Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

This chapter starts with motivation where area of problems and literature review are discussed. Then in proposed methodology section, an methodology is proposed to solve the problem. Then, in thesis contribution, contributions of thesis are outlined. Finally, chapter ends with conclusion.

1.2 Motivation

Computer Science is a highly technical and practical field. As each individual discipline in computer science grows rapidly, their courses has become more broader and specialized in scope and complexity. Since only small portion of individual discipline can be taught within short time-span of undergraduate studies, these courses focus more on theoretical foundations with many higher levels of abstraction [1]. As a result, experiments associated with these courses focus on experiments related to individual courses and are scattered from other courses [2].

Several courses are offered focusing on computer system design like digital electronics, computer architecture, interfacing, compiler etc. An important purpose of these courses is to provide understanding on how computer works as a whole. These courses teach details of each level of abstraction well but they fail to teach how they are interconnected with each other in computer system as a whole [3, 4].

Several attempts have been made to address these problems in literature as follows:

1. Reference [5, 6, 7] introduced an tiny computer system which combines processor, assembler, compiler and interfacing. But it does not follow building block approach which is building system by combining previous building blocks. Processor uses stack architecture which is very simple.

2. Reference [3, 8] introduced software based computer system. Computer system can be developed using building block approach. But it uses simple HDL other than real-life HDL. Its processor is simple.

3. Reference [9, 10] combines digital electronics and computer architecture but its relationship with assembler, compiler etc are ignored.

4. Other references like [11, 12, 13, 14] etc. generally focuses on computer architecture. Some of them focuses on processor only. Some of them focuses on advanced processor.

[2] suggested to make practical associated with individual courses interconnected together by building computer system from scratch. It will include digital electronics, computer architecture, assembler, compiler and operating system. It will help students understand concepts of computer systems and their interconnection in computer system very well.

1.3 Proposed Methodology

Computer system can be built which can combine digital electronics, computer architecture, assembler and interfacing together. It not only focuses on individual courses but also focuses on inter-relationship of these courses within computer system. Following figure shows the proposed methodology:

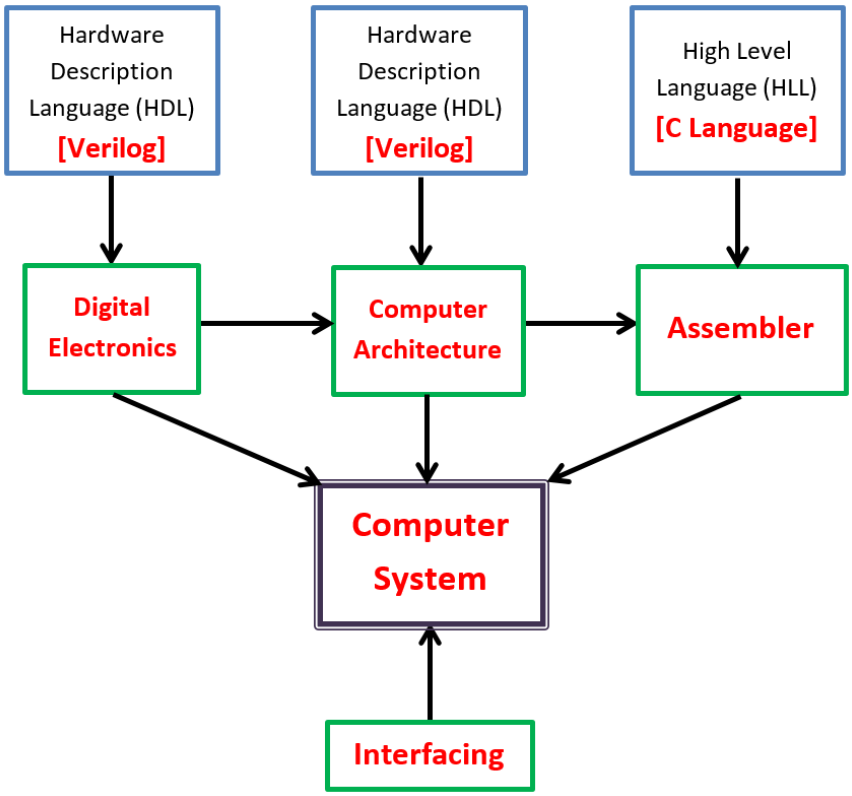


Figure 1.1: Proposed Methodology

As shown in figure, first digital electronics building blocks can be implemented. These building blocks can be used to implement computer architecture building blocks. Computer architecture building blocks can be combined to implement central processing unit (CPU). Then assembler can be built based on computer's instruction set architecture (ISA). Finally, computer can be interfaced with output devices to display output of computer.

1.4 Thesis Contributions

The main contribution of this thesis is the computer system which will combine four individual concepts of computer science and engineering together. It focuses on building simpler com-

puter system with importance given on combining those concepts rather on individual concepts themselves. This computer system can be used by students for educational purpose as reference. Students can develop this computer system from scratch. This will help them understand concepts of computer system in more better way. To make computer system more education friendly, computer system has following features:

1. **Building block approach:** Implement building block for the next design and next design will use it to implement building block for next design and so on.
2. **HDL:** Use Verilog for hand on experience on digital system design.
3. **Processor:** Implement processor with a complete instruction set which is simpler than MIPS.
4. **Assembler:** Implement assembler using any high level language to make it easier to program.
5. **Simulation:** Simulate processor in a FPGA and software.
6. **Interfacing:** Interface processor with external devices to understand how processor interact with external devices.
7. **Specialty:** Design a computer system suitable for education which is not introductory or more advanced.

1.5 Why using FPGA?

Field programmable gate array (FPGA) is a fully fabricated IC chip having thousands of logic gates in which the interconnections can be programmed to implement different functions. It has three components: I/O block, configurable logic block (CLB) and routing channel. FPGA architecture is shown in figure:

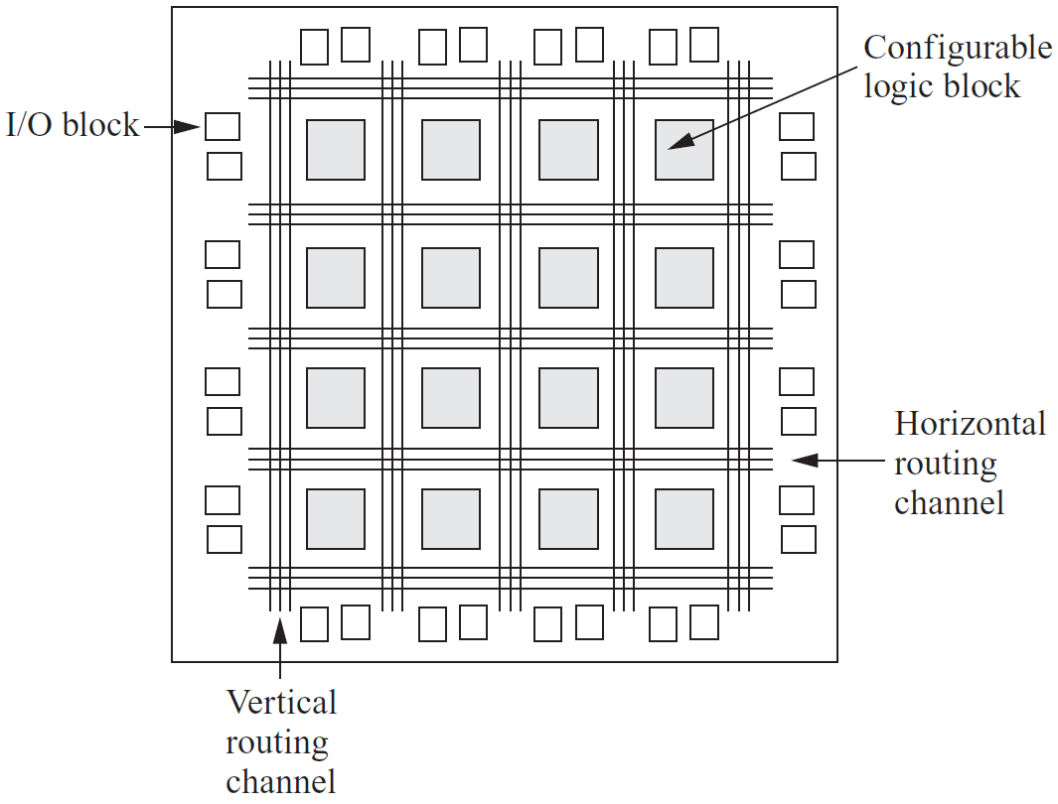


Figure 1.2: FPGA architecture

FPGA has following advantages:

1. Implement random logic and re-configurable hardware
2. Suitable for faster Prototyping
3. Implement fast logic for parallel operation
4. Implement customized designs

1.6 Thesis Organization

The rest of the thesis is organized as follows:

Chapter 2 - Digital Electronics Building Block

This chapter describes digital electronics building blocks which can be built by logical gates. It includes combinational circuits like adder, multiplier etc. and sequential circuits like D flip-flop, register etc.

Chapter 3 - Computer Architecture Building Block

This chapter first describes instruction set architecture (ISA) of computer itself. Then computer architecture building blocks are described which can be built by combining digital electronics building block developed in previous chapter. It includes arithmetic and logic unit (ALU), register file, control unit (CU) and instruction memory. Finally, computer architecture building blocks are combined to build central processing unit (CPU).

Chapter 4 - Interfacing and Assembler

This chapter describes interfacing of computer with LED, seven segment display and VGA monitor. Then assembler is described which is built on the top of computer to make programming task easier.

Chapter 5 - Verification, Simulation on FPGA and Result Analysis

This chapter shows simulation of assembler and computer on software for verification. Then, it shows simulation of computer system on fpga. Finally, the results are analyzed.

Chapter 6 - Conclusion and Future works

This chapter concludes the thesis and shows direction of future work.

1.7 Conclusion

This chapter discussed motivation behind thesis, proposed methodology with its contribution, importance of FPGA and finally thesis organization. Next chapter will discuss digital electronics building blocks.

Chapter 2

Digital Electronics Building Block

2.1 Introduction

This chapter focuses on digital electronics building blocks. This chapter begins with digital circuits. Then two types of digital circuits, combinational and sequential circuits with their respective components are discussed. Finally, chapter ends with conclusion.

2.2 Digital Circuits

Digital electronics building blocks are built by using logic gates. These blocks can be written in Verilog. To make block design easier, dataflow and behavioral style can also be used. There are two types of digital circuits that are used in computer. They are: combinational and sequential circuits. These circuits can be found in any traditional digital electronics books (like [10, 9]).

2.3 Combinational Circuit

Combinational circuits are mainly used for arithmetic, datapath selection operations etc.

2.3.1 Adder/Subtracter

Adder/Subtracter is used to perform add or subtract operation in computer. In this computer design, specification of adder/subtracter is:

1. **Input:** 32-bit a (input 1)
2. **Input:** 32-bit b (input 2)

3. **Input:** 1-bit addorsub [if addorsub == 1 then a - b = a + (-b) = a + not(b) + 1 = a + (b xor addorsub) + addorsub] [if addorsub == 0 then a + b = a + b = a + b + 0 = a + (b xor addorsub) + addorsub]
4. **Output:** 32-bit s (result)

Example of 4-bit carry look ahead adder is shown below.

