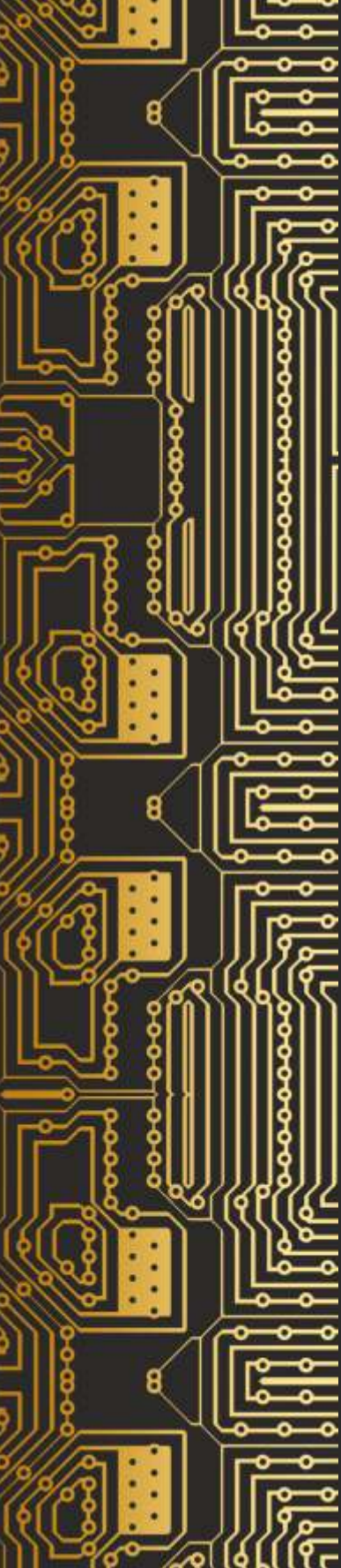
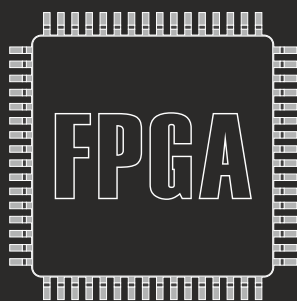




Projektowanie systemów wbudowanych w układach FPGA

Valery Salauyou
Adam Klimowicz



Dedykowana naszym rodzinom

PROJEKTOWANIE SYSTEMÓW WBUDOWANYCH W UKŁADACH FPGA

Valery Salaouyou, Adam Klimowicz



OFICyna WYDAWNICZA POLITECHNIKI BIAŁOSTOCKIEJ
BIAŁYSTOK 2022

Recenzenci:
prof. dr hab. inż. Alexander Barkalov
prof. dr hab. inż. Dariusz Kania

Redaktor naukowy dyscypliny informatyka techniczna i telekomunikacja:
prof. dr hab. Jarosław Stepaniuk

Korekta językowa:
mgr Agnieszka Polecka

Skład, grafika i okładka:
Marcin Dominów

Zdjęcie na okładce: geralt
<https://pixabay.com/pl/photos/p%C5%82ytka-drukowana-digitalizacja-2528399/>

© Copyright by Politechnika Białostocka, Białystok 2022

ISBN 978-83-67185-34-9
ISBN 978-83-67185-35-6 (eBook)
DOI: 10.24427/978-83-67185-35-6



Publikacja jest udostępniona na licencji
Creative Commons Uznanie autorstwa-Użycie niekomercyjne-Bez utworów zależnych 4.0
(CC BY-NC-ND 4.0).

Pełną treść licencji udostępniono na stronie
creativecommons.org/licenses/by-nc-nd/4.0/legalcode.pl.
Publikacja jest dostępna w Internecie na stronie Oficyny Wydawniczej PB.

Druk: PPH Remigraf sp. z o.o.

Oficina Wydawnicza Politechniki Białostockiej
ul. Wiejska 45C, 15-351 Białystok
e-mail: oficina.wydawnicza@pb.edu.pl
www.pb.edu.pl

Spis treści

Wykaz skrótów.....	13
Przedmowa.....	15
Część I. Projektowanie pamięci.....	21
1. Bloki pamięci wbudowanej.....	22
1.1. Pamięć systemów wbudowanych.....	22
1.2. Pamięć wbudowana (pamięć FPGA)	23
1.2.1. Zalety i wady pamięci wbudowanej	23
1.2.2. Zakres zastosowania pamięci wbudowanej.....	24
1.3. Architektura pamięci wbudowanej	25
1.3.1. Typy pamięci wbudowanej.....	25
1.3.2. Typy pamięci wbudowanej układów FPGA rodziny Cyclone V	27
1.3.3. Tryby pracy pamięci wbudowanej.....	28
1.3.4. Bity parzystości w blokach pamięci wbudowanej	34
1.3.5. Tryby synchronizacji pamięci wbudowanej.....	34
1.3.6. Tryby działania wyjść pamięci	36
1.3.7. Konfiguracja portu mieszanego.....	38
1.3.8. Zezwolenie na zapis bajtów danych	40
1.3.9. Obsługa trybu spakowanego	40
1.3.10. Zezwolenie synchronizacji adresu.....	41
1.4. Zalecenia dotyczące projektowania pamięci wbudowanej.....	41
1.4.1. Wybór bloku pamięci	41
1.4.2. Zewnętrzne rozwiązywanie konfliktów	42
1.4.3. Konfigurowanie trybu działania wyjść pamięci	42
1.4.4. Stan i inicjalizacja pamięci po włączeniu zasilania	42
1.4.5. Sterowanie synchronizacją w celu zmniejszenia zużycia energii	43
1.5. Wnioski	43
2. Opis pamięci wbudowanej w języku Verilog.....	46
2.1. Metody opisu pamięci na przykładzie jednoportowej pamięci typu RAM (Single-Port RAM) z jednym sygnałem synchronizacji.....	46

2.1.1. Deklaracja macierzy pamięci	46
2.1.2. Tryby działania wyjść pamięci	47
2.1.3. Opis pamięci ze „starymi danymi” na wyjściu	47
2.1.4. Opis pamięci z „nowymi” danymi na wyjściu	52
2.1.5. Opis pamięci z wyjściami kombinacyjnymi.....	56
2.1.6. Wstawianie rejestrów na wejściach adresowych pamięci	58
2.2. Inicjalizacja pamięci	60
2.3. Jednoportowa pamięć z dwoma sygnałami synchronizacji	63
2.4. Prosta dwuportowa pamięć typu RAM (Simple Dual-Port RAM)	64
2.5. Rzeczywista dwuportowa pamięć typu RAM (True Dual-Port RAM)	66
2.6. Pamięć typu ROM (Read-Only Memory)	68
2.7. Implementacja rejestrów przesuwających w blokach pamięci wbudowanej FPGA	71
2.8. Pamięć typu FIFO.....	75
2.9. Pamięć typu LIFO.....	79
2.10. Używanie atrybutów syntezy podczas opisywania pamięci w języku Verilog	82
2.11. Wnioski	85
3. Projektowanie pamięci z wykorzystaniem bloków IP i edytora parametrów systemu Quartus	89
3.1. Katalog IP do tworzenia modułów pamięci.....	89
3.2. Tworzenie modułów pamięci typu RAM i ROM	90
3.3. Tworzenie modułu pamięci typu FIFO	99
3.4. Tworzenie rejestru przesuwającego w bloku pamięci wbudowanej FPGA	103
3.5. Wnioski	105
Część II. Projektowanie układów sterowania	107
4. Projektowanie automatów mikroprogramowanych	108
4.1. Mikroprogramowanie i automaty mikroprogramowane	108
4.2. Reprezentacja AM. Sieci działań	110
4.3. Projektowanie automatów mikroprogramowanych zgodnie z siecią działań (GSA)	112
4.3.1. Metodyka projektowania AM zgodnie z GSA	112
4.3.2. Oznaczanie GSA	113
4.3.3. Tworzenie tablicy przejść.....	116
4.3.4. Dodatkowe oznaczenia GSA i pseudorównoważne AM	118
4.4. Implementacja AM na FPGA	119

4.4.1. Implementacja AM Mealy'ego	119
4.4.2. Implementacja AM Moore'a.....	120
4.4.3. Analiza funkcjonowania AM Mealy'ego i AM Moore'a.....	122
4.4.4. Implementacja AM Moore'a klasy C.....	124
4.5. Wykorzystanie AM przy realizacji synchronicznych układów mnożących	126
4.5.1. Implementacja układu mnożącego za pomocą algorytmu a.....	127
4.5.2. Projektowanie układu operacyjnego	128
4.5.3. Projektowanie układu sterującego.....	130
4.5.4. Implementacja układu mnożącego na FPGA	132
4.5.5. Porównanie różnych sposobów implementacji synchronicznych układów mnożących	137
4.6. Wnioski	139
5. Algorytmiczne automaty skończone (ASM)	142
5.1. Problemy GSA przy opisie algorytmów działania sprzętu	142
5.2. Język ASM.....	143
5.3. Metodyka budowy ASM	145
5.4. Przykład budowy ASM dla automatu Moore'a.....	146
5.5. Przykład budowy ASM dla automatu Mealy'ego	147
5.6. Implementacja automatów skończonych na FPGA zgodnie z ich opisem za pomocą ASM	148
5.6.1. Implementacja automatu Moore'a.....	149
5.6.2. Implementacja automatu Mealy'ego.....	150
5.7. Opis za pomocą ASM wspólnych modeli automatów Mealy'ego i Moore'a	152
5.8. Opis procesów równoległych za pomocą ASM	155
5.9. Opis układów kombinacyjnych za pomocą ASM	157
5.10. Wnioski.....	160
6. Automaty skończone ze ścieżką przetwarzania danych (FSMD).....	163
6.1. Sieć działań ASMD i automaty FSMD	163
6.2. Język ASMD	164
6.3. Wykorzystanie FSMD Moore'a do implementacji algorytmu mnożenia a.....	165
6.4. Wykorzystanie FSMD Mealy'ego do implementacji algorytmu mnożenia a.....	168
6.5. Zwiększenie wydajności synchronicznego układu mnożącego	173
6.6. Porównanie różnych sposobów implementacji synchronicznego układu mnożącego	176

6.7. Wnioski	182
Część III. Projektowanie procesorów wbudowanych	185
7. Podstawy projektowania procesorów wbudowanych.....	186
7.1. Architektury zestawów instrukcji procesora.....	186
7.2. Architektura zestawu instrukcji realizowanego procesora.....	188
7.2.1. Architektura zestawu instrukcji mikrokontrolera PIC16F84A	188
7.2.2. Projektowanie zestawu instrukcji procesora PIC	191
7.3. Podstawowe struktury procesorów RISC	194
7.3.1. Jednocyklowy procesor PIC	195
7.3.2. Wielocyklowy procesor PIC.....	196
7.3.3. Potokowy procesor PIC	198
7.3.4. Wykorzystanie pamięci podręcznej	201
7.3.5. Procesory superskalarne.....	202
7.3.6. Procesory wielowątkowe.....	203
7.3.7. Procesory wielordzeniowe.....	203
7.4. Metodyka projektowania procesorów na FPGA.....	204
7.5. Wnioski	206
8. Projektowanie jednocyklowego procesora PIC	209
8.1. Projektowanie układu operacyjnego	209
8.1.1. Elementy pamięci procesora	209
8.1.2. Określenie adresu następnej instrukcji.....	211
8.1.3. Realizacja instrukcji transferu danych (instrukcje movlw, movwf i movf).....	212
8.1.4. Realizacja operacji na bajtach (instrukcji formatu a)	213
8.1.5. Cykliczne operacje przesunięcia (instrukcje rlf i rrf)	213
8.1.6. Realizacja operacji inkrementacji i dekrementacji z pominięciem następnej instrukcji, jeśli wynik jest zerowy (instrukcje incfsz i decfsz)	215
8.1.7. Realizacja operacji bitowych (instrukcji formatu b).....	216
8.1.8. Realizacja operacji na stałych (instrukcji formatu c).....	217
8.1.9. Realizacja instrukcji skoku do adresu (goto), wywołania podprogramu (call) i powrotu z podprogramu (return, retlw).....	217
8.1.10. Realizacja ładowania wartości akumulatora z pamięci danych (instrukcja lw) i zapisu wartości akumulatora w pamięci danych (instrukcja sw)	219
8.1.11. Realizacja skoku do adresu w zależności od wyniku poprzedniej operacji (instrukcje gotoz i gotonz).....	219
8.2. Projektowanie układu arytmetyczno-logicznego	222

8.3. Projektowanie układu sterowania.....	224
8.4. Realizacja procesora jednocyklowego PIC.....	227
8.4.1. Struktura realizowanego procesora.....	227
8.4.2. Opis komponentów układu operacyjnego w języku Verilog	228
8.4.3. Opis układu operacyjnego (datapath)	231
8.4.4. Opis procesora PIC	234
8.4.5. Opis pamięci danych.....	235
8.4.6. Opis pamięci instrukcji.....	236
8.4.7. Opis modułu najwyższego poziomu	237
8.5. Debugowanie procesora jednocyklowego PIC	238
8.5.1. Debugowanie komponentów układu operacyjnego.....	239
8.5.2. Debugowanie układu sterowania.....	239
8.5.3. Debugowanie pamięci danych i pamięci instrukcji	241
8.5.4. Debugowanie całego projektu procesora PIC.....	241
8.6. Ocena wydajności procesora jednocyklowego	246
8.7. Modyfikacja procesora PIC.....	250
8.8. Wnioski	250
9. Projektowanie procesorów wielocyklowych PIC	253
9.1. Tradycyjne projektowanie procesora wielocyklowego PIC	253
9.1.1. Projektowanie układu operacyjnego.....	253
9.1.2. Projektowanie układu sterującego.....	256
9.1.3. Opis komponentów procesora wielocyklowego w języku Verilog	263
9.1.4. Debugowanie procesora wielocyklowego.....	270
9.1.5. Analiza projektu procesora wielocyklowego.....	270
9.2. Projektowanie procesora dwucyklowego PIC	271
9.2.1. Projektowanie układu procesora dwucyklowego	271
9.2.2. Opis w języku Verilog komponentów procesora dwucyklowego PIC.....	273
9.2.3. Debugowanie procesora dwucyklowego	276
9.3. Projektowanie czterocyklowego procesora PIC	276
9.3.1. Projektowanie układu operacyjnego.....	276
9.3.2. Projektowanie układu sterującego.....	281
9.3.3. Opis w języku Verilog komponentów procesora czterocyklowego....	282
9.3.4. Debugowanie procesora czterocyklowego	283
9.4. Ocena kosztów realizacji i wydajności procesorów PIC.....	284
9.5. Wnioski	286

Część IV. Projektowanie bloków cyfrowego przetwarzania sygnałów	289
10. Projektowanie filtrów cyfrowych	291
10.1. Systemy cyfrowego przetwarzania sygnałów na FPGA	291
10.1.1. Wprowadzenie do cyfrowego przetwarzania sygnałów.....	291
10.1.2. Zastosowanie FPGA do realizacji systemów DSP	292
10.2. Filtry cyfrowe	294
10.2.1. Klasyfikacja filtrów cyfrowych według częstotliwości przepustowej.....	295
10.2.2. Filtry FIR i IIR.....	296
10.2.3. Wybór między filrami IIR i FIR	297
10.2.4. Typy filtrów FIR.....	298
10.3. Opis filtrów cyfrowych w języku Verilog.....	298
10.4. Metodyka projektowania filtrów cyfrowych na FPGA	302
10.4.1. Definiowanie specyfikacji filtrów	303
10.4.2. Obliczanie współczynników filtra	304
10.4.3. Definiowanie struktury filtra	305
10.4.4. Analiza wpływu skończonej długości słowa	305
10.4.5. Realizacja filtra.....	306
10.5. Projektowanie filtrów cyfrowych w systemie MATLAB	307
10.6. Symulacja filtrów cyfrowych w systemie ModelSim.....	316
10.7. Realizacja filtrów cyfrowych w systemie Quartus	318
10.7.1. Realizacja w systemie Quartus filtrów cyfrowych opracowanych w systemie MATLAB	318
10.7.2. Badanie realizacji na FPGA różnych typów filtrów cyfrowych.....	319
10.8. Wnioski	322
Część V. Projektowanie podsystemu synchronizacji	325
11. Realizacja projektów synchronicznych.....	326
11.1. Zasady funkcjonowania projektów synchronicznych.....	326
11.2. Zalecenia firmy Intel dotyczące projektowania niezawodnych projektów synchronicznych	327
11.2.1. Unikanie pętli kombinacyjnych.....	327
11.2.2. Unikanie nieumyślnego wprowadzania zatrzaśków do projektu	328
11.2.3. Unikanie opóźnień w ścieżkach synchronizacji.....	329
11.2.4. Używanie układów synchronicznych do wygenerowania pojedynczego impulsu	329
11.2.5. Generacja zegara wewnętrznego	330
11.2.6. Wykorzystanie wewnętrznej logiki FPGA jako źródła sygnału zegarowego lub sygnałów sterujących rejestrami.....	331

11.2.7. Domeny synchronizacji	331
11.2.8. Dzielniki częstotliwości	332
11.2.9. Korzystanie z liczników kaskadowych (ripple counters)	332
11.2.10. Zastosowanie sieci synchronizacji FPGA.....	334
11.3. Multipleksowanie sygnałów zegarowych.....	335
11.4. Bramkowanie sygnałów zegarowych	338
11.4.1. Bramkowanie sygnałów zegarowych w projektach na FPGA.....	338
11.4.2. Metody bramkowania sygnałów zegarowych	338
11.4.3. Realizacja rejestrów z bramkowaniem sygnału zegarowego.....	343
11.4.4. Badanie metod bramkowania sygnałów zegarowych	348
11.4.5. Zastosowanie opcji Auto Gated Clock Conversion	349
11.4.6. Bramkowanie sygnałem asynchronicznym.....	350
11.5. Zalecenia dotyczące tworzenia szybkich i złożonych projektów.....	351
11.5.1. Planowanie realizacji fizycznej	351
11.5.2. Planowanie zasobów FPGA	352
11.5.3. Spełnienie ograniczeń czasowych	353
11.6. Optymalizacja podsystemu synchronizacji.....	353
11.6.1. Retiming	353
11.6.2. Duplikacja rejestrów	354
11.6.3. Synteza fizyczna logiki kombinacyjnej	354
11.6.4. Opcje systemu Quartus wpływające na optymalizację synchronizacji	355
11.6.5. Optymalizacja ścieżek krytycznych	358
11.7. Projektowanie sygnałów resetowania	359
11.7.1. Synchroniczne resetowanie rejestrów	359
11.7.2. Asynchroniczne resetowanie rejestrów	363
11.7.3. Badania metod resetowania rejestrów	368
11.8. Atrybuty syntezy wpływające na synchronizację projektu	369
11.9. Wnioski	371
12. Bloki pętli synchronizacji fazowej PLL	374
12.1. Zasady funkcjonowania bloku PLL.....	374
12.2. Specyfika architektury bloków PLL w rodzinie FPGA Cyclone V	376
12.3. Cechy funkcjonowania bloków PLL w układach FPGA z rodziny Cyclone V	378
12.4. Tryby synchronizacji sprzężenia zwrotnego	379
12.4.1. Tryb synchronizacji źródłowej.....	379
12.4.2. Tryb kompensacji LVDS.....	380

12.4.3. Tryb bezpośredniej kompensacji.....	381
12.4.4. Tryb kompensacji normalnej.....	381
12.4.5. Tryb bufora z zerowym opóźnieniem ZDB	382
12.4.6. Tryb zewnętrznego sprzężenia zwrotnego	382
12.5. Ułamkowy i całkowity tryb pracy bloków PLL	383
12.6. Przełączanie zegarów referencyjnych.....	383
12.7. Konfigurowanie bloków PLL za pomocą bloków IP i Edytora parametrów systemu Quartus	384
12.8. Rekonfiguracja bloków PLL	390
12.9. Rekonfiguracja PLL za pomocą interfejsu Avalon-MM.....	392
12.10. Rekonfiguracja bloków PLL za pomocą strumieniowego przesyłania pliku MIF ze wbudowanego bloku pamięci FPGA	393
12.11. Dynamiczne przesunięcie fazowe bloku PLL	394
12.11.1. Wykonywanie dynamicznego przesunięcia fazowego przy użyciu Bloku IP do rekonfiguracji PLL (Altera PLL Reconfig)	394
12.11.2. Wykonywanie dynamicznego przesunięcia fazowego przy użyciu Bloku IP Altera PLL	394
12.12. Wnioski	397
Zakończenie	401
Literatura	403
Indeks tematyczny	405
Spis tabel	417
Spis rysunków	419
Streszczenie	429
Abstract.....	431

Wykaz skrótów

ADC	– Analog-to-Digital Converter
ALM	– Adaptive Logic Modules
ALU	– Arithmetic Logical Unit
AM	– Automat Mikroprogramowalny
ASIC	– Application-Specific Integrated Circuit
ASM	– Algorithmic State Machine
ASMD	– Algorithmic State Machine with Datapath
Avalon-MM	– Avalon Memory-Mapped
DAC	– Digital-to-Analog Converter
DM	– Data Memory
DRAM	– Dynamic RAM
DSP	– Digital Signal Processing
ECC	– Error Checking and Correcting
EEPROM	– Electrically Erasable Programmable Read-Only Memory
EPROM	– Erasable Programmable Read-Only Memory
FIFO	– First-In, First-Out
FIR	– Finite Impulse Response
FPGA	– Field Programmable Gate Array
fPLL	– fractional PLL
FSM	– Finite State Machine
FSMD	– Finite State Machine with Datapath
FU	– Functional Unit
HDL	– Hardware Description Language
HPF	– HighPass Filter
GCLK	– Global Clock Networks
IIR	– Infinite Impulse Response
ILP	– Instruction-Level Parallelism
IM	– Instruction Memory
IOE	– Input-Output Element
IR	– Instruction Register
ISA	– Instruction Set Architecture
JVM	– Java Virtual Machine
LAB	– Logic Array Block
LIFO	– Last-In, First-Out

LPF	– LowPass Filter
LUT	– Look-Up Table
MAC	– Multiply and Accumulate
MLAB	– Memory Logic Array Block
PC	– Program Counter
PCB	– Printed Circuit Board
PFD	– Phase Frequency Detector
PIC	– Peripheral Interface Controller
PLL	– Phase-Locked Loop
PMA	– Physical Medium Attachment
RAM	– Random-Access Memory
RCLK	– Regional Clock Networks
RISC	– Reduced Instruction Set Computer
ROM	– Read Only Memory
RTL	– Register Transfer Level
SDM	– Sigma-Delta Modulator
SDRAM	– Synchronous Dynamic RAM
SoC	– System on Chip
SRAM	– Static Random-Access Memory
SSRAM	– Synchronous Static RAM
TLP	– Thread Level Parallelism
VCO	– Voltage-Controlled Oscillator
ZDB	– Zero Delay Buffer

Przedmowa

Książka omawia projektowanie bloków funkcjonalnych na *układach programowalnych* (field programmable gate array – FPGA), które są najczęściej używane w *systemach wbudowanych* (embedded systems). Wszystkim prezentowanym tematom towarzyszy duża liczba przykładów opisanych w języku Verilog. Do zademonstrowania wyników syntezy i symulacji rozpatrywanych przykładów został wykorzystany system Quartus Prime w wersji 18.1, a płyta ewaluacyjna Intel DE1-SoC została wykorzystana do fizycznej implementacji. Jednak materiał w książce nie koncentruje się na FPGA konkretnej firmy lub konkretnym systemie projektowania. Aby opanować materiał zaprezentowany w monografii, wystarczy żeby użyte narzędzie projektowe wspierało język wejściowy Verilog przy opisie projektów. Wszystkie prezentowane przykłady można także zaimplementować na układach FPGA Xilinx przy użyciu systemów ISE lub Vivado.

Osobliwością książki jest to, że każdy rozważany problem jest kompleksowo badany i proponowane są najskuteczniejsze metody jego rozwiązania. Omówienie wielu zagadnień podsumowane jest badaniami eksperymentalnymi, na podstawie których formułowane są zalecenia dotyczące najbardziej efektywnego wykorzystania różnych podejść do projektowania konkretnej jednostki lub bloku. Realizacja zadań zaproponowanych w książce przyczynia się do lepszego przyswojenia materiału, a także do zdobycia nieocenionego doświadczenia w projektowaniu bloków systemów wbudowanych na układach FPGA.

Książka oparta jest na kursach wykładowych prowadzonych przez autora przez wiele lat dla studentów Politechniki Białostockiej i dla profesjonalnych projektantów systemów wbudowanych oraz na specjalnych seminariach, przeznaczonych dla profesjonalnych projektantów systemów wbudowanych na specjalnych seminariach.

Monografia składa się z 12 rozdziałów, które są tematycznie połączone w pięć części.

Pierwsza część poświęcona jest projektowaniu pamięci systemów wbudowanych. Żaden z nowoczesnych systemów wbudowanych nie może obejść się bez pamięci. Ograniczyliśmy się do rozpatrzenia problemów projektowania tylko pamięci wbudowanej zaimplementowanej w układach FPGA. Pamięć w systemach wbudowanych jest szeroko stosowana do przechowywania kodu programu, generowania złożonych funkcji matematycznych w postaci dużej *tablicy przeglądowej* (look-up table – LUT), pamięci podręcznej procesora, pamięci typu FIFO do buforowania danych sieciowych itp.

Rozdział 1 omawia główne typy pamięci systemów wbudowanych, podaje ich klasyfikację i pokazuje miejsce w tej klasyfikacji pamięci na układzie FPGA (pamięci wbudowanej). W rozdziale opisano również architekturę pamięci wbudowanej i przedstawiono zalecenia firmy Intel dotyczące jej projektowania.

Rozdział 2 omawia projektowanie pamięci wbudowanej za pomocą języka Verilog. Różne sposoby opisu pamięci podano na przykładzie pamięci jednoportowej. Pokazano cechy opisu pamięci dwuportowej i pamięci typu ROM. Osobno została rozpatrzona kwestia inicjalizacji pamięci. Bloki pamięci wbudowanej FPGA umożliwiają wydajną implementację rejestrów przesuwających z odczepami stosowanymi w aplikacjach do *cyfrowego przetwarzania sygnałów* (digital signal processing – DSP). Podano metody projektowania w języku Verilog pamięci, takich jak FIFO (First-In, First-Out) i LIFO (Last-In, First-Out). Przedstawiono również atrybuty syntezy używane do opisu pamięci w języku Verilog.

Moduły pamięci w systemach wbudowanych wykorzystujących układy FPGA można również budować przy użyciu bloków IP (Intellectual Property Core) i edytora parametrów systemu Quartus. W tym przypadku wykorzystywane są specjalne megafunkcje, opracowane przez firmę Intel. Megafunkcje wykorzystują wszystkie właściwości architektur zintegrowanych bloków pamięci FPGA. Rozdział 3 omawia tworzenie przy użyciu bloków IP systemu Quartus modułów pamięci typu RAM, ROM i FIFO, a także rejestrów przesuwających.

Druga część poświęcona jest projektowaniu układów sterowania. Twórca systemów wbudowanych staje przed zadaniem projektowania układów sterowania za każdym razem, gdy powstaje nowy projekt. W niektórych przypadkach, cały system wbudowany może składać się z jednego układu sterującego lub kilku systemów sterowania współpracujących ze sobą. Jako urządzenie sterowania tradycyjnie używane są *automaty skończone* (finite state machine – FSM).

Wiele urządzeń i systemów sterowania opiera się na zasadach sterowania mikroprogramowego. Szeroko stosowane w systemach wbudowanych *automaty mikroprogramowane* (AM) są zbudowane na tych samych zasadach. Języki sterowania logicznego służą do opisu działania AM, a jednym z nich jest język sieci działań. Rozdział 4 omawia metodę projektowania AM zgodnie z siecią działań, a także implementację na FPGA AM Meal'ego, AM Moore'a i AM klasy C. Jako przykład zastosowania AM przedstawiono sprzętową implementację na FPGA synchronicznych układów mnożenia.

Rozdział 5 omawia nowy język graficzny do opisu automatów skończonych, zwany *algorytmicznym automatem skończonym* (algorithmic state machine – ASM) lub siecią działań algorytmu. Główną zaletą sieci działań algorytmu ASM w porównaniu z siecią działań jest to, że w języku Verilog automat skończony można bezpośrednio opisać poprzez jego reprezentację jako sieć działań ASM, bez szeregu formalnych etapów syntezy wykonywanych ręcznie, które często są źródłem trudnych do wykrycia błędów. Za pomocą sieci działań ASM można również opisać funkcjonowanie wspólnych modeli automatów skończonych, procesów równoległych i układów kombinacyjnych.

Rozdział 6 przedstawia nową technikę projektowania układów cyfrowych opartą na *sieci działań ASM ze ścieżką przetwarzania danych* (algorithmic state machine with datapath – ASMD) i *automatów skończonych ze ścieżką przetwarzania danych* (finite state machine with datapath – FSMMD), która została nazwana techniką ASMD-FSMMD. W przeciwieństwie do tradycyjnego podejścia, technika ASMD-FSMMD umożliwia natychmiastowe opisanie całego układu w języku Verilog bez dzielenia go na urządzenie operacyjne i urządzenie sterowania (kontroler – controller), co znacznie skraca czas i koszty projektowania, a także zwiększa niezawodność opracowanych projektów.

Trzecia część poświęcona jest projektowaniu procesorów wbudowanych. Nowoczesne systemy wbudowane często zawierają procesory zwane procesorami wbudowanymi (embedded processors), służące do wykonywania programów i algorytmów.

Rozdział 7 omawia ogólne problemy projektowania procesorów wbudowanych na układach FPGA: architekturę tradycyjną dla różnych list rozkazów procesorów, architekturę oraz sposób projektowania listy rozkazów dla procesorów PIC, podstawowe struktury procesorów RISC (reduced instruction set computer), a także metodologię projektowania procesorów na układach FPGA.

Rozdział 8 przedstawia proces projektowania jednordzeniowego procesora PIC oraz szczegółowo opisuje metodykę projektowania układu operacyjnego. Osobno rozważa się projekt jednostki arytmetyczno-logicznej i układu sterującego. Kody źródłowe Verilog są pokazane dla komponentów układu procesora, a także modułu najwyższego poziomu. Szczególną uwagę zwraca się na debugowanie procesora PIC i ocenę jego wydajności; podano zalecenia dotyczące sposobu zwiększenia wydajności procesora.

Rozdział 9 przedstawia trzy metodyki projektowania procesorów wielocyfrowych. Pierwsza technika polega na tradycyjnym podejściu do projektowania procesorów wielocyfrowych i obejmuje sekwencyjne wprowadzanie stanów układu sterującego: do wybierania i dekodowania polecenia, implementacji niektórych grup instrukcji i poszczególnych instrukcji. Druga technika maksymalnie wykorzystuje układ operacyjny i urządzenie sterowania procesorem jednocyfrowym. Aby to zrobić, automat skończony, który określa etapy wykonywania instrukcji, jest dodawany do układu procesora. Trzecia technika ma na celu zwiększenie liczby etapów wykonawczych procesora wielocyfrowego. Jest to konieczne, aby poprawić wydajność w przypadku przetwarzania danych potokowych.

Czwarta część poświęcona jest projektowaniu bloków cyfrowych przetwarzania sygnałów. Jedną z wyróżniających cech nowoczesnych systemów wbudowanych jest możliwość realizacji szerokiego zakresu zadań z dziedziny cyfrowego przetwarzania sygnałów DSP. Wynika to z faktu, że metody DSP są coraz częściej stosowane w różnych obszarach działalności człowieka.

Czwartą część przedstawiono tylko w jednym rozdziale – 10, który omawia projektowanie filtrów cyfrowych na układach FPGA. Podano krótkie wprowadzenie do cyfrowego przetwarzania sygnałów i wskazano możliwości FPGA przy implementacji systemów DSP. Określono główne parametry filtrów cyfrowych oraz metody opisywania filtrów cyfrowych w języku Verilog. Przedstawiono ogólną metodykę projektowania filtrów cyfrowych, rozważono przykład opracowania filtra cyfrowego

w systemie MATLAB, oraz symulację filtrów w systemie ModelSim i implementację filtrów w systemie Quartus. W podsumowaniu, zbadano implementację różnych typów filtrów dla różnych rodzin FPGA.

Piąta część poświęcona jest projektowaniu podsystemu synchronizacji. Problem skutecznej implementacji podsystemu synchronizacji staje się szczególnie istotny przy opracowywaniu szybkich i złożonych projektów. W przeciwieństwie do projektów na układach ASIC (application-specific integrated circuit), projekty na układach FPGA nie muszą obejmować całego podsystemu synchronizacji. W FPGA wiele elementów podsystemu synchronizacji zostało już zaprojektowanych i wdrożonych na poziomie sprzętowym przez projektantów FPGA. Elementy te obejmują sieci synchronizacji, pętle synchronizacji fazowej PLL (phase-locked loop) i bloki sterowania sygnałem zegarowym. Ponadto kompilator systemu Quartus automatycznie rozwiązuje wiele problemów związanych z synchronizacją projektu.

Rozdział 11 omawia różne problemy, przed którymi stoi inżynier podczas projektowania podsystemu synchronizacji złożonego projektu na FPGA. Realizacja projektów synchronicznych na FPGA ma swoją własną charakterystykę, której znajomość jest niezbędna do skutecznego wykorzystania możliwości sprzętowych FPGA. W tym rozdziale omówiono zalecenia firmy Intel dotyczące projektowania projektów synchronicznych na układach FPGA, multipleksowania i bramkowania sygnałów zegara, zalecenia dotyczące opracowywania szybkich i złożonych logicznie projektów, projektowania sygnałów „reset”, a także atrybutów syntezy wpływających na synchronizację projektu.

Rozdział 12 opisuje bloki pętli synchronizacji fazowej PLL. Bloki PLL są zaprojektowane do tworzenia wysokiej jakości wewnętrznych sygnałów zegarowych o zadanej częstotliwości, współczynniku wypełnienia i przesunięciu fazowym na podstawie referencyjnych sygnałów zegarowych. W rozdziale omówiono zasady działania bloków PLL, architekturę i funkcjonowanie bloków PLL w FPGA rodziny Cyclone V, a także sposób konfiguracji bloków PLL w projektach FPGA.

Każdy rozdział kończy się wnioskami. Wnioski podsumowują główne punkty rozważane w danym rozdziale i prezentują najważniejsze rezultaty badań oraz skupiają czytelnika na najważniejszych zagadnieniach, które mogły zostać pominięte podczas czytania tego rozdziału. W książce zaprezentowano dużą liczbę przykładów oraz wyników ich syntezy i symulacji. Wszystkie przykłady są sprawdzone i mogą być używane w projektach systemów wbudowanych.

Monografia przeznaczona jest przede wszystkim dla studentów odpowiednich specjalności uczelni technicznych. Materiał w niej zawarty może być wykorzystywany przez nauczycieli do prowadzenia wykładów, ćwiczeń laboratoryjnych i zajęć praktycznych. Wiele zagadnień poruszonych w książce nie zostało jeszcze w pełni zbadanych, dlatego też mogą zainteresować młodych naukowców. Książka może być również wykorzystana przez inżynierów jako samouczek do studiowania różnych problemów związanych z projektowaniem systemów wbudowanych na FPGA. Niektóre tematy przedstawione w tej publikacji są całkiem nowe i mogą zainteresować nawet doświadczonych projektantów.

Badania zostały zrealizowane w ramach pracy nr WZ/WI-IIT/4/2020 w Politechnice Białostockiej i sfinansowane z subwencji badawczej przekazanej przez Ministra Edukacji i Nauki.

Część I

Projektowanie pamięci

Projektowanie pamięci jest ważnym zadaniem w procesie tworzenia systemu wbudowanego na FPGA, ponieważ efektywne wykorzystanie pamięci znacznie zwiększa wydajność systemu. Pamięć w systemach wbudowanych jest szeroko stosowana do przechowywania kodu programu, do generowania wartości złożonych funkcji matematycznych w postaci dużej *tablicy wyszukiwania* (look-up table – LUT), jako pamięć podręczna procesora, jako pamięć typu FIFO do buforowania danych sieciowych, podczas realizacji zadań specjalnych, na przykład w algorytmie konwersji kolorów itp.

Wymagania dotyczące pamięci są w dużej mierze zależne od charakteru aplikacji, które mają być uruchamiane w systemie wbudowanym. W przypadku prostych i tanich systemów, pojemność i wydajność pamięci mogą być niewielkie. Jednak, w przypadku złożonych i szybkich systemów, przepustowość pamięci może być kluczowym wymogiem, tzn. wąskim gardłem w całym systemie. Dlatego do projektowania struktury pamięci w systemach wbudowanych powinno się podchodzić szczególnie ostrożnie.

1. Bloki pamięci wbudowanej

W tym rozdziale omówiono główne typy pamięci systemów wbudowanych, podano ich klasyfikację i pokazano miejsce pamięci w układzie FPGA (*pamięć wbudowana*). Opisano też architekturę pamięci wbudowanej oraz przedstawiono zalecenia firmy Intel, dotyczące projektowania pamięci wbudowanej.

1.1. Pamięć systemów wbudowanych

Pamięć systemów wbudowanych można podzielić na dwie duże klasy: *pamięć ulotną* (volatile memory) i *pamięć nieulotną* (non-volatile memory).

Pamięć ulotna zachowuje swoją zawartość tylko wtedy, gdy nie zostanie od niej odłączone napięcie zasilania. Po wyłączeniu zasilania zawartość pamięci ulotnej zostaje utracona. Dlatego w systemach wbudowanych nie można używać pamięci ulotnej do przechowywania informacji, która powinna być zachowana po wyłączeniu zasilania.

Przykłady pamięci ulotnej:

- *pamięć statyczna o dostępie swobodnym* (static random-access memory – SRAM);
- *synchroniczna statyczna pamięć typu RAM* (synchronous static RAM – SSRAM);
- *synchroniczna dynamiczna pamięć typu RAM* (synchronous dynamic RAM – SDRAM);
- *pamięć wbudowana w układ FPGA* (FPGA on-chip memory).

Pamięć nieulotna zachowuje swoją zawartość po wyłączeniu zasilania systemu. Z tego powodu pamięć nieulotna służy do przechowywania informacji pobieranych przez system między momentem włączenia i wyłączenia zasilania. Zazwyczaj pamięć nieulotna przechowuje *kod rozruchowy procesora* (boot-code), stałe ustawienia aplikacji, a także dane konfiguracyjne FPGA.

Pomimo tak pozytywnej cechy, jak zapisywanie danych po wyłączeniu zasilania, pamięć nieulotna jest znacznie wolniejsza podczas zapisu niż pamięć ulotna. Dlatego pamięć nieulotna ma bardziej złożone procedury zapisu i kasowania. Pamięć nieulotną można kasować ograniczoną liczbę razy, co gwarantuje producent, po czym może ona przestać działać.

Przykłady pamięci nieulotnej:

- wszystkie rodzaje pamięci flash;
- *kasowalna programowalna pamięć tylko do odczytu* (erasable programmable read-only memory – EPROM);
- *elektrycznie kasowana programowalna pamięć tylko do odczytu* (electrically erasable programmable read-only memory – EEPROM).

Nowoczesne systemy wbudowane najczęściej wykorzystują pamięć flash jako pamięć nieulotną. Większość aplikacji wbudowanych wymaga zarówno pamięci ulotnej, jak i nieulotnej. Dlatego systemy wbudowane zazwyczaj zawierają obie klasy pamięci.

1.2. Pamięć wbudowana (pamięć FPGA)

Pamięć wbudowana w układ FPGA (FPGA on-chip memory), zwana także *pamięcią wbudowaną* (embedded memory), jest najprostszym i najtańszym rodzajem pamięci do zastosowania w systemach wbudowanych na FPGA. Pamięć wbudowana jest pamięcią ulotną, więc zapisane w niej informacje są tracone po wyłączeniu zasilania.

Pamięć wbudowana jest zaimplementowana bezpośrednio w układzie FPGA. Podczas korzystania z niej zewnętrzne wyjścia FPGA i połączenia na płycie drukowanej nie są wykorzystywane, jak w przypadku korzystania z innych rodzajów pamięci.

Pamięć wbudowaną można stworzyć w systemie Quartus za pomocą narzędzia **IP Catalog** zwanego *Edytorem parametrów*. Tutaj można zdefiniować rozmiar, liczbę bitów słowa i typ pamięci wbudowanej, a także cechy specjalne pamięci wbudowanej, na przykład *dostęp dwuportowy* (dual-port access) do macierzy pamięci.

1.2.1. Zalety i wady pamięci wbudowanej

Pamięć wbudowana ma najniższe opóźnienia i najwyższą przepustowość w systemach wbudowanych na FPGA. Zazwyczaj opóźnienie pamięci wbudowanej jest równe jednemu taktowi sygnału zegara. Transfer danych w pamięci może być *potokowy* (pipelined), co daje typową przepustowość równą jednej operacji w każdym takcie.

Pamięć wbudowana umożliwia pracę w trybie dwuportowym, z osobnymi portami dla operacji zapisu i odczytu. *Tryb dwuportowy* (Dual-port mode) efektywnie podwaja potencjalną przepustowość pamięci, co pozwala zapisywać do pamięci za pomocą jednego portu i jednocześnie odczytywać poprzez drugi port.

Zawartość pamięci wbudowanej można zainicjować podczas konfiguracji FPGA. Pamięć wbudowana umożliwia zmianę zawartości początkowej za każdym razem, gdy układ FPGA jest konfigurowany.

Pamięć wbudowana ma następujące zalety:

- najmniejsze opóźnienie, najwyższa przepustowość spośród wszystkich typów pamięci w systemach wbudowanych na FPGA;
- każda transakcja jest wykonywana w jednym cyklu synchronizacji (takcie sygnału zegara);
- przepustowość pamięci można podwoić dzięki dwuportowemu trybowi pracy;
- nie wymaga stosowania zewnętrznych wyprowadzeń FPGA oraz połączeń na płycie drukowanej;
- użycie pamięci wbudowanej może znacznie skrócić czas opracowania i koszt projektu ze względu na brak konieczności stosowania interfejsu pamięci zewnętrznej;
- zmiany zawartości pamięci można wprowadzić podczas konfiguracji FPGA.

Pamięć wbudowana ma również pewne wady, z których główną jest niewielka pojemność przechowywanej informacji (od dziesiątek kilobajtów do kilku megabajtów). Ponadto pamięć wbudowana jest ulotna, dlatego traci swoją zawartość, gdy zasilanie zostanie wyłączone.

1.2.2. Zakres zastosowania pamięci wbudowanej

Ze względu na małe opóźnienie pamięć wbudowana jest najbardziej skuteczna jako pamięć podręczna procesora, ponieważ ograniczona pojemność pamięci wbudowanej nie jest krytycznym parametrem dla pamięci podręcznej. Na przykład procesor Nios II wykorzystuje pamięć wewnętrzną do przechowywania kodu programu i danych w pamięci podręcznej.

Pamięć wbudowana jest często używana jako pamięć tzw. *ściśle powiązana*. *Ściśle powiązana pamięć* (tightly coupled memory) jest pamięcią odwzorowaną na normalną przestrzeń adresową procesora, ma dedykowany interfejs do komunikacji z procesorem i małe opóźnienie.

Czasami najszybszym sposobem na zaimplementowanie złożonych funkcji matematycznych jest zapisanie gotowych wyników w *tablicy wyszukiwania* (look-up table – LUT). Pamięć wewnętrzna może służyć jako duża tablica wyszukiwania do implementacji takich funkcji. Jednak liczba możliwych wartości funkcji musi odpowiadać dostępnej pojemności pamięci wbudowanej.

Pamięć wbudowana idealnie nadaje się jako bufor typu *FIFO* (First-In, First-Out) w aplikacjach telekomunikacyjnych i do komunikacji między systemami działającymi na różnych częstotliwościach synchronizacji.

Zatem najbardziej efektywne wykorzystanie pamięci wbudowanej to:

- pamięć podręczna procesora;
- pamięć *ściśle powiązana*;
- duże tablice wyszukiwania do implementacji funkcji matematycznych i specjalnych;
- pamięć typu FIFO.

Pamięć wbudowana jest słabo dostosowana do aplikacji wymagających dużych pojemności pamięci. Jeśli aplikacja korzysta z wielu małych bloków danych i nie wszystkie bloki są umieszczone w pamięci układu, należy dokładnie zbadać, które z tych bloków są zaimplementowane w wewnętrznej pamięci FPGA, a które w pamięci zewnętrznej. W pamięci układu należy umieścić bloki, do których aplikacja odwołuje się najczęściej.

1.3. Architektura pamięci wbudowanej

W tej sekcji omówiono cechy architektury pamięci wbudowanej układów FPGA firmy Intel. Bloki pamięci wbudowanej układów FPGA firmy Xilinx są omówione w [7].

1.3.1. Typy pamięci wbudowanej

Różne rodziny FPGA zazwyczaj zawierają różne typy pamięci wbudowanej. W układach FPGA firmy Intel można znaleźć następujące bloki pamięci wbudowanej: M512, MLAB, M4K, M9K, M10K, M20K, M144K, M-RAM i eSRAM. Tabela 1.1 pokazuje, które bloki pamięci wbudowanej są uwzględnione w architekturach różnych rodzin FPGA.

TAB. 1.1. Bloki pamięci wbudowanej różnych rodzin FPGA firmy Intel

	M512 (512 bits)	MLAB (640 bits)	M4K (4K bits)	M9K (9K bits)	M10K (10K bits)	M20K (20K bits)	M144K (144K bits)	M-RAM (512K bits)	eSRAM (18/47 Mb)
Agilex		V				V			V
Arria GX	V		V					V	
Arria GZ		V				V			
Arria II GX		V		V					
Arria II GZ		V		V			V		
Arria V		V			V				
Arria V GZ		V				V			
Arria 10		V				V			
Cyclone			V						
Cyclone II			V						
Cyclone III				V					
Cyclone IV				V					
Cyclone V		V			V				
Cyclone 10 GX		V				V			

	M512 (512 bits)	MLAB (640 bits)	M4K (4K bits)	M9K (9K bits)	M10K (10K bits)	M20K (20K bits)	M144K (144K bits)	M-RAM (512K bits)	eSRAM (18/47 Mb)
Cyclone 10 LP				V					
MAX 10				V					
Stratix	V		V					V	
Stratix II	V		V					V	
Stratix III		V		V			V		
Stratix IV		V		V			V		
Stratix V		V				V			
Stratix 10		V				V			V

Bloki pamięci M512, M4K, M9K, M10K, M20K, M144K i M-RAM to osobne bloki pamięci, które są wbudowane w strukturę FPGA i realizują tylko i wyłącznie funkcje pamięci. Bloki te są idealne dla dużych macierzy pamięci i zapewniają dużą liczbę niezależnych portów.

Bloki pamięci matrycy logicznej MLAB (memory logic array block) są konfigurowane z *bloków macierzy logicznej* (logic array block – LAB), które można wykorzystywać zarówno do implementacji logiki, jak i do tworzenia pamięci wbudowanej. Bloki MLAB są dobrze dostosowane do dużych i małych macierzy pamięci; są zoptymalizowane pod kątem implementacji rejestrów przesuwających do aplikacji cyfrowego przetwarzania sygnałów DSP, buforów FIFO i linii opóźnień filtrów cyfrowych.

Bloki pamięci eSRAM nazywane są inaczej systemami pamięci wbudowanej typu RAM. Zostały one zaprojektowane do przechowywania dużych ilości informacji podczas interakcji z procesorem. Możliwe konfiguracje bloków pamięci wbudowanej układów FPGA firmy Intel pokazano w tabeli 1.2.

TAB. 1.2. Dopuszczalne konfiguracje bloków pamięci wbudowanej układów FPGA firmy Intel

M512 (512bits)	MLAB (640 bits)	M4K (4K Bits)	M9K (9K Bits)	M10K (10K Bits)	M20K (20K Bits)	M144K (144K Bits)	M-RAM (512K bits)
512x1	64x10	4Kx1	8Kx1	8Kx1	16Kx1	16Kx8	64Kx8
256x2	32x16	2Kx2	4Kx2	4Kx2	8Kx2	16Kx9	64Kx9
128x4	32x18	1Kx4	2Kx4	2Kx4	4Kx5	8Kx16	32Kx16
64x8	32x20	512x8	1Kx8	2Kx5	2Kx10	8Kx18	32Kx18
64x9		512x9	512x16	1Kx8	1Kx20	4Kx32	16Kx32
32x16		256x16	512x18	1Kx10	512x40	4Kx36	16Kx36
32x8		256x18	256x32	512x16		2Kx64	8Kx64
		128x32	256x36	512x20		2Kx72	8Kx72
		128x36		256x32			4Kx128
				256x40			4Kx144

1.3.2. Typy pamięci wbudowanej układów FPGA rodziny Cyclone V

Każdy projektant systemów wbudowanych powinien dobrze znać architekturę *docelowego układu FPGA*, na którym powinien być realizowany system wbudowany. Rozpatrzmy typy pamięci wbudowanej dla układu 5CSEMA5F31C6N z rodziny Cyclone V SE, który jest zainstalowany na płycie dydaktycznej DE1-SoC.

Układy z rodziny Cyclone V mają dwa rodzaje pamięci wbudowanej: bloki M10K i bloki MLAB. Jeden układ 5CSEMA5F31C6N zawiera 397 bloków M10K o pojemności 10 Kb każdy, które stanowią 3970 Kb pamięci oraz 768 bloków MLAB o pojemności 640 bitów każdy, które stanowią 480 Kb pamięci. W ten sposób jeden układ 5CSEMA5F31C6N zawiera 4450 Kb pamięci wbudowanej.

Jeden blok M10K zawiera 10 Kb pamięci. Aby utworzyć większą pamięć lub zwiększyć szerokość słowa, można połączyć kilka bloków M10K razem, przy tym mogą być też zastosowane elementy logiczne FPGA.

Każdy blok MLAB składa się z dziesięciu *adaptacyjnych modułów logicznych* (adaptive logic modules – ALMs).

Główne właściwości wewnętrznej pamięci FPGA rodziny Cyclone V podano w tabeli 1.3.

TAB. 1.3. Właściwości pamięci wbudowanej rodziny Cyclone V SE

Właściwość	M10K	MLAB
Maksymalna częstotliwość robocza	315 MHz	420 MHz
Pojemność jednego bloku wraz z bitem parzystości	10240	640
Bit parzystości	Obsługiwany	Obsługiwany
Dostęp do poszczególnych bajtów	Obsługiwany	Obsługiwany
Tryb pakietowy	Obsługiwany	Nieobsługiwany
Zezwolenie na synchronizację adresów	Obsługiwany	Obsługiwany
Mieszana szerokość magistrali danych w trybie prostej pamięci dwuportowej	Obsługiwany	Nieobsługiwany
Mieszana szerokość magistrali danych w trybie rzeczywistej pamięci dwuportowej	Obsługiwany	Nieobsługiwany
Mieszana szerokość magistrali danych bufora FIFO	Obsługiwany	Nieobsługiwany
Plik inicjalizacji pamięci (.mif)	Obsługiwany	Obsługiwany
Mieszany tryb synchronizacji	Obsługiwany	Obsługiwany
Pamięć w pełni synchroniczna	Obsługiwany	Obsługiwany

Właściwość	M10K	MLAB
Pamięć asynchroniczna	Nieobsługiwany	Tylko do operacji odczytu strumieniowego (flow-through)
Stan po włączeniu zasilania	Porty wyjściowe są wyzerowane	Rejestrowe porty wyjściowe wyzerowane; kombinacyjne porty wyjściowe są równe wynikowi odczytu pamięci
Reset asynchroniczny	Rejestry wyjściowe i zatraski wyjściowe są wyzerowane	Rejestry wyjściowe i zatraski wyjściowe są wyzerowane
Inicjowanie operacji odczytu i zapisu	Na dodatnim zboczu sygnału synchronizacji	Na dodatnim zboczu sygnału synchronizacji
Tryb odczytu podczas zapisywania tego samego portu	Porty wyjściowe są ustawione na „new data” lub „don't care”	Porty wyjściowe są ustawione na „don't care”
Tryb odczytu podczas zapisywania portu mieszanego	Porty wyjściowe są ustawione na „old data” lub „don't care”	Porty wyjściowe są ustawione na „old data”, „new data”, „don't care” lub „constrained don't care”
Obsługa sprawdzania i korekcji błędów (error checking and correcting – ECC)	Obsługiwane przez oprogramowanie IP w systemie Quartus	Obsługiwane przez oprogramowanie IP w systemie Quartus

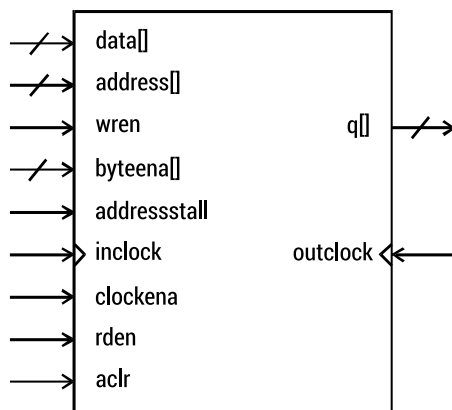
Przed zaprojektowaniem pamięci systemu wbudowanego zaleca się dokładne przestudiowanie typów pamięci wbudowanej rodziny FPGA, na której planowane jest wdrożenie projektu. Projektant jest proszony o określenie głównych cech pamięci wbudowanej używanej rodziny FPGA.

1.3.3. Tryby pracy pamięci wbudowanej

Istnieją następujące tryby pracy pamięci wbudowanej:

- *jednoportowa pamięć typu RAM* (single-port RAM);
- *prosta dwuportowa pamięć typu RAM* (simple dual-port RAM);
- *rzeczywista dwuportowa pamięć typu RAM* (true dual-port RAM);
- *rejestr przesuwający* (shift-register);
- *pamięć tylko do odczytu* (read only memory – ROM);
- *pamięć typu FIFO* (first-in, first-out – pierwsze weszło, pierwsze wyszło).

W trybie jednoportowym pamięci typu RAM (rys. 1.1) w jednym cyklu zegara, można wykonać tylko jedną operację: albo odczyt, albo zapis.



RYS. 1.1. Jednoportowa pamięć typu RAM

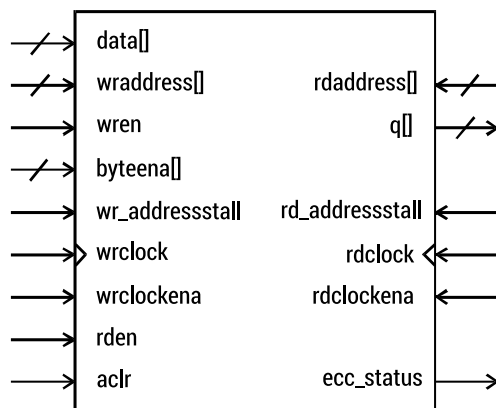
Podczas tworzenia jednoportowej pamięci typu RAM (a także innych typów pamięci) za pomocą Catalog IP systemu Quartus można wygenerować nie wszystkie porty, ale tylko te, które są potrzebne. Jednoportowa pamięć typu RAM może mieć następujące porty wejściowe:

- *data[]* – wejściowa magistrala danych;
- *adres[]* – magistrala adresu;
- *wren* – sygnał zezwolenia na zapis;
- *byteena[]* – magistrala zezwolenia dla bajtów zapisywanych danych;
- *addressstall* – sygnał zezwolenia na synchronizację adresu;
- *inclock* – sygnał zegarowy portów wejściowych;
- *outclock* – sygnał zegarowy portów wyjściowych;
- *clockena* – sygnał zezwolenia synchronizacji;
- *rden* – sygnał zezwolenia na odczyt;
- *aclr* – asynchroniczny sygnał resetowania;
- *q[]* – magistrala danych wyjściowych.

Zauważmy, że jednoportowa pamięć typu RAM ma tylko jedną magistralę adresową *address[]*, za pomocą której można adresować tylko jedną lokalizację pamięci. Funkcja większości portów wynika z ich nazwy. Jednak poszczególne porty (na przykład *rden*) mogą wykonywać dodatkowe funkcje. Funkcje każdego portu zostaną wyjaśnione po dalszym przedstawieniu architektury pamięci wbudowanej.

Gdy pamięć wbudowana działa w trybie dwuportowym, porty są oznaczone jako *port A* i *port B*.

W trybie prostej dwuportowej pamięci typu RAM (rys. 1.2) można jednocześnie wykonać jedną operację odczytu i jedną operację zapisu w różnych komórkach pamięci, na przykład operację zapisu na porcie A i operację odczytu na porcie B.

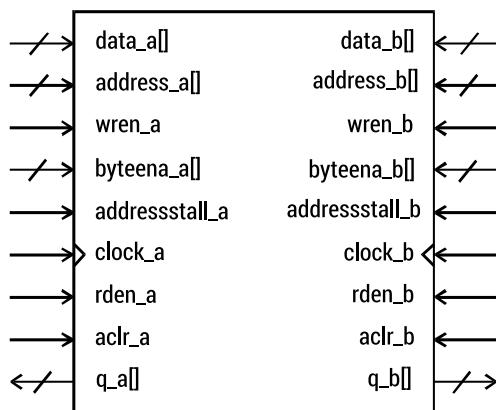


RYS. 1.2. Prosta dwuportowa pamięć typu RAM

Prosta dwuportowa pamięć typu RAM może mieć następujące porty wejściowe:

- *data[]* – magistrala wejściowych danych;
- *wraddress[]* – magistrala adresu zapisu;
- *rdaddress[]* – magistrala adresu odczytu;
- *wren* – sygnał zezwolenia na zapis;
- *byteena[]* – magistrala zezwolenia bajtów zapisywanych danych;
- *wr_addresstall* – sygnał zezwolenia synchronizacji adresu zapisu;
- *rd_addresstall* – sygnał zezwolenia synchronizacji adresu odczytu;
- *wrclock* – sygnał zegarowy zapisu;
- *rdclock* – sygnał zegarowy odczytu;
- *wrclockena* – sygnał zezwolenia synchronizacji zapisu;
- *rdclockena* – sygnał zezwolenia synchronizacji odczytu;
- *rden* – sygnał zezwolenia na odczyt;
- *aclr* – asynchroniczny sygnał resetowania;
- *q[]* – magistrala danych wyjściowych;
- *ecc_status[]* – magistrala sprawdzenia stanu i korekcji błędów.

W trybie rzeczywistym dwuportowym pamięci typu RAM (rys. 1.3) można wykonać dowolną kombinację dwóch operacji na portach A i B: dwie operacje odczytu, dwie operacje zapisu lub jedną operację odczytu i jedną operację zapisu przy dwóch różnych częstotliwościach sygnału zegarowego. Zauważmy, że bloki pamięci wbudowanej M10K mają dwa porty fizyczne: port A i port B. Ten tryb nie jest obsługiwany przez bloki MLAB.

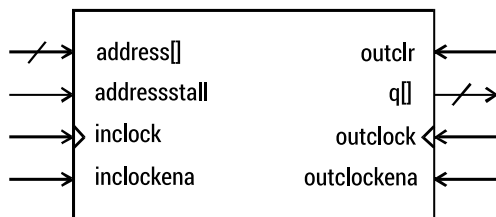


RYS. 1.3. Rzeczywista dwuportowa pamięć typu RAM

Rzeczywista dwuportowa pamięć typu RAM może mieć następujące porty:

- *data_a[], data_b[]* – magistrale danych wejściowych;
- *adres_a[], adres_b[]* – magistrale adresowe;
- *wren_a, wren_b* – sygnały zezwolenia na zapis;
- *byteena_a[], byteena_b[]* – magistrale zezwolenia bajtów zapisywanych danych;
- *addressstall_a, addressstall_b* – sygnały zezwolenia synchronizacji adresów;
- *clock_a, clock_b* – sygnały zegarowe odpowiednich portów;
- *rden_a, rden_b* – sygnały zezwolenia na odczyt;
- *aclr_a, aclr_b* – sygnały asynchronicznego resetowania;
- *q_a[], q_b[]* – magistrale danych wyjściowych.

W trybie pamięci tylko do odczytu można używać bloków pamięci wbudowanej jako pamięci typu ROM. W trybie jednoportowej pamięci typu ROM (rys. 1.4) można wykonać tylko jedną operację odczytu.



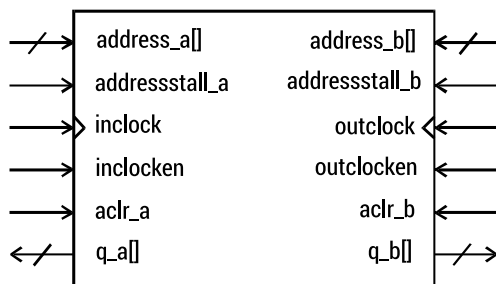
RYS. 1.4. Jednoportowa pamięć typu ROM

Jednoportowa pamięć typu ROM może mieć następujące porty:

- *adres[]* – magistrala adresu na odczyt;
- *addressstall* – sygnał zezwolenia synchronizacji adresu;
- *inclock* – sygnał zegarowy portów wejściowych;
- *inclockena* – sygnał zezwolenia synchronizacji portów wejściowych;

- *outclock* – sygnał zegarowy portu wyjściowego;
- *outclocken* – sygnał zezwolenia synchronizacji portu wyjściowego;
- *outclr* – sygnał resetowania rejestru portu wyjściowego;
- *q[]* – magistrala danych wyjściowych.

Podobnie jak pamięć typu RAM, pamięć typu ROM może mieć również dwa porty (rys. 1.5).



RYŚ. 1.5. Dwuportowa pamięć typu ROM

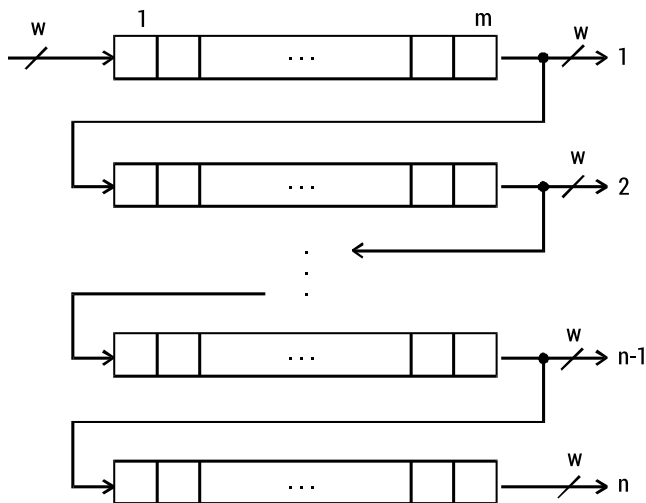
Dwuportowa pamięć typu ROM może mieć następujące porty:

- *adres_a[], adres_b[]* – magistrale adresowe;
- *addressstall_a, addressstall_b* – sygnały zezwolenia synchronizacji adresów;
- *inclock* – sygnał zegarowy portu wejściowego;
- *inclocken* – sygnał zezwolenia synchronizacji portów wejściowych;
- *outclock* – sygnał zegarowy portu wyjściowego;
- *outclocken* – sygnał zezwolenia synchronizacji portu wyjściowego;
- *aclr_a, aclr_b* – sygnały asynchronicznego resetowania;
- *q_a[], q_b[]* – magistrale danych wyjściowych.

Pamięć typu ROM musi być zawsze inicjowana przy użyciu plików *.mif* lub *.hex*. Magistrale adresowe pamięci typu ROM są rejestrowe w blokach M10K lub kombinacyjne w blokach MLAB. Kombinacyjna magistrala adresowa bloków MLAB jest obsługiwana tylko dla trybu prostej dwuportowej pamięci typu RAM. Wyjścia pamięci typu ROM mogą być rejestrowe lub kombinacyjne. Rejestry wyjściowe można „resetować” asynchronicznie. Operacja odczytu pamięci typu ROM jest identyczna z operacją odczytu w konfiguracji jednoportowej pamięci typu RAM.

W trybie rejestru przesuwającego można użyć bloków pamięci wbudowanej FPGA jako dużego rejestru przesuwającego (shift register), aby oszczędzić zasoby elementów logicznych i zasoby połączeń FPGA. Ten tryb jest przydatny w aplikacjach DSP, które wymagają przechowywania danych lokalnych, takich jak *filtry ze skończoną odpowiedzią impulsową* (finite impulse response – FIR), *generatory liczb pseudolosowych* (pseudo-random number generators), układach *filtrowania wielokanałowego* (multi-channel filtering), w *funkcjach automatycznej i krzyżowej korelacji* (auto- and cross-correlation functions).

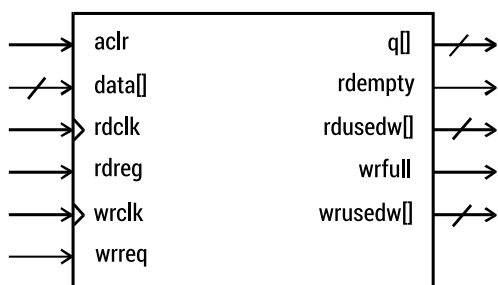
W bloku pamięci wbudowanej rejestr przesuwający magistrali jest realizowany w sposób pokazany na rysunku 1.6.



RYS. 1.6. Rejestr przesuwający magistrali w bloku pamięci wbudowanej

Przy szerokości magistrali w , liczbie odczepów n i długości odczepów m rozmiar zajmowany przez rejestr w pamięci jest równy $(w \times m \times n)$. Aby zaimplementować duże rejestry przesuwające, bloki pamięci wbudowanej FPGA można łączyć kaskadowo.

Bloki pamięci wbudowanej mogą być również używane jako pamięć typu FIFO (rys. 1.7).



RYS. 1.7. Pamięć typu FIFO

Pamięć typu FIFO może mieć następujące porty:

- $data[]$ – magistrala danych wejściowych;
- $q[]$ – magistrala danych wyjściowych;
- $aclr$ – sygnał asynchronicznego resetowania portów wyjściowych;
- $rdclk$ – sygnał zegarowy odczytu, sygnał zegarowy dla portów q , $rdreq$, $rdfull$, $rdempty$, $rdusedw$;

- *wrcclk* – sygnał zegarowy zapisu, sygnał zegarowy dla portów *data*, *wrreq*, *wrfull*, *wrempty*, *wrusedw*;
- *clock* – główny sygnał zegarowy;
- *wrreq* – żądanie operacji zapisu, sygnał nie jest ustawiony, gdy *full*=1 lub *wrfull*=1;
- *rdreq* – żądanie operacji odczytu, sygnał *rdreq* działa inaczej w trybie normalnym i trybie przewidywania (show-ahead mode);
- *full*, *wrfull*, *rdfull* – flagi wskazujące, że pamięć typu FIFO jest pełna;
- *empty*, *wrempty*, *rdempty* – flagi wskazujące, że pamięć typu FIFO jest pusta;
- *almost_full* – sygnał „prawie pełny”, ustawiany jest, gdy wartość sygnału *usedw[]* jest mniejsza niż parametr *almost_full_value*, używany dla wczesnej diagnostyki sygnału *full*;
- *almost_empty* – sygnał „prawie pusty”, ustawiany jest, gdy wartość użytego sygnału *usedw[]* jest mniejsza niż parametr *almost_empty_value*, używany dla wczesnej diagnostyki sygnału *empty*;
- *usedw[]*, *wrusedw[]*, *rdusedw[]* – pokazuje liczbę słów zapisanych w pamięci typu FIFO;
- *eccstatus[]* – dwubitowy port stanu kontroli ECC, może mieć następujące wartości: 00 – brak błędu; 01 – nieprawidłowa kombinacja; 10 – wystąpił błąd; 11 – wystąpił nieusuwalny błąd.

W trybie pamięci typu FIFO można używać bloków pamięci wbudowanej FPGA jako bufora FIFO. Bloki FIFO MLAB są idealne dla projektów zawierających wiele małych buforów FIFO. Jednak bloki MLAB nie obsługują trybu FIFO o *mieszanej szerokości* (mixed-width).

1.3.4. Bity parzystości w blokach pamięci wbudowanej

W blokach M10K *bit parzystości* (parity bit) jest piątym bitem powiązanim z każdymi 4 bitami danych dla słów o szerokości 5, 10, 20 i 40 bitów (bity 4, 9, 14, 19, 24, 29, 34 i 39). W danych bez bitu parzystości, bity parzystości są pomijane podczas operacji odczytu lub zapisu.

W blokach MLAB bit parzystości jest dziewiątym bitem powiązanim z każdym bajtem. Dziewiąty bit może przechowywać bit parzystości lub służyć jako dodatkowy bit.

Bity parzystości mogą być używane jako zwykłe bity danych. W takim przypadku funkcje parzystości w pamięci wbudowanej nie są wykonywane. Zastosowanie bitów parzystości zależy od konfiguracji bloków pamięci wbudowanej.

1.3.5. Tryby synchronizacji pamięci wbudowanej

Pamięć wbudowana FPGA, niezależnie od tego, w którym z bloków (M10K czy MLAB) jest zaimplementowana, obsługuje cztery tryby synchronizacji:

- *synchronizacja pojedyncza* lub *synchronizacja z jednym zegarem* (Single Clock Mode);
- *synchronizacja odczytu/zapisu* (Read/Write Clock Mode);
- *synchronizacja wejścia/wyjścia* (Input/Output Clock Mode);
- *synchronizacja niezależna* (Independent Clock Mode).

Tabela 1.4 pokazuje tryby synchronizacji pamięci wbudowanej, obsługiwane w każdym trybie pracy.

TAB. 1.4. Tryby synchronizacji bloków pamięci wbudowanej

Tryb synchronizacji	Tryb pracy pamięci wbudowanej				
	Single-Port	Simple Dual-Port	True Dual-Port	ROM	FIFO
Single Clock Mode	tak	tak	tak	tak	tak
Read/Write Clock Mode	–	tak	–	–	tak
Input/Output Clock Mode	tak	tak	tak	tak	–
Independent Clock Mode	–	–	tak	tak	–

W trybie synchronizacji z jednym zegarem, sygnał zegarowy *CLK* wraz z sygnałem zezwolenia synchronizacji *CE* (clock enable) steruje wszystkimi rejestrami bloku pamięci.

W trybie synchronizacji odczytu/zapisu dostępne są dwa sygnały synchronizacji: *zegar odczytu* (read clock) i *zegar zapisu* (write clock). Zegar odczytu steruje rejestrami dla *wyjścia danych* (data-output), *adresu odczytu* (read-address) i *zezwolenia na odczyt* (read-enable). Zegar zapisu steruje rejestrami *wejściowych danych* (data-input), *adresu zapisu* (write-address), *zezwolenia na zapis* (write-enable) i *zezwolenia bajtów* (byte enable).

W trybie zegara wejścia/wyjścia dostępne są również dwa sygnały synchronizacji: *zegar wejściowy* (input clock) i *zegar wyjściowy* (output clock). Zegar wejściowy steruje wszystkimi rejestrami związanymi z wprowadzaniem danych do bloku pamięci, w tym rejestrami: danych (data), adresu (address), zezwolenia bajtów (byte enables), zezwolenia na odczyt (read enables) i zezwolenia na zapis (write enables). Zegar wyjściowy steruje rejestrami danych wyjściowych (data output registers).

W trybie niezależnej synchronizacji dostępne są również dwa sygnały zegarowe: sygnał zegarowy portu A (clock A) i sygnał zegarowy portu B (clock B). Sygnał zegarowy portu A steruje wszystkimi rejestrami po stronie portu A, a sygnał zegarowy wyjścia portu B steruje wszystkimi rejestrami po stronie portu B. Aby oszczędzać energię, można utworzyć *niezależne zezwolenie synchronizacji* (independent clock enable), umożliwiające różnym rejestrom wejściowym i wyjściowym sterowanie zegarem określonego rejestru. W tym celu wystarczy wybrać w Edytorze parametrów systemu Quartus, obok opcji **clock enable** zezwolenia synchronizacji, opcję **More Options** i ustawić wymaganą niezależną rozdzielczość synchronizacji.

We wszystkich trybach synchronizacji *czas ustawiania* (setup time) lub *czas utrzymywania* (hold time) rejestrów wejściowych nie może zostać naruszony dla żadnego bloku pamięci wbudowanej. W przeciwnym razie, zawartość pamięci może zostać uszkodzona.

We wszystkich trybach synchronizacji asynchroniczne sygnały „reset” działają tylko dla zatrząsków i rejestrów wyjściowych.

Jeśli konieczne jest jednoczesne wykonanie odczytu i zapisu pod ten sam adres w trybie synchronizacji odczytu/zapisu, dane wyjściowe są nieznane. Jeśli chcemy, aby dane wyjściowe miały znaną wartość, musimy użyć trybu synchronizacji z jednym zegarem lub trybu synchronizacji wejścia/wyjścia i wybrać w Edytorze parametrów odpowiednie zachowanie *odczytu podczas zapisu* (read-during-write).

Zauważmy, że bloki pamięci MLAB obsługują tylko współbieżne operacje odczytu/zapisu podczas pracy w trybie z jednym zegarem.

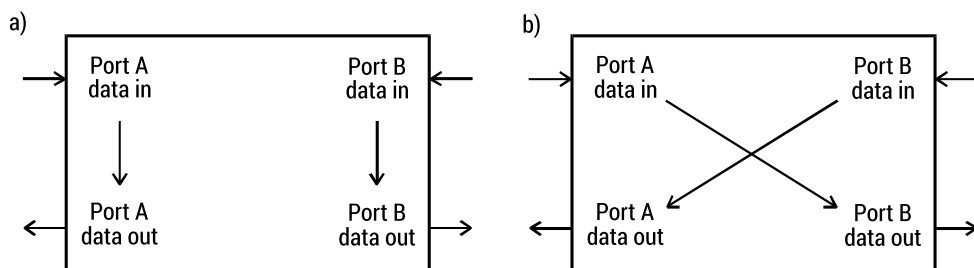
Niezależne zezwolenie synchronizacji jest obsługiwane w następujących trybach synchronizacji:

- w trybie synchronizacji odczytu/zapisu – obsługiwany zarówno zegar do odczytu tak i zegar do zapisu;
- w trybie niezależnej synchronizacji – obsługiwane dla rejestrów obu portów.

Aby oszczędzać energię, należy sterować taktowaniem określonego rejestru za pomocą sygnałów zezwolenia synchronizacji.

1.3.6. Tryby działania wyjść pamięci

Pamięć wbudowana FPGA umożliwia operację *odczytu podczas zapisu* (Read-During-Write). W związku z tym pojawia się pytanie, które dane zostaną odczytane: *stare dane* (old data), *nowe dane* (new data) czy *wartość nieokreślona* (don't care). Istnieją dwa tryby odczytu podczas zapisu: *tryb odczytu podczas zapisu tego samego portu* (Same-Port Read-During-Write Mode) i *mieszany tryb odczytu podczas zapisu portu* (Mixed-Port Read-During-Write Mode). Różnicę między tymi dwoma trybami pokazano na rysunku 1.8.



RYS. 1.8. Strumień danych: a – w trybie odczytu podczas zapisywania tego samego portu; b – w trybie mieszanym odczytu podczas zapisu portu

W trybie odczytu, podczas zapisu tego samego portu, dane mogą być odbierane na wejściu danych portu A i odczytywane z wyjścia danych tylko portu A (rys. 1.8a). To samo zachowanie dla portu B. W trybie mieszanym odczytu podczas zapisywania portu (rys. 1.8b), dane mogą być odbierane na wejściu danych portu A i odczytywane z wyjścia danych portu B lub dane mogą być odbierane na wejściu danych portu B oraz odczytane z wyjścia danych portu A.

Tryb odczytu podczas zapisu tego samego portu jest stosowany do jednoportowej pamięci typu RAM lub do tego samego portu w rzeczywistej dwuportowej pamięci typu RAM. Cechy tego trybu dla różnych bloków pamięci przedstawiono w tabeli 1.5.

TAB. 1.5. Wartości wyjść pamięci wbudowanej w trybie odczytu podczas zapisu tego samego portu

Tryb wyjścia	Blok pamięci FPGA	Opis
„new data” (flow-through)	M10K	Nowe dane są dostępne na rosnącym zboczach zegara tego samego cyklu, w którym nowe dane są zapisywane. Nowa wartość na wyjściach jest możliwa tylko w trybie strumieniowym
„don't care”	M10K, MLAB	Wartości wyjść pamięci typu RAM są nieznane

Jak widać z tabeli 1.5, w trybie odczytu przy zapisywaniu tego samego portu na wyjściach pamięci, możliwe są tylko dwie wartości: nowe dane i nieznana wartość, a nowe dane są możliwe tylko na wyjściach bloków M10K.

W jednoportowej pamięci typu RAM port umożliwiający odczyt *rden* (read-enable port) służy do sterowania portami wyjściowymi w trybie odczytu podczas zapisu. Aby zachować poprzednie wartości wyjściowe, należy wykonać operację zapisu z wyzerowanym sygnałem zezwolenia na odczyt *rden*. Aby pokazać nowo zapisane dane, nie należy generować sygnału *rden* lub ustawiać sygnału *rden*.

Tryb mieszany odczytu podczas zapisu portu dotyczy dwuportowych trybów pamięci typu RAM, w których dwa porty wykonują operacje odczytu i zapisu pod tym samym adresem pamięci przy użyciu tego samego zegara: jeden port odczytuje adres, a drugi zapisuje do niego. Cechy tego trybu dla różnych bloków pamięci przedstawiono w tabeli 1.6.

TAB. 1.6. Tryby wyjściowe dla bloków pamięci wbudowanej w trybie mieszanym do odczytu podczas zapisu portu

Tryb wyjścia	Typ pamięci	Opis
„new data”	MLAB	Rejestrowe wyjście MLAB ustawia nowe dane na następnym narastającym zboczach zegara po zapisaniu danych do pamięci. Ten tryb jest dostępny tylko wtedy, gdy wyjście jest rejestrowe

Tryb wyjścia	Typ pamięci	Opis
„old data”	M10K, MLAB	Wyjście pamięci typu RAM zachowuje starą wartość danych. W przypadku MLAB ten tryb jest dostępny tylko wtedy, gdy wyjście jest rejestrowe
„don't care”	M10K, MLAB	Wartości na wyjściach RAM są nieznane, gdy: - dla bloku M10K Quartus nie analizuje synchronizacji pomiędzy zapisem i odczytem; - dla bloku MLAB Quartus analizuje czas pomiędzy zapisem i odczytem (domyślnie)
„constrained don't care” (ograniczona wartość nieznana)	MLAB	Wartości na wyjściach RAM są nieznane. Oprogramowanie Quartus analizuje czas między zapisem i odczytem w MLAB

Zauważmy, że domyślnie system Quartus analizuje czas pomiędzy zapisami i odczytami w blokach MLAB. Aby wyłączyć to zachowanie, włącz opcję **„Do not analyze the timing between write and read operation. Metastability issues are prevented by never writing and reading at the same address at the same time”** (Nie analizuj czasu między operacjami zapisu i odczytu). Problemom z metastabilnością należy zapobiegać poprzez niepisanie i nieczytanie spod tego samego adresu w tym samym czasie).

Z tabeli 1.6 widać, że w trybie mieszanym odczytu, podczas zapisu portu, możliwe są cztery tryby wyjść:

- „new data” – nowe dane;
- „old data” – stare dane;
- „don't care” – wartość nieznana;
- „constrained don't care” – ograniczona wartość nieznana.

Nowe dane można odczytać tylko na wyjściach bloków MLAB. Na wyjściach bloków MLAB i M10K możliwe są stare dane i wartość nieznana. Możliwy jest również tryb wyjść o *ograniczonej nieznanej wartości* (constrained don't care), który występuje dla wyjść bloków MLAB. Wartości wyjść pamięci są nieznane w tym trybie, jednak oprogramowanie Quartus analizuje czas między zapisami i odczytami w MLAB.

1.3.7. Konfiguracja portu mieszanego

Konfiguracja portu o różnej szerokości lub mieszanego (Mixed-Width Port Configuration) oznacza, że port zapisu (write port) może mieć jedną szerokość magistrali danych, a port odczytu (read port) może mieć inną. Bloki MLAB nie obsługują konfiguracji portu o różnej szerokości. W przypadku bloków M10K konfiguracje portów o różnej szerokości są obsługiwane w trybach prostej dwuportowej pamięci typu RAM i rzeczywistej dwuportowej pamięci typu RAM. W tabeli 1.7 pokazano możliwe parametry portów odczytu i zapisu w trybie prostej dwuportowej pamięci RAM oraz tabeli 1.8 – w trybie rzeczywistej dwuportowej pamięci typu RAM.

TAB. 1.7. Konfiguracje portów o różnej szerokości dla jednostek M10K w trybie prostej dwuportowej pamięci typu RAM

Read Port	Write Port									
	8Kx1	4Kx2	2Kx4	2Kx5	1Kx8	1Kx10	512x16	512x20	256x32	256x40
8Kx1	tak	tak	tak	–	tak	–	tak	–	tak	–
4Kx2	tak	tak	tak	–	tak	–	tak	–	tak	–
2Kx4	tak	tak	tak	–	tak	–	tak	–	tak	–
2Kx5	–	–	–	tak	–	tak	–	tak	–	tak
1Kx8	tak	tak	tak	–	tak	–	tak	–	tak	–
1Kx10	–	–	–	tak	–	tak	–	tak	–	tak
512x16	tak	tak	tak	–	tak	–	tak	–	tak	–
512x20	–	–	–	tak	–	tak	–	tak	–	tak
256x32	tak	tak	tak	–	tak	–	tak	–	tak	–
256x40	–	–	–	tak	–	tak	–	tak	–	tak

W trybie prostej dwuportowej pamięci typu RAM porty odczytu i zapisu M10K można skonfigurować jako 8Kx1, 4Kx2, 2Kx4, 2Kx5, 1Kx8, 1Kx10, 512x16, 512x20, 256x32 i 256x40. Jednak nie każda konfiguracja portu zapisu odpowiada każdej konfiguracji portu odczytu i na odwrót. Pokazano to w tabeli 1.7.

TAB. 1.8. Konfiguracje portów o różnej szerokości dla jednostek M10K w trybie rzeczywistej dwuportowej pamięci typu RAM

Port B	Port A							
	8Kx1	4Kx2	2Kx4	2Kx5	1Kx8	1Kx10	512x16	512x20
8Kx1	tak	tak	tak	–	tak	–	tak	–
4Kx2	tak	tak	tak	–	tak	–	tak	–
2Kx4	tak	tak	tak	–	tak	–	tak	–
2Kx5	–	–	–	tak	–	tak	–	tak
1Kx8	tak	tak	tak	–	tak	–	tak	–
1Kx10	–	–	–	tak	–	tak	–	tak
512x16	tak	tak	tak	–	tak	–	tak	–
512x20	–	–	–	tak	–	tak	–	tak

W trybie rzeczywistej dwuportowej pamięci typu RAM porty odczytu i zapisu bloku M10K mogą mieć następujące konfiguracje: 8Kx1, 4Kx2, 2Kx4, 2Kx5, 1Kx8, 1Kx10, 512x16, 512x20. Konfiguracje 256x32 i 256x40 nie są tutaj możliwe.

1.3.8. Zezwolenie na zapis bajtów danych

Pamięć wbudowana ma jeszcze jedną ciekawą funkcję: *zezwolenie na zapis bajtów danych* (Byte Enable). Ta funkcja steruje maską danych wejściowych w taki sposób, aby zapisywane były tylko określone bajty danych. Bajty, które nie są zapisywane, pozostają niezmienione. Zapis do bloków pamięci wbudowanej jest sterowany przez dwa sygnały: *zezwolenie na zapis (wren)* i *zezwolenie na bajt (byteena)*. Domyślnie sygnał *byteena* ma aktywny stan wysoki. Sygnał *wren* jest wspólny dla całego bloku pamięci, a sygnały *byteena* są określone dla każdego bajtu zapisywanych danych, tworząc maskę zapisywanych danych. Zauważmy, że sygnałów *byteena* nie można wyzerować, ponieważ rejestry sygnałów *byteena* nie mają portu resetowania.

W blokach MLAB w najszerszym trybie, funkcja zezwolenia bajtów steruje wszystkimi 10 bitami. Jeśli w blokach M10K są używane bity parzystości, funkcja zezwolenia bajtów steruje 8 bitami danych i 2 bitami parzystości. Tabele 1.9 i 1.10 pokazują wartości sygnałów *byteena* i sterowanie zapisem bajtów dla szerokości danych, odpowiednio 20 i 40 bitów.

TAB. 1.9. Wartości sygnałów *byteena* i zapisywane bajty dla szerokości danych 20 bitów

byteena[1:0]	Zapisywane bity danych	
11	[19:10]	[9:0]
10	[19:10]	–
01	–	[9:0]

TAB. 1.10. Wartości sygnałów *byteena* i zapisywane bajty dla szerokości danych 40 bitów

byteena[3:0]	Zapisywane bity danych			
1111	[39:30]	[29:20]	[19:10]	[9:0]
1000	[39:30]	–	–	–
0100	–	[29:20]	–	–
0010	–	–	[19:10]	–
0001	–	–	–	[9:0]

W blokach M10K, dane wyjściowe zamaskowanych bajtów mają wartość „*don't care*”. W blokach MLAB dane wyjściowe zamaskowanych bajtów mają albo wartość „*don't care*”, albo wartość odpowiadającą danym z pamięci. W tym przypadku wartość zamaskowanych bajtów dla bloków MLAB jest sterowana przez oprogramowanie Quartus.

1.3.9. Obsługa trybu spakowanego

Tryb spakowany (packed mode) jest obsługiwany tylko przez bloki pamięci M10K. Funkcja pakowania umożliwia zapakowanie dwóch niezależnych jednoportowych

jednostek pamięci typu RAM do jednej jednostki pamięci. Oprogramowanie Quartus automatycznie implementuje tryb spakowany, gdy jest to konieczne, umieszczając fizyczny blok pamięci typu RAM w trybie rzeczywistym dwuportowym i wykorzystuje najbardziej znaczący bit adresu do rozróżnienia między dwoma logicznymi blokami pamięci typu RAM. W takim przypadku rozmiar każdej niezależnej jednoportowej pamięci typu RAM nie powinien przekraczać połowy rozmiaru bloku docelowego. Przypominamy, że bloki MLAB nie obsługują trybu spakowanego.

1.3.10. Zezwolenie synchronizacji adresu

Bloki pamięci wbudowanej obsługują *zezwolenie synchronizacji adresu* (address clock enable), które pozwala zachować poprzednią wartość adresu, dopóki sygnał *address-stall* = 1. Gdy bloki pamięci są skonfigurowane w trybie dwuportowym, każdy port ma swój własny sygnał zezwolenia synchronizacji adresu. Domyślnie *addressstall* = 0.

1.4. Zalecenia dotyczące projektowania pamięci wbudowanej

Oto kilka zaleceń firmy Intel dotyczących projektowania pamięci wbudowanej.

1.4.1. Wybór bloku pamięci

Oprogramowanie Quartus automatycznie przydziela pamięć zdefiniowaną przez użytkownika do bloków pamięci wbudowanej, na podstawie wymagań dotyczących szybkości działania projektu określonych przez ograniczenia czasowe. Na przykład, aby zwiększyć szybkość działania projektu, system Quartus może rozdzielić pamięć na wiele bloków pamięci.

Użytkownik może samodzielnie przypisać określony rozmiar do modułu pamięci i określić, w którym bloku pamięci wbudowanej ma on zostać zaimplementowany. W tym celu trzeba użyć IP-bloku **RAM: 1-PORT** lub **RAM: 2-PORT** w menu **IP Catalog** systemu Quartus i dokonać odpowiednich przypisań (co omówiono w rozdziale 3).

Ze względu na dwufunkcyjną architekturę bloków MLAB (do implementacji logiki i pamięci), w bloku MLAB dostępne są tylko: wejście danych i rejestry wyjściowe. Rejestry adresowe odczytu dla bloków MLAB są zewnętrzne i zostały zaimplementowane w adaptacyjnych modułach logicznych ALM. Jednak rejestry dla adresu zapisu i danych odczytu w blokach MLAB są rejestrami wewnętrznymi.

1.4.2. Zewnętrzne rozwiązywanie konfliktów

W trybie rzeczywistej dwuportowej pamięci typu RAM można wykonać dwa zapisy do tej samej komórki pamięci. Jednak bloki pamięci nie mają wewnętrznego schematu rozwiązywania konfliktów. Aby uniknąć nieznanych danych zapisywanych pod jakimś adresem, należy zaimplementować zewnętrzną logikę, służącą do rozwiązywania konfliktów podczas uzyskiwania dostępu do tej samej komórki pamięci.

1.4.3. Konfigurowanie trybu działania wyjść pamięci

Konfigurowanie trybu działania wyjść pamięci dotyczy określenia trybów pracy wyjść pamięci podczas zapisu (read-during-write).

Zachowanie odczytu podczas zapisu powinno być starannie skonfigurowane, aby odpowiadało wymaganiom projektowym. W tym celu dla każdego bloku pamięci wbudowanej konieczne jest prawidłowe ustawienie trybu odczytu podczas zapisu oraz trybu wyjść bloku pamięci wbudowanej.

1.4.4. Stan i inicjalizacja pamięci po włączeniu zasilania

Jeśli projektowana jest logika, która uwzględnia wartości początkowe po włączeniu zasilania, należy wziąć pod uwagę stan funkcjonalny po włączeniu zasilania dla różnych typów jednostek pamięci, co pokazano w tabeli 1.11.

TAB. 1.11. Początkowe wartości bloków pamięci wbudowanej po włączeniu zasilania

Typ pamięci	Rejestry wyjściowe	Wartość zasilania
MLAB	Używane	Zero
	Nie używane	Zawartość czytanej komórki pamięci
M10K	Używane	Zero
	Nie używane	Zero

Domyślnie, jeśli nie określono pliku .mif do inicjalizacji pamięci, oprogramowanie Quartus dla układu Cyclone V inicjalizuje komórki pamięci jako zera. Wszystkie bloki pamięci obsługują inicjalizację za pomocą pliku .mif. Aby zainicjować pamięć, należy utworzyć pliki .mif za pomocą Quartusa i zdefiniować ich użycie z odpowiednim blokiem IP pamięci typu RAM. Nawet jeśli pamięć jest wstępnie zainicjalizowana (na przykład przy użyciu pliku .mif), po włączeniu zasilania zaczyna działać z resetowanym wyjściem.

1.4.5. Sterowanie synchronizacją w celu zmniejszenia zużycia energii

Zmniejszenie zużycia energii odbywa się poprzez sterowanie taktowaniem każdego bloku pamięci.

Sygnał odczytu *rden* powinien być używany w celu zapewnienia, że operacje odczytu są wykonywane tylko wtedy, gdy jest to konieczne. Jeśli projekt nie wymaga odczytu podczas zapisu, zużycie energii można zmniejszyć, wyłączając sygnał *rden* podczas operacji zapisu lub gdy nie ma dostępu do pamięci.

Aby zmniejszyć statyczne zużycie energii, można użyć oprogramowania Quartus do umieszczenia nieużywanych bloków pamięci w *trybie niskiego poboru mocy* (low-power mode).

1.5. Wnioski

Pamięć systemów wbudowanych jest podzielona na dwie duże klasy: *pamięć ulotna* (volatile memory) i *pamięć nieulotna* (non-volatile memory).

Pamięć ulotna nie zachowuje swojej zawartości po wyłączeniu zasilania. Pamięć nieulotna zachowuje swoją zawartość po wyłączeniu zasilania.

Pamięć wbudowana (embedded memory) jest pamięcią w układzie FPGA. Pamięć wbudowana jest ulotna, więc zapisana w niej informacja będzie utracona po wyłączeniu zasilania.

Pamięć wbudowana nie jest odpowiednia dla aplikacji wymagających dużej pojemności pamięci.

Różne rodziny FPGA zawierają różne typy pamięci wbudowanej. Układy Intel FPGA mogą zawierać następujące bloki pamięci wbudowanej: M510 (512 bitów), MLAB (640 bitów), M4K (4 kbity), M9K (9 kbitów), M10K (10 kbów), M20K (20 kbów), M144K (144 kb), M-RAM (512 kb), eSRAM (18,432 lub 47,25 Mb).

Układy FPGA z rodziny Cyclone V mają dwa typy pamięci wbudowanej: MLAB i M10K.

Pamięć wbudowana współczesnych układów FPGA ma szereg różnych właściwości, których znajomość jest niezbędna do wydajnej implementacji pamięci w systemach wbudowanych.

Bloki pamięci M10K to oddzielne bloki pamięci, które są wbudowane w strukturę FPGA i realizują tylko funkcje pamięci.

Bloki MLAB są konfigurowane z bloków LAB macierzy logicznych o podwójnym przeznaczeniu, które mogą być używane do implementacji logiki lub tworzenia pamięci wbudowanej.

Przed zaprojektowaniem pamięci systemu wbudowanego zaleca się dokładne przestudiowanie typów pamięci wbudowanej używanych w rodzinie FPGA, na której planowana jest realizacja projektu.

Pamięć wbudowana może działać w następujących trybach:

- *jednoportowa pamięć typu RAM* (single-port RAM);
- *prosta dwuportowa pamięć typu RAM* (simple dual-port RAM);
- *rzeczywista dwuportowa pamięć typu RAM* (true dual-port RAM);
- *rejestr przesuwający* (shift-register);
- *pamięć tylko do odczytu* (read only memory – ROM);
- *pamięć typu FIFO* (first-in, first-out – pierwsze weszło, pierwsze wyszło).

Pamięć jednoportowa wykonuje tylko jedną operację na jeden cykl zegara: odczyt lub zapis. Pamięć jednoportowa ma jedną magistralę adresową.

Prosta dwuportowa pamięć może jednocześnie wykonywać jeden odczyt i jeden zapis w różnych komórkach pamięci w jednym cyklu zegara.

Rzeczywista dwuportowa pamięć w jednym cyklu zegara może wykonywać dowolną kombinację dwóch operacji na portach A i B: dwa odczyty, dwa zapisy lub jeden odczyt i jeden zapis przy dwóch różnych częstotliwościach zegara.

W trybie rejestru przesuwającego, pamięć wbudowana FPGA może być używana jako duży rejestr przesuwający magistrali po to, aby oszczędzić zasoby logiczne i zmniejszyć liczbę połączeń.

Pamięć typu ROM musi być zawsze inicjowana plikami .mif lub .hex.

Bloki pamięci wbudowanej zawierają bity parzystości, na przykład w blokach M10K co piąty bit, a w blokach MLAB co dziewiąty bit jest bitem parzystości. Bity parzystości mogą być używane jako zwykłe bity danych. W takim przypadku funkcje parzystości w pamięci wbudowanej nie są wykonywane. Użycie bitów parzystości jest określone przez konfigurację bloków pamięci wbudowanej.

Pamięć wbudowana obsługuje cztery tryby synchronizacji:

- *synchronizację z jednym zegarem* (Single Clock Mode);
- *synchronizację odczytu/zapisu* (Read/Write Clock Mode);
- *synchronizację wejścia/wyjścia* (Input/Output Clock Mode);
- *synchronizację niezależną* (Independent Clock Mode).

Pamięć wbudowana może wykonywać operację *odczytu podczas zapisu* (Read-During-Write). Istnieją dwa tryby odczytu podczas zapisu: *tryb odczytu podczas zapisu tego samego portu* (Same-Port Read-During-Write Mode) i *mieszany tryb odczytu podczas zapisu portu* (Mixed-Port Read-During-Write Mode).

W trybie odczytu podczas zapisu tego samego portu możliwe są następujące wartości wyjść pamięci:

- „new data” – nowe dane dla bloków M10K;
- „don't care” – nieznana wartość dla bloków M10K i MLAB.

W trybie mieszanym odczytu podczas zapisu portu możliwe są następujące wartości wyjść pamięci:

- „new data” – nowe dane, dla bloków MLAB;
- „old data” – stare dane, dla bloków M10K i MLAB;

- „don't care” – nieznana wartość, dla bloków M10K i MLAB;
- „constrained don't care” – ograniczona nieznana wartość, dla bloków MLAB.

Konfiguracja portu o różnej szerokości (Mixed-Width Port Configuration) oznacza, że port zapisu (write port) może mieć jedną szerokość danych, a port odczytu (read port) może mieć inną.

Zezwolenie bajtów (Byte Enable) steruje maską danych wejściowych po to, aby można było zapisywać tylko określone bajty danych.

Tryb spakowany (packed mode) umożliwia spakowanie dwóch niezależnych jednoportowych jednostek pamięci typu RAM w jeden blok M10K.

Zezwolenie na synchronizację adresu (address clock enable) pozwala na zachowanie wartości adresu tak długo, dopóki sygnał *addressstall* = 1.

Zalecenia firmy Intel dotyczące projektowania pamięci wbudowanej są następujące:

- moduł pamięci wbudowanej o wymaganych właściwościach architektury można wygenerować za pomocą katalogu IP w edytorze parametrów systemu Quartus;
- bloki pamięci wbudowanej FPGA nie mają mechanizmu rozwiązywania konfliktów w przypadku jednoczesnego zapisu do tej samej komórki pamięci, dlatego takie konflikty muszą być rozwiązywane za pomocą logiki zewnętrznej;
- należy poprawnie ustawić tryb odczytu podczas zapisu oraz tryb wyjść bloków pamięci wbudowanej;
- należy wziąć pod uwagę stany wyjść modułów pamięci wbudowanej po włączeniu zasilania;
- aby zmniejszyć pobór mocy bloków pamięci wbudowanej, należy użyć sygnałów *zezwolenia synchronizacji* (clock enable), a także sygnałów *zezwolenia odczytu* *rden* (read enable);
- aby zmniejszyć statyczne zużycie energii, można użyć oprogramowania Quartus, aby ustawić nieużywane bloki pamięci FPGA w *tryb niskiego poboru mocy* (low-power mode).

2. Opis pamięci wbudowanej w języku Verilog

W tym rozdziale omówiono projektowanie pamięci wbudowanej przy użyciu języka Verilog. Na przykładzie prostej pamięci jednoportowej podano różne sposoby opisu pamięci. Zwrócono uwagę na cechy opisu pamięci dwuportowej i pamięci typu ROM. Osobno jest rozpatrywana kwestia inicjalizacji pamięci. Bloki pamięci wbudowanej FPGA umożliwiają skuteczną implementację rejestrów przesuwających magistrali z odczepami używanymi w aplikacjach cyfrowego przetwarzania sygnałów (digital signal processing – DSP). Opisano metody projektowania w języku Verilog pamięci typu FIFO (First-In, First-Out) i LIFO (Last-In, First-Out). Na końcu rozdziału przedstawiono atrybuty syntezy opisujące pamięć w języku Verilog.

2.1. Metody opisu pamięci na przykładzie jednoportowej pamięci typu RAM (Single-Port RAM) z jednym sygnałem synchronizacji

Przyjrzyjmy się sposobom opisu pamięci na przykładzie pamięci jednoportowej. Przypomnijmy sobie, że pamięć jednoportowa może wykonywać tylko jedną operację: odczyt lub zapis w jednym cyklu zegara.

Rozpatrzmy pamięć jednoportową, która ma jedną magistralę wejściową *data_in*, jedną magistralę wyjściową *data_out* i jedną magistralę adresową *address*. W tym przykładzie magistrala adresowa jest używana zarówno do zapisu, jak i odczytu danych. W naszym przypadku do zarządzania pamięcią wystarczą dwa sygnały: *we* – dla zezwolenia zapisu (write enable) i *clk* – dla synchronizacji operacji odczytu i zapisu. Gdy *we* = 1, dane są zapisywane z magistrali *data_in* do komórki pamięci, której adres jest ustawiony na magistrali *address*, gdy *we* = 0, zapis do pamięci jest zabroniony i dane są tylko odczytywane z komórki pamięci wskazanej przez adres na magistrali *address*.

2.1.1. Deklaracja macierzy pamięci

Niezależnie od liczby portów i liczby sygnałów zegarowych, macierz pamięci w języku Verilog jest zadeklarowana jako jednowymiarowa tablica zmiennych *n*-bitowych typu **reg**, na przykład:

```
reg [7:0] mem[31:0];
```

Powyżej zadeklarowano tablicę *mem*, która zawiera 32 zmienne o rozmiarze 8 bitów.

Pamięć w projektach Verilog jest zwykle opisywana za pomocą osobnego modułu. W tym przypadku wygodnie jest określić rozmiar przestrzeni adresowej pamięci na podstawie szerokości magistrali adresowej, a rozmiar każdego słowa pamięci – przez szerokość magistrali danych. Aby to zrobić, można użyć parametrów modułu, na przykład poprzednio zadeklarowaną pamięć można opisać w następujący sposób:

```
module <nazwa_modułu>
```

```
#(parameter DATA_WIDTH=8, parameter ADDR_WIDTH=5)
```

```
...
```

```
reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];
```

gdzie DATA_WIDTH – szerokość magistrali danych; ADDR_WIDTH – szerokość magistrali adresowej.

2.1.2. Tryby działania wyjść pamięci

Podczas implementacji pamięci w układach FPGA firmy Intel, często pozwala się na wysyłanie zapisanych danych do wyjść pamięci w tym samym cyklu zegara. Ten tryb działania wyjść pamięci nazywany jest „*nowymi danymi*” («new data»). Jeżeli poprzednia wartość jest zachowana na wyjściach pamięci podczas zapisu, ten tryb działania wyjść pamięci jest nazywany „*starymi danymi*” («old data»). W niektórych przypadkach dozwolone są również tryby pracy wyjść pamięci „*wartość nieznana*” («don't care»), gdy wartość wyjść pamięci jest nieznana podczas zapisu, oraz „*ograniczona wartość nieznana*” («constrained don't care»), gdy wartość wyjść pamięci jest nieznana podczas zapisu, jednak program Quartus analizuje synchronizację pamiędzi operacjami zapisu i odczytu w blokach MLAB.

2.1.3. Opis pamięci ze „starymi danymi” na wyjściu

Opis w języku Verilog jednoportowej pamięci ze „starymi danymi” wyjściowymi może wyglądać następująco:

```
module single_port_ram_single_clk
```

```
#(parameter DATA_WIDTH=8, parameter ADDR_WIDTH=5)
```

```
(input clk, we,
```

```
input [DATA_WIDTH-1:0] data_in,
```

```
input [ADDR_WIDTH-1:0] address,
```

```

output reg [DATA_WIDTH-1:0] data_out);

// deklaracja macierzy pamięci
reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];

always @ (posedge clk)
begin
    if (we)
        ram[address] <= data_in;    // zapis
        data_out <= ram[address];    // odczyt
end
endmodule

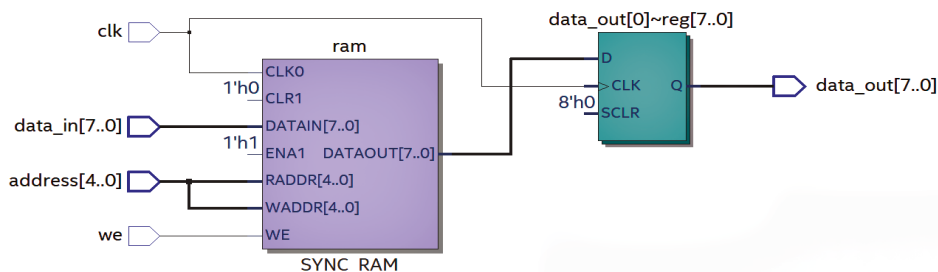
```

Tutaj działanie pamięci jest opisane za pomocą pojedynczego procesu **always**, który zaczyna działać, gdy nadejdzie narastające zbocze sygnału zegarowego *clk* (**posedge** *clk*). Jeśli zapis jest włączony (*we* = 1), to dane z magistrali *data_in* są zapisywane pod adres komórki pamięci *address*. Zauważmy, że dla dowolnej wartości sygnału *we*, zawartość komórki pamięci pod adresem *address* zostanie wygenerowana na wyjściu *data_out*. Ponieważ wartości wyjściowe *data_out* są przypisywane w bloku **always**, zmienna *data_out* jest zadeklarowana jako **reg**.

Statystyki wyników kompilacji (rys. 2.1) pokazują, że do implementacji pamięci wymagany jest jeden blok logiczny ALM, a macierz pamięci jest zaimplementowana w bloku pamięci wbudowanej FPGA i zajmuje 256 bitów. Wyniki syntezy projektu *single_port_ram_single_clk* przedstawiono na rysunku 2.2.

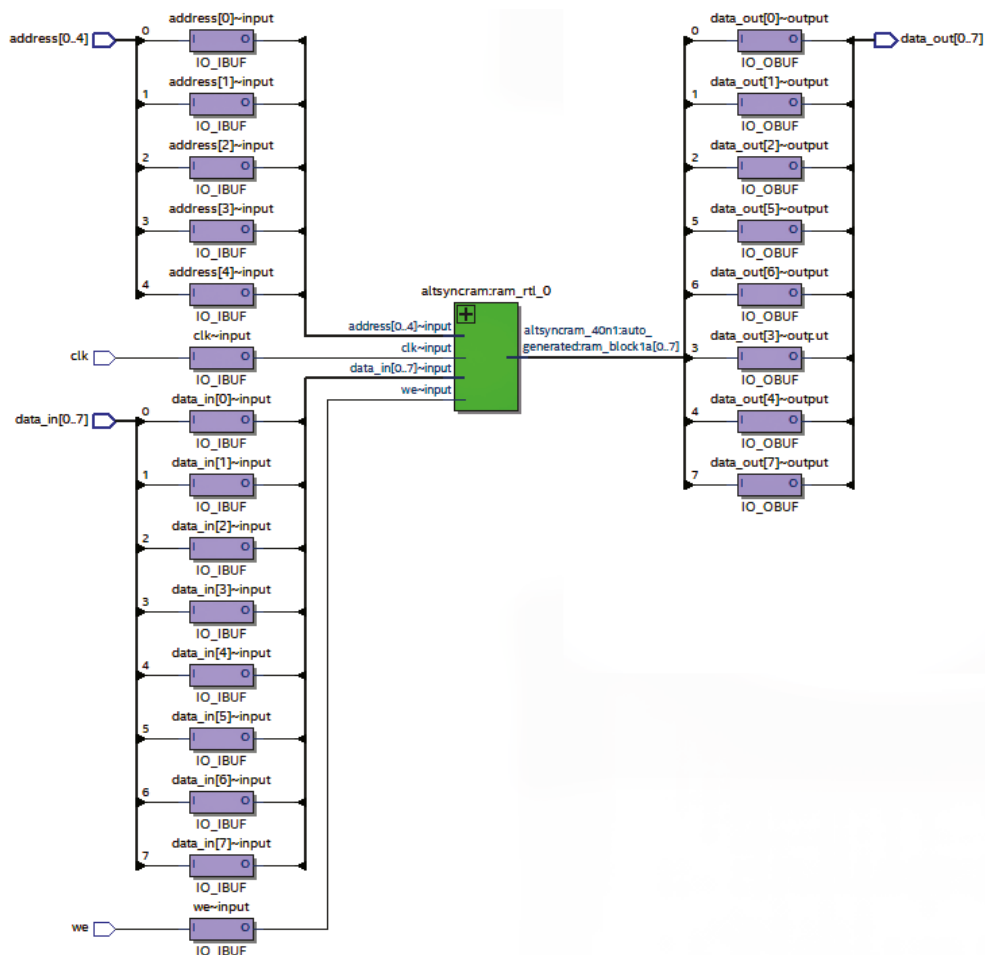
Logic utilization (in ALMs)	1 / 56,480 (< 1 %)
Total registers	0
Total pins	23 / 268 (9 %)
Total virtual pins	0
Total block memory bits	256 / 7,024,640 (< 1 %)

RYS. 2.1. Statystyki kompilacji dla projektu *single_port_ram_single_clk*



RYS. 2.2. Widok projektu *single_port_ram_single_clk* na poziomie RTL

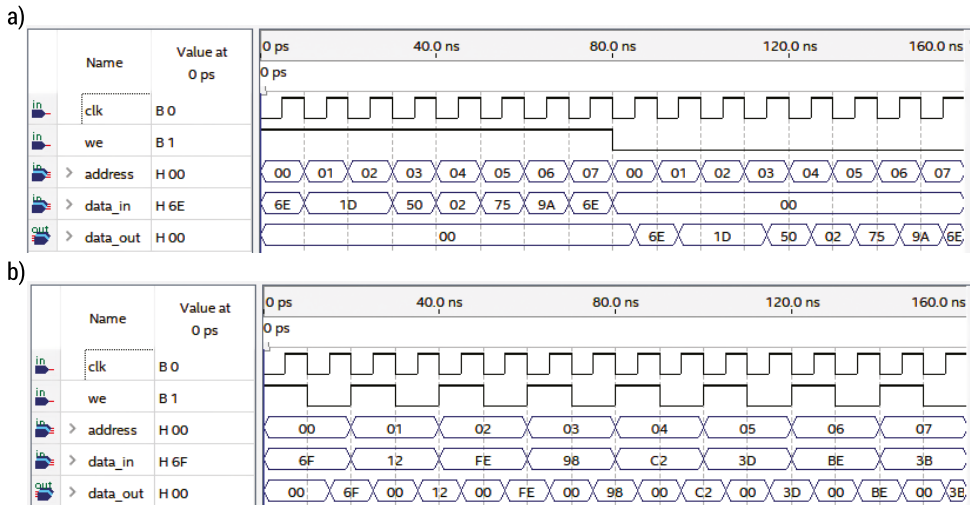
Analiza rysunku 2.2 pokazuje, że pamięć jest zaimplementowana przy użyciu prymitywu SYNC_RAM systemu Quartus, a na wyjściach pamięci ustawiany jest rejestr. Ponieważ pamięć zaimplementowana w blokach pamięci wbudowanej FPGA jest synchroniczna, rejestr wyjściowy na rysunku 2.2 jest zaimplementowany w bloku RAM i nie wymaga dodatkowych zasobów logicznych. Widać to na mapie technologicznej projektu (rys. 2.3).



RYŚ. 2.3. Mapa technologiczna projektu *single_port_ram_single_clk*

Wyniki symulacji modułu *single_port_ram_single_clk* pokazano na rysunku 2.4. Rysunek 2.4a pokazuje, że przy zapisie do pamięci stan sygnału na wyjściu jest równy zeru, a odczyt danych z pamięci następuje na dodatnim zboczach sygnału zegarowego *clk*. Podczas naprzemiennego zapisu i odczytu pod tym samym adresem (rys. 2.4b) zera na wyjściu występują na przemian z odczytanymi wartościami. Wynika to z użycia nieblokującego operatora przypisania („<=”). W przypadku zapisu

do pamięci ($we = 1$) wartość $ram[address]$ jest nieznana, ale wymagane jest przypisanie tej wartości do wyjścia bez blokowania (instrukcja $data_out \leq ram[address]$), więc kompilator generuje zera na wyjściu. Ten sposób opisu pamięci jest zalecany przez firmę Intel do implementacji pamięci ze „starymi danymi” wyjściowymi.



RYS. 2.4. Wyniki symulacji projektu *single_port_ram_single_clk*: a – w przypadku oddzielnych operacji zapisu i odczytu; b – w przypadku naprzemiennych operacji odczytu i zapisu

Tryb wyjścia pamięci „stare dane” można jawnie zaimplementować w następującym kodzie:

```

module single_port_ram_single_clk_v2
#(parameter DATA_WIDTH=8, parameter ADDR_WIDTH=5)
(input clk, we,
input [DATA_WIDTH-1:0] data_in,
input [ADDR_WIDTH-1:0] address,
output reg [DATA_WIDTH-1:0] data_out);

// deklaracja macierzy pamięci
reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];

always @ (posedge clk)
begin
    if (we) // zapis
        ram[address] <= data_in;
    else // odczyt

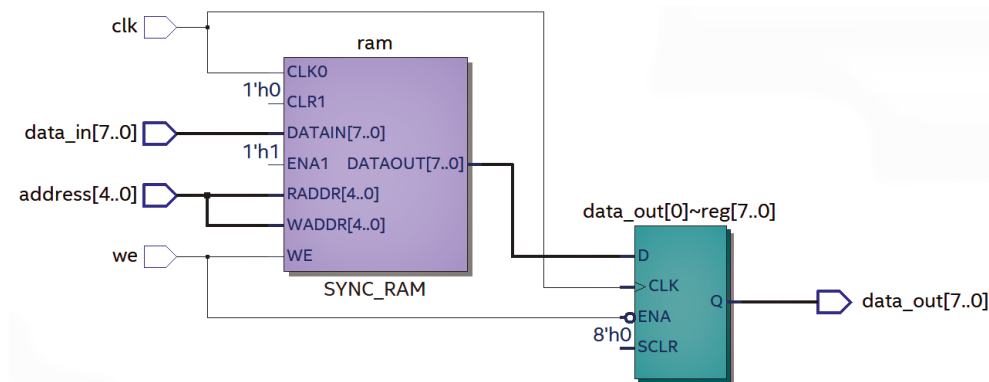
```

```
data_out <= ram[address];
```

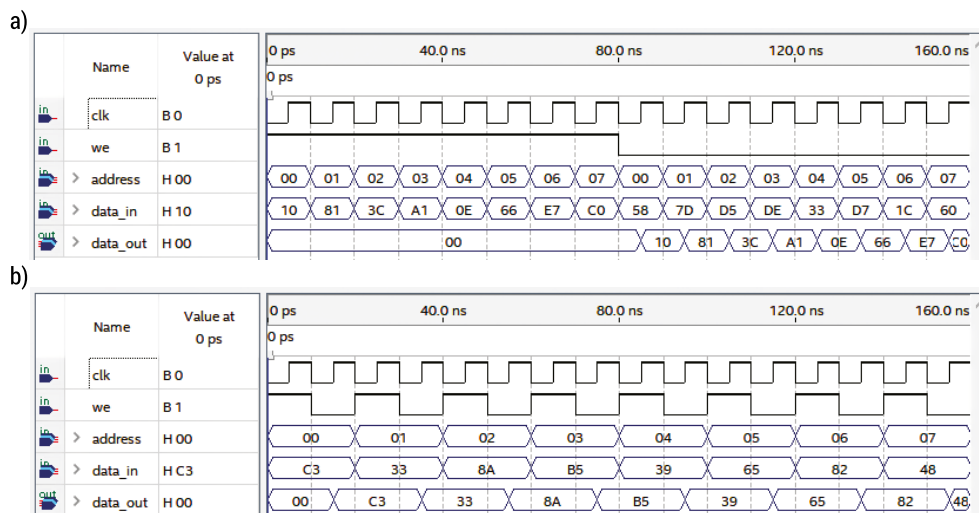
```
end
```

```
endmodule
```

Tutaj w jednym cyklu zegarowym, w zależności od wartości sygnału *we*, wykonywana jest tylko jedna operacja: zapis lub odczyt. Wyniki syntezy i symulacji projektu *single_port_ram_single_clk_v2* przedstawiono, odpowiednio, na rysunkach 2.5 i 2.6.



RYS. 2.5. Widok projektu *single_port_ram_single_clk_v2* na poziomie RTL



RYS. 2.6. Wyniki symulacji projektu *single_port_ram_single_clk_v2*: a – w przypadku oddzielnych operacji odczytu i zapisu; b – w przypadku naprzemiennych operacji odczytu i zapisu

Rysunek 2.5 pokazuje, że sygnał *we* dodatkowo steruje wejściem ENA zezwolenia pracy dla wyjściowego rejestru pamięci. W przypadku zapisu umożliwia to zachowanie na wyjściach pamięci wcześniej odczytanych danych, tzn. „starych danych”.

Symulacja projektu *single_port_ram_single_clk_v2* (rys. 2.6) w pełni potwierdza zachowanie wyjść pamięci w trybie „starych danych”. Zwróćmy uwagę, że w przeciwieństwie do rysunku 2.3b, podczas naprzemiennych operacji zapisu i odczytu na wyjściach pamięci, wektory zerowe nie są tworzone. W związku z tym projekt *single_port_ram_single_clk_v2* może być zalecany do implementacji pamięci w trybie wyjścia „stare dane”.

Wniosek. Projekt *single_port_ram_single_clk_v2* działa lepiej w trybie wyjściowym „starych danych” niż zalecany przez firmę Intel projekt *single_port_ram_single_clk*.

2.1.4. Opis pamięci z „nowymi” danymi na wyjściu

Aby można było wyświetlić dane zapisywane na wyjściach pamięci, wystarczy w rozwiązaniu kodzie modułu *single_port_ram_single_clk* zastąpić nieblokujący operator „<=” operatorem blokującym „=”. Ta metoda implementacji pamięci z „nowymi danymi” na wyjściu jest zalecana przez firmę Intel. Na przykład:

```
module single_port_ram_blocking_assign
#(parameter DATA_WIDTH=8, parameter ADDR_WIDTH=5)
(input clk, we,
input [DATA_WIDTH-1:0] data_in,
input [ADDR_WIDTH-1:0] address,
output reg [DATA_WIDTH-1:0] data_out);

// deklaracja macierzy pamięci
reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];

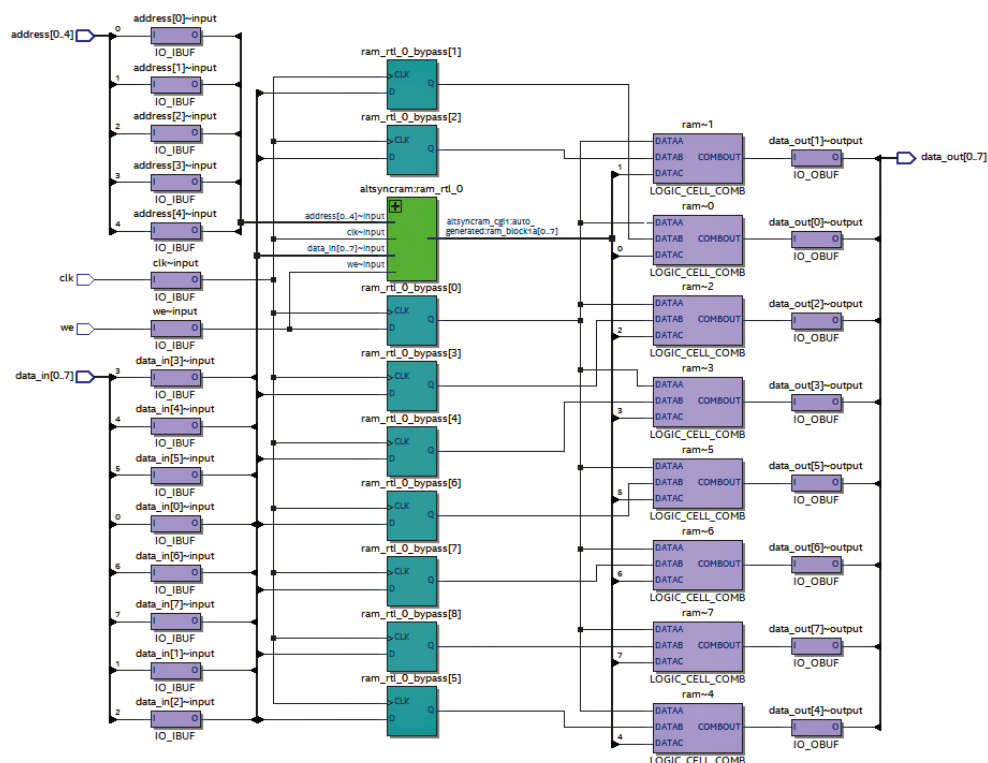
always @ (posedge clk)
begin
    if (we)
        ram[address] = data_in;
        data_out = ram[address];
end
endmodule
```

Przypomnijmy sobie, że ten tryb działania wyjść pamięci nazywa się *nowymi danymi*. Stan danych zapisywanych na wyjściu można wyjaśnić w następujący sposób. W jednym bloku **always** operator „=” zawsze blokuje wykonanie następnego operatora. Dlatego najpierw do pamięci zostaną zapisane nowe dane (*ram[address] = data_in;*), a następnie te nowe dane zostaną przesłane na wyjście pamięci (*data_out = ram[address];*). Aby zaimplementować to zachowanie, syntezy wprowadza

dotatkowe komórki logiczne do schematu projektu, co widać ze statystyk wyników syntezy (rys. 2.7) i technologicznej mapy projektu (rys. 2.8).

Logic utilization (in ALMs)	5 / 56,480 (< 1 %)
Total registers	9
Total pins	23 / 268 (9 %)
Total virtual pins	0
Total block memory bits	256 / 7,024,640 (< 1 %)

RYS. 2.7. Statystyki kompilacji dla projektu *single_port_ram_blocking_assign*

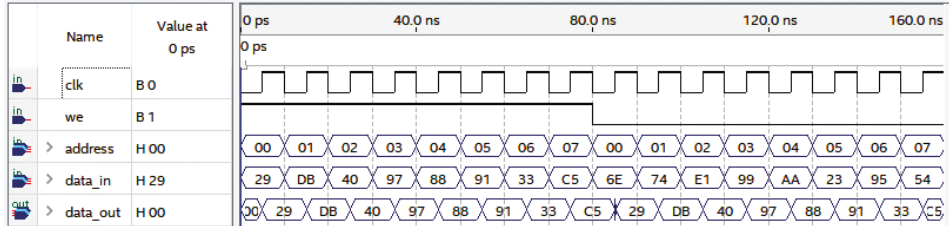


RYS. 2.8. Mapa technologiczna projektu *single_port_ram_blocking_assign*

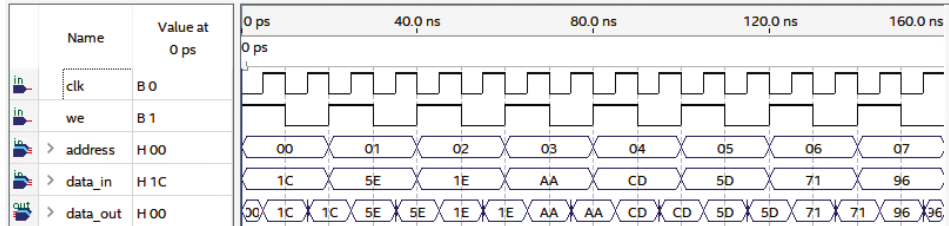
Rysunek 2.8 pokazuje, że do układu zostały dodane elementy logiczne pełniące funkcję multiplexerów, które w zależności od wartości sygnału *we* łączą wyjścia pamięci do macierzy pamięci, albo do rejestrów wejściowych. Zauważmy, że widok projektu *single_port_ram_blocking_assign* na poziomie RTL jest taki sam, jak na rysunku 2.2, tzn. na tym poziomie nie widać wprowadzanej dodatkowej logiki.

Wyniki symulacji modułu *single_port_ram_blocking_assign* pokazano na rysunku 2.9.

a)



b)



RYŚ. 2.9. Wyniki symulacji projektu *single_port_ram_blocking_assign*: a – w przypadku oddzielnych operacji zapisu i odczytu; b – w przypadku naprzemiennych operacji odczytu i zapisu

Analiza rysunku 2.9b pokazuje, że podczas zapisu, dane zapisywane w tym samym cyklu pojawiają się na wyjściu pamięci, a następnie są powtarzane w następnym cyklu podczas odczytu. To zachowanie dokładnie odpowiada trybowi „nowych danych” wyjść pamięci.

Tryb wyjść pamięci „nowe dane” można również jawnie zaimplementować za pomocą operatora nieblokującego „<=” w następującym kodzie:

```

module single_port_ram_single_clk_v3
#(parameter DATA_WIDTH=8, parameter ADDR_WIDTH=5)
(input clk, we,
input [DATA_WIDTH-1:0] data_in,
input [ADDR_WIDTH-1:0] address,
output reg [DATA_WIDTH-1:0] data_out);

// deklaracja pamięci
reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];

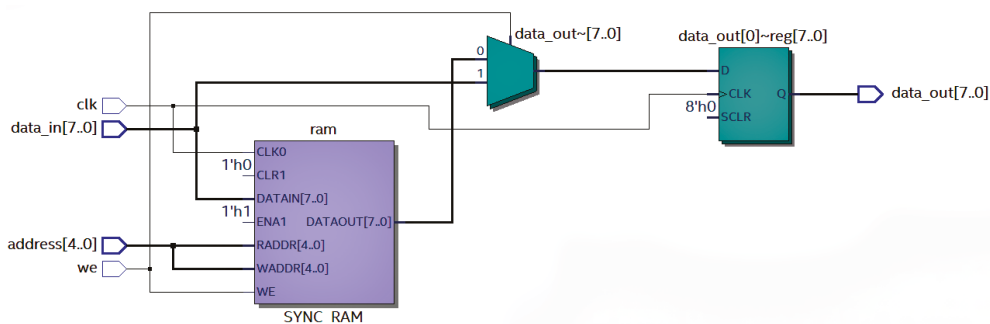
always @ (posedge clk)
begin
    if (we)
    begin
        ram[address] <= data_in;    // zapis
        data_out <= data_in;        // zapisywane dane wyprowadza się
    
```

```

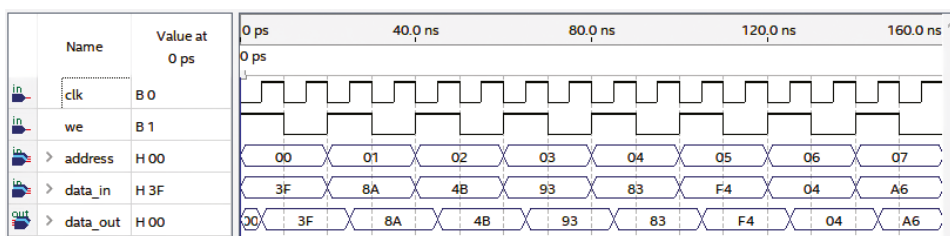
// na wyjście
end
else
    data_out <= ram[address]; // odczyt
end
endmodule

```

Tutaj operacje odczytu i zapisu w bloku **always** są również ściśle oddzielone, ale podczas zapisu, w przeciwieństwie do poprzedniego kodu, dane wejściowe *data_in* są dodatkowo przesyłane na wyjście pamięci (instrukcja *data_out <= data_in;*). Należy zauważyć, że w tym kodzie wszędzie jest używany operator nieblokujący „<=”, co jest bardziej naturalne przy opisywaniu układów synchronicznych. Wyniki syntezy i symulacji projektu *single_port_ram_single_clk_v3* przedstawiono, odpowiednio, na rysunkach 2.10 i 2.11.



RYS. 2.10. Widok projektu *single_port_ram_single_clk_v3* na poziomie RTL



RYS. 2.11. Wyniki symulacji projektu *single_port_ram_single_clk_v3*

Na rysunku 2.10 można zauważyć, że selektor (multiplexer) sterowany sygnałem *we*, który jest jawnie wprowadzony do schematu projektu, służy do wybierania wartości wyjściowych z wyjść macierzy pamięci lub z wejścia *data_in*.

Analiza wyników symulacji (rys. 2.11) pokazuje, że działanie projektu *single_port_ram_single_clk_v3* dokładnie odpowiada trybowi wyjść pamięci z „nowymi danymi”. W związku z tym, projekt *single_port_ram_single_clk_v3* może być zalecany do implementacji pamięci w trybie wyjścia „nowe dane”.

Wniosek. Projekt *single_port_ram_single_clk_v3* działa lepiej w trybie wyjściowym „nowe dane” niż zalecany przez firmę Intel projekt *single_port_ram_blocking_assign*.

2.1.5. Opis pamięci z wyjściami kombinacyjnymi

Czytelnik może mieć naturalne pytanie: czy można zaimplementować pamięć na FPGA z wyjściami kombinacyjnymi? Spróbujmy to zrobić za pomocą następującego kodu:

```
module single_port_ram_assign
#(parameter DATA_WIDTH=8, parameter ADDR_WIDTH=5)
(input clk, we,
input [DATA_WIDTH-1:0] data_in,
input [ADDR_WIDTH-1:0] address,
output [DATA_WIDTH-1:0] data_out);

// deklaracja macierzy pamięci
reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];

always @ (posedge clk)
begin
    if (we)
        ram[address] <= data_in;
end

assign data_out = ram[address];

endmodule
```

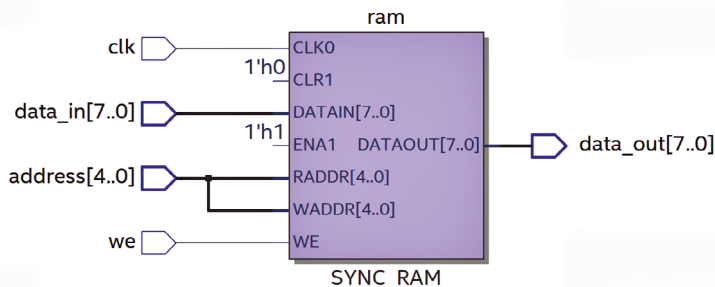
Tutaj zapis do pamięci jest wykonywany synchronicznie na narastającym zboczu sygnału zegarowego *clk*, a wyjścia pamięci *data_out* są wyjściami kombinacyjnymi.

Statystyki wyników syntezy (rys. 2.12) pokazują, że pamięć została zaimplementowana na komórkach logicznych, podczas gdy zupełnie nie zastosowano bloków pamięci wbudowanej FPGA.

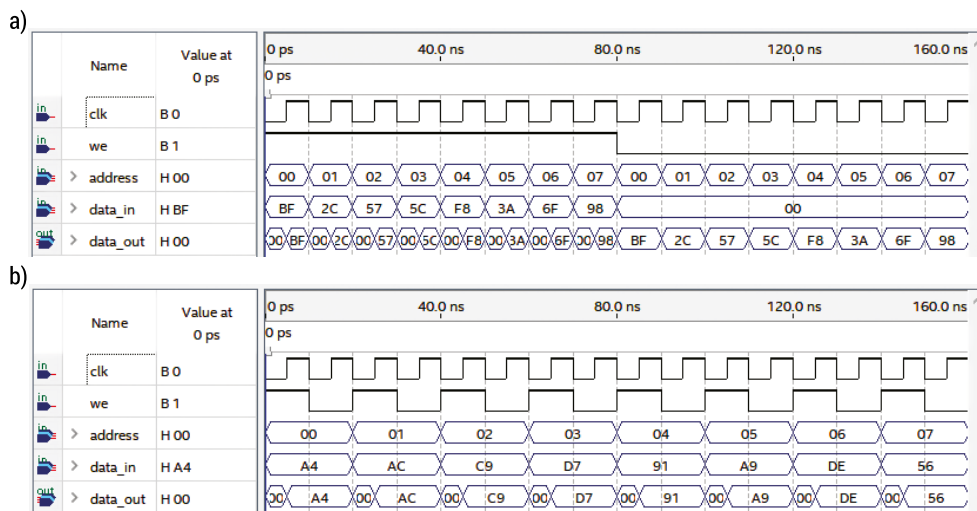
Logic utilization (in ALMs)	134 / 56,480 (< 1 %)
Total registers	256
Total pins	23 / 268 (9 %)
Total virtual pins	0
Total block memory bits	0 / 7,024,640 (0 %)

RYŚ. 2.12. Statystyki wyników syntezy dla projektu *single_port_ram_assign*

Wyniki syntezy i symulacji projektu *single_port_ram_assign* przedstawiono, odpowiednio, na rysunkach 2.13 i 2.14.



RYS. 2.13. Widok projektu *single_port_ram_assign* na poziomie RTL



RYS. 2.14. Wyniki symulacji projektu *single_port_ram_assign*: a – w przypadku oddzielnych operacji zapisu i odczytu; b – w przypadku naprzemiennych operacji odczytu i zapisu

Rysunek 2.13 pokazuje, że projekt *single_port_ram_assign* ma wyjścia kombi-nacyjne. Analiza rysunku 2.14a wykazuje, że na początku cyklu zapisu na wyjściu pamięci pojawiają się zera, a wraz z nadejściem dodatniego zbocza sygnału synchronizacji, zapisane dane pojawiają się na wyjściu. W przypadku naprzemiennych operacji zapisu i odczytu (rys. 2.14b), w momencie zapisu na wyjściu pojawiają się zera, gdyż w tym momencie nie jest znana wartość $ram[address]$, a następnie zapisywane dane są generowane na wyjściu. Zerowe wartości sygnałów na wyjściach ograniczają zakres zastosowania tego sposobu opisu pamięci. Z tego powodu Intel nie zaleca tej metody opisywania pamięci do implementacji w układach FPGA.

2.1.6. Wstawianie rejestrów na wejściach adresowych pamięci

W rozpatrywanych wcześniej projektach (z wyjątkiem *single_port_ram_assign*) rejestr był ustawiony na wyjściach pamięci. Można również wstawić rejestr na wejściach adresowych pamięci za pomocą następującego kodu:

```
module single_port_ram_addr_reg
#(parameter DATA_WIDTH=8, parameter ADDR_WIDTH=5)
(input clk, we,
input [DATA_WIDTH-1:0] data_in,
input [ADDR_WIDTH-1:0] address,
output [DATA_WIDTH-1:0] data_out);

// deklaracja macierzy pamięci
reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];

// zmienna do przechowywania adresu odczytu
reg [ADDR_WIDTH-1:0] addr_reg;

always @ (posedge clk)
begin
    if (we)
        ram[address] <= data_in;
        addr_reg <= address;
end

assign data_out = ram[addr_reg];

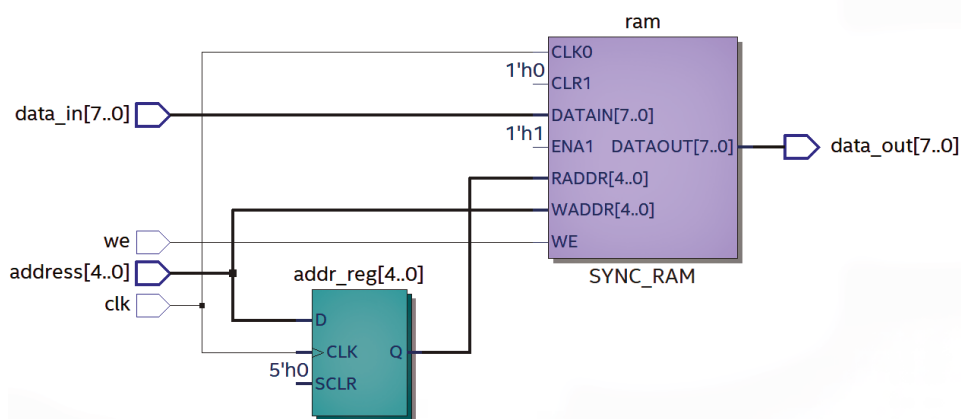
endmodule
```

W powyższym kodzie, aby ustawić rejestr na magistrali adresowej, deklarowana jest dodatkowa zmienna *addr_reg* typu **reg** do przechowywania wartości adresu. W bloku **always**, z nadejściem dodatniego zbocza sygnału zegarowego *clk* przy *we* = 1, dane wejściowe są zapisywane do komórki pamięci, a wartość zmiennej *addr_reg* jest zawsze aktualizowana. Wartość danych wyjściowych pamięci *data_out* jest generowana przy użyciu operatora przypisania ciągłego **assign**, gdzie zmienna *addr_reg* jest używana jako adres pamięci.

