RMIT University

School of Engineering

EEET2261 – Computer Architecture and Organisation

Laboratory Group Project Report

ISA and CPU Datapath for Binary Search

Sneha Chakravarthy (s3838647)

Hao Wan (s3655236)

Rishitkrishwa Rao (s3843246)

Contents

1. Introduction

The aim of this project is to:

1. Develop an Instruction Set Architecture to perform binary search for an element in a sorted array (see figure 1).
2. Design and simulate a single-cycle CPU datapath capable of executing the instructions defined in the ISA.

The components necessary to implement the CPU Datapath such as ALU, Program Counter, Register File, Control Unit are also designed and verified.

2. Background

2.1 Binary Search

Binary search is a search algorithm that finds the position of a target value within a sorted array. Binary search compares the target value to the middle element of the array. If they are not equal, the half in which the target cannot lie is eliminated and the search continues on the remaining half, again taking the middle element to compare to the target value, and repeating this until the target value is found. If the search ends with the remaining half being empty, the target is not in the array [1].

```
function search(A, n, T) is
    L := 0
    R := n − 1
    while L ≤ R do
        m := floor((L + R)/2)
        if A[m] < T then
            L := m + 1
        else if A[m] > T then
            R := m − 1
        else:
            return m
    return unsuccessful
```

Figure 1: The binary search algorithm [1]

2.2 Harvard Architecture

This project utilises the Harvard Architecture model where separate memory modules are used to store the instructions and data in a CPU. The instructions and data also have separate pathways as a result [2].

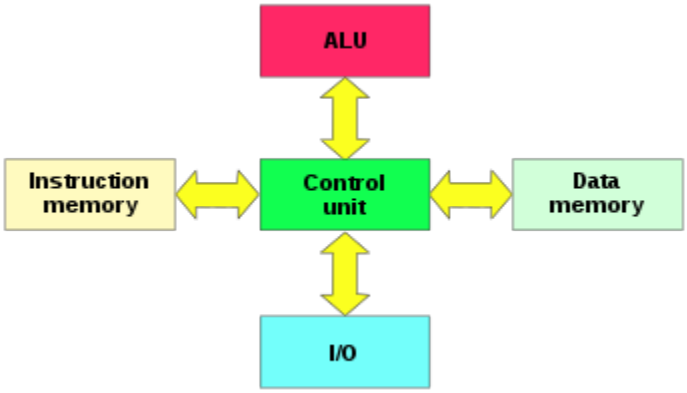

Figure 2: Harvard Architecture [2]

2.3 Instruction Cycle

The central processing unit (CPU) follows the fetch-decode-execute cycle (also known as the fetch-execute cycle, or simply the fetch-execute cycle) from boot-up until the computer has shut down in order to process instructions.

The fetch stage, the decode stage, and the execute stage are the three primary stages that make up this process. [3]

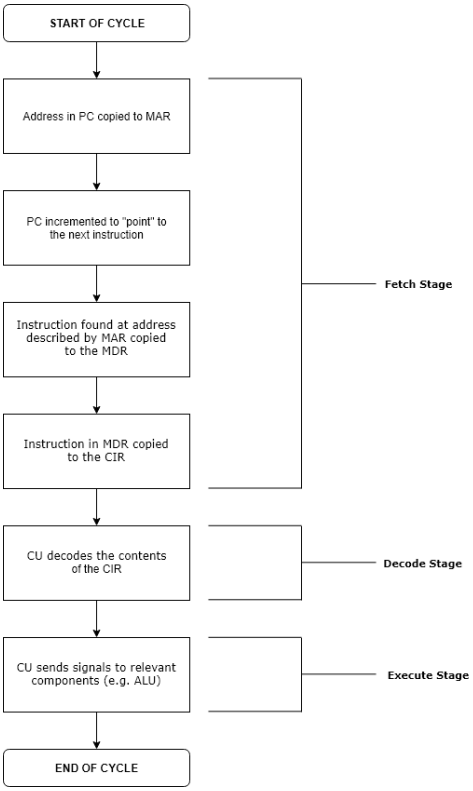

Figure 3: Fetch-Decode-Execute Cycle [3]

3. Methods

3.1 Instruction Set Architecture

The Instruction Set Architecture (ISA) for the CPU is designed based on the binary search algorithm seen in figure 1.

From the algorithm, it can be seen that the CPU is required to perform the following functions:

1. Arithmetic operations: Addition, Subtraction and Division
2. Logic operations: Comparison
3. Control transfer operations: Branch and Jump (for the if-else and loop control)

Since RISC-V is a widely used and familiar open-source instruction set, the ISA for the CPU was mainly derived from the RISC-V ISA. To identify the exact instructions required, the algorithm was implemented in the RISC-V assembly language as seen in figure 2.

```
main:
       lui    a0, data          # address of the data vector
       lw   a1, n             # number of elements in the data vector
       srai  a7, a1, 2
       slli  a2, a7, 2
       add   a2, a0, a2
       lw    a2, -4(a2)        # the search target
       addi t0, x0, 0         # L = 0
       addi t1, a1, -1        # R = n-1
       addi a3, x0, -1        # target = -1
loop:
       add t3, t1, t0         # M = R + L
       srai t3, t3, 1         # M = M/2 = (R+L)/2
       slli t5, t3, 2
       add t5, a0, t5
       lw t6, 0(t5)           # t6 = A[M]
       bgeu t6, a2, elseif    # A[M] > search target
       # if body
       addi t0, t3, 1         # L = M + 1
       j endif
       elseif:
       bgeu a2, t6, else      # A[M] < search target
       # else if body
       addi t1, t3, -1        # R = M - 1
       j endif
       else:
       # else body
       add a3, x0, t3
       j end
endif:
       bgeu t1, t0, loop      # L < R
       j end
end:
       add a0, x0, a3
```

Figure 4: Assembly code to execute the binary search algorithm

For designing the ISA, we also take into to consideration the memory elements used in simulating the actual datapath. The LPM_ROM mega function in Quartus Prime used as the Instruction Memory and Data Memory of the CPU allows storage of n words that can be addressed from 0, using 2^n address lines. The addresses required to access the words inside the memory would therefore range from $0\ to\ n-1$ only. As a result, the code no longer needs to calculate the address of an array element using its index. The index can be directly used as an address instead. This eliminates the need for the SLLI instruction.

The RISC-V ISA's jump instruction (J) is a pseudo-instruction that executes a JAL (jump and link) instruction by linking the return address to the register x0. Generally, since the x0 register cannot be modified, this instruction effectively does not store any return address. However, since the register file used in our design cannot prevent the register x0 from being modified, we adapt the jump instruction from the MIPS ISA to avoid this issue. This jump instruction updates the PC using an immediate value and does not store the return address.

Based on these considerations, the code that will be loaded into the instruction memory of the CPU is as follows:

```
      lui    a0, 0          # address of the data vector = 0
      lui    a1, 32         # number of elements in the data vector = 32
      lui    a2, 20         # search target = 20
      lui x0, 0
      addi t0, x0, 0        # L = 0
      addi t1, a1,  -1      # R = n-1
      addi a3, x0,  -1      # target = -1

      add t3, t1, t0        # M = R + L
      srai t3, t3, 1        # M = M/2 = (R+L)/2
      lw t4, 0(t3)
      bgeu t4, a2, elseif   # if A[M] > search target, branch to elseif
      addi t0, t3, 1        # L = M + 1
      j endif
elseif:
      bgeu a2,t4, else      # if A[M] < search target, branch to else
      addi t1, t3, -1       # R = M - 1
      j endif
else:
      add a3, x0, t3
      j end
endif:
      bgeu t1, t0, loop     # if L < R, branch to loop
      j end

end:
      add a0, x0, a3
```

Figure 5 : Binary Search Code

Based on the code in figure 5, the ISA is as follows:

Table 1: Instruction Set Architecture for the CPU Datapath implementing Binary Search

No	Instruction	Type
1	LUI - Load Upper Immediate	U Type
2	LW - Load Word	I Type
3	ADD - Add	R Type
4	ADDI - Add Immediate	I Type
5	SRAI - Shift Right Arithmetic Immediate	I Type
6	BGEU - Branch if Greater than or Equal to Unsigned	B Type
7	J - Jump	J Type

These instructions are encoded using the format seen in table 2 below. The colour black indicates bits of the instruction that are not used in an instruction. The "FUNC" field is used to identify the operation performed by the ALU while executing the instruction and OPCODE is a unique identifier for each instruction that is also used to generate control signals by

passing it through a control unit. The OPCODE's are designed such that the rightmost two bits (Instruction[1..0]) can be used directly as the select signal to choose what data should be written into the register file via a multiplexer.

There are three registers: RS1 (source register 1), RS2 (source register 2) and RD (destination register) as well as an immediate value IMM that are encoded according to the nature of the instruction. In total, each instruction is 20 bits long.

