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, wrzesień 2020

Od autora

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
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
PLL - phase-locked loop
Pmod - Peripheral Module interface
ALU - Arithmetic Logic Unit

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. W celu weryfikacji działania układów, są wykorzystywane symulatory języków opisu sprzętu takie jak Riviera-PRO.

SoC powinien składać się z mikroprocesora, mikrokontrolera lub rdzenia DSP. Każdy mikroprocesor posiada 'Model programowy procesora' (ang Instruction Set Architecture, ISA). ISA definiuje jak mikroprocesor powinien działać, jego listę rozkazów, typ danych, tryby adresowania, rejestry dostępne dla programisty, zasady obsługi przerwań i wyjątków. Przykładowe komercyjne ISA: ARM, MIPS, Power ISA. Jest również otwarty model programowy procesora, który jest oparty o zasady RISC, jest nim RISC-V. Otwarta standard ISA oznacza, że dostęp nie jest limitowany prawnie, finansowo lub tajemnicą handlową firmy.

Przykładem mikroprocesora wykorzystującego ISA RISC-V jest Ibex. Jest on tworzony przez lowRISC, wywodzącego się z Uniwersytety w Cambridge. Mikroprocesor ten jest 32bit, składa się z 2-stage pipileine i został zaimplementowany na bazie RV32IMC.

1.2 Cel i zakres pracy

Celem pracy jest implementacja SoC na podstawie mikroprocesora Ibex RISC-V. Mikroprocesor należy przystosować do implantacji na płytce FPGA NEXYS4DDR oraz dodać odpowiednie peryferia. Następnie przeprowadzić weryfikację zaimplementowanego systemu na chipie poprzez przeprowadzenie symulacji korzystając z biblioteki UVM 1.2 i testów RISCV compliance. Weryfikacji zostanie poddany cały SoC jak i poszczególne peryferia.
Zakres pracy obejmuje:

- Implementacje mikroprocesora IBEX
- Implementacje peryferii:
  1. RAM
  2. SPI
  3. I2C
  4. UART
  5. GPIO
  6. Timer
- Kompilacja toolchaina i przystosowanie go dla SoC
- Przeprowadzenie weryfikacji
- Porównanie wyników dla poszczególnych architektur i pamięci
- Podsumowanie wyników pracy

1.3 Zarys pracy

Praca składa się z 6 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, schematy blokowe i FSM. Czwarty rozdział przedstawia weryfikacje, opisuje przebiegające fazy biblioteki UVM 1.2 oraz jej wyniki. Następnie pokazuje symulację przeprowadzaną z instrukcjami wygenerowanymi przez RISCV-DV, Wyniki tej symulacji zostaną porównane z ISS Ovpsim i Spike. W piątym rozdziale zostaną porównane wyniki symulacji oraz syntezy architektury Von Neumanna z architekturą Harvardzką, pamięć RAM jedno-portowa z pamięcią RAM dwu-portową. Ostatni rozdział to podsumowanie oraz propozycję dalszego rozwoju projektu.

2 Część teoretyczna

2.1 RISC V

2.1.1 Instruction set architecture ISA

RISC-V to otwarta ISA bazująca na architekturze RISC. Oznacza to, że licensja jest typu Open-source, która 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 przywilejów
3. Debug Specification - specyfikacja debugowania

Podstawowe cechy architektury RISC to:

- Zredukowana lista rozkazów, jest ich kilkadziesiąt
- Przepustowość procesora zbliżona do jednej instrukcji na cylk
- Zredukowana tryby adresowania, kody rozkazów są prostsze
- Powiększenie liczby rejestrów
- Minimalizacja komunikacja 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. Instrukcje są 32-bit. Tabela 3 przedstawia formaty tych instrukcji. Korzystają one z sześciu formatów:

- 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 12bitowej stałej ze znakiem, wynik jest zapisywany w rejestrze *rd*.
- Upper immediate (U) - format wykorzystywany dla dwóch instrukcji: *LUI*, *AUIPC*. Służy do przypisywania liczb 20bitowych 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*.

Tabela 1: ISA base and extensions[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

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 2: Rejestry RISC-V[3]

Nazwa rejestry	Nazwa symboliczna	Opis	Właściciel
x0	Zero	zawsze zero	
x1	ra	adres powrotu	wywołujący
x2	sp	wskaźnik stosu	wołany (callee<?>)
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	wołany
x9	s1	zapisany rejestr	wołany
x10-11	a0-1	argument funkcji / wartość zwracana	wywołujący
x12-17	a-2-7	argument funkcji	wołany
x18-27	s2-11	zapisane rejestry	wołany
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	wołany
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 12bitowej *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 imm=+8	00010 rs1=2	010 LW	10000 rd=16	0000011 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 imm[11:0]=+8	10000 rs2=16	00010 rs1=2	010 SW	01000	0100011 STORE

Podobnie jak w instrukcjach *load* func3 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*. Zaś 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*. Przykład działania rozkazu *add*, wykorzystuje on format instrukcji *R*:

