POLITECHNIKA ŚLĄSKA

WYDZIAŁ AUTOMATYKI, ELEKTRONIKI I INFORMATYKI

Praca dyplomowa magisterska

Implementacja SoC na podstawie mikroprocesora RISC-V Ibex

Autor: inż. Dawid Zimończyk

Kierujący pracą: dr hab. inż. Robert Czerwiński, prof. Pol. Śl.

Gliwice, listopad 2020

Spis treści

Spis rysunków

Spis ważniejszych oznaczeń

SoC - System on Chip
ISA - Instruction Set Architecture
RISC - Reduced Instruction Set Computing
UVM - Universal Verification Methodology
I2C - Inter-Integrated Circuit
SPI - Serial Peripheral Interface
UART - Universal asynchronous receiver-transmitter
RAM - Random-access memory
PWM - Pulse-Width Modulation
GPIO - General-purpose input/output
RoT - Root of trust
FPGA - Field-programmable gate array
ISS - Instruction set simulator
SV - SystemVerilog
DV - Design verification
ISP - In-System Programming
JTAG - Joint Test Action Group
PC - Program counter
LSB - Least significant bit
MSB - Most significant bit
IP - Intellectual property
TLM - Transaction Level Modeling
DUT - Device under test
TCL - Tool Command Language
HDL - Hardware description language
PLL - Phase-locked loop
Pmod - Peripheral Module interface
ALU - Arithmetic Logic Unit
LUT - Look-up Table
BRAM - Blocks RAM

1 Wprowadzenie

1.1 Wstęp

Systemy na chipie znane również jako SoC, występują między innymi w naszych telefonach czy samochodach. Również są częścią systemów wbudowanych, te zaś są wykorzystywane w każdej dziedzinie życia, od zegarków elektronicznych po zaawansowane roboty medyczne. Ważne jest więc by układy te były niezawodne i działały w zamierzony sposób. By zweryfikować działanie zaimplementowanego systemu na chipie wykorzystano symulator języków opisu sprzętu Riviera-PRO.
SoC powinien składać się z mikroprocesora, mikrokontrolera lub rdzenia DSP. Każdy mikroprocesor posiada 'Model programowy procesora' (ang Instruction Set Architecture, ISA). Model ISA definiuje jak mikroprocesor powinien działać, jego listę rozkazów, typ danych, tryby adresowania, rejestry dostępne dla programisty, zasady obsługi przerwań i wyjątków. Przykładowe komercyjne modele ISA to: ARM, MIPS. Jest również otwarty model programowy procesora, który jest oparty o zasady RISC, jest nim RISC-V. Otwarty standard ISA oznacza, że dostęp nie jest limitowany prawnie, finansowo lub tajemnicą handlową firmy.
Przykładem mikroprocesora wykorzystującego architekturę RISC-V jest Ibex. Jest on tworzony przez stowarzyszenie lowRISC, powstałym na Uniwersytecie w Cambridge. Mikroprocesor ten jest 32-bitowy, składa się z 2-poziomowego potoku i został zaimplementowany na bazie RV32IMC.

1.2 Cel i zakres pracy

Celem pracy jest implementacja systemu na chipie z użyciem mikroprocesora Ibex RISC-V. Praca obejmuje zadania związane z przystosowaniem mikroprocesora Ibex RISC-V do implantacji na płycie uruchomieniowej NEXYS4DDR wraz implementacją modułów peryferyjnych. Zakres pracy obejmuje również weryfikację zaimplementowanego systemu z wykorzystaniem metodyki UVM (wer. 1.2).
Szczegółowy zakres pracy obejmuje:

- implementację mikroprocesora Ibex RISC-V,
- implementację modułów peryferyjnych, w skład których wchodzą:
    1. pamięć RAM,
    2. interfejs SPI,
    3. interfejs I2C,
    4. interfejs UART,
    5. moduł wejść/wyjść GPIO,

6. moduł Timera,

- przeprowadzenie syntezy systemu oraz w rozbiciu na poszczególne moduły,
- kompilację kompilatora skrośnego i przystosowanie go dla opracowanego systemu,
- przeprowadzenie weryfikacji,
- napisanie pracy.

1.3 Zarys pracy

Praca składa się z sześciu rozdziałów. Pierwszy zawiera krótkie omówienie tematu pracy, jej celu i zarys. Drugi rozdział jest poświęcony teorii. Opisuje on zagadnienia związane z ISA RISC-V, SoC, mikroprocesorem Ibex, kompilatorem, weryfikacją, płytce FPGA Nexys4 DDR i programem wykorzystanym do syntezy oraz programem do symulacji. Trzeci rozdział skupia się na implementacji poszczególnych części systemu na chipie, przedstawione zostaną w nim fragmentu opisu sprzętu i schematy blokowe. Czwarty rozdział przedstawia weryfikacje, opisuje testy z wykorzystaniem biblioteki UVM 1.2 oraz jej wyniki. Następnie pokazuje symulację przeprowadzaną z instrukcjami wygenerowanymi przez RISCV-DV. Szósty rozdział przedstawia wyniki syntezy poszczególnych modułów oraz eksperymenty związane ze zmianą parametrów syntezy. Ostatni rozdział to podsumowanie.

2 Część teoretyczna

2.1 RISC V

2.1.1 Specyfikacja ISA

Idea RISC-V jest związana z otwartym ISA bazująca na architekturze RISC. Oznacza to, że licencja jest typu Open-source. Pozwala na wprowadzanie dowonlych modyfikacji[1], również jest nie wymaga żadnych opłat za wykorzystywanie jej w komercyjnych celach. Dokumentacja składa się z trzech części[2]:

1. User-Level ISA Specification - specyfikacja ISA poziomu użytkownika.
2. Privileged ISA Specifiation - specyfikacja ISA trybu przywilejów.
3. Debug Specification - specyfikacja *debugowania*.

Podstawowe cechy architektury RISC to:

- zredukowana lista rozkazów - jest ich kilkadziesiąt,
- przepustowość procesora zbliżona do jednej instrukcji na cykl,
- zredukowana tryby adresowania, kody rozkazów są prostsze,
- minimalizacja komunikacji między procesorem a pamięcią,
- instrukcje mogą operować na dowolnych rejestrach,
- instrukcje zajmują w pamięci taką samą liczbę bajtów,
- procesor posiada architekturę Harwardzką,
- procesor używa przetwarzania potokowego

Są cztery podstawowe zestawy instrukcji oraz piętnaście ich rozszerzeń. W tabeli 1 przedstawiono ich podział.

Tabela 1: Podstawowa specyfikacja ISA i jej rozszerzenia[3]

Nazwa	Opis
Podstawowe	
RV32I	Base Integer Instruction Set, 32-bit
RV32E	Base Integer Instruction Set (embedded), 32-bit, 16 registers
RV64I	Base Integer Instruction Set, 64-bit
RV128I	Base Integer Instruction Set, 128-bit
Rozszerzenia	
M	Standard Extension for Integer Multiplication and Division
A	Standard Extension for Atomic Instructions
F	Standard Extension for Single-Precision Floating-Point
D	Standard Extension for Double-Precision Floating-Point
G	Shorthand for the base and above extensions
Q	Standard Extension for Quad-Precision Floating-Point
L	Standard Extension for Decimal Floating-Point
C	Standard Extension for Compressed Instructions
B	Standard Extension for Bit Manipulation
J	Standard Extension for Dynamically Translated Languages
T	Standard Extension for Transactional Memory
P	Standard Extension for Packed-SIMD Instructions
V	Standard Extension for Vector Operations
N	Standard Extension for User-Level Interrupts
H	Standard Extension for Hypervisor

Instrukcje są 32-bitowe. Tabela 2 przedstawia formaty tych instrukcji. Korzystają one z sześciu trybów adresowania:

- Register (R) - instrukcje realizują działania na dwóch rejestrach *rs1* i *rs2*, wynik jest zapisywany w rejestrze *rd*.
- Immediate (I) - instrukcje realizują działania rejestrze *rs1* i liczbie 12-bitowej stałej ze znakiem, wynik jest zapisywany w rejestrze *rd*.
- Upper immediate (U) - format wykorzystywany dla dwóch instrukcji: *LUI*, *AUIPC*. Służy do przypisywania liczb 20-bitowych do rejestru *rd*.
- Store (S) - instrukcje realizują zapis do pamięci, pobierany jest bazowy adres z rejestru *rs1* + offset pochodzący z *imm*, rejestr *rs2* przechowuje.
- Branch (SB) - instrukcje realizują skoki warunkowe.
- Jump (UJ) - instrukcje służące do skoków, dodają wartość *imm* do *PC*.

Każda instrukcja posiada kod operacji, jest to fragment rozkazu przekazywana do rdzenia. Informuje on jaka operacja powinna być wykonana. W instrukcjach architektury RISC-V liczba na zajmuje siedem najmniej ważnych bitów.

Tabela 2: 32-bit RISC-V formaty instrukcji[3]

Format	Bit																													
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	funct7							rs2					rs1					funct3			rd					opcode						
I	imm[11:0]												rs1					funct3			rd					opcode						
U	imm[31:12]																				rd					opcode						
S	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						
SB	[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	opcode						
UJ	[20]	imm[10:1]										[11]	imm[19:12]								rd					opcode						

2.1.2 Rejestry

RISC-V posiada 32 rejestry (tryb *embeded* posiada tylko 16). Jeśli korzystamy z rozszerzenia zawierającego liczby zmiennoprzecinkowe, dodane zostają kolejne 32 rejestry. Pierwszy rejestr nazywany jest rejestrem zerowym. Zawsze przyjmuje wartość zera, a wszystkie dane zapisywane do niego są tracone. Służy on jako rejestr pomocniczy w wielu instrukcjach.

Tabela 3: Rejestry RISC-V[3]

Nazwa rejestru	Nazwa symboliczna	Opis	Właściciel
x0	Zero	zawsze zero	
x1	ra	adres powrotu	wywołujący
x2	sp	wskaźnik stosu	wywołany
x3	gp	wskaźnik globalny	
x4	tp	wskaźnik wątku	
x5	t0	zmienna tymczasowa / alternatywny adres powrotu	wywołujący
x6-7	t1-2	zmienne tymczasowe	wywołujący
x8	s0/fp	zapisany rejestr / wskaźnik ramki	wywołany
x9	s1	zapisany rejestr	wywołany
x10-11	a0-1	argument funkcji / wartość zwracana	wywołujący
x12-17	a-2-7	argument funkcji	wywołany
x18-27	s2-11	zapisane rejestry	wywołany
x28-31	t3-6	zmienne tymczasowe	wywołujący
32 rejestry dla zmiennoprzecinkowego rozszerzenia			
f0-7	ft0-7	tymczasowe zmienne zmiennoprzecinkowe	wywołujący
f8-9	fs0-1	zapisane rejestry zmiennoprzecinkowe	wywołany
f10-11	fa0-1	argumenty/wartość zwaracana zmiennoprzecinkowe	wywołujący
f12-17	fa2-7	argumenty zmiennoprzecinkowe	wywołujący
f18-27	fs2-11	zapisane rejestry zmiennoprzecinkowe	wywołujący
f28-31	fs8-11	tymczasowe zmienne zmiennoprzecinkowe	wywołujący

2.1.3 Dostęp do pamięci

Dostęp do pamięci odbywa się za pomocą instrukcji *load/store*. W instrukcjach *load* adres bazowy znajduje się w rejestrze *rs1*, offset jest pobierany z liczby całkowitej 12-bitowej *imm*. Rejestr docelowy znajduje się w *rd*. Przykład działania instrukcji *LW*:

```
lw x16, 8(x2)
```

imm[11:0]	rs1	func3	rd	opcode
offset[11:0]	base_addr	width	dst_addr	LOAD
000000001000	00010	010	10000	0000011
imm=+8	rs1=2	LW	rd=16	LOAD

Wartość w *funct3* służy do dekodowania rozmiaru i znaku ładowanej wartości. Wartość ta jest zależna od użytego rozkazu, tabela 4 przedstawia zależność między instrukcją a wartością *func3*.

Tabela 4: Zależność między *func3* a instrukcją *load*[3]

func3	instrukcja
000	LB
001	LH
010	LW
100	LBU
101	LHU

Kolejnymi instrukcjami są rozkazy *store*. Potrzebują one dwóch rejestrów, rejestr *rs1* zawiera bazowy adres pamięci, natomiast do rejestru *rs2* zostanie ona przypisana. Wartość offsetu jest pobierana z *imm*. Przykład działania instrukcji *SW*:

`sw x16, 8(x2)`

imm[11:5]	rs2	rs1	func3	imm[4:0]	opcode
offset[11:5]	store_addr	base_addr	width	offset[4:0]	STORE
00000000	10000	00010	010	01000	0100011
imm[11:0]=+8	rs2=16	rs1=2	SW		STORE

Podobnie jak w instrukcjach *load funct3* służy dekodowania rozmiaru i jest zależna od przekazanego rozkazu. Tabela 5 przedstawia tą zależność.

Tabela 5: Zależność między *func3* a instrukcją *store*[3]

func3	instrukcja
000	SB
001	SH
010	SW

2.1.4 Instrukcje arytmetyczne i logiczne

RISC-V zawiera zestaw instrukcji matematycznych przeznaczony dla liczb całkowitych, w którego skład wchodzą: dodawanie, odejmowanie, przesuwanie, operacje logiczne i porównywanie liczb. Instrukcje dla mnożenia i dzielenia liczb znajdują się w rozszerzeniu ISA *M*. Rozszerzenie ISA *F* zawiera instrukcje matematyczne dla liczb zmiennoprzecinkowych pojedynczej precyzji, rozszerzenie *D* zawiera instrukcje matematyczne dla liczb zmiennoprzecinkowych podwójnej precyzji[3]. Instrukcje te wykorzystują format *R* i *I*. Na przykład działanie rozkazu *add* wykorzystuje format instrukcji *R*:

`add x6, x7, x8`

funct7	rs2	rs1	func3	rd	opcode
0000000	01000	00111	000	00110	0110011

Pierwszy argument jest zalokowany w rejestrze *rd*, kolejny *rs1* ostatni w *rs2*. Pola *Funct7* i *funct3* służą do rozpoznania operacji i są one zależne od przekazanej instrukcji. Tabela 6 przedstawia te zależności.

Tabela 6: Zależność między *func7* i *func3* a instrukcjami arytmetycznymi[3]

func7	func3	OPCODE	instrukcja
0000000	000	0110011	ADD
0100000	000	0110011	SUB
0000000	001	0110011	SLL
0000000	010	0110011	SLT
0000000	011	0110011	SLTU
0000000	100	0110011	XOR
0000000	101	0110011	SRL
0100000	101	0110011	SRA
0000000	110	0110011	OR
0000000	111	0110011	AND

Instrukcja *addi* wykorzystuje format *I*, więc trzeci argument rozkazu jest liczbą całkowitą. Przykład tej instrukcji ma następujący format:

```
addi x6, x0, 50
```

imm[11:0]	rs1	func3	rd	opcode
000000110010	00000	000	00110	0010011

Pole *func3* jest wykorzystywane w celu dekodowania instrukcji. Rozkazy przesunięcia bitowego wykorzystują pięć najmniej znaczących bitów z *imm*. Siedem pozostałych bitów służy do rozpoznawania instrukcji.

2.1.5 Instrukcje skokowe

Instrukcje skokowe dzielą się na dwa rodzaje: skoki bezwarunkowe i skoki warunkowe. Pierwszą z nich reprezentują dwa rozkazy: *JAL* (format *UJ*) i *JALR* (format *I*). Pierwszy z nich pozwala dodać do rejestru *PC* liczbę ze znakiem o szerokości 20-bitów. Dzięki rozkazowi *JALR* i *AUIPC* można stworzyć skok o szerokości 32-bitów. Rozkaz *AUIPC* zapisuje do rejestru aktualną wartość *PC*, a rozkaz *JALR*, zamienia dwanaście najmniej znaczących bitów na wartość przekazanego argumentu. Przykładowe programy z użyciem instrukcji skoków bezwarunkowych są następujące:

```
addi x31, x0 ,0
auipc x2, 0
addi x31, x31, 1
addi x31, x31, 2
jalr x1, x2, 8
```

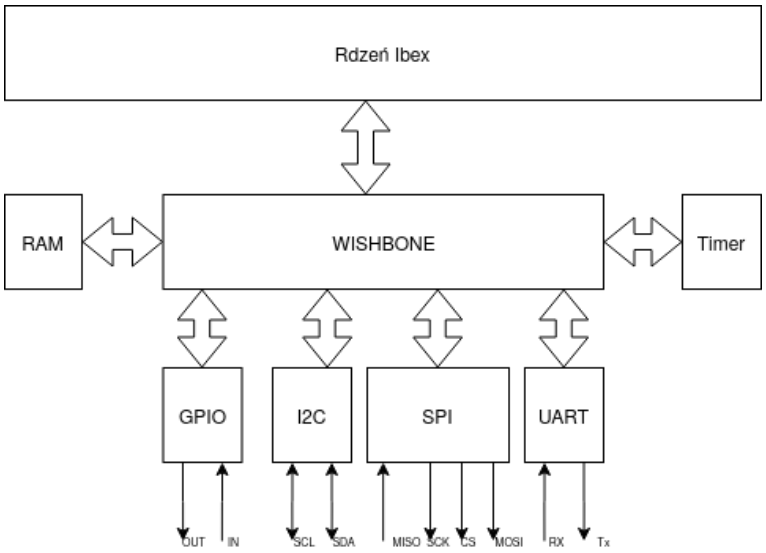
Program wpisuje do rejestru *x2* aktualną wartość *PC*, następnie po wykonaniu dwóch instrukcji *addi* następuje rozkaz *jalr*, który dodaje wartość 8 do zapisanej wartości *PC*, więc kolejnym rozkazem wykonanym będzie *addi x31, x31, 2*.
Kolejnym rodzajem są skoki warunkowe, jest ich sześć i są zakodowane w formacie *SB*:

- BEQ - gdy zapisane liczby w rejestrach są równe to wykonuje skok,
- BNE - gdy zapisane liczby w rejestrach są różne to wykonuje skok,
- BLT - gdy liczba z rejestru *rs1* jest większa to wykonuje skok,
- BLTU - gdy liczba z rejestru *rs1* jest większa bądź równa to wykonuje skok,
- BHE - gdy liczba z rejestru *rs2* jest większa to wykonuje skok,
- BGEU - gdy liczba z rejestru *rs2* jest większa to wykonuje skok,

2.2 System na Chipie

2.2.1 Architektura systemu

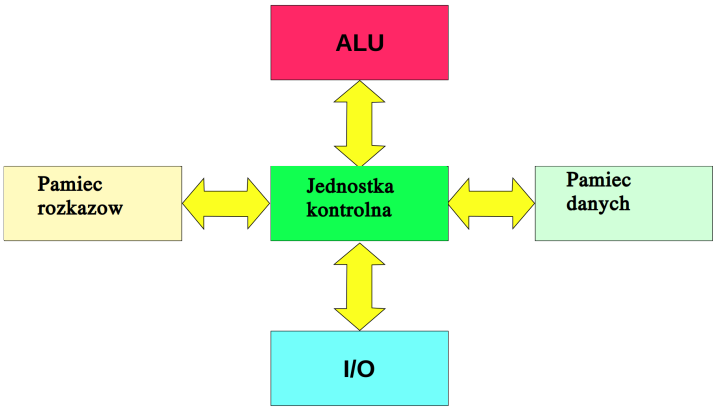
W celu realizacji projektu wybrano architekturę Harvardzką. Schemat blokowy przedstawiono na rysunku 1.



Rysunek 1: Schemat blokowy projektowanego systemu na chipie

2.2.2 Architektura Harvardzka

Architektura Harvardzka to rodzaj architektury komputera, która została wybrana w celu realizacji systemu. Posiada ona dwie oddzielne magistrale dla danych i rozkazów. Można w tym samym czasie pobierać argument wykonywanej funkcji i pobierać następny rozkaz. Zwiększa ta szybkość pracy. Rysunek 2 przedstawia schemat blokowy tej architektury.



