ISTANBUL TECHNICAL UNIVERSITY
COMPUTER AND INFORMATICS FACULTY

A PROCESSOR DESIGN ON FPGA

Graduation Project

Tuğrul Yatağan
040100117

Department : Computer Engineering

Advisor : Assoc. Prof. Dr. Mustafa Ersel Kamaşak

May 2015

ISTANBUL TECHNICAL UNIVERSITY
COMPUTER AND INFORMATICS FACULTY

A PROCESSOR DESIGN ON FPGA

Graduation Project

Tuğrul Yatağan
040100117

Department : Computer Engineering

Advisor : Assoc. Prof. Dr. Mustafa Ersel Kamaşak

May 2015

Declaration of Originality

I declare,

1. In this study, all citations made from other sources, are clearly indicated by reference to relevant sources,
2. The other parts except citations, especially the theoretical studies and the software which forms the main subject of the project,

are prepared by me.

Istanbul, 29 May 2015

Tuğrul Yatağan

A PROCESSOR DESIGN ON FPGA

(SUMMARY)

Using advantage of field programmable gate arrays (FPGA) dynamic digital design capability, every processor architecture and organization can be simulated on it. This allows compare different processor architecture design's advantages and disadvantages. Also new architecture designs can be tested on FPGA's before production. In this project a RISC processor is implemented with a newly designed architecture and instruction set. Commercial processor architectures were examined and mixture of their design inspired the new architecture. Learning, design & working mechanism of a processor is main aim of this project. Final result of this project is a soft processor which is able to execute instructions like fully functional standard microprocessor.

In this project, a soft processor is designed and implemented on Xilinx FPGA in Verilog hardware description language. Mixture of ARM Corporation's THUMB and MIPS Corporation's MIPS architectures were used to inspire new architecture. These designs were chosen because these are common RISC architectures on market. The processor is in Harvard architecture category so it has separate instruction and data memory. Also it has fixed length Load/Store instruction set architecture which means only load and store instructions can access to data memory. This leads to single cycle instruction execution capability with easier pipeline design. The classic RISC pipeline architecture is used for pipeline design so processor's pipeline has 5 stages. The processor has 16 Bit CPU, ALU, register and data bus architecture. Address bus for data and instruction memory is 12 bit so maximum addressable instruction and data memory is 4K word and it is equal to 8KB. This restriction is due to the FPGA's usable internal memory size. Processor's peripheral modules have their unique addresses in data memory address space so peripheral units uses memory mapped I/O access system.

Processor has original design instruction set architecture. An assembler is written in Python programming language for this new instruction set. This assembler can transform instructions into binary machine codes and it can load this binary codes into FPGA's instruction and data memory. Thus assembler can program soft processor in FPGA without any external tool.

In this project, soft processor and assembler gives an integrated development environment to processor programmers.

FPGA ÜZERİNDE BİR İŞLEMCİ TASARIMI

(ÖZET)

FPGA'lerin dinamik sayısal tasarım yeteneğinden faydalanılarak bütün işlemci mimarileri ve organizasyonları FPGA'ler üzerinde denenebilir. Bu, farklı işlemci mimari tasarımlarının avantaj ve dezavantajlarının kıyaslanabilmesini sağlar. Ayrıca FPGA'ler yeni işlemci mimarisi tasarımlarının kolaylıkla geliştirilebilmesini ve üretimden önce test edilebilmesini sağlar. Bu projede özel tasarım mimari ve komut setine sahip bir RISC işlemci tasarlanıp, uygulamaya konulmuştur. Bu yeni mimari tasarımı için piyasada bulunan RISC işlemci mimarilerinden ilham alınmıştır. İşlemci tasarımı ve işlemcilerin çalışma mekanizmasının derinlemesine öğrenilmesi bu bitirme projesinin asıl amacıdır. Proje sonunda komut işleyebilen tüm fonksiyonları yerinde FPGA üzerinde çalışabilen bir sanal (yazılımsal) işlemci tasarlanmıştır.

Bu projedeki sanal işlemci Verilog donanım tanımlama dilinde (HDL), Xilinx firması üretimi bir FPGA üzerinde geliştirilmiştir. Tasarım olarak ARM firması tarafından geliştirilen THUMB ve MIPS Teknoloji firması tarafından geliştirilen MIPS mimarilerinden esinlenilmiştir. Bu tasarımların seçilme sebebi piyasada yaygın olarak kullanılan RISC mimarisi tasarımları olmalarıdır. İşlemci Harvard mimari yapısında tasarlanmıştır yani ayrı veri ve komut belleğine sahiptir. Ayrıca sabit uzunlukta Yükle/Kaydet (Load/Store) komut seti mimarisine sahiptir bu yüzden yalnızca Yükle ve Kaydet komutları veri belleğine erişebilir. Bu daha basit bir iş hattı tasarımının yapılmasına ve tek çevrimde komut işlenmesine olanak sağlamaktadır. İş hattı mimarisi için klasik RISC iş hattı mimarisi seçilmiştir yani işlemci 5 aşamalı iş hattına sahiptir. İşlemci 16 Bit genişliğinde CPU, ALU, yazmaç ve veri yolu mimarisine sahiptir. Veri ve komut belleği için adres yolu genişliği 12 bittir. Bu yüzden en fazla adreslenebilir veri ve komut belleği boyutu 4K kelime uzunluğunda yani 8KB boyutundadır. Bu kısıtlamanın sebebi FPGA içerisindeki en fazla kullanılabilir dâhili bellek boyutunun kısıtlı olmasındandır. İşlemcinin çevresel giriş ve çıkış birimleri veri belleği üzerinden adreslenmiştir.

İşlemci özgün bir komut seti mimarisine sahiptir. Bu yeni komut seti için Python programlama dilinde bir komut çevirici (assembler) yazılmıştır. Bu komut çevirici, işlemci komutlarını ikili makine koduna çevirip FPGA üzerindeki işlemcinin komut ve veri belleğine yazabilmektedir. Böylece sanal işlemci sadece komut çevirici üzerinden başka bir araca gerek kalmadan programlanabilmektedir.

Projede sanal işlemci ve komut çevirici ile tam ve bütünleşik bir geliştirme ortamı oluşturulmuştur.

TABLE OF CONTENTS

1. INTRODUCTION

1.1. General View of FPGA and soft processors

FPGA's are configurable logic integrated circuits. They may contain thousands of programmable logic blocks. It is possible to create any logic design on a FPGA. FPGA boards can contain various I/O and memory units. There are several available soft processors on market. These soft processor can be implemented on FPGA devices. Official soft processor for FPGA manufacturer Xilinx Corporation is MicroBlaze [13] and FPGA manufacturer Altera Corporation is NIOS [1]. These two are most common soft processors on market. Xilinx Spartan-6 FPGA is used in this project. MicroBlaze is a 32-bit RISC Harvard architecture, closed code soft processor. There are several other open and commercial soft processor out there in market.

1.2. Work Done and Results

In this project a soft RISC processor is implemented with a newly designed architecture and instruction set on FPGA. Final results of this project are a soft processor which can execute instruction like normal microprocessor and an assembler for this soft processor. Project has started on October 2014 and finished on May 2015. Detailed project plan is described in chapter 2.

Mixture of THUMB and MIPS architectures were used to inspire new architecture so RISC architecture was chosen as main architecture. The processor is in Harvard architecture, it has separate instruction and data memory. The classic RISC pipeline architecture is used for pipeline design so processor's pipeline has 5 stages. [6] The processor has 16 Bit CPU, ALU, register and data bus architecture. Address bus for data and instruction memory is 12 bit. Detailed architectural design is described in chapter 3.1. Also processor has fixed length Load/Store instruction set architecture which means only load and store instructions access to data memory. Detailed instruction set design is described in chapter 3.1.2.

In this project, the soft processor is designed and implemented on Xilinx FPGA in Verilog hardware description language. Detailed hardware and software resources are described in chapter 3.2.

There are mainly three books to be a reference for this project. Also some online books and lecture notes are used for project's research. All sources which are used for research are referenced in chapter 5 according to APA style and cited in the text as square brackets ([#]).

2. PROJECT DECRIPTION AND PLAN

2.1. Project Description

Project consists of three main phases. Research, design and implementation. Detailed research on processor architectures, organizations and instruction sets on market are done for research phase.

A new 16-Bit RISC instruction set architecture is designed in design phase. Also a new processor organization is design which consists connection of ALU, registers, memory, I/O, fetch unit, decode unit, control unit, pipeline control/hazard unit, branch unit and interrupt unit.

All these design are first implemented on Logisim simulation program and after successful simulation run, all these modules are implemented on Xilinx ISim simulation program one by one. After successful simulation results, design implemented for FPGA board. Meanwhile a Python assembler is written for new instruction set for ease of development.

After successful processor work on FPGA, assembler improved for a direct programing tool for FPGA.

2.2. Project Plan

Project has started November 2014 and finished May 2015. My first graduation project advisor was Prof. Dr. Ahmet Coşkun Sönmez. He passed away on February 2015. He was a very cheerful teacher. Above all he was a very good person. All theoretical studies and works were done with him. Assoc. Prof. Dr. Mustafa Ersel Kamaşak became graduation project advisor after Prof. Dr. Ahmet Coşkun Sönmez.

Graduation project has these major steps:

- Research on processor architectures and processor design
- Instructions set design
- Architectural design, interaction of main processor units
- Organization design, connection of all processor units
- Design of processor core units in order;
    - ALU
    - Register file
    - Fetch unit
    - Decode unit
    - Control unit
    - Pipeline control/hazard unit
    - Branch unit
    - I/O and memory unit
    - Interrupt unit
    - Processor core organization

are implemented on;
    - Firstly, Logisim simulation program
    - After successful processor work on Logisim they are implemented on Xilinx ISim simulation program with Verilog
    - Lastly processor implemented on FGPA
- Development of assembler for new instruction set in Python

Schedule of these steps are shown following diagrams.

Research and design schedule:

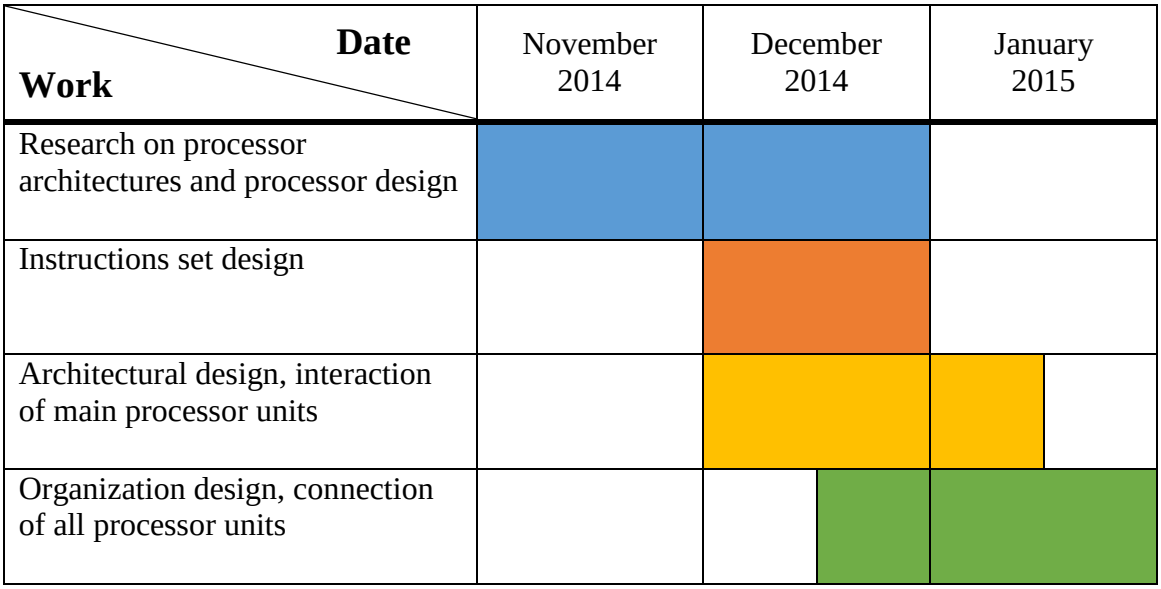

Figure 1: Gantt Diagram of the project's first three months

Implementation schedule:

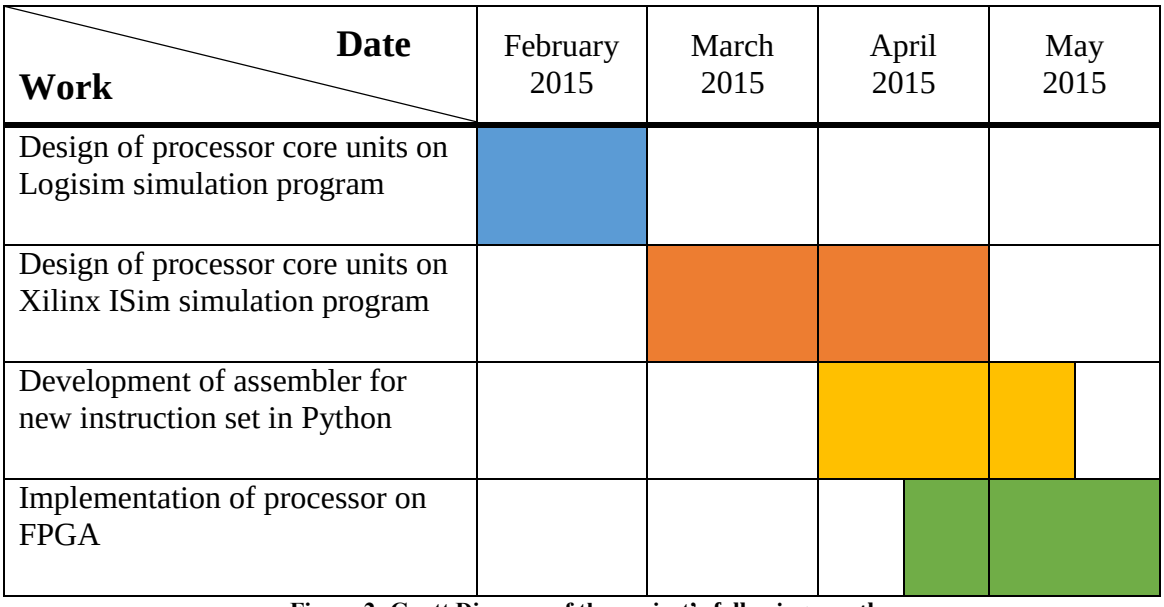

Figure 2: Gantt Diagram of the project's following months

3. THEORETICAL INFORMATION

3.1. Processor Architecture

RISC architecture is reduced instruction set computing which means simplified instruction set and microprocessor architecture. RISC design leads fewer and fixed cycle per instruction. Depth of pipeline defines cycle per instruction. Also instruction lengths are fixed in RISC architecture. Opposite architecture of RISC is complex instruction set computing (CISC). In CISC architecture instruction lengths and cycle per instruction can vary according to instruction. RISC reduced complexity and simplifies design process.