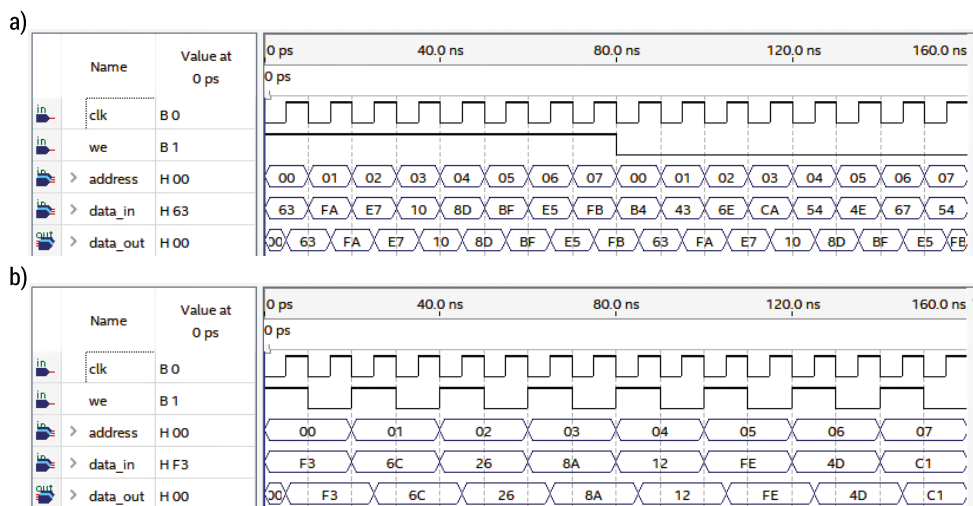
Statystyki wyników syntezy (rys. 2.15) pokazują, że projekt *single_port_ram_addr_reg* jest zaimplementowany w bloku pamięci wbudowanej FPGA. Wyniki syntezy i symulacji projektu *single_port_ram_addr_reg* przedstawiono, odpowiednio, na rysunkach 2.16 i 2.17.

Logic utilization (in ALMs)	1 / 56,480 (< 1 %)
Total registers	0
Total pins	23 / 268 (9 %)
Total virtual pins	0
Total block memory bits	256 / 7,024,640 (< 1 %)

RYS. 2.15. Statystyki wyników syntezy dla projektu *single_port_ram_addr_reg*



RYS. 2.16. Widok projektu *single_port_ram_addr_reg* na poziomie RTL



RYS. 2.17. Wyniki symulacji projektu *single_port_ram_addr_reg*: a – w przypadku oddzielnych operacji odczytu i zapisu; b – w przypadku naprzemiennych operacji odczytu i zapisu

Na rysunku 2.16 można zauważyć, że wyjścia pamięci są kombinacyjne, a rejestr jest ustawiony na wejściu adresowym pamięci. Wyniki symulacji projektu (rys. 2.17)

pokazują, że wyjścia pamięci działają w trybie „nowych danych”, podobnie jak w projekcie *single_port_ram_single_clk_v3* (rys. 2.11). Jednak Intel nie zaleca stosowania tego stylu opisu do implementacji pamięci w układach FPGA, ponieważ jeśli wyjścia pamięci są podłączone do rejestru, to dane na jego wyjściach będą tworzone z opóźnieniem o jeden cykl zegara.

Wniosek ogólny. Do implementacji pamięci na FPGA można polecić

- projekt *single_port_ram_single_clk_v2* w trybie wyjścia „stare dane”,
- projekt *single_port_ram_single_clk_v3* w trybie wyjścia „nowe dane”.

2.2. Inicjalizacja pamięci

Architektura układów FPGA firmy Intel nie obsługuje asynchronicznego zerowania zawartości pamięci za pomocą sygnału *reset*, jak w przypadku przerzutników lub rejestrów. Jednak rejestry wyjściowe bloków pamięci wbudowanej FPGA są zerowane po włączeniu zasilania. Dlatego, po włączeniu zasilania, sygnały na wyjściach bloków pamięci wbudowanej przyjmują stan zero.

Jeśli do pamięci typu RAM trzeba zapisać pewne wartości początkowe, można to zrobić w kodzie projektu. Zazwyczaj inicjalizacja pamięci jest opisana przy użyciu bloku **initial**, jak w poniższym kodzie:

```
module single_port_ram_with_initial
#(parameter DATA_WIDTH=8, parameter ADDR_WIDTH=5)
(input clk, we,
input [DATA_WIDTH-1:0] data_in,
input [ADDR_WIDTH-1:0] address,
output reg [DATA_WIDTH-1:0] data_out);

reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];
integer i;

initial begin
                                // inicjalizacja pamięci
    for (i = 0; i < 32; i = i + 1)
        ram[i] = i[7:0];
end

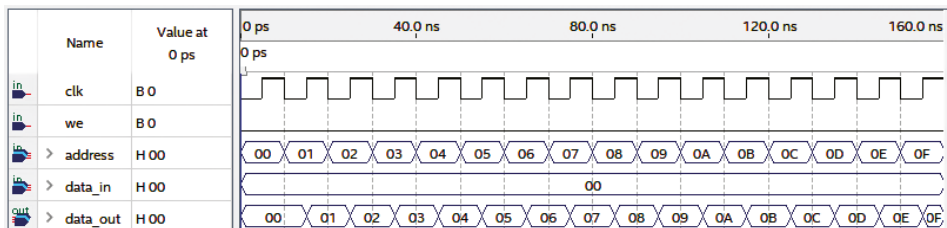
always @ (posedge clk) begin
    if (we)
        ram[address] <= data_in;
    else
```

```
data_out <= ram[address];
```

```
end
```

```
endmodule
```

Wyniki symulacji projektu *single_port_ram_with_initial* pokazano na rysunku 2.18.



RYS. 2.18. Wyniki symulacji projektu *single_port_ram_with_initial*

Wartości początkowe można również zapisać do pamięci za pomocą funkcji systemowych **\$readmemb** i **\$readmemh** języka Verilog [6], które mają następujący format:

```
$readmemb ("file_name", variable_array, start_address, end_address);
```

```
$readmemh ("file_name", variable_array, start_address, end_address);
```

gdzie *file_name* – nazwa pliku, z którego odczytywane są dane; *variable_array* – dwuwymiarowa tablica reprezentująca macierz pamięci; *start_address* i *end_address* – adresy początkowe i końcowe komórek pamięci, do których ładowane są dane. Adresy *start_address* i *end_address* to parametry opcjonalne. Plik, z którego odczytywane są dane, jest plikiem tekstowym ASCII, w którym dane muszą być reprezentowane w postaci binarnej (dla funkcji **\$readmemb**) lub szesnastkowej (dla funkcji **\$readmemh**).

Poniższy kod służy do inicjalizacji zawartości pamięci za pomocą funkcji **\$readmemh**, wykorzystując plik *ram.txt*. Dane zapisywane są od adresu 0 do adresu 31:

```
module single_port_ram_initial_by_file
#(parameter DATA_WIDTH=8, parameter ADDR_WIDTH=5)
(input clk, we,
input [DATA_WIDTH-1:0] data_in,
input [ADDR_WIDTH-1:0] address,
output reg [DATA_WIDTH-1:0] data_out);

reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];
```

```
initial begin
```

```
// inicjalizacja pamięci
```

```

                                $readmemh("ram.txt", ram, 0, 31);
end

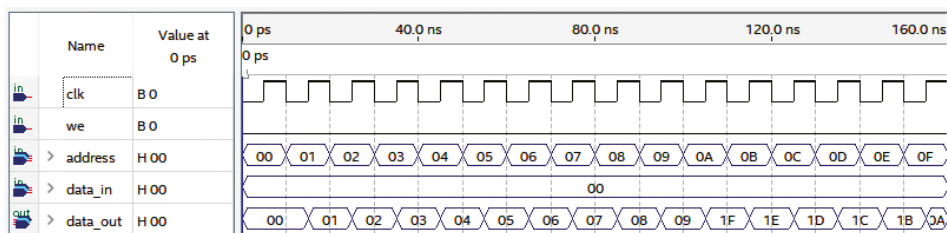
always @ (posedge clk) begin
    if (we)
        ram[address] <= data_in;
    else
        data_out <= ram[address];
end
endmodule

```

W powyższym przykładzie, wstępne dane do pamięci są odczytywane z pliku *ram.txt*, który wygląda następująco:

0 1 2 3 4 5 6 7 8 9 1f 1e 1d 1c 1b A B C D E F 10 11 12 13 14 15 16 17 18 19 1A

Wyniki symulacji projektu *single_port_ram_initial_by_file* pokazano na rysunku 2.19.



RYS. 2.19. Wyniki symulacji projektu *single_port_ram_initial_by_file*

Projektanci systemu Quartus nie zalecają resetowania zawartości pamięci w bloku **always** za pomocą sygnału *reset*, jak to ma zwykle miejsce podczas resetowania rejestrów, na przykład:

```

always @ (posedge clk, negedge reset)
begin
    if (!reset)
        mem[address] <= 0;
    else if (we)
        mem[address] <= data_in;
    data_out <= mem[address];
end

```

Faktem jest, że asynchroniczne resetowanie pamięci nie jest obsługiwane przez architekturę bloków pamięci wbudowanej FPGA Intel, a implementacja powyższego kodu może wymagać dodatkowych zasobów logicznych FPGA do realizacji. Zauważmy,

że w tym podejściu tylko jedna komórka pamięci jest zerowana w każdym cyklu. Aby wyzerować całą pamięć, potrzebne są $2^{\text{ADDR_WIDTH}}$ cykle zegara, w których należy sekwencyjnie wygenerować adresy od 0 do $2^{\text{ADDR_WIDTH}}-1$.

2.3. Jednoportowa pamięć z dwoma sygnałami synchronizacji

Rozpatrzmy wersję jednoportowej pamięci, która wykorzystuje dwa sygnały zegarowe, jeden do operacji zapisu, a drugi – odczytu. Kod takiej pamięci może wyglądać następująco:

```
module single_port_ram_dual_clk
#(parameter DATA_WIDTH=8, parameter ADDR_WIDTH=5)
(input write_clk, read_clk, we,
input [DATA_WIDTH-1:0] data_in,
input [ADDR_WIDTH-1:0] address,
output reg [DATA_WIDTH-1:0] data_out);

// macierz pamięci
reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];

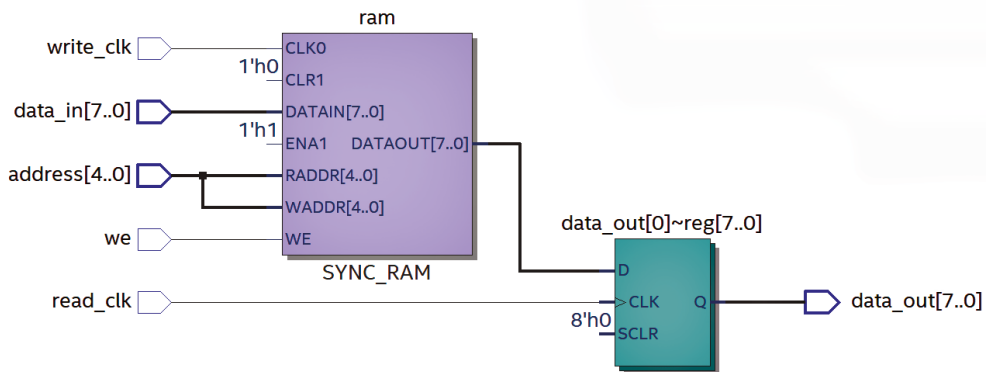
always @ (posedge write_clk)           // zapis
    if (we)
        ram[address] <= data_in;

always @ (posedge read_clk)           // odczyt
    data_out <= ram[address];

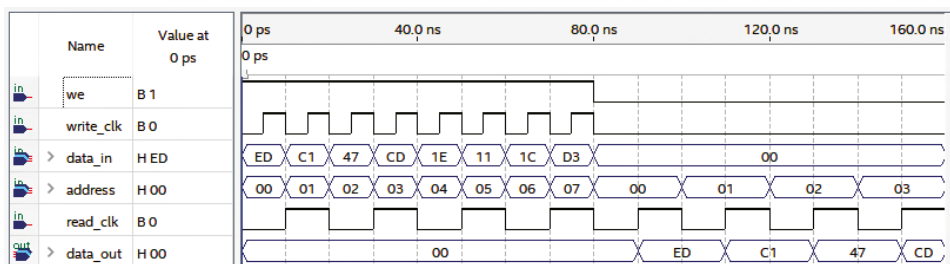
endmodule
```

Wyniki syntezy i symulacji projektu *single_port_ram_dual_clk* przedstawiono, odpowiednio, na rysunkach 2.20 i 2.21.

W symulacji (rys. 2.21) odczyt zawartości pamięci jest wykonywany z częstotliwością dwa razy mniejszą niż częstotliwość zapisu. Zauważmy, że podczas zapisywania do pamięci, jej wyjścia przyjmują stan zero.



RYS. 2.20. Widok projektu *single_port_ram_dual_clk* na poziomie RTL



RYS. 2.21. Wyniki symulacji projektu *single_port_ram_dual_clk*

2.4. Prosta dwuportowa pamięć typu RAM (Simple Dual-Port RAM)

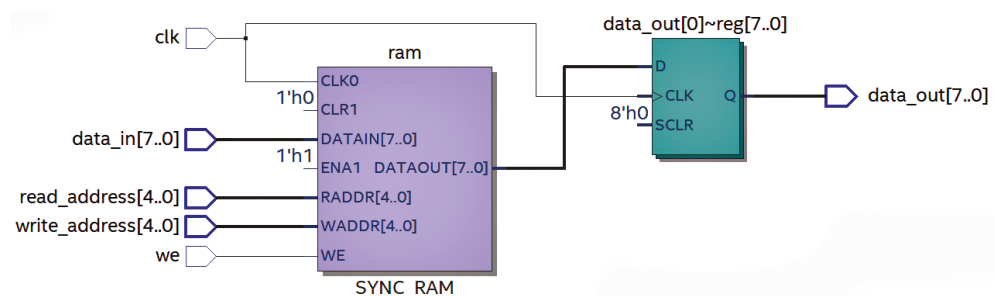
Prosta dwuportowa pamięć typu RAM ma dwa porty adresowe, jeden do operacji zapisu (na przykład *read_address*), a drugi do operacji odczytu (na przykład *write_address*). Jednak ta pamięć ma tylko jeden port wejściowy i jeden port wyjściowy. W języku Verilog, prostą dwuportową pamięć można opisać dla trybu „starych danych” lub dla trybu „nowych danych”. Poniżej przedstawiono kod prostej pamięci dwuportowej z jednym zegarem *clk* i wyjściami w trybie „starych danych”.

```
module simple_dual_port_ram_single_clk_old_data
#(parameter DATA_WIDTH=8, parameter ADDR_WIDTH=5)
(input clk, we,
input [DATA_WIDTH-1:0] data_in,
input [ADDR_WIDTH-1:0] write_address, read_address,
output reg [DATA_WIDTH-1:0] data_out);
```

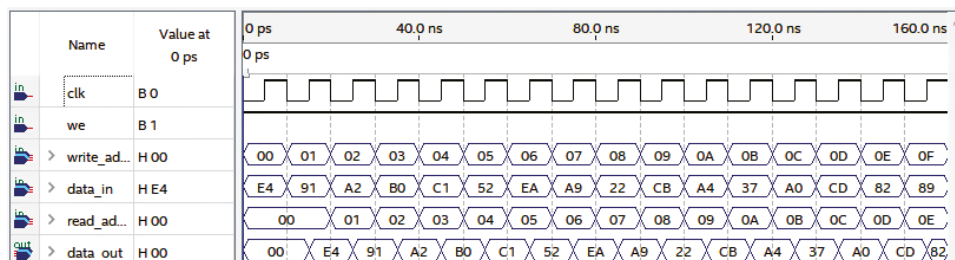
```
// macierz pamięci
reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];

always @ (posedge clk)
begin
    if (we)
        ram[write_address] <= data_in;
        data_out <= ram[read_address];
end
endmodule
```

Wyniki syntezy i symulacji projektu *simple_dual_port_ram_single_clk_old_data* przedstawiono, odpowiednio, na rysunkach 2.22 i 2.23.



RYS. 2.22. Widok projektu *simple_dual_port_ram_single_clk_old_data* na poziomie RTL



RYS. 2.23. Wyniki symulacji projektu *simple_dual_port_ram_single_clk_old_data*

Czytelnik jest proszony o samodzielne opracowanie kodów dla prostej dwupor-towej pamięci typu RAM z trybem wyjścia „nowe dane” oraz z dwoma sygnałami zegarowymi.

2.5. Rzeczywista dwuportowa pamięć typu RAM (True Dual-Port RAM)

Rzeczywista dwuportowa pamięć typu RAM ma dwa niezależne porty (port A i port B) z własnymi magistralami danych, adresu i sygnałami sterującymi. Jednak w tej pamięci macierz pamięci pozostaje współdzielona. Na przykład rzeczywista dwuportowa pamięć może mieć dwa porty wejściowe danych *data_a* i *data_b*, dwa porty wyjścia danych *q_a* i *q_b*, dwa porty adresowe *addr_a* i *addr_b*, dwa sygnały zezwolenia zapisu *we_a* i *we_b*, jak również pamięć dwuportowa może mieć jeden lub dwa sygnały zegarowe. Każdy port rzeczywistej pamięci dwuportowej może działać niezależnie, tzn. wykonywać dowolne operacje odczytu i zapisu, z wyjątkiem jednoczesnego zapisu do tej samej komórki pamięci. W tym ostatnim przypadku, zawartość komórki pamięci jest uważana za nieokreśloną.

Poniżej znajduje się kod rzeczywistej pamięci dwuportowej z dwoma sygnałami synchronizacji i trybem wyjścia „nowe dane”.

```
module true_dual_port_ram_dual_clk_new_data
#(parameter DATA_WIDTH=8, parameter ADDR_WIDTH=5)
(input [DATA_WIDTH-1:0] data_a, data_b,
input [ADDR_WIDTH-1:0] addr_a, addr_b,
input we_a, we_b, clk_a, clk_b,
output reg [DATA_WIDTH-1:0] q_a, q_b);
```

```
// macierz pamięci
```

```
reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];
```

```
always @ (posedge clk_a)      // port A
```

```
begin
```

```
    if (we_a)
```

```
    begin
```

```
        ram[addr_a] <= data_a;
```

```
        q_a <= data_a;
```

```
    end
```

```
    else
```

```
        q_a <= ram[addr_a];
```

```
end
```

```
always @ (posedge clk_b)      // port B
```

```
begin
```

```
    if (we_b)
```

```
    begin
```


Czytelnik jest proszony o opracowanie kodu dla innych typów pamięci dwuportowych (z trybem wyjścia „stare dane”, z oddzielnymi sygnałami synchronizacji odczytu i zapisu).

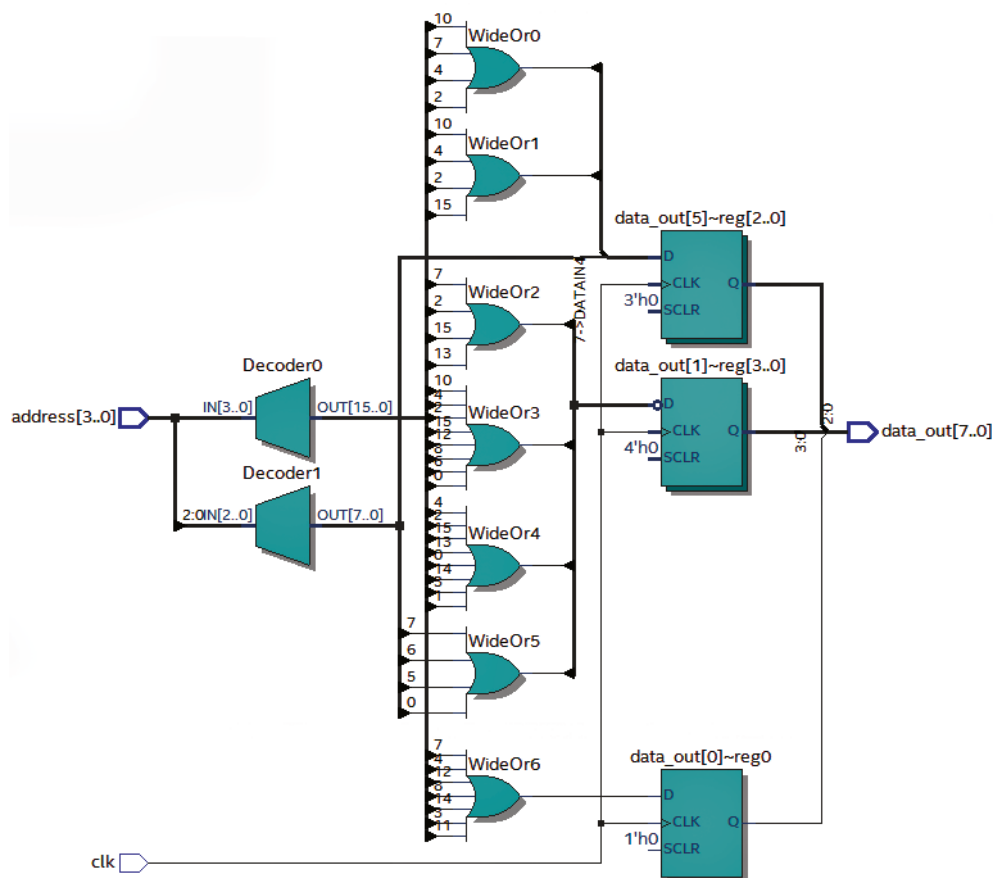
2.6. Pamięć typu ROM (Read-Only Memory)

Cechą implementacji pamięci typu ROM na FPGA jest to, że jej działanie można opisać za pomocą instrukcji **case**, której stałe elementy określają adresy komórek pamięci, a odpowiadające im instrukcje – zawartość komórek, na przykład:

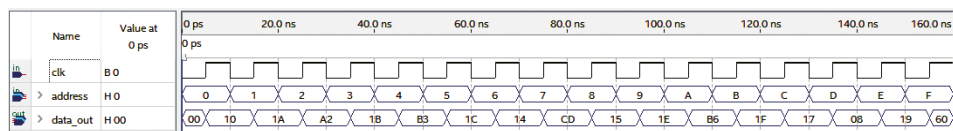
```
module rom_case
#(parameter DATA_WIDTH=8, parameter ADDR_WIDTH=4)
(input clk,
input [ADDR_WIDTH-1:0] address,
output reg [DATA_WIDTH-1:0] data_out);

always @ (posedge clk)
    case (address)
        4'h0: data_out = 8'h10;
        4'h1: data_out = 8'h1A;
        4'h2: data_out = 8'hA2;
        4'h3: data_out = 8'h1B;
        4'h4: data_out = 8'hB3;
        4'h5: data_out = 8'h1C;
        4'h6: data_out = 8'h14;
        4'h7: data_out = 8'hCD;
        4'h8: data_out = 8'h15;
        4'h9: data_out = 8'h1E;
        4'hA: data_out = 8'hB6;
        4'hB: data_out = 8'h1F;
        4'hC: data_out = 8'h17;
        4'hD: data_out = 8'h08;
        4'hE: data_out = 8'h19;
        4'hF: data_out = 8'h60;
    endcase
endmodule
```

Wyniki syntezy i symulacji projektu *rom_case* przedstawiono, odpowiednio, na rysunkach 2.26 i 2.27.



RYS. 2.26. Widok projektu *rom_case* na poziomie RTL



RYS. 2.27. Wyniki symulacji projektu *rom_case*

Pamięć typu ROM na rysunku 2.26 jest zaimplementowana w komórkach logicznych FPGA.

Wadą powyższego opisu jest to, że kompilator systemu Quartus nie może rozpoznać opisu pamięci w kodzie. Dlatego przy tym stylu opisywania pamięci typu ROM, pamięć zawsze będzie implementowana w komórkach logicznych, a nie w bloku pamięci wbudowanej FPGA.

Podany sposób opisu pamięci typu ROM jest dobrze dostosowany do małych rozmiarów pamięci. W przypadku dużej pojemności pamięci typu ROM wygodniej jest opisać ją jako pamięć typu RAM z inicjalizacją i tylko operacją odczytu. Zauważmy, że pamięć typu ROM, podobnie jak pamięć typu RAM, może być jedno- i dwuportowa, a także może mieć jeden lub dwa sygnały synchronizacji. Poniżej znajduje się kod dwuportowej pamięci typu ROM z dwoma sygnałami zegarowymi.

```

module dual_port_rom
#(parameter DATA_WIDTH=8, parameter ADDR_WIDTH=4)
(input clk_a, clk_b,
input [ADDR_WIDTH-1:0] addr_a, addr_b,
output reg [DATA_WIDTH-1:0] q_a, q_b);

reg [DATA_WIDTH-1:0] rom[2**ADDR_WIDTH-1:0];

initial                                     // ładowanie zawartości pamięci
    $readmemh(„rom_init.txt”, rom);

always @ (posedge clk_a)                // odczyt portu A
    q_a <= rom[addr_a];

always @ (posedge clk_b)                // odczyt portu B
    q_b <= rom[addr_b];

endmodule

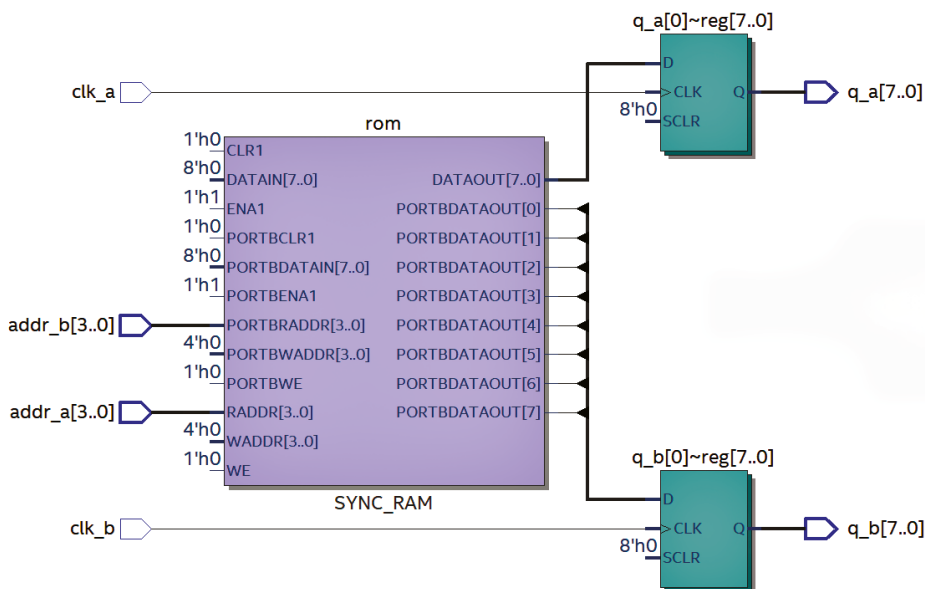
```

W powyższym przykładzie ładowanie zawartości pamięci typu ROM odbywa się z pliku *rom_init.txt*, który wygląda następująco:

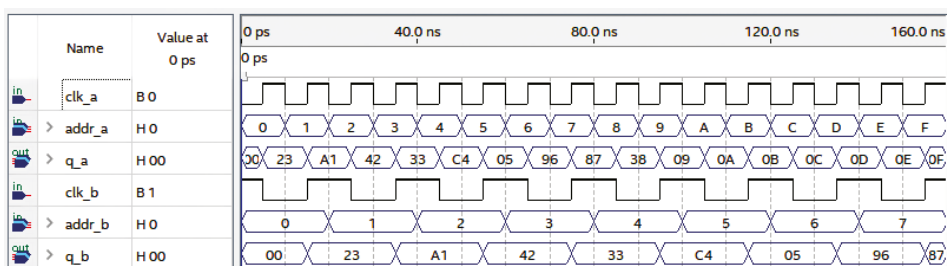
```
23 A1 42 33 C4 05 96 87 38 09 0A 0B 0C 0D 0E 0F.
```

Wyniki syntezy i symulacji projektu *dual_port_rom* przedstawiono, odpowiednio, na rysunkach 2.28 i 2.29.

Z rysunku 2.28 wynika, że kompilator rozpoznał opis pamięci w kodzie i zaimplementował go za pomocą prymitywu SYNC_RAM (mimo, że kod opisuje pamięć typu ROM).



RYS. 2.28. Widok projektu *dual_port_rom* na poziomie RTL

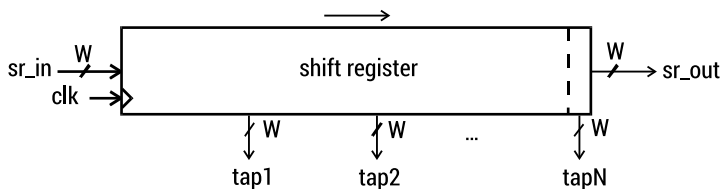


RYS. 2.29. Wyniki symulacji projektu *dual_port_rom*

Rysunek 2.29 pokazuje, że odczyt zawartości pamięci typu ROM jest wykonywany z różnymi częstotliwościami taktowania dla portu A i portu B.

2.7. Implementacja rejestrów przesuwających w blokach pamięci wbudowanej FPGA

Układy FPGA firmy Intel umożliwiają implementację złożonych rejestrów przesuwających w blokach pamięci wbudowanej FPGA, oszczędzając zasoby logiczne. Uogólnioną strukturę rejestru przesuwającego magistrali pokazano na rysunku 2.30.



RYS. 2.30. Uogólniona struktura rejestru przesuwającego magistrali z odczepami

Powyższy rejestr przesuwający umożliwia przesuwanie zawartości magistrali o szerokości W bitów. Każdy cykl zegara przesuwą zawartość rejestru o jeden bit w prawo. W tym przypadku, dane wejściowe są odbierane przez magistralę sr_in , a wynik jest wysyłany do magistrali sr_out . Ponadto rejestr przesuwający ma N pośrednich odczepów $tap1, \dots, tapN$, za pomocą których zawartość niektórych bitów jest wyprowadzana na wyjście. Zwykle odległość (w bitach) między zaczepami jest taka sama i jest oznaczona parametrem L . Przy $W = 1$ i $N = 0$ mamy zwykły rejestr przesuwający.

Kompilator systemu Quartus nie zawsze implementuje rejestry przesuwające w blokach pamięci wbudowanej FPGA. Robi to tylko wtedy, gdy spełnione są następujące warunki:

- szerokość magistrali $W = 1$, a całkowita długość rejestru jest większa niż 67 ($N \times L > 67$);
- szerokość magistrali $W > 1$, a całkowita liczba bitów rejestru jest większa niż 32 ($W \times N \times L > 32$).

We wszystkich innych przypadkach rejestr przesuwający jest realizowany za pomocą komórek logicznych FPGA. Ponadto odległość L między zaczepami musi być potęgą dwójki. Jeśli tak nie jest, syntezytor wykorzystuje dodatkowe zasoby logiczne FPGA do budowy rejestru. W rezultacie zwiększa się opóźnienie sygnału i niweluje to korzyści wynikające z implementacji rejestru przesuwającego w bloku pamięci wbudowanej FPGA.

Należy również zauważyć, że poszczególne bity rejestru przesuwającego zaimplementowane w bloku pamięci wbudowanej nie są dostępne dla symulatora.

Poniżej pokazano kod 68-bitowego rejestru przesuwającego, który jest zaimplementowany w bloku pamięci wbudowanej FPGA.

```

module shift_reg_68_bits (
input clk, shift,
input sr_in,
output sr_out);

reg [67:0] sr;

always @ (posedge clk)

```

```

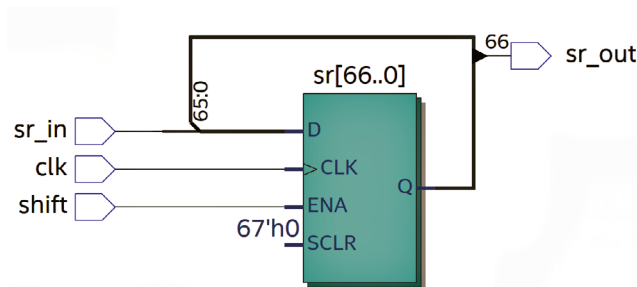
begin
    if (shift == 1'b1)
    begin
        sr[67:1] <= sr[66:0];
        sr[0] <= sr_in;
    end
end
assign sr_out = sr[66];
endmodule

```

Statystyki wyników syntezy (rys. 2.31) pokazują, że projekt *shift_reg_68_bits* jest realizowany na 64 bitach bloku pamięci wbudowanej i 8 logicznych modułach ALM. Wyniki syntezy projektu *shift_reg_68_bits* pokazano na rysunku 2.32.

Logic utilization (in ALMs)	8 / 32,070 (< 1 %)
Total registers	15
Total pins	4 / 457 (< 1 %)
Total virtual pins	0
Total block memory bits	64 / 4,065,280 (< 1 %)

RYŚ. 2.31. Statystyki wyników syntezy projektu *shift_reg_68_bits*



RYŚ. 2.32. Widok projektu *shift_reg_68_bits* na poziomie RTL

Poniższy kod opisuje 64-bitowy rejestr przesuwający magistrali o szerokości magistrali $W = 8$ i czterech odczepach na bitach 16, 32, 48 i 64.

```

module shift_reg_64_bits_4_taps
(input clk, shift,
input [7:0] sr_in,
output [7:0] sr_tap_one, sr_tap_two, sr_tap_three, sr_out);

reg [7:0] sr [64:0];

```



```

integer i;

always @ (posedge clk)
begin
    if (shift)
    begin
        for (i = 64; i>0; i = i-1)
        begin
            sr[i] <= sr[i-1];
        end
        sr[0] <= sr_in;
    end
end

assign sr_tap_one = sr[16];
assign sr_tap_two = sr[32];
assign sr_tap_three = sr[48];
assign sr_out = sr[64];
endmodule

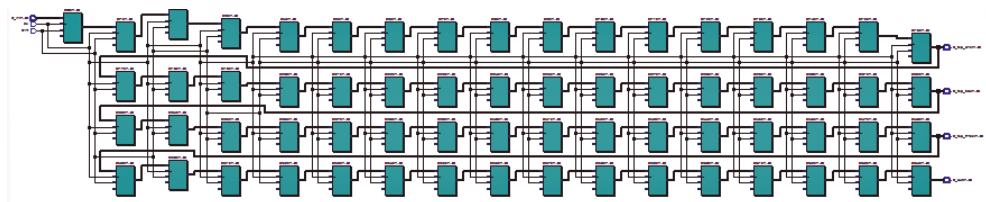
```

Statystyki wyników kompilacji (rys. 2.33) pokazują, że rejestr przesuwający magistrali jest zaimplementowany we wbudowanych blokach pamięci typu RAM, ale dodatkowo wykorzystywane są również komórki logiczne FPGA.

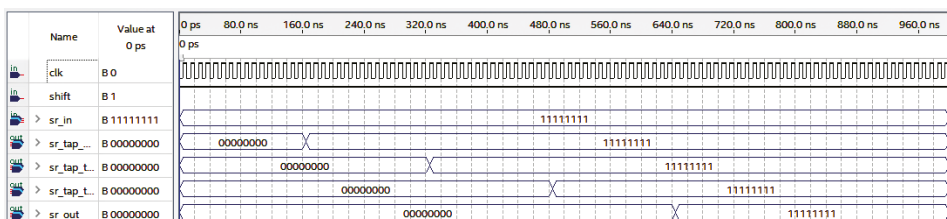
Logic utilization (in ALMs)	23 / 56,480 (< 1 %)
Total registers	80
Total pins	42 / 268 (16 %)
Total virtual pins	0
Total block memory bits	448 / 7,024,640 (< 1 %)

RYŚ. 2.33. Statystyki wyników syntezy projektu *shift_reg_64_bits_4_taps*

Wyniki syntezy i symulacji projektu *shift_reg_64_bits_4_taps* przedstawiono, odpowiednio, na rysunkach 2.34 i 2.35.



RYŚ. 2.34. Widok projektu *shift_reg_64_bits_4_taps* na poziomie RTL

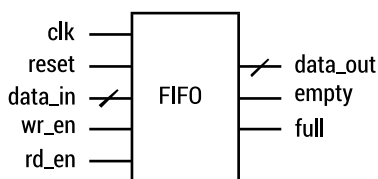


RYS. 2.35. Wyniki symulacji projektu *shift_reg_64_bits_4_taps*

2.8. Pamięć typu FIFO

Pamięć typu FIFO (First-In, First-Out) to rodzaj pamięci jednoportowej, w której dane, które zapisywane są do niej jako pierwsze, odczytywane są z pamięci także jako pierwsze. Pamięć typu FIFO jest szeroko stosowana w systemach wbudowanych podczas przesyłania danych z jednego podsystemu do drugiego. *Pojemność* (głębokość) pamięci typu FIFO jest określana na podstawie wyników symulacji na poziomie systemu. Uwzględnia się tu szybkość, z jaką dane są odbierane ze źródła i szybkość, z jaką odbiornik odczytuje dane.

Pamięć typu FIFO może być synchroniczna lub asynchroniczna. W przypadku synchronicznej pamięci typu FIFO można zastosować dwa sygnały synchronizacyjne, jeden do zapisu danych, drugi do odczytu danych. Taka pamięć typu FIFO może być używana do łączenia podsystemów pracujących na różnych częstotliwościach zegara. W przypadku asynchronicznej pamięci typu FIFO operacje zapisu i odczytu są sterowane odpowiednio wejściowymi i wyjściowymi sygnałami strobującymi. Rysunek 2.36 przedstawia symbol pamięci typu FIFO.



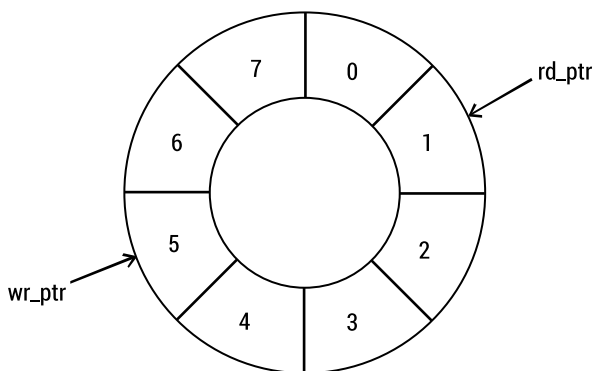
RYS. 2.36. Symbol pamięci typu FIFO

Pamięć typu FIFO (rys. 2.36) posiada dwa porty danych: port wejściowy *data_in*, za pomocą którego zapisywane dane są wprowadzane do pamięci, oraz port wyjściowy *data_out*, przez który odczytane dane są wyprowadzane z pamięci. Oprócz tradycyjnych sygnałów sterujących *clk* i *reset*, pamięć typu FIFO ma wejście zezwolenia na zapis *wr_en* i wejście zezwolenia na odczyt *rd_en*. Pamięć typu FIFO ma również dwa dodatkowe wyjścia: *empty*, które wskazuje, że pamięć jest pusta (tzn. nie ma zapisanych danych), oraz *full*, które wskazuje, że pamięć jest pełna. Gdy pamięć typu FIFO jest pusta, nie można z niej odczytać żadnych danych, a gdy pamięć typu FIFO

jest pełna, żadne dane nie mogą być do niej zapisane bez utraty wcześniej zapisanych danych. Zauważmy, że pamięć typu FIFO nie ma wejścia adresowego.

Oprócz wspomnianych wyjść (*empty* i *full*), pamięć typu FIFO może mieć wyjścia *almost_full* (prawie pełna) i *almost_empty* (prawie pusta). Na przykład, gdy pamięć jest zapełniona w 90% lub więcej, generowany jest sygnał *almost_full*, a gdy pamięć jest zapełniona poniżej 10%, generowany jest sygnał *almost_empty*. Sygnały *almost_full* i *almost_empty* umożliwiają systemowi wbudowanemu podjęcie pewnych działań, zanim pamięć typu FIFO będzie pusta lub pełna.

Logikę działania pamięci typu FIFO można łatwo wyjaśnić na przykładzie *pamięci pierścieniowej* (ring memory) (rys. 2.37).



RYŚ. 2.37. Funkcjonowanie pamięci typu FIFO

Wskaźnik *wr_ptr* zawiera adres komórki pamięci, do której zostaną zapisane dane, gdy sygnał *wr_en* jest aktywowany, wskaźnik *rd_ptr* zawiera adres komórki pamięci, z której dane będą odczytywane, gdy aktywowany jest sygnał *rd_en*. Przy każdym zapisie do pamięci typu FIFO, wskaźnik *wr_ptr* jest zwiększany o jeden, a dla każdego odczytu z pamięci zwiększany o jeden jest wskaźnik *rd_ptr*. Po przekroczeniu maksymalnej wartości (pojemności pamięci pomniejszonej o 1) każdy ze wskaźników przyjmuje wartość 0.

Głównym problemem przy projektowaniu pamięci typu FIFO jest zdefiniowanie wartości sygnałów *empty* i *full*. Na początku działania pamięci typu FIFO lub po przyjęściu sygnału resetowania *reset*, oba wskaźniki ustawiane są na komórkę o adresie 0, pamięć jest uważana za pustą i niepełną, tzn. *wr_ptr* = *rd_ptr* = 0, *empty* = 1 i *full* = 0. Zwykle pierwszą operacją jest zapis do pamięci typu FIFO, odpowiednio, wskaźnik *wr_ptr* jest zwiększany o jeden, a sygnał *empty* jest zerowany.

W trakcie swojej pracy wskaźniki *wr_ptr* i *rd_ptr* mogą zmieniać swoje wartości na różne sposoby, w zależności od potrzeby zapisu do pamięci i odczytu z pamięci. Dlatego może się zdarzyć, że wskaźnik *rd_ptr* zmienia się szybciej i „dogania” wskaźnik *wr_ptr*. Odpowiada to temu, że pamięć typu FIFO jest pusta i należy ustawić sygnał *empty*. Jeśli jednak wskaźnik *wr_ptr* rośnie szybciej, to wskaźnik *wr_ptr* „dogoni” wskaźnik *rd_ptr*. Odpowiada to pełnej pamięci typu FIFO i musi być ustawiony sygnał

full. Innymi słowy, w przypadku równości wskaźników *wr_ptr* i *rd_ptr* konieczne jest rozpoznanie stanu pamięci typu FIFO: pamięć jest pełna lub pamięć jest pusta.

Istnieje kilka podejść do rozwiązywania tego problemu:

- zliczanie ilości słów zapisanych w pamięci;
- określenie trendu operacji zapisu i odczytu, pamięć jest zapełniana (filling) lub opróżniana (emptying);
- przechowywanie stanów pamięci i zastosowanie automatu skończonego itp.

Jako przykład opisu pamięci typu FIFO rozpatrzmy kod projektu oparty na liczeniu liczby zapisanych w pamięci słów.

```
module fifo
#(parameter DATA_WIDTH=8, ADDR_WIDTH=2, DEPTH=(1<<ADDR_WIDTH))
(input [DATA_WIDTH-1:0] data_in,
input clk, reset, rd, wr,
output empty, full,
output reg [ADDR_WIDTH:0] fifo_cnt,
output reg [DATA_WIDTH-1:0] data_out);

// macierz pamięci
reg [DATA_WIDTH-1:0] fifo_ram[0:DEPTH-1];
reg [ADDR_WIDTH-1:0] rd_ptr, wr_ptr; // wskaźniki odczytu i zapisu

// określanie stanu pamięci
assign empty = (fifo_cnt==0);           // pamięć jest pusta
assign full = (fifo_cnt==DEPTH);        // pamięć jest pełna

always @(posedge clk)                  // zapis
    if (wr && !full) fifo_ram[wr_ptr] <= data_in;
    else if (wr && rd)    fifo_ram[wr_ptr] <= data_in;

always @(posedge clk)                  // odczyt
    if (rd && !empty)      data_out <= fifo_ram[rd_ptr];
    else if (rd && wr && empty) data_out <= fifo_ram[rd_ptr];

always @(posedge clk)                  // modyfikacja wskaźników
    if (~reset)
    begin
        wr_ptr <= 0;
        rd_ptr <= 0;
```

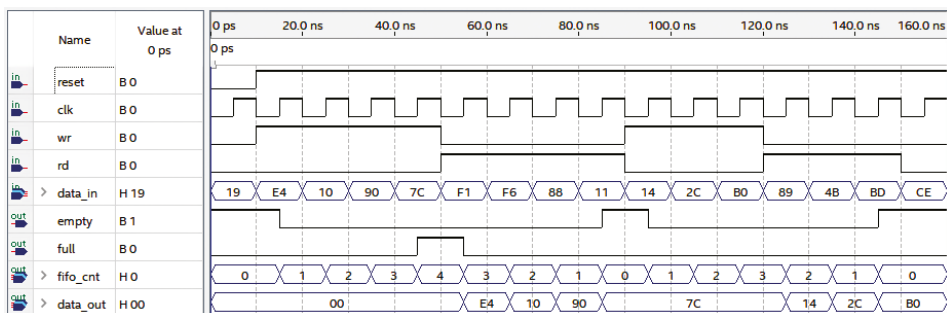
```

end
else
begin
    wr_ptr <= ((wr && !full)|| (wr && rd)) ? wr_ptr + 1'b1 : wr_ptr;
    rd_ptr <= ((rd && !empty)|| (wr && rd)) ? rd_ptr + 1'b1 : rd_ptr;
end
end

always @(posedge clk) // zliczanie liczby elementów w pamięci
    if (~reset)        fifo_cnt <= 0;
    else
    case ({wr,rd})
        2'b00 : fifo_cnt <= fifo_cnt;
        2'b01 : fifo_cnt <= (fifo_cnt==1'b0) ? 1'b0 : fifo_cnt - 1'b1;
        2'b10 : fifo_cnt <= (fifo_cnt==DEPTH) ? DEPTH : fifo_cnt + 1'b1;
        2'b11 : fifo_cnt <= fifo_cnt;
    endcase
endmodule

```

Projekt *fifo* ma trzy parametry: szerokość danych DATA_WIDTH, szerokość wskaźnika adresu ADDR_WIDTH i głębokość pamięci DEPTH. Zauważmy, że głębokość pamięci typu FIFO niekoniecznie wynosi $2^{\text{ADDR_WIDTH}}$. Do zliczenia liczby słów zapisanych do pamięci służy zmienna *fifo_cnt*, której wartość jest również widoczna na wyprowadzeniach zewnętrznych. Pozwala to oszacować stopień zapelnienia pamięci i podjąć niezbędne kroki w przypadku zbliżania się do sytuacji, gdy mamy pełną lub pustą pamięć typu FIFO. Wyniki symulacji projektu *fifo* przedstawiono na rysunku 2.38.



RYS. 2.38. Wyniki symulacji projektu *fifo*

W naszym przykładzie pamięć typu FIFO ma głębokość 4. W pierwszym cyklu (rys. 2.38) sygnał *reset* zeruje pamięć typu FIFO i ustawiany jest sygnał *empty*. W następnych 4 cyklach zegara wartości E4, 10, 90 i 7C są sekwencyjnie zapisywane do pamięci. Gdy pamięć jest pełna w momencie czasu 45ns, ustawiony jest sygnał

full. Następnie odczytywane są cztery wcześniej zapisane słowa pamięci, ich wartości można zobaczyć na magistrali *data_out*. Po przeczytaniu wszystkich słów sygnał *empty* jest ustawiony w momencie czasu 85ns.

Czytelnik jest proszony o opracowanie projektów dla synchronicznej pamięci typu FIFO z dwoma sygnałami synchronizacyjnymi, oddzielnie dla operacji zapisu i oddzielnie dla operacji odczytu, oraz asynchronicznej pamięci typu FIFO.

2.9. Pamięć typu LIFO

Pamięć typu LIFO (Last-In, First-Out) jest pamięcią, w której ostatnie do niej zapisywane dane są najpierw odczytywane z pamięci. Pamięć typu LIFO jest szeroko stosowana w mikroprocesorach i mikrokontrolerach jako *pamięć stosowa*. Na przykład stos przechowuje adresy powrotu dla wywołań podprogramów i programów obsługi przerwań. W zaawansowanych architekturach procesorów stos może przechowywać dane przekazywane do podprogramów lub wskaźniki do zmiennych.

Istnieje wiele sposobów implementacji stosu, tzn. pamięci typu LIFO. Poniższy kod przedstawia jeden ze sposobów opisanie pamięci typu LIFO w języku Verilog.

```
module lifo
#(parameter      DATA_WIDTH=8,           // szerokość danych
      ADDR_WIDTH=2,           // szerokość adresu
      DEPTH=(1<<ADDR_WIDTH)) // głębokość stosu
(input [DATA_WIDTH-1:0] data_in,           // zapisywane dane
input clk, reset,
input rd,wr,                             // odczyt, zapis
output reg [DATA_WIDTH-1:0] data_out,     // czytane dane
output reg error);

reg [DATA_WIDTH-1:0] ram [DEPTH-1:0]; // macierz pamięci
reg empty,full;

reg [ADDR_WIDTH:0] sp;                // wskaźnik stosu

always @(posedge clk)
    if(~reset) begin
        sp = DEPTH;                // wskaźnik ustawić na 1 więcej
        full = sp ? 1'b0 : 1'b1;   // definicja flag
        empty = sp[ADDR_WIDTH]; // pamięć jest pusta
        data_out = {DATA_WIDTH{1'b0}}; // na wyjściu 0
    end
```

```

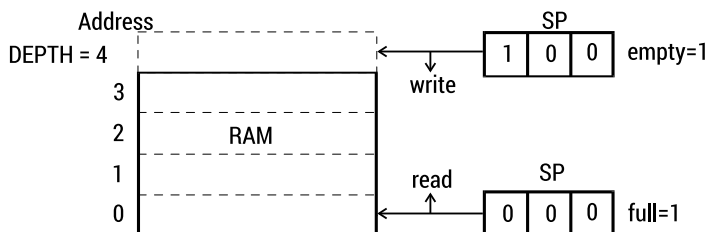
        error = 1'b0;
    end
    else begin
        if(wr && !full) begin           // zapis
            sp = sp - 1'b1;             // modyfikacja wskaźnika
            ram[sp] = data_in;          // wykonanie zapisu
            full = sp ? 1'b0 : 1'b1;    // modyfikacja flag
            empty = sp[ADDR_WIDTH];
            error = 1'b0;
        end
        else if(rd && !empty) begin     // odczyt
            data_out = ram[sp];         // wykonanie odczytu
            sp = sp + 1'b1;             // modyfikacja wskaźnika
            full = sp ? 1'b0 : 1'b1;    // modyfikacja flag
            empty = sp[ADDR_WIDTH];
            error = 1'b0;
        end
    end
    // sprawdzanie błędów
    else if (((empty && rd) || (full && wr) || (rd && wr)))
        error = 1'b1;                  // błąd
    else
        error = 1'b0;                  // nie ma błędu
    end
endmodule

```

Powyższy kod przedstawia sparymetryzowany moduł pamięci typu LIFO. Parametry określają szerokość danych (DATA_WIDTH), szerokość adresu (ADDR_WIDTH) i głębokość (DEPTH), tzn. liczbę słów pamięci typu LIFO.

Domyślna głębokość stosu to $2^{\text{ADDR_WIDTH}}$. Aby otrzymać tę wartość, wystarczy przesunąć w lewo jedynkę o ADDR_WIDTH bitów.

Projekt pamięci typu LIFO zawiera również zmienną *sp* (stack pointer) – wskaźnik wierzchołka stosu oraz flagi *empty* (pamięć jest pusta) i *full* (pamięć jest pełna). Logikę działania pamięci typu LIFO przedstawiono na rysunku 2.39.



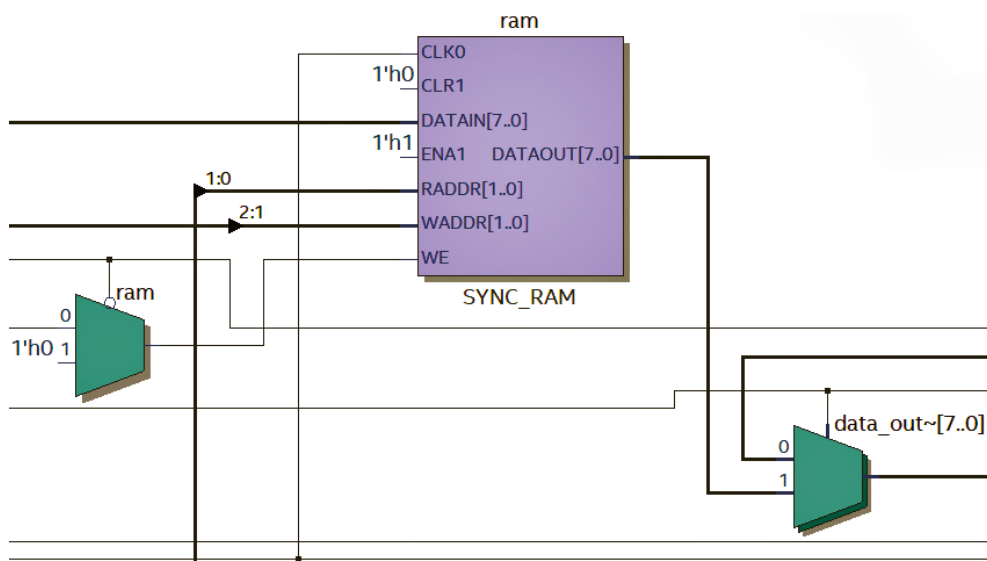
RYS. 2.39. Funkcjonowanie pamięci typu LIFO kiedy ADDR_WIDTH = 2

Zmienna *sp* wskazuje na komórkę pamięci, która ma być odczytana w przypadku operacji odczytu. W projekcie przyjęto następującą regułę: przy zapisie do pamięci (*wr* = 1) wartość *sp* maleje o jeden, a przy odczycie z pamięci (*rd* = 1) wartość *sp* jest zwiększana o jeden. Rozmiar *sp* jest o 1 bit większy niż jest potrzebny do adresowania komórek pamięci. Jest to celowe, ponieważ przed rozpoczęciem pracy wartość *sp* jest o jeden większa niż liczba komórek pamięci, tzn. *sp* wskazuje na komórkę pamięci, która nie istnieje. Pojedyncza wartość najbardziej znaczącego bitu *sp* jest znakiem, że pamięć typu LIFO jest pusta (*empty* = 1). Gdy dane są zapisywane do pamięci typu LIFO, *sp* może spaść do 0. Jeśli *sp* wynosi zero, pamięć typu LIFO jest pełna (*full* = 1).

Do prawidłowego funkcjonowania naszego projektu krokiem obowiązkowym jest synchroniczny reset pamięci typu LIFO za pomocą niskiego poziomu sygnału resetowania *reset*. Kiedy pamięć typu LIFO jest opróżniona, wskaźnik *sp* jest ustawiany na wartość początkową (*DEPTH*), określane są flagi *empty* i *full*, a wyjście pamięci jest ustawiane na zero.

Zapis do pamięci typu LIFO jest przeprowadzany po ustawieniu sygnału *wr* i skasowaniu sygnału *full*, tzn. zapis można wykonać tylko do niepełnej pamięci typu LIFO. Operacja zapisu polega na zmniejszeniu *sp* o jeden, zapisaniu danych wejściowych do komórki pamięci wskazywanej przez *sp* i ocenie flag *full* i *empty*. Odczyt z pamięci typu LIFO jest wykonywany po ustawieniu sygnału *rd* i skasowaniu sygnału *empty*, tzn. odczyt można wykonać tylko z niepustej pamięci typu LIFO. Wykonywanie operacji odczytu polega na umieszczeniu wartości komórki pamięci wskazywanej przez *sp* na magistrali wyjściowej, inkrementacji *sp* oraz ocenie flag *full* i *empty*.

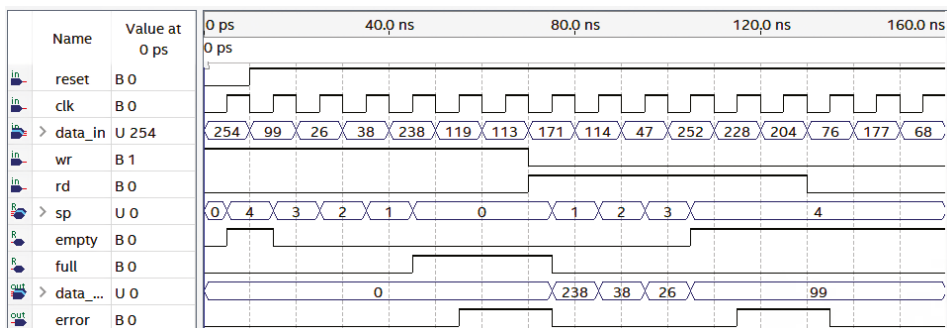
Za błąd podczas pracy z pamięcią typu LIFO uważa się odczyt z pustej pamięci, zapis do pełnej pamięci, a także jednoczesny odczyt i zapis do pamięci. Fragment schematu projektu *lifo*, zsyntetyzowanego przez kompilator, pokazano na rysunku 2.40.



RYS. 2.40. Fragment schematu projektu *lifo*, zsyntetyzowany przez kompilator

Projekt *lifo* oparty jest na prymitywie Quartusa SYNC_RAM, który ma wejściowe i wyjściowe magistrale danych, magistralę adresową odczytu (RADDR) i magistralę adresową zapisu (WADDR), a także sterowany jest sygnałem zegarowym i sygnałem zezwolenia na zapis WE.

Wyniki symulacji funkcjonalnej projektu *lifo* przedstawiono na rysunku 2.41. W przykładzie zapisujemy dane do pamięci typu LIFO w czterech komórkach przez sześć cykli zegara i odczytujemy dane z pamięci danych w sześciu cyklach zegara.



RYS. 2.41. Wyniki symulacji funkcjonalnej projektu *lifo*

Pierwszy cykl zegara zeruje pamięć typu LIFO, co powoduje ustawienie sygnału *empty*, wskazującego że pamięć jest pusta. Następnie, w ciągu czterech cykli zegara, niektóre wartości z magistrali *data_in* są zapisywane do pamięci typu LIFO. W momencie czasu 45ns sygnał *full* jest ustawiany, aby wskazać, że pamięć typu LIFO jest pełna. Próba zapisu do pamięci w następnym cyklu zegara ustawia sygnał *error* w momencie czasu 55ns.

W momencie czasu 70 ns jest ustawiany sygnał *rd* i zaczyna się odczyt danych z pamięci typu LIFO. Dane są odczytywane w odwrotnej kolejności, w jakiej zostały zapisane w pamięci. Próba odczytania nieistniejącej komórki pamięci pod adresem 4 ustawia sygnał *error* w momencie czasu 115ns. Na rysunku 2.41 widać, że pamięć typu LIFO działa poprawnie.

Zachęca się czytelnika do opracowania innego sposobu implementacji pamięci typu LIFO, na przykład polegającego na zliczaniu liczby słów zapisanych w pamięci.

2.10. Używanie atrybutów syntezy podczas opisywania pamięci w języku Verilog

Atrybuty syntezy (Synthesis Attributes) pozwalają na uwzględnienie cech architektury konkretnej rodziny FPGA podczas implementacji pamięci w blokach pamięci wbudowanej. Przy opisywaniu pamięci do implementacji w układach FPGA firmy

Intel można użyć następujących atrybutów syntezy: **ram_init_file**, **ramstyle**, **romstyle** i **max_depth**.

Atrybut **ram_init_file** określa początkową zawartość bloku pamięci w formie *pliku inicjalizacji pamięci* (Memory Initialization File – MIF). Jak każdy atrybut, w kodzie projektu można go zdefiniować w formacie komentarza podczas deklarowania macierzy pamięci, na przykład:

```
reg [7:0] mem [31:0] /* synthesis ram_init_file = "my_ram_init.mif" */
```

lub jako prefiks przed deklaracją macierzy pamięci:

```
(* ram_init_file = "my_ram_init.mif" *) reg [7:0] mem [31:0];
```

W poniższym kodzie zawartość pamięci jest poprzedzona prefiksem podczas deklarowania macierzy pamięci.

```
module single_port_ram_initial_by_attribute
#(parameter DATA_WIDTH=8, parameter ADDR_WIDTH=5)
(input clk, we,
input [DATA_WIDTH-1:0] data_in,
input [ADDR_WIDTH-1:0] address,
output reg [DATA_WIDTH-1:0] data_out);

(* ram_init_file = "ram_init.mif" *)
reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];

always @ (posedge clk)
begin
    if (we)
        ram[address] = data_in;
        data_out = ram[address];
end
endmodule
```

Aby ustawić początkowe wartości komórek pamięci, należy również utworzyć plik inicjujący pamięć z rozszerzeniem .mif, na przykład:

```
WIDTH=8;
DEPTH=32;
```

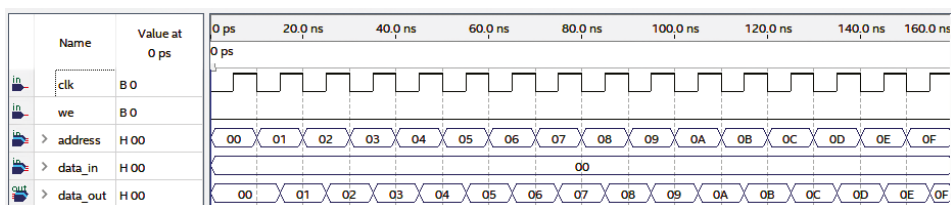
```
ADDRESS_RADIX=UNS;
```

```
DATA_RADIX=UNS;
```

```
CONTENT BEGIN
```

```
0          :    0;
1          :    1;
2          :    2;
...
31         :   31;
END;
```

Wyniki symulacji projektu *single_port_ram_initial_by_attribute* pokazano na rysunku 2.42.



RYS. 2.42. Wyniki symulacji projektu *single_port_ram_initial_by_attribute*

Atrybut **ramstyle** określa typ bloku pamięci wbudowanej FPGA, w którym zaimplementowana zostanie pamięć zadeklarowana w kodzie. Należy pamiętać, że aby zastosować atrybut **ramstyle**, syntezytor musi przydzielić pamięć z kodu projektu, tzn. rozpoznać opis pamięci w kodzie projektu. Aby to zrobić, należy przestrzegać metod opisu pamięci omówionych w podrozdziale 2.1.

Drugim warunkiem zastosowania atrybutu **ramstyle** jest to, żeby architektura FPGA obsługiwała określony blok pamięci wbudowanej FPGA. Atrybut **ramstyle** może przyjmować następujące wartości: „M-RAM”, „M-512”, „M4K”, „M9K”, „M10K”, „M20K”, „M144K” lub „MLAB”. Liczby w oznaczeniu wskazują rozmiar bloku pamięci. Wartość „MLAB” oznacza bloki o podwójnym przeznaczeniu MLAB, które mogą być używane zarówno do implementacji pamięci, jak i logiki. Rozmiar jednego MLAB to 640 bitów. Parametr „M-RAM” odnosi się do bloku pamięci o wielkości 589824 bitów, czasami nazywanego strukturą 4Kx144 bitów. Cechą bloku M-RAM jest to, że on nie może być inicjowany z pliku.

Oprócz tego, w atrybucie **ramstyle** można dodatkowo ustawić wartość „**no_rw_check**”, która informuje syntezytor, że projekt nie ma zamiaru czytać danych podczas zapisu do pamięci. Podanie tego parametru pozwala obniżyć koszt implementacji i zwiększyć szybkość projektowanego modułu pamięci z powodu tego, że syntezytor nie będzie generował *logiki obejścia* (bypass logic) bloku pamięci.

Jako przykład podamy kod pamięci, który jest zaimplementowany w bloku M10K i nie ma możliwości odczytu danych podczas zapisu do pamięci.

```

module single_port_ram_in_M10K
#(parameter DATA_WIDTH=8, parameter ADDR_WIDTH=5)
(input clk, we,
input [DATA_WIDTH-1:0] data_in,
input [ADDR_WIDTH-1:0] address,
output reg [DATA_WIDTH-1:0] data_out);

(* ramstyle = "M10K, no_rw_check" *)
reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];

always @ (posedge clk)
begin
        if (we)
                ram[address] = data_in;
        data_out = ram[address];
end
endmodule

```

Atrybut **romstyle** jest podobny do atrybutu **ramstyle**, z tą różnicą, że odnosi się do pamięci typu ROM.

Atrybut **max_depth** określa maksymalną głębokość (liczbę słów) pamięci, którą można zaimplementować w pojedynczym bloku pamięci wbudowanej. W niektórych przypadkach użycie atrybutu **max_depth** umożliwia bardziej wydajne wykorzystanie pamięci wbudowanej w FPGA. Na przykład wpis:

```
(* max_depth = 512 *)    reg [7:0] ram [1023:0];
```

definiuje maksymalną głębokość pamięci 512 słów. Jednak, aby zaimplementować tablicę RAM, wymagana jest głębokość 1024 słów. W rezultacie syntezy jest zmuszony użyć dwóch bloków M512 po 512 słów każdy w celu realizacji pamięci *ram*. W przypadku braku atrybutu **max_depth** = 512 syntezy wybrałby blok M4K, który owszem przechowuje macierze pamięci *ram*, ale 3K słów pamięci pozostałoby niewykorzystanych.

2.11. Wnioski

Metody opisu pamięci w języku Verilog są bardzo różnorodne i ich wybór pozostaje w gestii projektanta. Podczas implementacji pamięci na FPGA należy przestrzegać następującej zasady: kompilator systemu Quartus musi rozpoznać opis

pamięci w kodzie i zaimplementować pamięć w układzie, używając jednego z prymitywów.

Aby zwiększyć szybkość projektu i zaoszczędzić zasoby logiczne, ważne jest zastosowanie opisu, w którym pamięć jest zaimplementowana we wbudowanym bloku pamięci FPGA. Ze statystyk kompilacji można dowiedzieć się, gdzie została zaimplementowana pamięć – czy we wbudowanym bloku pamięci FPGA, czy przy użyciu komórek logicznych.

Macierz pamięci w języku Verilog jest zadeklarowana jako jednowymiarowa tablica zmiennych wektorowych typu **reg**, gdzie rozmiar zmiennych określa rozmiar słów pamięci, a rozmiar tablicy określa liczbę słów w pamięci. W języku Verilog rozmiary pamięci mogą być wygodnie określane jako parametry modułu.

Niezależnie od liczby portów, wszystkie rozważane typy pamięci mają tylko jedną macierz pamięci.

Przed opisaniem pamięci w języku Verilog należy określić wymagany tryb pracy wyjść pamięci: „nowe dane” lub „stare dane”.

Jednoportowa pamięć typu RAM może wykonywać tylko jedną operację w danym momencie: odczyt lub zapis. Pamięć jednoportowa ma jedną wejściową magistralę danych, jedną wyjściową magistralę danych i jedną magistralę adresową. Ponadto pamięć jednoportowa może mieć jeden lub dwa sygnały synchronizacji, a także inne sygnały sterujące.

Do praktycznego wykorzystania podczas implementacji pamięci na FPGA, możemy polecić projekt *single_port_ram_single_clk_v2* w trybie wyjścia „stare dane” oraz projekt *single_port_ram_single_clk_v3* w trybie „nowe dane”. W tych projektach wartości wyjściowe są jawnie określone w kodzie projektu.

W języku Verilog można również opisać pamięć z wyjściami kombinacyjnymi i z rejestrami na wejściach adresowych. Jednak firma Intel nie zaleca tych metod opisu pamięci do praktycznego zastosowania.

Architektura FPGA firmy Intel nie zapewnia zerowania zawartości pamięci przez sygnał *reset*, jak w przerzutnikach czy rejestrach. Jednak rejestry wyjściowe bloków pamięci wbudowanej FPGA są resetowane po włączeniu zasilania. Dlatego po włączeniu zasilania stan sygnałów na wyjściach bloków pamięci wbudowanej jest zawsze równy zero.

Wartości początkowe można zapisywać w pamięci jawnie w kodzie projektu przy użyciu proceduralnego bloku **initial** lub przy użyciu funkcji systemowych **\$readmemb** i **\$readmemh**. W tym drugim przypadku zawartość pamięci jest odczytywana z pliku tekstowego. Dane w pliku tekstowym są przedstawiane w postaci binarnej (dla funkcji **\$readmemb**) lub szesnastkowej (dla funkcji **\$readmemh**).

Pamięć jednoportowa może mieć dwa sygnały zegarowe: jeden synchronizuje operację zapisu, a drugi synchronizuje operację odczytu.

Prosta dwuportowa pamięć ma dwa porty adresowe: jeden do zapisu, a drugi do odczytu. Wyjścia prostej pamięci dwuportowej mogą również działać jako „nowe dane” lub „stare dane”. Ponadto pamięć dwuportowa może mieć jeden lub dwa sygnały synchronizacji.

Rzeczywista pamięć dwuportowa ma dwa niezależne porty: port A i port B, z własnymi magistralami danych, adresami i sygnałami sterującymi. Każdy port rzeczywistej pamięci dwuportowej może działać niezależnie, tzn. wykonywać dowolne operacje odczytu i zapisu, z wyjątkiem jednoczesnego zapisu do tej samej komórki pamięci. W tym ostatnim przypadku, zawartość komórki pamięci jest uważana za nieokreśloną. Wyjścia rzeczywistej pamięci dwuportowej mogą również działać jako „nowe dane” lub „stare dane”.

Typ pamięci ROM w języku Verilog można opisać za pomocą instrukcji **case**, gdzie zawartość komórek pamięci jest opisana w kodzie projektu. Pamięć typu ROM można również opisać jako RAM z obowiązkową inicjalizacją pamięci i tylko operacją odczytu. Pamięć typu ROM, podobnie jak pamięć typu RAM, może być jedno- lub dwuportowa, a także może mieć jeden lub dwa sygnały synchronizacji.

Duże rejestry przesuwające można zaimplementować w pamięci wbudowanej FPGA, aby poprawić wydajność projektu i zaoszczędzić zasoby logiczne FPGA. Rejestr przesuwający zostanie zaimplementowany przez kompilator w bloku pamięci wbudowanej FPGA, jeśli całkowita długość rejestru jest większa niż 67 bitów lub, w przypadku magistrali, całkowita liczba bitów rejestru jest większa niż 32.

Pamięć typu FIFO (First-In, First-Out) to rodzaj pamięci jednoportowej, w której dane, które docierają (zapisują) do niej jako pierwsze, są najpierw odczytywane z pamięci. Pamięć typu FIFO posiada dwa porty danych: wejściowy i wyjściowy. Oprócz tradycyjnych sygnałów sterujących *clk* i *reset*, pamięć typu FIFO ma również wejście zezwolenia zapisu *wr_en* i wejście umożliwiające odczyt *rd_en*. Pamięć typu FIFO ma dwa dodatkowe wyjścia: *empty*, sygnalizujące że pamięć jest pusta i *full*, sygnalizujące że pamięć jest pełna.

Głównym problemem przy projektowaniu pamięci typu FIFO jest zdefiniowanie wartości sygnałów *empty* i *full*. Istnieje kilka podejść do rozwiązania tego problemu: liczenie liczby zapisanych słów w pamięci, określenie tendencji operacji zapisu i odczytu – czy pamięć jest zapełniana (filling) lub opróżniana (emptying), przechowywanie stanów pamięci i używanie automatu skończonego itp.

Pamięć typu FIFO może być synchroniczna lub asynchroniczna. W przypadku synchronicznej pamięci typu FIFO można zastosować dwa sygnały synchronizacyjne, jeden do zapisu danych, drugi do odczytu danych. Pamięć typu FIFO może mieć również wyjścia *almost_full* (prawie pełne) i *almost_empty* (prawie puste).

Pamięć typu LIFO (Last-In, First-Out) jest pamięcią typu stosowego, w której ostatnio do niej zapisywane dane są najpierw odczytywane z pamięci.

Przy implementacji pamięci na układach FPGA firmy Intel w systemie Quartus można wykorzystać następujące atrybuty syntezy języka Verilog: **ram_init_file**, **ram_style**, **romstyle** i **max_depth**.

Atrybut **ram_init_file** określa początkową zawartość przydzielonego bloku pamięci w formacie pliku inicjalizacji pamięci .mif (Memory Initialization File).

W ten sposób można ustawić początkowe wartości modułu pamięci:

- w kodzie projektu w języku Verilog w bloku **initial** przy użyciu instrukcji **for** lub **case**;

- z pliku tekstowego używając funkcji systemowych **\$readmemb** i **\$readmemh**;
- z pliku inicjalizacji pamięci MIF (.mif) używając atrybutu **ram_init_file**.

Atrybut **ramstyle** określa typ bloku pamięci wbudowanej FPGA, w którym zaimplementowana zostanie pamięć zadeklarowana w kodzie. Dodatkowo atrybut **ramstyle** można ustawić na wartość „**no_rw_check**”, która mówi syntezaotorowi, że projekt nie powinien czytać danych podczas zapisywania do pamięci. Podanie tego parametru pozwala obniżyć koszt realizacji i zwiększyć szybkość działania modułu pamięci.

Atrybut **romstyle** jest podobny do atrybutu **ramstyle**, z tą różnicą, że odnosi się do pamięci typu ROM.

Atrybut **max_depth** określa maksymalną głębokość (liczbę słów) pamięci, którą można zaimplementować w pojedynczym bloku pamięci wbudowanej. W niektórych przypadkach użycie atrybutu **max_depth** umożliwia bardziej wydajne wykorzystanie pamięci wbudowanej FPGA.

3. Projektowanie pamięci z wykorzystaniem bloków IP i edytora parametrów systemu Quartus

Moduły pamięci w systemach wbudowanych realizowanych w FPGA można również budować przy użyciu bloków IP i *Edytora parametrów* (Parameter Editor) systemu Quartus. W tym celu wykorzystuje się specjalne megafunkcje opracowane przez firmę Intel. Megafunkcje pozwalają w pełni wykorzystać wszystkie właściwości architektur bloków pamięci wbudowanej FPGA, z których część nie jest dostępna podczas opisywania pamięci w języku Verilog.

W tym rozdziale omówiono sposoby tworzenia modułów pamięci typu RAM, ROM i FIFO, a także rejestrów przesuwających przy użyciu bloków IP systemu Quartus.

3.1. Katalog IP do tworzenia modułów pamięci

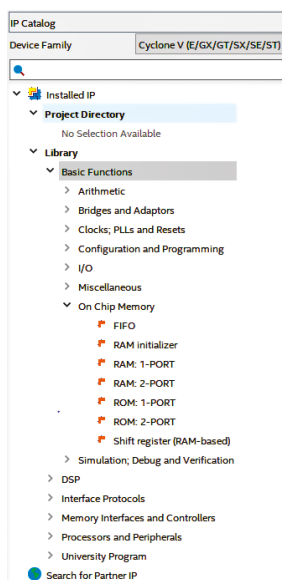
System Quartus umożliwia tworzenie gotowych projektów z wykorzystaniem bloków IP. Odbywa się to za pomocą narzędzia programowego Quartus zwanego *Katalogiem IP* (IP Catalog) lub *Edytorem parametrów*.

Aby wyświetlić okno Katalogu IP (jeśli nie jest wyświetlane na ekranie), wybierz w menu Quartus **Tools > IP Catalog**. Aby utworzyć moduł pamięci przy użyciu bloku IP, wybierz **Library > Basic Functions > On Chip Memory** w oknie Katalogu IP. Pojawi się okno dialogowe (rys. 3.1) zawierające listę wszystkich typów pamięci, które można utworzyć za pomocą Katalogu IP.

Na rysunku 3.1 można zauważyć, że korzystając z Katalogu IP można stworzyć jedno- i dwuportowe pamięci typu RAM i ROM, pamięć typu FIFO oraz rejestr przesuwający oparty na blokach pamięci wbudowanej FPGA. Zauważmy, że przy pomocy Katalogu IP możliwe jest najpełniejsze wykorzystanie właściwości architektur pamięci wbudowanej FPGA, co może nie być dostępne podczas opisywania pamięci w języku Verilog.

Bloki IP modułów pamięci są budowane przy użyciu megafunkcji. Megafunkcje ALTSYNCRAM i ALTDPRAM służą do tworzenia modułów pamięci typu RAM i ROM [15]. Oprogramowanie Quartus automatycznie wybiera jeden z tych bloków

do implementacji modeli pamięci. Wybór zależy od urządzenia docelowego, modeli pamięci oraz właściwości pamięci typu RAM i ROM.



RYS. 3.1. Okno Katalogu IP podczas tworzenia modułu pamięci

Pamięć typu FIFO oparta jest na dwóch megafunkcjach: SCFIFO (single-clock FIFO) i DCFIFO (dual-clock FIFO) [13]. Ponadto megafunkcja DCFIFO_MIXED_WIDTHS służy do budowy pamięci typu FIFO o różnych szerokościach danych wejściowych i wyjściowych.

Rejestr przesuwający w bloku RAM jest konstruowany przy użyciu megafunkcji ALTSHIFT_TAPS [16].

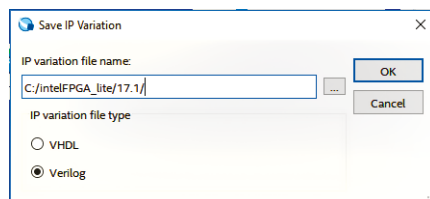
Bardziej szczegółowe informacje na temat portów i parametrów tych megafunkcji oraz specyfikacji ich wykorzystania można znaleźć w [13,15,16].

3.2. Tworzenie modułów pamięci typu RAM i ROM

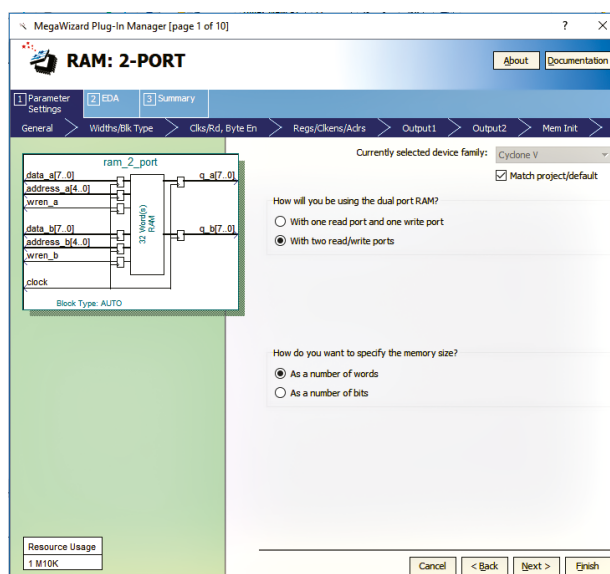
Jako przykład rozpatrzmy proces tworzenia dwuportowej pamięci typu RAM za pomocą narzędzia Katalog IP. Aby to zrobić, w oknie na rysunku 3.1 wybierz **RAM: 2-PORT**. Pojawi się okno dialogowe (rys. 3.2), w którym należy określić katalog do zapisania utworzonych plików oraz język projektowania (VHDL lub Verilog) używany do opisu tworzonego modułu.

Po kliknięciu OK uruchamiany jest MegaWizard Plug-In Manager, zwany także *Edytorem parametrów*. Edytor parametrów poprowadzi cię krok po kroku przez proces tworzenia bloków IP. W naszym przypadku, na ekranie zostanie wyświetlona pierwsza

strona (rys. 3.3) z 10 stron służących do sparametryzowania dwuportowej pamięci typu RAM. Zwróćmy uwagę, że podczas tworzenia konkretnego modułu nie wszystkie z 10 stron pojawiają się na ekranie. Zależy to od wartości wprowadzonych parametrów.



RYŚ. 3.2. Okno do zapisywania plików modułu pamięci utworzonego za pomocą Katalogu IP



RYŚ. 3.3. Pierwsza z dziesięciu stron Edytora parametrów podczas tworzenia pamięci dwuportowej

Pierwsze dwa wiersze poniżej nagłówka każdej strony Edytora parametrów pokazują stan procesu tworzenia bloków IP. Na pierwszej stronie Edytora parametrów podane są ogólne parametry tworzonego modułu pamięci:

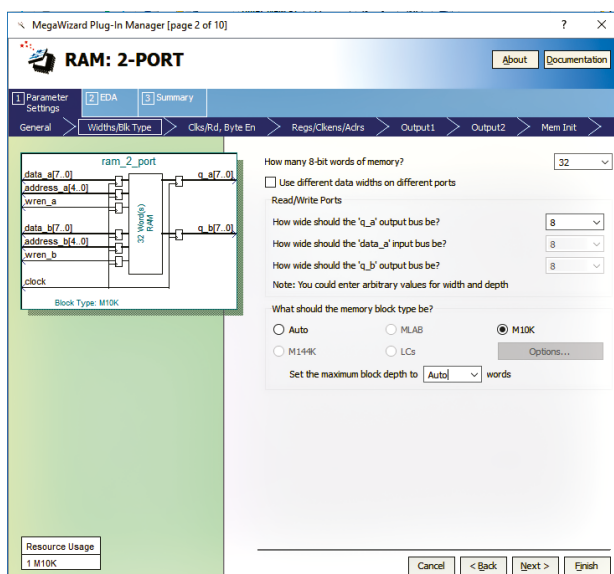
- rodzina docelowego układu FPGA;
- zgodność z projektem (ustawienie domyślne);
- jak będzie używana pamięć dwuportowa:
 - z jednym portem odczytu i jednym portem zapisu;
 - z dwoma portami odczytu/zapisu;
- w jaki sposób określany jest rozmiar pamięci:
 - jako liczba słów;
 - jako liczba bitów.

Struktura tworzonego modułu pamięci jest pokazana w lewym górnym rogu każdej strony Edytora parametrów. W procesie określania wartości parametrów struktura ta może ulec zmianie. Używane zasoby FPGA są wyświetlane w lewym dolnym rogu każdej strony Edytora parametrów.

Dolna linia każdej strony Edytora parametrów sugeruje możliwe działania ze strony użytkownika: przerwanie procesu (**Cancel**), cofnięcie się o jeden krok (< Back), przejście do następnego kroku (Next >) lub zakończenie procesu (**Finish**). Typową czynnością w edytorze jest kliknięcie przycisku **Next**.

Na kolejnej stronie Edytora parametrów (rys. 3.4) można określić niektóre parametry tworzonego modułu pamięci. W naszym przykładzie proponuje się wskazanie:

- liczba słów w pamięci;
- czy na różnych portach będą używane różne szerokości magistrali danych;
- szerokość magistral wyjściowych q_a i q_b ;
- szerokość wejściowej magistrali $data_a$;
- typ bloku pamięci: Auto, MLAB, M10K, M144K lub użycie komórek logicznych do implementacji tworzonego modułu pamięci;
- maksymalna głębokość bloku pamięci, mierzona liczbą słów.

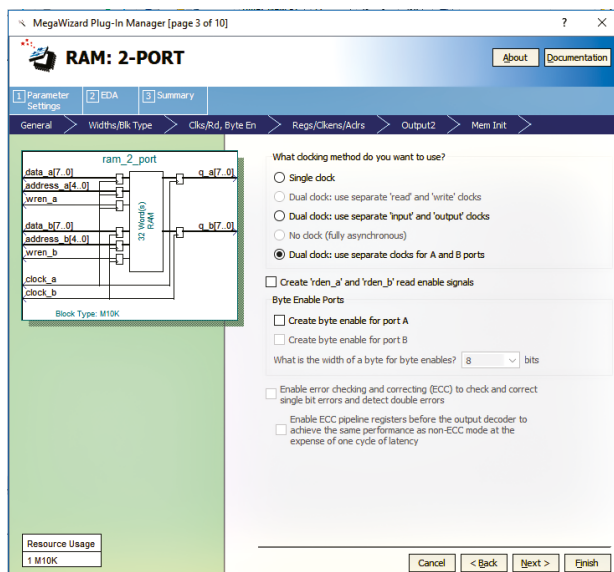


RYŚ. 3.4. Druga strona Edytora parametrów podczas tworzenia pamięci dwuportowej

Na kolejnej stronie Edytora parametrów (rys. 3.5) można określić dla tworzonego modułu pamięci następujące parametry:

- metoda synchronizacji:
 - z jednym sygnałem synchronizacji;
 - z dwoma sygnałami synchronizacji, przy użyciu oddzielnych sygnałów zegarowych do odczytu i zapisu;

- z dwoma sygnałami synchronizacji, przy użyciu oddzielnych sygnałów zegarowych dla wejść i wyjść;
- brak sygnału synchronizacji (pamięć asynchroniczna);
- z dwoma sygnałami synchronizacji, przy użyciu oddzielnych sygnałów zegarowych dla portu A i portu B;
- czy tworzyć sygnały zezwolenia na odczyt *rden_a* i *rden_b*;
- czy tworzyć sygnały zezwolenia na bajty dla portu A;
- czy tworzyć sygnały zezwolenia na bajty dla portu B;
- szerokość (w bitach) bajtu dla mechanizmu zezwolenia bajtów;
- włączać lub wyłączać *sprawdzanie i korygowanie błędów* (error checking and correcting – ECC) w celu sprawdzania i korygowania pojedynczych błędów oraz wykrywania błędów podwójnych;
- włączać lub wyłączać potokowe rejestry ECC przed dekodерem wyjściowym, aby zachować taką samą wydajność, jak w trybie innym niż ECC kosztem jednego cyklu opóźnienia.



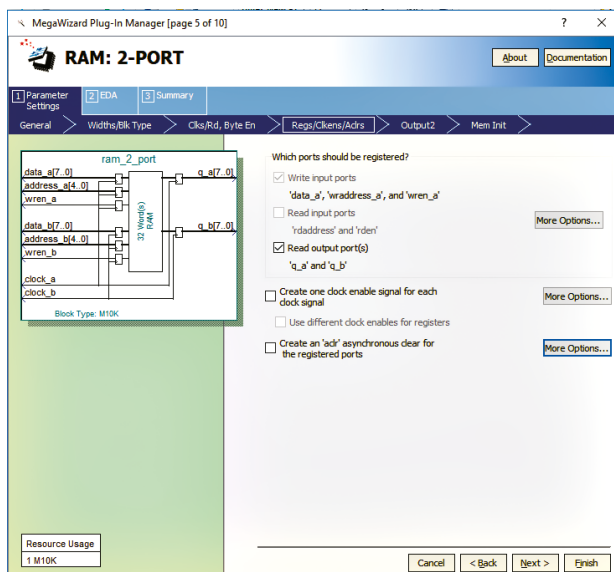
RYS. 3.5. Trzecia strona Edytora parametrów podczas tworzenia pamięci dwuportowej

Zwróć uwagę, że podczas korzystania z mechanizmu zezwolenia bajtów rozmiar słów (bajtów) może wynosić 5, 8, 9 lub 10 bitów.

Na kolejnej stronie Edytora parametrów (rys. 3.6) można:

- określić, które porty powinny być rejestrowe:
 - porty wejściowe do zapisu (*data_a*, *wraddress_a*, *wren_a*);
 - porty wejściowe do odczytu (*rdaddress*, *rden*);
 - porty wyjściowe do odczytu (*q_a*, *q_b*);

- utworzyć jeden sygnał zezwolenia synchronizacji dla każdego sygnału synchronizacji, przy czym można określić:
 - czy używać różnych sygnałów zezwolenia synchronizacji dla rejestrów;
- utworzyć asynchroniczny sygnał resetowania *aclr* dla wszystkich portów rejestrowych.

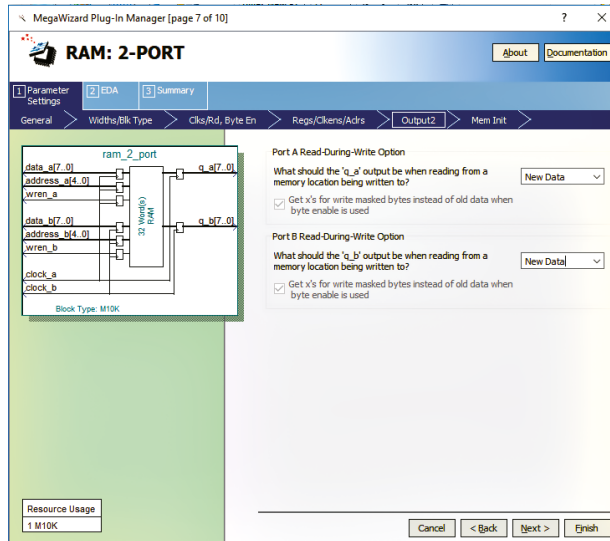


rys. 3.6. Piąta strona Edytora parametrów podczas tworzenia pamięci dwuportowej

Ponadto dla każdej grupy parametrów na rysunku 3.6 można dodatkowo określić wartości parametrów, które staną się dostępne po kliknięciu przycisku **More Options...**

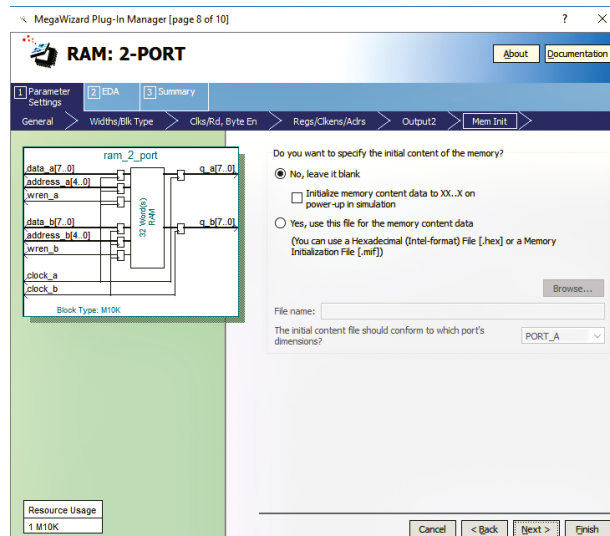
Na kolejnej stronie Edytora parametrów (rys. 3.7) dla każdego z portów A i B wskazane są wartości wyjść w trybie *odczytu podczas zapisu* (Read-During-Write-RDW). W tym przypadku dla każdego portu jest określone:

- jakie powinno być wyjście portu *q_a* lub *q_b* podczas odczytu z komórki pamięci podczas zapisu: *nowe dane* (New Data), *stare dane* (Old Data) lub *nieznane wartości* (Don't Care);
- czy uzyskać nieznane wartości x, aby zapisać zamaskowane bajty zamiast starych danych, gdy używana jest opcja zezwolenia bajtów.



RYS. 3.7. Siódma strona Edytora parametrów podczas tworzenia pamięci dwuportowej

Kolejna strona Edytora parametrów (rys. 3.8) pozwala określić sposób inicjalizacji zawartości pamięci.



RYS. 3.8. Ósma strona Edytora parametrów podczas tworzenia pamięci dwuportowej

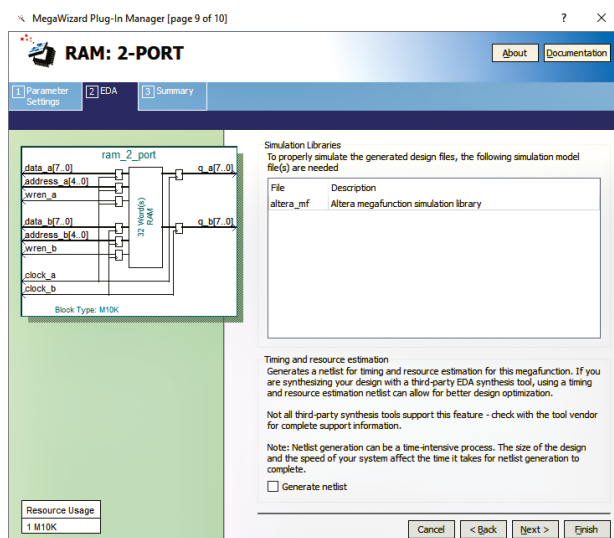
Na ósmej stronie Edytora parametrów można sprecyzować czy:

- początkowe wartości pamięci mają być określone;
- podczas symulacji, po włączeniu zasilania, zawartość pamięci jest inicjalizowana nieznanymi wartościami XX...X;

- początkowe wartości pamięci są opisane za pomocą pliku szesnastkowego (.hex) lub pliku inicjalizacji pamięci (.mif).

Podczas wyboru ostatniej opcji użyj przycisku **Browse...**, aby określić lokalizację pliku z początkowymi wartościami pamięci. Ponadto konieczne jest uszczegółowienie, wymiarom którego portu (A lub B) powinien odpowiadać plik początkowych wartości pamięci.

Następna strona Edytora parametrów (rys. 3.9) dotyczy narzędzi programowych używanych do symulacji projektu.



RYŚ. 3.9. Dziewiąta strona podczas tworzenia pamięci dwuportowej

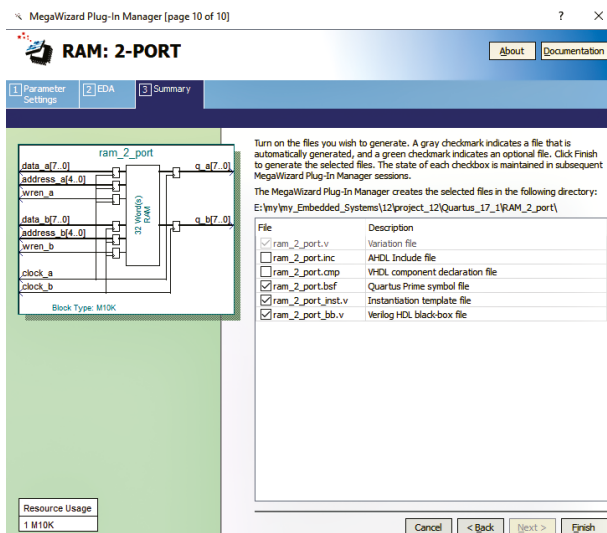
Na dziewiątej stronie Edytora parametrów możesz określić pliki związane z oprogramowaniem używanym do symulacji projektu. Domyślnie do symulacji projektu używana jest megafunkcja *altera_mf* firmy Intel (Altera). Jednak użytkownik może określić inne narzędzie programowe, takie jak oprogramowanie firmy Aldec lub firmy Cadence. Na tej samej stronie można wskazać, czy na potrzeby symulacji ma zostać wygenerowany plik *listy sieci* (netlist).

Ostatnia strona Edytora parametrów (rys. 3.10) pozwala zdefiniować wygenerowane pliki wyjściowe.

Szare znaczniki wyboru na tej stronie oznaczają pliki, które są generowane automatycznie, a zielone znaczniki wyboru oznaczają pliki, które są generowane na żądanie użytkownika. Można wygenerować następujące pliki:

- plik opisujący moduł pamięci w języku Verilog;
- plik załącznika w języku AHDL (.inc);
- plik deklaracji komponentu w języku VHDL (.cmp);
- plik obrazu graficznego modułu pamięci (.bsf);

- szablon pliku do tworzenia instancji modułu pamięci (<name>_inst.v);
- plik „czarnej skrzynki” (black box) modułu pamięci w języku Verilog (<name>_bb.v).



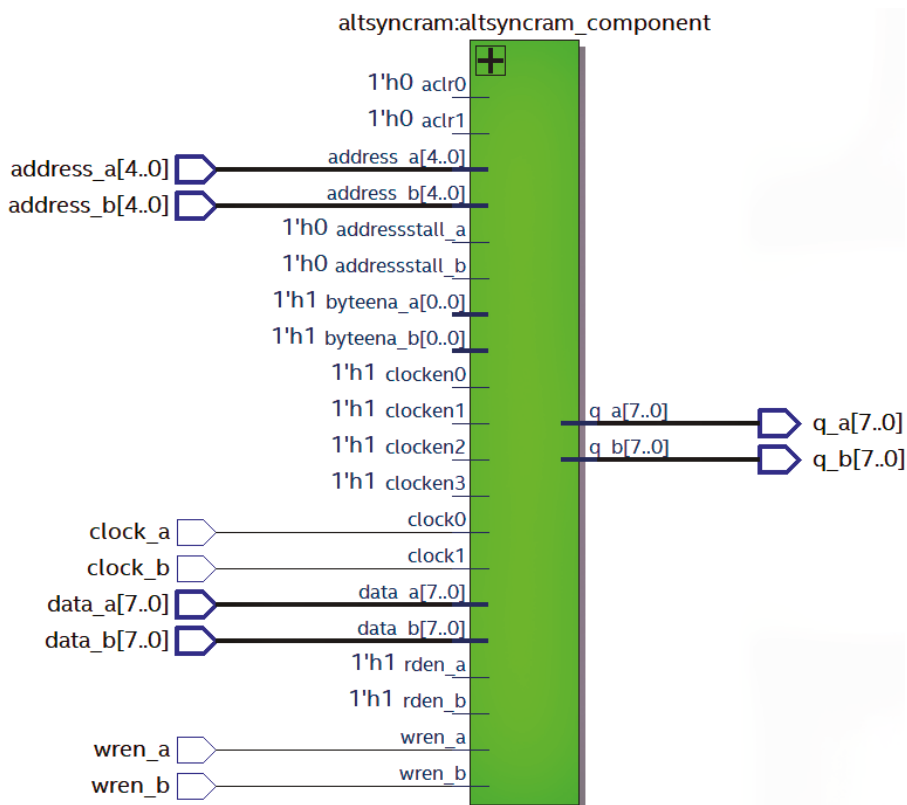
RYS. 3.10. Dziesiąta strona podczas tworzenia pamięci dwuportowej

Po kliknięciu przycisku **Finish** pojawi się okno dialogowe, w którym zostanie wyświetlona propozycja dodania pliku modułu pamięci do aktywnego projektu. Plik modułu pamięci Verilog (<name>.v) wygenerowany przez Edytor parametrów można przeglądać w dowolnym edytorze tekstu. Próba otwarcia tego pliku w Edytorze tekstowym systemu Quartus powoduje automatyczne uruchomienie Edytora parametrów bloków IP dla modułu pamięci. Można tu wprowadzić dowolne zmiany parametrów modułu pamięci, które zostaną następnie zapisane w odpowiednich plikach wyjściowych.

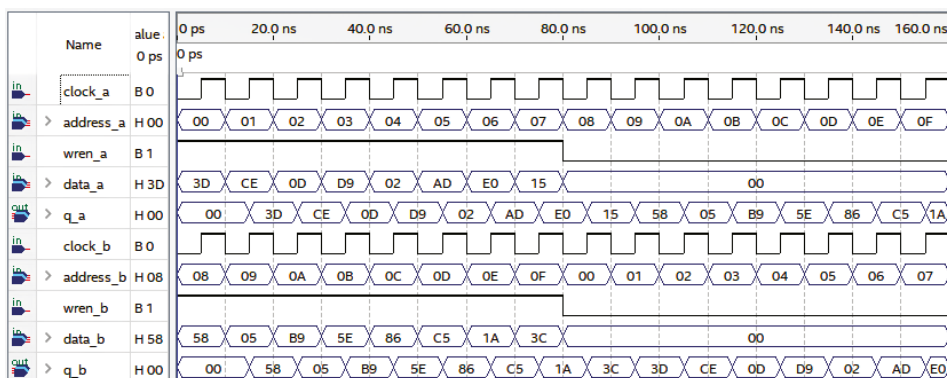
Wyniki syntezy i symulacji rozważanego przykładu dwuportowego modułu pamięci typu RAM pokazano, odpowiednio, na rysunkach 3.11 i 3.12.

Rysunek 3.11 pokazuje, że rozważany moduł pamięci jest zbudowany na podstawie megafunkcji **altsyncram**. Wskazuje na to połączenie portów utworzonego modułu z portami megafunkcji **altsyncram**, a także wartości, które są domyślnie przekazywane do innych portów megafunkcji **altsyncram**. Więcej informacji na temat wykorzystania funkcji **altsyncram** można znaleźć w [15].

Rysunek 3.12 przedstawia przykład jednoczesnego zapisu danych w pierwszych ośmiu cyklach zegara za pomocą portów A i B do różnych komórek pamięci: przez port A dane są zapisywane do komórek pamięci o adresach 00, ..., 07, a przez port B – do komórek o adresach 08, ..., 0F. W kolejnych ośmiu cyklach zegarowych następuje odczyt krzyżowy: z portu A odczytywane są dane z komórek o adresach 08, ..., 0F, a z portu B – z komórek o adresach 00, ..., 07.



RYS. 3.11. Widok projektu *ram_2_port* na poziomie RTL

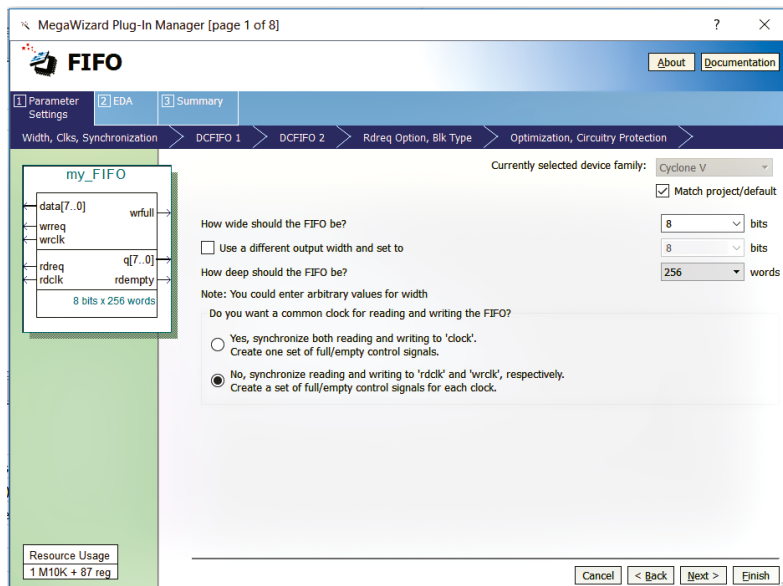


RYS. 3.12. Wyniki symulacji projektu *ram_2_port*

W podobny sposób można tworzyć pamięci typu ROM z jednym lub dwoma portami za pomocą Katalogu IP i Edytora parametrów. Cechą szczególną tworzenia modułów pamięci typu ROM jest to, że konieczne jest wskazanie pliku, który określa zawartość pamięci.

3.3. Tworzenie modułu pamięci typu FIFO

Rozpatrzmy tworzenie modułu pamięci typu FIFO z wykorzystaniem Katalogu IP systemu Quartus. Aby to zrobić, wybierzmy **Library > Basic Functions > On Chip Memory** w oknie Katalogu IP. W wyświetlonym oknie wybierzmy **FIFO**. Na ekranie pojawi się okno do określenia lokalizacji generowanych plików (rys. 3.2). Po określeniu katalogu roboczego ukaże się pierwsze okno Edytora parametrów (rys. 3.13), służące do tworzenia modułu pamięci typu FIFO.



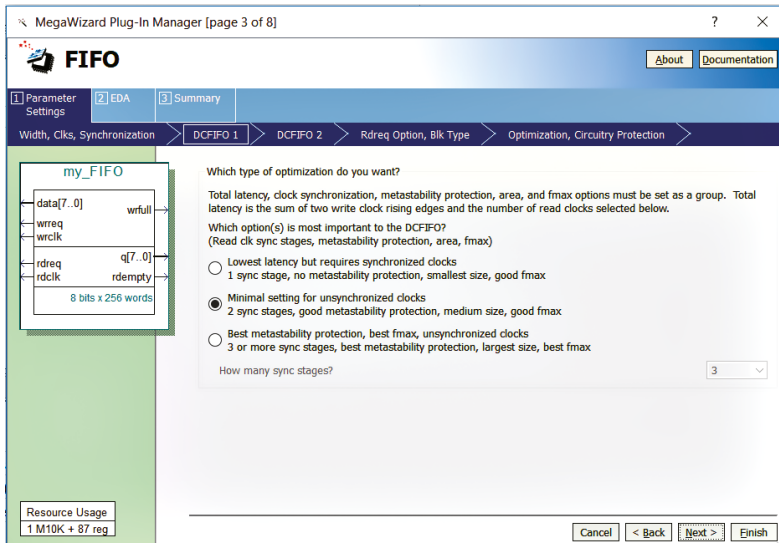
RYS. 3.13. Pierwsza z ośmiu stron Edytora parametrów podczas tworzenia modułu pamięci typu FIFO

W oknie na rysunku 3.13 wskazane są następujące parametry:

- wybrana docelowa rodzina FPGA;
- zgodność z projektem (ustawienia domyślne);
- szerokość (w bitach) słów w pamięci typu FIFO;
- czy używać różnych szerokości portów;
- głębokość (wyrażona liczbą słów) pamięci typu FIFO;
- czy synchronizować operacje odczytu i zapisu jednym sygnałem zegarowym *clock* i tworzyć jeden zestaw sygnałów sterujących *full/empty*;
- czy synchronizować operacje odczytu i zapisu różnymi sygnałami synchronizacji, odpowiednio *rdclk* i *wrclk*, i utworzyć własny zestaw sygnałów sterujących *full/empty* dla każdego sygnału synchronizacji.

Kolejna strona Edytora parametrów (rys. 3.14) definiuje grupę parametrów związanych z optymalizacją projektu: liczbę etapów synchronizacji, stopień ochrony przed

metastabilnością, zajmowany obszar na chipie (koszt realizacji) oraz maksymalną częstotliwość pracy (wydajność projektu).



rys. 3.14. Trzecia strona Edytora parametrów podczas tworzenia pamięci typu FIFO

Na stronie na rysunku 3.14 można określić następujące trzy grupy parametrów:

- 1 stopień synchronizacji sygnałów, najmniejsze opóźnienie, brak zabezpieczenia przed metastabilnością, najniższy koszt realizacji, dobra częstotliwość pracy;
- 2 stopnie synchronizacji sygnałów, dobra ochrona przed metastabilnością, średni koszt wykonania, dobra częstotliwość pracy;
- 3 lub więcej etapów synchronizacji, lepsza ochrona przed metastabilnością, wyższy koszt realizacji, najwyższa częstotliwość pracy.

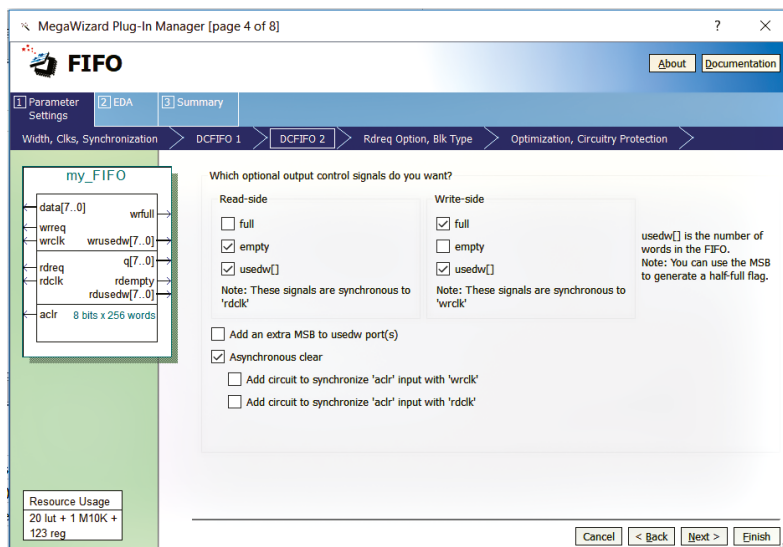
Kolejna strona Edytora parametrów (rys. 3.15) pozwala na zdefiniowanie niezbędnych dodatkowych sygnałów pamięci typu FIFO.

Na czwartej stronie Edytora parametrów, zarówno od strony zapisu, jak i od strony odczytu pamięci typu FIFO, można wygenerować następujące sygnały:

- *full* – wskazuje, że pamięć typu FIFO jest pełna;
- *empty* – wskazuje, że pamięć typu FIFO jest pusta;
- *usedw[]* – wskazuje liczbę słów w pamięci typu FIFO.

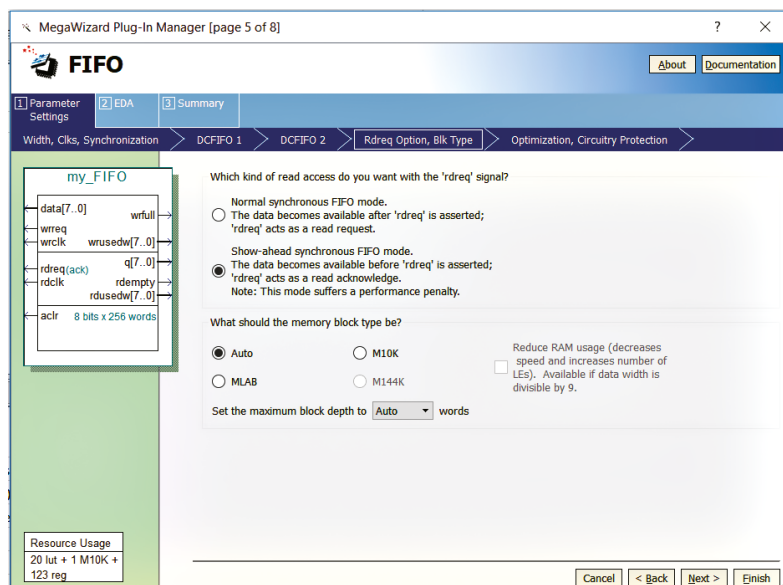
Ponadto na tej samej stronie można:

- dodać dodatkowy bit MSB do portu *usedw*, aby poinformować, że pamięć typu FIFO jest w połowie zapełniona;
- dodać układ synchronizujący wejście *aclr* z zegarem *wrclk* lub *rdclk*.



RYS. 3.15. Czwarta strona Edytora parametrów podczas tworzenia pamięci typu FIFO

Kolejna strona Edytora parametrów (rys. 3.16) definiuje tryb odczytu oraz typ bloku pamięci wbudowanej FPGA.



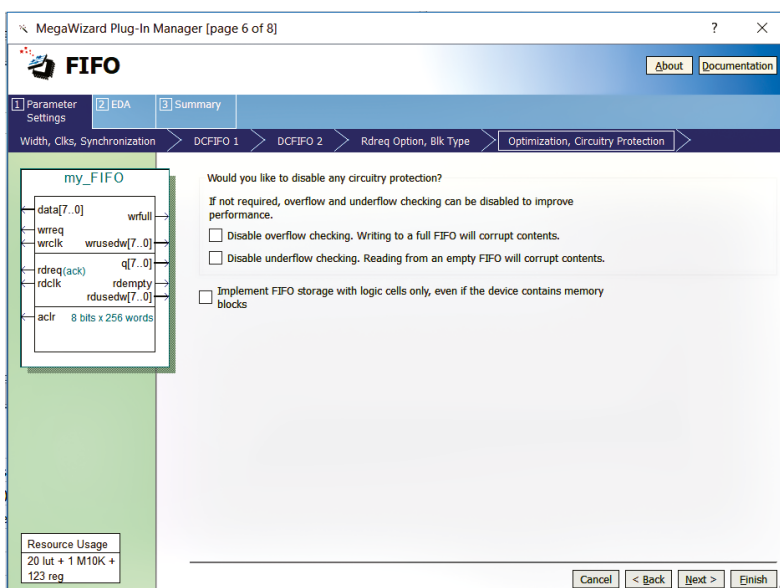
RYS. 3.16. Piąta strona Edytora parametrów podczas tworzenia pamięci typu FIFO

Na piątej stronie Edytora parametrów można określić następujące parametry:

- normalny tryb synchronizacji odczytu z pamięci typu FIFO: dane stają się dostępne po ustawieniu sygnału *rdreq*, sygnał *rdreq* działa jak żądanie odczytu;

- tryb synchronizacji typu show-ahead: dane są udostępniane przed ustawieniem sygnału *rdreg*, *rdreg* działa jak potwierdzenie odczytu;
- rodzaj bloku pamięci: Auto, MLAB, M10K, M144K lub komórki logiczne wykorzystywane do implementacji tworzonego modułu pamięci;
- maksymalna głębokość bloku pamięci (w słowach).

Kolejna strona Edytora parametrów (rys. 3.17) określa stopień optymalizacji projektu modułu pamięci typu FIFO oraz zabezpieczenia przed cyklicznym zapisem do pamięci.



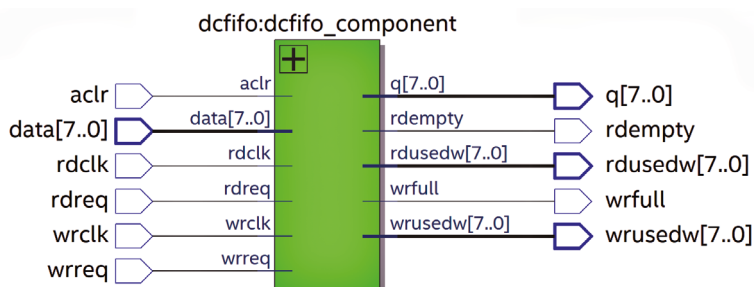
RYŚ. 3.17. Szósta strona Edytora parametrów podczas tworzenia pamięci typu FIFO

Na szóstej stronie Edytora parametrów można:

- wyłączyć sprawdzanie *przepełnienia* (overflow) (zapis do pełnej pamięci typu FIFO spowoduje uszkodzenie zawartości pamięci);
- wyłączyć sprawdzanie *niedomiaru* (underflow) (odczyt z pustej pamięci typu FIFO spowoduje uszkodzenie zawartości pamięci);
- implementować moduł pamięci typu FIFO tylko na komórkach logicznych, nawet jeśli FPGA zawiera bloki pamięci.

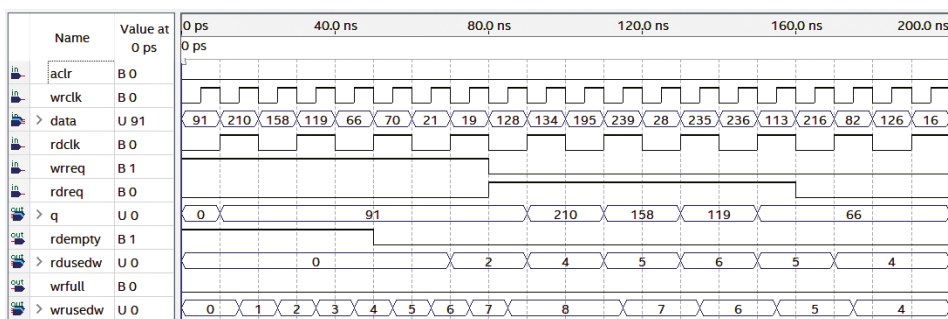
Kolejne dwie strony Edytora parametrów faktycznie pokrywają się ze stronami na rysunkach 3.9 i 3.10 dla pamięci typu RAM. Definiują oprogramowanie używane do symulacji projektu i generowane pliki wyjściowe.

Wyniki syntezy i symulacji rozważanego przykładu pamięci typu FIFO pokazano, odpowiednio, na rysunkach 3.18 i 3.19.



RYS. 3.18. Widok projektu *FIFO* na poziomie RTL

Rysunek 3.18 pokazuje, że rozważany moduł pamięci typu FIFO jest oparty na megafunkcji **dcfifo**. Pokazano tu połączenie portów utworzonego modułu z portami megafunkcji **dcfifo**, a także wartości, które są domyślnie przekazywane do innych portów megafunkcji **dcfifo**. Więcej informacji na temat wykorzystania funkcji **dcfifo** można znaleźć w [13].



RYS. 3.19. Wyniki symulacji projektu *FIFO*

Na rysunku 3.19, w pierwszych ośmiu cyklach synchronizacji, dane są zapisywane do pamięci typu FIFO w kolejności: 91, 210, 158, 119, 66, 70, 21, 19. W kolejnych ośmiu cyklach dane są odczytywane z pamięci typu FIFO. Ponieważ częstotliwość zegara odczytu *rdclk* (20 MHz) jest dwa razy większa niż częstotliwość zegara zapisu *wrclk* (10 MHz), wyjście *q[]* sekwencyjnie generuje wartości 91, 210, 158, 119, 66. Liczbę słów zapisanych w pamięci typu FIFO prawidłowo pokazuje wyjście *wrusedw*.

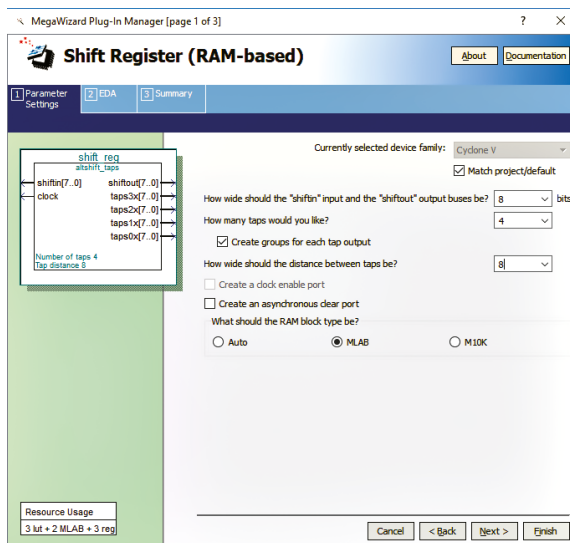
3.4. Tworzenie rejestru przesuwającego w bloku pamięci wbudowanej FPGA

Aby utworzyć rejestr przesuwający w bloku pamięci wbudowanej FPGA przy użyciu bloku IP systemu Quartus, należy wybrać **Shift register (RAM-based)** w oknie

Katalogu IP (rys. 3.1). W rezultacie edytor parametrów zaczyna tworzyć rejestr przesuwający w bloku pamięci typu RAM.

W pierwszym oknie Edytora parametrów (rys. 3.20) można określić:

- szerokość (w bitach) magistrali wejściowej (*shiftin*) i magistrali wyjściowej (*shiftout*);
- liczbę odczepów (taps);
- czy tworzyć grupy dla każdego wyjścia odczepu;
- odległość (w bitach) między odczepami;
- czy utworzyć port zezwolenia synchronizacji;
- czy utworzyć port asynchronicznego resetowania;
- typ wbudowanego bloku pamięci FPGA (Auto, MLAB lub M10K), w którym będzie realizowany rejestr przesuwający.

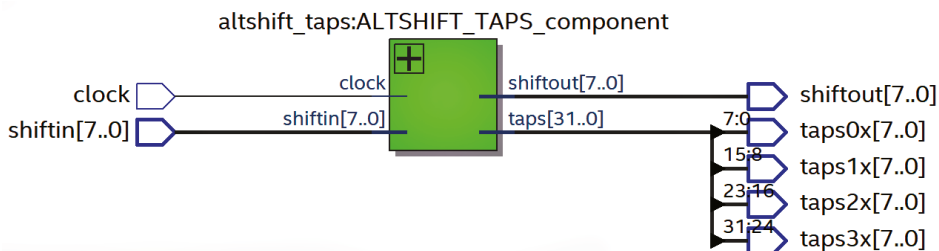


RYŚ. 3.20. Pierwsze okno Edytora parametrów podczas tworzenia rejestru przesuwającego

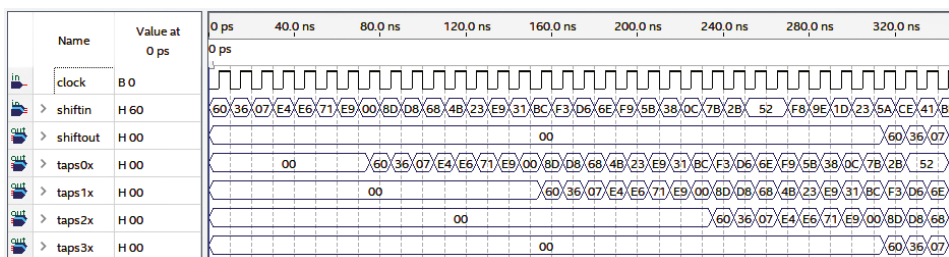
Drugie i trzecie okno Edytora parametrów definiuje typ używanego symulatora i wygenerowane pliki wyjściowe. Okna te w rzeczywistości pokrywają się z podobnymi oknami podczas tworzenia modułu pamięci typu RAM (rys. 3.9 i 3.10).

Wyniki syntezy i symulacji rozważanego przykładu rejestru przesuwającego przedstawiono, odpowiednio, na rysunkach 3.21 i 3.22.

Analiza rysunku 3.21 pokazuje, że megafunkcja **altshift_taps** firmy Intel (Altera) została użyta do utworzenia w bloku RAM 8-bitowego rejestru przesuwającego z czterema odczepami *taps0x*,...,*taps3x*. Więcej informacji na temat korzystania z megafunkcji **altshift_taps** można znaleźć w [16].



RYS. 3.21. Widok projektu *shift_reg* na poziomie RTL



RYS. 3.22. Wyniki symulacji projektu *shift_reg*

Rysunek 3.22 pokazuje, że odczep *taps0x[]* odpowiada przesunięciu magistrali *shiftin[]* o 8 bitów, odczep *taps1x[]* odpowiada 16 bitom, odczep *taps2x[]* odpowiada 24 bitom, a odczep *taps3x[]* odpowiada magistrali wyjściowej *shiftout[]*.

3.5. Wnioski

Wykorzystanie bloków IP za pomocą narzędzi: Katalogu IP (IP Catalog) oraz Edytora parametrów systemu Quartus pozwala na tworzenie gotowych projektów dla systemów wbudowanych (w tym modułów pamięci). Jednocześnie dostępna staje się większa różnorodność właściwości architektury projektowanego modułu pamięci niż przy opisie projektu w języku Verilog.

Bloki IP modułów pamięci są zbudowane z wykorzystaniem megafunkcji firmy Intel (Altera).

Katalog IP zawiera menu, które umożliwia tworzenie modułów dla jednoportowej pamięci typu RAM (RAM: 1-POR), dwuportowej pamięci typu RAM (RAM: 2-POR), jednoportowej pamięci typu ROM (ROM: 1-POR), dwuportowej pamięci typu ROM, pamięci typu FIFO i rejestru przesuwającego (Shift register (RAM-based)) w bloku pamięci wbudowanej FPGA.

Proces tworzenia modułu pamięci jest prowadzony przez MegaWizard Plug-In Manager, zwany także *Edytorem parametrów*.

Podczas tworzenia modułów pamięci, takich jak RAM lub ROM, edytor parametrów wyświetla na ekranie sekwencję 10 stron. Każda strona zawiera menu służące do definiowania parametrów i właściwości architektury utworzonego modułu pamięci. Nie wszystkie 10 stron pojawia się na ekranie, zależy to od wprowadzonych parametrów.

Po wpisaniu przez użytkownika niezbędnych parametrów, zdefiniowaniu wymaganych właściwości architektury i kliknięciu przycisku Finish, Edytor parametrów automatycznie generuje moduł pamięci w określonym języku (w naszym przypadku – w języku Verilog). Opis modułu można przeglądać w dowolnym edytorze tekstowym.

Moduły pamięci typu FIFO i rejestru przesuwającego są tworzone przy użyciu Katalogu IP i Edytora parametrów w bardzo podobny sposób jak pamięci typu RAM i ROM.

Część II

Projektowanie układów sterowania

Projektant systemów wbudowanych staje przed zadaniem zaprojektowania układów sterowania za każdym razem, gdy tworzy nowy projekt. Faktem jest, że jeśli poszczególne standardowe bloki funkcjonalne systemów wbudowanych mogą się powtarzać w różnych projektach, to układy sterujące zawsze wymagają projektowania od nowa. W systemie wbudowanym może być kilka układów sterowania. Złożone jednostki funkcjonalne, oddzielne części systemu wbudowanego, podsystemy systemu wbudowanego mogą mieć własne układy sterujące, jak również cały system wbudowany może mieć nadrzędny układ sterowania.

Jako układy sterowania często stosuje się *automaty skończone* (finite state machine – FSM), których projektowanie na FPGA zostało omówione w [10]. Układy sterujące można również tworzyć jako układy kombinacyjne. Przykład układu kombinacyjnego, działającego jako układ sterowania, zostanie podany później, przy projektowaniu procesora jednocyklowego. W niektórych przypadkach cały system wbudowany może składać się z jednego układu sterującego, na przykład system sterowania robotem, lub z kilku układów sterujących, na przykład system wbudowany samolotu, w którym istnieje kilka autonomicznych, a także współdziałających ze sobą systemów sterowania.

Wiele układów i systemów sterowania opiera się na zasadach *sterowania mikroprogramowego*. *Automaty mikroprogramowane* (AM), szeroko stosowane w systemach wbudowanych, są zbudowane na tych samych zasadach. Do opisu funkcjonowania AM są używane języki sterowania logicznego, jednym z nich jest język *sieci działań*. Języki sterowania logicznego były dalej rozwijane w *sieci działań automatów algorytmicznych* (algorithmic state machine – ASM), a automaty skończone – w *automaty skończone ze ścieżką przetwarzania danych* (FSM with datapath – FSM-D).

4. Projektowanie automatów mikroprogramowanych

Algorytmy sterowania logicznego są często opisywane przez *sieci działań* i implementowane jako *automaty mikroprogramowane* (AM). AM opiera się na *zasadach mikroprogramowania*, których pionierem był Wilkes. W tym rozdziale omówiono metodologię projektowania AM zgodnie z *siecią działań algorytmów* (SDA), a także implementację AM Mealy’ego, AM Moore’a i AM klasy C w układzie FPGA. Jako przykład aplikacji AM przedstawiono implementację sprzętową synchronicznych układów mnożących na FPGA.

4.1. Mikroprogramowanie i automaty mikroprogramowane

Obecnie architektury komputerów są zwykle podzielone na dwie duże klasy: *komputer ze złożonym zestawem instrukcji* (complex instruction set computer – CISC) i *komputer ze zredukowanym zestawem instrukcji* (reduced instruction set computer – RISC). Większość nowoczesnych mikroprocesorów jest projektowana zgodnie z architekturą RISC, ale historycznie pierwsze komputery zostały zbudowane zgodnie z architekturą CISC. Należą do nich znany IBM-360, wszystkie mikroprocesory INTEL x86, Motorola 68000 itp.

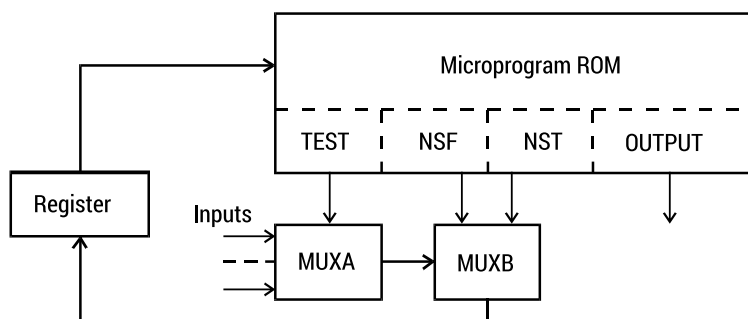
Komputery i mikroprocesory o architekturze CISC zawierają złożone instrukcje i różne metody adresowania z rejestrami bazowymi i indeksowymi. Złożone rozkazy były niezwykle trudne do realizacji w sprzęcie. Dlatego w 1951 roku Maurice Wilkes zaproponował zasady *mikroprogramowania* (microprogramming), zgodnie z którymi złożona instrukcja komputerowa jest wykonywana pod kontrolą *mikroprogramu* w kilku cyklach zegarowych. Oprogramowanie sprzętowe złożonych instrukcji jest zapisywane jako sekwencja *słów sterujących*, przechowywane w pamięci typu ROM i wywoływane w razie potrzeby.

Oprogramowanie składa się z pojedynczych *mikroinstrukcji* (microcodes). Z kolei mikroinstrukcja składa się z pojedynczych *mikrooperacji* (microoperations), które wykonują elementarne czynności. Każda mikroinstrukcja jest wykonywana w jednym cyklu zegara.

Kolejność wykonania mikroinstrukcji jest kontrolowana przez *warunki logiczne*. Warunki logiczne są sygnałami stanu i flagami *układu operacyjnego*.

W taki sposób mikroprogram jest realizowany przez system składający się z *układu operacyjnego* i *układu sterującego*. Układ operacyjny realizuje najprostsze mikrooperacje wykonywane w jednym cyklu zegara. Kilka mikrooperacji może być wykonywanych jednocześnie. Wówczas stanowią one *mikrorozkaz*. Układ sterowania jest *automatem skończonym*. Automat skończony, który implementuje mikroprogram, nazywany jest *automatem mikroprogramowanym* (AM – microprogrammed state machine). Wejścia AM to warunki logiczne, a wyjścia to sygnały sterujące, które sterują działaniem układu operacyjnego.

Tradycyjna implementacja sprzętowa sterowania mikroprogramowego została przedstawiona na rysunku 4.1.



RYS. 4.1. Tradycyjna implementacja sterowania mikroprogramowego

Struktura na rysunku 4.1 składa się z pamięci mikroprogramów typu ROM (Microprogram ROM), dwóch multiplexerów MUXA i MUXB oraz rejestru Register. Słowo sterujące zawiera cztery pola: TEST, NSF, NST i OUTPUT. Pole TEST zawiera kod warunku logicznego do sprawdzenia. Na podstawie tego kodu multiplexer MUXA wybiera jeden z sygnałów wejściowych AM, którego wartość należy sprawdzić. Wyjście MUXA jest wykorzystywane przez MUXB do wyboru adresu następnego mikrorozkazu (mikroinstrukcji) do wykonania z pól NSF lub NST. Pola NSF i NST zawierają adresy mikroinstrukcji, które powinny zostać wykonane w następnym cyklu w przypadku fałszywej (0) lub prawdziwej (1) wartości sprawdzanego sygnału wejściowego. Pole OUTPUT zawiera wartości sygnałów sterujących (wyjścia AM).

Z punktu widzenia teorii automatów skończonych, struktura na rysunku 4.1 jest automatem typu Moore'a. Stany automatu skończonego są mikroinstrukcjami. Kody stanów to adresy mikroinstrukcji w pamięci typu ROM, przechowywane w rejestrze Register. Następny stan automatu skończonego jest określany przez adres mikroinstrukcji (kod stanu) załadowany do rejestru Register. Układ kombinacyjny realizujący funkcje przejścia składa się z multiplexerów MUXA i MUXB. W strukturze automatu skończonego nie ma układu kombinacyjnego, który implementowałby funkcje wyjściowe, ponieważ wartości sygnałów wyjściowych są wyraźnie określone w polu OUTPUT słowa sterującego dla każdego stanu AM.

Możliwa jest również realizacja sterowania mikroprogramowego za pomocą automatu typu Mealy'ego. W tym przypadku pole OUTPUT jest reprezentowane przez dwa pola: OUTPUTF i OUTPUTT, tzn. wartości sygnałów wyjściowych generowanych odpowiednio przy przejściach false (0) i true (1). Ze względu na to, że taka implementacja wymaga znacznego zwiększenia długości słowa sterującego, w praktyce AM Mealy'ego nie są powszechnie stosowane.

Struktura sterowania mikroprogramowego na rysunku 4.1 ma następujące wady:

- duża długość słowa sterującego;
- w każdym cyklu zegara sprawdzana jest wartość tylko jednego warunku logicznego.

Następnie zaproponowano wiele różnych rozwiązań, w celu eliminacji wskazanych wad, ale nie udało się ich całkowicie pozbyć. W rezultacie nowoczesne systemy komputerowe są zwykle budowane w architekturze RISC.

Jednak sama idea sterowania mikroprogramowego okazała się bardzo skuteczna i została rozwinięta w różnych dziedzinach techniki, na przykład w robotyce, w sterownikach przemysłowych, a także w systemach wbudowanych, gdzie różnorodne układy sterujące, często nazywane *kontrolerami* (controllers), budowane są na podstawie sterowania mikroprogramowego.

4.2. Reprezentacja AM. Sieci działań

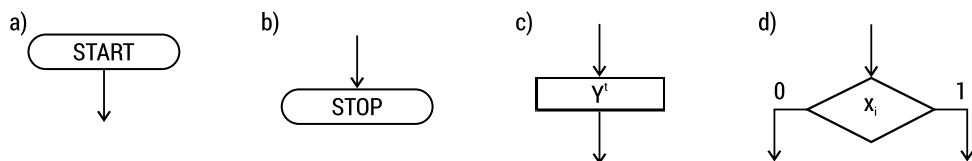
Algorytmy działania AM należą do klasy algorytmów sterowania logicznego. Istnieje wiele sposobów opisywania algorytmów sterowania logicznego, wśród których najbardziej rozpowszechnione są:

- *diagramy logiczne algorytmów*;
- *diagramy macierzowe algorytmów*;
- *sieci działań* lub *graficzne schematy algorytmów* (GSA).

Diagramy logiczne algorytmów odwołują się do językowych metod opisu algorytmów. Diagram macierzowy algorytmów (DMA) to macierz, której kolumny i wiersze odpowiadają mikrooperacjom (operatorom). W komórkach DMA zapisywane są warunki przejścia od jednego operatora do drugiego. Pusta komórka DMA oznacza, że nie ma przejścia między operatorami. Bezwarunkowy skok jest określany przez wpis 1 w odpowiedniej komórce DMA. Ponadto znanych jest wiele języków opisu algorytmów sterowania logicznego.

Spośród wszystkich metod opisu AM *sieci działań* lub *graficzne schematy algorytmów* (GSA) wyróżnia się prostotą i czytelnością. Język GSA został po raz pierwszy zaproponowany w pracy [3] jako graficzny odpowiednik schematów logicznych algorytmów. GSA to ukierunkowany skończony połączony graf, który zawiera jeden wierzchołek początkowy, jeden wierzchołek końcowy oraz pewną liczbę wierzchołków operacyjnych i wierzchołków warunkowych. Wierzchołek początkowy (rys. 4.2a)

ma tylko jedno wyjście i nie ma żadnych wejść, wierzchołek końcowy (rys. 4.2b), przeciwnie, ma jedno wejście i nie ma wyjść; wierzchołek operacyjny (rys. 4.2c) ma jedno wejście i jedno wyjście; wierzchołek warunkowy (rys. 4.2d) posiada jedno wejście i dwa wyjścia, które są oznaczone symbolami „0” i „1”.