Rysunek 2: Schemat blokowy architektury Harvardzkiej

2.2.3 Peryferia

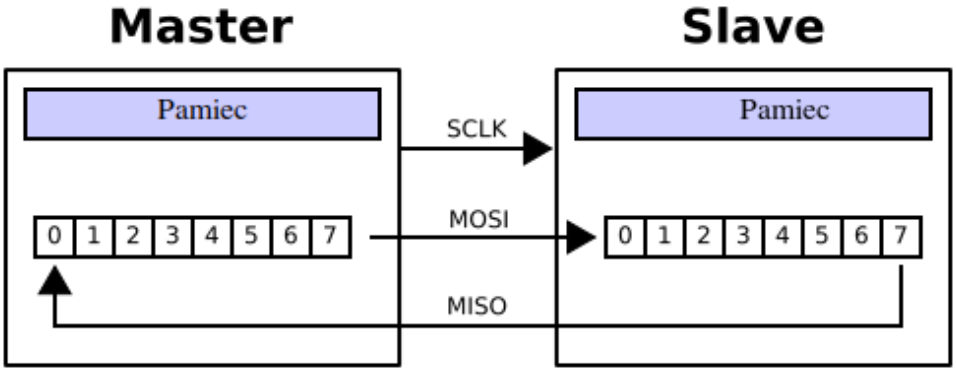
W projekcie została dodana pamięć RAM i następujące peryferia:

1. RAM - pamięć o dostępnie swobodnym, jest to podstawowy rodzaj pamięci cyfrowej. Może być ona odczytywana i zmieniana w dowolnej kolejności. Służy ona do przechowywania danych i kodu maszynowego. W projekcie zaimplementowano pamięć jedno-portową i dwu-portową. Pamięć jedno-portowa posiada tylko jedną parę sterujących portów, więc może być czytana lub zapisywana w jednej chwili czasu. Pamięć dwu-portowa zawiera dwie pary portów sterujących, więc może być czytana i zapisywana w jednej chwili czasu.[4]

2. SPI - interfejs służący do transmisji, głównie używany w systemach wbudowanych. Wykorzystuje się tryb *master-slave*, dzięki temu jest zapewniona komunikacja full-duplex. Interfejs ten posiada następujące porty:

   - $SCLK$ - zegar, wyjście z mastera,
   - $MOSI$ - *Master Out Slave In*,
   - $MISO$ - *Master In Slave Out*,
   - $\overline{SS}$ - *Slave Select*

By rozpocząć transmisje, *Master* konfiguruje *SCLK*, następnie ustawia stan niski na linii *SS* w celu wybrania odpowiedniego *Slave'a*. *Master* wysyła bit poprzez *MOSI* i *slave'a* odczytuje go i wysyła bit poprzez *MISO*. Rysunek 3 obrazuję przebieg transmisji[5].



Rysunek 3: Przykład transmisji SPI[17]

3. I2C - magistrala szeregowa, dwukierunkowa, synchroniczna służąca do komunikacji. Wykorzystuje tryb *master-slave*. Posiada dwa porty:

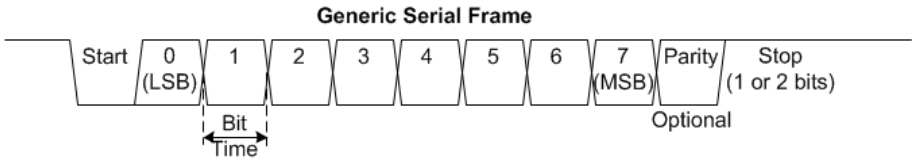
   - SDA - Linia dla *mastera* i *slave'a* służąca do komunikacji między nimi,
   - SCL - linia przenosząca sygnał zegarowy.

   I2C może pracować z wieloma *slave'ami* i *masterami*. By rozpocząć transmisję,*master* wysyła sygnał startowy. By to uzyskać sygnał na linii *SDA* zmienia się z wysokiego na niski przed zmianą sygnały z wysokiego na niski na linii *SCL*. Następnie jest przesyłany adres *slave'a*. *Slave* porównuje nadesłany adres i odsyła bit *ACK* ustawiając na linii *SDA* bit na stan niski. Po każdej udanej transmiji *slave* przysyła *masterowi* bit *ACK*. W celu zakończenia transmisji należy w czasie wysokiego stanu *SCL* zmienić stan z niskiego na wysoki na linii *SDA*[6].

4. UART - urządzenie służące do asynchronicznej szeregowej komunikacji. Odbiera, jak i wysyła informacje poprzez port szeregowy. Zawiera on on konwertery:

   - szeregowo-równoległy - do konwersji danych wysyłanych do komputera,
   - równoległy-szeregowy - do konwersji danych pochodzących z komputera.

   Rysunek 4 przedstawia ramkę UARTu. Bit parzystości jest opcjonalny i służy jako bit kontrolny[7].



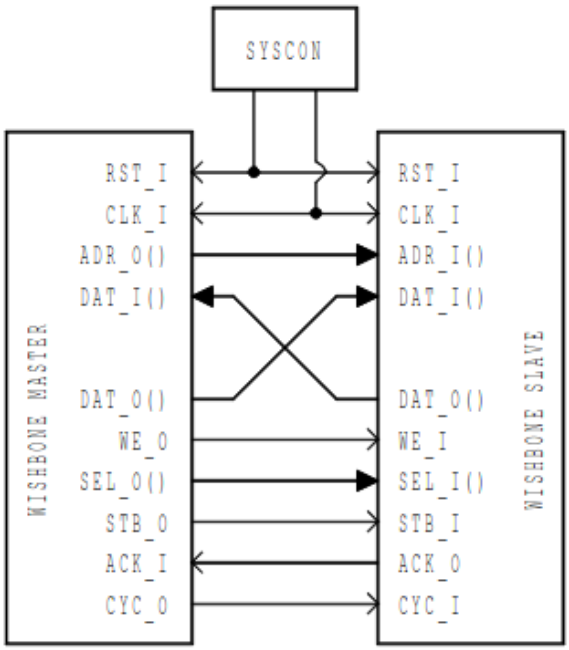
Rysunek 4: Ramka UART[18]

5. GPIO - wyprowadzenia służące do komunikacja między mikroprocesorem a peryferiami[8].

6. Timer.

2.2.4 Wishbone

Wishobone to darmowa magistrala służąca do łączenia ze sobą wielu modułów w systemie *master/slave*. Rysunek 5 przedstawia połączenia w tym interfejsie.



Rysunek 5: Interfejs Wishbone master/slave[9]

Podczas implementacji tej magistrali należy trzymać się zasad które definiuje standard:

- wszystkie sygnały interfejsu muszą być aktywne w wysokim stanie,
- wszystkie interfejsy *WISHOBONE* muszą zainicjować siebie podczas stanu wysokiego sygnału *RST_I*. Muszą zostać zainicjowane aż do narastającego zbocza *CLK_I*, której następuje po negacji *RST_I*,
- *RST_I* musi pozostać przynajmniej przez jeden pełny cykl zegarowy w stanie wysokim,
- wszystkie interfejsy *WISHBONE* muszą być przygotowane na reakcję na *RST_I* w każdym momencie,
- *RST_I* może pozostać w stanie wysokim dłużej niż jeden cykl zegarowy.

Porty używane przez ten interfejs obejmują[10]:

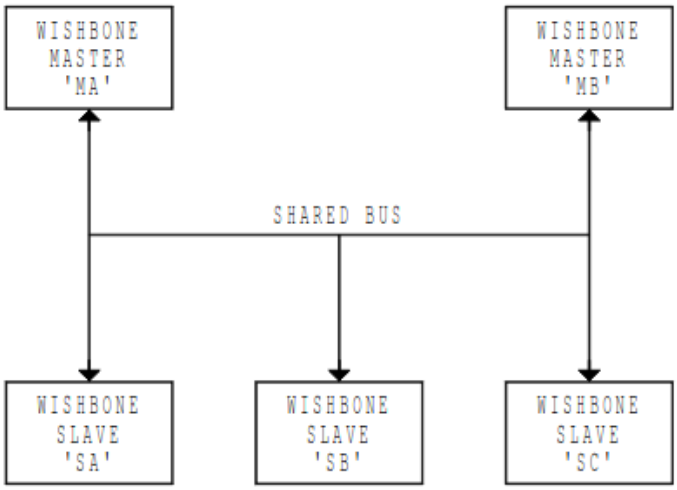
- *RST_I* - sygnał resetu otrzymywany z *SYSCON*,

- *CLK_I* - sygnał zegarowy otrzymywany z *SYSCON*,
- *ADR_O/I* - linia adresu, wyjście z *mastera*, wejście do *slave'a*,
- *DAT_I/O* - linia danych,
- *WE_O/I* - pozwolenie na zapis, wyjście z *master*, wejście do *slave*,
- *SEL_O/I* - selekcja bajtu, wyjście z *master*, wejście do *slave*,
- *STB_O/I* - potwierdzenie nadania danych przez *mastera*, wyjście z *master*, wejście do *slave*,
- *ACK_I/O* - potwierdzenie przyjęcia danych przez *slave'a*, wyjście z *slave*, wejście do *master*,
- *CYC_O/I* - cykl magistrali, wyjście z *master*, wejście do *slave*.

Są dostępne trzy topologie:

1. Data Flow Interconnection.
2. Crossbar Switch Interconnection.
3. Shared Bus Interconnection.

Ostatnia topologia została użyta w projekcie. Ma ona miejsce gdy wiele peryferii typu *slave* jest dołączonych do tych samych *masterów*. Rysunek 6 przedstawia przykład tej topologii.



Rysunek 6: Wishbone wspólne połączenie magistrali[9]

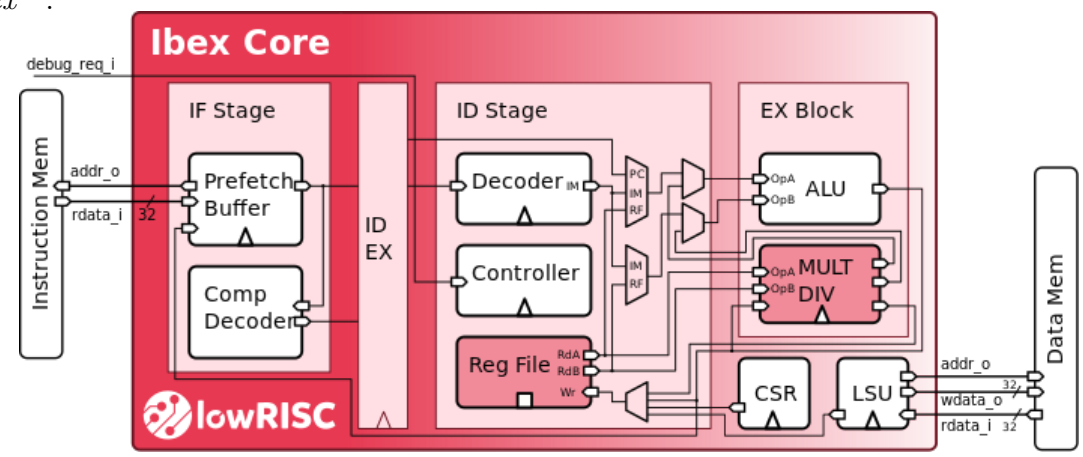
W celu rozpoznania odpowiedniego *slave'a* przypisuje im się adresy. Adresy te tworzą mapę. Opis tejże mapy znajduje się w rozdziale 3.2.

2.3 Ibex

Ibex jest to mikroprocesor tworzony przez organizację *LowRISC*. Jest on dwupotokowy:

1. Pobieranie instrukcji - pobiera instrukcje z pamięci.
2. Dekodowanie i wykonanie instrukcji - zdekodowanie pobranej instrukcji i natychmiastowe jej wykonanie

Implementuje on *ISA RV32IMC*. Wspiera on również rozszerzenie *E* i eksperymentalne *B*. Można je włączyć poprzez prawidłowe ustawienie parametrów[11]. Dla rdzenia została stworzona obszerna weryfikacja, wykorzystuje ona między innymi generator rozkazów *RISCV-DV*. Jest on również częścią projektu *OpenTitan*, jest to *RoT*, wspierany między innymi przez *Google*[12]. Rysunek 7 przedstawia schemat blokowy mikroprocesora *Ibex*[11].



Rysunek 7: Schemat blokowy mikroprocesora[11]

2.4 Kompilator

2.4.1 Budowanie kompilatora skrośnego

Kompilator skrośny można pobrać z oficjalnego repozytorium *RISC-V*[13]. By zbudować kompatibilną wersję kompilatora dla mikroprocesora *Ibex*, należy do konfiguracji podać argumenty *–with-abi=ilp32 –with-arch=rv32imc –with-cmodel=medany* lub skorzystać z *–multilib*. Opcja ta spowoduje zbudowanie kompilatora dla 64-bit, lecz po podaniu odpowiednich argumentów podczas kompilacji programu wspiera również architektury 32-bit.

2.4.2 Przykładowa kompilacja

By skompilować przykładowy program dla mikroprocesora *Ibex* należy użyć następujących komend:

Listing 1: Przykładowa kompilacja

```
riscv32-unknown-elf-gcc -march=rv32imc -mabi=ilp32 -static -mcmodel=medany -nostdlib \
-nostartfiles -Wall -g -Os -MMD -c  -o led.o led.c

riscv32-unknown-elf-gcc -march=rv32imc -mabi=ilp32 -static -mcmodel=medany -nostdlib \
-nostartfiles -Wall -g -Os -MMD -c  -o crt0.o crt0.S

riscv32-unknown-elf-gcc -march=rv32imc -mabi=ilp32 -static -mcmodel=medany -nostdlib \
-nostartfiles -Wall -g -Os -T link.ld led.o crt0.o -o led.elf

riscv32-unknown-elf-objcopy -O binary led.elf led.bin

srec_cat led.bin -binary -offset 0x0000 -byte-swap 4 -o led.vmem -vmem

riscv32-unknown-elf-objcopy -O verilog --interleave-width=4 \
--interleave=4 --byte=0 led.elf led.hex
```

Pierwsze dwie komendy tworzą biblioteki, trzecia komenda spaja ze sobą potrzebne biblioteki i konsolidatora i tworzy plik *bin*. Następnie plik *bin* jest konwertowany do plików *vmem* i *hex*.

2.5 Weryfikacja

2.5.1 UVM

UVM jest to biblioteka oparta na języku *SystemVerilog* służąca do tworzenia testów weryfikacyjnych. UVM zawiera bazowe klasy z metodami, które pomagają w weryfikacji. Ważniejsze klasy bazowe biblioteki:

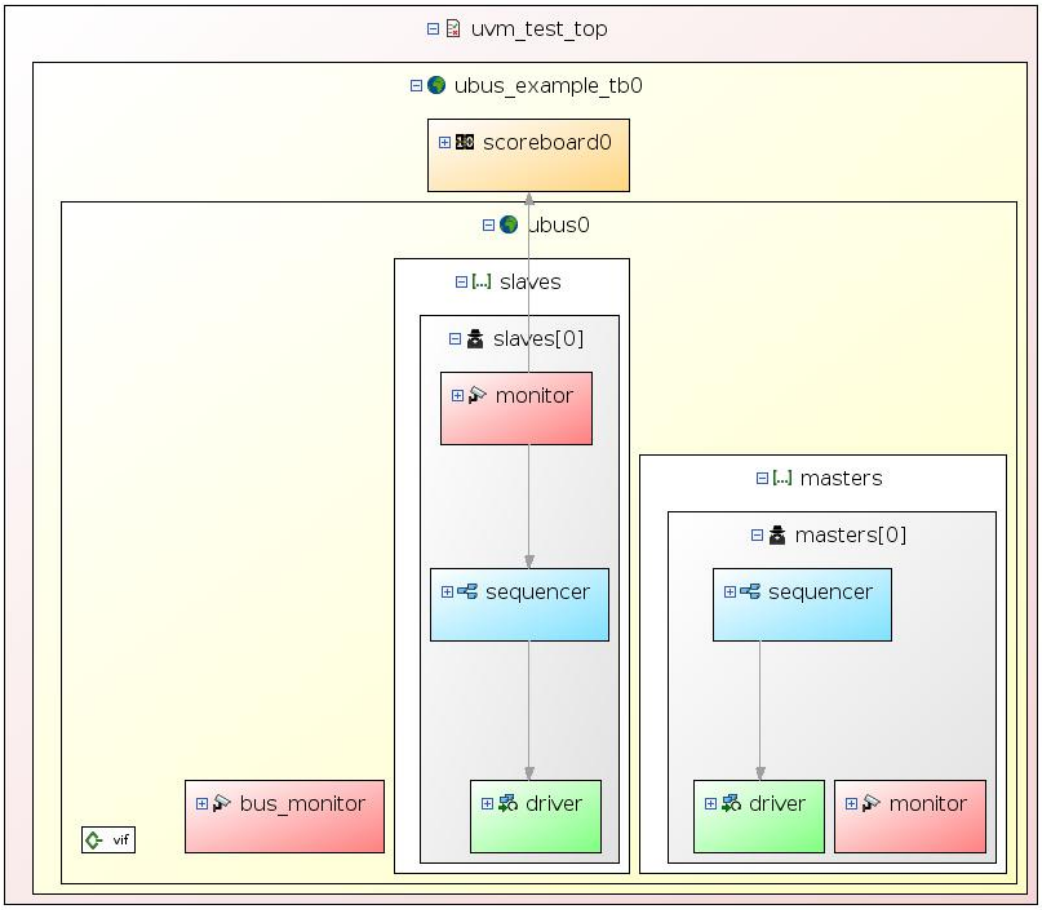
1. uvm_object - podstawowa klasa bazowa, zawierająca metody: *create, copy, clone, compare, print, record*. Zazwyczaj używana do budowy architektury testu,
2. uvm_component - wszystkie komponenty architektury testu takie jak *scoreboards, monitor, driver* pochodzą z tej klasy.
3. uvm_sequence - jest klasą bazową wszystkich sekwencji zawartych w teście.

UVM test składa się z następujących elementów:

- UVM test - jest odpowiedzialny za konfigurację testu, rozpoczęcie symulacji poprzez inicjalizację sekwencji, stworzenie wszystkich komponentów, której znajdują się poniżej w hierarchii na przykład: *uvm_env*,
- UVM env - grupuje *uvm_agent* i *uvm_scoreboard*
- UVM Agent - łączy ze sobą *uvm_components* na przykład:*uvm_driver, uvm_monitor, uvm_suquence, uvm_sequencer* za pomocą interfejsów TLM,
- UVM Driver - jest odpowiedzialny za wysyłanie pakietów do DUT,

- UVM Sequence - generuje pakiety,
- UVM Sequencer - jest odpowiedzialny za ruch między *uvm_sequence* i *uvm_driver*,
- UVM Monitor - obserwuje sygnały, następnie wysyła je do *uvm_scoreboard*,
- UVM Scoreboard - odbiera dane z *uvm_monitor* i porównuje z spodziewanymi wartościami. Wartości te mogą pochodzić z modelu referencyjnego lub *golden pattern*.

Rysunek 8 przedstawia przykładowy graf UVM testu.



Rysunek 8: Przykładowy graf UVM

2.5.2 RISCV DV

RISCV-DV to narzędzie służące do generacji programów w języku assembler do testowania danych aspektów procesora. Współpracuje z ISA: *RV32IMAFDC, RV64IMAFDC*. Programy są tworzone losowo. By korzystać z tego nardzędzia należy posiadać symulator wspierający UVM, na przykład: Riviera-PRO[14].

2.6 FPGA

SoC jest implementowany na płytce NEXYS4 DDR wyposażony w programowalny układ logiczny Artix-7 XC7A100T-1CSG324C. Ważniejsze zasoby płytki:[15]

- 15850 plastrów logicznych, każdy złożony z czterech elementów LUT o 6-wejściach i 8 przerzutników,
- Pojemność 4860 kb szybkiego bloku pamięci RAM,
- Sześć bloków zarządzania sygnałem zegarowym (CMT), każdy z pętlą fazową (PLL),
- 240 plastrów DSP,
- 16 przełączników użytkownika,
- Mostek USB-UART,
- Port USB-JTAG Digilent do komunikacji i programowania FPGA,
- Cztery porty Pmod,
- 100MHz rezonator kwarcowy.