Table 2: Instruction Encoding Format

Index	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Type																				
R			RS2				RS1				FUNC		RD				OPCODE			
U	IMM												RD				OPCODE			
I	IMM						RS1				FUNC		RD				OPCODE			
B	IMM		RS2				RS1				IMM						OPCODE			
J	IMM																OPCODE			

3.2 Components

To fetch, decode and execute instructions, the CPU requires the following components:

1. Combinational Logic Elements
    a. Adder/Subtractor
    b. Arithmetic Right Shift
    c. Comparator
    d. Decoder
    e. Control Unit (Instruction Decoder)
    f. Multiplexers
    g. ALU (Arithmetic Logic Unit)
2. State Elements
    a. ROM (Instruction and Data Memory)
    b. Register
    c. Register File
    d. Program Counter

The design and implementation of these components are discussed in the following sections.

3.2.1 Adder Subtractor

To perform 8-bit addition and subtraction operations, a single component based on the ripple carry adder is designed. The input and output ports of the component are described below:

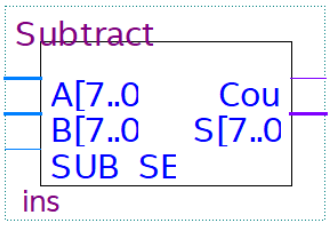

Figure 6: Subtractor Module 5

Table 3: Subtractor Module

Module Name: Subtractor				
Inputs			*Outputs*	
A	***B***	***SUB_SEL***	***S***	***Cout***
8-bit operand	8-bit operand	1-bit control	8-bit result	1-bit carry out

When the ***SUB_SEL*** signal is set to 0, addition is performed with operands and the result ***S = A + B***. Conversely, when the ***SUB_SEL*** is set to 1, subtraction is performed with the operands and the result ***S = A – B.***

To design this module, two sub modules are used: the 8-bit ripple carry adder and an 8-bit XOR module. The design and purpose of these modules is described below:

1. The 8-bit adder is composed of 8 1-bit full adders with the carry output of each full adder connected to the carry input port of the next full adder. The individual outputs of each full adder and connected to an 8-bit wide bus, giving an 8-bit output.

Table 4: Eight Bit Adder Module

Module Name: Eight Bit Adder				
Inputs			*Outputs*	
A	***B***	***Cin***	***S***	***Cout***
8-bit operand	8-bit operand	1-bit carry in	8-bit result	1-bit carry out

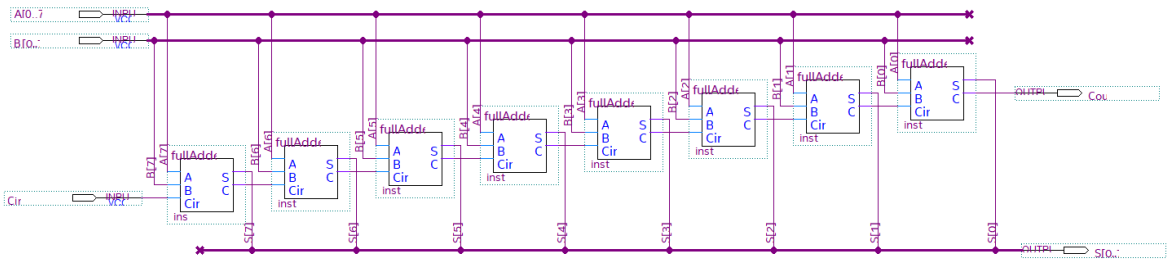

Figure 7: Eight Bit Ripple Carry Adder Schematic6

The result ***S = A + B + Cin*** and ***Cout*** is the carry out of the addition operation.

2. The adder currently performs the operation ***A + B***. To perform subtraction with the adder module, the subtractor module needs to convert the input B to its negative form such that the adder will now perform the operation ***A + (-B) = A – B.*** Negation can be achieved by adding the value 1 to the 2's complement of B ***(-B = ~B + 1).***
   The 2's complement of B can be found by perform bitwise XOR operations with the value 1. To perform bitwise XOR operations on an 8-bit value the following module is designed:

Table 5: Eight Bit XOR

Module Name: EightBitXOR		
Inputs		*Outputs*
A	***B***	***OUT***
8-bit operand	1-bit operand	8-bit result

When ***B = 0,*** the input is unchanged and ***OUT = A.*** Conversely when ***B = 1,*** the output is the 2's complement of input ***OUT = ~A.***

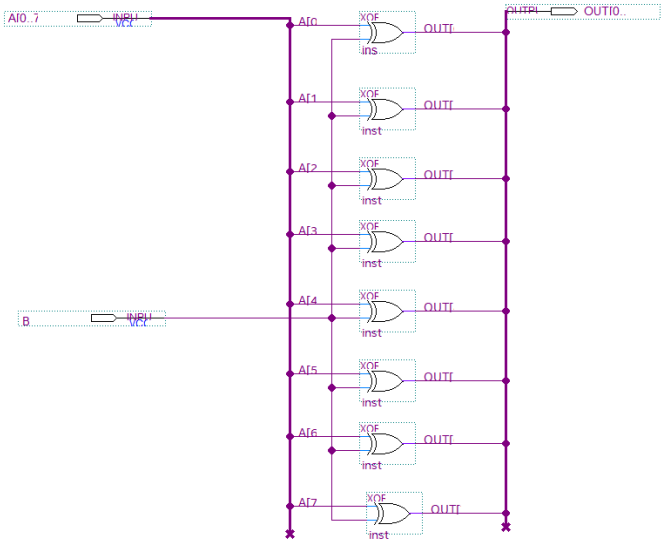

Figure 8: Eight Bit XOR7

The Adder and XOR modules are used to build the subtractor as follows:

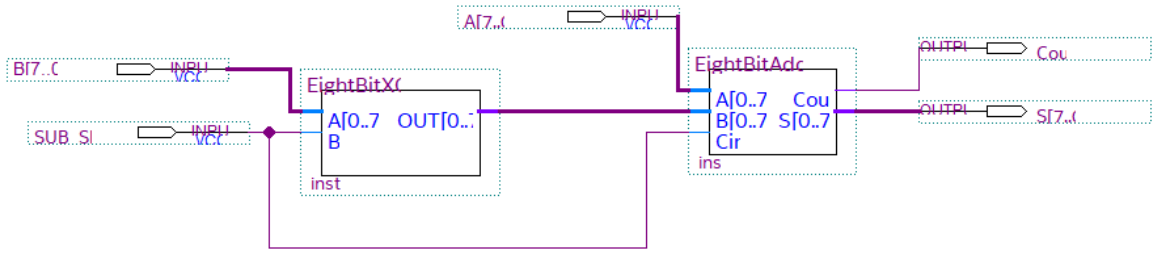

Figure 9: Eight Bit Adder/Subtractor8

The ***B*** and S***UB_SEL*** values are passed into the ***EightBitXOR*** module before being connected to the input of the ***EightBitAdder*** module. When the ***SUB_SEL = 0***, the output of the EightBitXOR module ***OUT = B*** and the value fed to the ***Cin*** port of the EightBitAdder is 0. Therefore, regular addition takes place.

Alternatively, when ***SUB_SEL = 1,*** the output of the EightBitXOR module ***OUT = ~B*** and the value fed to the ***Cin*** port of the EightBitAdder is 1. Therefore, the following operation takes place in the adder:

S = A + ~B + 1 = A – B.

3.2.2 Arithmetic Right Shift

Since the Binary Search program only requires the CPU to perform an arithmetic right shift operation by 1, a simple module that performs arithmetic right shift by the value of 1 is designed.

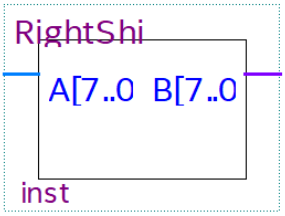

Figure 10: Arithmetic Right Shifter9

Table 6: Right Shift Module

Module Name: RightShift	
Inputs	*Outputs*
A	***B***
8-bit operand	8-bit result

The output ***B = A/2*** at all times.

3.2.3 Comparator

Since the Binary Search program only requires the CPU to perform a Greater than Equal to comparison, an 8-bit comparator that outputs whether the first input is greater than or equal to the second input is designed.