RYŚ. 4.2. Wierzchołki GSA: a – początkowy; b – końcowy; c – operacyjny; d – warunkowy

W każdym wierzchołku operacyjnym zapisana jest mikroinstrukcja Y^t – podzbiór zbioru mikrooperacji $Y = \{y_1, \dots, y_N\}$, $Y^t \subseteq Y$. Dozwolone jest zapisywanie identycznych mikroinstrukcji w różnych wierzchołkach operacyjnych, jak również $Y^t = \emptyset$, gdzie $\emptyset$ jest zbiorem pustym. Wierzchołki operacyjne definiują zestaw mikrooperacji, które muszą być wykonywane jednocześnie w jednym cyklu czasu maszynowego. W każdym wierzchołku warunkowym jest zapisywany jeden z elementów zbioru warunków logicznych $X = \{x_1, \dots, x_l\}$, te same warunki logiczne można zapisać w różnych wierzchołkach warunkowych. Wierzchołki warunkowe to punkty rozgałęzienia algorytmu sterowania logicznego w zależności od wartości zapisanych w nich warunków logicznych (wystąpienie lub brak określonych zdarzeń). Wewnątrz wierzchołków początkowego i końcowego znajdują się odpowiednio słowa „START” i „STOP”. Wierzchołek początkowy definiuje punkt wejścia do algorytmu, a wierzchołek końcowy określa koniec wykonywania algorytmu. Czasami w algorytmach cyklicznych brakuje wierzchołka końcowego.

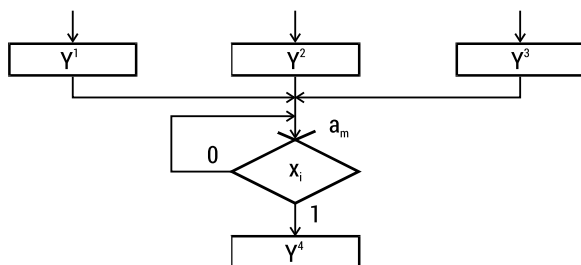
Zasady budowy GSA nie są sformalizowane, ale każdy GSA musi spełniać następujące warunki:

- wejścia i wyjścia wierzchołków są połączone za pomocą strzałek, zawsze skierowanych od wyjścia do wejścia;
- każde wyjście jest podłączone tylko do jednego wejścia;
- dowolny wierzchołek operacyjny lub wierzchołek warunkowy leży na co najmniej jednej ścieżce od wierzchołka początkowego do końcowego;
- przy opisie funkcjonowania AM, pętle w GSA muszą przechodzić przez wierzchołki operacyjne.

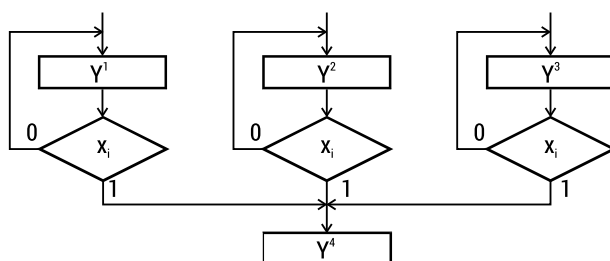
Wyjaśnijmy ostatni warunek dotyczący budowy GSA. Załóżmy, że mamy pewien fragment GSA pokazany na rysunku 4.3.

Na rysunku 4.3 etykieta a_m oznacza wejście wierzchołka warunkowego przy syntezie AM Mealy’ego. Z fragmentu na rysunku 4.3 wartości na wyjściach AM są nieznane, gdy automat jest w stanie a_m . Jednak w przypadku sprzętowej implementacji AM, wyjścia automatu powinny być zawsze określane przez wartości istotne: 0 lub 1. Dlatego podczas budowy GSA do syntezy AM zawsze konieczne jest jednoznaczne

zdefiniowanie mikrooperacji, które powinny być tworzone w stanach oczekiwania. Jeśli wychodzimy z założenia, że w stanie a_m powinna powstać wartość ostatnia, to fragment na rysunku 4.3 należy przepisać, jak pokazano na rysunku 4.4.



RYS. 4.3. Błędny fragment GSA odpowiadający stanowi oczekiwania



RYS. 4.4. Prawidłowy fragment GSA odpowiadający stanowi oczekiwania na rys. 4.3

4.3. Projektowanie automatów mikroprogramowanych zgodnie z siecią działań (GSA)

4.3.1. Metodyka projektowania AM zgodnie z GSA

Niech algorytm sterowania logicznego zostanie przedstawiony w postaci GSA. Zadanie polega na zbudowaniu takiego AM, który implementuje dany algorytm sterowania logiką. Problem w tym, że stany AM w żaden sposób nie są wskazywane w GSA. Projektant musi wprowadzić je samodzielnie. W [2] proponuje się wprowadzenie do GSA *etykiet* do oznaczania stanów AM, które następnie będą identyfikowane ze stanami wewnętrznymi AM. Istota tego problemu polega na tym, że GSA i AM odnoszą się do obiektów o różnym charakterze: GSA jest językiem graficznym do opisu algorytmów sterowania logicznego lub mikroprogramów, a AM jest automatem skończonym definiowanym przez stany wewnętrzne i przejścia między nimi.

W ogólnym przypadku metodologię projektowania AM dla GSA można przedstawić w postaci następującego algorytmu.

Algorytm 4.1. Metodyka projektowania AM zgodnie z GSA

1. Oznaczenie zgodnie z [2] GSA dla syntezy AM określonego typu: Mealy'ego lub Moore'a.
2. Identyfikacja etykiet GSA ze stanami wewnętrznymi AM.
3. Budowanie tabeli przejść AM.

Podczas budowania AM zgodnie z GSA mogą pojawić się problemy nawet na etapie tworzenia GSA, ponieważ etap ten jest wykonywany ręcznie przez człowieka i często też, nie przez projektanta AM. Zauważmy, że ten sam algorytm sterowania logicznego można opisać na różne sposoby, tzn. różnymi GSA. Niektóre z tych GSA prowadzą do skutecznej implementacji AM na układach FPGA (pod względem kosztów realizacji i szybkości), podczas gdy inne – nie.

Ponadto różne sposoby oznaczania GSA będą definiować różne AM, które implementują ten sam algorytm sterowania logicznego. Niektóre sposoby oznaczania GSA prowadzą do skutecznej implementacji AM na układach FPGA, podczas gdy inne – nie. Zwróćmy uwagę, że nie wszystkie oznaczenia GSA są prawidłowe, tzn. niektóre oznaczenia nie pozwalają na skonstruowanie AM, który prawidłowo implementuje zadany algorytm sterowania.

W [2] zaproponowano metody oznaczania GSA, które pozwalają na wprowadzenie minimalnej liczby znaczników (etykiet) GSA do syntezy AM określonego typu: Mealy'ego lub Moore'a. Jednak metody oznaczania GSA w [2] nie zakładają budowy wspólnych modeli AM. Ponadto GSA można oznaczyć w inny sposób niż [2]. W rezultacie przy użyciu tego samego GSA można zbudować kilka różnych AM o różnej wydajności i koszcie implementacji na FPGA.

Podczas identyfikowania etykiet GSA ze stanami wewnętrznymi może pojawić się pytanie: do jakiego typu AM, Mealy'ego czy Moore'a należy przypisać określoną etykietę GSA?

Pytania pojawiają się również podczas budowy tablicy przejść AM Mealy'ego. Gdzie powinno się kończyć przejście: przy spotkaniu pierwszej etykiety na ścieżce czy dopiero po minięciu wierzchołka operacyjnego? W pierwszym przypadku na przejściu jest tworzony zerowy wektor wyjściowy, ale czy zerowy wektor wyjściowy jest dopuszczalny w algorytmie sterowania? W drugim przypadku na ścieżce GSA mogą wystąpić wierzchołki warunkowe, w których zapisane są napotkane wcześniej warunki logiczne. Jak przejść przez takie wierzchołki warunkowe? Czy do tego czasu wartość warunku logicznego może się zmienić?

Spróbujmy zrozumieć niektóre z tych problemów na przykładzie syntezy konkretnego automatu mikroprogramowanego.

4.3.2. Oznaczanie GSA

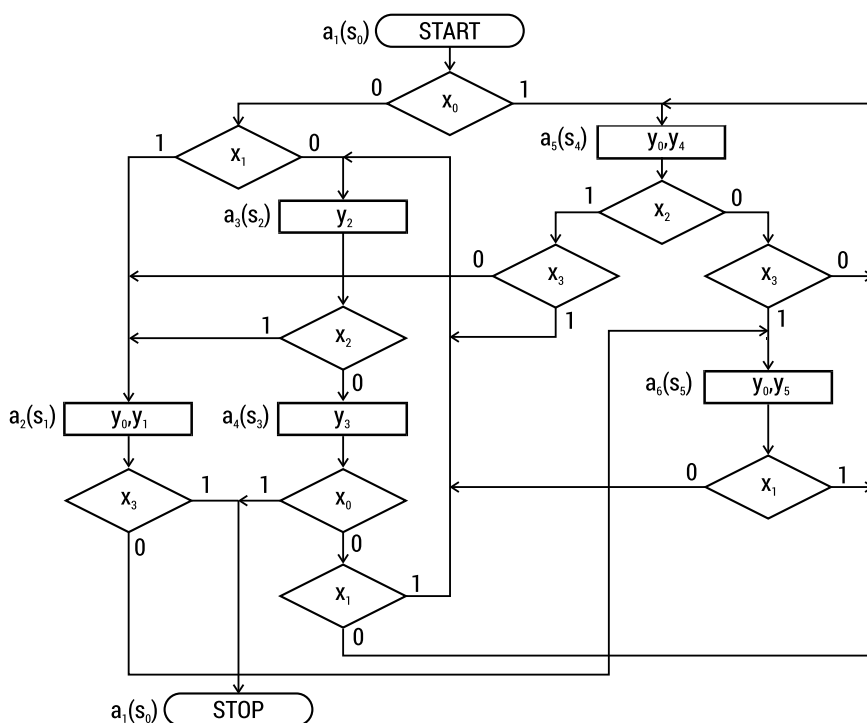
Najłatwiej jest oznaczyć GSA do syntezy AM Moore'a. Reguły oznaczania GSA dla syntezy AM Moore'a można przedstawić w postaci następującego algorytmu.

Algorytm 4.2. Oznaczanie GSA dla syntezy AM Moore'a

1. Etykieta a_1 , odpowiadająca początkowemu stanowi AM, oznacza początkowy i końcowy wierzchołek GSA.
2. Etykiety $a_2, \dots, a_M$ kolejno oznaczają wszystkie wierzchołki operacyjne GSA, przy tym każdy wierzchołek operacyjny jest oznaczany tylko jeden raz.
3. Koniec.

Dwa wierzchołki w GSA są oznaczane jednocześnie etykietą a_i : początkowy i końcowy. Oznacza to, że po wykonaniu algorytmu sterowania logicznego AM przechodzi do stanu początkowego a_1 i jest ponownie gotowy do wykonania algorytmu. W kroku 2 algorytmu 4.2 każdy wierzchołek operacyjny jest oznaczany etykietą tylko jeden raz. Dlatego liczba M etykiet GSA (stanów wewnętrznych AM) będzie równa $Q + 1$, gdzie Q jest liczbą wierzchołków operacyjnych GSA.

Przykład oznaczenia GSA do syntezy AM Moore'a pokazano na rysunku 4.5. Tutaj etykieta a_1 oznacza wierzchołek początkowy i końcowy. Następnie etykiety $a_2, \dots, a_6$ oznaczają wierzchołki operacyjne GSA. W nawiasach obok każdej etykiety na rysunku 4.5 zapisano stan wewnętrzny AM.

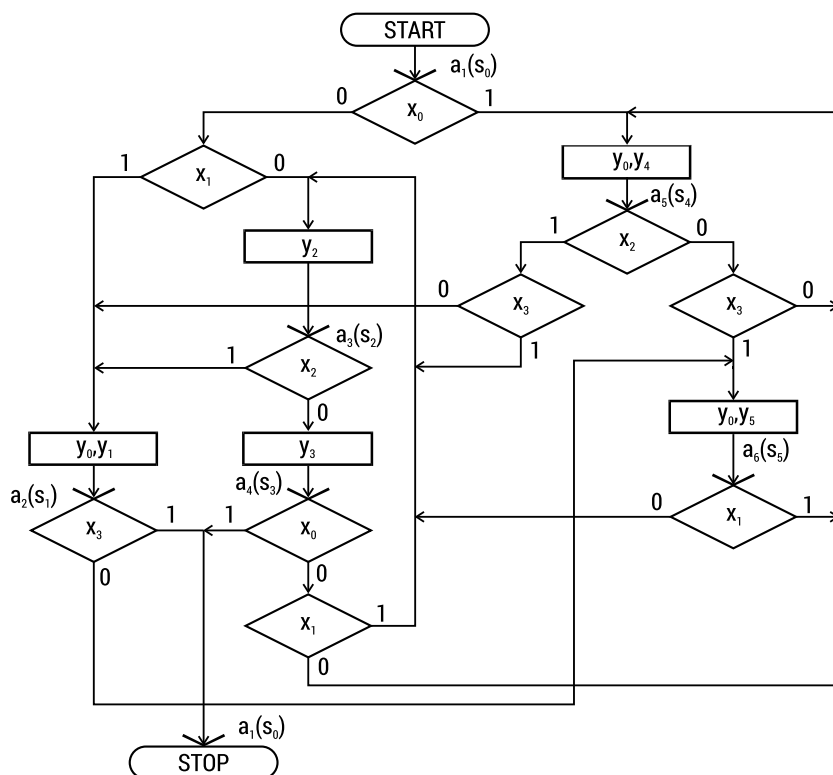


RYS. 4.5. Oznaczanie GSA dla syntezy AM Moore'a

Nieco trudniej jest oznaczyć GSA do syntezy AM Mealy'ego. Pomimo faktu, że liczba wprowadzanych etykiet jest powiązana z liczbą wierzchołków operacyjnych

Algorytm 4.3. Oznaczanie GSA do syntezy AM Mealy'ego

1. Etykieta a_j , odpowiadająca początkowemu stanowi AM, oznacza wejście wierzchołka bezpośrednio po początkowym wierzchołku, jak również wejście końcowego wierzchołka.
2. Etykiety $a_2, \dots, a_M$ oznaczają wejścia wierzchołków bezpośrednio następujących po wierzchołkach operacyjnych, przy tym wejścia wierzchołków zaznaczane są tylko raz.
3. Koniec.



RYS. 4.6. Oznaczenie GSA do syntezy AM Mealy'ego

115

pojawia się wierzchołek warunkowy oczekiwania, w którym sprawdzany jest sygnał zezwalający na działanie algorytmu. Informacja zwrotna z takiego warunkowego wierzchołka GSA nie przechodzi przez żaden z wierzchołków operacyjnych GSA. Jest to jedyny przypadek, w którym pętla sprzężenia zwrotnego w GSA nie przechodzi przez wierzchołek operacyjny. We wszystkich innych przypadkach, pętle sprzężenia zwrotnego muszą przechodzić przez wierzchołki operacyjne GSA.

Przykład oznaczania GSA do syntezy AM Mealy'ego pokazano na rysunku 4.6. Tutaj a_1 oznacza wejście wierzchołka warunkowego następującego bezpośrednio po wierzchołku początkowym GSA, jak również wejście ostatniego wierzchołka. Następnie etykiety $a_2, \dots, a_6$ oznaczają wejścia wierzchołków bezpośrednio następujących po wierzchołkach operacyjnych. W naszym przykładzie liczba etykiet GSA (stanów AM) do syntezy automatów Mealy'ego i Moore'a jest taka sama, ale często zdarza się, że liczba etykiet do syntezy AM Mealy'ego jest mniejsza niż liczba etykiet do syntezy AM Moore'a.

4.3.3. Tworzenie tablicy przejść

Niech A będzie zbiorem etykiet GSA, X będzie zbiorem warunków logicznych GSA, a Y będzie zbiorem mikrooperacji GSA. Aby utworzyć tablicę przejść AM Mealy'ego, etykiety GSA są rozpatrywane sekwencyjnie. Niech pewna kolejna etykieta $a_m, a_m \in A$, będzie zdefiniowana jako *stan obecny* automatu skończonego (present state), w którym rozpoczyna się przejście. Następnie znajduje się ścieżkę od wierzchołka GSA, którego wejście jest oznaczone etykietą a_m , w kierunku orientacji strzałek do pewnej etykiety $a_s, a_s \in A$, która jest definiowana jako *stan następny* (next state) lub *stan przejścia*. W trakcie przejścia przez wierzchołki warunkowe GSA ich zawartość jest wypisywana. Warunki logiczne zapisane w takich wierzchołkach warunkowych określają koniunkcję zmiennych wejściowych $X(a_m, a_s)$, której wartość równa jeden powoduje przejście ze stanu a_m do stanu a_s , a podzbiór mikrooperacji zapisanych w wierzchołku operacyjnym określa zbiór zmiennych wyjściowych $Y(a_m, a_s)$, które w tym przejściu przyjmują wartość 1. Zmienna $x_i, x_i \in X$, wchodzi w koniunkcję $X(a_m, a_s)$ w postaci prostej, jeśli warunkowy wierzchołek GSA, w którym jest zapisany, przechodzi wzdłuż strzałki oznaczonej jedynką (1), zmienna $x_i, x_i \in X$ wchodzi w koniunkcję $X(a_m, a_s)$ w postaci zanegowanej, jeśli strzałka jest oznaczona zerem (0).

Przejścia AM Moore'a są konstruowane w taki sam sposób jak dla automatu Mealy'ego, z tą różnicą, że zbiór $Y(a_m)$ zmiennych wyjściowych przyjmujących wartość 1 w stanie a_m jest określony przez zawartość wierzchołka operacyjnego, który jest oznaczony etykietą a_m . Dla wierzchołka początkowego $Y(a_1) = \emptyset$, gdzie $\emptyset$ jest zbiorem pustym.

Zestaw przejść AM jest zapisywany jako *tablica przejść* (transition table) z czterema kolumnami. Każdy wiersz tablicy przejść odpowiada jednemu przejściu AM. Kolumny tablicy przejść są oznaczone jako $a_m, a_s, X(a_m, a_s)$ i $Y(a_m, a_s)$. W przypadku automatu Mealy'ego w kolumnie a_m zapisywany jest stan obecny automatu, w którym rozpoczyna

się przejście; w kolumnie a_s – stan następny, w którym przejście się kończy; w kolumnie $X(a_m, a_s)$ – koniunkcja zmiennych wejściowych, której wartość równa 1 powoduje dane przejście; w kolumnie $Y(a_m, a_s)$ – wektor zmiennych wyjściowych, które przyjmują jedynkowe wartości w danym przejściu. W przypadku automatu Moore’a ostatnia kolumna jest oznaczona jako $Y(a_m)$; zawiera wektor zmiennych wyjściowych, które przyjmują wartości 1 w stanie a_m . Na przykład tablicę przejść automatu Mealy’ego, zbudowaną zgodnie z GSA z rysunku 4.6, podano w tabeli 4.1 oraz tablicę przejść automatu Moore’a, zbudowaną zgodnie z GSA z rysunku 4.5, pokazano w tabeli 4.2.

TAB. 4.1. Tablica przejść AM Mealy’ego, zbudowana zgodnie z GSA z rys. 4.6

a_m	$X(a_m, a_s)$	a_s	$Y(a_m, a_s)$
s_0	$\overline{x_0}$	s_4	$y_0 y_4$
	$\overline{x_0} x_1$	s_1	$y_0 y_1$
	$x_0 \overline{x_1}$	s_2	y_2
s_1	x_3	s_0	–
	x_3	s_5	$y_0 y_5$
s_2	x_2	s_1	$y_0 y_1$
	x_2	s_3	y_3
s_3	$\overline{x_0}$	s_0	–
	$\overline{x_0} x_1$	s_2	y_2
	$x_0 \overline{x_1}$	s_4	$y_0 y_4$
s_4	$x_2 x_3$	s_2	y_2
	$x_2 x_3$	s_1	$y_0 y_1$
	$\overline{x_2} x_3$	s_5	$y_0 y_5$
	$x_2 \overline{x_3}$	s_4	$y_0 y_4$
s_5	x_1	s_4	$y_0 y_4$
	x_1	s_2	y_2

TAB. 4.2. Tablica przejść AM Moore’a, zbudowana zgodnie z GSA z rys. 4.5

a_m	$X(a_m, a_s)$	a_s	$Y(a_m)$
s_0	$\overline{x_0}$	s_4	–
	$\overline{x_0} x_1$	s_1	
	$x_0 \overline{x_1}$	s_2	
s_1	x_3	s_0	$y_0 y_1$
	x_3	s_5	
s_2	x_2	s_1	y_2
	x_2	s_3	
s_3	$\overline{x_0}$	s_0	y_3
	$\overline{x_0} x_1$	s_2	
	$x_0 \overline{x_1}$	s_4	

a_m	$X(a_m, a_s)$	a_s	$Y(a_m)$
s_4	x_2x_3	s_2	y_0y_4
	$\overline{x_2}x_3$	s_1	
	$x_2\overline{x_3}$	s_5	
	$\overline{x_2}\overline{x_3}$	s_4	
s_5	x_1	s_4	y_0y_5
	$\overline{x_1}$	s_2	

Zbiór przejść z tego samego stanu nazywany jest *grupą przejść*. W tablicy przejść, grupy przejść są oddzielone poziomymi liniami. Niech zgodnie z [2] wykonano oznaczenie GSA. W tablicy przejść poprawnie utworzonej, zgodnie z GSA, dysjunkcja wszystkich koniunkcji $X(a_m, a_s)$, które inicjują przejścia każdej grupy przejść jest równa jeden, tzn. dla każdego stanu a_m , $a_m \in A$, powinno być prawdziwe:

$$\bigvee_{a_s \in A(a_m)} X(a_m, a_s) = 1,$$

gdzie $\vee$ jest znakiem dysjunkcji; $A(a_m)$ to zbiór stanów, do których możliwe jest przejście ze stanu a_m .

Zwracamy również uwagę, że przy rozpatrywanym podejściu do tworzenia AM z GSA budowane są automaty skończone w pełni określone.

4.3.4. Dodatkowe oznaczenia GSA i pseudorównoważne AM

Wiele algorytmów syntezy AM używa etykiet GSA, które różnią się od tych omówionych powyżej. Dlatego zbiór etykiet GSA $A = \{a_1, \dots, a_M\}$, wprowadzony zgodnie z zasadami [2] rozważanymi powyżej, będziemy nazywać *głównymi etykietami GSA*, a pozostałe – *dotatkowymi etykietami GSA*.

Niech $z_1, \dots, z_k$ będzie pewnym ciągiem zbiorów wartości zmiennych wejściowych, a $w_1, \dots, w_k$ – odpowiednim ciągiem zbiorów wartości zmiennych wyjściowych, generowanym przez automat S_1 . Dwa AM S_1 i S_2 są *równoważne*, jeżeli dla dowolnej sekwencji zbiorów wartości zmiennych wejściowych generują identyczne sekwencje zbiorów wartości zmiennych wyjściowych. Automat S_2 nazywany jest *pseudorównoważnym* do automatu S_1 , jeśli dla jakiegoś ciągu $z_1, \dots, z_k$ generuje ciąg zbiorów wartości zmiennych wyjściowych, w którym sekwencja zbiorów wyjściowych odpowiada ciągowi $w_1, \dots, w_k$, ale może zawierać dodatkowe zbiory zerowe.

Należy zauważyć, że w wielu praktycznych zastosowaniach dodatkowe zbiory zerowe w sekwencji $w_1, \dots, w_k$ są całkiem dopuszczalne. Dodatkowy zestaw zerowy AM odpowiada ścieżce w GSA, która nie przechodzi przez wierzchołek operacyjny (dla automatu Mealy'ego) lub nie kończy się na wierzchołku operacyjnym (dla automatu Moore'a). Ta ścieżka kończy się dodatkową etykietą GSA. W niektórych przypadkach wprowadzenie dodatkowych etykiet pozwala uprościć syntezę AM.

4.4. Implementacja AM na FPGA

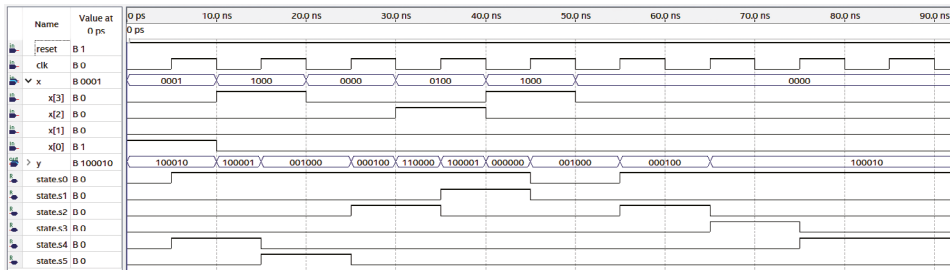
Po oznaczeniu GSA i zbudowaniu tablicy przejść, implementacja AM na FPGA nie różni się w żaden sposób od implementacji na FPGA automatów skończonych.

4.4.1. Implementacja AM Mealy'ego

Kod AM Mealy'ego w języku Verilog, w pełnej zgodności z tablicą przejść zawartą w tabeli 4.1, przedstawia się następująco:

```
module AM_Mealy(  
  input clk, reset,           // sygnały sterujące  
  input [3:0] x,             // wektor wejściowy  
  output reg [0:5] y;        // wektor wyjściowy  
  
  reg [2:0] state, next;      // zmienne stanu  
  
  localparam [2:0]          // definicja stanów  
  s0=3'd0,s1=3'd1,s2=3'd2,s3=3'd3,s4=3'd4,s5=3'd5;  
  
  always @(posedge clk, negedge reset) // opis pamięci automatu  
    if(~reset) state <= s0;  
    else       state <= next;  
  
  always @(*) // opis działania automatu  
  case(state)  
    s0: if(x[0]) begin next=s4; y=6'b100010; end  
        else if(~x[0] && x[1]) begin next=s1; y=6'b110000; end  
        else begin next=s2; y=6'b001000; end  
    s1: if(x[3]) begin next=s0; y=6'b000000; end  
        else begin next=s5; y=6'b100001; end  
    s2: if(x[2]) begin next=s1; y=6'b110000; end  
        else begin next=s3; y=6'b000100; end  
    s3: if(x[0]) begin next=s0; y=6'b000000; end  
        else if(~x[0] && x[1]) begin next=s2; y=6'b001000; end  
        else begin next=s4; y=6'b100010; end  
    s4: if(x[2] && x[3]) begin next=s1; y=6'b110000; end  
        else if(x[2] && ~x[3]) begin next=s2; y=6'b001000; end  
        else if(~x[2] && x[3]) begin next=s5; y=6'b100001; end  
        else begin next=s4; y=6'b100010; end  
    s5: if(x[1]) begin next=s4; y=6'b100010; end  
        else begin next=s2; y=6'b001000; end  
  
  endcase  
endmodule
```

Wykorzystano tutaj styl opisu automatu skończonego z dwoma procesami, w którym opis funkcji przejścia i wyjścia jest łączony w jeden proces [10]. Wyniki syntezy i symulacji AM Mealy'ego przedstawiono, odpowiednio, na rysunkach 4.7 i 4.8.



RYS. 4.8. Wyniki symulacji projektu *AM_Mealy*

4.4.2. Implementacja AM Moore'a

Kod AM Moore'a w języku Verilog, w pełnej zgodności z tablicą przejść zawartą w tabeli 4.2, przedstawia się następująco:

```

module AM_Moore(
input clk, reset,           // sygnały sterujące
input [3:0] x,              // wektor wejściowy
output reg [0:5] y);        // wektor wyjściowy

reg [2:0] state, next;      // zmienne stanu

localparam [2:0]           // definicja stanów
s0=3'd0,s1=3'd1,s2=3'd2,s3=3'd3,s4=3'd4,s5=3'd5;

always @(posedge clk, negedge reset) // opis pamięci
    if(~reset) state <= s0;
    else state <= next;

always @(*)                // opis funkcji przejść
    case(state)
        s0: if(x[0]) next=s4;
             else if(~x[0] && x[1]) next=s1;
             else next=s2;
        s1: if(x[3]) next=s0;
    
```

```

        else                next=s5;
s2:   if(x[2])              next=s1;
        else                next=s3;
s3:   if(x[0])              next=s0;
        else if(~x[0] && x[1]) next=s2;
        else                next=s4;
s4:   if(x[2] && x[3])       next=s1;
        else if(x[2] && ~x[3]) next=s2;
        else if(~x[2] && x[3]) next=s5;
        else                next=s4;
s5:   if(x[1])              next=s4;
        else                next=s2;
    endcase

```

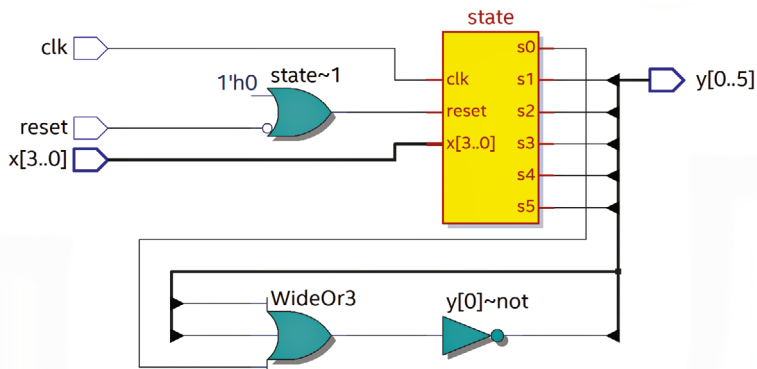
```

always @(*)                // opis funkcji wyjść
    case(state)
s0:  y=6'b000000;
s1:  y=6'b110000;
s2:  y=6'b001000;
s3:  y=6'b000100;
s4:  y=6'b100010;
s5:  y=6'b100001;
    endcase

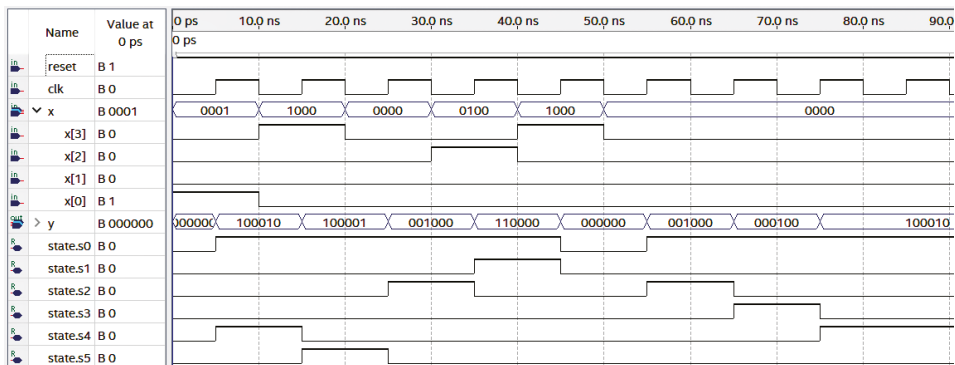
```

endmodule

Tutaj został wykorzystany tradycyjny styl opisu automatów skończonych z trzema procesami [10]. Wyniki syntezy i symulacji AM Moore'a przedstawiono, odpowiednio, na rysunkach 4.9 i 4.10.



RYS. 4.9. Wyniki syntezy projektu AM_Moore na poziomie RTL



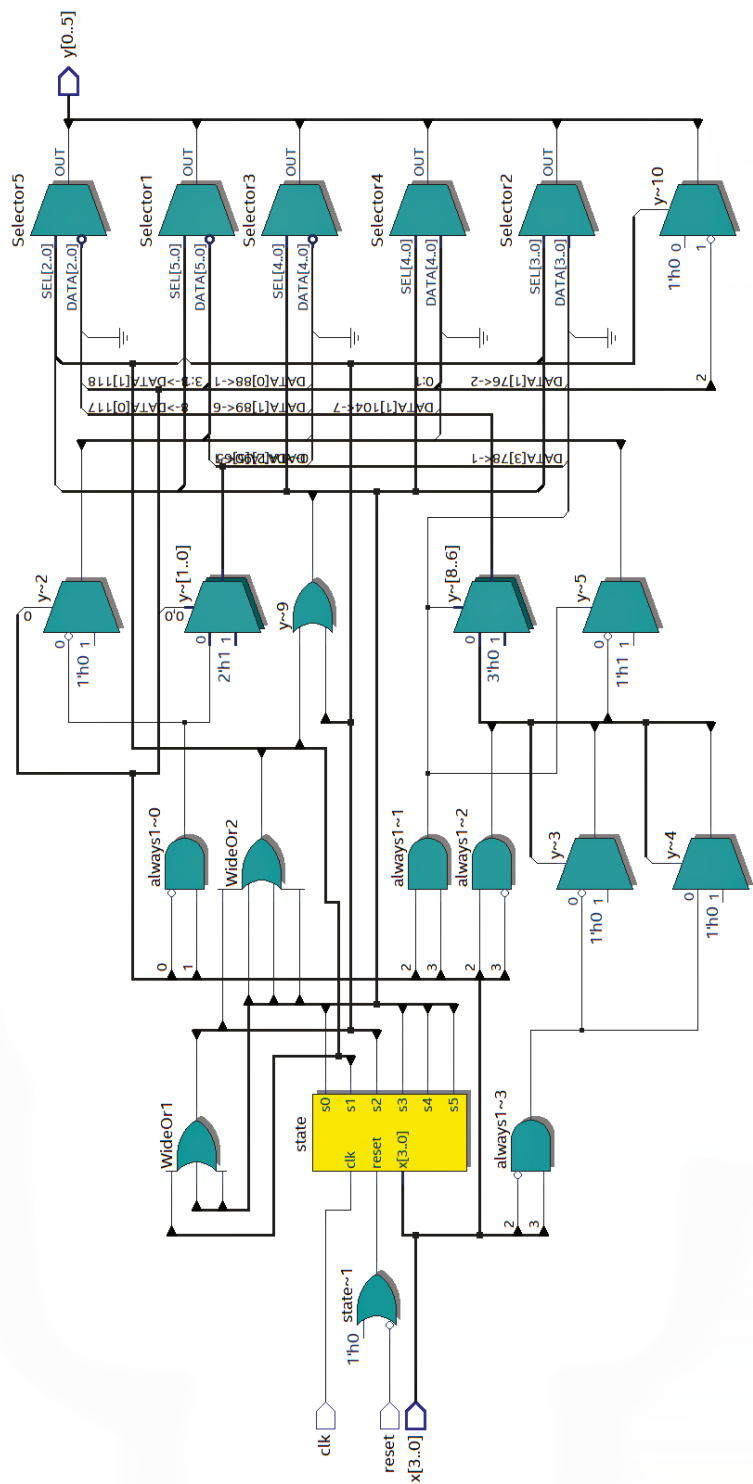
RYS. 4.10. Wyniki symulacji projektu *AM_Moore*

4.4.3. Analiza funkcjonowania AM Mealy'ego i AM Moore'a

Porównajmy wyniki symulacji funkcjonalnej AM Mealy'ego i Moore'a pokazane na rysunkach 4.8 i 4.10. Wyniki tego porównania przedstawiono w tabeli 4.3, gdzie *Time* – czas symulacji; x_i – wartość warunku logicznego; *s* – stan AM; *y* – wartość wektora wyjściowego wygenerowanego przez AM.

TAB. 4.3. Porównanie wyników symulacji funkcjonalnej AM Mealy'ego i Moore'a

Time (ns)	x_i	Mealy		Moore	
		<i>s</i>	<i>y</i>	<i>s</i>	<i>y</i>
0–5	$x_0 = 1$	s_0	100010	s_0	000000
5–10	$x_0 = 1$	s_4	100010	s_4	100010
10–15	$x_3 = 1$	s_4	100001	s_4	100010
15–20	$x_3 = 1$	s_5	001000	s_5	100001
20–25	0	s_5	001000	s_5	100001
25–30	0	s_2	000100	s_2	001000
30–35	$x_2 = 1$	s_2	110000	s_2	001000
35–40	$x_2 = 1$	s_1	100001	s_1	110000
40–45	$x_3 = 1$	s_1	000000	s_1	110000
45–50	$x_3 = 1$	s_0	001000	s_0	000000
50–55	0	s_0	001000	s_0	000000
55–60	0	s_2	000100	s_2	001000
60–65	0	s_2	000100	s_2	001000
65–70	0	s_3	100010	s_3	000100
70–75	0	s_3	100010	s_3	000100
75–80	0	s_4	100010	s_4	100010



RYS. 4.7. Wyniki syntezy projektu AM_Mealy na poziomie RTL

Time (ns)	x_i	Mealy		Moore	
		s	y	s	y
80–85	0	s_4	100010	s_4	100010
85–90	0	s_4	100010	s_4	100010

Analiza tabeli 4.3 pokazuje, że AM Mealy’ego i AM Moore’a działają prawie tak samo, ale AM Moore’a tworzy sygnały wyjściowe z opóźnieniem 5 ns. Ponadto, w chwilach czasu 15, 25, 35, 45, 55 i 65 ns, na wyjściach AM Mealy’ego pojawiają się wektory wyjściowe, których nie ma w sekwencji wektorów wyjściowych AM Moore’a. To ostatnie wyjaśnia fakt, że wyjścia AM Mealy’ego zależą nie tylko od stanu wewnętrznego, ale także od wartości sygnałów wejściowych. Dlatego, gdy zmieniają się wartości sygnałów wejściowych, również zmienia się wartość wektora wyjściowego.

Wniosek. Funkcjonowanie AM Mealy’ego i AM Moore’a podczas implementacji tego samego algorytmu sterowania logicznego jest różne. Okoliczność tę należy wziąć pod uwagę przy wyborze typu realizowanego AM.

4.4.4. Implementacja AM Moore’a klasy C

Ponieważ, po skonstruowaniu tablicy przejść, dowolny AM można uznać za zwykły automat skończony, wszystkie metody syntezy automatów skończonych można zastosować do AM. Zastosujemy do naszego AM metodę syntezy automatu klasy C, rozważaną w [10]. Macierz W do kodowania stanów wewnętrznych AM klasy C jest podana w tabeli 4.4.

TAB. 4.4. Macierz W do kodowania stanów wewnętrznych AM klasy C

a_i	$g_0g_1g_2g_3g_4g_5$
s_0	0 0 0 0 0
s_1	1 1 0 0 0
s_2	0 0 1 0 0
s_3	0 0 0 1 0
s_4	1 0 0 0 1
s_5	1 0 0 0 1

Opis AM klasy C w języku Verilog jest następujący:

```

module AM_Moore_C (
input clk, reset,           // sygnały sterujące
input [3:0] x,              // wektor wejściowy
output [0:5] y);           // wektor wyjściowy

```

```

reg [5:0] state, next;                // zmienne stanu

localparam [5:0]                      // definicja stanów
    s0=6'b000000,
    s1=6'b110000,
    s2=6'b001000,
    s3=6'b000100,
    s4=6'b100010,
    s5=6'b100001;

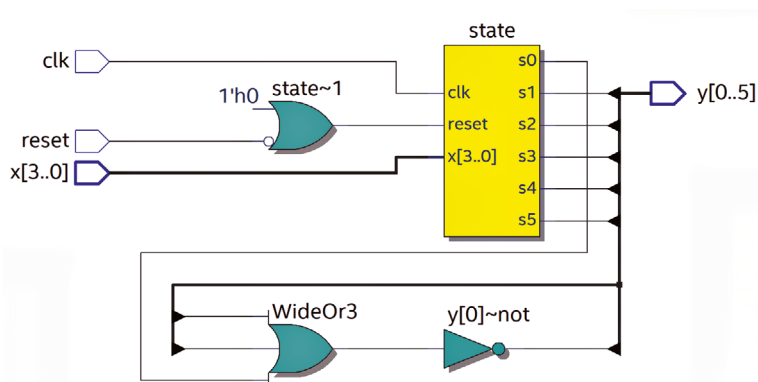
always @(posedge clk, negedge reset) // opis pamięci automatu
    if(~reset) state <= s0;
    else      state <= next;

always @(*)                          // opis funkcji przejść
    case(state)
        s0: if(x[0])                next=s4;
              else if(~x[0] && x[1]) next=s1;
              else                   next=s2;
        s1: if(x[3])                next=s0;
              else                   next=s5;
        s2: if(x[2])                next=s1;
              else                   next=s3;
        s3: if(x[0])                next=s0;
              else if(~x[0] && x[1]) next=s2;
              else                   next=s4;
        s4: if(x[2] && x[3])          next=s1;
              else if(x[2] && ~x[3]) next=s2;
              else if(~x[2] && x[3]) next=s5;
              else                   next=s4;
        s5: if(x[1])                next=s4;
              else                   next=s2;
        default                     next=s0;
    endcase

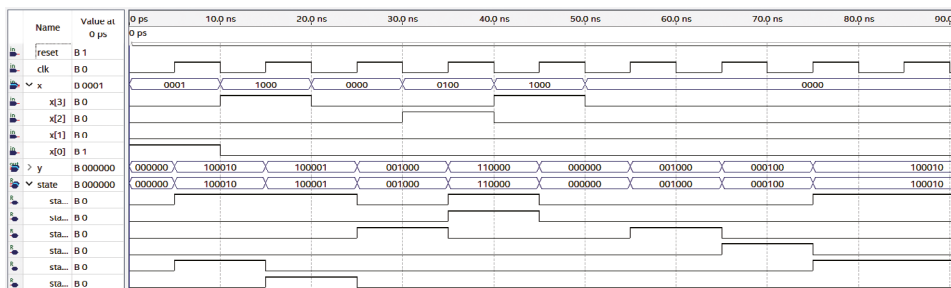
assign y=state;                       // opis funkcji wyjść
endmodule

```

Wyniki syntezy i symulacji AM klasy C przedstawiono, odpowiednio, na rysunkach 4.11 i 4.12.



RYS. 4.11. Wyniki syntezy projektu *AM_Moore_C* na poziomie RTL



RYS. 4.12. Wyniki symulacji projektu *AM_Moore_C*

Komentarz. W przypadku syntezy AM klasy C w systemie Quartus, opcja syntezy **State Machine Processing** musi być ustawiona na **User-Encoded**.

Analiza rysunku 4.12 pokazuje, że AM klasy C działa dokładnie tak, jak AM Moore'a.

Zachęcamy czytelnika do samodzielnej syntezy zgodnie z [10] AM klas D, E i F realizujących algorytm sterowania logicznego z naszego przykładu.

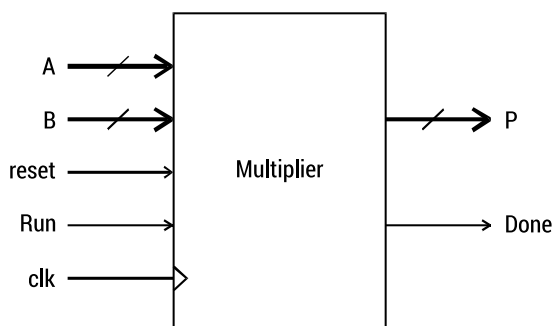
4.5. Wykorzystanie AM przy realizacji synchronicznych układów mnożących

W pracy [10] przedstawiono cztery algorytmy mnożenia za pomocą przesunięcia i dodawania iloczynów cząstkowych: algorytmy *a*, *b*, *c* i *d*. W [10] algorytmy te zostały opisane w języku Verilog przy użyciu operatorów proceduralnych, aby pokazać, że algorytmy można opisać w języku Verilog. Jednak ta implementacja układów mnożących jest podobna do programowej implementacji algorytmu mnożenia i wymaga dużej ilości zasobów logicznych FPGA.

W tej sekcji przedstawimy układ mnożący w postaci układu operacyjnego i układu sterującego. Opiszemy zachowanie układu sterującego w języku GSA i zaimplementujemy go jako automat skończony. Taka implementacja nazywana jest *sprzętową implementacją algorytmu mnożenia*.

4.5.1. Implementacja układu mnożącego za pomocą algorytmu a

Sprzętową implementacją układu mnożącego za pomocą algorytmu a jest układ synchroniczny (rys. 4.13), do którego wejść trafiają wartości mnożonych słów: mnożnej A i mnożnika B , a na wyjściu jest wartość iloczynu P . Niech szerokość słów A i B będzie taka sama i równa N bitów, wtedy iloczyn P będzie miał szerokość $2N$ bitów. Układ jest sterowany sygnałem zegara clk i sygnałem resetowania $reset$. Po ustawieniu wartości mnożonych słów na wejściach A i B , rozpoczyna się proces mnożenia przez ustawienie sygnału Run na jeden. Koniec procesu mnożenia jest wskazywany przez wartość równą 1 na wyjściu $Done$, od tej chwili wartość iloczynu można odczytać z wyjścia P .



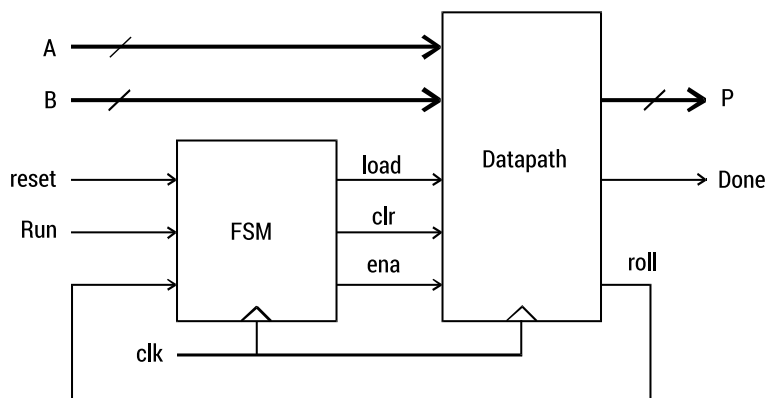
RYŚ. 4.13. Ogólna struktura synchronicznego układu mnożącego

Rysunek 4.14 przedstawia schemat blokowy synchronicznego układu mnożącego w postaci *układu operacyjnego* (datapath) i *układu sterującego* (controller), który jest zaimplementowany w postaci automatu skończonego FSM. Wartości mnożonych słów A i B są podawane na wejście układu operacyjnego. Iloczyn P i sygnał $Done$ są tworzone na wyjściach układu operacyjnego. Ponadto urządzenie operacyjne generuje sygnał $roll$, który jest wyjściem licznika modulo i wskazuje koniec procesu mnożenia.

Wejścia FSM to zewnętrzne sygnały $reset$ i Run oraz wewnętrzny sygnał $roll$. Automat skończony generuje następujące sygnały sterujące:

- clr – resetowanie rejestrów układu operacyjnego;
- $load$ – ładowanie wartości słów A i B do rejestrów układu operacyjnego;
- ena – zezwolenie operacji przesuwania rejestrów przesuwających.

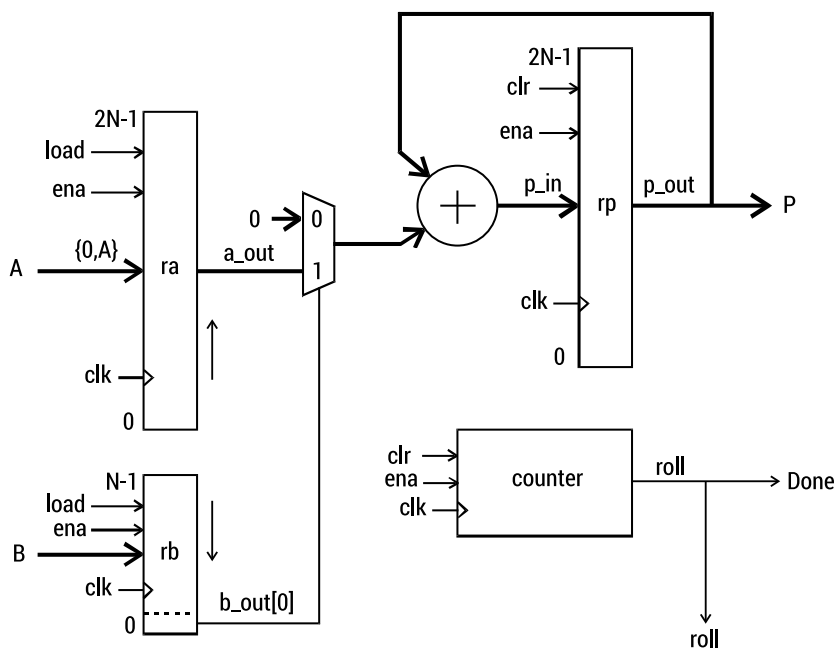
Układy operacyjny i sterujący są synchronizowane tym samym sygnałem zegarowym clk .



RYS. 4.14. Schemat blokowy synchronicznego układu mnożącego w postaci układu operacyjnego i układu sterującego

4.5.2. Projektowanie układu operacyjnego

Schemat układu operacyjnego realizującego algorytm mnożenia a przedstawiono na rysunku 4.15.



RYS. 4.15. Schemat układu operacyjnego, który implementuje algorytm mnożenia a

Zgodnie z algorytmem mnożenia a [10], mnożna A jest ładowana do najmniej znaczących bitów rejestru przesuwającego ra o szerokości $2N$ bitów. W każdej iteracji zawartość rejestru ra jest przesuwana o jeden bit w lewo (w kierunku bardziej znaczących bitów). Wartość mnożnika B jest ładowana do rejestru przesuwającego rb , który jest przesuwany o jeden bit w prawo przy każdej iteracji (w kierunku najmniej znaczącego bitu). Wartość iloczynu P jest przechowywana w $2N$ -bitowym zwykłym rejestrze rp .

Schemat układu operacyjnego zawiera również $2N$ -bitowy multiplekser magistrali, $2N$ -bitowy sumator i licznik modulo *counter*, który generuje sygnał *roll*, aby wskazać koniec algorytmu mnożenia. Sygnał *roll* jest tym samym sygnałem, co zewnętrzny sygnał *Done*.

Algorytm a jest wykonywany w N iteracjach (cyklach zegara). W każdej iteracji sprawdzany jest bit $rb[0]$. Jeżeli bit $rb[0]$ jest równy 0, to jako pierwszy operand sumatora wybierana jest wartość zerowa, w przeciwnym razie wybierana jest zawartość rejestru ra , która jest przesuniętą wartością mnożnika A . Drugim operandem sumatora jest zawartość rejestru rp , tzn. aktualna wartość iloczynu.

Wyjście sumatora jest wejściem do rejestru rp . W rzeczywistości sumator i rejestr rp są akumulatorem.

Zwróćmy uwagę, że podczas działania układu mnożącego wartości na wyjściu P będą ulegać zmianie. Natomiast wynik będzie wiarygodny (poprawny) dopiero po ustawieniu sygnału *Done*. Jeżeli takie zachowanie wyjścia układu mnożącego w systemie jest niedopuszczalne, to na wyjściu układu mnożącego należy umieścić dodatkowy rejestr, który jest przełączany na narastającym zboczku sygnału *Done*.

Wszystkie rejestry układu operacyjnego są taktowane zegarem clk . Rejestry przesuwające ra i rb mają sygnał sterujący *load*, służący do załadowania wartości początkowej i sygnał *ena*, do uaktywnienia operacji przesunięcia zawartości rejestru. Ponieważ rejestry ra i rb nie są bezpośrednio podłączone do wyjścia układu mnożącego, mogą nie mieć sygnału sterującego do resetowania. Rejestr wyjściowy rp ma sygnał sterujący *clr* służący zarówno do resetowania systemowego, jak i do czyszczenia rejestru rp przed rozpoczęciem iteracji oraz sygnał *ena* do zezwolenia przełączania.

Licznik *counter*, oprócz sygnału synchronizacji clk , posiada sygnały sterujące *clr* do resetowania i *ena* do zezwolenia zliczania.

Zwróćmy uwagę, że w układzie na rysunku 4.15 sygnał $b_out[0]$, który steruje wyborem wejść multipleksa, jest generowany nie przez urządzenie sterowania, ale jest wartością bitu zerowego rejestru rb . Jest to łatwiejsze pod względem implementacji sprzętowej. W takim przypadku urządzenie operacyjne wykonuje funkcje układu sterującego.

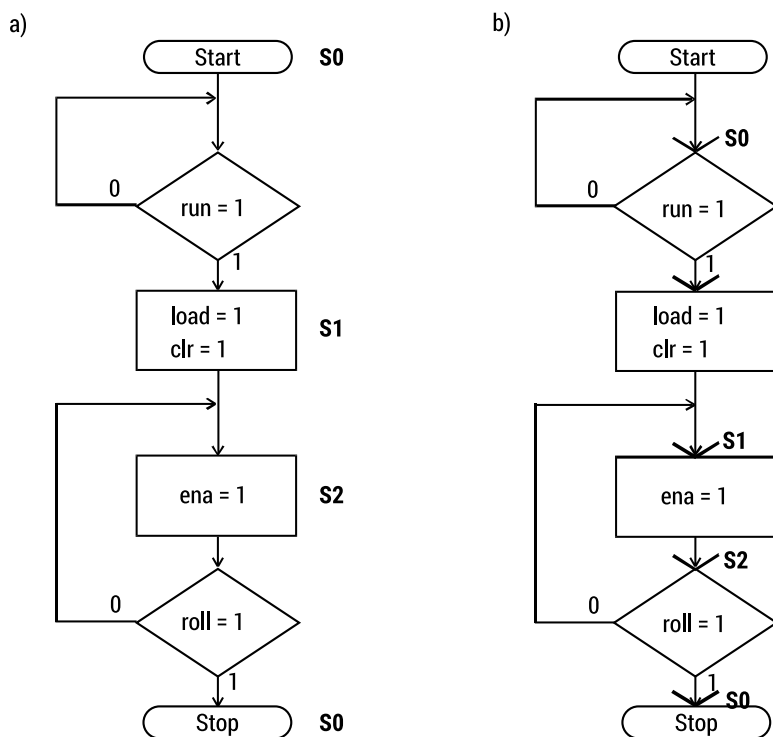
Zasadniczo nie ma ścisłej granicy między funkcjami układu operacyjnego i układu sterującego. Niektóre funkcje układu sterującego mogą być realizowane przez urządzenie sterowania (jak w naszym przykładzie) i odwrotnie.

4.5.3. Projektowanie układu sterującego

Zadaniem układu sterującego jest generowanie sygnałów sterujących dla układu operacyjnego, zapewniających poprawność działania określonego algorytmu przetwarzania danych.

W naszym przypadku przy implementacji algorytmu mnożenia *a* sygnałami wejściowymi układu sterującego są *reset*, *Run* and *roll*, na podstawie których układ sterowania tworzy wartości sygnałów *load*, *clr* i *ena* (rys. 4.14).

Algorytm działania układu sterującego jest w całości tworzony przez projektanta. Nie ma tutaj ograniczeń, ale ten etap często jest źródłem błędów w działaniu systemu. Jeden z możliwych sposobów opisu działania sterownika w języku GSA pokazany jest na rysunku 4.16.



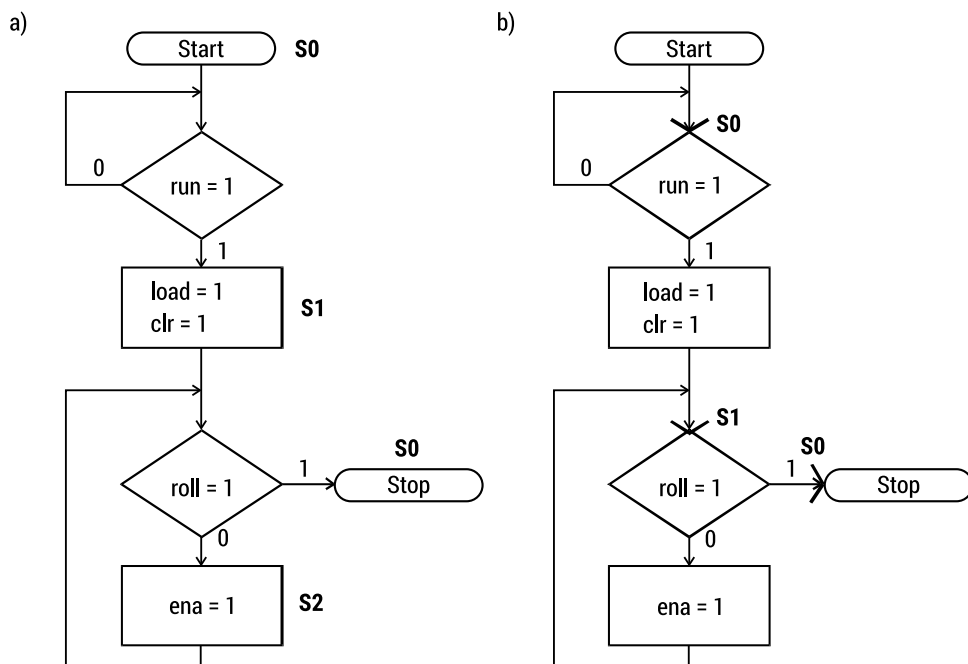
RYŚ. 4.16. Układ sterowania GSA do realizacji algorytmu mnożenia *a*: a – oznaczony etykietami do syntezy AM Moore'a, b – oznaczony do syntezy AM Mealy'ego

Rysunek 4.16a przedstawia GSA do syntezy AM Moore'a, a rysunek 4.16b przedstawia GSA do syntezy AM Mealy'ego. Pierwszy wierzchołek po początkowym jest wierzchołkiem warunkowym oczekującym, w którym sprawdzana jest wartość sygnału *Run*. Jeśli sygnał *Run* jest ustawiony, wykonywana jest mikroinstrukcja, w której wykonywane są mikrooperacje *load* (ładowanie rejestrów przesuwających *ra* i *rb*) oraz *clr* (resetowanie rejestru *rp*). Następnie wykonywana jest pętla, w której przez

ustawienie sygnału *ena* przesuwa się zawartość rejestrów *ra* i *rb*, a rejestr *rp* jest przełączany. Pętla jest wykonywana tak długo, jak długo sygnał *roll* = 0. W przypadku ustawienia sygnału *roll*, algorytm mnożenia kończy się.

Analiza rysunków 4.16a i 4.16b pokazuje, że AM Moore'a i AM Mealy'ego, które implementują dany algorytm sterowania logicznego, mają taką samą liczbę stanów (S0, S1 i S2).

Ten sam algorytm mnożenia *a* można opisać za pomocą GSA pokazanego na rysunku 4.17.



RYS. 4.17. Druga wersja GSA dla układu sterującego: a – przy oznaczaniu etykietami do syntezy AM Moore'a, b – przy oznaczaniu do syntezy AM Mealy'ego

Różnica między tymi dwoma GSA polega na tym, że w GSA z rysunku 4.16, w pętli najpierw ustawiany jest sygnał *ena*, a następnie sprawdzany jest sygnał *roll*, a w GSA (rys. 4.17), odwrotnie, najpierw sprawdzany jest sygnał *roll*, a następnie ustawiany jest sygnał *ena*. GSA na rysunku 4.17 dla syntezy AM Moore'a oznaczono trzema znacznikami (etykietami), a GSA na rysunku 4.17b dla syntezy AM Mealy'ego jest oznaczony tylko dwoma znacznikami. Oznacza to, że AM Mealy'ego, który realizuje GSA na rysunku 4.17, będzie miał tylko dwa stany, dla których wystarczy jeden bit do kodowania.

Wniosek. Ten sam algorytm sterowania logicznego można opisać przy pomocy różnych GSA, które mogą różnić się liczbą stanów AM.

4.5.4. Implementacja układu mnożącego na FPGA

Poniżej znajdują się opisy komponentów jednostki operacyjnej układu mnożącego w języku Verilog.

```
module shl_load                                // rejestr przesuwający w lewo
#(parameter N=4)
(input clk, load, ena,
input [N-1:0] d,
output reg [N-1:0] q);
```

```
always @(posedge clk, posedge load)
    if(load)    q <= d;
    else if(ena)  q <= {q[N-2:0],1'b0};
    else        q <= q;
```

```
endmodule
```

```
module shr_load                                // rejestr przesuwający w prawo
#(parameter N=4)
(input clk, load, ena,
input [N-1:0] d,
output reg [N-1:0] q);
```

```
always @(posedge clk, posedge load)
    if(load)    q <= d;
    else if(ena)  q <= {1'b0,q[N-1:1]};
    else        q <= q;
```

```
endmodule
```

```
module mux2                                    // multiplexer magistrali 2-1
#(parameter WIDTH = 32)
(input [WIDTH-1:0] d0, d1,
input s,
output [WIDTH-1:0] y);
```

```
assign y = s ? d1 : d0;
```

```
endmodule
```

```
module adder                                    // sumator
```

```

#(parameter N=4)
(input [N-1:0] a, b,
output [N-1:0] y);

assign y = a + b;

endmodule

module register // rejestr z wejściem zezwolenia synchronizacji
#(parameter WIDTH = 4)
(input clk, reset, ena,
input [WIDTH-1:0] d,
output reg [WIDTH-1:0] q);

always @(posedge clk, posedge reset)
    if (reset) q <= 0;
    else if (ena) q <= d;

endmodule

module counter_modulo_M // licznik modulo
#(parameter M=4)
(input clk, reset, ena,
output roll);

reg [N-1:0] Q;

localparam N=clogb2(M-1);

function integer clogb2(input [M-1:0] v);
    for (clogb2 = 0; v > 0; clogb2 = clogb2 + 1)
        v = v >> 1;
endfunction

always @(posedge clk, posedge reset)
    if (reset) Q <= {N{1'b0}};
    else if (ena)
        if (Q == M-1) Q <= {N{1'b0}};
        else Q <= Q + 1'b1;

```

```
assign roll = (Q == M-1);
endmodule
```

```
module my_dff(                // przerzutnik typu D
input clk, reset, ena, d,
output reg q);

always @(posedge clk, posedge reset)
    if (reset) q <= 1'b0;
    else if (ena) q <= d;

endmodule
```

endmodule

Opis elementów układu operacyjnego w pełni odpowiada tradycyjnemu opisowi odpowiednich standardowych bloków funkcjonalnych w języku Verilog [10].

Opis układu operacyjnego w języku Verilog jest następujący:

```
module datapath_mult
#(parameter N=4)           // N – szerokość słów wejściowych
(input clk,                 // zegar
input [N-1:0] a, b,        // słowa wejściowe
input load, clr, enable,   // sygnały sterujące
output roll,               // wyjście licznika modulo
output [2*N-1:0] p,        // iloczyn
output done);              // flaga końca mnożenia
```

```
wire [N-1:0] b_out;        // magistrale pośrednie
wire [2*N-1:0] op_2, p_in, p_out, a_out;
```

// rejestry przesuwające

```
shl_load #(2*N)    ra(clk,load, enable, {{N{1'b0}},a}, a_out);
shr_load #(N)      rb(clk,load, enable, b, b_out);
```

// multiplexer

```
mux2 #(2*N) ex_mux({2*N{1'b0}}, a_out, b_out[0], op_2);
```

// sumator

```
adder #(2*N) ex_adder(p_out, op_2, p_in);
```

// rejestr wyjściowy

```
register #(2*N) rp(clk, clr, enable, p_in, p_out);
```

```
// licznik modulo
counter_modulo_M #(N) ex_counter(clk, clr, enable, roll);

assign p = p_out;                // ustalenie wyjścia

my_dff ex_dff(clk, clr, 1'b1, roll, done);    // zapamiętanie sygnału done
endmodule
```

Zastosowano tutaj styl strukturalny opisu projektu w języku Verilog [10], gdzie komponenty układu operacyjnego są tworzone na podstawie opisanych wcześniej modułów i określone są połączenia między nimi.

Opis w języku Verilog AM Mealy'ego, odpowiadający GSA na rysunku 4.17b, jest następujący:

```
module fsm_Mealy(
input clk, reset, run, roll,      // sygnały wejściowe
output load, clr, ena);          // generowane sygnały wyjściowe

reg [2:0] y;                      // wektor wyjściowy

localparam [1:0] s0=0,s1=1;      // definicja stanów

reg [1:0] state, next;           // zmienne stanu

always @(posedge clk)            // opis pamięci automatu
    if(reset) state <= s0;
    else      state <= next;

always @(*)                      // opis przejść i wyjść automatu
    case (state)
        s0: if(run)          begin next = s1; y = 3'b110; end
            else          begin next = s0; y = 3'b000; end
        s1: if(roll)         begin next = s0; y = 3'b000; end
            else          begin next = s1; y = 3'b001; end
        default:          begin next = s0; y = 3'b000; end
    endcase

// określenie wartości sygnałów wyjściowych
assign {load, clr, ena} = y;
endmodule
```

W tym przykładzie opisu automatu skończonego użyliśmy stylu automatu Mealy'ego z dwoma procesami, w którym opis funkcji przejścia i wyjścia jest połączony w jeden proces. Cechą powyższego opisu jest to, że do określenia wartości wyjść automatu używany jest wektor bitowy, a wartości poszczególnych sygnałów wyjściowych są określane za pomocą operatora przypisania ciągłego **assign**.

Moduł najwyższego poziomu układu mnożącego odpowiadający algorytmowi *a* jest następujący:

```

module mult_a
#(parameter N=4)                // N – szerokość słów wejściowych
(input clk, reset, run,         // sygnały sterujące
input [N-1:0] a,b,             // słowa wejściowe
output [2*N-1:0] p,            // iloczyn
output done);                  // flaga końca procesu mnożenia

wire load, clr, ena, roll;    // sygnały pośrednie

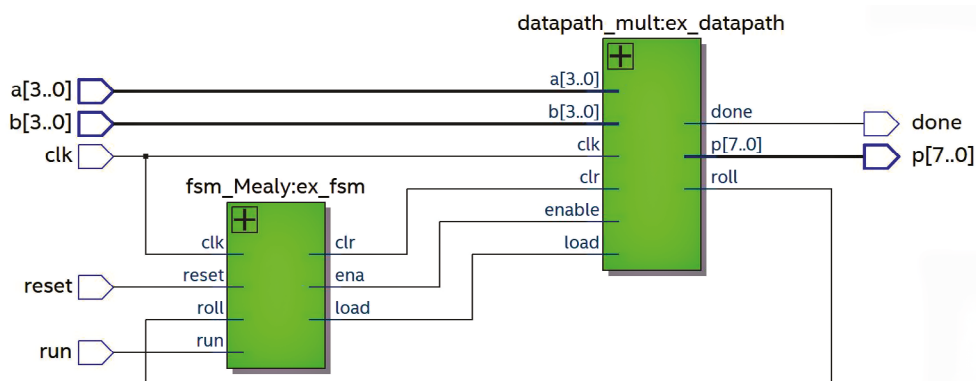
// opis układu operacyjnego
datapath_mult #(N) ex_datapath
    (clk, a, b, load, clr, ena, roll, p, done);

// opis układu sterującego
fsm_Mealy'ego ex_fsm
    (clk, reset, run, roll, load, clr, ena);

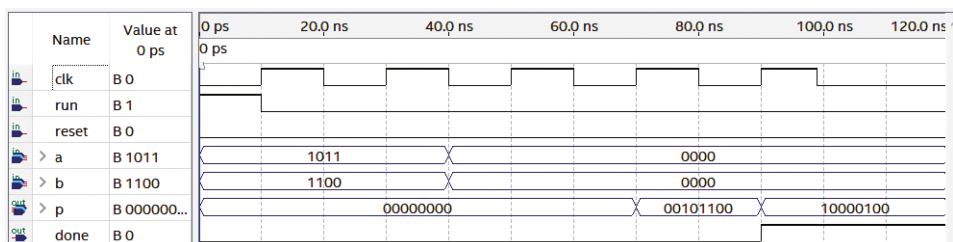
endmodule

```

Wyniki syntezy i symulacji modułu *mult_a* przedstawiono, odpowiednio, na rysunkach 4.18 i 4.19.



RYS. 4.18. Widok projektu *mult_a* na poziomie RTL



RYS. 4.19. Wyniki symulacji projektu *mult_a*

Czytelnik jest proszony o samodzielne wykonanie sprzętowej implementacji algorytmów mnożenia *b*, *c* i *d* z pracy [10].

Komentarz. W algorytmie mnożenia *d* każdy cykl wymaga załadowania i przesunięcia zawartości rejestru *rp*. Ponieważ operacje ładowania i przesunięcia tego samego rejestru nie mogą być wykonywane w jednym cyklu zegara, każdy cykl mnożenia jest wykonywany w dwóch cyklach zegara.

4.5.5. Porównanie różnych sposobów implementacji synchronicznych układów mnożących

W pracy [10] rozpatrywano projekty synchronicznych układów mnożących, opisując algorytm działania układu mnożącego w języku Verilog. Ponadto każda operacja w kodzie Verilog w układzie mnożącym odpowiada pewnemu komponentowi, na przykład sumator odpowiada operacji dodawania. Liczba sumatorów w układzie mnożącym jest określana przez liczbę iteracji (szerokość słowa *N*) podczas obliczania i dodawania iloczynów częściowych. Ponadto wymiar sumatorów jest wprost proporcjonalny do parametru *N*. Dlatego koszt realizacji układu mnożącego rośnie szybko wraz ze wzrostem *N*.

W tej sekcji zaproponowano sprzętową implementację algorytmicznych układów mnożących w postaci układu operacyjnego i układu sterującego. W tym przypadku w układzie mnożącym stosowany jest tylko jeden sumator, ale operacja mnożenia wykonywana jest w kilku taktach synchronizacji: dla algorytmu *a* – w *N* taktach zegarowych, dla algorytmu *b* – w *2N* taktach zegarowych.

Powstaje pytanie: które z podejść jest lepsze pod względem kosztów i szybkości realizacji oraz dla jakich wartości *N*.

Aby odpowiedzieć na to pytanie, algorytmy mnożenia *a*, *b*, *c* i *d* z [10] zostały zaimplementowane w sprzęcie. Do badań wybrano algorytmy *a* i *d* (projekty *mult_a* i *mult_d*), które porównano z opisem algorytmu *a* w języku Verilog (projekt *algorithm_a*). Dodatkowo prezentowane projekty porównano z metodą systemu Quartus, która realizuje operację mnożenia arytmetycznego: $P = A * B$ (projekt *Quartus*).

Badania eksperymentalne przeprowadzono w systemie Quartus w wersji 18.1 (dla układów FPGA z rodziny Cyclone IV E). Aby dokładniej określić maksymalną

częstotliwość pracy na wejściach i wyjściach wszystkich układów mnożących, ustawiono rejestry i określono ograniczenia czasowe, korzystając z pliku ograniczeń czasowych SDC [9]. Wyniki badań przedstawiono w tabeli 4.5, gdzie N jest szerokością słów wejściowych A i B ; L to liczba elementów logicznych FPGA (w przypadku projektu Quartus liczba użytych wbudowanych w FPGA bloków mnożenia jest podana w nawiasach); F to maksymalna częstotliwość sygnału zegarowego w megahercach (MHz) dla danego projektu.

TAB. 4.5. Parametry układów mnożących z uwzględnieniem liczby taktów sygnału zegara

Method	N=4		N=8		N=16		N=32		N=64		N=128	
	L	F	L	F	L	F	L	F	L	F	L	F
<i>mult_a</i>	36	61.25	66	22.75	123	9.75	236	4.41	462	1.66	919	0.25
<i>mult_d</i>	37	28.38	56	13.00	99	6.84	181	2.66	343	1.11	666	0.21
<i>algorithm_a</i>	34	130.00	129	75.00	513	30.00	2043	11.00	8197	4.86	32897	–
<i>Quartus</i>	32 (0)	170.00	0 (1)	219.6	0 (2)	193.00	97 (8)	111.70	455 (32)	73.00	2329 (112)	25.10

Najpierw przeanalizujemy opracowane przez nas projekty układów mnożących *mult_a*, *mult_d* i *algorithm_a*, a następnie porównajmy je z metodą systemu Quartus.

Z tabeli 4.5 wynika, że dla $N = 4$ koszt projektów *mult_a*, *mult_d* i *algorithm_a* jest w przybliżeniu taki sam (34–37), ale wydajność projektu *algorithm_a* jest 2,1 i 4,6 razy wyższa niż odpowiednio, projektów *mult_a* i *mult_d*. Wraz ze wzrostem wartości N , koszt realizacji projektu *algorithm_a* rośnie szybciej. Zatem dla $N = 64$ koszt realizacji projektu *algorithm_a* jest wyższy niż koszt realizacji projektu *mult_a* 17,7 razy, a projektu *mult_d* 23,9 razy. Jednak pod względem wydajności projekt *algorithm_a* nadal przewyższa projekt *mult_a* 2,9 razy, a projekt *mult_d* 4,4 razy. Przy $N = 128$ projekt *algorithm_a* nie pozwala na implementację układu mnożącego na układach FPGA z rodziny Cyclone IV E.

Zatem spośród projektów *mult_a*, *mult_d* i *algorithm_a* projekt *mult_d* ma najniższy koszt realizacji, a projekt *algorithm_a* ma najwyższą wydajność.

Analizując tabelę 4.5 można zauważyć, że przy implementacji układów mnożących metodą systemu Quartus są szeroko stosowane wbudowane moduły mnożące, które znajdują się we wbudowanych blokach DSP (digital signal processing). Liczbę wbudowanych modułów mnożenia użytych w tabeli 4.5 podano w nawiasach.

Przy $N = 4$ wbudowane układy mnożące FPGA nie są używane. Koszt realizacji projektu *Quartus* wynosi 32, co jest wartością współmierną do wartości uzyskanych dla rozpatrywanych projektów układów mnożących. Maksymalna częstotliwość pracy projektu *Quartus* (170 MHz) również nieznacznie przewyższa maksymalną częstotliwość pracy projektu *algorithm_a* (130 MHz).

Przy $N = 8$ i $N = 16$ projekt *Quartus* w ogóle nie wykorzystuje komórek logicznych. Oznacza to, że układy mnożące są w pełni zaimplementowane w blokach wbudowanych DSP. Przy tym wydajność projektu *Quartus* przewyższa wydajność projektu

algoritm_a w 2,9 razy dla $N = 8$ i w 6,4 razy dla $N = 16$. Wraz ze wzrostem parametru N utrzymuje się podobny trend: projekt *Quartus* znacznie przewyższa projekty *mult_a*, *mult_d* i *algoritm_a* pod względem szybkości.

Jednak przy $N = 64$ koszt realizacji projektu *Quartus* (455) staje się współmierny do projektów *mult_a* i *mult_d* (462 i 343) i to pomimo zastosowania wbudowanych bloków FPGA. A przy $N = 128$ koszt projektu *Quartus* przekracza 2,5-krotnie projekt *mult_a*, i 3,5-krotnie projekt *mult_d*.

Podsumowując, metoda mnożenia systemu *Quartus* znacznie przewyższa wszystkie opracowane przez nas metody pod względem szybkości dla różnych wartości N . Jednak przy $N = 128$ metoda *Quartus* jest 3,5-krotnie słabsza od projektu *mult_d* pod względem kosztów realizacji. Przewaga szybkości metody *Quartus* wynika z efektywnego wykorzystania bloków wbudowanych DSP.

Wnioski. Przy implementacji układów mnożących na układach FPGA, aby osiągnąć wysoką wydajność przy dowolnych wartościach N , należy zastosować metodę mnożenia systemu *Quartus*.

Przy implementacji układów mnożących na układach FPGA, aby uzyskać najniższy koszt implementacji dla $N < 64$ należy zastosować metodę systemową *Quartus*, a dla $N \geq 64$ zastosować sprzętową implementację algorytmu *d* (projekt *mult_d*).

4.6. Wnioski

Złożony rozkaz komputerowy jest wykonywany pod kontrolą *mikroprogramu* w kilku cyklach zegarowych. Mikroprogram składa się z oddzielnych *mikroinstrukcji* (microcodes). Z kolei mikroinstrukcja składa się z pojedynczych *mikrooperacji* (microoperations), które wykonują elementarne czynności. Każda mikroinstrukcja jest wykonywana w jednym cyklu zegara. Kolejność wykonania mikroinstrukcji jest kontrolowana przez *warunki logiczne* (logical conditions). Automat skończony, który implementuje mikroprogram, nazywany jest *automatem mikroprogramowanym* (AM – microprogrammed state machine). Wejścia AM są warunkami logicznymi, a wyjścia są mikrooperacjami.

Sterowanie mikroprogramowe rozwinęło się w robotyce, przy projektowaniu sterowników przemysłowych, w technologii komputerowej i w systemach wbudowanych, gdzie w oparciu o zasady sterowania mikroprogramowego budowane są różne układy sterujące, często nazywane kontrolerami (controllers).

Graficzne schematy algorytmów (GSA) (sieć działań) to ukierunkowany graf, który zawiera jeden wierzchołek początkowy, jeden końcowy oraz kilka wierzchołków operacyjnych i warunkowych. *Mikroinstrukcja* (zestaw mikrooperacji) jest zapisana w każdym wierzchołku operacyjnym. Warunki logiczne są zapisywane w wierzchołkach warunkowych. GSA musi spełniać następujące warunki: wejścia i wyjścia wierzchołków są połączone za pomocą strzałek, zawsze skierowanych od wyjścia do wejścia; każde wyjście jest podłączone tylko do jednego wejścia; dowolny wierzchołek

operacyjny lub wierzchołek warunkowy leży na co najmniej jednej ścieżce od wierzchołka początkowego do wierzchołka końcowego; opisując działanie AM, sprzężenia zwrotne w GSA muszą przechodzić przez wierzchołki operacyjne.

Projektowanie AM zgodnie z GSA polega na oznaczeniu etykietami (znacznikami) GSA do syntezy AM określonego typu (Mealy'ego lub Moore'a), identyfikacji etykiet GSA ze stanami wewnętrznymi AM i budowy tablicy przejść AM.

Ten sam algorytm sterowania logicznego można opisać na różne sposoby, tzn. różnymi GSA. Niektóre z tych GSA prowadzą do skutecznej (pod względem kosztów realizacji i szybkości) implementacji AM na układach FPGA, podczas gdy inne – nie.

Różne sposoby oznaczania GSA definiują różne AM, które realizują ten sam algorytm sterowania logicznego. Niektóre sposoby oznaczania GSA prowadzą do skutecznej implementacji AM na FPGA, podczas gdy inne – nie. Ponadto nie wszystkie oznaczenia GSA są prawidłowe, tzn. niektóre oznaczenia nie pozwalają na budowę AM, który implementuje dany algorytm sterowania. W rezultacie, przy użyciu tego samego GSA, można zbudować kilka różnych AM o różnej wydajności i różnym koszcie realizacji na FPGA.

W [2] zaproponowano metody oznaczania GSA minimalną liczbą znaczników (etykiet) do syntezy AM typu Mealy'ego i AM typu Moore'a. Etykiety wprowadzone zgodnie z zasadami [2] nazywane są *głównymi etykietami* GSA, a pozostałe – *dodatkowymi etykietami* GSA.

Dwa AM S_1 i S_2 są *równoważne*, jeśli dla dowolnej sekwencji zbiorów wartości zmiennych wejściowych $w_1, \dots, w_k$ generują identyczne sekwencje zbiorów wartości zmiennych wyjściowych $z_1, \dots, z_k$. Automat S_2 nazywany jest *pseudorównoważnym* do automatu S_1 , jeśli dla jakiegoś ciągu $z_1, \dots, z_k$ automat S_2 generuje ciąg zbiorów wartości zmiennych wyjściowych, w którym sekwencja zbiorów wyjściowych odpowiada ciągowi $w_1, \dots, w_k$, ale może zawierać dodatkowe zbiory zerowe. W wielu praktycznych zastosowaniach dodatkowe zbiory zerowe w sekwencji $w_1, \dots, w_k$ są całkiem dopuszczalne. W niektórych przypadkach wprowadzenie dodatkowych etykiet pozwala uprościć syntezę AM.

Po oznaczeniu GSA i zbudowaniu tablicy przejść, implementacja AM opisanych za pomocą GSA nie różni się od implementacji na FPGA automatów skończonych.

Funkcjonowanie AM Mealy'ego i Moore'a przy implementacji tego samego algorytmu sterowania logicznego jest różne. Okoliczność tę należy wziąć pod uwagę przy wyborze rodzaju realizowanego AM.

Schemat blokowy synchronicznego układu mnożącego można przedstawić w postaci *układu operacyjnego* i *układu sterującego*, który jest realizowany w postaci automatu mikroprogramowanego.

Zasadniczo nie ma ścisłej granicy między funkcjami układu operacyjnego i układu sterującego. Niektóre funkcje układu sterującego mogą być realizowane przez urządzenie operacyjne i odwrotnie.

Algorytm działania układu sterującego jest opracowywany przez użytkownika, na przykład w postaci GSA. Nie ma tutaj ograniczeń, ale ten etap często jest źródłem błędów w działaniu systemu.

Ten sam algorytm sterowania logicznego można opisać za pomocą różnych GSA, mogących różnić się liczbą stanów AM, które je implementują.

Przy implementacji układów mnożących w strukturach FPGA należy zastosować metodę mnożenia systemu Quartus, aby uzyskać wysoką wydajność przy dowolnych wartościach szerokości słów wejściowych N .

Przy implementacji układów mnożących w strukturach FPGA, aby uzyskać najniższy koszt implementacji dla $N < 64$ należy zastosować metodę systemową Quartus, a dla $N \geq 64$ zastosować sprzętową implementację algorytmu d [10] (projekt *mult_d*).

5. Algorytmiczne automaty skończone (ASM)

W tym rozdziale omówiono język graficzny służący do opisu automatów skończonych, nazywany *algorytmicznym automatem skończonym* (algorithmic state machine – ASM). Przypomnijmy sobie, że do opisu algorytmów w programowaniu używane są tradycyjne *schematy blokowe*. Wcześniej rozważane *graficzne schematy algorytmów* (GSA) są przeznaczone do opisu algorytmów sterowania logicznego i nie są dostatecznie wygodne do opisu automatów skończonych. Z tego powodu zaproponowano język ASM do opisu funkcjonowania automatów skończonych.

Główną zaletą ASM w porównaniu z GSA jest to, że w języku Verilog automat skończony można bezpośrednio opisać poprzez jego reprezentację w postaci ASM, bez użycia oznaczania GSA, identyfikacji etykiet GSA ze stanami automatu skończonego i budowania tablicy przejść.

Ten rozdział zawiera opis języka ASM, metodologię budowy ASM, przykłady budowy ASM dla automatów Mealy’ego i Moore’a, cechy implementacji na FPGA automatów Mealy’ego i Moore’a opisywanych w języku ASM, a także inne cechy języka ASM: opis wspólnych modeli automatów Mealy’ego i Moore’a, procesów równoległych i układów kombinacyjnych.

Można stwierdzić, że ASM jest potężnym narzędziem do projektowania automatów skończonych, które nie zyskało jeszcze powszechnej akceptacji wśród projektantów systemów wbudowanych.

5.1. Problemy GSA przy opisie algorytmów działania sprzętu

W poprzednim rozdziale rozważaliśmy kwestie projektowania *automatów mikroprogramowanych* (AM), których zachowanie opisano w języku *graficznych schematów algorytmów* (GSA). Ogólnie rzecz biorąc, GSA są przeznaczone do opisanego algorytmów sterowania logicznego, a nie algorytmów funkcjonowania sprzętu, tzn. systemów wbudowanych na FPGA. Stąd pojawiają się pewne sprzeczności. Zwróćmy uwagę na niektóre z nich.

Przypomnijmy, że w przypadku przejścia z GSA do AM, przeprowadza się formalne oznaczenie GSA dla syntezy AM określonego typu: Mealy’ego lub Moore’a. Następnie etykiety GSA są identyfikowane ze stanami wewnętrznymi AM i budowana

jest tablica przejść automatu skończonego. Ponieważ oznaczenie GSA jest wykonywane metodami formalnymi, może się okazać, że stany AM nie odpowiadają stanom, których projektant oczekiwał w projektowanym sprzęcie.

Szereg problemów powoduje również budowa tablicy przejść AM w oparciu o oznaczony GSA. Na przykład, jak zaimplementować wierzchołki oczekiwania i pętle sprzężenia zwrotnego GSA, jak przejść wierzchołki warunkowe z powtarzającymi się warunkami logicznymi, w której etykiecie zakończyć przejście podczas budowy AM Mealy'ego, jeśli po drodze nie napotkano wierzchołka operacyjnego itp.

Ponadto metodyka budowy AM na podstawie GSA, rozważana w rozdziale 4, nie pozwala na sprawdzenie warunków logicznych przy generowaniu sygnałów wyjściowych automatu Mealy'ego, ponieważ etykieta oznacza wejście wierzchołka następującego po wierzchołku operacyjnym, tzn. przejście musi być zakończone natychmiast po napotkaniu wierzchołka operacyjnego.

Zauważmy też, że w GSA nie ma połączenia z sygnałem synchronizacji, tzn. nie można dokładnie określić, za ile cykli zegara będzie wykonywany określony fragment GSA.

Używając GSA do opisu działania AM, można zauważyć następujące wady GSA:

- nie ma bezpośredniego związku między algorytmem funkcjonowania urządzenia a jego stanami wewnętrznymi;
- nie ma jednoznaczności w realizacji cykli oczekiwania (wierzchołki oczekiwania GSA);
- nie można sprawdzić warunków logicznych po uformowaniu sygnałów wyjściowych automatu Mealy'ego;
- nie ma powiązania działania algorytmu z cyklami zegara.

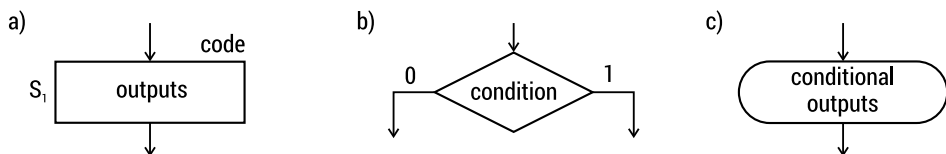
Zauważmy, że w praktyce projektowania inżynierskiego bardzo ważne jest, aby dokładnie wiedzieć, ile cykli zegarowych zajmie dana gałąź algorytmu działania układu.

Wszystkie te wady są nieobecne przy zastosowaniu języka algorytmicznych automatów skończonych ASM do opisu działania układów sterowania.

5.2. Język ASM

Język *algorytmicznych automatów skończonych* (algorithmic state machine – ASM) lub *sieć działań algorytmu* to graf skierowany, zawierający trzy typy wierzchołków (rys. 5.1):

- prostokąty – *wierzchołki stanów* (state box);
- romby – *wierzchołki warunkowe* (decision box);
- owale – *wierzchołki wyjść warunkowych* (conditional output box).



RYS. 5.1. Elementy ASM: a – wierzchołek stanów; b – wierzchołek warunkowy; c – wierzchołek wyjść warunkowych

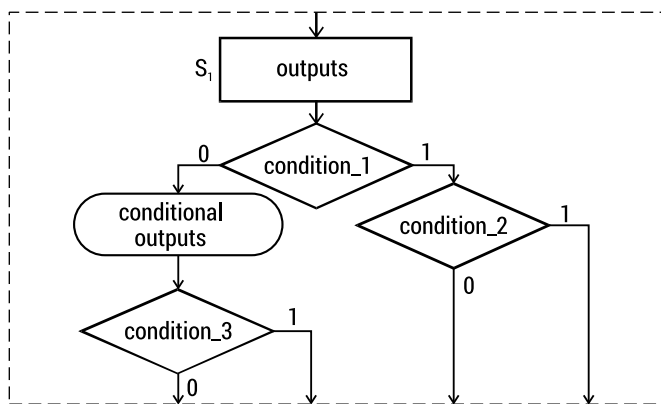
Wierzchołek stanu ASM określa stan automatu. W górnej części stanu można zapisać nazwę stanu (na przykład S0, START, INITIAL itp.), a także kod binarny stanu. W przypadku automatu Moore’a, wewnątrz wierzchołka stanu, zapisywane są te sygnały wyjściowe, które w tym stanie przyjmują wartość jeden. Domyślnie przyjmuje się, że wszystkie inne wyjścia w tym stanie mają wartość zero. Należy zauważyć, że ASM nie pozwala na opisywanie automatów skończonych z nieokreślonymi wyjściami.

Wierzchołek warunkowy ASM w pełni odpowiada wierzchołkowi warunkowemu GSA: zapisywany jest w nim sprawdzany warunek. Wyjścia węzła warunkowego są oznaczone wartościami 0 i 1, które odpowiadają przejściom w przypadku zerowej (fałszywej) lub jedynkowej (prawdziwej) wartości wyniku sprawdzania warunku. Jako warunek można zastosować zmienną wejściową automatu, wyrażenie logiczne, pojedynczy bit wektora bitowego itp. Podobnie jak w GSA, warunkowy wierzchołek ASM jest punktem rozgałęzienia algorytmu.

Wierzchołek wyjścia warunkowego (owal) odpowiada wierzchołkowi operacyjnemu GSA dla automatu Mealy’ego. W nim zapisywane są sygnały wyjściowe automatu Mealy’ego, które przyjmują wartość jeden przy tym przejściu. Sygnały wyjściowe zapisane w owalach nazywane są *wyjściami warunkowymi* (conditional output).

W ASM dla automatów Moore’a nie ma wierzchołków wyjść warunkowych (owali), a w ASM dla automatów Mealy’ego nic nie jest zapisane w wierzchołkach stanów.

Głównym blokiem konstrukcyjnym sieci działań ASM jest *blok ASM* (ASM block), którego przykład pokazano na rysunku 5.2.

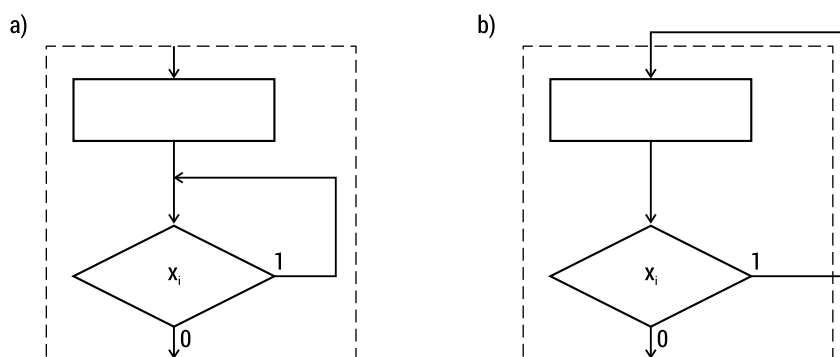


RYS. 5.2. Blok ASM

W ogólnym przypadku, blok ASM zawiera tylko jeden wierzchołek stanu i może mieć kilka wierzchołków warunkowych i wierzchołków wyjść warunkowych. Blok ASM może mieć dowolną liczbę rombów i owali (lub nie mieć ich wcale), a romby mogą poprzedzać owale lub następować po nich. Wejścia i wyjścia wierzchołków są połączone za pomocą strzałek (krawędzi skierowanych). Blok ASM ma tylko jedno wejście i może mieć jedno lub więcej wyjść. Blok ASM opisuje zachowanie automatu w jednym stanie podczas jednego cyklu synchronizacji.

Sieć działań ASM (zwana dalej po prostu ASM) to zbiór połączonych ze sobą bloków ASM. Użytkownik musi przestrzegać następujących zasad tworzenia ASM:

- każde wyjście dowolnego wierzchołka ASM można podłączyć tylko do jednego wejścia innego wierzchołka, tzn. rozgałęzienie algorytmu jest możliwe tylko w wierzchołkach warunkowych;
- sprzężenia zwrotne są zabronione wewnątrz bloku ASM (rys. 5.3).



RYŚ. 5.3. Implementacja pętli oczekiwania w ASM: a – źle, b – poprawnie

Zwróćmy uwagę na niektóre charakterystyczne cechy ASM. ASM wyraźnie definiuje stany wewnętrzne automatu za pomocą bloków ASM. Przejście z jednego bloku ASM do drugiego odbywa się zawsze w jednym cyklu zegarowym, tzn. algorytm działania automatu jest bezpośrednio powiązany z synchronizacją. Dzięki temu, zgodnie z ASM, zawsze można prześledzić, ile cykli potrzeba na wykonanie danego fragmentu algorytmu. Jednocześnie ASM zachowuje przejrzystość GSA i sieci działań algorytmów. Ponadto ASM wstępnie określa typ automatu: Mealy'ego lub Moore'a.

5.3. Metodyka budowy ASM

Ponieważ automat skończony jest matematycznym modelem dowolnego układu sekwencyjnego, ASM można wykorzystać do opisu działania dowolnego sprzętu. Przedstawimy metodykę budowy ASM dla automatu skończonego w postaci następującego algorytmu.

Algorytm 5.1. Metodyka budowy ASM

1. Określane są stany automatu skończonego.
2. Dla każdego stanu budowany jest blok ASM. Przy czym, wszystkie przejścia od danego stanu są określane dla wszystkich możliwych wartości zmiennych wejściowych. Jeśli jakaś zmienna wejściowa nie wpływa na przejścia z danego stanu, to nie występuje wewnątrz bloku ASM.
3. Dla automatu Moore'a w wierzchołkach stanów zapisywane są zmienne wyjściowe, które przyjmują wartość jeden w danym stanie.
4. W przypadku automatu Mealy'ego zmienne wyjściowe, które przyjmują wartość jedynkę na danym przejściu, są zapisywane w wierzchołkach wyjść warunkowych (owale).
5. Bloki ASM są ze sobą łączone zgodnie z algorytmem działania układu. W ten sposób każde wyjście bloku ASM może być podłączone tylko do jednego wejścia tego lub innego bloku ASM.
6. Koniec.

Zauważmy, że konstrukcja ASM zaczyna się od określenia stanów automatu skończonego, tzn. inżynier od samego początku jednoznacznie definiuje wszystkie stany projektowanego układu. Następnie określane są przejścia z każdego stanu, tzn. definiuje się przejścia między stanami. W dalszej kolejności wyznaczane są wartości zmiennych wyjściowych: osobno dla automatu Mealy'ego i osobno dla automatu Moore'a. Ostatni etap, czyli połączenie ze sobą bloków ASM, ma raczej charakter formalny i pozwala jeszcze raz sprawdzić poprawność budowy ASM.

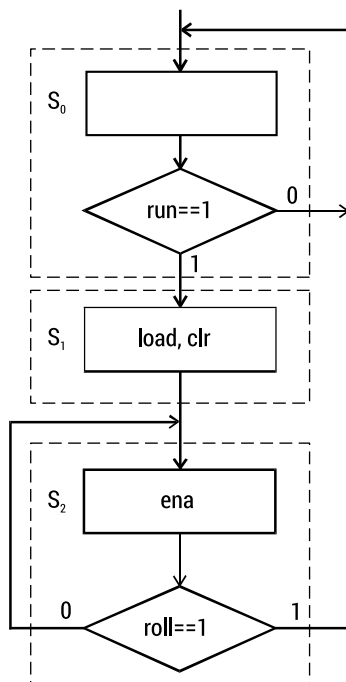
Biorąc pod uwagę fakt, że w ASM wartości wyjść (0 lub 1) są zawsze jednoznacznie określone, a także fakt, że dla każdego stanu wyznaczane są wszystkie możliwe przejścia dla wszystkich wartości zmiennych wejściowych, można twierdzić, że ASM opisują w pełni zdefiniowane automaty skończone.

Przeanalizujemy przykłady budowy ASM oddzielnie dla automatu Moore'a i dla automatu Mealy'ego. Ponieważ większość inżynierów zajmujących się projektowaniem sprzętu myśli w kategoriach automatu Moore'a, najpierw rozpatrzmy przykład budowy ASM w przypadku automatu Moore'a.

5.4. Przykład budowy ASM dla automatu Moore'a

Rozpatrzmy budowę ASM dla automatu Moore'a sterującego urządzeniem operacyjnym dla algorytmu mnożenia a (podobnego do GSA pokazanego na rysunku 4.16a). W stanie początkowym S_0 (rys. 5.4) nic nie jest tworzone, wówczas w tym bloku sprawdzana jest wartość sygnału uruchomienia run . Gdy $run = 0$, następuje powrót do stanu początkowego (w GSA jest to tylko wierzchołek oczekujący). W następnym stanie S_1 , sygnały $load$ i clr są ustawione na jeden. Ten blok ASM składa się z pojedynczego wierzchołka stanu. Następny blok ASM ze stanem S_2 implementuje cykl mnożenia.

W tym celu ustawiany jest sygnał *ena*, pozwalając rejestrom *ra* i *rb* na przesuwanie i przełączanie rejestru *rp*. Następnie sprawdzany jest sygnał *roll*; w przypadku *roll* = 0 pętla jest powtarzana, a w przypadku *roll* = 1 ASM powraca do stanu początkowego. Zbudowany ASM pokazano na rysunku 5.4. Bloki ASM na rysunku 5.4 są zaznaczone liniami przerywanymi.



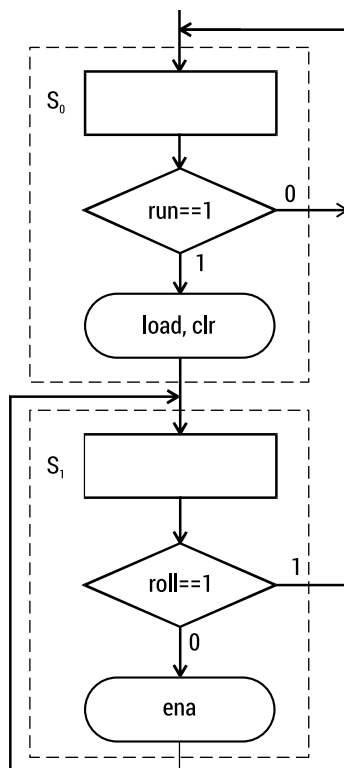
RYC. 5.4. ASM automatu Moore'a sterującego algorytmem mnożenia *a*

Porównanie GSA z rysunku 4.16a i ASM z rysunku 5.4 pokazuje, że są one bardzo podobne. Różnica polega na tym, że w ASM nie ma wierzchołków początkowych i końcowych, a także na tym, że pętla oczekiwania jest zaimplementowana poza blokiem ASM, wyraźnie wskazując, w jaki stan przechodzi automat w trybie oczekiwania. Ponadto, zamiast przechodzić do wierzchołka końcowego, ASM po wykonaniu algorytmu wyraźnie wykonuje przejście do stanu początkowego.

5.5. Przykład budowy ASM dla automatu Mealy'ego

Należy zbudować ASM dla automatu Mealy'ego, który steruje algorytmem mnożenia *a*. W bloku ASM ze stanem początkowym S_0 (rys. 5.5) sprawdzana jest wartość sygnału *run*. Przy *run* = 0 wykonywana jest pętla oczekiwania, a przy *run* = 1 ustawiane są sygnały *load* i *clr*. Następny blok ASM, odpowiadający stanowi S_1 , sprawdza

wartość sygnału *roll*. Jeśli $roll = 0$, ustawiany jest sygnał *ena* i wykonywany jest powrót do stanu S_1 , w przeciwnym razie algorytm mnożenia kończy się i następuje powrót do stanu początkowego. Kompletny ASM dla automatu Mealy'ego pokazano na rysunku 5.5.



RYS. 5.5. ASM automatu Mealy'ego sterującego algorytmem mnożenia *a*

ASM dla automatu Mealy'ego z rysunku 5.5 najbardziej odpowiada GSA z rysunku 4.17b. Zachęcamy czytelnika do porównania GSA z rysunku 4.17b i ASM z rysunku 5.5 i znalezienia ich podobieństw i różnic.

5.6. Implementacja automatów skończonych na FPGA zgodnie z ich opisem za pomocą ASM

Aby zaimplementować automat skończony na FPGA, określony za pomocą ASM, wystarczy opisać automat skończony w języku Verilog i wykonać jego syntezę za pomocą systemu Quartus. Możliwe jest opisanie automatu skończonego w języku Verilog bezpośrednio przy użyciu ASM, bez budowy tablicy przejść automatu skończonego (jak to robiono podczas syntezy AM przy użyciu GSA).

5.6.1. Implementacja automatu Moore'a

ASM z rysunku 5.4, odpowiadający automatowi Moore'a, zawiera trzy stany S_0 , S_1 i S_2 . Ze stanu S_0 możliwe jest przejście do stanu S_1 przy $run = 1$, jak również powrót do stanu S_0 przy $run = 0$. Jest tylko jedno przejście ze stanu S_1 do stanu S_2 . Przejścia ze stanu S_2 są wykonywane do stanu S_0 przy $roll = 1$ i z powrotem do stanu S_2 przy $roll = 0$.

Pojedyncze wartości sygnałów wyjściowych *load* i *clr* generowane są w stanie S_1 , a sygnału wyjściowego *ena* – w stanie S_2 . W stanie S_0 nie są generowane żadne sygnały wyjściowe.

Aby opisać rozpatrywany automat Moore'a w języku Verilog, konieczne jest określenie stanów automatu skończonego, opisanie pamięci automatu, określenie przejść między stanami, a także sygnałów wyjściowych generowanych w każdym stanie. Poniżej znajduje się możliwy opis w języku Verilog automatu Moore'a, wykonany zgodnie z ASM z rysunkiem 5.4.

```
module ASM_Moore(
input clk, reset, run, roll,      // deklaracja sygnałów wejściowych
output load, clr, ena);          // deklaracja sygnałów wyjściowych

localparam [1:0] s0=0,s1=1,s2=2; // deklaracja stanów

reg [1:0] state, next;           // deklaracja zmiennych stanów

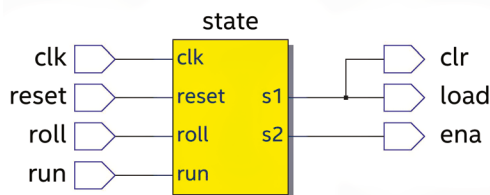
always @(posedge clk)            // opis pamięci automatu
    if(reset) state <= s0;
    else state <= next;

always @(*)                      // opis przejść między stanami
    case (state)
        s0: if(run) next = s1;
             else next = s0;
        s1: next = s2;
        s2: if(roll) next = s0;
             else next = s2;
        default: next = s0;
    endcase

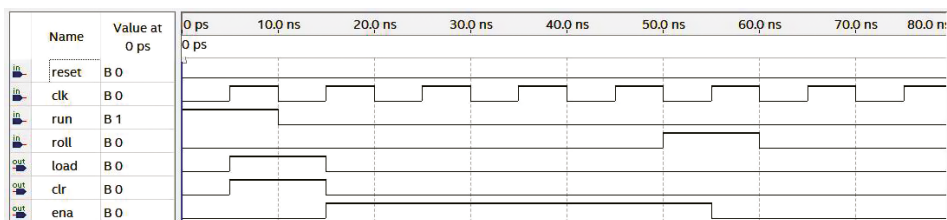
assign load = state==s1;          // opis sygnałów wyjściowych
assign clr = state==s1;
assign ena = state==s2;
endmodule
```

W powyższym kodzie zastosowano styl opisujący automat skończony Moore'a z dwoma procesami [10]. Pierwszy proces opisuje pamięć automatu, drugi proces opisuje przejścia między stanami, a wartości sygnałów wyjściowych są określone za pomocą przypisania ciągłego **assign**.

Wyniki syntezy i symulacji projektu *ASM_Moore* przedstawiono, odpowiednio, na rysunkach 5.6 i 5.7.



RYS. 5.6. Wyniki syntezy projektu *ASM_Moore* na poziomie RTL



RYS. 5.7. Wyniki symulacji projektu *ASM_Moore*

5.6.2. Implementacja automatu Mealy'ego

ASM automatu Mealy'ego na rysunku 5.5 zawiera dwa stany S_0 i S_1 . Przejścia ze stanu S_0 są możliwe z powrotem do stanu S_0 przy $run = 0$, jak również do stanu S_1 przy $run = 1$, a przy przejściu do stanu S_1 generowane są jedynkowe wartości sygnałów *load* i *clr*. Ze stanu S_1 przejścia prowadzą do stanu S_0 przy $roll = 0$ oraz z powrotem do stanu S_1 przy $roll = 1$, a przy przejściu do stanu S_1 generowana jest wartość 1 sygnału *ena*.

Poniżej znajduje się możliwy opis w języku Verilog automatu Mealy'ego.

```

module ASM_Mealy(
input clk, reset, run, roll,           // deklaracja sygnałów wejściowych
output load, clr, ena);               // deklaracja sygnałów wyjściowych

reg [2:0] y;                          // deklaracja wektora bitowego

localparam [1:0] s0=0,s1=1;          // deklaracja stanów

```

```

reg [1:0] state, next;           // deklaracja zmiennej stanu

always @(posedge clk)           // opis pamięci automatu
    if(reset) state <= s0;
    else state <= next;

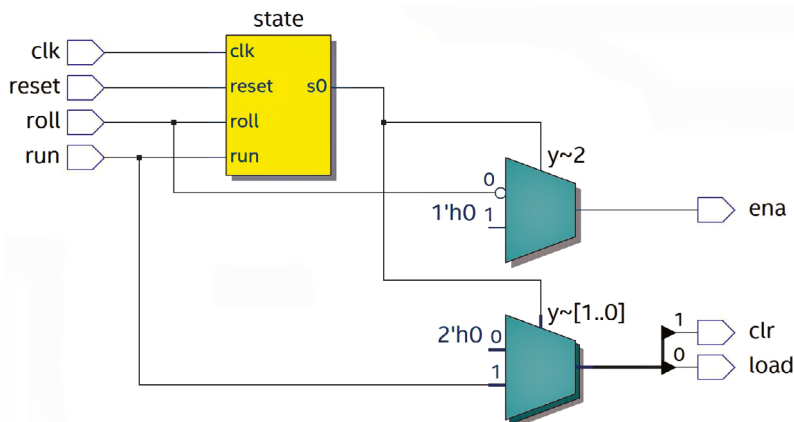
always @(*)                     // opis funkcji przejściowych i wyjściowych
    case (state)
        s0: if(run) begin next = s1; y = 3'b110; end
             else begin next = s0; y = 3'b000; end
        s1: if(roll) begin next = s0; y = 3'b000; end
             else begin next = s1; y = 3'b001; end
        default: begin next = s0; y = 3'b000; end
    endcase

assign {load, clr, ena} = y;     // określenie wartości sygnałów wyjściowych
endmodule

```

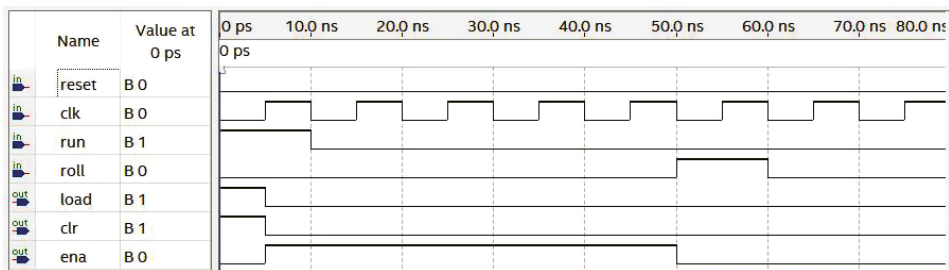
W powyższym kodzie zastosowano styl opisujący automaty skończone Mealy'ego z dwoma procesami [10]. Cechą tego opisu jest to, że wartości wszystkich sygnałów wyjściowych w każdym przejściu są definiowane jako wektor bitowy $y[]$, a wartości każdego pojedynczego sygnału wyjściowego są następnie określone przy użyciu operatora przypisania ciągłego **assign** i operacji konkatencji („{ }”).

Wyniki syntezy i symulacji projektu *ASM_Mealy* przedstawiono, odpowiednio, na rysunkach 5.8 i 5.9.



RYS. 5.8. Wyniki syntezy projektu *ASM_Mealy* na poziomie RTL

Porównanie wyników symulacji z rysunków 5.7 i 5.9 pokazuje, że wyjścia automatu Moore'a są generowane i resetowane później niż Mealy'ego. Oprócz tego, automaty Mealy'ego i Moore'a działają tak samo.



RYS. 5.9. Wyniki symulacji projektu *ASM_Mealy*

Należy zauważyć, że w przypadku implementacji na FPGA automatu skończonego, opisanego za pomocą ASM, w porównaniu z implementacją AM zgodnie z GSA, nie ma potrzeby:

- oznaczania GSA;
- identyfikowania etykiet GSA ze stanami AM;
- budowania tablicy przejść AM.

W ten sposób użycie ASM do reprezentowania automatu skończonego, w porównaniu z GSA, eliminuje szereg formalnych kroków syntezy wykonywanych ręcznie przez inżyniera, które mogą być źródłem trudnych do znalezienia błędów.

Oprócz opisu algorytmów działania automatów skończonych Mealy'ego i Moore'a, ASM można wykorzystać do innych celów. Spójrzmy na niektóre z możliwych zastosowań ASM.

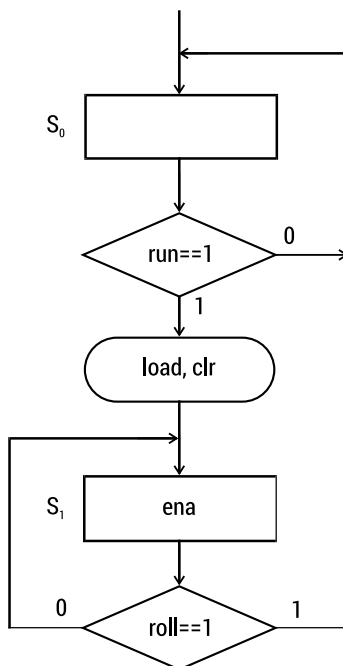
5.7. Opis za pomocą ASM wspólnych modeli automatów Mealy'ego i Moore'a

Ponieważ ASM można wykorzystać do opisanego zachowania zarówno automatów Mealy'ego, jak i Moore'a, ASM są bardzo wygodne w opisywaniu wspólnych modeli automatów Mealy'ego i Moore'a [5]. Aby użyć ASM do opisu algorytmu działania wspólnego modelu, wystarczy zapisywać sygnały wyjściowe zarówno w wierzchołkach stanów (wyjścia automatu Moore'a), jak i w wierzchołkach wyjść warunkowych (wyjścia automatu Mealy'ego).

Przypomnijmy, że sygnały wyjściowe automatów Mealy'ego i Moore'a są generowane w różnym czasie, dlatego należy upewnić się, że wyjścia automatu Mealy'ego nie przecinają się z wyjściami automatu Moore'a. Innymi słowy, jeśli jakaś zmienna wyjściowa jest zapisana w wierzchołku stanu (w prostokącie) ASM, to ona nie powinna występować w wierzchołkach wyjść warunkowych (w owalach) i odwrotnie.

Jako przykład rozpatrzmy ASM dla wspólnego modelu automatu, sterującego urządzeniem operacyjnym układu mnożącego według algorytmu *a*, pokazany na rysunku 5.10. W tym przypadku blok ASM stanu S_0 odpowiada automatu

Mealy'ego, a blok ASM stanu S_1 odpowiada automatu Moore'a, tzn. wyjścia *load* i *clr* są wyjściami automatu Mealy'ego, a sygnał wyjściowy *ena* jest wyjściem automatu Moore'a.



RYS. 5.10. ASM wspólnego modelu automatów Mealy'ego i Moore'a

Poniżej znajduje się opis w języku Verilog wspólnego modelu automatów Mealy'ego i Moore'a według ASM z rysunku 5.10.

```

module ASM_Mealy_Moore(
input clk, reset, run, roll,      // deklaracja sygnałów wejściowych
output load, clr, ena);          // deklaracja sygnałów wyjściowych

reg [2:0] y;                      // deklaracja wektora bitowego

localparam [1:0] s0=0,s1=1;      // deklaracja stanów

reg [1:0] state, next;           // deklaracja zmiennych stanów

always @(posedge clk)            // opis pamięci automatu
    if(reset) state <= s0;
    else state <= next;
  
```



```

always @(*)                                // opis funkcji przejściowych i wyjściowych
case (state)
    // opis stanu automatu Mealy'ego
    s0: if(run)          begin next = s1; y = 3'b110; end
        else          begin next = s0; y = 3'b000; end

    // opis stanu automatu Moore'a
    s1: begin
        y = 3'b001;
        if(roll)      next = s0;
        else          next = s1;
    end
    default:      begin next = s0; y = 3'b000; end
endcase

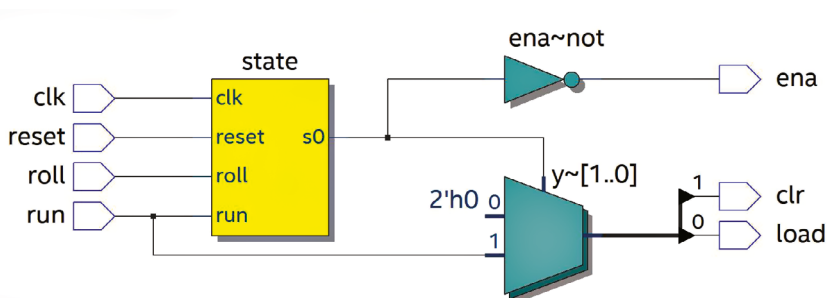
assign {load, clr, ena} = y;              // określenie wartości wyjściowych
endmodule

```

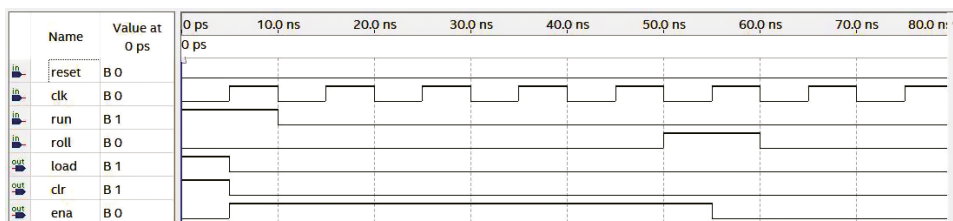
Podany kod odpowiada stylowi opisu automatów skończonych z dwoma procesami, w których opisy funkcji przejściowych i funkcji wyjściowych są łączone w jeden proces [10]. Opisując stan *s0* automatu Mealy'ego, dla każdego warunku przejścia określany jest następny stan i wektor wyjściowy utworzony przy tym przejściu. Opisując stan *s1* automatu Moore'a, najpierw określany jest wektor wyjściowy utworzony w tym stanie, a następnie opisywane są przejścia z tego stanu.

Konkretne wartości każdego sygnału wyjściowego są definiowane za pomocą oddzielnej instrukcji przypisania ciągłego **assign** i operacji konkatencji.

Wyniki syntezy i symulacji projektu *ASM_Mealy_Moore* przedstawiono na rysunkach 5.11 i 5.12.



RYŚ. 5.11. Wyniki syntezy projektu *ASM_Mealy_Moore* na poziomie RTL



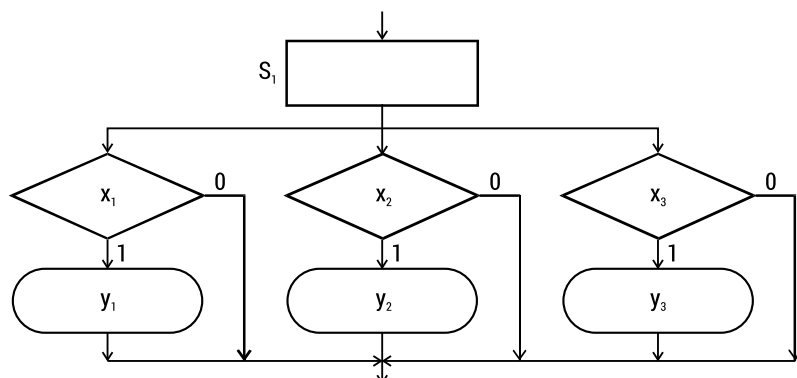
RYS. 5.12. Wyniki symulacji projektu *ASM_Mealy_Moore*

Porównanie diagramów czasowych na rysunkach 5.7 i 5.9 z wykresem czasowym na rysunku 5.12 pokazuje, że we wspólnym modelu automatów, sygnały wyjściowe automatu Mealy'ego *load* i *clr* są ustawiane i resetowane w taki sam sposób, jak w automacie Mealy'ego, jednak sygnał wyjściowy *ena* automatu Moore'a jest ustawiany, jak w automacie Mealy'ego i jest resetowany, jak w automacie Moore'a. Dlatego należy zachować ostrożność podczas używania wspólnych modeli automatów Mealy'ego i Moore'a.

5.8. Opis procesów równoległych za pomocą ASM

Jedną z zasad tworzenia stanów ASM brzmi: każde wyjście dowolnego wierzchołka ASM może być połączone tylko z jednym wejściem innego wierzchołka. Zasada ta została wprowadzona po to, aby rozgałęzienie algorytmu było możliwe tylko przy wierzchołkach warunkowych ASM. Innymi słowy, algorytm opisany za pomocą ASM jest zawsze wykonywany sekwencyjnie: wierzchołki ASM są sprawdzane sekwencyjnie, jeśli wierzchołek jest warunkowy, to następuje rozgałęzienie, które jest określone przez wartość warunku, jeśli wierzchołek jest wierzchołkiem stanu lub wierzchołkiem wyjścia warunkowego, następuje przejście do następnego wierzchołka.

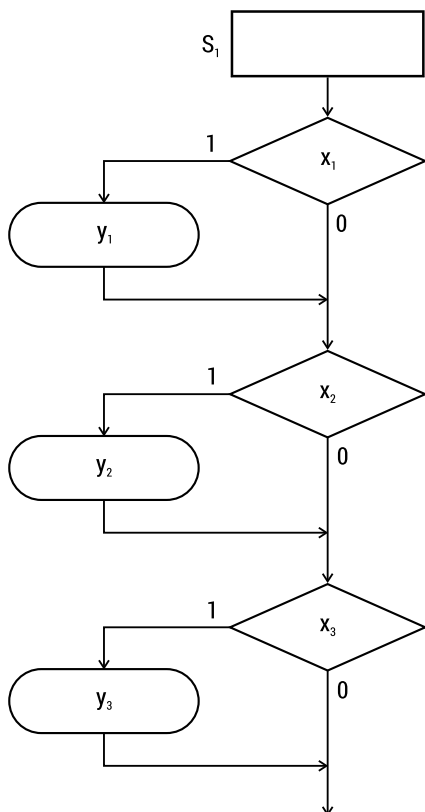
Powstaje pytanie: jak opisać procesy równoległe za pomocą ASM. W niektórych przypadkach jest to możliwe. Rozpatrzmy blok ASM pokazany na rysunku 5.13.



RYS. 5.13. Opis procesu równoległego

Powyższa zasada jest tutaj naruszona: wyjście wierzchołka stanu S_1 jest połączone z wejściami trzech wierzchołków warunkowych, które są początkiem równoległych gałęzi algorytmu. W każdej gałęzi sprawdzany jest warunek logiczny x_1 , x_2 lub x_3 i w zależności od jego wartości, odpowiedni sygnał wyjściowy jest ustawiany lub nie. Wszystkie gałęzie algorytmu działają równoległe i niezależnie od siebie.

W takim przypadku równoległe działanie algorytmu można opisać sekwencyjnie, przestrzegając wszystkich zasad budowy ASM, jak pokazano na rysunku 5.14.



RYS. 5.14. Opis równoległego procesu za pomocą ASM

Tutaj każdy warunek jest testowany sekwencyjnie i jeśli jest on spełniony, ustawiany jest odpowiedni sygnał wyjściowy. Ponieważ blok ASM jest zawsze wykonywany w jednym cyklu zegara, implementacja ASM na rysunku 5.14 będzie odpowiadać (wygląd dla zewnętrznego obserwatora) równoległemu działaniu algorytmu.

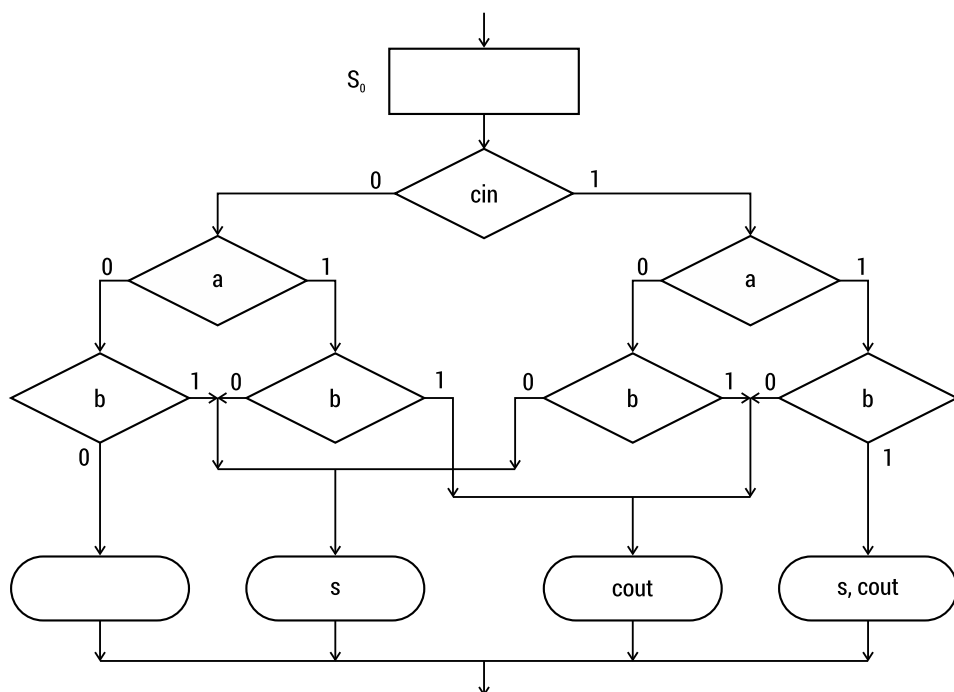
Czytelnik jest proszony o samodzielne opisanie w języku Verilog fragmentu ASM z rysunku 5.14 w postaci automatu skończonego, wykonanie jego implementacji na FPGA i sprawdzenie jego działania.

5.9. Opis układów kombinacyjnych za pomocą ASM

Ponieważ układy kombinacyjne mogą również pracować jako układy sterujące (na przykład w procesorach jednocyklowych), pojawia się naturalne pytanie: czy ASM można wykorzystać do opisu układów kombinacyjnych. Z drugiej strony, każdy układ kombinacyjny można rozpatrywać jako jednostanowy automat skończony. ASM mają za zadanie opisywać działanie automatów skończonych niezależnie od liczby ich stanów. Dlatego można założyć, że ASM można również wykorzystać do opisu układów kombinacyjnych. Należy zauważyć, że układ kombinacyjny jest zawsze opisywany tylko jednym blokiem ASM.

Jako przykład rozpatrzmy użycie ASM do opisu działania sumatora jednobitowego i rozpatrzmy dwie opcje reprezentacji sumatora: w postaci tablicy prawdy oraz w postaci równań logicznych.

ASM odpowiadający jednobitowej tablicy prawdy sumatora pokazano na rysunku 5.15. Wszystkie możliwe kombinacje zmiennych wejściowych są przedstawione tutaj w wierzchołkach warunkowych. Wartości zmiennych wyjściowych są zapisane w owalach, tak jak w przypadku automatu Mealy'ego. Aby określić zerowy wektor wyjściowy, używany jest owal, w którym nic nie jest zapisywane.



RYS. 5.15. ASM jednobitowego sumatora, odpowiadający tablicy prawdy

ASM pokazany na rysunku 5.15 składa się tylko z jednego bloku ASM, tzn. zawiera tylko jeden stan, który nie wymaga żadnych bitów kodu do kodowania, ponieważ

$\text{intlog}_2 1 = 0$. Z tego powodu, do opisu ASM w języku Verilog można użyć dowolnego stylu opisywania układów kombinacyjnych. Poniżej znajduje się jeden z możliwych sposobów opisu ASM z rysunku 5.15.

```

module add_1_true_table(
input a,b,cin,
output s,cout);

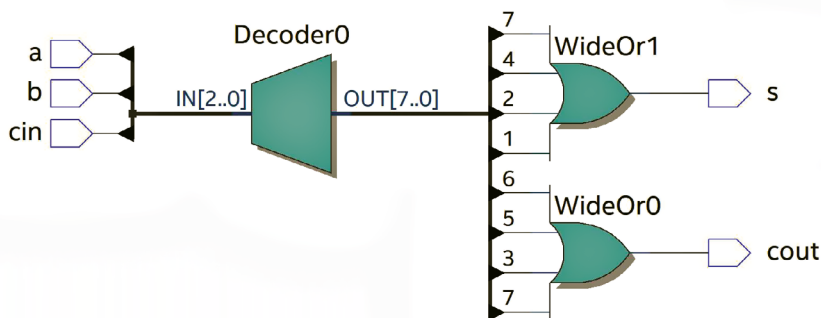
reg [1:0] y;

always @(*)
    case ({cin,a,b})
        3'b000: y=2'b00;
        3'b001: y=2'b01;
        3'b010: y=2'b01;
        3'b011: y=2'b10;
        3'b100: y=2'b01;
        3'b101: y=2'b10;
        3'b110: y=2'b10;
        3'b111: y=2'b11;
    endcase

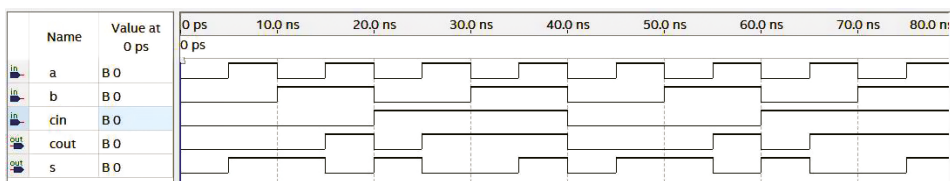
assign {cout,s} = y;
endmodule

```

Wyniki syntezy i symulacji projektu *add_1_true_table* przedstawiono na rysunkach 5.16 i 5.17.



RYS. 5.16. Wyniki syntezy projektu *add_1_true_table* na poziomie RTL

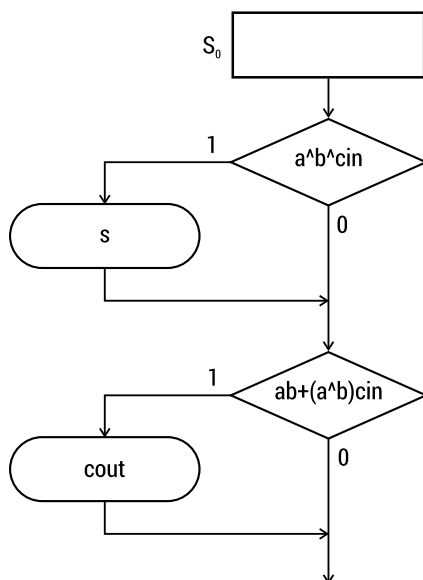


RYS. 5.17. Wyniki symulacji projektu *add_1_true_table*

Rysunek 5.18 przedstawia ASM jednobitowego sumatora, który odpowiada następującym równaniom logicznym:

$$s = a \oplus b \oplus \text{cin};$$

$$\text{cout} = a \cdot b + (a \oplus b) \cdot \text{cin}.$$



RYS. 5.18. ASM jednobitowego sumatora odpowiadający równaniom logicznym

Opis ASM z rysunku 5.18 w języku Verilog może wyglądać następująco:

```

module add_1_equations(
input a,b,cin,
output reg s,cout);

```

```

always @(*)
begin

```

```

    if (a ^ b ^ cin)          s = 1'b1;
    else                     s = 1'b0;

```

```

    if (a & b | (a ^ b) & cin)    cout = 1'b1;
    else                          cout = 1'b0;

```

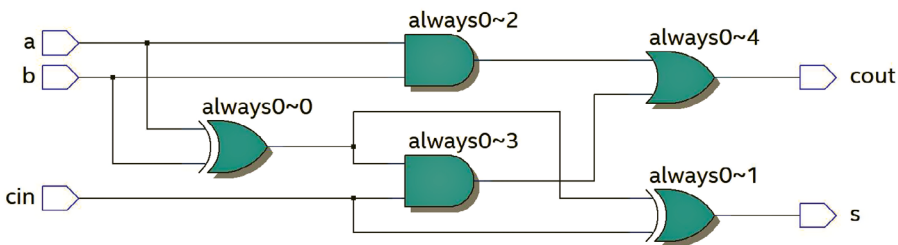
```

end
endmodule

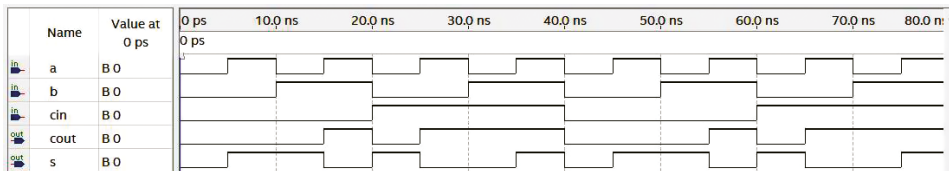
```

W powyższym kodzie instrukcje **if** są wykonywane sekwencyjnie, ponieważ w bloku **always** instrukcje proceduralne są wykonywane sekwencyjnie w takiej kolejności, w jakiej są zapisane w kodzie źródłowym projektu, co w pełni odpowiada algorytmowi przedstawionemu przez ASM z rysunku 5.18. Z drugiej strony, ASM z rysunku 5.18 opisuje dwa równoległe procesy (patrz sekcja 5.1.8). Dlatego można oczekiwać, że wyjścia *s* i *cout* będą obliczane równoległe.

Wyniki syntezy i symulacji projektu *add_1_equation* przedstawiono na rysunkach 5.19 i 5.20.



RYS. 5.19. Wyniki syntezy projektu *add_1_equation*



RYS. 5.20. Wyniki symulacji projektu *add_1_equation*

Rysunek 5.19 pokazuje, że rzeczywiście wyjścia *s* i *cout* są obliczane równoległe.

Wadą języka ASM jest fakt, że nie pozwala on na opisywanie nie w pełni określonych automatów skończonych.

5.10. Wnioski

Sieć działań *algorytmicznych automatów skończonych* (algorithmic state machine – ASM) jest graficznym językiem opisującym działanie automatów skończonych. Główną zaletą ASM w porównaniu z GSA jest to, że w języku Verilog automat skończony można bezpośrednio opisać poprzez jego reprezentację w postaci ASM, bez użycia oznaczania

GSA, identyfikowania etykiet GSA ze stanami automatu skończonego i budowania tablicy przejść.

ASM to ukierunkowany graf zawierający trzy typy wierzchołków:

- prostokąty – *wierzchołki stanów* (state box);
- romby – *wierzchołki warunkowe* (decision box);
- owale – *wierzchołki wyjść warunkowych* (conditional output box).

Wierzchołek stanu ASM określa stan automatu. Wierzchołek warunkowy ASM w pełni odpowiada wierzchołkowi warunkowemu GSA. Wierzchołek wyjścia warunkowego ASM odpowiada wierzchołkowi operacyjnemu GSA dla automatu Mealy'ego.

Przy opisie automatu Moore'a, w ASM nie ma wierzchołków wyjść warunkowych (owale), a przy opisie automatu Mealy'ego, nic nie jest zapisywane w wierzchołkach stanów.

Głównym blokiem konstrukcyjnym sieci działań ASM jest *blok ASM* (ASM block). Blok ASM zawiera tylko jeden węzeł stanu i może mieć wiele wierzchołków warunkowych i wierzchołków wyjść warunkowych. Blok ASM opisuje zachowanie automatu w jednym stanie podczas jednego cyklu synchronizacji. Sieć działań ASM składa się z połączonych ze sobą bloków ASM.

Zasady budowy sieci działań ASM:

- każde wyjście dowolnego wierzchołka ASM można podłączyć tylko do jednego wejścia innego wierzchołka, tzn. rozgałęzienie algorytmu jest możliwe tylko na wierzchołkach warunkowych;
- sprzężenia zwrotne wewnątrz bloku ASM są zabronione.

Cechy szczególne ASM są następujące: za pomocą bloków ASM określa się stany wewnętrzne automatu; przejście z jednego bloku ASM do drugiego odbywa się zawsze w jednym cyklu zegarowym, tzn. algorytm działania automatu jest bezpośrednio powiązany z czasem działania automatu, zgodnie z ASM zawsze można śledzić, ile cykli zegara zajmie ten lub inny fragment algorytmu. ASM zachowuje przejrzystość GSA i schematów blokowych. W ASM od początku określony jest typ automatu: Mealy'ego lub Moore'a. ASM opisują w pełni określone automaty skończone.

Metodyka budowy ASM jest następująca: określa się stany automatu skończonego, następnie dla każdego stanu budowany jest blok ASM. Dla automatu Moore'a zmienne wyjściowe, które przyjmują jedynkową wartość w danym stanie są zapisywane w wierzchołkach stanów, dla automatu Mealy'ego w wierzchołkach wyjść warunkowych zapisywane są zmienne wyjściowe, przyjmując jedynkową wartość przy danym przejściu. Bloki ASM łączą się ze sobą zgodnie z algorytmem działania urządzenia.

Aby zaimplementować na FPGA automat skończony zdefiniowany za pomocą ASM, wystarczy opisać automat skończony w języku Verilog i syntetyzować go za pomocą używanego narzędzia projektowego. Opisać automat skończony w języku Verilog można bezpośrednio na podstawie ASM, bez użycia żadnych pośrednich etapów.

Podczas implementacji na FPGA automatu skończonego zgodnie z ASM, w porównaniu z implementacją AM zgodnie z GSA, nie ma potrzeby stosowania oznaczania GSA; identyfikowania etykiet GSA ze stanami AM; budowy tablicy przejść AM. Używanie ASM do reprezentowania automatu skończonego, w porównaniu z GSA, eliminuje szereg formalnych kroków syntezy wykonywanych ręcznie przez inżyniera, które mogą być źródłem trudnych do znalezienia błędów.

Przy opisywaniu wspólnych modeli automatów Mealy'ego i Moore'a za pomocą ASM, sygnały wyjściowe są zapisywane zarówno w wierzchołkach stanów (wyjścia automatu Moore'a), jak i w wierzchołkach wyjść warunkowych (wyjścia automatu Mealy'ego). W takim przypadku dane wyjściowe automatu Mealy'ego nie mogą przecinać się z wyjściami automatu Moore'a. Innymi słowy, jeśli jakaś zmienna wyjściowa jest zapisana w wierzchołku stanów (w prostokącie) ASM, to nie powinna występować w wierzchołkach wyjść warunkowych (owale) i odwrotnie.

Należy zachować ostrożność podczas wykorzystywania wspólnych modeli automatów Mealy'ego i Moore'a, ponieważ wyjścia automatu Mealy'ego są ustawiane i resetowane w innym czasie niż wyjścia automatu Moore'a.

Za pomocą ASM można opisywać równoległe algorytmy działania układów cyfrowych.

ASM mogą być również wykorzystywane do opisu układów kombinacyjnych. W takiej sytuacji ASM zawiera tylko jeden stan.

6. Automaty skończone ze ścieżką przetwarzania danych (FSMD)

W niniejszym rozdziale przedstawiono nową metodykę projektowania układów cyfrowych na podstawie sieci działań *algorytmicznego automatu skończonego ze ścieżką przetwarzania danych* (algorithmic state machine with datapath – ASMD) oraz *automatu skończonego ze ścieżką przetwarzania danych* (finite state machine with datapath – FSMD), zwaną *metodyką ASMD-FSMD*. Opisano również język ASMD. W przeciwieństwie do tradycyjnego podejścia, metodyka ASMD-FSMD pozwala na opisanie całego układu od razu w języku Verilog bez dzielenia go na automaty: operacyjny (wykonawczy) i sterujący, co znacznie skraca czas i koszt projektowania, a także zwiększa niezawodność opracowywanych projektów.