2.7 SystemVerilog

Język opisu sprzętu, jest rozszerzeniem języka Verilog. Dodaje on nowe typy danych: *logic, enum, byte, shortint, int, longint, struct, union*, wielowymiarowe tablice. Dodano również nowe bloki proceduralne: *always_comb, always_latch, always_ff*. Wprowadzono interfejsy wraz z *modportami*, pomagają one zapanować nad portami w projekcie. Udoskonalono weryfikację poprzez dodanie nowego typu danych: *string*, klas, asercji oraz *constrained random generation* pozwalający narzucić ograniczenia podczas randomizacji[16]. Narzędziem syntezującym ten język jest Vivado Design Suite - oprogramowanie firmy Xilinx dla syntezy i analizy projektów HDL. Posiada wbudowany symulator *ISIM* oraz *Vivado IP Integrator* pozwalający na szybkie zarządzanie IP. Programem ułatwiającym pracę jest Riviera-PRO komercyjny symulator HDL firmy Aldec. Obsługuje on bibliotekę UVM, randomizacje, asercje oraz może być wykorzystany do generacji programów assembler w celu weryfikacji działania systemu na chipie. Narzędziem które generuje skryptu dla kompilacji jak i syntezy jest Bender.

3 Implementacja

3.1 System na chipie

Modułem głównym projektu jest *ibex_soc*. Nazwy jego portów, parametru i ich przeznaczenie zostały przedstawione w tabeli 7

Tabela 7: Porty i parametry modułu *ibex_soc*

typ parametru/kierunek portu	nazwa parametru / portu	przeznaczenie
localparam	SPI_SLAVE_NUMBER	ilość portów SS SPI
input	I_CLK	wejście sygnału zegarowego
input	I_RST_N	wejście sygnału resetu
output	O_LED	wyjście GPIO
input	I_BTM	wejście GPIO
input	I_UART_RX	wejście UART receive
output	O_UART_TX	wyjście UART transmit
inout	IO_SDA	dwukierunkowa linia danych I2C
inout	IO_SCL	dwukierunkowa linia zegara I2C
input	I_MISO	wejście Master In Slave Out SPI
input	I_MOSI	wejście Master Out Slave In SPI
output	O_MOSI	wyjście Master Out Slave In SPI
output	O_MISO	wyjście Master In Slave Out SPI
output	O_SCK	wyjście linii zegara SPI
input	I_SCK	wejście linii zegara SPI
input	I_CS	wejście wyboru slave SPI
output	O_CS	wyjście wyboru slave SPI

Parametr *SPI_SLAVE_NUMBER* definiuje ilość wyjść wybory slave. Wejście sygnału zegarowego zostało podłączone do rezonatora kwarcowego o częstotliwości 100MHz. Wejście resetu zostało podłączone do przełącznika znajdującego się na płytce FPGA, jest on aktywny w stanie niskim. Sygnały *GPIO* zostały podłączone do diod LED oraz przełączników. Sygnały *UART* zostały podłączone do znajdującego się na płytce konwertera *USB-UART*. Pozostałe sygnały zostały połączone z portami *Pmod*.
Tabela 8 przedstawia nazwy modułów i odpowiadające im instancje, zainicjowane w *ibex_soc*.

Tabela 8: Instancje modułów znajdujących się w *ibex_soc*

nazwa modułu/interfejsu	nazwa instancji	przeznaczenie
clkgen	clkgen	buforowanie sygnału zegarowego oraz jego skalowanie
ibex_wb	ibex_wishbone	wraper rdzenia Ibex przystosowany do interfejsu *Wishbone*
wishbone_sharedbus	wb_share_bus	komunikacja masterów ze slave'ami
wb_1p_ram_instr	ram_instr	jednoportowa pamięć RAM przeznaczona dla instrukcji
wb_1p_ram_data	ram_data	jednoportowa pamięć RAM przeznaczona dla danych
wb_2p_ram_instr	ram_instr	dwuportowa pamięć RAM przeznaczona dla instrukcji
wb_2p_ram_data	ram_data	dwuportowa pamięć RAM przeznaczona dla danych
wb_gpio	wb_gpio	wraper GPIO przystosowany do interfejsu *Wishbone*
wb_uart	wb_uart	wraper UART przystosowany do interfejsu *Wishbone*
wb_i2c	wb_i2c	wraper I2C przystosowany do interfejsu *Wishbone*
wb_spi_master	wb_spi_master	wraper SPI master przystosowany do interfejsu *Wishbone*
wb_spi_slave	wb_spi_slave	wraper SPI slave przystosowany do interfejsu *Wishbone*
wb_timer	wb_timer	wraper timera przystosowany do interfejsu *Wishbone*
wishbone_if	wb_master	tablica interfejsów przeznaczona dla rdzenia
wishbone_if	wb_slave	tablica interfejsów przeznaczona dla peryferii

Szczegółowy opis powyższych modułów znajduje się w kolejnych podrozdziałach. Moduł ten importuje również paczkę z mapą pamięci. Są w niej zdefiniowane parametry opisujące adres bazowy jak i rozmiar. Listing 2 przedstawia te parametry.

Listing 2: Mapa pamięci

```
package addr_map_pkg;
        parameter NUM_MASTER = 2;
        parameter NUM_SLAVE = 7;
        parameter RAM_INSTR_BASE_ADDR   = 'h00000000;
        parameter RAM_INSTR_SIZE        = 'h10000;
        parameter RAM_DATA_BASE_ADDR    = 'h00100000;
        parameter RAM_DATA_SIZE         = 'h10000;
        parameter LED_BASE_ADDR         = 'h10000000;
        parameter LED_SIZE              = 'h0fff;
        parameter UART_BASE_ADDR        = 'h10001000;
        parameter UART_SIZE             = 'h0fff;
        parameter I2C_BASE_ADDR         = 'h10002000;
        parameter I2C_SIZE              = 'h0fff;
        parameter SPI_BASE_ADDR         = 'h10003000;
        parameter SPI_SIZE              = 'h0fff;
        parameter TIMER_BASE_ADDR       = 'h10004000;
        parameter TIMER_SIZE            = 'h0fff;
endpackage
```

Parametr *NUM_MASTER* definiuje ilość *masterów* w projekcie. Są dwa, pierwszy przeznaczony dla linii danych rdzenia, drugi przeznaczony dla linii instrukcji rdzenia. Parametr *NUM_SLAVE* definiuje ilość użytych peryferii. Parametry te służą również do określenia wielkości tablic instancji interfejsu *wishbone_if*.

3.2 Rdzeń Ibex

3.2.1 *Ibex wishbone*

Głównym modułem rdzenia jest *ibex_core*. By poprawnie działał z magistralą *Wishbone*, należy opisać *wrapper* w odpowiedni sposób. W tym celu powstał moduł *ibex_wishbone*, jego zadaniem jest poprawne przeniesienie sygnałów do interfejsów magistrali *Wishbone*. Zostały w nim zainicjowane następujące moduły/interfejsy:

- *data_core* - interfejs *ibex_if* z sygnałami danych,
- *instr_core* - interfejs *ibex_if* z sygnałami instrukcji,
- *u_core* - instancja modułu *ibex_core*,
- *data_core2wb* - instancja modułu *ibex_to_wb*,
- *instr_core2wb* - instancja modułu *ibex_to_wb*.

Wykorzystuje on przypisanie ciągłe by w instancji *instr_core* wymusić stan niski na sygnałach: *we*, *be* i *wdata* w celu zabezpieczenia przypadkowego zapisu w pamięci instrukcji.

3.2.2 *Data core* i *Instr core*

Data_core i *instr_core* są to instancje interfejsu *ibex_if*. Są w nich zdefiniowane sygnały pochodzące z linii instrukcji i linii danych. Interfejs ten zawiera w sobie dwa modporty: *master* i *slave*. W zależności od potrzeby możemy odczytywać wartości sygnałów używając modportu *slave*, modport *master* daje możliwość zapisywania wartości sygnałów. Tabela 9 przedstawia listę sygnałów wraz z ich kierunkami w zależności od używanego modportu.

Tabela 9: Porty i parametry modułu *ibex_soc*

Kierunek wyprowadzenia		Nazwa wyprowadzenia	Przeznaczenie
Modport master	Modport slave		
input	input	clk_i	sygnał zegarowy
input	input	rst_ni	sygnał resetu
output	input	reg	żądanie zapytania
input	output	gnt	sygnał akceptacji zapytania
input	output	rvalid	sygnał prawidłowego odczytu danych
output	input	we	zezwolenie zapisu
output	input	be	sygnał bajtu
output	input	addr	sygnał adresowy
output	input	wdata	dane przeznaczone do zapisu
input	output	rdata	odczytane dane
input	output	err	sygnał błędu

3.2.3 *Ibex core*

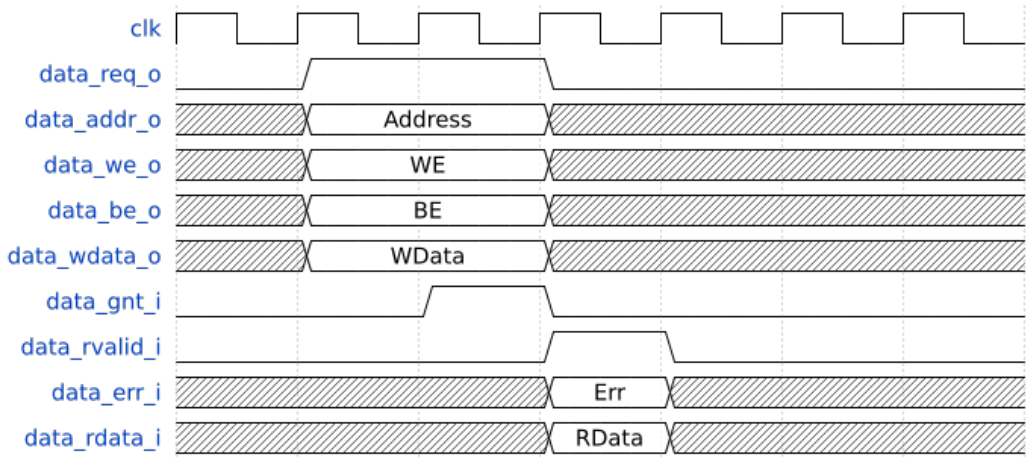
Moduł *ibex_core* został zainjonowany jako *u_core*. Zawiera on w sobie wszystkie submoduły rdzenia,a są to:

- *Clock gating* - moduł zawierający bufor sygnału zegarowego,
- *Instruction Fetch* - odpowiedzialny za pobieranie instrukcji. W jednym cyklu dostarcza instrukcję do *ibex_id_stage* o ile pamięć jest zdolna do wysłania jednej instrukcji na cykl. Instrukcje są przechwytywane do *ibex_prefetch_buffer* w celu optymalizacji wydajności. Rozkazy są zapisywane wraz licznikiem rozkazów i pochodzą z *ibex_fetch_fifo*. Gdy *FIFO* jest puste, instrukcja natychmiast zostaje przekazana na jego wyjście,
- *Instruction Decode* - odpowiedzialny za dekodowanie instrukcji. Zawiera on w sobie multipleksery, kontrolujące przepływ danych do ALU,
- *Instruction Execute* - odpowiedzialny za wykonanie instrukcji, zawiera on w sobie ALU i moduły odpowiedzialne za mnożenie i dzielnie,
  - *Arithmetic Logic Unit* - jednostka arytmetyczno-logiczna, blok kombinacyjny wykonujący obliczenia liczb całkowitych oraz operacje porównawcze. Dodatkowo jest wykorzystywany:
    - w wykonywaniu dodawania w ramach algorytmów mnożenia i dzielenia,
    - w obliczaniu $PC+Imm$,
    - w obliczaniu adresu pamięci $Reg+Imm$,
  - *Multiplier/Divider Block* - blok wykorzystywany do mnożenia i dzielenia. Są dostępne dwa tryby: szybki i wolny. Oba wykorzystują algorytm długiego podziału oraz jednostkę arytmetyczno-logiczną.
- *Register File* - zawiera trzydzieści dwa lub szesnaście 32-bit rejestrów. Liczba ich jest zależna od rozszerzenia *RV32E*. Rejestr *x0* jest zawsze zerem. Moduł ten posiada dwa porty przeznaczone dla odczytu i jeden dla zapisu. Gdy dany rejestr jest równocześnie zapisywany i odczytywany, zwróci on wartość aktualną a nie zapisywaną,
- *Control and Status Registers* - zawiera rejestry kontrolne i statusu,
- *Load-Store Unit* - odpowiedzialny za dostęp do pamięci danych. Pozwala działać na słowach (32-bit) pół-słowach(16-bit) i bajtach(8-bit). Każda operacja zapisania lub odczytu danych powoduje zatrzymacie bloków *ID/EX* na przynajmniej

jeden cykl w celu oczekiwania na odpowiedź. Potrafi obsłużyć źle ustawiony dostęp do pamięci, czyli dostęp, który nie jest w domyślnych granicach słowa. Potrzeba na to co najmniej dwóch cykli ponieważ są robione dwa osobne zapisania. Komunikacja z pamięcią odbywa się w następujący sposób:

1. Jednsotka LSU wysyła adres poprzez *data_addr_o*, konfiguruje wyjścia *data_be_o*, *data_wdata_o*, ustawia stan wysoki sygnału *data_req_o* i *data_we_o*. Gdy pamięć będzie gotowa do obsługi żądania, odpowiada stanem wysokim sygnału *data_gnt_i*.
2. Po otrzymaniu potwierdzenia gotowości, *LSU* może zmienić wartość sygnału *data_addr_o*.
3. Pamięć wysyła wysoki stan sygnału *data_rvalid_i* wraz z informacją o wystąpieniu błędów, jeśli takowe się pojawią zostanie to zasygnalizowane stanem wysokim sygnału *data_err_i*. Odczytane dane są przekazywane dostępne na linii *data_rdata_i*.
4. W przypadku wielu żądań, są one obsługiwane w kolejności ich nadania.

Rysunek 9 przedstawia przykład komunikacji między modułem *LSU* a pamięcią.



Rysunek 9: Komunikacja *LSU* z pamięcią[11]

3.2.4 Komunikacja rdzenia z magistralą *Wishbone*

Instancje *data_core2wb* i *instr_core2wb* modułu *ibex_to_wb* są odpowiedzialne za komunikację rdzenia z magistralą. Moduł ten posiada dwa porty:

1. *core* - modport *slave* pochodzący z interfejsu *ibex_if*. Dla instancji *data_core2wb* została przypisana instancja *data_core*, dla instancja *instr_core2wb* została przypisana instancja *insftr_core*.

2. *wb* - modport *master* pochodzący z interfejsu *wishbone_if*. Dla instancji *data_core2wb* została przypisana instancja *data_wb*. Dla instancji *instr_core2wb* została przypisana instancja *instr_wb*.

W module tym zostało wykorzystane przypisanie ciągłe w celu przekazania wartości sygnałów. Listing 3 przedstawia te przypisania.

Listing 3: Przypisanie ciągłe modułu *ibex_to_wb*

```
assign core.gnt    = core.req & ~wb.stall;
assign core.rvalid = wb.ack;
assign core.err    = wb.err;
assign core.rdata  = wb.data_s;
assign wb.stb      = core.req;
assign wb.addr     = core.addr;
assign wb.data_m   = core.wdata;
assign wb.we       = core.we;
assign wb.sel      = core.we ? core.be : '1;

always_ff @(posedge wb.clk_i or posedge wb.rst_ni)
        if (!wb.rst_ni)
                cyc <= 1'b0;
        else
        if (core.req)
                cyc <= 1'b1;
        else if (wb.ack || wb.err)
                cyc <= 1'b0;

assign wb.cyc = core.req | cyc;}
```

Dzięki temu zabiegowi, każda zmiana sygnału zostanie przeniesiona na magistralę *Wishbone*.

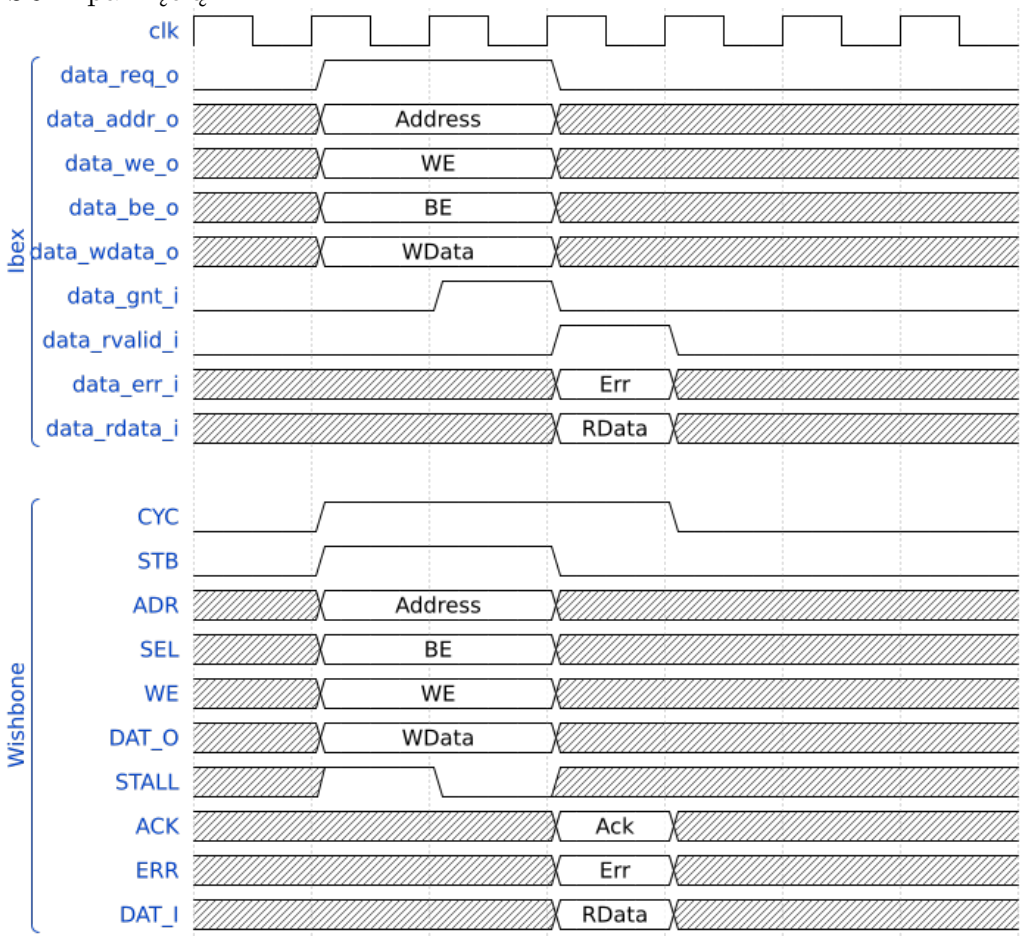
3.3 *Wishbone*

Magistrala *Wishbone* składa się z następujących komponentów:

- Instancje interfejsu *wishbone_if*: *wb_master* i *wb_slave*,
- instancji modułu *wishbone_sharedbus*: *wb_share_bus*,
- modułów służących do podłączenia komponentów systemu na chipie do magistrali:
  - *wb_gpio* - moduł łączący GPIO z magistralą,
  - *wb_uart* - moduł łączącu UART z magistralą,
  - *wb_i2c* - moduł łączący I2C z magistralą,
  - *wb_spi_master* - moduł łączący SPI master z magistralą,
  - *wb_spi_slave* - moduł łączący SPI slave z magistralą,
  - *wb_timer* - moduł łączący timer z magistralą,

– *wb_ram* - moduł łączący pamięć RAM z magistralą.

Rysunek 10 przedstawia porównanie komunikacji magistrali *Wishbone* z komunikacją *LSU* z pamięcią.



Rysunek 10: Porównanie komunikacji Ibex z *Wishbone*

3.3.1 Interfejs magistrali *Wishbone*

Interfejs magistrali posiada dwie instancje:

- *wb_master* - tablica interfejsu, wielkość tej tablicy definiowana jest przez parametr *NUM_MASTER*. Jest ona przeznaczona dla urządzeń typu *master*,
- *wb_slave* - tablica interfejsu, wielkość tej tablicy definiowana jest przez parametr *NUM_SLAVE*. Jest ona przeznaczona dla urządzeń typu *slave*.

W celu zarządzania kierunkami sygnałów, zostały utworzone dwa modporty: *master* i *slave*. Wyprowadzenia oraz ich przeznaczenie zostały opisane w tabeli 10.