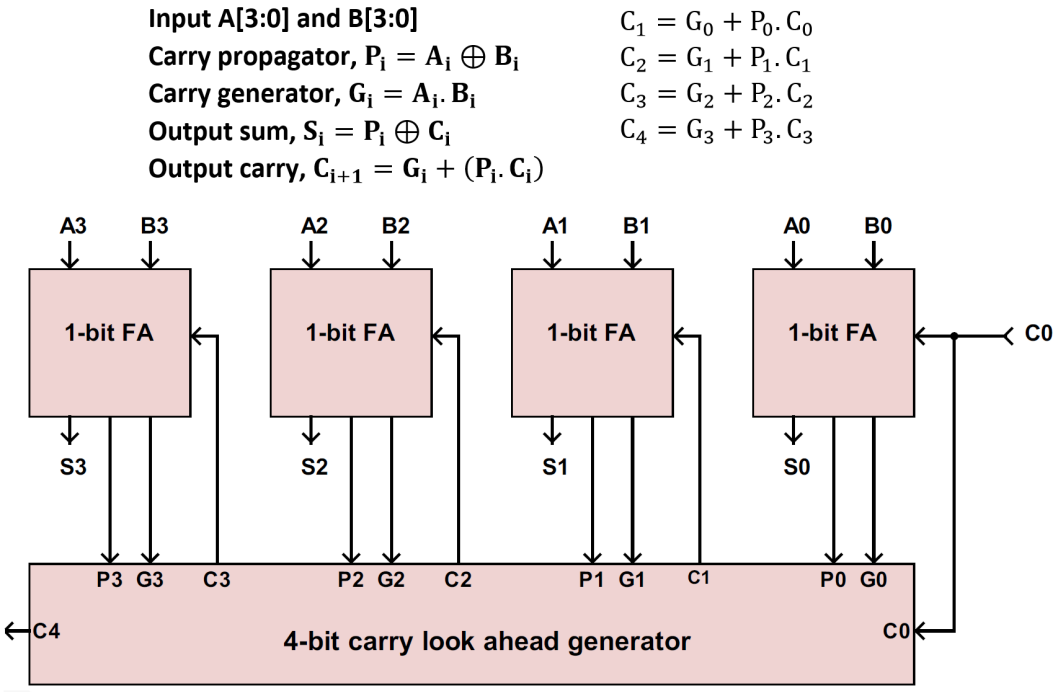


Figure 2.1: 4-bit carry look ahead adder

As shown in figure, adder takes A[3:0], B[3:0] and C[0] as input and produces S[3:0] and C[4] as output. Each input A[i] and B[i] go through 1-bit full adder to produce sum S[i]. Logic diagram of full adder [10] is shown in figure.

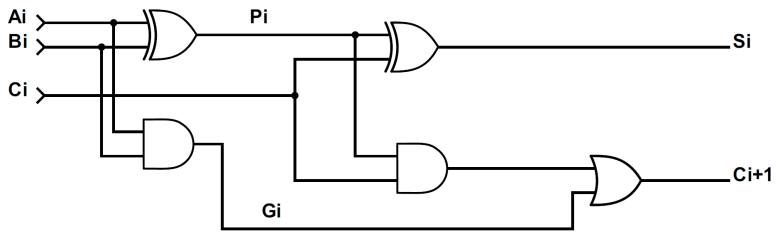


Figure 2.2: Logic diagram of full adder

Since all the carry C[3:0] will be generated at the same time, C[4] will not have to wait for C[3] and C[2] to propagate. Logic diagram of a look ahead carry generator [10] is shown in figure.

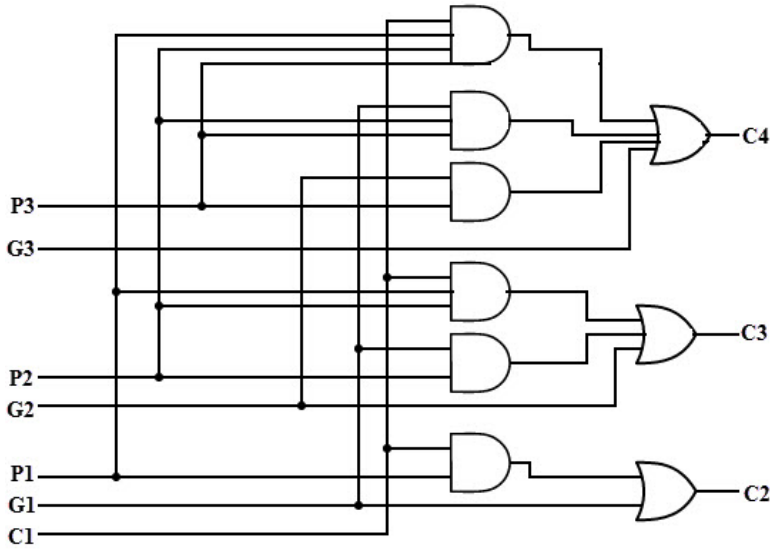


Figure 2.3: Logic diagram of 4-bit carry look ahead adder

2.3.2 Multiplier

Multiplier is used to perform multiplication operation in computer. In this computer design, specification of multiplier is:

1. **Input:** 16-bit a (input 1)
2. **Input:** 16-bit b (input 2)
3. **Output:** 32-bit r (result)

Example of multiplier using partial product accumulation method is described below. Multiplier takes A[3:0] and B[3:0] as input where A is multiplicand and B is multiplier. The multiplication of A and B is shown in figure. Each ANDed term is a partial product. The resulting product is accumulated as shown in figure.

				A3	A2	A1	A0
				B3	B2	B1	B0
				A2 B0	A2 B0	A1 B0	A0 B0
			A3 B1	A2 B1	A1 B1	A0 B1	
		A3 B2	A2 B2	A1 B2	A0 B2		
	A3 B3	A2 B3	A1 B3	A0 B3			
S7	S6	S5	S4	S3	S2	S1	S0

Figure 2.4: Product accumulation

The building block of multiplier [15] is shown in figure. Building block takes previous sum (Sum In), partial product X & Y and carry-in (Cin) as input. These inputs go through AND gate and full adder. Finally, building block produces carry-out (Cout) and next sum (Sum Out).

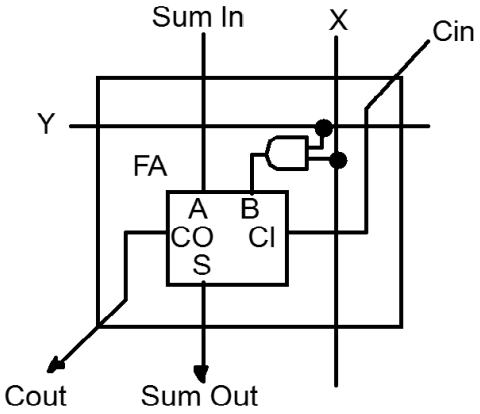


Figure 2.5: Building block of multiplier

4-by-4 combinational multiplier [15] is shown in figure.

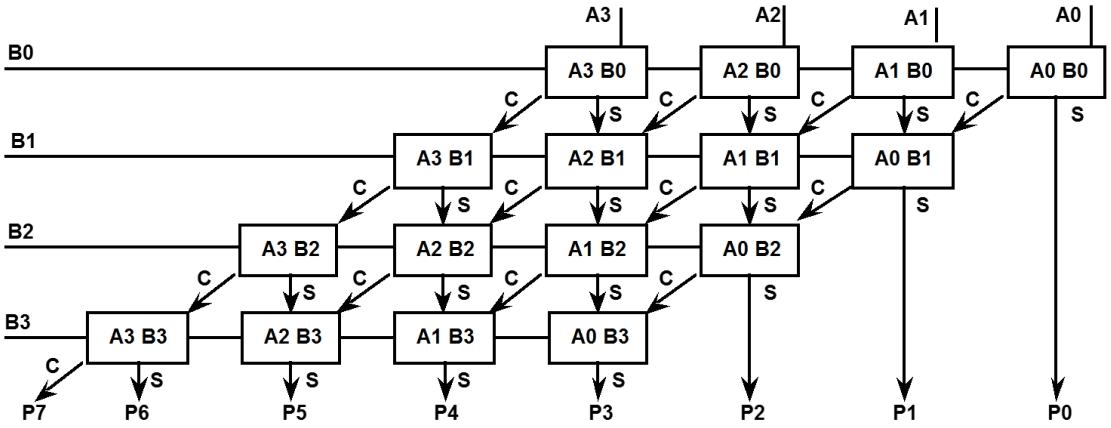


Figure 2.6: Multiplier

2.3.3 Shifter

Shifter is used to perform shifting operation on data in computer. In this computer design, specification of shifter is:

1. **Input:** 32-bit d (data)
2. **Input:** 2-bit sa (shift amount)
3. **Input:** 1-bit r (shift right) [shift right:1 and shift left:0]
4. **Input:** 1-bit a (arithmetic shift) [arithmetic:1 and logical:0]
5. **Output:** 32-bit sh (shifted data)

Example of 4-bit shifter [16] is shown below:

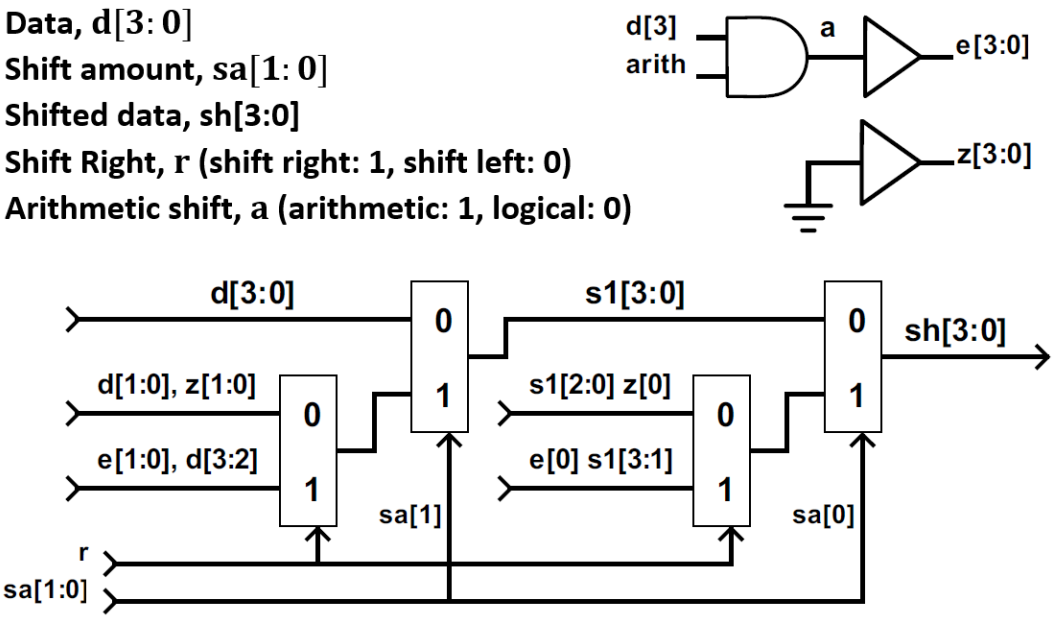


Figure 2.7: 4-bit shifter

As shown in figure, If arith is a 0 (logical shift), e will be zero which is generated by connecting each bit of e to ground. If arith is a 1 (arithmetic shift), every bit of e is equal to the sign bit of d which is generated by an AND gate. Two 2-to-1 multiplexers are used to select path depending on right or left shift bit, r. Other 2-to-1 multiplexers are used to select path depending on shift amount, sa.