Harvard architecture is a mainframe computer architecture with separate instruction and data memory. Harvard architecture utilize instruction and data memory at the same time so this leads simultaneous instruction fetch and data access. On the other hand Von Neumann architecture is a mainframe architecture with single instruction and data memory. In Von Neumann architecture instruction fetch and data access must be sequential. Von Neumann architecture makes it difficult to design pipeline. Harvard architecture is commonly used with RISC architecture. Harvard architecture leads simple RISC pipeline design [6].

RISC architecture is usually preferred on low power, mobile or embedded systems. Important examples for RISC architecture are ARM Corporation's ARM and THUMB, MIPS Corporation's MIPS, Atmel Corporation's AVR, Microchip Technology's PIC architecture [4].

Load/Store architecture is a sub architecture of RISC systems. Only specific load and store instructions can access to data memory. All other instruction operands are register operands. Load/Store architecture prevents possible stalls in pipeline and simplifies pipeline design. Load and store operations can only be done in specific phase of pipeline so no data access instruction intersect other data access instruction. Load/Store architecture is commonly used with RISC architecture. Important examples for Load/Store architecture are ARM Corporation's ARM and THUMB, MIPS Corporation's MIPS [4].

3.2. Pipeline Architectures

Classic RISC architecture pipeline has 5 stages [6]. Which are;

- Instruction fetch
    - Instruction is fetched from instruction memory to instruction register where program counter points
- Instruction decode
    - Instruction type, operands and offsets are determined
    - Necessary control signals are generated according to instruction type
    - Instruction enters to pipeline
    - Instruction control signals enters to pipeline control registers
    - Operand registers are read end of this stage

- Execute (ALU)
  - Instruction operands enter to ALU
  - Actual computation is done
- Memory access
  - If instruction is memory access instruction, memory access is done
  - If instruction doesn't need memory access, instruction is forwarded to next stage
- Write back
  - Instruction execution results or memory access results are written into register file

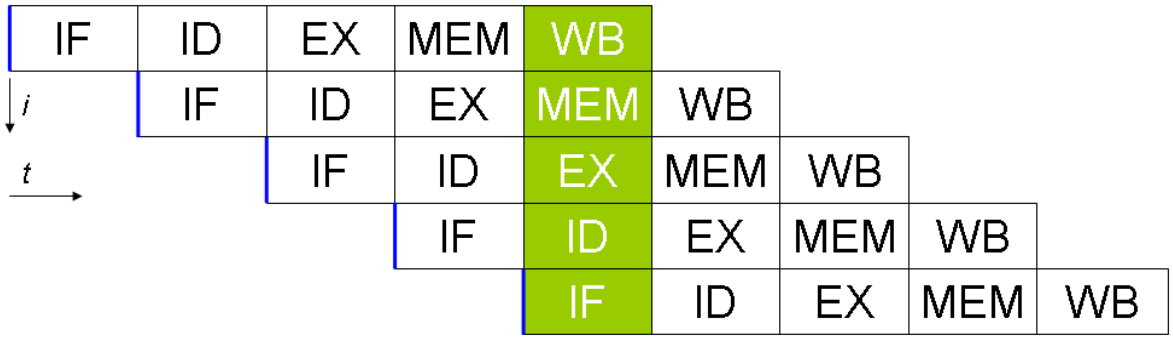

Figure 3: 5 stage classic RISC pipeline timing diagram (Retrieved from [10])

Pipeline hazards are[4]:

- Data hazards
  - Occurs if register type operand needs to be read but previous instruction is not finished
  - Data forwarding solves register operand data hazards
  - Occurs if memory type instruction result need to be read by following instruction.
  - Pipeline stall operation solves memory type data hazard but solution has a one clock cycle pipeline penalty
- Control hazards
  - Occurs if branch is taken
  - Instruction after the branch must not be executed so pipeline flush solves branch hazards but solution has a one clock cycle pipeline penalty

Hazard unit capable of detect data hazards and branch hazards. If a hazard detected, than hazard unit can create forward signal for data forwarding, it can create stall signal for wait necessary operation to finish, or it can create flush signal to flush wrong execution.

3.2.1. Data Forwarding

If register type operand needs to be fetched but previous instruction is not finished then a data hazard occurs. Forwarding data across the pipeline will solve this hazard. Otherwise invalid old value of register would be fetched.

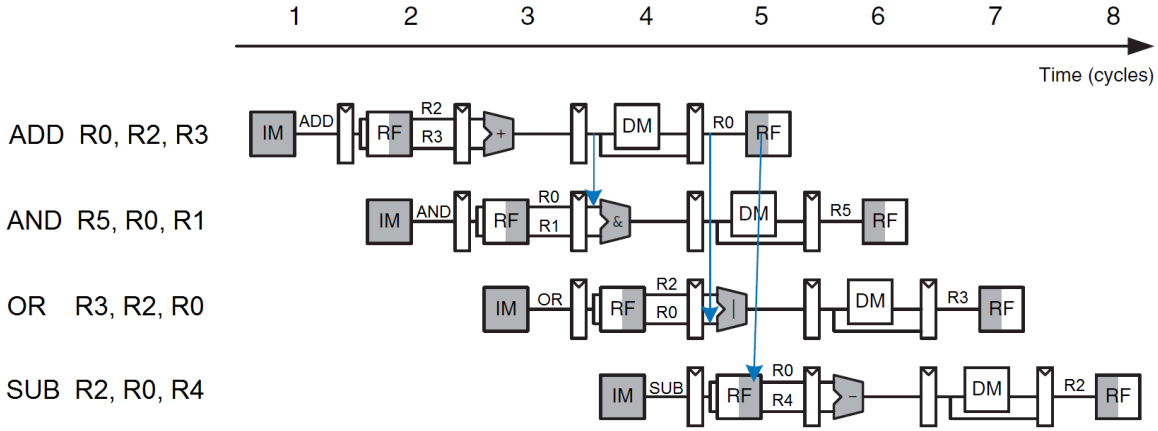

Figure 4: Data forwarding timing diagram (Retrieved from [5] and modified)

Data forwarding solves register operand data hazards.

3.2.2. Pipeline Stall

If memory type operand needs to be read but memory read instruction is not finished then a data hazard occurs. Register fetch is in stage two and memory read is in stage four. It is impossible to solve this hazard without waiting. Instruction which reads previous memory type instruction's result needs to be stall. Otherwise invalid old value of register would be fetched.

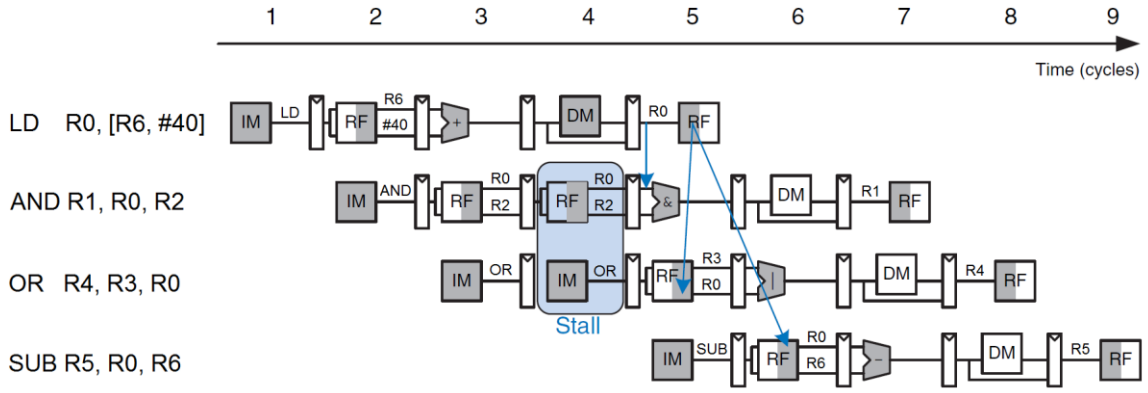

Figure 5: Pipeline stall timing diagram (Retrieved from [5] and modified)

Pipeline stall operation solves memory type data hazard but solution has a one clock cycle pipeline penalty

3.2.3. Pipeline Flush

Branch decision is made at decode stage. If branch instruction decides to take branch, following instruction needs to be removed from pipeline because following instructions already entered to pipeline when branch decision is made. Otherwise undesired instruction execution occurs.

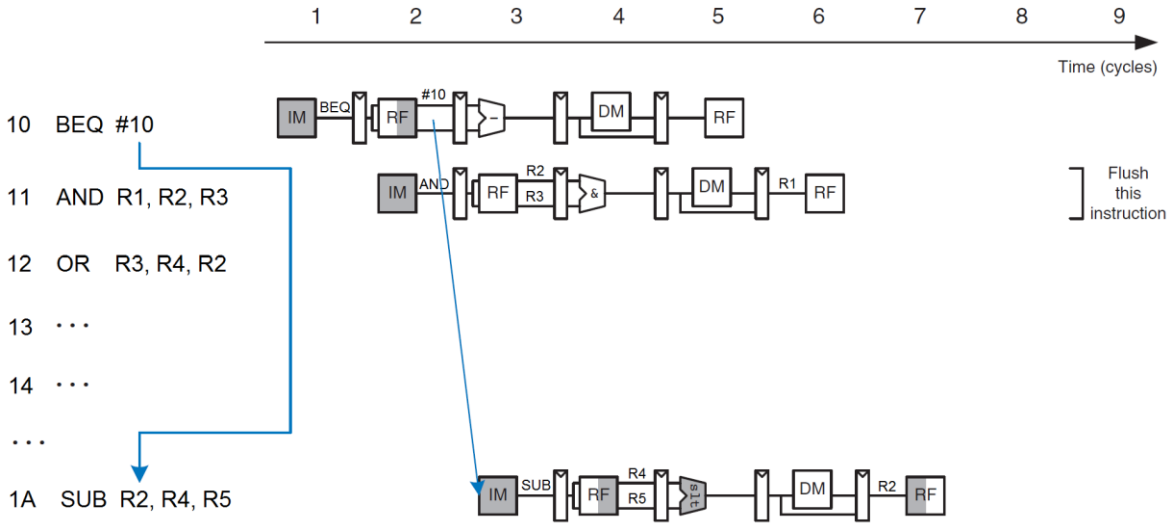

Figure 6: Pipeline flush timing diagram (Retrieved from [5] and modified)

Pipeline flush solves branch hazards but solution has a one clock cycle pipeline penalty

3.3. Instruction Set Architecture

Instruction set defines all the instructions can be executed in a processor. Each instruction set is a unique for a specific processor. Instruction set design directly effects processor micro architecture and processor organization, and vice versa.

RISC architecture instructions can be categorized by operations as follow [4]:

- Register transfer operations
- Memory transfer operations
- Program flow control operations
  - Branch
  - Subroutines call and return
  - Interrupt
- Arithmetic and logic operations

Instruction set design depends to multiple factor like usage area, performance requirement, cost etc. Mainly requirements determines instruction set architecture, there is no perfect set for all requirements. But an instruction set should include some properties.

Expected properties of an instruction set are [9]:

- Regularity
  - Expected instruction opcodes and operands should be available
  - Example; if add with carry is available then subtract with carry should also be available
- Orthogonality
  - Choice of addressing mode should be independent from the choice of instruction
  - Example; if add with register offset (register addressing mode) is available then subtract with register offset should also be available
- Completeness
  - Any operation that are needed most of the times should be available in instruction set
  - Some operations may not be provided in an instruction set due to decrease complexity, but desired operation should be executed with a combination of provided instructions in a reasonable amount of time

3.4. FPGA

A typical FPGA consists of thousands of logic cells. Each of these cell can be used as logic block and memory. FPGA uses lookup tables to generate necessary logic blocks. All input and output combination for desired logic block is generated while logic synthesis and on startup FPGA loads this synthesis into its lookup tables so all FPGA behaves like a big logic circuit.

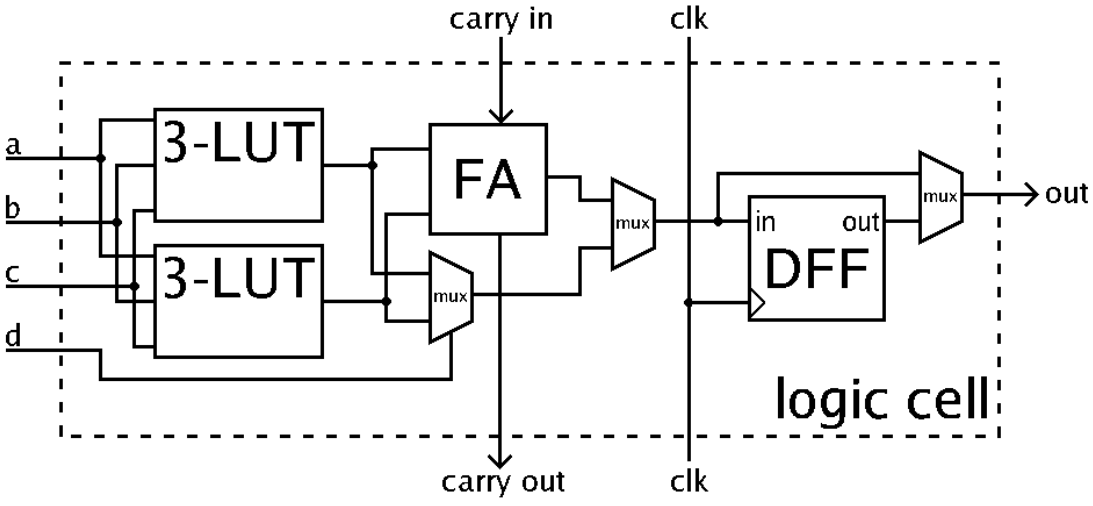

Figure 7: FPGA logic cell illustration (Retrieved from [8])

There may be also synchronous block RAM modules in a FPGA chip. These RAM modules can be used separate or combined. These modules can be used as ROM or RAM.