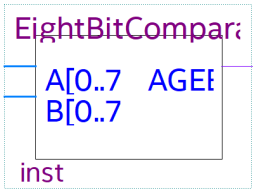

Figure 11: Eight Bit Comparator10

Table 7: Eight Bit Comparator Module

Module Name: EightBitComparator		
Inputs		*Outputs*
A	***B***	***AGEB***
8-bit operand	8-bit operand	1-bit result

To build an 8-bit comparator, 4-bit and 1-bit comparators are used in combination. The design of the 1-bit comparator is as follows:

Table 8: 1 bit Comparator

Module Name: comparator				
Inputs		*Outputs*		
A	***B***	***lesserA***	***equal***	***lesserB***
1-bit operand	1-bit operand	1-bit result	1-bit result	1-bit result

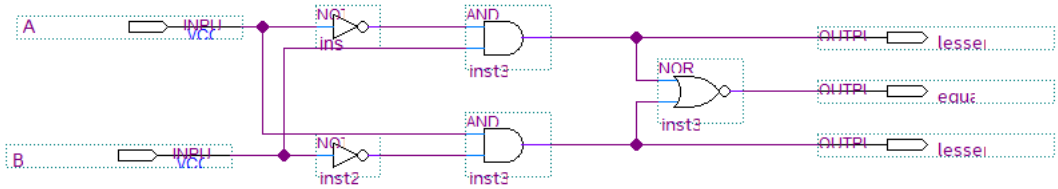

Figure 12: 1 bit comparator

The operation of the 1-bit comparator can be summarised as follows:

Table 9: 1 bit comparator operation

Condition	**lesserA**	**equal**	**lesserB**
A > B (eg: A = 2, B = 1)	0	0	1
A = B (eg: A = 2, B = 2)	0	1	0
A < B (eg: A = 2, B = 4)	1	0	0

Four of the 1-bit comparators are used to build a 4-bit comparator

Table 10: Four Bit comparator module

Module Name: FourBitComparator				
Inputs		*Outputs*		
A	***B***	***lesserA***	***equal***	***lesserB***
4-bit operand	4-bit operand	1-bit result	1-bit result	1-bit result

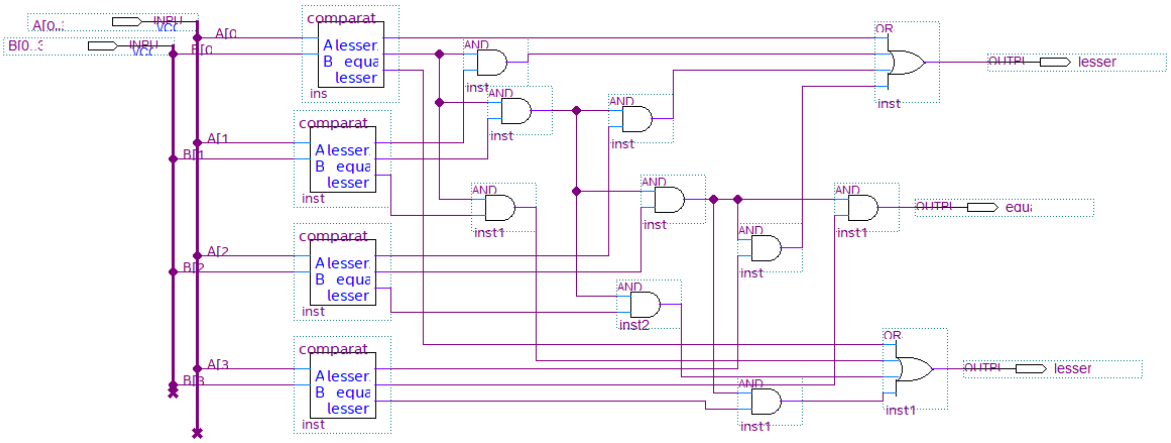

Figure 13: Four Bit Comparator

The operation of the 4-bit comparator is exactly the same as the 1-bit comparator and expected results are summarized in table 11.

Finally, the 8-bit comparator is built using two 4-bit comparators with one module comparing the most significant bits and the other comparing the least significant bits of the 8-bit inputs.

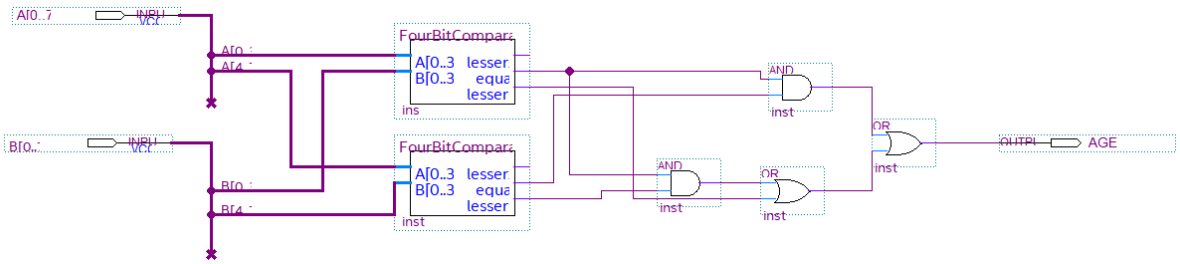

Figure 14: Eight Bit Comparator

The operation of the 8-bit comparator is summarised below:

Table 11: Eight Bit comparator operation

Condition	AGEB
A > B (eg: A = 2, B = 1)	1
A = B (eg: A = 2, B = 2)	1
A < B (eg: A = 2, B = 4)	0

3.2.4 Decoder

In the Register File, a 4-bit address is used to write to one of the 16 registers present in it by asserting the enable signal of the target register and de-asserting the enable signals of all other registers. In order to so, a module is required to get a 4-bit address and decode it into 16 signals of 1-bit each that are connected in order to the registers. Depending on the input, exactly one output signal is set to 1 at any given time.

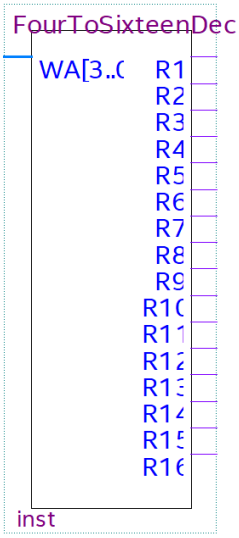

Figure 15: Four to Sixteen Decoder module

Table 12: Four to Sixteen Decoder Module

Module Name: FourToSixteenDecoder	
Input	*Outputs*
WA	***R1 – R16***
4-bit address	1-bit enable signal

The operation of the FourtoSixteenDecoder can be summarised as follows:

Table 13

WA	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16
0000	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0001	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0010	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
0011	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0100	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
0101	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
0110	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0111	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
1000	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
1001	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
1010	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
1011	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
1100	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
1101	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
1110	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
1111	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

To build the FourToSixteenDecoder, two TwoToFourDecoder decoders designed in previous lab exercises are used. The TwoToFour Decoder works similarly to the FourToSixteenDecoder by converting a 2-bit address into 4 enable signals.

Table 14

Module Name: TwoToFourDecoder	
Input	*Outputs*
WA	***R1 – R4***
2-bit address	1-bit enable signal

The operation of the TwoToFourDecoder can be summarized as follows:

Table 15

WA	**R1**	**R2**	**R3**	**R4**
00	1	0	0	0
01	0	1	0	0
10	0	0	1	0
11	0	0	0	1

The decoder is built using logic AND, OR gates to produce the required outcome

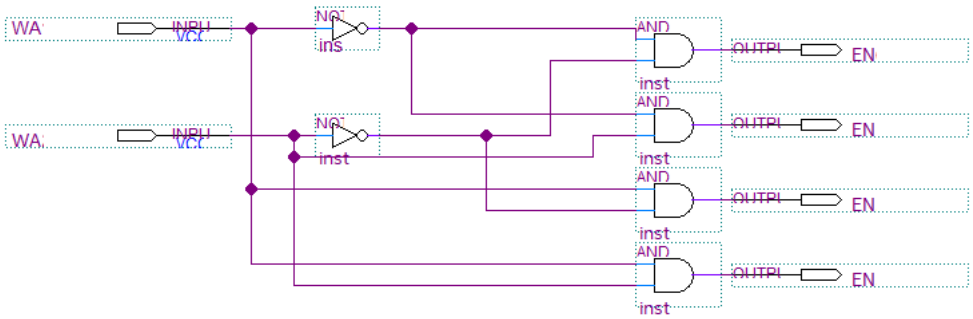

Figure 16: Four to One Decoder

The FourToSixteenDecoder is built with the smaller decoder and some logic gates as follows:

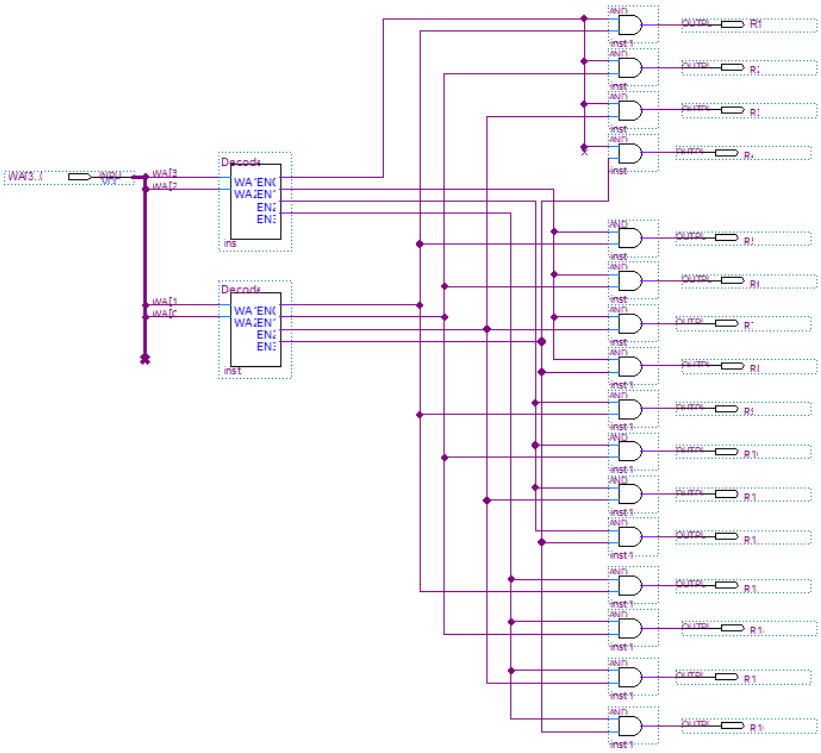

Figure 17: Four to Sixteen Decoder

The Control Unit receives the opcode of an instruction and uses logic gates to produce the required control signals for the datapath. The following signals are produced:

1. PCWrite – Enables/Disables the update functionality of the program counter when asserted/de-asserted respectively
2. branch – asserted when opcode of branch instruction is read as input. It is used to control the input entering the IMM (immediate) port of the program counter via a multiplexer.
3. jump – asserted when opcode of jump instruction is read as input. It is used to control the input entering the IMM (immediate) port of the program counter via a multiplexer.
4. RegWriteEN – enables/disables the write functionality of the register file when asserted/de-asserted respectively
5. BSel - It is used to control the input entering the B (input) port of the ALU via a multiplexer. It is asserted only when the opcode of an I-type function is read as input.

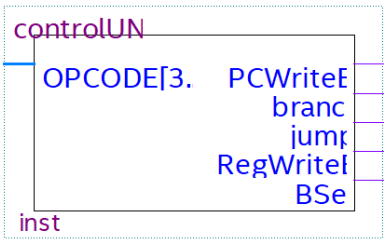

Figure 18: Program Counter Module

Table 16: Program Counter Module

Module Name: ControlUNIT					
Inputs	*Outputs*				
OPCODE	***RegWrite***	***Branch***	***Jump***	***PCWrite***	***Bsel***
4-bit opcode	1-bit signal	1-bit signal	1-bit signal	1-bit signal	1-bit signal

Table 17: Operation of the Control unit

		RegWrite	Branch	Jump	PCWrite	BSel
INSTR	OPCODE	1	0	0	1	0
LW	0001	1	0	0	1	1
ADD	0000	1	0	0	1	1
ADDI	0100	1	0	0	1	1
SRAI	1100	1	0	0	1	1
BGEU	0101	0	1	0	1	0
J	0011	0	0	1	1	0
LUI	1011	1	0	0	1	0

The logic presented by the above table is implemented using AND, OR and NOT logic gates as seen below.

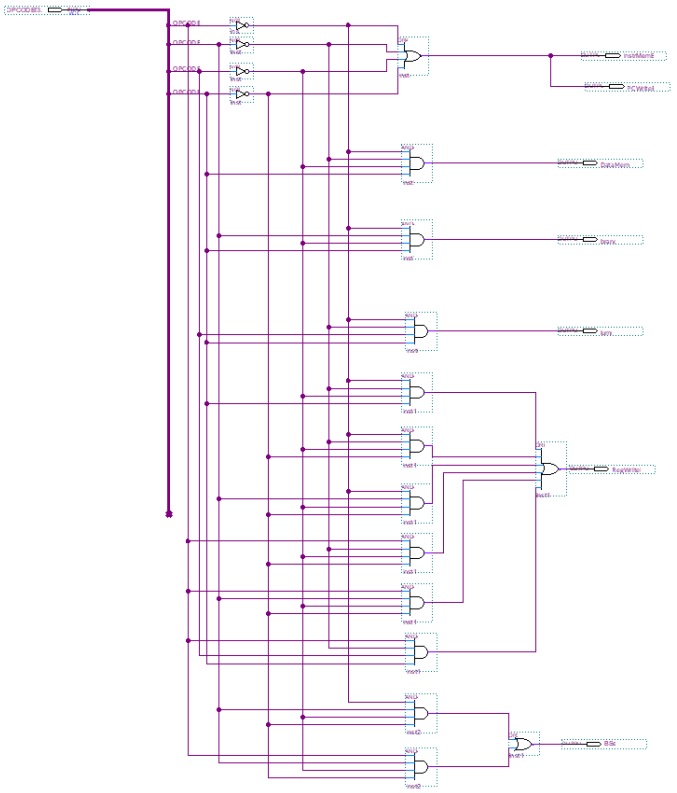

Figure 19: Control Unit

3.2.5 Multiplexers

To read values from the register file, multiplexers are used. These multiplexers take the values of the all the registers and output the value corresponding to the address provided by using it as a select signal. Since the CPU has a 16x8-bit register file, a 16-way 8-bit multiplexer is required.

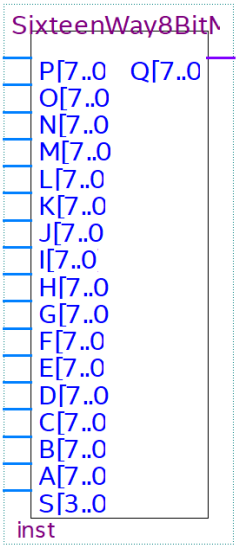

Figure 20: Sixteen Way Eight Bit Multiplexer Module

Table 18: Sixteen Way Eight Bit Multiplexer Module

Module Name: SixteenWay8BitMUX		
Inputs		*Outputs*
A - P	***S***	***Q***
8-bit values	4-bit select signal	8-bit value

The operation of the SixteenWay8BitMUX can be summarized as follows:

Table 19

S3	**S2**	**S1**	**S0**	**Q**
0	0	0	0	A
0	0	0	1	B
0	0	1	0	C
0	0	1	1	D
0	1	0	0	E
0	1	0	1	F
0	1	1	0	G
0	1	1	1	H
1	0	0	0	I
1	0	0	1	J
1	0	1	0	K

1	0	1	1	L
1	1	0	0	M
1	1	0	1	N
1	1	1	0	O
1	1	1	1	P

To build a 16-way 8-bit multiplexer, we first need to design a 16-way 1-bit multiplexer. Eight such 16-way 1-bit multiplexers are then used to choose a single bit each, resulting in a 16-way 8-bit multiplexer.

Table 20: Sixteen Way One Bit Multiplexer Module

Module Name: SixteenWayOneBitMultiplexer		
Inputs		*Outputs*
D0-D15	***S***	***Q***
1-bit values	4-bit select signal	1-bit value

The 16-way 1-bit multiplexer is designed based on the following truth table:

Table 21: Sixteen Way One Bit Multiplexer Operation

S3	**S2**	**S1**	**S0**	**Q**
0	0	0	0	D0
0	0	0	1	D1
0	0	1	0	D2
0	0	1	1	D3
0	1	0	0	D4
0	1	0	1	D5
0	1	1	0	D6
0	1	1	1	D7
1	0	0	0	D8
1	0	0	1	D9
1	0	1	0	D10
1	0	1	1	D11
1	1	0	0	D12
1	1	0	1	D13
1	1	1	0	D14
1	1	1	1	D15

From the truth table, the following Boolean expression is derived:

$$Q = \overline{S3}\,\overline{S2}\,\overline{S1}\,\overline{S0}\,D0 + \overline{S3}\,\overline{S2}\,\overline{S1}\,S0\,D1 + \overline{S3}\,\overline{S2}\,S1\,\overline{S0}\,D2 + \overline{S3}\,\overline{S2}\,S1\,S0\,D3 + \overline{S3}\,S2\,\overline{S1}\,\overline{S0}\,D4 + \overline{S3}\,S2\,\overline{S1}\,S0\,D5 + \overline{S3}\,S2\,S1\,\overline{S0}\,D6 + \overline{S3}\,S2\,S1\,S0\,D7 + S3\,\overline{S2}\,\overline{S1}\,\overline{S0}\,D8 + S3\,\overline{S2}\,\overline{S1}\,S0\,D9 + S3\,\overline{S2}\,S1\,\overline{S0}\,D10 + S3\,\overline{S2}\,S1\,S0\,D11 + S3\,S2\,\overline{S1}\,\overline{S0}\,D12 + S3\,S2\,\overline{S1}\,S0\,D13 + S3\,S2\,\overline{S1}\,S0\,D14 + S3\,S2\,S1\,S0\,D15$$

The multiplexer is designed using AND, NOT and OR gates to implement the logic presented in the Boolean expression as seen in figure 12.