2.3.4 Multiplexer

Multiplexer is used to select data path in computer. In this computer design, two types of multiplexer are used. They are: 2-to-1 multiplexer and 4-to-1 multiplexer. Specification of 2-to-1 multiplexer is:

1. **Input:** 32-bit d0 (data 0)
2. **Input:** 32-bit d1 (data 1)
3. **Input:** 1-bit s (select bit)
4. **Output:** 32-bit y (output)

Specification of 4-to-1 multiplexer is:

1. **Input:** 32-bit d0 (data 0)
2. **Input:** 32-bit d1 (data 1)
3. **Input:** 32-bit d2 (data 2)
4. **Input:** 32-bit d3 (data 3)
5. **Input:** 2-bit s (select bit)
6. **Output:** 32-bit y (output)

Example of 2-to-1 multiplexer is shown below:

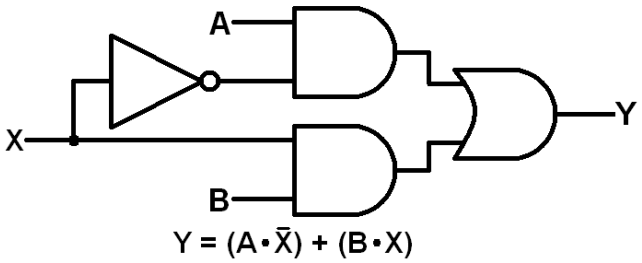


Figure 2.8: 2-to-1 multiplexer

As shown in figure, 2-to-1 multiplexer takes A and B as input and X as select bit. Depending on X, it connects A or B with Y.

2.3.5 Decoder

Decoder is used to decode input and to enable one of outputs depending on input in computer. In this computer design, specification of 5-to-32 line decoder is:

1. **Input:** 5-bit a (input)
2. **Input:** 1-bit e (master enable)
3. **Output:** 32-bit y (output)

Example of 2-to-4 line decoder is shown below:

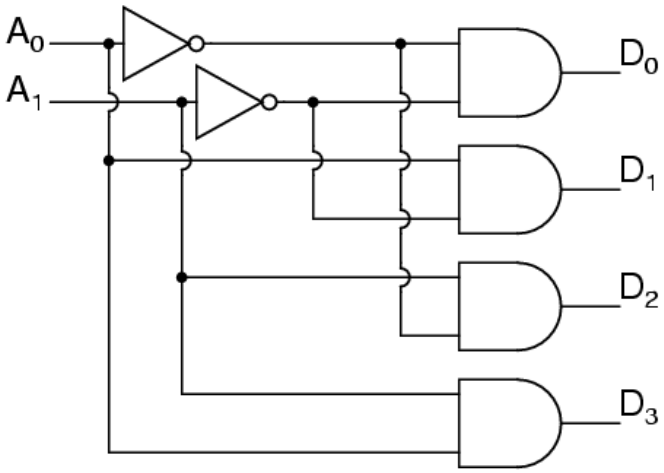


Figure 2.9: 2-to-4 line decoder

As shown in figure, 2-to-4 line decoder takes takes A[1:0] as input. Depending on A, One of y[3:0] outputs will be active High.

2.4 Sequential Circuit

Sequential circuits are mainly used to store data temporarily.

2.4.1 D Flip-flop

D flip-flop is used as a basic memory cell to store data in computer. In this computer design, specification of D flip-flop is:

1. **Input:** 1-bit d (data)
2. **Input:** 1-bit en (enable)
3. **Input:** 1-bit c (clock input)
4. **Input:** 1-bit clrn (clear)
5. **Output:** 1-bit q (output)

Example of D Flip-flop [10] is shown below:

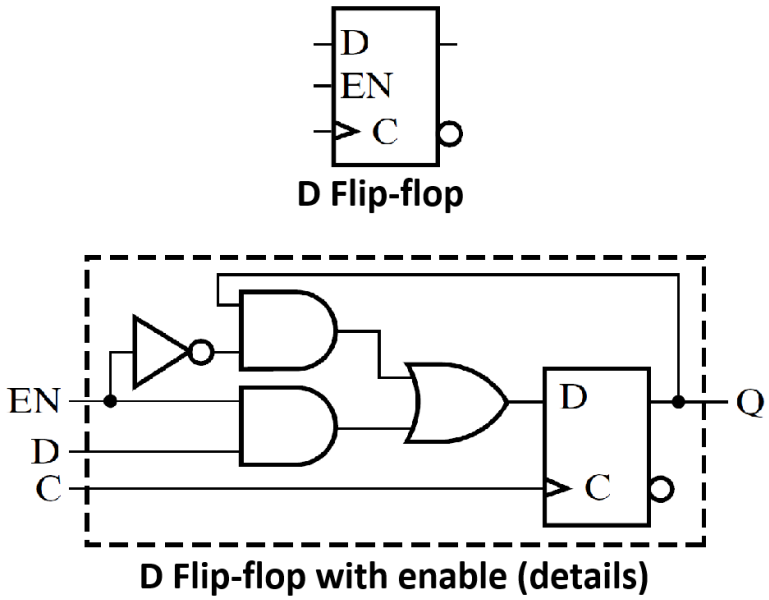


Figure 2.10: D Flip-flop with its detailed view

As shown in figure, D flip-flop takes 1-bit data (d), enable (en), clock input (c) and clear bit (clrn) as inputs. It stores the value d when clock is on positive edge. If clear bit is on, then it will be reset. It generates q which is data stored in D flip-flop.

2.4.2 Register

Register is a combination of D flip-flops used to store data temporarily. In this computer design, specification of 32-bit register is:

1. **Input:** 32-bit d (data)
2. **Input:** 1-bit load (enable)
3. **Input:** 1-bit c (clock input)
4. **Input:** 1-bit clrn (clear)
5. **Output:** 32-bit q (output)

Example of D Flip-flop and register [10] are shown below:

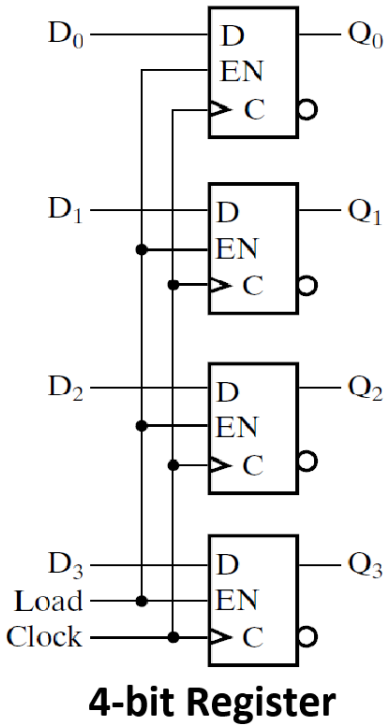


Figure 2.11: 4-bit register

As shown in figure, 4-bit register can be formed by connecting 4 D flip-flop in parallel. It takes 4-bit data d[3:0], enable load and clock as inputs. It stores 4-bit data when clock is on positive edge and load is enabled. It produces its stored data as output 4-bit q.

2.5 Conclusion

This chapter explained all necessary digital electronics building blocks needed to implement computer. These blocks can be described by using Verilog language. Necessary testing should be performed to verify those blocks. Next chapter will combine all these blocks to implement computer architecture building blocks.

Chapter 3

Computer Architecture Building Block

3.1 Introduction

This chapter discusses computer architecture building blocks. This chapter starts with instruction set architecture (ISA) of computer. Then, computer architecture blocks will be implemented from digital architecture blocks implemented in previous chapter. Finally, conclusion will end the chapter.

3.2 Instruction Set Architecture (ISA)

Instruction Set Architecture (ISA) is the contact point where software and hardware meet together. It provides interface to software to connect with hardware.

3.2.1 Instruction Format

Size of instruction opcode is 6-bit. First 2-bit determines type of instruction and last 4-bit determines the operation. Computer has three types of instruction format. They are:

1. **Register Instruction:** First 2-bit (30 to 31) of 6-bit opcode (26 to 31) in register instruction is 00. Last 4-bit (26 to 29) of 6-bit opcode determines operation to be performed. It performs operation on registers **Ra** (16 to 20) and **Rb** (11 to 15) and stores the result in register **Rc** (21 to 25).

Register Instruction

31 26	25 21	20 16	15 11	10 0
00XXXX	Rc	Ra	Rb	Unused

Figure 3.1: Register instruction format

2. **Immediate Instruction:** First 2-bit (30 to 31) of 6-bit opcode (26 to 31) in immediate instruction is 01. Last 4-bit (26 to 29) of 6-bit opcode determines operation to be performed. It performs operation on registers **Ra** (16 to 20) and immediate value, **immediate** (0 to 15) and stores the result in register Rc (21 to 25).

Immediate Instruction

31 26	25 21	20 16	15 0
01XXXX	Rc	Ra	immediate

Figure 3.2: Immediate instruction format

3. **Jump Instruction:** First 2-bit (30 to 31) of 6-bit opcode (26 to 31) in jump instruction is 11. Last 4-bit (26 to 29) of 6-bit opcode determines operation to be performed. It sets program counter to address **addr** (18 to 25).

Jump Instruction

31 26	25 18	17 0
11XXXX	addr	unused

Figure 3.3: Jump instruction format

3.2.2 Instruction Types

There are three types of instruction based on operations. They are:

1. **Arithmetic Operation:** It includes **addition**, **subtraction** and **multiplication** operation in both register and immediate instruction format.
2. **Logical Operation:** It includes **AND**, **OR**, **XOR**, **shift logical left**, **shift logical right**, **shift arithmetic right** and **compare** operation in both register and immediate instruction format.
3. **Transfer of control:** It includes **jump**, **jump if equal**, **jump if not equal**, **jump if less**, **jump if less or equal**, **jump if greater**, **jump if greater or equal**, **jump if carry**, **jump if not carry**, **jump if zero** and **jump if not zero** operation in jump instruction format.

Following table shows list of instructions.

Table 3.1: List of instructions

Type of operation	Operation Name	Register Instruction	Immediate Instruction	Jump Instruction
Arithmetic	Addition	ADD	ADDI	-
	Subtraction	SUB	SUBI	-
	Multiplication	MUL	MULI	-
Logical	AND	AND	ANDI	-
	OR	OR	ORI	-
	XOR	XOR	XORI	-
	Shift logical left	SHL	SHLI	-
	Shift logical right	SHR	SHRI	-
	Shift arithmetic right	SAR	SARI	-
	Compare	CMP	CMPI	-
Transfer of control	Jump	-	-	JMP
	Jump if equal	-	-	JE
	Jump if not equal	-	-	JNE
	Jump if less	-	-	JL
	Jump if less or equal	-	-	JLE
	Jump if greater	-	-	JG
	Jump if greater or equal	-	-	JGE
	Jump if carry	-	-	JC
	Jump if not carry	-	-	JNC
	Jump if zero	-	-	JZ
	Jump if not zero	-	-	JNZ

3.3 Computer Architecture Blocks

There are six computer architecture blocks implemented from digital electronics building block. They are: arithmetic and logic unit (ALU), register file, control unit (CU), instruction memory, flag register and program counter. Finally, these blocks are combined to create central processing unit (CPU).

3.3.1 Arithmetic and Logic Unit (ALU)

Arithmetic and logic unit (ALU) is implemented from Adder/Subtracter, multipler, barrel shifter, 2-to-1 multiplexer and 4-to-1 multiplexer digital electronics building blocks. ALU takes **a** and **b** as input and performs operations on them based on **op**. It produces result,**r** as output. It also produces zero flag (**zf**), sign flag (**sf**) and carry flag (**cf**). ALU is shown in figure.