4. ANALAYSIS AND MODELLING

4.1. Analysis and Requirements

Programmer should assume that they can program processor without knowing internal micro architecture. Programmer shouldn't need to change the instruction order or programmer shouldn't need to insert stall instructions for correctness of execution. The processor itself must provide correctness of execution so processor's internal architecture must prevent any pipeline, data or branch hazard. Processor's pipeline must utilize one instruction per cycle without any hazard. Correctness of execution is the most important requirement for the whole processor project. Efficiency of executions is also important requirements for the pipeline architecture of the processor. If processor met these requirements, it can execute any desired operation to be expected from a standard processor.

4.2. Architectural Model

Mixture of THUMB and MIPS architectures were used to inspire new architecture so RISC architecture was chosen as frame architecture. The processor is in Harvard architecture so it has separate instruction and data memory. Also it has fixed length Load/Store instruction set architecture which means only load and store instructions access to data memory [12]. This leads to single cycle instruction execution capability with easier pipeline design.

The classic RISC pipeline architecture is used for pipeline design so processor's pipeline has 5 stages [6]:

- Instruction fetch
    - Instruction is fetched from instruction memory to instruction register where program counter points
- Instruction decode
    - Instruction type, operands and offsets are determined
    - Necessary control signals are generated according to instruction type
    - Instruction enters to pipeline
    - Instruction control signals enters to pipeline control registers
    - Operand registers are read end of this stage
- Execute (ALU)
    - Instruction operands enter to ALU
    - Actual computation is done
- Memory access
    - If instruction is memory access instruction, memory access is done
    - If instruction doesn't need memory access, instruction is forwarded to next stage
- Write back
    - Instruction execution results or memory access results are written into register file

Timing diagram of pipeline stages is shown in Figure 2. Fetch and memory read/write stages slightly overflow next clock. Register write and register read operations can be done in one clock cycle. Execute and decode operations run exactly one clock cycle.

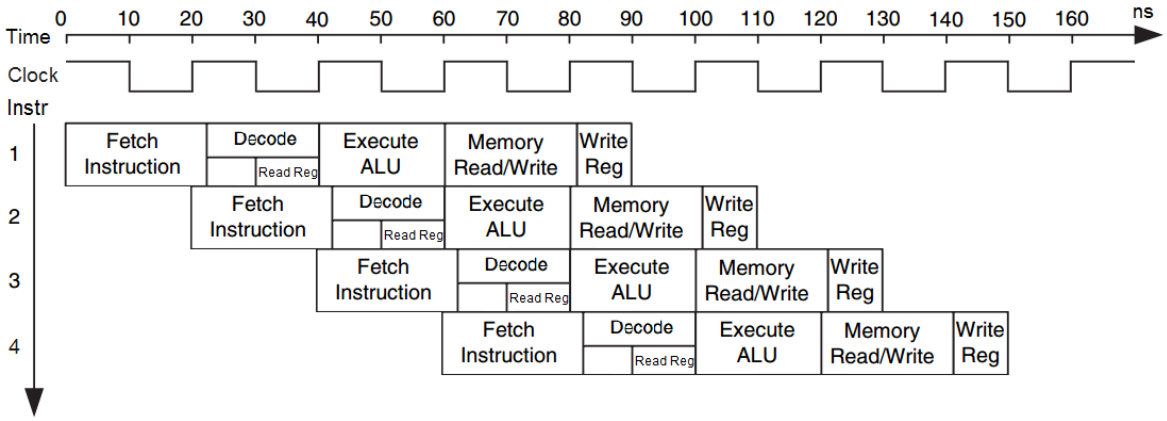

Figure 8: Timing diagram of pipeline stages (Retrieved from [5] and modified)

The processor has 16 Bit CPU, ALU, register and data bus architecture. Address bus for data and instruction memory is 12 bit so maximum addressable instruction and data memory is 4K word and it is equal to 8KB.

4.2.1. Organization

Slightly modified MIPS architecture is used for organizational design.

Organization of blocks with 5 stage pipeline is shown in Figure 3. [5] Detailed organization will be given in design section.

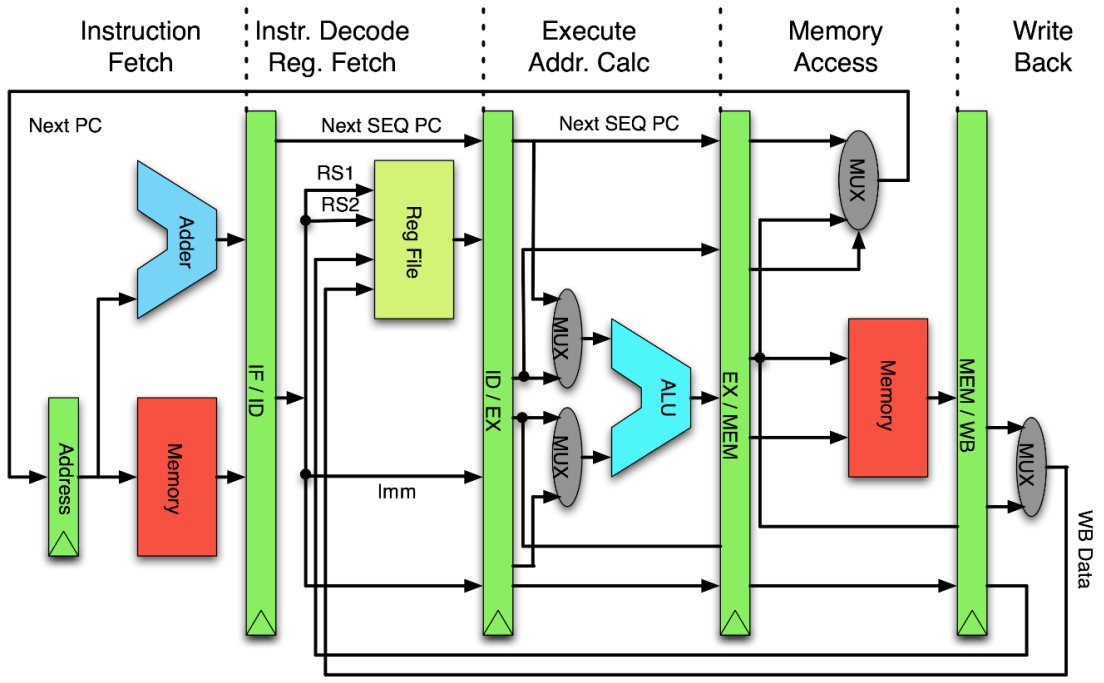

Figure 9: Classic 5 stage RISC pipeline organization (Retrieved from [10] and modified)

There are 8 general purpose 16 bit registers in register file. Also there are 16 bit instruction and condition code register, 12 bit program counter and address register. All general purpose register accessible and they can transfer data between them. PC can only be modifiable via branch instructions, its content is not accessible. IR is not accessible. CCR's some condition bits can be changed via some instructions and they can be usable via only branch instructions.

Condition code register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
								IEN	IRQ	I		N	Z	C	V

Bit	Assignment
N	Negative/Less than
Z	Zero
C	Carry/Borrow
V	Overflow
IEN	Interrupt Enable
IRQ	Interrupt Request

Registers

Abbreviation	Equivalent	Bit
R0	General purpose register 0	16
R1	General purpose register 1	16
R2	General purpose register 2	16
R3	General purpose register 3	16
R4	General purpose register 4	16
R5	General purpose register 5	16
R6	General purpose register 6	16
R7	General purpose register 7	16
IR	Instruction register	16
PC	Program counter	12
AR	Address register	12
CCR	Condition code register	16

4.2.2. Instruction Set

All instructions are fixed 16 Bit long. There are 11 type of instruction. Load/Store instruction set, so only load and store instructions can access to data memory. ARM Corporation's THUMB instruction set architecture were used to inspire new instruction set [11].

Instruction Types

1. Inherent Instructions
2. Shift Instructions
3. Add/Subtract Register Immediate Instructions
4. Move/Compare/Add/Subtract Immediate Instructions
5. ALU Instructions
6. Load/Store with Register Offset Instructions
7. Load/Store with Immediate Offset Instructions
8. Conditional Branch Instructions
9. Unconditional Branch Instruction
10. Branch to Subroutine Instruction
11. Move Long Immediate

Instruction Format

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	X	X	X	X	X	X	X	X	X	Op		
2	0	0	0	1	Op		Off4				Rs			Rd		
3	0	0	1	Op	Off6						Rs			Rd		
4	0	1	0	Op		Off8								Rd		
5	0	1	1	Op			Rn				Rs			Rd		
6	1	0	0	X	X	X	Op	Ro			Rb			Rd		
7	1	0	1	Op	Off6						Rb			Rd		
8	1	1	0	Cond				Off9								
9	1	1	1	0	1	Off11										
10	1	1	1	0	1	Off11										
11	1	1	1	1	Off12											

Operand Table

Abbreviation	Equivalent	Bit
Rd	Destination Register	3
Rs	Source Register	3
Rb	Base Register	3
Ro	Offset Register	3
Off#	Immediate Offset	#
Op	Operation	X
Cond	Condition	X

Complete Instruction Set

Assembly	Action	Updates
NOP	No operation	
ION	Interrupts on	I
IOF	Interrupts off	I
RTI	Return from interrupt	
RTS	Return from subroutine	
LSL Rd, Rs, #Off4	Logic shift left by #Off4	N Z C
LSR Rd, Rs, #Off4	Logic shift right by #Off4	N Z C
ASR Rd, Rs, #Off4	Arithmetic shift right by #Off4	N Z C
CSL Rd, Rs, #Off4	Circular shift left by #Off4	N Z C
ADD Rd, Rs, #Off6	Rd := Rs + #Off6	N Z C V
SUB Rd, Rs, #Off6	Rd := Rs - #Off6	N Z C V
MOV Rd, #Off8	Rd := #Off8	N Z
CMP Rd, #Off8	Rd - #Off8	N Z C V
ADD Rd, #Off8	Rd := Rd + #Off8	N Z C V
SUB Rd, #Off8	Rd := Rd - #Off8	N Z C V
AND Rd, Rs, Rn	Rd := Rs AND Rn	N Z
OR Rd, Rs, Rn	Rd := Rs OR Rn	N Z
XOR Rd, Rs, Rn	Rd := Rs XOR Rn	N Z
LSL Rd, Rs, Rn	Rd := Rs << Rn	N Z C
LSR Rd, Rs, Rn	Rd := Rs >> Rn	N Z C

ASR Rd, Rs, Rn	Rd := Rs ASR Rn	N Z C
CSL Rd, Rs, Rn	Rd := Rs CSL Rn	N Z C
ADD Rd, Rs, Rn	Rd := Rs + Rn	N Z C V
SUB Rd, Rs, Rn	Rd := Rs - Rn	N Z C V
NEG Rd, Rs	Rd := -Rs	N Z
NOT Rd, Rs	Rd := NOT Rs	N Z
CMP Rs, Rn	Rs – Rn	N Z C V
TST Rs, Rn	Rs AND Rn	N Z C V
LD Rd, [Rb, Ro]	Rd := MEM[Rb + Ro]	
LD Rd, [Rb, #Off6]	Rd := MEM[Rb + #Off6]	
STR [Rb, #Off6], Rd	MEM[Rb + #Off6] := Rd	
BEQ #Off9	Branch if equal	
BNE #Off9	Branch if not equal	
BCS #Off9	Branch if unsigned higher or same	
BCC #Off9	Branch if unsigned lower	
BMI #Off9	Branch if negative	
BPL #Off9	Branch if positive or zero	
BVS #Off9	Branch if overflow	
BVC #Off9	Branch if no overflow	
BHI #Off9	Branch if unsigned higher	
BLS #Off9	Branch if unsigned lower or same	
BGE #Off9	Branch if signed greater than or equal	
BLT #Off9	Branch if signed less than	
BGT #Off9	Branch if signed greater than	
BLE #Off9	Branch if signed less than or equal	
BAL #Off11	Branch always	
BTS #Off11	Branch to subroutine	
MOVL R0 #Off12	R0 := #Off12	

4.2.2.1. Inherent Instructions

These instructions are implied (inherent) instructions which means they have not any operand. Operation table for inherent instructions is incomplete, it can be extend.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	X	X	X	X	X	X	X	X	X	Op		

Op	Assembly	Action	Updates
000	NOP	No operation	
001	ION	Interrupts on	I
010	IOF	Interrupts off	I
011	RTI	Return from interrupt	
100	RTS	Return from subroutine	

Example Instructions:

```
NOP
ION
```

4.2.2.2. Shift Instructions

Immediate shift operation instructions with two register operands. Logical left, logical right, arithmetic right, circular right shift operations are implemented.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	Op		Off4				Rs			Rd		

Op	Assembly	Action	Updates
00	LSL Rd, Rs, #Off4	Logic shift left by #Off4	N Z C
01	LSR Rd, Rs, #Off4	Logic shift right by #Off4	N Z C
10	ASR Rd, Rs, #Off4	Arithmetic shift right by #Off4	N Z C
11	CSL Rd, Rs, #Off4	Circular shift left by #Off4	N Z C

Example Instruction:

```
LSR  R2, R5, #10
```

4.2.2.3. Add/Subtract Register Immediate Instructions

Add and subtract operations with two register operands and immediate positive 6 bit offset.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	Op	Off6						Rs			Rd		

Op	Assembly	Action	Updates
0	ADD Rd, Rs, #Off6	Rd := Rs + #Off6	N Z C V
1	SUB Rd, Rs, #Off6	Rd := Rs - #Off6	N Z C V

Example Instruction:

```
ADD  R1, R4, #60
```

4.2.2.4. Move/Compare/Add/Subtract Immediate Instructions

Move, compare, add and subtract operations with one register operand and immediate positive 8 bit offset. Compare operations only updates condition code register, no write back stage.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	Op		Rd			Off8							