Tabela 3: 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						

`add x6, x7, x8`

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

Pierwszy argument trafił to rejestru *rd*, kolejny do rejestru *rs1* ostatni do rejestru *rs2*. *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:

`addi x6, x0, 50`

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

Func3 jest wykorzystywana w celu dekodowania instrukcji. Rozkazu przesunięcia bitowego wykorzystują pięć najmłodszych 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 20bitów. Dzięki rozkazowi *JALR* i *AUIPC* można stworzyć skok o szerokości 32bitów. Rozkaz *AUIPC* zapisuje do rejestru aktualną wartość PC, a rozkaz *JALR*, zamienia dwanaście najmłodszych bitów na wartość przekazanego argumentu. Przykładowe programy z użyciem instrukcji skoków bezwarunkowych.

```
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*.
Kolejną rodzajem są skoki warunkowe, jest ich sześć i są zakodowane w formacie *SB*:

- BEQ - gdy zapisane liczby w rejestrach są równe wykonuje skok
- BNE - gdy zapisane liczby w rejestrach są różne wykonuje skok

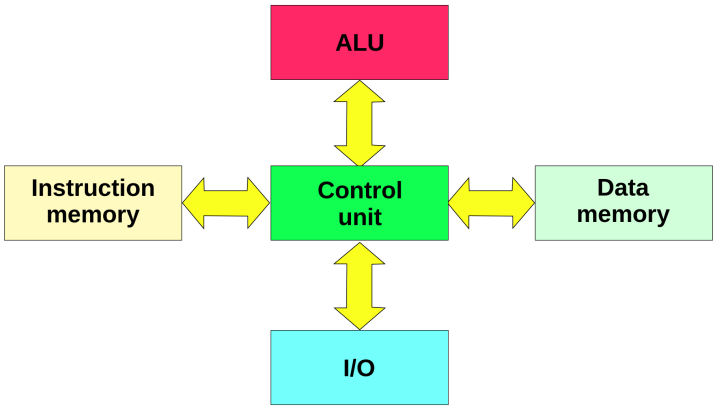
- BLT - gdy liczba z rejestru *rs1* jest większa wykonuje skok
- BLTU - gdy liczba z rejestru *rs1* jest większa bądź równa wykonuje skok
- BHE - gdy liczba z rejestru *rs2* jest większa wykonuje skok
- BGEU - gdy liczba z rejestru *rs2* jest większa wykonuje skok

2.2 System on Chip

2.2.1 Architektura Harvardzka

Architektura Harvardzka to rodzaj architektury komputera. Posiada ona dwie oddzielne szyny dla danych i rozkazów. Można w tym samym czasie pobierać argument wykonywanej funkcji i pobierać następnego rozkazu. Zwiększa ta szybkość pracy. Rysunek 1 przedstawia schemat blokowy tej architektury.