Tabela 10: Sygnały interfejsu *wishbone_if*

Kierunek sygnału		Nazwa sygnału	Przeznaczenie
Modport master	Modport slave		
input	input	clk_i	sygnał zegarowy
input	input	rst_ni	sygnał resetu
output	input	addr	sygnał adresu
output	input	data_m	sygnał danych mastera
input	output	data_s	sygnał danych slave'a
output	input	we	zezwolenie zapisu
output	input	sel	selekcja bajtu
output	input	stb	potwierdzenie nadania danych
input	output	ack	potwierdzenie przyjęcia danych
output	input	cyc	cykl magistrali
input	output	err	sygnał błędu
input	output	stall	sygnał zajętości

3.3.2 Połączenia magistrali *Wishbone*

W celu komunikacji urządzeń typu *slave* z urządzeniami typu *master* należało przygotować odpowiedni moduł, kontrolujący tę komunikację. Jest on parametryzowany:

- *num_master = -1* - określa liczbę urządzeń typu *master*,
- *num_slave = -1* - określa liczbę urządzeń typu *slave*,
- *bit [31:0] base_addr[num_slave] = '{-1}* - tablica adresów początkowych urządzeń typu *slave*. Jej szerokość definiowana jest poprzez parametr *num_slave*, każde jej pole to liczba całkowita 32-bitowa.
- *bit [31:0] size[num_slave] = '{-1}* - tablica szerokości adresu pod jakim znajduje się urządzenie typu *slave*. Jej szerokość definiowana jest poprzez parametr *num_slave*, każde jej pole to liczba całkowita 32-bitowa.

Wartość domyślna parametrów to -1, ma to uchronić przed złym przypisaniem wartości podczas inicjalizacji tego modułu. Lista portów tego modułu składa się z dwóch modportów:

- *wishbone_if.slave wb_master[num_master]* - port przeznaczony dla odczytu informacji z urządzeń typu *master*. Został użyty modport *slave* w celu zabezpieczenia przed przypadkowym nadpisaniem sygnałów,
- *wishbone_if.master wb_slave[num_slave]* - port przeznaczony do odczytu informacji z urządzeń typu *slave*. Został użyty modport *master* w celu zabezpieczenia przed przypadkowym nadpisaniem sygnałów.

Przykładowa inicjalizacja modułu została pokazana na Listingu 4. Kolejność parametrów podanych do przypisania tablicy *base_addr* i *size* musi się zgadzać z indeksem przypisanym dla poszczególnego komponentu.

Listing 4: Przykładowa inicjalizacja modułu *wishbone_sharedbus*

```
wishbone_sharedbus
#(      .num_master       (NUM_MASTER),
        .num_slave        (NUM_SLAVE),
        .base_addr ('{RAM_INSTR_BASE_ADDR,RAM_DATA_BASE_ADDR, GPIO_BASE_ADDR,
         UART_BASE_ADDR, I2C_BASE_ADDR, SPI_BASE_ADDR, TIMER_BASE_ADDR}),
        .size       ('{RAM_INSTR_SIZE, RAM_DATA_SIZE, GPIO_SIZE, UART_SIZE,
         I2C_SIZE, SPI_SIZE, TIMER_SIZE}))
wb_share_bus(
        .wb_master(wb_master),
        .wb_slave(wb_slave));
```

Dla sygnałów pochodzących z urządzeń zostały utworzone pomocnicze tablice zmiennych tymczasowych. Szerokość tych tablic definiują parametry *num_master* i *num_slave*. Zdefiniowane zostały również sygnały wspólne, mające na celu przekazywanie wartości między komponentami.
W pierwszym kroku należy odczytać/przypisać wartości dla danych modportów. Listing 5 przedstawia tą operację.

Listing 5: Przykładowa inicjalizacja modułu *wishbone_sharedbus*

```
for (genvar i = 0; i < num_master; i++)
    begin
        assign wb_master_cyc[i] = wb_master[i].cyc;
        .
        .
        .
        assign wb_master[i].ack = wb_master_ack[i];
     end

    for (genvar i = 0; i < num_slave; i++)
    begin
        assign wb_slave[i].cyc = wb_slave_cyc[i];
        .
        .
        .
        assign wb_slave_ack[i] = wb_slave[i].ack;
     end
```

Pętla *for* z użyciem zmiennej typu *genvar* pozwala tworzyć bloki generyczne. Dzięki nim i przypisaniu ciągłemu wartości poszczególnych sygnałów zawsze zostaną przypisane gdy nastąpi ich zmiana.
Wybór aktywnego urządzenia *slave* jest dokonywany poprzez iterację po tablicy adresów i sprawdzenie poprzez operator *inside* czy adres podany przez *mastera* znajduje się w przestrzeni adresowej urządzenia *slave*. Gdy jest to prawdą, operator zwróci jedynkę logiczną, która jest przypisywana do tablicy *slave_select* w komórkę odpowiadającej

danemu urządzeniu *slave*. Operacja ta została umieszczona w bloku proceduralnym *always_comb* więc przy każdej zmianie adresu operacja ta jest ponawiana. Listing 6 przedstawia ten proces.

Listing 6: Wybór urządzenia *slave*

```
always_comb
        for (int i = 0; i < num_slave; i++)
                ss[i] = addr inside {[base_addr[i]:(base_addr[i]+size[i])]};

always_ff @(posedge wb_slave[0].clk_i or posedge wb_slave[0].rst_ni)
        if (!wb_slave[0].rst_ni)
                ss1 <= '0;
        else
        if (cyc && stb)
                ss1 <= ss;
```

Informacja o wyborze danego urządzenia jest przekazywana za pomocą nie blokującego przypisania do tablicy *slave_select_1* w celu zapewnienia potokowości. Blok proceduralny *always_comb* zapewnia komunikacje między *masterem* a urządzeniem *slave*. Listing 7 przedstawia ten zabieg.

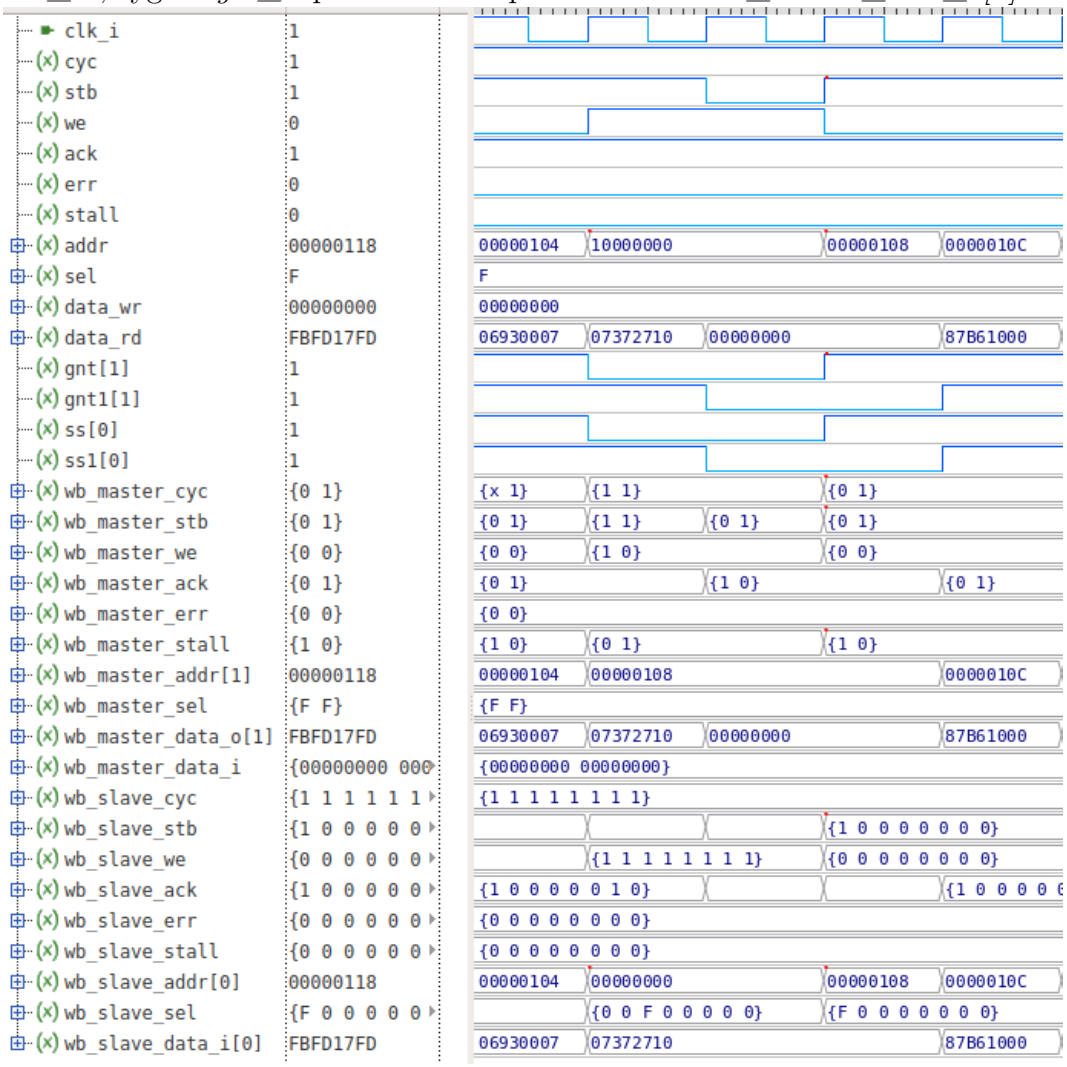
Listing 7: Komunikacja urządzenia *master* z urządzeniem *slave*

```
always_comb begin
        ack    = 1'b0;
        err    = 1'b0;
        stall  = 1'b0;
        data_rd = '0;
        for (int i = 0; i < num_slave; i++) begin
                ack   |= wb_slave_ack[i];
                err   |= wb_slave_err[i];
                stall |= wb_slave_stall[i];
                wb_slave_cyc[i]   = cyc;
                wb_slave_addr[i]  = '0;
                wb_slave_stb[i]   = 1'b0;
                wb_slave_we[i]    = we;
                wb_slave_sel[i]   = '0;
                wb_slave_data_o[i] = '0;
                if (ss[i]) begin
                        wb_slave_addr[i]   = addr;
                        wb_slave_stb[i]    = cyc & stb;
                        wb_slave_sel[i]    = sel;
                        wb_slave_data_o[i] = data_wr;
                end
                if (ss1[i])
                        data_rd = wb_slave_data_i[i];
        end
end
```

Jeśli urządzenie *slave* zostało wybrane, przekazywane są do niego sygnały z urządzenia *master* w kolejnym cyklu zegarowym dane są przypisywane do urządzenia *master*.
Obsługa urządzeń *master* jest analogiczna. Gdy pojawi się sygnał *cyc* informujący o żądaniu mastera następuje zapisanie stanu wysokiego dla sygnału *gnt*, w kolejnym

cyklu zegarowym wartość ta jest przekazywana do *gnt_1* w celu zachowania potokowości. Gdy urządzenie *master* jest gotowe do działania, zapisuje dane do sygnałów wspólnych, te przekazują je urządzeniu *slave*. Potwierdzeniem udanej transmisji jest przekazanie sygnału *ack* potwierdzającego odczyt danych przez peryferia i sygnału *err*, który komunikuje o problemach.

Rysunek 11 przedstawia przebiegi sygnałów tego modułu uzyskanych dzięki symulacji. Po otrzymaniu prawidłowego adresu sygnał *ss* zmienił swój stan na wysoki, w kolejnym cylku zegarowym stan wysoki przyjmuje sygnał *ss1* - co odpowiada opisowi. Pojawienie się jedynki logicznej w sygnale *stb* spowodowało aktywację sygnału *gnt* a z kolejnym cyklem zegarowym wartość ta zostaje przepisana na sygnał *gnt1*. Wysoki stan *ss* spowodował aktywację urządzenie *slave*, można to zauważyć poprzez pojawienie się adresu na linii *wb_slave_addr*, dzięki *ss1* dane z linii *wb_slave_data_i[0]* zostały przekazane do *data_rd*, sygnał *gnt_1* pozwolił na zapis ich na linii *wb_master_data_o[1]*



Rysunek 11: Przebiegi sygnałów podczas symulacji

3.4 Pamięć *RAM*

3.4.1 Komunikacja pamięci z magistralą *Wishbone*

W celu poprawnej komunikacji z magistralą należało opisać moduł, którego zadaniem jest poprawna konwersja i przekazywanie sygnałów między magistralą a pamięcią *RAM*. Moduł posiada parametr: *SIZE*, informujący o pojemności tejże pamięci, oraz lokalny parametr:*ADDR_WIDTH* informujący o szerokości pola adresowego, powstaje on dzięki obliczeniu logarytmu o podstawie dwa z parametru *SIZE*. Parametry te są dalej przekazywane dla modułów opisujących pamięć *RAM*. Listing 8 przedstawia komunikacje między pamięcią a magistralą. Adres zostaje wybrany poprzez wybranie odpowiedniej części wektora *addr*. Sygnał *valid* umożliwia zapis/odczyt z pamięci. Jest on aktywny gdy nadejdzie potwierdzenie nadania danych wraz z wysokim stanem sygnału cyklu magistrali. Pozwolenie na zapis danych jest równe koniunkcji sygnałów selekcji oraz powielonemu cztery razy pozwoleniu na zapis pochodzącego od rdzenia. Sygnały zajętości pamięci i błędu zostały podpięte do stanu niskiego ponieważ w modelu pamięci nie istnieje możliwość ich wystąpienia.

Listing 8: Komunikacja pamięci z magistralą

```
assign ram_addr = wb.addr[ADDR_WIDTH-1:2];
assign ram_valid = valid;
assign ram_we = {4{wb.we}} & wb.sel;
assign ram_data_i = wb.data_m;
assign wb.data_s = ram_data_o;
assign valid = wb.cyc & wb.stb;
assign wb.stall = 1'b0;
assign wb.err = 1'b0;

always_ff @(posedge wb.clk_i or posedge wb.rst_ni)
    if (!wb.rst_ni)
        wb.ack <= 1'b0;
    else
        wb.ack <= valid & ~wb.stall;
```

3.4.2 Pamięć jednoportowa

Listing 9 przedstawia opis modelu pamięci RAM.

Listing 9: Model pamięci RAM

```
 logic /*sparse*/ [31:0] mem [SIZE];

always @(posedge clk_i)
  if (valid_i)
    begin
       if (we_i[0]) mem[addr_i][7:0]   <= data_i[7:0];
       if (we_i[1]) mem[addr_i][15:8]  <= data_i[15:8];
       if (we_i[2]) mem[addr_i][23:16] <= data_i[23:16];
       if (we_i[3]) mem[addr_i][31:24] <= data_i[31:24];
    end

always_ff @(posedge clk_i)
  if (valid_i)
    data_o <= mem[addr_i];

 parameter MEM_FILE = "blink_slow.mem";
 initial begin
   $display("Initializing %s", MEM_FILE);
   $readmemh(MEM_FILE, mem);
 end
```

Komórka pamięci składa się z 32-bitów, ilość komórek jest definiowana przez parametr *SIZE*. Argument */*sparse*/* został użyty w celu optymalizacji symulacji. Parametr *MEM_FILE* określa ścieżkę do pliku, który zostanie załadowany do pamięci. Podczas wysokiego stanu sygnału *valid_i* zostaje odczytana komórka pamięci ze wskazanego adresu. Zezwala on również na zapis do komórki pamięci, gdy odpowiedni bit sygnału *we_i* przejdzie w stan wysoki. Sygnał ten jest 4-bitowy, każdy bit odpowiada jednemu bajtu w komórce pamięci. Pamięć jest jednoportowa więc w danej chwili czasu dozwolona jest operacja zapisu lub odczytu danych. By zapobiec kolizji, zaimplementowano dwie pamięci *RAM*, pierwsza odpowiedzialna za przechowywanie instrukcji, druga odpowiedzialna za przechowywanie danych. Pamięć instrukcji została przypisana do zerowej komórki tablicy instancji *wb_slave* interfejsu *wishbone_if*, natomiast pamięć danych do pierwszej komórki. Rysunek 12 przedstawia przebiegi sygnałów oraz fragment pamięci RAM. Podczas wysokiego stanu sygnału *valid*, pojawiła się informacja o chęci odczytu z danych z komórki o adresie *0020*. Pod tym adresem zapisana jest wartość *0100006F* która w następnym cyklu zegarowym trafia na wyprowadzenie *data_o*, sytuacja ta powtarza się do momentu pojawienia się stanu niskiego sygnału *valid*. Rysunek 13 przedstawia sytuacje zapisu do pamięci. Na linii adresowej pojawia się *00*, sygnał *valid_i* jest w stanie wysokim. Oznacza to, że w następnym cyklu zegarowym do komórki pamięci o adresie *0* zostanie przypisana wartość znajdująca się w *data_i*. Jak widać na przebiegu sygnałów wartość ta została

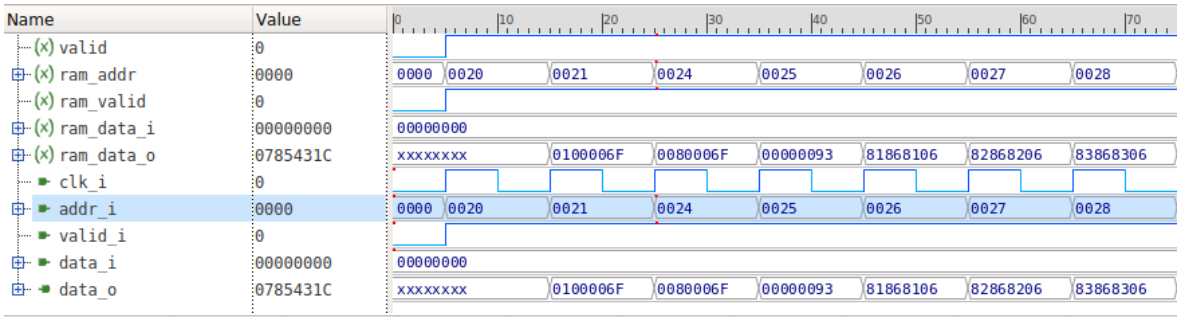
poprawnie zapisana. Gdy sygnał *valid_i* jest w stanie niskim, wartość z linii *data_o* nie powinna zostać przekazana do pamięci. Na przebiegach również została ukazana taka sytuacja, linia adresowa przyjmuje wartość *1D* lecz komórka pamięci o tym adresie nie zostaje zapisana. Listing 10 i 11 przedstawiają instancje modułów pamięci.

Listing 10: Instancja pamięci instrukcji

```
p1_ram_instr#(
      .SIZE(SIZE),
      .AW(ADDR_WIDTH))
ram(
      .clk_i(wb.clk_i),
      .addr_i(ram_addr),
      .valid_i(ram_valid),
      .data_i(ram_data_i),
      .data_o(ram_data_o)
);
```

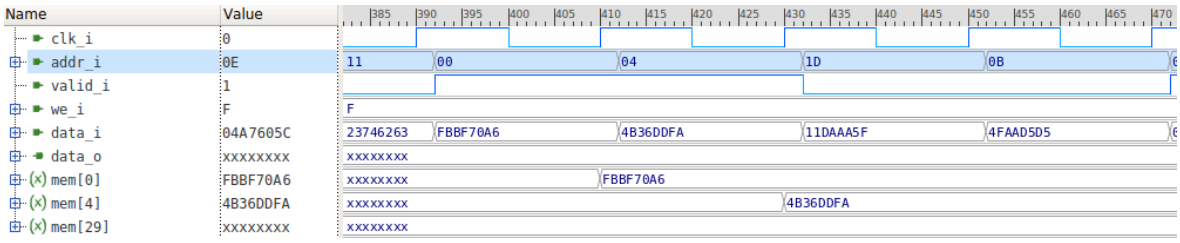
Listing 11: Instancja pamięci danych

```
p1_ram_data#(
      .SIZE(SIZE),
      .AW(ADDR_WIDTH))
ram(
      .clk_i(wb.clk_i),
      .addr_i(ram_addr),
      .valid_i(ram_valid),
      .we_i(ram_we),
      .data_i(ram_data_i),
      .data_o(ram_data_o));
```



	0000	0001	0002	0003	0004	0005	0006	0007	0008	0009
0020	0100006F	0080006F	0040006F	0000006F	00000093	81868106	82868206	83868306	84868406	85868506
0030	8C868C06	8D868D06	8E868E06	8F868F06	00010117	F3010113	13000D13	13000D93	01BD5763	000D2023

Rysunek 12: Przebiegi sygnałów pamięci *RAM* oraz jej dane podczas odczytu.



Rysunek 13: Przebiegi sygnałów pamięci *RAM* podczas zapisu.