Metodykę ASMD-FSMD zademonstrowano na przykładzie realizacji układu mnożącego synchronicznego, odpowiadającego algorytmowi mnożenia a [10]. Różne podejścia metodyki ASMD-FSMD porównano między sobą oraz z tradycyjnym podejściem do projektowania układów cyfrowych.

Na podstawie przeprowadzonych badań sformułowano zalecenia dotyczące efektywnego stosowania metodyki ASMD-FSMD.

6.1. Sieć działań ASMD i automaty FSMD

Tradycyjnie, urządzenie cyfrowe lub system cyfrowy jest zwykle przedstawiany jako układ operacyjny (wykonawczy) i układ sterujący. Układy operacyjny i sterowania są zwykle projektowane oddzielnie: układ wykonawczy jako zbiór komponentów (standardowe bloki funkcjonalne), a układ sterowania jako automat skończony. Może pojawić się naturalne pytanie, czy można połączyć te dwa procesy razem.

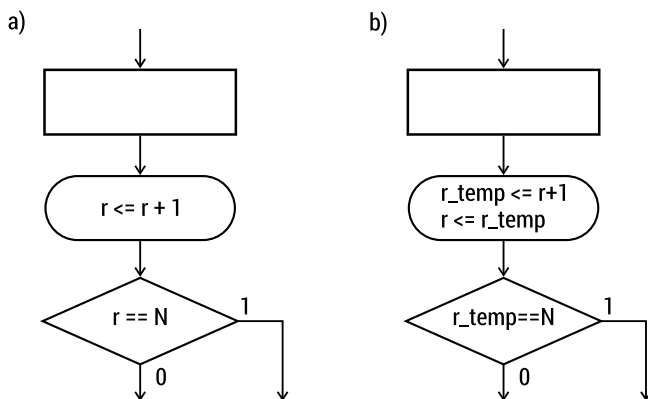
Połączenie procesów projektowania układu operacyjnego i układu sterującego ułatwia fakt, że obie klasy tych układów opisane są tym samym językiem Verilog. Ponadto urządzenie operacyjne i urządzenie sterowania są generalnie automatami skończonymi. Jednak metodyka projektowania automatów skończonych na FPGA [10] nie jest w pełni odpowiednia do projektowania układów operacyjnych, ponieważ koncentruje się na projektowaniu głównie układów sterowania.

Następnym problemem jest to, jak opisać połączone działanie układu operacyjnego i układu sterującego. Grafy automatów i tablice przejść automatów nie są do tego odpowiednie. Odpowiedzią na ten problem jest sieć działań nazywana *algorytmicznym automatem ze ścieżką przetwarzania danych* (ASM with datapath – ASMD). Automat skończony realizowany przez sieć działań ASMD jest nazywany *automatem skończonym ze ścieżką przetwarzania danych* (FSM with datapath – FSMDD).

6.2. Język ASMD

Sieć działań ASMD to zwykła sieć ASM, w której w prostokątach i owalach można zapisywać dowolne operacje na rejestrach, które są dozwolone w języku Verilog, a w wierzchołkach warunkowych można sprawdzić dowolne wyrażenia logiczne języka Verilog. Sieć działań ASMD, podobnie jak ASM, składa się z bloków. Czynności opisane w bloku ASMD wykonywane są w jednym cyklu zegarowym.

Należy jednak mieć na uwadze, że wszystkie operacje na rejestrach opisanych wewnątrz bloku ASMD będą wykonywane przy wyjściu z bloku, tzn. na kolejnym dodatnim zboczku sygnału synchronizacji. Ta właściwość ASMD może być źródłem subtelnych błędów, gdy zmieniona wartość rejestru jest sprawdzana w tym samym bloku ASMD (rys. 6.1).



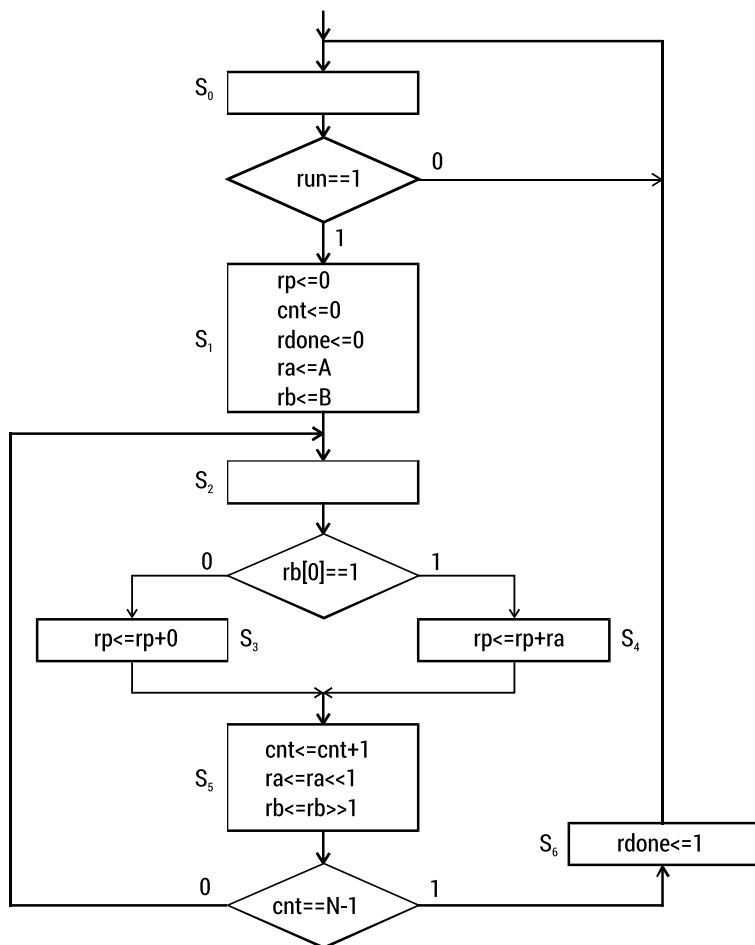
RYŚ. 6.1. Sprawdzenie zmienionej wartości rejestru: a – błędna, b – poprawna

Na rysunku 6.1a w bloku ASMD najpierw zwiększa się wartość rejestru r , a następnie sprawdzana jest jego zawartość, która zmieni się dopiero po wyjściu z bloku. Aby uniknąć takiego błędu, można wprowadzić zmienną pośrednią r_temp typu **wire**, w której będzie przechowywana tymczasowa wartość rejestru r (rys. 6.1b).

Należy zauważyć, że koncepcja FSMDD jest nową metodyką projektowania układów i systemów cyfrowych, która znacznie upraszcza i przyspiesza proces projektowania różnych układów cyfrowych.

6.3. Wykorzystanie FSMD Moore'a do implementacji algorytmu mnożenia a

Jako przykład rozpatrzmy, jak można wykorzystać sieć działań ASMD do opisanego algorytmu mnożenia a [10] i zaimplementować go jako pojedynczy automat FSMD Moore'a. ASMD opisująca algorytm mnożenia a została pokazana na rysunku 6.2.



RYS. 6.2. ASMD opisująca algorytm mnożenia a jako FSMD Moore'a

W stanie S_0 oczekujemy na sygnał uruchomienia run , który rozpoczyna proces mnożenia. W stanie S_1 rejestr iloczynu rp , licznik cnt i przerzutnik $rdone$ są zerowane. Dodatkowo w stanie S_1 ładowane są początkowe wartości rejestrów ra i rb .

Następny stan S_2 jest punktem wejścia w cykl procesu mnożenia. Jeden cykl procesu mnożenia odpowiada wytworzeniu iloczynu cząstkowego, dodaniu go do wyniku i wykonaniu niezbędnych przesunięć rejestrów, a także zwiększeniu wartości licznika.

W stanie S_2 , w zależności od wartości bitu $rb[0]$, konieczne jest dodanie do rejestru rp albo wartości zerowej, albo wartości rejestru ra . Ponieważ jednak ASMD opisuje automat typu Moore'a, czynności te mogą być wykonywane tylko w oddzielnych stanach S_3 i S_4 .

Po stanie S_3 lub S_4 następuje stan S_5 , w którym jest zwiększany licznik cnt , a wartości w rejestrach ra i rb są przesuwane. Dodatkowo w stanie S_5 sprawdzana jest wartość licznika cnt . Jeśli wartość cnt jest równa $N-1$, następuje przejście do stanu S_6 , w przeciwnym razie do stanu S_2 , aby wykonać nową iterację procesu mnożenia.

W stanie S_6 przerzutnik $rdone$, który wskazuje na zakończenie procesu mnożenia, jest ustawiony na jeden i następuje przejście do stanu początkowego S_0 .

W taki sposób ASMD dla algorytmu mnożenia a zaimplementowanego za pomocą automatu typu Moore'a zawiera 7 stanów. Zwróć uwagę, że automat sterujący zbudowany zgodnie z ASM z rysunku 5.4 ma tylko trzy stany.

Opis FSM Moore'a w języku Verilog, co odpowiada ASMD na rysunku 6.2, może wyglądać następująco:

```

module mult_FSM Moore
#(parameter N=4)
(input clk, reset, run,           // sygnały sterujące
input [N-1:0] a,b,              // słowa wejściowe
output [2*N-1:0] p,             // iloczyn
output done);                  // flaga o zakończeniu mnożenia

reg [2*N-1:0] ra,rp;            // deklaracja rejestrów
reg [N-1:0] rb;
reg rdone;

reg [N_cnt-1:0] cnt;            // licznik

localparam N_cnt=clogb2(N-1); // określenie wielkości licznika

function integer clogb2(input [N-1:0] v); // funkcja stała
    for (clogb2 = 0; v > 0; clogb2 = clogb2 + 1)
        v = v >> 1;
endfunction

localparam [2:0]                // deklaracja stanów
s0=0,s1=1,s2=2,s3=3,s4=4,s5=5,s6=6;

reg [2:0] state;                // zmienna stanu

always @(posedge clk)           // procesu mnożenia
    if(reset) state <= s0;       // stan resetowania

```

```

else // rozpoczęcie cyklu mnożenia
  case (state)
    s0: if(run) state <= s1;
        else state <= s0;
    s1: begin
        rp <= 0; cnt <= 0; rdone <= 0;
        ra <= {{N{1'b0}},a};
        rb <= b;
        state <= s2;

        end
    s2: if(rb[0]) state <= s4;
        else state <= s3;
    s3: begin
        rp <= rp + {2*N{1'b0}};
        state <= s5;

        end
    s4: begin
        rp <= rp + ra;
        state <= s5;

        end
    s5: begin
        cnt <= cnt + 1'b1;
        rb <= rb >> 1;
        ra <= ra << 1;
        if (cnt == N-1) state <= s6;
        else state <= s2;

        end
    s6: begin
        rdone <= 1'b1;
        state <= s0;

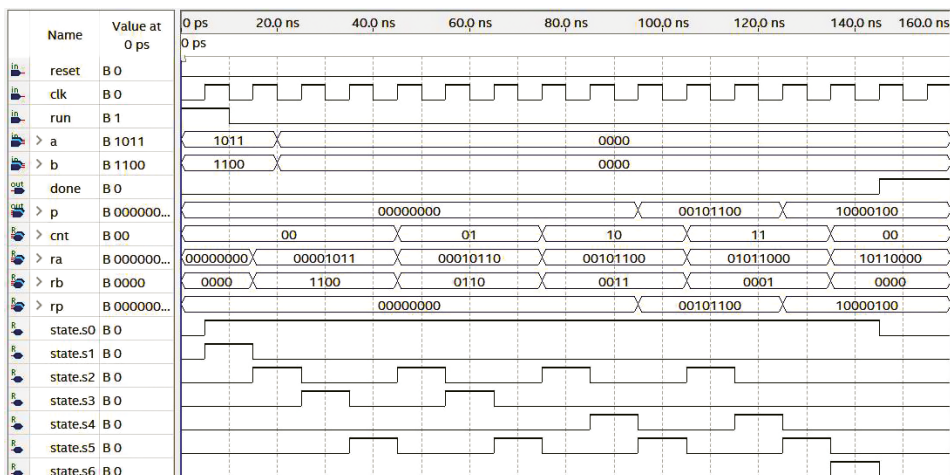
        end
    endcase

assign p = rp; // określenie wartości wyjściowych
assign done = rdone;
endmodule

```

W powyższym kodzie do określenia wielkości licznika używana jest funkcja stała *clogb2* [6], która oblicza najmniejszą liczbę całkowitą większą lub równą logarytmowi binarnemu v . Do opisu automatu Moore'a jest używany styl opisu z jednym procesem. W takim przypadku kolejny stan jest określany w każdym stanie, a wszystkie działania wykonywane są na rejestrach zgodnie z ASMD z rysunku 6.2.

Wyniki syntezy i symulacji projektu *mult_FSMD_Moore* przedstawiono na rysunkach 6.3 i 6.4.



RYS. 6.4. Wyniki symulacji projektu *mult_FSM_D_Moore*

Analiza rysunku 6.4 pokazuje, że wynik mnożenia na wyjściu *p* jest generowany w 13 cyklu zegarowym, a sygnał wykonania *done* ustawiany jest w 15 cyklu zegarowym.

Czytelnik jest proszony, aby samodzielnie obliczyć liczbę cykli zegara potrzebnych do mnożenia 4-bitowych słów *A* i *B* dla ASMD z rysunku 6.2 i porównać wyniki z wykresem czasowym z rysunku 6.4.

Z wykresu czasowego pokazanego na rysunku 6.4 widać, że automat FSMD zaczyna pracę od stanu S_0 , w którym nic się nie dzieje. Po ustawieniu sygnału *run* automat FSMD przechodzi w stan S_1 , w którym następuje inicjalizacja rejestrów. Jeden cykl mnożenia jest wykonywany w trzech cyklach zegarowych. W tym przypadku automat FSMD przechodzi przez stany $S_2 - S_3 - S_5$ lub $S_2 - S_4 - S_5$ w zależności od wartości bitu *rb*[0]. Po zakończeniu procesu mnożenia automat przechodzi w stan S_6 , w którym wartość przerzutnika *rdone* ustawiana jest na jeden. Wyjściowy sygnał *done* pojawia się na wyjściu tylko w następnym takcie synchronizacji.

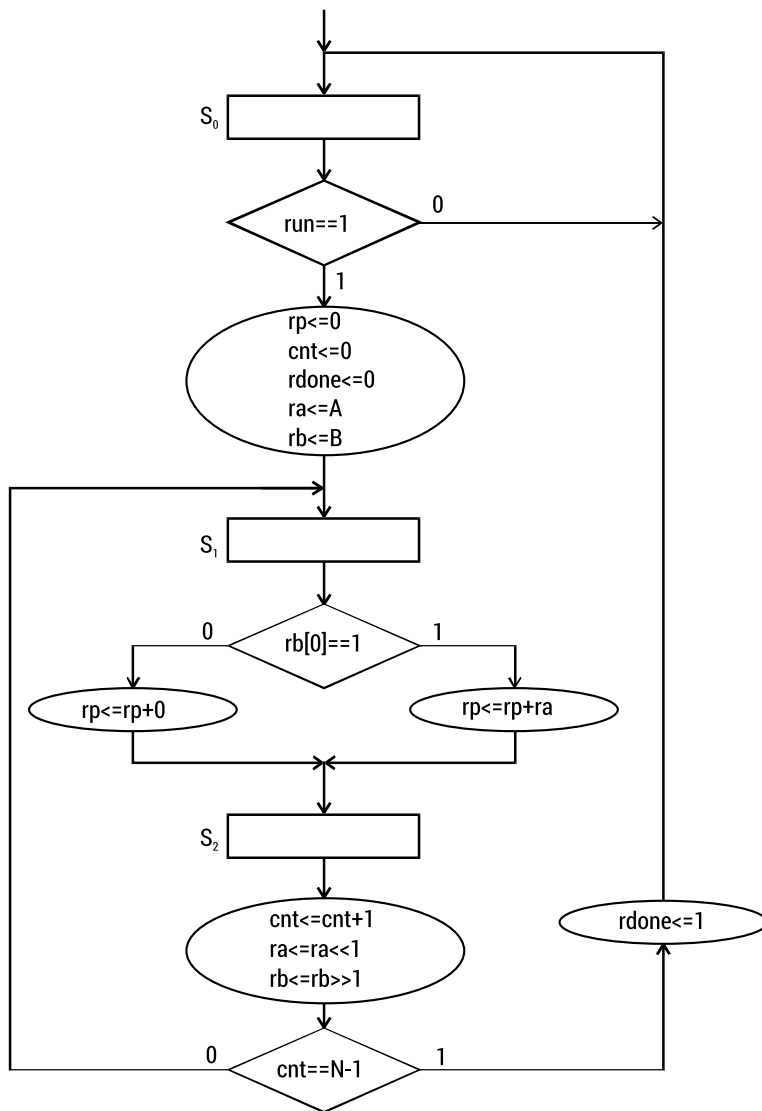
W ten sposób FSMD dla automatu Moore'a wykonuje mnożenie *N*-bitowych liczb binarnych za n_{Moore} taktów zegara synchronizacji, gdzie:

$$n_{Moore} = 3 * N + 3. \quad (6.1)$$

6.4. Wykorzystanie FSMD Mealy'ego do implementacji algorytmu mnożenia *a*

Rysunek 6.5 przedstawia ASMD do implementacji algorytmu mnożenia *a* przy użyciu FSMD Mealy'ego.

W stanie początkowym S_0 automat oczekuje na nadejście jedynkowej wartości sygnału *run*. Po ustawieniu sygnału *run* na 1, wykonywana jest niezbędna inicjalizacja wszystkich rejestrów układu mnożącego.



RYS. 6.5. ASMD do implementacji algorytmu mnożenia *a* przy użyciu automatu FSMD typu Mealy'ego

Następny stan S_1 jest punktem wejścia w cykl procesu mnożenia. W stanie S_1 sprawdzana jest wartość bitu $rb[0]$ i zależnie od jego wartości, do zawartości rejestru rp dodawana jest wartość zerowa, albo zawartość rejestru ra . Zauważmy, że dwa oddzielne stany nie są tutaj wymagane, jak w FSMD Moore'a (S_3 i S_4 na rys. 6.2).

W stanie S_2 wykonywane są niezbędne przesunięcia zawartości rejestrów ra i rb oraz inkrementacja licznika. Następnie w stanie S_2 sprawdzana jest poprzednia wartość licznika, jeśli jest równa $N-1$, to algorytm mnożenia kończy się i automat przechodzi

do stanu S_0 z ustawionym rejestrem *rdone*. W przeciwnym wypadku cykl procesu mnożenia jest powtarzany.

Opis automatu FSMD Mealy’ego w języku Verilog, odpowiadający ASMD z rysunku 6.5, może wyglądać następująco:

```

module mult_FSMD_Mealy
#(parameter N=4)
(input clk, reset, run,           // sygnały sterujące
input [N-1:0] a,b,              // słowa wejściowe
output [2*N-1:0] p,             // iloczyn
output done);                  // flaga o zakończeniu mnożenia

reg [2*N-1:0] ra,rp;            // deklaracja rejestrów
reg [N-1:0] rb;
reg rdone;

reg [N_cnt-1:0] cnt;           // licznik

localparam N_cnt=clogb2(N-1); // obliczanie rozmiaru licznika

function integer clogb2(input [N-1:0] v); // funkcja stała
    for (clogb2 = 0; v > 0; clogb2 = clogb2 + 1)
        v = v >> 1;
endfunction

localparam [1:0] s0=0,s1=1,s2=2; // deklaracja stanów

reg [1:0] state;               // zmienna stanu

always @(posedge clk)          // rozpoczęcie procesu mnożenia
    if(reset) state <= s0;
    else
        case (state)
        s0: if(run)
            begin
                rp <= 0; cnt <= 0; rdone <= 0;
                ra <= {{N{1'b0}},a};
                rb <= b;
                state <= s1;
            end

```

```

        else
            state <= s0;
s1:    if(rb[0])
        begin
            rp <= rp + ra;
            state <= s2;
        end
    else
        begin
            rp <= rp + {2*N{1'b0}};
            state <= s2;
        end
s2:    begin
            cnt <= cnt + 1'b1;
            rb <= rb >> 1;
            ra <= ra << 1;
            if (cnt == N-1)
                begin
                    rdone <= 1'b1;
                    state <= s0;
                end
            else
                state <= s1;
            end
    default: state <= s0;
endcase

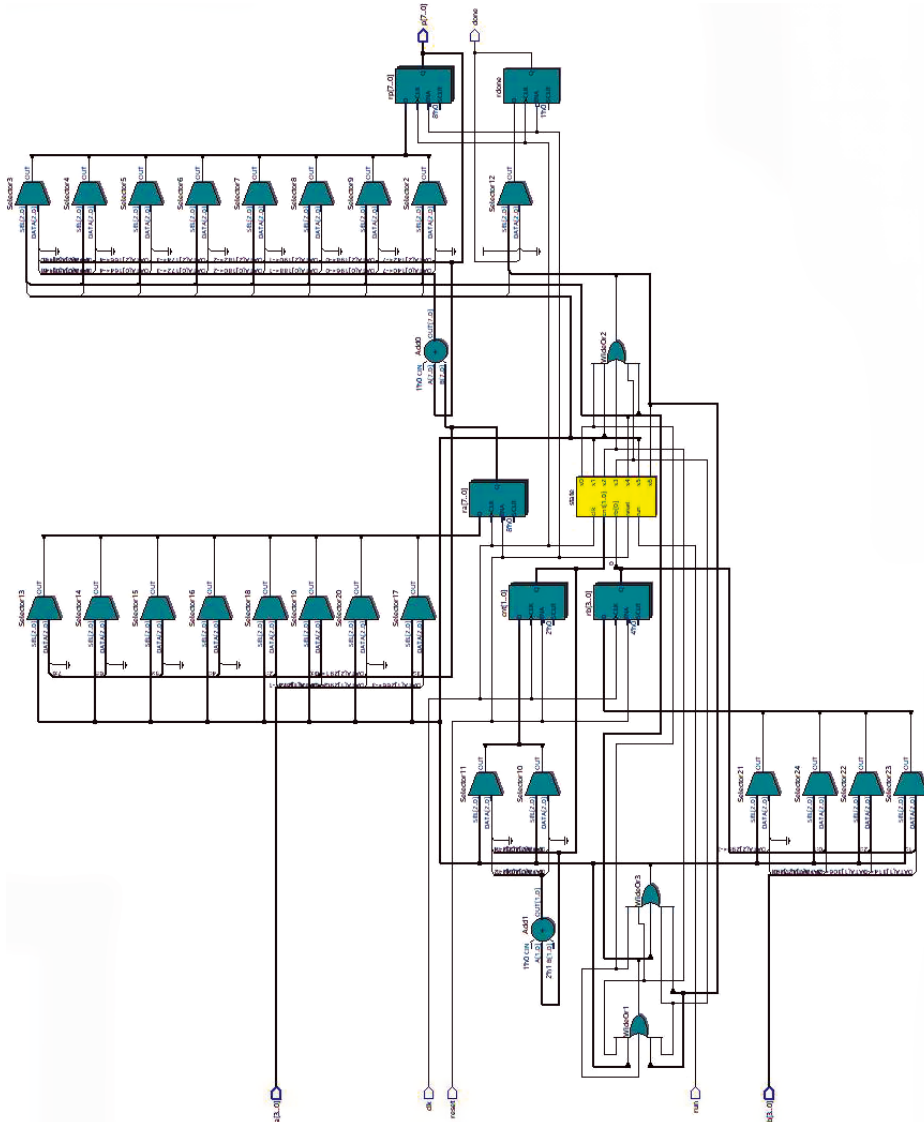
```

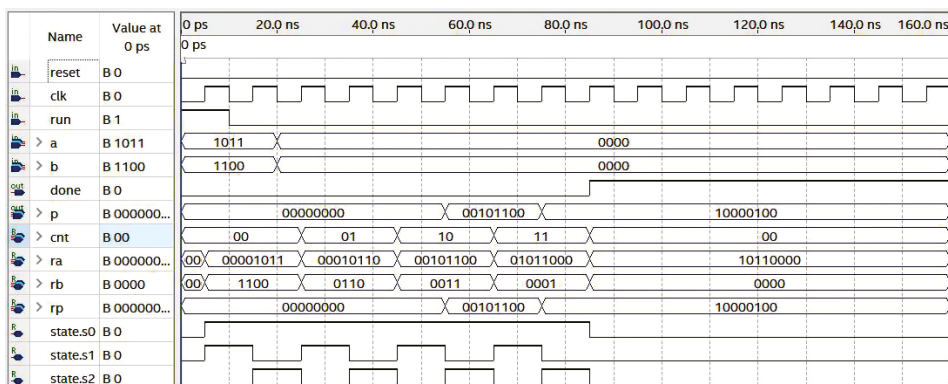
```

assign p = rp;           // określenie wartości wyjściowych
assign done = rdone;
endmodule

```

Moduł *mult_FSMD_Mealy* jest zapisany w stylu opisu automatów skończonych z jednym procesem [10]. W naszym przypadku jest to dopuszczalne, ponieważ iloczyn *p* i sygnał *done* w układzie mnożącym generowane są na wyjściach rejestrów. Wyniki syntezy i symulacji projektu *mult_FSMD_Mealy* przedstawiono na rysunkach 6.6 i 6.7.





RYS. 6.7. Wyniki symulacji projektu *mult_FSM Mealy*

Na rysunku 6.7 widać, że jeden cykl mnożenia jest wykonywany w dwóch cyklach synchronizacji, dlatego w projekcie *mult_FSM Mealy* mnożenie N -bitowych liczb binarnych odbywa się w n_{Mealy} cyklach synchronizacji, gdzie:

$$n_{Mealy} = 2 * N + 1. \quad (6.2)$$

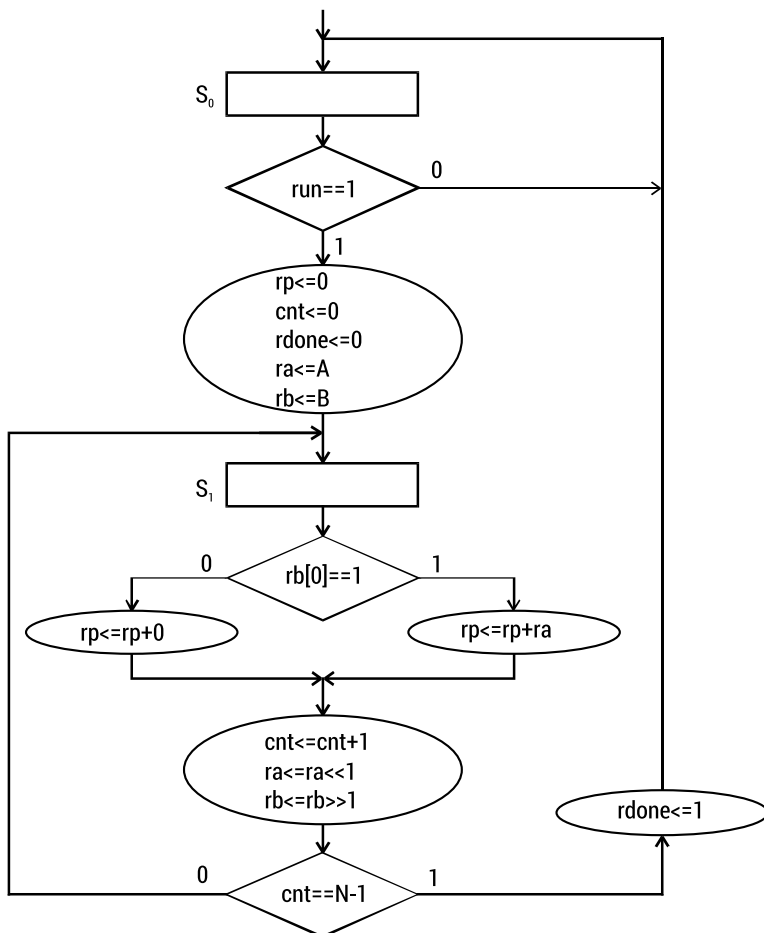
Porównanie FSM Moore'a i FSM Mealy'ego z naszego przykładu pokazuje, że FSM Mealy'ego ma znacznie mniej stanów (odpowiednio 3 i 7), a także działa znacznie szybciej, ponieważ każdy cykl procesu mnożenia jest wykonywany w dwóch cyklach zegara (w FSM Moore'a w trzech cyklach zegara).

6.5. Zwiększenie wydajności synchronicznego układu mnożącego

Fakt, że jeden cykl procesu mnożenia jest wykonywany w kilku cyklach zegarowych, sprawia że praktyczne zastosowanie omówionych powyżej projektów układów mnożących jest wyjątkowo nieefektywne, ponieważ znacznie wydłuża czas mnożenia.

Pytanie brzmi: jak zaprojektować FSM, który wykonuje pojedynczy cykl mnożenia w jednym cyklu zegarowym? Aby to zrobić, konieczne jest zbudowanie ASMD, w którym jeden blok ASMD odpowiada jednemu cyklowi mnożenia. ASMD spełniający to wymaganie pokazano na rysunku 6.8.

Cechą szczególną ASMD z rysunku 6.8 jest to, że mamy tu tylko dwa bloki ASMD: w stanie S_0 rejestry są inicjalizowane, a stan S_1 odpowiada pełnemu cyklowi mnożenia.



RYS. 6.8. ASMD dla automatu Mealy’ego zapewniająca największą szybkość synchronicznego układu mnożącego

Poniżej znajduje się kod projektu automatu FSMD Mealy’ego w języku Verilog, który odpowiada ASMD z rysunku 6.8.

```

module mult_FSMD_Mealy_1
#(parameter N=4)


```

```

localparam N_cnt=clogb2(N-1); // określenie wielkości licznika

function integer clogb2(input [N-1:0] v); // funkcja stała
    for (clogb2 = 0; v > 0; clogb2 = clogb2 + 1)
        v = v >> 1;
endfunction

localparam [0:0] s0=0,s1=1; // deklaracja stanów

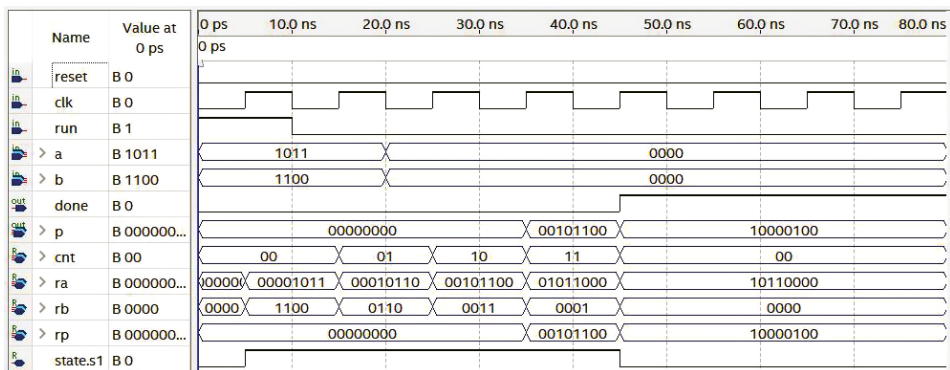
reg [0:0] state; // zmienna stanu

always @(posedge clk) // rozpoczęcie procesu mnożenia
    if(reset) state <= s0;
    else
        case (state)
            s0: if(run)
                begin
                    rp <= 0; cnt <= 0; rdone <= 0;
                    ra <= {{N{1'b0}},a};
                    rb <= b;
                    state <= s1;
                end
            else
                state <= s0;
            s1: begin
                if(rb[0]) rp <= rp + ra;
                else rp <= rp + {2*N{1'b0}};
                cnt <= cnt + 1'b1;
                rb <= rb >> 1;
                ra <= ra << 1;
                if (cnt == N-1)
                    begin
                        rdone <= 1'b1;
                        state <= s0;
                    end
                end
            else
                state <= s1;
            end
        default: state <= s0;
    endcase

assign p = rp; // określenie wartości wyjściowych
assign done = rdone;
endmodule

```

Wyniki syntezy i symulacji FSM D, zbudowanego zgodnie z ASMD na rysunku 6.8, pokazano na rysunkach 6.9 i 6.10.



RYS. 6.10. Wyniki symulacji projektu *mult_FSM Mealy_1*

Rysunek 6.10 pokazuje, że jeden cykl mnożenia w projekcie *mult_FSM Mealy_1* jest wykonywany w jednym cyklu synchronizacji, dlatego w projekcie *mult_FSM Mealy_1* mnożenie N -bitowych liczb binarnych odbywa się w $n_{Mealy-1}$ cyklach zegara synchronizacji, gdzie:

$$n_{Mealy-1} = N + 1. \quad (6.3)$$

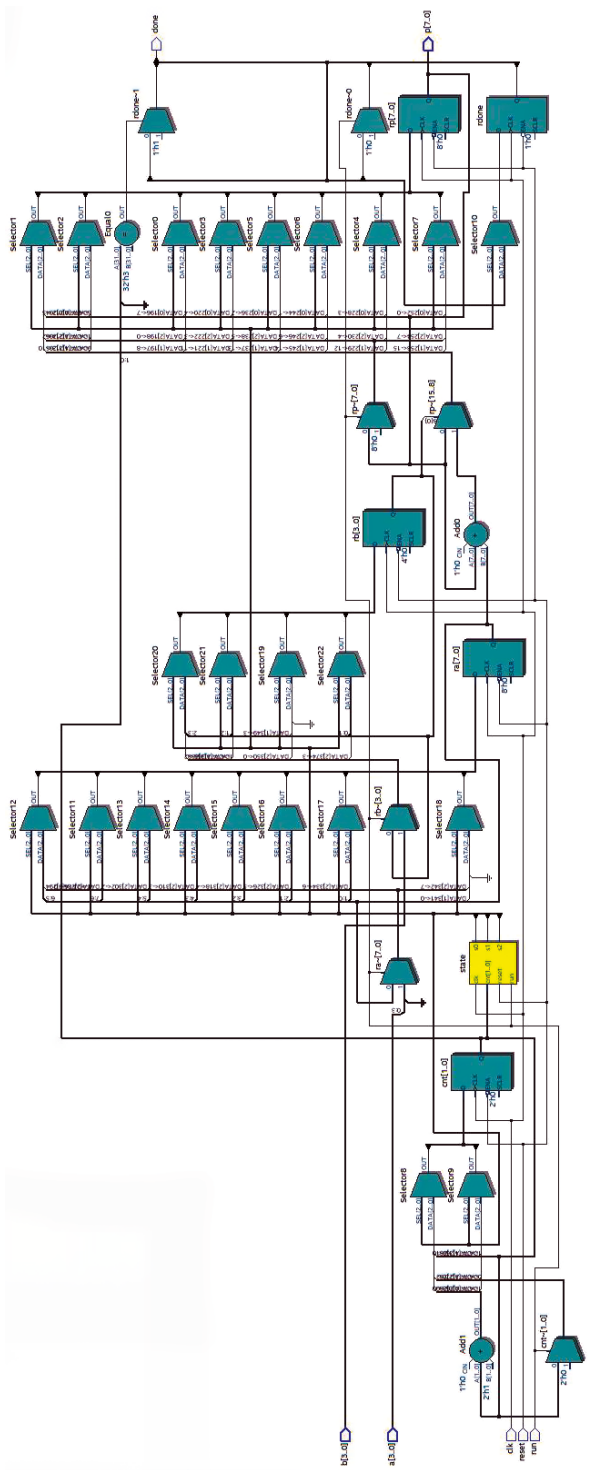
Rozważane przykłady opracowania synchronicznego układu mnożącego według algorytmu *a* pokazały, że szybkość opracowywanego projektu w dużej mierze zależy od budowy ASMD. W takim przypadku należy zwrócić uwagę na typ automatu: Mealy'ego lub Moore'a. W naszym przypadku użycie FSM Mealy'ego pozwoliło nam zbudować szybki układ mnożący, podczas gdy szybkość działania FSM Moore'a była trzykrotnie niższa.

Wnioski: aby budować szybkie układy należy:

- używać ASMD dla automatu Mealy'ego;
- starać się budować ASMD tak, aby miał jak najmniejszą liczbę stanów, szczególnie w cyklach algorytmu;
- opisać wszystkie akcje w pętli algorytmu jednym blokiem ASMD (działanie pożądane).

6.6. Porównanie różnych sposobów implementacji synchronicznego układu mnożącego

Pytanie postawione wcześniej pozostaje jednak otwarte: która z metodyk projektowania układów cyfrowych jest lepsza – tradycyjna, z oddzielnym projektowaniem układu operacyjnego i układu sterującego, czy też z wykorzystaniem FSM? W tym celu zbadano następujące projekty synchronicznych układów mnożących, implementujących algorytm mnożenia *a*:



- *mult_FSMD_Moore* – to układ mnożący zaprojektowany zgodnie z ASMD dla automatu Moore’a z rysunku 6.2;
- *mult_FSMD_Mealy* – to układ mnożący zaprojektowany zgodnie z ASMD dla automatu Mealy’ego z rysunku 6.5;
- *mult_FSMD_Mealy_1* – to układ mnożący zaprojektowany zgodnie z ASMD dla automatu Mealy’ego z rysunku 6.8;
- *mult_a_Mealy* – to układ mnożący z rozdziału 4 (projekt *mult_a*), zaprojektowany tradycyjnie, z oddzielnym projektowaniem układu operacyjnego i układu sterującego, gdy urządzenie sterowania jest realizowane przy użyciu AM Mealy’ego.

Wyniki takiego badania w systemie Quartus Prime w wersji 18.1 dla układów FPGA z rodziny Cyclone IV E przedstawiono w tabelach 6.1–6.4, gdzie N jest liczbą bitów słów wejściowych układu mnożącego; L – to liczba użytych bramek logicznych FPGA (koszt realizacji); F – to maksymalna częstotliwość projektu w megahercach (szybkość działania); T – to czas trwania jednego cyklu synchronizacji w nanosekundach; n – to liczba cykli zegara potrzebnych do obliczenia wyniku; t – to czas (w nanosekundach) wymagany do obliczenia wyniku, $t = T \cdot n$. Wartość n dla projektu *mult_FSMD_Moore* określa się zgodnie z rozdziałem (lub podrozdziałem, punktem, sekcją) 6.1, dla projektu *mult_FSMD_Mealy* – zgodnie z rozdziałem (lub podrozdziałem, punktem, sekcją) 6.2, a dla projektów *mult_FSMD_Mealy_1* i *mult_a_Mealy* – zgodnie z rozdziałem (lub podrozdziałem, punktem, sekcją) 6.3.

TAB. 6.1. Wyniki badań projektu *mult_FSMD_Moore*

N	L	F	T	n	t	T_{SDC}
4	47	197.55	5.062	15	75.93	10
8	102	172.71	5.790	27	156.33	10
16	192	147.91	6.761	51	344.81	10
32	309	149.32	6.679	99	663.00	10
64	583	106.47	9.392	195	1831.44	10
128	1139	31.64	31.606	387	12231.52	35

TAB. 6.2. Wyniki badań projektu *mult_FSMD_Mealy*

N	L	F	T	n	t	T_{SDC}
4	34	270.12	3.702	9	33.32	10
8	55	230.26	4.343	17	73.83	10
16	96	202.02	4.950	33	163.35	10
32	178	149.81	6.675	65	433.88	10
64	345	116.67	8.571	129	1105.66	10
128	662	32.95	30.349	257	7799.69	35

TAB. 6.3. Wyniki badań projektu *mult_FSMD_Mealy_1*

<i>N</i>	<i>L</i>	<i>F</i>	<i>T</i>	<i>n</i>	<i>t</i>	<i>T_{SDC}</i>
4	28	285.88	3.498	5	17.49	10
8	49	228.89	4.369	9	39.32	10
16	90	202.18	4.946	17	84.08	10
32	172	172.06	5.812	33	191.80	10
64	335	111.64	8.957	65	582.21	10
128	657	32.50	30.769	129	3969.20	35

TAB. 6.4. Wyniki badań projektu *mult_a_Mealy*

<i>N</i>	<i>L</i>	<i>F</i>	<i>T</i>	<i>n</i>	<i>t</i>	<i>T_{SDC}</i>
4	36	245.28	4.077	5	20.39	10
8	66	212.95	4.696	9	42.26	10
16	123	224.77	4.449	17	75.63	10
32	236	155.23	6.442	33	212.59	10
64	464	110.29	9.067	65	589.36	10
128	919	31.43	31.817	129	4104.39	35

W celu dokładniejszego określenia wydajności dla wszystkich układów mnożących, określono plik SDC ograniczeń czasowych [9]. Parametr T_{SDC} w tabelach 6.1–6.4 definiuje czas trwania cyklu zegara i wirtualne sygnały zegarowe dla wejść-wyjść, przy których kompilator spełnił wszystkie ograniczenia czasowe. Poniżej znajduje się przykład użytego pliku SDC.

```
create_clock -name clk -period 10.000 [get_ports {clk}]
create_clock -name virt_clk_in -period 10.000
create_clock -name virt_clk_out -period 10.000
derive_pll_clocks -create_base_clocks
derive_clock_uncertainty
set_input_delay -clock { virt_clk_in } -min 0 [get_ports {reset run a[*] b[*]}]
set_input_delay -clock { virt_clk_in } -max 1 [get_ports {reset run a[*] b[*]}]
set_output_delay -clock { virt_clk_out } -min 0 [get_ports {p[*] done}]
set_output_delay -clock { virt_clk_out } -max 1 [get_ports {p[*] done}]
```

Pomimo tego, że parametry F i T wszystkich projektów układów mnożących są do siebie podobne, czas t wykonania operacji mnożenia różni się znacznie dla różnych projektów. Wynika to z różnicy w liczbie n cykli zegara wymaganych do obliczenia wyniku.

Aby uzyskać lepiej widoczne porównanie rozpatrywanych projektów między sobą, w tabeli 6.5 przedstawiono wartości kosztu realizacji L i czasu t wykonania operacji mnożenia dla różnych wartości N .

TAB. 6.5. Badanie różnych metodyk projektowania sprzętu na przykładzie synchronicznego układu mnożącego

Projects	N=4		N=8		N=16		N=32		N=64		N=128	
	L	t	L	t	L	t	L	t	L	t	L	t
<i>mult_a_Mealy</i>	36	20.39	66	42.26	123	75.63	236	212.59	464	589.36	919	4104.39
<i>mult_FSM_D_Moore</i>	47	75.93	102	156.33	192	344.81	309	663.00	583	1831.44	1139	12231.52
<i>mult_FSM_D_Mealy</i>	34	33.32	55	73.83	96	163.35	178	433.88	345	1105.66	662	7799.69
<i>mult_FSM_D_Mealy_1</i>	28	17.49	49	39.32	90	84.08	172	191.80	335	582.21	657	3969.20
Best	28	17.49	49	39.32	90	75.63	172	191.80	335	582.21	657	3969.20

Na podstawie analizy tabeli 6.5 można zauważyć, że najlepszym projektem, zarówno pod względem kosztów realizacji, jak i szybkości działania, jest projekt *mult_FSM_D_Mealy_1*, z wyjątkiem czasu t przy $N = 16$.

Przewaga szybkości działania układu z projektu *mult_FSM_D_Mealy_1* nad projektami *mult_FSM_D_Moore* i *mult_FSM_D_Mealy* jest łatwa do wyjaśnienia. W projekcie *mult_FSM_D_Moore* jeden cykl procesu mnożenia wykonywany jest w trzech cyklach synchronizacyjnych, w projekcie *mult_FSM_D_Mealy* w dwóch cyklach synchronizacyjnych, a w projekcie *mult_FSM_D_Mealy_1* w jednym cyklu synchronizacyjnym.

W projekcie *mult_a_Mealy* jeden cykl procesu mnożenia jest również wykonywany w jednym cyklu zegarowym, jednak czas trwania cyklu dla projektu *mult_FSM_D_Mealy_1* jest krótszy (poza przypadkiem $N = 16$). Przewaga szybkości projektu *mult_FSM_D_Mealy_1* nad projektem *mult_a_Mealy* może być wyjaśniona skuteczną realizacją operacji języka Verilog przez kompilator Quartus.

Projekt *mult_FSM_D_Mealy_1* przewyższa również wszystkie inne projekty pod względem kosztów realizacji. Przewaga projektu *mult_FSM_D_Mealy_1* nad wszystkimi projektami pod względem kosztów realizacji jest również wyjaśniona skuteczną realizacją operacji języka Verilog przez kompilator Quartus.

Projekt *mult_FSM_D_Mealy_1*, w odróżnieniu od projektu *mult_a_Mealy*, został opisany w języku Verilog bezpośrednio zgodnie z ASMD na rysunku 6.8, bez wyodrębnienia układu operacyjnego i układu sterującego. Zatem zastosowanie metodyki projektowania układów cyfrowych opartej na ASMD ze ścieżką przetwarzania danych w postaci automatu skończonego ze ścieżką przetwarzania danych FSM_D (zwanej dalej po prostu metodyką ASMD-FSM_D), pozwoliło znacznie obniżyć koszty realizacji i zwiększyć szybkość działania układu mnożącego. Ponadto metodyka ASMD-FSM_D

zdecydowanie skraca czas projektowania w porównaniu z podejściem tradycyjnym, ponieważ nie ma potrzeby projektowania układu operacyjnego i wszystkich jego elementów składowych, a także układu sterującego.

Należy zauważyć, że metodyka projektowania ASMD-FSMD może również poprawić niezawodność projektów. Faktem jest, że wiele etapów tradycyjnego projektowania jest wykonywanych przez inżyniera ręcznie, co jest źródłem trudnych do znalezienia błędów. W metodyce ASMD-FSMD, po zbudowaniu ASMD, opis projektu w języku Verilog wykonywany jest bezpośrednio zgodnie z ASMD, bez opisów pośrednich. Oczywiście opanowanie metodyki ASMD-FSMD wymaga opanowania pewnych umiejętności, ale jak widać, warto to uczynić.

W metodyce ASMD-FSMD, budowa ASMD obarczona jest specjalnymi wymaganiami. Doświadczenie w stosowaniu metodyki ASMD-FSMD wykazało, że dla jej efektywnego stosowania konieczne jest przestrzeganie pewnych zasad. Przede wszystkim należy dążyć do zbudowania schematów blokowych ASMD dla automatów typu Mealy'ego. Jeżeli algorytm działania układu zawiera cykle, to cały cykl w ASMD należy opisać jednym stanem. Jeśli nie jest to możliwe, należy użyć minimalnej liczby stanów. W prostokątach i owalach ASMD należy zapisywać tylko operacje Verilog, bez używania złożonych wyrażeń.

6.7. Wnioski

Połączenie procesów projektowania układu operacyjnego i układu sterującego ułatwia fakt, że obie klasy tych układów mogą być opisane tym samym językiem Verilog. Ponadto urządzenie operacyjne i urządzenie sterowania są generalnie automatami skończonymi.

ASM ze ścieżką przetwarzania danych (ASM with datapath – ASMD) to normalny ASM, w którym w prostokątach i owalach można zapisywać dowolne operacje na rejestrach, dozwolone w języku Verilog, a w wierzchołkach warunkowych sprawdzać dowolne wyrażenia logiczne w języku Verilog. Automat skończony zaimplementowany w postaci ASMD nazywany jest *automatem skończonym ze ścieżką przetwarzania danych* (FSM with datapath – FSMD).

Zauważmy, że wszystkie operacje na rejestrach opisane wewnątrz bloku ASMD będą wykonywane przy wyjściu z bloku, tzn. na kolejnym dodatnim zboczku sygnału synchronizacji. Ta właściwość ASMD może być źródłem pewnych błędów, gdy zmieniana wartość rejestru jest sprawdzana w tym samym bloku ASMD. Aby uniknąć takiego błędu, można wprowadzić zmienną pośrednią typu **wire**, w której będzie przechowywana tymczasowa wartość rejestru.

Szybkość realizacji projektu w dużej mierze zależy od budowy ASMD. W takim przypadku należy zwrócić uwagę na typ budowanego automatu: Mealy'ego lub Moore'a. Zwykle wykorzystanie modelu automatu typu Mealy'ego do budowy ASMD pomaga zmniejszyć liczbę stanów FSMD i zwiększyć wydajność projektu.

Aby zbudować szybkie urządzenia, należy:

- używać ASMD dla automatu Mealy'ego;
- starać się budować ASMD tak, aby miał jak najmniejszą liczbę stanów, szczególnie w cyklach algorytmu;
- wszystkie akcje w pętli algorytmu opisywać jednym blokiem ASMD, jeśli jest to możliwe.

Przewagę pod względem kosztów realizacji i szybkości realizacji projektów opracowanych zgodnie z metodyką ASMD-FSMD, w porównaniu z podejściem tradycyjnym, w którym projekt prezentowany jest w postaci układu i układu sterującego, tłumaczy sprawna implementacja operacji języka Verilog przez kompilator systemu Quartus.

Metodyka ASMD-FSMD znacznie skraca czas projektowania w porównaniu z podejściem tradycyjnym, ponieważ nie ma potrzeby projektowania układu operacyjnego i wszystkich jego elementów składowych, a także układu sterującego.

Metodyka projektowania ASMD-FSMD poprawia niezawodność projektów, ponieważ w metodyce ASMD-FSMD po zbudowaniu ASMD, opis projektu w języku Verilog wykonywany jest bezpośrednio według ASMD, bez żadnych opisów pośrednich, podczas gdy w podejściu tradycyjnym wiele etapów realizuje inżynier ręcznie, co jest źródłem trudnych do znalezienia błędów.

W metodyce projektowania ASMD-FSMD stawia się specjalne wymagania co do budowy ASMD. Należy dążyć do tworzenia schematów blokowych ASMD dla automatu typu Mealy'ego. Jeżeli algorytm pracy układu zawiera cykle, to cały cykl w ASMD należy opisać jednym stanem. Jeśli nie jest to możliwe, należy użyć minimalnej liczby stanów. W prostokątach i owalach ASMD należy zapisywać tylko operacje Verilog, bez używania złożonych wyrażeń.

Metodyka ASMD-FSMD do projektowania układów i systemów cyfrowych pozwala znacznie skrócić czas i koszt projektowania, a także poprawić parametry takie jak: koszt realizacji, szybkość i niezawodność projektu. Jednak, aby osiągnąć wysokie wskaźniki wydajności, należy użyć FSMD Mealy'ego i dokładnie przemyśleć każdy stan wewnętrzny automatu opisanego w ASMD.

Część III

Projektowanie procesorów wbudowanych

Nowoczesne systemy wbudowane często zawierają procesory zwane *procesorami wbudowanymi* (embedded processors), służące do wykonywania programów i algorytmów. W układach FPGA procesor wbudowany może być zaimplementowany „sztywno” (hard), zajmując część fizycznej struktury FPGA, lub może być zaimplementowany „międko” (soft), za pomocą konfiguracji wewnętrznej struktury FPGA. W tym drugim przypadku do implementacji procesora zostaną wykorzystane zasoby FPGA: elementy logiczne oraz bloki pamięci wbudowanej. Przykładami takich procesorów są Nios II i MicroBlaze. Przy sztywnej implementacji procesora w strukturze FPGA układ programowalny nazywany jest *systemem w jednym układzie scalonym* lub *systemem jednoukładowym* (System on Chip – SoC).

Konieczność zaprojektowania oryginalnego procesora na FPGA może pojawić się w następujących przypadkach:

- przy rozwiązywaniu bardzo prostych zadań;
- przy rozwiązywaniu specyficznych zadań;
- przy braku w strukturze FPGA wbudowanego procesora;
- przy braku wolnych zasobów FPGA do realizacji wbudowanego soft-procesora typu Nios II i in;
- gdy używany jest tylko ograniczony zestaw instrukcji znanego procesora i zaimplementowany jest podzbiór instrukcji określonego procesora w celu oszczędzania zasobów FPGA;
- gdy procesor wymagany jest do wykonania jednego konkretnego programu, w tym przypadku możliwa jest znaczna oszczędność wykorzystywanych zasobów FPGA;
- gdy oryginalny procesor jest zaimplementowany z własnym zestawem instrukcji i in.

Ponadto za pomocą układów FPGA są często implementowane prototypy nowych procesorów do późniejszych badań.

7. Podstawy projektowania procesorów wbudowanych

Rozdział obejmuje ogólne zagadnienia projektowania procesorów wbudowanych na FPGA: tradycyjną architekturę zestawu instrukcji procesora, architekturę i projekt zestawu instrukcji procesora PIC, podstawowe struktury procesorów RISC, a także metodologię projektowania procesorów na FPGA.

Na pierwszy rzut oka może się wydawać, że zaprojektowanie procesora na FPGA to bardzo trudne zadanie. Jednak nie jest to do końca prawdą. Dysponując znajomością metodyki projektowania procesorów na FPGA oraz metod projektowania poszczególnych elementów systemów wbudowanych, studenci w laboratorium mogą zbudować prosty procesor na FPGA w ciągu dwóch godzin akademickich.

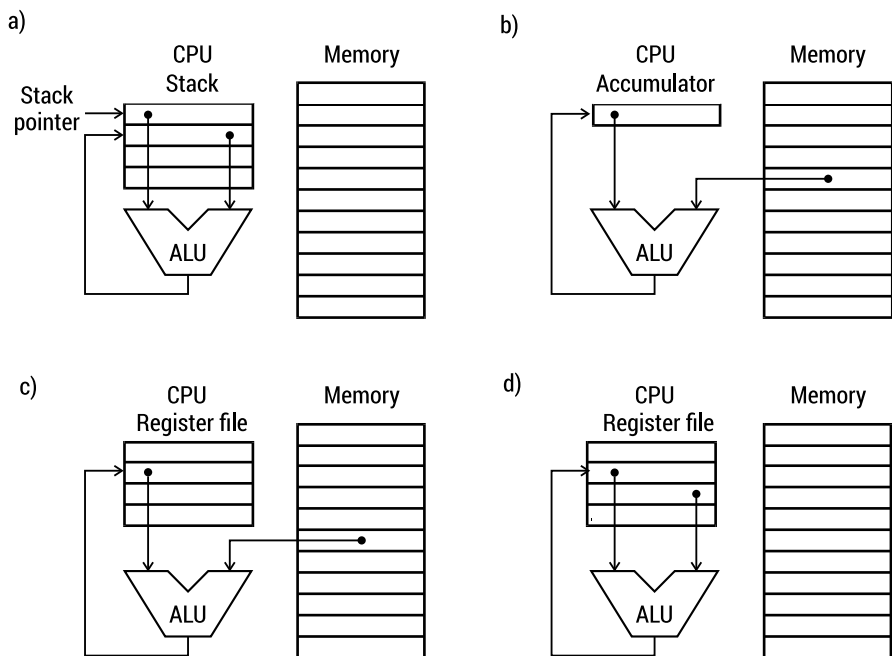
Przykłady projektowania procesorów jedno- i wielocyklowych zostaną omówione w następnych rozdziałach tej książki.

7.1. Architektury zestawów instrukcji procesora

Procesor zaimplementowany w FPGA jest zwykle zbudowany na bazie jednej ze znanych architektur. Ułatwia to tworzenie i debugowanie programów na procesorach wbudowanych, ponieważ umożliwia to korzystanie z narzędzi do tworzenia oprogramowania dla niektórych popularnych procesorów. Ponadto można korzystać z wcześniej utworzonych programów, a także standardowych systemów operacyjnych.

Głównym węzłem każdego procesora jest jego *jednostka arytmetyczno-logiczna* lub *układ arytmetyczno-logiczny* (arithmetic logical unit – ALU). Tradycyjnie ALU pobiera dwa operandy, wykonuje na nich określoną operację i zapisuje wynik w określonym miejscu procesora.

Lokalizacja operandów i wyniku determinuje *architekturę zestawu instrukcji* lub *architekturę listy rozkazów* (instruction set architecture – ISA) procesora. Obecnie znane są cztery główne architektury zestawu instrukcji procesora: stosowa (rys. 7.1a), z akumulatorem (rys. 7.1b), rejestr-pamięć (rys. 7.1c) i rejestr-rejestr (rys. 7.1d).



RYS. 7.1. Architektury zestawu instrukcji procesora: a – stosowa; b – z akumulatorem; c – rejestr-pamięć; d – rejestr-rejestr

W architekturze stosowej (rys. 7.1a) oba operandy zawsze znajdują się na szczycie stosu, a wynik jest również wstawiany na stos. Do współdziałania z pamięcią służą polecenia zapisu na stosie (odłożenia na stos) **push** i odczytu ze stosu (zdjęcia ze stosu) **pop**. Wirtualna maszyna Javy (Java virtual machine – JVM) jest właśnie zbudowana na podstawie architektury stosowej. Zaletą architektury stosowej jest to, że instrukcje procesora mogą w ogóle nie zawierać adresów argumentów. W rezultacie rozmiary instrukcji i kodu maszynowego mają minimalną długość.

W architekturze z akumulatorem (rys. 7.1b), jeden z argumentów jest zawsze umieszczany w akumulatorze, a drugi operand jest odczytywany z pamięci. Wynik operacji jest zawsze przechowywany w akumulatorze. Architektura zestawu instrukcji z akumulatorem jest używana przez procesory Z80 i 6502. Instrukcje procesora z akumulatorem zawierają adres tylko jednego argumentu. Wadą tej architektury są częste odwołania do pamięci.

W architekturze rejestr-pamięć (rys. 7.1c), jeden operand jest umieszczany jawnie w *pliku rejestrów* (register file), a drugi w pamięci. Przypomnijmy, że plik rejestrów to niewielka liczba bardzo szybkich rejestrów ogólnego przeznaczenia. Wynik operacji w architekturze rejestr-pamięć jest zawsze zapisywany do pliku rejestrów. Zaletą tej architektury instrukcji jest jej wszechstronność: w każdej instrukcji można zaadresować zarówno oddzielne rejestry ogólnego przeznaczenia, jak i komórki pamięci. Wadą tej architektury jest złożoność adresowania operandów i instrukcji, stąd wynika duży

rozmiar instrukcji i programów. Wszystkie procesory Intel x86 są oparte na architekturze rejestr-pamięć.

Większość komputerów o zredukowanym zestawie instrukcji (reduced instruction set computer – RISC) ma architekturę rejestr-rejestr (rys. 7.1d). W tej architekturze oba operandy znajdują się w pliku rejestrów, wynik operacji jest również zapisywany w pliku rejestrów. Dlatego wiele instrukcji procesora zawiera jednocześnie trzy adresy: dwa argumenty i wynik. Jednak są to adresy o małym rozmiarze. Aby załadować dane z pamięci do pliku rejestrów, należy użyć polecenia **load**, a aby zapisać rejestr w pamięci, należy użyć polecenia **store**. Zaletą architektury instrukcji rejestr-rejestr jest prostota adresowania operandów i wyniku, mała liczba formatów instrukcji, możliwość użycia stałego rozmiaru instrukcji, duża szybkość wykonywania instrukcji, możliwość wykonywania instrukcji potokowo itp. Oczywiście architektura rejestr-rejestr ma pewne wady, ale zalet jest więcej.

Tabela 7.1 pokazuje fragmenty kodów w języku assemblera służących do wykonywania operacji dodawania dwóch liczb dla różnych architektur zestawu instrukcji procesora.

TAB. 7.1. Przykładowe kody dla różnych architektur zestawu instrukcji procesora

Stack	Accumulator	Register-memory	Register-register
push x	load x	load r1, x	load r1, x
push y	add y	add r1, y	load r2, y
add	store z	store z, r1	add r3, r1, r2
pop z			store z, r3

7.2. Architektura zestawu instrukcji realizowanego procesora

Załóżmy, że konieczne jest stworzenie wbudowanego procesora, który implementuje podzbiór instrukcji popularnego mikrokontrolera PIC16F84A [11]. Projektowany przez nas procesor będziemy nazywać – *procesorem PIC*.

7.2.1. Architektura zestawu instrukcji mikrokontrolera PIC16F84A

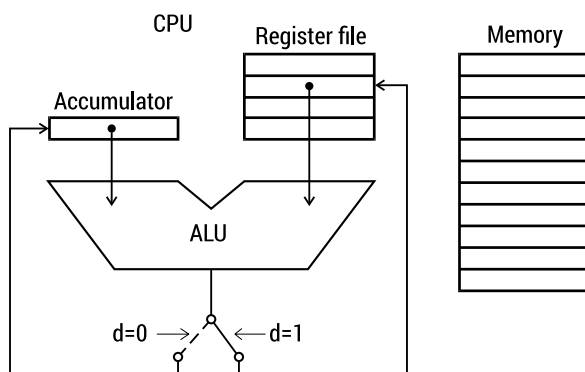
Mikrokontroler PIC16F84A należy do średniej klasy mikrokontrolerów PIC (Peripheral Interface Controller) i zawiera tylko 35 instrukcji (tab. 7.2).

TAB. 7.2. Zestaw instrukcji mikrokontrolera PIC16F84A

Mnemoniki, operandy	Opis	Kod maszynowy	Cykle
Operacje bajtowe			
ADDWF f, d	Dodawanie W i f ($W + f$)	00 0111 dfff ffff	1
ANDWF f, d	Iloczyn logiczny W i f ($W \& f$)	00 0101 dfff ffff	1
CLRF f	Zerowanie f ($f = 0$)	00 0001 1fff ffff	1
CLRW –	Zerowanie W ($W = 0$)	00 0001 0xxx xxxx	1
COMF f, d	Uzupełnienie f ($f = \sim f$)	00 1001 dfff ffff	1
DECF f, d	Dekrementacja f ($f = f - 1$)	00 0011 dfff ffff	1
DECFSZ f, d	Dekrementacja f, z pominięciem następnego polecenia, jeśli 0 ($(f - 1) == 0$, skip = 1)	00 1011 ffff ffff	1 (2)
INCF f, d	Inkrementacja f ($f = f + 1$)	00 1010 dfff ffff	1
INCFSZ f, d	Inkrementacja f, z pominięciem następnego polecenia, jeśli 0 ($(f + 1) == 0$, skip = 1)	00 1111 dfff ffff	1 (2)
IORWF f, d	Suma logiczna W z f ($W f$)	00 0100 dfff ffff	1
MOVF f, d	Zapis f ($f \rightarrow W$ lub $f \rightarrow f$)	00 1000 dfff ffff	1
MOVWF f	Zapis W do f ($W \rightarrow f$)	00 0000 1fff ffff	1
NOP –	Brak operacji (zapis z W do W)	00 0000 0xx0 0000	1
RLF f, d	Cykliczne przesunięcie f w lewo przez flagę przeniesienia	00 1101 dfff ffff	1
RRF f, d	Cykliczne przesunięcie f w prawo przez flagę przeniesienia	00 1100 dfff ffff	1
SUBWF f, d	Odejmowanie W od f ($f - W$)	00 0010 dfff ffff	1
SWAPF f, d	Zamiana półbajtów w f	00 1110 dfff ffff	1
XORWF f, d	XOR dla W i f	01 11bb bfff ffff	1
Operacje bitowe			
BCF f, b	Zerowanie bitu w f	01 00bb bfff ffff	1
BSF f, b	Ustawianie bitu w f	01 01bb bfff ffff	1
BTFSC f, b	Sprawdź bit w f, pomiń polecenie, jeśli 0	01 10bb bfff ffff	1 (2)
BTFSS f, b	Sprawdź bit w f, pomiń polecenie, jeśli 1	01 11bb bfff ffff	1 (2)
Stałe operacje i polecenia sterujące			
ADDLW k	Dodawanie stałej do W ($k1 + W$)	11 111x kkkk kkkk	1
ANDLW k	Iloczyn logiczny stałej i W ($W \& k1$)	11 1001 kkkk kkkk	1
IORLW k	Suma logiczna stałej i W ($W k1$)	11 1000 kkkk kkkk	1
MOVLW k	Zapis stałej w W ($k1 \rightarrow W$)	11 00xx kkkk kkkk	1
SUBLW k	Odejmowanie W od stałej ($k1 - W$)	11 110x kkkk kkkk	1
XORLW k	XOR dla stałej i W ($W \wedge k1$)	11 1010 kkkk kkkk	1
CALL k	Wywołanie podprogramu ($\rightarrow k2$)	10 0kkk kkkk kkkk	2

Mnemoniki, operandy	Opis	Kod maszynowy	Cykle
GOTO k	Skok do adresu ($\rightarrow k2$)	10 1kkk kkkk kkkk	2
RETURN -	Powrót z podprogramu	00 0000 0000 1000	2
RETLW k	Powrót z podprogramu ze stałą w W ($k1 \rightarrow W$)	11 01xx kkkk kkkk	2
RETFIE -	Powrót po przerwaniu	00 0000 0000 1001	2
CLRWDT -	Resetowanie timera watchdoga	00 0000 0110 0100	1
SLEEP -	Przełączanie w tryb uśpienia	00 0000 0110 0011	1

Architektura zestawu instrukcji PIC (rys. 7.2) łączy zalety architektury z akumulatorem i architektury rejestr-rejestr.



RYŚ. 7.2. Architektura zestawu instrukcji mikrokontrolera PIC

Jeden z operandów tej architektury znajduje się zawsze w akumulatorze, a drugi pochodzi z rejestru ogólnego przeznaczenia znajdującego się w pliku rejestrów. Lokalizacja wyniku operacji jest określana przez wartość bitu *d*, który znajduje się w kodzie polecenia. Przy *d* = 0 wynik jest umieszczany w akumulatorze W, a przy *d* = 1 wynik jest ponownie zapisywany w rejestrze ogólnego przeznaczenia.

Mikrokontroler PIC16F84A ma stosunkowo dużą (68) liczbę rejestrów ogólnego przeznaczenia, które tworzą *plik rejestrów* (Register File). Mikrokontroler PIC16F84A nie ma tradycyjnej pamięci danych. Zamiast tego mikrokontroler PIC16F84A wykorzystuje nieulotną pamięć typu EEPROM z dość złożonym mechanizmem odczytu i zapisu.

W projektowanym procesorze PIC wykorzystamy tradycyjną pamięć danych, taką jak RAM. Aby procesor współdziałał z pamięcią danych, wprowadzimy do zestawu instrukcji naszego procesora PIC dwa dodatkowe polecenia: **lw** – załaduj wartość akumulatora z pamięci i **sw** – zapisz wartość akumulatora do pamięci.

Fragment kodu języka assembler dla naszego procesora podczas dodawania dwóch liczb będzie wyglądał następująco:

```
lw x           // x → W
addlw y        // W = x + y → W
sw z           // W → z
```

co odpowiada architekturze zestawu instrukcji z akumulatorem.

7.2.2. Projektowanie zestawu instrukcji procesora PIC

Jak wspomniano wcześniej, implementowany przez nas procesor PIC ma tradycyjną pamięć danych, czyli RAM, natomiast mikrokontroler PIC16F84A ma pamięć nieulotną typu EEPROM. Rozmiar słowa pamięci danych wynosi 8 bitów, jak w mikrokontrolerze PIC16F84A. Problem pojawia się z rozmiarem adresu pamięci danych. Faktem jest, że lista instrukcji mikrokontrolera PIC16F84A ma bardzo niewiele możliwości dodawania nowych instrukcji. Listę instrukcji musimy rozszerzyć o dwa polecenia: **sw** – zapis do pamięci danych i **lw** – odczyt z pamięci danych. Aby to zrobić, rozpatrzmy polecenie **movlw**. Kod maszynowy **movlw** ma dwa niezdefiniowane bity [9:8] i pole stałej [7:0]. Bity [9:8] kodu maszynowego instrukcji **movlw** zostaną wykorzystane do zdefiniowania kodów nowych instrukcji. Załóżmy następujące kodowanie rozpatrywanych instrukcji za pomocą bitów [9:8]: **movlw** – 00, **lw** – 01 i **sw** – 10.

Jako adres pamięci danych zostanie użyte pole stałej [7:0] polecenia **movlw**. Z tego powodu pamięć danych naszego procesora PIC może mieć 256 słów po 8 bitów, tzn. ma konfigurację 256x8. Oczywiście 8-bitowy adres ogranicza ilość *pamięci danych* (Data Memory – DM). Problem ten można rozwiązać, zwiększając kod polecenia do 16 bitów.

Kolejne pytanie dotyczy tego, skąd pochodzą dane, które mają być zapisane w pamięci danych i gdzie trafiają dane podczas odczytu z pamięci danych. W procesorach o dużym rozmiarze instrukcji możliwa jest wymiana danych między rejestrami pliku rejestrów i pamięcią danych. W naszym przypadku, ponieważ w słowie polecenia jest miejsce tylko na jeden adres, wymiana danych jest możliwa tylko między pamięcią danych a akumulatorem W.

Plik rejestrów mikrokontrolera PIC16F84A zawiera 68 rejestrów ogólnego przeznaczenia i ma dość złożoną strukturę: jest podzielony na dwa banki i ma określone adresy na stałe powiązane z rejestrami specjalnymi. Rejestry specjalne służą do: komunikacji z urządzeniami peryferyjnymi, obsługi mechanizmu przerwań, zapisu do pamięci typu EEPROM itp.

Nasz procesor PIC nie ma żadnych układów peryferyjnych, żadnych przerwań i możliwości zapisu do EEPROM. Dlatego uprościmy plik rejestrów naszego procesora i zdefiniujemy sobie go jako zwykłą pamięć typu RAM o konfiguracji 128x8 (128 słów po 8 bitów). W ten sposób nasz procesor PIC będzie miał dużo (128) rejestrów ogólnego przeznaczenia. Jeśli zajdzie konieczność wymiany danych z urządzeniami zewnętrznymi, wówczas można w tym celu wykorzystać oddzielne komórki pamięci danych.

Długość instrukcji mikrokontrolera PIC16F84A wynosi 14 bitów, więc rozmiar słowa pamięci rozkazów (Instruction Memory – IM) również wynosi 14 bitów. W naszym procesorze PIC przyjmuje się, że rozmiar słowa pamięci instrukcji wynosi 16 bitów (biorąc pod uwagę przyszłe rozszerzenie zbioru instrukcji), ale użyjemy tylko 14 najmniej znaczących bitów.

Rozmiar adresu pamięci instrukcji mikrokontrolera PIC16F84A wynosi 13 bitów, ale używany jest tylko obszar pamięci adresowany przez 11 najmniej znaczących bitów. Ponieważ nasz procesor PIC nie jest przeznaczony do dużych programów, ograniczymy rozmiar adresu pamięci instrukcji do 10 bitów. Zatem pamięć instrukcji naszego procesora PIC ma konfigurację 1024x14 (1024 słowa po 14 bitów).

Nasz procesor PIC nie jest mikrokontrolerem, dlatego nie posiada ani timera watchdog, ani trybu uśpienia, ani mechanizmu obsługi przerwań. Dlatego też polecenia **clrwlt**, **sleep** i **retfie** zostaną wykluczone z listy instrukcji mikrokontrolera PIC16F84A.

Jest jeszcze jeden problem podczas pisania programów dla naszego procesora PIC. Implementacja rozgałęzień algorytmu w mikrokontrolerze PIC16F84A przebiega zazwyczaj w następujący sposób: sprawdzana jest flaga zerowa z w rejestrze stanu i w zależności od jej wartości kolejne polecenie jest pomijane lub nie. Nasz procesor PIC nie ma rejestru stanu, więc takie podejście do implementacji rozgałęziania algorytmów nie jest możliwe.

Aby zapewnić możliwość rozgałęzienia algorytmu zaimplementowanego przez program, wprowadzamy dwie nowe instrukcje: **gotoz** – skok do adresu, jeśli wynikiem poprzedniej operacji jest zero, oraz **gotonz** – skok do adresu, jeśli wynik poprzedniej operacji nie jest zerowy.

Aby wprowadzić te polecenia do listy instrukcji mikrokontrolera PIC16F84A, należy zwrócić uwagę, że kod maszynowy polecenia **retlw** dla powrotu z podprogramu ze stałą ma wartość nieokreśloną w bitach [9:8]. Do polecenia **gotoz** przypiszemy kod operacji 110101 w bitach [13:8], do polecenia **gotonz** przypiszemy kod operacji 110110, a kod 110100 pozostawimy dla polecenia **retlw**. Wadą tej definicji instrukcji **gotoz** i **gotonz** jest to, że przeskok przy użyciu tych instrukcji jest możliwy w zakresie adresów od 0 do 255. Można rozwiązać ten problem, wydłużając kod polecenia do 16 bitów.

Biorąc to wszystko pod uwagę, zestaw instrukcji naszego procesora PIC ma postać przedstawioną w tabeli 7.3.

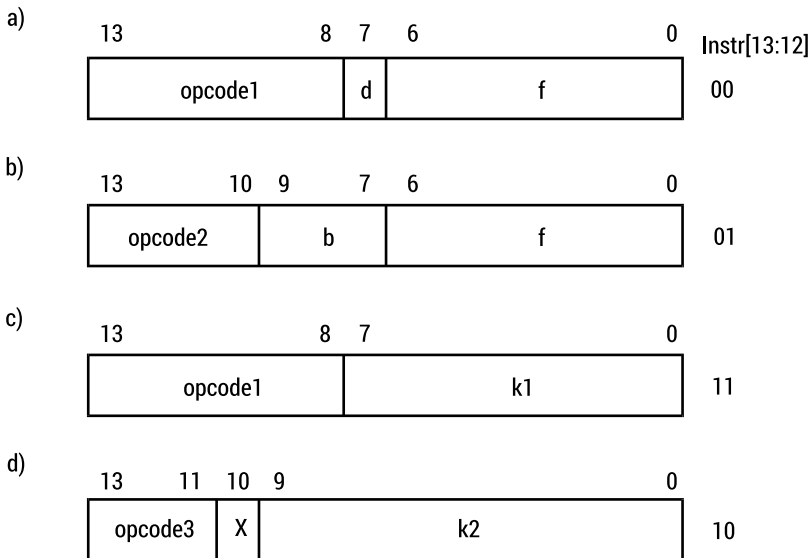
TAB. 7.3. Zestaw instrukcji procesora PIC

Mnemoniki, operandy	Opis	Kod maszynowy
Operacje na bajtach (polecenia formatu a)		
ADDWF f, d	Dodawanie W i f (W + f)	00 0111 dfff ffff
ANDWF f, d	Iloczyn logiczny W i f (W i f)	00 0101 dfff ffff
CLRF f	Zerowanie f (f = 0)	00 0001 1fff ffff

Mnemoniki, operandy	Opis	Kod maszynowy
CLRW –	Zerowanie W ($W = 0$)	00 0001 0xxx xxxx
COMF f, d	Uzupełnienie F ($f = \sim f$)	00 1001 dfff ffff
DECF f, d	Dekrementacja f ($f = f - 1$)	00 0011 dfff ffff
DECFSZ f, d	Dekrementacja f z pominięciem następnego polecenia, jeśli 0 ($((f - 1) == 0, \text{skip} = 1)$)	00 1011 ffff ffff
INCF f, d	Inkrementacja f ($f = f + 1$)	00 1010 dfff ffff
INCFSZ f, d	Inkrementacja f z pominięciem następnego polecenia, jeśli 0 ($((f + 1) == 0, \text{skip} = 1)$)	00 1111 dfff ffff
IORWF f, d	Suma logiczna W z f ($W f$)	00 0100 dfff ffff
MOVF f, d	Zapis f ($((f) \rightarrow W$ lub $(f) \rightarrow f$)	00 1000 dfff ffff
MOVWF f	Zapis W do f ($W \rightarrow f$)	00 0000 1fff ffff
NOP –	Brak operacji (zapis z W do W)	00 0000 0xx0 0000
RLF f, d	Cykliczne przesunięcie f w lewo przez flagę przeniesienia	00 1101 dfff ffff
RRF f, d	Cykliczne przesunięcie f w prawo przez flagę przeniesienia	00 1100 dfff ffff
SUBWF f, d	Odejmovanie W od f ($f - W$)	00 0010 dfff ffff
SWAPF f, d	Zamiana półbajtów w f	00 1110 dfff ffff
XORWF f, d	XOR dla W i f	00 0110 dfff ffff
Operacje na bitach (polecenia formatu b)		
BCF f, b	Zerowanie bitu b w f	01 00bb bfff ffff
BSF f, b	Ustawienie bitu b w f	01 01bb bfff ffff
BTFSC f, b	Sprawdź bit b w f, pomiń polecenie, jeśli jest równe 0	01 10bb bfff ffff
BTFSS f, b	Sprawdź bit b w f, pomiń polecenie, jeśli jest równe 1	01 11bb bfff ffff
Operacje na stałych (polecenia formatu c)		
ADDLW k1	Dodawanie stałej i W ($k1 + W$)	11 111x kkkk kkkk
ANDLW k1	Iloczyn logiczny stałej i W ($W \& k1$)	11 1001 kkkk kkkk
IORLW k1	Suma logiczna stałej i W ($W k1$)	11 1000 kkkk kkkk
MOVLW k1	Zapisywanie stałej w W ($k1 \rightarrow W$)	11 0000 kkkk kkkk
LW k1	Ładowanie W z pamięci ($\text{mem} \rightarrow W$)	11 0001 kkkk kkkk
SW k1	Zapis zawartości W w pamięci ($W \rightarrow \text{mem}$)	11 0010 kkkk kkkk
SUBLW k1	Odejmovanie W od stałej ($k1 - W$)	11 110x kkkk kkkk
XORLW k1	XOR stałej i W ($W \wedge k1$)	11 1010 kkkk kkkk
Polecenia sterujące (polecenia formatu d)		
CALL k2	Wywołanie programu ($\rightarrow k2$)	10 0kkk kkkk kkkk
GOTO k2	Przejdźcie do adresu ($\rightarrow k2$)	10 1kkk kkkk kkkk
RETURN –	Powrót z podprogramu	00 0000 0000 1000
RETLW k1	Powrót z podprogramu z stałej w W ($k1 \rightarrow W$)	11 0100 kkkk kkkk

Mnemoniki, operandy	Opis	Kod maszynowy
GOTOZ k1	Przejdźcie do adresu, jeśli wynikiem poprzedniego polecenia jest 0	11 0101 kkkk kkkk
GOTONZ k1	Przejdźcie do adresu, jeśli wynik poprzedniego polecenia nie wynosi 0	11 0110 kkkk kkkk

Wszystkie polecenia naszego procesora PIC można warunkowo podzielić na cztery formaty (rys. 7.3):



RYS. 7.3. Formaty zestawu instrukcji procesora PIC

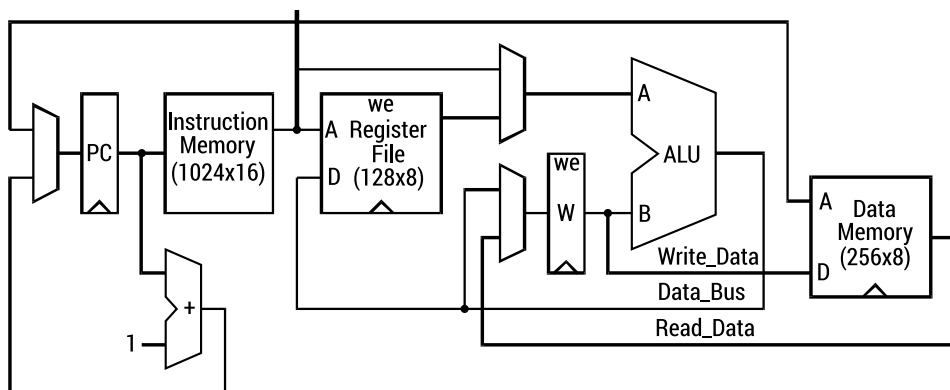
Zauważmy, że tylko dwa polecenia odpowiadają formatowi: **call** i **goto**. Pozostałe polecenia sterujące (**return**, **retlw**, **gotoz** i **gotonz**) nie odpowiadają żadnemu z powyższych formatów instrukcji.

7.3. Podstawowe struktury procesorów RISC

Z reguły procesory wbudowane w FPGA budowane są na architekturze o zredukowanym zestawie instrukcji, tzn. RISC. Rozpatrzymy najczęściej używane struktury procesorów RISC.

7.3.1. Jednocyklowy procesor PIC

Rysunek 7.4 przedstawia uproszczoną strukturę jednocyklowego (Single-Cycle) procesora PIC. Procesor zawiera następujące główne elementy: *licznik rozkazów* lub *licznik programu* (program counter – PC), *pamięć instrukcji* (Instruction Memory – IM), *plik rejestrów* (Register File – RF), *jednostkę arytmetyczno-logiczną* (arithmetic-logic unit – ALU), *akumulator* (accumulator) W i *pamięć danych* (Date Memory – DM). Aby nie zaśmiecać rysunku 7.4, *układ sterowania* (Control Unit), sygnały sterujące i niektóre elementy układów tutaj nie są pokazane.



RYS. 7.4. Uproszczona struktura jednocyklowego procesora PIC

Licznik rozkazów PC wskazuje wykonywane polecenie, tzn. wartość licznika PC jest używana jako adres pamięci instrukcji. Polecenie odczytane z pamięci jest dekodowane: dzielone na osobne pola i używane przez układ sterujący do tworzenia niezbędnych sygnałów sterujących.

Do utworzenia ciągu adresów w pamięci rozkazów służy osobny sumator, którego jedno wejście przyjmuje wartość licznika rozkazów PC, a drugie – stałą 1. W przypadku przejścia pod adres podany w kodzie rozkazu, wartość licznika rozkazów PC można wczytać bezpośrednio z kodu rozkazu. W tym celu na wejściu licznika rozkazów PC zainstalowany jest multiplexer.

Pamięć instrukcji IM jednocyklowego procesora PIC jest w rzeczywistości układem kombinacyjnym, tzn. nie zależy od sygnału synchronizacji. Kiedy na wejściu do pamięci instrukcji z wyjścia licznika rozkazów PC pojawia się nowy adres, pamięć instrukcji na jego wyjściu z niewielkim opóźnieniem tworzy kod maszynowy instrukcji.

Dekodowanie instrukcji w jednocyklowym procesorze PIC jest realizowane poprzez prosty wybór oddzielnych linii magistrali wyjściowej pamięci rozkazów. Dlatego do dekodowania instrukcji nie jest wymagane żadne specjalne urządzenie.

Plik rejestrów zawiera 128 8-bitowych rejestrów ogólnego przeznaczenia. W rzeczywistości plik rejestrów to jednoportowa pamięć typu RAM. Adres pliku rejestrów A pochodzi z kodu instrukcji. Na wyjściu pliku rejestrów tworzona jest

wartość odpowiadająca zawartości jednego z rejestrów, który jest operandem A jednostki arytmetyczno-logicznej ALU. Dane zapisywane w pliku rejestrów są tworzone na wyjściu ALU.

Jednostka arytmetyczno-logiczna ALU przyjmuje argument A albo z pliku rejestrów, albo z kodu instrukcji, a argument B zawsze przyjmuje wartość akumulatora W. Aby wybrać operand A na wejściu ALU, umieszczono multiplexer. Wyjście ALU tworzy wewnętrzną magistralę danych Data Bus procesora PIC. Na wyjściu ALU, tzn. na magistrali danych Data Bus tworzony jest wynik wykonywanej operacji. Operację, jaka ma zostać wykonana przez ALU, określa kod instrukcji, który jest tworzony przez układ sterowania i przekazywany do ALU jako sygnały sterujące (na rysunku 7.4 tego nie pokazano). Wynik wykonanej operacji, w zależności od wartości bitu *d*, można zapisać do pliku rejestrów lub do akumulatora W. W tym celu plik rejestrów ma wejście we zezwolenia zapisu. Podobne wejście ma również akumulator W.

Cechą szczególną naszego procesora PIC jest to, że dane zapisane w pamięci danych DM nie pochodzą z magistrali danych Data Bus, ale z wyjścia akumulatora W. Jak wspomniano wcześniej, wynika to z niewielkich rozmiarów kodu instrukcji procesora PIC, w którym można umieścić tylko jeden adres: albo adres rejestru ogólnego przeznaczenia w pliku rejestrów (aby wskazać operand pewnej operacji), albo adres miejsca w pamięci danych (aby zapisać wynik operacji). Dlatego też wymiana danych w naszym procesorze PIC jest możliwa tylko między pamięcią danych a akumulatorem W. W tym przypadku adres komórki pamięci jest określany w kodzie instrukcji, a zapisywane dane zawsze znajdują się w akumulatorze W. Podobnie, podczas odczytu danych z pamięci, adres komórki określany jest w kodzie instrukcji, a dane są zawsze ładowane do akumulatora W. W tym celu wyjście pamięci danych jest połączone z wejściem akumulatora W poprzez multiplexer. Multiplexer na wejściu akumulatora W określa, czy dane w akumulatorze pochodzą z wyjścia ALU, czy też z wyjścia pamięci danych.

Układ sterowania procesora jednocyklowego jest układem kombinacyjnym (nie automatem stanu), ponieważ musi generować sygnały sterujące w jednym cyklu zegarowym, tzn. nie zależy od sygnału synchronizacji.

Licznik rozkazów PC jest zwykłym rejestrem i zależy od sygnału zegara. Plik rejestrów, pamięć danych i akumulator W również zależą od sygnału zegara.

Główną cechą struktury procesora pokazanego na rysunku 7.4 jest to, że każdą operację można wykonać w jednym cyklu zegara. Dlatego konstrukcja na rysunku 7.4 nazywana jest *procesorem jednocyklowym* (single-cycle CPU).

7.3.2. Wielocyklowy procesor PIC

Szybkość procesora jednocyklowego zależy od czasu wykonywania najwolniejszej operacji, która może być znacznie dłuższa niż czas wykonywania większości innych operacji. Dlatego też, aby zwiększyć szybkość działania procesora, zwykle dzieli się cykl wykonywania operacji na *etapy* lub *fazy* (stages). Wykonanie każdego etapu zajmuje

mniej czasu niż najwolniejsza operacja. Może to zwiększyć częstotliwość procesora. Tak budowane są *procesory wielocyklowe* (multiple-cycle CPU).

W przypadku większości procesorów wykonanie instrukcji jest zwykle podzielone na cztery etapy:

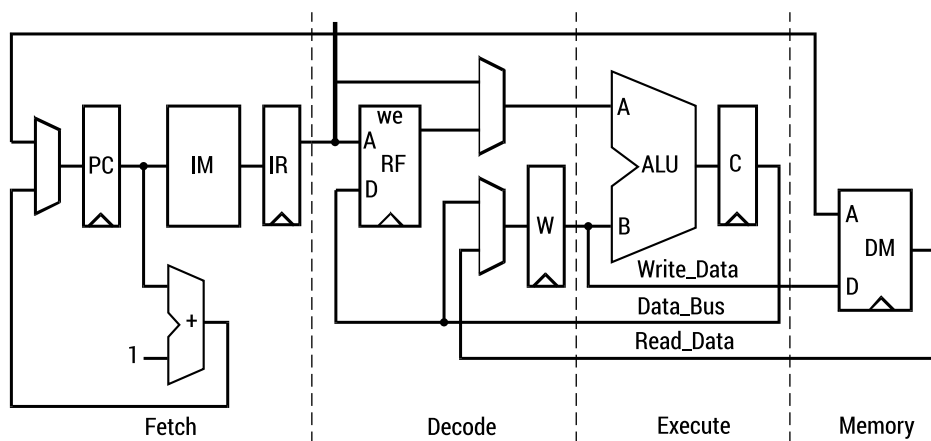
- *wybór instrukcji* (Fetch);
- *dekodowanie instrukcji* (Decode);
- *wykonanie instrukcji* (Execute);
- *odczyt lub zapis pamięci* (Memory).

W fazie Fetch polecenie jest odczytywane z pamięci instrukcji. W niektórych implementacjach na tym etapie instrukcja może być również przechowywana w oddzielnym *rejestrze instrukcji* (Instruction Register – IR). Na etapie dekodowania Decode instrukcja jest „odszyfrowywana”, podczas gdy urządzenie sterowania generuje niezbędne sygnały sterujące do wykonania instrukcji, a także wybierane są operandy z pliku rejestrów. Na etapie wykonania Execute jednostka arytmetyczno-logiczna ALU wykonuje określoną operację na podstawie kodu operacji i sygnałów sterujących, generowanych przez układ sterowania. W etapie Memory zapisuje się zawartość jednego z rejestrów z pliku rejestrów do pamięci danych lub ładuje dane z pamięci do jednego z rejestrów.

W przypadku procesorów z architekturą zestawu instrukcji rejestr-rejestr, może być dodany etap *zwrotnego zapisywania danych do pliku rejestrów* (Writeback).

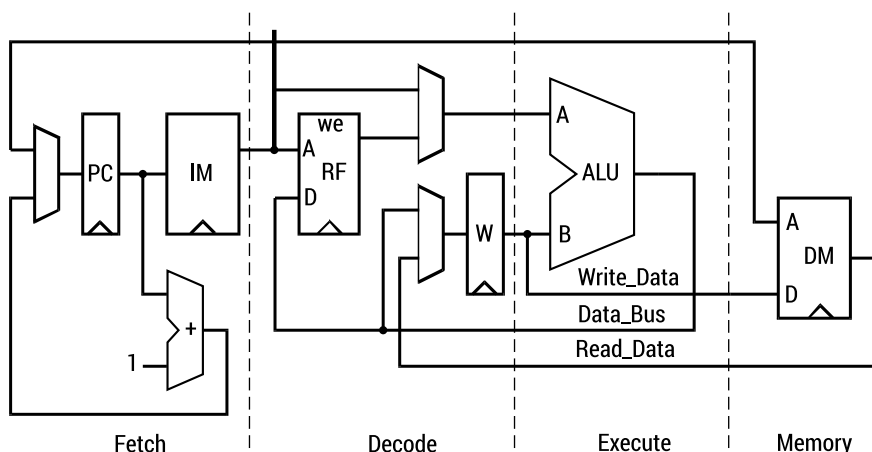
Na pierwszy rzut oka może się wydawać, że etapów jest zbyt wiele. Ma to na celu skrócenie czasu potrzebnego do wykonania każdego etapu. Należy zauważyć, że w niektórych procesorach liczba etapów wykonywania instrukcji może sięgać nawet kilkadziesiątu. W naszym przypadku liczymy na czterokrotny wzrost wydajności.

Rysunek 7.5 przedstawia tradycyjną strukturę procesora wielocyklowego, która różni się od struktury procesora jednocyklowego obecnością rejestru instrukcji IR na wyjściu pamięci instrukcji, a także rejestru C na wyjściu ALU.



RYS. 7.5. Tradycyjna struktura procesora wielocyklowego

Zauważmy, że rejestr instrukcji IR w układzie pokazanym na rysunku 7.5 jest zbędny. Można uczynić pamięć instrukcji synchroniczną, wtedy rejestr IR nie jest wymagany. Rejestr C na wyjściu ALU jest również zbędny, ponieważ jego funkcję pełni akumulator W. Uwzględniając powyższe uwagi, strukturę wielocyklowego procesora PIC pokazano na rysunku 7.6.



RYS. 7.6. Zaawansowana struktura wielocyklowego procesora PIC

Struktura wielocyklowego procesora PIC z rysunku 7.6 jest bardzo podobna do struktury jednocyklowego procesora z rysunku 7.4. Jedyna różnica polega na tym, że pamięć instrukcji IM jest synchroniczna.

Należy zwrócić uwagę, że etap Memory jest wykonywany tylko podczas zapisywania i odczytywania danych do i z pamięci danych DM, tzn. podczas uruchamiania instrukcji **sw** i **lw**. W przypadku wszystkich innych instrukcji procesora PIC wystarczy wykonać pierwsze trzy etapy: pobieranie Fetch, dekodowanie Decode i wykonywanie Execute.

Inkrementacja licznika rozkazów PC może być wykonana w dowolnym etapie, w zależności od wykonywanej instrukcji.

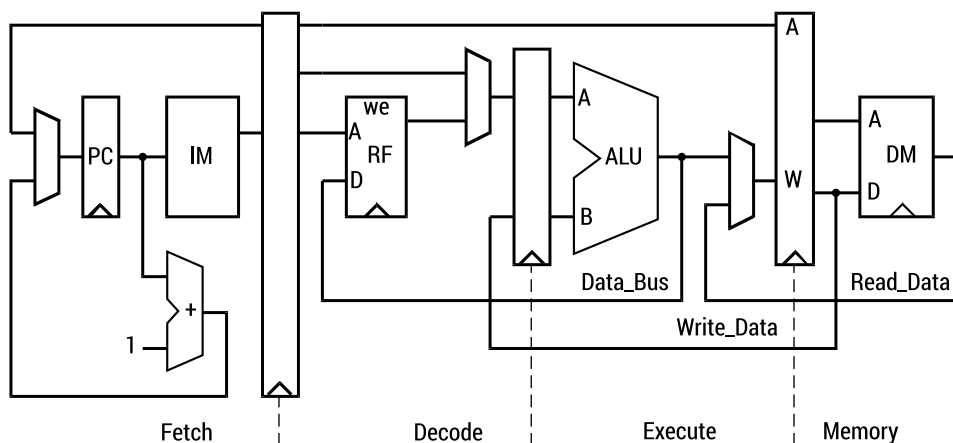
7.3.3. Potokowy procesor PIC

W procesorze wielocyklowym, na każdym etapie użyteczne działania są wykonywane tylko przez te elementy struktury, które odpowiadają danemu etapowi, a pozostałe elementy konstrukcji nie wykonują żadnej użytecznej pracy.

Istnieje możliwość takiego zorganizowania pracy procesora wielocyklowego, aby w każdym cyklu taktowania elementy konstrukcji każdego stopnia wykonywały użyteczne czynności. W tym celu wyniki każdego etapu są przechowywane w rejestrach pośrednich. Po wykonaniu określonej czynności elementy struktury etapu w kolejnym cyklu mogą wykonać pożyteczną pracę dla następnej instrukcji.

W związku z tym możliwe jest wykonywanie równolegle tylu instrukcji, ile jest cykli procesora wielocyklowego. Taki procesor nazywany jest *procesorem potokowym* (pipeline CPU).

Strukturę procesora potokowego pokazano na rysunku 7.7. W dużej mierze pokrywa się ona ze strukturą procesora wielocyklowego. Jedyna różnica polega na tym, że pomiędzy elementami każdego etapu wstawione są rejestry, w których przechowywane są wyniki pośrednie wykonania instrukcji.

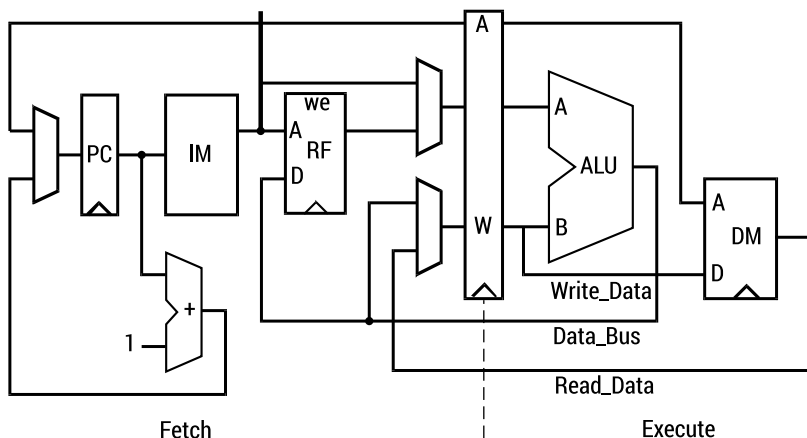


RYS. 7.7. Struktura potokowego procesora PIC z czterema etapami

Wydajność procesora potokowego w naszym przypadku jest prawie czterokrotnie wyższa niż procesora wielocyklowego. Stąd chęć inżynierów do zwiększania liczby cykli procesora wielocyklowego. Słowo „prawie” uwzględnia fakt, że pierwsze cztery cykle ładują potok, a ostatnie cztery – rozładowują go.

Zauważmy, że potok działa dobrze tylko wtedy, gdy instrukcje wykonywane są sekwencyjnie. Jeśli wartość licznika rozkazów PC ulegnie zmianie (na przykład, gdy polecenie zostanie pominięte lub nastąpi skok bezwarunkowy, wywołanie podprogramu, powrót z podprogramu), proces ładowania potoku musi zostać powtórzony.

Można również zauważyć, że w przypadku naszego procesora PIC niektóre etapy potoku są zbędne. Na przykład w przypadku większości instrukcji wykonywane są tylko etapy Fetch, Decode i Execute, podczas gdy dla instrukcji **lw** i **sw** wykonywane są tylko etapy Fetch i Memory. Ponadto dekodowanie instrukcji (etap dekodowania Decode) w naszym procesorze PIC odbywa się poprzez prosty wybór oddzielnych linii magistrali wyjściowej pamięci instrukcji i jest realizowane za pomocą połączeń przewodowych, które nie wymagają dodatkowego zegara synchronizacji. Dlatego też można zmniejszyć liczbę etapów procesora potokowego PIC do dwóch etapów (rys. 7.8). Zauważmy, że mikrokontroler PIC16F84A jest mikrokontrolerem potokowym z dwoma etapami wykonywania instrukcji.



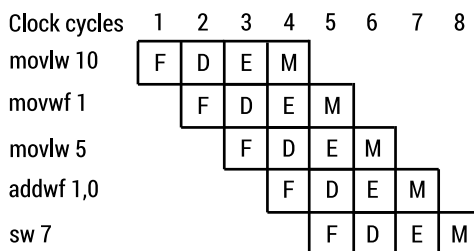
RYS. 7.8. Potokowa struktura procesora PIC z dwoma etapami

Rozpatrzmy potokowe działanie naszego procesora PIC w przypadku wykonania następującego fragmentu kodu:

```
movlw 10      // 10 → W
movwf 1       // W=10 → rf[1]
movlw 5       // 5 → W
addwf 1,0     // rf[1] + W = 15 → W
sw 7          // W → DM[7].
```

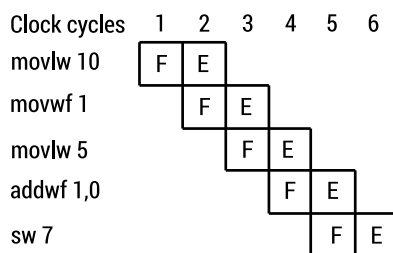
W tym przypadku dodawane są dwie liczby 10 i 5, a wynik 15 jest zapisywany w komórce pamięci danych pod adresem 7.

W przypadku procesora PIC z czterema etapami, działanie procesora potokowego pokazano na rysunku 7.9, gdzie F – etap wybierania instrukcji Fetch; D – etap dekodowania instrukcji Decode; E – etap wykonania instrukcji Execute; M – etap odczytu lub zapisu pamięci Memory.



RYS. 7.9. Działanie potokowego procesora PIC z czterema etapami

W przypadku procesora PIC z dwoma etapami działanie procesora potokowego podczas wykonywania tego samego fragmentu kodu pokazano na rysunku 7.10.

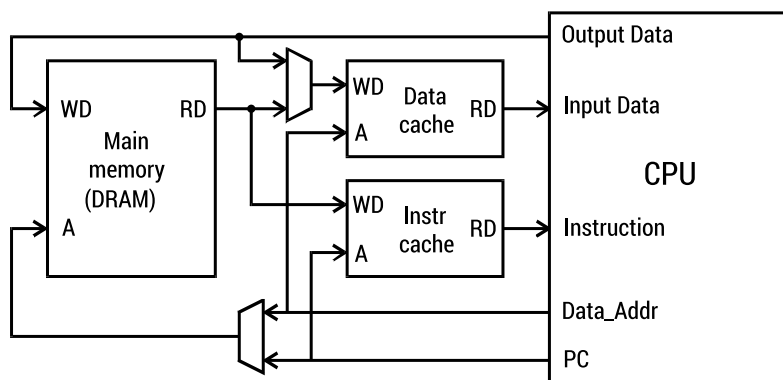


RYS. 7.10. Działanie potokowego procesora PIC z dwoma etapami

Rozpatrywany fragment kodu zawiera pięć instrukcji, do wykonania których procesor jednocyklowy wymaga 5 cykli synchronizacji, a rozpatrywany procesor wielocyklowy – 20. Porównanie rysunków 7.9 i 7.10 pokazuje, że wykonanie tego fragmentu kodu na procesorze potokowym z czterema etapami wymaga 8 cykli synchronizacji, a przy dwóch etapach – 6 cykli (czyli pod względem liczby cykli synchronizacji jest zbliżony do procesora jednocyklowego).

7.3.4. Wykorzystanie pamięci podręcznej

Wąskim gardłem wszystkich procesorów jest wykonywanie operacji związanych z dostępem do pamięci. Faktem jest, że *pamięć statyczna* (Static Random Access Memory – SRAM) jest zbyt droga, aby zbudować z niej całą pamięć systemu wbudowanego, a *pamięć dynamiczna* (Dynamic RAM – DRAM) jest zbyt wolna. Dlatego też w strukturze nowoczesnych procesorów zastosowano pośrednią *pamięć podręczną* (cache), realizowaną w technologii SRAM (rys. 7.11).



RYS. 7.11. Struktura procesora z pamięcią podręczną instrukcji i danych

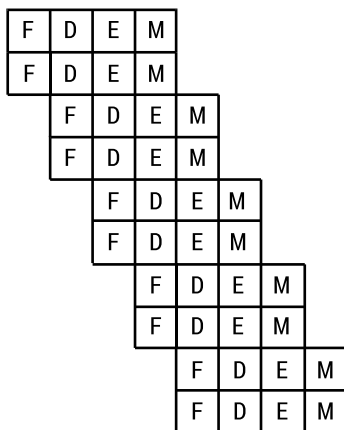
Pamięć podręczna jest również podzielona na *pamięć podręczną danych* (Data cache) i *pamięć podręczną instrukcji* (Instruction cache), które przechowują najczęściej używane dane i instrukcje. Jednak zastosowanie pamięci podręcznej wymaga użycia pewnych specjalnych mechanizmów sterowania, co wyraża się wzrostem złożoności

struktury procesora. Jednak wszystkie nowoczesne procesory mają pamięć podręczną danych i pamięć podręczną instrukcji.

7.3.5. Procesory superskalarne

Procesor superskalarny (superscalar processor) obsługuje *równoległość na poziomie instrukcji* (instruction-level parallelism – ILP). W tym celu wszystkie bloki funkcjonalne procesora potokowego (IM, RF, ALU, DM) powiela się wielokrotnie. W rezultacie staje się możliwe potokowe wykonanie wielu instrukcji.

Na rysunku 7.12 przedstawiono warunkowe działanie procesora superskalarnego z podwojoną liczbą bloków funkcjonalnych. W każdym cyklu zegara można potokowo wykonać dwie instrukcje.



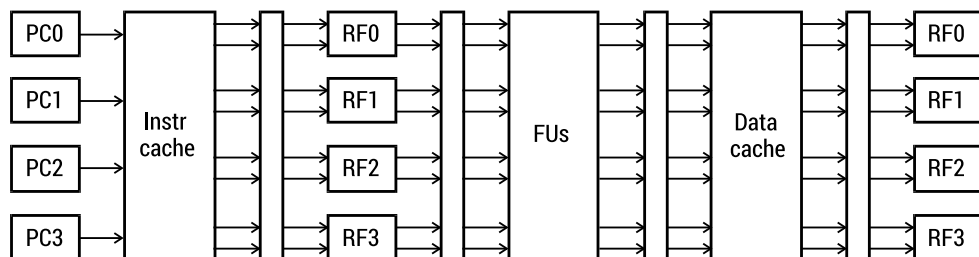
RYS. 7.12. Warunkowe działanie superskalarnego procesora

Implementacja procesora superskalarnego wymaga znacznego obciążenia sprzętowego. Oprócz wielokrotnego powielenia bloków funkcjonalnych dla każdego etapu, konieczne jest również zaimplementowanie sprzętowej *zmiany nazw rejestrów* (register renaming), *stacji rezerwacji* (reservation stations), a także *zmiany kolejności buforów* (reorder buffers).

W przypadku dwukrotnego wzrostu liczby bloków funkcjonalnych (jak na rysunku 7.12) wydajność procesora wzrasta tylko o 20%. Dalszy wzrost wydajności jest zwykle utrudniony przez brak równoległości na poziomie instrukcji ILP. Kompilatory mogą używać kilku metodyk do ulepszenia ILP, takich jak *rozwijanie pętli* (loop-unrolling) i *statyczne planowanie instrukcji* (static instruction scheduling), ale ograniczenia wydajności procesorów superskalarnych pozostają.

7.3.6. Procesory wielowątkowe

Wątek (thread) to sekwencja wykonywalnych instrukcji. Strukturę *procesora wielowątkowego* (multithreading CPU) pokazano na rysunku 7.13.



RYS. 7.13. Uproszczona struktura procesora wielowątkowego

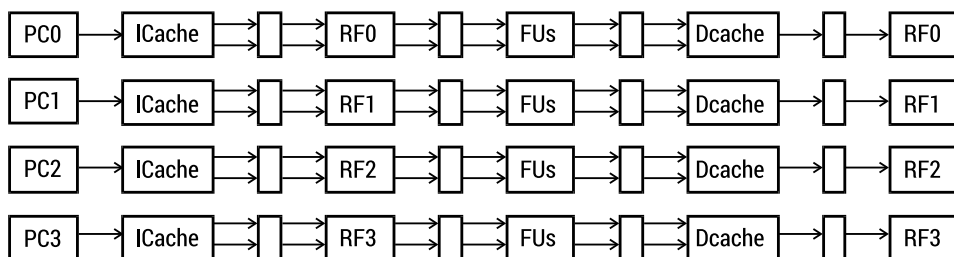
Procesory wielowątkowe wykonują równoległe wiele wątków instrukcji. W procesorach wielowątkowych wydajność jest poprawiana dzięki zastosowaniu *równoległości na poziomie wątków* (thread level parallelism – TLP). Struktura na rysunku 7.13 jest przeznaczona do równoległego wykonywania czterech wątków instrukcji.

Każdy wątek ma oddzielny licznik rozkazów PC i plik rejestrów RF. Jednak pamięci podręczne instrukcji i danych oraz *jednostki funkcjonalne* (functional units – FUs) są współużytkowane przez wszystkie wątki. Współdzielenie pamięci podręcznej i bloków funkcjonalnych zwiększa aktywne wykorzystanie tych komponentów, ale utrudnia sterowanie.

7.3.7. Procesory wielordzeniowe

Procesor wielordzeniowy (multicore CPU) to układ scalony zawierający kilka *zwykłych procesorów* (ordinary CPU), zwanych *rdzeniami* (cores). Rdzenie mogą być procesorami potokowymi, superskalarnymi lub wielowątkowymi.

Rysunek 7.14 przedstawia strukturę procesora wielordzeniowego, który zbudowany jest z czterech procesorów potokowych.



RYS. 7.14. Uproszczona struktura procesora wielordzeniowego

Procesor wielordzeniowy wykorzystuje dwa poziomy pamięci podręcznej: L1 i L2. Każdy rdzeń ma oddzielną pamięć podręczną instrukcji i danych na poziomie L1. Wszystkie rdzenie mogą współużytkować pamięć podręczną L2.

W porównaniu z procesorem wielowątkowym, w procesorze wielordzeniowym pamięć podręczna L1 nie może być współużytkowana przez wszystkie rdzenie. Zmniejsza to wydajność pamięci podręcznej L1, ale ułatwia projektowanie i implementację procesora.

Procesor wielordzeniowy może być traktowany jako *system wieloprocesorowy zaimplementowany na jednym układzie* (chip-multiprocessor).

7.4. Metodyka projektowania procesorów na FPGA

Niedoświadczonemu czytelnikowi może się wydawać, że projektowanie procesora jest niczym magia: pewni bardzo mądrzy ludzie łączą elementy układów i pojawia się procesor. W ogólnym przypadku tak to wygląda. Jeśli natomiast znamy metodologię projektowania procesorów, stworzenie procesora nie wydaje się już magią.

Struktura procesora jest zwykle podzielona na dwie główne części: *jednostkę operacyjną* (datapath) i *jednostkę sterującą* (control unit).

Urządzenie operacyjne wykonuje operacje na danych: przetwarzanie, przechowywanie, transmisję, separację, łączenie itp. Urządzenie operacyjne zawiera rejestry, pamięć, ALU, multiplexery, dekodery i różne elementy układów. W naszym procesorze PIC układ operacyjny wykonuje operacje na bajtach.

Układ sterowania pobiera z pamięci instrukcji kod rozkazu do wykonania i generuje sekwencję sygnałów sterujących dla układu operacyjnego, które sterują etapami wykonywania instrukcji. Sygnały sterujące mogą być sygnałami wyboru multiplexera, zezwoleniem na zapis do rejestrów i pamięci, kodem operacji ALU itp. Zwykle układ sterowania ma postać automatu skończonego, ale dla procesora jednocyklowego układ sterujący jest układem kombinacyjnym.

W ogólnym przypadku procesor można traktować jako bardzo duży automat skończony, w którym pamięć automatu to wszystkie rejestry i cała pamięć procesora. Można również myśleć o procesorze jako o zbiorze oddzielnych, współdziałających ze sobą automatów skończonych.

Najpierw należy określić, które elementy układu określają stan procesora. Procesor można traktować raczej jako złożony układ sekwencyjny i w przeciwieństwie do automatu skończonego, stan procesora jest określany nie tylko przez rejestr pamięci (jak w automacie skończonym), ale przez komponenty, które mogą przechowywać podstawowe informacje. Takimi komponentami układu, w przypadku procesora jednocyklowego, są licznik instrukcji PC, plik rejestrów RF, pamięć instrukcji IM i pamięć danych DM. Reszta układu procesora może również zawierać elementy pamięci takie, jak pamięć automatu skończonego (układu sterowania) lub rejestry pośrednie procesora potokowego. Natomiast elementy te nie przechowują informacji podstawowych.

Po zidentyfikowaniu elementów pamięci procesora, należy dodać inne komponenty do układu procesora. Możliwych jest tutaj kilka podejść. Można dodawać komponenty w celu wykonania każdego etapu działania procesora: pobrania instrukcji, dekodowania instrukcji, wykonania instrukcji, zapisu-odczytu pamięci itp. Komponenty można również dodać do układu procesora w celu wykonywania instrukcji o określonym formacie (dla naszego procesora są to formaty *a*, *b*, *c* i *d*). Ponadto można dodawać komponenty w celu wykonywania określonych instrukcji. W praktyce wszystkie powyższe podejścia są często stosowane jednocześnie lub w różnych kombinacjach. Wszystko zależy od doświadczenia i gustu inżyniera.

Kwestia konstrukcji ALU wymaga osobnego rozważenia. Istnieje kilka podejść. Możliwe jest zaimplementowanie wszystkich operacji na danych (operacji logicznych i arytmetycznych, przesuwania operandów, obliczania adresu następnej instrukcji) w jednej wspólnej jednostce ALU lub przy użyciu oddzielnych elementów układu procesora. Samą ALU można również zaimplementować na kilka sposobów:

- wykonać ręczną syntezę ALU, a następnie opisać jej strukturę w języku Verilog;
- opisać działanie ALU przy użyciu operatorów proceduralnych języka Verilog.

O sposobie budowy ALU do zastosowania w konkretnym przypadku decyduje inżynier na podstawie analizy szybkości i kosztów realizacji ALU, a także całego procesora jako całości.

Istotne miejsce w strukturze układu operacyjnego zajmują multiplexery. Zwykle są one używane do wyboru źródła danych pewnej jednostki funkcjonalnej, na przykład do wyboru argumentów ALU, wyboru wartości licznika rozkazów PC itp. W strukturze układu operacyjnego stosowane są multiplexery magistrali zarówno z dwoma, jak i wieloma wejściami.

Dekodowanie instrukcji można wykonać za pomocą określonego układu lub za pomocą prostych połączeń przewodowych.

Ponadto do układu procesora można wprowadzić oddzielne elementy logiczne, aby wykonywać operacje logiczne na pojedynczych sygnałach. Zbiór wszystkich oddzielnych elementów logicznych procesora nazywa się *logiką sklejającą* (glue logic) [4].

Osobnego rozpatrzenia wymaga kwestia wykonania układu sterującego. Wejściem do układu sterującego jest kod wykonywanej operacji lub kod całego rozkazu, a także sygnały generowane przez układ operacyjny, które informują o przebiegu wykonywania operacji. Przykładami takich sygnałów są: *przeniesienie* (carry out), *przepełnienie* (overflow) i *wynik zerowy* (zero) – generowane przez jednostkę ALU, *przepełnienie stosu* (full) lub *pusty stos* (empty) – generowane przez pamięć stosu itp.

Zwykle urządzenie sterowania jest realizowane jako automat skończony. Wyjątkiem jest układ sterowania procesorem jednocyklowym, który jest układem kombinacyjnym. Liczba stanów automatu skończonego niekoniecznie jest równa liczbie etapów wykonania instrukcji. Liczba ta może być wyższa. Ważne jest tylko, aby pełny cykl wykonania instrukcji (od stanu początkowego automatu skończonego do powrotu do stanu początkowego) był wykonywany w określonej liczbie cykli

zegarowych. W każdym stanie automatu skończonego tworzony jest zestaw wartości sygnałów sterujących, które mają wpływ na ten etap wykonywania instrukcji.

Strukturę i algorytm działania układu sterującego można budować równolegle z tworzeniem struktury układu operacyjnego.

Biorąc pod uwagę powyższe założenia, metodologię projektowania procesorów na FPGA można przedstawić w postaci następującego algorytmu.

Algorytm 7.1. Metodyka projektowania procesorów na FPGA

1. Wyznaczenie elementów stanu procesora: licznik instrukcji PC, plik rejestrów RF, pamięć instrukcji IM i pamięć danych DM.
2. Dodanie elementów do układu operacyjnego w celu realizacji grup instrukcji o określonym formacie (dla procesora PIC są to formaty *a*, *b*, *c* i *d*).
3. Dodanie elementów do układu operacyjnego w celu realizacji określonych etapów wykonywania instrukcji (pobieranie, dekodowanie i odczyt operandów, wykonanie operacji, zapis-odczyt pamięci, zapis zwrotny do pliku rejestrów).
4. Dodanie elementów do układu operacyjnego w celu realizacji indywidualnych instrukcji procesora.
5. Dodanie elementów do układu operacyjnego w celu określenia adresu następnej instrukcji.
6. Projektowanie ALU.
7. Projektowanie układu sterującego.
8. Udoskonalenie układu procesora w celu obsługi sytuacji konfliktowych (dla procesorów potokowych).
9. Opis komponentów procesora w języku Verilog.
10. Opracowanie *modułu do testowania* (testbench) procesora.
11. Przeprowadzenie syntezy i testowanie procesora, a w razie potrzeby, wprowadzenie zmian w projekcie procesora.
12. Koniec.

Powyższa metodyka ma bardzo ogólny charakter i może się różnić w zależności od gustu i doświadczenia inżyniera.

7.5. Wnioski

Procesory zaimplementowane w układach FPGA nazywane są *procesorami wbudowanymi* (embedded processor). W układach FPGA procesor wbudowany może być zaimplementowany „sztywno” (hard), na stałe zajmując część układu scalonego lub może być zaimplementowany „międko” (soft), poprzez konfigurację wewnętrznej struktury FPGA. Przy sztywnej implementacji procesora w strukturze FPGA programowalny układ nazywany jest *systemem w jednym układzie* (System on Chip – SoC).

Architektura zestawu instrukcji (instruction set architecture – ISA) procesora determinuje miejsce argumentów *jednostki arytmetyczno-logicznej* (arithmetic logical unit – ALU). Tradycyjne architektury procesorów to: stosowa, z akumulatorem, rejestr-pamięć i rejestr-rejestr. Większość *komputerów o zredukowanym zestawie instrukcji* (reduced instruction set computer – RISC) ma architekturę rejestr-rejestr. Architektura zestawu instrukcji PIC łączy zalety architektury z akumulatorem i architektury rejestr-rejestr.

Ważne jest to, aby dokładnie przemyśleć zestaw instrukcji przyszłego procesora, ponieważ od tego zależy struktura i złożoność implementacji sprzętowej, a także działanie projektowanego procesora, tzn. złożoność pisania programów.

Główne podstawowe struktury procesorów RISC to:

- procesor jednocyklowy;
- procesor wielocyklowy;
- procesor potokowy.

Aby poprawić wydajność procesorów i znaleźć kompromis między ilością pamięci statycznej i dynamicznej, do procesora wprowadzono *pamięć podręczną* (cache). Pamięć podręczną można podzielić na: *pamięć podręczną danych* (Data cache) i *pamięć podręczną instrukcji* (Instruction cache). Pamięć podręczna może mieć kilka poziomów: L1, L2 itp.

Dalszy wzrost wydajności procesora jest możliwy dzięki budowie procesorów superskalarnych, wielowątkowych i wielordzeniowych.

Struktura procesora jest zwykle podzielona na dwie główne części: *układ operacyjny* (datapath) i *układ sterowania* (control unit).

Układ operacyjny wykonuje operacje na danych: przetwarzanie, transformację, przechowywanie, transmisję, separację, łączenie itp. Układ operacyjny zawiera: rejestry, pamięć, jednostki ALU, multipleksery, dekodery i różne inne komponenty układów.

Urządzenie sterujące pobiera kod instrukcji do wykonania z pamięci instrukcji i generuje sekwencję sygnałów sterujących dla układu operacyjnego, które sterują etapami wykonywania instrukcji. Sygnały sterujące mogą być sygnałami wyboru multipleksorów, zezwolenia zapisu do rejestrów i pamięci, sygnałami kodu operacji ALU itp. Zwykle urządzenie sterowania ma postać automatu skończonego, ale dla procesora jednocyklowego jest ono układem kombinacyjnym.

W ogólnym przypadku procesor można traktować jako bardzo duży automat skończony, w którym pamięć automatu to wszystkie rejestry i cała pamięć procesora. Można również myśleć o procesorze jako o zbiorze oddzielnych, współpracujących ze sobą automatów skończonych. Stan procesora określają następujące elementy struktury:

- *licznik rozkazów* (program counter – PC);
- *plik rejestrów* (register file);
- *pamięć instrukcji* (instruction memory);
- *pamięć danych* (data memory);
- *akumulator* (accumulator), dla architektur z akumulatorem.

Projektowanie struktury jednostki operacyjnej procesora rozpoczyna się od określenia elementów stanu procesora. Następnie dopracowywana jest struktura układu operacyjnego, aby umożliwić realizację określonych instrukcji. Równolegle można kształtować strukturę i algorytm działania układu sterującego.

Do struktury układu operacyjnego można dodawać komponenty do:

- realizacji grupy instrukcji o określonym formacie (dla procesora PIC są to formaty *a*, *b*, *c* i *d*);
- realizacji określonych etapów wykonywania instrukcji (pobieranie, dekodowanie i odczytywanie operandów, wykonanie operacji, zapis-odczyt pamięci, zapis z powrotem do pliku rejestrów);
- realizacji poszczególnych instrukcji procesora;
- określenia adresu następnej instrukcji.

W ogólnym przypadku metodyka projektowania procesorów na FPGA nie jest sformalizowana i może się różnić w zależności od gustu i doświadczenia inżynierów.

8. Projektowanie jednocyklowego procesora PIC

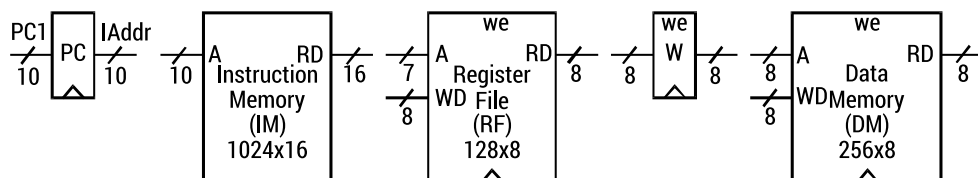
W tym rozdziale przedstawiono proces projektowania jednocyklowego procesora PIC, którego zestaw instrukcji omówiono w rozdziale 7. Szczegółowo opisano projekt układu operacyjnego. Projekt układu arytmetyczno-logicznego i układu sterującego jest rozpatrywany osobno. Podano kody w języku Verilog komponentów układu procesora, a także modułu najwyższego poziomu. Szczególną uwagę zwraca się na debugowanie procesora PIC i ocenę jego wydajności, podano też zalecenia dotyczące zwiększenia wydajności procesora PIC. Omawiany procesor PIC jest łatwy w modyfikacji, którą czytelnik może wykonać samodzielnie.

8.1. Projektowanie układu operacyjnego

Zaprojektujmy urządzenie operacyjne jednocyklowego procesora PIC zgodnie z metodyką przedstawioną w rozdziale 7. W tej metodyce projektowanie procesora rozpoczyna się od zdefiniowania elementów pamięci procesora.

8.1.1. Elementy pamięci procesora

Elementy pamięci procesora PIC to (rys. 8.1): *licznik rozkazów* lub *licznik programu* (Program Counter – PC), *pamięć instrukcji* (Instruction Memory – IM), *plik rejestrów* (Register File – RF), *pamięć danych* (Data Memory – DM) i *akumulator* W.



RYS. 8.1. Elementy pamięci jednocyklowego procesora PIC

Licznik rozkazów PC jest zwykłym 10-bitowym rejestrem, którego wyjście (magistrala IAddr) określa adres wykonywanej instrukcji, a wejściowa magistrala PC1 podaje adres następnej instrukcji. Jeśli instrukcje w procesorze są wykonywane sekwencyjnie (bez skoków), to $PC1 = IAddr + 1$.

Pamięć instrukcji IM w naszym procesorze to *pamięć tylko do odczytu* (read only memory – ROM). W rzeczywistych procesorach pamięć instrukcji to *pamięć o dostępie swobodnym* (random access memory – RAM), tzn. pamięć wielokrotnego zapisu. Właściwość ta jest potrzebna po to, aby system operacyjny mógł ładować nowe programy do pamięci. Pamięć instrukcji, oparta na 10-bitowym adresie podawanym na wejście adresowe A, tworzy na wyjściu RD kod maszynowy instrukcji, która ma być wykonana przez procesor.

W naszym procesorze PIC kod instrukcji ma szerokość 16 bitów, z których tylko 14 najmniej znaczących bitów jest używanych. Dwa bity kodu instrukcji są zarezerwowane dla przyszłych modyfikacji procesora PIC.

W jednocyklowym procesorze PIC pamięć instrukcji IM jest niezależna od sygnału zegarowego. Oznacza to, że po zmianie adresu na wyjściu RD (z pewnym opóźnieniem) zostanie wygenerowany kod maszynowy instrukcji zapisanej pod tym adresem. W rzeczywistości pamięć instrukcji IM jednocyklowego procesora PIC jest układem kombinacyjnym, którego tablica prawdy jest ładowana do pamięci na etapie syntezy z pliku inicjalizacji pamięci.

Należy zauważyć, że każdy układ kombinacyjny można zaimplementować w podobny sposób w układach FPGA.

Plik rejestrów RF składa się ze 128 rejestrów ogólnego przeznaczenia po 8 bitów każdy, do adresowania których wystarczy 7 bitów. Zapis do pliku rejestrów jest wykonywany z wartością 1 na wejściu zezwolenia zapisu *we*. W rzeczywistości plik rejestrów RF to jednoportowa pamięć RAM o konfiguracji 128×8 z wejściem zezwolenia zapisu.

Akumulator W jest zwykłym rejestrem z wejściem zezwolenia zapisu *we*.

Pamięć danych DM jest bardzo podobna do pliku rejestrów RF. Jest to również jednoportowa pamięć typu RAM z wejściem zezwolenia zapisu. Jedyną różnicą jest to, że adres ma szerokość 8 bitów, więc pamięć danych DM ma konfigurację 256×8 (256 słów po 8 bitów w każdym słowie).

Specjalnym elementem pamięci w procesorze PIC, który nie jest często spotykany w architekturze procesora, jest *akumulator W*. Akumulator W odgrywa ważną rolę w naszej architekturze procesora PIC. Zawiera on jeden z argumentów w przypadku operacji dwuargumentowych. Wynik operacji może być przechowywany w akumulatorze W lub w jednym z rejestrów pliku rejestrów RF. Ponadto akumulator W jest używany, gdy procesor PIC współdziela z pamięcią danych DM, ponieważ dane z pamięci DM można załadować tylko do akumulatora W, a dane do pamięci DM można zapisać tylko z akumulatora W.

Dane wszystkich elementów pamięci procesora (licznik rozkazów, pamięć instrukcji, plik rejestrów, pamięć danych i akumulator) odczytywane są w sposób kombinacyjny. Oznacza to, że jeśli na przykład adres na wejściu A pamięci danych zmieni się,

to nowe dane (po pewnym opóźnieniu) pojawiają się na wyjściu RD, przy czym nie jest tu wymagany sygnał zegarowy.

Jednocześnie dane są zapisywane w elementach pamięci na dodatnim zboczu sygnału synchronizacji. W tym celu adres, dane i sygnał zezwolenia na zapis muszą być ustawione w *czasie ustawienia* (setup time) t_{CO} przed nadejściem zbocza zegara i muszą pozostać stabilne przez *czas utrzymywania* (hold time) t_H po nadejściu zbocza zegara.

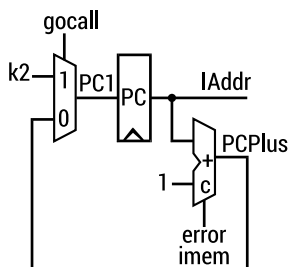
8.1.2. Określenie adresu następnej instrukcji

Bez prawidłowego określenia adresów instrukcji żaden procesor nie będzie działał. Dlatego zaczniemy projektowanie elementów naszego procesora PIC od zdefiniowania adresu następnej instrukcji.

Aby określić adres następnej instrukcji, w przypadku sekwencyjnego wykonywania rozkazów, konieczne jest zwiększenie o jeden wartości na wyjściu licznika rozkazów PC. W tym celu wystarczy wprowadzić do obwodu procesora sumator, którego jeden operand jest wartością licznika PC, a drugi stałą 1. Podobny schemat pokazano na rysunku 7.4.

Aby zaimplementować możliwość skoku do adresu podanego w kodzie instrukcji (dla instrukcji **goto** i **call**), konieczne jest zapewnienie możliwości załadowania określonej wartości do licznika rozkazów PC. Aby to zrobić, na wejściu licznika rozkazów PC instalujemy multiplekser magistrali 2-1 o szerokości 10 bitów, który jest sterowany sygnałem *gocall*. Na jedno wejście multipleksera 2-1 podawana jest wartość stałej *k2* z kodu programu, a na drugie wejście podaje się wartość wyjścia sumatora licznika rozkazów. Sygnał *gocall* jest generowany przez jednostkę sterującą. Podczas realizacji instrukcji **goto** i **call** sygnał *gocall* jest równy 1, we wszystkich innych przypadkach sygnał *gocall* jest równy 0.

Uwzględniając powyższe założenia, schemat generowania adresu następnej instrukcji wygląda tak, jak pokazano na rysunku 8.2.



RYS. 8.2. Schemat tworzenia adresu następnej instrukcji

Zwróćmy uwagę, że w układzie pokazanym na rysunku 8.2 sygnał przeniesienia sumatora sygnalizuje błąd w adresie pamięci instrukcji poprzez generowanie sygnału

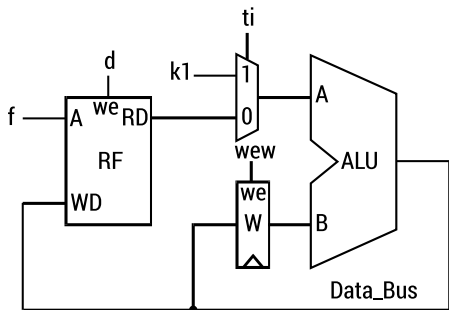
error_imem. W miarę wdrażania kolejnych instrukcji, schemat z rysunku 8.2 zostaje zaktualizowany.

Kontynuujemy projektowanie naszego procesora, dodając komponenty do układu procesora, które są niezbędne do implementacji określonych grup instrukcji.

8.1.3. Realizacja instrukcji transferu danych (instrukcji movlw, movwf i movf)

Grupa instrukcji, bez której żaden procesor nie może się obejść, to *instrukcje transferu* lub *przesyłania danych*. Udział tych instrukcji w programach stanowi ponad 50% całkowitej liczby instrukcji maszynowych.

Aby zaimplementować polecenie **movlw** (zapis stałej *kl* z kodu rozkazu do akumulatora W), należy podać wartość stałej *kl* na wejście A układu ALU, przenieść ją na wyjście ALU i umożliwić zapis do akumulatora W (rys. 8.3).



RYS. 8.3. Fragment schematu układu operacyjnego do implementacji instrukcji przesyłania danych `movlw`, `movwf` i `movf`

Multiplexer na wejściu A ALU (rys. 8.3) sterowany jest sygnałem *ti*, który pozwala na podanie na wejście A ALU wartości stałej *k1* z kodu rozkazu lub wartości rejestru *f* z pliku rejestrów. Zezwolenie na zapis do akumulatora W (rys. 8.3) jest sterowane przez sygnał *wew*, który jest generowany przez układ sterowania i nie zawsze jest inwersją sygnału *d* (zezwolenie na zapis do pliku rejestrów).

Aby zaimplementować polecenie **movwf** (zapis wartości z akumulatora W do pliku rejestrów) należy ustawić wartość adresu rejestru ogólnego przeznaczenia *f* na wejściu adresowym pliku rejestrów, podać wartość akumulatora W na wejście B ALU, przesłać ją na wyjście ALU i umożliwić zapis do pliku rejestrów RF.

Aby zaimplementować instrukcję **movf** (zapis wartości rejestru ogólnego przeznaczenia f do akumulatora W lub z powrotem do rejestru f), konieczne jest ustawienie wartości adresu rejestru f na wejściu adresowym pliku rejestrów RF. Na wejściu A ALU należy ustawić wartość rejestru f , przekazać go na wyjście ALU i zezwolić na zapis do akumulatora W lub do pliku rejestrów RF.

W taki oto sposób wszystkie instrukcje przesyłania danych mogą być realizowane przez schemat z rysunku 8.3 i odpowiednie ustawienie sygnałów sterujących.

8.1.4. Realizacja operacji na bajtach (instrukcji formatu *a*)

Najliczniejszą grupą instrukcji w naszym procesorze są instrukcje formatu *a*, które realizują operacje na bajtach. Przyjrzyjmy się, jak instrukcje w formacie *a* są zaimplementowane w naszym procesorze PIC.

Wszystkie operacje bajtowe są wykonywane w jednostce arytmetyczno-logicznej ALU. W przypadku wykonywania operacji dwuargumentowych, wartość rejestru *f* dociera do wejścia A ALU z pliku rejestrów RF, a wartość akumulatora W jest podawana na wejście B ALU. W przypadku operacji jednoargumentowych do generowania wyniku wykorzystywane jest wejście A lub wejście B ALU. Jeśli żadne działanie na operandach nie jest wymagane, wyjście ALU jest równe wartości argumentu B, tzn. akumulatora W.

Wynik operacji jest generowany na wyjściu ALU, którym jest wewnętrzna magistrala danych Data_Bus w procesorze PIC. Wynik operacji jest zapisywany albo do akumulatora W, albo do rejestru *f* pliku rejestrów RF, w zależności od wartości bitu *d* kodu instrukcji. Gdy *d* = 0, wynik operacji zapisywany jest do akumulatora W, a gdy *d* = 1 – do rejestru *f* pliku rejestrów RF.

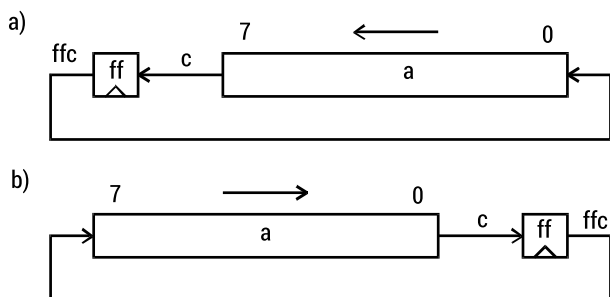
Generalnie naszym zadaniem jest opisanie ALU w języku Verilog w celu implementacji instrukcji formatu *a*. Bezpośredni opis ALU w języku Verilog zostanie podany poniżej. Na tym etapie projektowania procesora PIC konieczne jest określenie struktury układu operacyjnego zapewniającego realizację instrukcji w formacie *a*. W tym celu posłużymy się strukturą z rysunku 8.3.

W taki oto sposób, w celu realizacji najliczniejszej grupy instrukcji formatu *a*, żadne dodatkowe komponenty nie są wprowadzane do układu procesora. Należy również zauważyć, że implementacja instrukcji **addwf**, **andwf**, **clrf**, **clrw**, **comf**, **decf**, **incf**, **iorwf**, **nop**, **subwf** i **xorwf** nie powoduje żadnych specjalnych problemów, ponieważ instrukcje te wykonują tradycyjne operacje arytmetyczne i logiczne, które można łatwo zaimplementować w języku Verilog.

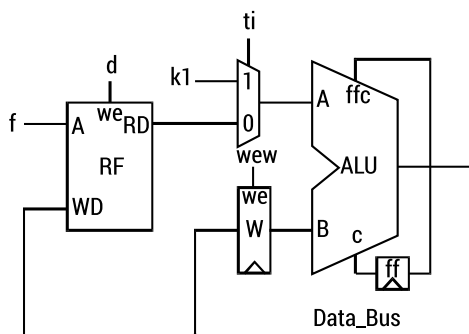
8.1.5. Cykliczne operacje przesunięcia (instrukcje *rlf* i *rff*)

Cechą szczególną instrukcji **rlf** i **rff** jest to, że wykonują one cykliczne przesunięcie zawartości rejestru *f* o jeden bit przez flagę przeniesienia *c* (rys. 8.4). Przesunięcie jednobitowe jest wykonywane w czasie jednego cyklu zegarowego procesora PIC.

Aby zrealizować operacje przesunięcia cyklicznego, sygnał przeniesienia *c* musi być zapamiętany w zewnętrznym przerzutniku *ff*, a wartość na wyjściu przerzutnika *ff* musi zostać podana na wejście *ffc* ALU (rys. 8.5).



RYS. 8.4. Instrukcji przesunięcia cyklicznego przez znak przeniesienia w lewo **rlf** (a) i w prawo **rrf** (b)



RYS. 8.5. Modyfikacja obwodu układu operacyjnego w celu realizacji instrukcji przesunięcia cyklicznego **rlf** i **rrf**

Instrukcję **rlf** przesunięcia cyklicznego w lewo można opisać w języku Verilog następująco:

```
c = a[7];
out = {a[6:0], ffc};
```

Podobnie instrukcję **rrf** przesunięcia cyklicznego w prawo można opisać w języku Verilog następująco:

```
c = a[0];
out = {ffc, a[7:1]};
```

W ten sposób instrukcje przesunięcia cyklicznego **rlf** i **rrf** są realizowane w jednostce ALU, a sprzężenie zwrotne z przerzutnikiem jest dodawane do obwodu ALU w celu zachowania znaku przeniesienia.