Alternatively, a 16-way 1 bit multiplexer could be designed by using 15 2- way 1-bit multiplexers, where to form a 16-way 1-bit multiplexer. Although this design is conceptually simple, this design is avoided as the it has a higher propagation delay. This is because, each 2-way multiplexer receives its input signal only after the propagation delay of the two multiplexers feeding into it has passed. As a 16-way multiplexer would require about 4 levels of such multiplexers (see figure 21), the propagation delay would be about 4 times the usual.

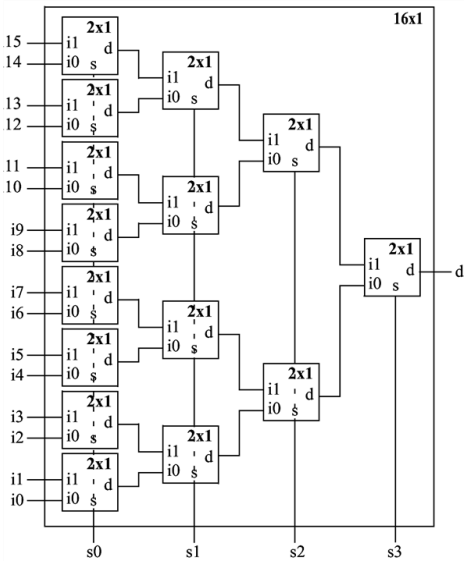

Figure 21: 16-way multiplexer design using 15 2 way multiplexers

Eight such 16-way 1-bit multiplexers are then used to multiplex each individual bit of the eight-bit input value as seen in figure 23.

Other multiplexers used in the datapath include: 8-bit 4-way and 8-bit 2-way multiplexers that work similarly.

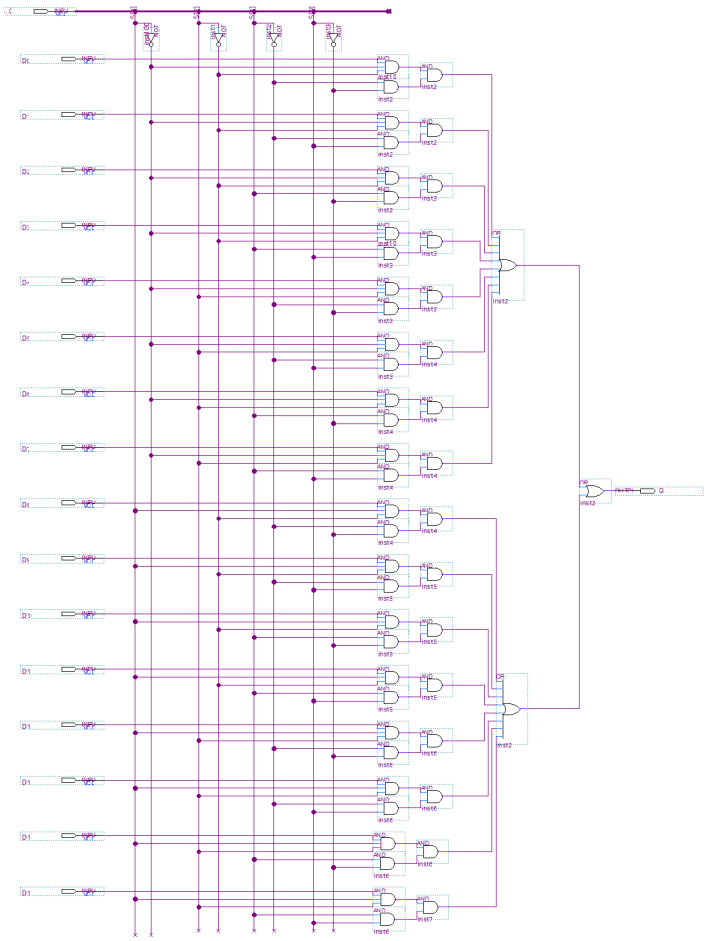

Figure 21: 16 way 1 bit multiplexer

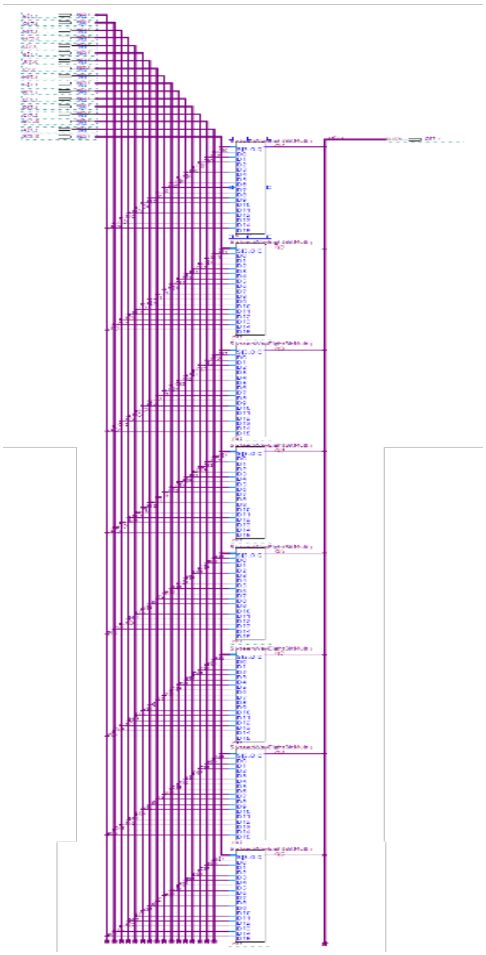

Figure 23: 16 way 8-bit multiplexer

3.2.6 Arithmetic Logic Unit

The ALU in this CPU produced the output of 3 arithmetic operations and 1 logic operation: addition, subtraction, arithmetic right shift by 1 and comparison to see if A is greater than or equal to B depending on the ALU_SEL signal supplied to it. It should be noted that the comparison's result is always available as an output regardless of the ALU_SEL signal supplied.

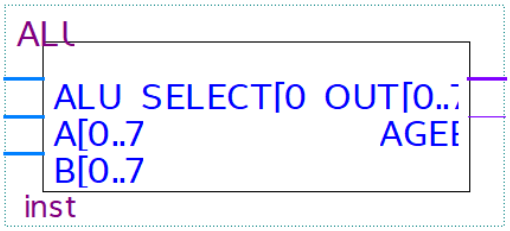

Figure 24: ALU Module

Table 22: ALU Module

Module Name: ALU

Inputs			*Outputs*	
A	***B***	***ALU_SEL***	***OUT***	***AGEB***
8-bit operand	8-bit operand	2-bit control	8-bit result	1-bit signal

The operation of the ALU can be summarized as follows:

Table 23: Operation of the ALU

ALU_SELECT	Operation
00	Addition
01	Subtraction
10	Arithmetic Right Shift
X (does not depend on ALU_SELECT)	Comparison

It should be noted that the comparison's result is always available as an output regardless of the ALU_SEL signal supplied.

The ALU is designed using the Subtractor, RightShift, EightBitComparator and FourWayFourBitMultiplexer modules developed in the previous sections.

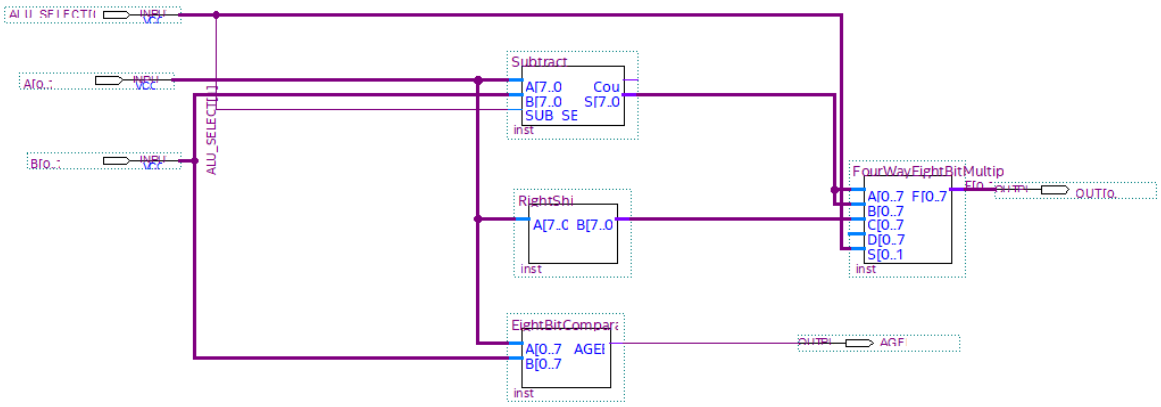

Figure 24: ALU

3.2.7 Read Only Memory

Since both instruction and data memory are never modified, a ROM is used to store the instructions and data required to execute a binary search application. The LPM_ROM megafunction present in the Quartus Prime Lite Software are preloaded with the instructions and data before the program is executed.