Op	Assembly	Action	Updates
00	MOV Rd, #Off8	Rd := #Off8	N Z
01	CMP Rd, #Off8	Rd - #Off8	N Z C V
10	ADD Rd, #Off8	Rd := Rd + #Off8	N Z C V
11	SUB Rd, #Off8	Rd := Rd - #Off8	N Z C V

Example Instructions:

```
MOV  R1, #130
CMP  R2, #25
ADD  R5, #40
SUB  R3, #200
```

4.2.2.5. ALU Instructions

These instructions performs ALU operations like logical, shift and compare instructions with two or three register operands. Bitwise logical operations, add with carry and subtract with borrow operations, shift operations are implemented with three register operands. Complement and negations operations, compare and test operations are implemented with two register operands. Operation table for ALU instructions is incomplete, it can be extend.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	Op				Rn			Rs			Rd		

Op	Assembly	Action	Updates
0000	AND Rd, Rs, Rn	Rd := Rs AND Rn	N Z
0001	OR Rd, Rs, Rn	Rd := Rs OR Rn	N Z
0010	XOR Rd, Rs, Rn	Rd := Rs XOR Rn	N Z
0011	LSL Rd, Rs, Rn	Rd := Rs << Rn	N Z C
0100	LSR Rd, Rs, Rn	Rd := Rs >> Rn	N Z C
0101	ASR Rd, Rs, Rn	Rd := Rs ASR Rn	N Z C
0110	CSL Rd, Rs, Rn	Rd := Rs CSL Rn	N Z C
0111	ADD Rd, Rs, Rn	Rd := Rs + Rn	N Z C V
1000	SUB Rd, Rs, Rn	Rd := Rs - Rn	N Z C V
1001	NEG Rd, Rs	Rd := -Rs	N Z
1010	NOT Rd, Rs	Rd := NOT Rs	N Z
1011	CMP Rs, Rn	Rs – Rn	N Z C V
1100	TST Rs, Rn	Rs AND Rn	N Z C V

Example Instructions:

```
XOR  R2, R5, R3
CSL  R1, R3, R4
ADD  R4, R2, R6
NOT  R5, R6
CMP  R3, R5
```

4.2.2.6. Load/Store with Register Offset Instructions

These instructions transfer 16 bit word between registers and memory. Memory address operands are base register, offset register and 3 bit immediate offset.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	X	X	X	Op	Ro			Rb			Rd		

Op	Assembly	Action
0	LD Rd, [Rb, Ro]	Rd := MEM[Rb + Ro]

Example Instructions:

```
LD   R3, [R5, R2]
```

4.2.2.7. Load/Store with Immediate Offset Instructions

These instructions transfer 16 bit word between registers and memory. Memory address operands are base register and 6 bit immediate offset.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	Op	Off6						Rb			Rd		

Op	Assembly	Action
0	LD Rd, [Rb, #Off6]	Rd := MEM[Rb + #Off6]
1	STR [Rb, #Off6], Rd	MEM[Rb + #Off6] := Rd

Example Instructions:

```
LD   R3, [R5, #10]
LD   R1, [R0]       // offset is #0
STR  [R5, #26], R1
STR  [R3], R2       // offset is #0
```

4.2.2.8. Conditional Branch Instructions

These instructions perform a conditional branch depending on the state of the condition code N Z C V bits with 8 bit offset from PC register. [2]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	Cond				Off9								

Cond	Assembly	Condition Logic	Condition
0000	BEQ #Off9	Z = 1	equal
0001	BNE #Off9	Z = 0	not equal
0010	BCS #Off9	C = 1	unsigned higher or same
0011	BCC #Off9	C = 0	unsigned lower
0100	BMI #Off9	N = 1	negative
0101	BPL #Off9	N = 0	positive or zero
0110	BVS #Off9	V = 1	overflow
0111	BVC #Off9	V = 0	no overflow
1000	BHI #Off9	Z' * C	unsigned higher
1001	BLS #Off9	Z + C'	unsigned lower or same
1010	BGE #Off9	(N * V) + (N' * V')	signed greater than or equal
1011	BLT #Off9	N XOR V	signed less than
1100	BGT #Off9	(N * Z' * V) + (N' * Z' * V')	signed greater than
1101	BLE #Off9	Z + (N XOR V)	signed less than or equal

Example Instructions:

```
BEQ  #50  // BEQ    Label
BGT  #25  // BGT    Label
```

4.2.2.9. Unconditional Branch Instruction

This instruction performs an unconditional branch with 12 bit offset from PC register.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	#Off11										

Assembly	Action
BAL #Off11	PC := PC + #Off11

Example Instruction:

```
BAL  Label     // Label may be #534 offset
```

4.2.2.10. Branch to Subroutine Instruction

This instruction performs a subroutine call with 11 bit offset from PC register.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	#Off11										

Assembly	Action
BTS #Off11	PC := PC + #Off11

Example Instruction:

```
BTS  Label     // Label may be #532 offset
```

4.2.2.11. Move Long Immediate

Move long moves 12 bit immediate offset to R0 register.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	Off12											

Assembly	Action
MOVL R0 #Off12	R0 := #Off12

Example Instruction:

```
MOVL R0 #1368
```

5. DESIGN, IMPLEMENTATION AND TEST

Processor is first implemented on Logisim simulation program. After successful simulation result, processor is implemented with Verilog description language and tested on Xilinx ISim simulation tool. After successful design and simulation result, finally processor is implemented on Xilinx FPGA.

5.1. Structure of the Processor

All necessary control signals are generated by control unit. These control signal enters to pipeline control registers and control signals follows instruction through pipeline. Hazard unit watches these control signals and detects possible hazards.

Following Figure shows; main core organization, control unit, pipeline control registers, hazard unit, branch unit and interrupt unit.

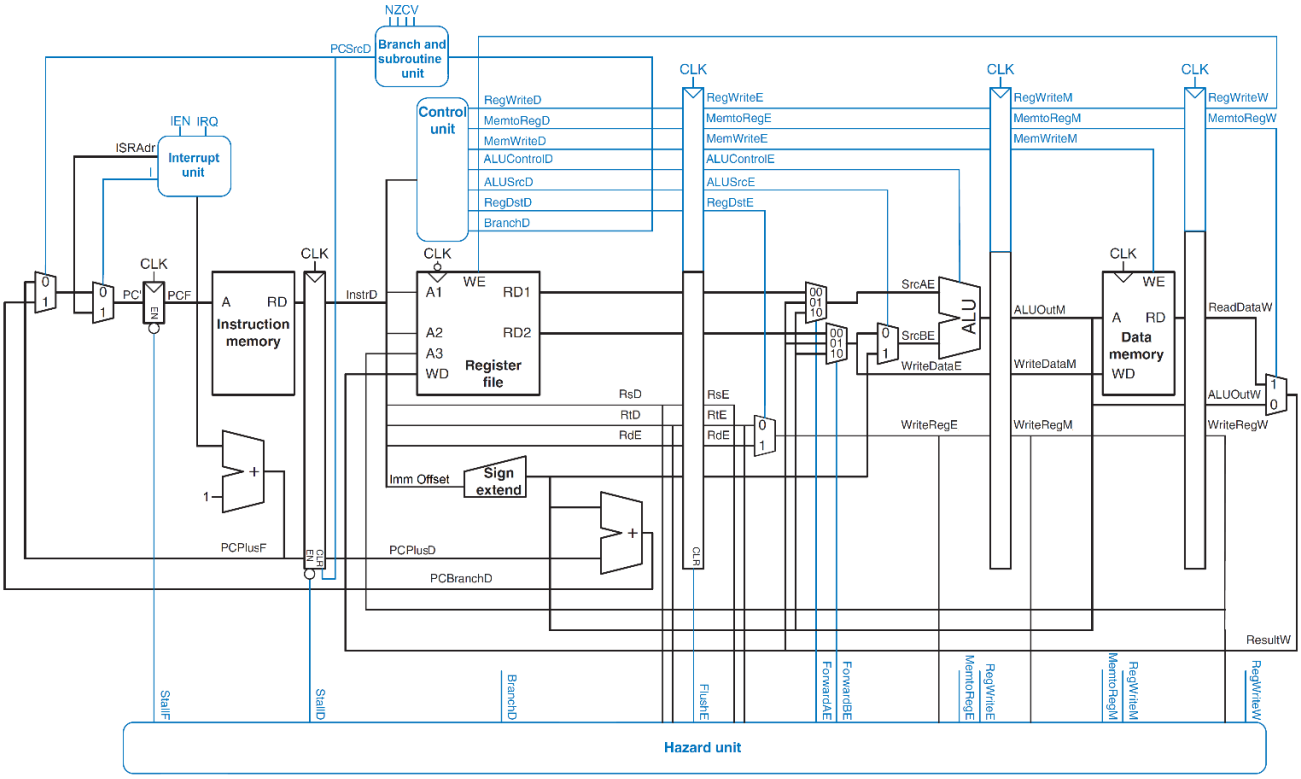

Figure 10: Processor organization schematic with control and hazard units (Retrieved from [5] and modified)

5.2. Design and Simulation on Logisim

Logisim is an educational tool for designing and simulating digital logic circuits. It can be used to design and simulate entire CPUs. It can give detailed look to processor before the FPGA implementation. Logisim version 2.7.1 is used for logic simulation.

5.2.1. ALU

Arithmetic and logic unit of processor. It has 16 bit data bit length, two data input and one output port. There are four flags which are zero (Z), negative (N), overflow (V), carry (C) flags. It has 8 bit selection port. First 4 bit selects ALU operation and last 4 bit selects which flags are going to update. ALU can perform these operations:

- Logical AND
- Logical OR
- Logical XOR
- Logical left shift
- Logical right shift
- Arithmetic shift right
- Circular shift left
- Arithmetic add
- Arithmetic subtract
- Arithmetic negation
- Logical NOT

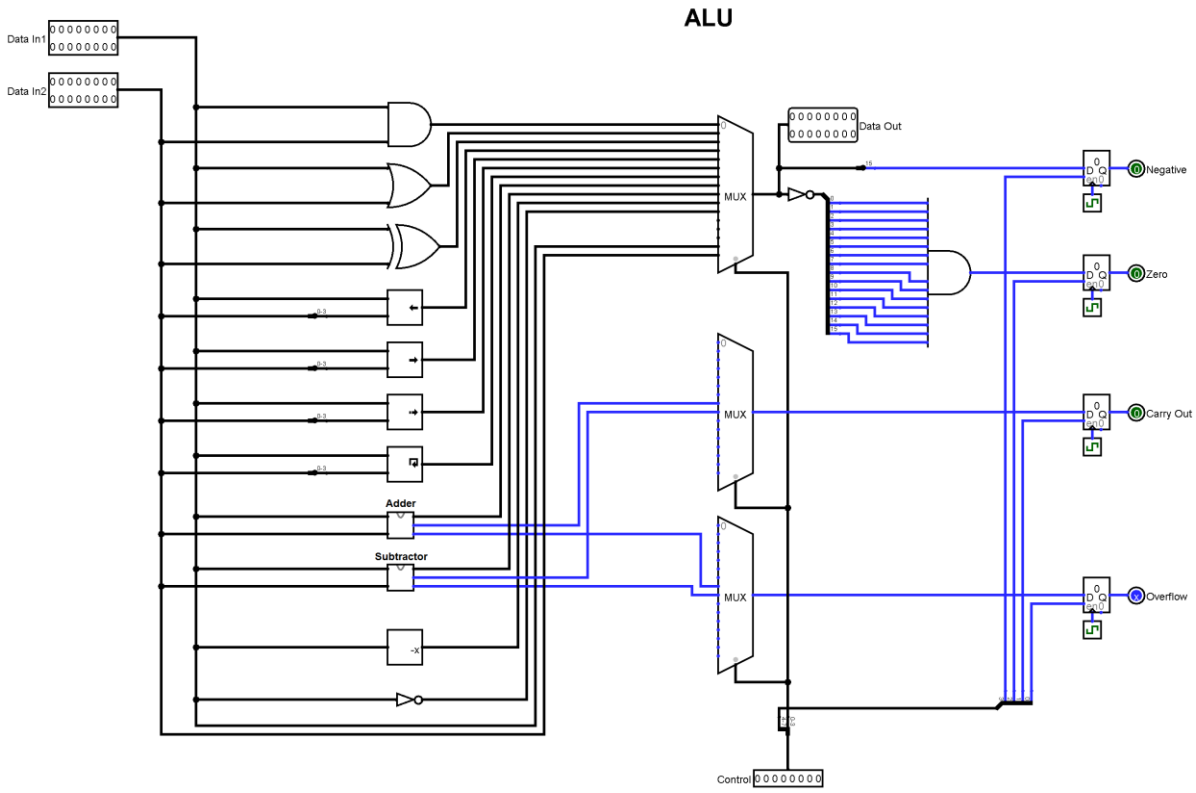

Figure 11: ALU on Logisim

5.2.2. Register File

All general purpose registers resides in register file. These registers can be actively used for instruction operands. There are eight 16 bit registers in register file. There are one data in and two data output ports, all of these ports have their own 3 bit address selection port. Also, there is Read/Write selection input port.