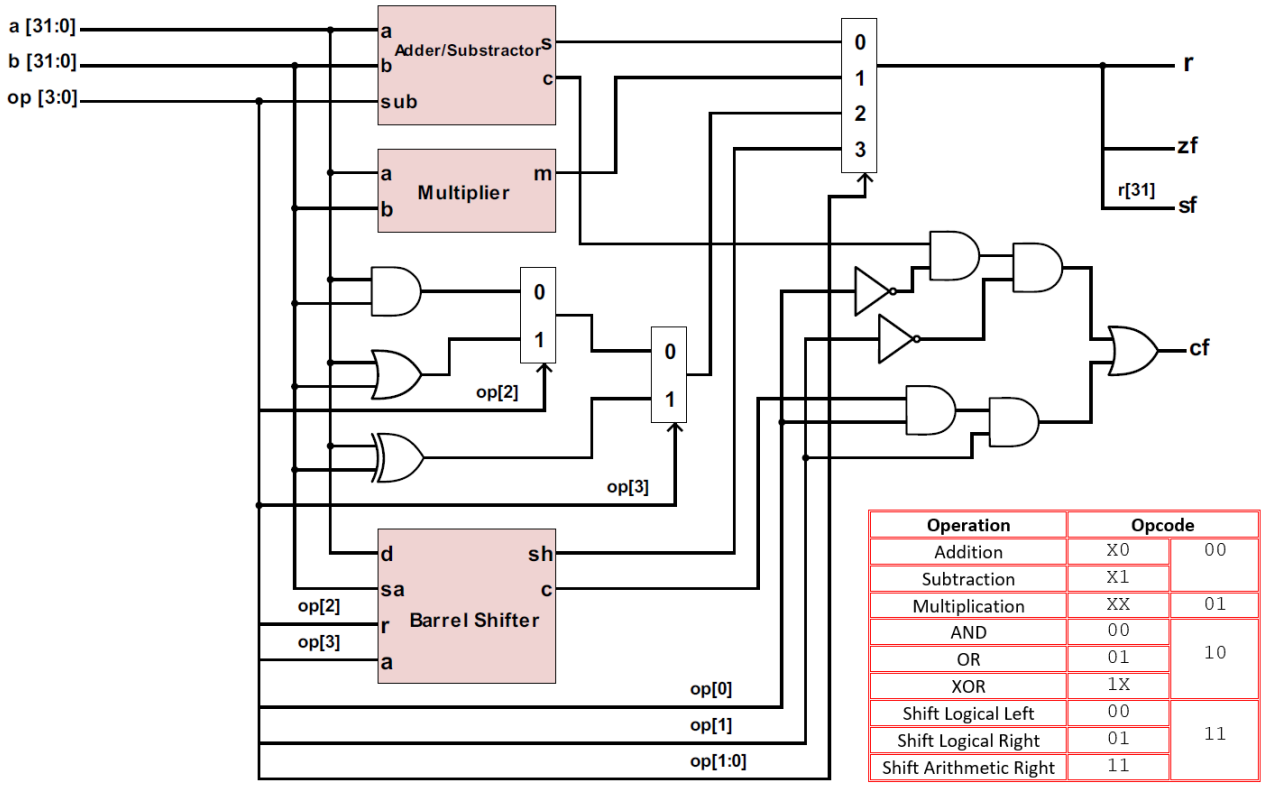


Figure 3.4: Arithmetic and logic unit (ALU)

op[1:0] is used to select one of the 4 paths in 4-to-1 multiplexer to r. op[3:2] is used to select various operations within op[1:0]. There are four types of operation based on op[1:0]. They are:

1. **00:** It has addition and subtraction operations.
    - **Addition (X000):** For addition operation, op[3] is don't care (X) and op[2] must be 0.
    - **Subtraction (X100):** For subtraction operation, op[3] is don't care (X) and op[2] must be 1.
2. **01:** It has only multiplication operation.
    - **Multiplication (XX01):** For multiplication operation, op[3] is don't care (X) and op[2] is don't care (X).

3. **10:** It has AND, OR and XOR operations. op[2] is used in 2-to-1 multiplexer to select between AND and OR operation. op[3] is used in 2-to-1 multiplexer to select between XOR and operation selected by op[2].

    - **AND (0010):** For AND operation, op[3] must be 0 and op[2] must be 0.
    - **OR (0110):** For OR operation, op[3] must be 0 and op[2] must be 1.
    - **XOR (1X10):** For XOR operation, op[3] must be 1 and op[2] is don't care (X).

4. **11:** It has shift logical left, shift logical right and shift arithmetic left operations. op[2] is used as input r to barrel shifter. op[3] is used as input a to barrel shifter.

    - **Shift Logical Left (0011):** For shift logical left operation, op[3] must be 0 and op[2] must be 0.
    - **Shift Logical Right (0111):** For shift logical right operation, op[3] must be 0 and op[2] must be 1.
    - **Shift Arithmetic Left (1111):** For shift arithmetic left operation, op[3] must be 1 and op[2] must be 1.

It produces three flag outputs. They are:

1. **zero flag (zf):** It is active High when result, **r** is zero.
2. **sign flag (sf):** It is active High when most significant bit (MSB) or sign bit of result **r** is 1.
3. **carry flag (cf):** It is active High when c of adder/subtracter is 1 and op[1:0]=00 or c of barrel shifter is 1 and op[1:0]=11.

3.3.2 Register File

Register file can be implemented from decoder, register and multiplexers implemented in previous chapter. Example of register file [10] is shown below:

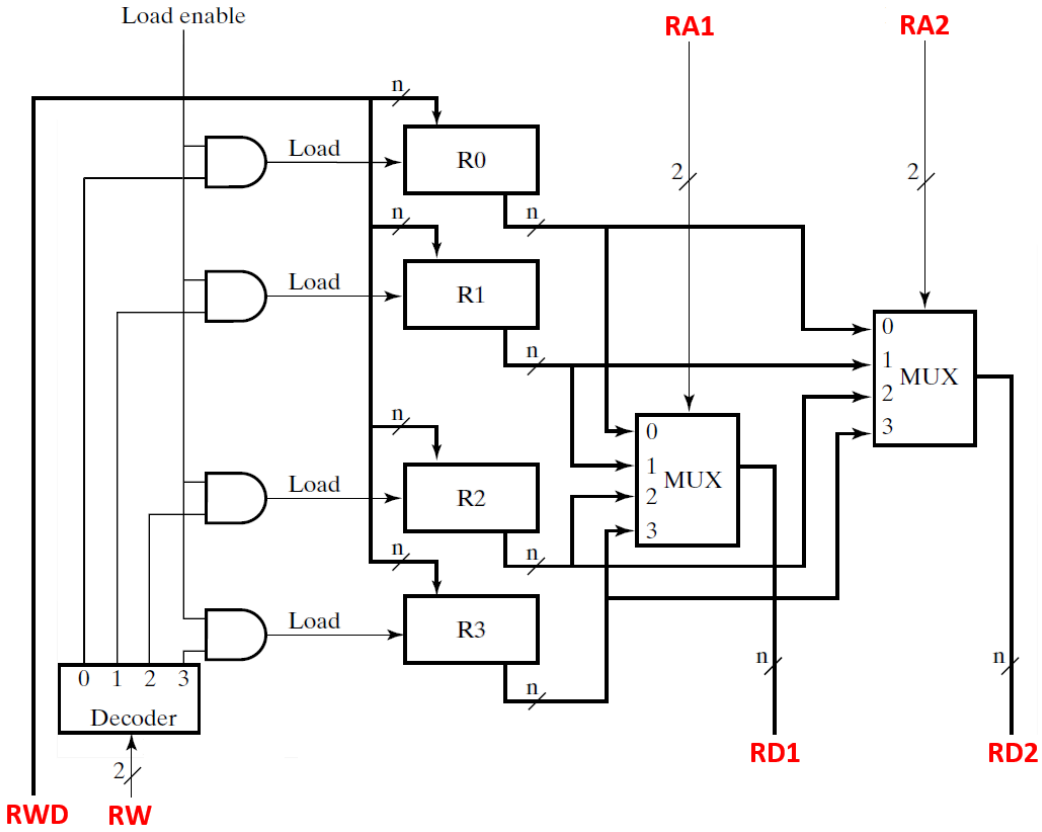


Figure 3.5: Register file

In this design, for write operation, **RW** contains address of register to be written and **RWD** contains data to be written. **RW** is used as an input to 2-to-4 line decoder and enables of one of register to be written. Finally, **RWD** is written to register enabled.

For read operation, **RA1** is the address of first register to be read and **RD1** will hold data of first register to be read. **RA1** is used as input to select register in 4-to-1 multiplexer which connects selected register to **RD1**. Similarly, **RA2** is the address of second register to be read and **RD2** will hold data of second register.

In original computer design, **R0** register is connected to ground. It means it only holds 0 all time. It is done to make implementation of pseudo-instruction like **MOV** easier.

3.3.3 Control Unit (CU)

Control unit (CU) is implemented as combinational circuit. It takes opcode **op[5:0]** and flag values **zf**, **sf** & **cf** as input and produces active High on **wreg**, **imm_select** and **jmp_select** as output.

Combinational logic of control unit (CU) is shown in table below:

Table 3.2: Combinational logic of control unit

	op[5]	op[4]	op[3]	op[2]	op[1]	op[0]	zf	sf	cf		wreg	imm_ select	jmp_ select
Register Instruction													
ADD	0	0	0	0	0	0	X	X	X		1	0	0
SUB			0	0	0	1							
MUL			0	0	1	0							
AND			0	0	1	1							
OR			0	1	0	0							
XOR			0	1	0	1							
SHL			0	1	1	0							
SHR			0	1	1	1							
SAR			1	0	0	0							
CMP			1	0	0	1					0		
Immediate Instruction													
ADDI	0	1	0	0	0	0	X	X	X		1	1	0
SUBI			0	0	0	1							
MULI			0	0	1	0							
ANDI			0	0	1	1							
ORI			0	1	0	0							
XORI			0	1	0	1							
SHLI			0	1	1	0							
SHRI			0	1	1	1							
SARI			1	0	0	0							
CMPI			1	0	0	1					0		
Jump Instruction													
JMP	1	1	0	0	0	0	X	X	X		0	0	1
JE			0	0	0	1	1	0	X				
JNE			0	0	1	0	0	X	X				
JL			0	0	1	1	0	1	X				
JLE			0	1	0	0	sf=1 or zf=1		X				
JG			0	1	0	1	0	0	X				
JGE			0	1	1	0	sf=0 or zf=1		X				
JC			0	1	1	1	X	X	1				
JNC			1	0	0	0	X	X	0				
JZ			1	0	0	1	1	X	X				
JNZ			1	0	1	0	0	X	X				

3.3.4 Instruction Memory

Instruction memory is implemented as read only memory (ROM). ROM is called a memory but it is actually a combinational circuit. All values in ROM have fixed value and are loaded with all the machine instructions prior running the CPU.
Example of instruction memory is shown in figure below:

	0	1	2	3	4	5	6	7
0								
1								
2								
3								
4								
5								
6								
7								

Figure 3.6: Instruction memory (8X8 ROM)

3.3.5 Flag register

Flag register is used to hold zero flag **zf**, sign flag **sf** and carry flag **cf** values. It is implemented as 3 D flip-flops derived in previous chapter. It updates its flag values after execution of each instruction. Flag values are specially helpful for jump instruction.