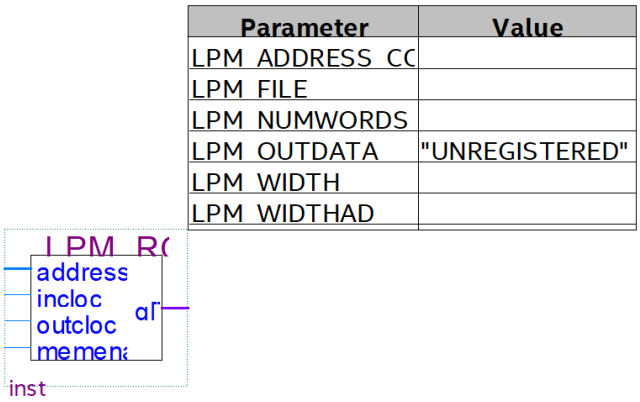

Figure 25: LPM_ROM Module

These modules are configured as:

	Name	Value	Type	Description
1	LPM_ADDRESS_CONTROL	"REGISTERED"		Should the address and control ports be registered?
2	LPM_FILE	"C:/Users/sneha/Downloads/InstrMem.mif"		File containing initial contents of memory array
3	LPM_NUMWORDS	32		Number of memory words, default is 2^LPM_WIDTHAD
4	LPM_OUTDATA	"UNREGISTERED"		Should the output data be registered?
5	LPM_WIDTH	20		Data width in bits, any integer > 0
6	LPM_WIDTHAD	5		Number of address lines, any integer > 0

Figure 26: LMP_ROM Configurations for Instruction Memory

Since data needs to be read from these memory modules on a rising clock edge, the address control is set to “REGISTERED”. The LPM_FILE field specifies a Memory Initialization File that contains the instructions for the instruction memory and the data (32 word array) for the data memory (figure 6 and 7).

The configuration of the instruction memory depends on the number of instructions that need to be executed. From figure 3, it can be seen that the program requires 21 instructions to be executed. To address 21 elements, the number of bits required can be calculated by:

$$\log_2(n) = address\ width$$

Where $n = number\ of\ instructions$

Since, $\log_2(21) = 4.39$, 5 bits are required to address the instruction memory. Consequently, the maximum number of words in memory can be calculated by:

$$2^n = no.of\ memory\ words$$

Where $n = address\ width$

Therefore, the instruction memory can store a maximum of $2^5 = 32$ instructions.

Additionally, since each instruction is 20 bits long (as seen in table 2), the width of each word is set to 20.

	Name	Value	Type	Description
1	LPM_ADDRESS_CONTROL	"REGISTERED"		Should the address and control ports be registered?
2	LPM_FILE	"C:/Users/sneha/Downloads/DataMem.mif"		File containing initial contents of memory array
3	LPM_NUMWORDS	32		Number of memory words, default is 2^LPM_WIDTHAD
4	LPM_OUTDATA	"UNREGISTERED"		Should the output data be registered?
5	LPM_WIDTH	8		Data width in bits, any integer > 0
6	LPM_WIDTHAD	5		Number of address lines, any integer > 0

Figure 27: LPM_ROM Configuration for Data Memory

Similarly, the data memory is also configured with all the necessary parameters.

ıdd	+0	+1	+2	+3	+4	+5	+6	+7
0	11	8219	5163	251	15412	17732	32088	53344
8	23148	6257	39989	22580	291	116789	22852	291
16	251984	323	577717	323	97280	0	0	0
24	0	0	0	0	0	0	0	0

Figure 28: Instruction Memory Initialization File

20 bits are used to store one instruction. The instruction consists of opcode, address of destination register, function code, address of source registers, immediate value. each of the components sits in its designated position and bit range. the binary outcome is then converted into unsigned integer. When fetching instructions, corresponding bit range will be fetched from the bus line and then fed into the next-stage component.

ıdd	+0	+1	+2	+3	+4	+5	+6	+7
0	1	2	4	5	10	14	20	32
8	33	61	71	84	93	110	113	119
16	121	136	138	139	151	155	188	191
24	193	196	204	220	244	245	248	252

Figure 29: Data Memory Initialization file

An assumption of all unsigned and sorted data array is taken to ease the workloads.

3.2.8 Register

Since the CPU deals with 8-bit values, 8-bit registers are designed to store these values. N-bit registers are made using N DFFE's (D Flip-Flops with Enable) and share a clock and enable signal.

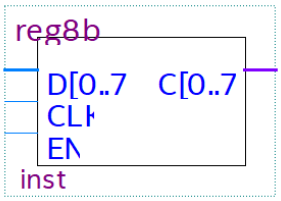

Figure 30: Eight Bit Register Module

Table 24: Eight Bit Register Module

Module Name: reg8bit			
Inputs			*Outputs*
D	***CLK***	***EN***	***C***
8-bit value	1-bit clock signal	1-bit signal	8-bit value

The register writes values only when the ***EN*** signal is asserted and the clock is on a rising edge. The values are read even if the ***EN*** signal is not asserted.

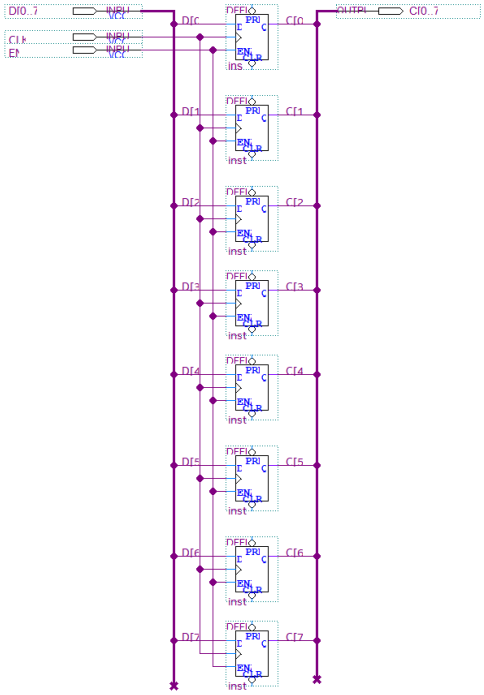

Figure 31: 8-bit register

3.2.9 Register File

The register file contains registers that store the internal state of the CPU. From figure 3, a minimum of 9 registers are required to store the internal state of the CPU and execute the code.

To address 9 registers, the ceiling of $\log_2(9)$ = 4 bits are required. With four address lines, we can address a maximum of $2^4 = 16$ registers. Therefore, a 16x8-bit register file is designed such that it reads two registers and writes one register in one clock cycle.

The schematic representation of the register file is as follows:

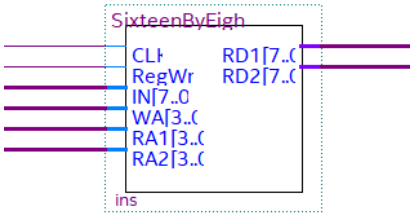

Figure 32: Schematic Diagram of a 16x8-bit register

Table 25: Sixteen By Eight Register File Module

Module Name: SixteenByEightBitRF							
Inputs						*Outputs*	
IN	***WA***	***RA1***	***RA2***	***CLK***	***RegWrite***	***RD1***	***RD2***
8-bit value	4-bit address	4-bit address	4-bit address	1-bit clock	1-bit control	8-bit value	8-bit value

The operation of the 16x8-bit register file can be described as follows: When the ***CLK*** is on a rising edge and the ***RegWrite*** signal is enabled, the 8-bit value presented by ***IN*** is written to the register whose address is given by the 4-bit address ***WA.*** At every rising edge of the ***CLK*** signal, the values at the registers whose 4-bit addresses are given by ***RA1 and RA2*** are read at ports ***RD1 and RD2*** respectively.

To read the values from the registers based on the address provided, a 16-way 8-bit multiplexer is used at each read port. The corresponding 4-bit read address is used as a selection signal to choose between the 16 registers connected to the input of the multiplexer.

To write the values to the correct register, a decoder (see figure 15) is used. The outputs of the decoder are connected to the enable signal of the registers, thereby making sure that only the register that needs to be written to has an asserted enable signal.