3.4.3 Pamięć dwuportowa

Pamięć dwu portowa składa się z dwóch zestawów linii adresowych, danych i sterujących. Pozwala to na jednoczesny dostęp do pamięci dwóm niezależnym procesom do wspólnych danych. Komórka pamięci składa się z 32-bitów a ilość komórek jest definiowana przez parametr *SIZE*. Inicjalizacja pamięci odbywa się poprzez podanie ścieżki do pliku w parametrze *MEM_FILE*. Zapis i odczyt działa w sposób analogiczny jak w przypadku pamięci jednoportowej. Gdy obie linie adresowe wskazują tą samą komórkę,

pierwszeństwo ma linia z instrukcji oznaczona literą *b*. Listing 12 przedstawia inicjalizację tego modułu. Sygnały *b_we_i* i *b_data_i* zostały przypisane do zera ponieważ w aktualnej wersji przewiduje jedynie wgrywanie kodu maszynowego poprzez podanie odpowiedniej ścieżki za pomocą parametru *MEM_FILE*.

Listing 12: Inicjalizacja dwuportowej pamięci RAM

```
p2_ram#(
      .SIZE(SIZE),
      .AW(ADDR_WIDTH))
ram(
      .clk_i(wb_data.clk_i),
      .a_addr_i(a_ram_addr),
      .a_valid_i(a_ram_valid),
      .a_we_i(a_ram_we),
      .a_data_i(a_ram_data_i),
      .a_data_o(a_ram_data_o),

      .b_addr_i(b_ram_addr),
      .b_valid_i(b_ram_valid),
      .b_we_i('0),
      .b_data_i('0),
      .b_data_o(b_ram_data_o)      );
```

3.5 *GPIO*

3.5.1 Komunikacja z magistralą *Wishbone*

W celu poprawnej komunikacji należało przygotować moduł odpowiedzialny za konwersję i przekazywanie danych między *GPIO* a magistralą. Rolę tę pełni *wb_gpio*. Moduł ten zawiera instancję *GPIO* oraz sygnały pomocnicze do komunikacji. Listing 13 przedstawia fragment tego modułu.

Listing 13: Komunikacja *GPIO* z magistralą

```
assign valid     = wb.cyc & wb.stb;
assign select_output    = wb.addr[11:2] == 0;
assign select_input    = wb.addr[11:2] == 1;
assign wb.stall = 1'b0;
assign wb.err    = 1'b0;

always_ff @(posedge wb.clk_i or posedge wb.rst_ni)
  if (!wb.rst_ni)
    wb.ack <= 1'b0;
  else
    wb.ack <= valid & ~wb.stall;

assign wb.data_s = {28'h0000000, data_s};
```

Do linii *data_s* został przypisany sygnał pochodzący z instancji *GPIO data_s*, jest on 4-bitowy więc by w pełni zapełnić przestrzeń zastosowano konkatenację. Sygnały *wb.err* i *wb.stall* zostały przypisane do zera. Projekt nie przewiduje sytuacji by te

sygnały mogą się pojawić. Kierunek transmisji jest wybierany poprzez ustawienie odpowiedniego bitu w sygnale *wb.addr*. Jeśli wartość tego wektora będzie równa *0*, dane zostaną przekazane do wyjścia, jeśli wartość wektora będzie równa *1*, dane zostaną odczytane z wejść. Sygnał *valid* jest równy koniunkcji sygnałów *wb.cyc* i *wb.stb*. Sygnał *wb.ack* przyjmuje stan wysoki jeden cykl zegarowy po pojawieniu się sygnału *valid*.

3.5.2 Moduł *GPIO*

Opis modułu *GPIO* został przedstawiony na listingu 14. Podczas resetu wszystkie sygnały są zerowane. Następnie w zależności od wybranego trybu, dane są przekazywane na diody LED lub odczytywane z przełączników znajdujących się na płytce FPGA. Następnie infomracje są przekazywane do sygnału *data_s*

Listing 14: Model *GPIO*

```
always @(posedge clk_i or posedge rst_ni)
        if (!rst_ni)
                led <= '0;
        else
        if (valid && we && sel_led) begin
                led <= data_m[3:0];
                data_s <= led;
        end

always @(posedge clk_i or posedge rst_ni)
        if (!rst_ni)
                data_output <= '0;
        else
        if (valid && we && sel_but) begin
                data_input <= button;
                data_s <= data_input;
        end
```

3.6 *UART*

3.6.1 Komunikacja z magistralą *Wishbone*

W celu poprawnej komunikacji z magistralą *Wishbone* został stworzony moduł *wb_uart*. Znajduje się w nim instancja modułu głównego *UART*, oraz sygnały potrzebne do poprawnego połączenia z magistralą. Listing 15 przedstawia inicjalizację modułu głównego *UART* i sygnały pomocnicze.

Listing 15: Model *GPIO*

```
uart#(.clk_freq(50000000),
        .baud_rate(19200),
        .data_bits(8),
        .parity_type(0),
        .stop_bits(0)) uart_top (
        .rx_i(uart_rx_i),
        .tx_data_i(wb.data_m[8-1:0]),
        .tx_data_vld_i(valid_i),
        .rst_i(~wb.rst_ni),
        .clk_i(wb.clk_i),
        .we_i(wb.we),
        .rx_data_vld_o(valid_o),
        .rx_data_o(uart_data_rx),
        .rx_parity_err_o(wb.err),
        .tx_o(uart_tx_o),
        .tx_active_o(wb.stall));

assign valid_i = wb.cyc & wb.stb;
assign wb.data_s = {24'h000000, uart_data_rx};
always_ff @(posedge wb.clk_i or posedge wb.rst_ni)
        if (!wb.rst_ni)
                wb.ack <= 1'b0;
        else
                wb.ack <= valid_o & ~wb.stall;
```

W celu przypisania wartości sygnału *uart_data_rx* do wektora *wb.data_s* należy użyć konkatenacji z zerami, ponieważ sygnał ten jest 8-bitowy. Zera chronią przed zapisałem niechcianych sygnałów. Sygnał *valid_i* jest równy koniunkcji sygnałów *wb.cyc* i *wb.stb*. Potwierdzenie nadania informacji poprzez transmiter jest uzyskiwane poprzez mnożenie logiczne sygnału *valid_o* i negacją sygnału *wb.stall*.

3.6.2 Moduł główny *UART*

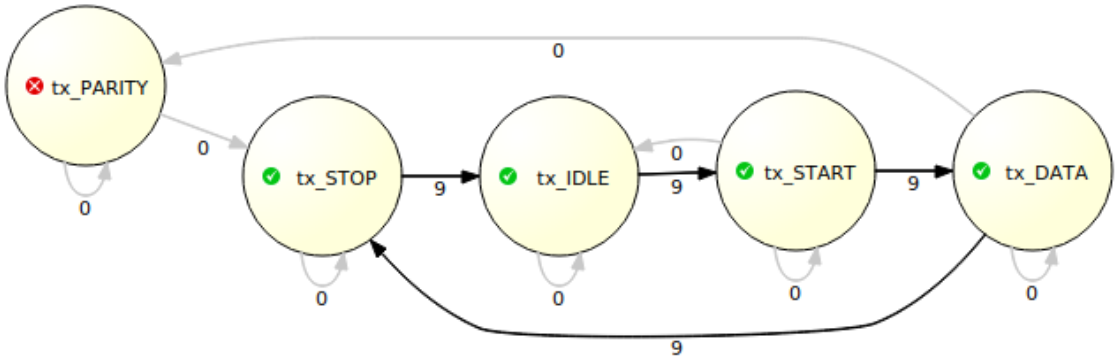
W module głównym znajdują się instancje transmitera i odbiornika *UART*. Poprzez parametryzowanie go można określić następujące cechy:

- *clk_freq* - określa częstotliwość zegara systemu na chipie,
- *baud_rate* - określa szybkość transmisji, w projekcie jego wartość jest równa 19200bps,
- *data_bits* - określa szerokość wektora danych. W projekcie użyto 8-bitowej szerokości,
- *parity_type* - określa bit parzystości, przypisanie zera wyłączy go, jedynki ustawienie go jako bit nieparzystości, dwójki ustawienie go jako bit parzysty. W projekcie bit ten jest wyłączony,

- *stop_bits* - ilość bitów stopu, dostępny jest wybór między jednym a dwoma bitami. W projekcie występuje jeden bit stopu.

3.6.3 Implementacja transmitera *UART*

Transmiter został zaimplementowany w oparciu o graf FSM przedstawiony na rysunku 14



Rysunek 14: Graf *FSM* transmitera *UART*.

Stany te zostały przepisane do lokalnych parametrów, za poruszanie się między nimi odpowiadają zmienne: *tx_STATE* i *tx_NEXT*. Podczas resetu sygnały są zerowane i zostaje ustawiony stan *tx_IDLE*. Po jego zwolnieniu wartości zostają przypisywane do poszczególnych sygnałów. Przedstawia to listing 16.

Listing 16: Transmiter *UART* po resecie

```
tx_STATE <= tx_NEXT;
clk_div_reg <= clk_div_next;
tx_out_reg <= tx_out_next;
tx_data_reg <= tx_data_next;
index_bit_reg <= index_bit_next;
stop_bits_remaining <= stop_bits_remaining_next;
```

Podczas stanu *tx_IDLE* zostają przypisane wartości domyślne, dla sygnału *tx* ustawiony jest stan wysoki. Gdy nadejdzie potwierdzenie przesłania danych przez rdzeń, stan zostaje zmieniony na *tx_START*.

Stan *tx_START* ustawia sygnał *tx* na niski, rozpoczynając w ten sposób transmisje. Następnie do zmiennej *tx_NEXT* przypisywany jest stan *tx_DATA*.

Stan *tx_DATA* został przedstawiony na listingu 17. Do sygnału *tx* jest przypisywana wartość wybranego bitu wektora *tx_data_reg*. Kolejny krok to sprawdzanie czasu bitu, jeśli licznik czasu dojdzie do samego końca, wskaźnik bitu zwiększa swoją wartość w przeciwnym razie zachodzi inkrementacja licznika czasu i zapętlenie stanu. Przepełnienie wskaźnika bitu skutkuje przejściem w kolejny stan *tx_STOP* lub *tx_PARITY* jeśli ustawiony jest parametr.

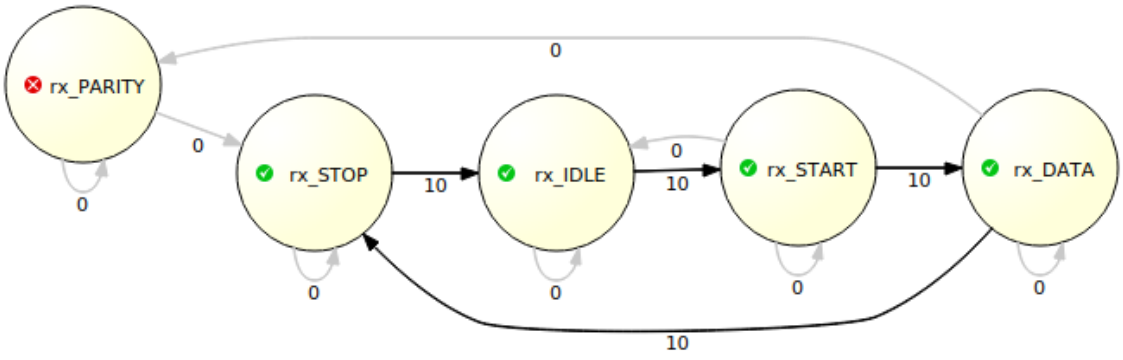
Listing 17: Stan *tx_DATA*

```
tx_DATA: begin
        tx_out_next = tx_data_reg[index_bit_reg];
        if(clk_div_reg < clock_divide[$clog2(clock_divide):0]-1'b1) begin
                clk_div_next = clk_div_reg + 1'b1;
                tx_NEXT = tx_DATA;
        end
        else begin
                clk_div_next = 0;
                if(index_bit_reg < (data_bits-1)) begin
                        index_bit_next = index_bit_reg + 1'b1;
                        tx_NEXT = tx_DATA;
                end
                else begin
                        index_bit_next = 0;
                        if(parity_type == 0) begin
                                tx_NEXT = tx_STOP;
                        end
                        else begin
                                tx_NEXT = tx_PARITY;
                        end
                end
        end
end
```

Parzystość zostaje sprawdzana przy pomocy operatorów redukcji *XOR* lub *NXOR* użytych na całym wektorze *tx_data_reg*. Po wysłaniu tej informacji, do zmiennej *tx_NEXT* zostaje przypisany stan *tx_STOP*.
W stanie *tx_STOP* zostaje wysłana odpowiednia ilość bitów stopu. Po dokonaniu tej operacji, stan wraca do *tx_IDLE*.

3.6.4 Implementacja odbiornika *UART*

Odbiornik został zaimplementowany w oparciu o graf FSM przedstawiony na rysunku 15



Rysunek 15: Graf *FSM* odbiornika *UART*.

Stany te zostały zdefiniowane jako lokalne parametry, za poruszanie się między nimi odpowiedzialne są dwie zmienne: *rx_STATE* oraz *rx_NEXT*. Podczas resetu sygnały

są zerowane, do zmiennej *rx_STATE* zostaje przypisany stan *rx_IDLE*. Po jego zwolnieniu następuje przypisanie nie blokujące, które ma na celu przypisania wartości w następnym cyklu zegarowym. W ten sposób aktualizacja stanu nastąpi zawsze na początku cyklu. Listing 18 pokazuje wszystkie przypisania. Całość mieści się w bloku proceduralnym *always_ff*.

Listing 18: Odbiornik *UART* po resecie

```
rx_STATE <= rx_NEXT;
clk_div_reg <= clk_div_next;
rx_data_reg <= rx_data_next;
index_bit_reg <= index_bit_next;
rx_data_vld <= rx_data_vld_next;
rx_parity_err <= parity_err_next;
stop_bits_remaining <= stop_bits_remaining_next;
```

Pierwszą fazą która występuje po resecie jest: *rx_IDLE*. Podczas niej blok czeka, aż na linii *rx* pojawi się zero logiczne, oznaczające początek transmisji. Jeśli ten warunek zostanie spełniony do zmiennej *rx_NEXT* zostaje przypisana faza *rx_START*. Jeśli na linii *rx* wciąż pozostaje stan wysoki do *rx_NEXT* przypisany zostanie *rx_IDLE*. Podczas tej fazy zostają ustawione wartości początkowe dla sygnałów.
W fazie *rx_START* następuje ponowne sprawdzenie stanu sygnału *rx*, sprawdzenie te następuje w połowie trwania bitu. Jeśli pozostał w stanie niskim nastąpi przypisanie do *rx_NEXT* kolejnego stanu, którym jest *rx_DATA*. Jeśli sygnał powrócił do stanu wysokiego, stan wraca do *rx_IDLE*.
Faza *rx_DATA* została przedstawiona na listingu 19. Pierwszym krokiem w tym stanie, jest sprawdzanie w jakim czasie trwania bitu znajduje się sygnał. W warunku sprawdzającym ograniczono wielkość wektora *clock_divide* na niezbędnej ilości bitów w celu optymalizacji projektu. Gdy licznik przyjmie wartość równą końcu czasu bitu, następuje jego wyzerowanie i przypisanie wartości sygnały *rx* do wektora *rx_data_next*. Następnie jest sprawdzana ilość odebranych bitów, jeśli zostanie ona przekroczona, następuje kolejny stan *rx_STOP* lub *rx_PARITY* w zależności od ustawień parametru odpowiedzialnego za parzystość bitu. W przeciwnym razie, wartość indeksu wektora zostaje zwiększona i stan się zapętla.

Listing 19: Stan *rx_DATA*

```
rx_DATA: begin
        if(clk_div_reg < clock_divide[$clog2(clock_divide):0]-1'b1) begin
                clk_div_next = clk_div_reg + 1'b1;
                rx_NEXT = rx_DATA;
        end
        else begin
                clk_div_next = 0;
                rx_data_next[index_bit_reg] = rx;
                if(index_bit_reg < (data_bits-1)) begin
                        index_bit_next = index_bit_reg + 1'b1;
                        rx_NEXT = rx_DATA;
                end
                else begin
                        index_bit_next = 0;
                        if(parity_type == 0) begin
                                rx_NEXT = rx_STOP;
                        end
                        else begin
                                rx_NEXT = rx_PARITY;
                        end
                end
        end
end
```

Parzystość zostaje sprawdzana za pomocą operatorów redukcji *XOR* i *NXOR* zastosowanych na całym wektorze *rx_data_reg*.
W fazie *rx_STOP* blok czeka na pojawienie się określonej ilości bitów stopu. Gdy warunek ten zostanie spełniony stan wraca do *rx_IDLE* oraz ustawia logiczną jedynkę dla sygnału potwierdzającego odbiór transmisji.

3.7 Interfejs *SPI*

3.7.1 Komunikacja z magistralą *Wishbone*

W celu poprawnej komunikacji z magistralą *Wishbone* , należało stworzyć moduły *wb_spi_master* i *wb_spi_slave*. Moduły te zawierają instancje *SPI*, oraz pomocnicze sygnały dla poprawnego przekazywania informacji. Listingi 20 i 21 przedstawiają instancję tych modułów. Parametr *SPI_SLAVE* określa szerokość wektora wyboru urządzeń *slave*. Sygnał *valid* dla modułu *spi_slave* jest równy koniunkcji sygnałów *wb.cyc* i *wb.stb*. Sygnał wyjściowy powstaje poprzez konkatenację wektora *data_out* z zerami. Sygnały błędu i zajętości zostały przypisane do zera, ponieważ projekt nie przewiduje ich wystąpienia.

Listing 20: Instancja pamięci instrukcji

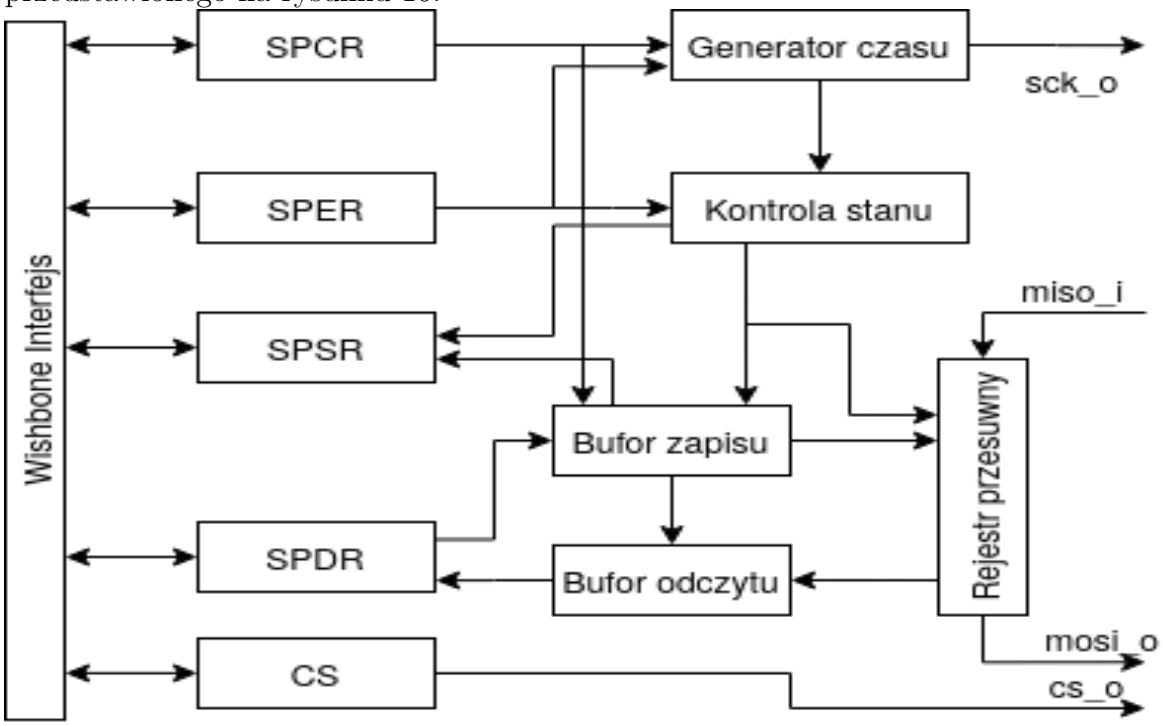
```
spi_master#(SPI_SLAVE) spi_master(
.clk_i(wb.clk_i),
.rst_i(~wb.rst_ni),
.cyc_i(wb.cyc),
.stb_i(wb.stb),
.adr_i(wb.addr[4:2]),
.we_i(wb.we),
.dat_i(wb.data_m[8-1:0]),
.dat_o(data_out),
.ack_o(wb.ack),
.inta_o(irq_o),
.sck_o(sck_o),
.cs_o(cs_o),
.mosi_o(mosi_o),
.miso_i(miso_i));
```

Listing 21: Instancja pamięci danych

```
spi_slave spi_slave(
.clk_i(wb.clk_i),
.rst_i(~wb.rst_ni),
.tx_dv_i(valid),
.tx_byte_i(wb.data_m[8-1:0]),
.rx_byte_o(data_out),
.rx_dv_o(wb.ack),
.spi_clk_i(sck_i),
.cs_i(cs_i),
.mosi_i(mosi_i),
.miso_o(miso_o));
```

3.7.2 *SPI Master*

Moduł *SPI Master* został zaimplementowany na podstawie schematu blokowego przedstawionego na rysunku 16.