Rysunek 1: Schemat blokowy architektury Harvardzkiej



2.2.2 Peryferia

W projekcie zostały dodane 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 jeden dane/adres port, więc może być czytana lub zapisywana w jednej chwili czasu. Pamięć dwu-portowa zawiera dwa dane/adres porty, więc może być czytana i zapisywana w jednej chwili czasu.[4]

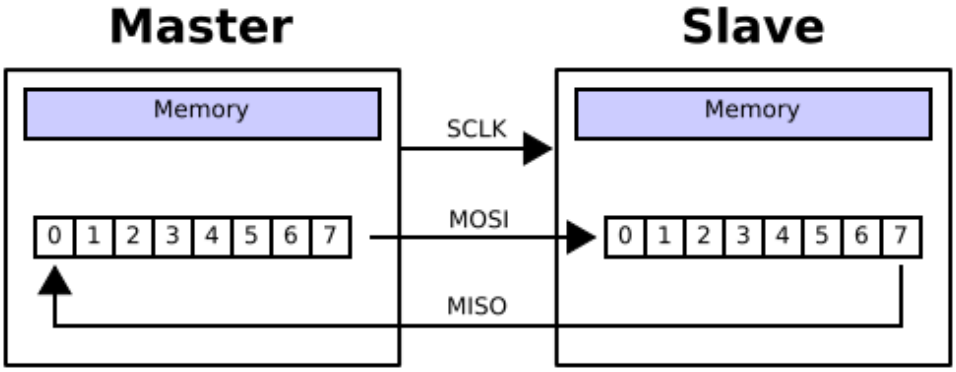
2. SPI - interfejs służący do transmisji, głównie używany w systemach wbudowanych. Wykorzystuje się tryb *master-slave*, dzięki temu jest zapewniona komunikacja full-duplex. Interfejs ten posiada następujące porty:

   - $SCLK$ - 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 *SS* w celu wybrania odpowiedniego *Slave'a*. *Master* wysyła bit poprzez *MOSI* i *slave'a* odczytuje go i wysyła bit poprzez *MISO*. Rysunek 2 obrazuję przebieg transmisji[5].

Rysunek 2: Przykład transmisji SPI



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

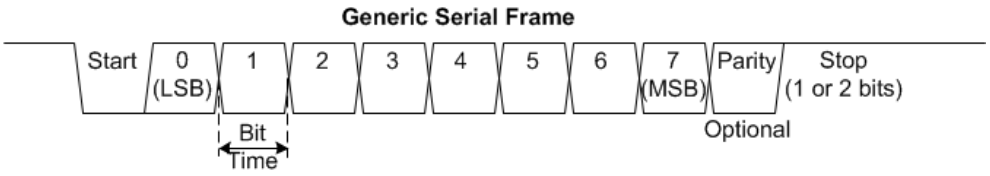
   I2C może pracować z wieloma *slave'ami* i *masterami*. Rysunek **??** przedstawia wygląd ramki I2C. By rozpoacząć transmisje *master* wysyła sygnał startowy. By to uzyskać sygnał na linii *SDA* zmienia się z wysokiego na niski przed zmianą sygnały z wysokiego na niski na linii *SCL*. Następnie jest przesyłany 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*. Rysunek **??** przedstawia przykładowy przebieg transmisji.[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 3 przedstawia ramkę UARTu. Bit parzystości jest opcjonalny i służy jako bit kontrolny.[7]

Rysunek 3: Ramka UART



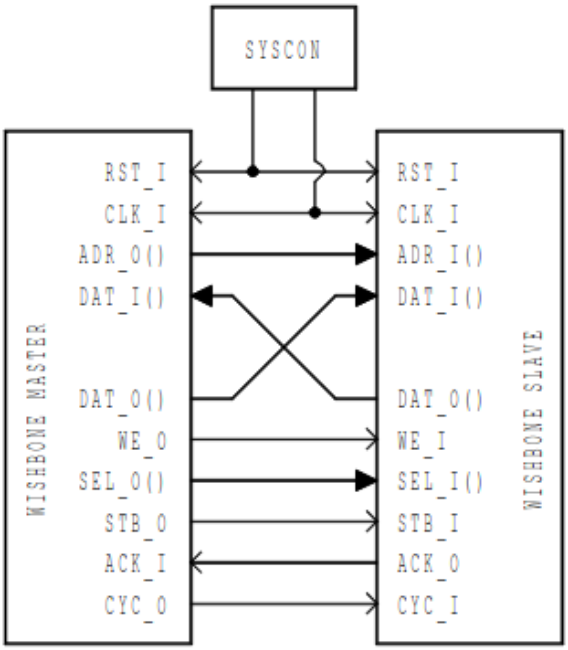
5. GPIO - wyprowadzenia służące do komunikacja między mikroprocesorem a peryferiami[8]

6. Timer

2.2.3 Wishbone