3.3.6 Program counter (PC)

Program counter is used to keep track of instruction pointer which is used to determine instruction to be fetched next. It is implemented as register built in previous chapter. Its value is incremented by 1 via an adder after execution of each instruction (except jump instruction) or is set to address specified after execution of jump instruction.

3.4 Central Processing Unit (CPU)

All computer architecture building blocks are combined to create central processing unit (CPU). It has different datapaths for register, immediate and jump instruction. They are explained below:

1. **CPU datapath for register instruction:** CPU datapath for register instruction is shown in below:

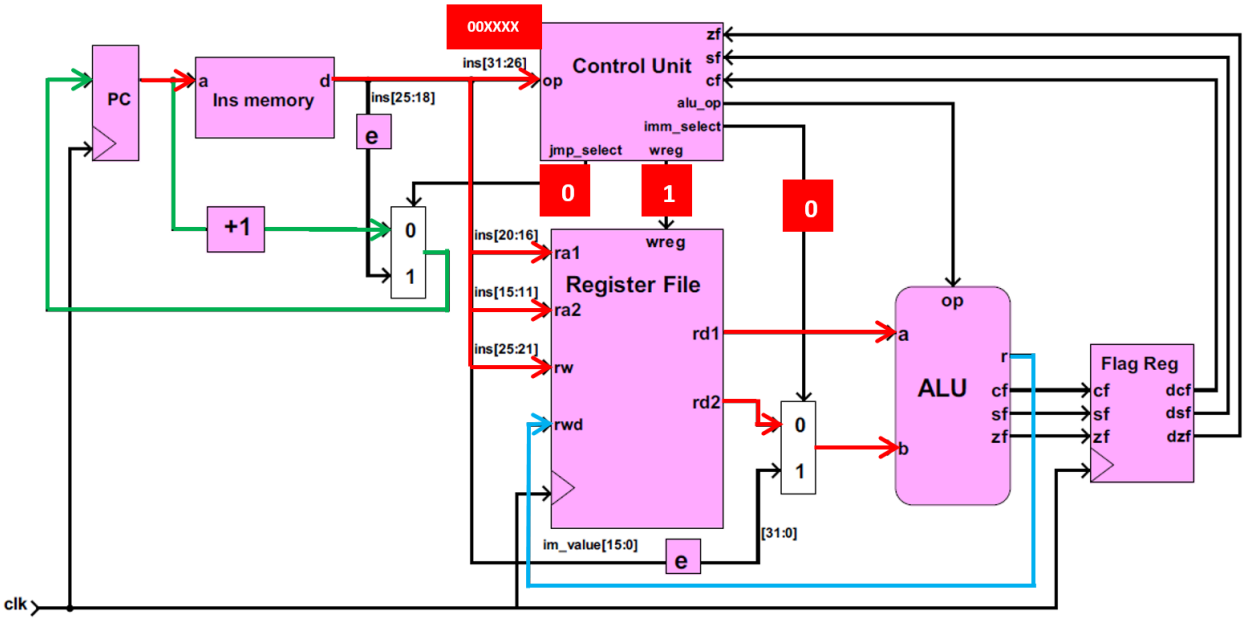


Figure 3.7: CPU datapath for register instruction

- **Red path:** Red path shows that program counter (PC) is used as address a to fetch next instruction to be executed from instruction memory (Ins memory). Then instruction goes to control unit (CU) which decodes the instruction and sends operation code (alu_op) to ALU. Control unit (CU) sets three control signal as follows:
  - **jmp_select:** It selects path 0 in first 2-to-1 multiplexer which will increment PC by 1.
  - **wreg:** It enables wreg in register file which will allow register, Rc to be written with new value .
  - **imm_select:** It selects path 0 in second 2-to-1 multiplexer which will pass value of register Rb to ALU.

  Instruction also goes to register file which receives register number of Ra, Rb and Rc at ra1, ra2 and rw ports respectively. Values of Ra and Rb are loaded to rd1 and rd2 ports respectively which are sent directly to ALU to perform operation.

- **Sky blue path:** Sky blue path shows that ALU produces result, r which is sent to register file as rwd to store it in register Rc.
- **Green path:** Green path shows that PC is incremented by 1 and is stored in PC.

2. **CPU datapath for immediate instruction:** CPU datapath for immediate instruction is shown in below:

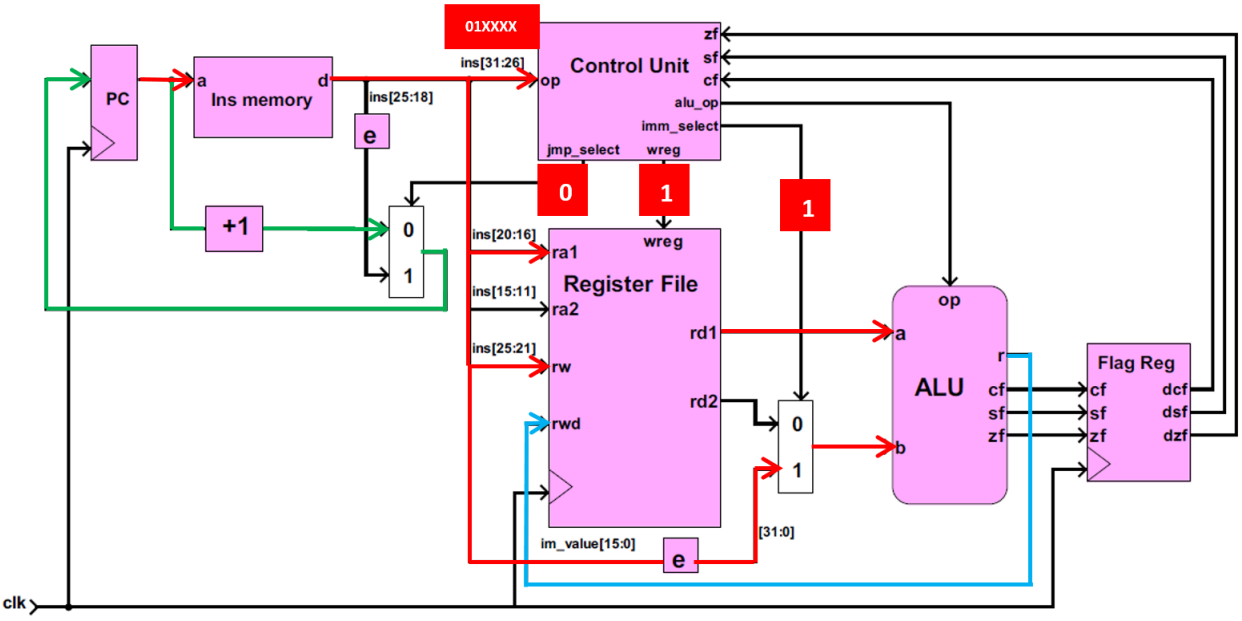


Figure 3.8: CPU datapath for immediate instruction

- **Red path:** Similarly next instruction to be executed is fetched from instruction memory (Ins memory) by using program counter (PC). Then control unit decodes the instruction and generates operation code (alu_op) to ALU. Control unit (CU) sets three control signal as follows:
  - **jmp_select:** It selects path 0 in first 2-to-1 multiplexer to increment PC by 1.
  - **wreg:** It enables wreg in register file to allow update of register, Rc.
  - **imm_select:** It selects path 1 in second 2-to-1 multiplexer which will select immediate value in instruction (0 to 15) instead of value of register Rb and pass it to AlU.

  Instruction also goes to register file which receives register number of Ra and Rc at ra1 and rw ports respectively. Values of Ra is loaded to rd1 port which is sent to ALU to perform operation.
- **Sky blue path:** Similarly, ALU generates result, r to store it in register Rc.
- **Green path:** Similarly, PC is incremented and stored in PC.

3. **CPU datapath for jump instruction:** CPU datapath for jump instruction is shown in below:

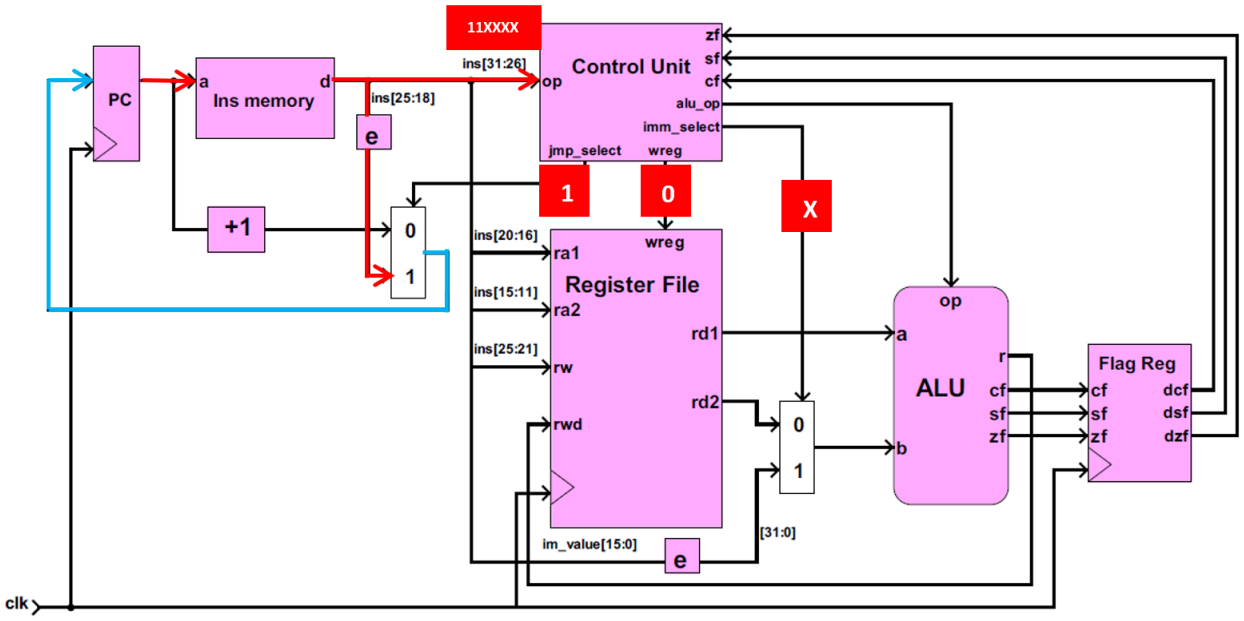


Figure 3.9: CPU datapath for jump instruction

- **Red path:** Similarly, next instruction is fetched which is decoded by control unit. Control unit (CU) sets three control signal as follows:
  - **jmp_select:** It selects path 1 in first 2-to-1 multiplexer which will load program counter (PC) with address specified in instruction (18 to 25).
  - **wreg:** It disables wreg in register file to prevent register Rc to be written.
  - **imm_select:** It is don't care because this instruction will not perform any operation in ALU.
- **Sky blue path:** Sky blue path shows PC is loaded with new address.

3.5 Conclusion

This chapter described all computer architecture building blocks and constructed computer from these blocks. In next chapter, computer will be interfaced and an assembler will be built.

Chapter 4

Interfacing and Assembler

4.1 Introduction

This chapter describes interfacing where computer is connected to LED, seven segment display and VGA monitor to display output. Then, assembler is designed which is built on the top of computer. Finally, chapter ends with conclusion.

4.2 Interfacing

Since computer do not have instructions to display output, computer can be interfaced to display contents of registers, output of ALU and value of program counter (PC) via LED, seven segment display or VGA monitor.