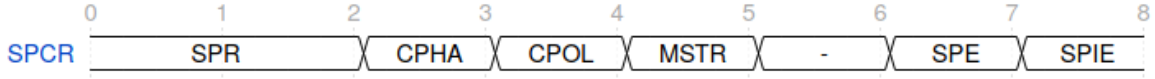
Rysunek 16: Model *SPI Master*.

Rejestry *SPCR*, *SPER*. *SPSR* i *SPDR* znajdują się pod odpowiednim adresem, tabela 11 przedstawia je.

Tabela 11: Lista rejestrów *SPI*

Nazwa	adres	ilość bitów	dostęp	opis
SPCR	0x00	8	zapis/odczyt	rejestr sterujący
SPSR	0x01	8	zapis/odczyt	rejestr statusu
SPDR	0x02	8	zapis/odczyt	rejestr danych
SPER	0x03	8	zapis/odczyt	rejestr rozszerzeń
CS	0x04	[SPI_SLAVE-1:0]	zapis/odczyt	rejestr wyboru *slave*

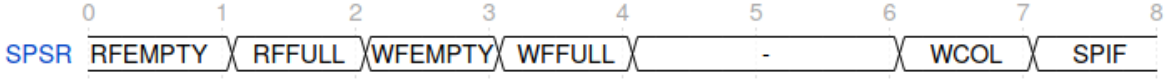
Zawartość rejestru *SPCR* jest przedstawiona na rysunku 17. Dostęp do wszystkich jego bitów jest zapis/odczyt.



Rysunek 17: Rejestr *SPCR*.

- Bit 7 *SPIE* - Uaktywnienie przerwań SPI - gdy bit ten jest jedynką logiczną, następuje włączenie przerwań, działa tylko gdy bit *SPIF* w rejestrze *SPER* również jest ustawiony na jedynkę logiczną
- Bit 6 *SPE* - Włączenie SPI - gdy bit ten jest jedynką logiczną, następuje włączenie interfejsu SPI,
- Bit 4 *MSTR* - Selekcja trybu *master* - gdy bit ten jest jedynką logiczną, następuje przełączenie urządzenia w tryb *master*, w projekcie bit ten zawsze jest jedynką logiczną,
- Bit 3 *CPOL* - Polaryzacja zegara, gdy bit ten jest jedynką logiczną, sygnał *SCK* ma wartość wysoką w stanie nieaktywnym. Gdy bit ten jest zerem logicznym, sygnał *SCK* ma wartość niską w stanie nieaktywnym,
- Bit 2 *CPHA* - Faza zegara, gdy bit ten jest jedynką logiczną na zboczu narastającym następuje przygotowanie, na zboczu opadającym następuje próbkowanie. Gdy bit ten jest zerem logicznym, zbocze narastające powoduje fazę próbkowania a opadające fazę przygotowania,
- Bit 0 i 1 *SPR* - Wybór częstotliwości zegarowej - bity te kontrolują częstotliwość sygnału *SCK*. Zależność ta, została pokazana w tabeli 13.

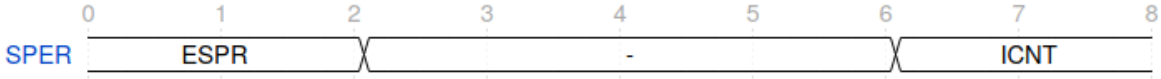
Zawartość rejestru *SPSR* została przedstawiona na rysunku 18.



Rysunek 18: Rejestr *SPSR*.

- Bit 7 *SPIF* - Znacznik przerwania SPI - po zakończeniu transferu, bitowi jest przypisywana jedynka logiczna, jeśli bit *SPIE* również jest jedynką logiczną, generowany jest sygnał przerwania,
- Bit 6 *WCOL* - Znacznik kolizji zapisu - bitowi jest przypisywana jedynka logiczna jeśli bit *WFFULL* jest w stanie wysokim i odbywa się zapis do rejestru danych,
- Bit 3 *WFFULL* - Znacznik zapełnienia *FIFO* przeznaczonego do zapisu - bitowi jest przypisywana jedynka logiczna, gdy *FIFO* zostanie zapełnione,
- Bit 2 *WFEMPTY* - Znacznik pustego *FIFO* przeznaczonego do zapisu - bitowi jest przypisywana jedynka logiczna, gdy *FIFO* jest puste,
- Bit 1 *WEFULL* - Znacznik zapełnienia *FIFO* przeznaczonego dla odczytu - bitowi jest przypisywana jedynka logiczny, gdy *FIFO* zostanie zapełnione,
- Bit 0 *WEEMPTY* - Znacznik pustego *FIFO* przeznaczonego dla odczytu - bitowi jest przypisywana jedynka logiczna, gdy *FIFO* jest puste.

W rejestrze *SPDR* znajdują się dane, przechowywane one są w dwóch *FIFO*, zapisu i odczytu. Ustawienie zera logicznego na bicie *SPE* powoduje wyczyszczenie *FIFO*. Zawartość rejestru *SPER* jest przedstawiona na rysunku 19.



Rysunek 19: Rejestr *SPER*.

- Bit 6 i 7 *ICNT* - Licznik przerwań - określa potrzebną ilość zakończonych cykli transferowych po których bitowi *SPIF* zostanie przypisana jedynka logiczna. Tabela 12 przedstawia te zależności.

Tabela 12: Lista rejestrów *SPI*

ICNT	Opis
2'b00	*SPIF* jest ustawiany po każdym zakończonym cyklu transferu
2'b01	*SPIF* jest ustawiany po dwóch zakończonych cyklach transferu
2'b10	*SPIF* jest ustawiany po trzech zakończonych cyklach transferu
2'b11	*SPIF* jest ustawiany po czterech zakończonych cyklach transferu

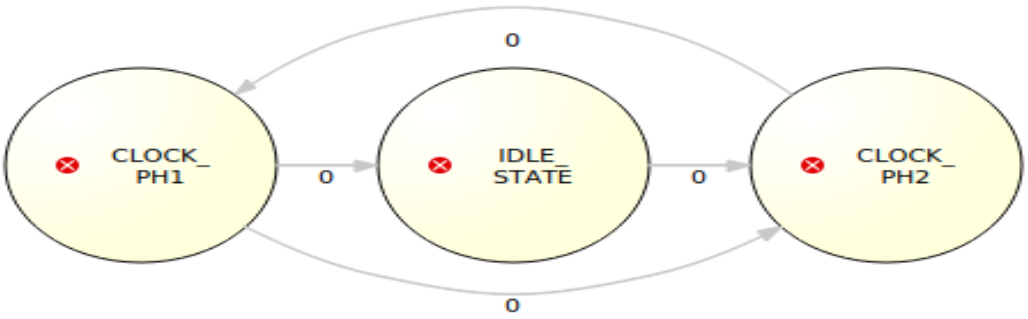
- Bit 0 i 1 *ESPR* - Rozszerzony wybór częstotliwości czasu - dodaje dodatkowe dwa bity pozwalające ustalić częstotliwość *SCK*. Tabela 13 przedstawia te zależności.

Tabela 13: Dzielnik zegara

ESPR	SPR	Dzielnik zegara
2'b00	2'b00	2
2'b00	2'b01	4
2'b00	2'b10	16
2'b00	2'b11	32
2'b01	2'b00	8
2'b01	2'b01	64
2'b01	2'b10	128
2'b01	2'b11	256
2'b10	2'b00	512
2'b10	2'b01	1024
2'b10	2'b10	2048
2'b10	2'b11	4096

Rejestr *CS* to rejestr wybory urządzeń typu *slave*.

Graf przedstawiony na rysunku 20 przedstawia maszynę stanów odpowiedzialną za transfer informacji.



Rysunek 20: Graf *FSM SPI master*.

Fazą domyślna jest *IDLE_STATE*. Jej przebieg został przedstawiony na listingu 22.

Listing 22: Faza *IDLE_STATE*

```
IDLE_STATE: begin
        bcnt  <= 3'h7;
        treg  <= wfdout;
        sck_o <= cpol;

        if (~wfempty) begin
                wfre  <= 1'b1;
                state <= CLOCK_PH2;
                if (cpha) sck_o <= ~sck_o;
        end
end
```

Ustawiany jest pomocniczy licznik przesłanych bitów *bcnt*, przypisywany bajt danych do wektora *treg* pochodzących z *FIFO* oraz ustawiana wartość początkowa *sck_o*. Jeśli *FIFO* nie było puste rozpoczyna się transmisja. Sygnał *wfre* zezwala na odczyt informacji z *FIFO*, stan przechodzi do następnej fazy *CLOCK_PH2* oraz ustawiana jest faza zegara.
W stanie *CLOCK_PH2* sygnał *sck_o* zostaje zanegowany i następuje kolejny stan *CLOCK_PH1*.
Stan *CLOCK_PH1* odpowiedzialny jest za aktualizację wektora *treg*. Jej przebieg jest przedstawiony na listingu 23.

Listing 23: Faza *CLOCK_PH1*

```
CLOCK_PH1:
        if (ena) begin
                treg <= {treg[6:0], miso_i};
                bcnt <= bcnt -3'h1;

                if (~|bcnt) begin
                        state <= IDLE_STATE;
                        sck_o <= cpol;
                        rfwe  <= 1'b1;
                end else begin
                        state <= CLOCK_PH2;
                        sck_o <= ~sck_o;
                end
        end
```

Jeśli sygnał *ena* jest w stanie wysokim następuje przesunięcie rejestru i wpisanie na pozycji najmniej znaczącego bitu, informacji pochodzącej z urządzenia *slave*. Następnie licznik przekazanych bitów zostaje dekrementowany, jeśli jego wartość będzie równa zero, stan powraca do *IDLE_STATE* oraz zezwala na zapis do *FIFO* przechowującego dane z urządzenia *slave*. Do wyprowadzenia *mosi_o* zostało zastosowane przypisanie ciągłe najbardziej znaczącego bitu wektora *treg*.
Implementacja generatora sygnału zegarowego *sck_o* została przedstawiona na listingu 24.

Listing 24: Generowanie sygnału zegarowego

```
always @(posedge clk_i)
        if(spe & |clkcnt & |state)
                clkcnt <= clkcnt - 11'h1;
        else
                case (espr)
                        .
                        .
                endcase
        reg ena = ~|clkcnt;
```

Podczas narastającego zbocza sygnału zegarowego pochądzącego z rdzenia następuje sprawdzanie czy interfejs *SPI* został włączone, poprzedni cykl generowania sygnału czasu *sck_o* został zakończony i sprawdzenie aktualnego statusu *FSM*, jeśli znajduje się w stanie *IDLE_STATE* następuje ponowne przypisanie wartości do wektora *clkcnt*. Wartości zapisywane do tego wektora są przedstawione w tabeli 13. Jeśli wszystkie warunki zostały spełnione, następuje dekrementacja rejestru *clkcnt*. Jeśli będzie równy zeru, stan wysoki zostanie przypisany do sygnału *ena*, jest on używany w fazach *FSM*. Do magazynowania danych przeznaczono dwa bufory *FIFO*, dla danych wejściowych i wyjściowych, każdy o parametryzowanej głębokości, w projekcie parametr ten jest równy cztery. Posiada znacznik zapełnienia jak i pustego buforu. Gdy nadejdzie sygnał zapisu, licznik zapełnienia *FIFO* zostaje inkrementowany, dane zostają zapisane do komórki i pozycja komórki przeznaczonej dla zapisu zostaje inkrementowana. Podczas odczytu licznik zapełnienia zostaje dekrementowany, dana zostaje odczytana z buforu, znacznik pozycji odczyty zostaje inkrementowany. Gdy bufor jest pełny, zapis jest niedostępny, jeśli bufor jest pusty, odczyt jest niedostępny. Listing 25 przedstawia przebieg tych faz.

Listing 25: Model *FIFO*

```
case ({rd_i, wr_i})
        2'b00: begin
                full_indicator <= full_indicator;
        end
        2'b01: begin
                if  (full_indicator < 2**FIFO_BIT)      begin
                        full_indicator <= full_indicator + 1;
                        fifo_mem[wr_ptr] <= data_i;
                        wr_ptr <= wr_ptr + 1;
                end
        end
        2'b10: begin
                if (full_indicator > 0) begin
                        data_o <= fifo_mem[rd_ptr];
                        full_indicator <= full_indicator - 1;
                        rd_ptr <= rd_ptr + 1;
                end
        end
        2'b11: begin
                wr_ptr <= wr_ptr + 1;
                rd_ptr <= rd_ptr + 1;
                data_o <= fifo_mem[rd_ptr];
                fifo_mem[wr_ptr] <= data_i;
                full_indicator <= full_indicator;
        end
endcase
```

3.8 Interfejs *I2C*

3.8.1 Komunikacja z magistralą *Wishbone*

W celu poprawnej komunikacji z magistralą należało stworzyć dodatkowy moduł. Jego celem jest poprawne połączenia instancji *I2C* która została przedstawiona na listingu 26 z interfejsem.

Listing 26: Bufor trójstanowy

```
i2c_top i2(
        .wb_clk_i(wb.clk_i),
        .wb_rst_i(!wb.rst_ni),
        .arst_i(1'b0),
        .wb_adr_i(wb.addr[4:2]),
        .wb_dat_i(wb.data_m[7:0]),
        .wb_dat_o(data_o),
        .wb_we_i(wb.we),
        .wb_stb_i(wb.stb),
        .wb_cyc_i(wb.cyc),
        .wb_ack_o(wb.ack),
        .wb_inta_o(),
        .scl_pad_i(scl_pad_i),
        .scl_pad_o(scl_pad_o),
        .scl_padoen_o(scl_padoen_o),
        .sda_pad_i(sda_pad_i),
        .sda_pad_o(sda_pad_o),
        .sda_padoen_o(sda_padoen_o)      );
```

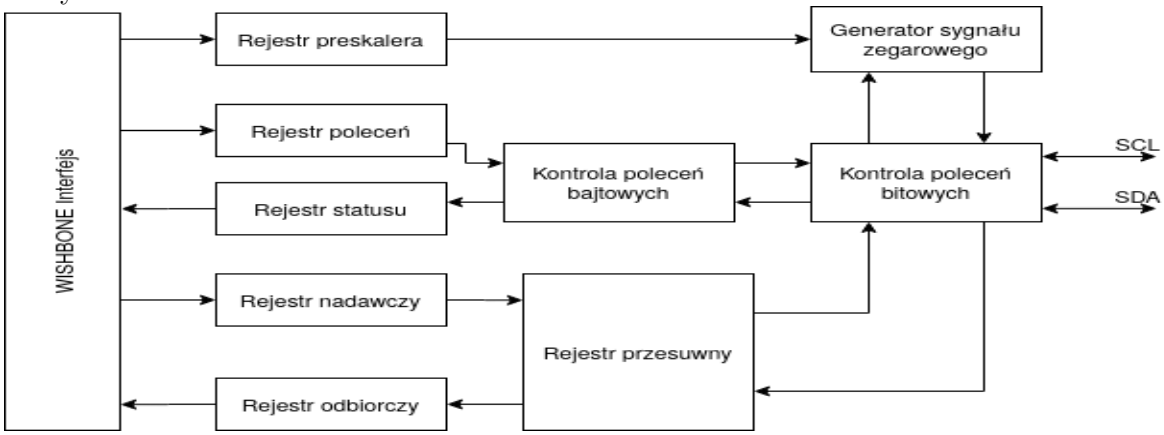
Sygnały *SDA* i *SCL* są dwukierunkowe, należało więc zastosować bufory trójstanowe w celu wprowadzenia stanu wysokiej impedancji by unikać konfliktów stanu podczas transmisji innego urządzenia. Jego opis został przedstawiony na listing 27.

Listing 27: Bufor trójstanowy

```
assign IO_SCL = scl_padoen_o ? 'bz : scl_pad_o;
assign IO_SDA = sda_padoen_o ? 'bz : sda_pad_o;
assign scl_pad_i = IO_SCL;
assign sda_pad_i = IO_SDA;
```

3.8.2 Implementacja *I2C*

Interfejs *I2C* został zaimplementowany na podstawie architektury przedstawionej na rysunku 21.



Rysunek 21: Architektura interfejsu *I2C*.

By uzyskać dostęp dla danego rejestru należy odwołać się do poprawnego adresu. Tabela 14 przedstawia przypisane adresy dla danego rejestru.

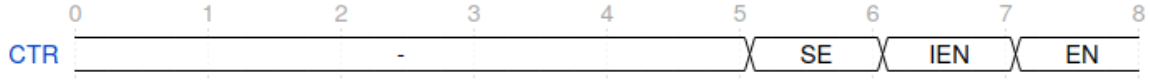
Tabela 14: Lista rejestrów interfejsu*I2C*

Nazwa	adres	ilość bitów	dostęp	opis
PRERlo	0x00	8	zapis/odczyt	Preskaler zegara, niski baj
PRERhi	0x01	8	zapis/odczyt	Preskaler zegara, wysoki bajt
CTR	0x02	8	zapis/odczyt	Rejestr kontrolny
TXR	0x03	8	zapis	Rejestr nadawczy
RXR	0x03	8	odczyt	Rejestr odbiorczy *slave*
CR	0x04	8	zapis	Rejestr poleceń
SR	0x04	8	odczyt	Rejestr stanu
SLA	0x07	7	zapis/odczyt	Rejestr adresu slave

Rejestr preskalera służy do ustalenia częstotliwości sygnału zegarowego *SCL*. Wartość jaką należy wpisać do tego rejestru można obliczyć wzorem 1.

$$preskaler = \frac{wb\_clk\_i}{5 * SCL} - 1 \tag{1}$$

Rejestr *CTR* został przedstawiony na rysunku 22.

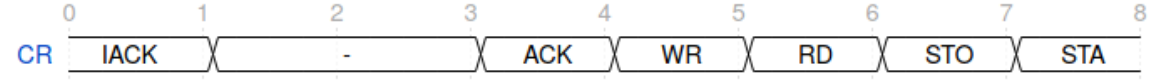


Rysunek 22: Rejestr kontrolny.

- Bit 7 - EN - Uaktywnienie interfejsu *I2C* - gdy bit ten jest logiczną jedynką interfejs *I2C* jest aktywny, przypisanie zera logicznego dezaktywuje go,
- Bit 6 - IEN - Uaktywnienie przerwań - gdy bit ten jest logiczną jedynką przerwania interfejsu *I2C* zostają włączone,
- Bit 5 - SE - Uaktywnienie trybu *slave* - gdy bit ten jest logiczną jedynką następuje zmiana trybu z *master* na *slave*.

Rejestr nadawczy i odbiorczy składają się z jednego bajtu który zostanie nadany lub został odebrany.

Rejestr *CR* został przedstawiony na rysunku 23.

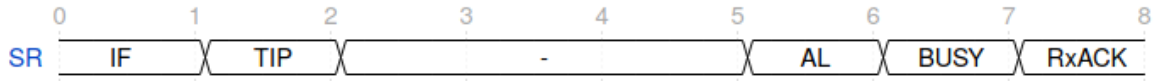


Rysunek 23: Rejestr komend.