8.1.6. Realizacja operacji inkrementacji i dekrementacji z pominięciem następnej instrukcji, jeśli wynik jest zerowy (instrukcje *incfsz* i *decfsz*)

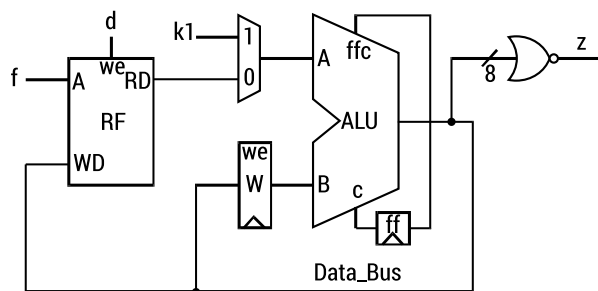
Realizacja instrukcji **incfsz** (**decfsz**) zakłada sekwencyjne wykonanie następujących akcji:

- zwiększyć (zmniejszyć) wartość rejestru *f* o jeden;
- sprawdzić wynik: jeśli wynik wynosi zero, pominąć następny rozkaz;
- zapisać wynik operacji z powrotem do rejestru *f* lub do akumulatora *W*, w zależności od wartości bitu *d* w kodzie instrukcji.

Należy zauważyć, że wszystkie te kroki należy wykonać w ciągu jednego cyklu zegara PIC.

Rozpatrzmy implementację instrukcji **incfsz** (**decfsz**) na poziomie sprzętowym. Aby zaimplementować instrukcję **incfsz** (**decfsz**), konieczne jest przesłanie wartości rejestru *f* na wejście A ALU. Zwiększanie (zmniejszanie) operandu A w jednostce ALU jest wykonywane przez proste dodawanie (odejmowanie) stałej 1.

Aby sprawdzić wynik porównania z zerem, podłączamy 8-wejściową bramkę NOR do magistrali danych *Data_Bus* w celu utworzenia flagi zero *z* (rys. 8.6).

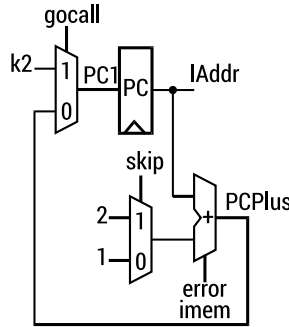


RYC. 8.6. Modyfikacja wyjścia ALU w celu sprawdzenia zerowego wyniku

Aby pominąć następną komendę, licznik rozkazów PC musi być zwiększony nie o stałą 1, jak przy sekwencyjnym wykonywaniu rozkazów, ale o stałą 2. Aby móc zwiększyć licznik rozkazów PC o 1 lub 2, należy zainstalować multiplexer na drugim wejściu sumatora licznika rozkazów (rys. 8.2). Multiplexer wybiera wartość stałej 1 lub 2 w zależności od sygnału sterującego pominięcia *skip*, jak pokazano na rysunku 8.7.

Sygnał pominięcia *skip* będzie generowany przez urządzenie sterowania na podstawie flagi zero *z* i kodu instrukcji **incfsz** (**decfsz**).

W ten oto sposób instrukcje **incfsz** i **decfsz** zostały zaimplementowane poprzez dodanie bramki NOR na wyjściu ALU, w celu wygenerowania flagi zero *z* i poprzez zmodyfikowanie układu generacji adresu dla następnej instrukcji.



RYS. 8.7. Modyfikacja schematu generowania adresu następnej instrukcji w celu umożliwienia pominięcia kolejnej instrukcji

8.1.7. Realizacja operacji bitowych (instrukcji formatu b)

Najpierw przyjrzymy się implementacji instrukcji **bsf**, która ustawia określony bit w rejestrze *f* na jeden. Numer bitu jest wskazywany bitami [9:7] kodu instrukcji. Niech *bn* będzie numerem bitu do ustawienia na jeden i niech, jak poprzednio, wartość rejestru *f* będzie wchodzić na wejście A ALU. Aby wykonać polecenie **bsf**, trzeba stałą 8'b00000001 przesunąć w lewo o *bn* pozycji i wynik dodać logicznie do zawartości rejestru *f*. W języku Verilog będzie to wyglądać tak:

```
out = a | (8'b00000001 << bn);
```

Podobnie, aby wyzerować bit o numerze *bn* w rejestrze *f*, trzeba stałą 8'b00000001 przesunąć w lewo o *bn* pozycji, odwrócić, a następnie pomnożyć logicznie przez zawartość rejestru *f*. W języku Verilog polecenie **bcb** jest implementowane w następujący sposób:

```
out = a & (~(8'b00000001 << bn));
```

Wynik instrukcji **bsf** lub **bcb** jest przechowywany w rejestrze *f*. Jest to realizowane w urządzeniu sterującym poprzez ustawienie sygnału sterującego *d* na 1.

Instrukcje **btfsc** i **btfss** sprawdzają wartość określonego bitu w rejestrze *f* i w zależności od wartości 0 lub 1 na tym bicie, pomijają lub nie pomijają następnej instrukcji. Zauważmy, że wartość bitu w rejestrze *f* nie ulega zmianie.

Aby zaimplementować instrukcje **btfsc** i **btfss**, należy najpierw sprawdzić bit o numerze *bn* w rejestrze *f* (czyli bit *f*[*bn*]). Jeśli podczas wykonywania instrukcji **btfsc**, bit *f*[*bn*] = 0, należy pominąć następne polecenie. Podobnie w przypadku instrukcji **btfss**, należy pominąć następne polecenie, jeśli bit *f*[*bn*] ma wartość 1.

Aby sprawdzić wartość bitu *f*[*bn*], trzeba przesunąć stałą 8'b00000001 w lewo o *bn* pozycji i pomnożyć logicznie przez zawartość rejestru *f*. W języku Verilog można to opisać w następujący sposób:

```
out = a & (8'b00000001 << bn);
```

Kolejne polecenie jest pomijane w urządzeniu sterującym przy użyciu logiki do generowania jedynkowej wartości sygnału pominięcia *skip*. W przypadku instrukcji **btfsc** następne polecenie jest pomijane, jeśli flaga $z = 1$, a w przypadku instrukcji **btfss**, następne polecenie jest pomijane, jeśli flaga $z = 0$.

Wynik wykonania instrukcji **btfsc** lub **btfss**, w przeciwieństwie do instrukcji **bsf** i **bcf**, jest przechowywany w akumulatorze W, przy czym zawartość rejestru *f* nie ulega zmianie.

W ten oto sposób instrukcje operacji bitowych w naszym procesorze PIC zostały implementowane w jednostce ALU i jednostce sterującej bez zmiany struktury jednostki operacyjnej.

8.1.8. Realizacja operacji na stałych (instrukcji formatu c)

Implementacja instrukcji **movlw** do zapisu stałej *k1* do akumulatora została omówiona wcześniej przy implementacji instrukcji przesyłania danych. Realizację wprowadzonych przez nas nowych poleceń **lw** i **sw** rozważymy później.

Realizacja operacji na stałych jest podobna do implementacji operacji na bajtach. Jedyna różnica polega na tym, że wartość stałej *k1*, która znajduje się w kodzie instrukcji, jest podawana na wejście A ALU zamiast wartości rejestru *f*. Z tego powodu, na wejściu A ALU znajduje się multiplexer sterowany sygnałem *ti* generowanym przez jednostkę sterującą. Dla instrukcji, które działają ze stałymi $ti = 1$, w przeciwnym razie $ti = 0$.

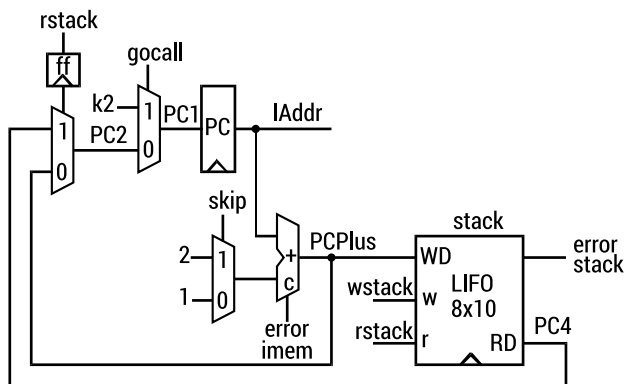
Implementacja instrukcji **retfie** służącej do powrotu z podprogramu ze stałą oraz nowe instrukcje skoku **gotoz** i **gotonz** zostaną omówione poniżej.

Tym sposobem, instrukcje formatu *c* działające na stałych, zostały zaimplementowane w jednostce ALU bez modyfikowania schematu układu operacyjnego.

8.1.9. Realizacja instrukcji skoku do adresu (goto), wywołania podprogramu (call) i powrotu z podprogramu (return, retlw)

Te instrukcje nazywane są instrukcjami *transferu sterowania*. Najprostszą z nich jest polecenie **goto**, które powoduje przeskok pod bezpośredni adres wskazany w kodzie instrukcji przez stałą *k2*. Aby zaimplementować polecenie **goto**, wystarczy załadować wartość stałej *k2* do licznika rozkazów PC. W tym celu na wejściu licznika PC (rys. 8.7) instalowany jest multiplexer, który wybiera wartość PC albo z wyjścia sumatora licznika instrukcji (magistrała PCPlus), albo z kodu rozkazu (stała *k2*). Multiplexer ten sterowany jest przez sygnał *gocall*, który jest ustawiony na jeden dla instrukcji **goto** i **call**.

Pozostałe instrukcje z tej grupy są realizowane za pomocą *stosu* (stack), w sposób pokazany na rysunku 8.8.



RYS. 8.8. Modyfikacja schematu generowania adresu następnej instrukcji w celu realizacji instrukcji **return** i **retlw** dla wyjścia z podprogramu

Stos jest pamięcią typu LIFO (Last In, First Out), tzn. jako pierwsze stos opuszcza słowo, które przyszło jako ostatnie. W przypadku wywołania podprogramu (instrukcja **call**) adres następnej instrukcji należy *odłożyć na stos* (push), a przy powrocie z podprogramu (instrukcje **return** i **retlw**) poprzednio zapisany adres musi zostać *pobrany ze stosu* (pop). Stos posiada magistralę danych wejściowych WD, danych wyjściowych RD oraz dwa sygnały sterujące: *w* – do zapisania adresu instrukcji na stosie oraz *r* – do odczytu adresu instrukcji ze stosu. Ponadto, w przypadku przepełnienia stosu lub próby odczytu z pustego stosu, generowany jest sygnał *error_stack*.

Aby zaimplementować polecenie **call** wywołania podprogramu, wystarczy podłączyć wyjście PCPlus sumatora PC do wejścia stosu WD. Stos jest kontrolowany przez dwa sygnały: *wstack* – zapis na stos i *rstack* – odczyt ze stosu. Sygnały *wstack* i *rstack* są generowane przez układ sterowania. Wyjście stosu jest podłączone do wejścia multipleksera, który wybiera adres następnej instrukcji albo z wyjścia sumatora PC (magistrala PCPlus), albo z wyjścia stosu RD. Wyjściem multipleksera jest magistrala PC2.

Podczas implementacji instrukcji **return** i **retlw** służących do wyjścia z podprogramu pojawiają się problemy. Chodzi o to, że wykonanie instrukcji wyjścia z podprogramu wymaga dwóch cykli zegara sygnału synchronizacji: w pierwszym cyklu zegara instrukcja wyjścia z podprogramu jest odczytywana z pamięci instrukcji, a w drugim cyklu zegara adres powrotu jest ładowany do licznika rozkazów PC.

Problem polega na tym, że w drugim cyklu zegara sygnał *rstack* sterujący stosem w procesorze jednocyklowym zostanie skasowany. Aby zapamiętać wartość sygnału *rstack* i wykorzystać go w drugim cyklu zegara, do układu wprowadza się przerzutnik *ff*, którego wyjście steruje multiplekserem tworzącym magistralę PC2 (rys. 8.8).

Należy zauważyć, że w naszym procesorze PIC wywołanie podprogramu trwa jeden cykl zegara, a wyjście z podprogramu zajmuje dwa cykle zegarowe. W kodzie programu naszego procesora PIC, po poleceniu **return** i **retlw**, aby powrócić z podprogramu, należy zapisać rozkaz **nop** (brak operacji).

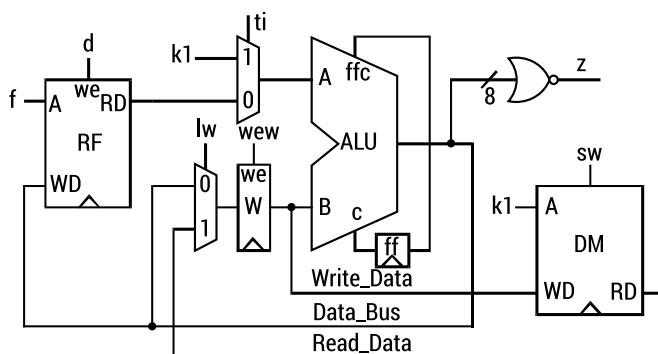
Tak więc, aby zaimplementować polecenie wywołania podprogramu **call** i instrukcje powrotu z podprogramu **return** i **retlw**, do układu generacji adresu następnej

instrukcji wprowadza się stos i multiplexer. Dodatkowo wartość sygnału *rstack* jest przechowywana w przerzutniku.

8.1.10. Realizacja ładowania wartości akumulatora z pamięci danych (instrukcja lw) i zapisu wartości akumulatora w pamięci danych (instrukcja sw)

Aby zaimplementować instrukcję **sw** (zapamiętanie wartości akumulatora W w pamięci danych DM) wystarczy podłączyć wyjście akumulatora W do wejścia WD pamięci danych DM za pomocą magistrali Write_Data i umożliwić zapis do pamięci danych za pomocą sygnału *sw*.

Aby zaimplementować instrukcję **lw** (załadowanie danych z pamięci danych DM do akumulatora W), na wejściu akumulatora W należy zainstalować multiplexer, który będzie wybierał dane ładowane do akumulatora W albo z magistrali Data_Bus, albo z wyjścia RD (magistrala Read_Data) pamięci danych DM (rys. 8.9).



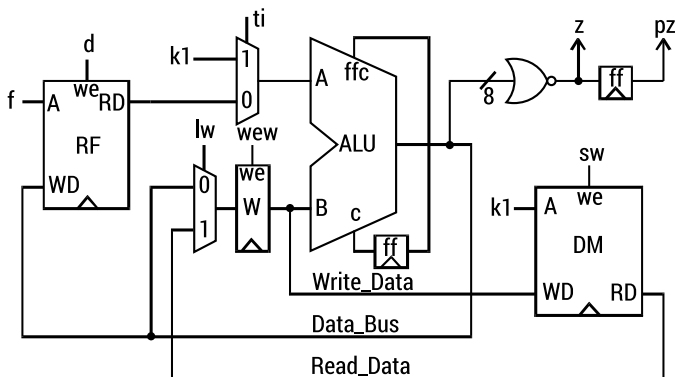
RYŚ. 8.9. Modyfikacja układu operacyjnego w celu wykonania instrukcji załadowania wartości akumulatora z pamięci danych **lw** i zapisania wartości akumulatora w pamięci danych **sw**

Sygnał sterujący multiplexsera *lw* i sygnał zezwolenia na zapis *sw* w pamięci danych DM są generowane przez układ sterowania na podstawie odpowiednich kodów instrukcji.

W ten oto sposób rozkazy **lw** i **sw** są realizowane poprzez wprowadzenie do układu operacyjnego magistral Write_Data i Read_Data, a także multiplexsera na wejściu akumulatora W.

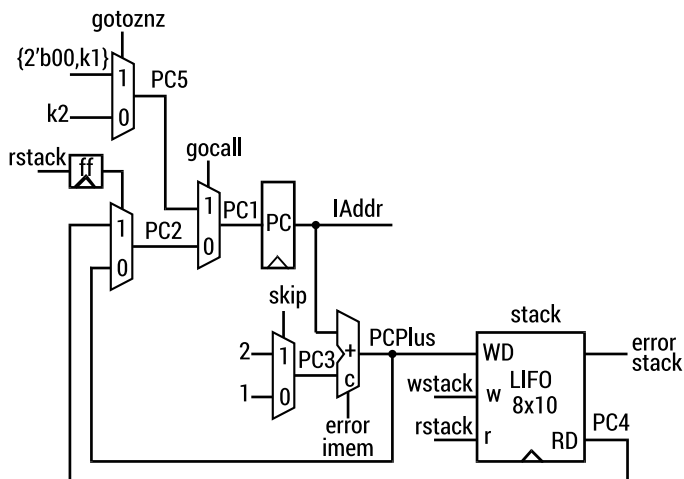
8.1.11. Realizacja skoku do adresu w zależności od wyniku poprzedniej operacji (instrukcje gotoz i gotonz)

Aby zaimplementować instrukcje **gotoz** i **gtonz**, konieczne jest zapamiętanie wartości flagi *z* w przerzutniku, którego wyjście *pz* wskazuje wartość flagi zero z poprzedniej operacji (rys. 8.10).



rys. 8.10. Modyfikacja obwodu wyjściowego ALU w celu zachowania flagi z poprzedniej operacji

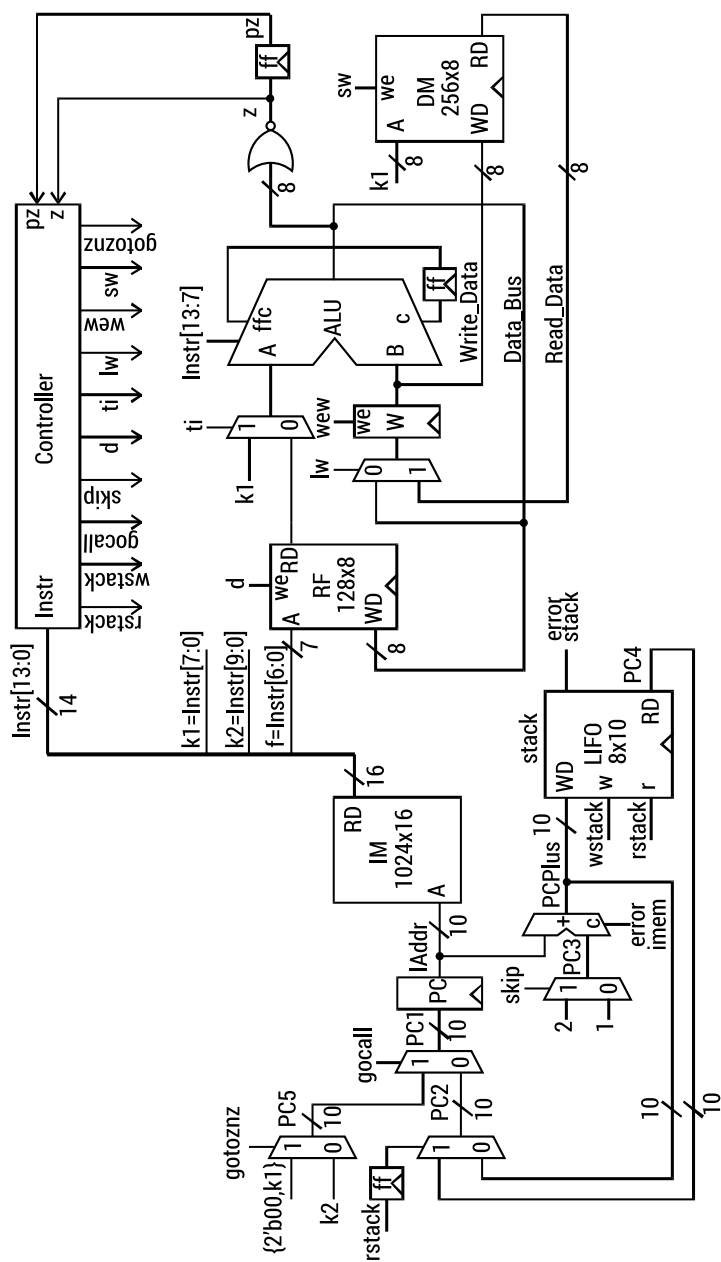
Następnie należy zapewnić ładowanie stałej $k1$ z kodu instrukcji **gotoz** i **gotonz** do licznika rozkazów PC. W tym celu do układu wprowadzany jest multiplexer z wyjściową magistralą PC5, generujący adres następnej instrukcji, który przyjmuje albo stałą $k2$, albo stałą $k1$ rozszerzoną o zera do 10 bitów (rys. 8.11). Wprowadzony multiplexer sterowany jest sygnałem *gotoznz*, który jest generowany w układzie sterującym i przy wykonywaniu instrukcji **gotoz** lub **gotonz** przyjmuje wartość 1.



rys. 8.11. Schemat tworzenia adresu następnej instrukcji przy implementacji instrukcji **gotoz** i **gotonz**

Tak więc, aby zaimplementować instrukcje **gotoz** i **gotonz**, przerzutnik jest ustawiany na wyjściu bramki NOR układu ALU, a do układu generacji adresu następnej instrukcji wprowadzany jest multiplexer.

Na tym kończy się projektowanie układu operacyjnego dla naszego procesora PIC. Ostateczny schemat zaprojektowanego jednocyklowego procesora PIC pokazano na rysunku 8.12.



RYS. 8.12. Schemat jednocyklowego procesora PIC

8.2. Projektowanie układu arytmetyczno-logicznego

Jednostka arytmetyczno-logiczna (ALU) naszego procesora PIC (rys. 8.12) ma wejścia danych A i B, wejście sterujące *in_instr*, które jest częścią kodu wykonywanej instrukcji (*in_instr* = *Instr*[13: 7]) i wejście *ffc* – wyjście przerzutnika, który przechowuje wartość sygnału przeniesienia *c*. ALU będzie dawać wynik na wyjściu *out*, jak również sygnał przeniesienia *c*.

Wejście A ALU przyjmuje pierwszy operand wykonywanej operacji, którym może być wartość rejestru ogólnego przeznaczenia znajdującego się w pliku rejestrów RF lub stała *k1* z kodu instrukcji. Drugim operandem na wejściu B jednostki ALU jest zawsze wartość akumulatora W.

Jeśli ALU nie wykonuje żadnych czynności, to wartość akumulatora W nie powinna się zmieniać. To ostatnie można zaimplementować albo wykorzystując sygnał *wew* zezwolenia zapisu do akumulatora W, albo przekazując wartość z wejścia B jednostki ALU na wyjście *out* i umożliwiając zapis do akumulatora W. Dla naszego ALU przyjmujemy następującą zasadę: jeśli kod wykonywanej operacji nie jest zdefiniowany, to domyślnie, wartość akumulatora W z wejścia B ALU zostanie przekazana na wyjście *out*.

Analiza zestawu instrukcji procesora PIC (tab. 7.3) pokazuje, że bity kodu maszynowego [13:12] dla instrukcji formatu *a* mają wartość 00, formatu *b* – 01, formatu *c* – 11 i formatu *d* – 10. Istnieje również kilka instrukcji, które wyraźnie nie są specyficzne dla żadnego formatu. Operacja, która ma być wykonana dla instrukcji formatu *a* i *c* jest określona przez bity [11:8], formatu *b* – przez bity [11:10], formatu *d* – przez bit [11].

W związku z powyższym opis jednostki ALU procesora PIC w języku Verilog może wyglądać następująco:

```
module alu
#(parameter DATA_WIDTH=8)
(input [DATA_WIDTH-1:0] a,           // a - (f)
 input [DATA_WIDTH-1:0] b,           // b - (w)
 input [6:0] in_instr,               // in_instr = Instr[13:7]
 input ffc,                           // ffc = ffc
 output reg [DATA_WIDTH-1:0] out,     // wynik
 output reg c);                       // przeniesienie carry out

wire [1:0] icode = in_instr[6:5];     // kod grupy instrukcji
wire [3:0] opcode = in_instr[4:1];    // kod operacji
wire [2:0] bn = in_instr[2:0];        // numer bitu

always @(*) begin
    c = 0;                             // resetowanie przeniesienia
    case (icode)
```

2'b00: // instrukcje dla operacji na bajtach

case (opcode)

```
4'b0111: {c,out} = a + b; // addwf
4'b0101: out = a & b; // andwf
4'b0001: out = 8'b0; // clrf, clrw
4'b1001: out = ~a; // comf
4'b0011, // decf
4'b1011: {c,out} = a - 1'b1; // decfsz
4'b1010, // incf
4'b1111: {c,out} = a + 1'b1; // incfsz
4'b0100: out = a | b; // iorwf
4'b1000: out = a; // movf
4'b0000: out = b; // movwf, nop
4'b1101: begin // rlf
    c = a[DATA_WIDTH-1];
    out = {a[DATA_WIDTH-2:0],ffc};
end
4'b1100: begin // rrf
    c = a[0];
    out = {ffc,a[DATA_WIDTH-1:1]};
end
4'b0010: {c,out} = a - b; // subwf
4'b1110: out = {a[3:0],a[7:4]}; // swapf
4'b0110: out = a ^ b; // xorwf
default: out = b;
```

endcase

2'b01: // instrukcje dla operacji na bitach

case (opcode[3:2])

```
2'b00: out = a & (~(8'b00000001 << bn)); // bcf
2'b01: out = a | (8'b00000001 << bn); // bsf
2'b10, // btfsc
2'b11: out = a & (8'b00000001 << bn); // btfss
```

endcase

2'b11: // instrukcje dla operacji na stałych

casex (opcode)

```
4'b111x: {c,out} = a + b; // addlw
4'b1001: out = a & b; // andlw
4'b1000: out = a | b; // iorlw
4'b0000: out = a; // movlw
4'b0001, // lw
```

```

4'b0010: out = b;      // sw
4'b110x: {c,out} = a - b; // sublw
4'b1010: out = a ^ b;   // xorlw
4'b0100: out = a;       // retlw
4'b0101: out = b;       // gotoz
4'b0110: out = b;       // gotonz
default: out = b;

    endcase
default: out = b;
endcase

end
endmodule

```

W powyższym kodzie ALU, szerokość danych jest określana przez parametr `DATA_WIDTH`. Za pomocą niejawnych operatorów ciągłego przypisywania, podczas deklarowania odpowiednich zmiennych z kodu instrukcji *in_instr* jest wyodrębniany kod grupy instrukcji o określonym formacie *icode*, kod operacji do wykonania *opcode* oraz numer bitu *bn* (dla operacji na bitach).

Funkcjonowanie ALU jest opisane pojedynczym blokiem **always**. Na początku bloku **always** sygnał przeniesienia *c* jest resetowany, aby zapobiec automatycznemu wstawianiu zatrasku przez kompilator. Pierwszy rozkaz **case** w bloku **always** sprawdza kod grupy instrukcji. Dla każdej grupy instrukcji mamy oddzielną instrukcję **case** służącą do określenia kodu operacji do wykonania. Większość operacji jest realizowana przy pomocy odpowiednich operatorów języka Verilog. Wyjątkami są omówione wcześniej instrukcje **rlf** i **rrf**, a także operacje bitowe.

W instrukcjach **case**, w przypadku niezgodności między kodem operacji lub kodem grupy instrukcji z którymkolwiek z elementów stałych [6], jako wynik operacji *out* przyjmuje się wartość operandu *B*.

8.3. Projektowanie układu sterowania

W procesorze jednocyklowym PIC wszystkie instrukcje są wykonywane w ramach jednego cyklu zegara (z wyjątkiem instrukcji wyjścia z podprogramu). Z tego powodu, układ sterowania procesora jednocyklowego jest układem kombinacyjnym.

Urządzenie sterujące dowolnego procesora, na podstawie kodu wykonywanej instrukcji i flag generowanych przez urządzenie operacyjne, musi utworzyć wartości sygnałów sterujących, które sterują procesem wykonywania instrukcji. Flagi generowane przez urządzenie operacyjne informują urządzenie sterowania o stanie procesu wykonania instrukcji. W przypadku naszego procesora PIC te flagi to sygnały *z* i *pz*. Sygnał *z* wskazuje zerową wartość wyniku operacji, a sygnał *pz* wskazuje na zerową wartość poprzedniej operacji.

Układ sterowania procesora PIC generuje następujące sygnały sterujące:

- sygnały wyboru dla multiplekserów: *gotoznz*, *gocall*, *skip*, *ti*, *lw*;
- sygnały zezwolenia na zapis do rejestrów i pamięci: *d*, *wew*, *sw*;
- sygnały do zapisu i odczytu stosu: *wstack*, *rstack*.

Opis w języku Verilog układu sterującego dla naszego procesora PIC jest następujący:

```
module controller(  
input [13:0] Instr,           // kod maszynowy instrukcji  
input z, pz,                 // flagi zer  
output rstack, wstack, gocall, skip, // generowane sygnały sterujące  
output reg d,  
output ti, lw, wew,  
output sw,  
output gotoznz);  
  
// deklaracja flag instrukcji  
wire call, goto, retlw, wreturn, decfsz, incfsz, btfsc, btfss, gotoz, gotonz;  
  
// tworzenie sygnałów instrukcji  
assign lw = (Instr[13:8] == 6'b110001);  
assign sw = (Instr[13:8] == 6'b110010);  
assign call = (Instr[13:11] == 3'b100);  
assign goto = (Instr[13:11] == 3'b101);  
assign wreturn = (Instr[13:0] == 14'b00000000001000);  
assign retlw = (Instr[13:10] == 6'b110100);  
assign decfsz = (Instr[13:8] == 6'b001011);  
assign incfsz = (Instr[13:8] == 6'b001111);  
assign btfsc = (Instr[13:10] == 4'b0110);  
assign btfss = (Instr[13:10] == 4'b0111);  
assign gotoz = (Instr[13:8] == 6'b110101);  
assign gotonz = (Instr[13:8] == 6'b110110);  
  
// tworzenie sygnałów sterujących  
assign rstack = (wreturn || retlw);           // instrukcja return lub retlw  
assign wstack = (Instr[13:11] == 3'b100);     // instrukcja call  
assign gocall = (goto || call || gotoznz);  
assign skip = ((z && decfsz) || (z && incfsz) || (z && btfsc) || (!z && btfss));  
assign ti = (Instr[13:12] == 2'b11) ? 1'b1 : 1'b0;
```



```

assign wew = (!d || lw || retlw) && (!sw) && (!gocall) && (!wreturn);
assign gotoznz = ((gotoz && pz) ? 1'b1:
                  ((gotonz && !pz) ? 1'b1 : (1'b0)));

// obliczenie d
always @(*)
    if (Instr[13:12]==2'b00)      d = Instr[7];      // instrukcje formatu a
    else if ((Instr[13:10]==4'b0100) ||           // instrukcja bcf
            (Instr[13:10]==4'b0101))           // instrukcja bsf
        d = 1'b1;
    else                                d = 1'b0;    // inne instrukcje
endmodule

```

W podanym kodzie najpierw określone są sygnały wykonywanych instrukcji (które pokrywają się z mnemonikami instrukcji), a następnie tworzone są wartości sygnałów sterujących.

Sygnał zezwolenia na odczyt stosu *rstack* jest generowany dla instrukcji **return** i **retlw** w celu umożliwienia wyjścia z podprogramu. Sygnał *wstack* umożliwiający zapis na stosie jest generowany dla instrukcji wywołania podprogramu **call**.

Sygnał *gocall* do wyboru wartości magistrali PC1 jest generowany dla instrukcji skoku na adres **goto** i instrukcji wywołania podprogramu **call**, a także po ustawieniu sygnału *gotoznz*.

Sygnał *skip* pominięcia następnego polecenia jest generowany podczas wykonywania instrukcji **incfsz**, **decfsz**, **btfsc** i **btfss**, a także odpowiedniej wartości sygnału *z*.

Sygnał *ti* do wyboru pierwszego argumentu ALU jest generowany dla wszystkich instrukcji formatu *c*.

Sygnał *wew* umożliwiający zapis do akumulatora W jest generowany, gdy sygnał *d* umożliwiający zapis do pliku rejestrów jest zerowy, a także gdy jest wykonywana instrukcja *lw* służąca do załadowania W z pamięci danych i powrotu **retlw** z podprogramu ze stałą w akumulatorze W. Ponadto sygnał *wew* nie powinien być generowany, gdy wykonywane są instrukcje **sw**, **goto**, **call** i **return**.

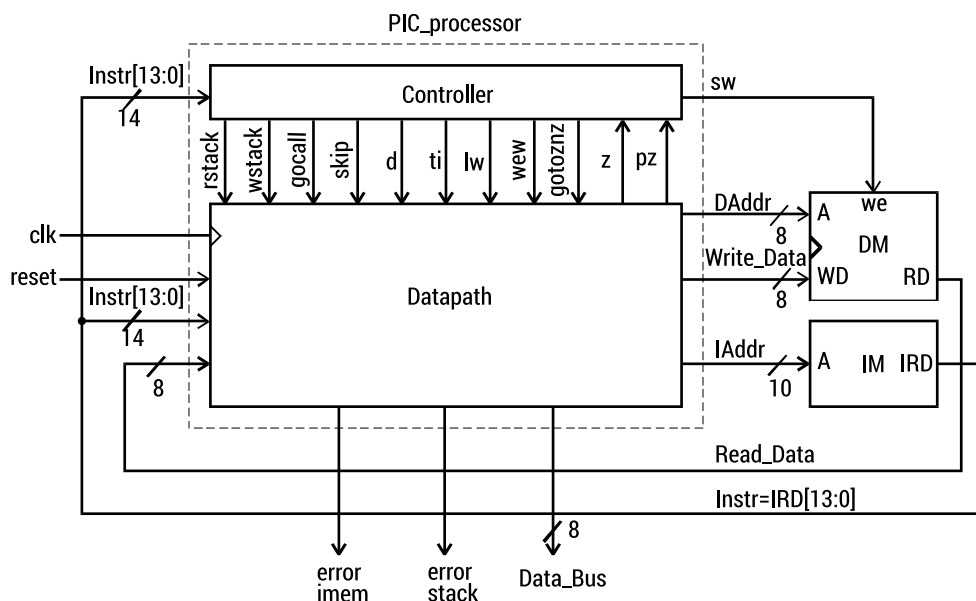
Sygnał *gotoznz* służący do wyboru wartości magistrali PC5 jest generowany dla instrukcji rozgałęzienia **gotoz** w przypadku zerowego wyniku poprzedniej operacji, a także dla instrukcji **gotonz** w przypadku niezerowego wyniku poprzedniej operacji.

Sygnał sterujący *d* umożliwia zapis do pliku rejestrów RF. W przypadku instrukcji formatu *a*, wartość bitu *d* jest wskazywana w bicie [7] kodu instrukcji. Jednak dla instrukcji **bcf** (wyczyść bit w rejestrze *f*) i **bsf** (ustaw bit w rejestrze *f*), wynik operacji również musi zostać zapisany z powrotem do pliku rejestrów. Dlatego w kodzie układu sterującego wartość sygnału sterującego *d* jest obliczana oddzielnie.

8.4. Realizacja procesora jednocyklowego PIC

8.4.1. Struktura realizowanego procesora

W omówionych wcześniej układach procesor PIC był przedstawiany w połączeniu z pamięcią instrukcji IM i pamięcią danych DM. Jednak przy praktycznej realizacji procesora, bloki pamięci są oddzielone od procesora i są urządzeniami zewnętrznymi w stosunku do procesora. Rysunek 8.13 przedstawia strukturę jednocyklowego procesora PIC z zewnętrznymi blokami pamięci instrukcji i pamięci danych.



RYS. 8.13. Uogólniona struktura jednocyklowego procesora PIC z pamięcią instrukcji i pamięcią danych

Struktura pokazana na rysunku 8.13 obejmuje rzeczywisty procesor PIC (**PIC_processor**), który składa się ze sterownika (**Controller**) i układu operacyjnego (**Datapath**). Procesor PIC jest podłączony do pamięci instrukcji DM i IM poprzez następujące interfejsy:

- **DAddr** – magistrala adresowa pamięci danych;
- **Write_Data** – magistrala danych wejściowych pamięci danych;
- **Read_Data** – magistrala danych wyjściowych pamięci danych;
- **IAddr** – magistrala adresowa pamięci instrukcji;
- **Instr** – 14 najmniej znaczących bitów magistrali **IRD** danych wyjściowych pamięci instrukcji – reprezentuje kod maszynowy wykonywanej instrukcji.

Procesor jest sterowany zegarem *clk* i sygnałem resetowania *reset*. Pamięć danych DM jest sterowana tylko sygnałem zegarowym *clk*, na którego narastającym zboczu następuje zapis do pamięci. Pamięć instrukcji IM nie ma sygnałów sterujących.

Procesor generuje następujące sygnały błędów:

- *error_imem* – błąd podczas odczytu instrukcji z pamięci instrukcji;
- *error_stack* – błąd podczas uzyskiwania dostępu do stosu.

Dodatkowo procesor posiada zewnętrzną magistralę danych *Data_Bus*, której wartość odpowiada wewnętrznej magistrali danych *Data_Bus*, tzn. wyjściu ALU.

Kod maszynowy instrukcji *Instr* trafia zarówno do układu operacyjnego *Datapath*, jak i układu sterującego *Controller*. Na podstawie kodu instrukcji *Instr* oraz flag *z* (wynik zerowy) i *pz* (wynik zerowy z poprzedniej operacji), układ sterowania generuje wartości sygnałów sterujących dla układu operacyjnego oraz sygnał *sw* zezwolenia zapisu do pamięci danych DM.

8.4.2. Opis komponentów układu operacyjnego w języku Verilog

Komponenty układu operacyjnego są reprezentowane przez następujące moduły w języku Verilog:

- *mux2_1* – multiplexer magistrali 2-1, szerokość magistrali jest określona przez parametr *WIDTH*;
- *ffc* – przerzutnik typu D z sygnałem synchronizacji i resetowaniem asynchronicznym na zboczu ujemnym;
- *register* – rejestr z sygnałem synchronizacji i zerowaniem asynchronicznym na zboczu ujemnym, ilość bitów rejestru określona jest parametrem *WIDTH*;
- *register_we* – ten sam układ, co *register*, ale dodano wejście zezwolenia zapisu *we*;
- *adder_PCPlus* – sumator dwóch liczb binarnych *a* i *b* z wyjściem przeniesienia *cout*; szerokość wejść i wyjść określa parametr *WIDTH*;
- *regfile* – plik rejestrów w postaci jednoportowej pamięci typu RAM; zapis do pamięci odbywa się na dodatnim zboczu sygnału zegarowego, odczyt odbywa się asynchronicznie przy każdej zmianie adresu; pamięć jest definiowana przez następujące parametry: *ADDR_SIZE* – szerokość magistrali adresowej, *DEPTH* – liczba słów, *WORD_SIZE* – rozmiar słowa w bitach;
- *alu* – jednostka arytmetyczno-logiczna ALU; liczba bitów w operandach *a* i *b*, jak również wynik wyjściowy, są określane przez parametr *DATA_WIDTH*;
- *stos* – pamięć typu LIFO, służąca do implementacji stosu procesora; opisują ją następujące parametry: *IADDR_WIDTH* – szerokość słowa, *STACK_ADDR_WIDTH* – szerokość magistrali adresowej, *STACK_DEPTH* – liczba słów w pamięci (głębokość stosu).

Poniżej znajdują się kody dla powyższych modułów. Opis ALU w języku Verilog został rozpatrzony wcześniej.

```

module mux2_1
#(parameter WIDTH = 8)
(input [WIDTH-1:0] d0, d1,
input s,
output [WIDTH-1:0] y);

assign y = s ? d1 : d0;
endmodule

module ffc(
input clk, reset,
input d,
output reg q);

always @(posedge clk, negedge reset)
    if (~reset)    q <= 1'b0;
    else          q <= d;

endmodule

module register
#(parameter WIDTH = 8)
(input clk, reset,
input [WIDTH-1:0] d,
output reg [WIDTH-1:0] q);

always @(posedge clk, negedge reset)
    if (~reset)    q <= 0;
    else          q <= d;

endmodule

module register_we
#(parameter WIDTH = 8)
(input clk, reset, we,
input [WIDTH-1:0] d,
output reg [WIDTH-1:0] q);

always @(posedge clk, negedge reset)
    if (~reset)    q <= 0;
    else if (we)    q <= d;
    else          q <= q;

```

endmodule

```
module adder_PCPlus
#(parameter WIDTH=8)
(input [WIDTH-1:0] a, b,
output [WIDTH-1:0] y,
output cout);
```

```
assign {cout,y} = a + b;
endmodule
```

```
module regfile
#(parameter ADDR_SIZE=7, DEPTH=128, WORD_SIZE=8)
(input clk, we,
input [ADDR_SIZE-1:0] a,      // adres
input [WORD_SIZE-1:0] wd,    // zapisywane dane
output [WORD_SIZE-1:0] rd);  // odczytane dane
```

```
reg [WORD_SIZE-1:0] RF[DEPTH-1:0]; // macierz pamięci
```

```
always @(posedge clk)
    if (we) RF[a] <= wd;          // zapis
```

```
assign rd = RF[a];              // odczyt
endmodule
```

```
module stack
#(parameter IADDR_WIDTH=10,      // szerokość słowa
STACK_ADDR_WIDTH=3,             // szerokość adresu
STACK_DEPTH=8)                 // głębokość stosu
(input [IADDR_WIDTH-1:0] data_in, // dane zapisywane
input clk, reset,
input rd,wr,                    // odczyt, zapis
output reg [IADDR_WIDTH-1:0] data_out, // odczytane dane
output reg error);
```

```
reg [IADDR_WIDTH-1:0] ram [STACK_DEPTH-1:0]; // pamięć stosu
reg empty,full;                          // flagi: stos pusty, stos pełny
```

```
reg [STACK_ADDR_WIDTH:0] sp;             // wskaźnik stosu
```

```

always @(posedge clk)
    if(~reset) begin
        sp = STACK_DEPTH; // wskaźnik głębokości o 1 większą
        full = sp ? 1'b0 : 1'b1; // definicja flagi
        empty = sp[STACK_ADDR_WIDTH]; // pamięć jest pusta
        data_out = {IADDR_WIDTH{1'b0}}; // na wyjściu 0
    end
    else begin
        if(wr && !full) begin // zapis na stosie
            sp = sp - 1'b1; // modyfikacja wskaźnika
            ram[sp] = data_in; // wykonanie zapisu
            full = sp ? 1'b0 : 1'b1; // definicja flagi
            empty = sp[STACK_ADDR_WIDTH];
        end
        else if(rd && !empty) begin // odczyt ze stosu
            data_out = ram[sp]; // wykonanie odczytu
            sp = sp + 1'b1; // modyfikacja wskaźnika
            full = sp ? 1'b0 : 1'b1; // definicja flagi
            empty = sp[STACK_ADDR_WIDTH];
        end
        // sprawdzanie błędów
        else if (((empty && rd) || (full && wr) || (rd && wr)))
            error = 1'b1;
        else
            error = 1'b0;
    end
end
endmodule

```

Zauważmy, że jednym z parametrów w pliku rejestrów jest DEPTH (liczba słów w pamięci). Pozwala to na utworzenie pliku rejestrów ogólnego przeznaczenia z liczbą rejestrów mniejszych niż dwa do potęgi ADDR_SIZE, na przykład można określić DEPTH = 68, jak w mikrokontrolerze PIC16F84A. To samo dotyczy stosu instrukcji: można zdefiniować liczbę słów w stosie, niekoniecznie równą potęgę dwójki.

8.4.3. Opis układu operacyjnego (datapath)

Układ operacyjny jest opisany przy użyciu strukturalnego stylu opisu języka Verilog. Poniższy kod układu operacyjnego jest podzielony na logikę licznika instrukcji, plik rejestrów, logikę ALU i akumulator W.

```

module datapath
#(parameter
DATA_WIDTH=8,           // szerokość danych
DADDR_WIDTH=8,          // szerokość adresu pamięci danych
INSTR_WIDTH=14,         // rozmiar instrukcji
IADDR_WIDTH=10,         // szerokość adresu pamięci instrukcji
STACK_ADDR_WIDTH=3,     // szerokość adresu stosu
REGFILE_ADDR_WIDTH=7,   // szerokość adresu pliku rejestrów
STACK_DEPTH=(1<<STACK_ADDR_WIDTH), // głębokość stosu
REGFILE_DEPTH=(1<<REGFILE_ADDR_WIDTH)) // głębokość pliku rejestrów

(input clk, reset,           // sygnał synchronizacji i resetowania
input [INSTR_WIDTH-1:0] Instr, // instrukcja
input rstack, wstack, gocall, skip, // sygnały sterujące
input d,                    // kierunek zapisu wyniku ALU
// d = 0 do W, d = 1 do pliku rejestrów
input ti, lw, wew,          // sygnały sterujące
input [DATA_WIDTH-1:0] Read_Data, // dane z pamięci danych
input gotozn,              // sygnał sterujący dla instrukcji
// gotoz i gotonz
output [IADDR_WIDTH-1:0] IAddr, // adres pamięci instrukcji
output [DADDR_WIDTH-1:0] DAddr, // adres pamięci danych
output [DATA_WIDTH-1:0] Write_Data, // dane do pamięci danych
output error_imem, error_stack, // flagi błędów
output z,                  // flaga zero
output [DATA_WIDTH-1:0] Data_Bus, // magistrala danych
output pz);               // flaga zero poprzedniej instrukcji

wire [7:0] k1=Instr[7:0]; // stała dla instrukcji formatu c
wire [9:0] k2=Instr[9:0]; // stała dla instrukcji formatu d

wire [6:0] f=Instr[6:0]; // numer rejestru w pliku rejestrów
wire [IADDR_WIDTH-1:0] PCPlus, PC, PC1, PC2, PC3, PC4, PC5; // magistrale PC
wire [DATA_WIDTH-1:0] A_ALU, RF_out, W, W1; // magistrale W
wire c, wffc, ff_rstack; // sygnały sterujące

// logika licznika rozkazów
register #(IADDR_WIDTH) ex_pc(clk, reset, PC1, PC); // rejestr instrukcji
mux2_1 #(IADDR_WIDTH) ex_pc1(PC2, PC5, gocall, PC1); // multipleksery
mux2_1 #(IADDR_WIDTH) ex_pc2(PCPlus, PC4, ff_rstack, PC2);
mux2_1 #(IADDR_WIDTH) ex_pc3(10'd1, 10'd2, skip, PC3);

```

```

mux2_1 #(IADDR_WIDTH) ex_pc5(k2,{2'b00,k1},gotoznz,PC5);
ffc ex_rstack(clk,reset,rstack,ff_rstack);

// sumator PC
adder_PCPlus #(IADDR_WIDTH) ex_adder(PC,PC3,PCPlus,error_imem);

// stos
stack #(IADDR_WIDTH,STACK_ADDR_WIDTH,STACK_DEPTH)
        ex_stack(PCPlus,clk,reset,rstack,wstack,PC4,error_stack);

// plik rejestrów
regfile #(REGFILE_ADDR_WIDTH,REGFILE_DEPTH,DATA_WIDTH)
        ex_regfile(clk,d,f,Data_Bus,RF_out);

// logika ALU i akumulatora W
// multipleksery
mux2_1 #(DATA_WIDTH) ex_A_ALU(RF_out,k1,ti,A_ALU);
mux2_1 #(DATA_WIDTH) ex_W1(Data_Bus,Read_Data,lw,W1);

// akumulator W
register_we #(DATA_WIDTH) ex_W(clk,reset,wew,W1,Write_Data);

// ALU
alu #(.DATA_WIDTH(DATA_WIDTH))
        ex_alu(.a(A_ALU),.b(Write_Data),.in_instr(Instr[13:7]),
        .ffc(wffc),.out(Data_Bus),.c(c));

assign z = (!(|Data_Bus)); // określenie flagi zera

ffc ex_ffc(clk,reset,c,wffc);    // przerzutnik przeniesienia
ffc ex_ffpz(clk,reset,z,pz);    // przerzutnik zera

assign IAddr=PC;                // definicja adresu pamięci instrukcji
assign DAddr=k1;               // definicja adresu pamięci danych
endmodule

```

Możliwa jest zmiana parametrów struktury układu operacyjnego naszego procesora PIC, co pozwala nam tworzyć procesory o różnych charakterystykach. Służą do tego następujące parametry:

- DATA_WIDTH – szerokość danych;
- DADDR_WIDTH – szerokość adresu pamięci danych;
- INSTR_WIDTH – rozmiar instrukcji;

- IADDR_WIDTH – szerokość adresu pamięci instrukcji;
- STACK_ADDR_WIDTH – szerokość adresu stosu;
- REGFILE_ADDR_WIDTH – szerokość adresu pliku rejestrów;
- STACK_DEPTH – głębokość (liczba słów) stosu;
- REGFILE_DEPTH – głębokość pliku rejestrów (liczba rejestrów).

Dzięki tym parametrom można łatwo tworzyć instancje układów operacyjnych dla różnych procesorów PIC. Należy również zwrócić uwagę, że wprowadzenie parametrów do opisu procesora pozwala uniknąć trudnych do znalezienia błędów związanych z modyfikacją różnych komponentów procesora.

8.4.4. Opis procesora PIC

Po opisanu układu operacyjnego i układu sterującego, opis procesora PIC jest prosty. Zgodnie z rysunkiem 8.13, procesor PIC składa się z jednej instancji układu operacyjnego i jednej instancji układu sterującego. Poniżej znajduje się kod procesora PIC w języku Verilog.

```

module PIC_processor
#(parameter
DATA_WIDTH=8,           // szerokość danych
DADDR_WIDTH=8,          // szerokość adresu pamięci danych
INSTR_WIDTH=14,         // rozmiar instrukcji
IADDR_WIDTH=10,         // szerokość adresu pamięci instrukcji
STACK_ADDR_WIDTH=3,     // szerokość adresu stosu
REGFILE_ADDR_WIDTH=7,   // szerokość adresu pliku rejestrów
STACK_DEPTH=(1<<STACK_ADDR_WIDTH), // głębokość stosu
REGFILE_DEPTH=(1<<REGFILE_ADDR_WIDTH)) // głębokość pliku rejestrów

input [INSTR_WIDTH-1:0] Instr, // instrukcja
input [DATA_WIDTH-1:0] Read_Data, // dane z pamięci danych
output [IADDR_WIDTH-1:0] IAddr, // adres pamięci instrukcji
output [DADDR_WIDTH-1:0] DAddr, // adres pamięci danych
output [DATA_WIDTH-1:0] Write_Data, // dane do pamięci danych
output sw, // instrukcja sw
output error_imem, error_stack, // wskaźniki błędów
output [DATA_WIDTH-1:0] Data_Bus); // magistrala danych

wire gocal, skip, ti, lw, wew, z; // sygnały sterujące

```

```
wire rstack,wstack,d,pz,gotoznz;
```

```
// układ operacyjny
datapath #(.DATA_WIDTH(DATA_WIDTH), .DADDR_WIDTH(DADDR_WIDTH), .INSTR_
WIDTH(INSTR_WIDTH), .IADDR_WIDTH(IADDR_WIDTH), .STACK_ADDR_WIDTH(STACK_ADDR_
WIDTH), .REGFILE_ADDR_WIDTH(REGFILE_ADDR_WIDTH),
.STACK_DEPTH(STACK_DEPTH), .REGFILE_DEPTH(REGFILE_DEPTH))
ex_datapath(.clk(clk), .reset(reset), .Instr(Instr), .rstack(rstack), .wstack(wstack),
.gocall(gocall), .skip(skip), .d(d), .ti(ti), .lw(lw), .wew(wew), .Read_Data(Read_Data),
.gotoznz(gotoznz), .IAddr(IAddr), .DAddr(DAddr), .Write_Data(Write_Data),
.error_imem(error_imem), .error_stack(error_stack), .z(z),
.Data_Bus(Data_Bus), .pz(pz));
```

```
// układ sterowania
controller ex_controller(.Instr(Instr), .z(z), .pz(pz), .rstack(rstack), .wstack(wstack),
.gocall(gocall), .skip(skip), .d(d), .ti(ti), .lw(lw), .wew(wew),
.sw(sw), .gotoznz(gotoznz));
```

```
endmodule
```

Ponieważ układy: operacyjny i sterowania zawierają dużą liczbę portów, ich nazwy są używane do przesyłania sygnałów do instancji modułów. To samo dotyczy przesyłania wartości parametrów do układu operacyjnego. Zauważmy, że przekazywanie sygnałów przez nazwy portów pozwala uniknąć trudnych do znalezienia błędów dla modułów o dużej liczbie portów [6].

Należy również zauważyć, że procesor PIC ma takie same parametry jak układ operacyjny. Umożliwia to tworzenie różnych instancji procesora PIC z różnymi parametrami.

8.4.5. Opis pamięci danych

Pamięć danych jest pamięcią jednoportową z wejściem zezwolenia zapisu *we* oraz asynchronicznym odczytem danych z pamięci. Kod pamięci danych w języku Verilog jest następujący:

```
module dmem
#(parameter ADDR_SIZE=8, WORD_SIZE=8)
(input clk, we, // sygnał synchronizacji, zezwolenia zapisu
input [ADDR_SIZE-1:0] a, // adres
input [WORD_SIZE-1:0] wd, // zapamiętywane dane
output [WORD_SIZE-1:0] rd); // czytelne dane
```

```
reg [WORD_SIZE-1:0] RAM[2**ADDR_SIZE-1:0]; // macierz pamięci
```

```
assign rd = RAM[a]; // czytanie z pamięci
```

```
always @(posedge clk) // zapis do pamięci  
    if (we) RAM[a] <= wd;
```

```
endmodule
```

Dane są zapisywane do pamięci synchronicznie, na narastającym zboczach sygnału zegarowego, przy wysokiej wartości na wejściu zezwolenia zapisu *we*. Natomiast dane są odczytywane z pamięci asynchronicznie, po każdej zmianie adresu. Pamięć danych ma następujące parametry: ADDR_SIZE – rozmiar magistrali adresowej; WORD_SIZE – rozmiar słów pamięci.

8.4.6. Opis pamięci instrukcji

Pamięć instrukcji naszego procesora PIC to pamięć typu ROM, z której można tylko odczytywać dane. Aby opisać pamięć typu ROM w języku Verilog, konieczne jest wpisanie do niej danych początkowych. Można to zrobić w kodzie źródłowym za pomocą instrukcji przypisania lub za pomocą funkcji systemowych **\$readmemb** lub **\$readmemh**, odczytując dane z wcześniej przygotowanego pliku tekstowego.

Inną cechą pamięci instrukcji naszego procesora PIC jest to, że jest ona asynchroniczna, tzn. odczyt danych z pamięci i wygenerowanie wartości na wyjściu następuje w momencie zmiany wartości adresu. Ta ostatnia właściwość jest typowa dla procesorów jednocyfrowych: instrukcja jest odczytywana z pamięci instrukcji w tym samym cyklu zegara, w którym jest wykonywana. W rzeczywistości pamięć instrukcji naszego procesora PIC jest układem kombinacyjnym, którego tablica prawdy jest określona przez zewnętrzny plik tekstowy.

Poniżej znajduje się kod pamięci instrukcji w języku Verilog.

```
module imem
```

```
#(parameter ADDR_SIZE=10, DEPTH=1024, WORD_SIZE=16)
```

```
(input [ADDR_SIZE-1:0] a, // adres
```

```
output [WORD_SIZE-1:0] rd); // czytane dane
```

```
reg [WORD_SIZE-1:0] RAM[DEPTH-1:0]; // macierz pamięci
```

```
initial // inicjalizacja pamięci  
    $readmemb(„imemfile_14.txt”, RAM);
```

```
assign rd = RAM[a]; // odczyt pamięci
```

endmodule

Pamięć instrukcji naszego procesora PIC ma następujące parametry: ADDR_SIZE – rozmiar magistrali adresowej; WORD_SIZE – rozmiar słowa; DEPTH – głębokość (liczba słów).

8.4.7. Opis modułu najwyższego poziomu

Moduł najwyższego poziomu naszego projektu procesora PIC jest w pełni zgodny ze strukturą na rysunku 8.13. Składa się z instancji właściwego procesora PIC oraz instancji pamięci danych i pamięci instrukcji.

Moduł najwyższego poziomu ma tylko dwa wejścia: zegar *clk* i sygnał resetowania *reset*. Aby móc monitorować wykonywanie poszczególnych instrukcji, do portów zewnętrznych wyprowadzono następujące magistrale:

- IAddr – adres pamięci instrukcji;
- DAddr – adres pamięci danych;
- Instr – kod maszynowy wykonywanej instrukcji;
- Write_Data – dane zapisane w pamięci danych;
- Read_Data – dane odczytane z pamięci danych;
- Data_Bus – wewnętrzna magistrala danych procesora PIC, która jest wyjściem ALU.

Dodatkowo na portach zewnętrznych wyświetlane są flagi błędów: *error_imem* – błąd pamięci instrukcji oraz *error_stack* – błąd podczas zapisu lub odczytu stosu.

Moduł najwyższego poziomu ma takie same parametry jak procesor PIC, kod modułu najwyższego poziomu w języku Verilog jest następujący:

```
module single_cycle_PIC_CPU
```

```
#(parameter
```

```
DATA_WIDTH=8,           // szerokość danych
```

```
DADDR_WIDTH=8,          // szerokość adresu pamięci danych
```

```
INSTR_WIDTH=14,          // rozmiar instrukcji
```

```
IADDR_WIDTH=10,          // szerokość adresu pamięci instrukcji
```

```
STACK_ADDR_WIDTH=3,       // szerokość adresu stosu
```

```
REGFILE_ADDR_WIDTH=7,     // szerokość adresu pliku rejestrów
```

```
STACK_DEPTH=(1<<STACK_ADDR_WIDTH), // głębokość stosu
```

```
REGFILE_DEPTH=(1<<REGFILE_ADDR_WIDTH)) // głębokość pliku rejestrów
```

```
(input clk, reset,          // zegar, reset
```

```
output [IADDR_WIDTH-1:0] IAddr, // adres pamięci instrukcji
```

```
output [DADDR_WIDTH-1:0] DAddr, // adres pamięci danych
```

```

output [DATA_WIDTH-1:0] Write_Data,    // dane zapisywane w pamięć
output [DATA_WIDTH-1:0] Read_Data,     // dane odczytane z pamięci
output error_imem, error_stack,        // flagi błędów
output [DATA_WIDTH-1:0] Data_Bus,     // magistrala danych
output [INSTR_WIDTH-1:0] Instr        // wykonywana instrukcja
);

wire sw;
wire [15:0] IRD;                      // wyjście pamięci instrukcji

// procesor
PIC_processor #( .DATA_WIDTH(DATA_WIDTH), .DADDR_WIDTH(DADDR_WIDTH), .INSTR_
WIDTH(INSTR_WIDTH), .IADDR_WIDTH(IADDR_WIDTH), .STACK_ADDR_WIDTH(STACK_ADDR_
WIDTH), .REGFILE_ADDR_WIDTH(REGFILE_ADDR_WIDTH), .STACK_DEPTH(STACK_DEPTH),
.REGFILE_DEPTH(REGFILE_DEPTH))
ex_PIC_processor(.clk(clk), .reset(reset), .Instr(Instr), .Read_Data(Read_Data),
.IAddr(IAddr), .DAddr(DAddr), .Write_Data(Write_Data), .sw(sw),
.error_imem(error_imem), .error_stack(error_stack), .Data_Bus(Data_Bus));

// pamięć instrukcji
imem #(IADDR_WIDTH,2**IADDR_WIDTH,32'd16) ex_imem(IAddr, IRD);

assign Instr = IRD[13:0];

// pamięć danych
dmem #(DADDR_WIDTH, DATA_WIDTH) ex_dmem(clk,sw,DAddr,Write_Data,Read_Data);
endmodule

```

8.5. Debugowanie procesora jednocyklowego PIC

Zwykle czas, który projektanci spędzają na tworzeniu kodu i debugowaniu złożonego projektu, ma stosunek 10:90 lub nawet 5:95, tzn. 5–10% całkowitego czasu jest poświęcane na tworzenie kodu projektu, a 90–95% na jego debugowanie. Na podstawie tych statystyk możemy wywnioskować, że debugowanie projektu jest najbardziej czasochłonnym procesem.

Nie będziemy tutaj rozwodzić się nad problemami debugowania złożonych projektów realizowanych na FPGA (do tego może nie wystarczyć nawet objętość całej książki), ale skupimy się tylko na debugowaniu projektu naszego procesora PIC.

8.5.1. Debugowanie komponentów układu operacyjnego

Wszystkie podstawowe czynności dotyczące przetwarzania informacji podczas wykonywania każdej instrukcji zachodzą w układzie operacyjnym. Dlatego też elementy składowe układu operacyjnego muszą być szczególnie i dokładnie sprawdzane. Nawet jeśli wcześniej opracowano jakiś węzeł (na przykład przerzutnik, rejestr lub sumator), należy go ponownie przetestować przed użyciem go w urządzeniu operacyjnym.

Aby przetestować określony komponent, należy go sprawdzić pod kątem wszystkich możliwych kombinacji danych wejściowych. Często jest to niemożliwe ze względu na dużą ilość danych wejściowych (na przykład, jak sprawdzić pamięć o szerokości magistrali adresowej 16, 32 lub 64 bitów). W takich przypadkach istnieje możliwość zmniejszenia szerokości danych lub adresu. Do tego celu, w miarę możliwości, można zaprojektować komponenty sparametryzowane (jak w naszym projekcie procesora PIC).

Podczas debugowania komponentów należy zwrócić szczególną uwagę na wyjątki, często określane jako *wartości graniczne* (boundaries). Na przykład w projekcie sumatora wartościami granicznymi są: tworzenie sygnału przeniesienia *cout*, w projekcie stosu – tworzenie sygnału błędu *error_stack* i inne.

Szczególną uwagę należy zwrócić na debugowanie ALU, sprawdzając poprawność wykonania każdej instrukcji i ustawianie flag.

Zachęcamy czytelnika do niezależnego testowania każdego komponentu układu operacyjnego naszego procesora PIC. W tym celu można skorzystać z dowolnych dostępnych narzędzi do symulacji i analizy projektów na FPGA, np. narzędzi systemu Quartus.

8.5.2. Debugowanie układu sterowania

Układ sterowania jednocyklowego procesora PIC jest układem kombinacyjnym, którego wejściem jest magistrala Instr, reprezentująca kod maszynowy wykonywanej instrukcji, a także flagi *z* i *pz*. Tablica prawdy układu sterującego jest przedstawiona w tabeli 8.1, gdzie *x* jest wartością dowolną (don't care), *d* jest wartością bitu Instr[7], *!d* jest odwrotną wartością bitu Instr[7].

TAB. 8.1. Tablica prawdy układu sterującego procesorem PIC

Instrukcja	Kod maszynowy (Instr [13: 0])															z	pz	Sygnały sterujące									
																		rstack	wstack	gocall	skip	d	ti	lw	wew	sw	gotozn
addwf	0	0	0	1	1	1	d	x	x	x	x	x	x	x	x	x	x	0	0	0	0	d	0	0	!d	0	0
andwf	0	0	0	1	0	1	d	x	x	x	x	x	x	x	x	x	x	0	0	0	0	d	0	0	!d	0	0
clrf	0	0	0	0	0	1	1	x	x	x	x	x	x	x	x	x	x	0	0	0	0	1	0	0	0	0	0
clrw	0	0	0	0	0	1	0	x	x	x	x	x	x	x	x	x	x	0	0	0	0	0	0	0	1	0	0

Instrukcja	Kod maszynowy (Instr [13: 0])														z	pz	Sygnały sterujące									
																	rstack	wstack	gocall	skip	d	ti	lw	wew	sw	gotonz
comf	0	0	1	0	0	1	d	x	x	x	x	x	x	x	x	x	0	0	0	0	d	0	0	!d	0	0
decf	0	0	0	0	1	1	d	x	x	x	x	x	x	x	x	x	0	0	0	0	d	0	0	!d	0	0
decfsz	0	0	1	0	1	1	d	x	x	x	x	x	x	x	1	x	0	0	0	1	d	0	0	!d	0	0
incf	0	0	1	0	1	0	d	x	x	x	x	x	x	x	x	x	0	0	0	0	d	0	0	!d	0	0
incfsz	0	0	1	1	1	1	d	x	x	x	x	x	x	x	1	x	0	0	0	1	d	0	0	!d	0	0
iorwf	0	0	0	1	0	0	d	x	x	x	x	x	x	x	x	x	0	0	0	0	d	0	0	!d	0	0
movf	0	0	1	0	0	0	d	x	x	x	x	x	x	x	x	x	0	0	0	0	d	0	0	!d	0	0
movwf	0	0	0	0	0	0	1	x	x	x	x	x	x	x	x	x	0	0	0	0	1	0	0	0	0	0
nop	0	0	0	0	0	0	0	x	x	0	0	0	0	0	x	x	0	0	0	0	0	0	0	1	0	0
rlf	0	0	1	1	0	1	d	x	x	x	x	x	x	x	x	x	0	0	0	0	d	0	0	!d	0	0
rrf	0	0	1	1	0	0	d	x	x	x	x	x	x	x	x	x	0	0	0	0	d	0	0	!d	0	0
subwf	0	0	0	0	1	0	d	x	x	x	x	x	x	x	x	x	0	0	0	0	d	0	0	!d	0	0
swapf	0	0	1	1	1	0	d	x	x	x	x	x	x	x	x	x	0	0	0	0	d	0	0	!d	0	0
xorwf	0	0	0	1	1	0	d	x	x	x	x	x	x	x	x	x	0	0	0	0	d	0	0	!d	0	0
bcf	0	1	0	0	x	x	x	x	x	x	x	x	x	x	x	x	0	0	0	0	1	0	0	0	0	0
bsf	0	1	0	1	x	x	x	x	x	x	x	x	x	x	x	x	0	0	0	0	1	0	0	0	0	0
btfs	0	1	1	0	x	x	x	x	x	x	x	x	x	x	1	x	0	0	0	1	0	0	0	1	0	0
btfs	0	1	1	1	x	x	x	x	x	x	x	x	x	x	0	x	0	0	0	1	0	0	0	1	0	0
addlw	1	1	1	1	1	x	x	x	x	x	x	x	x	x	x	x	0	0	0	0	0	1	0	1	0	0
andlw	1	1	1	0	0	1	x	x	x	x	x	x	x	x	x	x	0	0	0	0	0	1	0	1	0	0
iorlw	1	1	1	0	0	0	x	x	x	x	x	x	x	x	x	x	0	0	0	0	0	1	0	1	0	0
movlw	1	1	0	0	0	0	x	x	x	x	x	x	x	x	x	x	0	0	0	0	0	1	0	1	0	0
lw	1	1	0	0	0	1	x	x	x	x	x	x	x	x	x	x	0	0	0	0	0	1	1	1	0	0
sw	1	1	0	0	1	0	x	x	x	x	x	x	x	x	x	x	0	0	0	0	0	1	0	0	1	0
sublw	1	1	1	1	0	x	x	x	x	x	x	x	x	x	x	x	0	0	0	0	0	1	0	1	0	0
xorlw	1	1	1	0	1	0	x	x	x	x	x	x	x	x	x	x	0	0	0	0	0	1	0	1	0	0
call	1	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	0	1	1	0	0	0	0	0	0	0
goto	1	0	1	x	x	x	x	x	x	x	x	x	x	x	x	x	0	0	1	0	0	0	0	0	0	0
return	0	0	0	0	0	0	0	0	0	0	1	0	0	0	x	x	1	0	0	0	0	0	0	0	0	0
retlw	1	1	0	1	0	0	x	x	x	x	x	x	x	x	x	x	1	0	0	0	0	1	0	1	0	0
gotoz	1	1	0	1	0	1	x	x	x	x	x	x	x	x	x	1	0	0	1	0	0	0	0	0	0	1
gotons	1	1	0	1	1	0	x	x	x	x	x	x	x	x	x	x	0	0	0	1	0	0	0	0	0	1

Aby przetestować układ sterowania, konieczne jest sprawdzenie wartości generowanych sygnałów sterujących dla każdej instrukcji procesora PIC. Zauważmy, że niektóre sygnały wyjściowe układu sterującego zależą od bitu $d = \text{Instr}[7]$ kodu maszynowego.

8.5.3. Debugowanie pamięci danych i pamięci instrukcji

Debugowanie pamięci danych DM nie nastręcza szczególnych trudności, wystarczy upewnić się, że zapisane dane zostały poprawnie odczytane.

Aby sprawdzić pamięć instrukcji IM, musimy najpierw zapisać w niej dane inicjalizacyjne. W tym celu należy przygotować plik tekstowy z kodami maszynowymi wykonywanego programu.

8.5.4. Debugowanie całego projektu procesora PIC

Sensowne jest przetestowanie całego projektu procesora PIC wraz z podłączonymi blokami pamięci danych i instrukcji, tzn. należy przetestować moduł najwyższego poziomu *single_cycle_PIC_CPU*.

Cechą szczególną testowania tego projektu jest to, że kompilator systemu Quartus postrzega pamięć instrukcji jako układ kombinacyjny. Dlatego też kompilator syntezuje projekt w zależności od instrukcji programu zapisanych w pamięci (czyli zestawu kodów maszynowych). W takim przypadku możliwa jest sytuacja, gdy niektóre elementy struktury procesora nie zostaną w ogóle zaimplementowane, a ich porty wejściowe i wyjściowe są automatycznie podłączane do masy przez kompilator.

Innymi słowy, konkretna implementacja procesora PIC na FPGA zależy od programu wykonywanego przez procesor i tylko te komponenty procesora, które są niezbędne do realizacji określonego programu, zostaną zaimplementowane w FPGA. Dlatego należy ponownie skompilować projekt za każdym razem, gdy modyfikujesz plik inicjujący pamięć instrukcji IM. W takim przypadku za każdym razem w FPGA zostanie utworzony inny procesor.

Debugowanie jednocyklowego procesora PIC polega na sprawdzeniu każdej instrukcji procesora. Aby przetestować daną instrukcję niezbędne jest:

- napisanie programu do testowania określonej instrukcji;
- stworzenie odpowiedniego pliku do inicjalizacji pamięci instrukcji;
- skompilowanie i przeprowadzenie symulacji projektu procesora, przeanalizowanie wyników symulacji, wprowadzenie zmian w projekcie, jeśli to konieczne.

Aby uprościć ten proces, czytelnik jest proszony o opracowanie kompilatora języka asemblera tworzącego kody maszynowe dla naszego procesora PIC.

Poniżej znajdują się programy w kodzie maszynowym do testowania niektórych instrukcji w naszym procesorze PIC.

Program do sprawdzania instrukcji **movlw**, **movwf**, **addwf**, **andwf**, **clrw**, **comf**:

imemfile_1

// kod maszynowy	mnemoniki	komentarze
0011000000000011	movlw 3	// 3 → W
0000000010000001	movwf 1	// 3 → rf[1]

0011000000000111	movlw 7 // 7 → W
0000011100000001	addwf 1,0 // (rf[1]) + W = 10 → W
0000000010000010	movwf 2 // 10 → rf[2]
0011000000001001	movlw 9 // 9 → W
0000010100000010	andwf 2,1 // rf[2] & W → W
0000000100000000	clrw // 0 → W
0000100100000001	comf 1,0 // !(rf[1]) → W

Program do testowania instrukcji **decf** i **incf**:

imemfile_2

0011000000001111	movlw 15 // 15 → W
0000000010000001	movwf 1 // W=15 → rf[1]
0000001110000001	decf 1,1 // rf[1] = rf[1] - 1
0000001110000001	decf 1,1 // rf[1] = rf[1] - 1
0000001110000001	decf 1,1 // rf[1] = rf[1] - 1
0000001100000001	decf 1,0 // W = rf[1] - 1
0000101010000001	incf 1,1 // rf[1] = rf[1] + 1
0000101010000001	incf 1,1 // rf[1] = rf[1] + 1
0000101000000001	incf 1,0 // W = rf[1] + 1

Program do testowania instrukcji **nop** i **addwf**:

imemfile_3

0011000000000011	movlw 3 // 3 → W
0000000010000001	movwf 1 // W=3 → rf[1]
0011000000000111	movlw 7 // 7 → W
0000000000000000	nop
0000000000000000	nop
0000000000000000	nop
0000000000000000	nop
0000011100000001	addwf 1,0 // rf[1] + W = 10 → W

Program do sprawdzania instrukcji **rlf**:

imemfile_4

0011000000010111	movlw 23 // 23 → W
0000000010000001	movwf 1 // W=23 → rf[1]
0000110110000001	rlf 1,1 // rf[1] = rf[1] << 1
0000110110000001	rlf 1,1 // rf[1] = rf[1] << 1
0000110110000001	rlf 1,1 // rf[1] = rf[1] << 1
0000110110000001	rlf 1,1 // rf[1] = rf[1] << 1

```

0000110110000001    rlf 1,1        // rf[1] = rf[1] << 1
0000110110000001    rlf 1,1        // rf[1] = rf[1] << 1
0000110110000001    rlf 1,1        // rf[1] = rf[1] << 1
0000110110000001    rlf 1,1        // rf[1] = rf[1] << 1
0000110110000001    rlf 1,1        // rf[1] = rf[1] << 1
0000110110000001    rlf 1,1        // rf[1] = rf[1] << 1
0000110110000001    rlf 1,0        // W = rf[1] << 1

```

Program do weryfikacji instrukcji **rrf**:

imemfile_5

```

0011000000010111    movlw 23 // 23 → W
0000000010000001    movwf 1  // W=23 → rf[1]
0000110010000001    rrf 1,1    // rf[1] = rf[1]>> 1
0000110010000001    rrf 1,1    // rf[1] = rf[1]>> 1
0000110010000001    rrf 1,1    // rf[1] = rf[1]>> 1
0000110010000001    rrf 1,1    // rf[1] = rf[1]>> 1
0000110010000001    rrf 1,1    // rf[1] = rf[1]>> 1
0000110010000001    rrf 1,1    // rf[1] = rf[1]>> 1
0000110010000001    rrf 1,1    // rf[1] = rf[1]>> 1
0000110010000001    rrf 1,1    // rf[1] = rf[1]>> 1
0000110010000001    rrf 1,1    // rf[1] = rf[1]>> 1
0000110010000001    rrf 1,1    // rf[1] = rf[1]>> 1
0000110010000001    rrf 1,0    // W = rf[1]>> 1

```

Program do sprawdzania instrukcji **subwf**, **swpf** i **xorwf**:

imemfile_6

```

0011000000000111    movlw 7  // 7 → W
0000000010000001    movwf 1  // W=7 → rf[1]
0011000000000011    movlw 3  // 3 → W
0000001000000001    subwf 1,0 // rf[1]=7-3 = 4 → W
0000111000000001    swpf 1,0 // swpf(rf[1]) → W
0000011000000001    xorwf 1,0 // rf[1] ^ W → W

```

Program do sprawdzania instrukcji **bcf**, **bsf** i **movf**:

imemfile_7

```

0011000000001111    movlw 15 // 15 → W
0000000010000001    movwf 1  // W=15 → rf[1]
0001000010000001    bcf 1,1    // rf[1][1] = 0
0001000100000001    bcf 1,1    // rf[1][2] = 0
0000100000000001    movf 1,0 // rf[1] → W

```

```

0001010010000001    bsf 1,1        // rf[1][1] = 1
0001010100000001    bsf 1,2        // rf[1][2] = 1
0000100000000001    movf 1,0    // rf[1] → W

```

Program do sprawdzania instrukcji **btfsc** i **btfss**:

imemfile_8

```

0011000000001001    movlw 5    // 5 → W
0000000010000001    movwf 1    // W=5 → rf[1]
0001100100000001    btfsc 1,2    // rf[1][2] = 0 ?
0011000000001010    movlw 10   // 10 → W
0011000000001111    movlw 15   // 15 → W
0001101100000001    btfsc 1,3    // rf[1][3] = 1 ?
0011000000000111    movlw 7    // 7 → W
0011000000000011    movlw 3    // 3 → W

```

Program do testowania instrukcji **decfsz** i **goto**:

imemfile_9

```

0011000000000100    movlw 4    // 4 → W
0000000010000001    movwf 1    // W=4 → rf[1]
0000101110000001    decfsz 1,1 // rf[1] = rf[1] - 1
0010100000000010    goto 2      // → decfsz 1,1
0011000000000111    movlw 15   // 15 → W
00110000000001010    movlw 10   // 10 → W

```

Program do sprawdzania instrukcji **incfsz** i **goto**:

imemfile_10

```

0011000011111101    movlw -3   // -3 → W
0000000010000001    movwf 1    // W=-3 → rf[1]
0000111110000001    incfsz 1,1 // rf[1] = rf[1] + 1
0010100000000010    goto 2      // → incfsz 1,1
0011000000000111    movlw 15   // 15 → W
00110000000001010    movlw 10   // 10 → W

```

Program do sprawdzania instrukcji **call**, **goto**, **iorwf**, **clrf**, **btfss**, **addlw**, **andlw**, **iorlw**, **sublw**, **xorlw**, **return**, **nop**:

imemfile_11

```

0011000000000111    movlw 15   // 15 → W
0000000010000000    movwf 0    // W=15 → rf[0]
00110000000001010    movlw 10   // 10 → W
00100000000000111    call 7     // → iorwf 1,1

```

```

0011000000000011    movlw 3    // 3 → W
0011000000000100    movlw 4    // 4 → W
00101000000010010    goto 18    // → nop
0000010010000001    iorwf 1,1    // rf[1] | W=10 → rf[1]=10
0000000110000001    clrf 1      // rf[1] = 0
0001110100000000    btfss 0,2    // rf[0][2] = 1 ?
00110000000001010    movlw 10   // 10 → W
00110000000001001    movlw 9    // 9 → W
00111110000000011    addlw 3     // 3 + W = 12
00111001000000111    andlw 7     // 7 & W = 4
00111000000000011    iorlw 3     // 3 | W = 7
00111100000001011    sublw 11    // 11 - W = 4 → W
00111010000001101    xorlw 13    // 13 ^ W = 9
00000000000001000    return
00000000000000000    nop

```

Program do sprawdzania instrukcji **sw** i **lw**:

imemfile_12

```

00110000000001010    movlw 10    // 10 → W
00110010000000000    sw 0       // W → DM[0]
00110000000001111    movlw 15    // 15 → W
00110010000000001    sw 1       // W → DM[1]
00110001000000000    lw 0        // d[0]=10 → W
00110001000000001    lw 1        // d[1]=15 → W

```

Program do sprawdzania instrukcji **gotoz** i **gotonz**:

imemfile_13

```

00110000000001010    movlw 10    // 10 → W
00000000100000001    movwf 1    // W=10 → rf[1]
00110000000000101    movlw 5     // 5 → W
00000101000000001    andwf 1,0  // rf[1] & W → W
00110101000001000    gotoz 8    // if (z==1) goto 8 (iorwf 1,0)
00000000000000000    nop
00000000000000000    nop
00000000000000000    nop
00000100100000001    iorwf 1,0  // rf[1] | W → W
00110110000000011    gotonz 3   // if (z==0) goto 3 (andwf 1,0)

```

imemfile_14 – ogólny plik testujący łączący w sobie wszystkie pliki **imemfile_1** - **imemfile_13**.

Uwaga. W celu praktycznego wykorzystania podanych kodów maszynowych nie powinny one zawierać tekstowych mnemoników instrukcji i komentarzy, ponieważ są one tutaj podane tylko w celu lepszego zrozumienia kodów maszynowych.

Ostatnim etapem debugowania procesora jest testowanie procesora w strumieniu zadań. W tym celu opracowanych zostało wiele programów rozwiązujących różne problemy z obszaru problemów aplikacyjnych, do których ten procesor jest przeznaczony. Jeśli istnieją referencyjne zadania aplikacyjne, programy są opracowywane w celu rozwiązania problemów referencyjnych. Uzyskane wyniki są sprawdzane i analizowane. W razie potrzeby przeprowadzana jest modyfikacja projektu procesora.

Ostatnim etapem debugowania jest próbna praca procesora w określonym środowisku.

8.6. Ocena wydajności procesora jednocyklowego

Istnieje wiele sposobów pomiaru wydajności procesora. Tradycyjnie o wydajności (lub szybkości działania) systemu cyfrowego decyduje opóźnienie najdłuższej ścieżki danych od wejścia do wyjścia systemu, zwane także *opóźnieniem dla najgorszego przypadku* (worst-case delay). Aby to zrobić, musimy skorzystać z *analizatora czasowego* (time analyzer), na przykład TimeQuest [9], za pomocą którego można znaleźć najdłuższą ścieżkę danych. Aby zwiększyć wydajność należy zastosować różne rozwiązania układowe lub zmienić strukturę całego systemu. Takie podejście jest raczej przeznaczone w przypadku implementacji procesora w układach ASIC.

W przypadku budowy procesora z elementów o niskim i średnim stopniu skali integracji znane są opóźnienia poszczególnych elementów układu. W tym przypadku najdłuższe opóźnienie jest określane dość łatwo, po prostu poprzez dodawanie opóźnień elementów obwodu, przez które przechodzi najdłuższa ścieżka danych. To podejście również nie jest odpowiednie, ponieważ implementujemy nasz procesor na FPGA, gdzie nie ma elementów małego i średniego stopnia skali integracji.

Spróbujemy ocenić wydajność procesora jednocyklowego, korzystając z narzędzi projektowych dostarczonych przez system Quartus. Sama ocena wydajności procesora daje nam niewiele informacji o tym, jak poprawić tę wydajność. Dlatego ocenimy każdy element obwodu oddzielnie, aby znaleźć elementy, które wprowadzają największe opóźnienie w ścieżce danych procesora. Aby to zrobić, skompilujemy każdy element obwodu procesora osobno i spróbujemy określić główne parametry syntetyzowanego elementu.

Wyniki takich badań przedstawiono w tabeli 8.2, gdzie LE – liczba elementów logicznych wymaganych do realizacji projektu; *Register* – liczba używanych przerzutników; *Pins* – liczba wyprowadzeń (portów) projektu; *Memory bits* – liczba wykorzystanych bitów pamięci we wbudowanych blokach pamięci FPGA; D_{max} – maksymalna liczba warstw logicznych elementów LUT w schemacie projektu; D_{av} – średnia liczba warstw logicznych elementów LUT w schemacie projektu; F_{max} – maksymalna

częstotliwość pracy układu w megahercach (MHz); WCS (worst-case slack) – *zapas czasowy* (slack) ścieżki sygnału dla najgorszego przypadku w nanosekundach (ns).

TAB. 8.2. Wyniki syntezy komponentów procesora

Komponent	LE	Registers	Pins	Memory bits	D_{max}	D_{av}	F_{max}	WCS
mux 2_1	8	0	25	0	1.0	1.0	–	–
ffc	1	1	4	0	0	0	–	–
register	0	8	18	0	0	0	–	–
register_we	8	8	19	0	0	0	–	–
regfile	1330	1024	25	0	7.00	5.43	–	–
adder_PCPlus	9	0	25	0	1.80	1.52	–	–
alu	177	0	33	0	7.70	6.06	–	–
stack	112	97	25	0	3.00	2.31	216	11.891
datapath	1659	1141	71	0	19.10	14.27	75	1.278
controller	20	0	26	0	3.00	2.72	–	–
dmem	2670	2048	26	0	8.00	5.92	–	–
imem	126	0	26	0	4.00	3.66	–	–
PIC_processor	1639	1141	61	0	22.00	16.75	72	1.146
single_cycle_PIC_CPU	2308	1525	60	0	23.90	11.29	67	0.293

Badania elementów układu procesora PIC przeprowadzono z wykorzystaniem systemu Quartus Prime wersja 18.1. Do tego celu wybrana została rodzina FPGA Cyclone IV E, jako najdogodniejsza, z następujących powodów. Liczba n wejść elementów LUT w FPGA Cyclone IV E wynosi 4, co pozwala dokładniej określić koszt realizacji (w układach FPGA z rodziny Cyclone V $n = 6$, koszt implementacji jest determinowany nie przez liczbę elementów logicznych, ale przez liczbę modułów logiki adaptacyjnej – adaptive logic modules – ALM); dla rodziny Cyclone IV E określa się maksymalną (D_{max}) i średnią (D_{av}) liczbę poziomów elementów LUT (D_{max} i D_{av} nie są zdefiniowane dla niektórych rodzin FPGA).

Aby poprawnie określić parametry czasowe, dla każdego projektu zdefiniujemy plik ograniczeń czasowych SDC [9] z podanym okresem głównego sygnału synchronizacji, a także wirtualnymi sygnałami synchronizacji wejść i wyjść o wartości 20 ns.

Analiza tabeli 8.2 pokazuje, że parametry F_{max} i WCS nie są określone dla większości elementów obwodu. Chodzi o to, że maksymalne opóźnienie, na podstawie którego obliczane jest F_{max} , jest określane przez najgorszą ścieżkę sygnału między dwoma rejestrami. Elementy *register*, *register_we*, *regfile* i *dmem* zawierają rejestry, ale te elementy nie mają ścieżek sygnałów między dwoma rejestrami.

Pośrednio szybkość działania elementów można ocenić na podstawie maksymalnej D_{max} i średniej liczby poziomów logicznych D_{av} elementów LUT w projekcie. Dla tych parametrów najgorszym elementem w urządzeniu operacyjnym jest

alu ($D_{max} = 7,7$, $D_{av} = 6,06$), następnie *regfile* ($D_{max} = 7,0$, $D_{av} = 5,43$), *stack* ($D_{max} = 3,0$, $D_{av} = 2,31$) i *adder_PCPlus* ($D_{max} = 1,8$, $D_{av} = 1,52$). Dla pozostałych elementów układu sterującego $D_{max} = 0$ (tzn. schemat nie zawiera elementów logicznych i jest realizowany tylko przez połączenia przewodowe lub w oddzielnych rejestrach) lub $D_{max} = 1$ (sygnały przechodzą tylko przez jeden poziom LUT, czyli układ ma maksymalną szybkość dla obwodów złożonych z elementów logicznych). W rezultacie urządzenie operacyjne *datapath* ma $D_{max} = 19,1$ i $D_{av} = 14,27$.

Układ sterowania ma $D_{max} = 3,0$ i $D_{av} = 2,72$. Dlatego można założyć, że układ sterowania jest szybszy niż układ operacyjny.

Należy szczególnie zwrócić uwagę na to, że kompilator systemu Quartus zaimplementował wszystkie komponenty pamięci (*regfile*, *dmem* i *imem*) procesora PIC w logice FPGA, bez użycia wbudowanych bloków pamięci FPGA. Wynika to z asynchronicznego trybu odczytu danych z pamięci.

Najgorszym elementem pamięci jest *dmem* ($D_{max} = 8,0$, $D_{av} = 5,92$), czego należało się spodziewać, ponieważ pamięć danych *dmem* jest znacznie większa (2048 bitów) niż pamięć *regfile* lub *imem*.

Aby określić parametry F_{max} i WCS dla wszystkich elementów układu procesora, opracowano nowe projekty, w których ustawiono rejestry na wejściach i wyjściach danych każdego elementu. Przyrostek „_r” został dodany do nazw projektów nowych elementów procesora. Wyniki badania elementów z rejestrami na wejściu i wyjściu danych podano w tabeli 8.3.

TAB. 8.3. Wyniki oceny szybkości komponentów procesora

Komponent	LE	Registers	Pins	Memory bits	D_{max}	D_{av}	F_{max}	WCS
mux 2_1_r (8)	16	24	27	0	1.00	0.49	305	13.346
ffc_r	1	2	4	0	0	0	847	12.136
register_r	8	16	18	0	0	0	432	13.436
register_we_r	8	16	19	0	0	0	468	13.443
regfile_r	17	9	26	1024	1.00	0.40	1420	7.982
adder_PCPlus_r	25	24	27	0	1.80	0.85	406	9.442
alu_r	182	24	35	0	9.30	6.44	133	7.898
stack	112	97	25	0	3.00	2.31	216	11.891
datapath	1659	1141	71	0	19.10	14.27	75	1.278
controller_r	24	15	28	0	3.00	1.91	787	8.461
dmem_r	17	9	27	2048	1.00	0.39	1420	8.248
imem_r	27	11	28	16384	1.00	0.60	801	7.430
PIC_processor	1639	1141	61	0	22.00	16.75	72	1.146
single_cycle_PIC_CPU	2308	1525	60	0	23.90	11.29	67	0.293

Po ustawieniu rejestrów na wejściach i wyjściach danych, parametry D_{max} i D_{av} są znacznie mniejsze dla pliku rejestrów *regfile*, pamięci instrukcji *imem* i pamięci danych *dmem*. Dzieje się tak, ponieważ projekty *regfile_r*, *imem_r* i *dmem_r* są zaimplementowane we wbudowanych blokach FPGA, co potwierdza parametr *Memory bits*.

Najważniejszy parametr w tabeli 8.3 to F_{max} , maksymalna częstotliwość pracy, na podstawie której można określić szybkość działania każdego elementu obwodu. Spośród elementów układu operacyjnego najwolniejszy jest *alu* ($F_{max} = 133$), następnie *stack* ($F_{max} = 216$), multiplexer *mux_2_1_r* ($F_{max} = 305$), *adder_PCPlus* ($F_{max} = 406$), *register* ($F_{max} = 432$) itp. Układ sterowania ma znacznie wyższą wydajność: $F_{max} = 787$. Jednak najwyższą wydajność mają komponenty pamięci zaimplementowane w oddzielnych blokach pamięci, na przykład dla *imem_r* mam $F_{max} = 801$, a dla *dmem_r* i *regfile_r* $F_{max} = 1420$.

Tym niemniej szybkość działania układu operacyjnego ($F_{max} = 75$), procesora PIC ($F_{max} = 72$) i procesora jednocyklowego ($F_{max} = 67$) pozostaje niewielka.

Zatem wąskim gardłem w konstrukcji naszego jednocyklowego procesora PIC pod względem szybkości (tab. 8.2) są następujące komponenty (w malejącej kolejności wpływu): *datapath* ($D_{max} = 19,1$), *dmem* ($D_{max} = 8,0$), *alu* ($D_{max} = 7,7$), *regfile* ($D_{max} = 7,0$), *imem* ($D_{max} = 4,0$), *stack*, *controller* ($D_{max} = 3,0$), *adder_PCPlus* ($D_{max} = 1,8$), *mux_2_1* ($D_{max} = 1,0$).

Pomimo tego, że dla układu sterującego (projekt *controller*) $D_{max} = 3,0$, ma on stosunkowo dużą szybkość ($F_{max} = 787$, tab. 8.3). W przypadku większości projektów wystarczy, aby urządzenie sterowania było szybsze niż urządzenie operacyjne. W naszym przypadku ten warunek jest spełniony.

Szybkość komponentów pamięci (*dmem*, *regfile*, *imem*) można zwiększyć, jeśli są one zaimplementowane w oddzielnych blokach pamięci FPGA (tab. 8.3). Aby to zrobić, konieczne jest ustawienie rejestrów na ich wyjściach danych lub zapewnienie synchronicznego odczytu z pamięci, co jest możliwe w procesorze wielocyklowym.

Szybkość działania układu operacyjnego zależy od jego komponentów składowych: *alu*, *regfile*, *stack*, *adder_PCPlus*, *mux_2_1*. Wąskim gardłem jest tutaj *alu*. Dlatego też, aby zwiększyć szybkość wszystkich procesorów PIC, ważne jest, aby dołożyć wszelkich starań mających na celu poprawienie szybkości *alu*. Aby to zrobić, musimy przeanalizować wydajność każdej operacji wykonywanej w ALU. Należy znaleźć najwolniejsze operacje i spróbować przeprojektować je tak, aby poprawić wydajność. Czytelnik jest proszony o samodzielne wykonanie tych czynności.

Szybkość stosu można zwiększyć, jeśli macierz pamięci jest zaimplementowana w oddzielnym bloku FPGA, a logika stosu (wskaźnik wierzchołka stosu, liczba wpisów, sprawdzanie wypełnienia stosu itp.) jest zaimplementowana oddzielnie przy użyciu logiki FPGA.

Czytelnik jest proszony o przeprojektowanie modułu stosu w celu poprawy jego wydajności.

Niektóre z proponowanych podejść skutkujących zwiększeniem wydajności procesora zostaną wykorzystane w konstrukcji procesora wielocyklowego PIC.

8.7. Modyfikacja procesora PIC

Czytelnik jest proszony o zmodyfikowanie rozpatrywanego procesora PIC poprzez zwiększenie rozmiaru instrukcji do 16 bitów. Spowoduje to zdefiniowanie instrukcji zapisu **sw** i odczytu **lw** pamięci danych w formacie *d*, co z kolei pozwala na użycie stałej *k2* jako adresu pamięci danych DM. W rezultacie pamięć danych DM może być skonfigurowana jako 1024×8 lub nawet 2048×8. Można również zdefiniować kod maszynowy dla instrukcji **gotoz** i **gotonz** w formacie *d*, aby można było nawigować po tych instrukcjach w zakresie adresów od 0 do 2047.

Czytelnik jest również proszony o zaimplementowanie zmodyfikowanego procesora PIC z magistralą danych o szerokości 16, 32 i 64 bitów (to ostatnie odbywa się po prostu przez określenie odpowiedniej wartości parametru DATA_WIDTH w kodzie procesora).

8.8. Wnioski

Projektowanie jednocyklowego procesora PIC rozpoczyna się od zdefiniowania elementów pamięci procesora. Następnie komponenty są wstawiane do układu procesora w celu określenia adresów: następnej instrukcji, instrukcji przesyłania danych, instrukcji formatu *a* i innych grup instrukcji.

Aby zaimplementować największą grupę instrukcji formatu *a*, do układu procesora nie są wprowadzane żadne dodatkowe komponenty. Większość instrukcji PIC jest łatwa do realizacji, ponieważ instrukcje te wykonują tradycyjne operacje arytmetyczne i logiczne, które można łatwo zaimplementować w języku Verilog.

Aby zaimplementować instrukcje przesunięcia cyklicznego **rlf** i **rrf**, w celu zapamiętania sygnału przeniesienia, do układu procesora wprowadzany jest dodatkowy przerzutnik zewnętrzny w stosunku do ALU.

Instrukcje **incfsz** i **decfsz** służące do inkrementacji i dekrementacji z pominięciem następnej instrukcji wymagają wprowadzenia 8-wejściowej bramki NOR do układu procesora, w celu wygenerowania flagi zera *z*, a także zmiany w układzie generowania adresu następnej instrukcji.

Instrukcje formatu *b*, które wykonują operacje na bitach, są w pełni implementowane przez jednostkę ALU i nie wymagają dodatkowych komponentów w układzie procesora.

Instrukcje formatu *c*, które działają na stałych, są również zaimplementowane w jednostce ALU bez konieczności modyfikowania schematu układu operacyjnego.

Instrukcje skoku do adresu (**goto**), wywołania podprogramu (**call**) i powrotu z podprogramu (**return**, **retlw**) wymagają wprowadzenia stosu (pamięci typu LIFO), a także modyfikacji układu generacji adresu kolejnej instrukcji.

W naszym procesorze PIC wywołanie podprogramu trwa jeden cykl zegarowy, a wyjście z podprogramu zajmuje dwa cykle zegarowe. W kodzie programu naszego procesora PIC, po instrukcji **return** i **retlw**, aby powrócić z podprogramu, należy zapisać instrukcję **nop** (brak operacji).

Realizacja instrukcji **sw** i **lw** wymaga dodania do schematu procesora magistral **Write_Data** i **Read_Data** w celu ustanowienia interfejsu pamięci danych DM. Ponadto realizacja instrukcji **lw** wymaga zainstalowania multipleksera na wejściu akumulatora W. Sygnał sterujący multipleksera *lw* i sygnał zezwolenia na zapis *sw* w pamięci danych DM są generowane przez urządzenie sterowania na podstawie odpowiednich kodów instrukcji.

Aby zaimplementować instrukcje **gotoz** i **gotonz** skoku pod adres w zależności od wyniku poprzedniej operacji, do układu procesora jest wprowadzany przerzutnik, który zapamiętuje flagę z poprzedniej operacji, modyfikowany jest również układ generujący adres następnej instrukcji.

Układ arytmetyczno-logiczny ALU procesora PIC na podstawie kodu instrukcji, wartości dwóch operandów A i B oraz poszczególnych flag tworzy na wyjściu wynik operacji. Jeśli nie zdefiniowano żadnego kodu instrukcji lub nie jest wymagana żadna operacja, domyślnie na wyjście ALU jest przesyłana wartość akumulatora W (argument B).

Układ sterowania procesora jednocyklowego jest układem kombinacyjnym, który generuje wszystkie niezbędne sygnały sterujące na podstawie kodu instrukcji i poszczególnych flag. Sygnały sterujące PIC można podzielić na sygnały wyboru dla multipleksów, sygnały zezwalające na zapis do rejestrów i elementów pamięci oraz sygnały zapisu i odczytu stosu.

Struktura zaimplementowanego procesora różni się od rozważanej konstrukcji procesora tym, że pamięć danych i pamięć instrukcji są urządzeniami zewnętrznymi w stosunku do procesora.

Aby zmniejszyć liczbę błędów, a także ułatwić zmiany i modyfikacje, komponenty procesora i moduł procesora należy opisać w języku Verilog jako moduły sparametryzowane. W tym samym celu przekazywanie wartości parametrów i sygnałów do instancji modułów musi odbywać się przy użyciu nazw parametrów i portów.

Debugowanie złożonego projektu jest znacznie bardziej czasochłonne niż tworzenie kodu projektu. Debugowanie projektu rozpoczyna się od przetestowania wszystkich komponentów projektu, zaczynając od poziomu najniższego. Należy jednak zwrócić szczególną uwagę na sytuacje wyjątkowe i spełnienie warunków granicznych. Podczas debugowania procesora PIC należy zwrócić szczególną uwagę na testowanie ALU i układu sterującego.

Ukończony projekt procesora musi zostać zweryfikowany pod kątem prawidłowego wykonania każdej instrukcji, grupy instrukcji, przykładów referencyjnych, przepływu zadań i działania testowego. Jeśli wprowadzisz zmiany w układzie procesora, proces debugowania należy powtórzyć od samego początku.

Wąskim gardłem pod względem wydajności procesora PIC jest jednostka arytmetyczno-logiczna i stos. Dlatego należy zwrócić szczególną uwagę na proces projektowania tych jednostek.

Rozważany procesor PIC zapewnia szerokie możliwości modyfikacji jego struktury. Do tego celu szerokość słowa pamięci instrukcji PIC wynosi 16 bitów, podczas gdy długość słowa pamięci instrukcji PIC16F84A wynosi 14 bitów. Ponadto układ operacyjny, pamięć danych, pamięć instrukcji i procesor PIC są opisane w języku Verilog przy użyciu sparametryzowanych modułów.

9. Projektowanie procesorów wielocyklowych PIC

W procesorze wielocyklowym wykonanie każdej instrukcji jest podzielone na etapy. Czas wykonania najwolniejszego etapu jest krótszy niż czas wykonania całej instrukcji. Dlatego też częstotliwość taktowania procesora wielocyklowego jest wyższa niż procesora jednocyklowego. Na podstawie procesorów wielocyklowych budowane są procesory potokowe, których wydajność jest znacznie wyższa od procesorów jednocyklowych.

W tym rozdziale przedstawiono trzy metodyki projektowania procesorów wielocyklowych. Pierwsza technika jest tradycyjnym podejściem do projektowania procesorów wielocyklowych i polega na sekwencyjnym wprowadzaniu stanów układu sterującego: do wybierania i dekodowania instrukcji, do realizacji określonych grup instrukcji i pojedynczych instrukcji. Druga technika jest oryginalna, maksymalnie wykorzystuje układ operacyjny i układ sterowania procesora jednocyklowego. Zgodnie z nią do obwodu procesora dodawany jest automat skończony, który określa etapy wykonywania instrukcji. Trzecia technika ma na celu zwiększenie liczby etapów procesora wielocyklowego. Czasami jest to konieczne, aby zwiększyć wydajność procesora potokowego, który jest oparty na procesorze wielocyklowym.

Podano również porównania kosztów implementacji, szybkości i wydajności procesorów wielocyklowych w porównaniu z procesorem jednocyklowym.

9.1. Tradycyjne projektowanie procesora wielocyklowego PIC

Aby zaprojektować procesor wielocyklowy PIC, użyjemy tej samej metodyki, co przy projektowaniu procesora jednocyklowego, omówionej w rozdziale 7.

9.1.1. Projektowanie układu operacyjnego

Nasze zadanie upraszcza fakt, że struktura procesora wielocyklowego w dużej mierze jest zgodna ze strukturą procesora jednocyklowego (rys. 8.12). W szczególności, wiele

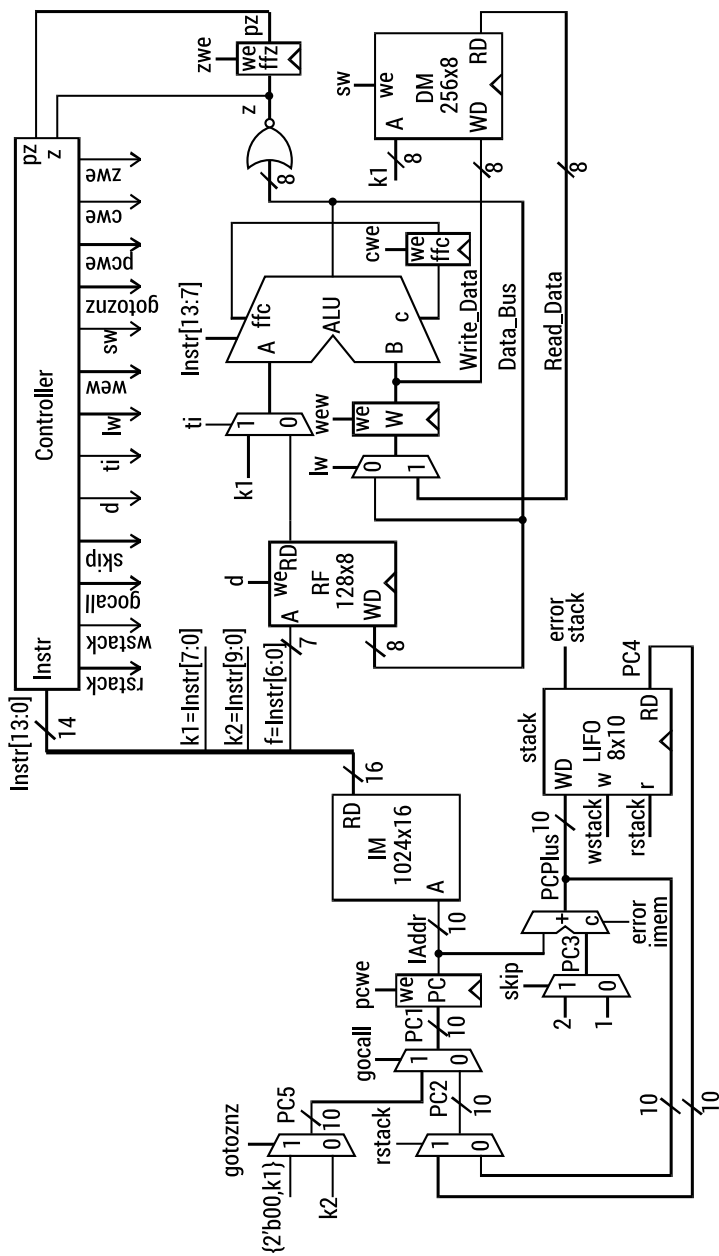
komponentów układu operacyjnego zostało już opracowanych i może być ponownie wykorzystanych w procesorze wielocyklowym.

Zwróćmy uwagę na różnice w strukturze procesora charakterystyczne dla procesora wielocyklowego. Przede wszystkim, aby móc wybrać kod instrukcji z pamięci instrukcji IM na etapie Fetch, rejestr licznika instrukcji PC musi mieć wejście zezwolenia zapisu *we*, które jest sterowane sygnałem *pcwe*.

Co więcej, w strukturze procesora jednocyklowego (rys. 8.12) przerzutniki *ffc* i *ffz* służące do przechowywania flag przeniesienia *c* i zero *z* nie mają wejścia umożliwiającego zapis, ponieważ wartości flag musiały być zapisane tylko na jeden cykl zegarowy i sprawdzone w następnym cyklu. W procesorze wielocyklowym instrukcja może być wykonywana przez kilka cykli (których liczba nie jest z góry znana), a wartości flag *c* i *z* muszą być również zachowane przez kilka cykli. Dlatego w procesorze wielocyklowym przerzutniki *ffc* i *ffz* muszą mieć wejścia zezwolenia zapisu *we*, które są sterowane odpowiednio przez sygnały *cwe* i *zwe*.

Pomijając wymienione różnice, schemat procesora wielocyklowego (rys. 9.1) jest w dużej mierze podobny do schematu procesora jednocyklowego.

Zwróćmy uwagę na różnice w schematach procesorów jedno- i wielocyklowego. Na schemacie na rysunku 9.1, w przeciwieństwie do schematu z rysunku 8.12, nie ma przerzutnika, który zapamiętałby sygnał *rstack*. W procesorze wielocyklowym nie ma potrzeby zapamiętywania sygnału *rstack*, ponieważ pobieranie instrukcji i wykonywanie instrukcji są realizowane w różnych cyklach zegarowych. Na schemacie z rysunku 9.1 rejestr licznika rozkazów PC oraz przerzutniki do przechowywania flagi przeniesienia *c* i zera *z* posiadają również wejścia zezwolenia na zapis, które są sterowane odpowiednio sygnałami *pcwe*, *cwe* i *zwe*. Ponadto urządzenie sterowania w układzie pokazanym na rysunku 9.1 generuje trzy dodatkowe sygnały sterujące: *pcwe*, *cwe* i *zwe*.



9.1.2. Projektowanie układu sterującego

Główne różnice w projektach procesorów wielocyklowego i jednocyklowego dotyczą ich układów sterowania.

W procesorze jednocyklowym układ sterowania jest układem kombinacyjnym, który na podstawie kodu rozkazu oraz znaczników *z* i *pz* generuje wartości sygnałów sterujących niezbędnych do wykonania odpowiedniej instrukcji.

W procesorze wielocyklowym wykonywanie instrukcji jest podzielone na kilka etapów. Dlatego też urządzeniem sterującym jest automat skończony. Kiedy instrukcja jest wykonywana, automat skończony przechodzi przez kilka stanów, w których generowane są sygnały sterujące niezbędne do wykonania określonego etapu.

Typowe etapy wykonywania instrukcji w procesorach wielocyklowych to:

- *wybór instrukcji* (Fetch);
- *dekodowanie instrukcji* (Decode);
- *wykonanie instrukcji* (Execute);
- *odczyt-zapis pamięci* (Memory).

Liczba stanów automatu skończonego może, ale nie musi, być taka sama jak liczba etapów procesora wielocyklowego. Charakterystyczną cechą automatów skończonych, które są układem sterującym procesora wielocyklowego, jest reprezentacja automatu skończonego w postaci zbioru ścieżek (pętli) rozpoczynających się i kończących w stanie początkowym. Przejście automatu przez stany jednej pętli odpowiada etapom wykonania danej instrukcji. Po tym czasie automat jest gotowy do wykonania następnej instrukcji.

Istnieje wiele podejść do projektowania układu sterującego procesora wielocyklowego:

- liczba stanów automatu skończonego pokrywa się z liczbą etapów;
- każda pętla automatu skończonego implementuje jedną instrukcję;
- każda pętla automatu skończonego implementuje grupę instrukcji itp.

Przyjrzyjmy się procesowi projektowania układu sterującego dla naszego wielocyklowego procesora PIC. Automat skończony będziemy reprezentować jako automat typu Moore'a, w którym niezbędne wartości sygnałów sterujących są generowane w stanach automatu. Przejścia między stanami mogą być bezwarunkowe (oznaczone wartością logiczną 1, czyli zawsze prawdziwe) lub warunkowe, wykonywane pod pewnymi warunkami (np. w przypadku jedynkowej wartości flagi wykonywanej instrukcji). Aby uprościć prezentację, znaki instrukcji będą oznaczone mnemonikami odpowiedniej instrukcji.

Wyjścia automatu skończonego są sygnałami zezwalającymi na zapis do pamięci, rejestrów i przerzutników układu operacyjnego oraz sygnałami selekcji używanymi przez multipleksery układu operacyjnego. W pobliżu stanów automatu skończonego napiszemy sygnały sterujące, które w tym stanie mają wartość jedynkową. Domyślnie wszystkie inne sygnały sterujące przyjmują w tym stanie wartość zero.

9.1.2.1. Wybór kodu rozkazu z pamięci instrukcji

Wybór kodu rozkazu z pamięci instrukcji IM odbywa się w stanie początkowym S0 poprzez ustawienie sygnału *pcwe* zezwolenia na zapis do licznika rozkazów PC (rys. 9.2). Zakłada się, że wartość adresu następnej instrukcji została już obliczona i znajduje się na magistrali PC1 (rys. 9.1).

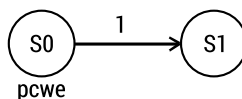


RYS. 9.2. Stan S0 wyboru kodu instrukcji

W stanie S0 wartość magistrali PC1 jest zapisywana do rejestru PC, a na wyjściu rejestru PC pojawia się wartość adresu następnej instrukcji, tzn. na magistrali IAddr. Ponieważ magistrala IAddr jest wejściem adresowym pamięci instrukcji IM, kod wykonywanej instrukcji zostanie ustawiony na wyjściu RD pamięci instrukcji IM.

9.1.2.2. Dekodowanie instrukcji

W jednocyklowym procesorze PIC dekodowanie instrukcji odbywa się za pomocą połączeń przewodowych bez udziału sygnałów sterujących. W procesorze wielocyklowym sygnały sterujące również nie są wymagane, ale wymagany jest oddzielny etap, określony przez stan S1 (rys. 9.3).

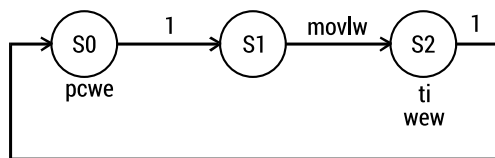


RYS. 9.3. Stan S1 dekodowania instrukcji

Przejście ze stanu S0 do stanu S1 jest bezwarunkowe, tzn. jest wykonywane zawsze. Wskazuje na to jedynka logiczna na strzałce prowadzącej ze stanu S0 do stanu S1. Należy zauważyć, że w stanie S1 nie są generowane żadne sygnały sterujące. W przypadku sekwencyjnego wykonywania instrukcji żadne sygnały sterujące nie są wymagane do tworzenia adresu następnej instrukcji. Dlatego w stanie S1 na magistrali PC1 zostanie wygenerowany adres kolejnej instrukcji.

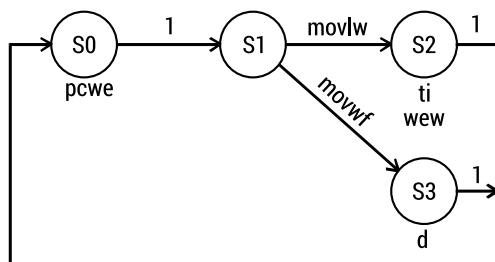
9.1.2.3. Implementacja instrukcji transferu danych (instrukcji *movlw*, *movwf* i *movf*)

Aby zaimplementować instrukcje do przesyłania (transferu) danych **movlw**, **movwf** i **movf**, wprowadzamy do automatu skończonego, odpowiednio, trzy stany S2, S3 i S4. Aby zaimplementować polecenie **movlw**, które zapisuje stałą *k1* do akumulatora W (rys. 9.4), należy podać wartość stałej *k1* na wejście A ALU (ustawić sygnał *ti*), przesłać na wejście akumulatora W (domyślnie *lw* = 0) i zezwolić na zapis do akumulatora (ustawić sygnał *wew*).



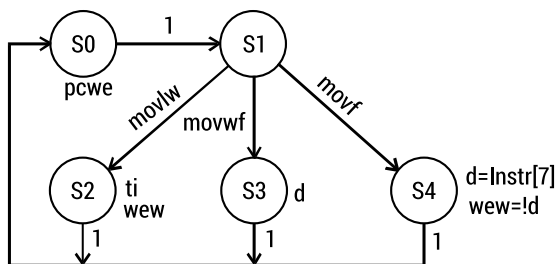
RYS. 9.4. Stan S2 do wykonania instrukcji **movlw**

Aby zaimplementować polecenie **movwf**, które zapisuje wartość akumulatora W do pliku rejestrów RF (rys. 9.5), konieczne jest ustawienie wartości adresu rejestru f (już ustawionego w wyniku dekodowania rozkazu w stanie S1) na wejściu adresowym pliku rejestrów RF. Należy też podać wartość akumulatora na wejście B ALU (zawsze podawane bez sygnałów sterujących) i umożliwić zapis do pliku rejestrów ($d = 1$).



RYS. 9.5. Stan S3 do wykonania instrukcji **movwf**

Aby zaimplementować polecenie **movf**, które zapisuje wartość rejestru f do akumulatora W lub z powrotem do rejestru f (rys. 9.6), konieczne jest ustawienie wartości adresu rejestru f na wejściu adresowym pliku rejestrów RF (już ustawionego w wyniku dekodowania rozkazu), dodatkowo na wejściu A ALU, wartość rejestru f (domyślnie $ti = 0$), umożliwia zapis albo do pliku rejestrów RF (wartość bitu $d = \text{Instr}[7]$), albo do akumulatora W ($wew = !d$).



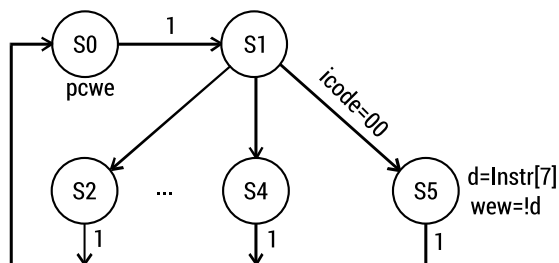
RYS. 9.6. Stan S4 do wykonania instrukcji **movf**

Zwróćmy uwagę, że wartości sygnałów d i wew nie są określone jednoznacznie, ale zależą od wartości bitu $Instr[7]$, tzn. od wejścia automatu skończonego. W tym przypadku mamy sygnały wyjściowe d i wew automatu Mealy'ego. Dlatego nasz automat skończony nie jest automatem Moore'a w „czystej” formie, ale jest wspólnym modelem automatów typu Mealy'ego i Moore'a.

9.1.2.4. Realizacja operacji na bajtach (instrukcji formatu a)

Operacje bajtowe są realizowane w jednostce ALU w zależności od kodu instrukcji. Podczas wykonywania operacji na bajtach jeden operand podawany jest z pliku rejestrów RF na wejście A ALU ($ti = 0$), a drugi z wyjścia akumulatora W (nie są wymagane żadne sygnały sterujące). Wynik operacji utworzony na wyjściu ALU, na magistrali Data_Bus, jest zapisywany do pliku rejestrów ($d = 1$) lub do akumulatora W ($wew = !d$).

Aby zaimplementować instrukcje formatu a , do automatu skończonego wprowadzamy stan S5 (rys. 9.7), do którego przejście odbywa się pod warunkiem $icode == 2'b00$, gdzie $icode = Instr[6:5]$ jest wspólnym kodem dla instrukcji formatu a .



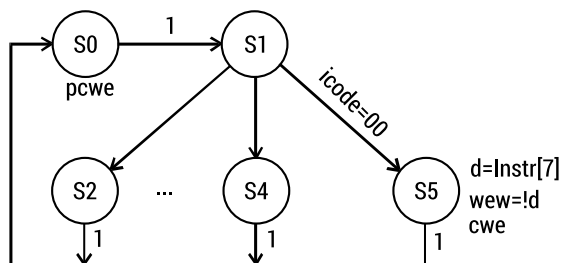
RYŚ. 9.7. Stan S5 do implementacji instrukcji formatu a

Na tym etapie projektowania automatu w stanie S5 generowane są wartości tylko dwóch sygnałów: d i wew . Następnie będziemy dodawać sygnały sterujące generowane w stanie S5.

9.1.2.5. Cykliczne operacje przesunięcia (instrukcje rlf i rff)

Instrukcje **rlf** i **rff** przesuwają cyklicznie w lewo i w prawo zawartość rejestru f przez bit przeniesienia c , a wynik jest zapisywany z powrotem do rejestru f ($d = 1$) lub do akumulatora W ($wew = !d$). Wartość sygnału przeniesienia c jest przechowywana w przerzutniku ffc na zewnątrz jednostki ALU. Zapisem do przerzutnika ffc steruje sygnał zezwolenia na zapis cwe (c write enable).

Ponieważ sygnał przeniesienia może być generowany w innych instrukcjach, ustawimy sygnał cwe tak, aby umożliwiał zapisanie sygnału przeniesienia do przerzutnika ffc dla wszystkich operacji na bajtach (rys. 9.8).

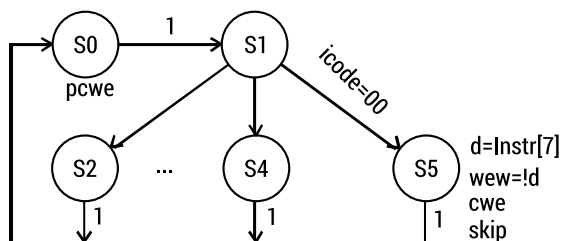


RYS. 9.8. Stan S5 do realizacji cyklicznych operacji przesunięcia

9.1.2.6. Realizacja operacji inkrementacji i dekrementacji z pominięciem następnej instrukcji, jeśli wynik jest zerowy (instrukcje **incfsz** i **decfsz**)

Realizacja instrukcji **incfsz** (**decfsz**) zakłada zwiększenie (zmniejszenie) o jeden wartości rejestru *f*; sprawdzenie wyniku, jeśli wynik wynosi zero, pominięcie następnej instrukcji; zapis wyniku z powrotem do rejestru *f* lub do akumulatora *W*. Zwiększanie (zmniejszanie) wartości rejestru *f* jest wykonywane w jednostce ALU na podstawie kodu instrukcji. Kolejne polecenie jest pomijane przez zwiększenie licznika rozkazów PC nie o 1, ale o 2. W tym celu należy ustawić sygnał pominięcia *skip*. Zapisanie wyniku z powrotem do rejestru *f* lub do akumulatora *W* nie różni się od innych instrukcji formatu *a*.

Aby zaimplementować instrukcje **incfsz** i **decfsz**, wystarczy w stanie S5 (rys. 9.9) wygenerować sygnał pominięcia za pomocą następującego wyrażenia logicznego:
 $\text{skip} = z \ \&\& \ \text{decfsz} \ || \ z \ \&\& \ \text{incfsz}$.



RYS. 9.9. Stan S5 do realizacji instrukcji **incfsz** i **decfsz**

W ten sposób pętla automatu skończonego przechodząca przez stan S5 implementuje grupę instrukcji w formacie *a*. Wyjątkiem są instrukcje **movf** i **movwf**, które zostały zaimplementowane wcześniej.

Pytanie do uważnego czytelnika: czy można w naszym automacie skończonym połączyć stany S4 i S5 w jeden stan?

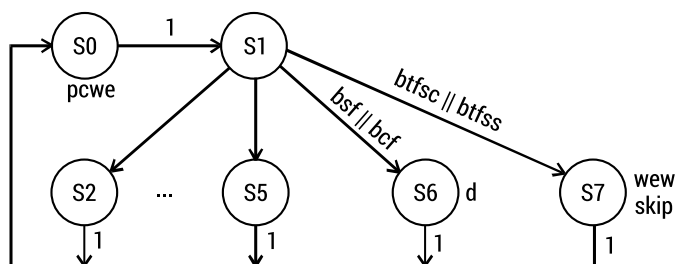
9.1.2.7. Realizacja operacji bitowych (instrukcji formatu *b*)

Format *b* zawiera cztery instrukcje, które można podzielić na dwie grupy: zmiana wartości bitu (instrukcje **bsf** i **bcf**) oraz sprawdzenie wartości bitu i pominięcie następnej instrukcji w zależności od wyniku (instrukcje **btfsc** i **btfss**).

Instrukcja **bsf** (**bcf**) ustawia (zeruje) określony bit w rejestrze *f*. Wynik jest generowany w ALU i zapisywany w rejestrze *f*. Aby zapisać wynik w rejestrze *f*, należy ustawić sygnał *d* i wyzerować sygnał *wew*.

Aby wykonać polecenie **btfsc** lub **btfss**, musimy sprawdzić wartość określonego bitu w rejestrze *f*. Aby to zrobić, przesunemy stałą 8'b00000001 w lewo o określoną liczbę bitów i dodamy ją logicznie do zawartości rejestru *f*. Wynik będzie przechowywany w akumulatorze W, a zawartość rejestru *f* pozostanie niezmieniona. Następną instrukcją jest pomijana, jeśli *z* = 0 dla **btfss** i *z* = 1 dla **btfsc**.

Aby zaimplementować instrukcje formatu *b*, wprowadzamy dwa oddzielne stany (rys. 9.10): S6 – dla instrukcji **bsf** i **bcf**; S7 – dla instrukcji **btfsc** i **btfss**.



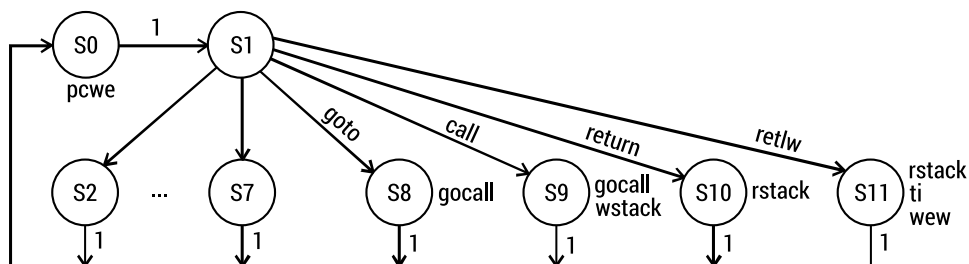
RYS. 9.10. Stany S6 i S7 do implementacji instrukcji formatu *b*

W stanie S6 generowane są sygnały *d* = 1 i *wew* = 0, a w stanie S7 – *d* = 0 i *wew* = 1. Dodatkowo w stanie S7 wartość sygnału pominięcia *skip* jest określana za pomocą następującego wyrażenia logicznego:

$skip = z \ \&\& \ btfsc \ || \ !z \ \&\& \ btfss.$

9.1.2.8. Realizacja instrukcji skoku (instrukcji goto, call, return i retlw)

Aby zaimplementować cztery instrukcje skoków **goto**, **call**, **return** i **retlw**, wprowadzamy cztery stany S8, S9, S10 i S11, odpowiadające każdej ze wskazanych instrukcji (rys. 9.11).



RYS. 9.11. Stany S8, S9, S10 i S11 do implementacji instrukcji skoku

Aby zaimplementować polecenie **goto**, wystarczy ustawić sygnał *gocall*. W rezultacie wartość stałej *k2* (rys. 9.1) wraz z adresem instrukcji, do której wykonuje się

przejście, zostanie przesłana na wejście licznika rozkazów PC (domyślnie sygnał *gotoznz* ma wartość 0).

Aby zaimplementować polecenie **call**, dodatkowo ustawiany jest sygnał *wstack* do zapisu adresu następnej instrukcji na stos.

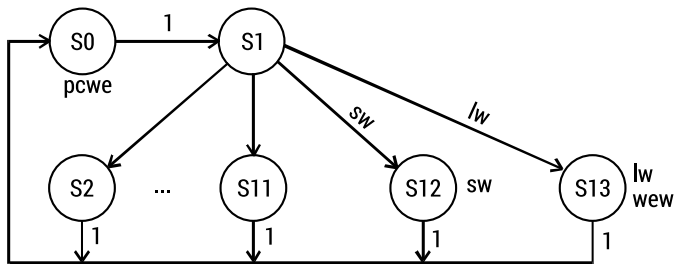
Polecenie powrotu **return** jest realizowane przez ustawienie sygnału *rstack*. W rezultacie adres instrukcji na szczycie stosu zostanie odczytany ze stosu (rys. 9.1) i przesłany na wejście rejestru licznika rozkazów PC (domyślnie sygnał *gocall* = 0).

Aby zaimplementować polecenie **retlw**, oprócz sygnału *rstack* należy ustawić sygnały *ti* i *wew*. W rezultacie stała *k1* (rys. 9.1), znajdująca się w kodzie rozkazu, zostanie przekazana na wejście akumulatora W (domyślnie *lw* = 0) i zapisana do akumulatora w następnym cyklu zegarowym.

Należy zwrócić uwagę, że wykonanie instrukcji **return** i **retlw** nie wymaga dodatkowego taktu synchronizacji, jak w procesorze jednocyklowym, ponieważ wybór kodu instrukcji z pamięci instrukcji i wybór kodu instrukcji ze stosu odbywa się na różnych etapach wykonywania instrukcji.

9.1.2.9. Realizacja operacji zapisu i odczytu pamięci danych (instrukcje sw i lw)

Aby zaimplementować instrukcję **sw** zapisu zawartości akumulatora do pamięci danych (zapis do pamięci danych) oraz instrukcję **lw** ładowania zawartości akumulatora z pamięci danych (odczyt z pamięci danych), wprowadzamy dwa odrębne stany S12 i S13 (rys. 9.12).



RYS. 9.12. Stany S12 i S13 do implementacji instrukcji **sw** i **lw**

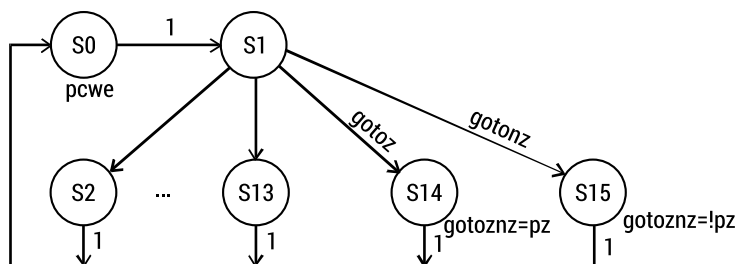
Adres komórki pamięci, do której zapisywane są dane lub z której odczytywane są dane, wskazuje stała $k1$ (stała $k1$ określana jest na etapie dekodowania w stanie S1). Aby zapamiętać wartość akumulatora W w pamięci danych (rozkaz **sw**), wystarczy ustawić sygnał sterujący sw . Aby zawartość komórki pamięci danych DM załadować do akumulatora W (rozkaz **lw**) należy ustawić sygnał lw i umożliwić zapis do akumulatora ($wew = 1$).

9.1.2.10. Realizacja skoków do adresu w zależności od wyniku poprzedniej operacji (instrukcje gotoz i gotonz)

Aby zapisać wynik poprzedniej operacji w układzie operacyjnym, używany jest przerzutnik *ffz*, który przechowuje wartość flagi zera *z*. Zapis do przerzutnika *ffz* odbywa

się za pomocą sygnału *zwe* (z write enable – zezwolenie na zapis flagi *z*). Sygnał *zwe* musi być generowany we wszystkich stanach automatu, w których obliczany jest jakikolwiek wynik, tzn. w stanach S2, S3, S4, S5, S6, S7, S12 i S13.

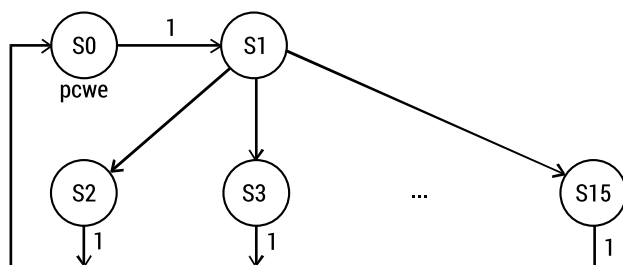
Aby zaimplementować instrukcje *gotoz* i *gotonz*, wprowadzono dwa stany S14 i S15 (rys. 9.13).



RYS. 9.13. Stany S14 i S15 do implementacji instrukcji **gotoz** i **gotonz**

W stanie S14 sygnał sterujący *gotoznz* jest generowany, jeśli *pz* = 1, a w stanie S15 generowany jest sygnał sterujący *gotoznz*, jeśli *pz* = 0.

W ten oto sposób automat skończony odpowiadający układowi sterującemu procesora wielocyklowego PIC ma postać pokazaną na rys. 9.14.



RYS. 9.14. Ogólny diagram automatu skończonego do sterowania procesorem wielocyklowym

Na rysunku 9.14 można zauważyć, że zaprojektowany procesor jest procesorem trzycyklowym, ponieważ każda instrukcja jest wykonywana w trzech cyklach (taktach) zegarowych.

9.1.3. Opis komponentów procesora wielocyklowego w języku Verilog

W procesorze wielocyklowym pamięć instrukcji IM, pamięć danych DM i większość komponentów układu operacyjnego są identyczne z odpowiednimi komponentami procesora jednocyklowego. Rozpatrzmy opis w języku Verilog tylko tych

komponentów, które znacznie różnią się od analogicznych komponentów procesora jednocyklowego.

9.1.3.1. Opis układu operacyjnego

W procesorze wielocyklowym są używane przerzutniki z wejściem zezwolenia zapisu *we*. Opis takiego przerzutnika w języku Verilog może wyglądać następująco:

```
module my_ff(  
input clk, reset, we,  
input d,  
output reg q);  
  
always @(posedge clk, negedge reset)  
    if (~reset)        q <= 1'b0;  
    else if(we)         q <= d;  
  
endmodule
```

Pozostałe komponenty jednostki operacyjnej dla procesora wielocyklowego są dokładnie takie same, jak dla procesora jednocyklowego.

Opis układu operacyjnego procesora wielocyklowego w pełni odpowiada schematowi na rysunku 9.1 i wygląda następująco:

```
module datapath  
#(parameter  
DATA_WIDTH=8,           // szerokość danych  
DADDR_WIDTH=8,          // szerokość adresu pamięci danych  
INSTR_WIDTH=14,          // rozmiar instrukcji  
IADDR_WIDTH=10,          // szerokość adresu pamięci instrukcji  
STACK_ADDR_WIDTH=3,      // szerokość adresu stosu  
REGFILE_ADDR_WIDTH=7,    // szerokość adresu pliku rejestrów  
STACK_DEPTH=(1<<STACK_ADDR_WIDTH), // głębokość stosu  
REGFILE_DEPTH=(1<<REGFILE_ADDR_WIDTH)) // głębokość pliku rejestrów  
  
(input clk, reset,           // zegar, reset  
input [INSTR_WIDTH-1:0] Instr, // polecenie (instrukcja)  
input pcwe,                // zezwolenie zapisu w PC  
input cwe,zwe,             // zezwolenie zapisu w ffc i ffpz  
input rstack, wstack, gocall, skip, // sygnały sterujące  
input d,                   // kierunek zapisu wyniku ALU  
// d = 0 – do W, d = 1 – do pliku rejestrów  
input ti, lw, wew,         // sygnały sterujące
```

```

input [DATA_WIDTH-1:0] Read_Data, // dane z pamięci danych
input gotoznz, // sygnały sterujące dla realizacji
// instrukcji gotoz i gotonz
output [IADDR_WIDTH-1:0] IAddr, // adres pamięci instrukcji
output [DADDR_WIDTH-1:0] DAddr, // adres pamięci danych
output [DATA_WIDTH-1:0] Write_Data, // dane do pamięci danych
output error_imem, error_stack, // flagi błędów
output z, // flaga zero
output [DATA_WIDTH-1:0] Data_Bus, // magistrala danych
output pz; // flaga wyniku zerowego

wire [7:0] k1=Instr[7:0]; // stała formatu c
wire [9:0] k2=Instr[9:0]; // stała formatu d

wire [6:0] f=Instr[6:0]; // numer rejestru w pliku rejestrów
wire [IADDR_WIDTH-1:0] PCPlus,PC,PC1,PC2,PC3,PC4,PC5; // magistrale PC
wire [DATA_WIDTH-1:0] A_ALU, RF_out, W, W1; // magistrale W
wire c, wffc; // przeniesienie i wyjście przerzutnika przeniesienia

// logika licznika rozkazów PC
register_we #(IADDR_WIDTH) ex_pc(clk,reset,pcwe,PC1,PC); // rejestr instrukcji

mux2_1 #(IADDR_WIDTH) ex_pc1(PC2,PC5,gocall,PC1); // multipleksery
mux2_1 #(IADDR_WIDTH) ex_pc2(PCPlus,PC4,rstack,PC2);
mux2_1 #(IADDR_WIDTH) ex_pc3(10'd1,10'd2,skip,PC3);
mux2_1 #(IADDR_WIDTH) ex_pc5(k2,{2'b00,k1},gotoznz,PC5);

adder_PCPlus #(IADDR_WIDTH)
ex_adder(PC,PC3,PCPlus,error_imem); // sumator PC

stack #(IADDR_WIDTH,STACK_ADDR_WIDTH,STACK_DEPTH)
ex_stack(PCPlus,clk,reset,rstack,wstack,PC4,error_stack); // stos

// plik rejestrów
regfile #(REGFILE_ADDR_WIDTH,REGFILE_DEPTH,DATA_WIDTH)
ex_regfile(clk,d,f,Data_Bus,RF_out);

// Logika ALU i akumulatora W.
mux2_1 #(DATA_WIDTH) ex_A_ALU(RF_out,k1,ti,A_ALU); // multipleksery
mux2_1 #(DATA_WIDTH) ex_W1(Data_Bus,Read_Data,lw,W1);

```



```

register_we #(DATA_WIDTH)
ex_W(clk,reset,wew,W1,Write_Data);          // akumulator W

alu #(.DATA_WIDTH(DATA_WIDTH))              // ALU
    ex_alu(.a(A_ALU),.b(Write_Data),.in_instr(Instr[13:7]),
    .ffc(wffc),.out(Data_Bus),.c(c));

assign z = (!(Data_Bus));                    // obliczanie wyniku zerowego

my_ff ex_ffc(clk,reset,cwe,c,wffc);          // przerzutnik przeniesienia
my_ff ex_ffpz(clk,reset,zwe,z,pz);           // przerzutnik zero

assign lAddr=PC;                            // definicja adresu pamięci instrukcji
assign DAddr=k1;                            // definicja adresu pamięci danych
endmodule

```

Różnice między konstrukcją układu operacyjnego dla procesora jednocyklowego a konstrukcją procesora wielocyklowego są następujące:

- dodano wejściowe sygnały sterujące *pcwe*, *cwe* i *zwe*;
- usunięto przerzutnik *ex_rstack*;
- rejestr licznika rozkazów PC realizowany jest za pomocą modułu *register_we*, który posiada wejście zezwolenia zapisu *we*, sterowane sygnałem *pcwe*;
- przerzutniki *ex_ffc* i *ex_ffpz* są zaimplementowane przy użyciu modułu *my_ff*, który ma wejście zezwolenia zapisu *we* sterowane przez sygnał *cwe* dla instancji *ex_ffc* i przez sygnał *zwe* dla instancji *ex_ffpz*.