4.2.1 Light Emitting Diode (LED)

Registers, ALU output or program counter can be connected to LEDs. Since register can be large in size and numbers of LED are limited, least significant portion of register can be connected to LEDS.

4.2.2 Seven Segment Display

Seven segment display requires the following circuit [17]:

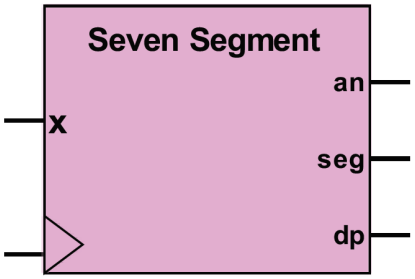


Figure 4.1: Seven segment display circuit

Seven segment circuit uses 2-bit LED-activating counter. Counter will repeatedly count from zero to three for continuously activating and updating the four seven-segment LEDs. It takes following inputs:

1. **x:** Values of registers, ALU output or program counter.
2. **clk:** Clock input which is usually 190Hz.

Then circuit produces following outputs:

1. **an:** Anode signals of the 7-segment LED display.
2. **seg:** Cathode patterns of the 7-segment LED display.
3. **dp:** Decimal point.

4.2.3 VGA Monitor Display

VGA monitor display requires the following circuits [18]

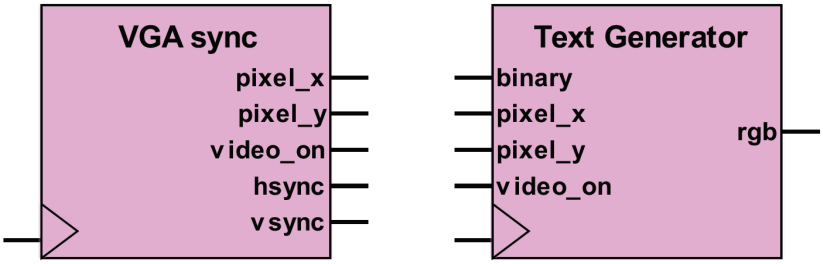


Figure 4.2: VGA monitor display circuits

VGA sync and text generator circuits are described below:

- **VGA sync circuit:** VGA sync circuit produces timing and synchronization signals based on resolution of VGA monitor. It takes following input:
  1. **clk:** Clock input which is usually 50MHz.

  Then circuit produces following outputs:

  1. **hsync:** Control horizontal scan of monitor.
  2. **vsync:** Control vertical scan of monitor.
  3. **video_on:** Enable or disable of display.
  4. **pixel_x:** X coordinate of current pixel.
  5. **pixel_y:** Y coordinate of current pixel.

- **Text generator circuit:** Text generator circuit produces texts to be displayed on VGA monitor. It generally uses font rom which stores all the font patterns to generate text. It takes following inputs:
  1. **binary:** Contents of registers, ALU output and program counter.
  2. **clk:** Clock input which is usually 50MHz.
  3. **video_on:** Enable or disable of display.
  4. **pixel_x:** X coordinate of current pixel.
  5. **pixel_y:** Y coordinate of current pixel.

  Then circuit produces following output:

  1. **rgb:** Color signals (Red, Green and Blue).

4.2.4 Complete Design of Interfacing Circuit

Complete design of interfacing circuit is shown in figure:

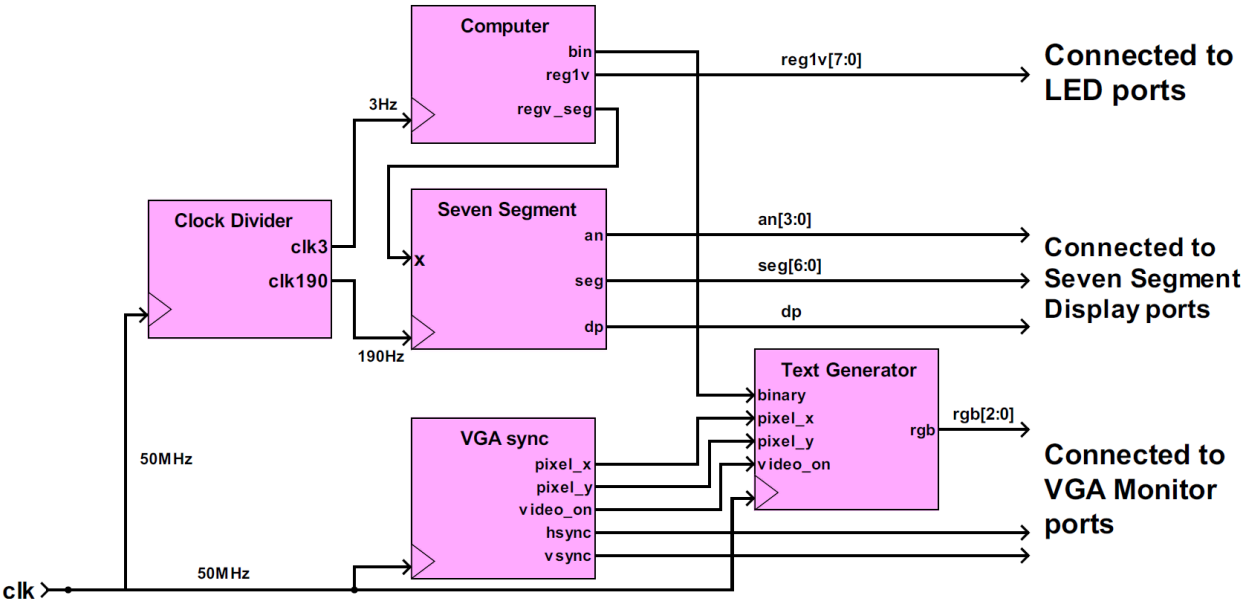


Figure 4.3: Complete design of interfacing circuit

As shown in figure, Clock divider circuit is used to generate 3Hz and 190Hz clock inputs for computer and seven segment circuit respectively. **reg1v**, **regv_seg** and **bin** contains data necessary to be displayed in LED, seven segment and VGA monitor respectively. **reg1v** port of computer circuit is connected to LED ports of FPGA. **regv_seg** port of computer circuit is input **x** to seven segment display circuit. Outputs of seven segment circuit are connected to seven segment ports of FPGA. **bin** port of computer circuit is used as input **binary** in text generator circuit. Output of text generator is connected to VGA ports of FPGA.

4.3 Assembler

Since writing instructions in machine code is difficult, assembler is used. Assembler translates assembly program to machine code which matches with instruction set architecture of computer. It makes programming task easier. Following figure shows different phases of assembler:

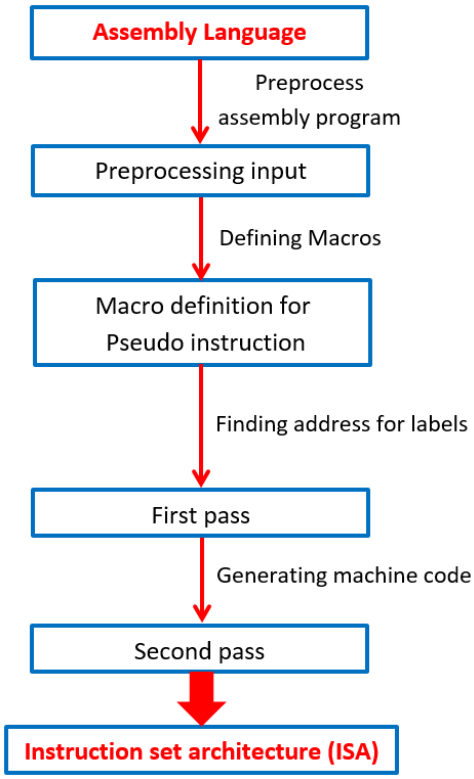


Figure 4.4: Different phases of assembler

4.3.1 Preprocessing Input (Lexical Analysis)

In first step, assembly program will be preprocessed. It will remove all the unnecessary contents of program. It performs following tasks:

1. Remove comment after ";".
2. Replace all consecutive white-spaces into one white-space.
3. Remove all white-spaces before and after instruction.
4. Tokenize every word in instruction.

4.3.2 Pseudo Extension (Parsing)

In second step, all pseudo-instructions in assembly program will be extended. It will also perform error checking. It performs following tasks:

1. Perform extension of pseudo instruction.

Table 4.1: Example of extension of pseudo instruction

Pseudo Instruction	Extension
MOV(R1,R2)	ADD(R1,R1,R2)
MOV(R1,2)	ADDI(R1,R1,2)
SUB(R2,R4)	SUB(R2,R2,R4)
AND(R4,100)	ANDI(R4,R4,100)
CMP(R5,R6)	CMP(R5,R5,R6)
SHL(R7,10)	SHLI(R7,R7,10)

2. Show error message if illegal pseudo instruction or register is found.
3. Show error message if immediate value is out of range.
4. Show error message if illegal instruction format is found.

4.3.3 Pass 1

In third step, all assembly instructions will be converted to machine instructions except instruction containing labels and symbols. It will also construct symbol and label tables. It performs following tasks:

1. Translate every instruction to its corresponding machine code except jump instruction.
2. Find all the label outside instruction and store them in a label table.
3. Find all the label inside instruction and store them in a symbol with its address.

4.3.4 Pass 2

In fourth step, whole assembly program will be translated to machine code with appropriate addressing. It performs following tasks:

1. Translate every jump instruction to its corresponding machine code.
2. Show error message if label's address is not found in symbol table.

4.4 Conclusion

This chapter explained interfacing of computer and design of assembler. Next chapter will verify computer system in software and run computer system on fpga for result analysis.

Chapter 5

Verification, Simulation on FPGA and Result Analysis

5.1 Introduction

This chapter focuses on running an assembly language program on assembler to generate machine code which will be loaded within instruction memory and run on fpga. Then, values of registers, ALU output and program counter (PC) of computer will be viewed on LED, seven segment display and VGA monitor. Finally, chapter ends with conclusion.