- Bit 7 - STA - wysyła sygnał startu,
- Bit 6 - STOP - wysyła sygnał stopu,
- Bit 5 - RD - Odczyt - gdy zostanie wpisana jedynka logiczna do tego bitu, dane zostaną odczytane z urządzenia *slave*,

- Bit 4 - WR - Zapis - gdy zostanie wpisana jedynka logiczna do tego bitu, dane zostaną zapisane do urządzenia *slave*,
- Bit 3 - ACK - Bit potwierdzenia - wysłanie bitu *NACK* gdy jest przypisane logiczna jedynka lub wysłanie bitu *ACK* gdy jest przypisane logiczne zero,
- Bit 1 - IACK - Bit przerwania potwierdzenia - gdy zostanie przypisana jedynka, czyści wysyłanie przerwania.

Rejestr *SR* został przedstawiony na rysunku 24



Rysunek 24: Rejestr statusu.

- Bit 7 - RxACK - potwierdzenie pochodzące z urządzenia typu *slave*. Gdy bit przyjmuje jedynkę logiczną sygnał to otrzymano*NACK*, gdy jest zerem logicznym to otrzymano *ACK*,
- Bit 6 - BUSY - sygnał zajętości, bit przyjmuje jedynkę logiczną po sygnale *START* , zero logiczne pojawia się po sygnale *STOP*,
- Bit 5 - AL - utrata arbitrażu - bit przyjmuje jedynkę logiczną gdy pojawi się sygnał *STOP* bez żądania lub wysyłany jest wysoki stan sygnału *SDA* lecz znajduje się na nim niski,
- Bit 1 - TIP - transfer w trakcie - bit przyjmuje jedynkę logiczną gdy dane są przekazywane,
- Bit 0 - IF - Flaga przerwań - bit przyjmuje jedynkę logiczną gdy jeden bajt danych zostanie przesłany lub arbitraż zostanie utracony.

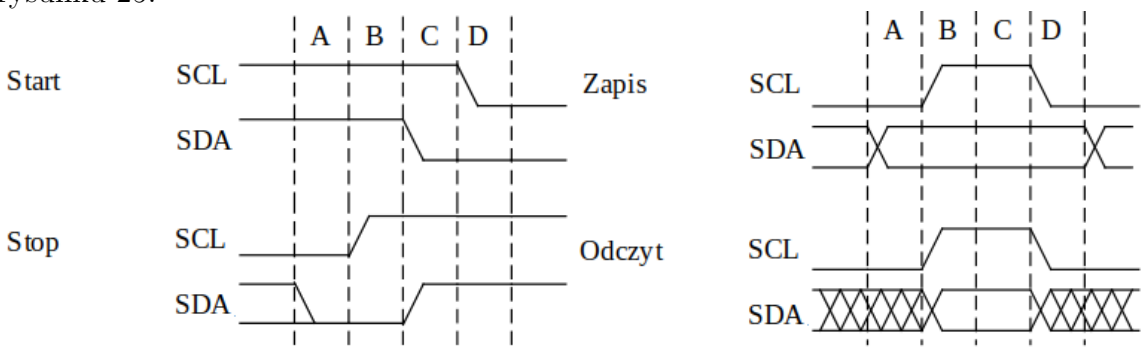
Transmisja składa się z czterech cykli:

1. Sygnał START - urządzenie *master* generuje sygnał START jeśli bit *STA* pochodzący z rejestru komend jest jedynką logiczną i gdy jeden z bitów *WR* lub *RD* jest w stanie wysokim.
2. Wysłanie adresu *slave* - urządzenie *master* wysyła adres, składa się on z siedmiu bitów, ósmy bit oznacz chęć zapisu(0) lub odczytu(1), bit ten jest najmniej znaczący. Urządzenie *slave* odpowiada wysyłając potwierdzenie ustawiając linię *SDA* na stan niski przy dziewiątym cyklu *SCL*.
3. Przesył danych - po otrzymaniu potwierdzenie z danego adresu urządzenia *slave*, rozpoczyna się transmisja bajtowa. Kierunek jej określił bit *WR/RD*. Po każdym bajcie występuje bit potwierdzenia, jeśli *slave* przekaże sygnał *NACK*, urządzenie

master może wygenerować sygnał *STOP* by zakończyć transmisje lub powtórzyć sygnał *START*. Gdy urządzenie *master* po odbiorze danych nie przekaże sygnału *ACK*, urządzenie *slave* zwalnia linię *SDA* by urządzenie *master* mogło wygenerować sygnał *STOP* lub ponowny *START*.

4. Sygnał STOP - urządzenie *master* kończy transmisje.

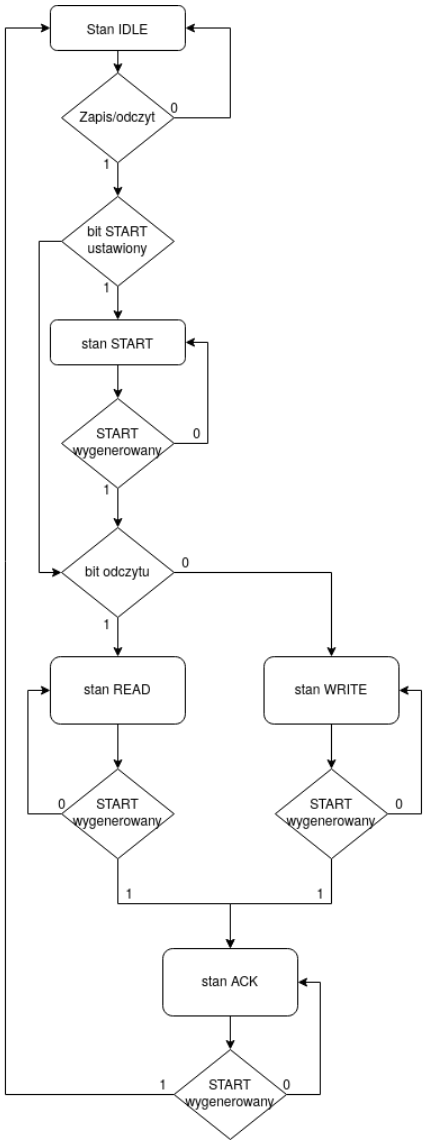
Moduł kontroli poleceń bajtowych odpowiedzialny jest za ruch bajtowy interfejsu *I2C*. Pobiera dane z rejestru komend i z nich sekwencje bajtowe. Jeśli w rejestrze bity *START*. *STOP* i *READ* są w stanie wysokim. Moduł stworzy sekwencje startującą transmisje, odpowiedzialną za odczyt danych z urządzenia *slave* oraz wygeneruje syngał *STOP*. Każda sekwencja jest rozbijana na pojedyncze bity, które są przekazywany do modułu kontroli komend bitowych.
Moduł kontroli komend bitowych jest odpowiedzialny za poprawne ustawianie stanów na linii *SDA* i *SCL*. Otrzymuje on informację jaka komenda jest aktualnie wykonywana. Każda operacja jest podzielona na cztery części tak jak zostało to przedstawione na rysunku 25.



Rysunek 25: Podział operacji na części.

Algorytm działania modelu został przedstawiony na rysunku 26

.



Rysunek 26: Algorytm działania modelu.

3.9 Timer

3.9.1 Komunikacja z magistralą *Wishbone*

W celu poprawnej komunikacji z magistralą, należało opisać moduł łączący. Posiada on w sobie inicjalizację modułu timera oraz przypisanie ciągłe do zera sygnału *stall*.

3.9.2 Implementacja timera

Timer posiada cztery rejestry, zostały one przedstawione w tabeli 15. Z każdym wysokim zboczem zegara wzrasta wartość 64-bitowego licznika *mtime*. Przerwanie zostanie generowane gdy wartość *mtime* będzie równa z *mtimecmp*.

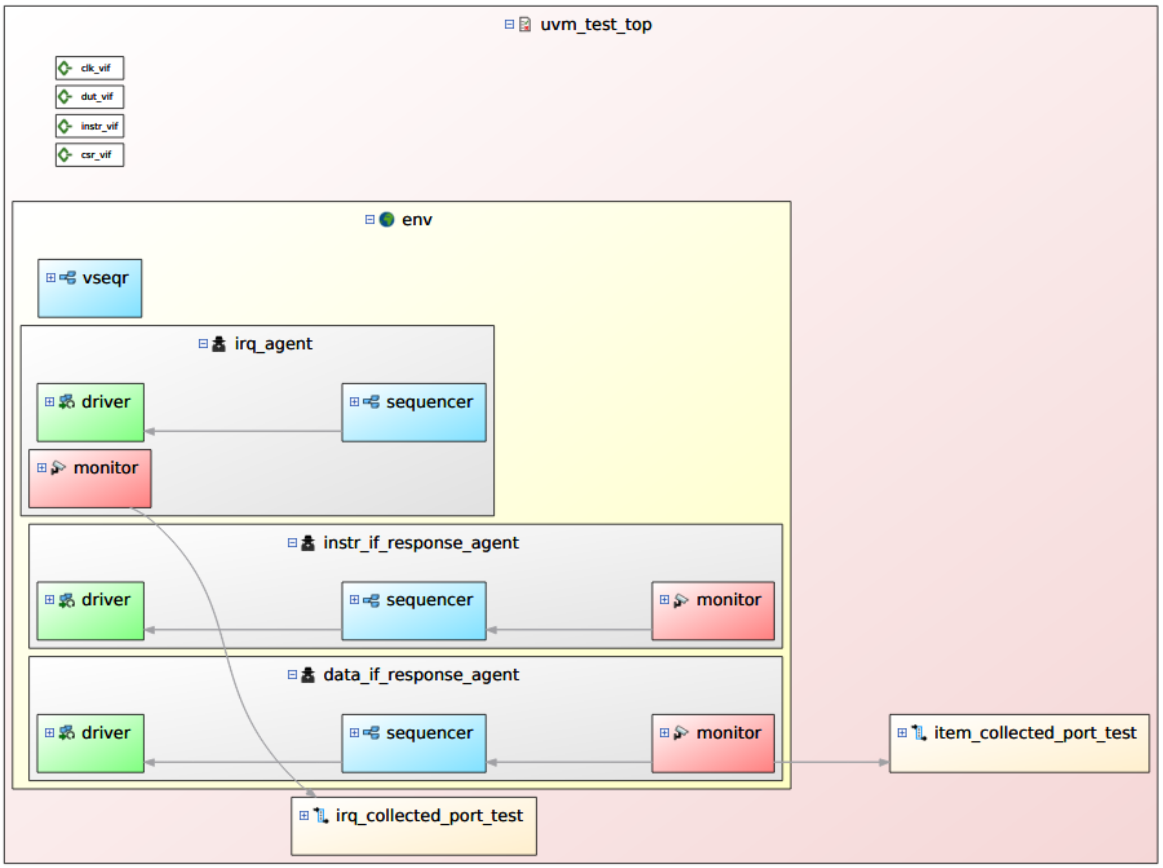
Tabela 15: Lista rejestrów timera

Nazwa	adres	ilość bitów	dostęp	opis
MTIME_LOW	0x00	32	odczyt	Aktualna wartość licznika, niskie słowo
MTIME_HIGH	0x04	32	odczyt	Aktualna wartość licznika, wysokie słowo
MTIMECMP_LOW	0x08	32	zapis/odczyt	Wartość do porównania, niskie słowo
MTIMECMP_HIGH	0x0f	32	zapis/odczyt	Wartość do porównania, wysokie słowo

4 Weryfikacja

4.1 Rdzeń Ibex z RISCV DV

W celu weryfikacja pracy rdzenia przeprowadzono testy, wykorzystując bibliotekę UVM i generator rozkazów RISCV-DV. Wygenerowane programy są symulowane na poziomie RTL, następnie poprzez program OVPSim jest generowany złoty wzór, który jest porównywany do symulacji. Po dokonaniu komparacji pojawia się podsumowanie, informujące ile testów ma wynik pozytywny lub negatywny. Graf testu UVM jest przedstawiony na rysunku 27.



Rysunek 27: Graf UVM testu rdzenia Ibex

Architektura UVM składa się z dwóch agentów przeznaczonych na obsługę pamięci. Jeden z nich obsługuje *LSU*, drugi jest odpowiedzialny za przechwytywanie instrukcji. Czekają one na żądanie pochodzące z rdzenia, następnie udzielają potrzebnych danych lub instrukcji. Agent jest przeznaczony do losowego wysyłania przerwań rdzeniowi. Wszystkie testy dziedziczą po głównej klasie *core_ibex_base_test* i koordyunują one przepływ pojedynczego testu, od wczytania skomplikowanego programu binarnego i sprawdzanie stanu rdzenia.

Jest 35 programów sprawdzających prace rdzenia. Każdy z nich testuje daną funkcjonalność między innymi: instrukcje arytmetyczne, instrukcje skokowe pomiędzy wieloma podprogramami, wprowadzenie błędnych instrukcji, resetowanie rdzenia w losowych

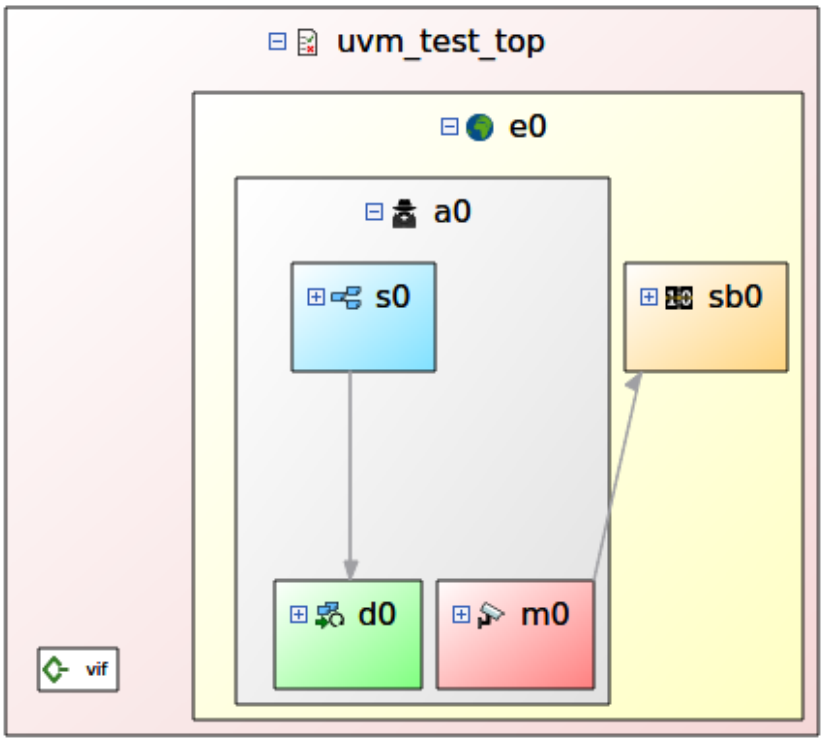
momentach, używanie przerwań w losowych momentach, odczyt z pamięci o błędnym adresie itp. Przykładowe wyjście konsoli zostało przedstawione na listingu 28. Symulator OVPSim wytworzył 10184 instrukcje gdy symulacja RTL 10185. Różnica jednej instrukcji wynika z symulatora Riviera-PRO, po zakończeniu symulacji programu dokłada on przypisanie zera do rejestru x0. Po symulacji wszystkich testów otrzymano trzydzieści pięć wyników pozytywnych, co oznacza, że rdzeń działa poprawnie.

Listing 28: Przykładowe porównanie

```
Comparing ovpsim/DUT sim result :/asm_tests/riscv_arithmetic_basic_test.0.o
Processing ibex log :/riscv_arithmetic_basic_test.0/trace_core_00000000.log
Processed instruction count : 10184
CSV saved to :/rtl_sim/riscv_arithmetic_basic_test.0/trace_core_00000000.csv
Processing ovpsim log :/instr_gen/ovpsim_sim/riscv_arithmetic_basic_test.0.log
Processed instruction count : 10185
CSV saved to :/instr_gen/ovpsim_sim/riscv_arithmetic_basic_test.0.csv
```

4.2 Pamięć RAM

Weryfikacja polega na losowym generowaniu sygnału zezwolenia na zapis, poprawności danych, adresu i samych danych. Ilość tych pakietów również jest losowa. Dane które zostały wygenerowane przekazywane są poprzez wirtualny interfejs do pamięci RAM. Jeśli sygnały *valid* i *we* są w stanie wysokim, dane są przypisywane do tablicy asocjacyjnej o typie komórek 4-bitów do których przypisywany jest aktualny adres. Po zakończonym teście w fazie *check_phase*, wartości z tablicy są porównywane z stanem pamięci. Jeśli wszystkie wartości są zgodne test uzyskał wynik pozytywny. Graf UVM został przedstawiony na rysunku 28.



Rysunek 28: Graf UVM testu pamięci RAM

Testy zostały przeprowadzone dla pamięci jednoportowej i dwuportowej. Ilość generowanych sekwencji jest mieści się w puli 500:1000. Ilość komórek testowanej pamięci jest równa 32 co sprawia, że adres mieści się w czterech bitach. Listing 29 przedstawia raport UVM.

Listing 29: UVM raport

```
# KERNEL: ** Report counts by severity
# KERNEL: UVM_INFO : 2871
# KERNEL: UVM_WARNING :    0
# KERNEL: UVM_ERROR :    0
# KERNEL: UVM_FATAL :    0
# KERNEL: ** Report counts by id
# KERNEL: [DRV]    940
# KERNEL: [MON]    947
# KERNEL: [RNTST]      1
# KERNEL: [SEQ]      1
# KERNEL: [TEST_DONE]      1
# KERNEL: [UVM/RELNOTES]      1
# KERNEL: [test_pass]    32
# KERNEL: [test_pass_final]      1
# KERNEL: [uvm_test_top.e0.sb0]    947
```

Można z niego odczytać, że:

- Test przeszedł bez problemów, o czym świadczy zerowa ilość wiadomości typu *UVM_WARNING*, *UVM_ERROR* i *UVM_FATAL*,
- zostało wysłanych 940 sekwencji, świadczy o tym liczba wiadomości *DRV*

- Monitor zaraportował, że dane zostały zmienione 947 razy, różnica między *DRV* powstała ponieważ, 7 sekwencji zostało wysłanych ręcznie w celu resetu pamięci itp.
- informacja o *test_pass* pojawiła się 32 razy, co zgadza się z dostępna liczbą komórek pamięci
- informacja o *test_pass_final* sygnalizuje, że test pokrył wszystkie komórki pamięci

4.3 GPIO

Weryfikacja polega na losowym generowania sygnałów pochodzących z zewnątrz, sygnałów wychodzących, zezwolenia na zapis i adresu. Ilość wytworzonych pakietów jest losowa. Wygenerowane dane są przekazywane poprzez wirtualny interfejs do modułu GPIO. Wysłane dane są zapisywane w dynamicznych tablicach i porównywane z wartością sygnałów na liniach *O_LED* i *data_s*. Struktura testu UVM jest identyczna jak w weryfikacji pamięci RAM i została przedstawiona na rysunku 28.