9.1.3.2. Opis układu sterującego

Opis układu sterującego w języku Verilog dla procesora wielocyklowego może wyglądać następująco:

```

module controller(
input clk,reset,                // zegar i reset
input [13:0] Instr,            // kod maszynowy instrukcji
input z,pz,                     // flagi zero
output reg rstack, wstack, gocal, skip, // generowane sygnały sterujące
output reg d,
output reg ti, lw, wew,
output reg sw,
output reg gotoznz,
output reg pcwe,
output reg cwe, zwe);

wire call,goto,retlw,preturn,decfsz,incfsz,bcf,bsf, // znaki instrukcji

```

btfsc,btfss,gotoz,gotonz,movlw,movwf,movf,plw,psw;

wire [1:0] icode = Instr[13:12]; // kod formatu instrukcji

// tworzenie znaków instrukcji

```
assign movlw = (Instr[13:8] == 6'b110000);
assign movwf = (Instr[13:7] == 7'b0000001);
assign movf = (Instr[13:8] == 6'b001000);
assign plw = (Instr[13:8] == 6'b110001);
assign psw = (Instr[13:8] == 6'b110010);
assign call = (Instr[13:11] == 3'b100);
assign goto = (Instr[13:11] == 3'b101);
assign preturn = (Instr[13:0] == 14'b00000000001000);
assign retlw = (Instr[13:8] == 6'b110100);
assign decfsz = (Instr[13:8] == 6'b001011);
assign incfsz = (Instr[13:8] == 6'b001111);
assign bcf = (Instr[13:10] == 4'b0100);
assign bsf = (Instr[13:10] == 4'b0101);
assign btfsc = (Instr[13:10] == 4'b0110);
assign btfss = (Instr[13:10] == 4'b0111);
assign gotoz = (Instr[13:8] == 6'b110101);
assign gotonz = (Instr[13:8] == 6'b110110);
```

reg [3:0] state,next; // zmienne stanu automatu

// określanie stanów automatu skończonego

localparam [3:0] s0=0,s1=1,s2=2,s3=3,s4=4,s5=5,s6=6,s7=7,s8=8,
 s9=9,s10=10,s11=11,s12=12,s13=13,s14=14,s15=15;

always @(posedge clk, negedge reset) // opis pamięci automatu

```
    if (~reset)        state <= s0;
    else              state <= next;
```

always @(*) // opis przejść automatu

```
    case (state)
        s0:    next = s1;
        s1:    if(preturn) next = s10;
              else if(movwf) next = s3;
              else if(movlw) next = s2;
              else if(movf) next = s4;
              else if(retlw) next = s11;
```

```

else if(psw) next = s12;
else if(plw) next = s13;
else if(gotoz) next = s14;
else if(gotonz) next = s15;
else if(bsf || bcf) next = s6;
else if(btfsc || btfss) next = s7;
else if(goto) next = s8;
else if(call) next = s9;
else if(icode == 2'b00) next = s5;
else next = s0;
default: next = s0;
endcase

```

```

always @(*) // opis funkcji wyjściowych
begin

```

// domyślne wartości wyjściowe

```

pcwe=1'b0; ti=1'b0; wew=1'b0; zwe=1'b0; d=1'b0; cwe=1'b0; skip=1'b0;
gocall=1'b0; wstack=1'b0; rstack=1'b0; sw=1'b0; lw=1'b0;

```

gotoznz=1'b0;

```

case (state)
s0: pcwe=1'b1;
s1: ;
s2: begin ti=1'b1; wew=1'b1; zwe=1'b1; end
s3: begin d=1'b1; zwe=1'b1; end
s4: begin d=Instr[7]; wew=!d; zwe=1'b1; end
s5: begin
    d=Instr[7]; wew=!d;
    cwe=1'b1;
    skip= (z && decfsz) || (z && incfsz);
    zwe=1'b1;
end
s6: begin d=1'b1; zwe=1'b1; end
s7: begin
    wew=1'b1;
    skip= (z && btfsc) || (!z && btfss);
    zwe=1'b1;
end
s8: gocall=1'b1;
s9: begin gocall=1'b1; wstack=1'b1; end
s10: rstack=1'b1;
s11: begin rstack=1'b1; ti=1'b1; wew=1'b1; end

```

```

s12: begin sw=1'b1; zwe=1'b1; end
s13: begin lw=1'b1; wew=1'b1; zwe=1'b1; end
s14: gotoznz=pz;
s15: gotoznz=!pz;
default ;

endcase

end

endmodule

```

Układ sterowania jest reprezentowany jako automat skończony, którego opis składa się z trzech procesów (bloków **always**). Na początku, dla wygody opisu, na podstawie kodu instrukcji tworzone są znaki instrukcji: sygnały wewnętrzne, których nazwy pokrywają się z mnemonikami wykonywanej instrukcji.

Automat skończony, zgodnie z rysunkiem 9.14, zawiera 16 stanów s0 – s15, do których kodowania wystarczą cztery bity.

Pierwszy proces opisuje pamięć automatu w tradycyjny sposób.

Drugi proces opisuje przejścia między stanami. Większość przejść jest bezwarunkowa. Wyjątkiem są przejścia ze stanu s1. W powyższym opisie zagnieżdżono instrukcje **if** do sprawdzenia, czy warunki przejścia są spełnione. Zwracamy szczególną uwagę na to, że procedura sprawdzania warunków przejścia w tym przypadku jest niezbędna. Przede wszystkim należy sprawdzić warunki, w których sprawdzana jest największa liczba bitów kodu instrukcji. Jeśli ta reguła zostanie naruszona, przejścia do niektórych stanów automatu skończonego nigdy nie zostaną wykonane, tzn. niektóre instrukcje nie zostaną nigdy wykonane.

Czytelnik jest proszony o samodzielne opisanie przejść ze stanu s1 za pomocą instrukcji **case**.

Trzeci proces opisuje funkcje wyjściowe automatu skończonego. Służy do tego instrukcja **case**. Domyślne wartości sygnałów wyjściowych są opisane przed instrukcją **case**. W każdym stanie generowane są niezbędne wartości sygnałów wyjściowych automatu skończonego, które są sygnałami sterującymi dla układu operacyjnego.

Należy zauważyć, że w stanach s4, s5, s7, s14 i s15 sygnały wyjściowe automatu skończonego zależą od sygnałów wejściowych, tzn. są sygnałami wyjściowymi automatu Mealy'ego. W tym przypadku mamy połączony model automatów Mealy'ego i Moore'a.

Opis procesora (moduł *PIC_processor*) dla procesora wielocyklowego jest taki sam, jak dla procesora jednocyklowego, z tym wyjątkiem, że urządzenie sterowania (moduł *controller*) ma trzy dodatkowe porty wyjściowe, a urządzenie operacyjne (moduł *datapath*) ma trzy dodatkowe porty wejściowe, które odpowiadają sygnałom *pcwe*, *cwe* i *zwe*. Czytelnik jest proszony o samodzielne opisanie modułu *PIC_processor*.

Opisy pamięci instrukcji IM (moduł *imem*) i pamięci danych DM (moduł *dmem*), a także opis modułu najwyższego poziomu, całkowicie pokrywają się z analogicznymi opisami dla procesora jednocyklowego.

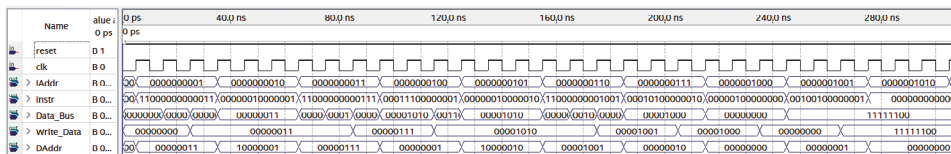
9.1.4. Debugowanie procesora wielocyklowego

Z symulacji naszego procesora wielocyklowego wynika, że działa on poprawnie tylko wtedy, gdy pierwsza instrukcja w programie to **nop** (brak operacji). Poniżej znajduje się program do testowania instrukcji **movlw**, **movwf**, **addwf**, **andwf**, **clrw**, **comf** z poleceniem **nop** jako pierwszym poleceniem.

imemfile_15

```
0000000000000000    nop
0011000000000001    movlw 3    // 3 → W
0000000010000001    movwf 1    // 3 → rf[1]
0011000000000111    movlw 7    // 7 → W
0000011100000001    addwf 1,0 // (rf[1]) + W = 10 → W
0000000010000010    movwf 2    // 10 → rf[2]
0011000000001001    movlw 9    // 9 → W
0000010100000010    andwf 2,1 // rf[2] & W → W
0000000100000000    clrw      // 0 → W
0000100100000001    comf 1,0 // !(rf[1]) → W
```

Wyniki symulacji procesora wielocyklowego PIC przedstawiono na rysunku 9.15.



RYS. 9.15. Wyniki symulacji procesora wielocyklowego PIC

Czytelnik jest proszony o wyjaśnienie, dlaczego pierwsza instrukcja w programie musi być instrukcją **nop**, aby procesor wielocyklowy działał poprawnie.

9.1.5. Analiza projektu procesora wielocyklowego

Analizując nasz projekt dla procesora wielocyklowego PIC widać, że wszystkie instrukcje są wykonywane w ciągu trzech cykli (taktów) zegarowych. Widać to wyraźnie na rysunku 9.14: każda pętla od stanu s0 do stanu s0 przechodzi przez dokładnie trzy stany automatu skończonego.

Analizując rysunek 9.14 można też zauważyć, że stany s0 i s1 można połączyć w jeden stan, ponieważ etap dekodowania kodu instrukcji w naszym procesorze odbywa się za pomocą połączeń przewodowych i nie wymaga sygnału synchronizacji. W przypadku łączenia stanów s0 i s1 w jeden stan, adres kolejnego rozkazu zostanie wyliczony w jednym ze stanów S2 – S15, ponieważ adres następnej instrukcji jest wyznaczany przez układ kombinacyjny bez udziału sygnałów sterujących i nie wymaga cyklu synchronizacji.

Czytnik ma możliwość samodzielnej modyfikacji układu sterującego rozważanego projektu wielocyklowego procesora PIC tak, aby każde polecenie było wykonywane w dwóch cyklach zegarowych.

Wadą tego modelu procesora wielocyklowego PIC jest to, że działa on poprawnie tylko wtedy, gdy pierwsza instrukcja programu jest instrukcją **nop**.

9.2. Projektowanie procesora dwucyklowego PIC

W poprzedniej sekcji czytelnik był zachęcany do zbudowania procesora dwucyklowego PIC przez połączenie stanów S0 i S1 automatu skończonego. Czy jest możliwe zbudowanie procesora PIC z dwoma cyklami, którego automat skończony zawiera tylko dwa stany? Spróbujmy to zrobić, obierając za bazę wyjściową procesor jednocyklowy.

9.2.1. Projektowanie układu procesora dwucyklowego

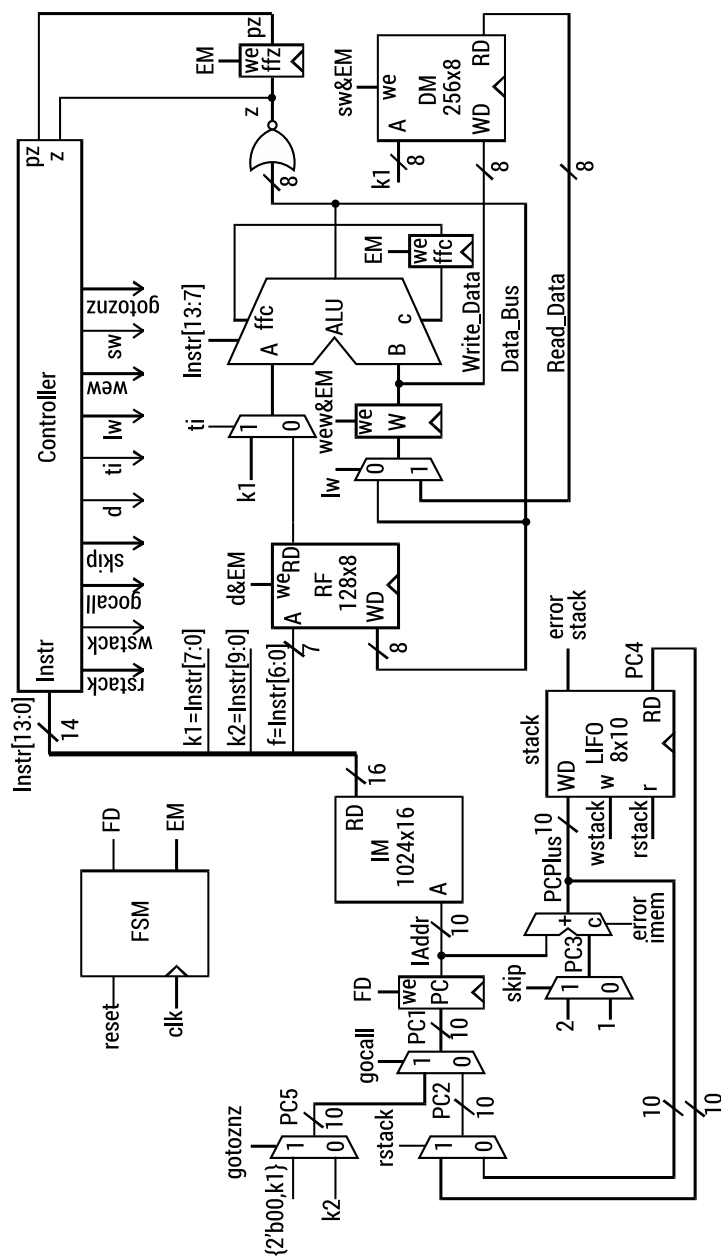
W procesorze PIC etapy pobierania (Fetch) i dekodowania (Decode) instrukcji można połączyć w jeden etap, ponieważ w PIC dekodowanie instrukcji odbywa się za pomocą prostych połączeń przewodowych i nie wymaga sygnału zegarowego. Połączone etapy pobierania Fetch i dekodowania Decode instrukcji oznaczmy jako FD.

W procesorze PIC można również połączyć wykonywanie instrukcji (Execute) i etap zapisu i odczytu pamięci (Memory). Faktem jest, że w procesorze PIC wynik każdej operacji nie jest umieszczany w pamięci danych, ale w akumulatorze W lub w pliku rejestrów RF. Dlatego nie ma potrzeby stosowania etapu Memory. Instrukcje **sw** i **lw** są również wykonywane na etapie wykonywania Execute, podobnie jak inne instrukcje. Połączony etap wykonania instrukcji Execute i zapisu-odczytu pamięci Memory oznaczmy jako EM.

Do układu procesora jednocyklowego PIC wprowadzamy automat skończony, który na przemian generuje sygnały sterujące FD i EM (z nazwy pokrywającej się z zainicjowanymi etapami procesora). Sygnały FD i EM sterują przerzutnikami, rejestrami i blokami pamięci procesora, które muszą być przełączane na odpowiednim etapie.

Na etapie pobierania instrukcji licznik rozkazów PC musi być przełączony, aby sygnał FD umożliwiał zapis do rejestru PC. Wszystkie inne elementy pamięci naszego procesora (plik rejestrów RF, akumulator W, pamięć danych DM, a także przerzutniki *ffc* i *ffz*) przełączają się, gdy aktywny jest sygnał EM. W tym celu sygnał EM jest mnożony logicznie z odpowiednimi sygnałami zezwolenia na zapis każdego komponentu. Na przykład akumulator W przełącza się, gdy spełnione są warunki *wew* & EM.

W pozostałych przypadkach układ operacyjny procesora dwucyklowego PIC całkowicie pokrywa się ze schematem układu operacyjnego dla procesora jednocyklowego (rys. 9.16).



RYS. 9.16. Schemat procesora dwucyklowego PIC

Wszystkie sygnały sterujące są generowane przez kontroler zastosowany w procesorze jednocyklowym PIC. Dlatego nie ma potrzeby projektowania układu sterującego dla dwucyklowego procesora PIC.

9.2.2. Opis w języku Verilog komponentów procesora dwucyklowego PIC

Nowym elementem w schemacie dwucyklowego procesora PIC jest automat skończony FSM_2_cycle (rys. 9.16), który można opisać w języku Verilog w następujący sposób.

```

module FSM_2_cycle(
input clk,reset,
output reg FD,EM);

reg [0:0] state, next;

localparam [0:0] s0=1'b0, s1=1'b1;

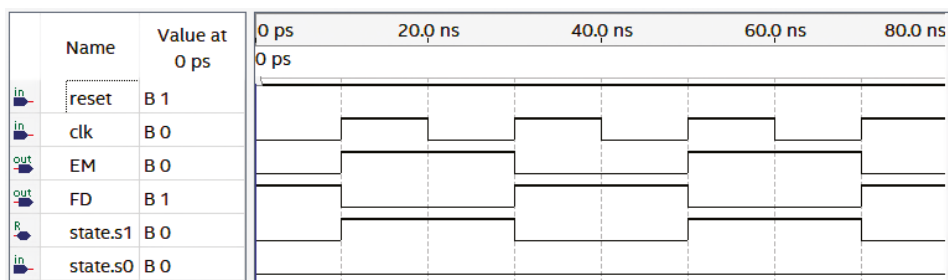
always @(posedge clk, negedge reset)    // opis pamięci automatu
    if (~reset)      state <= s0;
    else             state <= next;

always @(*)                                // opis funkcji przejścia i wyjścia
    case (state)
        s0:  begin next = s1; FD = 1'b1; EM = 1'b0; end
        s1:  begin next = s0; FD = 1'b0; EM = 1'b1; end
    endcase

endmodule

```

Powyższy opis odpowiada opisowi automatów skończonych z dwoma procesami. Wyniki symulacji projektu FSM_2_cycle przedstawiono na rysunku 9.17.



RYS. 9.17. Wyniki symulacji automatu skończonego dla procesora dwucyklowego PIC

Opis w języku Verilog układu sterowania dla dwucyklowego PIC jest dokładnie taki sam, jak opis układu sterowania dla procesora jednocyklowego.

Opis w języku Verilog układu operacyjnego dwucyklowego procesora PIC jest następujący:

```
module datapath_2_cycle
#(parameter
DATA_WIDTH=8,           // szerokość danych
DADDR_WIDTH=8,          // szerokość adresu pamięci danych
INSTR_WIDTH=14,          // rozmiar instrukcji
IADDR_WIDTH=10,          // szerokość adresu pamięci instrukcji
STACK_ADDR_WIDTH=3,      // szerokość adresu stosu
REGFILE_ADDR_WIDTH=7,    // szerokość adresu pliku rejestrów
STACK_DEPTH=(1<<STACK_ADDR_WIDTH), // głębokość stosu
REGFILE_DEPTH=(1<<REGFILE_ADDR_WIDTH)) // głębokość pliku rejestrów

(input clk, reset,           // zegar, reset
input [INSTR_WIDTH-1:0] Instr, // polecenie (instrukcja)
input rstack, wstack, gocall, skip, // sygnały sterujące
input d,                    // kierunek zapisu wyniku ALU
// d = 0 – do W, d = 1 – do pliku rejestrów
input ti, lw, wew,
input [DATA_WIDTH-1:0] Read_Data, // dane z pamięci danych
input gotoznz,              // sygnał sterujący dla instrukcji
// gotoz i gotonz
input FD, EM,               // sygnały sterujące cyklami

output [IADDR_WIDTH-1:0] IAddr, // adres pamięci instrukcji
output [DADDR_WIDTH-1:0] DAddr, // adres pamięci danych
output [DATA_WIDTH-1:0] Write_Data, // dane do pamięci danych
output error_imem, error_stack, // flagi błędów
output z,                   // flaga zero
output [DATA_WIDTH-1:0] Data_Bus, // magistrala danych
output pz);                // znak zera

wire [7:0] k1=Instr[7:0]; // stała formatu c
wire [9:0] k2=Instr[9:0]; // stała formatu d

wire [6:0] f=Instr[6:0]; // numer rejestru w pliku rejestrów
wire [IADDR_WIDTH-1:0] PCplus, PC, PC1, PC2, PC3, PC4, PC5; // magistrale PC
```

```

wire [DATA_WIDTH-1:0] A_ALU, RF_out, W, W1;           // magistrale W.
wire c, wffc, ff_rstack;    // przeniesienie i wyjście przerzutnika przeniesienia

// logika licznika rozkazów
register_we #(IADDR_WIDTH) ex_pc(clk,reset,FD,PC1,PC); // rejestr instrukcji

mux2_1 #(IADDR_WIDTH) ex_pc1(PC2,PC5,gocall,PC1);    // multipleksery
ffc ex_rstack(clk,reset,rstack,ff_rstack);
mux2_1 #(IADDR_WIDTH) ex_pc2(PCPlus,PC4,ff_rstack,PC2);
mux2_1 #(IADDR_WIDTH) ex_pc3(10'd1,10'd2,skip,PC3);
mux2_1 #(IADDR_WIDTH) ex_pc5(k2,{2'b00,k1},gotoznz,PC5);

// sumator PC
adder_PCPlus #(IADDR_WIDTH) ex_adder(PC,PC3,PCPlus,error_imem);

// stos
stack #(IADDR_WIDTH,STACK_ADDR_WIDTH,STACK_DEPTH)
    ex_stack(PCPlus,clk,reset,rstack,wstack,PC4,error_stack);

// plik rejestrów
regfile #(REGFILE_ADDR_WIDTH,REGFILE_DEPTH,DATA_WIDTH)
    ex_regfile(clk,d & EM,f,Data_Bus,RF_out);

// logika ALU i akumulatora W
mux2_1 #(DATA_WIDTH) ex_A_ALU(RF_out,k1,ti,A_ALU);    // multipleksery
mux2_1 #(DATA_WIDTH) ex_W1(Data_Bus,Read_Data,lw,W1);

// akumulator W
register_we #(DATA_WIDTH) ex_W(clk,reset,wew & EM,W1,Write_Data);

// ALU
alu #(.DATA_WIDTH(DATA_WIDTH))
    ex_alu(.a(A_ALU),.b(Write_Data),.in_instr(Instr[13:7]),
    .ffc(wffc),.out(Data_Bus),.c(c));

assign z = (!(|Data_Bus));    // obliczenie zera

my_ff ex_ffc(clk,reset,EM,c,wffc);    // przerzutnik przeniesienia
my_ff ex_ffpz(clk,reset,EM,z,pz);    // przerzutnik zera

```

```

assign IAddr=PC;           // definicja adresu pamięci instrukcji
assign DAddr=k1;          // definicja adresu pamięci danych
endmodule

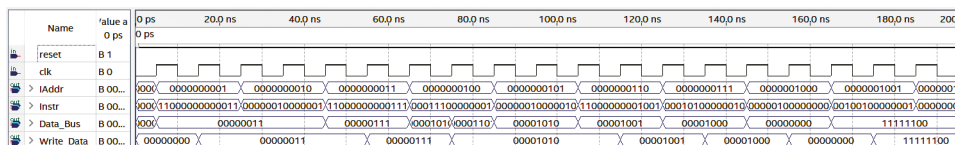
```

Opis pozostałej części procesora dwucyklowego PIC jest identyczny z opisem dla procesora jednocyklowego. Wyjątkami są przerzutniki *ffc* i *ffz*, które są opisane dla procesora wielocyklowego przez moduł *my_ff*.

Czytelnik proszony jest o samodzielne uzupełnienie opisu procesora (moduł *PIC_processor_2_cycle*) oraz modułu najwyższego poziomu dla procesora dwucyklowego PIC.

9.2.3. Debugowanie procesora dwucyklowego

Symulacja działania naszego procesora dwucyklowego PIC pokazuje, że działa on poprawnie tylko wtedy, gdy pierwsza instrukcja w programie to **nop**. Rysunek 9.18 przedstawia wyniki symulacji dla programu *imemfile_15* (patrz podrozdział 9.1.4).



RYS. 9.18. Wyniki symulacji dwucyklowego procesora PIC

Rysunek 9.18 pokazuje, że każde polecenie jest wykonywane w ciągu dwóch taktów synchronizacji.

9.3. Projektowanie czterocyklowego procesora PIC

Zaprojektowaliśmy jedno-, dwu- i trzycyklowe procesory PIC. Może pojawić się pytanie: jak zaprojektować czterocyklowy procesor z tradycyjnymi etapami przetwarzania instrukcji: pobieranie, dekodowanie, wykonywanie i odczyt-zapis pamięci? Na przykład do stworzenia procesora potokowego może być potrzebny czterocyklowy procesor, w którym konieczne jest zwiększenie liczby etapów wykonywania każdej instrukcji w celu zwiększenia wydajności całego procesora.

9.3.1. Projektowanie układu operacyjnego

Stworzymy czterocyklowy procesor PIC oparty na procesorze jednocyklowym, tak jak to zrobiliśmy przy projektowaniu procesora dwucyklowego.

Tradycyjna konstrukcja procesora wielocyklowego (rys. 7.5) zakłada obecność rejestru instrukcji (Instruction Register – IR) w układzie procesora, do którego zapis jest dozwolony przez sygnał *irwe*. Sygnał *irwe* jest generowany na etapie Fetch pobierania instrukcji.

W naszym procesorze nie wprowadzimy osobnego rejestru IR do układu procesora. Zamiast tego modyfikujemy pamięć instrukcji IM, dodając wejście *re* (read enable) zezwolenia odczytu z pamięci, które jest sterowane przez sygnał *irwe*, oraz synchroniczny odczyt z pamięci. W takim przypadku działanie procesora nie ulegnie zmianie: jeśli sygnał *irwe* zostanie ustawiony, to wraz z nadejściem aktywnego zbocza sygnału zegarowego kod instrukcji zostanie odczytany z pamięci instrukcji i ustawiony na magistrali Instr (rys. 9.19).

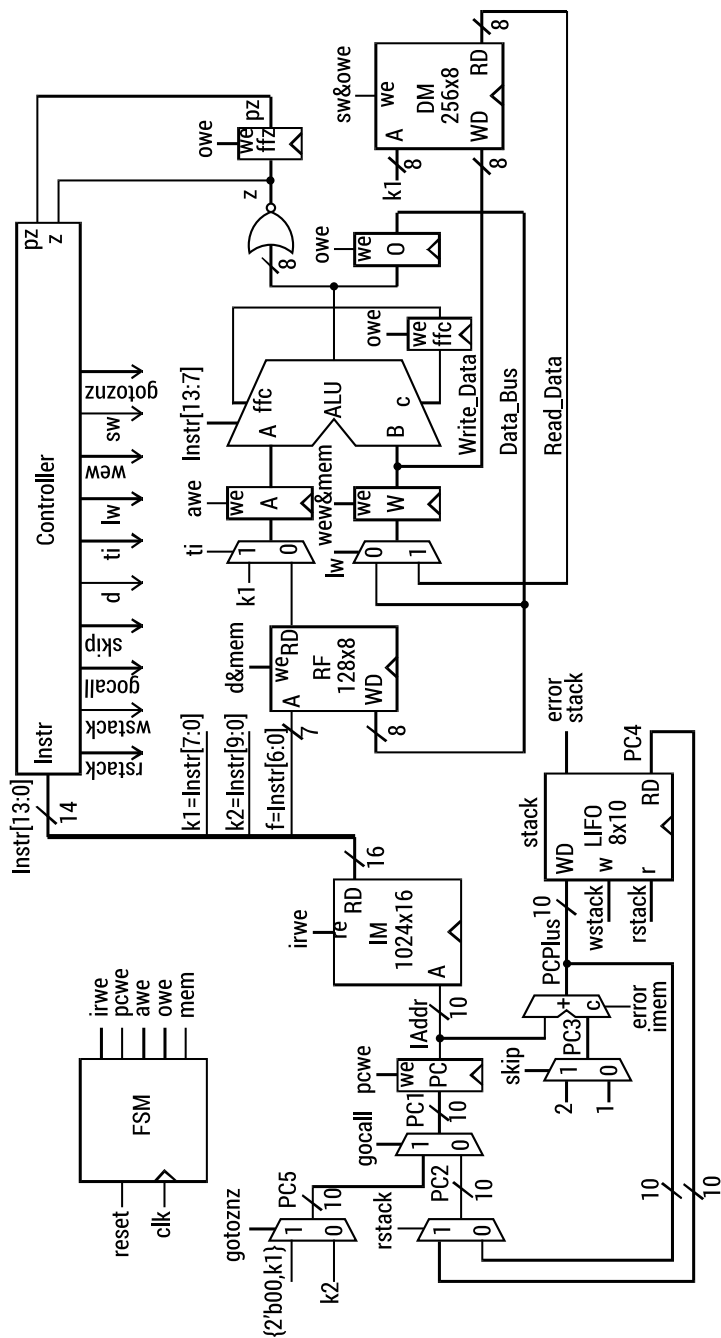
Tradycyjna struktura procesora wielocyklowego na etapie dekodowania zakłada dekodowanie kodu instrukcji i odczytanie z pamięci operandów ALU. W procesorze PIC dekodowanie kodu instrukcji odbywa się za pomocą połączeń przewodowych bez cyklu zegarowego. Argument A pochodzi albo z pliku rejestrów RF, albo jest częścią kodu instrukcji (stała *k1*). Operand B jest zawsze obecny w akumulatorze W i nie wymaga sygnału synchronizacji do jego utworzenia na etapie dekodowania.

Aby jakoś wykorzystać cykl synchronizacji na etapie dekodowania, wprowadzamy rejestr A do układu procesora na wejściu A ALU (rys. 9.19). Zapis do rejestru A jest sterowany przez sygnał *awe*, który jest generowany na etapie dekodowania. Zauważmy, że na etapie dekodowania naszego procesora PIC generowany jest również sygnał *pcwe*, aby umożliwić zapis do rejestru licznika rozkazów PC.

W tradycyjnej strukturze procesora wielocyklowego etap wykonania obejmuje wykonanie operacji, tzn. obliczenie wyniku za pomocą ALU i zapisanie go do rejestru wyjściowego C (rys. 7.5). W naszym czterocyklowym procesorze PIC nazwiemy rejestr wyjściowy O (out), aby nie mylić go z sygnałem przeniesienia *c* (rys. 9.19). Rejestr O jest zapisywany, gdy ustawiony jest sygnał *owe*, który jest generowany na etapie Execute wykonywania instrukcji. Dodatkowo sygnał sterujący *owe* pozwala na zapisanie flag przeniesienia *c* i zera *z* do przerzutników *ffc* i *ffz*.

Zwróćmy uwagę, że instrukcje *sw* i *lw* (zapisu wartości akumulatora W w pamięci danych DM i ładowania akumulatora W z pamięci danych) są wykonywane na etapie wykonywania Execute jak zwykłe instrukcje. Dlatego wejście zezwolenia zapisu *we* w pamięci danych DM jest kontrolowane przez wyrażenie *sw & owe*.

W naszym procesorze PIC wynik operacji zapisywany jest albo do pliku rejestrów RF albo do akumulatora W. Dlatego też pamięcią w naszym procesorze PIC jest plik rejestrów RF i akumulator W. Na etapie Memory naszego procesora wynik operacji zapisywany jest do pliku rejestrów RF lub akumulatora W. W tym celu na etapie Memory jest generowany sygnał *mem*. Zatem wejście *we* pliku rejestrów RF jest sterowane przez wyrażenie *d & mem*, a wejście *we* akumulatora W jest sterowane przez wyrażenie *wew & mem*.



Ostateczny schemat procesora czterocyklowego pokazano na rysunku 9.19, a opis układu operacyjnego w języku Verilog jest następujący:

module datapath_4_cycle

#(parameter

DATA_WIDTH=8, // szerokość danych
DADDR_WIDTH=8, // szerokość adresu pamięci danych
INSTR_WIDTH=14, // rozmiar instrukcji
IADDR_WIDTH=10, // szerokość adresu pamięci instrukcji
STACK_ADDR_WIDTH=3, // szerokość adresu stosu
REGFILE_ADDR_WIDTH=7, // szerokość adresu pliku rejestrów
STACK_DEPTH=(1<<STACK_ADDR_WIDTH), // głębokość stosu
REGFILE_DEPTH=(1<<REGFILE_ADDR_WIDTH)) // głębokość pliku rejestrów

(input clk, reset, // zegar, reset

input [INSTR_WIDTH-1:0] Instr, // polecenie (instrukcja) z pamięci IM

input irwe, pcwe, awe, owe, mem, // sygnały od FSM sterujące cyklami

input rstack, wstack, gocall, skip, // sygnały sterujące

input d, // kierunek zapisu wyniku ALU

// d = 0 – do W, d = 1 – do pliku rejestrów

input ti, lw, wew, // sygnały sterujące

input [DATA_WIDTH-1:0] Read_Data, // dane z pamięci danych

input gotoznz, // sygnał sterujący dla instrukcji gotoz i gotonz

output [IADDR_WIDTH-1:0] IAddr, // adres pamięci instrukcji

output [DADDR_WIDTH-1:0] DAddr, // adres pamięci danych

output [DATA_WIDTH-1:0] Write_Data, // dane do pamięci danych

output error_imem, error_stack, // flagi błędów

output z, // flaga zero

output [DATA_WIDTH-1:0] Data_Bus, // magistrala danych

output pz; // znak zera

wire [7:0] k1=Instr[7:0]; // stała formatu c

wire [9:0] k2=Instr[9:0]; // stała formatu d

wire [6:0] f=Instr[6:0]; // numer rejestru w pliku rejestrów

wire [IADDR_WIDTH-1:0] PCPlus, PC, PC1, PC2, PC3, PC4, PC5; // magistrale PC

wire [DATA_WIDTH-1:0] A_ALU, RF_out, W, W1; // magistrale W

wire c, wffc, ff_rstack; // przeniesienia i wyjście przerzutnika przeniesienia

```

wire [DATA_WIDTH-1:0] IA, ALU_out;      // wejście rejestru A, wyjście ALU

// logika licznika rozkazów
register_we #(IADDR_WIDTH) ex_pc(clk,reset,pcwe,PC1,PC);    // rejestr instrukcji

mux2_1 #(IADDR_WIDTH) ex_pc1(PC2,PC5,gocall,PC1);    // multipleksery
mux2_1 #(IADDR_WIDTH) ex_pc2(PCPlus,PC4,rstack,PC2);
mux2_1 #(IADDR_WIDTH) ex_pc3(10'd1,10'd2,skip,PC3);
mux2_1 #(IADDR_WIDTH) ex_pc5(k2,{2'b00,k1},gotoznz,PC5);

// sumator PC
adder_PCPlus #(IADDR_WIDTH) ex_adder(PC,PC3,PCPlus,error_imem);

stack #(IADDR_WIDTH,STACK_ADDR_WIDTH,STACK_DEPTH)    // stos
    ex_stack(PCPlus,clk,reset,rstack,wstack,PC4,error_stack);

// plik rejestrów
regfile #(REGFILE_ADDR_WIDTH,REGFILE_DEPTH,DATA_WIDTH)
    ex_regfile(clk,d & mem,f,Data_Bus,RF_out);

// Logika ALU i akumulatora W
mux2_1 #(DATA_WIDTH) ex_A_ALU(RF_out,k1,ti,IA);      // multipleksery
mux2_1 #(DATA_WIDTH) ex_W1(Data_Bus,Read_Data,lw,W1);

// rejestr wejścia A ALU
register_we #(DATA_WIDTH) ex_A(clk,reset,awe,IA,A_ALU);

// akumulator W
register_we #(DATA_WIDTH) ex_W(clk,reset,wew & mem,W1,Write_Data);

// ALU
alu #(.DATA_WIDTH(DATA_WIDTH))
    ex_alu(.a(A_ALU),.b(Write_Data),.in_instr(Instr[13:7]),
    .ffc(wffc),.out(ALU_out),.c(c));

// rejestr wyjściowy ALU
register_we #(DATA_WIDTH) ex_O(clk,reset,owe,ALU_out,Data_Bus);

assign z = (!(|ALU_out));    // obliczenie zera

```

```

my_ff ex_ffc(clk,reset,owe,c,wffc); // przerzutnik przeniesienia
my_ff ex_ffpz(clk,reset,owe,z,pz); // przerzutnik zero

assign IAddr=PC; // definicja adresu pamięci instrukcji
assign DAddr=k1; // definicja adresu pamięci danych
endmodule

```

9.3.2. Projektowanie układu sterującego

Układ sterowania naszego procesora czterocyklowego jest połączeniem układu sterowania procesorem jednocyklowym i automatu skończonego typu Moore'a, który zawiera cztery stany: S0, S1, S2 i S3. Stany S0, S1, S2 i S3 odpowiadają etapom pobierania Fetch, dekodowania Decode, wykonywania Execute i zapis-odczyt pamięci Memory. W stanie S0 generowany jest sygnał *irwe*, umożliwiający odczytanie instrukcji z pamięci instrukcji IM; w stanie S1 generowany jest sygnał *pcwe*, który umożliwia zapis do licznika rozkazów PC oraz sygnał *awe*, który umożliwia zapisanie argumentu A do rejestru A; w stanie S2 generowany jest sygnał *owe*, pozwalający na zapis do rejestru wyniku operacji O, a także do pamięci danych DM oraz do przerzutników ffz i ffc; w stanie S3 generowany jest sygnał *mem* umożliwiający zapis do pliku rejestrów RF i akumulatora W.

Opis automatu skończonego w języku Verilog może wyglądać następująco:

```

module FSM_4_cycle(
input clk,reset,
output reg irwe, pcwe, awe, owe, mem);

reg [1:0] state, next; // zmienne stanu

localparam [1:0] // stany automatu
s0=0,s1=1,s2=2,s3=3;

always @(posedge clk, negedge reset) // opis pamięci automatu
if (~reset) state <= s0;
else state <= next;

always @(*) // opis funkcji przejścia
case (state)
s0: next = s1;
s1: next = s2;
s2: next = s3;
s3: next = s0;

```



```

        default: next = s0;
    endcase

always @(*) // opis funkcji wyjściowych
    begin
        irwe=1'b0; pcwe=1'b0; awe=1'b0; owe=1'b0; mem=1'b0;
        case (state)
            s0: irwe = 1'b1;
            s1: begin pcwe = 1'b1; awe = 1'b1; end
            s2: owe = 1'b1;
            s3: mem = 1'b1;
        endcase
    end
endmodule

```

9.3.3. Opis w języku Verilog komponentów procesora czterocyklowego

Schemat działania procesora czterocyklowego został opisany wcześniej. Jednostka sterująca procesora czterocyklowego jest identyczna z jednostką sterującą procesora jednocyklowego. Do układu procesora jest dodawany, opisany powyżej automat skończony (moduł *FSM_4_cycle*). Reszta komponentów procesora czterocyklowego jest repliką elementów z projektu procesora dwucyklowego. Wyjątkiem jest pamięć instrukcji IM.

Pamięć instrukcji IM dla procesora czterocyklowego ma dodatkowo wejście synchronizacji *clk* i zezwolenie odczytu *re*. Pamięć instrukcji IM można opisać w języku Verilog w następujący sposób:

```

module imem_4_cycle
#(parameter ADDR_SIZE=10, DEPTH=1024, WORD_SIZE=16)
(input clk, re,
input [ADDR_SIZE-1:0] a,
output reg [WORD_SIZE-1:0] rd);

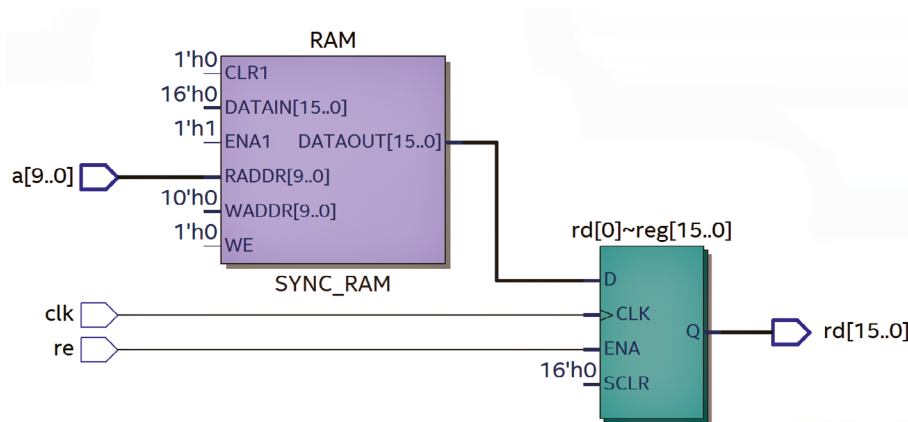
reg [WORD_SIZE-1:0] RAM[DEPTH-1:0];

initial
    $readmemb("imemfile_1.txt", RAM);

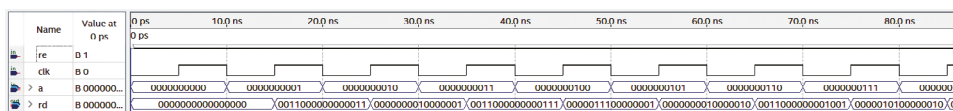
always @(posedge clk)
    if (re) rd = RAM[a];
endmodule

```

Wyniki syntezy i symulacji projektu *imem_4_cycle* przedstawiono, odpowiednio, na rysunkach 9.20 i 9.21.



RYS. 9.20. Wyniki syntezy projektu *imem_4_cycle*

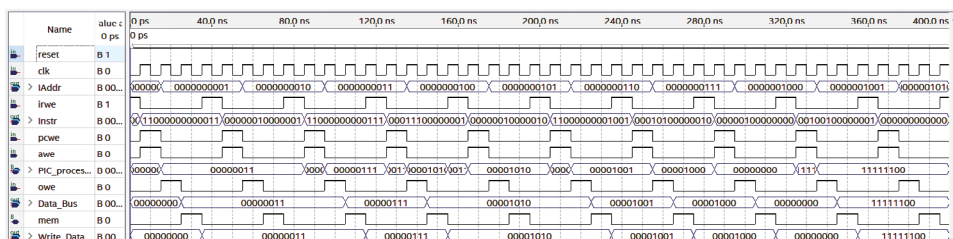


RYS. 9.21. Wyniki symulacji projektu *imem_4_cycle*

Czytelnik jest proszony o samodzielne uzupełnienie opisu w języku Verilog procesora czterocyklowego (moduł *PIC_processor_4_cycle*) oraz modułu najwyższego poziomu.

9.3.4. Debugowanie procesora czterocyklowego

Sprawdźmy działanie procesora czterocyklowego podczas wykonywania programu z pliku *imemfile_1* (patrz rozdział 9.1). Wyniki symulacji przedstawiono na rysunku 9.22.



RYS. 9.22. Wyniki symulacji procesora czterocyklowego PIC

Rysunek 9.22 pokazuje, że każda instrukcja w czterocyklowym procesorze jest wykonywana w czterech cyklach zegarowych. Procesor nie wymaga użycia instrukcji **nop** jako pierwszej instrukcji w programie.

9.4. Ocena kosztów realizacji i wydajności procesorów PIC

Wprowadźmy następujące oznaczenia dla projektów opracowanych procesorów:

- *single-cycle* – dla procesora jednocyklowego z rozdziału 8;
- *multi-cycle* – dla procesora wielocyklowego (trzyktyklowego) z rozdziału 9.1;
- *multi-2-cycle* – dla procesora dwucyklowego z rozdziału 9.2;
- *multi-4-cycle* – dla procesora czterocyklowego z rozdziału 9.3.

Dla dokładniejszego oszacowania kosztu realizacji, konieczne jest usunięcie atrybutu *keep* we wszystkich opisach projektów, który był używany na etapie debugowania. Aby poprawnie określić parametry czasowe, dla każdego projektu ustawiono plik ograniczeń czasowych z okresem (period) 20 ns dla głównego sygnału zegarowego i wirtualnych sygnałów zegarowych wejść i wyjść. Plik ograniczeń ma następującą postać:

```
create_clock -name clk -period 20.000 [get_ports {clk}]
create_clock -name virt_clk_in -period 20.000
create_clock -name virt_clk_out -period 20.000
derive_pll_clocks -create_base_clocks
derive_clock_uncertainty
set_input_delay -clock { virt_clk_in } -min 0 [get_ports {reset}]
set_input_delay -clock { virt_clk_in } -max 1 [get_ports {reset}]
set_output_delay -clock { virt_clk_out } -min 0 [get_ports {lAddr[*] DAddr[*] \
Write_Data[*] Read_Data[*] error_imem error_stack Data_Bus[*] Instr[*]}]
set_output_delay -clock { virt_clk_out } -max 1 [get_ports {lAddr[*] DAddr[*] \
Write_Data[*] Read_Data[*] error_imem error_stack Data_Bus[*] Instr[*]}]
```

Ponieważ koszt realizacji naszych projektów zależy od programu inicjującego pamięć instrukcji IM, projekty *multi-cycle* i *multi-2-cycle* zostały przetestowane z plikiem inicjalizacyjnym pamięci *imemfile_15*, a projekty *single-cycle* i *multi-4-cycle* z plikiem inicjującym pamięć *imemfile_1*. Badania eksperymentalne przeprowadzono przy użyciu systemu Quartus Prime w wersji 18.1 na układach FPGA z rodziny Cyclone IV SE.

Wyniki badań kosztu realizacji jednocyklowego i wielocyklowych procesorów PIC oraz ich elementów przedstawiono w tabeli 9.1, gdzie *L* to liczba użytych elementów logicznych FPGA; *R* to liczba używanych przerzutników; *M* to liczba używanej pamięci w bitach; Σ – suma arytmetyczna.

Analiza tabeli 9.1 pokazuje, że projekty procesorów nie wykorzystują pamięci wbudowanej FPGA. Wyjątkiem jest projekt *multi-4-cycle*, który wykorzystuje 2176 bitów z bloków pamięci wbudowanej.

Najmniejsza liczba elementów logicznych jest wymagana, aby zaimplementować procesor czterocyklowy *multi-4-cycle* ($L = 285$), a następnie projekt *multi-cycle* ($L = 545$). Projekty *single-cycle* i *multi-2-cycle* wykorzystują w przybliżeniu taką samą liczbę elementów logicznych i przerzutników (odpowiednio: 717–719 i 410–411). Najniższy koszt realizacji projektu *multi-4-cycle* interpretowany jest efektywnym wykorzystaniem bloków pamięci wbudowanej FPGA.

TAB. 9.1. Wyniki badania kosztów realizacji procesorów PIC

Komponent	single-cycle			multi-cycle			multi-2-cycle			multi-4-cycle		
	<i>L</i>	<i>R</i>	<i>M</i>	<i>L</i>	<i>R</i>	<i>M</i>	<i>L</i>	<i>R</i>	<i>M</i>	<i>L</i>	<i>R</i>	<i>M</i>
PIC_processor	369	154	0	361	158	0	365	154	0	242	58	128
controller	10	0	0	20	10	0	8	0	0	11	0	0
datapath	359	154	0	341	148	0	358	154	0	231	58	128
dmem	330	256	0	169	128	0	330	256	0	19	19	2048
imem	20	0	0	20	0	0	20	0	0	23	10	0
FSM	0	0	0	0	0	0	1	1	0	6	4	0
Σ	719	410	0	545	286	0	717	411	0	285	91	2176

Spośród komponentów procesora, największa liczba elementów logicznych jest wymagana do implementacji układu operacyjnego *datapath* (361–329) i pamięci danych *dmem* (341–330). Wyjątkiem jest projekt *multi-4-cycle*, w którym plik rejestrów i pamięć danych są zaimplementowane w blokach pamięci wbudowanej FPGA.

Wyniki badania szybkości rozpatrywanych projektów procesorów podano w tabeli 9.2, gdzie F jest maksymalną częstotliwością roboczą (w megahercach); D_{max} i D_{av} – maksymalną i średnią głębokością warstw elementów logicznych w układzie.

TAB. 9.2. Wyniki badania szybkości procesorów PIC

Projekt	F	D_{max}	D_{av}
single-cycle	77.67	17.0	8.80
multi-cycle	79.40	14.0	8.84
multi-2-cycle	78.27	17.0	8.77
multi-4-cycle	101.1	11.9	7.26

Analiza tabeli 9.2 pokazuje, że projekt *multi-4-cycle* ($F = 101,1$ MHz) charakteryzuje się najwyższą szybkością działania, inne projekty mają w przybliżeniu tę samą szybkość działania (od 77,61 do 79,40 MHz). Największa wydajność projektu

multi-4-cycle wynika również z zastosowania bloków pamięci wbudowanej FPGA do implementacji komponentów pamięci.

Maksymalna (D_{max}) i średnia (D_{av}) liczba warstw elementów LUT wpływa na szybkość działania procesorów, ale wpływ ten nie zawsze jest oczywisty.

Oszacujmy wydajność rozpatrywanych procesorów w przypadku ich wykorzystania w potokowym przetwarzaniu danych. Wyniki takich badań przedstawiono w tabeli 9.3, gdzie T jest czasem trwania sygnału synchronizacji; n_1 , n_{10} , n_{100} i n_{1000} to liczba cykli zegarowych wymaganych do wykonania odpowiednio jednego, dziesięciu, stu i tysiąca instrukcji; t_1 , t_{10} , t_{100} i t_{1000} to czas potrzebny procesorowi do wykonania odpowiednio jednej, dziesięciu, stu i tysiąca instrukcji. Wartości czasu w tabeli 9.3 są podane w nanosekundach.

TAB. 9.3. Wyniki badania wydajności procesorów PIC

Projekt	T	n_1	t_1	n_{10}	t_{10}	n_{100}	t_{100}	n_{1000}	t_{1000}
single-cycle	12.87	1	12.87	10	128.70	100	1287.00	1000	12870.00
multi-cycle	12.59	3	37.77	12	151.08	102	1284.18	1002	12615.18
multi-2-cycle	12.78	2	25.56	11	140.58	101	1290.78	1001	12792.78
multi-4-cycle	9.89	4	39.56	13	128.57	103	1018.67	1003	9919.67

Analiza tabeli 9.3 pokazuje, że gdy wykonywana jest jedna instrukcja (t_1), procesor jednocyklowy *single-cycle* jest niekwestionowanym liderem wydajności. W przypadku dziesięciu instrukcji (t_{10}) procesor czterocyklowy *multi-4-cycle* ma niewielką przewagę nad procesorem jednocyklowym *single-cycle*. Przy stu instrukcjach (t_{100}) przewaga procesora czterocyklowego *multi-4-cycle* staje się oczywista. Ponadto procesor wielocyklowy *multi-cycle* przewyższa również procesor jednocyklowy *single-cycle*. W przypadku wykonania tysiąca instrukcji (t_{1000}) wszystkie procesory wielocyklowe mają przewagę nad procesorem jednocyklowym.

Zauważmy, że wraz ze wzrostem liczby instrukcji w programie rośnie przewaga procesorów wielocyklowych nad procesorem jednocyklowym. W taki sposób w pełni przejawia się wydajność procesorów wielocyklowych w przypadku potokowego przetwarzania danych.

9.5. Wnioski

W rozdziale tym przedstawiono kilka metodyk projektowania procesorów wielocyklowych.

W pierwszej metodyce (projekt *multi_cycle*), układ sterowania jest zaimplementowany jako automat skończony. W tym przypadku zakłada się sekwencyjne przechodzenie stanów automatu skończonego: wybranie kodu instrukcji z pamięci programu, dekodowanie instrukcji, zaimplementowanie określonych grup rozkazów

i poszczególnych instrukcji. Liczba stanów automatu skończonego przy takim podejściu może być dowolna. Każdej instrukcji odpowiada określona pętla automatu skończonego, która zaczyna się i kończy się w stanie początkowym. Pętle automatu skończonego mogą odpowiadać jednemu rozkazowi, grupie rozkazów, grupie rozkazów o określonym formacie itp. Liczba etapów (cykli) wykonania każdej instrukcji jest określona przez liczbę stanów w pętli odpowiadającej danej instrukcji. Różne instrukcje mogą być wykonywane w różnej liczbie cykli zegarowych, tzn. odpowiadające im pętle przechodzą przez różną liczbę stanów. Liczba cykli procesora zależy od długości najdłuższej pętli. Zwykle liczba stanów automatu skończonego w tym podejściu przekracza liczbę cykli procesora wielocyklowego.

Druuga metodyka (projekt *multi_2_cycle*) zakłada maksymalne wykorzystanie układu operacyjnego i układu sterującego procesora jednocyklowego. Aby określić cykle procesora, do układu procesora wprowadzany jest dodatkowy automat skończony, który generuje sygnały sterujące (w naszym przykładzie sygnały FD i EM), które inicjują wykonanie pewnych etapów procesora wielocyklowego. Wprowadzone sygnały są mnożone logicznie przez odpowiadające im sygnały sterujące procesora jednocyklowego. Liczba stanów automatu skończonego, w przypadku zastosowania tej metodyki, jest równa liczbie cykli procesora wielocyklowego (w naszym przykładzie 2 cykle).

Trzecia metodyka ma na celu zwiększenie liczby etapów procesora wielocyklowego. W takim przypadku do układu operacyjnego, w porównaniu z procesorem jednocyklowym, dodaje się rejestry, aby stworzyć wymaganą liczbę etapów procesora. Aby zbudować urządzenie sterowania, można zastosować zarówno pierwszą, jak i drugą metodykę.

W procesorze wielocyklowym elementy pamięci (rejestry, przerzutniki, plik rejestrów i bloki pamięci) z reguły muszą mieć wejścia zezwolenia zapisu *we*, które są sterowane sygnałami generowanymi na różnych etapach wykonywania instrukcji.

Układ sterowania procesora wielocyklowego jest zbudowany w postaci automatu skończonego. W tym przypadku automat skończony może być zarówno automatem typu Moore'a, jak i automatem typu Mealy'ego. Ponadto automat skończony może być typu mieszanego, tzn. automat skończony może mieć funkcje wyjściowe zarówno automatu Moore'a, jak i automatu Mealy'ego.

Układ sterowania procesora wielocyklowego może również składać się z układu sterującego procesorem jednocyklowym i automatu skończonego. Sygnały wyjściowe automatu skończonego określają etapy procesora i są mnożone logicznie przez odpowiadające im wartości wyjściowe układu sterowania procesorem jednocyklowym.

Wykorzystanie bloków pamięci wbudowanej FPGA do implementacji komponentów pamięci procesora może znacząco obniżyć koszty realizacji i zwiększyć wydajność projektu.

Wydajność procesorów wielocyklowych przejawia się w pełni w przypadku potokowego przetwarzania danych, dla dużych programów zawierających 1000 lub więcej instrukcji.

Część IV

Projektowanie bloków cyfrowego przetwarzania sygnałów

Jedną z cech wyróżniających nowoczesne systemy wbudowane jest wykonywanie szerokiego zakresu zadań z dziedziny *cyfrowego przetwarzania sygnałów* (Digital Signal Processing – DSP). Wynika to z faktu, że metody DSP są coraz częściej wykorzystywane w różnych sferach działalności człowieka. Wspomnijmy tylko o niektórych obszarach wykorzystania DSP:

- przeznaczenie ogólne: filtrowanie, filtrowanie adaptacyjne, wykrywanie i korelacja sygnałów, analiza widmowa i transformata Fouriera;
- przetwarzanie mowy: kodowanie i dekodowanie, ocena tekstu, rozpoznawanie i synteza mowy, usuwanie echa;
- przetwarzanie sygnałów audio: tłumienie szumów, korekcja dźwięku, emulacja akustyki otoczenia, miksowanie i edycja dźwięku, synteza dźwięku;
- przetwarzanie obrazu: widzenie maszynowe, rozpoznawanie wzorców, rotacja i dekompozycja obrazu, poprawa jakości obrazu, animacja, faksymile (szczegółowe odwzorowanie oryginału graficznego), kompresja i dekompresja danych;
- systemy informacyjne: poczta głosowa, faksy, modemy, telefony komórkowe, korektory, szyfrowanie i deszyfrowanie danych, sieci przewodowe i bezprzewodowe, radio i telewizja;
- sterowanie: silniki, nawigacja, wibracje, pobór mocy, robotyka, serwo sterowanie.

Systemy wbudowane z wykorzystaniem cyfrowego przetwarzania sygnałów znajdują zastosowanie w takich dziedzinach jak:

- kosmos: fotografia z kosmosu, kompresja danych, inteligentna analiza danych sensorycznych;
- medycyna: przetwarzanie sygnałów biomedycznych, tomografia komputerowa, rezonans magnetyczny, elektrokardiografia, fluoroskopia, aparaty do obserwacji pacjenta, aparaty słuchowe;
- telekomunikacja: kompresja głosu i obrazu, eliminacja echa, multipleksowanie sygnałów, filtrowanie, korekcja adaptacyjna, rozpraszanie widma;
- sfera wojskowa: radar, sonar, systemy naprowadzania, komunikacja zamknięta, selektywne zagłuszanie komunikacji radiowej;

- przemysł: monitoring i kontrola procesów przemysłowych, testowanie nieniszczące, kontrola położenia i prędkości, redukcja hałasu, poszukiwanie złóż surowców;
- badania naukowe: zbieranie danych, analiza spektralna, symulacja i modelowanie;
- sprzęt codziennego użytku: aparaty cyfrowe, kamery cyfrowe, odtwarzacze audio i wideo, telefony komórkowe, automatyczne sekretarki, faksy, modemy.

Z powyższego przeglądu wynika, że obszaru projektowania systemów cyfrowego przetwarzania sygnałów na układach FPGA nie da się objąć jedną książką. W tej pozycji ograniczymy się tylko do rozważenia zagadnienia projektowania filtrów cyfrowych w FPGA.

10. Projektowanie filtrów cyfrowych

Rozdział zawiera krótkie wprowadzenie do *cyfrowego przetwarzania sygnałów* (Digital Signal Processing – DSP) i wskazuje możliwości FPGA w zakresie implementacji systemów DSP. Określono główne parametry filtrów cyfrowych, przedstawiono metody opisu filtrów cyfrowych w języku Verilog. Przedstawiono ogólną metodologię projektowania filtrów cyfrowych, rozważono przykład opracowania filtra cyfrowego w systemie MATLAB, symulację filtra w systemie ModelSim oraz implementację filtra w systemie Quartus. Na koniec pokazano wyniki badań realizacji różnych typów filtrów dla różnych rodzin FPGA.

10.1. Systemy cyfrowego przetwarzania sygnałów na FPGA

10.1.1. Wprowadzenie do cyfrowego przetwarzania sygnałów

W dziedzinie cyfrowego przetwarzania sygnałów przez sygnał rozumie się zmienną zawierającą pewnego rodzaju informację, z którą można wykonywać określone czynności: konwertować (przetwarzać), przysyłać itp. Generalnie sygnały dzielą się na *sygnały analogowe (ciągłe)* i *sygnały dyskretne (cyfrowe)*. Konwersja sygnału analogowego do postaci cyfrowej nazywa się *kwantyzacją* (kwantowaniem). *Cyfrowe przetwarzanie sygnałów* (DSP) to zbiór metod i algorytmów matematycznych oraz realizujących je programów i urządzeń technicznych służących do konwersji sygnałów dyskretnych.

W otaczającej nas rzeczywistości mamy do czynienia z sygnałami analogowymi, dlatego zastosowanie układów DSP (rys. 10.1) wiąże się zwykle z zastosowaniem *przetworników analogowo-cyfrowych* (analog-to-digital converter – ADC) oraz *przetworników cyfrowo-analogowych* (digital-to-analog converter – DAC).



RYS. 10.1. Typowe zastosowanie układu DSP: $x(t)$ – wejście analogowe; $x(k)$ – wejście dyskretne; $y(k)$ – wyjście dyskretne; $y(t)$ – wyjście analogowe

Już w XIX wieku wykazano, że każdy sygnał można przedstawić jako sumę sygnałów sinusoidalnych. Pojedynczy sygnał sinusoidalny nazywany jest *harmoniczną* (harmonic). Każda harmoniczna charakteryzuje się własną częstotliwością i amplitudą. Reprezentacja sygnału jako suma harmonicznych nazywana jest *widmem sygnału* (signal spectrum). Harmoniczna o największej amplitudzie w widmie sygnału nazywana jest *główną harmoniczną* (main harmonic). Jeżeli liczba harmonicznych w widmie sygnału jest skończona, to zgodnie z twierdzeniem Nyquista-Shannona-Kotelnikowa, przy próbkowaniu sygnału analogowego bez utraty informacji, częstotliwość próbkowania musi być co najmniej dwa razy większa od maksymalnej częstotliwości harmonicznej w widmie sygnału.

10.1.2. Zastosowanie FPGA do realizacji systemów DSP

Ponieważ systemy DSP należą do klasy systemów cyfrowych, do ich realizacji można wykorzystać dowolne elementy, za pomocą których budowane są systemy cyfrowe. Jednak do budowy systemów DSP, ze względu na ich specyfikę, najczęściej stosuje się procesory DSP, układy ASIC i układy FPGA.

W porównaniu z procesorami DSP użycie układów FPGA może znacznie zwiększyć wydajność systemu ze względu na:

- dużą wydajność FPGA;
- szerszy kanał transmisji danych;
- możliwość podzielenia strumienia danych na kilka równoległych bloków przetwarzania;
- możliwość wykonywania operacji potokowych.

W porównaniu z układami ASIC układy FPGA mają następujące zalety:

- redukcja kosztów i czasu związanych z projektowaniem i wdrożeniem systemu;
- elastyczność w realizacji projektów, tzn. możliwość modyfikacji projektu na dowolnym etapie projektowania (np. można wybrać inną metodę obliczania współczynników filtra, gdy płytką drukowaną została już opracowana);
- łatwość integracji projektu na poziomie systemu;
- możliwość przejścia na układy ASIC po pełnym przetestowaniu projektu.

Przyjrzyjmy się bliżej właściwościom architektur nowoczesnych układów FPGA, które pozwalają efektywnie realizować systemy DSP. Właściwości te obejmują obecność w układzie FPGA:

- wbudowanych bloków DSP;
- bloków pamięci wbudowanej i pamięci rozproszonej;
- interfejsu z pamięcią zewnętrzną;
- wbudowanego procesora (wbudowanych rdzeni procesorów);
- dużej liczby zewnętrznych wyprowadzeń ogólnego przeznaczenia;
- sieci synchronizacji i bloków PLL.

Ponadto nowoczesne układy FPGA oferują możliwość konfiguracji bloków pamięci jako pamięci typu FIFO i dużych rejestrów przesuwających, a także zawierają w swojej strukturze dużą liczbę rejestrów, co ułatwia potokowe przetwarzanie danych.

Wbudowane bloki DSP pozwalają na efektywną implementację takich elementów składowych układów DSP jak układy mnożące, *układy mnożące z akumulacją* (Multiply and Accumulate – MAC), sumatory (subtraktory), pre-sumatory.

Bloki pamięci wbudowanej FPGA umożliwiają różne tryby konfiguracji, takie jak jedno- i dwuportowe pamięci typu RAM i ROM oraz pamięć typu FIFO. Bloki pamięci wbudowanej mogą przechowywać wartości współczynników filtra, a także wartości pośrednie wyników. Pamięć typu FIFO może służyć jako bufor wejściowych i wyjściowych strumieni danych podczas budowania systemów DSP czasu rzeczywistego. Ponadto istnieje możliwość skonfigurowania bloków pamięci wbudowanej do implementacji dużych rejestrów przesuwających. Takie rejestry przesuwające są używane do przechowywania wartości próbek wejściowych podczas implementacji filtrów cyfrowych. W przypadku braku pamięci wbudowanej można skorzystać z interfejsu do pamięci zewnętrznej.

Jeśli zasobów logicznych FPGA nie wystarcza do zbudowania układu DSP, można skorzystać z procesora wbudowanego dostępnego w architekturze FPGA (w tym przypadku układ FPGA nazywamy *systemem na jednym układzie* – System on Chip – SoC). Procesor wbudowany można również zaimplementować, konfigurując część logiki FPGA przy użyciu bloków IP.

Duża liczba zewnętrznych wyprowadzeń FPGA pozwala na budowanie wielokanałowych układów DSP. Ponadto bloki wejścia-wyjścia niektórych układów FPGA zapewniają dodatkowe opcje wstępnego przetwarzania informacji.

Złożone systemy DSP zwykle używają kilku różnych sygnałów taktowania. Aby zbudować podsystem synchronizacji w architekturze FPGA, mamy do dyspozycji globalne i lokalne sieci sygnałów zegarowych i sygnałów sterujących rejestrami, a także bloki *pętli synchronizacji fazowej* (phase-locked loop – PLL).

Rozważane właściwości architektur FPGA pozwalają stwierdzić, że architektury nowoczesnych układów FPGA są ukierunkowane na wydajną implementację systemów DSP. Ponadto architektury FPGA ułatwiają równoległe i potokowe przetwarzanie danych.

Zwróćmy uwagę, że niektóre narzędzia projektowe pozwalają dostosować algorytmy DSP do określonej architektury docelowego układu FPGA (na przykład liczba wejść LUT, liczba równoległych kanałów przetwarzania danych, liczba bloków DSP w łańcuchu kanałów przetwarzania danych itp.). W tym przypadku korzyści płynące z wykorzystania FPGA do implementacji systemu DSP mogą być ogromne.

Ponadto algorytmy DSP można skonfigurować tak, aby dostosowywały się do wymagań systemu „w locie” (on the fly), tzn. podczas działania systemu (na przykład mogą być załadowane inne współczynniki filtra). Te możliwości są zapewniane przez właściwości rekonfiguracji i częściowej rekonfiguracji FPGA.

Architektura FPGA daje użytkownikowi duże możliwości przy wyborze struktury systemu DSP. FPGA umożliwia zbudowanie struktury równoległej lub w pełni

szeregowej, a także struktury hybrydowej, równoległo-szeregowej. Dlatego też architektury FPGA i oparte na nich narzędzia projektowe zapewniają projektantom systemów DSP elastyczność w implementacji projektów, znajdując kompromis między zajmowaną powierzchnią (koszt) a szybkością działania projektu (wydajność).

Efektywne wykorzystanie architektury FPGA do implementacji funkcji DSP jest możliwe dzięki zaawansowanym narzędziom projektowym FPGA (Quartus, Vivado), które udostępniają bloki IP do realizacji określonych zadań DSP (np. synteza filtrów FIR lub CIC). Ponadto, osobno są dostarczane specjalne pakiety do rozwiązywania zadań z obszaru DSP. Na przykład, w systemie Quartus takim pakietem jest DSP Builder, a w systemie Vivado jest to System Generator for DSP. Dostępnych jest również wiele bloków IP DSP innych producentów.

Wiele systemów (lub środowisk) do rozwiązywania problemów matematycznych z dziedziny DSP obsługuje również implementację systemów DSP w układach FPGA. Przykładem takiego systemu jest MATLAB.

Ostatnio architektury FPGA ewoluowały w kierunku jeszcze wydajniejszej implementacji systemów DSP. Na przykład rodzina Intel Agilex obsługuje nowe funkcje do implementacji systemów DSP, takie jak:

- arytmetyka liczb całkowitych i zmiennoprzecinkowych, gdzie w szczególności do zadań związanych z arytmetyką zmiennoprzecinkową przeznaczono:
 - bank rejestrów wejściowych;
 - rejestry potokowe;
 - układy mnożące dla arytmetyki zmiennoprzecinkowej;
 - sumatory-kalkulatory dla arytmetyki zmiennoprzecinkowej;
 - bank rejestrów wyjściowych;
 - obsługę wyjątków;
- układy mnożące dla liczb zespolonych;
- tryb kaskadowego (systolic) filtra FIR.

10.2. Filtry cyfrowe

Filtracja cyfrowa oznacza eliminację (tłumienie) niektórych częstotliwości w widmie sygnału i transmisję innych częstotliwości. Celem filtracji może być:

- poprawa jakości sygnału (na przykład eliminacja lub redukcja szumów);
- ekstrakcja informacji z mieszaniny sygnałów (użytecznego sygnału);
- separacja wcześniej połączonych sygnałów (na przykład oddzielenie mowy z sygnału modulowanego).

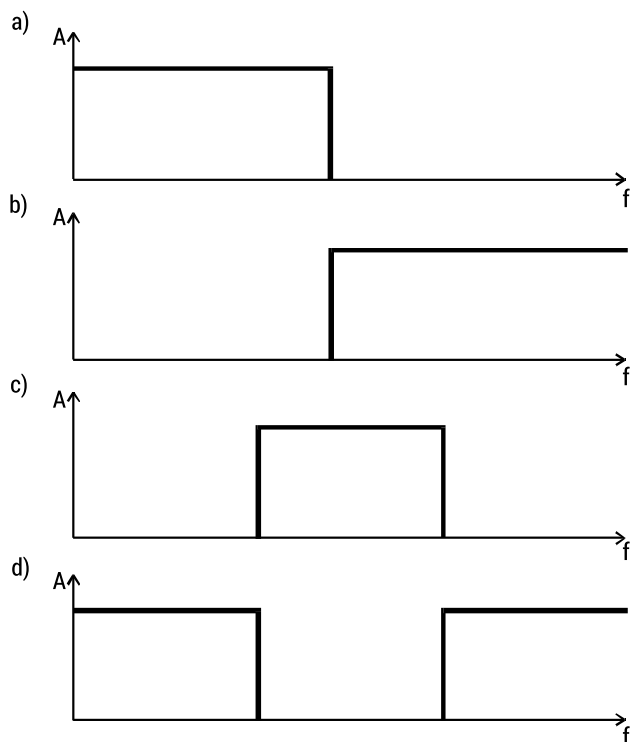
10.2.1. Klasyfikacja filtrów cyfrowych według częstotliwości przepustowej

Filtr cyfrowy to urządzenie elektroniczne, które przesyła sygnały w określonym zakresie częstotliwości i nie przepuszcza (tłumi) sygnałów o innych częstotliwościach. Obszar częstotliwości, w którym przechodzą sygnały, nazywany jest *pasmem przepustowym* (bandpass), a obszar częstotliwości, w którym sygnały są tłumione, nazywany jest *pasmem tłumienia* (bandstop). W filtrze cyfrowym może występować kilka pasm przepustowych i pasm tłumienia.

W rzeczywistości filtr cyfrowy przepuszcza sygnały w paśmie przepustowym z niewielkimi zniekształceniami i niecałkowicie przepuszcza sygnały w paśmie tłumienia, tzn. sygnały w paśmie tłumienia nadal przechodzą, ale mają znacznie zmniejszoną amplitudę.

W zależności od przepustowości filtry dzielą się na (rys. 10.2):

- filtry dolnoprzepustowe (lowpass filter – LPF);
- filtry górnoprzepustowe (highpass filter – HPF);
- filtry pasmowoprzepustowe (bandpass filter);
- filtry pasmowozaporowe (bandstop filter).



RYC. 10.2. Klasyfikacja filtra według częstotliwości przepustowości: a – dolnoprzepustowy; b – górnoprzepustowy; c – pasmowoprzepustowy; d – pasmowozaporowy (wycinający)

10.2.2. Filtry FIR i IIR

Inną ważną klasyfikacją filtrów cyfrowych jest podział ze względu na *odpowiedź impulsową filtra* (impulse response). Odpowiedź impulsowa filtra cyfrowego jest odpowiedzią na pojedynczy impuls przy zerowych warunkach początkowych. Według tej właściwości, filtry cyfrowe są podzielone na dwie duże klasy:

- filtry ze *skończoną odpowiedzią impulsową* (finite impulse response – FIR);
- filtry z *nieskończoną odpowiedzią impulsową* (infinite impulse response – IIR).

Innymi słowy, jeśli na wejście filtra FIR zostanie przyłożony pojedynczy impuls, to po skończonym czasie jego wartość wyjściowa będzie wynosić zero. Tego samego nie można powiedzieć o filtrze IIR: pojedynczy impuls na wejściu filtra IIR generuje nieskończoną sekwencję impulsów na jego wyjściu.

Zachowanie filtra IIR można opisać następującym równaniem:

$$y(n) = \sum_{k=0}^{\infty} h(k)x(n-k) = \sum_{k=0}^N b_k x(n-k) - \sum_{k=1}^M a_k y(n-k) \quad (10.1)$$

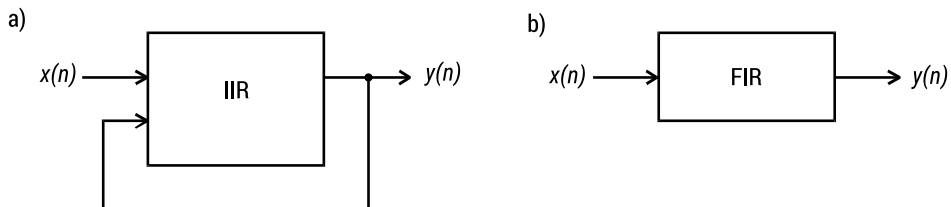
W podobny sposób zachowanie filtra FIR można opisać równaniem:

$$y(n) = \sum_{k=0}^{N-1} h(k)x(n-k) = \sum_{k=0}^{N-1} b_k x(n-k) \quad (10.2)$$

gdzie $h(k)$ jest odpowiedzią impulsową filtra; a_k i b_k to współczynniki filtra.

W filtrach FIR odpowiedź impulsowa $h(k)$ jest taka sama jak współczynniki filtra b_k .

Rzęd filtra FIR (filter order) to liczba N we wzorze (10.2), a rzęd filtra IIR to liczba M we wzorze (10.1). Z równań (10.1) i (10.2) wynika, że filtr IIR ma sprzężenia zwrotne, podczas gdy filtr FIR nie ma sprzężeń zwrotnych (rys. 10.3).



RYC. 10.3. Uogólniona struktura filtra IIR (a) i filtra FIR (b)

Ponieważ wartość wyjściowa $y(n)$ filtrów IIR zależy od poprzednich wartości wyjściowych (10.1), filtry IIR nazywane są *filtrami rekursywnymi*, a filtry FIR nazywane są *filtrami nierekursywnymi*.

Zachowanie filtrów można również wyrazić za pomocą *funkcji transmitancji* (transfer function). Funkcja transmitancji dla filtra FIR jest następująca:

$$H(z) = \sum_{k=0}^{N-1} h(k)z^{-k} \quad (10.3)$$

a funkcja transmitancji dla filtra IIR jest przedstawiona w następujący sposób:

$$H(z) = \frac{\sum_{k=0}^N b_k z^{-k}}{1 + \sum_{k=1}^M a_k z^{-k}} \quad (10.4)$$

Reprezentacje funkcji transmitancji filtrów są przydatne do szacowania odpowiedzi częstotliwościowych filtrów i są często używane do definiowania specyfikacji filtrów.

10.2.3. Wybór między filtrami IIR i FIR

Projektując filtry cyfrowe inżynier często zastanawia się, jaki typ filtra wybrać do projektu: IIR czy FIR. Zanim odpowiemy na to pytanie, należy zapoznać się z funkcjami filtrów IIR i FIR. Filtry IIR i FIR mają następujące cechy [1]:

1. Filtry FIR mają liniową odpowiedź fazową, więc nie zniekształcają sygnału wyjściowego. Odpowiedź fazowa filtrów IIR jest nieliniowa, zwłaszcza na krawędziach pasma.
2. Filtry FIR są zawsze stabilne. Ponieważ filtry IIR mają sprzężenie zwrotne (rys. 10.3a), mogą być niestabilne. Inżynier musi zagwarantować stabilność filtrów IIR.
3. Błędy (szumy) zaokrąglania i kwantyzacji są mniej istotne dla filtrów FIR niż dla filtrów IIR.
4. Do budowy filtrów wysokiej jakości (ze stromymi zboczami charakterystyki) filtry FIR wymagają więcej współczynników niż filtry IIR.
5. Istnieją analogowe realizacje filtrów IIR, w przeciwieństwie do filtrów FIR, których analogowe implementacje nie istnieją.
6. Filtry IIR mogą być ograniczone w użyciu ze względu na ich rekurencyjność.

Wybierając pomiędzy filtrami FIR i IIR, można kierować się następującymi zasadami:

- jeśli głównym wymaganiem jest stroma charakterystyka częstotliwości granicznej i szerokie pasmo, wówczas należy zastosować filtr IIR;
- jeśli głównym wymaganiem jest brak zniekształceń fazy lub minimalne zniekształcenia, należy zastosować filtr FIR.

W praktyce większość inżynierów wybiera filtry FIR. Dlatego też architektury FPGA są bardziej skoncentrowane na implementacji filtrów FIR. W następnych rozdziałach będziemy głównie rozważać projektowanie filtrów FIR.

10.2.4. Typy filtrów FIR

Filtry FIR z liniową odpowiedzią fazową mają właściwość symetrii odpowiedzi impulsowej $h(n)$. Istnieją dwa rodzaje symetrii odpowiedzi impulsowej $h(n)$:

- dodatnia, gdy $h(n) = h(N - 1 - n)$;
- ujemna, gdy $h(n) = -h(N - 1 - n)$.

W zależności od parzystości liczby współczynników N i rodzaju symetrii odpowiedzi impulsowej filtry FIR dzielą się na 4 typy (tab. 10.1).

TAB. 10.1. Typy filtrów FIR

Symetria odpowiedzi impulsowej $h(n)$	N	Rodzaj filtra FIR
Dodatnia	Nieparzysta	Typ 1
Dodatnia	Parzysta	Typ 2
Ujemna	Nieparzysta	Typ 3
Ujemna	Parzysta	Typ 4

Według [1] najbardziej uniwersalne są filtry typu 1. Filtry typu 2 nie są stosowane jako filtry górnoprzepustowe (HPF). Filtry typu 3 i 4 nie mogą być używane jako filtry dolnoprzepustowe (LPF). Ponadto filtrów typu 3 nie można używać jako filtrów górnoprzepustowych. Dlatego filtry 3 i 4 są najczęściej używane jako układy różniczkujące i układy do realizacji transformaty Hilberta.

10.3. Opis filtrów cyfrowych w języku Verilog

Filtr FIR można opisać w języku Verilog bezpośrednio jego równaniem (10.2). Stałe współczynniki filtra są opisane jako parametry. Jeżeli w trakcie pracy układu DSP przewidywana jest zmiana wartości współczynników, to w odpowiednim module należy przewidzieć oddzielny port do wprowadzania współczynników filtra.

Przy określaniu rozmiaru słowa wyjściowego należy mieć na uwadze, że jego rozmiar jest równy sumie długości próbek wejściowych (szerokość magistrali wejściowej) plus rozmiar słów przydzielonych dla współczynników filtra. Ponadto należy dodać pewną liczbę bitów, aby ewentualnie zwiększyć rozmiar wyniku ze względu na wykonywaną operację dodawania.

Poniższy przykład opisuje filtr dolnoprzepustowy FIR rzędu 8 ($N = 8$), który ma nieparzystą liczbę dodatnich współczynników symetrycznych (filtr FIR typu 1) b_k , $k = [0, ..., N]$, których wartości odpowiadają rozkładowi Gaussa. Szerokość $x(n)$ magistrali wejściowej wynosi 8 bitów, a 8 bitów jest również przydzielonych do reprezentowania współczynników. Rozmiar magistrali wyjściowej jest równy rozmiarowi wyniku mnożenia (8 + 8 bitów) plus 2 bity w przypadku przepełnienia ze względu na operację dodawania. Częstotliwość próbkowania na wejściu jest określana przez zegar *clk*. Kod Verilog dla rozważanego filtra FIR może wyglądać następująco:

```

module FIR_lowpass
#(parameter ORDER=8,                // rząd filtra
WORD_SIZE_IN=8,                    // rozmiar wejścia
WORD_SIZE_OUT=2*WORD_SIZE_IN+2,    // rozmiar wyjścia
b0=8'd7,                            // współczynniki filtra
b1=8'd17,
b2=8'd32,
b3=8'd46,
b4=8'd52,
b5=8'd46,
b6=8'd32,
b7=8'd17,
b8=8'd7)

(input clk, reset,                  // sygnały sterujące
input [WORD_SIZE_IN-1:0] data_in,  // wejście filtra
output [WORD_SIZE_OUT-1:0] data_out); // wyjście filtra

reg [WORD_SIZE_IN-1:0] x[1:ORDER]; // zmienne filtra
integer k;

// definiowanie wartości zmiennych filtra
always @(posedge clk)
    if(reset)
        for(k=1; k<=ORDER; k=k+1) x[k] <= 0;
    else begin
        x[1] <= data_in;
        for(k=2; k<=ORDER; k=k+1) x[k] <= x[k-1];
    end

// obliczenie wartości wyjściowej
assign data_out=

```

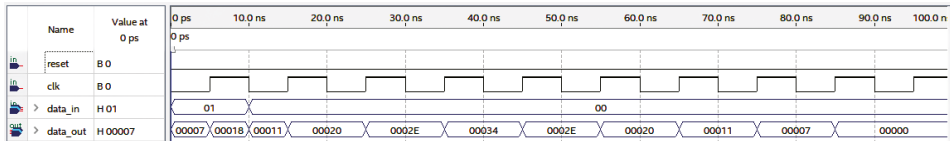
```

        b0*data_in+
        b1*x[1]+b2*x[2]+b3*x[3]+b4*x[4]+
b5*x[5]+b6*x[6]+b7*x[7]+b8*x[8];
endmodule

```

Zsyntetyzowany schemat filtra FIR zawiera 14 sumatorów i 3 układy mnożące.

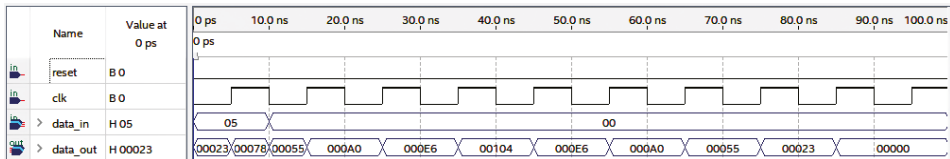
Wyniki symulacji dla przypadku, gdy pojedynczy impuls o amplitudzie równej 1 dociera na wejście filtra, pokazano na rysunku 10.4.



RYŚ. 10.4. Wyniki symulacji projektu *FIR_lowpass*, w przypadku gdy na wejście dociera pojedynczy impuls o amplitudzie 1

Na rysunku 10.4 widać, że po 9 cyklach taktowania ($N + 1$) wyjście filtra jest zerowane, tzn. zsyntetyzowany filtr jest w pełni zgodny z definicją filtra o skończonej odpowiedzi impulsowej.

Może pojawić się pytanie: jak wartość amplitudy pojedynczego impulsu na wejściu filtra FIR wpływa na czas trwania wartości niezerowych na wyjściu filtra FIR? Na rysunku 10.5 przedstawiono wyniki symulacji funkcjonalnej naszego filtra FIR w przypadku pojedynczego impulsu o amplitudzie 5 przyłożonego na wejście.



RYŚ. 10.5. Wyniki symulacji funkcjonalnego projektu *FIR_lowpass*, gdy na wejście dociera pojedynczy impuls o amplitudzie 5

Analiza rysunku 10.5 pokazuje, że wartość amplitudy pojedynczego impulsu nie ma znaczenia: po 9 cyklach sygnału zegara na wyjściu filtra powstaje stała wartość zerowa. Jednak wartość amplitudy pojedynczego impulsu wpływa na wartości wyjściowe generowane przez filtr.

Filtr IIR można również opisać w języku Verilog bezpośrednio, za pomocą równania (10.1). W naszym przykładzie $M = N = 8$. Współczynniki filtra a_k i b_k w kodzie Verilog są opisane jako parametry. Aby zsumować wartości *współczynników przewidywania* (coefficient feedforward) i *współczynników sprzężenia zwrotnego* (coefficient feedback) filtra IIR, wprowadza się dwie zmienne typu **wire**: *data_feedforward* i *data_feedback*. W pozostałej części kod filtra IIR jest podobny do kodu filtra FIR i wygląda następująco:

```

module IIR_lowpass
#(parameter ORDER=8,                                // rząd filtra
WORD_SIZE_IN=8,                                     // rozmiar wejścia
WORD_SIZE_OUT=2*WORD_SIZE_IN+2,                     // rozmiar wyjścia
b0=8'd7,b1=0,b2=0,b3=0,b4=0,b5=0,b6=0,b7=0,b8=0, // współczynniki b
a1=8'd46,a2=8'd32,a3=8'd17,a4=8'd0,                 // współczynniki a
a5=8'd17,a6=8'd32,a7=8'd46,a8=8'd52)

(input clk, reset,                                // sygnały sterujące
input [WORD_SIZE_IN-1:0] data_in,                 // wejście filtra
output [WORD_SIZE_OUT-1:0] data_out);              // wyjście filtra

reg [WORD_SIZE_IN-1:0] x[1:ORDER];                // zmienne x filtra
reg [WORD_SIZE_OUT-1:0] y[1:ORDER];               // zmienne y filtra

wire [WORD_SIZE_OUT-1:0] data_feedforward;        // wartość przewidywania
wire [WORD_SIZE_OUT-1:0] data_feedback;           // wartość sprzężenia zwrotnego

integer k;

// definiowanie wartości zmiennych filtra
always @(posedge clk)
    if(reset)
        for(k=1; k<=ORDER; k=k+1) begin
            x[k] <= 0;
            y[k] <= 0;
        end
    else begin
        x[1] <= data_in;
        y[1] <= data_out;
        for(k=2; k<=ORDER; k=k+1) begin
            x[k] <= x[k-1];
            y[k] <= y[k-1];
        end
    end

// obliczanie wartości przewidywania
assign data_feedforward=
    b0*data_in+
    b1*x[1]+b2*x[2]+b3*x[3]+b4*x[4]+

```

```
b5*x[5]+b6*x[6]+b7*x[7]+b8*x[8];
```

```
// obliczenie wartości sprzężenia zwrotnego
```

```
assign data_feedback=
```

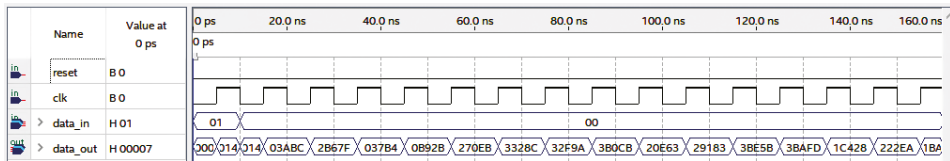
```
                a1*y[1]+a2*y[2]+a3*y[3]+a4*y[4]+  
a5*y[5]+a6*y[6]+a7*y[7]+a8*y[8];
```

```
// obliczenie wartości wyjściowej
```

```
assign data_out=data_feedforward+data_feedback;
```

```
endmodule
```

Zsyntetyzowany schemat filtra IIR dla naszego przykładu zawiera 11 sumatorów i 3 układy mnożące. Wyniki symulacji filtra IIR w sytuacji, gdy na wejście dociera pojedynczy impuls o amplitudzie 1, przedstawiono na rysunku 10.6.



RYS. 10.6. Wyniki symulacji projektu *IIR_lowpass*, gdy na wejście dociera pojedynczy impuls o amplitudzie 1

Rys. 10.6 pokazuje, że pojedynczy impuls na wejściu filtra IIR nie prowadzi do zerowych wartości na wyjściu filtra.

10.4. Metodyka projektowania filtrów cyfrowych na FPGA

Projektowanie filtrów cyfrowych na FPGA obejmuje sekwencyjne wykonanie następujących etapów:

- zdefiniowanie specyfikacji filtra;
- obliczanie współczynników filtra;
- definicja struktury filtra;
- analiza wpływu skończonej długości słowa na binarną reprezentację próbek wejściowych i współczynniki filtra;
- symulacja filtra;
- implementacja filtra.

Rozpatrzmy szczegółowo każdy z tych etapów.

10.4.1. Definiowanie specyfikacji filtrów

Zwykle specyfikacja filtra definiuje parametry, z którymi są powiązane:

- charakterystyka sygnałów: typ źródła i odbiornika sygnału, interfejs wejścia-wyjścia, szybkość transmisji danych, szerokość pasma przepustowości i szerokość pasma tłumienia, najwyższa częstotliwość;
- charakterystyka filtra: pożądana charakterystyka amplitudowa i fazowa, możliwe odchylenia pasma przepustowego i tłumienia, rzeczywisty lub nie, czas pracy filtra;
- metoda implementacji filtra: FPGA, procesor DSP, mikrokontroler;
- inne wymagania: koszt, wymiary, czas opracowania.

Inżynier ustala specyfikacje filtrów we współpracy z klientem i nie zawsze jest w stanie dokładnie określić wszystkie wymienione parametry. Należy jednak dążyć do sformułowania jak największej liczby wymagań, aby nie określać ich na kolejnych etapach projektowania.

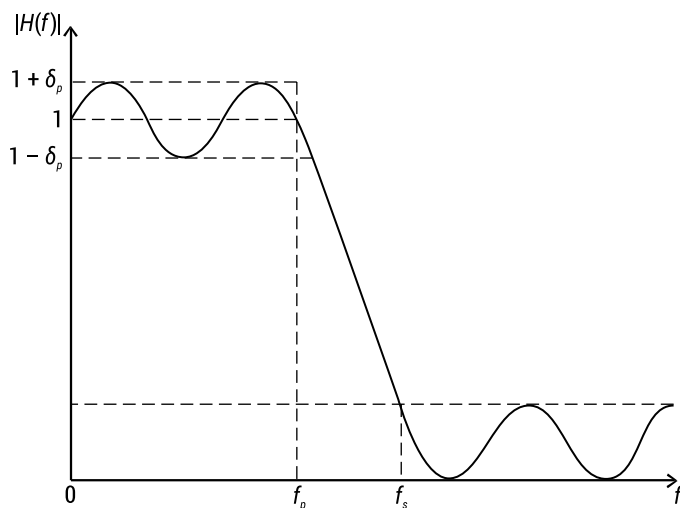
Parametry odpowiedzi *amplitudo-częstotliwościowej filtra* (filter amplitude-frequency response) są często przedstawiane jako wykres tolerancji. Przykład takiego wykresu dla filtra dolnoprzepustowego pokazano na rysunku 10.7, gdzie:

δ_p – odchylenie (dewiacja) w paśmie przepustowości (deviation in bandwidth);

δ_s – odchylenie (dewiacja) w paśmie tłumienia (deviation in bandstop);

f_p – częstotliwość graniczna pasma przepustowości (bandwidth cut-off frequency);

f_s – częstotliwość graniczna pasma tłumienia (bandstop cut-off frequency).



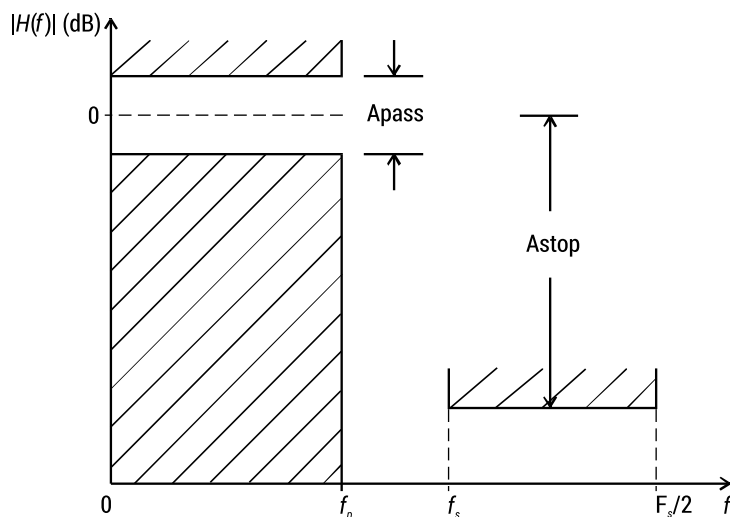
RYS. 10.7. Wykres tolerancji dla filtra dolnoprzepustowego

Dewiacje w paśmie przepustowości i w paśmie tłumienia można wyrazić jako liczby w skali liniowej (odpowiednio δ_p i δ_s) lub logarytmicznej (w decybelach). W tym drugim przypadku wyrażają one *niejednorodność* w paśmie przepustowości

(non-uniformity in bandwidth) A_p i tłumienność w paśmie zaporowym (attenuation in bandstop) A_s (rys. 10.8). Parametry te mają następującą zależność:

$$A_p = 20 \lg (1 + \delta_p) \quad (10.5)$$

$$A_s = -20 \lg (\delta_s) \quad (10.6)$$



RYŚ. 10.8. Alternatywne przedstawienie parametrów filtra dolnoprzepustowego

Graniczne częstotliwości f_p i f_s można również przedstawić w standardowych jednostkach częstotliwości (Hz, kHz, MHz) lub w formie znormalizowanej. W tym drugim przypadku wartości f_p i f_s są podzielone przez częstotliwość próbkowania (sampling frequency) F_s .

10.4.2. Obliczanie współczynników filtra

Do tej pory opracowano wiele różnych matematycznych metod obliczania współczynników filtra i liczba tych metod stale rośnie. Każda metoda ma swoją własną charakterystykę. Interesują nas tylko te metody, które nadają się do praktycznego zastosowania przy realizacji filtrów cyfrowych. Zwykle najbardziej wydajne metody obliczania współczynników filtra są implementowane w narzędziach programowych, na przykład MATLAB, Core Generator, TFilter, WinFilter, libdspl, TIBQ, miniDSP, dspGuru itp.

Do obliczania współczynników filtrów FIR najczęściej używane są metody optymalizacji (optimization methods), metoda ważenia (weighing method) i metoda próbkowania częstotliwości (frequency sampling method). Przy obliczaniu współczynników filtrów IIR najczęściej stosowaną metodą jest transformacja niezmienna odpowiedzi

impulsowej (invariant transformation of impulse response), *metoda transformacji dwuliniowej* (two-line transformation method) oraz *metoda rozmieszczania zer i biegunów* (method of placing zeros and poles).

Inżynier nie musi samodzielnie obliczać współczynników filtra. Aby to zrobić, wystarczy użyć odpowiedniego narzędzia programowego. Jednak w celu prawidłowego i skutecznego korzystania z takiego narzędzia inżynier musi znać zastosowaną metodę. Z wymienionymi metodami można zapoznać się w [1].

10.4.3. Definiowanie struktury filtra

Na tym etapie konieczne jest określenie najbardziej odpowiedniej struktury do realizacji filtra.

Tradycyjnie, do implementacji filtrów IIR używane są *struktury bezpośrednie* (direct structures), *struktury kaskadowe* (cascading structures) i *struktury równoległe* (parallel structures). Struktura bezpośrednia jest bezpośrednią implementacją filtra IIR zgodnie z równaniem (10.1). W przypadku stosowania struktury kaskadowej funkcja transmitancji filtra IIR $H(z)$ jest rozkładana na czynniki i wyrażana jako iloczyn połączeń filtrów drugiego rzędu. W strukturze równoległej funkcja przeniesienia $H(z)$ jest reprezentowana jako suma połączeń filtrów drugiego rzędu. Na bezpośrednią strukturę filtrów IIR największy wpływ mają skutki skończonej długości słowa. Dlatego w praktyce projektanci starają się unikać stosowania struktury bezpośredniej przy implementacji filtrów IIR.

Natomiast w przypadku filtrów FIR najczęściej stosowaną, ze względu na swoją prostotę, jest struktura bezpośrednia, która odpowiada równaniom (10.2). Bezpośrednia struktura filtra FIR nazywana jest również *linią opóźniającą z odczepami* (tapped delay line) lub *filtrem poprzecznym* (transversal filter). W praktyce inżynierskiej filtry FIR są często implementowane przy użyciu układu próbkowania i układu szybkiego splotu.

Istnieją również znane struktury, które są popularne w niektórych zastosowaniach filtrów. Na przykład struktura sieciowa jest szeroko stosowana w aplikacjach przetwarzania mowy.

Na ostateczny wybór konstrukcji filtra wpływają takie czynniki jak:

- typ filtra: FIR lub IIR;
- łatwość realizacji;
- wrażliwość struktury na skutki skończonej długości słowa.

10.4.4. Analiza wpływu skończonej długości słowa

Teoretycznie równania (10.1) i (10.2), które opisują działanie filtrów, przedstawiają wszystkie wielkości z bardzo dużą dokładnością. Jednak implementacja filtra na FPGA lub procesorze wymaga użycia słów o skończonej liczbie bitów. Projektując filtr inżynier musi przeanalizować wpływ skończonych długości słów i wybrać odpowiednią

długość słowa, aby przedstawić próbki wejściowe, współczynniki filtra, wyniki mnożenia i dodawania. Wszystko jest jasne, jeśli chodzi o długość wyniku mnożenia dwóch liczb $x*y$, jest ona równa sumie długości każdej liczby $N + M$, gdzie N jest długością słowa x , a M jest długością słowa y , ale nie wszystko jest oczywiste w przypadku wyników sumowania, ponieważ kolejnych słów składowych sumy może być dość dużo i należy przewidzieć dodatkowe bity na ewentualne przepełnienie.

Skończone długości słów mogą obniżyć wydajność filtra cyfrowego z powodu:

- *kwantyzacji sygnałów* (signal quantization) na wejściu i wyjściu, np. dowolny przetwornik analogowo-cyfrowy (ADC) generuje *szum ADC* (ADC noise), który wpływa na całą ścieżkę transmisji danych w filtrze cyfrowym;
- *kwantyzacji współczynników* (coefficient quantization) filtrów – czynnik ten prowadzi do zniekształcenia charakterystyki częstotliwościowej filtra i może prowadzić do niestabilności filtrów IIR;
- *błędy zaokrągleń* (rounding errors) – współczynnik ten generuje szum zaokrągleń i ma taki sam wpływ jak kwantyzacja współczynników; efekt błędów zaokrągleń można nieco zmniejszyć, wybierając odpowiednią metodę zaokrąglania;
- *przepełnienie* (overflow) – przepełnienie występuje w wyniku wielokrotnego dodawania i niewystarczającej długości słowa dla wyniku; przepełnienie prowadzi do nieprawidłowych wartości wyjściowych.

Z powyższego możemy wywnioskować, że inżynier powinien bardzo uważać przy doborze długości słów reprezentujących próbki wejściowe i wyjściowe, a także współczynników filtra.

Stopień wpływu skończonej długości słowa zależy od następujących czynników:

- długości słowa;
- rodzaju arytmetyki: zmiennoprzecinkowa lub całkowita;
- metody kwantowania próbek wejściowych i współczynników filtra;
- struktury filtra.

Opierając się na tych czynnikach, inżynier powinien pod koniec projektowania filtra ocenić wpływ skończonej długości słowa na wydajność filtra i podjąć odpowiednie działania, jeśli jest to konieczne.

10.4.5. Realizacja filtra

Po obliczeniu współczynników filtra, określeniu jego struktury i uwzględnieniu wpływu skończonej długości słowa, pojawia się pytanie o sposób realizacji filtra na określonej bazie elementów, w naszym przypadku na FPGA.

Aby zaimplementować filtr cyfrowy na FPGA, wymagane są następujące elementy obwodu:

- pamięć typu ROM (RAM – dla filtrów adaptacyjnych) do przechowywania współczynników filtrów;

- pamięć typu RAM do przechowywania wartości próbek wejściowych;
- pamięć typu RAM do przechowywania wartości próbek wyjściowych (dla filtrów IIR);
- układy mnożące (układy mnożące z akumulacją);
- sumatory.

Realizacja układu filtra w dużej mierze zależy od tego, w jakim czasie: rzeczywistym lub nie, filtr będzie działał.

Gdy filtr nie działa w czasie rzeczywistym, wszystkie dane początkowe są znane i przechowywane na dysku w postaci plików. Mogą to być wyniki badań eksperymentalnych, nagrania mowy, muzyki, zdjęć itp. Konieczne jest przefiltrowanie danych wyjściowych i uzyskanie określonego wyniku, na przykład w celu dalszej analizy. Cechą pracy w tym trybie jest to, że czas pracy filtra jest praktycznie nieograniczony. Jednak implementując filtr należy zadbać o sprawne ładowanie danych źródłowych z pamięci zewnętrznej.

Zupełnie inaczej sytuacja wygląda, gdy filtr działa w czasie rzeczywistym. W takim przypadku nakładane są bardzo surowe wymagania dotyczące czasu przetwarzania danych wejściowych. Każda próbka wejściowa musi zostać przetworzona, a wiarygodny wynik musi zostać uzyskany przed nadejściem następnej próbki wejściowej. Aby to zrobić, należy wziąć pod uwagę częstotliwość próbkowania F_s , częstotliwość sygnału zegarowego filtra i liczbę cykli zegarowych wymaganych do przetworzenia jednej próbki. Może się zdarzyć, że aby spełnić warunki filtra do działania w czasie rzeczywistym, konieczna jest zmiana rzędu filtra, długości słów, algorytmu obliczania współczynników filtra, struktury filtra, typu filtra (FIR lub IIR).

W przypadku potokowego przetwarzania danych, jak również przy zmiennej częstotliwości próbek wejściowych, do struktury filtra można dodać bufory wejściowe i wyjściowe zbudowane na blokach pamięci typu FIFO.

Wybierając układ FPGA do implementacji filtra, należy wziąć pod uwagę:

- długość słowa i ilość pamięci wewnętrznej do przechowywania współczynników filtrów, próbek wejściowych i wyjściowych (w przypadku filtra IIR);
- liczbę wbudowanych bloków DSP i liczbę układów mnożących w każdym bloku;
- ilość zasobów logicznych FPGA;
- możliwość konfiguracji bloków pamięci wbudowanej do implementacji pamięci typu FIFO i rejestrów przesuwających.

10.5. Projektowanie filtrów cyfrowych w systemie MATLAB

W tej części rozpatrzymy proces tworzenia filtra cyfrowego w systemie MATLAB na przykładzie projektowania filtra dolnoprzepustowego o następującej specyfikacji:

- częstotliwość próbkowania $F_s = 5100$ Hz;
- częstotliwość graniczna pasma przepustowego $f_p = 530$ Hz;

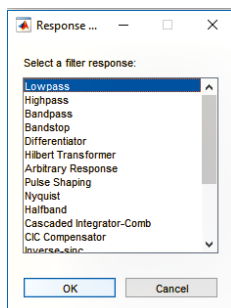
- częstotliwość graniczna pasma tłumienia $f_s = 785$ Hz;
- nierównomierność w paśmie przepustowym $A_p = 1$ dB;
- tłumienność w paśmie tłumienia $A_s = 60$ dB.

System MATLAB udostępnia kilka programów, za pomocą których można zaprojektować filtr cyfrowy. Użyjemy programu do tworzenia filtrów, który zapewnia zintegrowane środowisko do projektowania filtra cyfrowego, a także generuje kod w języku opisu sprzętu (hardware description language – HDL) w celu realizacji filtra na FPGA.

Przed uruchomieniem MATLAB-a stwórz na dysku katalog, w którym będą znajdować się pliki projektu naszego filtra. W systemie MATLAB wpisz:

>>filterbuilder

W odpowiedzi pojawi się okno (rys. 10.9), w którym należy wybrać z listy typ filtra.



RYŚ. 10.9. Wybór filtra dolnoprzepustowego do projektowania w MATLAB

Wybierz **Lowpass**, kliknij **OK**. Pojawi się okno **Lowpass Design** Projektanta filtra dolnoprzepustowego (rys. 10.10).

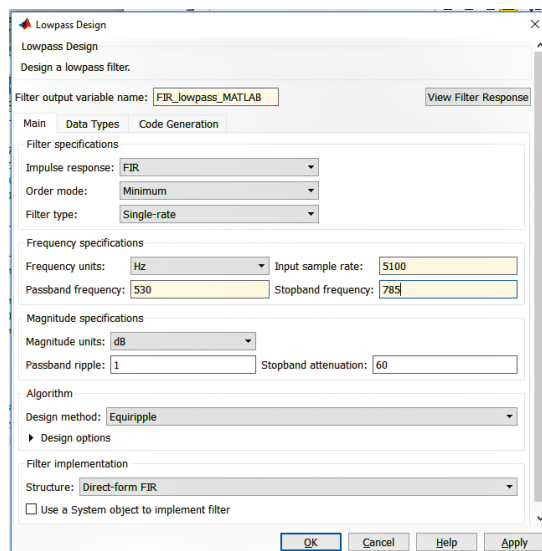
W polu **Filter output variable name** (Nazwa zmiennej wyjściowej filtra) określ nazwę zmiennej MATLAB, która będzie odpowiadać rezultatowi naszego filtra, na przykład **FIR_lowpass_MATLAB**.

Okno **Lowpass Design** (Projekt dolnoprzepustowy) ma trzy zakładki: **Main** (Główne), **Data Types** (Typy danych) i **Code Generation** (Generowanie kodu).

Na zakładce **Main** można określić specyfikację filtra, specyfikację częstotliwości, specyfikację amplitudy, algorytm obliczania współczynników filtra oraz sposób implementacji filtra.

Pole **Filter specification** (Specyfikacja filtra) zawiera:

- rodzaj odpowiedzi impulsowej filtra (**Impulse response**), **FIR** lub **IIR**;
- rząd filtra (**Order mode**), **Minimum** (Minimalny) lub **Specify** (Określ); jeśli wybrano **Specify**, pojawi się okno, w którym należy wpisać wartość rzędu filtra;
- typ filtra (**Filter type**), **Single-rate**, **Decimator**, **Interpolator** lub **Sample-rate converter**.



RYS. 10.10. Okno **Lowpass Design** Projektanta filtra dolnoprzepustowego

W naszym przykładzie pozostawiamy wszystkie wartości parametrów w polu **Filter specification** niezmienione (**FIR, Minimum, Single-rate**).

Pole **Frequency specification** (Specyfikacja częstotliwości) zawiera:

- jednostki częstotliwości, **Hz, kHz, MHz, GHz** lub znormalizowane (**Normalized (0 to 1)**);
- częstotliwość próbkowania (**Input sample rate**);
- częstotliwość pasma przepustowego (**Passband frequency**);
- częstotliwość pasma tłumienia (**Stopband frequency**).

Dla naszego filtra częstotliwość próbkowania wynosi 5100 Hz, częstotliwość graniczna jest ustawiona na 530 Hz, a częstotliwość tłumienia jest ustawiona na 785 Hz.

Pole **Magnitude specification** (specyfikacja amplitudy) zawiera:

- jednostki miary amplitudy (**Magnitude units**), **dB** (w decybelach) lub **Linear** (liniowy);
- poziom nierównomierności w paśmie przepuszczania (**Passband ripple** – tętnienie w paśmie przepustowym);
- poziom tłumienności w paśmie zaporowym (**Stopband ripple**).

W naszym przykładzie nierównomierność w paśmie przepuszczania jest ustawiona na 1 dB, a poziom tłumienności w paśmie zaporowym wynosi 60 dB.

W polu **Algorithm** można wybrać metodę obliczania współczynników filtra (**Design method**). W przypadku filtra FIR dostępne są następujące metody:

- **Equiripple** – o równomiernie pulsującej charakterystyce amplitudowo-częstotliwościowej;
- **Interpolated FIR** – filtr FIR interpolowany;
- **Kaiser window** – okno Kaisera;
- **Multistage equiripple** – wielostopniowy Equiripple.

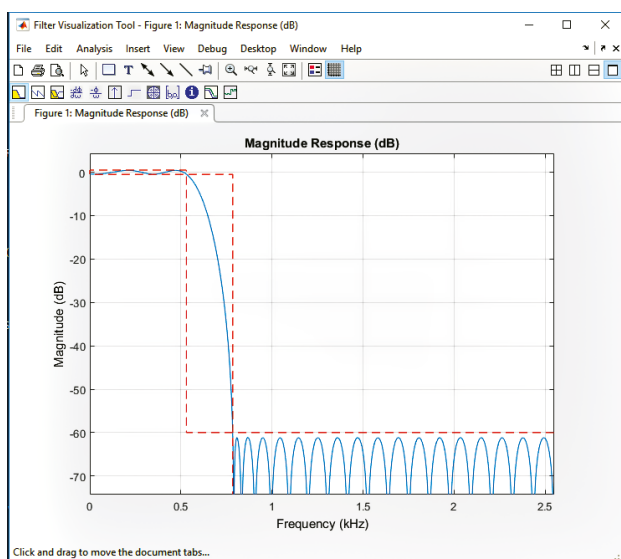
W naszym przypadku pozostawiamy wartość **Equiripple**.

W polu **Filter implementation** (implementacja filtra) struktura filtra jest wybierana z następujących możliwych struktur:

- **Direct-form FIR** – struktura bezpośrednia;
- **Direct-form FIR transposed** – transponowana struktura bezpośrednia;
- **Direct-form symmetric FIR** – symetryczna struktura bezpośrednia;
- **Overlap-add FIR** – struktura z dodatkowym nakładaniem.

Po wprowadzeniu wszystkich specyfikacji filtra kliknij przycisk **Apply** (Zastosuj). MATLAB wygeneruje wymagany filtr.

Aby wyświetlić główne parametry przyszłego filtra, wystarczy kliknąć przycisk **View Filter Response** (Wyświetl odpowiedź filtra). Pojawi się okno (rys. 10.11) narzędzia wizualizacji filtrów (**Filter Visualization Tool**), w którym można obserwować różne parametry tworzonego filtra na wykresach.



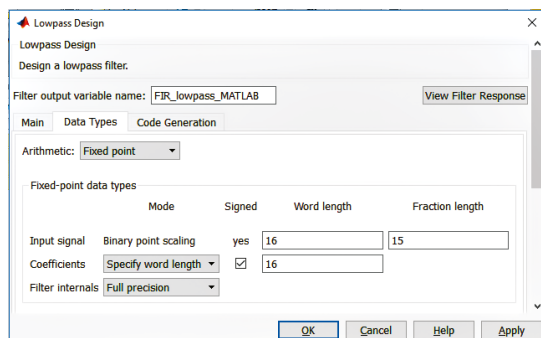
RYŚ. 10.11. Okno narzędzia wizualizacji filtra **Filter Visualization Tool**

Narzędzie wizualizacji filtrów **Filter Visualization Tool** umożliwia obserwację wizualną następujących parametrów filtra:

- Magnitude Response – odpowiedź amplitudowa;
- Phase Response – odpowiedź (charakterystyka) częstotliwościowa (fazowa);
- Magnitude and Phase Response – odpowiedź amplitudowo-częstotliwościowa;
- Group Delay Response – charakterystyka opóźnienia grupowego;
- Phase Delay – opóźnienie fazy;
- Impulse Response – odpowiedź impulsowa;
- Step Response – odpowiedź przejściowa;
- Pole/Zero Plot – wykres biegunów i zer;
- Filter Coefficient – współczynniki filtra;
- Filter Information – informacje o filtrze;
- Magnitude Response Estimate – oszacowanie charakterystyki amplitudy;
- Round-off Noise Power Spectrum – zaokrąglone widmo mocy szumu.

Jeśli niektóre parametry utworzonego filtra nie spełniają wymagań, należy wrócić do okna **Lowpass Design** i zdefiniować inną specyfikację filtra.

W oknie **Lowpass Design** zakładka **Data Types** (Typy danych) umożliwia zdefiniowanie typów danych dla sygnału wejściowego, współczynników filtra i wewnętrznych zmiennych filtra (rys. 10.12).

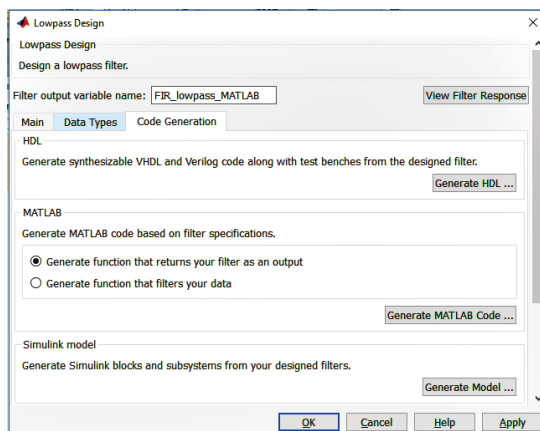


RYŚ. 10.12. Zakładka **Data Types** okna **Lowpass Design**

Pole **Arithmetic** określa format reprezentacji liczb: liczby zmiennoprzecinkowe podwójnej precyzji (**Double precision**) lub pojedynczej precyzji (**Single precision**) albo format stałoprzecinkowy (**Fixed point**).

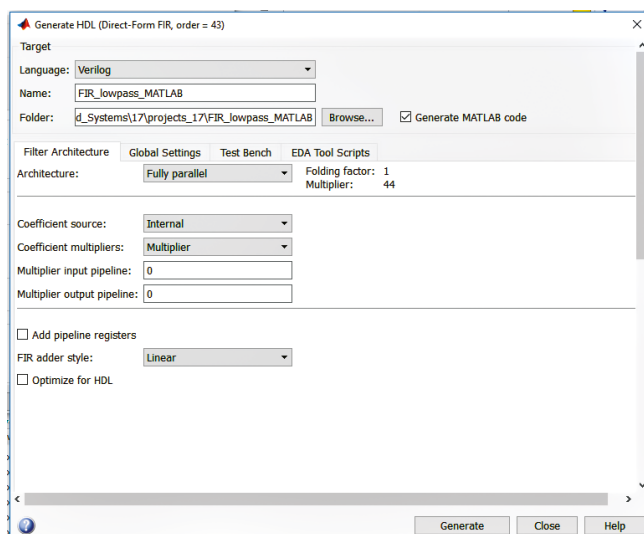
W naszym przykładzie ustawiamy format **Fixed point**. Pojawiają się pola, w których można określić format słowa do reprezentowania sygnału wejściowego (**Input signal**), współczynników (**Coefficients**) i elementów wewnętrznych filtra (**Filter internals**). Dla sygnału wejściowego długość słowa (**Word length**) i długość części ułamkowej (**Fraction length**) są określane w bitach.

Zakładka **Code Generation** (rys. 10.13) steruje generowaniem kodu w języku opisu sprzętu (HDL), w języku systemu **MATLAB** i programie do symulacji **Simulink**.



RYS. 10.13. Zakładka **Code Generation** okna **Lowpass Design**

Wybierz pole **Generate HDL...**, aby zdefiniować język opisu sprzętu, w którym ma zostać wygenerowany kod filtra. Pojawi się okno **Generate HDL** (rys. 10.14).



RYS. 10.14. Okno **Generate HDL**

W oknie **Generate HDL** należy wybrać jeden z języków: VHDL lub Verilog, podać nazwę pliku i katalog na dysku, w którym zostaną utworzone wygenerowane pliki z opisem naszego filtra. Wybieramy język Verilog, nazwa pliku to *FIR_lowpass_MATLAB* i za pomocą przycisku **Browse...** definiujemy katalog na dysku dla naszego projektu. Włączamy również opcję **Generate MATLAB code**, aby utworzyć funkcję MATLAB, która będzie zawierała wszystkie wprowadzone ustawienia.

Okno **Generate HDL** ma dodatkowo cztery zakładki: **Filter Architecture** (Architektura filtra), **Global Settings** (Ustawienia globalne), **Test Bench** (Program testowy) i **EDA Tool Scripts** (Skrypty dla narzędzi EDA).

Zakładka **Filter Architecture** definiuje:

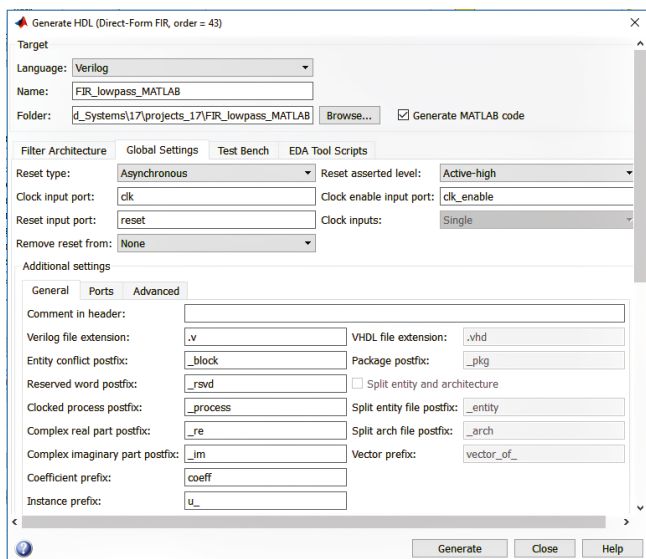
- **Coefficient source** – źródło współczynników: ustawiane w kodzie języka HDL (**Internal**) lub przekazywane poprzez interfejs z procesora systemowego (**Processor interface**);
- **Coefficient multipliers** – układy mnożące dla współczynników, można określić jako **Multiplier**, **CSD** lub **factored-CSD**;
- **Multiplier input pipeline** – liczba układów mnożących potoku wejściowego;
- **Multiplier output pipeline** – liczba układów mnożących potoku wyjściowego.

Opcja **Add pipeline registers** (Dodaj rejestry potoku) umożliwia dodawanie rejestrów potoku. Można także zdefiniować strukturę sumatora FIR (**FIR adder style**), strukturę liniową (**Linear**) lub drzewiastą (**Tree**). Oprócz tego, gdy włączona jest opcja **Optimize for HDL** (Optymalizuj dla HDL), wykonywana jest dodatkowa optymalizacja kodu filtra pod kątem szybkości i kosztu implementacji, ale jej zastosowanie może prowadzić do nieco innych wyników filtrowania. W naszym przykładzie należy pozostawić wszystkie parametry na karcie **Filter Architecture** niezmienione.

Zakładka **Global Settings** (rys. 10.15) pozwala:

- ustawić nazwy portów wejściowych synchronizacji (**Clock input port**), resetowania (**Reset input port**) i zezwolenia synchronizacji (**Clock enable input port**);
- ustawić typ resetowania (**Reset type**): asynchroniczny (**Asynchronous**) lub synchroniczny (**Synchronous**);
- usunąć sygnał resetowania (**Remove reset from**) z rejestru przesuwającego;
- określić zbocze sygnału resetowania (**Reset asserted level**): dodatnie (**Active-high**) lub ujemne (**Active-low**);
- ustalić liczbę wejść synchronizacji (**Clock inputs**).

Zakładka **Global Settings** zawiera trzy dodatkowe zakładki: **General** (Ogólne), **Ports** (Porty) i **Advanced** (Zaawansowane). Zakładka **General** umożliwia definiowanie przyrostków i przedrostków dla różnych sygnałów w kodzie HDL. Zakładka **Ports** umożliwia zdefiniowanie parametrów portu, w szczególności nazwy portu wejściowego (**Input port**) i portu wyjściowego (**Output port**) filtra, rzeczywiste lub zespolone wartości wejścia, a także określić, czy zostanie dodany rejestr wejściowy (**Add input register**) i (lub) wyjściowy (**Add output register**). W zakładce **Advanced** można zezwolić na stosowanie dyrektywy **timescale** w kodzie projektu, a także wskazać datę i czas w nagłówku kodu.



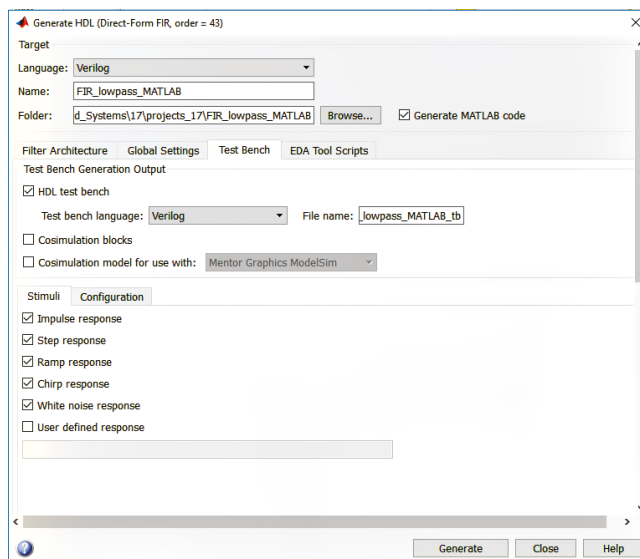
RYS. 10.15. Okno **Generate HDL** z otwartą zakładką **Global Settings**

Zakładka **Test Bench** (rys. 10.16) umożliwia wygenerowanie pliku testowego do przetestowania utworzonego filtra. W tym celu należy określić opcję **HDL test bench** i wybrać żądany język: Verilog lub VHDL, a także wprowadzić nazwę pliku dla jednostki testowej. Ponadto, korzystając z opcji **Cosimulation model for use with** (Model symulacyjny do użytku z), można określić program symulacyjny, który ma być użyty do przeprowadzenia symulacji: **Mentor Graphics ModelSim** lub **Cadence Incisive**.

Zakładka **Test Bench** ma dwie dodatkowe zakładki: **Stimuli** (Bodźce) i **Configuration** (Konfiguracja). Zakładka **Stimuli** pozwala wybrać rodzaj wpływu na wejścia filtrów:

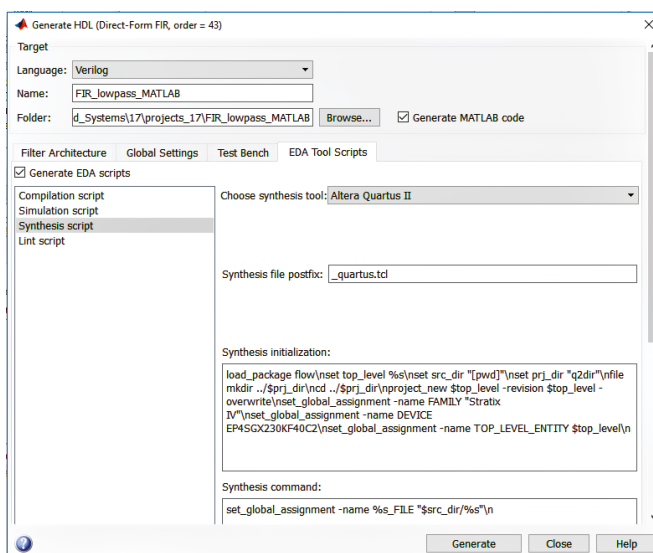
- **Impulse response** (Odpowiedź impulsowa) – impuls;
- **Step response** (Odpowiedź skokowa) – krokowa;
- **Ramp response** (Odpowiedź rampowa) – sygnał narastający;
- **Chirp response** (Odpowiedź Chirp) – sygnał z liniową zmianą modulacji częstotliwościowej;
- **White noise response** (Odpowiedź białego szumu) – biały szum gaussowski;
- **User defined response** (Odpowiedź użytkownika) – sygnał zdefiniowany przez użytkownika jako wyrażenie lub zmienna MATLAB.

Dodatkowo w zakładce **Configuration** można ustawić inny niż tradycyjny cykl pracy sygnału synchronizacji, czas ustawiania i podtrzymania, opóźnienie sygnału włączenia synchronizacji, czas trwania sygnału resetowania, opóźnienie danych wejściowych między próbkami, inicjalizację wejść modułu testowego, wieloplikowy moduł testowy, zdefiniować sufiksy dla nazwy pliku danych, pliku komunikatów dla modułu testowego i flagi symulacji.



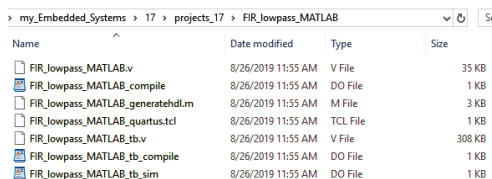
RYS. 10.16. Zakładka **Test Bench** w oknie **Generate HDL**

Zakładka **EDA Tool Scripts** (rys. 10.17) umożliwia skonfigurowanie plików skryptów do syntezy i symulacji projektu filtra w narzędziach projektowych innych firm, takich jak Quartus lub Vivado. W naszym przypadku dla parametru **Synthesis script** wybierz narzędzie syntezy **Altera Quartus II**.



RYS. 10.17. Zakładka **EDA Tool Scripts** okna **Generate HDL**

Na tym kończy się okno dialogowe do tworzenia wymaganego filtra i generowania kodu projektu w HDL, naciśnij przycisk **Generate**. We wskazanym przez nas katalogu można przeglądać wygenerowane pliki (rys. 10.18):



Name	Date modified	Type	Size
FIR_lowpass_MATLAB.v	8/26/2019 11:55 AM	V File	35 KB
FIR_lowpass_MATLAB_compile	8/26/2019 11:55 AM	DO File	1 KB
FIR_lowpass_MATLAB_generatehdl.m	8/26/2019 11:55 AM	M File	3 KB
FIR_lowpass_MATLAB_quartus.tcl	8/26/2019 11:55 AM	TCL File	1 KB
FIR_lowpass_MATLAB_tb.v	8/26/2019 11:55 AM	V File	308 KB
FIR_lowpass_MATLAB_tb_compile	8/26/2019 11:55 AM	DO File	1 KB
FIR_lowpass_MATLAB_tb_sim	8/26/2019 11:55 AM	DO File	1 KB

RYS. 10.18. Pliki projektu cyfrowego filtra, automatycznie wygenerowane przez system MATLAB

Na rysunku 10.18 pliki z rozszerzeniem .v – to kod filtra i jednostki testowej w języku Verilog dla naszego przykładu; pliki z rozszerzeniem .do – to pliki skryptów programu ModelSim do kompilacji modułu filtra i modułu do testowania, a także do wykonywania symulacji; plik z rozszerzeniem .tcl – to plik skryptowy w języku Tcl [9]; plik z rozszerzeniem .m jest plikiem systemu MATLAB.

Dalszą symulację wygenerowanego filtra można przeprowadzić za pomocą programu ModelSim.

10.6. Symulacja filtrów cyfrowych w systemie ModelSim

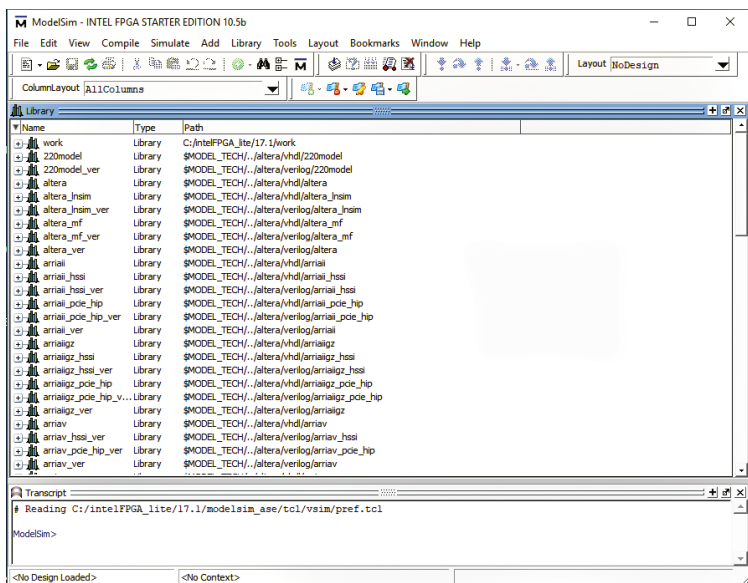
System ModelSim jest zwykle dostarczany z systemem Quartus. W naszym przypadku użyjemy ModelSim – Intel FPGA Starter Edition, który jest dołączony do Quartus Prime w wersji 18.1. Uruchamiamy program ModelSim, klikając odpowiednią ikonę na ekranie pojawi się okno, jak na rysunku 10.19.

Najpierw musimy określić katalog roboczy, w którym znajdują się pliki projektu, i utworzyć bibliotekę roboczą dla programu ModelSim. Aby określić katalog roboczy w menu głównym programu ModelSim, wybierz **File > Change Directory** i ustaw katalog z wygenerowanymi plikami filtrów ../FIR_lowpass_MATLAB. Aby utworzyć bibliotekę roboczą, wybierz **File > New > Library**, zaakceptuj określone ustawienia, kliknij **OK**. Biblioteka robocza **work** pojawi się w oknie **Library**.

Można wykonać wszystkie czynności związane z symulacją wygenerowanego filtra ręcznie lub skorzystać z plików wsadowych już utworzonych w systemie MATLAB:

- FIR_lowpass_MATLAB_compile.do – do kompilacji modułu filtra;
- FIR_lowpass_MATLAB_tb_compile.do – aby skompilować moduł do testowania filtra;
- FIR_lowpass_MATLAB_tb_sim.do – do symulacji filtra.

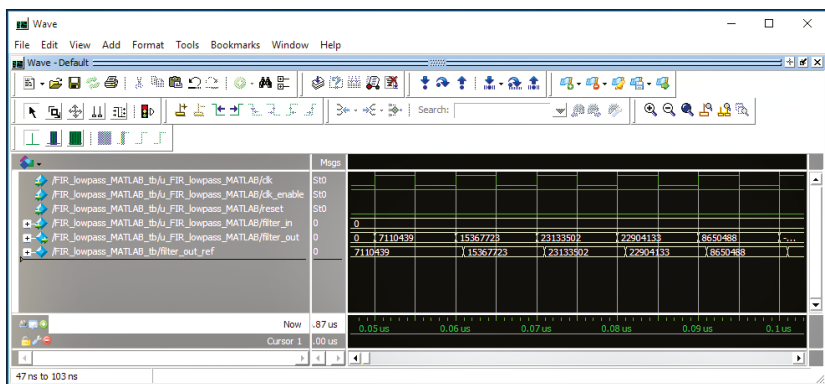
Aby skompilować moduł filtra, skopiuj zawartość pliku FIR_lowpass_MATLAB_compile.do do okna **Transcript** programu ModelSim i naciśnij **Enter**. Podobne czynności wykonujemy dla pliku FIR_lowpass_MATLAB_tb_compile.do.



RYS. 10.19. Ekran po uruchomieniu ModelSim

Aby przeprowadzić symulację projektu filtra wygenerowanego przez system MATLAB, powtórz te same kroki z plikiem `FIR_lowpass_MATLAB_tb_sim.do`. W efekcie wczytane zostaną niezbędne pliki do symulacji projektu i przeprowadzona zostanie symulacja.

Wyniki symulacji można wyświetlić graficznie w oknie **Wave** (rys. 10.20). Aby powiększyć lub zmniejszyć obraz w oknie **Wave**, użyj odpowiednich przycisków menu.



RYS. 10.20. Wyniki symulacji w programie ModelSim projektu filtra cyfrowego `FIR_lowpass_MATLAB` wygenerowane w systemie MATLAB

Na rysunku 10.20 widać, że wartości na wyjściu wygenerowanego filtra `filter_out` i wartości referencyjne `filter_out_ref` są takie same. Oznacza to, że nasz filtr działa poprawnie.

10.7. Realizacja filtrów cyfrowych w systemie Quartus

W systemie Quartus filtry cyfrowe można zaimplementować na kilka sposobów. Pierwsza z nich została omówiona w rozdziale 10.3, gdzie filtry zostały opisane w języku Verilog. W tym podejściu współczynniki filtra muszą być znane oraz muszą być reprezentowane jako liczby całkowite.

Drugi sposób implementacji filtrów cyfrowych w systemie Quartus polega na opracowaniu filtra z wykorzystaniem systemu MATLAB i wygenerowaniu kodu projektu filtra w jednym z języków opisu sprzętu: Verilog lub VHDL (patrz rozdział 10.5).

10.7.1. Realizacja w systemie Quartus filtrów cyfrowych opracowanych w systemie MATLAB

Założmy, że projekt filtra cyfrowego został wykonany w systemie MATLAB w pełnej zgodności z rozdziałem 10.5 i wygenerowano kody projektu filtra w języku Verilog i moduły do testowania filtra. Dalsza implementacja filtra na FPGA w systemie Quartus nie różni się niczym od realizacji jakiegokolwiek projektu pod względem jego opisu w języku Verilog. W tym celu wystarczy zdefiniować docelowy układ FPGA, skompilować projekt i skonfigurować układ FPGA.

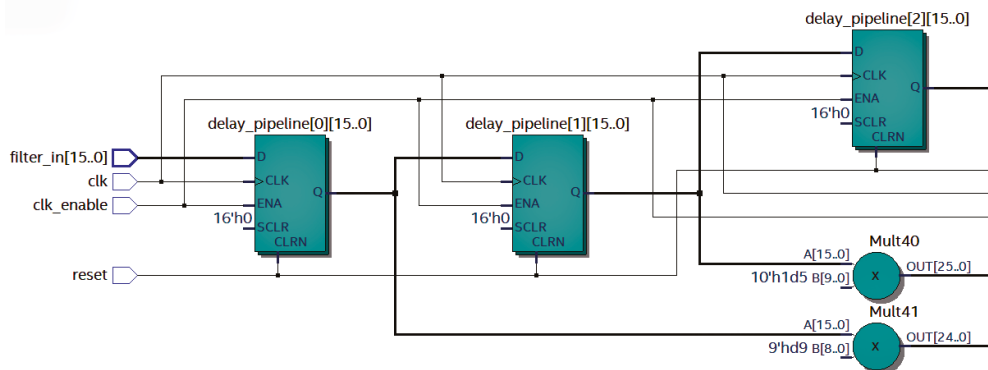
Jako przykład rozpatrzmy implementację w systemie Quartus projektu filtra *FIR_lowpass_MATLAB*, którego kod został wygenerowany w systemie MATLAB (patrz rozdział 10.5). Zdefiniujemy rodzinę Cyclone V jako docelowy układ FPGA. Ostateczne podsumowanie wyników syntezy pokazano na rysunku 10.21.

Top-level Entity Name	FIR_lowpass_MATLAB
Family	Cyclone V
Device	5CGXFC7C7F23C8
Timing Models	Final
Logic utilization (in ALMs)	206 / 56,480 (< 1 %)
Total registers	768
Total pins	53 / 268 (20 %)
Total virtual pins	0
Total block memory bits	0 / 7,024,640 (0 %)
Total DSP Blocks	41 / 156 (26 %)

RYS. 10.21. Statystyka wyników syntezy projektu *FIR_lowpass_MATLAB*

Na rysunku 10.21 widać, że aby zaimplementować nasz filtr na układach FPGA z rodziny Cyclone V, potrzebnych było 206 *adaptacyjnych modułów logicznych* (Adaptive Logic Module – ALM), 768 rejestrów i 41 wbudowanych bloków DSP.

Rysunek 10.22 przedstawia fragment widoku projektu na poziomie RTL.



RYS. 10.22. Fragment widoku projektu FIR_lowpass_MATLAB na poziomie RTL

Należy zwrócić uwagę, że nie ma potrzeby symulacji projektu *FIR_lowpass_MATLAB* w systemie Quartus, ponieważ projekt ten był wcześniej symulowany w systemie ModelSim (patrz rozdział 10.6).

10.7.2. Badanie realizacji na FPGA różnych typów filtrów cyfrowych

Rozpatrzmy projekty czterech klas filtrów (lowpass, highpass, bandpass i bandstop), których specyfikacje są do siebie podobne (tab. 10.2).