Wishobone to opensource magistrala służąca do łączenia ze sobą wielu IP w systemie *master/slave*. Rysunek 4 przedstawia połączenia w tym interfejsie.

Rysunek 4: Wishbone master/slave interfejs[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 asercji 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 asercji.
- Wszystkie interfejsy *WISHBONE* muszą być przygotowane na reakcję na *RST_I* w każdym momencie.
- *RST_I* może pozostać w stanie asercji dłużej niż jeden cykl zegarowy.

Porty używane przez ten interfejs[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*

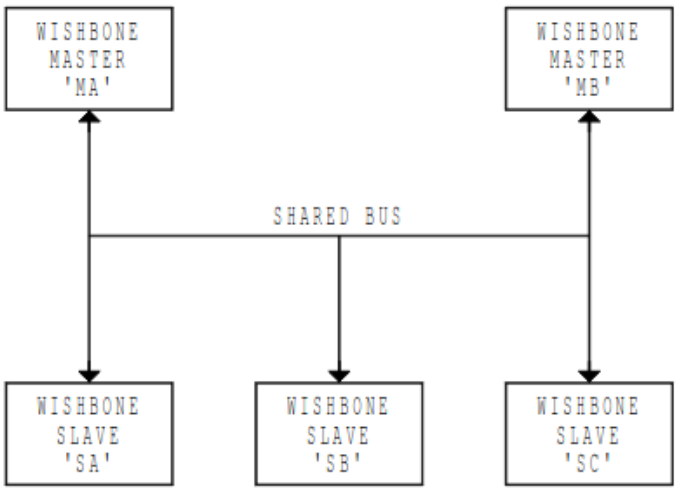
- *DAT_I/O* - linia danych
- *WE_O/I* - pozwolenie na zapis, wyjście z *master*, wejście do *slave*.
- *SEL_O/I* - selekcja bajtu, wyjście z *master*, wejście do *slave*.
- *STB_O/I* - potwierdzenie nadania danych przez *mastera*, wyjście z *master*, wejście do *slave*.
- *ACK_I/O* - potwierdzenie przyjęcia 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 podpięta do tych samych *masterów*. Rysunek 5 przedstawia przykład tej topologii.

Rysunek 5: Wishbone shared bus interconnection[9]



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

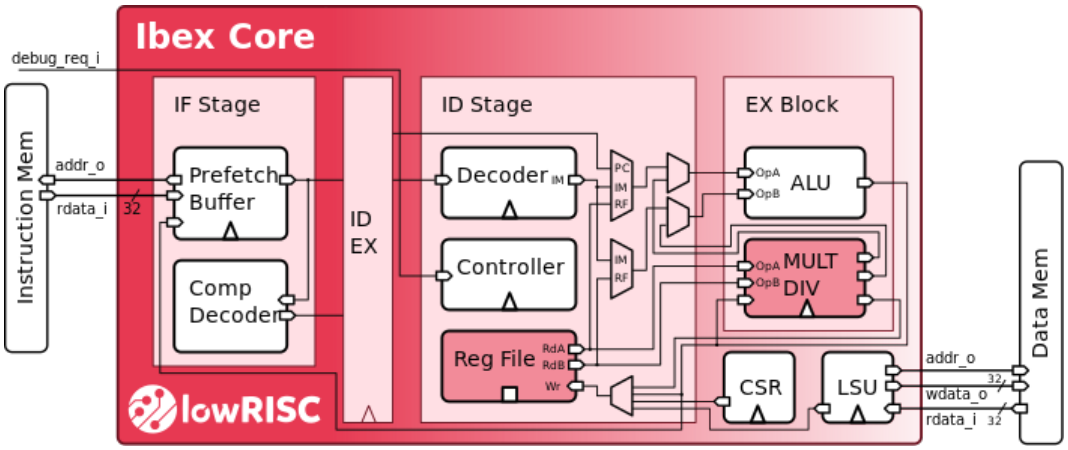
2.3 Ibex

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]. Mikroprocesor ma szeroko rozwiniętą weryfikacje, wykorzystuje on między innymi generator rozkazów *RISCV-DV*. Jest on również częścią projektu *OpenTitan*, jest to *RoT*, wspierany między innymi przez *Google*[12]. Rysunek 6 przedstawia schemat blokowy mikroprocesora *Ibex*[11].

Rysunek 6: Schemat blokowy mikroprocesora



2.4 Kompilator

2.4.1 Budowanie toolchaina

Toolchain 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 64bit, lecz po podaniu odpowiednich argumentów podczas kompilacji programu wspiera również architektury 32bit.