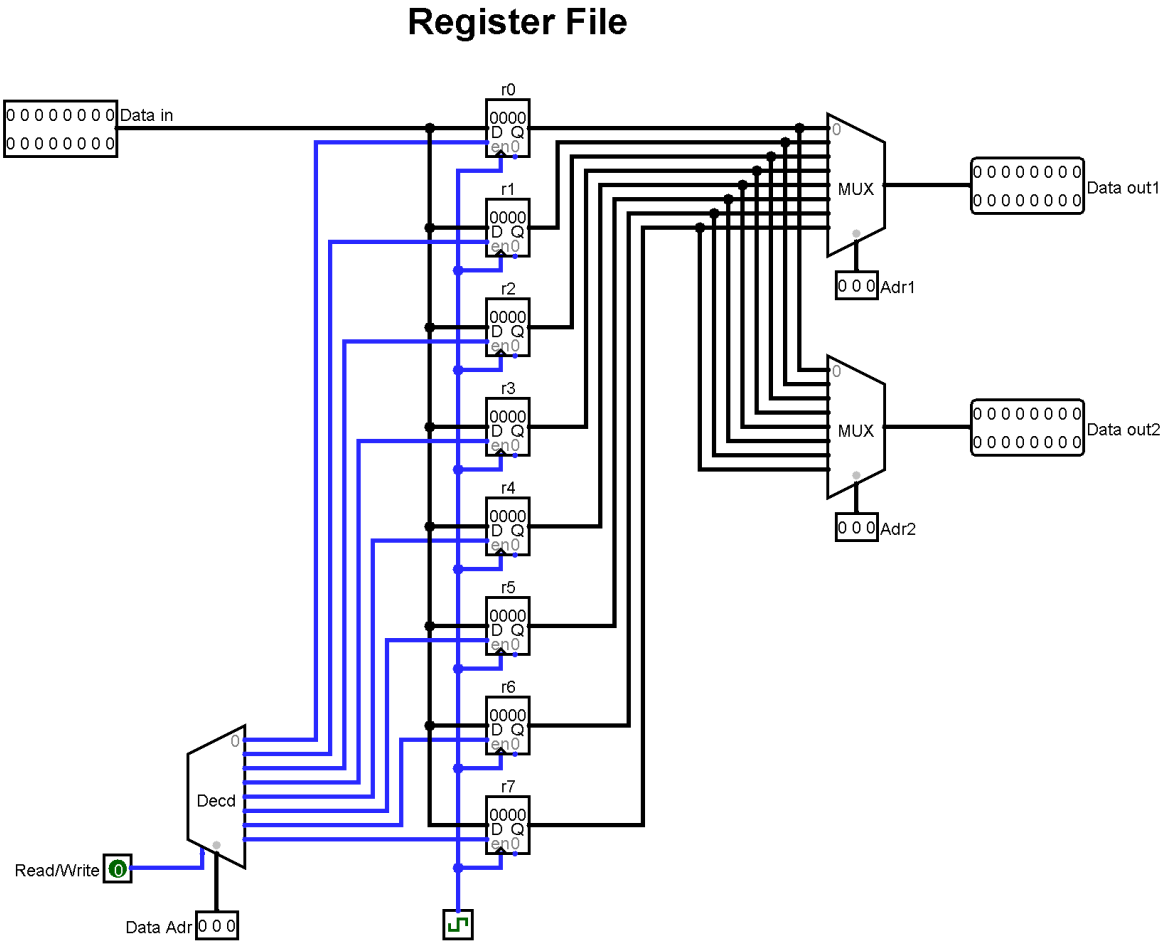

Figure 12: Register file on Logisim

5.2.3. Fetch Unit

Fetch unit fetches instruction from instruction memory where 12 bit program counter points then loads to 16 bit instruction register (IR). Normally program counter will be incremented by one. If branch operation or subroutine operation occurs then PC added with offset. If interrupt occurs then PC updated with predefined address of interrupt service routine. Hazard unit may stall or flush fetch. Instruction ROM has 16 bit data width and 12 bit address depth.

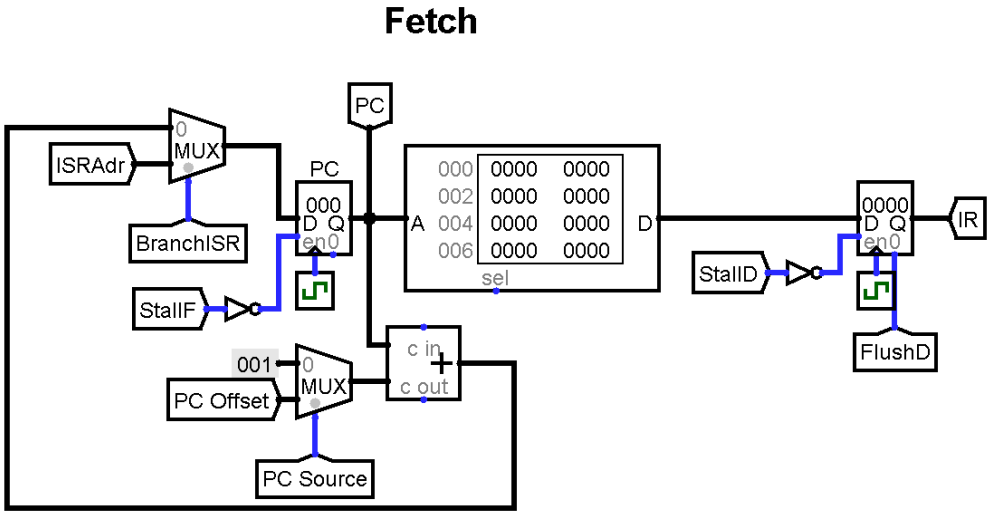

Figure 13: Fetch unit on Logisim

5.2.4. Decode Unit

After fetch stage, instruction is decoded. All decode unit is a combinational circuit. First instruction type is determined, then necessary control signals are generated according to instruction. Some control signal enters to pipeline control registers to reach their pipeline stage.

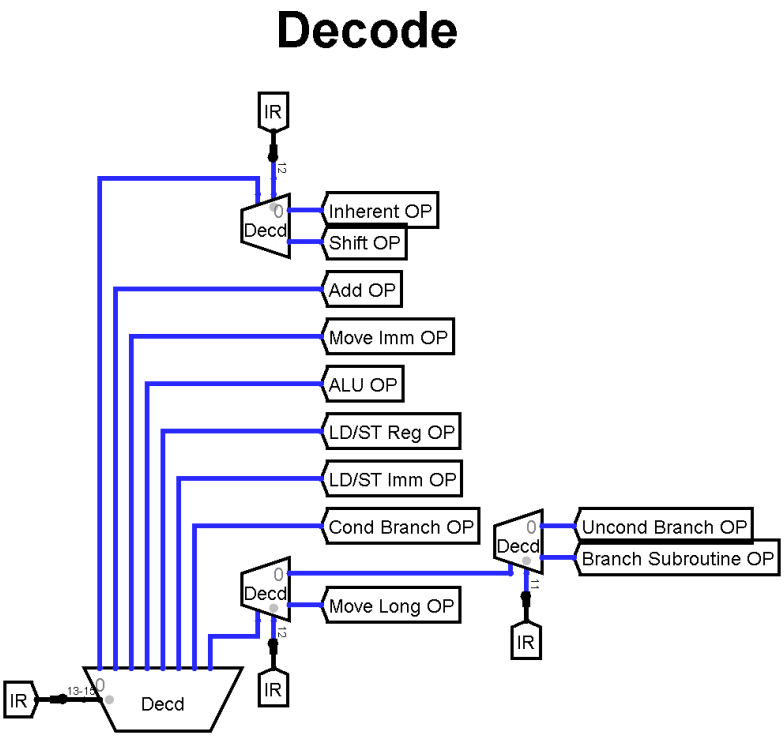

Figure 14: Main decode unit on Logisim

Necessary control signals are generated in instruction's decode logic. Example decode logic for Add/Subtract Register immediate instructions:

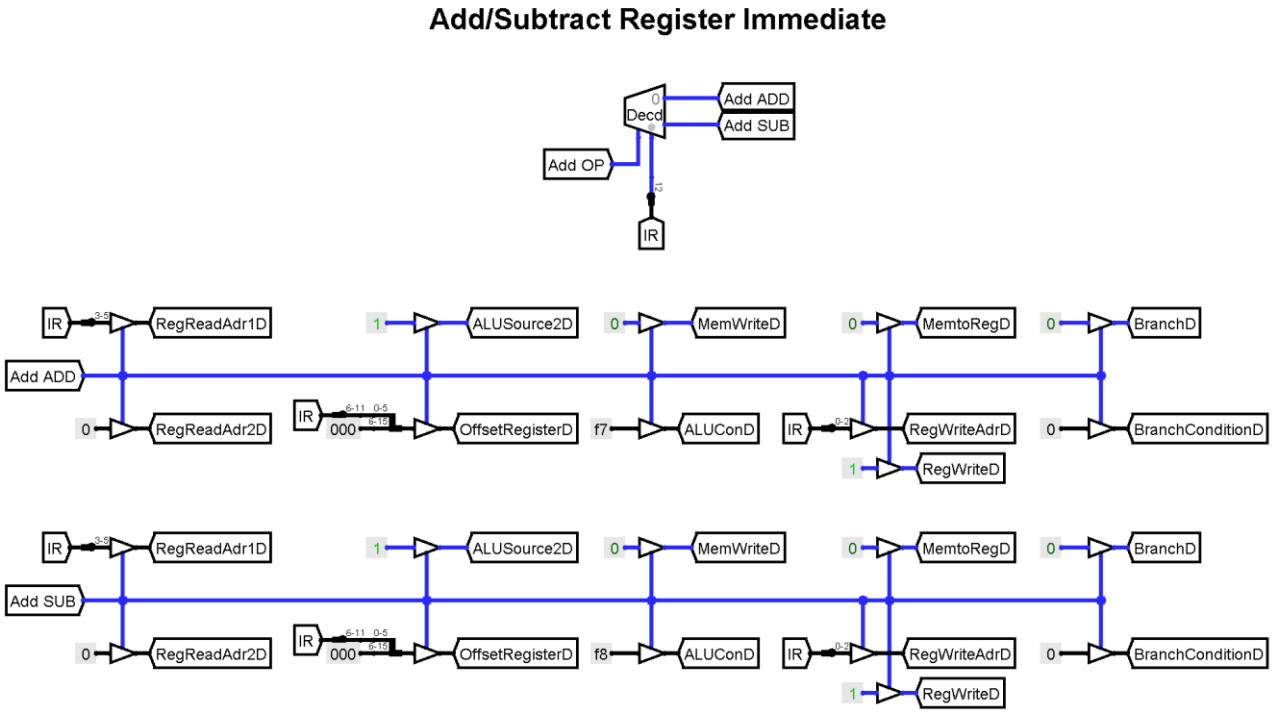

Figure 15: Decode for Add/Subtract Immediate on Logisim

5.2.5. I/O and Memory Unit

Processor has memory mapped I/O architecture so all peripherals can be accessed via only load and store instruction just like a memory. All I/O units their unique address in memory address space. Memory RAM has 16 bit data width and 12 bit address depth.

Processor input port address is FF0

Processor output port address is FF1

It is possible to add new I/O modules to empty address space.

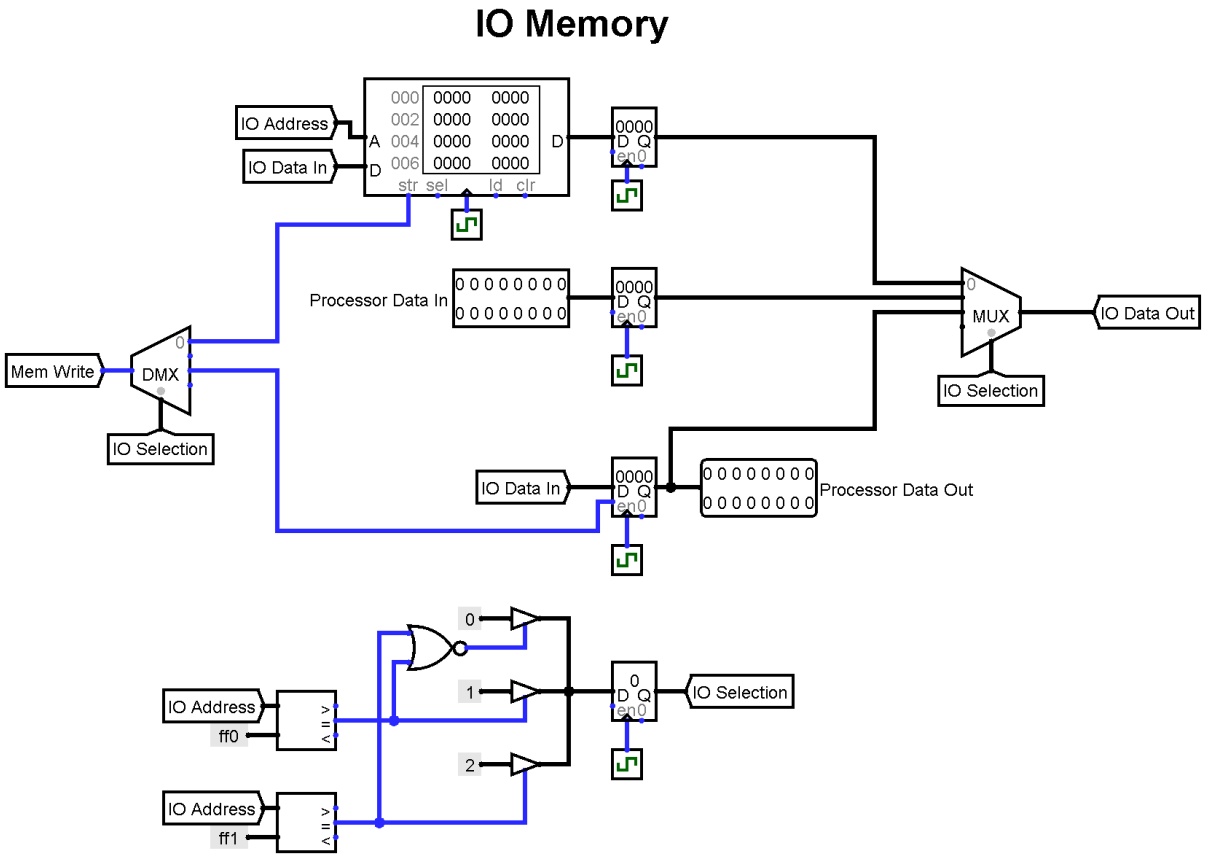

Figure 16: I/O and Memory unit on Logisim

5.2.6. Pipeline Control Registers

Pipeline registers transfers control signals made in decode stage. Control signals waits their execution stage. Control signals and their instruction flow on pipeline registers in parallel. Execution stage of pipeline can be flushed, this signals used for stall operation.