TAB. 10.2. Specyfikacje badanych filtrów

Parametr	Lowpass	Highpass	Bandpass	Bandstop
Częstotliwość próbkowania F_s (Hz)	5100	5100	5100	5100
Częstotliwość graniczna pasma f_s (Hz)	530	785	–	–
Częstotliwość graniczna f_p (Hz)	785	530	–	–
Pierwsza częstotliwość graniczna pasma tłumienia f_{s1} (Hz)	–	–	530	785
Pierwsza częstotliwość graniczna pasma przepustowego f_{p1} (Hz)	–	–	785	530
Druga częstotliwość graniczna pasma przepustowego f_{p2} (Hz)	–	–	1030	1285
Druga częstotliwość graniczna pasma tłumienia f_{s2} (Hz)	–	–	1285	1030
Nierównomierność przepustowości A_p (dB)	1	1	1	1
Tłumienność w paśmie zaporowym A_s (dB)	60	60	60	60

Każdy z tych filtrów może być zaimplementowany jako filtr FIR lub filtr IIR. Zsyntetyzujemy te filtry (w sumie 8 projektów) w systemie MATLAB, a następnie

zrealizujemy je w systemie Quartus, aby odpowiedzieć na pytanie: która rodzina FPGA najlepiej nadaje się do implementacji konkretnego filtra?

W systemie MATLAB wszystkie filtry FIR były generowane domyślnie ze strukturą **Direct form FIR**, a filtry IIR ze strukturą **Direct form II, Second-Order Sections**. Współczynniki dla filtrów FIR obliczono za pomocą algorytmu **equiripple**, a dla filtrów IIR – za pomocą algorytmu **butler**. Wyniki syntezy rozpatrywanych filtrów w systemie MATLAB przedstawiono w tabeli 10.3.

TAB. 10.3. Parametry filtrów generowanych w systemie MATLAB

Parametr	Lowpass		Highpass		Bandpass		Bandstop		Best
	FIR	IIR	FIR	IIR	FIR	IIR	FIR	IIR	
Liczba sekcji	–	9	–	9	–	7	–	9	7
Długość	44		45		47		45		44
Stabilność	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	
Liniowość fazowa	Yes	No	Yes	No	Yes	No	Yes	No	
Liczba układów mnożących	44	46	45	45	47	29	45	46	29
Liczba sumatorów	43	36	44	36	46	21	44	36	21
Liczba stanów	–	18	–	18	–	14	–	18	14

Z tabeli 10.3 można zauważyć, że najlepszym spośród wszystkich filtrów pod względem kosztów realizacji okazał się filtr pasmowoprzepustowy (bandpass) IIR, który wymaga 29 układów mnożących (44–47 dla pozostałych filtrów) i 21 sumatorów (36–46 dla pozostałych filtrów). Wszystkie filtry są stabilne, ale tylko filtry FIR mają liniowość fazową.

Wniosek. Przy syntezie w systemie MATLAB filtr pasmowoprzepustowy wymaga do swojej budowy najmniejszej liczby sumatorów i układów mnożących w porównaniu z innymi klasami filtrów (lowpass, highpass i bandstop).

Następnie wszystkie osiem projektów filtrów zostało zaimplementowanych w systemie Quartus w wersji 18.1 na różnych rodzinach FPGA. Jako koszt realizacji przyjęto liczbę wykorzystanych elementów logicznych i wbudowanych bloków DSP (w tab. 10.4 podano w nawiasach), a jako wydajność – maksymalną częstotliwość roboczą w megahercach. Koszt realizacji rozpatrywanych projektów filtrów przedstawiono w tabeli 10.4, a wydajność – w tabeli 10.5.

Analiza tabeli 10.4 pokazuje, że Cyclone V jest najlepszą rodziną pod względem kosztów realizacji ze względu na efektywne wykorzystanie wbudowanych bloków DSP. Zwróć uwagę, że rodzina Arria II ma również wbudowane bloki DSP w swojej strukturze, ale nie są one tak efektywnie wykorzystywane i tylko dla filtrów pasmowoprzepustowych (bandpass) i pasmowozaporowych (bandstop).

Spśród wszystkich filtrów dla wszystkich rodzin FPGA, filtr pasmowoprzepustowy IIR (22 bloki DSP) ma najlepszy koszt implementacji pod względem liczby użytych bloków DSP. Należy się tego spodziewać, ponieważ ten filtr miał

najmniejszą liczbę układów mnożących i sumatorów w przypadku syntezy w systemie MATLAB.

TAB. 10.4. Koszt realizacji filtrów

FPGA	Lowpass		Highpass		Bandpass		Bandstop		Best
	FIR	IIR	FIR	IIR	FIR	IIR	FIR	IIR	
Cyclone IV	5469	6398	4971	6014	5977	4052	4960	5776	4052
Cyclone V	206 (41)	712 (37)	325 (36)	541 (36)	249 (44)	448 (22)	782 (36)	553 (36)	448 (22)
Cyclone 10	5401	6371	4968	6004	5969	4046	4971	5777	4046
Arria II	4763	6143	3475	5815	4068 (10)	2498 (14)	3178 (14)	4705 (12)	2498 (14)
MAX 10	5436	6419	4988	6024	5970	4056	5018	5762	4056
Best	206 (41)	712 (37)	325 (36)	541 (36)	249 (44)	448 (22)	782 (36)	553 (36)	

Jeśli porównamy koszt realizacji filtrów FIR i IIR, to koszt realizacji filtrów FIR jest o 10–20% niższy niż filtrów IIR. Wyjątkami są filtr pasmowoprzepustowy IIR dla wszystkich rodzin FPGA i filtr pasmowozaporowy IIR dla rodziny Cyclone V.

Wniosek. Jeśli wymagany jest najniższy koszt realizacji, należy zbudować filtr FIR (wyjątkiem jest filtr pasmowy), a jako rodzinę FPGA należy wybrać Cyclone V.

TAB. 10.5. Wydajność filtrów

FPGA	Lowpass		Highpass		Bandpass		Bandstop		Best
	FIR	IIR	FIR	IIR	FIR	IIR	FIR	IIR	
Cyclone IV	35.94	7.82	26.50	8.16	27.89	9.95	29.95	8.53	35.94
Cyclone V	5.28	6.08	5.41	7.13	5.01	8.06	5.48	7.03	8.06
Cyclone 10	29.81	7.38	26.21	8.05	28.13	9.73	32.48	8.51	32.48
Arria II	51.65	8.63	40.97	10.00	52.44	9.59	49.06	9.70	52.44
MAX 10	23.26	5.94	19.68	6.38	22.57	7.95	22.23	6.64	23.26
Best	51.65	8.63	40.97	10.00	52.44	9.95	49.06	9.70	

Analiza tabeli 10.5 pokazuje, że filtry realizowane w układach FPGA z rodziny Arria II mają najwyższą wydajność. Wyjątkiem jest filtr pasmowoprzepustowy IIR, który ma najwyższą wydajność, gdy jest implementowany w rodzinie Cyclone IV. Filtry zaimplementowane w układach FPGA z rodziny Cyclone V mają najniższą wydajność.

Jednocześnie nie udało się ustalić, który typ filtra na jakiej rodzinie układów FPGA powinien zostać zaimplementowany, aby osiągnąć największą wydajność. Na przykład dla Cyclone IV najlepszą wydajnością charakteryzuje się FIR lowpass, dla Cyclone V – IIR bandpass, dla Cyclone 10 – IIR bandstop, dla Arria II – FIR bandpass, dla MAX 10 – FIR lowpass.

Jeśli porównamy wydajność filtrów FIR i IIR, to filtry FIR są 3–4 razy szybsze niż filtry IIR, z wyjątkiem filtrów zaimplementowanych w układach FPGA z rodziny Cyclone V.

Wniosek. Aby uzyskać najwyższą wydajność z filtrów FIR lub IIR, należy wybrać filtr FIR (z wyjątkiem filtrów zaimplementowanych w układach FPGA z rodziny Cyclone V), a jako rodzina FPGA powinniśmy zatrzymać się na Arria II (z wyjątkiem filtra pasmowoprzepustowego IIR dla Cyclone IV).

10.8. Wnioski

Nowoczesne architektury FPGA nastawione są na wydajną implementację systemów DSP.

Przy wyborze pomiędzy filtrem FIR i IIR można kierować się następującymi zasadami: jeśli głównym wymaganiem jest nachylenie charakterystyki częstotliwościowej i szerokie pasmo, należy zastosować filtr IIR; jeśli głównym wymaganiem jest brak zniekształceń fazowych lub minimalne zniekształcenia, należy zastosować filtr FIR.

Rozmiar słowa wyjściowego projektu filtra w języku Verilog jest równy szerokości magistrali wejściowej plus rozmiar współczynników filtra. Należy dodać pewną liczbę bitów, aby potencjalnie zwiększyć rozmiar wyniku ze względu na operację dodawania.

Projektowanie filtrów cyfrowych na FPGA obejmuje kolejne wykonanie następujących etapów:

- definiowanie specyfikacji filtra;
- obliczanie współczynników filtra;
- definicję struktury filtra;
- analizę wpływu skończonej długości słowa na binarną reprezentację próbek wejściowych i współczynników filtra;
- symulację filtra;
- realizację filtra.

Do obliczania współczynników filtrów FIR najczęściej używane są metody optymalizacji, metoda ważenia i metoda próbkowania częstotliwości. Przy obliczaniu współczynników filtrów IIR najczęściej stosowaną metodą jest transformacja niezmienna odpowiedzi impulsowej, metoda transformacji dwuliniowej oraz metoda rozmieszczania zer i biegunów. Najbardziej wydajne metody obliczania współczynników filtra są implementowane w narzędziach programowych, na przykład MATLAB, Core Generator, TFilter, WinFilter, libdspl, TIBQ, miniDSP, dspGuru itp.

Inżynier musi bardzo uważać przy doborze długości słów reprezentujących próbki wejściowe i wyjściowe, a także współczynniki filtra. Po zakończeniu projektowania filtra należy ocenić wpływ skończonej długości słowa na działanie filtra i, jeśli to konieczne, podjąć odpowiednie kroki.

Budując filtr, który nie działa w czasie rzeczywistym, konieczne jest zapewnienie ładowania danych wejściowych z pamięci zewnętrznej. Implementując filtr działający w czasie rzeczywistym, należy wziąć pod uwagę częstotliwość próbkowania F_s , częstotliwość synchronizacji filtra oraz liczbę cykli zegarowych wymaganych do przetworzenia jednej próbki.

Program do tworzenia filtrów, który jest częścią MATLAB, zapewnia zintegrowane środowisko do projektowania filtrów cyfrowych, a także generuje kod języka opisu sprzętu (Verilog lub VHDL) w celu realizacji filtra na FPGA.

Program ModelSim, wchodzący w skład systemu Quartus, pozwala na symulację filtra za pomocą plików wsadowych generowanych w systemie MATLAB.

Implementacja filtra, którego kod został wygenerowany w systemie MATLAB na FPGA w systemie Quartus, nie różni się od realizacji jakiegokolwiek projektu pod względem jego opisu w języku Verilog. W tym celu wystarczy zdefiniować docelowy układ FPGA, skompilować projekt i skonfigurować układ FPGA.

Przy syntezie w systemie MATLAB filtr pasmowoprzepustowy wymaga do swojej budowy najmniejszej liczby sumatorów i układów mnożących w porównaniu z innymi klasami filtrów (dolnoprzepustowy, górnoprzepustowy i pasmowozaporowy).

Jeśli wymagany jest najniższy koszt realizacji, należy zbudować filtr FIR (wyjątkiem jest filtr pasmowy), a rodzina Cyclone V powinna zostać wybrana jako FPGA.

Aby uzyskać najwyższą wydajność, należy wybrać filtr FIR (z wyjątkiem filtrów zaimplementowanych w układach FPGA z rodziny Cyclone V), a jako rodzinę FPGA należy wybrać Arria II, z wyjątkiem filtra pasmowoprzepustowego IIR dla Cyclone IV.

Część V

Projektowanie podsystemu synchronizacji

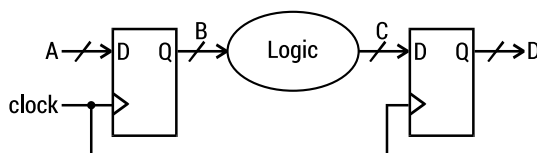
W przeciwieństwie do projektów na układach ASIC (application-specific integrated circuit), w projektach na FPGA nie ma potrzeby projektowania całego podsystemu synchronizacji. Faktem jest, że w FPGA wiele elementów podsystemu synchronizacji zostało już zaprojektowanych i zaimplementowanych na poziomie sprzętowym przez projektantów FPGA. Te elementy obejmują *sieci zegarowe* (clock networks), *bloki pętli synchronizacji fazowej* (PLL block), omówione w rozdziale 12 i *bloki sterowania zegarem* (clock control blocks). Ponadto Kompilator systemu Quartus automatycznie rozwiązuje wiele problemów związanych z synchronizacją projektów. Problem efektywnego wdrażania podsystemu synchronizacji nabiera szczególnego znaczenia przy opracowywaniu szybkich i złożonych projektów.

11. Realizacja projektów synchronicznych

W tym rozdziale omówiono różne kwestie, przed którymi stoi inżynier podczas projektowania podsystemu synchronizacji dla złożonego projektu FPGA. Realizacja projektów synchronicznych na FPGA ma swoją własną charakterystykę, której znajomość jest niezbędna do efektywnego wykorzystania możliwości sprzętowych FPGA, dostarczanych przez narzędzia projektowe. W tym rozdziale omówiono zalecenia firmy Intel dotyczące projektowania synchronicznych układów na FPGA, multipleksowania i bramkowania sygnału zegarowego, zalecenia projektowe dla szybkich i złożonych projektów, implementacje sygnału resetowania oraz atrybuty syntezy, które mają wpływ na elementy związane z synchronizacją układów.

11.1. Zasady funkcjonowania projektów synchronicznych

Wszystkie systemy wbudowane są obecnie generalnie implementowane jako *projekty synchroniczne*. W projekcie synchronicznym (rys. 11.1) aktywność wszystkich wejść i wyjść jest kontrolowana przez sygnał zegarowy *clock*. Oznacza to, że na każdym aktywnym zboczu sygnału zegarowego (w FPGA zbocze aktywne jest zboczem dodatnim) dane z wejść rejestrów są przenoszone na ich wyjścia, tzn. dane z wejścia A są przesyłane na wyjście B, a dane z wejścia C na wyjście D (rys. 11.1). Po nadejściu aktywnego zbocza sygnału zegarowego *clock*, zaczynają się zmieniać wartości wyjść logiki kombinacyjnej *Logic*, które zasilają wejścia rejestrów. Zmiany te inicjują okres niestabilności wyjść C kombinacyjnej logiki *Logic*, który po pewnym czasie się kończy, a wyjścia C są ustawiane na nową wartość.



rys. 11.1. Fragment projektu synchronicznego

Niestabilność wyjść logiki kombinacyjnej nie wpływa na działanie projektu, jeśli spełnione są następujące warunki:

- przed aktywnym zboczem sygnału zegarowego wejście danych (A i C na rys. 11.1) jest stabilne, nie krócej niż *czas ustawiania* (setup time) rejestru t_{SU} ;
- po aktywnym zboczach sygnału zegarowego wejście danych pozostaje stabilne przynajmniej przez *czas utrzymywania* (hold time) rejestru t_H .

Aby spełnić wymagania dotyczące czasu ustawiania i czasu utrzymywania, wyjścia logiki kombinacyjnej, które zasilają rejestr, muszą być podawane synchronicznie, zgodnie z sygnałem taktowania rejestru. Dotyczy to również danych wejściowych zewnętrznych (do FPGA). Jeśli sygnały zewnętrzne są asynchroniczne, należy je zsynchronizować, tzn. *powiązać z zegarem*, czyli uzależnić ich zmiany od sygnału synchronizacji, aby zapobiec naruszeniu czasów ustawiania i utrzymywania.

W przypadku przekroczenia czasu ustawiania t_{SU} lub czasu utrzymywania t_H rejestru, wyjście rejestru przechodzi do poziomu pośredniego (którego wartość znajduje się pomiędzy wysokim a niskim poziomem sygnału), zwanego *stanem metastabilnym* (metastable state). Wyjście rejestru może pozostawać w stanie metastabilnym przez długi czas (więcej niż jeden cykl zegarowy), a następnie nieprzewidywalnie powraca do jednego ze stanów stabilnych. Dlatego do stabilnej pracy projektu synchronicznego konieczne jest całkowite wykluczenie możliwości przejścia dowolnego rejestru projektu do stanu metastabilnego.

Sprawdzanie spełnienia wymagań dotyczących czasu ustawiania i czasu utrzymywania w projektach FPGA, a także prawdopodobieństwa przejścia rejestrów do stanu metastabilnego przeprowadza się w systemie Quartus za pomocą *analizatora czasowego* (time analyser) TimeQuest [9].

11.2. Zalecenia firmy Intel dotyczące projektowania niezawodnych projektów synchronicznych

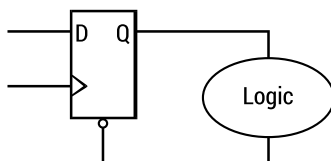
Szczególnie niebezpieczne w projektach synchronicznych są impulsy przejściowe zwane *hazardami* (hazard), *usterkami* (faults) lub *zakłóceniami* (glitches), a także krótsze impulsy zwane *szpilkami* (spikes), które mogą pojawić się w obwodach synchronizacji i resetowania rejestrów. Dlatego należy bardzo starannie zaprojektować logikę zasilającą takie obwody.

11.2.1. Unikanie pętli kombinacyjnych

Pętla kombinacyjna (combinational loop) jest to układ kombinacyjny ze sprzężeniem zwrotnym, który nie zawiera rejestrów. Pętle kombinacyjne są najczęstszymi przyczynami *niestabilności* (instability) i *zawodności* (unreliability) układów cyfrowych.

Pętle kombinacyjne są tworzone w projekcie w następujących przypadkach:

- kiedy w kodzie *języka opisu sprzętu* (hardware description language – HDL) lewa strona wyrażenia w instrukcji docelowej pojawia się również po prawej stronie;
- gdy wyjście rejestru poprzez logikę kombinacyjną powraca do asynchronicznego wejścia tego samego rejestru (rys. 11.2).



RYS. 11.2. Pętla kombinacyjna w sterowaniu rejestrem

Niebezpieczeństwo związane z pętlami kombinacyjnymi polega na tym, że zachowanie pętli zależy od *opóźnień propagacji* (propagation delays) przez układ logiczny tworzący pętlę. Jednak opóźnienia propagacji mogą się różnić ze względu na wahania temperatury lub napięcia, co powoduje nieprzewidywalne zachowanie pętli. Ponadto zachowanie pętli zawsze się zmienia podczas implementacji projektu w innym układzie FPGA.

Wniosek. W projekcie synchronicznym sprzężenia zwrotne powinny zawsze obejmować rejestry.

11.2.2. Unikanie nieumyślnego wprowadzania zatrząsków do projektu

Częstym błędem w kodzie HDL jest nieumyślne wstawianie *zatrząsk* (latches). W języku Verilog kompilator wprowadza zatrząski w następujących przypadkach:

- w przypadku braku co najmniej jednego z sygnałów wejściowych na liście czułości przy opisie układu kombinacyjnego, dlatego też należy opisać listę czułości dla układów kombinacyjnych w postaci @ * lub @(*);
- przy opisywaniu układu kombinacyjnego, jeśli dla niektórych kombinacji sygnałów wejściowych wartości wyjść nie są jednoznacznie zdefiniowane, dlatego też zawsze przy opisie układów kombinacyjnych w instrukcji **if** należy uwzględnić konstrukcję **else**, a w przypadku instrukcji **case** – konstrukcję **default** [10].

Architektura FPGA nie jest zoptymalizowana do implementacji zatrząsk. W związku z tym projekty w FPGA z zatrząskami mają niższą wydajność i wyższy koszt realizacji w porównaniu do równoważnych projektów wykorzystujących rejestry.

Zatrząski są bardzo trudne do analizy czasowej ze względu na ich przezroczystość w przypadku wysokiego poziomu na wejściu sterującym. Analizator czasowy TimeQuest nie może przeprowadzać analizy *pożyczek cykli* (cycle-borrowing)

w projektach z zatrzaskami. Zakłócenia przechodzą również przez zatrzaski bez przeszkód w trybie przezroczystym, co dodatkowo komplikuje analizę czasową.

Wniosek. W projektach synchronicznych, tam gdzie to możliwe, zamień zatrzaski na rejestry.

11.2.3. Unikanie opóźnień w ścieżkach synchronizacji

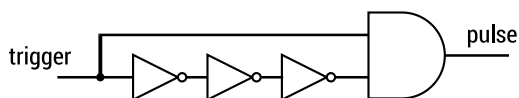
Projekty ASIC często wykorzystują łańcuchy opóźniające z szeregowo połączonych inwerterów (negacji), aby wyeliminować *wyścigi* (races) sygnałów. W FPGA problem wyścigów zegarowych jest rozwiązywany przez twórców architektury FPGA na poziomie sprzętowym: impuls zegarowy w domenie synchronizacji dociera do punktów końcowych (wejścia zegarowe rejestrów) prawie jednocześnie, tzn. z bardzo małym *opóźnieniem* (skew).

Za pomocą narzędzia syntezy FPGA, parzysta liczba inwerterów zostanie zoptymalizowana do bezpośredniego połączenia przewodowego, a nieparzysta liczba inwerterów zostanie zoptymalizowana do jednego negatora. Należy również pamiętać, że architektura FPGA nie zawiera oddzielnych bramek ani inwerterów. Najmniejszym elementem logiki FPGA jest generator funkcji LUT (Look-Up Table).

Wniosek. Obwody opóźniające w projektach FPGA nie mogą być budowane w postaci łańcuchów inwerterów. W tym celu należy używać rejestrów przesuwających, liczników i dzielników częstotliwości [10].

11.2.4. Używanie układów synchronicznych do wygenerowania pojedynczego impulsu

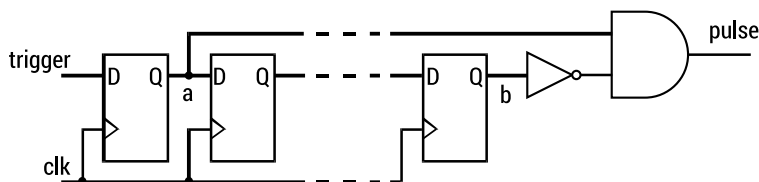
Czasami w projekcie konieczne jest wygenerowanie pojedynczego impulsu. Tradycyjnie używa się do tego dwuwejściowej bramki AND z łańcuchem inwerterów na jednym z wejść (rys. 11.3). Sygnał wyzwalający *trigger* zasila oba wejścia bramki AND. Szerokość impulsu zależy od różnicy czasu między bezpośrednią drogą sygnału *trigger* na wejście bramki AND a drogą przez inwertery.



RYS. 11.3. Tradycyjna realizacja pojedynczego impulsu

Jak wspomniano powyżej, ta implementacja pojedynczego impulsu nie nadaje się do układów FPGA ze względu na nieprzewidywalność opóźnienia w łańcuchu inwerterów. Ponadto impuls może nie być wystarczająco szeroki w niektórych zastosowaniach, szerokość impulsu zmienia się podczas przejścia na inny układ FPGA, a analiza taktowania nie może zweryfikować szerokości impulsu.

Aby zaimplementować pojedynczy impuls w projekcie FPGA, możesz użyć układu z rysunku 11.4. W tym przypadku szerokość impulsu zależy od liczby przerzutników w łańcuchu między punktami *a* i *b*. Dla jednego przerzutnika szerokość impulsu jest równa jednemu taktowi sygnału zegarowego *clk*, dla dwóch przerzutników – dwóm taktom, itd.



RYS. 11.4. Schemat do implementacji pojedynczego impulsu w FPGA

Wniosek. W projektach FPGA należy używać łańcucha przerzutników do wygenerowania pojedynczego impulsu, jak pokazano na rysunku 11.4.

11.2.5. Generacja zegara wewnętrznego

Układy FPGA mają zwykle niewielką liczbę (2–4) dedykowanych wyprowadzeń do podawania zewnętrznych sygnałów zegarowych. Te wyprowadzenia są często nazywane wejściami zegarowymi lub dedykowanymi wyprowadzeniami wejściowymi zegarowymi (dedicated clock input pins). Złożone projekty mogą wymagać znacznie większej liczby sygnałów zegarowych. Dlatego w projektach FPGA wejścia zegarowe służą do podawania *podstawowych* (basic) lub *referencyjnych* (reference) sygnałów zegarowych z zewnętrznego generatora sygnału zegarowego, na podstawie którego generowane są niezbędne wewnętrzne sygnały zegarowe.

Wewnętrzne sygnały zegarowe w układach FPGA są generowane przy użyciu bloków *pętli synchronizacji fazowej* PLL (omówionych w rozdziale 12). Do tego celu niepożądane jest użycie logiki FPGA. Faktem jest, że logika FPGA generuje nieprzewidywalne opóźnienia i zakłócenia, a generowane w taki sposób sygnały synchronizacji nie mogą być analizowane czasowo.

Wszystkie generowane wewnętrznie zegary muszą mieć zdefiniowane ograniczenia czasowe w pliku SDC projektu [9]. Te ograniczenia czasowe określają stosunek zegarów generowanych wewnętrznie do zegarów podstawowych. Spełnienie tego wymagania pozwala na prawidłowe przeprowadzenie analizy czasowej projektu.

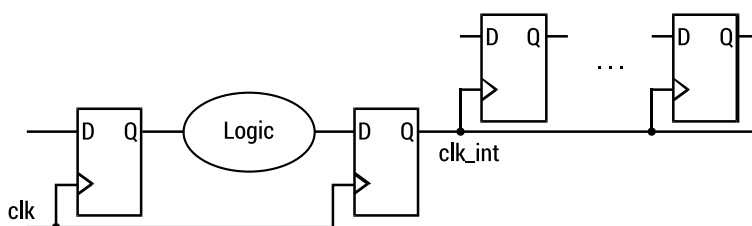
Do zegarów generowanych wewnętrznie należy przypisać dedykowane sieci synchronizacyjne, które mają dużą szybkość transferu i mają bardzo małe opóźnienie w porównaniu ze zwykłymi sieciami.

Wniosek. Zegary wewnętrzne w układach FPGA są generowane przez bloki PLL na podstawie referencyjnych zewnętrznych sygnałów zegarowych. Nie należy używać do tego wewnętrznej logiki FPGA.

11.2.6. Wykorzystanie wewnętrznej logiki FPGA jako źródła sygnału zegarowego lub sygnałów sterujących rejestrami

Czasami, ze względu na brak bloków PLL, nadal trzeba używać logiki FPGA jako źródła sygnałów zegarowych lub sygnałów sterujących rejestrami (resetowania lub ustawiania). Zazwyczaj wyjścia logiczne generują krótkotrwałe impulsy (zakłócenia lub szpilki). Jest to częste zjawisko i jest kompensowane przez rejestry w projektach synchronicznych.

Zupełnie inaczej jest, gdy wyjścia logiczne nie są podłączone do wejścia danych rejestru, ale na wejście zegarowe lub wejście sterujące rejestru. Na wejściu zegarowym rejestru zakłócenia powodują nieprzewidywalne przełączanie rejestrów, nieokreślony czas trwania sygnału zegarowego, naruszają czas ustawiania lub czas utrzymywania rejestru. To samo dotyczy sygnałów sterujących rejestrami. Z kolei wyjście takiego rejestru może spowodować zakłócenia w innym miejscu projektu. Dlatego, aby uniknąć tych problemów, wyjście logiki kombinacyjnej powinno być powiązane z zegarem przed użyciem go jako zegar lub sygnał sterowania (resetowania lub ustawiania) rejestru, jak pokazano na rysunku 11.5.



RYS. 11.5. Schemat do powiązania sygnału zegarowego *clk_int* generowanego przez logikę wewnętrzną z systemowym sygnałem zegarowym *clk*

W schemacie na rysunku 11.5 na wejściach i wyjściach logiki wewnętrznej *Logic* ustawiono rejestry do powiązania z sygnałem synchronizacji *clk*. W rezultacie sygnał zegarowy *clk_int* generowany przez wewnętrzną logikę będzie „powiązany” z systemowym zegarem *clk*.

11.2.7. Domeny synchronizacji

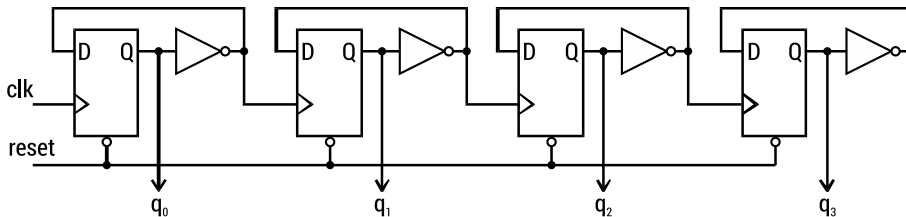
Każdy wewnętrznie wygenerowany zegar generuje własną *domenę synchronizacji* (clock domain), tzn. obszar FPGA, w którym wszystkie rejestry są taktowane przez ten sygnał zegarowy. W miarę możliwości należy unikać przesyłania danych między domenami synchronizacji. Jeśli nie jest to możliwe, można użyć pamięci typu FIFO jako bufora do wymiany danych między domenami synchronizacji. Aby skompensować zmienne opóźnienia między domenami synchronizacji, należy zastosować *niepewność sygnału zegarowego* (clock uncertainty) w ograniczeniach czasowych [9]. Zwykle niepewność synchronizacji wynosi od 10% do 15% opóźnienia sygnału zegarowego.

11.2.8. Dzielniki częstotliwości

Do dzielenia częstotliwości sygnałów zegarowych w projektach FPGA użyj wbudowanych bloków PLL. Jednak problem z dzielnikami częstotliwości może wynikać z niewystarczającej liczby PLL. W takim przypadku do podziału częstotliwości sygnału zegarowego należy wykorzystać liczniki synchroniczne lub automaty skończone z rejestrami na wyjściach [10]. Jednocześnie nie można wstawiać logiki kombinacyjnej na wyjściach liczników lub automatów skończonych, gdyż mogą być one źródłem zakłóceń dla sygnału synchronizacji.

11.2.9. Korzystanie z liczników kaskadowych (ripple counters)

Czasami, aby zmniejszyć użycie logiki, liczniki binarne są budowane w postaci kaskadowego połączenia przerzutników typu D (rys. 11.6), które nazywane są *licznikami kaskadowymi* (ripple counters). W liczniku kaskadowym wyjście przerzutnika każdego stopnia steruje wejściem zegarowym przerzutnika następnego stopnia przez negator, a ta sama wartość jest podawana przez obwód sprzężenia zwrotnego do wejścia przerzutnika.



RYC. 11.6. Schemat czterobitowego licznika tętnień

W architekturze FPGA nie ma negatorów, a przerzutniki nie mają wbudowanej inwersji wyjść. Dlatego inwertery w obwodzie licznika kaskadowego są realizowane za pomocą bramek logicznych FPGA. Poniżej znajduje się kod 4-bitowego licznika kaskadowego w języku Verilog.

```
module ripple_counter
(input clk,reset,
output [3:0] Q);

wire [3:0] c;

my_DFF ex0 (~c[0],clk,reset,c[0]);
my_DFF ex1 (~c[1],~c[0],reset,c[1]);
my_DFF ex2 (~c[2],~c[1],reset,c[2]);
```

```
my_DFF ex3 (~c[3],~c[2],reset,c[3]);
```

```
assign Q = c;
```

```
endmodule
```

```
module my_DFF(  
input d, clk, reset,  
output reg q);
```

```
always @(posedge clk, negedge reset)  
    if (!reset) q <= 1'b0;  
    else      q <= d;
```

```
endmodule
```

Tradycyjny opis licznika binarnego w języku Verilog jest następujący:

```
module binary_counter  
#(parameter N=4)  
(input clk,reset,  
output reg [N-1:0] Q);
```

```
always @(posedge clk, negedge reset)  
    if (!reset) Q <= {N{1'b0}};  
    else      Q <= Q + 1'b1;
```

```
endmodule
```

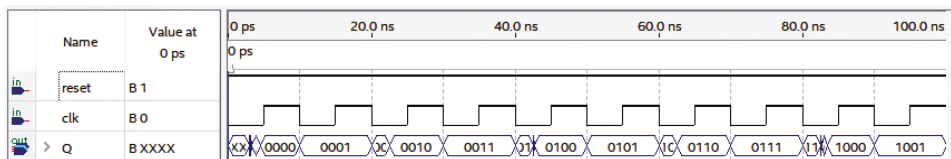
Zasoby FPGA wykorzystywane w obu projektach przedstawiono w tabeli 11.1, gdzie L to liczba użytych elementów logicznych FPGA; R to liczba użytych przerzutników. Z tabeli 11.1 wynika, że licznik kaskadowy wykorzystuje o 3 elementy logiczne więcej niż licznik binarny. Te trzy elementy logiczne są wymagane do implementacji inwerterów między przerzutnikami.

TAB. 11.1. Zasoby FPGA używane podczas realizacji licznika kaskadowego i licznika binarnego

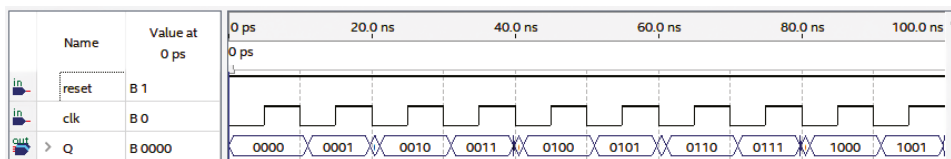
Projekt	L	R
ripple_counter	7	4
binary_counter	4	4

Problemy pojawiają się również przy analizie czasowej licznika kaskadowego, ponieważ wejścia zegarowe trzech przerzutników (rys. 11.6) są zasilane z wyjść przerzutników poprzedniego stopnia i nie jest możliwe wyznaczenie dla nich zapasów

czasowych w odniesieniu do sygnału zegarowego. Ponadto w licznikach kaskadowych zakłócenia rozprzestrzeniają się wzdłuż linii zegarowych przerzutników i zwiększają się one wraz ze wzrostem liczby bitów licznika. Wyraźnie widać to na diagramach symulacji czasowej licznika kaskadowego (rys. 11.7) i licznika binarnego (rys. 11.8).



RYS. 11.7. Wyniki symulacji czasowej projektu *ripple_counter*



RYS. 11.8. Wyniki symulacji czasowej projektu *binary_counter*

Zakłócenia pojawiają się również w liczniku binarnym, ale są one znacznie krótsze niż w liczniku kaskadowym.

Wniosek. Należy unikać używania liczników kaskadowych w projektach synchronicznych na FPGA, ponieważ zwiększa to tylko liczbę używanych elementów logicznych, uniemożliwia analizę czasową i wydłuża czas trwania zakłóceń.

11.2.10. Zastosowanie sieci synchronizacji FPGA

Architektura nowoczesnych układów FPGA obejmuje następujące specjalne sieci synchronizacji:

- *globalne* (global clock networks – GCLK);
- *regionalne* (regional clock networks – RCLK);
- *dedykowane szybkie regionalne* (dedicated fast regional clock networks).

Sieci te są zorganizowane w strukturę hierarchiczną i zapewniają wystarczającą ilość sygnałów zegarowych w każdym obszarze układu FPGA z małym opóźnieniem i niewielkim opóźnieniem (skew).

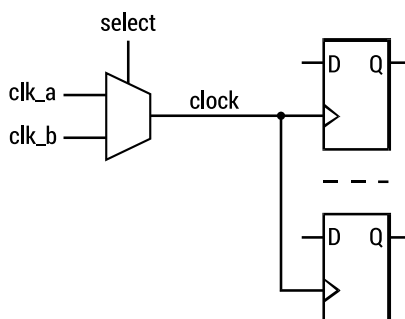
Oprogramowanie Quartus automatycznie wykorzystuje sieci globalne dla sygnałów sterujących przerzutnikami o dużym współczynniku rozgałęzienia na wyjściu, a także wyjść bloków PLL. Ponadto, korzystając z opcji **Global Signal**, użytkownik może przypisać trasowanie określonych sygnałów w globalnych sieciach synchronizacji.

Aby w pełni wykorzystać zasoby trasowania sieci synchronizacji, należy ograniczyć liczbę projektowanych zegarów do liczby dedykowanych sieci synchronizacji dostępnych w docelowym układzie FPGA.

Wniosek. Należy wykorzystywać w pełni sieci zegarowe FPGA do przesyłania sygnałów zegara i sygnałów sterowania rejestrami. Zwiększa to szybkość i niezawodność projektu. Jednak liczba sygnałów zegarowych i sygnałów sterujących rejestrami w projekcie nie powinna przekraczać liczby sieci synchronizacji w docelowym układzie FPGA.

11.3. Multipleksowanie sygnałów zegarowych

W projektach synchronicznych może być konieczne sterowanie tą samą funkcją logiczną z różnych źródeł sygnału zegarowego. Na przykład w aplikacjach telekomunikacyjnych, które używają różnych standardów częstotliwości, a także w trybie gotowości (stand-by), częstotliwość synchronizacji systemu może zostać zmniejszona, aby zmniejszyć zużycie energii. W takich projektach wykorzystuje się *multipleksowanie* (multiplexing) źródeł sygnału zegarowego (rys. 11.9).



RYŚ. 11.9. Multipleksowanie sygnałów zegarowych

Istotą multipleksowania sygnałów zegarowych jest podłączenie, w zależności od wartości sygnału sterującego *select*, wejścia sygnału zegarowego *clock* do różnych źródeł impulsów zegarowych: *clk_a* lub *clk_b*.

W projektach FPGA multipleksowanie sygnału zegarowego może być realizowane przy użyciu wbudowanych bloków PLL lub za pomocą logiki wewnętrznej. W układach Intel FPGA do multipleksowania sygnałów zegarowych również mogą być używane *bloki sterowania zegarem* (clock control blocks).

Wykorzystanie bloków PLL jako multiplekserów zegarowych zostanie omówione w następnym rozdziale.

Poniższy kod opisuje rejestr taktowany dwoma różnymi sygnałami *clk_a* i *clk_b*. Przełączanie między sygnałami synchronizacji jest realizowane za pomocą multipleksa 2-1 sterowanego sygnałem *select*.

```

module clock_MUX(      // przykład multipleksowania sygnałów zegarowych
input [3:0] data,      // magistrala wejściowa
input clk_a,clk_b,select,reset, // sygnały sterujące
output [3:0] out);    // magistrala wyjściowa

wire clock;           // zegar

my_MUX clk_mux(clk_a,clk_b,select,clock); // multiplekser

reg_N_bits #(4) my_reg(clock,reset,data,out); // rejestr
endmodule

module my_MUX(         // moduł multipleksera 2-1
input a,b,
input s,
output reg mux_out);

always @(*)
    if (!s)      mux_out = a;
    else        mux_out = b;

endmodule

module reg_N_bits      // moduł rejestru
#(parameter N=4)
(input clk, reset,
input wire [N-1:0] data,
output reg [N-1:0] out);

always @(posedge clk, negedge reset)
    if (~reset) out <= 0;
    else      out <= data;

endmodule

```

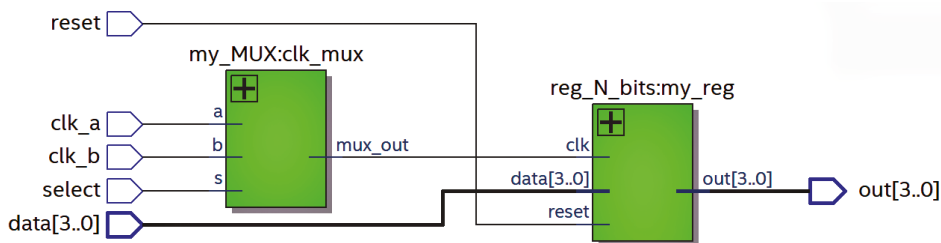
Plik ograniczeń czasowych SDC do multipleksowania sygnału zegarowego musi zawierać polecenie `set_clock_groups` z opcją – **exclusive** [9], jak w przykładzie poniżej.

```

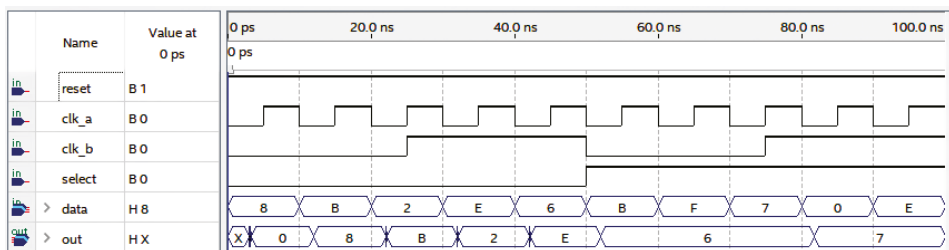
create_clock -name clk_a -period 10 [get_ports clk_a]
create_clock -name clk_b -period 50 [get_ports clk_b]
set_clock_groups -exclusive -group {clk_a} -group {clk_b}

```

Wyniki syntezy i symulacji projektu `clock_MUX` przedstawiono, odpowiednio, na rysunkach 11.10 i 11.11.



RYS. 11.10. Wyniki syntezy projektu *clock_MUX* na poziomie RTL



RYS. 11.11. Wyniki symulacji czasowej projektu *clock_MUX*

W pokazanym na rysunku 11.11 przykładzie symulacji, sygnał zegarowy *clk_a* ma okres 10 ns, a sygnał zegarowy *clk_b* ma okres 25 ns. Przez pierwsze 50 ns rejestr jest taktowany zegarem *clk_a*, a następnie zegarem *clk_b*. Wygenerowane wartości wyjściowe można obserwować na wyjściu *out*.

W projektach synchronicznych użycie logiki (tzn. multiplekserów, jak w powyższym projekcie) do multipleksowania sygnałów zegarowych jest dozwolone pod następującymi warunkami:

- sygnał zegarowy *clock* wykorzystuje globalne zasoby synchronizacji;
- logika multipleksowania sygnałów zegarowych nie zmienia się od początku konfiguracji;
- rejestry są zawsze kasowane przy przełączaniu sygnałów zegarowych;
- w przypadku chwilowo niepoprawnej odpowiedzi, następne przełączenie sygnałów zegarowych nie ma negatywnych konsekwencji.

Aby móc przełączać sygnały zegarowe w czasie rzeczywistym bez resetowania rejestrów, projekt musi zapewniać:

- brak naruszeń synchronizacji w rejestrach;
- brak zakłóceń w sygnałach synchronizacji;
- brak wyścigów w sygnałach synchronizacji.

Aby uniknąć większości tych trudności, podczas multipleksowania sygnałów zegarowych w układach Intel FPGA należy używać wbudowanych *bloków sterowania zegarem* (clock control blocks). Te dedykowane bloki sterujące synchronizacją zapewniają małe opóźnienie globalnego trasowania synchronizacji, brak opóźnień linii

synchronizacji, brak usterek i wyścigów sygnałów zegarowych. Na przykład układy FPGA Cyclone V mają cztery typy bloków sterowania zegarem: GCLK, RCLK, PCLK i External PLL Clock Output Control Block. Szczegółowe informacje na temat wykorzystania każdego z tych bloków można znaleźć w dokumentacji technicznej docelowego układu FPGA, na przykład [12].

11.4. Bramkowanie sygnałów zegarowych

11.4.1. Bramkowanie sygnałów zegarowych w projektach na FPGA

W praktyce, w celu zmniejszenia zużycia energii, powszechny stał się mechanizm bramkowania sygnału zegarowego. Jeśli przez jakiś czas część układu nie działa (nie wykonuje przydatnych czynności), to w celu zmniejszenia przez nią zużycia energii można wyłączyć synchronizację. To włączanie lub wyłączanie sygnału zegarowego nazywane jest *bramkowaniem* lub *strobowaniem synchronizacji* (clock gating).

Generalnie synchronizacja jest włączona, gdy sygnał bramki *gate* (sygnał *strobu-jący*) jest wysoki, a synchronizacja jest wyłączona, gdy sygnał bramki *gate* jest niski. Bramkowanie sygnału zegarowego może być wykonywane na źródle sygnału zegara, w każdym rejestrze (przerzutniku) lub gdzieś w węzle pośrednim *sieci zegarowej* (clock network). Aby uzyskać największe oszczędności energii, bramkowanie sygnału zegarowego powinno być wykonywane jak najbliżej źródła zegara.

Przypomnijmy, że wszystkie przerzutniki nowoczesnych układów FPGA mają wejście włączające zegar ENA. Dlatego układy FPGA pozwalają na bramkowanie każdego pojedynczego przerzutnika.

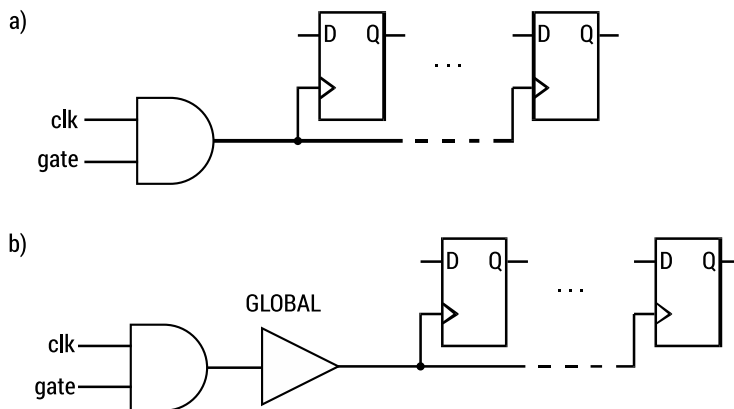
Bramkowanie polega na włączaniu (zezwoleniu) lub wyłączaniu określonego sygnału zegarowego *clk* w zależności od wartości sygnału bramki *gate*. Kiedy jakiś sygnał synchronizacji jest wyłączony, domena sterowana przez ten zegar jest zamykana i staje się funkcjonalnie nieaktywna. Wszystkie rejestry zawarte w domenie przestają się przełączać, zmniejszając w ten sposób dynamiczne zużycie energii.

Jednak bramkowanie sygnałów zegarowych może skomplikować realizację projektu i jego analizę czasową. *Zegary bramkowane* (gated clocks) mają tendencję do opóźnień (skew) i są wrażliwe na zakłócenia, które mogą powodować błędy w projekcie. Ponadto sygnał bramki jest zwykle generowany przez układ logiczny FPGA i może być synchroniczny lub asynchroniczny.

Spróbujmy przyjrzeć się metodom bramkowania sygnału zegarowego w projektach FPGA.

11.4.2. Metody bramkowania sygnałów zegarowych

Tradycyjnym sposobem bramkowania sygnału zegarowego, dodatnim aktywnym zboczem jest użycie bramki AND w sieci synchronizacji (rys. 11.12).



RYS. 11.12. Bramkowanie sygnału synchronizacji za pomocą bramki AND przy użyciu zwykłych obwodów trasowania (a) i za pomocą globalnych sieci synchronizacji (b)

W przypadku ujemnego aktywnego zbocza sygnału synchronizacji w obwodzie z rysunku 11.12 należy użyć bramki OR.

Wadą obwodu z rysunku 11.12a jest to, że wyjście bramki AND jest połączone z wejściami zegarowymi rejestrów za pomocą zwykłych obwodów trasowania, które nie są przeznaczone do przenoszenia sygnałów zegarowych. Aby wyeliminować tę wadę w systemie Quartus, można skorzystać z bufora GLOBAL (rys. 11.12b), którego wyjście jest zawsze podłączone do globalnych lub regionalnych sieci synchronizacji FPGA. Globalne i regionalne sieci synchronizacji FPGA są szybkie, charakteryzują się niewielkim opóźnieniem i są specjalnie zaprojektowane do przesyłania sygnałów sterujących rejestrami (*clock*, *reset* i *clock enable*).

Poniżej znajduje się opis projektu rejestru w języku Verilog, który odpowiada schematowi z rysunku 11.12a.

```

module reg_AND
#(parameter N=4)
(input [N-1:0] d,
input clk, reset, gate,
output reg [N-1:0] q);

wire gated_clock;

assign gated_clock = clk & gate;

always @(posedge gated_clock, negedge reset)
    if(~reset) q <= 0;
    else      q <= d;

endmodule

```

Rejestr odpowiadający schematowi z rysunku 11.12b, można opisać w języku Verilog w następujący sposób:

```
module reg_AND_GLOBAL
#(parameter N=4)
(input [N-1:0] d,
input clk, reset, gate,
output reg [N-1:0] q);

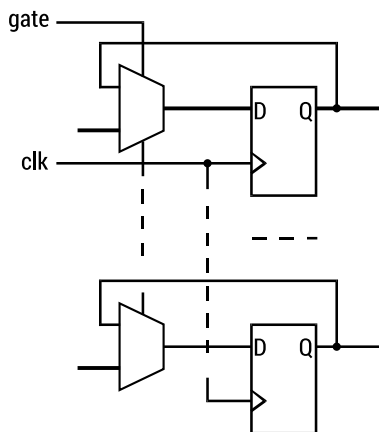
wire gated_clock;

GLOBAL ex1(.in(clk & gate), .out(gated_clock));

always @(posedge gated_clock, negedge reset)
    if(~reset) q <= 0;
    else      q <= d;
endmodule
```

Ostatni kod wykorzystuje prymityw GLOBAL systemu Quartus, który zmusza kompilator do wstawienia bufora GLOBAL do obwodu, którego wyjście jest zawsze połączone z globalną siecią synchronizacji.

Kolejny sposób bramkowania rejestrów wykorzystuje multipleksery na wejściu każdego przerzutnika (rys. 11.13).



RYŚ. 11.13. Bramkowanie danych z multipleksarami

Ta metoda nie strobuje sygnałów zegarowych, ale strobuje dane wejściowe rejestru. Gdy sygnał bramki *gate* jest wysoki, dane z magistrali wejściowej trafiają do wejścia danych przerzutników. Gdy sygnał bramki *gate* jest niski, wartości z wyjść przerzutników są wysyłane do wejścia przerzutnika.

Może się wydawać, że schemat na rysunku 11.13 jest bardziej złożony niż schemat na rysunku 11.12. Jednak w układach FPGA przerzutnik z multiplexerem wejściowym jest realizowany przy użyciu pojedynczego elementu logicznego FPGA. Dlatego, po zaimplementowaniu na FPGA, schemat z rysunku 11.13 nie jest bardziej skomplikowany niż schemat z rysunku 11.12.

Schemat rejestrów z rysunku 11.13 można opisać następującym kodem:

```

module reg_MUX
#(parameter N=4)
(input [N-1:0] d,
input clk, reset, gate,
output reg [N-1:0] q);

wire [N-1:0] gdata;

assign gdata = gate ? d : q;

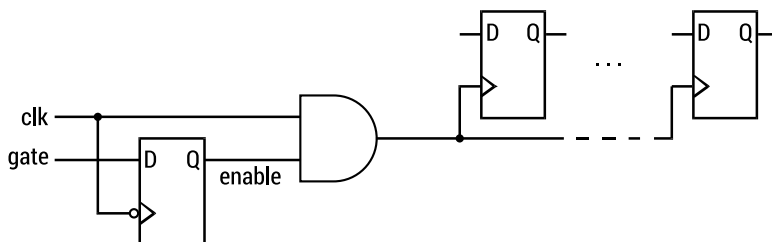
always @(posedge clk, negedge reset)
    if(~reset) q <= 1'b0;
    else      q <= gdata;

endmodule

```

Zaletą obwodu z rysunku 11.13 jest to, że nie ma on wpływu na zegar *clk*. Dlatego też sygnał synchronizacji ze źródła do rejestru może być przesyłany przez globalne sieci synchronizacji.

W przypadku złożonej logiki generowania sygnału bramki *gate*, sygnał ten może być asynchroniczny i zawierać zakłócenia. W takich sytuacjach firma Intel zaleca stosowanie schematu bramkowania pokazanego na rysunku 11.14.



RYS. 11.14. Synchronizacja sygnału bramki *gate*

Tutaj sygnał bramki *gate* trafia na wejście przerzutnika, które jest taktowane przez ujemne zbocze sygnału zegarowego *clk*. Na wyjściu pierwszego przerzutnika generowany jest synchroniczny sygnał zezwolenia *enable*, który jest używany na wejściu bramki AND do bramkowania sygnału zegarowego *clk*. Kod rejestru z rysunku 11.14 wygląda tak:

```

module reg_sync_AND
#(parameter N=4)
(input [N-1:0] d,
input clk, reset, gate,
output reg [N-1:0] q);

wire clk_e;
reg enable;

always @(negedge clk, negedge reset)
    if(~reset) enable <= 0;
    else      enable <= gate;

assign clk_e = clk & enable;

always @(posedge clk_e, negedge reset)
    if(~reset) q <= 0;
    else      q <= d;

endmodule

```

Na rysunku 11.14 wyjście bramki AND jest połączone z wejściami zegarowymi przerzutników za pomocą normalnych sieci trasowania.

Poniższy kod w układzie z rysunku 11.14 wprowadza bufor GLOBAL do transmisji sygnału zegarowego przez globalne sieci synchronizacji.

```

module reg_sync_AND_GLOBAL
#(parameter N=4)
(input [N-1:0] d,
input clk,reset,gate,
output reg [N-1:0] q);

wire clk_e;
reg enable;

always @(negedge clk, negedge reset)
    if(~reset) enable <= 0;
    else      enable <= gate;

GLOBAL ex1(.in(clk & enable), .out(clk_e));

```

```

always @(posedge clk_e, negedge reset)
    if(~reset) q <= 0;
    else      q <= d;

endmodule

```

Można również użyć wejścia do włączania zegara ENA, które znajduje się w każdym przerzutniku we współczesnych układach FPGA, do bramkowania sygnałów zegarowych. Kod projektu rejestru wykorzystującego wejście ENA do bramkowania sygnału zegarowego jest następujący:

```

module reg_enable
#(parameter N=4)
(input [N-1:0] d,
input clk,reset,gate,
output reg [N-1:0] q);

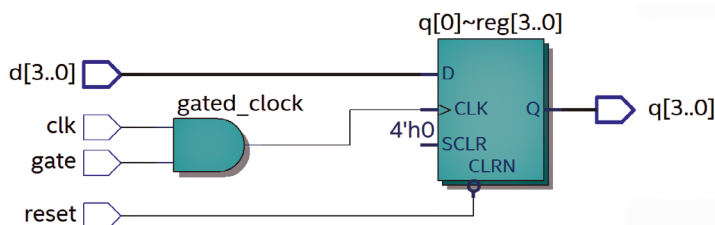
always @(posedge clk, negedge reset)
    if(~reset)      q <= 0;
    else if(gate)    q <= d;
    else           q <= q;

endmodule

```

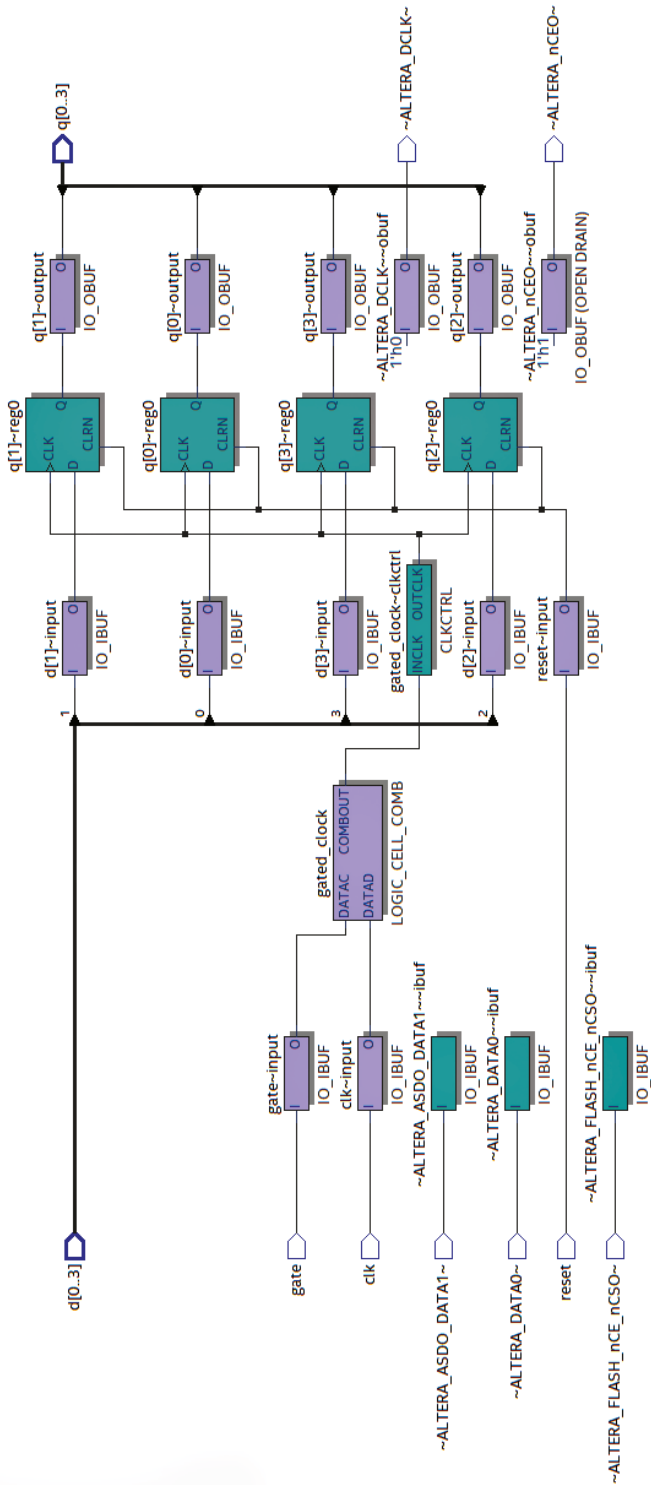
11.4.3. Realizacja rejestrów z bramkowaniem sygnału zegarowego

Wszystkie rozważane projekty rejestrów zostały zrealizowane w systemie Quartus w wersji 18.1 na układach FPGA z rodziny Cyclone IV E. Jako przykład rozpatrzmy realizację projektu *reg_AND*. Widok projektu na poziomie RTL przedstawia rysunek 11.15.



RYS. 11.15. Wyniki syntezy projektu *reg_AND* na poziomie RTL

W tym projekcie interesująca jest mapa technologiczna projektu po dopasowaniu (Technology Map Post-Fitter), pokazana na rysunku 11.16.

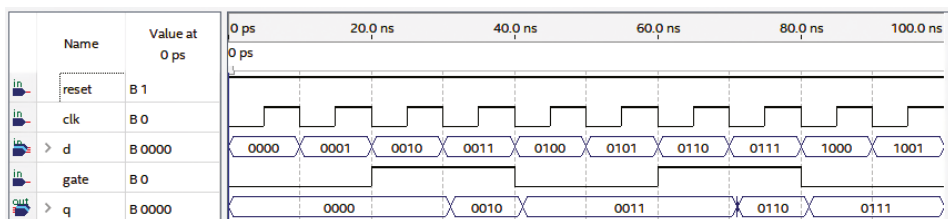


rys. 11.16. Mapa technologiczna po dopasowaniu projektu reg_4

Na rysunku 11.16 można zauważyć, że na wyjściu elementu logicznego LOGIC_CELL_COMB, który implementuje bramkę AND, wstawiony jest blok sterowania synchronizacją CLKCTRL, którego wyjście jest zawsze podłączone do globalnej sieci synchronizacji. Oznacza to, że bramkowany sygnał zegarowy *gated_clock* będzie przesyłany do wejść zegarowych przerzutników przez globalną sieć zegarową.

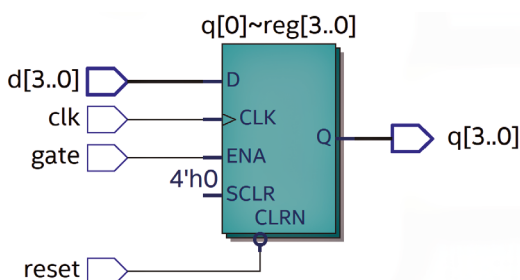
Kompilator automatycznie wprowadzi blok sterowania synchronizacją CLKCTRL, ponieważ opcja **Auto GLOBAL Clock** jest zawsze włączona w systemie Quartus Prime w wersji 18.1 i nowszych. Tej opcji nie ma na liście dostępnych opcji w wersji 18.1, ale była obecna we wcześniejszych wersjach systemu Quartus. Opcja **Auto GLOBAL Clock** mówi kompilatorowi, aby zawsze przypisywał zegary rejestrów do wolnych globalnych sieci zegarowych. Dlatego wprowadzenie bufora GLOBAL w projekcie *reg_AND_GLOBAL* nie zmienia projektu: wyniki projektu *reg_AND_GLOBAL* całkowicie pokrywają się z projektem *reg_AND*.

Wyniki symulacji czasowej projektów *reg_AND* i *reg_AND_GLOBAL* pokazano na rysunku 11.17.



RYS. 11.17. Wyniki symulacji czasowej projektów *reg_AND* i *reg_AND_GLOBAL*

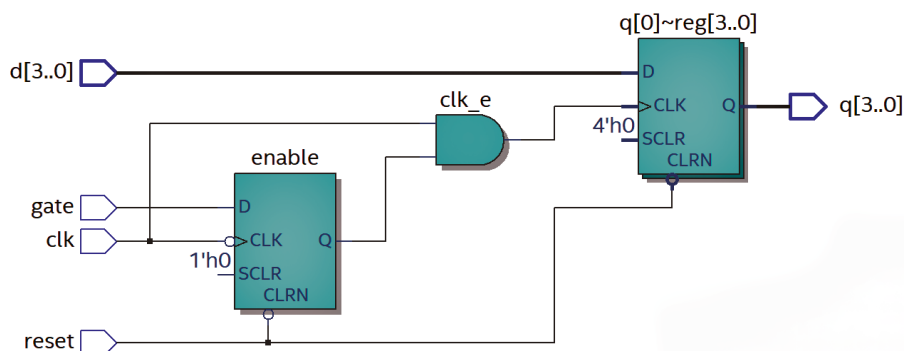
Wyniki syntezy projektu *reg_MUX* przedstawiono na rysunku 11.18.



RYS. 11.18. Wyniki syntezy projektu *reg_MUX* na poziomie RTL

Cechą szczególną projektu *reg_MUX* jest to, że kompilator automatycznie przypisał sygnał bramki *gate* do wejścia ENA przerzutników rejestru.

Realizacja projektu *reg_sync_AND* (rys. 11.19) w pełni odpowiada schematowi z rysunku 11.14. Z mapy technologicznej po dopasowaniu projektu (rys. 11.20) widać, że wygenerowany bramkowany sygnał zegarowy *clk_e* podawany jest na wejścia zegarowe przerzutników poprzez blok sterujący synchronizacją CLKCTRL, tzn. za pomocą globalnych sieci zegarowych. Innymi słowy, kompilator automatycznie wprowadzi blok CLKCTRL dla wewnętrznie generowanego sygnału zegarowego *clk_e*. Dlatego wyniki syntezy projektu *reg_sync_AND_GLOBAL* całkowicie pokrywają się z wynikami syntezy *reg_sync_AND*.



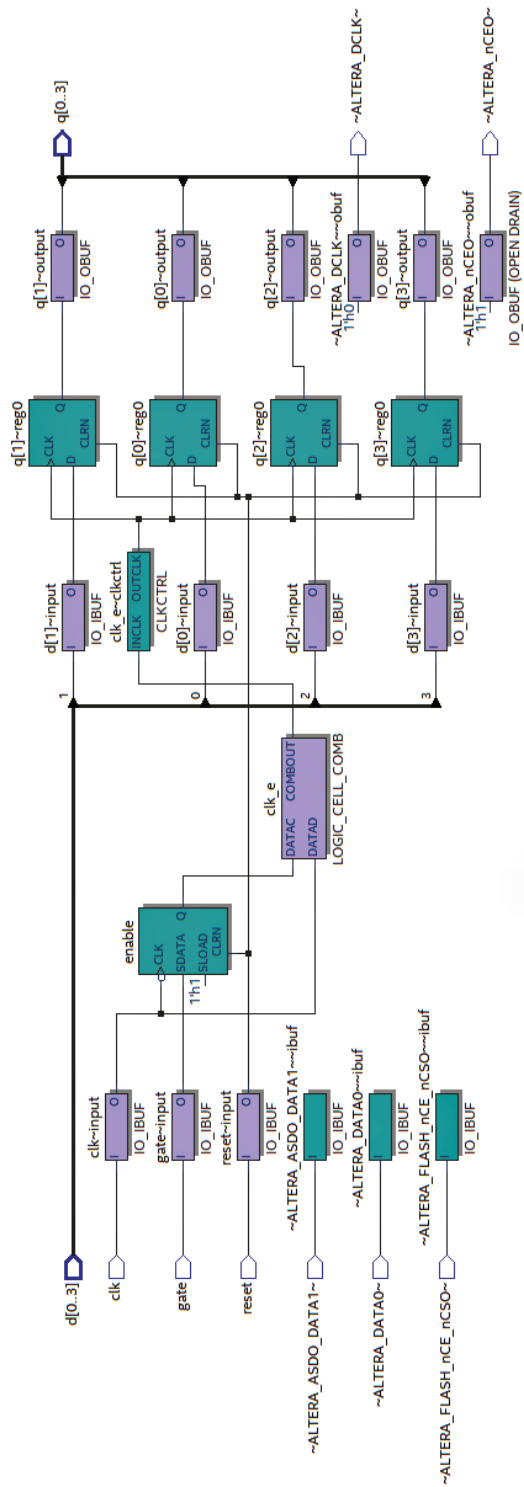
RYŚ. 11.19. Wyniki syntezy projektu *reg_sync_AND* na poziomie RTL

Realizacja projektu *reg_enable* jest całkowicie taka sama, jak projektu *reg_MUX*.

Rezultaty symulacji czasowej wszystkich rozpatrywanych projektów praktycznie pokrywają się (rys. 11.17). Niewielką różnicę obserwuje się tylko dla projektów *reg_MUX* i *reg_enable*, dla których wartości na wyjściach formują się na 1–2 ns wcześniej niż dla innych projektów.

Wnioski. W systemie Quartus sygnały synchronizacji generowane przez wewnętrzną logikę są automatycznie przypisywane do wolnych globalnych lub regionalnych sieci synchronizacji. Dlatego nie ma potrzeby przypisywania bufora GLOBAL do takich sygnałów zegarowych.

Multipleksery na wejściu przerzutników służących do bramkowania danych wejściowych są automatycznie zastępowane przez kompilator Quartus poprzez przypisanie sygnału strobuującego do wejścia ENA każdego przerzutnika.



RYS. 11.20. Mapa technologiczna po dopasowaniu projektu reg_sync_AND

11.4.4. Badanie metod bramkowania sygnałów zegarowych

Czytelnik może mieć naturalne pytanie: która z rozpatrywanych metod bramkowania sygnałów zegarowych jest najbardziej efektywna w projektach FPGA? Spróbujmy odpowiedzieć na to pytanie.

W tym celu zrealizujemy powyższe projekty rejestrów w Quartus Prime wersji 18.1. Jako parametry syntezy rozważmy liczbę L użytych elementów logicznych FPGA i liczbę R użytych przerzutników FPGA. Aby oszacować parametry czasowe dla wszystkich projektów, utworzymy następujący plik ograniczeń czasowych SDC:

```
create_clock -name clk -period 10.000 [get_ports {clk}]
create_clock -name virt_clk_in -period 10.000
create_clock -name virt_clk_out -period 10.000
derive_pll_clocks -create_base_clocks
derive_clock_uncertainty
set_input_delay -clock { virt_clk_in } -min 0.5 [get_ports { d[*] reset gate }]
set_input_delay -clock { virt_clk_in } -max 3 [get_ports { d[*] reset gate }]
set_output_delay -clock { virt_clk_out } -min 0.5 [get_ports { q[*] }]
set_output_delay -clock { virt_clk_out } -max 3 [get_ports { q[*] }]
```

Rozpatrzmy wartości parametrów czasowych dla najwolniejszego czasowego modelu FPGA z rodziny Cyclone IV E. Wyniki badań przedstawiono w tabeli 11.2, gdzie **Worst-case** – zapas czasowy dla ścieżki najgorszego przypadku; **Setup** – zapas czasowy dla czasu ustawiania; **Hold** – zapas czasowy dla czasu utrzymywania.

TAB. 11.2. Badania wyników syntezy projektów rejestrów z różnymi metodami bramkowania sygnałów zegarowych

Projekt	L	R	Slack		
			Worst-case	Setup	Hold
reg_AND	1	4	0.787	5.680	0.578
reg_AND_GLOBAL	1	4	0.787	5.680	0.578
reg_MUX	0	4	2.182	4.492	0.657
reg_sync_AND	1	5	0.863	1.842	0.625
reg_sync_AND_GLOBAL	1	5	0.863	1.842	0.625
reg_enable	0	4	2.182	4.492	0.657
Best	0	4	2.182	5.680	0.657

Analiza tabeli 11.2 pokazuje, że projekty *reg_MUX* i *reg_enable* mają najniższy koszt realizacji ($L = 0$, $R = 4$) i największy zapas czasowy w najgorszym przypadku **Worst-case** (2,182 ns).

Projekty *reg_AND* i *reg_AND_GLOBAL* używają jednej komórki logicznej do implementacji bramki AND. Te projekty mają najdłuższy zapas czasowy czasu ustawiania **Setup**.

Projekty *reg_sync_AND* i *reg_sync_AND_GLOBAL* z synchronicznym sygnałem zezwolenia wymagają dodatkowego przerzutnika ($L = 1$, $R = 5$) w celu wygenerowania zsynchronizowanego sygnału zezwolenia *enable*, w porównaniu z poprzednimi projektami.

Wnioski. Projekty *reg_MUX* i *reg_enable* wykazały najlepsze wyniki pod względem kosztów realizacji i wydajności.

Do praktycznego zastosowania możemy polecić metodę bramkowania sygnałów zegarowych, zaimplementowaną w projekcie *reg_enable*, jako najprostszą w opisie za pomocą języka Verilog.

11.4.5. Zastosowanie opcji Auto Gated Clock Conversion

Opcja **Auto Gated Clock Conversion** wskazuje kompilatorowi, aby zamiast logiki bramkowania używał wejścia zezwolenia synchronizacji ENA, które w nowoczesnych układach FPGA znajduje się w każdym przerzutniku. W tym przypadku logika bramkowania może zawierać elementy AND, OR, MUX i NOT. Domyślnie opcja **Auto Gated Clock Conversion** w systemie Quartus jest wyłączona. Tabela 11.3 przedstawia wyniki badań rozpatrywanych projektów rejestrów w przypadku włączenia opcji **Auto Gated Clock Conversion**.

TAB. 11.3. Wyniki syntezy rozpatrywanych projektów rejestrów z włączoną opcją **Auto Gated Clock Conversion**

Projekt	L	R	Slack		
			Worst-case	Setup	Hold
reg_AND	0	4	2.182	4.492	0.657
reg_AND_GLOBAL	1	4	0.787	5.680	0.578
reg_MUX	0	4	2.182	4.492	0.657
reg_sync_AND	1	5	2.182	2.699	0.671
reg_sync_AND_GLOBAL	1	5	0.863	1.842	0.625
reg_enabl	0	4	2.182	4.492	0.657
Best	0	4	2.182	5.680	0.671

Realizacja omawianych projektów z włączoną opcją **Auto Gated Clock Conversion** spowodowała zmianę tylko projektów *reg_AND* i *reg_sync_AND*. Projekt *reg_AND* z włączoną opcją **Auto Gated Clock Conversion** daje dokładnie takie same wyniki, jak projekty *reg_MUX* i *reg_enable*. Włączenie opcji **Auto Gated Clock Conversion**

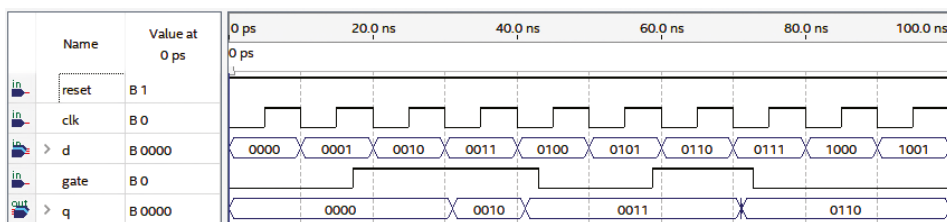
dla projektu *reg_sync_AND* poprawia zapasy czasowe w najgorszym przypadku **Worst-case**, czasu ustawiania **Setup** i utrzymywania **Hold**.

Wniosek. Włączenie opcji **Auto Gated Clock Conversion** w niektórych przypadkach pozwala zwiększyć wydajność projektu.

11.4.6. Bramkowanie sygnałem asynchronicznym

Gdy sygnał bramki *gate* jest synchroniczny (tzn. zmienia się na zboczach sygnału zegarowego), wszystkie projekty działają poprawnie. Rozpatrzmy przypadek asynchronicznego sygnału bramki *gate*. W naszym przykładzie, niech sygnał bramki *gate* będzie wysoki w dwóch przedziałach czasowych: od 17 ns do 42 ns i od 58 ns do 72 ns.

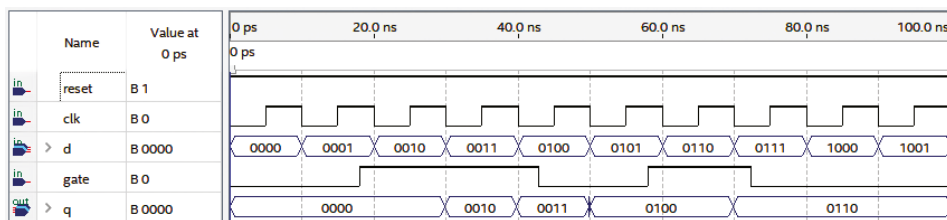
Wyniki symulacji czasowej dla projektów *reg_AND* i *reg_AND_GLOBAL* pokazano na rysunku 11.21.



RYS. 11.21. Wyniki symulacji czasowej dla projektów *reg_AND* i *reg_AND_GLOBAL*

Rysunek 11.21 pokazuje, że projekty *reg_AND* i *reg_AND_GLOBAL* działają poprawnie. Rzeczywiście, zmiany wartości wyjścia *q* zachodzą (z pewnym opóźnieniem) przy dodatnim zboczach sygnału zegarowego *clk* i przy wysokim poziomie sygnału bramki *gate*.

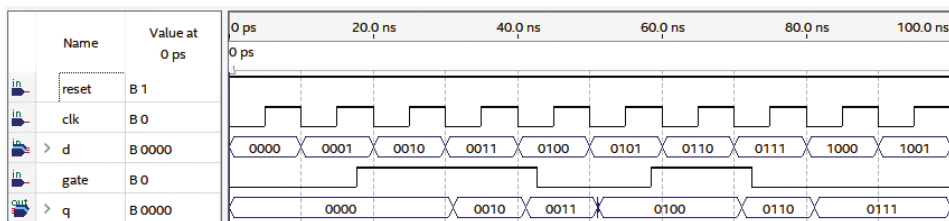
Diagramy czasowe wyników symulacji dla projektów *reg_MUX* i *reg_enable* pokazano na rysunku 11.22.



RYS. 11.22. Diagramy czasowe wyników symulacji dla projektów *reg_MUX* i *reg_enable*

Analiza rysunku 11.22 pokazuje, że w zakresie od 50 ns do 70 ns na wyjściu generowana jest nieprawidłowa wartość 0100.

Symulację czasową projektów *reg_sync_AND* i *reg_sync_AND_GLOBAL* pokazano na rysunku 11.23.



RYS. 11.23. Wyniki symulacji czasowej dla projektów *reg_sync_AND* i *reg_sync_AND_GLOBAL*

Na rysunku 11.23 widać, że na wyjściu generowane są nieprawidłowe wartości: 0100 w zakresie od 50 ns do 70 ns oraz 0111 w zakresie od 82 ns do 100 ns.

Wniosek. W przypadku asynchronicznego sygnału bramki *gate* należy zastosować projekt *reg_AND*, który pozwala na uzyskanie na wyjściach wiarygodnych wyników.

Ogólne wnioski dotyczące bramkowania sygnałów zegarowych są następujące:

- wprowadzenie bufora GLOBAL dla sygnałów zegarowych generowanych przez wewnętrzną logikę FPGA nie wpływa na wyniki syntezy, ponieważ kompilator automatycznie przypisuje sygnały zegarowe do globalnych sieci synchronizacji (opcja **Auto Global Clock** w systemie Quartus jest zawsze włączona);
- do praktycznego zastosowania można polecić metodę przedstawioną w projekcie *reg_enable*, jako najprostszą przy opisywaniu w języku Verilog;
- w niektórych przypadkach włączenie opcji **Auto Gated Clock Conversion** pozwala poprawić parametry czasowe projektów z bramkowanymi sygnałami zegarowymi;
- w przypadku asynchronicznego sygnału strobującego należy zastosować metodę zaimplementowaną w projekcie *reg_AND*.

11.5. Zalecenia dotyczące tworzenia szybkich i złożonych projektów

Ta sekcja zawiera wskazówki od firmy Intel podczas projektowania podsystemów synchronizacji dla szybkich i złożonych projektów.

11.5.1. Planowanie realizacji fizycznej

W przypadku złożonych i szybkich projektów konieczne jest zaplanowanie fizycznej realizacji projektu w docelowego układzie scalonym FPGA. Dlatego też należy wziąć pod uwagę następujące elementy związane z realizacją fizyczną:

- liczba unikalnych domen synchronizacji;
- ilość logiki w każdym bloku funkcjonalnym;
- lokalizacja i kierunek przepływu danych między blokami;
- w jaki sposób dane są trasowane do bloków funkcjonalnych z interfejsów wejścia-wyjścia.

Na przykład układ FPGA z rodziny Cyclone V SE 5CSEMA5F31C6N, który jest zainstalowany na płycie DE1-SoC, ma 16 globalnych i 66 regionalnych sieci synchronizacyjnych oraz 85K elementów logicznych. W FPGA z rodziny Cyclone V SE jeden *blok macierzy logicznej* (Logic Array Block – LAB) zawiera 10 *adaptacyjnych modułów logicznych* (Adaptive Logic Modules – ALM). Każdy moduł ALM ma 8 wejść i 4 rejestry (przerzutniki). Każdy blok LAB zawiera specjalną logikę do generowania sygnałów sterujących w modułach ALM. W FPGA z rodziny Cyclone V SE możliwe jest użycie dwóch zewnętrznych źródeł sygnałów zegarowych i trzech zewnętrznych źródeł sygnałów zezwolenia synchronizacji.

Pod względem regionalnych sieci synchronizacji FPGA jest podzielona na cztery ćwiartki. Jedna regionalna sieć synchronizacji zarządza jednym *kwadrantem* (quadrants). *Podwójne regionalne* (Dual-Regional) sieci synchronizacji obsługują dwie ćwiartki lub połowę obszaru FPGA. Dlatego też, aby poprawić kontrolę, można zorganizować hierarchię projektu w oddzielne bloki dla każdej strony urządzenia. Bloki projektowe mogą generować i wykorzystywać sygnały sterujące rejestrami po każdej stronie urządzenia. To podejście jest szczególnie skuteczne w projektach o charakterze nadawczo-odbiorczym, gdzie powinno się umieścić układ logiczny po stronie FPGA w pobliżu nadajników/odbiorników.

Komentarz. Podczas korzystania z potokowych sygnałów sterujących należy wyłączyć opcję **Auto Shift Register Replacement**.

Ponadto może wystąpić wyścig między sygnałami sterującymi interfejsu, gdy pochodzą z różnych stron urządzenia. W takim przypadku zaleca się wprowadzenie potokowych stopni rejestrowych (stages), aby sygnały te obejmowały całą szerokość urządzenia.

11.5.2. Planowanie zasobów FPGA

Bloki funkcjonalne projektu powinny być zaplanowane na wczesnym etapie rozwoju z wykorzystaniem odpowiednich globalnych, regionalnych i podwójnych regionalnych sieci synchronizacji.

Po zdefiniowaniu sygnałów zegarowych projektu, globalne sieci synchronizacji należy przypisać do sygnałów sterujących rejestrami o najwyższym współczynniku rozgałęzienia na wyjściu. Pozwala to na zarządzanie dowolnymi rejestrami w całym urządzeniu. Regionalne sieci synchronizacji umożliwiają zarządzanie rejestrami w jednej ćwiartce (quadrant) urządzenia, a podwójne sieci regionalne w połowie układu FPGA.

W zależności od przepływu danych i fizycznej lokalizacji wejść i wyjść bloku funkcjonalnego projektu, blok taki jest przypisywany do określonej ćwiartki lub określonej połowy układu.

Opracowując plan obszaru w układzie scalonym, należy wziąć pod uwagę bilans zasobów układu: ilość bloków logicznych, pamięci i DSP dla głównych bloków funkcjonalnych projektu. Logika, która współdziała z pamięcią, musi przejść przez cały

układ. W takim przypadku konieczne jest zastosowanie dostatecznie dużej liczby stopni rejestrów w ścieżkach danych i sterujących, aby uzyskać w odpowiednim czasie dostęp do niezbędnych zasobów fizycznych układu.

11.5.3. Spełnienie ograniczeń czasowych

W przypadku złożonych i szybkich projektów spełnienie wszystkich ograniczeń czasowych projektu może być trudne. Dlatego komunikaty analizatora czasowego TimeQuest systemu Quartus mają kolor czerwony w oknie **Table of Contents**. Aby tego uniknąć, firma Intel zaleca:

- wszędzie tam, gdzie w projekcie nie jest wymagana analiza czasu pojedynczego cyklu, należy zastosować *ograniczenia wielocyklowe* (multicycle constraints) [9];
- zastosować ograniczenie „*falszywe ścieżki*” (False Path) [9] do wszystkich asynchronicznych przecięć domen zegarowych lub sygnałów resetowania; poprawia to wyniki synchronizacji i skraca czas kompilacji;
- jeśli to możliwe, zachować domyślne wartości opcji kompilatora; pozwala to skrócić czas kompilacji;
- aby znaleźć najlepsze rozwiązanie, skompilować projekt z co najmniej trzema różnymi ograniczeniami czasowymi lub ustawieniami kompilatora.

11.6. Optymalizacja podsystemu synchronizacji

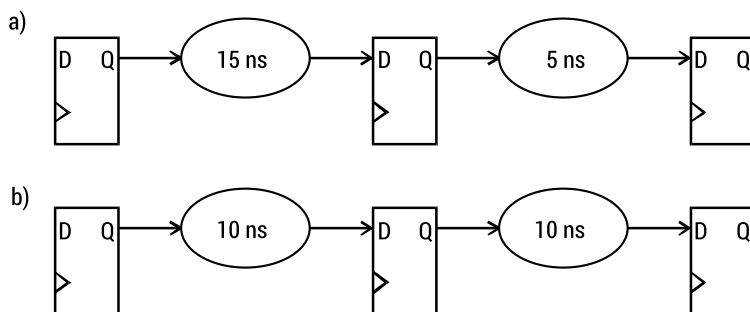
Aby zoptymalizować synchronizację w projektach FPGA, stosowane są następujące metody: *retiming rejestrów* (register retiming), *duplikacja rejestrów* (register duplication) i *synteza fizyczna logiki kombinacyjnej* (physical synthesis for combinational logic). Każda z tych metod może poprawić wydajność projektu.

11.6.1. Retiming

Jak wiadomo, każdy system cyfrowy na poziomie RTL można przedstawić jako zbiór rejestrów, pomiędzy którymi znajdują się fragmenty logiki.

O czasie trwania cyklu synchronizacji decyduje najdłuższe opóźnienie sygnału przechodzącego przez fragmenty logiczne. Istotą retimingu rejestrów jest ponowne przypisanie logiki między rejestrami w taki sposób, aby zmniejszyć maksymalne opóźnienie sygnału propagowanego przez największy fragment logiczny (rys. 11.24).

Retiming rejestrów jest najbardziej efektywny w przypadku projektów potokowych. Należy zauważyć, że podczas retimingu rejestrów, nowe rejestry nie są wprowadzane do projektu, a jedynie logika jest redystrybuowana pomiędzy fragmentami.

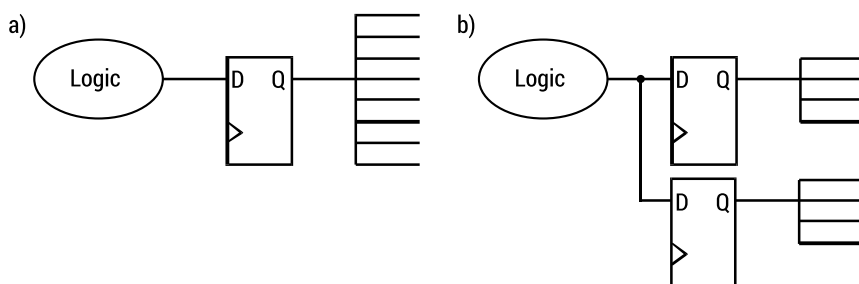


RYS. 11.24. Retiming rejestrów na poziomie RTL: a – przed; b – po

11.6.2. Duplikacja rejestrów

Niektóre rejestry projektów mogą mieć duży współczynnik *fan-out* rozgałęzienia wyjścia. Może to prowadzić do konieczności wprowadzenia dodatkowych buforów do wzmocnienia sygnału. Dodatkowo poszczególne punkty logiki sterowane przez wyjście rejestru mogą znajdować się daleko od źródła, powodując dodatkowe opóźnienia.

Duplikacja rejestrów (duplicated registers) polega na wprowadzeniu do projektu dodatkowych rejestrów w celu zmniejszenia dla poszczególnych rejestrów współczynnika rozgałęzienia wyjścia (rys. 11.25). Często duplikacja rejestrów prowadzi do przyspieszenia całego projektu.



RYS. 11.25. Duplikacja rejestrów: a – przed; b – po

11.6.3. Synteza fizyczna logiki kombinacyjnej

Zwykle synteza logiczna projektu jest wykonywana przez program zwany Syntezatorem, który jest częścią kompilatora.

Po syntezie logicznej należy przeprowadzić syntezę fizyczną, która polega na *umieszczeniu* (placing) elementów obwodu i *trasowaniu* (routing) połączeń

przewodowych między nimi. Synteza fizyczna jest wykonywana przez program o nazwie Fitter, który jest również częścią kompilatora.

Opóźnienia trasowania mogą być znaczące i mogą przekraczać opóźnienia elementów logicznych. Dlatego wydajność i koszt projektu w dużej mierze zależą od skutecznego rozwiązania problemów związanych z rozmieszczeniem i trasowaniem. Na przykład silnie powiązana logika może być umieszczona blisko siebie lub zaimplementowana w pojedynczym bloku funkcjonalnym FPGA.

Synteza fizyczna jest kontrolowana przez opcje Fittera (omówione poniżej), które optymalizują koszty realizacji i wydajność projektu.

11.6.4. Opcje systemu Quartus wpływające na optymalizację synchronizacji

Użytkownik systemu Quartus może wpływać na metody optymalizacji synchronizacji za pomocą opcji Syntezatora i Fittera. Typową opcją syntezy i fittingu jest opcja **Prevent register optimization** (Zapobiegaj retimingowaniu rejestrów). Domyślnie ta opcja jest wyłączona, tzn. przeprowadzony zostanie retiming rejestrów w celu poprawy wyników projektu.

W tabeli 11.4 podsumowano opcje syntezy, które mają wpływ na synchronizację.

TAB. 11.4. Opcje Syntezatora wpływające na synchronizację

Opcja	Domyślna wartość	Optymalizuje	Opis
Allow Register Duplication (Zezwalaj na duplikację rejestrów)	On	Szybkość	Pozwala na duplikację rejestrów z dużym współczynnikiem <i>fan-out</i>
Allow Register Merging (Zezwalaj na łączenie rejestrów)	On	Koszt (powierzchnia)	Umożliwia łączenie rejestrów generujących tę samą logikę
Allow Synchronous Control Signals (Zezwalaj na synchroniczne sygnały sterujące)	On	Koszt	Umożliwia użycie synchronicznego resetowania (<i>synchronous clear</i>) i/ lub synchronicznego ładowania (<i>synchronous load</i>) w normalnym trybie komórek logicznych
Force Use of Synchronous Clear Signals (Użyj synchronicznych sygnałów resetowania)	Off	Koszt	Nakazuje kompilatorowi użycie synchronicznych sygnałów resetowania (<i>synchronous clear</i>) w normalnym trybie komórek logicznych
Remove Duplicated Registers (Usuń duplikację rejestrów)	On	Koszt	Usuwa rejestr, jeśli jest identyczny z innym rejestrem
Timing-Driven Synthesis (synteza sterowana czasem)	On	Szybkość i koszt	Umożliwia kompilatorowi podczas syntezy wykorzystanie informacji o ograniczeniach czasowych w celu lepszej optymalizacji projektu

Opcja **Allow Register Duplication** (Zezwalaj na duplikację rejestrów) umożliwia użycie metody duplikacji rejestrów w celu poprawy wydajności projektu.

W dużych projektach może się zdarzyć, że wyjścia różnych rejestrów pełnią te same funkcje logiczne. W takim przypadku dodatkowe rejestry można usunąć, pozostawiając tylko jeden z nich. Opcje **Allow Register Merging** i **Remove Duplicated Registers** służą do kontrolowania procesu łączenia rejestrów.

W układach FPGA firmy Intel resetowanie asynchroniczne i ładowanie rejestrów asynchronicznych może wymagać dodatkowych zasobów logicznych. Aby włączyć użycie synchronicznych sygnałów sterujących, użyj opcji **Allow Synchronous Control Signals** i **Force Use of Synchronous Clear Signals**.

System Quartus ma kolejną interesującą opcję **Timing-Driven Synthesis**, która umożliwia Syntezatorowi wykorzystanie ograniczeń czasowych projektu w celu jego optymalizacji. Na przykład, jeśli wszystkie ograniczenia czasowe zostaną spełnione, kompilator nie będzie ponownie wykonywać retimingu rejestrów. Jednak, aby użyć informacji o ograniczeniach czasowych, w projekcie należy zdefiniować plik SDC.

Tabela 11.5 przedstawia opcje Fittera, które wpływają na czas i wyniki syntezy fizycznej.

TAB. 11.5. Opcje Fittera wpływające na synchronizację

Opcja	Domyślna wartość	Optymalizuje	Uwagi
Allow Register Duplication (Zezwalaj na duplikację rejestrów)	On	Szybkość	Pozwala na duplikację rejestrów z dużym współczynnikiem <i>fan-out</i>
Allow Register Merging (Zezwalaj na łączenie rejestrów)	On	Koszt (powierzchnia)	Umożliwia łączenie rejestrów generujących tę samą logikę
Auto Global Clock (Automatyczny wybór globalnego sygnału synchronizacji)	On	Szybkość i koszt	Pozwala kompilatorowi wybrać sygnał, który zasila większość wejść zegarowych w przerzutnikach jako zegar globalny
Auto Global Register Control Signals (Automatyczny wybór globalnych sygnałów sterujących rejestrami)	On	Szybkość i koszt	Pozwala kompilatorowi wybrać sygnały, które zasilają większość wejść sterujących przerzutników (z wyłączeniem sygnałów zegarowych) jako sygnały globalne
Auto Packed Registers (Automatyczne pakowanie rejestrów)	Auto	Koszt	Umożliwia kompilatorowi łączenie rejestrów i funkcji kombinacyjnych lub implementowanie rejestrów przy użyciu komórek we/wy, bloków pamięci typu RAM lub bloków DSP zamiast komórek logicznych

Opcja	Domyślna wartość	Optymalizuje	Uwagi
Auto Register Duplication (Automatyczna duplikacja rejestrów)	Auto	Szybkość	Pozwala Fitterowi automatycznie zduplikować rejestry w LAB zawierającym puste komórki logiczne. Powiązana z opcją Logic Cell Insertion – Logic Duplication
Perform Physical Synthesis for Combinational Logic for Fitting (Wykonaj fizyczną syntezę logiki kombinacyjnej dla fittingu)	Off	Koszt	Nieobsługiwane w Quartus Prime Pro Edition
Perform Physical Synthesis for Combinational Logic for Performance (Wykonaj fizyczną syntezę logiki kombinacyjnej w celu wydajności)	Off	Szybkość	Nieobsługiwane w Quartus Prime Pro Edition
Perform Register Duplication for Performance (Duplikacja rejestrów do wydajności)	Off	Szybkość	Nieobsługiwane w Quartus Prime Pro Edition
Perform Register Retiming for Performance (Ponownie zsynchronizuj rejestry pod kątem wydajności)	Off	Szybkość	Nieobsługiwane w Quartus Prime Pro Edition
Physical Synthesis Effort Level (Poziom wysiłku syntezy fizycznej)	Normal	Szybkość	Nieobsługiwane w Quartus Prime Pro Edition
Router Timing Optimization Level (Poziom optymalizacji synchronizacji przez trasowanie)	Normal	Szybkość	Nieobsługiwane w Quartus Prime Pro Edition

Opcje **Allow Register Duplication** i **Allow Register Merging** są podobne do opcji Syntezatora, z tą różnicą że dotyczą one Fittera.

Opcje **Auto Global Clock** i **Auto Global Register Control Signal** instruuja Fitter, aby wybrał sygnał, który zasila większość wejść zegarowych przerzutników i wejścia sygnału sterującego zezwoleniem synchronizacji jako sygnał globalny, tzn. sygnały te mają przypisane globalne lub regionalne sieci synchronizacji.

Opcja **Auto Packed Registers** mówi Fitterowi, aby połączyć razem rejestr i funkcję logiczną podczas implementacji przy użyciu komórek we/wy, bloków pamięci typu RAM lub bloków DSP. Opcja może mieć wartość **Auto**, **Minimize Area**, **Minimize Area with Chains**, **Normal**, **Off**, **Sparse** i **Sparse Auto**. Kiedy **Auto Packed Registers** jest ustawiona na dowolną wartość inną niż **Off**, rejestry są łączone z komórkami we/wy, aby poprawić taktowanie we/wy, a także bloki DSP i RAM w celu zmniejszenia

zajmowanej powierzchni i jeśli to możliwe, poprawy wydajności. Podczas ustawiania opcji na:

- **Auto** (domyślnie) – Fitter stara się osiągnąć najlepszą wydajność na niewielkiej powierzchni;
- **Sparse Auto** – Fitter stara się osiągnąć najwyższą wydajność na jak największej powierzchni;
- **Off** – Fitter nie łączy rejestrów z innymi funkcjami, przy czym powierzchnia projektu może znacznie wzrosnąć bez spełnienia ograniczeń czasowych;
- **Sparse** – Fitter łączy funkcje w sposób poprawiający wydajność w wielu projektach;
- **Normal** – Fitter łączy funkcje, w celu maksymalizacji wydajności i zmniejszenia zajmowanej powierzchni;
- **Minimize Area** – Fitter agresywnie łączy ze sobą funkcje, aby zmniejszyć powierzchnię kosztem wydajności;
- **Minimize Area with Chains** – Fitter jeszcze bardziej agresywnie łączy ze sobą funkcje, które są częścią kaskadowych łańcuchów rejestrów lub które można przekształcić w kaskadowe łańcuchy rejestrów.

Opcja **Auto Register Duplication** pozwala Fitterowi duplikować rejestry w bloku LAB, który ma puste komórki logiczne.

Opcje **Perform Physical Synthesis for Combinational Logic for Fitting** i **Perform Physical Synthesis for Combinational Logic for Performance** nakazują Fitterowi wykonywanie fizycznej syntezy projektu w celu optymalizacji powierzchni i wydajności.

Opcje **Perform Register Duplication for Performance** i **Perform Register Retiming for Performance** nakazują Fitterowi wykonywać duplikowanie i retiming rejestrów w celu zwiększenia wydajności projektu.

Opcje **Physical Synthesis Effort Level** i **Router Timing Optimization Level** pozwalają na osiągnięcie kompromisu między stopniem optymalizacji projektu a czasem kompilacji projektu.

Wniosek. Szereg opcji Syntezatora i Fittera może mieć wpływ na koszt realizacji i wydajność złożonych projektów.

11.6.5. Optymalizacja ścieżek krytycznych

Aby osiągnąć wysoką wydajność projektu, ważne jest przeanalizowanie ścieżek krytycznych, które mają największy wpływ na spełnienie ograniczeń czasowych projektu. Poprawienie choćby jednej takiej ścieżki może doprowadzić do znacznej poprawy synchronizacji.

Błędy synchronizacji (timing failures) sygnałów sterujących przy wysokim współczynniku *fan-out* rozgałęzienia na wyjściu mogą być spowodowane jedną z następujących przyczyn:

- nieoptymalne wykorzystanie globalnych sieci synchronizacji;

- przekroczenie rozmiaru układu fizycznego przy użyciu konwencjonalnych zasobów trasowania bez użycia potoków;
- nieudana korekta dużego współczynnika rozgałęzienia *fan-out* na wyjściu poprzez duplikację rejestrów.

Aby osiągnąć dużą szybkość i przepustowość projektu, konieczne jest zmniejszenie szerokości magistrali i użycia ścieżek. Na przykład niektóre funkcje logiczne można zaimplementować w blokach pamięci FPGA. Pamięć FPGA jest tańsza i bardziej zwarta niż rejestry i zmniejsza zużycie ścieżek.

Wniosek. Aby poprawić wydajność projektu, krytyczne ścieżki sygnałowe powinny zostać zoptymalizowane, na przykład poprzez implementację funkcji logicznych w blokach pamięci wbudowanej FPGA.

11.7. Projektowanie sygnałów resetowania

Oprócz sygnałów taktowania projektanci systemów wbudowanych często stają przed koniecznością zaprojektowania sygnałów sterujących rejestrami. Sygnały sterujące elementami pamięci FPGA można znaleźć w [8].

Wśród sygnałów sterujących szczególną pozycję zajmuje sygnał resetowania. Prawie każdy system wbudowany ma sygnał resetowania, który przywraca system do stanu początkowego. W komputerach osobistych przycisk resetowania znajduje się na przednim panelu jednostki systemowej. Samodzielne systemy wbudowane używają też timera *watchdog* do resetowania systemu.

Wszystkie nowoczesne układy FPGA mają oddzielne wyprowadzenie RESET do asynchronicznego resetowania wszystkich rejestrów. Asynchroniczny sygnał na wyprowadzeniu RESET jest podłączony do wszystkich przerzutników FPGA za pomocą oddzielnej sieci o bardzo niskim opóźnieniu (skew). To wyprowadzenie powinno być używane do ogólnego zerowania systemu. Jednak może być konieczne zresetowanie tylko części systemu lub oddzielnego bloku funkcyjnego. W tej sytuacji możliwe są następujące typy resetowania:

- *reset synchroniczny* (Synchronous Reset);
- *reset asynchroniczny* (Asynchronous Reset);
- *synchronizowany reset asynchroniczny* (Synchronized Asynchronous Reset).

11.7.1. Synchroniczne resetowanie rejestrów

Opisując rejestr w języku Verilog, tradycyjnie stosuje się dwie metody resetowania rejestrów: synchroniczną i asynchroniczną. Synchroniczne resetowanie rejestrów zapewnia, że projektowany układ jest całkowicie synchroniczny. W obwodzie synchronicznym wartości *zapasu czasowego* (slack) można łatwo określić za pomocą

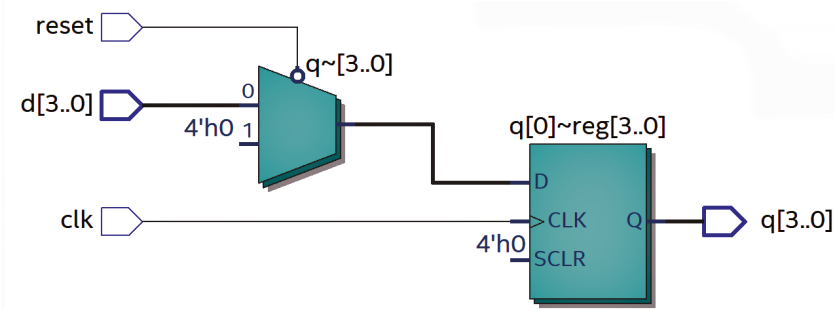
analizatora czasowego TimeQuest. W języku Verilog rejestr z synchronicznym resetowaniem można opisać w następujący sposób:

```
module sync_reset
#(parameter N=4)
(input [N-1:0] d,
input clk,reset,
output reg [N-1:0] q);

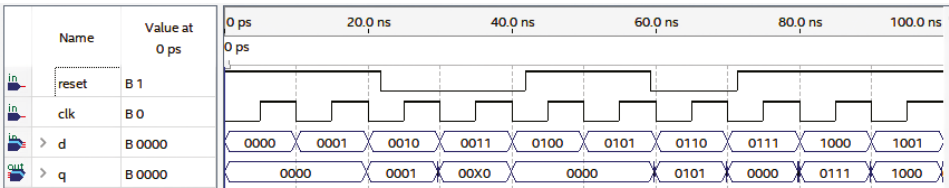
always @(posedge clk)
    if(~reset) q <= 0;
    else      q <= d;

endmodule
```

W przedstawionym kodzie na liście czułości nie ma sygnału resetowania *reset*, dlatego przy niskiej wartości sygnału *reset* i wraz z nadejściem dodatniego zbocza sygnału zegarowego *clk* wartość rejestru zostanie wyzerowana. Wyniki syntezy i analizy czasowej projektu *sync_reset* przedstawiono na rysunkach 11.26 i 11.27.



RYS. 11.26. Wyniki syntezy projektu *sync_reset* na poziomie RTL



RYS. 11.27. Wyniki symulacji czasowej projektu *sync_reset*

W projekcie *sync_reset* na wejściu danych rejestru kompilator automatycznie ustawia multiplexer sterowany przez inwersję sygnału *reset* w celu wyboru wartości zerowej lub danych z magistrali *d* (rys. 11.26). W rozważanym przykładzie symulacji zastosowano dwa przedziały aktywności sygnału resetowania *reset*: od 22 ns do 42 ns i od 58 ns do 72 ns.

Analiza rysunku 11.27 pokazuje, że rejestr jest kasowany ze znacznym opóźnieniem (18,1 ns i 10,4 ns) od momentu wystąpienia niskiego poziomu na wejściu *reset*. Średni czas opóźnienia wynosi 14,25 ns. Ponadto w czasie 30 ns wyjście rejestru jest ustawione na 00x0, co wskazuje na błąd rejestru. Dlatego ta metoda resetowania rejestru może być stosowana tylko wtedy, gdy sygnał resetowania *reset* jest zewnętrznie powiązany z sygnałem zegarowym, tzn. zmiana sygnału *reset* nie narusza czasu ustawiania i czasu utrzymywania rejestru.

Wejścia rejestrów mogą być również bramkowane za pomocą bramek AND, jak pokazano w poniższym kodzie:

```
module sync_reset_AND
#(parameter N=4)
(input [N-1:0] d,
input clk,reset,
output reg [N-1:0] q);

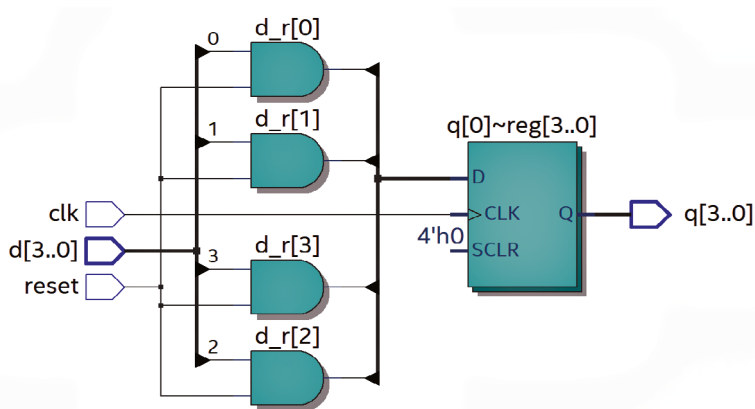
wire [N-1:0] d_r;

assign d_r = d & {N{reset}};

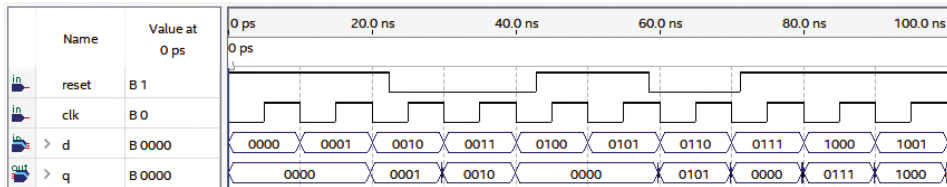
always @(posedge clk)
    q <= d_r;

endmodule
```

Wyniki syntezy i symulacji czasowej projektu *sync_reset_AND* przedstawiono na rysunkach 11.28 i 11.29.



RYS. 11.28. Wyniki syntezy projektu *sync_reset_AND* na poziomie RTL



RYS. 11.29. Wyniki symulacji czasowej projektu *sync_reset_AND*

Na rysunku 11.28 można zauważyć, że bramka AND jest ustawiona na wejściu każdego przerzutnika, aby bramkować dane wejściowe za pomocą sygnału resetowania *reset*. Resetowanie rejestru (rys. 11.29) również następuje ze znacznym opóźnieniem, średni czas opóźnienia wynosi 14,46 ns. Jednak w czasie 30 ns na wyjściu generowana jest prawidłowa wartość 0010.

Sygnał resetowania *reset* można zsynchronizować, tzn. połączyć z sygnałem zegarowym, wewnątrz FPGA, tak jak w następującym projekcie:

```

module sync_reset_int
#(parameter N=4)
(input [N-1:0] d,
input clk,reset,
output reg [N-1:0] q);

reg reg_a, reg_b;
wire rst_n;

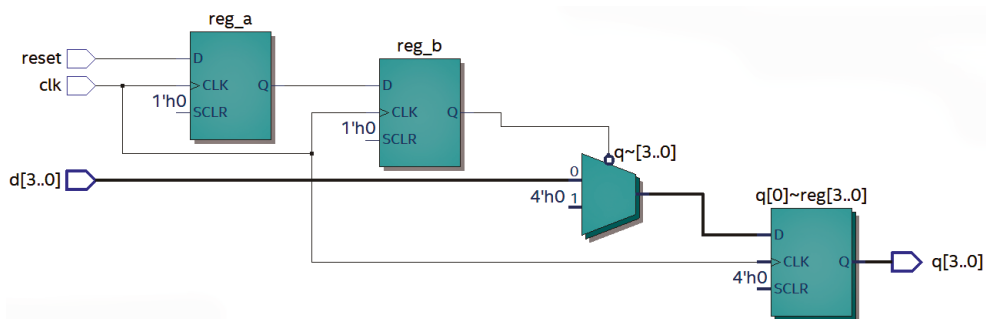
assign rst_n = reg_b;

always @(posedge clk)
    if(~rst_n) q <= 0;
    else      q <= d;

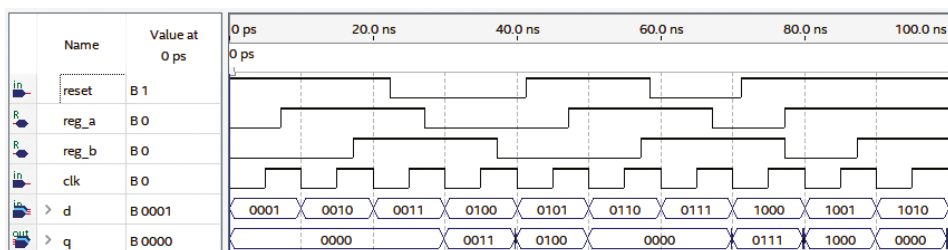
always @(posedge clk)
begin
        reg_a <= reset;
        reg_b <= reg_a;
end
endmodule

```

Wyniki syntezy i symulacji czasowej projektu *sync_reset_int* przedstawiono na rysunkach 11.30 i 11.31.



RYS. 11.30. Wyniki syntezy projektu *sync_reset_int* na poziomie RTL



RYS. 11.31. Wyniki symulacji czasowej projektu *sync_reset_int*

Na rysunku 11.30 widać, że za pomocą szeregowo połączonych przerzutników *reg_a* i *reg_b* sygnał resetowania *reset* jest synchronizowany z sygnałem zegarowym *clk*. Jednak nowa wartość na wyjściu przerzutnika *reg_a* ustawiana jest na wyjściu z opóźnieniem jednego taktu synchronizacji, a na wyjściu przerzutnika *reg_b* – dwóch taktów synchronizacji. Z rysunku 11.31 wynika, że opóźnienie zerowania rejestru wynosi około 28,6 ns. Dodatkowo, ze względu na to, że rejestry *reg_a* i *reg_b* są kasowane przy włączeniu zasilania, wektory wejściowe 0001 i 0010 nie trafiają na wejścia rejestrów.

11.7.2. Asynchroniczne resetowanie rejestrów

Asynchroniczne resetowanie rejestrów jest najczęstszym i najłatwiejszym sposobem resetowania projektu. Reset asynchroniczny jest niezależny od sygnału zegarowego i gdy tylko pojawi się impuls resetowania, rejestry są zerowane. Zazwyczaj kompilator systemu Quartus automatycznie przypisuje asynchroniczny sygnał resetowania do globalnych sieci synchronizacji. W przypadku asynchronicznego resetu z aktywnym niskim sygnałem resetowania *reset* kod projektu rejestru może wyglądać następująco:

```
module async_reset
#(parameter N=4)
```

```

(input [N-1:0] d,
input clk,reset,
output reg [N-1:0] q);

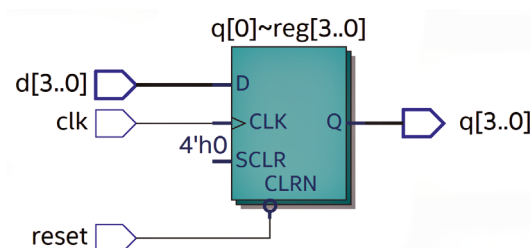
always @(posedge clk, negedge reset)
    if(~reset) q <= 0;
    else      q <= d;

endmodule

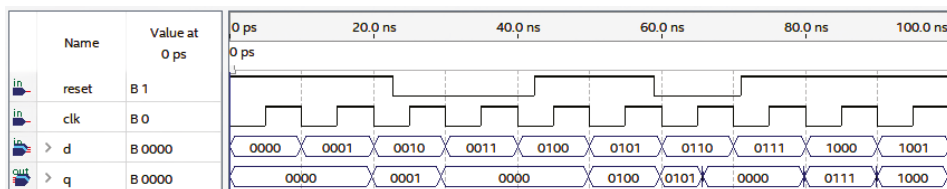
```

Ten kod zapewnia resetowanie (przypisanie wartości zerowych) rejestru na ujemnym zboczach sygnału *reset*, dlatego na liście czułości sygnał *reset* jest zapisany ze słowem kluczowym **negedge**, a w wyrażeniu warunkowym instrukcji **if** sprawdzana jest inwersja sygnału zerowania *reset*.

Wynik syntezy projektu *async_reset* pokazano na rysunku 11.32. Na podstawie wyników symulacji czasowej (rys. 11.33) można wyznaczyć czas opóźnienia od momentu zerowania do wartości zerowych na wyjściach rejestru, który wynosi około 6,58 ns.



RYS. 11.32. Wyniki syntezy projektu *async_reset* na poziomie RTL



RYS. 11.33. Wyniki symulacji czasowej projektu *async_reset*

Jednak asynchroniczny sygnał resetowania w żaden sposób nie pasuje do sygnału zegarowego i może powodować metastabilny stan rejestru, jeśli zostanie przekroczony czas ustawiania t_{SU} lub czas utrzymywania t_H . Aby tego uniknąć, w poniższym projekcie zaleca się stosowanie dodatkowych rejestrów (follower registers) w celu dopasowania danych wejściowych do sygnału synchronizacji.

```

module async_reset_with_follower_registers
#(parameter N=4)
(input [N-1:0] d,
input clk,reset,
output [N-1:0] q);

reg [N-1:0] reg1, reg2, reg3;

assign q = reg3;

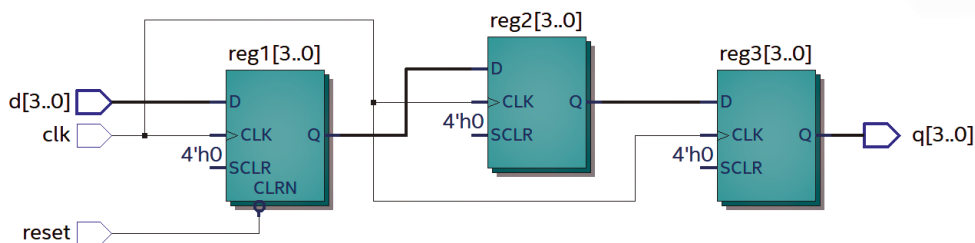
always @(posedge clk, negedge reset)
    if(~reset) reg1 <= 0;
    else      reg1 <= d;

always @(posedge clk)
begin
    reg2 <= reg1;
    reg3 <= reg2;
end
endmodule

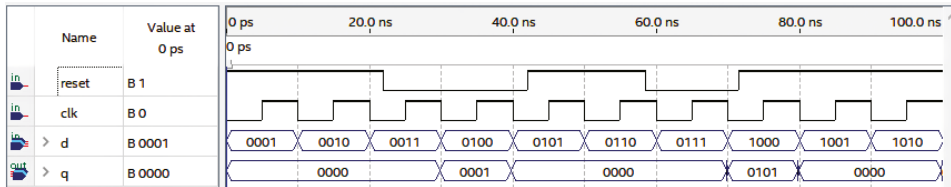
```

Trzy rejestry są tutaj połączone szeregowo: reg1, reg2 i reg3. Tylko pierwszy rejestr, reg1, jest resetowany asynchronicznie. Dane między rejestrami są przesyłane wyłącznie za pomocą sygnału synchronizacji *clk*.

Wyniki syntezy i symulacji projektu *async_reset_with_follower_registers* przedstawiono na rysunkach 11.34 i 11.35. Pomimo tego, że projekt ten wykorzystuje trzy razy więcej przerzutników w porównaniu do *async_reset*, rozwiązuje on problem metastabilności rejestrów.



RYS. 11.34. Wyniki syntezy projektu *async_reset_with_follower_registers* na poziomie RTL



RYS. 11.35. Wyniki symulacji czasowej projektu *async_reset_with_follower_registers*

Na rysunku 11.35 widać, że wartości na wyjściu *q* zmieniają się wraz ze zmianą sygnału zegarowego, ale z opóźnieniem dwóch cykli zegarowych. W fizycznej realizacji projektu *reg1*, *reg2* i *reg3* muszą być umieszczone blisko siebie, aby zmniejszyć opóźnienia trasowania podczas przesyłania sygnałów między rejestrami.

Aby uniknąć problemów związanych z czysto synchronicznymi i czysto asynchronicznymi sygnałami zerowania, w projektach na FPGA wykorzystuje się *zsynchronizowany reset asynchroniczny* (synchronized asynchronous reset), który łączy zalety resetowania synchronicznego i asynchronicznego. Ta metoda akceptuje asynchroniczny sygnał resetowania *reset*, który jest następnie powiązany z sygnałem zegarowym *clk* przy użyciu dwóch przerzutników (*reg1* i *reg2*). W rezultacie generowany jest sygnał *rst_n*, który służy do zerowania rejestru.

Poniżej znajduje się kod rejestru z synchronizowanym resetem asynchronicznym.

```

module sync_async_reset
#(parameter N=4)
(input [N-1:0] d,
input clk,reset,
output reg [N-1:0] q);

reg reg1, reg2;
wire rst_n;

assign rst_n = reg2;

always @(posedge clk, negedge reset)
    if(~reset)
    begin
        reg1 <= 0;
        reg2 <= 0;
    end
    else
    begin
        reg1 <= 1'b1;

```

```

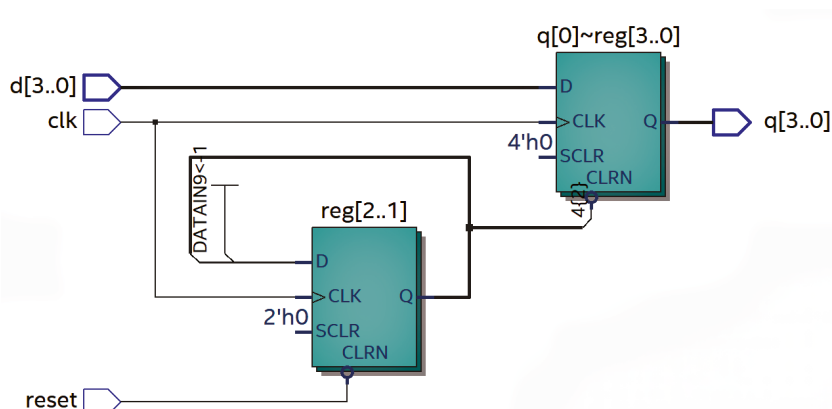
        reg2 <= reg1;
    end

always @(posedge clk, negedge rst_n)
    if(~rst_n) q <= 0;
    else      q <= d;

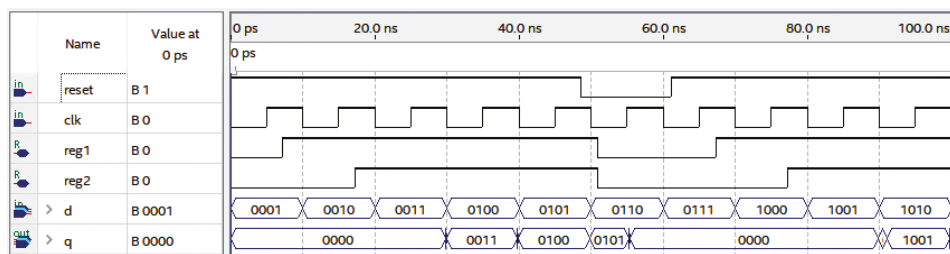
endmodule

```

Wyniki syntezy i symulacji czasowej projektu *sync_async_reset* przedstawiono na rysunkach 11.36 i 11.37.



RYS. 11.36. Wyniki syntezy projektu *sync_async_reset* na poziomie RTL



RYS. 11.37. Wyniki symulacji czasowej projektu *sync_async_reset*

Analizując rysunek 11.37 można zauważyć, że ponieważ wszystkie rejestry są resetowane po włączeniu zasilania, w tym reg1 i reg2, dane zaczynają być zapisywane do rejestru dopiero w trzecim cyklu zegarowym *clk* (wartość 0011). Po resecie asynchronicznym w zakresie od 48 ns do 62 ns kolejne dane również zaczynają być zapisywane dopiero na trzecim dodatnim zboczach sygnału zegara *clk* (wartość 1001). Jednak sam reset jest wykonywany po 7,1 ns na pierwszym dodatnim zboczach sygnału zegarowego *clk* (czas 55 ns).

11.7.3. Badania metod resetowania rejestrów

Jak zawsze pozostaje otwarte pytanie: która z rozpatrywanych metod resetowania rejestrów jest najlepsza? Aby odpowiedzieć na to pytanie, zrealizujemy rozpatrywane projekty rejestrów z wykorzystaniem systemu Quartus w wersji 18.1 na układach FPGA z rodziny Cyclone IV E. Wyniki badań przedstawiono w tabeli 11.6, gdzie L i R to odpowiednio: liczba użytych elementów logicznych i przerzutników FPGA; F_{max} – maksymalna częstotliwość pracy; D – przybliżona wartość opóźnienia czasowego pojawienia się wartości zerowych na wyjściu rejestru od momentu pojawienia się poziomu aktywnego (niskiego) na wejściu sygnału resetowania; *Worst-case* – zapas czasowy dla najgorszego przypadku; *Setup* – zapas czasowy czasu ustawiania; *Hold* – zapas czasowy czasu utrzymywania; *Recovery* – zapas czasowy czasu regeneracji; *Removal* – zapas czasowy czasu usuwania.

TAB. 11.6. Wyniki badań sposobów resetowania rejestrów

Projekt	L	R	D	F _{max}	Slack				
					Worst-case	Setup	Hold	Recovery	Removal
async_reset	0	4	6.58	–	2.110	4.344	1.927	4.309	2.322
sync_reset_AND	4	4	14.46	–	2.104	2.861	2.369	–	–
sync_reset_int	6	6	29.14	149.75	2.110	3.322	0.372	–	–
sync_reset	4	4	14.25	–	2.104	2.861	2.369	–	–
async_reset_with_follower_registers	8	12	19.45	468.38	0.845	4.302	0.371	4.206	2.372
sync_async_reset	2	6	7.1	1321.0	0.845	4.629	0.372	7.745	1.631
Best	0	4	6.58	1321.0	2.110	4.629	2.369	7.745	2.372

Analiza tabeli 11.6 pokazuje, że najlepszym sposobem ze względu na koszt realizacji ($L = 0$, $R = 4$) i uzyskania najniższego opóźnienia ($D = 6,58$ ns) jest tradycyjna metoda resetowania asynchronicznego (projekt *async_reset*).

Kolejnymi pod względem kosztu realizacji ($L = 4$, $R = 4$), z mniej więcej tymi samymi wartościami opóźnienia D (14,25 ns i 14,46 ns), są metody resetu synchronicznego (projekty *sync_reset* i *sync_reset_AND*).

Kolejną pod względem kosztów realizacji ($L = 6$, $R = 6$) jest metoda resetu synchronicznego z wewnętrzną synchronizacją sygnału resetu (projekt *sync_reset_int*). Ta metoda ma również największe opóźnienie ($D = 29,14$ ns).

Najwyższy koszt realizacji ($L = 8$, $R = 12$) ma metoda wykorzystująca dodatkowe rejestry (projekt *async_reset_with_follower_registers*). Ta metoda ma również znaczny czas opóźnienia ($D = 19,45$ ns).

Metoda z synchronizacją asynchronicznego sygnału resetowania (projekt *sync_async_reset*) ma stosunkowo niski koszt realizacji ($L = 2$, $R = 6$) i małe opóźnienie ($D = 7,1$ ns).

Wnioski. W przypadku synchronicznego resetowania rejestrów (projekt *sync_reset*) układ jest całkowicie synchroniczny, wartości zapasów czasowych są łatwe do określenia, jednakże rejestry zerowane są z zauważalnym opóźnieniem od momentu pojawienia się aktywnego poziomu sygnału resetowania. Ta metoda może być zalecana w przypadku zewnętrznej synchronizacji sygnału zerowania.

W metodzie bramkowania danych wejściowych do synchronicznych sygnałów resetujących *reset* za pomocą bramek AND (projekt *sync_reset_AND*), rejestry również są zerowane z zauważalnym opóźnieniem. Obie te metody (projekt *sync_reset* i projekt *sync_reset_AND*) mogą być zalecane dla projektów z zewnętrznym zsynchronizowanym sygnałem resetowania.

W przypadku wewnętrznej synchronizacji sygnału resetowania (projekt *sync_reset_int*) wymagane są dwa dodatkowe elementy logiczne, czas potrzebny do wyzerowania rejestrów znacznie się wydłuża, a rejestry nie przyjmują dwóch zestawów danych wejściowych bezpośrednio po włączeniu zasilania.

Tradycyjne asynchroniczne zerowanie rejestru (projekt *async_reset*) jest najłatwiejsze do realizacji; nie wykorzystuje dodatkowych elementów logicznych FPGA, ponieważ sygnał resetowania jest bezpośrednio podłączony do wejścia sterującego CLRN przerzutnika; zerowanie rejestrów następuje z minimalnym opóźnieniem; nie ma problemu z otrzymaniem dwóch zestawów danych wejściowych po włączeniu zasilania. Jednak ten sposób resetowania może prowadzić do metastabilnego stanu rejestrów.

Wykorzystanie rejestrów *follower registers* znacznie zwiększa koszt implementacji rejestru, a także opóźnienie zerowania (projekt *async_reset_with_follower_registers*).

W metodzie zsynchronizowanego resetu asynchronicznego (projekt *sync_async_reset*) koszt implementacji nieznacznie wzrasta, rejestry są zerowane z niewielkim opóźnieniem, ale pozostaje problem z odebraniem dwóch zestawów danych wejściowych bezpośrednio po włączeniu zasilania.

Jak widać z tej analizy, nie ma idealnego sposobu na zerowanie rejestrów. Wybór najbardziej odpowiedniej metody należy do projektanta.

11.8. Atrybuty syntezy wpływające na synchronizację projektu

Podobnie jak opcje syntezy, atrybuty syntezy mogą również wpływać na synchronizację projektu, a ostatecznie na jego wydajność i koszt realizacji. Przyjrzyjmy się atrybutom syntezy, które mogą mieć wpływ na synchronizację projektu.

Atrybut **maxfan** określa maksymalną wartość *współczynnika rozgałęzienia* (fan-out) na wyjściu rejestru lub kombinacyjnej komórki logicznej. Jeśli określona wartość zostanie przekroczona, kompilator zreplikuje komórki lub zduplikuje rejestry, aby spełnić warunek.

Atrybut **preserve** zapobiega wszelkiej optymalizacji rejestru przez kompilator (duplikacji, usunięcia lub konkatencji z innym rejestrem). Ten atrybut jest podobny do atrybutu **keep** dla sieci.

Atrybut **noprune** uniemożliwia kompilatorowi usunięcie lub optymalizację rejestru związane ze współczynnikiem rozgałęzienia na wyjściach *fan-out*.

Atrybut **dont_merge** zapobiega scalaniu rejestru ze zduplikowanym rejestrem.

Atrybut **dont_replicate** zapobiega kopiowaniu rejestru.

Atrybut **dont_retime** zapobiega retimingowi rejestru.

Atrybut **direct_enable** umożliwia skrócenie krytycznej ścieżki projektu poprzez przekształcenie złożonego warunku zezwolenia synchronizacji rejestru. Czasami rejestr ma złożony warunek zezwolenia synchronizacji, którego implementacja zawiera ścieżkę krytyczną. Atrybut **direct_enable** umożliwia skierowanie krytycznej części warunku bezpośrednio do wejścia zezwolenia synchronizacji rejestrów i zaimplementowanie pozostałej części warunku przy użyciu zwykłej logiki. Na przykład w poniższym projekcie rejestru zapas czasowy dla najgorszego przypadku został zwiększony z 0,845 ns do 2,182 ns poprzez przypisanie atrybutu **direct_enable** do e2.

```
module direct_enable
#(parameter N=4)
(input [N-1:0] d,
input clk,reset,
output reg [N-1:0] q);

wire e1;                // zmienne pośrednie
(* direct_enable *) wire e2;
wire e3;

assign e1 = ^d;          // pośrednie wartości zmiennych
assign e2 = |d;
assign e3 = d[0];

always @(posedge clk, negedge reset)
    if(~reset)           q <= 0;
    else
        if(e1 & e2 & e3) q <= d;      // warunek zapisu
        else             q <= q;

endmodule
```

Atrybut **useioff** umożliwia kompilatorowi używanie rejestrów komórek we/wy FPGA zamiast wewnętrznych rejestrów logicznych. W komórkach we/wy FPGA wygodnie jest umieścić rejestry buforów wejściowych i wyjściowych projektu, a także rejestry *aktywowania wyjść* (output enable). Użycie rejestrów komórek we/wy pozwala na zwiększenie wydajności projektu poprzez skrócenie czasu ustawiania, a także *czasu od zbocza sygnału zegara do ustawienia stabilnych danych na wyjściu* (clock-to-output) i *czasu od zbocza sygnału zegarowego do włączenia wyjść* (clock-to-output-enable).

11.9. Wnioski

Aby zapewnić stabilne działanie projektu synchronicznego, należy całkowicie wykluczyć możliwość przejścia dowolnego rejestru w stan metastabilny. Aby to zrobić, sygnały na wejściach danych każdego rejestru muszą spełniać wymagania dotyczące czasu ustawiania i czasu utrzymania rejestru.

W projekcie synchronicznym sprzężenia zwrotne powinny zawsze obejmować rejestry.

W projektach synchronicznych zatrzaski powinny być zastępowane rejestrami, gdy tylko jest to możliwe.

Układy opóźniające w projektach FPGA nie mogą być budowane w postaci łańcuchów inwerterów. W tym celu należy używać rejestrów przesuwających, liczników i dzielników częstotliwości [10].

Projekty na FPGA powinny wykorzystywać łańcuch przerzutników do generowania pojedynczego impulsu, a nie łańcuch inwerterów.

Sygnały wewnętrzne w FPGA są generowane przez bloki PLL na podstawie referencyjnych zewnętrznych sygnałów zegarowych. Nie należy używać do tego wewnętrznej logiki FPGA.

Przed użyciem wyjścia logiki kombinacyjnej jako sygnału zegarowego lub sygnału sterującego rejestrem, należy go uczynić wyjściem rejestrowym, jak pokazano na rysunku 11.5.

Aby podzielić częstotliwość sygnału zegarowego, należy użyć bloków PLL (najlepsze rozwiązanie), liczników lub automatów skończonych z rejestrami na wyjściach. W tym przypadku logika kombinacyjna nie może być umieszczona na wyjściach liczników lub automatów skończonych.

Liczniki kaskadowe nie powinny być używane w projektach FPGA, ponieważ zwiększa to koszt projektu, uniemożliwia analizę czasową i wydłuża czas trwania zakłóceń.

Do multipleksowania sygnałów zegarowych powinny być używane bloki PLL lub *bloki sterowania zegarem* (clock control blocks). Należy zachować ostrożność podczas używania do tego celu multiplekserów.

Włączanie i wyłączanie domeny zegarowej za pomocą oddzielnego sygnału sterującego nazywane jest *bramkowaniem synchronizacji* (clock gating). Sygnały

synchronizacji mogą być bramkowane za pomocą bramki AND (projekt *reg_AND*), multiplekserów na wejściach danych przerzutników (projekt *reg_MUX*), a także za pomocą wejścia przerzutników ENA (projekt *reg_enable*). Projekty *reg_MUX* i *reg_enable* wykazały najlepsze wyniki pod względem kosztów realizacji i wydajności. Do praktycznego zastosowania możemy polecić metodę bramkowania sygnałów zegarowych, zaimplementowaną w projekcie *reg_enable*, jako najprostszą do opisaną w języku Verilog. W przypadku asynchronicznego sygnału strobowującego należy skorzystać z metody zaimplementowanej w projekcie *reg_AND*.

Multipleksery na wejściu przerzutników służące do bramkowania danych wejściowych są automatycznie zastępowane przez kompilator Quartus przez przypisanie sygnału bramkowania do wejścia ENA każdego przerzutnika.

W systemie Quartus sygnały zegarowe generowane przez wewnętrzną logikę są automatycznie przypisywane do wolnych globalnych lub regionalnych sieci synchronizacji. Dlatego nie ma potrzeby przypisywania bufora GLOBAL do takich sygnałów zegarowych.

Włączenie opcji **Auto Gated Clock Conversion** w niektórych przypadkach poprawia wydajność projektu.

W przypadku złożonych projektów o dużej szybkości konieczne jest zaplanowanie fizycznej realizacji projektu w docelowym układzie FPGA. Opracowując plan fizyczny układu, należy wziąć pod uwagę bilans zasobów układu: ilość bloków logicznych, pamięci i DSP dla głównych bloków funkcjonalnych projektu.

Retiming rejestrów (register retiming) polega na ponownym przypisaniu logiki między rejestrami w taki sposób, aby zmniejszyć maksymalne opóźnienie sygnałów przez największy element logiczny. Retiming rejestrów jest najbardziej efektywny w przypadku projektów potokowych.

Duplikacja rejestrów (duplicated registers) polega na wprowadzeniu do projektu dodatkowych rejestrów w celu zmniejszenia współczynnika rozgałęzienia na wyjściu *fan-out* dla poszczególnych rejestrów. Duplikacja rejestrów często zwiększa szybkość całego projektu.

Synteza fizyczna jest wykonywana przez Fitter (program zawarty w kompilatorze) i jest związana z umieszczaniem i trasowaniem elementów projektu. Synteza fizyczna w niektórych przypadkach może poprawić koszt i wydajność złożonych projektów.

Szereg opcji Syntezatora i Fittera może mieć wpływ na koszt realizacji i szybkość działania złożonych projektów.

Aby spełnić wszystkie ograniczenia czasowe, należy postępować zgodnie z zaleceniami przedstawionymi w podrozdziale 11.5.3.

Aby poprawić wydajność projektu, krytyczne ścieżki sygnałowe powinny zostać zoptymalizowane, na przykład poprzez implementację funkcji logicznych w blokach pamięci wbudowanej FPGA.

Zastosowanie sieci synchronizacyjnych z oddzielnymi sygnałami sterującymi pomaga przyspieszyć projekt. Jednak liczba sygnałów taktowania i sygnałów sterujących w projekcie nie powinna przekraczać liczby dostępnych sieci synchronizacji w docelowym układzie FPGA.

Sygnały do resetowania rejestrów powinny być starannie zaprojektowane. W przypadku synchronicznego zerowania rejestrów (projekt *sync_reset*) schemat projektu jest całkowicie synchroniczny, wartości zapasów czasowych są łatwe do określenia, jednakże rejestry resetowane są z zauważalnym opóźnieniem od momentu pojawienia się aktywnego poziomu sygnału zerowania. W metodzie bramkowania danych wejściowych synchronicznym sygnałem resetu z wykorzystaniem bramek AND (projekt *sync_reset_AND*), zerowanie rejestrów również odbywa się z zauważalnym opóźnieniem. Obie te metody (projekt *sync_reset* i projekt *sync_reset_AND*) mogą być zalecane dla projektów z zewnątrz zsynchronizowanym sygnałem resetowania.

W przypadku wewnętrznej synchronizacji sygnału resetowania (projekt *sync_reset_int*) wymagane są dwa dodatkowe elementy logiczne, czas potrzebny do wyzerowania rejestrów znacznie się wydłuża, a rejestry nie przyjmują dwóch wektorów danych wejściowych bezpośrednio po włączeniu zasilania.

Tradycyjne asynchroniczne resetowanie rejestru (projekt *async_reset*) jest najłatwiejsze do realizacji; nie wykorzystuje elementów logicznych FPGA, ponieważ sygnał resetowania jest bezpośrednio podłączony do wejść sterujących CLRN przerzutników; zerowanie rejestrów następuje z minimalnym opóźnieniem; nie ma problemu z przyjęciem dwóch wektorów danych wejściowych po włączeniu zasilania. Jednak ten sposób resetowania może prowadzić do metastabilnego stanu rejestrów.

Wykorzystanie rejestrów *follower registers* znacznie zwiększa koszt implementacji rejestru, a także opóźnienie resetowania (projekt *async_reset_with_follower_registers*).

W metodzie synchronizowanego resetu asynchronicznego (projekt *sync_async_reset*) koszt implementacji nieznacznie wzrasta, rejestry są zerowane z niewielkim opóźnieniem, ale pozostaje problem z odebraniem dwóch wektorów danych wejściowych bezpośrednio po włączeniu zasilania.