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 testbenchu i konfiguracji testcase'u

2. uvm_component - wszystkie komponenty testbenchu takie jak *scoreboards, monitor, driver* pochodzą z tej klasy.

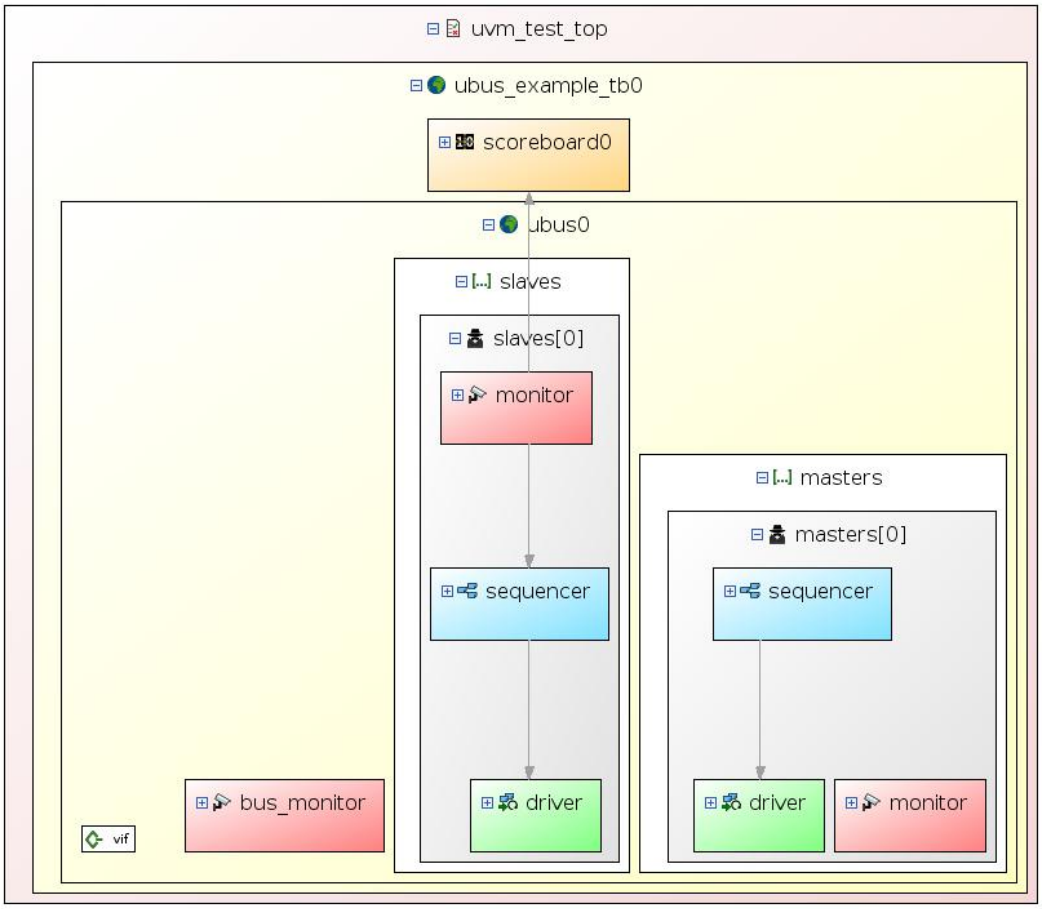
3. uvm_sequence - jest klasą bazową wszystkich sekwencji zawartych w testbenchu

UVM test składa się z następujących elementów:

- UVM test - jest odpowiedzialny za konfigurację testbenchu, rozpoczęcie symulacji poprzez inicjalizację sekwencji, stworzenie wszystkich komponentów, której znajdują się poniżej w hierarchii na przykład: uvm_env.
- UVM env - grupuje agentów i scoreborady
- 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 pa kiety
- 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 7 przedstawia przykładowy graf UVM testu

Rysunek 7: Przykładowy graf UVM



2.5.2 RISCV DV

RISCV-DV - narzędzie/IP 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/IP należy posiadać symulator wspierający UVM, na przykład: Riviera-PRO[14].

2.6 FPGA

SoC będzie działać 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]

2.7.1 Xilinx Vivado Design Suite

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.

2.7.2 Aldec Riviera-PRO

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 SoCa.

3 Implementacja

3.1 System na czipie

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:

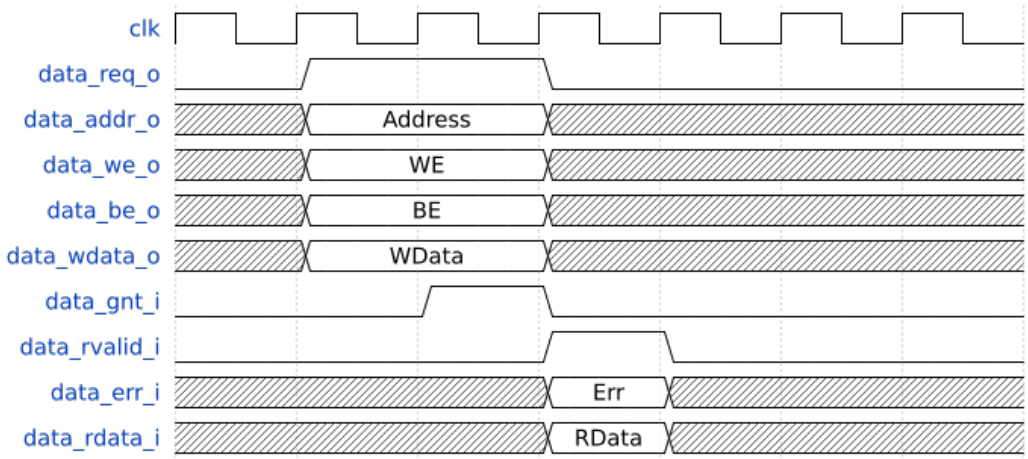
- *Clock gating* - moduł zawierający bufor sygnału zegarowego.
- *Instruction Fetch* - odpowiedzialny za pobieranie instrukcji. W jednym cyklu dostarcza instrukcję do *ibex_id_stage* o ile pamięć jest zdolna do wysłania jednej instrukcji na cykl. Instrukcje są przechwytywane do *ibex_prefetch_buffer* 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.