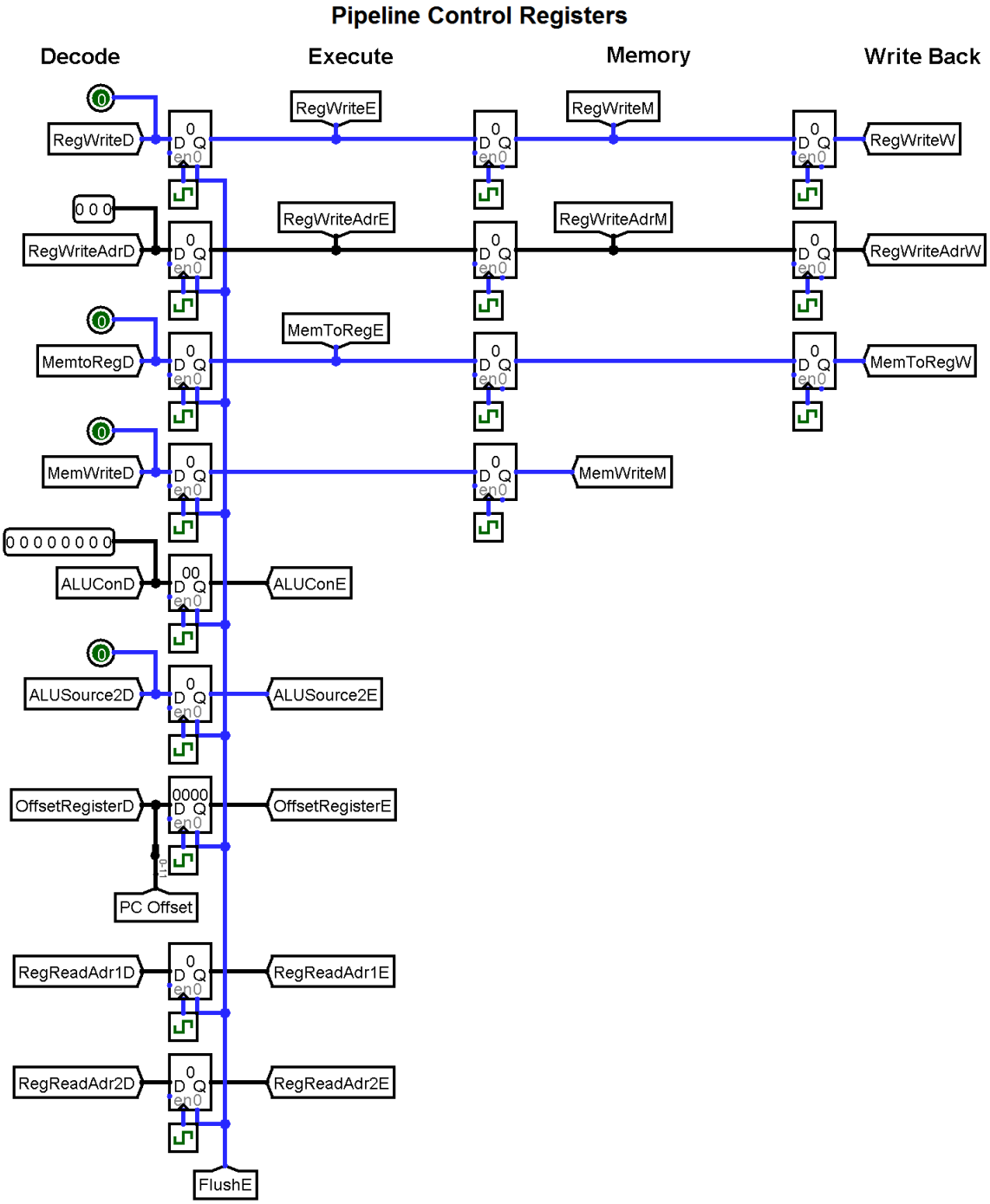

Figure 17: Pipeline control registers on Logisim

5.2.7. Branch Control Unit

Branch control unit is a combination logic circuit. It determines if branch is taken or not. Branching decision is made by using zero, negative, overflow and carry flags according to branch condition type.

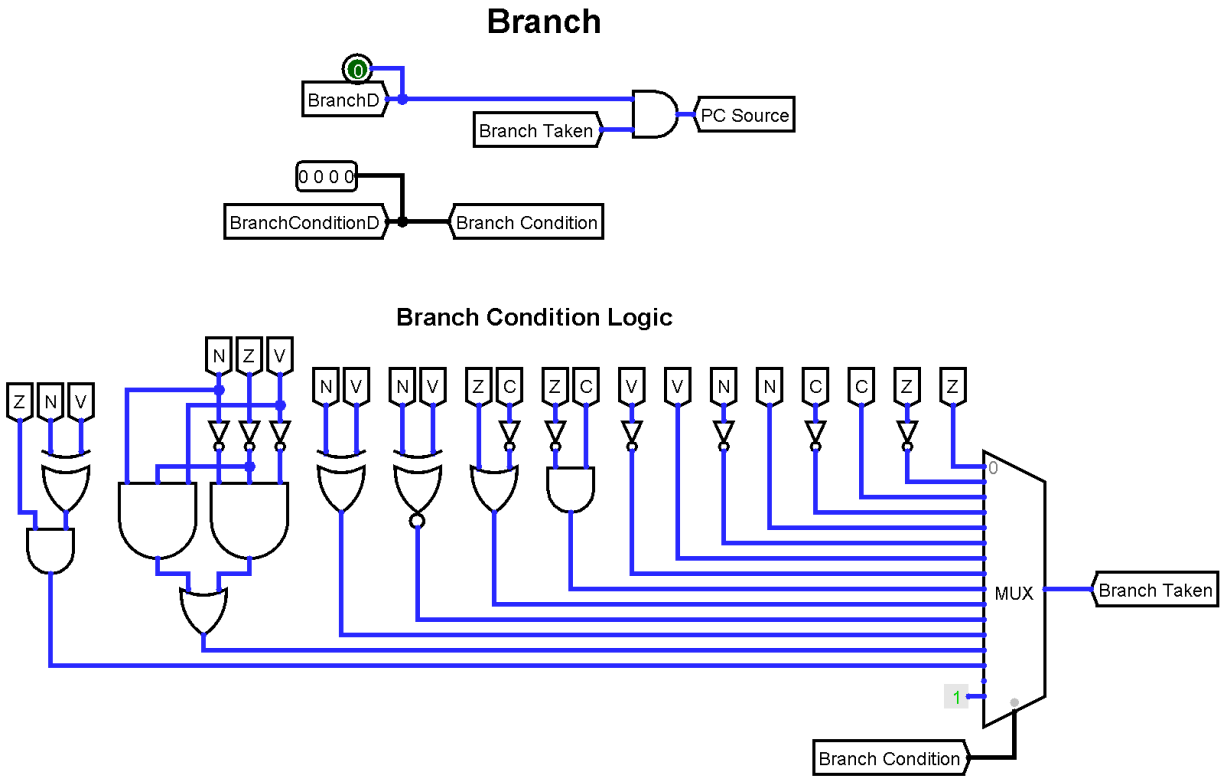

Figure 18: Branch control unit on Logisim

5.2.8. Interrupt Control Unit

If interrupt enable (IEN) flag is set then, interrupt request (IRQ) signal will branch processor to interrupt service routine (ISR) address. IEN flag can be set with ION and IOF instructions. Return address is preserved in register. Nested interrupts is not permitted. Processor can enter only one interrupt at a time. Default ISR address is 001.

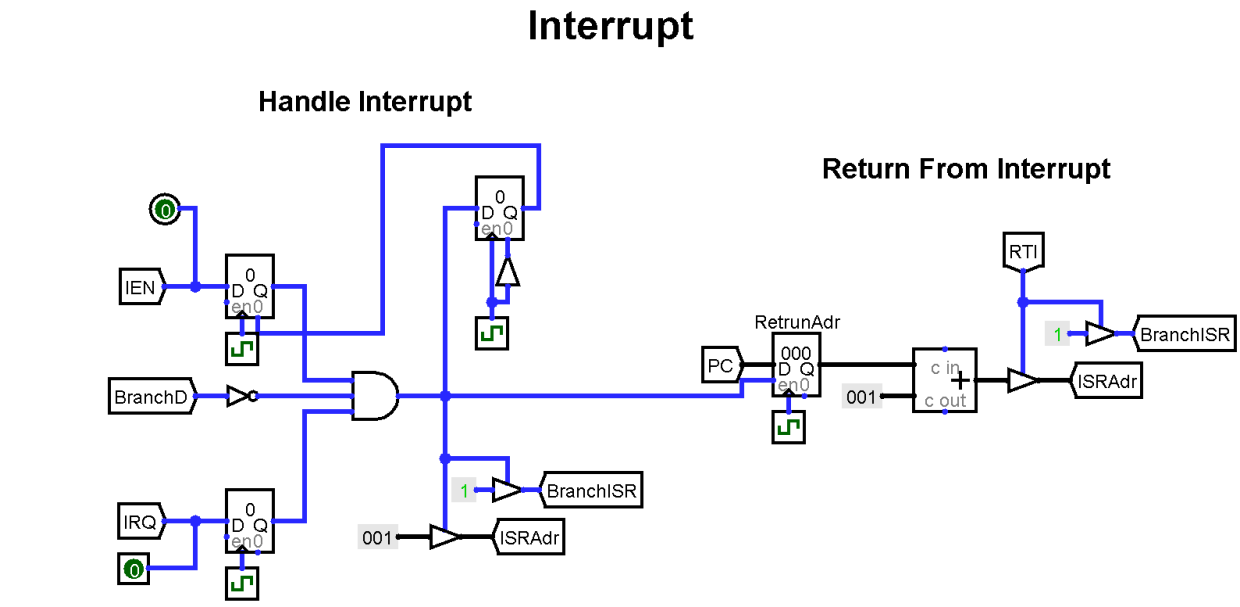

Figure 19: Interrupt unit on Logisim

5.2.9. Hazard Detection Unit

Hazard detection unit watches control signals on pipeline and if it detects a possible hazard then it creates forward, stall of flush signals to prevent hazard. Detection unit compares source and target operands of adjacent instructions. If source and target operands of adjacent instructions are same then detection unit checks for hazardous condition. Data forwarding solves register type instruction's data dependency. Flush and stall operations costs one clock cycle.

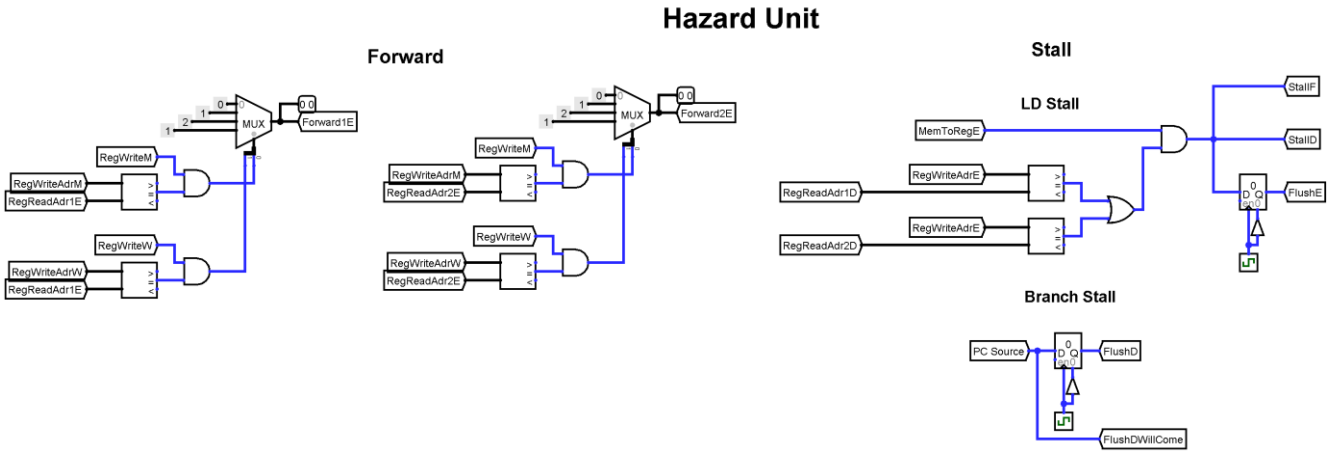

Figure 20: Hazard unit on Logisim

5.2.10. Subroutine Stack

Branch to subroutine (BTS) instruction pushes return address to the subroutine stack and return from subroutine (RTS) instruction pops return address from stack to program counter. Stack has 8 registers so at most 8 nested subroutine call can be made. After 8 subroutine call, stack will overflow and return addresses will be lost. This leads incorrect execution sequence. Programmer should consider stack size.

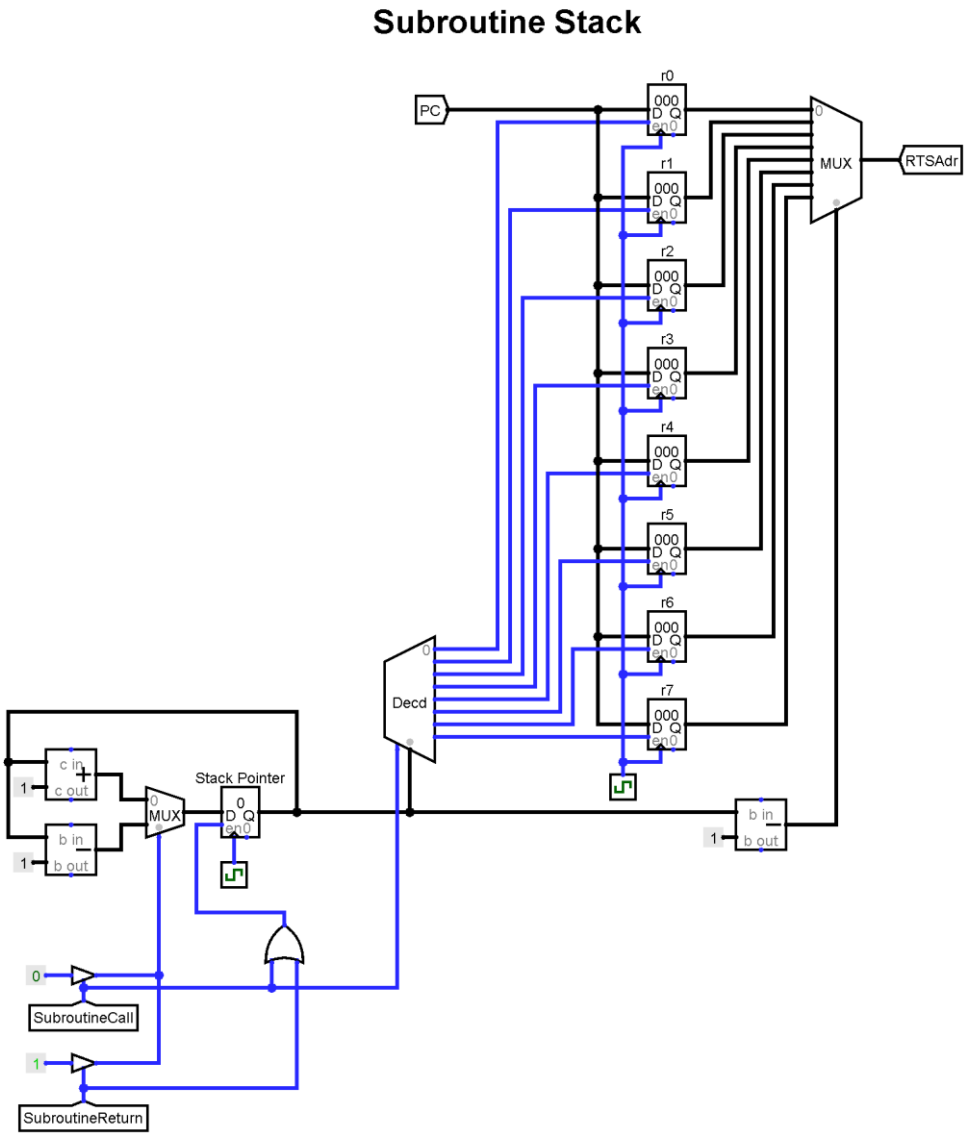

Figure 21: Subroutine stack on Logisim

5.2.11. Core Organization

All operational units resides in this block which are Register file, forwarding mechanism, ALU, data memory. This main block contains units which are responsible for register fetch, execute, memory access and write back stages of pipeline.