Nie ma idealnego sposobu na resetowanie rejestrów. Wybór najbardziej odpowiedniej metody należy do projektanta.

Atrybuty syntezy, w przeciwieństwie do opcji Syntezatora, można ustawić indywidualnie dla poszczególnych modułów i poszczególnych rejestrów w projektach w języku Verilog. Atrybuty syntezy, jak również opcje Syntezatora, mają wpływ na koszty realizacji i wydajność projektu.

12. Bloki pętli synchronizacji fazowej PLL

Bloki *pętli synchronizacji fazowej* (Phase-Locked Loop – PLL) są wbudowane w strukturę FPGA i konfigurowane razem z FPGA zgodnie z wymaganiami określonymi przez użytkownika. Układy PLL są szeroko stosowane w radiotechnice, telekomunikacji i informatyce. Układy PLL mogą być cyfrowe lub analogowe. Konstrukcje FPGA wykorzystują tylko cyfrowe układy PLL.

Bloki PLL są przeznaczone do generowania wewnętrznych sygnałów zegarowych wysokiej jakości zadaną częstotliwością, współczynnikiem wypełnienia i przesunięciem fazowym w oparciu o referencyjne sygnały zegarowe. Architektury bloków PLL mogą być różne dla różnych rodzin FPGA.

W rozdziale omówiono zasady działania bloków PLL, cechy architektury bloków PLL w układach FPGA z rodziny Cyclone V, a także sposoby konfiguracji bloków PLL w projektach FPGA.

W systemach wbudowanych FPGA bloki PLL są używane do następujących celów:

- generowanie wszystkich niezbędnych sygnałów zegarowych projektu o określonej częstotliwości, współczynniku wypełnienia i przesunięciu fazowym;
- generowanie sygnałów zegarowych wysokich częstotliwości przekraczających częstotliwość zewnętrznego referencyjnego sygnału zegarowego;
- generowanie sygnałów zegarowych dla transceiverów;
- multipleksowanie sygnałów zegarowych;
- przypisywanie sygnałów zegarowych do globalnych i regionalnych sieci synchronizacji.

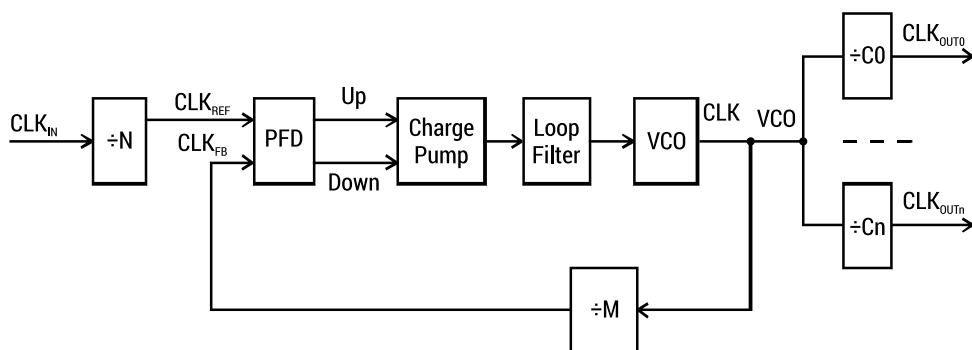
12.1. Zasady funkcjonowania bloku PLL

Blok PLL jest pętlą synchronizowaną fazowo ze sprzężeniem zwrotnym (rys. 12.1), która działa w oparciu o różnicę faz referencyjnego sygnału zegarowego CLK_{REF} i sygnału sprzężenia zwrotnego CLK_{FB} generowanego przez *oscylator sterowany napięciem* (voltage-controlled oscillator – VCO).

Główne elementy struktury blokowej PLL to:

- *licznik preskalera* (Pre-scale Counter) $\div N$;
- *detektor fazy* (phase frequency detector – PFD);

- układ pompy ładunku (Charge Pump), który generuje sygnał analogowy do sterowania generatorem VCO;
- dolnoprzepustowy filtr pętli (Loop Filter);
- generator sterowany napięciem VCO;
- licznik sprzężenia zwrotnego $\div M$;
- liczniki (podzielniki) wyjść postskalera $\div C_0, \dots, \div C_n$.



RYS. 12.1. Uogólniona struktura bloku PLL

Wejście bloku PLL odbiera sygnał zegarowy CLK_{IN} o częstotliwości wejściowej F_{IN} , który jest dzielony przez licznik $\div N$ o wielkości N . W rezultacie powstaje referencyjny sygnał zegarowy CLK_{REF} o częstotliwości F_{REF} .

Detektor fazy PFD porównuje częstotliwość odniesienia F_{REF} z częstotliwością sygnału sprzężenia zwrotnego F_{FB} . Detektor PFD ustawia sygnały *Up* lub *Down*, jeśli częstotliwość sprzężenia zwrotnego F_{FB} opóźnia lub wyprzedza częstotliwość odniesienia F_{REF} . Układ pompy ładunku Charge Pump wykorzystuje sygnały *Up* i *Down* do generowania sygnału analogowego do sterowania VCO. Ponieważ sygnał na wyjściu układu pompy ładunku może charakteryzować się *wahaniami częstotliwości* (jitter) i *zakłóceniami* (glitches), sygnał ten przechodzi przez filtr dolnoprzepustowy pętli Loop Filter.

Generator VCO na podstawie analogowego sygnału wejściowego zwiększa lub zmniejsza częstotliwość, generując sygnał zegarowy CLK_{VCO} o częstotliwości F_{VCO} .

Licznik sprzężenia zwrotnego $\div M$ jest w rzeczywistości powielaczem częstotliwości. Licznik $\div M$ mnoży częstotliwość F_{VCO} sygnału zegarowego CLK_{VCO} przez wartość M i generuje sygnał zegara sprzężenia zwrotnego CLK_{FB} o częstotliwości F_{FB} , który jest podawany przez pętlę sprzężenia zwrotnego na wejście detektora fazy PFD.

Liczniki wyjściowe postskalera $\div C_0, \dots, \div C_n$ dzielą sygnał wyjściowy CLK_{VCO} generatora VCO przez wartość odpowiadającą każdemu licznikowi, tworząc zegary wyjściowe $CLK_{OUT0}, \dots, CLK_{OUTn}$.

W taki oto sposób blok PLL pozwala na utworzenie następujących relacji między częstotliwościami zegarów:

- $F_{REF} = F_{IN} / N$;
- $F_{VCO} = F_{REF} * M = (F_{IN} * M) / N$; (12.1)

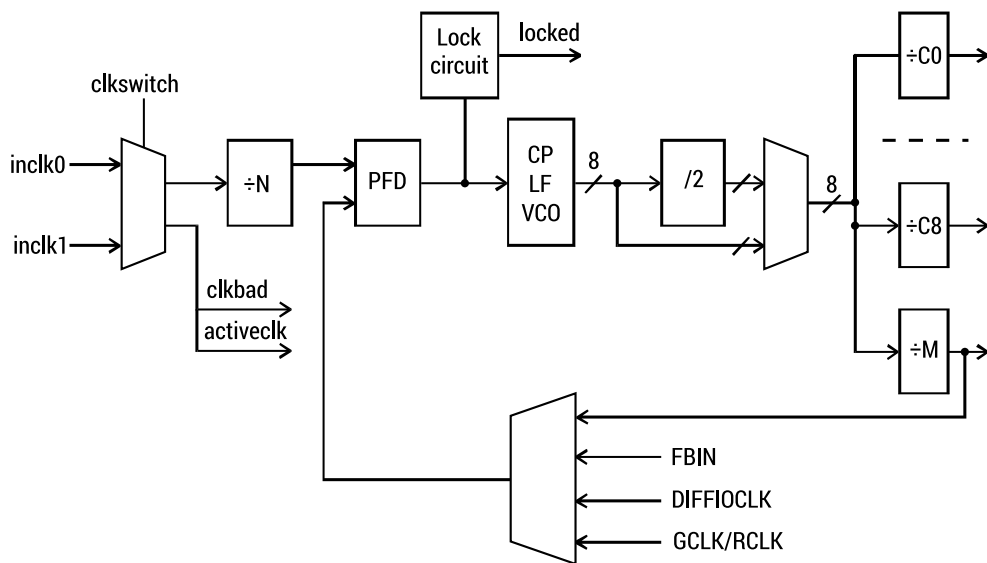
- $$F_{OUTn} = F_{VCO} / C_n = (F_{REF} * M) / C_n = (F_{IN} * M) / (N * C_n)$$

gdzie F_{IN} jest częstotliwością wejściowego sygnału zegarowego; M – wartością licznika sprzężenia zwrotnego $\div M$; N – wartością licznika preskalera $\div N$; C_n – wartością licznika wyjściowego postskalera $\div C_n$.

Bloki PLL układów FPGA firmy Intel wyrównują dodatnie zbocze sygnału zegara odniesienia CLK_{REF} do zbocza sygnału zegara sprzężenia zwrotnego CLK_{FB} . W tym przypadku ujemne zbocza wyjściowych sygnałów zegarowych $CLK_{OUT0}, \dots, CLK_{OUTn}$ są określane przez *współczynnik wypełnienia* (duty-cycle), który jest ustawiany przez użytkownika.

12.2. Specyfika architektury bloków PLL w rodzinie FPGA Cyclone V

Pomimo ogólnych zasad działania, wbudowane bloki PLL każdej rodziny FPGA mają pewne cechy. Aby to przeanalizować, rozpatrzmy implementację wbudowanych bloków PLL w rodzinie Cyclone V (rys. 12.2).



RYC. 12.2. Struktura bloków PLL rodziny FPGA Cyclone V

Bloki PLL FPGA z rodziny Cyclone V (rys. 12.2) mogą mieć maksymalnie dwa wejściowe referencyjne sygnały zegarowe, *inclk0* i *inclk1*, które są wybierane przez *układ wyboru sygnału zegara odniesienia* (reference clock selection circuit). Wejścia *inclk0* i *inclk1* mogą być sterowane z: dedykowanych zewnętrznych wejść zegarowych FPGA, globalnych lub regionalnych sieci synchronizacji, sąsiedniego bloku PLL w celu

utworzenia kaskadowego PLL i dedykowanego wyprowadzenia *refclk*. Wybór między sygnałami zegarowymi odniesienia może być dokonany za pomocą sygnału przełączania lub automatycznie, jeśli jeden z zewnętrznych generatorów sygnałów zegarowych przestanie działać. Układ wyboru sygnału zegara odniesienia generuje sygnał *activeclk*, który wskazuje aktywne zbocze sygnału zegarowego odniesienia (0 – *inclk0*, 1 – *inclk1*), jak również sygnał *clkbad*[1:0], który wskazuje na awarię zegara odniesienia.

Wyjście detektora fazy PFD jest dodatkowo połączone z *układem blokady* (Lock circuit), który generuje sygnał blokady *locked*. Sygnał blokady *locked* wskazuje punkt w czasie, w którym fazy sygnału zegara odniesienia i zegara sprzężenia zwrotnego są wyrównane przez PLL.

Generator VCO generuje 8 wyjść zegarowych, których częstotliwość można podzielić na dwa przez element /2.

Wartości wyjść generatora VCO podawane są na wejścia dziewięciu liczników $\div C0, \dots, \div C8$ i licznika sprzężenia zwrotnego $\div M$. W przeciwieństwie do rysunku 12.1, wyjście licznika sprzężenia zwrotnego $\div M$ można podłączyć do portu wyjściowego bloku PLL.

Porty wyjściowe bloku PLL można podłączyć do:

- globalnych i regionalnych sieci synchronizacji (GCLK i RCLK);
- sąsiedniego bloku PLL w celu utworzenia kaskadowego obwodu PLL;
- zewnętrznego wyjścia synchronizacji;
- portu wyjściowego FBOUT w celu utworzenia zewnętrznej pętli sprzężenia zwrotnego;
- styków Tx Serial Clock i Tx Load Enable transceiverów;
- bloku DLL interfejsu pamięci zewnętrznej;
- linii zegarowych *bloków physical medium attachment* (PMA).

W blokach PLL FPGA z rodziny Cyclone V sprzężenie zwrotne może być wewnętrzne lub zewnętrzne. Wewnętrzne sprzężenie zwrotne bloku PLL łączy detektor fazy PFD z wyjściem licznika $\div M$ i jest używane w *trybie bezpośredniej kompensacji* (Direct Compensation Mode). Zewnętrzne sprzężenie zwrotne PLL jest tworzone przez podłączenie wejścia detektora fazy PFD do jednego z portów wejściowych PLL: FBIN, DIFFIOCLK lub GCLK/RCLK.

Port wejściowy FBIN łączy się z portem wyjściowym FBOUT w celu zapewnienia sprzężenia zwrotnego w *trybie zewnętrznego sprzężenia zwrotnego* (External Feedback Mode) i *trybie bufora zerowego opóźnienia* (zero delay buffer – ZDB Mode). Port DIFFIOCLK jest używany w *trybie kompensacji LVDS* (LVDS Mode). Port GCLK/RCLK służy do dostarczania informacji zwrotnych w *trybie synchronizacji źródłowej* (Source Synchronous Mode) i *trybie normalnej kompensacji* (Normal Compensation Mode). Tryby zegarowe sprzężenia zwrotnego zostaną omówione w sekcji 12.4.

12.3. Cechy funkcjonowania bloków PLL w układach FPGA z rodziny Cyclone V

Bloki PLL rodziny Cyclone V zapewniają syntezę i zrównoważone sterowanie: wewnętrznym sygnałem zegara FPGA, zewnętrznym zegarem systemowym oraz szybkimi interfejsami we/wy. PLL rodziny Cyclone V to ułamkowe bloki, które mogą funkcjonować jako *ułamkowe PLL* (fractional PLL – fPLL) lub *całkowite PLL* (integer PLL). W trybie pracy bloku PLL na liczbach całkowitych wszystkie liczniki mogą przyjmować tylko wartości całkowite. W trybie ułamkowym wartość licznika sprzężenia zwrotnego $\div M$ może przyjmować wartość ułamkową. W rezultacie wartości częstotliwości generowanych wyjściowych sygnałów zegarowych mogą być dokładniej określone.

Liczniki $\div M$, $\div N$ i $\div C_n$ mogą mieć zakres od 1 do 512, tzn. przy pojedynczym PLL częstotliwość wejściowa F_{IN} może zostać zwiększona 512 razy i zmniejszona 262 144 razy. Jeśli te wartości są niewystarczające sąsiednie bloki PLL można łączyć kaskadowo, łącząc port Cascade Output jednego PLL z portem Cascade Input sąsiedniego PLL. Przypomnijmy, że jeden układ FPGA z rodziny Cyclone V SE A5 zawiera 5 bloków PLL.

Każdy blok PLL rodziny Cyclone V ma dwa sygnały sterujące: reset asynchroniczny *areset* (wejście) i wykrywanie blokady *locked* (wyjście).

Uzyskanie synchronizacji dla zdefiniowanej przez użytkownika zależności fazowej i częstotliwościowej między wejściowymi i wyjściowymi sygnałami zegarowymi PLL nazywane jest *trzymaniem – blokadą* (lock). W przypadku rozbieżności między tymi relacjami mówi się, że PLL *stracił blokadę* (lose-of-lock).

Po ustawieniu sygnału resetowania *areset* liczniki i wyjścia PLL są resetowane, a VCO powraca do swoich nominalnych ustawień. Gdy sygnał resetowania *areset* przejdzie w stan niski, blok PLL ponownie synchronizuje swoje wyjścia z zegarem wejściowym, tzn. wykonuje blokadę. Sygnał *areset* musi być ustawiony za każdym razem, gdy PLL traci blokadę, aby zapewnić prawidłową zależność fazową między wejściowymi i wyjściowymi sygnałami zegarowymi PLL.

Sygnał detekcji blokady *locked* wskazuje punkt w czasie, w którym zegar sprzężenia zwrotnego jest wychwytywany na zegarze odniesienia, zarówno w fazie, jak i częstotliwości, tzn. blok PLL generuje wyjściowe sygnały synchronizacji w pełnej zgodności z określonymi wymaganiami.

Bloki PLL z rodziny Cyclone V umożliwiają generowanie wyjściowych sygnałów zegarowych z określonym *przesunięciem fazowym* (Phase Shift). Minimalna wartość przesunięcia fazowego wynosi 1/8 okresu częstotliwości VCO. Na przykład, jeśli $F_{VCO} = 1000$ MHz, to minimalne przesunięcie fazowe wynosi $1 \text{ ns} / 8 = 125$ ps.

Gdy blok PLL generuje wiele wyjściowych sygnałów zegarowych na różnych częstotliwościach, częstotliwość F_{VCO} jest ustawiana na najniższą wspólną wielokrotność tych częstotliwości. Na przykład, jeśli trzy wyjścia bloku PLL mają 10, 33 i 66 MHz,

to Quartus ustawia F_{VCO} na 660 MHz, a następnie liczniki postskalera $\div Cn$ zmniejszają tę częstotliwość dla każdego portu wyjściowego [12].

Cykl pracy sygnałów zegarowych generowanych w PLL jest zwykle określany przez *współczynnik wypełnienia* (duty cycle), który jest wyrażony w procentach. Podany współczynnik wypełnienia określa szerokość impulsu w odniesieniu do okresu sygnału zegarowego, który wynosi 100%.

12.4. Tryby synchronizacji sprzężenia zwrotnego

Bloki PLL obsługują następujące *tryby zegara sprzężenia zwrotnego* (feedback clock modes):

- *synchronizacji źródłowej* (Source Synchronous Mode);
- *kompensacji LVDS* (LVDS Compensation Mode);
- *bezpośredniej kompensacji* (Direct Compensation Mode);
- *normalnej kompensacji* (Normal Compensation Mode);
- *bufora z zerowym opóźnieniem* (Zero-Delay Buffer Mode – ZDB Mode);
- *zewnętrznego sprzężenia zwrotnego* (External Feedback Mode).

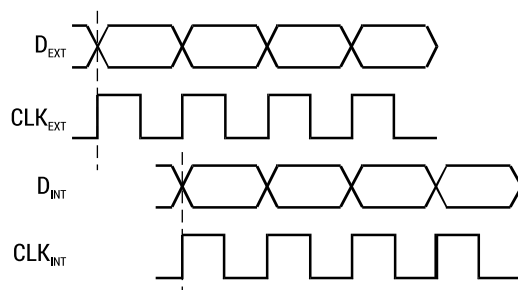
Każdy tryb pozwala na pomnożenie i podzielenie częstotliwości wejściowego sygnału zegara, określenie przesunięcia fazowego oraz ustawienie współczynnika wypełnienia wyjściowych sygnałów zegarowych.

Opóźnienia wejściowe i wyjściowe są w pełni kompensowane przez PLL tylko wtedy, gdy jako źródło sygnału zegarowego używane jest dedykowane zewnętrzne wejście zegarowe FPGA powiązane z tym PLL. Informacja o tym, które zewnętrzne wejścia zegarowe FPGA są powiązane z konkretnym PLL dla każdego układu FPGA, jest umieszczona w dokumentacji technicznej.

Dodatkowo wprowadzimy następujące nazewnictwo: zewnętrzne wyprowadzenia FPGA będą nazywane *wyprowadzeniami* (pins), a wejścia wewnętrznych rejestrów FPGA – *portami* (ports).

12.4.1. Tryb synchronizacji źródłowej

Tryb synchronizacji źródłowej (Source Synchronous Mode) służy do odbierania danych synchronizowanych przez źródło danych. W tym przypadku dane i sygnał zegarowy (D_{EXT} i CLK_{EXT}) docierają do wyprowadzeń wejściowych FPGA w tym samym czasie i muszą być jednocześnie przesyłane z elementów we/wy do portów rejestrów wewnętrznych FPGA (D_{INT} i CLK_{INT}), jak pokazano na rysunku 12.3, gdzie D_{EXT} to dane na wyprowadzeniach FPGA; CLK_{EXT} – referencyjny sygnał zegarowy na wyprowadzeniu FPGA; D_{INT} – dane na porcie wewnętrznego rejestru FPGA; CLK_{INT} – sygnał zegarowy na porcie zegarowym wewnętrznego rejestru FPGA.



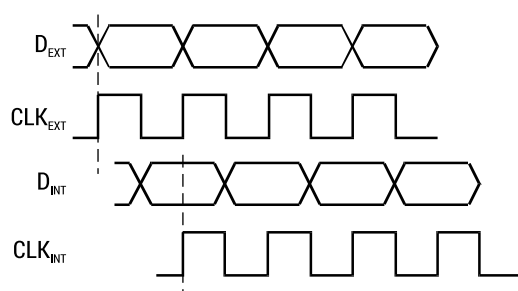
RYS. 12.3. Przykład zależności fazowej sygnałów w trybie synchronizacji źródłowej

W trybie synchronizacji źródłowej bloki PLL mogą kompensować opóźnienia sygnału od wyprowadzeń do wewnętrznych portów rejestru FPGA. Kompensuje to opóźnienie sieci synchronizacji i różnicę w opóźnieniu między dwiema następującymi ścieżkami:

- od wyprowadzenia danych do portu rejestru elementu we/wy (input-output element – IOE);
- od wyprowadzenia sygnału zegara do wejścia detektora PFD bloku PLL.

12.4.2. Tryb kompensacji LVDS

Tryb kompensacji LVDS (LVDS Compensation Mode) utrzymuje tę samą relację między danymi i zegarem (D_{EXT} i CLK_{EXT}) z wyprowadzeń FPGA do portów rejestru przechwytywania konwertera SERDES (D_{INT} i CLK_{INT}), przy czym sygnał zegara CLK_{INT} jest przesunięty w fazie o 180° w stosunku do danych D_{INT} (rys. 12.4).



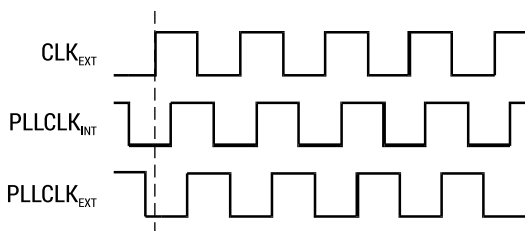
RYS. 12.4. Przykład zależności fazowej sygnałów w trybie kompensacji LVDS

Tryb kompensacji LVDS kompensuje opóźnienie sieci synchronizacji i różnicę opóźnień między następującymi dwiema ścieżkami:

- od wyprowadzenia danych do portu rejestru przechwytywania konwertera SERDES;
- od wyprowadzenia sygnału zegara do portu zegarowego rejestru przechwytywania konwertera SERDES.

12.4.3. Tryb bezpośredniej kompensacji

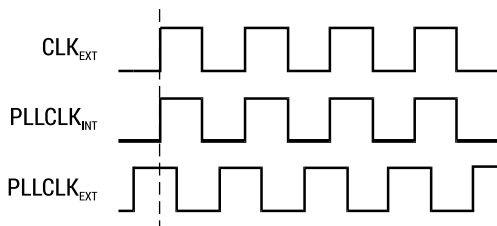
W trybie bezpośredniej kompensacji (Direct Compensation Mode) blok PLL nie kompensuje opóźnień sieci zegarowej. Dlatego też zegary $PLLCLK_{INT}$ i $PLLCLK_{EXT}$ generowane przez blok PLL mają przesunięcie fazowe względem sygnału zegara odniesienia CLK_{EXT} o wielkość opóźnienia trasowania (rys. 12.5), gdzie $PLLCLK_{INT}$ jest sygnałem zegarowym generowanym przez blok PLL na porcie zegarowym wewnętrznego rejestru FPGA; $PLLCLK_{EXT}$ – sygnał taktowania bloku PLL na wyprowadzeniu wyjściowym FPGA.



RYS. 12.5. Przykład zależności fazowej sygnałów w trybie bezpośrednim

12.4.4. Tryb kompensacji normalnej

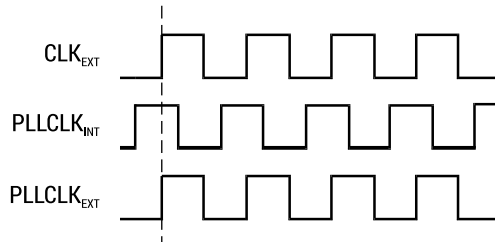
W trybie kompensacji normalnej (Normal Compensation Mode) blok PLL w pełni kompensuje opóźnienia wewnętrznego sygnału zegara $PLLCLK_{INT}$ wprowadzone przez sieci zegarowe GCLK i RCLK. Jednak sygnał zegarowy generowany przez blok PLL na wyprowadzeniu FPGA ($PLLCLK_{EXT}$) ma opóźnienie fazowe w stosunku do referencyjnego sygnału zegarowego CLK_{EXT} (rys. 12.6).



RYS. 12.6. Przykład zależności fazowej sygnałów w normalnym trybie kompensacji

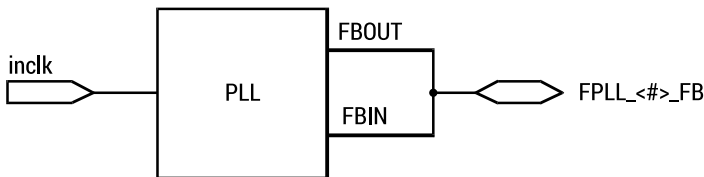
12.4.5. Tryb bufora z zerowym opóźnieniem ZDB

W trybie bufora z zerowym opóźnieniem ZDB (Zero-Delay Buffer), sygnał zegarowy generowany przez blok PLL na wyprowadzeniu wyjściowym PLL ($PLLCLK_{EXT}$) jest wyrównany fazowo z referencyjnym sygnałem zegarowym CLK_{EXT} , podczas gdy wewnętrzne sygnały zegara PLL ($PLLCLK_{INT}$) pozostają niewyrównane fazowo (rys. 12.7).



RYŚ. 12.7. Przykład zależności fazowej sygnałów w trybie bufora z zerowym opóźnieniem ZDB

Aby zapewnić wyrównanie faz w trybie ZDB, port wyjściowy FBOUT bloku PLL musi być podłączony do portu wejściowego FBIN za pomocą wyprowadzenia dwukierunkowego, jak pokazano na rysunku 12.8.

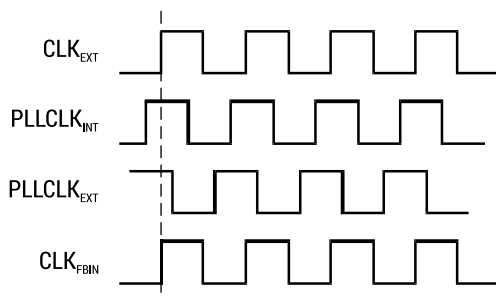


RYŚ. 12.8. Podłączanie portów PLL i wyprowadzeń FPGA w trybie bufora z zerowym opóźnieniem

Idea trybu ZDB polega na kompensacji opóźnienia sygnału podczas przechodzenia przez cały układ FPGA, tzn. sygnał przechodzi przez układ FPGA z „zerowym” opóźnieniem.

12.4.6. Tryb zewnętrznego sprzężenia zwrotnego

W trybie zewnętrznego sprzężenia zwrotnego (External Feedback Mode) blok PLL kompensuje opóźnienie sygnału na podstawie zewnętrznej ścieżki na płycie drukowanej PCB (printed circuit board). Aby to zrobić, port wyjściowy FBOUT bloku PLL łączy się na płycie z portem wejściowym FBIN bloku PLL. W trybie zewnętrznego sprzężenia zwrotnego wyrównuje się fazowo zegar CLK_{FBIN} z referencyjnym zegarem CLK_{EXT} na porcie FBIN bloku PLL. W tym przypadku wewnętrzny zegar $PLLCLK_{INT}$ i sygnał zegara $PLLCLK_{EXT}$ na wyprowadzeniu zewnętrznym bloku PLL pozostają niewyrównane fazowo (rys. 12.9).



RYS. 12.9. Przykład zależności fazowej sygnałów w trybie zewnętrznego sprzężenia zwrotnego

12.5. Ułamkowy i całkowity tryb pracy bloków PLL

Bloki PLL rodziny FPGA Cyclone V obsługują dwa tryby pracy: *ułamkowy* (Fractional) i *całkowity* (Integer). W trybie pracy na liczbach całkowitych struktura bloku PLL odpowiada strukturze z rysunku 12.2. W tym przypadku wartości liczników $\div M$ mogą przyjmować tylko wartości całkowite.

Tryb ułamkowy bloku PLL pozwala na dokładniejsze ustawienie stosunku częstotliwości F_{REF} i F_{FB} . Aby to zrobić, w pętli sprzężenia zwrotnego PLL zawarty jest *modulator sigma-delta* (sigma-delta modulator – SDM), który może tworzyć liczbę ułamkową z sumy liczb całkowitych. Wszystko zależy od tego, jak długo będzie sumowana każda liczba. Na przykład założymy, że chcesz ustawić współczynnik na 4,75. W tym przypadku modulator SDM $\frac{1}{4}$ czasu dodaje wartość 4, a $\frac{3}{4}$ czasu dodaje wartość 5, co daje $4 * (1/4) + 5 * (3/4) = 4,75$.

12.6. Przełączanie zegarów referencyjnych

Blok PLL z rodziny FPGA Cyclone V może mieć jeden lub dwa referencyjne sygnały zegarowe: *inclk0* i *inclk1* (rys. 12.2). Dwa referencyjne sygnały zegarowe w systemach wbudowanych są potrzebne z różnych powodów, na przykład: w przypadku awarii jednego z zewnętrznych generatorów, w celu zmniejszenia poboru mocy w trybie czuwania, do uwzględnienia zmian środowiskowych itp.

Blok PLL rodziny Cyclone V obsługują trzy tryby przełączania zegarów referencyjnych (reference clock switching modes):

- *automatyczny* (Automatic);
- *ręczny* (Manual);
- *automatyczny ze sterowaniem ręcznym* (Automatic switchover with manual override).

W trybie przełączania automatycznego, jeśli aktywne wejście synchronizacji przestanie działać w ciągu trzech cykli sygnału taktującego, sygnał zegarowy referencyjny jest przełączany na *inclk1* lub *inclk0*. Automatyczne rozpoznanie awarii jednego z zewnętrznych generatorów jest możliwe pod warunkiem, że ich częstotliwość różni się nie więcej niż o 20%.

W trybie ręcznego przełączania, jeśli sygnał *clkswitch* przechodzi z niskiego na wysoki i pozostaje wysoki przez trzy cykle taktowania, zegar referencyjny przełącza się z *inclk0* na *inclk1* lub odwrotnie. W trybie automatycznego przełączania z ręcznym sterowaniem, czas odniesienia jest wybierany w taki sam sposób, jak w trybie automatycznym, ale wysoki sygnał przełączający zapobiega przełączaniu wejściowych sygnałów zegarowych.

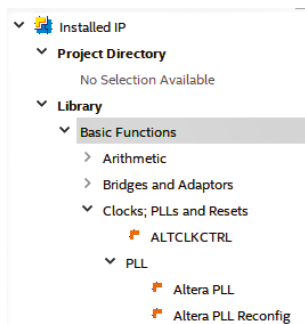
Należy zauważyć, że w przypadku dwóch zegarów referencyjnych, PLL może wykonywać funkcje multipleksowania sygnałów zegarowych.

12.7. Konfigurowanie bloków PLL za pomocą blików IP i Edytora parametrów systemu Quartus

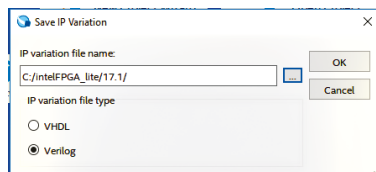
W strukturze FPGA bloki PLL są implementowane sprzętowo. Użytkownik musi jedynie skonfigurować potrzebne bloki PLL zgodnie z określonymi wymaganiami. Bloki PLL są konfigurowane równolegle z konfiguracją FPGA.

Utworzenie wymaganego bloku PLL w projekcie odbywa się za pomocą Katalogu IP i *Edytora parametrów* (Parameter Editor) MegaWizard systemu Quartus, podobnie jak przy tworzeniu modułów pamięci wbudowanej z rozdziału 1.

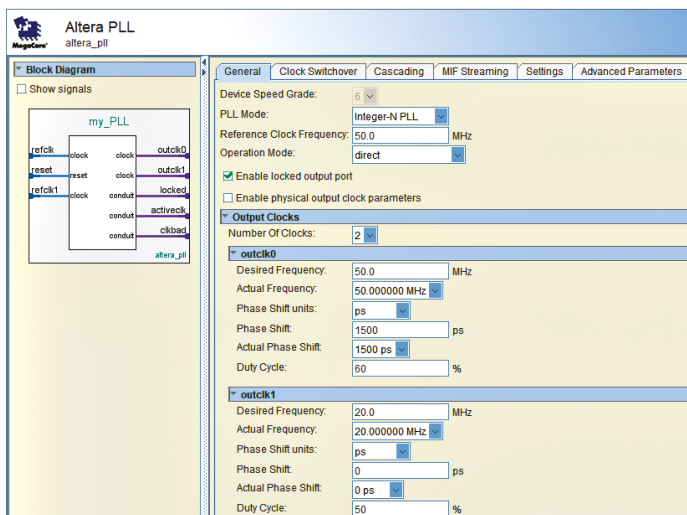
Aby utworzyć blok PLL projektu należy w oknie Katalogu IP wybrać **Library > Basic Function > Clocks; PLL and Resets > PLL > Altera PLL** (rys. 12.10). Po określeniu **Altera PLL** pojawi się okno (rys. 12.11), gdzie należy określić ścieżkę do katalogu na dysku, w którym będą przechowywane pliki projektu oraz język (Verilog lub VHDL) opisu projektu bloku PLL. Po naciśnięciu przycisku OK pojawi się okno Edytora parametrów umożliwiające zdefiniowanie konfiguracji bloku PLL (rys. 12.12).



RYS. 12.10. Okno Katalogu IP podczas tworzenia projektu bloku PLL



RYS. 12.11. Okno do określenia katalogu projektu i języka opisu projektu



RYS. 12.12. Okno Edytora parametrów z aktywną zakładką **General** podczas tworzenia bloku PLL

Okno na rysunku 12.12 zawiera kilka zakładek, z których główną jest zakładka **General**. W zakładce **General** można określić:

- *stopień szybkości układu FPGA* (Device Speed Grade);
- *tryb pracy bloku PLL* (PLL Mode): liczby całkowite (Integer-N PLL) lub ułamkowe (Fractional-N PLL);
- *częstotliwość referencyjnego sygnału zegarowego* (Reference Clock Frequency) w megahercach;
- *tryb pracy sprzężenia zwrotnego* (Operation Mode): bezpośredni (direct), zewnętrzne sprzężenie zwrotne (external feedback), normalny (normal), synchronizacja źródłowa (source synchronous), bufor zerowego opóźnienia (zero delay buffer – ZDB) lub LVDS (lvds).

Ponadto w zakładce **General** można określić opcje:

- uaktywnij port wyjściowy sygnału *locked* (Enable locked output port);
- uaktywnij fizyczne parametry sygnału zegara wyjściowego (Enable physical output clock parameters).

Kolejne pole (Output Clocks) w zakładce **General** określa liczbę zegarów wyjściowych (Number Of Clocks) generowanych przez PLL.

Dla każdego n-tego wyjścia zegarowego *outclk_n* określa się:

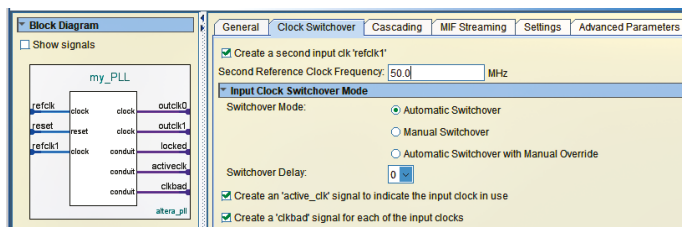
- *żądaną częstotliwość* (Desired Frequency) w megahercach;
- *aktualną częstotliwość* (Actual Frequency), którą wybiera się z listy sugerowanych wartości;
- *jednostki przesunięcia fazowego*: pikosekundy (ps) lub stopnie (degrees);
- *wartość przesunięcia fazowego* (Phase Shift);
- *rzeczywistą wartość przesunięcia fazowego* (Actual Phase Shift), która jest również wybierana z listy sugerowanych wartości;
- *współczynnik wypełnienia* (Duty Cycle) w procentach.

Należy zwrócić uwagę, że proces tworzenia bloku PLL jest wykonywany równoległe z wprowadzaniem parametrów przez użytkownika. Dlatego w lewej części okna Edytora parametrów, można zobaczyć graficzny obraz tworzonego bloku PLL.

W dolnej części Edytora parametrów wyświetlane są komunikaty informacyjne i ostrzegawcze oraz komunikaty o błędach przy wprowadzaniu parametrów.

Dodatkowo w prawym górnym rogu Edytora parametrów znajduje się przycisk Documentation, po którego kliknięciu otwiera się okno, w którym można przeczytać krótkie informacje o każdym parametrze, a także uzyskać link do dokumentacji technicznej.

Kolejna zakładka **Clock Switchover** (rys. 12.13) umożliwia zdefiniowanie parametrów *przełącznika zegara* (Clock Switchover). Na tej karcie można włączyć opcję tworzenia drugiego sygnału zegara referencyjnego *refclk1* (Create a second input clk 'refclk1') i ustawić jego częstotliwość (Second Reference Clock Frequency) w megahercach.



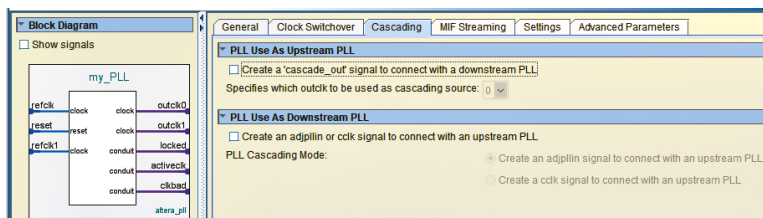
RYŚ. 12.13. Zakładka **Clock Switchover** Edytora parametrów podczas tworzenia bloku PLL

W przypadku tworzenia drugiego referencyjnego sygnału zegarowego konieczne jest zdefiniowanie *trybu pracy przełącznika zegarowego* (Switchover Mode):

- *automatyczny* (Automatic Switchover);
- *ręczny* (Manual Switchover);
- *automatyczny ze sterowaniem ręcznym* (Automatic Switchover with Manual Override).

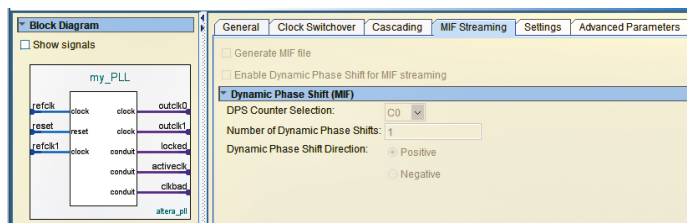
Można również wprowadzić wartość opóźnienia przełącznika (od 0 do 7), ustawić opcję wyjścia *active_clk*, aby wskazać, który zegar wejściowy jest używany, i utworzyć dwubitowy sygnał *clkbad*, który wskazuje niedziałający zegar referencyjny.

Zakładka **Cascading** (rys. 12.14) umożliwia zdefiniowanie portu wejściowego *adjpltn* i portu wyjściowego *cascade_out* dla kaskadowych połączeń bloków PLL.



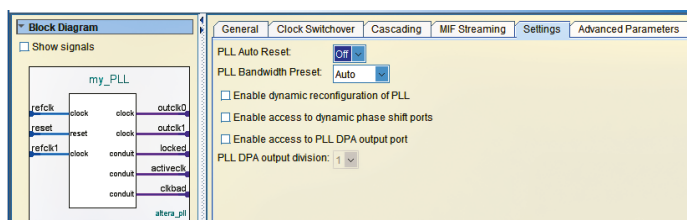
RYŚ. 12.14. Zakładka **Cascading** Edytora parametrów podczas tworzenia bloku PLL

Zakładka **MIF Streaming** (rys. 12.15) umożliwia zdefiniowanie parametrów w przypadku dynamicznej rekonfiguracji bloku PLL oraz zapis właściwości dynamicznego przesunięcia fazowego w pliku MIF. W zakładce **MIF Streaming** można włączyć opcję **Generate MIF file** (generuj plik MIF) i włączyć dynamiczne przesunięcie fazowe dla przesyłania strumieniowego pliku MIF (**Enable Dynamic Phase Shift for MIF streaming**). Ponadto zakładka **MIF Streaming** pozwala na wybór licznika wyjściowego lub licznika $\div M$ dla dynamicznego przesunięcia fazy (DPS), liczby dynamicznych przesunięć fazowych oraz kierunku dynamicznego przesunięcia fazowego.



RYŚ. 12.15. Zakładka **MIF Streaming** Edytora parametrów podczas tworzenia bloku PLL

Zakładka **Settings** (rys. 12.16) umożliwia włączenie (**On**) lub wyłączenie (**Off**) opcji automatycznego resetowania bloku PLL w przypadku utraty blokady-trzymania (**PLL Auto Reset**), a także określenie wartości *wstępnego ustawienia pasma PLL* (PLL Bandwidth Preset) jako wysokiej (**High**), niskiej (**Low**), średniej (**Medium**) lub definiowanej automatycznie (**Auto**).



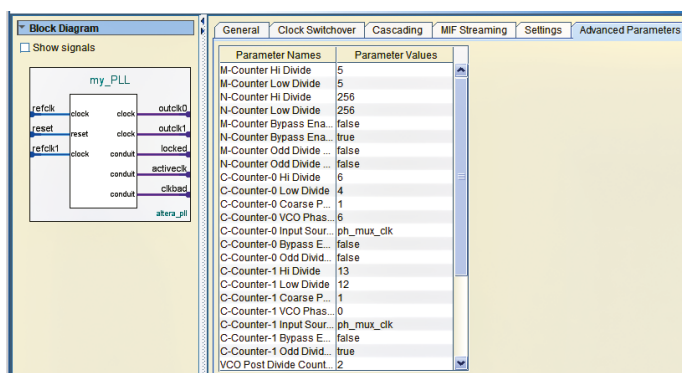
RYŚ. 12.16. Zakładka **Settings** Edytora parametrów podczas tworzenia bloku PLL

Dodatkowo w zakładce **Settings** można zezwolić na:

- *dynamiczną rekonfigurację PLL* (Enable dynamic reconfiguration of PLL);
- *dostęp do portów z dynamicznym przesunięciem fazowym* (Enable access to dynamic phase shift ports);
- *dostęp do portu wyjściowego DPA bloku PLL* (Enable access to PLL DPA output port).

Jeśli dostęp do portu DPA jest włączony, określana jest wartość dzielnika dla wyjścia DPA.

Zakładka **Advanced Parameters** (rys. 12.17) umożliwia szczegółowe zdefiniowanie dodatkowych parametrów bloku PLL.

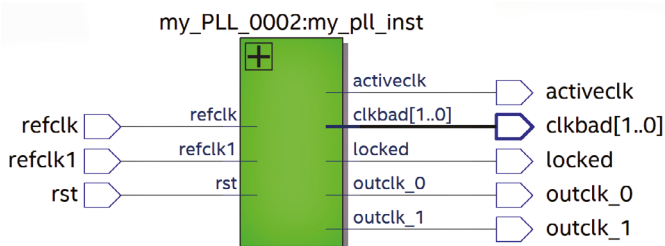


RYŚ. 12.17. Zakładka **Advanced Parameters** Edytora parametrów podczas tworzenia bloku PLL

Po zdefiniowaniu wszystkich niezbędnych parametrów utworzonego bloku PLL należy kliknąć przycisk Finish.

Po kliknięciu przycisku Finish edytor parametrów MegaWizard rozpoczyna generowanie bloku PLL zgodnie z określonymi parametrami. W naszym przykładzie plik *my_PLL.v* zostanie wygenerowany w języku Verilog, który można przeglądać w dowolnym edytorze tekstu, z wyjątkiem edytora systemu Quartus. Próba otwarcia pliku *my_PLL.v* w systemie Quartus powoduje ponowne wywołanie Edytora parametrów PLL, w którym można dokonać niezbędnych korekt.

Wyniki syntezy naszego projektu *my_PLL* pokazano na rys. 12.18.



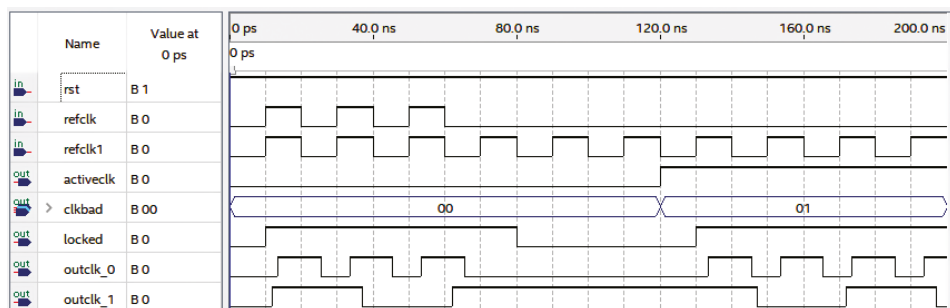
RYŚ. 12.18. Wyniki syntezy projektu *my_PLL* na poziomie RTL

Przy pomocy RTL Viewer (naciskając przycisk „+”) widać, że nasz projekt powstał przy użyciu megafunkcji `altera_pll`. Dalsze wgłębianie się w strukturę megafunkcji `altera_pll` pozwala zobaczyć strukturę zsyntetyzowanego bloku PLL, który jest zbudowany na bazie megafunkcji `altera_cyclonev_pll`.

Plik ograniczeń czasowych SDC w projektach zawierających PLL musi zawierać polecenie **`derive_pll_clocks`**. Polecenie **`derive_pll_clocks`** ma następujący format:
`derive_pll_clocks [-create_base_clocks] [-use_net_name]`

Polecenie **`derive_pll_clocks`** generuje instrukcje **`create_generate_clock`**, które na podstawie już znanych parametrów ograniczają zegary generowane przez blok PLL. Opcja **`-create_base_clocks`** dodaje instrukcje **`create_clock`** dla wejść zegarowych, które są zasilane z wyjść PLL. Domyślnie nazwa sygnałów zegarowych generowanych na wyjściach PLL jest taka sama, jak nazwa wyjścia PLL. Opcja **`-use_net_name`** umożliwia użycie nazwy sieci (węzła) jako nazwy sygnału zegara.

Wyniki symulacji czasowej projektu *my_PLL* przedstawiono na rys. 12.19.



RYS. 12.19 Wyniki symulacji czasowej projektu *my_PLL*

W pokazanym przykładzie projekt ma dwa wejściowe zegary referencyjne *refclk* i *refclk1* o tym samym okresie 20 ns. Na swoich wyjściach blok PLL generuje dwa wyjściowe sygnały zegarowe *outclk_0* i *outclk_1*. Zegar *outclk_0* ma częstotliwość 50 MHz (okres 20 ns), przesunięcie fazowe 1,5 ns i cykl pracy 60%. Zegar *outclk_1* ma częstotliwość 20 MHz (okres 50 ns) bez przesunięcia fazowego i cykl pracy wynoszący 50%. Domyślnie PLL używa sygnału zegara referencyjnego *refclk*.

Na rysunku 12.19 widać, że w chwili 70 ns nie ma kolejnego dodatniego zbocza na wejściu *refclk*. Oznacza to, że wystąpił błąd generatora synchronizacji związanego z wejściem *refclk*. W odpowiedzi jednostka PLL automatycznie przełącza zegar referencyjny na *refclk1* po trzech cyklach zegarowych w czasie 120 ns. Wyjściowe sygnały synchronizacji ponownie są generowane od czasu 120 ns.

Czytelnik jest proszony o przeanalizowanie rysunku 12.19 i wyjaśnienie zmian sygnałów *activeclk*, *clkbad* i *locked*, które są generowane przez blok PLL.

12.8. Rekonfiguracja bloków PLL

Do *rekonfiguracji bloków PLL* (PLL reconfiguration) w systemie Quartus wykorzystywane są *Bloki IP do rekonfiguracji PLL* (PLL Reconfig IP cores).

Rekonfiguracja bloków PLL umożliwia zmianę częstotliwości, współczynnika wypełnienia i przesunięcia fazowego generowanych sygnałów zegarowych w czasie rzeczywistym, bez konieczności rekonfiguracji całego FPGA. Rekonfiguracja bloku PLL jest możliwa tylko w trybie ułamkowego PLL (Fractional PLL – fPLL). Rekonfiguracja PLL jest również nazywana *dynamiczną rekonfiguracją fPLL* (fPLL dynamic reconfiguration).

Podczas dynamicznej rekonfiguracji fPLL zmieniane są parametry następujących elementów PLL:

- liczniki wyjściowe $\div C$;
- licznik sprzężenia zwrotnego $\div M$;
- licznik preskalera $\div N$;
- prąd układu pompy ładunku Charge pump;
- wartości R i C filtra pętli Loop Filter;
- przesunięcie fazowe każdego licznika;
- wartości dzielenia ułamkowego M_{FRAC} modulatora delta-sigma DSM.

Rekonfiguracja bloków PLL jest używana:

- w systemach wbudowanych pracujących na wielu częstotliwościach;
- przy dostosowaniu systemu wbudowanego do warunków środowiska pracy;
- podczas regulacji opóźnienia t_{CO} (clock-to-out) w czasie rzeczywistym.

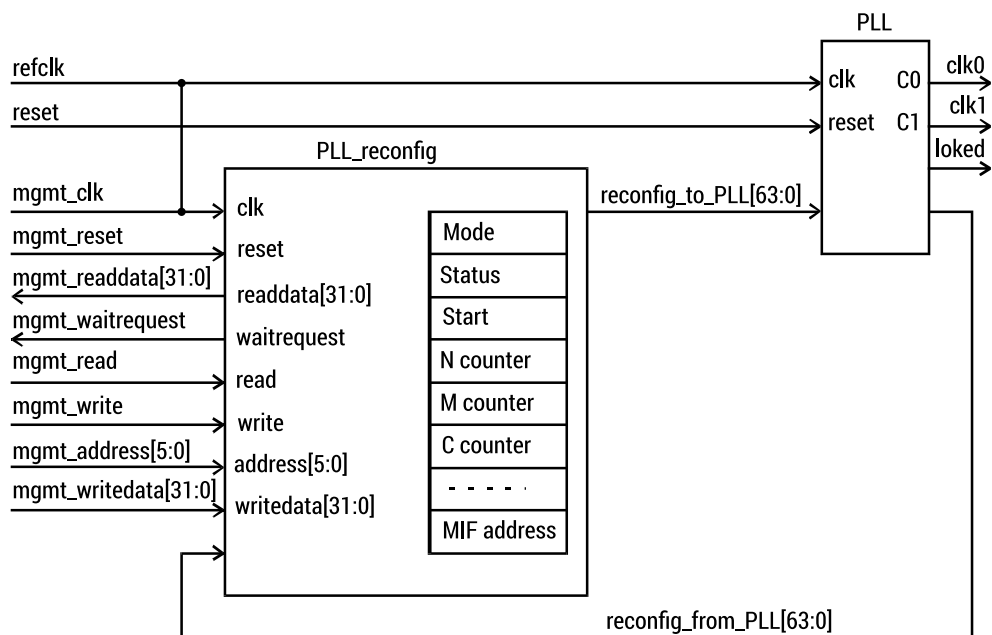
Przypomnijmy, że opóźnienie t_{CO} to czas od momentu nadejścia aktywnego zbocza sygnału zegarowego do pojawienia się stabilnych danych na wyjściu rejestru.

Rekonfiguracja PLL jest wykonywana przy użyciu oddzielnego Bloku IP do rekonfiguracji PLL (PLL Reconfig IP core), co pokazano na rysunku 12.20.

Blok IP rekonfiguracji PLL współpracuje z logiką użytkownika za pomocą *interfejsu Avalon Memory-Mapped* (Avalon-MM).

Na rysunku 12.20 wykorzystywane są następujące sygnały sterujące interfejsu Avalon-MM:

- *readdata[31:0]* – dane odczytane z Bloku IP rekonfiguracji PLL;
- *writedata[31:0]* – dane zapisywane do Bloku IP rekonfiguracji PLL;
- *address[5:0]* – adres rejestru do operacji odczytu lub zapisu;
- *read* – sygnał wskazujący na operację odczytu;
- *write* – sygnał wskazujący na operację zapisu;
- *reset* – sygnał resetowania;
- *waitrequest* – sygnał oczekiwania na żądanie; gdy ten sygnał jest wysoki, moduł PLL-reconfig ignoruje operacje odczytu lub zapisu.



RYS. 12.20. Schemat rekonfiguracji PLL przy użyciu Bloku IP do rekonfiguracji PLL

Moduł PLL-reconfig zawiera szereg rejestrów wewnętrznych (tab. 12.1), których zapis pozwala na rekonfigurację bloku PLL.

TAB. 12.1. Rejestry modułu PLL-reconfig

Nazwa	Rozmiar	Adres	Opis
Mode	1	000000	Rejestr trybu pracy modułu PLL-reconfig: 0 – oczekiwanie na żądanie (waitrequest); 1 – odpytywanie
Status	1	000001	Rejestr statusu bloku PLL: 0 – zajęty; 1 – gotowy
Start	1	000010	Rejestr startu procesu rekonfiguracji, zapis 0 lub 1 rozpoczyna rekonfigurację PLL lub proces dynamicznego przesunięcia fazowego
N Counter	18	000011	Rejestr zawierający parametry konfiguracyjne licznika $\div N$
M Counter	18	000100	Rejestr zawierający parametry konfiguracyjne licznika $\div M$
C Counter	23	000101	Rejestr zawierający parametry konfiguracyjne licznika $\div Cn$
Dynamic_Phase_Shift	22	000110	Rejestr zawierający parametry konfiguracyjne dynamicznego przesunięcia fazowego
M Counter Fractional Value (K)	32	000111	Wartość ułamkowa K licznika $\div M$ (dla DSM); $M_{FRAC} = K[X-1:0] / 2^X$, gdzie $X = 8, 16, 24$ lub 32
Bandwidth	4	001000	Ustawienie pasma bloku PLL

Nazwa	Rozmiar	Adres	Opis
Charge Pump	3	001001	Ustawienie prądu pompy ładunku dla generatora VCO
VCO DIV	1	011100	Włączenie dzielnika przez 2 dla częstotliwości F_{VCO}
.mif Base Address	9	011111	Adres bazowy profilu PLL w pliku inicjalizacji pamięci MIF (Memory Initialization File)

Sygnaly modułu PLL-reconfig są sterowane przez logikę użytkownika. Ponieważ sterowanie to może być dość złożone, w rzeczywistych systemach wbudowanych sygnały modułu PLL-reconfig są zwykle kontrolowane przez procesor wbudowany. W FPGA firmy Intel można do tego celu używać procesora Nios II.

12.9. Rekonfiguracja PLL za pomocą interfejsu Avalon-MM

Interfejs Avalon-MM pozwala na rekonfigurację PLL za pomocą procesora wbudowanego, na przykład Nios II. Aby zrekonfigurować ustawienia fPLL za pomocą interfejsu Avalon-MM, należy wykonać następujące kroki:

Algorytm 12.1. Rekonfiguracja PLL za pomocą interfejsu Avalon-MM

1. Uruchamiając blok IP rekonfiguracji PLL (Altera PLL Reconfig IP core), za pomocą operacji zapisu Avalon, wpisz wartość 0 lub 1 do rejestru trybu, aby określić tryb pracy bloku: *polling* lub *waitrequest*.
2. Użyj operacji zapisu interfejsu Avalon, aby zdefiniować adres rejestru i jego nową wartość. Adresy rejestrów bloku PLL podano w tabeli 12.1. Więcej informacji można znaleźć w [14].
3. Powtórz krok 2 dla wszystkich rekonfigurowalnych elementów PLL, które chcesz zmienić.
4. Użyj operacji zapisu Avalon, aby zapisać wartość 0 lub 1 w rejestrze Start. Zapisanie dowolnej wartości do rejestru Start inicjuje dynamiczną rekonfigurację PLL lub dynamiczne przesunięcie fazowe.

Jeśli rejestr trybu Mode jest ustawiony na 0 w kroku 1 (tryb oczekiwania zapytania *waitrequest*), blok IP rekonfiguracji PLL ustawia sygnał *mgmt_waitrequest* aż do zakończenia rekonfiguracji. Następną operacją odczytu lub zapisu interfejsu Avalon może być wykonana tylko po tym, jak blok IP rekonfiguracji PLL zresetuje sygnał *mgmt_waitrequest*.

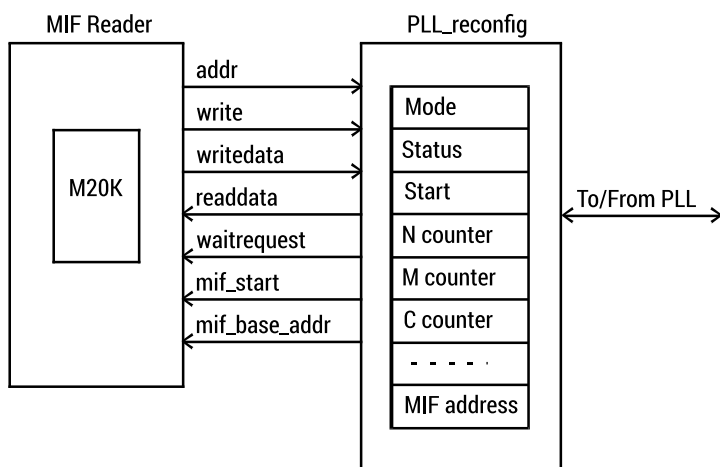
Jeśli rejestr trybu Mode jest ustawiony na 1 (tryb odpytywania *polling*) w kroku 1, blok IP rekonfiguracji PLL zapisze wartość 0 (zajęty) do rejestru stanu Status. Aby upewnić się, że rekonfiguracja została zakończona, należy okresowo sprawdzać wartość rejestru Status przy użyciu operacji odczytu interfejsu Avalon. Blok IP rekonfiguracji PLL ignoruje wszelkie nowe instrukcje rekonfiguracji (operacje zapisu Avalon) do momentu odczytania wartości 1 z rejestru stanu.

Ponadto PLL musi uzyskać dostęp do sygnału zegara referencyjnego przed wykonaniem rekonfiguracji lub dynamicznego przesunięcia fazowego.

12.10. Rekonfiguracja bloków PLL za pomocą strumieniowego przesyłania pliku MIF ze wbudowanego bloku pamięci FPGA

Bloki PLL można również rekonfigurować za pomocą pamięci wbudowanych, takich jak M20K. Informacje o rekonfiguracji PLL można zapisać do pamięci wbudowanej FPGA przy użyciu *pliku inicjalizacji pamięci MIF* (Memory Initialization File). Do tego celu, w pliku MIF zapisywane są informacje o rekonfiguracji bloku PLL (*profilu PLL*). Plik konfiguracyjny MIF jest tworzony przez Edytor parametrów PLL. Można stworzyć wiele plików MIF, wpisywać w nich własne instrukcje i łączyć pliki razem w celu uzyskania różnych opcji rekonfiguracji PLL.

Aby odczytać informacje o rekonfiguracji bloku PLL z pamięci wbudowanej, należy utworzyć osobny *moduł odczytu pliku MIF* (MIF Reader). Moduł MIF Reader jest tworzony przez blok IP Altera PLL Reconfig, po ustawieniu na 1 opcji **Enable MIF Streaming**. Interfejs modułu PLL-reconfig z modułem MIF Reader pokazano na rysunku 12.21.



RYS. 12.21. Struktura schematu rekonfiguracji PLL w trybie strumieniowania pliku MIF

Rekonfiguracja PLL poprzez odczytanie informacji z bloku pamięci wbudowanej nazywa się *rekonfiguracją w trybie strumieniowania MIF* (MIF streaming reconfiguration).

W celu rekonfiguracji bloku PLL na magistrali *writedata* ustawiany jest adres bazowy profilu PLL, a na magistrali *address* ustawiana jest wartość 011111. Po zmianie wartości rejestru Start, moduł PLL-reconfig rozpoczyna odczytywanie wartości nowych ustawień bloku PLL z pliku MIF, odpowiednio zmieniając konfigurację PLL. Moduł PLL-reconfig generuje sygnały wyjściowe, gdy wszystkie zmiany zostały dokonane i PLL przechwycił zegar referencyjny.

12.11. Dynamiczne przesunięcie fazowe bloku PLL

Funkcja dynamicznego przesunięcia fazowego umożliwia przesunięcie fazowe zegarów wyjściowych PLL względem siebie nawzajem i względem *sygnału zegara referencyjnego* (reference clock). Najmniejszy krok zmiany to $1/8$ okresu częstotliwości F_{VCO} . Podczas procesu dynamicznego przesuwania faz, zegary wyjściowe bloku PLL pozostają aktywne.

Istnieją dwie metody dynamicznego przesunięcia fazowego:

- użycie bloku IP rekonfiguracji PLL (Altera PLL Reconfig);
- użycie układu dynamicznego przesunięcia fazowego, wykorzystującego blok IP do tworzenia bloku PLL (Altera PLL).

12.11.1. Wykonywanie dynamicznego przesunięcia fazowego przy użyciu Bloku IP do rekonfiguracji PLL (Altera PLL Reconfig)

Dynamiczne przesunięcie fazowe z wykorzystaniem Bloku IP do rekonfiguracji PLL odbywa się w pełnej zgodności z algorytmem rekonfiguracji PLL z wykorzystaniem interfejsu Avalon-MM, gdzie w kroku 2 do rejestru `Dynamic_Phase_Shift[20:16]` musi zostać wpisany adres licznika wyjściowego C_n , którego wyjście zostanie przesunięte w fazie.

Implementacja dynamicznego przesunięcia fazowego w blokach PLL ma szereg cech, które można znaleźć w [14].

12.11.2. Wykonywanie dynamicznego przesunięcia fazowego przy użyciu Bloku IP Altera PLL

Aby wykonać dynamiczne przesunięcie fazowe za pomocą Bloku IP Altera PLL w Edytorze parametrów PLL w zakładce **Settings**, włącz opcję **Enable access to dynamic phase shift ports** (Włącz dostęp do portów dynamicznego przesunięcia fazowego). W efekcie do utworzonego modułu PLL zostaną dodane następujące porty wejściowe:

- *phase_en* – sygnał zezwolenia na przesunięcie fazy, dodatnie zbocze na tym porcie inicjuje jedno przesunięcie fazowe;
- *scanclk* – wejście zegarowe do włączania i wyłączania dynamicznego przesunięcia fazowego w połączeniu z sygnałem *phase_en*;
- *updn* – znak przesunięcia fazowego: 1 – przesunięcie dodatnie; 0 – przesunięcie ujemne;
- *cntsel[4:0]* – wybór licznika wyjściowego, pięciobitowa wartość binarna n na tym wejściu określa licznik $\div C_n$, którego wyjście będzie przesunięte w fazie.

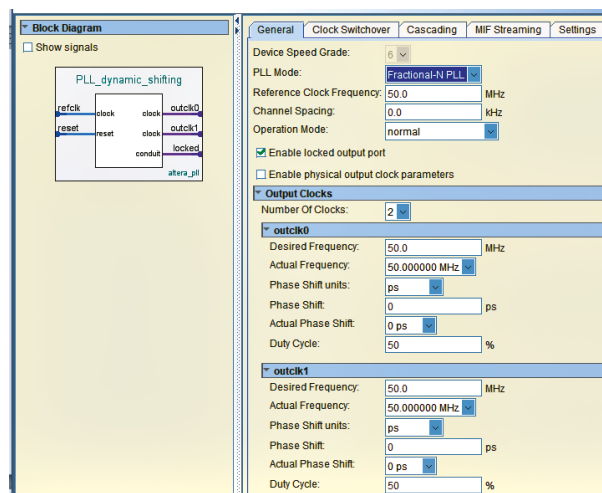
Ponadto do modułu PLL zostanie dodany port wyjściowy *phase_done*. Wysoka wartość na tym porcie wskazuje, że korekcja fazy jest zakończona i PLL jest gotowy do wykonania działań korekcyjnych na następnym zboczu narastającym wejścia *scanclk*.

Chcąc włączyć i skonfigurować dynamiczne przesunięcie fazowe za pomocą Bloku IP Altera PLL, należy wykonać poniższe kroki:

Algorytm 12.2. Wykonywanie dynamicznego przesunięcia fazowego przy użyciu Bloku IP Altera PLL

1. Ustaw wymagane wartości dla sygnałów *updn* i *cntsel*.
2. Ustaw sygnał *phase_en* na co najmniej dwa cykle *scanclk*. Każdy impuls sygnału *phase_en* umożliwia jedno przesunięcie fazowe.
3. Zresetuj sygnał *phase_en* po tym, jak *phase_done* osiągnie stan niski.

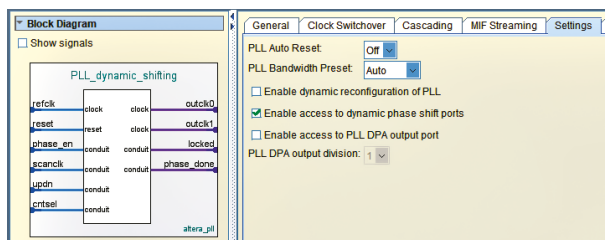
Jako przykład rozpatrzmy projekt bloku PLL *PLL_dynamic_shifting*, który wykorzystuje dynamiczne przesunięcie fazowe. Zakładka **General** Edytora parametrów podczas tworzenia bloku PLL została pokazana na rysunku 12.22.



RYŚ. 12.22. Zakładka **General** z przykładu dynamicznego przesunięcia fazowego

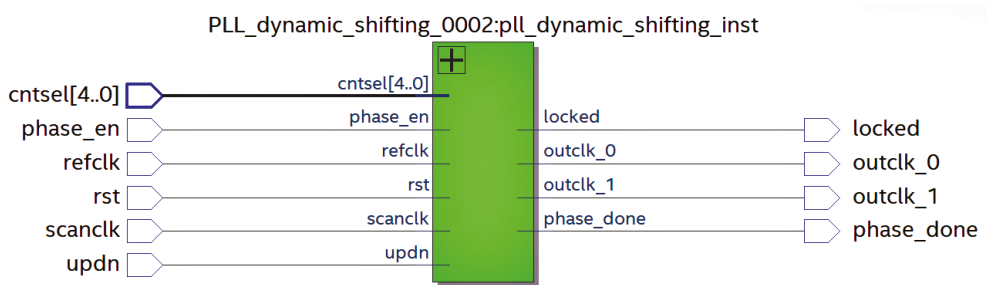
W tym oknie wybierany jest tryb ułamkowy pracy jednostki PLL. Referencyjny sygnał zegarowy ma częstotliwość 50 MHz. Wybrano normalny (normal) tryb sprzężenia zwrotnego jednostki PLL i dwa wyjściowe sygnały zegarowe *outclk0* i *outclk1* o tej samej częstotliwości 50 MHz.

Aby móc wykonywać dynamiczne przesunięcie fazowe, w zakładce **Settings** należy zaznaczyć opcję **Enable access to dynamic phase shift ports** (Włącz dostęp do portów dynamicznego przesunięcia fazowego), jak pokazano na rysunku 12.23. Pozostałe wartości parametrów Bloku IP należy pozostawić bez zmian.

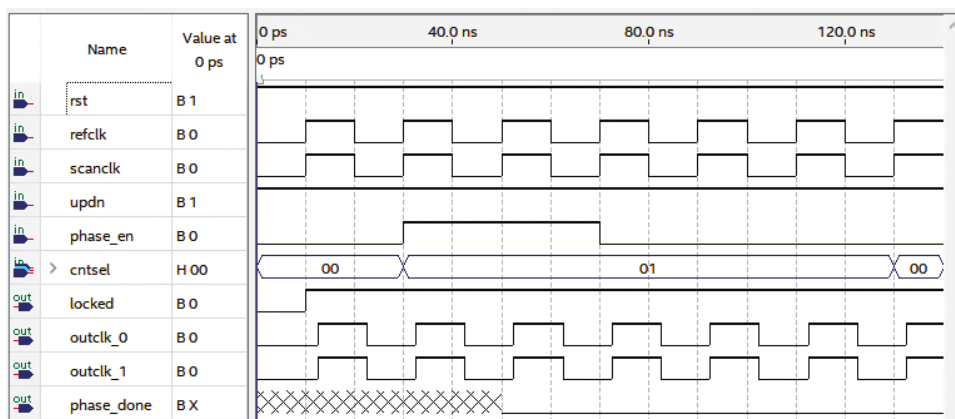


RYS. 12.23. Zakładka **Settings** z przykładu dynamicznego przesunięcia fazowego

Wyniki syntezy i symulacji projektu *PLL_dynamic_shifting* przedstawiono na rysunkach 12.24 i 12.25.



RYS. 12.24. Wyniki syntezy projektu *PLL_dynamic_shifting* na poziomie RTL



RYS. 12.25. Wyniki symulacji projektu *PLL_dynamic_shifting*

Na rysunku 12.25 widać, że po czasie 30 ns ustawiona została wartość 01h na magistrali cntsel, tzn. przesunięcie fazowe zostanie wykonane na wyjściu *outclk1*, a stan wysoki ustawiono na porcie *phase_en*, co zezwoliło na przesunięcie fazowe. Przesunięcie fazowe odbywa się wraz z nadejściem następnego dodatniego zbocza sygnału zegara *scanclk* w czasie 50 ns, o czym świadczy niski poziom sygnału *phase_done*.

12.12. Wnioski

Blok PLL jest pętlą synchronizacji fazowej ze sprzężeniem zwrotnym. W układach FPGA bloki PLL są używane do generowania wewnętrznych sygnałów zegarowych o określonej częstotliwości, współczynnika wypełnienia (duty cycle) i przesunięciu fazowym, na bazie sygnału zegara referencyjnego.

Głównymi elementami bloku PLL są licznik preskalera $\div N$, detektor fazy PFD, układ pompy ładunku Charge Pump, pętla filtra dolnoprzepustowego, generator przestrajany napięciem VCO, licznik sprzężenia zwrotnego $\div M$ oraz liczniki wyjściowe postskalera $\div C0, \dots, \div Cn$.

Bloki PLL układów FPGA z rodziny Cyclone V mogą mieć jeden lub dwa wejściowe sygnały zegarowe (*inclk0* i *inclk1*), mogą zawierać układ do wyboru sygnału zegara referencyjnego, układ blokady (Lock circuit) i 9 liczników wyjściowych postskalera ($\div C0, \dots, \div C8$). Dodatkowo wyjście licznika sprzężenia zwrotnego ($\div M$) jest również wyprowadzane do portu wyjściowego bloku PLL. Dodatkowo częstotliwość generatora sterowanego napięciem VCO można podzielić przez 2. Bloki PLL z rodziny Cyclone V, oprócz wewnętrznych, mogą mieć również zewnętrzne sprzężenia zwrotne.

Bloki PLL rodziny Cyclone V obsługują dwa tryby działania: całkowity i ułamkowy. W trybie ułamkowym, ze względu na to, że licznik $\div M$ może przyjmować wartości ułamkowe, wartości częstotliwości wyjściowych sygnałów zegarowych są wyznaczone z większą dokładnością.

Liczniki bloków PLL rodziny Cyclone V występują w rozmiarach od 1 do 512. Dzięki temu częstotliwość wejściowa może zostać zwiększona 512 razy lub zmniejszona 262 144 razy. Jeśli to nie wystarczy, sąsiednie bloki PLL można połączyć kaskadowo.

Liczbę bloków PLL i ich lokalizację w docelowym układzie FPGA można znaleźć w dokumentacji technicznej.

Minimalna wartość przesunięcia fazowego wynosi $1/8$ okresu częstotliwości generatora VCO. Częstotliwość generatora VCO jest automatycznie ustawiana przez system Quartus na najniższą wspólną wielokrotność częstotliwości wyjściowych zegarów bloku PLL.

Cykl roboczy sygnału zegara wyjściowego PLL jest określony przez współczynnik wypełnienia (duty cycle) i jest wyrażony w procentach.

Bloki PLL rodziny Cyclone V obsługują sześć trybów synchronizacji sprzężenia zwrotnego: tryb synchronizacji źródłowej (Source Synchronous Mode), tryb kompensacji LVDS (LVDS Compensation Mode), tryb bezpośredni (Direct Mode), tryb normalnej kompensacji (Normal Compensation Mode), tryb bufora zerowego opóźnienia ZDB (Zero-Delay Buffer Mode) i tryb zewnętrznego sprzężenia zwrotnego (External Feedback Mode).

Wejściowe i wyjściowe opóźnienia są w pełni kompensowane przez PLL tylko wtedy, gdy jako źródło sygnału zegara używane jest dedykowane zewnętrzne wyprowadzenie zegarowe FPGA, powiązane z tym blokiem PLL. To, które zewnętrzne wyprowadzenia zegarowe FPGA są powiązane z konkretnym PLL, jest określone w dokumentacji technicznej każdego układu FPGA.

W trybie synchronizacji źródłowej PLL służy do odbierania danych synchronizowanych przez źródło danych, przy jednoczesnej kompensacji opóźnień z zewnętrznych wyprowadzeń FPGA do wewnętrznych portów rejestrów FPGA.

W trybie kompensacji LVDS blok PLL utrzymuje relację sygnału zegara i danych z zewnętrznych wyprowadzeń FPGA na portach wewnętrznych rejestrów FPGA, podczas gdy sygnał zegara jest przesunięty o 180°.

W trybie bezpośrednim blok PLL nie kompensuje opóźnień sieci synchronizacji, dlatego generowane sygnały zegarowe mają przesunięcie fazowe względem referencyjnego sygnału zegarowego o rozmiar opóźnienia trasowania.

W normalnym trybie kompensacji blok PLL w pełni kompensuje opóźnienia globalnych sieci synchronizacji GCLK i regionalnych RCLK, tzn. fazy referencyjnego sygnału zegarowego i sygnałów zegarowych na portach rejestrów FPGA są takie same.

W trybie bufora zerowego opóźnienia ZDB wyrównywany fazowo jest zegar referencyjny i zegar na wyjściu FPGA. Aby zaimplementować tryb ZDB, należy użyć jednego dwukierunkowego wyprowadzenia FPGA.

W trybie zewnętrznego sprzężenia zwrotnego blok PLL może kompensować opóźnienie sygnału na płycie drukowanej PCB. W tym celu port wyjściowy FBOUT bloku PLL jest podłączony na płycie drukowanej do portu wejściowego FBIN tego samego bloku PLL. Tryb zewnętrznego sprzężenia zwrotnego wyrównuje fazę referencyjnego sygnału zegarowego i sygnału zegara na porcie wejściowym FBIN.

Bloki PLL rodziny Cyclone V obsługują trzy referencyjne tryby przełączania: automatyczne (Automatic), ręczne (Manual) i automatyczne przełączanie ze sterowaniem ręcznym (Automatic switchover with manual override).

W trybie automatycznego przełączania, jeśli aktywny zegar wejściowy przestanie działać, to w ciągu trzech cykli wejście zegarowe referencyjne jest przełączane na *inclk1* lub *inclk0*. Automatyczne rozpoznanie awarii jednego z zewnętrznych generatorów jest możliwe pod warunkiem, że ich częstotliwość różni się nie więcej niż o 20%.

W trybie ręcznym przełączanie między wejściowymi sygnałami zegarowymi sterowane jest wartością sygnału *clkswitch*.

Przełączanie automatyczne ze sterowaniem ręcznym jest podobne do trybu automatycznego, z tym że wysoka wartość sygnału *clkswitch* blokuje przełączanie wejściowych sygnałów zegarowych.

Multipleksowanie zegarów wejściowych za pomocą jednostki PLL odbywa się w trybie ręcznym.

Parametry bloku PLL są definiowane za pomocą Edytora parametrów Bloków IP systemu Quartus.

Plik ograniczeń czasowych SDC w projektach zawierających PLL musi zawierać polecenie **derive_pll_clocks**.

Dynamiczna rekonfiguracja bloku PLL jest możliwa tylko w trybie ułamkowym, pozwala na zmianę częstotliwości, współczynnika wypełnienia i przesunięcia fazowego generowanych sygnałów zegarowych w czasie rzeczywistym bez rekonfiguracji całego FPGA.

W systemach wbudowanych rekonfiguracja bloków PLL jest stosowana do:

- zapewnienia możliwości pracy na wielu częstotliwościach, np. w celu zmniejszenia zużycia energii;
- adaptacji do warunków zewnętrznych;
- korekty czasu t_{CO} .

Rekonfiguracja PLL jest wykonywana przy użyciu oddzielnych bloków IP rekonfiguracji PLL (PLL Reconfig IP cores), które mają szereg wewnętrznych rejestrów i są kontrolowane przez interfejs Avalon-MM ze wbudowanego procesora, takiego jak Nios II, lub ze wbudowanej jednostki pamięci, takiej jak M20K.

Rekonfiguracja PLL poprzez odczytanie informacji z bloku pamięci wbudowanej nazywana jest *rekonfiguracją strumieniowania MIF* (MIF streaming reconfiguration).

Dynamiczne przesunięcie fazowe umożliwia przesunięcie fazowe zegarów wyjściowych bloku PLL względem siebie nawzajem i względem zegaru referencyjnego. Istnieją dwie metody dynamicznego przesunięcia fazowego: użycie modułu Bloku IP rekonfiguracji (Altera PLL Reconfig) i wykorzystanie układu dynamicznego przesunięcia fazy z wykorzystaniem Bloku IP Altera PLL.

Dynamiczne przesunięcie fazowe z wykorzystaniem Bloku IP do rekonfiguracji PLL odbywa się w pełnej zgodności z algorytmem rekonfiguracji PLL z wykorzystaniem interfejsu Avalon-MM.

Wykonywanie dynamicznego przesunięcia fazowego przy użyciu Bloku IP Altera PLL jest realizowane poprzez włączenie opcji **Enable access to dynamic phase shift ports** (Dostęp do portów dynamicznego przesunięcia fazowego). W celu uzyskania dostępu i sterowania portami dynamicznego przesunięcia fazowego należy użyć algorytmu 12.2.

Zakończenie

W prezentowanej publikacji autorzy starali się odpowiedzieć na większość pytań, które zostały zadane w pierwszej książce [10] wraz obietnicą ich późniejszego omówienia, z wyjątkiem elementów języka opisu ograniczeń czasowych SDC, ponieważ ten ostatni jest dość dobrze opisany w [9]. Oczywiście pozostało wiele nierozwiązanych pytań dotyczących projektowania systemów wbudowanych na FPGA:

- wykorzystanie interfejsu do pamięci zewnętrznej i kart SD;
- wykorzystanie transceiverów (nadajników/odbiorników);
- zastosowanie procesorów wbudowanych, zaimplementowanych w FPGA;
- projektowanie systemów wbudowanych na podstawie bloków IP i tworzenie własnych bloków IP;
- implementacja interfejsów standardowych (USB, PS2, SPI, RS232, IrDA, VGA);
- modelowanie, analiza i testowanie systemów wbudowanych itp.

Jak wiadomo, życie nie stoi w miejscu, pojawiają się nowe obszary działalności człowieka i nowe obszary zastosowań systemów wbudowanych. Przykładami są różne akceleratory sprzętowe systemów wyszukiwania informacji w Internecie lub w algorytmach Big Data, systemy sztucznej inteligencji, systemy sterowania samochodami, maszynami latającymi itp. Nowe obszary zastosowań stawiają nowe wyzwania dla systemów wbudowanych, z którymi czytelnik być może będzie musiał sobie poradzić.

Wszelkie pytania i uwagi dotyczące książki i jej treści można kierować do autorów za pośrednictwem poczty elektronicznej na adresy: valsol@mail.ru i a.klimowicz@pb.edu.pl.

Literatura

1. Ifeachor E. C., Jervis B. W. Digital signal processing: a practical approach. – Pearson Education, 2002. – 960 pp.
2. Baranov S.I. Synteza automatów mikroprogramowanych. – Leningrad: Energy, 1979. – 232 str. (po rosyjsku)
3. Kaluzhnin L.A. Algorytmizacja problemów matematycznych // Problemy cybernetyki. – Moskwa: Fizmatgiz, 1959, wydanie. 2, str. 51–67. (po rosyjsku)
4. Soloviev V.V., Samal D.I. Metoda syntezy dowolnej logiki na programowalnych urządzeniach logicznych. // Automatyka i inżynieria komputerowa, 1997, nr 1, str. 57–64. (po rosyjsku)
5. Solov'ev V.V. Implementation of finite-state machines based on programmable logic ICs with the help of the merged model of Mealy and Moore machines // Journal of Communications Technology and Electronics, 2013, Vol. 58, No. 2, pp. 172–177.
6. Soloviev V.V. Podstawy języka projektowania sprzętu cyfrowego Verilog. – Moskwa: Gorąca linia – Telecom, 2014. – 208 str. (po rosyjsku)
7. Soloviev V.V. Architektury Xilinx FPGA: seria CPLD i FPGA 7. – Moskwa: Gorąca linia – Telecom. 2016. – 392 str. (po rosyjsku)
8. Soloviev V.V. Projektowanie elementów pamięci na podstawie FPGA // Komponenty i technologie, 2017, nr 11, s. 6–14. (po rosyjsku)
9. Soloviev V.V. Analiza czasowa układów programowalnych. – Moskwa: Gorąca linia – Telecom. 2018. – 360 s. (po rosyjsku)
10. Salauyou V., Grześ T., Bułatowa I., Klimowicz A. Język Verilog w projektowaniu systemów wbudowanych na układach FPGA. – Białystok: Oficyna Wydawnicza Politechniki Białostockiej, 2021. – 440 str.
11. Wilmschurst T. Designing embedded systems with PIC microcontrollers: principles and applications. – Elsevier, 2006. – 544 pp.
12. Cyclone V Device Handbook. Vol. 1: Device Interface and Integration. – Altera, 2018. – 320 p.
13. FIFO Intel FPGA IP. User Guide. – Intel. 2018.
14. Implementing Fractional PLL Reconfiguration with Altera PLL and Altera PLL Reconfig IP Cores. Altera, 2016, 27 p. (AN 661)
15. Internal Memory (RAM and ROM). User Guide. – Altera Corporation. 2014.
16. RAM-Based Shift Register (ALTSHIFT_TAPS) IP Core. User Guide. – Altera Corporation. 2014.

Indeks tematyczny

a

adaptacyjny moduł logiczny (adaptive logic modules – ALM)	352
adres odczytu (read-address)	35
adres zapisu (write-address)	35
akumulator (accumulator)	195, 207
algorytmiczny automat skończony (algorithmic state machine – ASM)	16, 142
algorytmiczny automat skończony ze ścieżką przetwarzania danych (ASM with datapath – ASMD)	164, 182
analizator czasowy (time analyzer)	246
architektura zestawu instrukcji (instruction set architecture – ISA)	186, 207
arytmetyczno-logiczny układ (arithmetic logical unit – ALU)	186, 207
atrybuty syntezy (Synthesis Attributes)	82
automat mikroprogramowany (AM)	108
automat skończony (finite state machine – FSM)	16, 107
automat skończony ze ścieżką przetwarzania danych (finite state machine with datapath – FSM-D)	17, 163
automaty mikroprogramowane pseudo-równoważne	118
automaty mikroprogramowane równoważne	118

b

bit parzystości (parity bit)	34
blok ASM (ASM block)	144, 161
blok macierzy logicznej (Logic Array Block – LAB)	26, 352
blok pamięci macierzy logicznej (memory logic array block – MLAB)	26
blok pętli synchronizacji fazowej (PLL block)	374
blok PMA (physical medium attachment)	377
bloki sterowania zegarem (clock control blocks)	325, 335, 337
błędy synchronizacji (timing failures)	358
błędy zaokrągleń (rounding errors)	306
blokada (lock)	378
bramkowanie sygnału synchronizacji (clock gating)	338

c

całkowite PLL (integer PLL)	378
cyfrowe przetwarzanie sygnałów (digital signal processing – DSP).....	16, 46, 289
czas ustawiania (setup time)	36, 211, 327
czas utrzymywania (hold time)	36, 211, 327
czas od zbocza sygnału zegarowego do ustawienia wiarygodnych danych na wyjściu (clock-to-output)	371
czas od zbocza sygnału zegarowego do włączenia wyjść (clock-to-output-enable)	371
częstotliwość aktualna (Actual Frequency)	386
częstotliwość graniczna pasma przepustowości (bandwidth cut-off frequency)	303
częstotliwość graniczna pasma tłumienia (bandstop cut-off frequency)	303
częstotliwość próbkowania (sampling frequency)	304
częstotliwość referencyjnego sygnału zegarowego (Reference Clock Frequency).....	385

d

dekodowanie instrukcji (Decode).....	197, 256
detektor fazy (phase frequency detector – PFD).....	374
diagramy logiczne algorytmów	110
diagramy macierzowe algorytmów	110
dolnoprzepustowy filtr pętli (Loop Filter)	375
domena synchronizacji (clock domain)	331
dostęp do portów z dynamicznym przesunięciem fazowym (access to dynamic phase shift ports)	388
dostęp do portu wyjściowego DPA bloku PLL (access to PLL DPA output port).....	388
dostęp dwuportowy (dual-port access)	25
duplikacja rejestrów (register duplication)	353
dynamiczna rekonfiguracja fPLL (fPLL dynamic reconfiguration)	390
dynamiczna rekonfiguracja PLL (dynamic reconfiguration of PLL)	388

e

Edytor parametrów (Parameter Editor)	89, 384
elektrycznie kasowana programowalna pamięć tylko do odczytu (electrically erasable programmable read-only memory – EEPROM)	23
etykiety (znaczniki) GSA	113
etykiety GSA dodatkowe	118
etykiety GSA główne	118

f

FIFO (First-In, First-Out)	16, 24
filtr dolnoprzepustowy (lowpass filter – LPF)	295
filtr górnoprzepustowy (highpass filter – HPF)	295
filtr nierekursywny	296
filtr pasmowoprzepustowy (bandpass filter)	295
filtr poprzeczny (transversal filter)	305
filtr rekursywny	296
filtr pasmowozaporowy (bandstop filter)	295
filtr z nieskończoną odpowiedzią impulsową (infinite impulse response filter – IIR filter)	296
filtr ze skończoną odpowiedzią impulsową (finite impulse response filter – FIR filter)	32, 296
filtrowanie wielokanałowe (multi-channel filtering)	32
funkcja transformaty (transfer function)	297
funkcje automatycznej i krzyżowej korelacji (auto- and cross- correlation functions)	32

g

generator sterowany napięciem (voltage-controlled oscillator – VCO)	374
generatory liczb pseudolosowych (pseudo-random number generators)	32
graficzne schematy algorytmów (GSA)	110, 139
grupa przejść	118

h

harmoniczna (harmonic)	292
harmoniczna główna (main harmonic)	292
hazard	327

i

inicjalizacja pamięci	44, 60
instrukcje transferu (przesyłania) danych	212
interfejs Avalon Memory-Mapped (Avalon-MM)	390
Blok IP do rekonfiguracji PLL (PLL Reconfig IP cores)	390

j

jednoportowa pamięć typu RAM (single-port RAM)	28, 44
jednostka arytmetyczno-logiczna	186, 196, 222

jednostki funkcjonalne (functional units – FUs)	203
jednostka operacyjna (datapath)	204
jednostka sterująca (control unit)	204
język opisu sprzętu (hard description language – HDL)	308, 328
jitter	375

k

kasowalna programowalna pamięć tylko do odczytu (erasable programmable read-only memory – EPROM)	23
Katalog IP (IP Catalog)	89
kod rozruchowy procesora (boot-code)	22
komputer ze złożonym zestawem instrukcji (complex instruction set computer – CISC)	108
komputer ze zredukowanym zestawem instrukcji (reduced instruction set computer – RISC)	108, 186
kwadrant (quadrant)	352
kwantyzacja sygnałów (signal quantization)	306
kwantyzacja współczynników (coefficient quantization)	306

l

licznik programu (program counter – PC)	195, 209
licznik sprzężenia zwrotnego	375
liczniki wejściowe preskalera (Pre-scale Counter)	374
liczniki kaskadowe (ripple counters)	332
liczniki wyjściowe postskalera	375
linia opóźniająca z odczepem (tapped delay line)	305
logika obejścia (bypass logic)	84
logika sklejanie (glue logic)	205

m

mikroinstrukcja/ mikrorozkaz (microcode)	108
mikrooperacja (microoperation)	108
mikroprogram (microprogram)	108
mikroprogramowanie (microprogramming)	108
metoda próbkowania częstotliwości (frequency sampling method)	304
metoda rozmieszczania zer i biegunów (method of placing zeros and poles)	305
metoda transformacji dwuliniowej (two-line transformation method)	305
metoda ważenia (weighing method)	304
metody optymalizacji (optimization methods)	304

metodyka ASMD-FSMD.....	163
metodyka budowy ASM.....	145
metodyka projektowania AM zgodnie z GSA.....	112
metodyka projektowania filtrów cyfrowych na FPGA	302
metodyka projektowania procesorów na FPGA	204, 206
układ mnożący z akumulacją (Multiply and Accumulate – MAC).....	293
moduł do testowania (testbench).....	206
moduł odczytu pliku MIF (MIF Reader)	393
multipleksowanie (multiplexing)	335

n

niedomiąg (underflow)	102
nieokreślona (nieznana) wartość (don't care)	28, 47
nieokreślona (nieznana) wartość ograniczona (constrained don't care)	38
niepewność zegara (clock uncertainty).....	331
niejednorodność w paśmie przepustowości (non-uniformity in bandwidth)	304
niestabilność (instability)	327
nowe dane (new data).....	36, 94

o

odchylenie w paśmie przepustowości (deviation in bandwidth).....	303
odchylenie w paśmie tłumienia (deviation in bandstop).....	303
odczyt podczas zapisu (read-during-write).....	36, 42
odczyt-zapis pamięci (Memory)	256, 276
odłożyć na stos (push)	187, 218
odpowiedź amplitudo-częstotliwościowa filtra (filter amplitude-frequency response)	303
odpowiedź impulsowa filtra (filter impulse response)	296
ograniczenia wielocyklowe (multicycle constraints).....	353
opóźnienie (skew)	329
opóźnienie najgorszego przypadku (worst-case delay).....	246
opóźnienie propagacji (propagation delay)	328
oscylator sterowany napięciem (voltage-controlled oscillator – VCO)	374

p

pamięć danych (Data Memory – DM)	191, 207
pamięć typu FIFO (first-in, first-out – pierwsze weszło, pierwsze wyszło).....	24
pamięć instrukcji (Instruction Memory – IM)	192, 195, 209

pamięć typu LIFO (Last-In, First-Out)	16, 46, 79
pamięć nieulotna (non-volatile memory)	22, 43
pamięć o dostępie swobodnym (random-access memory – RAM)	210
pamięć podręczna (cache)	201, 207
pamięć podręczna danych (Data cache)	201, 207
pamięć podręczna instrukcji (Instruction cache)	201, 207
pamięć prosta dwuportowa (simple dual-port RAM)	28, 44, 64
pamięć rzeczywista dwuportowa (true dual-port RAM)	28, 44, 66
pamięć synchroniczna dynamiczna typu RAM (synchronous dynamic RAM – SDRAM)	22
pamięć synchroniczna statyczna typu RAM (synchronous static RAM – SSRAM)	22
pamięć statyczna o dostępie swobodnym (static random-access memory – SRAM)	22
pamięć stosowa	79
pamięć ściśle powiązana (tightly coupled memory)	24
pamięć tylko do odczytu (read only memory – ROM)	28, 44, 210
pamięć ulotna (volatile memory)	22
pamięć wbudowana (embedded memory)	23, 43
pamięć wbudowana w układ FPGA (FPGA on-chip memory)	22, 23
pętla kombinacyjna (combinational loop)	327
pętla synchronizacji fazowej (phase-locked loop – PLL)	293, 374
planowanie instrukcji statyczne (static instruction scheduling)	202
plik inicjalizacji pamięci (Memory Initialization File – MIF)	83
plik rejestrów (Register File)	187, 190, 195
pobranie ze stosu (pop)	218
pojemność (głębokość) pamięci typu FIFO	75
port o szerokości mieszanej (Mixed-Width Port)	38
port odczytu (read port)	38
port zapisu (write port)	38
potokowy (pipelined)	23
pożyczka cykli (cycle-borrowing)	328
procesor jednocyklowy (single-cycle CPU)	195
procesor PIC	195
procesor potokowy (pipelined CPU)	199
procesor superskalarny (superscalar processor)	202
procesor wbudowany (embedded processors)	185
procesor wielocyklowy (multiple-cycle CPU)	197

procesor wielordzeniowy (multicore CPU).....	203
procesor wielowątkowy (multithreading CPU)	203
procesor zwykły (ordinary CPU).....	203
projekty synchroniczne (synchronous designs)	326
przełącznik zegara (Clock Switchover)	386
przeniesienie (carry out)	205
przepełnienie (overflow)	102, 205, 306
przepełnienie stosu (full).....	205
przesunięcie fazowe (Phase Shift)	378, 386
przetwornik analogowo-cyfrowy (analog-to-digital converter – ADC).....	291
przetwornik cyfrowo-analogowy (digital-to-analog converter – DAC)	291
pusty stos (empty)	205

r

rdzenie (cores)	203
rejestr instrukcji (Instruction Register – IR)	197, 277
rejestr przesuwający (shift-register)	28, 44, 72
rejestr follower (follower registers)	364
rekonfiguracja bloków PLL (PLL reconfiguration).....	390
rekonfiguracja strumieniowania MIF (MIF streaming reconfiguration)	393
reset asynchroniczny (Asynchronous Reset).....	359
reset synchroniczny (Synchronous Reset)	359
retiming rejestrów (register retiming)	372
rozmieszczanie (placing).....	354
rozwijanie pętli (loop-unrolling).....	202
równoległość na poziomie instrukcji (instruction-level parallelism – ILP).....	202
równoległości na poziomie wątków (thread level parallelism – TLP)	203
rząd filtra (filter order).....	296
rzeczywista wartość przesunięcia fazowego (Actual Phase Shift).....	386

s

schematy blokowe (block diagrams)	142
sieć działań algorytmu	143
sieć działań ASM	145, 161
sieć działań ASM ze ścieżką przetwarzania danych (algorithmic state machine with datapath – ASMD).....	17, 163
sieci synchronizacji:	
– dedykowane szybkie regionalne (dedicated fast regional clock networks).....	334
– globalne (global clock networks – GCLK)	334

– podwójne regionalne (Dual-Regional)	352
– regionalne (regional clock networks – RCLK)	334
sieci zegarowe (clock networks)	334, 352
słowa sterujące (control words)	109
sprawdzanie i korygowanie błędów (error checking and correcting – ECC)	93
sprzętowa implementacją algorytmu mnożenia	127
stacje rezerwacji (reservation stations)	202
stan metastabilny (metastable state)	327
stan następny (next state)	116
stan obecny (present state)	116
stan przejścia	116
stare dane (old data)	36, 47, 94
stopień szybkości FPGA (Device Speed Grade)	385
strata blokady-trzymania (lose-of-lock)	378
struktury bezpośrednie (direct structures)	305
struktury kaskadowe (cascading structures)	305
struktury równoległe (parallel structures)	305
sygnał podstawowy (basic signal)	330
sygnał referencyjny (reference signal)	330
sygnał strobuujący (gated signal)	338
sygnał zezwolenia synchronizacji (clock enable – CE)	35
sygnały analogowe (ciągłe)	291
sygnały dyskretne (cyfrowe)	291
system na jednym układzie (System on Chip - SoC)	185, 293
system wieloprocessorowy zaimplementowany na jednym układzie (chip-multiprocessor)	204
systemy wbudowane (embedded systems)	15
synchronizacja niezależna (Independent Clock)	35
synchronizacja odczytu/zapisu (Read/Write Clock)	35
synchronizacja wejścia/wyjścia (Input/Output Clock)	35
synchronizacja z jednym zegarem (Single Clock)	35
synchronizowany reset asynchroniczny (Synchronized Asynchronous Reset)	366
synteza fizyczna logiki kombinacyjnej (physical synthesis for combinational logic)	353
szerokość mieszana (mixed-width)	34
szum ADC (ADC noise)	306

t

tablica przeglądowa / wyszukiwania (look-up table – LUT)	15, 21, 329
tablica przejść (transition table)	116
tłumienność w paśmie zaporowym (attenuation in bandstop)	304
transformacja niezmienna odpowiedzi impulsowej (invariant transformation of impulse response)	305
trasowanie (routing)	354
tryb bezpośredniej kompensacji (Direct Compensation Mode)	377, 379
tryb bufora zerowego opóźnienia (zero delay buffer – ZDB Mode)	382
tryb dwuportowy (Dual-port mode)	23
tryb kompensacji LVDS (LVDS Mode)	377
tryb niskiego poboru mocy (low-power mode)	43
tryb normalnej kompensacji (Normal Compensation Mode)	377, 379, 381
tryb mieszany odczytu podczas zapisu na porcie (Mixed-Port Read-During-Write Mode)	36
tryb odczytu podczas zapisu tego samego portu (Same-Port Read-During-Write Mode)	36
tryb przewidywania (show-ahead mode)	34
tryb spakowany (packed mode)	40
tryb synchronizacji źródłowej (Source Synchronous Mode)	377, 379
tryb zewnętrznego sprzężenia zwrotnego (External Feedback Mode)	377, 379, 382
tryby przełączania zegarów referencyjnych (reference clock switching modes)	383
– automatyczny (Automatic)	383
– ręczny (Manual)	383
– automatyczne przełączanie ze sterowaniem ręcznym (Automatic switchover with manual override)	383, 386
tryby zegara sprzężenia zwrotnego (feedback clock modes)	379

u

układ programowalny (field programmable gate array – FPGA)	15
układ pompy ładunku (Charge Pump)	375
układ wyboru sygnału zegarowego odniesienia (reference clock selection circuit)	376
układ blokady (Lock circuit)	377, 397
ułamkowy PLL (fractional PLL – fPLL)	378, 390
układ operacyjny (datapath)	127

układ sterujący (control unit/ controller)	195, 204, 207
usterki (faults)	327

W

wartość nieznana (don't care)	36
wartości graniczne (boundaries)	239
warunki logiczne (logical conditions)	139
wejścia danych (data-input)	35
widmo sygnału (signal spectrum)	292
wierzchołki stanów (state box)	143
wierzchołki warunkowe (decision box)	143
wierzchołki wyjść warunkowych (conditional output box)	143
współczynnik przewidywania (coefficient feedforward)	300
współczynnik rozgałęzienia na wyjściu (fan-out)	354
współczynnik sprzężenia zwrotnego (coefficient feedback)	300
współczynnik wypełnienia (duty-cycle)	376
wstępne ustawienie przepustowości bloku PLL (PLL Bandwidth Preset)	387
wybór instrukcji (Fetch)	197
wyjścia danych (data-output)	35
wyjścia warunkowe (conditional output)	144
wykonanie instrukcji (Execute)	197
wynik zerowy (zero)	205

Z

zakłócenia (glitches)	327, 375
zapas czasowy (slack)	247, 359
zawodność (unreliability)	327
zegar odczytu (read clock)	35
zegar referencyjny (reference clock)	383, 394
zegar wejściowy (input clock)	35
zegar wyjściowy (output clock)	35
zegar zapisu (read clock)	35
zegary bramkowane (gated clocks)	338
zezwoleńie bajtów (byte enable)	35, 40, 45
zezwalenie bajtów zapisywanych danych (Byte Enable)	40
zezwoleńie na odczyt (read-enable)	35
zezwoleńie na zapis (write-enable)	35
zezwalenie synchronizacji adresu (address clock enable)	41, 45

zezwoleń synchronizacji niezależne (independent clock enable).....	35
zmiany kolejności buforów (reorder buffers)	202
zmiany nazw rejestrów (register renaming)	202
zsynchronizowany reset asynchroniczny (synchronized asynchronous reset).....	359
zwrotne zapisywanie danych do pliku rejestrów (Writeback).....	197
ż	
żądana częstotliwość (Desired Frequency)	386

Spis tabel

Tab. 1.1. Bloki pamięci wbudowanej różnych rodzin FPGA firmy Intel.....	25
Tab. 1.2. Dopuszczalne konfiguracje bloków pamięci wbudowanej układów FPGA firmy Intel	26
Tab. 1.3. Właściwości pamięci wbudowanej rodziny Cyclone V SE.....	27
Tab. 1.4. Tryby synchronizacji bloków pamięci wbudowanej.....	35
Tab. 1.5. Wartości wyjść pamięci wbudowanej w trybie odczytu podczas zapisu tego samego portu.....	37
Tab. 1.6. Tryby wyjściowe dla bloków pamięci wbudowanej w trybie mieszanym do odczytu podczas zapisu portu	37
Tab. 1.7. Konfiguracje portów o różnej szerokości dla jednostek M10K w trybie prostej dwuportowej pamięci typu RAM	39
Tab. 1.8. Konfiguracje portów o różnej szerokości dla jednostek M10K w trybie rzeczywistej dwuportowej pamięci typu RAM.....	39
Tab. 1.9. Wartości sygnałów <i>byteena</i> i zapisywane bajty dla szerokości danych 20 bitów	40
Tab. 1.10. Wartości sygnałów <i>byteena</i> i zapisywane bajty dla szerokości danych 40 bitów	40
Tab. 1.11. Początkowe wartości bloków pamięci wbudowanej po włączeniu zasilania	42
Tab. 4.1. Tablica przejść AM Mealy'ego, zbudowana zgodnie z GSA z rys. 4.6	117
Tab. 4.2. Tablica przejść AM Moore'a, zbudowana zgodnie z GSA z rys. 4.5	117
Tab. 4.3. Porównanie wyników symulacji funkcjonalnej AM Mealy'ego i Moore'a	122
Tab. 4.4. Macierz W do kodowania stanów wewnętrznych AM klasy C.....	124
Tab. 4.5. Parametry układów mnożących z uwzględnieniem liczby taktów sygnału zegara.....	138
Tab. 6.1. Wyniki badań projektu <i>mult_FSMD_Moore</i>	178
Tab. 6.2. Wyniki badań projektu <i>mult_FSMD_Mealy</i>	178
Tab. 6.3. Wyniki badań projektu <i>mult_FSMD_Mealy_1</i>	179
Tab. 6.4. Wyniki badań projektu <i>mult_a_Mealy</i>	179
Tab. 6.5. Badanie różnych metodyk projektowania sprzętu na przykładzie synchronicznego układu mnożącego	180
Tab. 7.1. Przykładowe kody dla różnych architektur zestawu instrukcji procesora	188

Tab. 7.2. Zestaw instrukcji mikrokontrolera PIC16F84A.....	189
Tab. 7.3. Zestaw instrukcji procesora PIC	192
Tab. 8.1. Tablica prawdy układu sterującego procesorem PIC.....	239
Tab. 8.2. Wyniki syntezy komponentów procesora	247
Tab. 8.3. Wyniki oceny szybkości komponentów procesora.....	248
Tab. 9.1. Wyniki badania kosztów realizacji procesorów PIC	285
Tab. 9.2. Wyniki badania szybkości procesorów PIC.....	285
Tab. 9.3. Wyniki badania wydajności procesorów PIC.....	286
Tab. 10.1. Typy filtrów FIR.....	298
Tab. 10.2. Specyfikacje badanych filtrów.....	319
Tab. 10.3. Parametry filtrów generowanych w systemie MATLAB.....	320
Tab. 10.4. Koszt realizacji filtrów.....	321
Tab. 10.5. Wydajność filtrów.....	321
Tab. 11.1. Zasoby FPGA używane podczas realizacji licznika kaskadowego i licznika binarnego.....	333
Tab. 11.2. Badania wyników syntezy projektów rejestrów z różnymi metodami bramkowania sygnałów zegarowych	348
Tab. 11.3. Wyniki syntezy rozpatrywanych projektów rejestrów z włączoną opcją Auto Gated Clock Conversion	349
Tab. 11.4. Opcje Syntezatora wpływające na synchronizację.....	355
Tab. 11.5. Opcje Fittera wpływające na synchronizację.....	356
Tab. 11.6. Wyniki badań sposobów resetowania rejestrów	368
Tab. 12.1. Rejestry modułu PLL-reconfig	391

Spis rysunków

Rys. 1.1. Jednoportowa pamięć typu RAM.....	29
Rys. 1.2. Prosta dwuportowa pamięć typu RAM.....	30
Rys. 1.3. Rzeczywista dwuportowa pamięć typu RAM.....	31
Rys. 1.4. Jednoportowa pamięć typu ROM	31
Rys. 1.5. Dwuportowa pamięć typu ROM	32
Rys. 1.6. Rejestr przesuwający magistrali w bloku pamięci wbudowanej	33
Rys. 1.7. Pamięć typu FIFO.....	33
Rys. 1.8. Strumień danych: a – w trybie odczytu podczas zapisywania tego samego portu; b – w trybie mieszanym odczytu podczas zapisu portu	36
Rys. 2.1. Statystyki kompilacji dla projektu <i>single_port_ram_single_clk</i>	48
Rys. 2.2. Widok projektu <i>single_port_ram_single_clk</i> na poziomie RTL.....	48
Rys. 2.3. Mapa technologiczna projektu <i>single_port_ram_single_clk</i>	49
Rys. 2.4. Wyniki symulacji projektu <i>single_port_ram_single_clk</i> : a – w przypadku oddzielnych operacji zapisu i odczytu; b – w przypadku naprzemiennych operacji odczytu i zapisu	50
Rys. 2.5. Widok projektu <i>single_port_ram_single_clk_v2</i> na poziomie RTL	51
Rys. 2.6. Wyniki symulacji projektu <i>single_port_ram_single_clk_v2</i> : a – w przypadku oddzielnych operacji odczytu i zapisu; b – w przypadku naprzemiennych operacji odczytu i zapisu	51
Rys. 2.7. Statystyki kompilacji dla projektu <i>single_port_ram_blocking_assign</i>	53
Rys. 2.8. Mapa technologiczna projektu <i>single_port_ram_blocking_assign</i>	53
Rys. 2.9. Wyniki symulacji projektu <i>single_port_ram_blocking_assign</i> : a – w przypadku oddzielnych operacji zapisu i odczytu; b – w przypadku naprzemiennych operacji odczytu i zapisu	54
Rys. 2.10. Widok projektu <i>single_port_ram_single_clk_v3</i> na poziomie RTL	55
Rys. 2.11. Wyniki symulacji projektu <i>single_port_ram_single_clk_v3</i>	55
Rys. 2.12. Statystyki wyników syntezy dla projektu <i>single_port_ram_assign</i>	56
Rys. 2.13. Widok projektu <i>single_port_ram_assign</i> na poziomie RTL.....	57

Rys. 2.14. Wyniki symulacji projektu <i>single_port_ram_assign</i> :	
a – w przypadku oddzielnych operacji zapisu i odczytu;	
b – w przypadku naprzemiennych operacji odczytu i zapisu	57
Rys. 2.15. Statystyki wyników syntezy dla projektu <i>single_port_ram_addr_reg</i>	59
Rys. 2.16. Widok projektu <i>single_port_ram_addr_reg</i> na poziomie RTL	59
Rys. 2.17. Wyniki symulacji projektu <i>single_port_ram_addr_reg</i> :	
a – w przypadku oddzielnych operacji odczytu i zapisu;	
b – w przypadku naprzemiennych operacji odczytu i zapisu	59
Rys. 2.18. Wyniki symulacji projektu <i>single_port_ram_with_initial</i>	61
Rys. 2.19. Wyniki symulacji projektu <i>single_port_ram_initial_by_file</i>	62
Rys. 2.20. Widok projektu <i>single_port_ram_dual_clk</i> na poziomie RTL	64
Rys. 2.21. Wyniki symulacji projektu <i>single_port_ram_dual_clk</i>	64
Rys. 2.22. Widok projektu <i>simple_dual_port_ram_single_clk_old_data</i> na poziomie RTL.....	65
Rys. 2.23. Wyniki symulacji projektu <i>simple_dual_port_ram_single_clk_old_data</i>	65
Rys. 2.24. Widok projektu <i>true_dual_port_ram_dual_clk_new_data</i> na poziomie RTL.....	67
Rys. 2.25. Wyniki symulacji projektu <i>true_dual_port_ram_dual_clk_new_data</i>	67
Rys. 2.26. Widok projektu <i>rom_case</i> na poziomie RTL	69
Rys. 2.27. Wyniki symulacji projektu <i>rom_case</i>	69
Rys. 2.28. Widok projektu <i>dual_port_rom</i> na poziomie RTL	71
Rys. 2.29. Wyniki symulacji projektu <i>dual_port_rom</i>	71
Rys. 2.30. Uogólniona struktura rejestru przesuwającego magistrali z odczepami	72
Rys. 2.31. Statystyki wyników syntezy projektu <i>shift_reg_68_bits</i>	73
Rys. 2.32. Widok projektu <i>shift_reg_68_bits</i> na poziomie RTL	73
Rys. 2.33. Statystyki wyników syntezy projektu <i>shift_reg_64_bits_4_taps</i>	74
Rys. 2.34. Widok projektu <i>shift_reg_64_bits_4_taps</i> na poziomie RTL.....	74
Rys. 2.35. Wyniki symulacji projektu <i>shift_reg_64_bits_4_taps</i>	75
Rys. 2.36. Symbol pamięci typu FIFO	75
Rys. 2.37. Funkcjonowanie pamięci typu FIFO	76
Rys. 2.38. Wyniki symulacji projektu <i>fifo</i>	78
Rys. 2.39. Funkcjonowanie pamięci typu LIFO kiedy ADDR_WIDTH = 2	80
Rys. 2.40. Fragment schematu projektu <i>lifo</i> , zsyntetyzowany przez kompilator	81
Rys. 2.41. Wyniki symulacji funkcjonalnej projektu <i>lifo</i>	82
Rys. 2.42. Wyniki symulacji projektu <i>single_port_ram_initial_by_attribute</i>	84
Rys. 3.1. Okno Katalogu IP podczas tworzenia modułu pamięci	90

Rys. 3.2. Okno do zapisywania plików modułu pamięci utworzonego za pomocą Katalogu IP.....	91
Rys. 3.3. Pierwsza z dziesięciu stron Edytora parametrów podczas tworzenia pamięci dwuportowej.....	91
Rys. 3.4. Druga strona Edytora parametrów podczas tworzenia pamięci dwuportowej.....	92
Rys. 3.5. Trzecia strona Edytora parametrów podczas tworzenia pamięci dwuportowej.....	93
Rys. 3.6. Piąta strona Edytora parametrów podczas tworzenia pamięci dwuportowej.....	94
Rys. 3.7. Siódma strona Edytora parametrów podczas tworzenia pamięci dwuportowej.....	95
Rys. 3.8. Ósma strona Edytora parametrów podczas tworzenia pamięci dwuportowej.....	95
Rys. 3.9. Dziewiąta strona podczas tworzenia pamięci dwuportowej.....	96
Rys. 3.10. Dziesiąta strona podczas tworzenia pamięci dwuportowej.....	97
Rys. 3.11. Widok projektu <i>ram_2_port</i> na poziomie RTL.....	98
Rys. 3.12. Wyniki symulacji projektu <i>ram_2_port</i>	98
Rys. 3.13. Pierwsza z ośmiu stron Edytora parametrów podczas tworzenia modułu pamięci typu FIFO.....	99
Rys. 3.14. Trzecia strona Edytora parametrów podczas tworzenia pamięci typu FIFO.....	100
Rys. 3.15. Czwarta strona Edytora parametrów podczas tworzenia pamięci typu FIFO.....	101
Rys. 3.16. Piąta strona Edytora parametrów podczas tworzenia pamięci typu FIFO.....	101
Rys. 3.17. Szósta strona Edytora parametrów podczas tworzenia pamięci typu FIFO	102
Rys. 3.18. Widok projektu <i>FIFO</i> na poziomie RTL	103
Rys. 3.19. Wyniki symulacji projektu <i>FIFO</i>	103
Rys. 3.20. Pierwsze okno Edytora parametrów podczas tworzenia rejestru przesuwającego.....	104
Rys. 3.21. Widok projektu <i>shift_reg</i> na poziomie RTL.....	105
Rys. 3.22. Wyniki symulacji projektu <i>shift_reg</i>	105
Rys. 4.1. Tradycyjna implementacja sterowania mikroprogramowego.....	109
Rys. 4.2. Wierzchołki GSA: a – początkowy; b – końcowy; c – operacyjny; d – warunkowy	111
Rys. 4.3. Błędny fragment GSA odpowiadający stanowi oczekiwania	112
Rys. 4.4. Prawidłowy fragment GSA odpowiadający stanowi oczekiwania na rys. 4.3.....	112
Rys. 4.5. Oznaczanie GSA dla syntezy AM Moore'a	114
Rys. 4.6. Oznaczenie GSA do syntezy AM Mealy'ego	115
Rys. 4.8. Wyniki symulacji projektu <i>AM_Mealy</i>	120
Rys. 4.9. Wyniki syntezy projektu <i>AM_Moore</i> na poziomie RTL	121
Rys. 4.10. Wyniki symulacji projektu <i>AM_Moore</i>	122

Rys. 4.7. Wyniki syntezy projektu <i>AM_Mealy</i> na poziomie RTL	123
Rys. 4.11. Wyniki syntezy projektu <i>AM_Moore_C</i> na poziomie RTL	126
Rys. 4.12. Wyniki symulacji projektu <i>AM_Moore_C</i>	126
Rys. 4.13. Ogólna struktura synchronicznego układu mnożącego	127
Rys. 4.14. Schemat blokowy synchronicznego układu mnożącego w postaci układu operacyjnego i układu sterującego.....	128
Rys. 4.15. Schemat układu operacyjnego, który implementuje algorytm mnożenia <i>a</i>	128
Rys. 4.16. Układ sterowania GSA do realizacji algorytmu mnożenia <i>a</i> : a – oznaczony etykietami do syntezy AM Moore'a, b – oznaczony do syntezy AM Mealy'ego	130
Rys. 4.17. Druga wersja GSA dla układu sterującego: a – przy oznaczaniu etykietami do syntezy AM Moore'a, b – przy oznaczaniu do syntezy AM Mealy'ego	131
Rys. 4.18. Widok projektu <i>mult_a</i> na poziomie RTL.....	136
Rys. 4.19. Wyniki symulacji projektu <i>mult_a</i>	137
Rys. 5.1. Elementy ASM: a – wierzchołek stanów; b – wierzchołek warunkowy; c – wierzchołek wyjść warunkowych	144
Rys. 5.2. Blok ASM.....	144
Rys. 5.3. Implementacja pętli oczekiwania w ASM: a – źle, b – poprawnie.....	145
Rys. 5.4. ASM automatu Moore'a sterującego algorytmem mnożenia <i>a</i>	147
Rys. 5.5. ASM automatu Mealy'ego sterującego algorytmem mnożenia <i>a</i>	148
Rys. 5.6. Wyniki syntezy projektu <i>ASM_Moore</i> na poziomie RTL.....	150
Rys. 5.7. Wyniki symulacji projektu <i>ASM_Moore</i>	150
Rys. 5.8. Wyniki syntezy projektu <i>ASM_Mealy</i> na poziomie RTL	151
Rys. 5.9. Wyniki symulacji projektu <i>ASM_Mealy</i>	152
Rys. 5.10. ASM wspólnego modelu automatów Mealy'ego i Moore'a	153
Rys. 5.11. Wyniki syntezy projektu <i>ASM_Mealy_Moore</i> na poziomie RTL.....	154
Rys. 5.12. Wyniki symulacji projektu <i>ASM_Mealy_Moore</i>	155
Rys. 5.13. Opis procesu równoległego	155
Rys. 5.14. Opis równoległego procesu za pomocą ASM.....	156
Rys. 5.15. ASM jednobitowego sumatora, odpowiadający tablicy prawdy.....	157
Rys. 5.16. Wyniki syntezy projektu <i>add_1_true_table</i> na poziomie RTL.....	158
Rys. 5.17. Wyniki symulacji projektu <i>add_1_true_table</i>	159
Rys. 5.18. ASM jednobitowego sumatora odpowiadający równaniom logicznym	159
Rys. 5.19. Wyniki syntezy projektu <i>add_1_equation</i>	160
Rys. 5.20. Wyniki symulacji projektu <i>add_1_equation</i>	160

Rys. 6.1. Sprawdzenie zmienionej wartości rejestru: a – błędna, b – poprawna	164
Rys. 6.2. ASMD opisująca algorytm mnożenia <i>a</i> jako FSMD Moore'a	165
Rys. 6.4. Wyniki symulacji projektu <i>mult_FSMD_Moore</i>	168
Rys. 6.5. ASMD do implementacji algorytmu mnożenia <i>a</i> przy użyciu automatu FSMD typu Mealy'ego	169
Rys. 6.3. Wyniki syntezy projektu <i>mult_FSMD_Moore</i> na poziomie RTL.....	172
Rys. 6.7. Wyniki symulacji projektu <i>mult_FSMD_Mealy</i>	173
Rys. 6.8. ASMD dla automatu Mealy'ego zapewniająca największą szybkość synchronicznego układu mnożącego	174
Rys. 6.10. Wyniki symulacji projektu <i>mult_FSMD_Mealy_1</i>	176
Rys. 6.6. Wyniki syntezy projektu <i>mult_FSMD_Mealy</i> na poziomie RTL	177
Rys. 6.9. Wyniki syntezy projektu <i>mult_FSMD_Mealy_1</i> na poziomie RTL.....	181
Rys. 7.1. Architektury zestawu instrukcji procesora: a – stosowa; b – z akumulatorem; c – rejestr-pamięć; d – rejestr-rejestr	187
Rys. 7.2. Architektura zestawu instrukcji mikrokontrolera PIC.....	190
Rys. 7.3. Formaty zestawu instrukcji procesora PIC	194
Rys. 7.4. Uproszczona struktura jednocyklowego procesora PIC.....	195
Rys. 7.5. Tradycyjna struktura procesora wielocyklowego.....	197
Rys. 7.6. Zaawansowana struktura wielocyklowego procesora PIC	198
Rys. 7.7. Struktura potokowego procesora PIC z czterema etapami.....	199
Rys. 7.8. Potokowa struktura procesora PIC z dwoma etapami	200
Rys. 7.9. Działanie potokowego procesora PIC z czterema etapami.....	200
Rys. 7.10. Działanie potokowego procesora PIC z dwoma etapami	201
Rys. 7.11. Struktura procesora z pamięcią podręczną instrukcji i danych.....	201
Rys. 7.12. Warunkowe działanie superskalarne procesora.....	202
Rys. 7.13. Uproszczona struktura procesora wielowątkowego	203
Rys. 7.14. Uproszczona struktura procesora wielordzeniowego	203
Rys. 8.1. Elementy pamięci jednocyklowego procesora PIC	209
Rys. 8.2. Schemat tworzenia adresu następnej instrukcji	211
Rys. 8.3. Fragment schematu układu operacyjnego do implementacji instrukcji przesyłania danych movlw , movwf i movf	212
Rys. 8.4. Instrukcji przesunięcia cyklicznego przez znak przeniesienia w lewo rlf (a) i w prawo rff (b).....	214
Rys. 8.5. Modyfikacja obwodu układu operacyjnego w celu realizacji instrukcji przesunięcia cyklicznego rlf i rff	214

Rys. 8.6. Modyfikacja wyjścia ALU w celu sprawdzenia zerowego wyniku	215
Rys. 8.7. Modyfikacja schematu generowania adresu następnej instrukcji w celu umożliwienia pominięcia kolejnej instrukcji.....	216
Rys. 8.8. Modyfikacja schematu generowania adresu następnej instrukcji w celu realizacji instrukcji return i retlw dla wyjścia z podprogramu.....	218
Rys. 8.9. Modyfikacja układu operacyjnego w celu wykonania instrukcji załadowania wartości akumulatora z pamięci danych lw i zapisania wartości akumulatora w pamięci danych sw	219
Rys. 8.10. Modyfikacja obwodu wyjściowego ALU w celu zachowania flagi z poprzedniej operacji	220
Rys. 8.11. Schemat tworzenia adresu następnej instrukcji przy implementacji instrukcji gotoz i gotonz	220
Rys. 8.12. Schemat jednocyklowego procesora PIC	221
Rys. 8.13. Uogólniona struktura jednocyklowego procesora PIC z pamięcią instrukcji i pamięcią danych.....	227
Rys. 9.1. Schemat procesora wielocyklowego.....	255
Rys. 9.2. Stan S0 wyboru kodu instrukcji	257
Rys. 9.3. Stan S1 dekodowania instrukcji	257
Rys. 9.4. Stan S2 do wykonania instrukcji movlw	258
Rys. 9.5. Stan S3 do wykonania instrukcji movwf	258
Rys. 9.6. Stan S4 do wykonania instrukcji movf	258
Rys. 9.7. Stan S5 do implementacji instrukcji formatu <i>a</i>	259
Rys. 9.8. Stan S5 do realizacji cyklicznych operacji przesunięcia.....	260
Rys. 9.9. Stan S5 do realisasji instrukcji incfsz i decfsz	260
Rys. 9.10. Stany S6 i S7 do implementacji instrukcji formatu <i>b</i>	261
Rys. 9.11. Stany S8, S9, S10 i S11 do implementacji instrukcji skoku.....	261
Rys. 9.12. Stany S12 i S13 do implementacji instrukcji sw i lw	262
Rys. 9.13. Stany S14 i S15 do implementacji instrukcji gotoz i gotonz	263
Rys. 9.14. Ogólny diagram automatu skończonego do sterowania procesorem wielocyklowym.....	263
Rys. 9.15. Wyniki symulacji procesora wielocyklowego PIC.....	270
Rys. 9.16. Schemat procesora dwucyklowego PIC	272
Rys. 9.17. Wyniki symulacji automatu skończonego dla procesora dwucyklowego PIC.....	273
Rys. 9.18. Wyniki symulacji dwucyklowego procesora PIC.....	276
Rys. 9.19. Schemat procesora czterocyklowego PIC	278
Rys. 9.20. Wyniki syntezy projektu imem_4_cycle	283

Rys. 9.21. Wyniki symulacji projektu <i>imem_4_cycle</i>	283
Rys. 9.22. Wyniki symulacji procesora czterocyklowego PIC.....	283
Rys. 10.1. Typowe zastosowanie układu DSP: $x(t)$ – wejście analogowe; $x(k)$ – wejście dyskretne; $y(k)$ – wyjście dyskretne; $y(t)$ – wyjście analogowe.....	291
Rys. 10.2. Klasyfikacja filtra według częstotliwości przepustowości: a – dolnoprzepustowy; b – górnoprzepustowy; c – pasmowoprzepustowy; d – pasmowozaporowy (wycinający)	295
Rys. 10.3. Uogólniona struktura filtra IIR (a) i filtra FIR (b)	296
Rys. 10.4. Wyniki symulacji projektu <i>FIR_lowpass</i> , w przypadku gdy na wejście dociera pojedynczy impuls o amplitudzie 1	300
Rys. 10.5. Wyniki symulacji funkcjonalnego projektu <i>FIR_lowpass</i> , gdy na wejście dociera pojedynczy impuls o amplitudzie 5.....	300
Rys. 10.6. Wyniki symulacji projektu <i>IIR_lowpass</i> , gdy na wejście dociera pojedynczy impuls o amplitudzie 1	302
Rys. 10.7. Wykres tolerancji dla filtra dolnoprzepustowego	303
Rys. 10.8. Alternatywne przedstawienie parametrów filtra dolnoprzepustowego	304
Rys. 10.9. Wybór filtra dolnoprzepustowego do projektowania w MATLAB	308
Rys. 10.10. Okno Lowpass Design Projektanta filtra dolnoprzepustowego.....	309
Rys. 10.11. Okno narzędzia wizualizacji filtra Filter Visualization Tool	310
Rys. 10.12. Zakładka Data Types okna Lowpass Design	311
Rys. 10.13. Zakładka Code Generation okna Lowpass Design	312
Rys. 10.14. Okno Generate HDL	312
Rys. 10.15. Okno Generate HDL z otwartą zakładką Global Settings	314
Rys. 10.16. Zakładka Test Bench w oknie Generate HDL	315
Rys. 10.17. Zakładka EDA Tool Scripts okna Generate HDL	315
Rys. 10.18. Pliki projektu cyfrowego filtra, automatycznie wygenerowane przez system MATLAB	316
Rys. 10.19. Ekran po uruchomieniu ModelSim	317
Rys. 10.20. Wyniki symulacji w programie ModelSim projektu filtra cyfrowego <i>FIR_lowpass_MATLAB</i> wygenerowane w systemie MATLAB	317
Rys. 10.21. Statystyka wyników syntezy projektu <i>FIR_lowpass_MATLAB</i>	318
Rys. 10.22. Fragment widoku projektu <i>FIR_lowpass_MATLAB</i> na poziomie RTL.....	319
Rys. 11.1. Fragment projektu synchronicznego	326
Rys. 11.2. Pętla kombinacyjna w sterowaniu rejestrem	328
Rys. 11.3. Tradycyjna realizacja pojedynczego impulsu	329
Rys. 11.4. Schemat do implementacji pojedynczego impulsu w FPGA	330

Rys. 11.5. Schemat do powiązania sygnału zegarowego <i>clk_int</i> generowanego przez logikę wewnętrzną z systemowym sygnałem zegarowym <i>clk</i>	331
Rys. 11.6. Schemat czterobitowego licznika tętnień	332
Rys. 11.7. Wyniki symulacji czasowej projektu <i>ripple_counter</i>	334
Rys. 11.8. Wyniki symulacji czasowej projektu <i>binary_counter</i>	334
Rys. 11.9. Multipleksowanie sygnałów zegarowych	335
Rys. 11.10. Wyniki syntezy projektu <i>clock_MUX</i> na poziomie RTL.....	337
Rys. 11.11. Wyniki symulacji czasowej projektu <i>clock_MUX</i>	337
Rys. 11.12. Bramkowanie sygnału synchronizacji za pomocą bramki AND przy użyciu zwykłych obwodów trasowania (a) i za pomocą globalnych sieci synchronizacji (b).....	339
Rys. 11.13. Bramkowanie danych z multiplekserami.....	340
Rys. 11.14. Synchronizacja sygnału bramki <i>gate</i>	341
Rys. 11.15. Wyniki syntezy projektu <i>reg_AND</i> na poziomie RTL.....	343
Rys. 11.16. Mapa technologiczna po dopasowaniu projektu <i>reg_AND</i>	344
Rys. 11.17. Wyniki symulacji czasowej projektów <i>reg_AND</i> i <i>reg_AND_GLOBAL</i>	345
Rys. 11.18. Wyniki syntezy projektu <i>reg_MUX</i> na poziomie RTL	345
Rys. 11.19. Wyniki syntezy projektu <i>reg_sync_AND</i> na poziomie RTL	346
Rys. 11.20. Mapa technologiczna po dopasowaniu projektu <i>reg_sync_AND</i>	347
Rys. 11.21. Wyniki symulacji czasowej dla projektów <i>reg_AND</i> i <i>reg_AND_GLOBAL</i>	350
Rys. 11.22. Diagramy czasowe wyników symulacji dla projektów <i>reg_MUX</i> i <i>reg_enable</i>	350
Rys. 11.23. Wyniki symulacji czasowej dla projektów <i>reg_sync_AND</i> i <i>reg_sync_AND_GLOBAL</i>	351
Rys. 11.24. Retiming rejestrów na poziomie RTL: a – przed; b – po	354
Rys. 11.25. Duplikacja rejestrów: a – przed; b – po	354
Rys. 11.26. Wyniki syntezy projektu <i>sync_reset</i> na poziomie RTL.....	360
Rys. 11.27. Wyniki symulacji czasowej projektu <i>sync_reset</i>	360
Rys. 11.28. Wyniki syntezy projektu <i>sync_reset_AND</i> na poziomie RTL	361
Rys. 11.29. Wyniki symulacji czasowej projektu <i>sync_reset_AND</i>	362
Rys. 11.30. Wyniki syntezy projektu <i>sync_reset_int</i> na poziomie RTL	363
Rys. 11.31. Wyniki symulacji czasowej projektu <i>sync_reset_int</i>	363
Rys. 11.32. Wyniki syntezy projektu <i>async_reset</i> na poziomie RTL.....	364
Rys. 11.33. Wyniki symulacji czasowej projektu <i>async_reset</i>	364
Rys. 11.34. Wyniki syntezy projektu <i>async_reset_with_follower_registers</i> na poziomie RTL.....	365

Rys. 11.35. Wyniki symulacji czasowej projektu <i>async_reset_with_follower_registers</i>	366
Rys. 11.36. Wyniki syntezy projektu <i>sync_async_reset</i> na poziomie RTL	367
Rys. 11.37. Wyniki symulacji czasowej projektu <i>sync_async_reset</i>	367
Rys. 12.1. Uogólniona struktura bloku PLL.....	375
Rys. 12.2. Struktura bloków PLL rodziny FPGA Cyclone V	376
Rys. 12.3. Przykład zależności fazowej sygnałów w trybie synchronizacji źródłowej.....	380
Rys. 12.4. Przykład zależności fazowej sygnałów w trybie kompensacji LVDS	380
Rys. 12.5. Przykład zależności fazowej sygnałów w trybie bezpośrednim.....	381
Rys. 12.6. Przykład zależności fazowej sygnałów w normalnym trybie kompensacji.....	381
Rys. 12.7. Przykład zależności fazowej sygnałów w trybie bufora z zerowym opóźnieniem ZDB.....	382
Rys. 12.8. Podłączanie portów PLL i wyprowadzeń FPGA w trybie bufora z zerowym opóźnieniem	382
Rys. 12.9. Przykład zależności fazowej sygnałów w trybie zewnętrznego sprężenia zwrotnego	383
Rys. 12.10. Okno Katalogu IP podczas tworzenia projektu bloku PLL	384
Rys. 12.11. Okno do określenia katalogu projektu i języka opisu projektu	385
Rys. 12.12. Okno Edytora parametrów z aktywną zakładką General podczas tworzenia bloku PLL.....	385
Rys. 12.13. Zakładka Clock Switchover Edytora parametrów podczas tworzenia bloku PLL.....	386
Rys. 12.14. Zakładka Cascading Edytora parametrów podczas tworzenia bloku PLL.....	387
Rys. 12.15. Zakładka MIF Streaming Edytora parametrów podczas tworzenia bloku PLL.....	387
Rys. 12.16. Zakładka Settings Edytora parametrów podczas tworzenia bloku PLL	387
Rys. 12.17. Zakładka Advanced Parameters Edytora parametrów podczas tworzenia bloku PLL.....	388
Rys. 12.18. Wyniki syntezy projektu <i>my_PLL</i> na poziomie RTL.....	388
Rys. 12.19 Wyniki symulacji czasowej projektu <i>my_PLL</i>	389
Rys. 12.20. Schemat rekonfiguracji PLL przy użyciu Bloku IP do rekonfiguracji PLL.....	391
Rys. 12.21. Struktura schematu rekonfiguracji PLL w trybie strumieniowania pliku MIF ...	393
Rys. 12.22. Zakładka General z przykładu dynamicznego przesunięcia fazowego.....	395
Rys. 12.23. Zakładka Settings z przykładu dynamicznego przesunięcia fazowego.....	396
Rys. 12.24. Wyniki syntezy projektu <i>PLL_dynamic_shifting</i> na poziomie RTL	396
Rys. 12.25. Wyniki symulacji projektu <i>PLL_dynamic_shifting</i>	396

Streszczenie

W książce przedstawiono zagadnienia projektowania systemów wbudowanych w układach FPGA, wykorzystując ich opis w języku Verilog i bloki IP pamięci różnych typów: jednoportowe i dwuportowe pamięci typu RAM, ROM, FIFO, LIFO oraz rejestry przesuwające we wbudowanych blokach pamięci. Omówiono kilka metod projektowania układów sterowania: w postaci automatu mikroprogramowanego (AM) z siecią działań, opartego na algorytmicznych automatach skończonych (algorithmic state machine – ASM), a także na algorytmicznych automatach skończonych ze ścieżką przetwarzania danych (algorithmic state machine with datapath – ASMD).

W monografii rozpatrywana jest metodyka projektowania procesora jednocyklowego na FPGA, a także trzy metodyki projektowania procesorów wielocyklowych. Szczególna uwaga zwracana jest na debugowanie procesorów i ocenę wydajności procesorów, podano też zalecenia dotyczące zwiększenia wydajności procesora. Przedstawiono ogólną metodykę projektowania filtrów cyfrowych, omówiono przykład opracowania filtra cyfrowego w systemie MATLAB, pokazano symulację filtrów w systemie ModelSim i implementację filtrów na FPGA w systemie Quartus. Uwzględniono kwestie projektowania podsystemów synchronizacji dla systemów wbudowanych na FPGA. Opisano zasady działania bloków pętli synchronizacji fazowej PLL, architekturę i funkcjonowanie bloków PLL w układach FPGA, a także sposób konfiguracji bloków PLL w projektach FPGA.

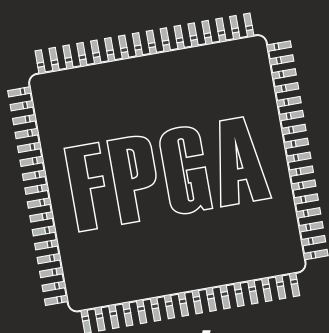
Książka przeznaczona jest dla studentów, doktorantów i nauczycieli uczelni technicznych, a także projektantów systemów wbudowanych.

Abstract

The book presents issues of designing embedded systems in FPGAs using their description in Verilog and IP memory blocks of various types: single-port and two-port RAM, ROM, FIFO, LIFO, and shift registers in embedded memory blocks. Several methods of designing control systems are discussed: in the form of a microprogrammed automaton with a graph scheme, based on algorithmic state machines (ASM), as well as on algorithmic state machines with a data processing path (ASMD).

The monograph considers the methodology for designing a single-cycle processor on FPGA and three methodologies for designing multi-cycle processors. Particular attention is paid to debugging processors and evaluating processor performance. The recommendations for increasing processor performance are also provided. The general methodology for designing digital filters is presented, an example of developing a digital filter in the MATLAB system is discussed, and the simulation of filters in the ModelSim system and the implementation of filters on FPGA in the Quartus system are shown. The issues of designing synchronization subsystems for embedded systems on FPGA are also considered. The principles of operation of PLL phase-locked loop blocks, architecture and operation of PLL blocks in FPGAs, as well as the configuration of PLL blocks in FPGA projects, are described.

The book is intended for students, doctoral students, and teachers of technical universities, as well as designers of embedded systems.



 Politechnika
Białostocka