4.4 UART

Weryfikacja modułu UART polega na generowaniu losowej wartości sygnału *TX* i danych przekazywanych do wejścia *RX*. Informacja o ilości transmisji przekazywana jest w głównym module testu. Gdy randomizacja wartości sygnału *TX* się powiedzie rozpoczyna się transmisja. Jeśli dane zostały poprawnie wysłane zwracana jest informacja o powodzeniu. W dalszej kolejności następuje randomizacja 8-bitowej zmiennej przeznaczonej do przekazania na port *RX*. Dane te są wysyłane bitowo, w tym celu wykorzystano zagnieżdzoną w pętli *for*, pętle *repeat*. Wartość tej pętli jest równa czasie trwania bitu i następuje w niej zmiana sygnału zegarowego. Po odczekaniu wymaganej ilości sygnałów zegarowych następuje wysłanie kolejnego bitu. Po wysłaniu wszystkich bitów następuje komparacja zmiennej przygotowanej do wysłania z wartością odczytaną. Po zakończeniu jest losowa kolejna transakcja. Struktura testu UVM została przedstawiona na rysunku 28, przykładowa transakcja została pokazana na listingu 30

Listing 30: Przykładowa transakcja UART

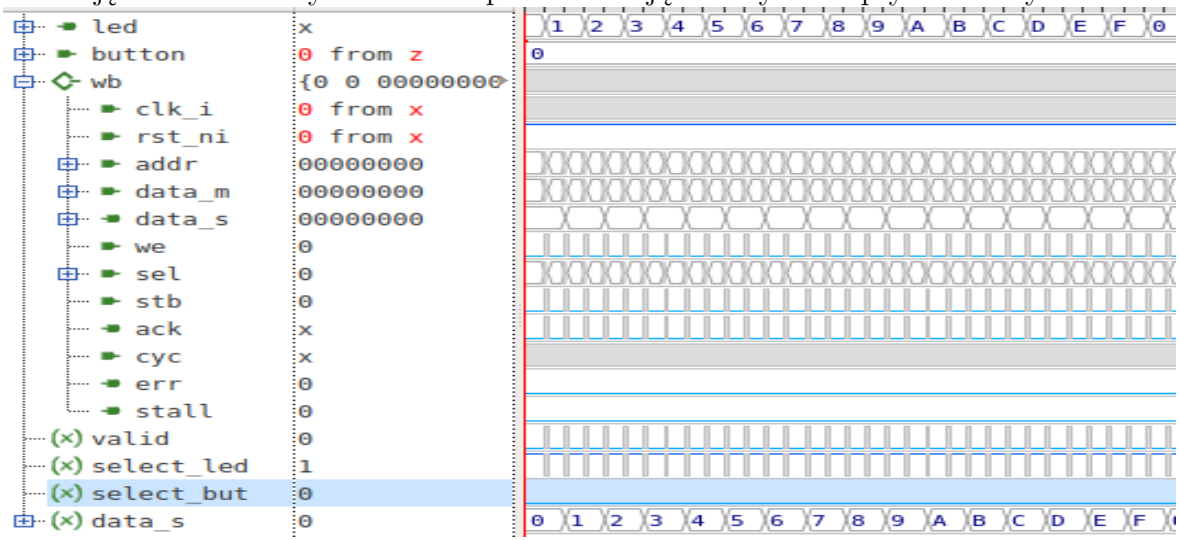
```
numer transakcji = 19
start = 1,        tx_data_in = b,          tx_active = 0
TX PASS
Oczekiwana = 77,          otrzymana = 77
RX PASS
```

4.5 SPI

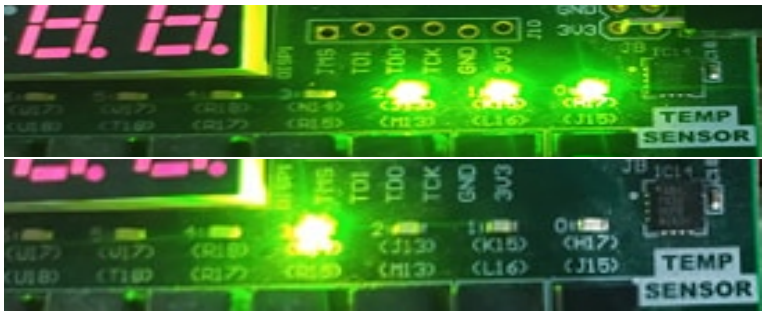
Weryfikacja modułu SPI polega na generowaniu losowych danych przeznaczonych do transmisji, następnie monitorowane wyjście *MOSI* w celu zebrania bitów i sprawdzenia poprawności transmisji. Pierwszą częścią jest odpowiednie ustawienie rejestrów w celu rozpoczęcia transmisji. Po każdym ustawieniu rejestrów ich wartości są sczytywane w celu weryfikacji. Gdy moduł *SPI* jest gotów do transmisji uruchamiany jest *task* z przesyłem danych oraz ich odbiorem. Poprawność jest sprawdzana po każdej wysłanej transakcji. Struktura testu UVM została przedstawiona na rysunku 28.

4.6 Przykładowy program

W celu weryfikacji działania systemu na chipie został przygotowany program testowy. Program ten wpisuje daną na adres wyjścia *GPIO*. Dana ta jest inkrementowana co pół sekundy. Wynik symulacji został przedstawiony na zdjęciu 29. Rysunek 30 przedstawiają diody na płytce Nexys4 DDR.



Rysunek 29: Symulacja działania programu



Rysunek 30: Działający program na płytce FPGA NEXYS4 DDR

5 Wyniki eksperymentów

5.1 Synteza modułów peryferyjnych

Została przeprowadzona synteza modułów peryferyjnych. Wykorzystane zasoby płytki *FPGA* zostały przedstawione w tabelach:

- Synteza modułu *SPI*, wykorzystanie zasobów zostało przedstawione w tabeli 16,
- synteza modułu *UART*, wykorzystanie zasobów zostało przedstawione w tabeli 17,
- synteza modułu *I2C*, wykorzystanie zasobów zostało przedstawione w tabeli 18,
- synteza modułu *RAM*, wykorzystanie zasobów zostało przedstawione w tabeli 19,
- synteza modułu timera, wykorzystanie zasobów zostało przedstawione w tabeli 20,
- synteza rdzenia Ibex, wykorzystanie zasobów zostało przedstawione w tabeli 21.

Tabela 16: Wykorzystanie zasobów zsyntezowanego modułu *SPI*

Zasób	Wykorzystanie	Dostępność	Wykorzystanie [%]
LUT	99	63400	0.16
LUTRAM	16	19000	0.08
FF	88	126800	0.07
IO	56	210	26.67
BUFG	1	32	3.13

Tabela 17: Wykorzystanie zasobów zsyntezowanego modułu *UART*

Zasób	Wykorzystanie	Dostępność	Wykorzystanie [%]
LUT	75	63400	0.12
FF	60	126800	0.05
IO	50	210	23.81
BUFG	1	32	3.13

Tabela 18: Wykorzystanie zasobów zsyntezowanego modułu *I2C*

Zasób	Wykorzystanie	Dostępność	Wykorzystanie [%]
LUT	227	63400	0.36
FF	178	126800	0.14
IO	53	210	25.24
BUFG	1	32	3.13

Tabela 19: Wykorzystanie zasobów zsyntezowanego modułu pamięci *RAM*

Zasób	Wykorzystanie	Dostępność	Wykorzystanie [%]
LUT	30	63400	0.05
FF	35	126800	0.03
BRAM	32	135	23.7
IO	145	210	69.05
BUFG	2	32	6.25

Tabela 20: Wykorzystanie zasobów zsyntezowanego modułu *timera*

Zasób	Wykorzystanie	Dostępność	Wykorzystanie [%]
LUT	151	63400	0.24
FF	163	126800	0.13
IO	81	210	38.57
BUFG	1	32	3.13

Tabela 21: Wykorzystanie zasobów zsyntezowanego modułu rdzenia *Ibex*

Zasób	Wykorzystanie	Dostępność	Wykorzystanie [%]
LUT	2858	63400	4.08
LUTRAM	48	63400	0.25
FF	929	126800	0.73
DSP	1	240	0.42
IO	254	210	120.95
BUFG	2	32	6.25

Duże wykorzystanie zasobów portów *In/Out* spowodowane jest użyciem magistrali *Wishbone*.

5.2 Synteza systemu na chipie

Tabela 22 przedstawia wykorzystanie zasobów dostępnych na płytce *FPGA Nexys4 DDR*. Po użyciu dyrektywy *(* ram_decomp = "power"*)*, która powoduje rozbicie pamięci na mniejsze części, spowodowało zmianę wykorzystania zasobów. Tabela 23 przedstawia wykorzystanie tych zasobów. Widoczny jest wzrost zapotrzebowania *LUT*, dlatego finalnie projekt jest syntezowany bez użycia dyrektywy. Próba wymuszenia zastąpienia pamięci rejestrami zakończyła się niepowodzeniem. Płytka FPGA Nexys4 DDR jest zbyt uboga w zasobach by zapewnić potrzebną ilość rejestrów. Wymagana liczba to 4194304, limit płytki to 1000000.

Tabela 22: Wykorzystanie zasobów zsyntezowanego modułu systemu na chipie z pamięcią dwu-portową

Zasób	Wykorzystanie	Dostępność	Wykorzystanie [%]
LUT	3333	63400	5.26
LUTRAM	64	19000	0.34
FF	1478	126800	1.17
BRAM	32	135	23.70
DSP	1	240	0.42
IO	22	210	10.48
BUFG	4	32	12.50
PLL	1	6	16.97

Tabela 23: Wykorzystanie zasobów zsyntezowanego modułu systemu na chipie wraz z dyrektywą *ram_decompz* pamięcią dwu-portową

Zasób	Wykorzystanie	Dostępność	Wykorzystanie [%]
LUT	4143	63400	6.53
LUTRAM	64	19000	0.34
FF	1492	126800	1.18
BRAM	32	135	23.70
DSP	1	240	0.42
IO	22	210	10.48
BUFG	4	32	12.50
PLL	1	6	16.97

Liczba wykorzystanych zasób zsyntezowanego systemu na chipie nie jest równa sumie wszystkich modułów. Przyczyną jest optymalizacja która zachodzi w trakcie syntezy.

Tabela 24 przedstawia wykorzystanie zasobów systemu z jedno-portową pamięcią RAM.

Tabela 24: Wykorzystanie zasobów zsyntezowanego modułu systemu na chipie z pamięcią jedno-portową

Zasób	Wykorzystanie	Dostępność	Wykorzystanie [%]
LUT	3379	63400	6.53
LUTRAM	64	19000	0.34
FF	1477	126800	1.17
BRAM	64	135	47.41
DSP	1	240	0.42
IO	22	210	10.48
BUFG	4	32	12.50
PLL	1	6	16.97

Zauważalny jest znaczący wzrost wykorzystania BRAM. Dlatego domyślnie projekt wykorzystuje dwu-portową pamięć RAM. W celu jeszcze większej optymalizacji wykorzystana dyrektywę *AreaMapLargeShiftRegToBRAM*, wymusza ona wykorzystanie BRAM dla rejestrów przesuwnych. Tabela 25 przedstawia wykorzystanie zasobów po przeprowadzonej syntezie.

Tabela 25: Wykorzystanie zasobów zsyntezowanego modułu systemu na chipie z pamięcią dwu-portową z użyciem dyrektywy *AreaMapLargeShiftRegToBRAM*

Zasób	Wykorzystanie	Dostępność	Wykorzystanie [%]
LUT	3327	63400	5.25
LUTRAM	64	19000	0.34
FF	1478	126800	1.17
BRAM	32	135	23.70
DSP	1	240	0.42
IO	22	210	10.48
BUFG	4	32	12.50
PLL	1	6	16.97

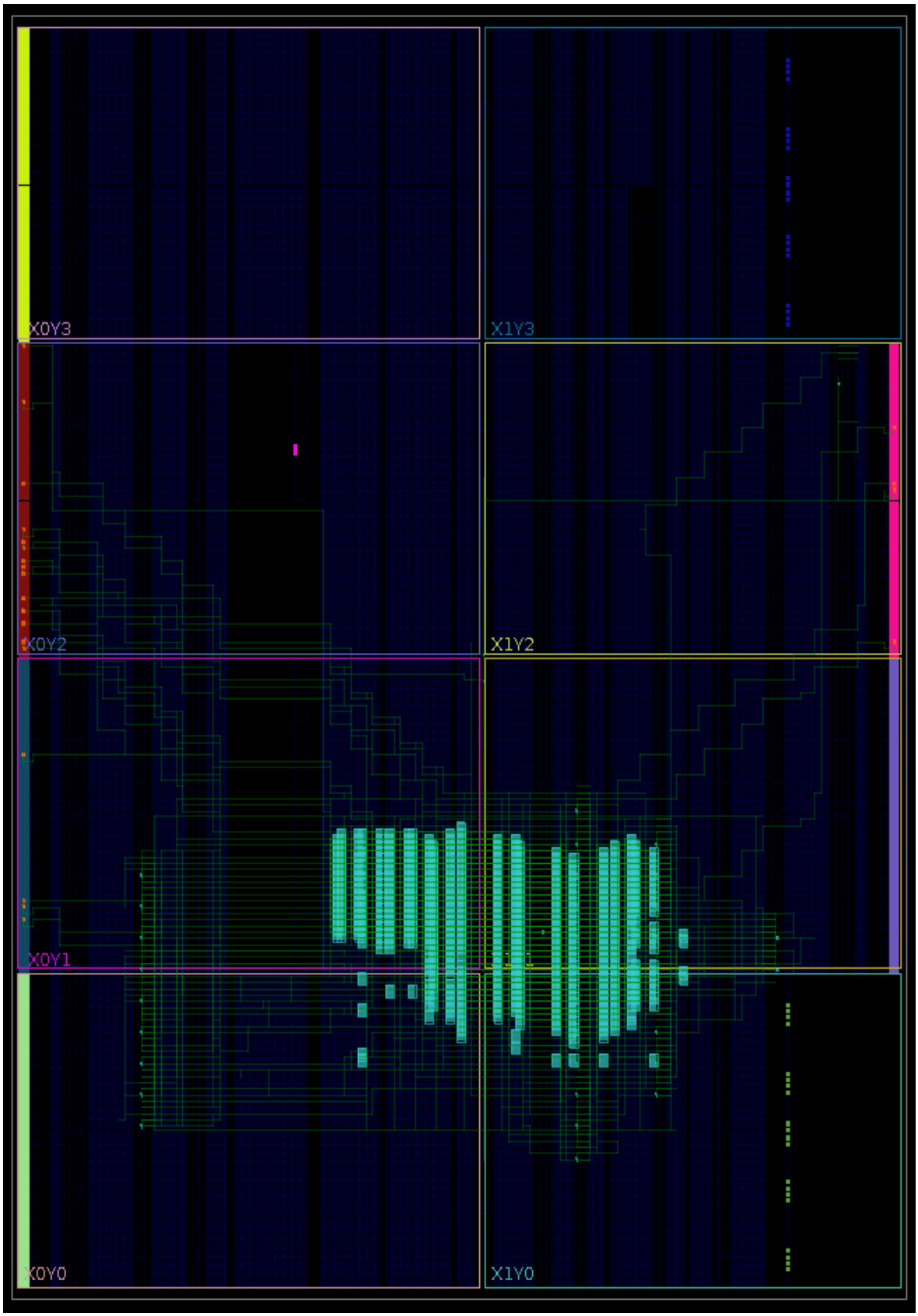
Dzięki temu zabiegowi zaoszczędzono sześć LUT i nie spowodowało to dodatkowego wykorzystania innych zasobów. Po zastosowaniu dodatkowej dyrektywy
Kolejna użyta dyrektywa ma na celu optymalizację bloków FSM to *sequential*. Tabela 26 przedstawia zużycie zasobów płytki.

Tabela 26: Wykorzystanie zasobów zsyntezowanego modułu systemu na chipie z pamięcią dwu-portową z użyciem dyrektywy *AreaMapLargeShiftRegToBRAM* i *sequential*

Zasób	Wykorzystanie	Dostępność	Wykorzystanie [%]
LUT	3318	63400	5.23
LUTRAM	64	19000	0.34
FF	1476	126800	1.16
BRAM	32	135	23.70
DSP	1	240	0.42
IO	22	210	10.48
BUFG	4	32	12.50
PLL	1	6	16.97

Użycie dodatkowych dyrektyw zmniejszyło zapotrzebowanie zasobów. Dlatego w finalnej wersji zostały użyte przedstawione dyrektywy.

Rysunek 31 przedstawia rozmieszczenie wykorzystywanych elementów na płytce.



Rysunek 31: Rozmieszczenie wykorzystywanych elementów

6 Podsumowanie

6.1 Podsumowanie

W ramach realizacji pracy został opracowany system na chipie wykorzystujący rdzeń Ibex RISC-V. Mikroprocesor ten spełnia wymagania specyfikacji ISA mikroprocesora RISC-V opracowanej przez . Do systemu została dodana pamięć RAM, z możliwością wyboru między pamięcią dwu-portową a jedno-portową. Ponadto zostały zaimplementowane peryferia takie jak:

- *GPIO* - moduł wejść/wyjść GPIO z czterema wyjściami i wejściami,
- *UART* - interfejs *UART*,
- *SPI* - interfejs *SPI*,
- *I2C* - interfejs *I2C*.

Integralną część systemu stanowi magistrala *Wishbone*, która odpowiada za komunikację między rdzeniem a peryferiami i pamięcią. W celu poprawnego działania tej magistrali stworzone zostały interfejsy do urządzeń typu *slave* i *master*, oraz moduł sterowania magistralą. W celu poprawnej komunikacji, pamięć RAM, jak i peryferia, zostały przypisane do poszczególnych adresów. Moduł, *wishbone_sharedbus* organizuje transfer danych magistrali.
W celu sprawdzenia działania systemu na chipie została przeprowadzona weryfikacja rdzenia, pamięci RAM i peryferiów. Wykorzystana została biblioteka UVM 1.2, która pozwala na tworzenie zaawansowanego środowiska testowego. Rdzeń został, przetestowany poprzez wykonywanie skomplikowanych programów assemblerowych. Programy te zostały utworzone dzięki RISCV-DV, który pozwala na generowanie programów sprawdzających daną funkcjonalność rdzenia. Wynik symulacji RTL został, porównany za „złotym" wzorem przygotowanym poprzez symulator *ISS* OVPSim. Wynik pozytywny uzyskały wszystkie testy. Pamięć RAM, jak i peryferia, również zostały poddane weryfikacji. Wszystkie testy przeszły z wynikiem pozytywnym.
Projekt jest w pełni syntezowalny. Eksperymenty i badania zostały przeprowadzone w środowisku *Vivado 2019.2*. System został zaimplementowany w układzie FPGA płyty prototypowej Nexys4DDR. Weryfikacja praktyczna została przeprowadzona z wykorzystaniem prostego programu napisanego w języku asembler.
Architektura RISC-V jest wciąż rozwijana. Rdzeń mikroprocesora warto rozbudować między innymi o interfejs *JTAG* oraz moduł *RISCV-DBG*. Warto również rozważyć zastąpienie magistrali *Wishbone* magistralą AXI z uwagi na jej coraz szersze zastosowanie. Pomimo pewnych pomysłów na dalszy rozwój systemu, należy podkreślić, że cel stawiany pracy został osiągnięty, a zakres w pełni zrealizowany.

7 Bibliografia

Literatura

[1] Karl Michael Popp. *Best Practices for commercial use of open source software*. Books On Demand 2015.

[2] [Online] RISC-V, ISA Specification, https://riscv.org/specifications/ [dostęp 10 sierpień 2020]

[3] Andrew Waterman, Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA version 2.2*. University of California, Berkeley. EECS-2016-118. Retrieved 7 May 2017.

[4] Kung Linliu. *DRAM-Dynamic Random Access Memory: The memory of computer, smart phone and notebook PC*. Independently Published 2018.

[5] [Online] Freescale Semiconductor, Inc., SPI Block Guide https://bit.ly/3iF2u3l [dostęp 10 sierpień 2020]

[6] Dominique Paret, Carl Fenger. *The I2C Bus: From Theory to Practice*. Wiley 1997

[7] Adam Osborne. *An Introduction to Microcomputers Volume 1: Basic Concepts*. McGraw-Hill; 2nd edition 1980.

[8] [Online] Sasang Balachandran, General PurposeInput/Output (GPIO), https://bit.ly/2SylFkq [dostęp 10 sierpień 2020]

[9] [Online] OpenCores, Wishbone B4, https://bit.ly/37vmPGi [dostęp 10 sierpień 2020]

[10] [Online] Gisselquist Technology, Building Formal Assumptions to Describe Wishbone Behaviour, https://bit.ly/35wlJYl [dostęp 10 sierpień 2020]

[11] [Online] lowRISC, Ibex: An embedded 32 bit RISC-V CPU core, https://bit.ly/35wmPDa [dostęp 10 sierpień 2020]

[12] [Online] Techcrunch, Google launches OpenTitan, https://tcrn.ch/2PIjSrN [dostęp 10 sierpień 2020]

[13] [Online] RISC-V, Riscv-gnu-toolchain, https://bit.ly/3dTBcW2 [dostęp 10 sierpień 2020]

[14] [Online] Google, Riscv-dv documentation, https://bit.ly/33Vh8zI [dostęp 10 sierpień 2020]

[15] [Online] DIGILENT, Nexys4DDR™ FPGA Board Reference Manual, https://bit.ly/3mhBhG8 [dostęp 10 sierpień 2020]

[16] [Online] IEEE, IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language https://bit.ly/34pxnVr [dostęp 10 sierpień 2020]

[17] [Online] Serial Peripheral Interface, https://bit.ly/37BbIvm [dostęp 10 sierpień 2020]

[18] [Online] UART, https://bit.ly/3onZY5g [dostęp 10 sierpień 2020]

[19] [Online] Accellera, Universal Verification Methodology (UVM) 1.2 User's Guide, https://bit.ly/2FZRa4w [dostęp 10 sierpień 2020]