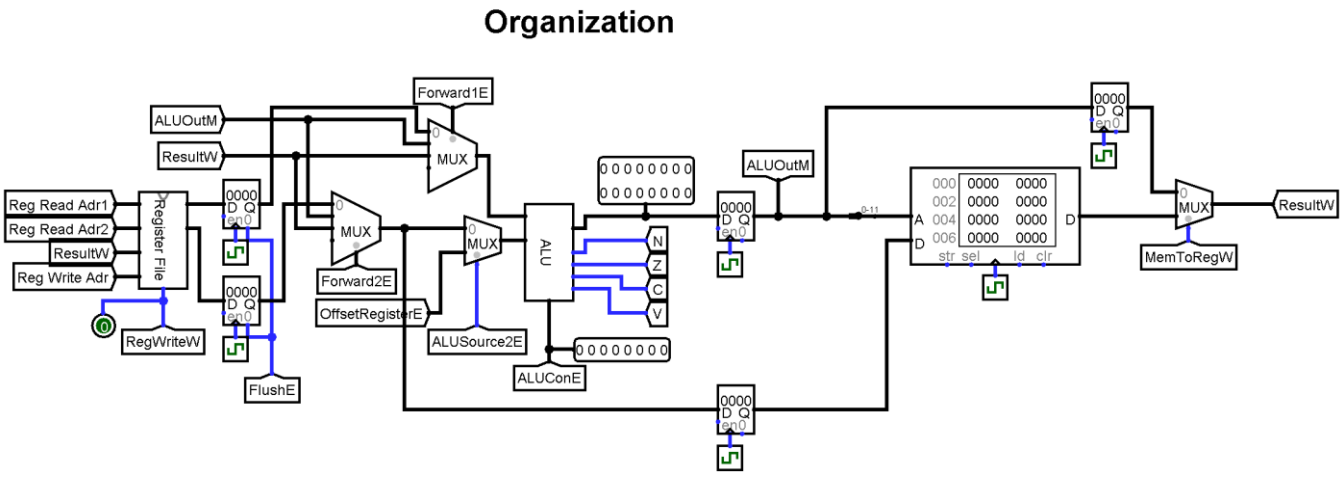

Figure 22: Register File on Logisim

5.3. Assembler

A Python script is written for assemble instructions and create binary machine instructions. Assembler uses simple regular expressions to parse instruction mnemonics. Every instruction has unique name and addressing name so simple instruction assemble function is run through assembler file within a loop. After generating instruction binaries, these are converted to Logisim memory input file format, ISim memory input file format and Xilinx FPGA mem input file format.

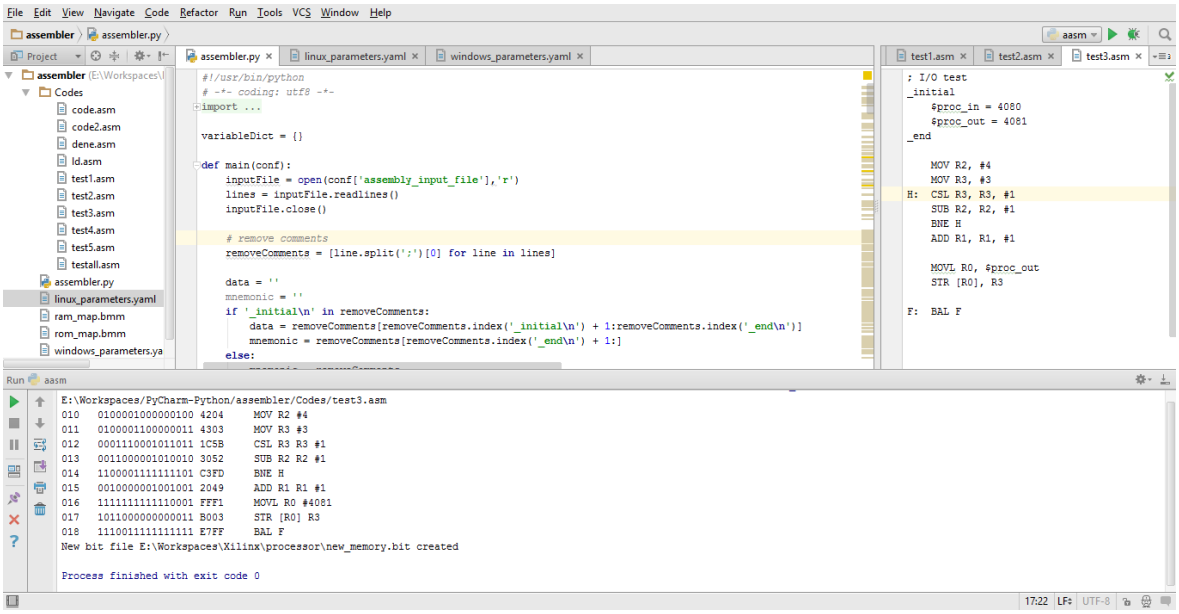

Figure 23: Assembler in PyCharm Python IDE

Assembler takes configuration file as a command line argument. Assembler only reads these configuration file for necessary parameters. Configuration file is in YAML data serialization format. "**PyYAML**" module for Python can parse this YAML configuration file.

Assembler read assembly input file and generates assembly instructions. Assembler can also use FPGA bit configuration file to update program and data memory of processor for FPGA implementation. Assembler takes original FPGA bit file and uses Xilinx's "**data2mem**" tool to create new FPGA bit file. Original bit file path, modified bit file path and data2mem tool executable path must be gives to assembler as a parameter in configuration file.

Assembler can program FPGA via Xilinx's "**iMPACT**" tool. If iMPATC tool executable path is given to assembler, assembler can program FPGA directly.

Assembler uses Python's "**subprocess**" module for execute iMACT and data2mem external executables in Python script.

Assembler can assemble instructions, create memory file for Logisim, create memory file for ISim, create bit file for FPGA and program FPGA in one command. It makes easy to develop and run programs for processor. Assembler needs root access to program FPGA.

Assembler configuration parameters are;

- assembly_input_file
- logisim_rom_output_file
- logisim_ram_output_file
- fpga_simulation_rom_output_file
- fpga_simulation_ram_output_file
- xilinx_data2mem_path
- fpga_top_module_bit_input_file
- fpga_rom_bmm_input_file
- fpga_ram_bmm_input_file
- fpga_modified_bit_output_file
- xilinx_impact_path

Example assembler YAML configuration file:

```
### Parameters for Linux ###
# Assembly input file path
assembly_input_file: /media/BELGELER/Workspaces/PyCharm-Python/assembler/Codes/test5.asm
## Simulation
# Logisim simulation memory file path for instruction ROM and data RAM
logisim_rom_output_file: /media/BELGELER/Workspaces/Logisim/code.rom
logisim_ram_output_file: /media/BELGELER/Workspaces/Logisim/data.ram
# Xilinx ISIM simulation memory file path for instruction ROM and data RAM
fpga_simulation_rom_output_file: /media/BELGELER/Workspaces/Xilinx/processor/ROM_4K.mif
fpga_simulation_ram_output_file: /media/BELGELER/Workspaces/Xilinx/processor/RAM_4K.mif
# Xilinx data2mem tool path
xilinx_data2mem_path: /opt/Xilinx/13.2/ISE_DS/ISE/bin/lin64/data2mem
## FPGA
# FPGA device top module bit file
fpga_top_module_bit_input_file: /media/BELGELER/Workspaces/Xilinx/processor/Atlys_Spartan6.bit
# FPGA memory map file for instruction ROM and data RAM
fpga_rom_bmm_input_file: /media/BELGELER/Workspaces/PyCharm-Python/assembler/rom_map.bmm
fpga_ram_bmm_input_file: /media/BELGELER/Workspaces/PyCharm-Python/assembler/ram_map.bmm
# FPGA device result bit file
fpga_modified_bit_output_file: /media/BELGELER/Workspaces/Xilinx/processor/new_memory.bit
# Xilinx Impact tool path for program FPGA
xilinx_impact_path: /opt/Xilinx/13.2/ISE_DS/ISE/bin/lin64/impact
```

Assembler has some special words for create predefined variables.

- Assembler variables can resides between "`_initial`" and "`_end`" block.
- " **$** " character assigns decimal variable.
- " **&** " character gives starting address of variable.
- " **;** "character makes afterwards a comment in a line.

Example input assembly file:

```
; I/O test asm file
_initial
    $proc_in = 4080
    $proc_out = 4081
_end

    MOV R2, #4
    MOV R3, #3
H:  CSL R3, R3, #1
    SUB R2, R2, #1
    BNE H
    ADD R1, R1, #1
    MOVL R0, $proc_out
    STR [R0], R3

F:  BAL F
```

To run assembler on command prompt:

```
$ python assembler.py configuration.yaml
```

Assembler output:

```
010   0100001000000100 4204     MOV R2 #4
011   0100001100000011 4303     MOV R3 #3
012   0001110001011011 1C5B     CSL R3 R3 #1
013   0011000001010010 3052     SUB R2 R2 #1
014   1100001111111101 C3FD     BNE H
015   0010000001001001 2049     ADD R1 R1 #1
016   1111111111110001 FFF1     MOVL R0 #4081
017   1011000000000011 B003     STR [R0] R3
018   1110011111111111 E7FF     BAL F
```

5.4. Verilog HDL Implementation

All of the previously explained units are implemented on Verilog HDL language. Same notation is preserved for processor units and Verilog modules. Their internal structure, input ports, output ports and module names are also same in Verilog implementation.

Designed processor units and their Verilog module names are:

- Processor [`processor`]
  - Organization unit [`organization_unit`]
    - I/O and data memory unit [`IO_memory`]
      - Data memory [`data_ram`]
    - Register file [`register_file`]
    - ALU [`ALU`]
  - Fetch unit [`fetch_unit`]
    - Instruction memory [`instruction_rom`]
  - Decode unit [`decode_unit`]
  - Branch control unit [`branch_control_unit`]
  - Subroutine stack [`subroutine_stack`]
  - Hazard detection unit [`hazard_detection_unit`]
  - Interrupt control unit [`interrupt_control_unit`]
  - Pipeline control registers [`pipeline_registers`]

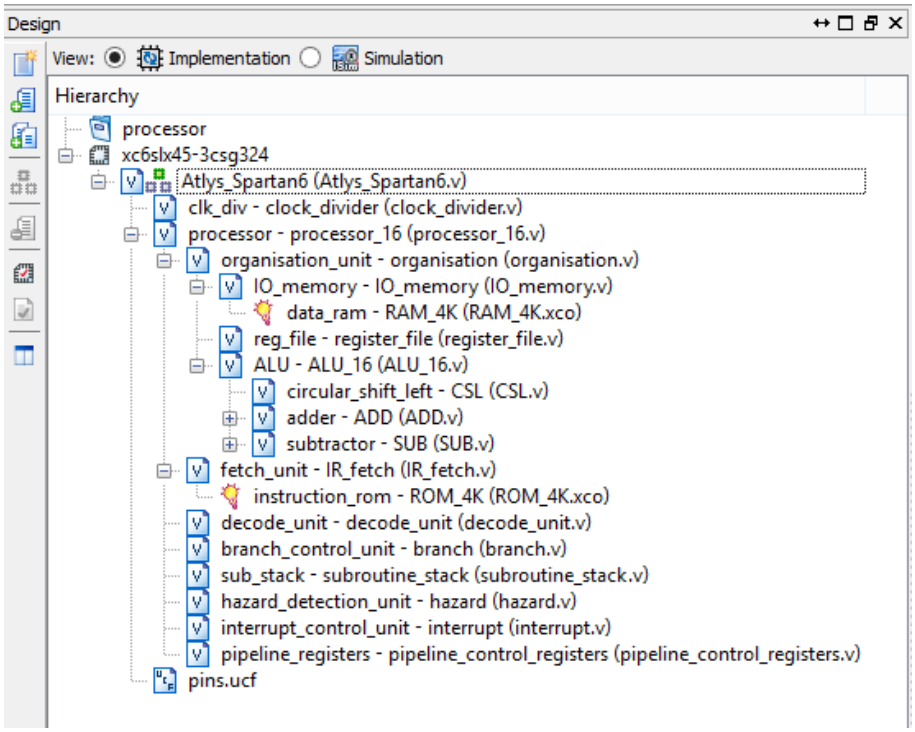

Figure 24: Module hierarchy on Xilinx ISE

5.4.1. Simulation on ISim

ISim is Xilinx FPGA's default simulation software. It has behavioral and Post-Route simulation capabilities. Behavioral simulation option uses Verilog codes to simulate processor. Post-Route simulation option uses synthesis output to simulate processor and it gives actual logic results as like FPGA run.