5.2 Verification

In order to verify computer, the following series will be programmed:

$$(1*1)*2-(1*2)/2+(1*2*3)*2-(1*2*3*4)/2 \quad (5.1)$$

Following figure shows pseudo-code and desired output of the program:

Pseudo Code

```
R1=0;
R2=0;
R3=0;

while(R2<=4)
{
      R2++;
      R1=R1*R2;

      if(R2 is odd)
      {
            R5=R1;
            R5=R5*2;
            R3=R3+R5;
      }
      else
      {
            R5=R1;
            R5=R5/2;
            R3=R3-R5;
      }
}
```

Desired Output

```
//Multiplicand value = (1*2*3*4)
R1=24;
//Multiplier value = 4
R2=4;
//Final result of series = 1
R3=1;
```

Figure 5.1: Pseudo-code of program with desired output

5.2.1 Program on assembler

Following figure shows assembly code of program:

	Assembly Code
(00)	XOR R1, 1
(01)	XOR R2, 0
(02)	XOR R3, 0
(03)	
(04)	JP:
(05)	ADD R2, 1
(06)	MUL R1, R2
(07)	
(08)	AND R4, 0
(09)	MOV R4, R2
(10)	SHR R4, 1
(11)	JNC JP1
(12)	
(13)	AND R5, 0
(14)	MOV R5, R1
(15)	SHL R5, 1
(16)	ADD R3, R5
(17)	JMP JP2
(18)	
(19)	JP1:
(20)	AND R5, 0
(21)	MOV R5, R1
(22)	SAR R5, 1
(23)	SUB R3, R5
(24)	
(25)	JP2:
(26)	CMP R2, 4
(27)	JNE JP

Figure 5.2: Pseudo-code of program with desired output

Following figure shows preprocessing and pseudo extension of assembly code:

	Preprocessing & Pseudo Extension
(00)	XORI R1 R1 1
(01)	XORI R2 R2 0
(02)	XORI R3 R3 0
(03)	JP: ADDI R2 R2 1
(04)	MUL R1 R1 R2
(05)	ANDI R4 R4 0
(06)	ADD R4 R0 R2
(07)	SHRI R4 R4 1
(08)	JNC JP1
(09)	ANDI R5 R5 0
(10)	ADD R5 R0 R1
(11)	SHLI R5 R5 1
(12)	ADD R3 R3 R5
(13)	JMP JP2
(14)	JP1: ANDI R5 R5 0
(15)	ADD R5 R0 R1
(16)	SARI R5 R5 1
(17)	SUB R3 R3 R5
(18)	JP2: CMPI R2 R2 4
(19)	JNE JP

Figure 5.3: Preprocessing and pseudo extension of code

Following figure shows pass 1 of assembly code:

	Pseudo Extension	Pass 1
(00)	XORI R1 R1 1	010101_00001_00001_0000000000000001
(01)	XORI R2 R2 0	010101_00010_00010_0000000000000000
(02)	XORI R3 R3 0	010101_00011_00011_0000000000000000
(03)	JP: ADDI R2 R2 1	010000_00010_00010_0000000000000001
(04)	MUL R1 R1 R2	000010_00001_00001_00010_00000000000
(05)	ANDI R4 R4 0	010011_00100_00100_0000000000000000
(06)	ADD R4 R0 R2	000000_00100_00000_00010_00000000000
(07)	SHRI R4 R4 1	010111_00100_00100_0000000000000001
(08)	JNC JP1	111000
(09)	ANDI R5 R5 0	010011_00101_00101_0000000000000000
(10)	ADD R5 R0 R1	000000_00101_00000_00001_00000000000
(11)	SHLI R5 R5 1	010110_00101_00101_0000000000000001
(12)	ADD R3 R3 R5	000000_00011_00011_00101_00000000000
(13)	JMP JP2	110000
(14)	JP1: ANDI R5 R5 0	010011_00101_00101_0000000000000000
(15)	ADD R5 R0 R1	000000_00101_00000_00001_00000000000
(16)	SARI R5 R5 1	011000_00101_00101_0000000000000001
(17)	SUB R3 R3 R5	000001_00011_00011_00101_00000000000
(18)	JP2: CMPI R2 R2 4	011001_00010_00010_0000000000000100
(19)	JNE JP	110010

Figure 5.4: Pass 1 of code

Label and symbol table are shown below:

Table 5.1: Label and symbol table

Label	Address
JP	00000011 (03)
JP1	00001110 (14)
JP2	00010010 (18)

Symbol	Address
JP1	00001000 (08)
JP2	00001101 (13)
JP	00010011 (19)

Following figure shows pass 2 of assembly code:

	Pseudo Extension	Pass 2
(00)	XORI R1 R1 1	010101_00001_00001_0000000000000001
(01)	XORI R2 R2 0	010101_00010_00010_0000000000000000
(02)	XORI R3 R3 0	010101_00011_00011_0000000000000000
(03)	JP: ADDI R2 R2 1	010000_00010_00010_0000000000000001
(04)	MUL R1 R1 R2	000010_00001_00001_00010_00000000000
(05)	ANDI R4 R4 0	010011_00100_00100_0000000000000000
(06)	ADD R4 R0 R2	000000_00100_00000_00010_00000000000
(07)	SHRI R4 R4 1	010111_00100_00100_0000000000000001
(08)	JNC JP1	111000_00001110_00000_00000_00000000
(09)	ANDI R5 R5 0	010011_00101_00101_0000000000000000
(10)	ADD R5 R0 R1	000000_00101_00000_00001_00000000000
(11)	SHLI R5 R5 1	010110_00101_00101_0000000000000001
(12)	ADD R3 R3 R5	000000_00011_00011_00101_00000000000
(13)	JMP JP2	110000_00010010_00000_00000_00000000
(14)	JP1: ANDI R5 R5 0	010011_00101_00101_0000000000000000
(15)	ADD R5 R0 R1	000000_00101_00000_00001_00000000000
(16)	SARI R5 R5 1	011000_00101_00101_0000000000000001
(17)	SUB R3 R3 R5	000001_00011_00011_00101_00000000000
(18)	JP2: CMPI R2 R2 4	011001_00010_00010_0000000000000100
(19)	JNE JP	110010_00000011_00000_00000_00000000
(20)	NOP	000011_00010_00010_00010_00000000000

Figure 5.5: Pass 2 of code

5.2.2 Program on processor

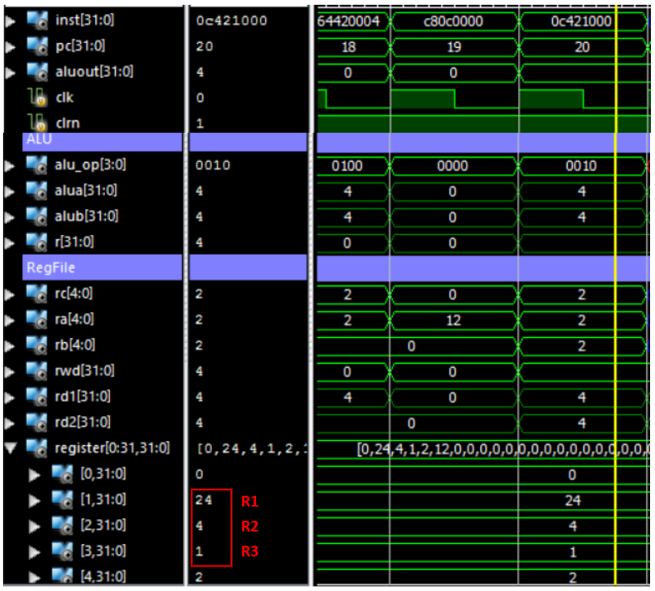
Following figure shows program loaded into instruction memory in processor:

```
module instmem (a,inst);                // instruction memory, rom
    input  [31:0] a;                      // address
    output [31:0] inst;                   // instruction
    wire   [31:0] rom [0:20];             // rom cells: 32 words * 32 bits

    // rom[word_addr] = instruction      // (pc) label   instruction
   assign rom[5'h0] = 32'b010101_00001_00001_0000000000000001; //XORI R1 R1 1
   assign rom[5'h1] = 32'b010101_00010_00010_0000000000000000; //XORI R2 R2 0
   assign rom[5'h2] = 32'b010101_00011_00011_0000000000000000; //XORI R3 R3 0
   assign rom[5'h3] = 32'b010000_00010_00010_0000000000000001; //JP: ADDI R2 R2 1
   assign rom[5'h4] = 32'b000010_00001_00001_00010_00000000000; //MUL R1 R1 R2
   assign rom[5'h5] = 32'b010011_00100_00100_0000000000000000; //ANDI R4 R4 0
   assign rom[5'h6] = 32'b000000_00100_00000_00010_00000000000; //ADD R4 R0 R2
   assign rom[5'h7] = 32'b010111_00100_00100_0000000000000001; //SHRI R4 R4 1
   assign rom[5'h8] = 32'b111000_00001110_00000_00000_00000000; //JNC JP1
   assign rom[5'h9] = 32'b010011_00101_00101_0000000000000000; //ANDI R5 R5 0
   assign rom[5'hA] = 32'b000000_00101_00000_00001_00000000000; //ADD R5 R0 R1
   assign rom[5'hB] = 32'b010110_00101_00101_0000000000000001; //SHLI R5 R5 1
   assign rom[5'hC] = 32'b000000_00011_00011_00101_00000000000; //ADD R3 R3 R5
   assign rom[5'hD] = 32'b110000_00010010_00000_00000_00000000; //JMP JP2
   assign rom[5'hE] = 32'b010011_00101_00101_0000000000000000; //JP1: ANDI R5 R5 0
   assign rom[5'hF] = 32'b000000_00101_00000_00001_00000000000; //ADD R5 R0 R1
   assign rom[5'h10] = 32'b011000_00101_00101_0000000000000001; //SARI R5 R5 1
   assign rom[5'h11] = 32'b000001_00011_00011_00101_00000000000; //SUB R3 R3 R5
   assign rom[5'h12] = 32'b011001_00010_00010_0000000000000100; //JP2: CMPI R2 R2 4
   assign rom[5'h13] = 32'b110010_00000011_00000_00000_00000000; //JNE JP
   assign rom[5'h14] = 32'b000011_00010_00010_00010_00000000000; //NOP

    //assign inst = rom[a];           // use word address to read rom
    assign inst = (a >= 5'h15)? 32'b000011_00010_00010_00010_00000000000 : rom[a];
endmodule
```

Figure 5.6: Loading machine code into instruction memory

As shown in figure, same instruction NOP will executed repeatedly rest of the time after reaching end of program.

Following figure shows simulation result of processor in software:



Desired Output
//Multiplicand value = (1*2*3*4)
R1=24;
//Multiplier value = 4
R2=4;
//Final result of series = 1
R3=1;

Figure 5.7: Simulation result of processor in software

From the figure, it can be said that processor is **verified** in software.

5.3 Simulation on FPGA

Computer system will now be simulated in FPGA. For simulation on FPGA, Nexys3 board based on Xilinx Spartan-6 is used [19, 20] . It is shown in figure:

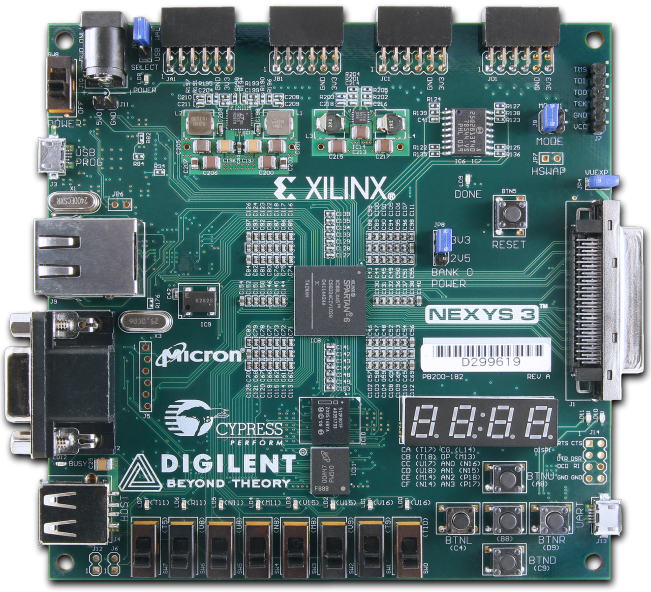


Figure 5.8: Nexys3 board based on Xilinx Spartan-6

Following figure shows FPGA simulating processor with displaying outputs:

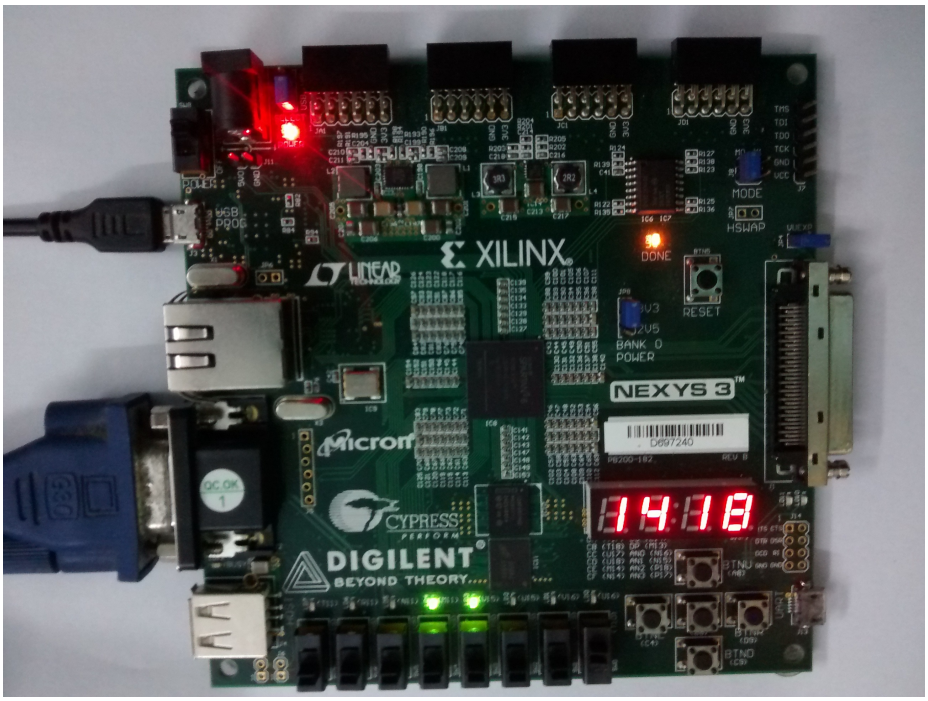


Figure 5.9: FPGA simulating processor with displaying outputs

Following figure shows VGA monitor displaying outputs:

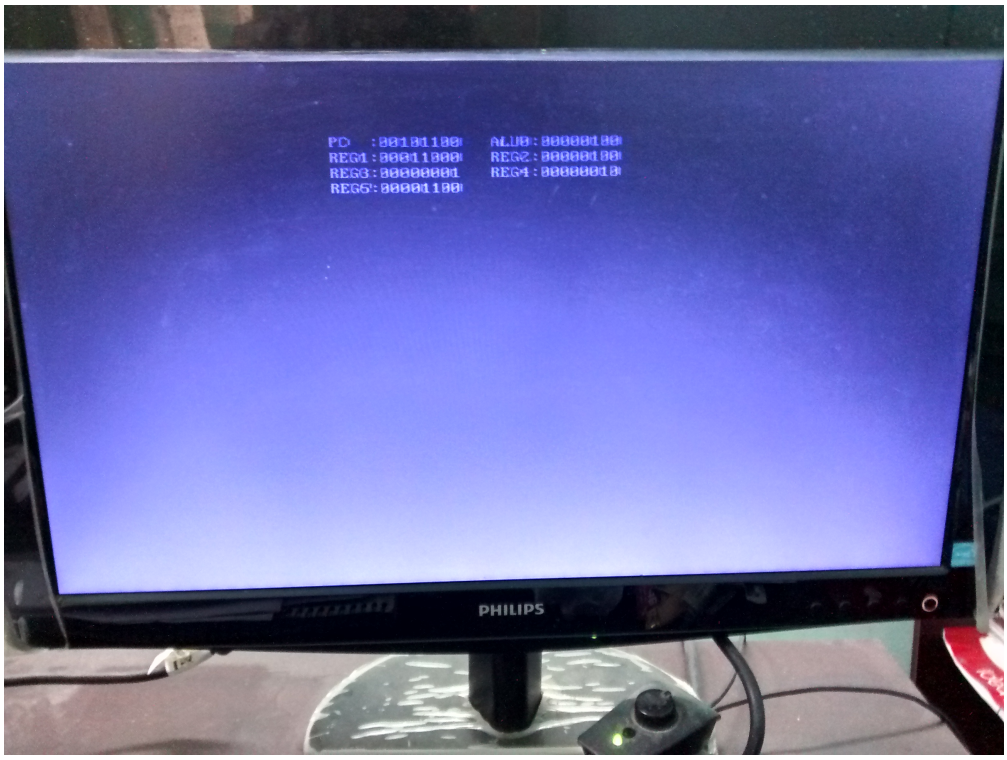


Figure 5.10: VGA monitor displaying outputs

5.3.1 LED output

Following figure shows LED displaying value of register R1:

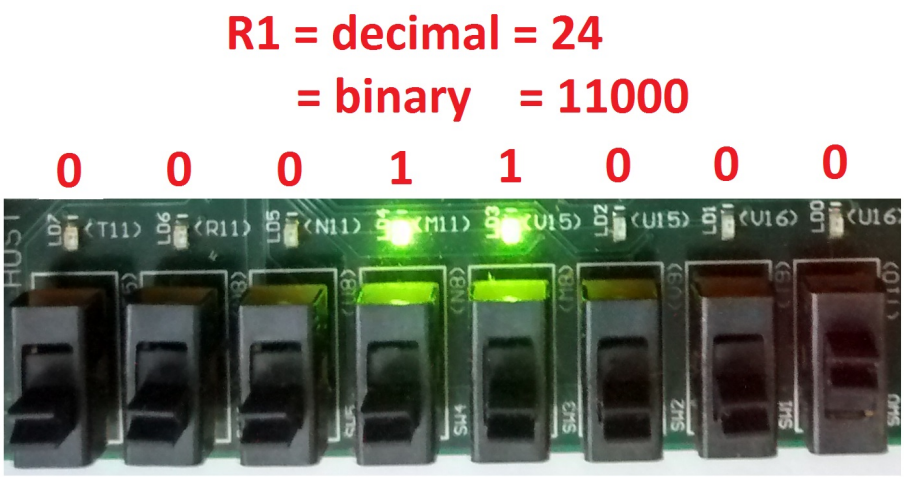


Figure 5.11: LED output

5.3.2 Seven segment display

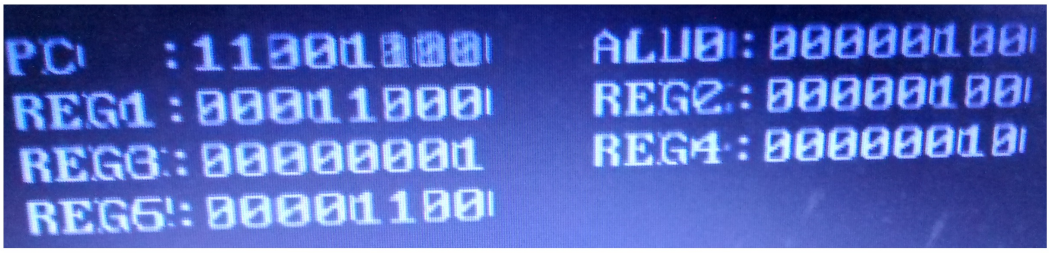
Following figure shows seven segment displaying values of register R1, R2 and R3:

R1 = decimal = 24
= hexadecimal = 18
R2 = decimal = hexadecimal = 4
R3 = decimal = hexadecimal = 1

Figure 5.12: Seven segment display

5.3.3 VGA monitor display

Following figure shows VGA monitor displaying values of register R1, R2, R3, R4 and R5, ALU output and Program counter :



REG1 = binary = 00011000 (24)
REG2 = binary = 00000100 (4)
REG3 = binary = 00000001 (1)
REG4 = binary = 00000010 (2)
REG5 = binary = 00001100 (12)

Figure 5.13: VGA monitor display

5.4 Result Analysis

From simulation result of processor in FPGA and software, it can be concluded that computer system is verified.

5.5 Conclusion

This chapter described simulation of processor on software and FPGA. Next chapter will conclude the thesis and discuss future work direction.

Chapter 6

Conclusion and Future Works

6.1 Conclusion

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

6.2 Future Works

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

References

[1] A. Clements, "Computer architecture education," *IEEE Micro*, vol. 20, no. 3, pp. 10–12, 2000.

[2] L. Wang and R. Huang, "A comprehensive experiment scheme for computer science and technology," in *Proceedings of the International Conference on Frontiers in Education: Computer Science and Computer Engineering (FECS)*, p. 1, The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2012.

[3] S. Schocken, N. Nisan, and M. Armoni, "A synthesis course in hardware architecture, compilers, and software engineering," in *ACM SIGCSE Bulletin*, vol. 41, pp. 443–447, ACM, 2009.

[4] J. T. O'Donnell, "Connecting the dots: Computer systems education using a functional hardware description language," *arXiv preprint arXiv:1301.5075*, 2013.

[5] K. Nakano, K. Kawakami, K. Shigemoto, Y. Kamada, and Y. Ito, "A tiny processing system for education and small embedded systems on the fpgas," in *Embedded and Ubiquitous Computing, 2008. EUC'08. IEEE/IFIP International Conference on*, vol. 2, pp. 472–479, IEEE, 2008.

[6] K. Nakano and Y. Ito, "Processor, assembler, and compiler design education using an fpga," in *Parallel and Distributed Systems, 2008. ICPADS'08. 14th IEEE International Conference on*, pp. 723–728, IEEE, 2008.

[7] R. Nakamura, Y. Ito, and K. Nakano, "Tinycse: Tiny computer system for education," in *Computing and Networking (CANDAR), 2013 First International Symposium on*, pp. 639–641, IEEE, 2013.

[8] N. Nisan and S. Schocken, *The elements of computing systems: building a modern computer from first principles*. MIT press, 2005.

[9] D. Harris and S. Harris, *Digital design and computer architecture*. Morgan Kaufmann, 2010.

[10] M. M. Mano, C. R. Kime, T. Martin, *et al.*, *Logic and computer design fundamentals*, vol. 3. Prentice Hall, 2008.

[11] H. Oztekin, F. Temurtas, and A. Gulbag, "Bzk. sau. fpga10. 0: Microprocessor architecture design on reconfigurable hardware as an educational tool," in *Computers & Informatics (ISCI), 2011 IEEE Symposium on*, pp. 385–389, IEEE, 2011.

[12] N. Bhardwaj, M. Senftleben, and K. Schneider, "Abacus: A processor family for education," in *Proceedings of the WESE'14: Workshop on Embedded and Cyber-Physical Systems Education*, p. 2, ACM, 2014.

[13] M. Holland, J. Harris, and S. Hauck, "Harnessing fpgas for computer architecture education," in *Microelectronic Systems Education, 2003. Proceedings. 2003 IEEE International Conference on*, pp. 12–13, IEEE, 2003.

[14] H. Park, Y.-W. Ko, J. So, and J.-G. Lee, "Manycore processor education platform with fpga for undergraduate level computer architecture class," in *The 2nd International Conference on Information Science and Technology*, 2013.

[15] R. H. Katz and G. Borriello, "Contemporary logic design," 2005.

[16] Y. Li, *Computer Principles and Design in Verilog HDL*. John Wiley & Sons, 2015.

[17] R. E. Haskell and D. M. Hanna, "Introduction to digital design using digilent fpga boards-block diagram/verilog examples," *Oakland University, Rochester, Michigan, ISBN*, pp. 978–0, 2009.

[18] P. P. Chu, *FPGA prototyping by Verilog examples: Xilinx Spartan-3 version*. John Wiley & Sons, 2011.

[19] DIGILENT, "Nexys3 fpga board reference manual." `https://reference.digilentinc.com/_media/nexys:nexys3:nexys3_rm.pdf`.

[20] Xilinx, “Spartan-6 family overview.” https://www.xilinx.com/support/documentation/data_sheets/ds160.pdf.