- *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 8 przedstawia przykład komunikacji między modułem *LSU* a pamięcią.

Rysunek 8: Komunikacja *LSU* z pamięcią



- *Register File* - zawiera trzydzieści jeden lub piętnaście 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.

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*.

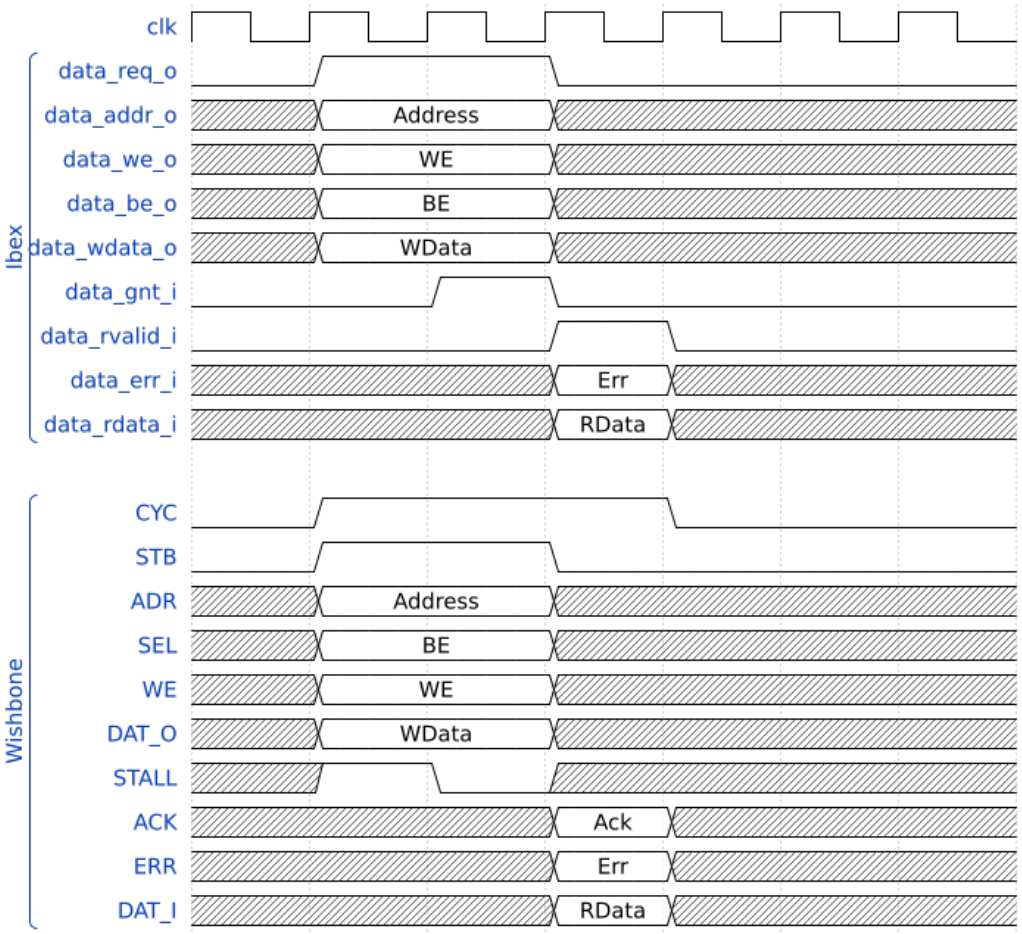
3.3 Wishbone

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 czipie 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 9 przedstawia porównanie komunikacji magistrali *Wishbone* z komunikacją *LSU* z pamięcią.

Rysunek 9: 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 32bitowa.
- *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 32bitowa.

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

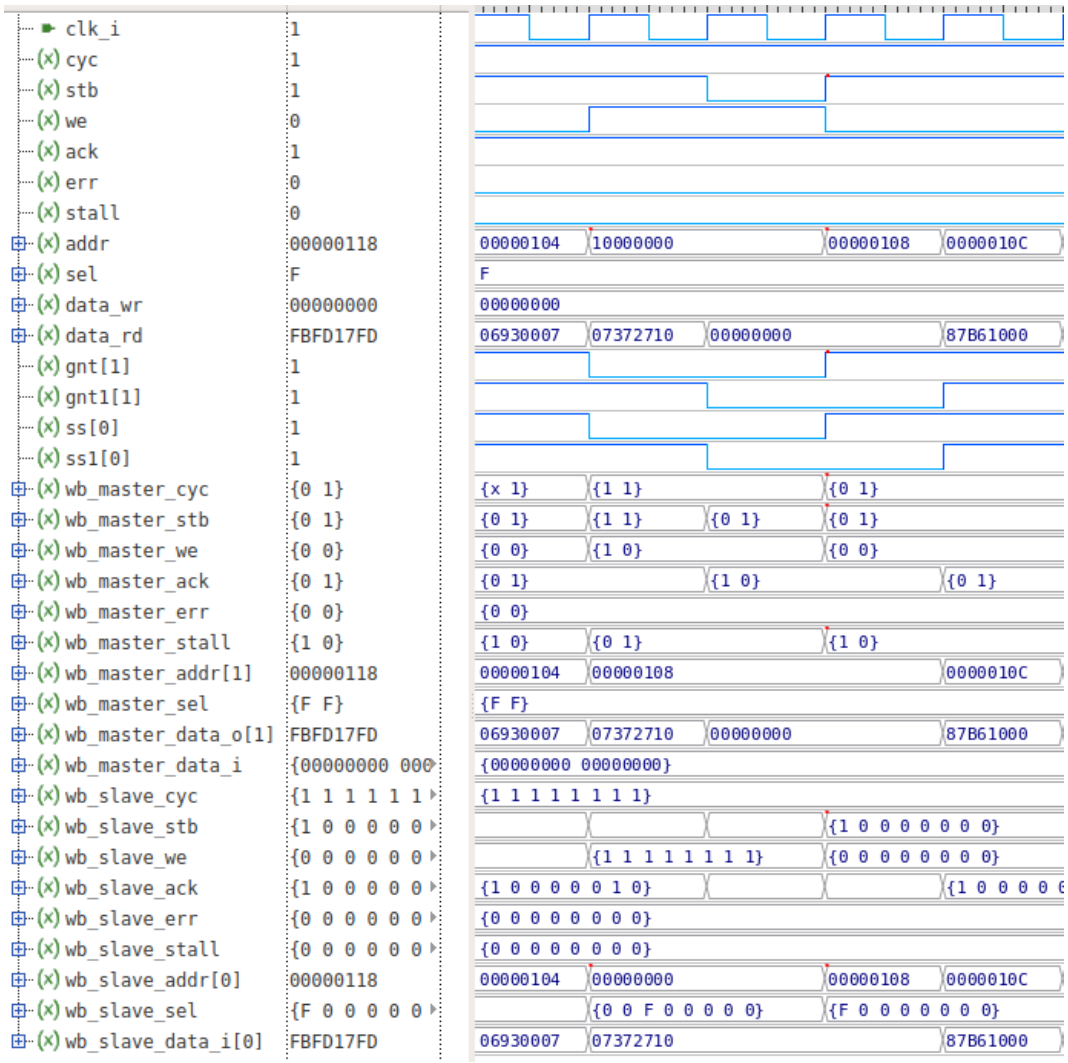
przypisania do tablicy *slave_select_1* w celu zapewnienia potokowości. Blok proceduralny *always_comb* zapewnia komunikacje między *masterem* a urządzeniem *slave*. Listing 7 przedstawia ten zabieg.

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 10 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 10: 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 32bitów, 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 4bitowy, 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 11 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 12 przedstawia sytuacje zapisu do pamięci. Na linii adresowej pojawia się *00*, sygnał *valid_i* jest w stanie wysokim. Oznacza to, że w następnym cyklu ze-

garowym 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.

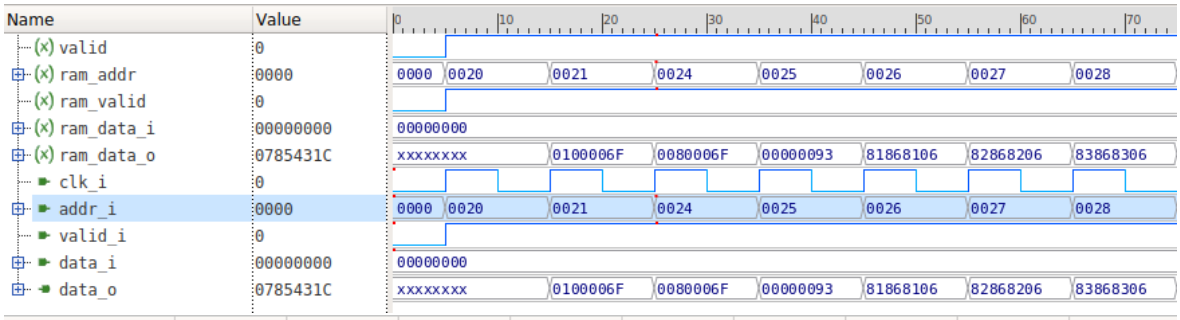
Listing 10: Instancja pamięci instrukcji