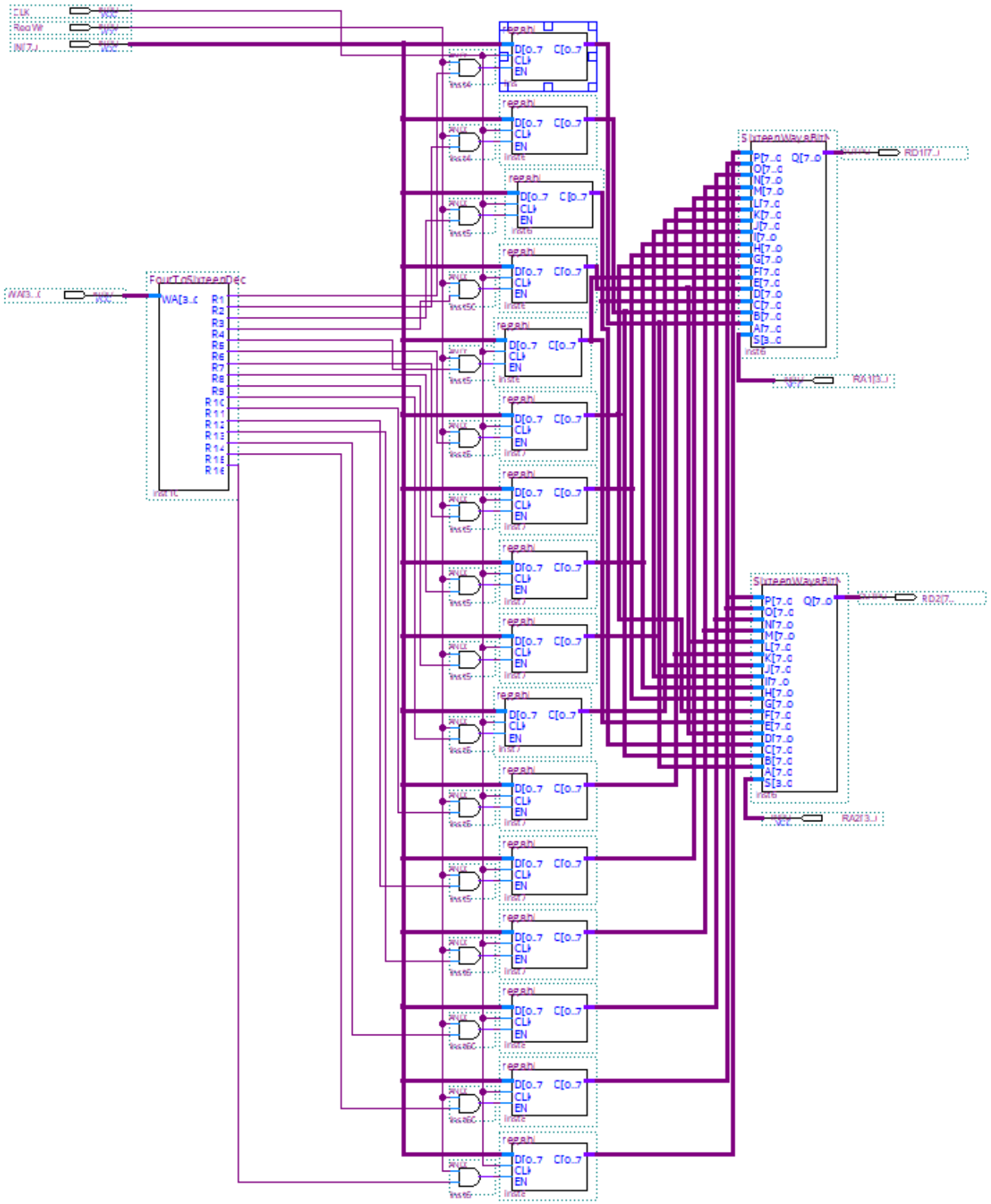

Figure 33: Sixteen By Eight Register File

3.2.10 Program Counter

The Program Counter is a register that stores the Instruction Memory address of the current instruction to be executed. Since the Instruction Memory of this CPU requires a 5-bit address, and since the standard register size in this CPU is 8-bits, an 8-bit program counter is designed. The unnecessary bits are discarded before connecting the output of the PC to the instruction memory.

The program counter uses a PCSrc signal to control whether the PC is incremented by a fixed value K or set to an immediate value. A PCWt signal is also used to determine whether the program counter's value should be updated or not.

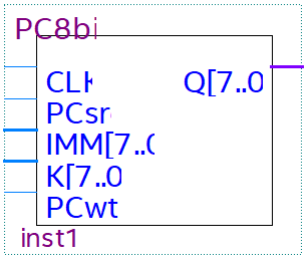

Figure 34: Eight Bit Program Counter

Table 26: Eight Bit Program Counter Module

Module Name: PC8bit					
Inputs					*Outputs*
IMM	***K***	***CLK***	***PCSrc***	***PCWt***	***Q***
8-bit value	8-bit value	1-bit clock	1-bit control	1-bit control	8-bit value

The operation of the ***PC8Bit*** module can be summarized as follows: When the ***PCSrc = 0*** and the ***CLK*** is on a rising edge and the ***PCWt*** signal is asserted, the value of the program counter is incremented by a fixed value ***K.*** Alternatively, when the ***PCSrc = 1***and the ***CLK*** is on a rising edge and the ***PCWt*** signal is asserted, the value of the program counter is set to the immediate value ***IMM.*** If ***PCwt = 0,*** regardless of the value of the input or other control signals, no change occurs in the PC.

To build the PC, an 8-bit adder, 8-bit 2-way multiplexer and 8-bit register are used. The Adder is used to increment the PC by a fixed value K and the multiplexer is used to choose between the incremented value or the immediate value by using PCSrc as a select signal.

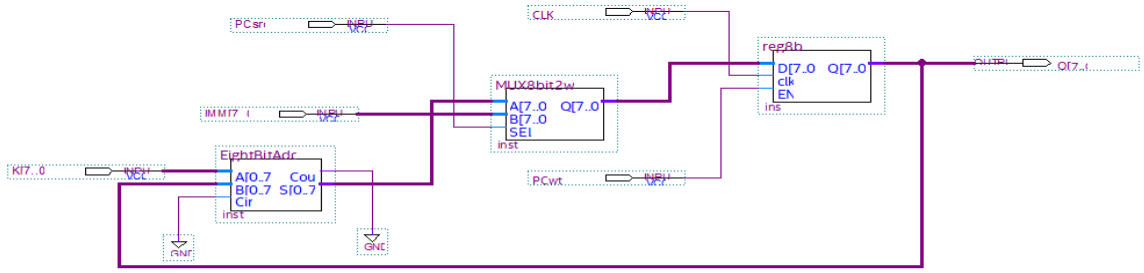

Figure 35: 8-bit Program Counter

3.3 Datapath

The complete datapath for the CPU is built step-by-step by considering each instruction in the ISA and using the components developed in the previous section.

1. R-Type Instruction:

The 20-bit instruction is split into the 5 different values seen above using buses. The 4-bit addresses *RS1* and *RS2* are fed to the *RA1* and *RA2* read address ports of the Register File. The two 8-bit values read from the Register File, *RD1* and *RD2* are in turn fed to the *ALU*. The 2-bit *FUNC* code is used as the *ALU_SEL* signal for the *ALU* to determine which operation to perform. The 8-bit result of the operation is routed from the *ALU* back to the *IN* port of the Register File and is written into the target register whose address, *RD*, is fed to port *WA* of the Register File. The *OPCODE* is fed to the control unit and all necessary control signals are routed to their destination.

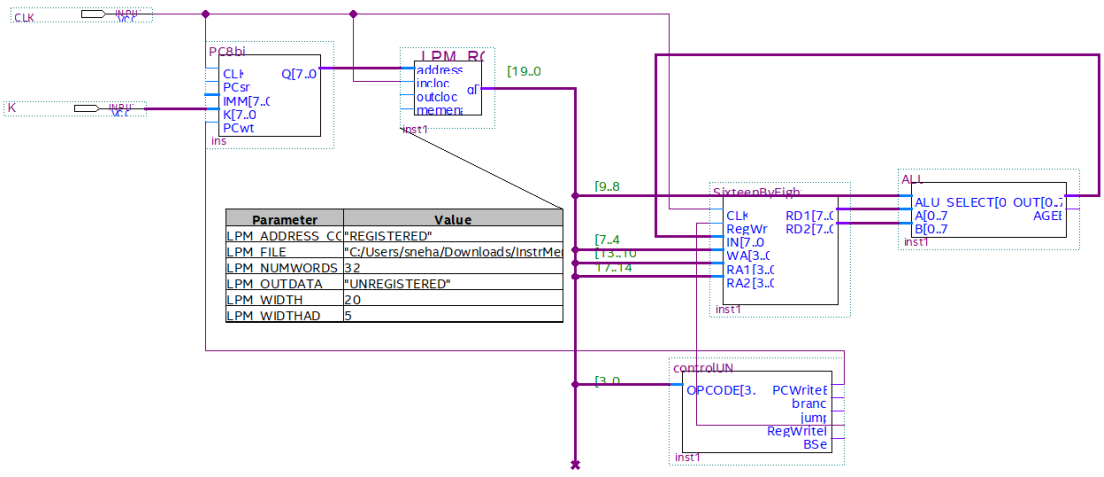

Figure 36: R Type instruction

2. U-Type Instruction:

IMM	RD	OPCODE

The 20-bit instruction is split into 3 different values seen above using buses. The 8 rightmost bits of the immediate value are fed into the *IN* port of the Register file to be written to the Register addressed by *RD* at the *WA* port of the Register File. However, a path already exists between the output of the ALU and the *IN* port of the Register File. In order to allow both inputs to the register file to exist, a multiplexer is introduced before the *IN* port of the register file. The last two bits of the opcode are used as the select signal for this multiplexer.