ISim simulation tool can use memory initialization file for instruction and data memory. This gives a chance for simulate different assembly files without synthesis.

Variable names and their timing diagrams can be viewed in ISim simulation run.

Example run for "`test_processor`" testbench is in figure 25. This simulation uses previous assembly file given in section 5.3. Program shifts number 3 by 4 times and gives it to processor output LEDs. Result will be 0030 in hexadecimal, 00110000 in binary notation.

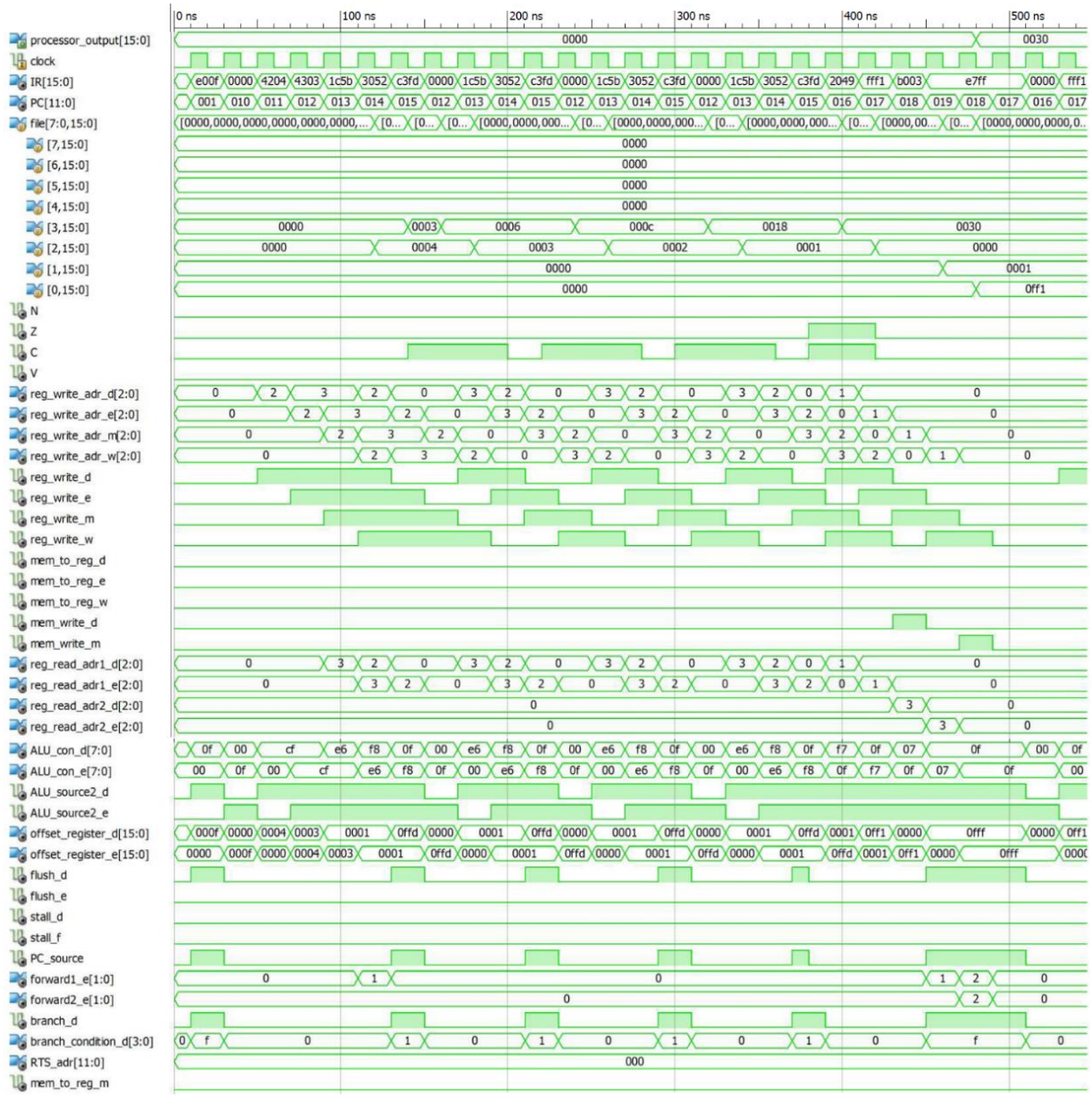

Figure 25: Timing diagram of processor's signals on ISim

5.4.2. Implementation on FPGA

DIGILENT ATLYS development board has Xilinx Spartan-6 XC6SLX45 FPGA chip which has [3]:

- 6,822 slices, each containing four 6-input LUTs and eight flip-flops
- 2,088 Kb block ram capacity
- 218 I/O pins, some pins hardwired to LEDs, switches, and etc.

These features are more than enough for implementation of the processor. Processor uses only %02 of logic cells on FPGA.

FPGA can run processor with at most 50MHz clock frequency.

Xilinx ISE synthesis place and route report:

```
Device Utilization Summary:

Slice Logic Utilization:

  Number of Slice Registers:                 415 out of  54,576    1%

    Number used as Flip Flops:               301

    Number used as Latches:                  114

  Number of Slice LUTs:                      623 out of  27,288    2%

    Number used as logic:                    579 out of  27,288    2%

      Number using O6 output only:           453

      Number using O5 output only:            31

      Number using O5 and O6:                 95

    Number used as Memory:                    25 out of   6,408    1%

      Number used as Dual Port RAM:           24

        Number using O5 and O6:               24

      Number used as Single Port RAM:          0
```

XC6SLX45 has 116 16Kb block ram. Instruction and data memories have 16 bit data length and 12 bit address depth. So each memory needs total of $16 * 2^{12} = 64$ Kb.

Each memory generated with combination of 4 16Kb FPGA block ram. Total 8 16Kb FPGA block ram is used.

6. EXPERIMENT RESULTS

Synthesis of processor takes approximately one minute on Xilinx ISE. Programming FPGA takes about 8 seconds.

Example program:

```
; I/O test asm file
_initial
    $proc_out = 4081
_end
    MOV R2, #4
    MOV R3, #3
H:  CSL R3, R3, #1
    SUB R2, R2, #1
    BNE H
    ADD R1, R1, #1
    MOVL R0, $proc_out
    STR [R0], R3
F:  BAL F
```

After successful synthesis and successful program of FPGA is shown in figure 26. Example program is used for demonstration. Program shifts number 3 by 4 times and gives it to processor output LEDs. Result will be 0030 in hexadecimal, 00110000 in binary notation.

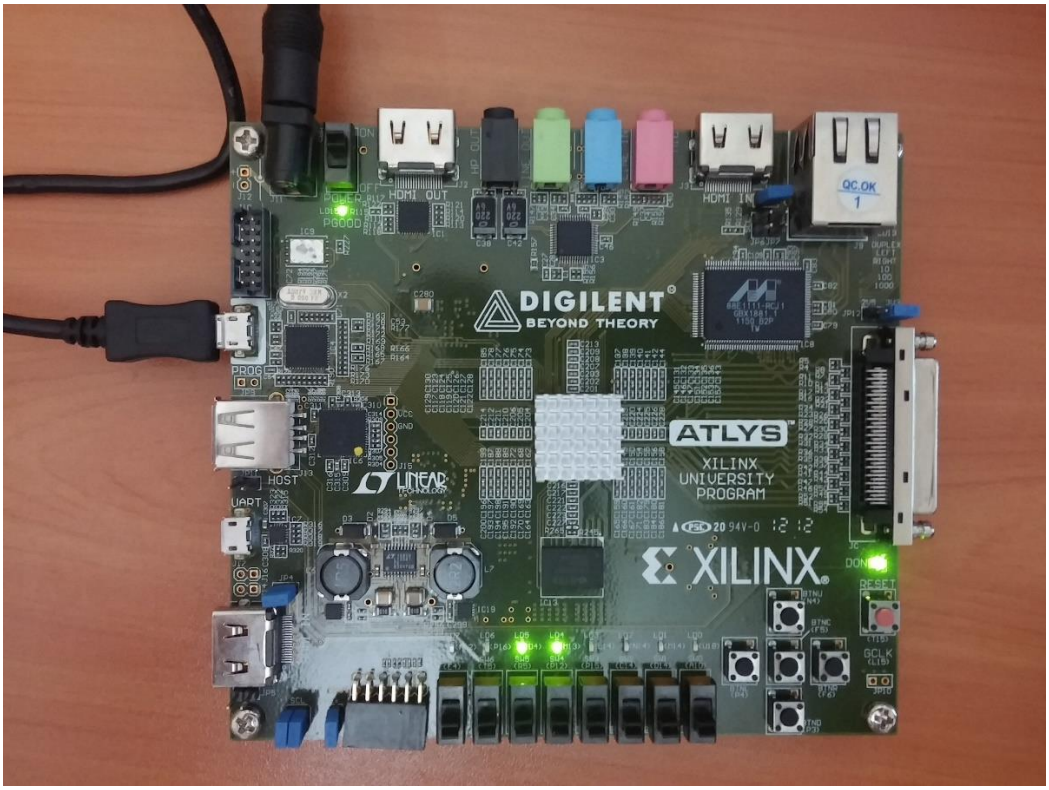

Figure 26: DIGILENT ATLYS Spartan-6 FPGA development board example run

With the help of pipeline, processor can achieve instruction level parallelism. Nominal cycles per instruction (CPI) is 1. Without pipelining CPI would be 5. Penalty for stall and flush instructions is 1 clock cycle. Pipeline stall and flush operations increase cycle per instruction. But in long execution run, CPI converges to 1.

$$\text{CPI} \cong 1$$

FPGA can run processor with max 50MHz clock frequency.

$$\text{Clock frequency} = 50000000$$

Processor performance can be calculated as [7]:

$$\text{Effective processor performance} = \frac{\text{Clock frequency}}{\text{CPI} \times 1000000} = \frac{50000000}{1 \times 1000000} \cong 50 \text{ MIPS}$$

Execution time for a given instruction count can be calculated as [7]:

$$\text{Execution time } (T) = \frac{\text{CPI} \times \text{Instruction count}}{\text{Clock frequency}}$$

Instruction count for example program is 25. Execution time for example program can be calculated as:

$$\text{Execution time } (T) = \frac{1 \times 25}{50000000} \cong 0.5 \text{ ms}$$

7. CONCLUSION AND PROPOSALS

7.1. Educational Attainment

In this project a RISC processor is implemented with a newly designed architecture and instruction set. Commercial processor architectures were examined and mixture of their design inspired the new architecture. Learning, design & working mechanism of a processor is main aim of this project. Final result of this project is a soft processor which is able to execute instructions like fully functional standard microprocessor. After all whole project, detailed experience is gained about computer architecture, computer organization, processor architecture, micro architecture, instruction set architecture, FPGA, Verilog HDL, assembly language parsing and Python programing language topics.

7.2. Conclusion

FPGA is very effective to design and test new processor architectures. Design and implementation of a new processor is a time consuming and costly operation. FPGA increases development time and cost.

The processor itself is a very efficient, it has simple design and implementation. It uses very little FPGA resource. Also it can execute instructions with complete correctness. Programmer doesn't need to change the instruction order or programmer doesn't need to insert stall instructions for correctness of execution. The processor itself provides correctness of execution, programmers doesn't need to worry about it. Processor's pipeline utilizes approximately one instruction per cycle. Correctness of execution is the most important requirements for the processor overall. And processor meets this requirements successfully. It can execute any desired operation to be expected from a standard processor.

7.3. Proposals

Processor achieve fair performance for average size program. Memory access pipeline stalls can be decrease by changing order of instructions. Processor always predicts branch to not take so there is not much that can be done about branch penalty. The processor overcome a lot of problem but still it also lacks a lot of feature. Processor can be improved by adding branch prediction mechanism, dynamic ram support, cache memory mechanism. Processor even can be make a multi core. But this improvements would increase complexity of processor. As always there is a tradeoff between development and complexity. The processor is more suitable for general purpose simple I/O control applications.

8. REFERENCES

[1] Altera Inc. (2015). *Nios II Classic Processor Reference Guide.* Retrieved from https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/nios2/n2cpu_nii5v1.pdf

[2] Berger, A. S. (2005). *Hardware and Computer Organization: The Software Perspective.* Burlington: Elsevier Inc.

[3] Digilent Inc. (2013). *Atlys Board Reference Manual.* Retrieved from http://www.digilentinc.com/data/products/atlys/atlys_rm_v2.pdf

[4] Flynn, M. J. (1995). *Computer architecture: Pipelined and parallel processor design.* Boston, MA: Jones and Bartlett.

[5] Harris, D. & Harris, S. (2012). *Digital Design and Computer Architecture* (2nd ed.). San Francisco: Morgan Kaufmann Publishers Inc.

[6] Hennessy, J. L. & Patterson, D. A. (2011). *Computer Architecture: A Quantitative Approach* (5th ed.). San Francisco: Morgan Kaufmann Publishers Inc.

[7] Hennessy, J. L., & Patterson, D. A. (1998). *Computer organization and design: The hardware/software interface.* San Francisco, Calif: Morgan Kaufmann Publishers.

[8] Lawlor, O. (2014). *CS 441: System Architecture* [Lecture Note]. Retrieved from UAF Computer Science Department website: https://www.cs.uaf.edu/courses/cs441/notes/encoding/

[9] The McGraw-Hill Companies Inc. (2009). *Schaum's Outline of Theory and Problems of Computer Architecture*. Retrieved from http://www.dauniv.ac.in/downloads/CArch_PPTs/CompArchCh04L07InstructionSetFeatures.pdf

[10] Microprocessor Design. (n.d.). In *Wikibooks*. Retrieved from http://en.wikibooks.org/wiki/Microprocessor_Design/Print_Version

[11] University of Waterloo. (2012). *THUMB Instruction Set*. Retrieved from University of Waterloo website: https://ece.uwaterloo.ca/~ece222/ARM/ARM7-TDMI-manual-pt3.pdf

[12] Weatherspoon, H. (2011). *CS 3410: Computer System Organization and Programming* [Lecture Slides]. Retrieved from Cornell University website: http://www.cs.cornell.edu/courses/CS3410/2011sp/schedule.html

[13] Xilinx Inc. (2011). *MicroBlaze Processor Reference Guide.* Retrieved from http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_2/mb_ref_guide.pdf