```
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

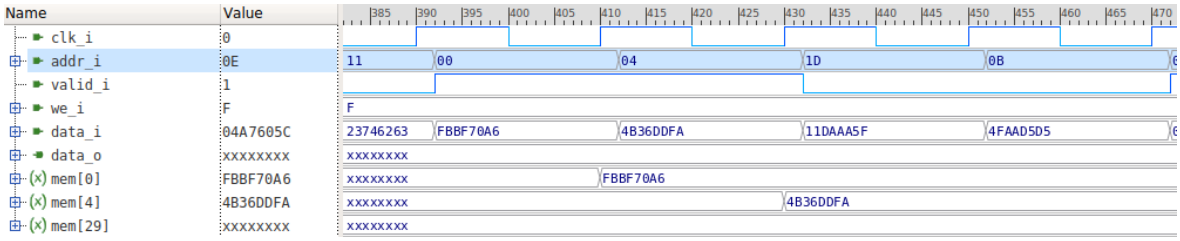
```
p1_ram_data#(
      .SIZE(SIZE),
      .AW(ADDR_WIDTH))
ram(
      .clk_i(wb.clk_i),
      .addr_i(ram_addr),
      .valid_i(ram_valid),
      .we_i(ram_we),
      .data_i(ram_data_i),
      .data_o(ram_data_o));
```

Rysunek 11: Przebiegi sygnałów pamięci *RAM* oraz jej dane podczas odczytu.



	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* 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 32bitów 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 4bitowy więc by w pełni zapełnić przestrzeń zastosowano konkatenację. Sygnały *wb.err* i *wb.stall* zostały przypisane do zera. Projekt sytuacji by te sygnały mogą się pojawić. Wybór kierunku 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),
        .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 8bitowy. 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*

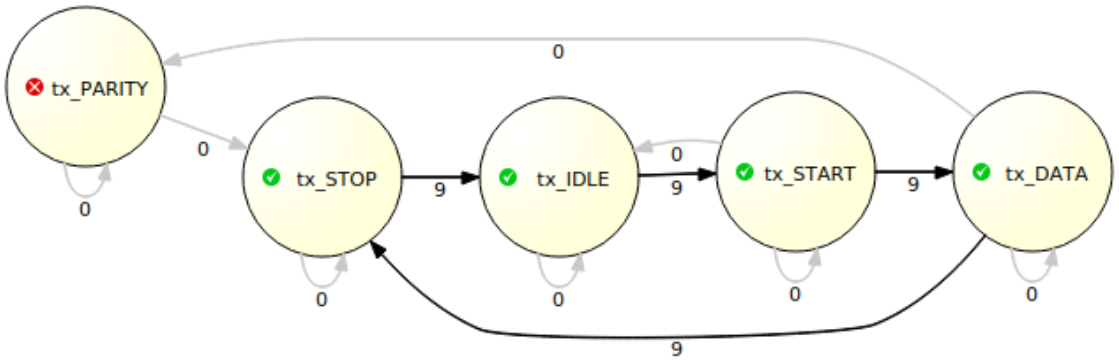
W module głównym znajdują się instancje transmitera i odbiornika *UART*. Poprzez parametryzowanie go można określić następujące cechy:

- *clk_freq* - określa częstotliwość zegara systemu na czipie,
- *baud_rate* - określa szybkość transmisji, w projekcie jego wartość jest równa 19200bps,
- *data_bits* - określa szerokość wektora danych. W projekcie użyto 8bitowej 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 Transmiter *UART*

Transmiter został zaimplementowany w oparciu o graf FSM przedstawiony na rysunku 13

Rysunek 13: Graf *FSM* odbiornika *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*

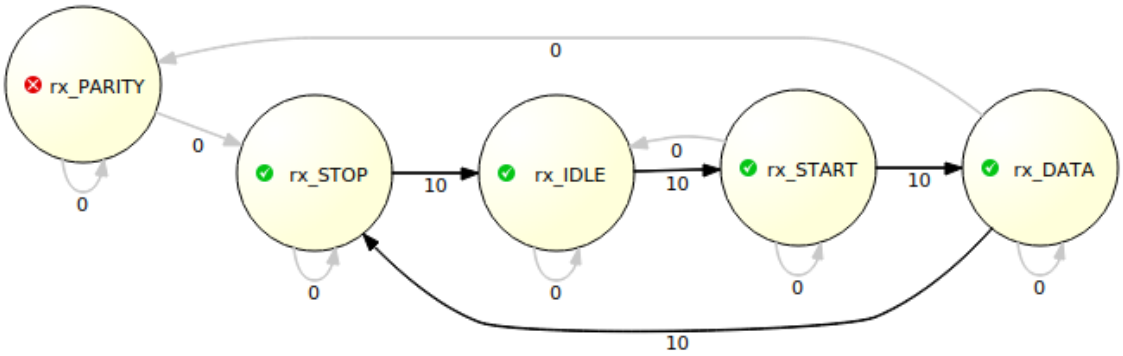
```
tx_DATA: begin
        tx_out_next = tx_data_reg[index_bit_reg];
        if(clk_div_reg < clock_divide[$clog2(clock_divide):0]-1'b1) begin
                clk_div_next = clk_div_reg + 1'b1;
                tx_NEXT = tx_DATA;
        end
        else begin
                clk_div_next = 0;
                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 Odbiornik *UART*

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

Rysunek 14: 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 I2C

3.7.1 master

opis i2c master

3.7.2 slave

opis i2c slave

3.8 SPI

3.8.1 master

opis spi master

3.8.2 slave

opis spi slave

3.9 Timer

opis timera

4 Weryfikacja

4.1 RISCV DV

4.1.1 riscv arithmetic basic test

krotko o tym tescie i wynik z simstatus jak rowniez fragment logu komparacji z spike-/ovpsim

4.1.2 riscv rand instr test

krotko o tym tescie i wynik z simstatus jak rowniez fragment logu komparacji z spike-/ovpsim

4.1.3 riscv illegal instr test

krotko o tym tescie i wynik z simstatus jak rowniez fragment logu komparacji z spike-/ovpsim

4.2 ibex core

4.3 pamiec ram

4.4 gpio

4.5 uart

4.6 spi

4.7 i2c

5 Benchmarki

pamiec 1p ram vs 2p ram

6 Uruchomienie przykładowego programu

7 Podsumowanie i wnioski

7.1 dalszy rozwoj

text

8 Bibliografia

Literatura

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

[2] 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] https://www.nxp.com/files-static/microcontrollers/doc/ref_manual/S12SPIV4.pdf [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] https://bit.ly/2DJ1Y5F [dostęp 10 sierpień 2020]

[9] http://cdn.opencores.org/downloads/wbspec_b4.pdf [dostęp 10 sierpień 2020]

[10] http://zipcpu.com/zipcpu/2017/11/07/wb-formal.html [dostęp 10 sierpień 2020]

[11] https://ibex-core.readthedocs.io/en/latest/index.html [dostęp 10 sierpień 2020]

[12] https://tcrn.ch/2PIjSrN [dostęp 10 sierpień 2020]

[13] https://github.com/riscv/riscv-gnu-toolchain [dostęp 10 sierpień 2020]

[14] https://bit.ly/33Vh8zI [dostęp 10 sierpień 2020]

[15] https://dl.btc.pl/kamami_wa/digilent_nexys4-ddr_1.pdf [dostęp 10 sierpień 2020]

[16] https://standards.ieee.org/standard/1800-2017.html [dostęp 10 sierpień 2020]