Table 27

S[0..1]	**F[0..7]**
00	A[0..7] – Output ***OUT*** from the ALU
11	D[0..7] – Immediate value ***INSTR[15..8]***

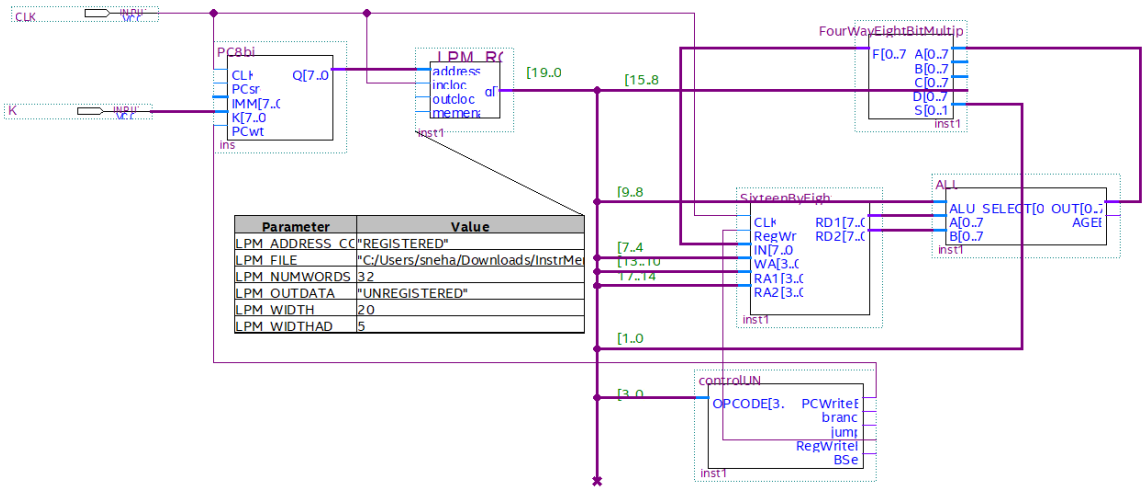

Figure 37: U Type Instruction

3. I-Type Instruction:

IMM	RS1	FUNC	RD	OPCODE

The 20-bit instruction is split into the 5 different values seen above using buses. The 6-bit immediate value IMM is converted into an 8-bit value by adding two bits of the value 0 to the left of the existing bits. This value is then fed into the *B* port of the ALU. However, a path already exists between the *RD2* port of the Register File and the *B* port of the ALU. To allow both inputs to the ALU, a multiplexer is introduced to choose between these two inputs. The *BSel* signal produced by the control unit is used as the select signal for this multiplexer. When *BSel = 0,* the value read from register file is fed to the ALU. When *Bsel = 1,* the immediate value encoded in the instruction from bits 15 to 8 is fed into the ALU.

Additionally, to perform the Load Word instruction, a data memory needs to be included. The address of the word to be loaded is calculated using *RS1 and IMM* being fed to the ALU and the output of the ALU is used as the address. The value read from the data memory is fed into the *IN* port of the Register File through the multiplexer.

Table 28: Working of Multiplexer to select Write Data

S[0..1]	**F[0..7]**
00	A[0..7] – Output ***OUT*** from the ALU
01	B[0..7] – Data value ***q*** from Data Memory
11	D[0..7] – Immediate value ***INSTR[15..8]***

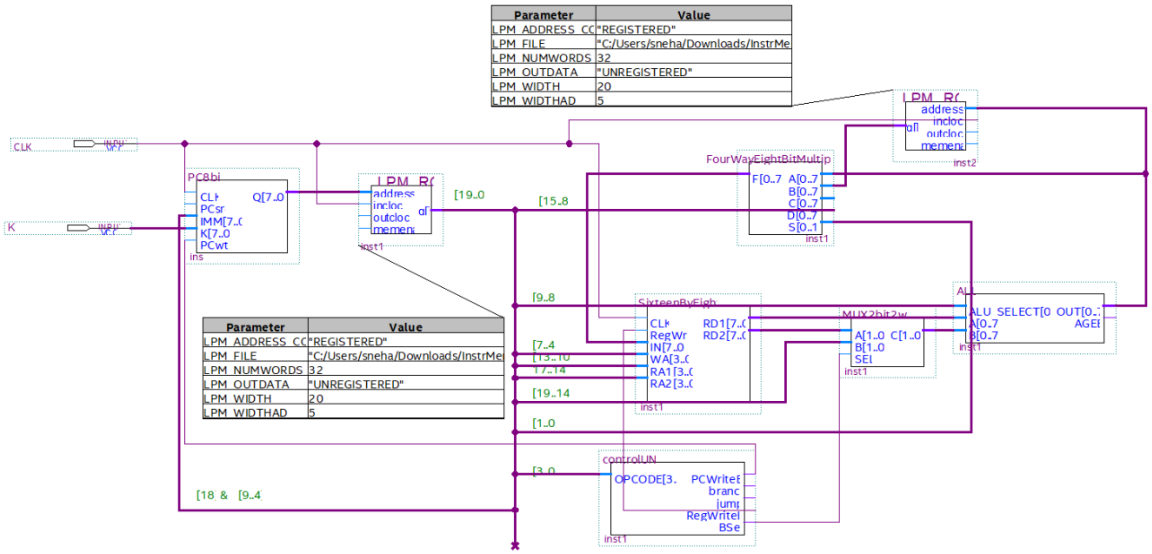

Figure 38: I Type Instruction

4. B-Type Instruction:

IMM	RS2	RS1	IMM	OPCODE

The 20-bit instruction is split into the 4 different values seen above using buses. The 8-bit immediate value IMM is obtained from two different locations in the instruction and is joined together using buses. This immediate value is used to calculate the address that the program counter should jump to if the branch condition is satisfied. To do so, an adder is used to add the current value of the program counter to the branch immediate and this value is fed into the program counter's *IMM* port. To decide whether the *PCSrc* control signal should be asserted or de-asserted, the *AGEB* output of the ALU and the *branch* output produced by the Control Unit are used. If the *branch* signal is asserted AND the branch condition is satisfied (*AGEB* is asserted), the *PCSrc* signal is set to 1. If either of these signals are de-asserted, *PCSrc* is set to 0 and the PC is incremented by K as usual.

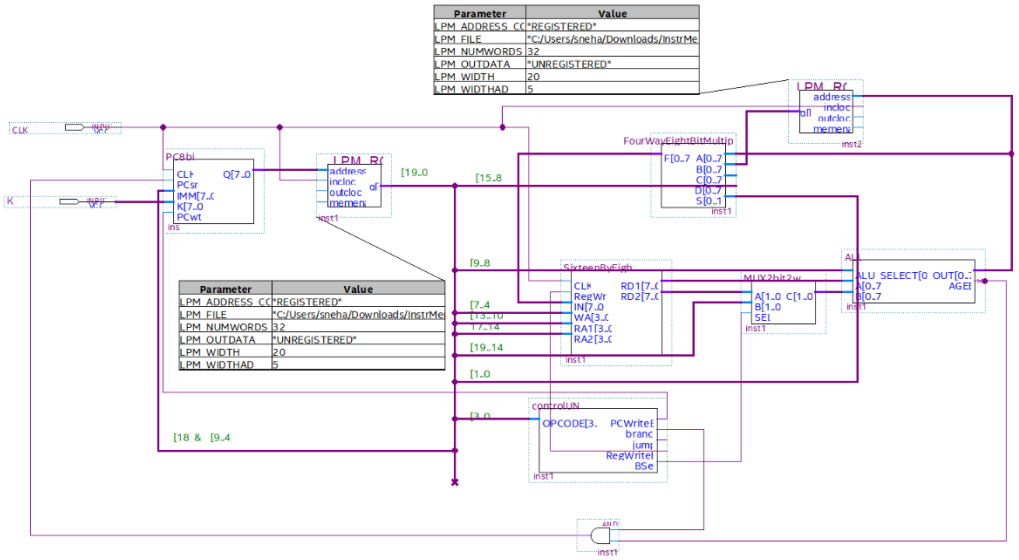

Figure 39: B Type Instruction

5. J-Type Instruction:

IMM	OPCODE

The 20-bit address is split into the 2 values seen above using buses. The 8 rightmost bits of the immediate value contain the address that the program counter should jump to. In order to set the PC to that value, the IMM should be fed to the *IMM* port of the PC. However, the value calculated using the branch immediate is already being fed into this port. To allow both inputs to the *IMM* port of the PC, another multiplexer is used. The *jump* signal from the Control Unit is used to control this new multiplexer. When *jump* is asserted, the jump immediate is written to the *IMM* port of the PC. If not, the branch immediate is written. The *PCSrc* signal is also modified a little to accommodate jump. To ensure that IMM value is selected when either *branch* OR *jump* are asserted, an OR gate is also added.

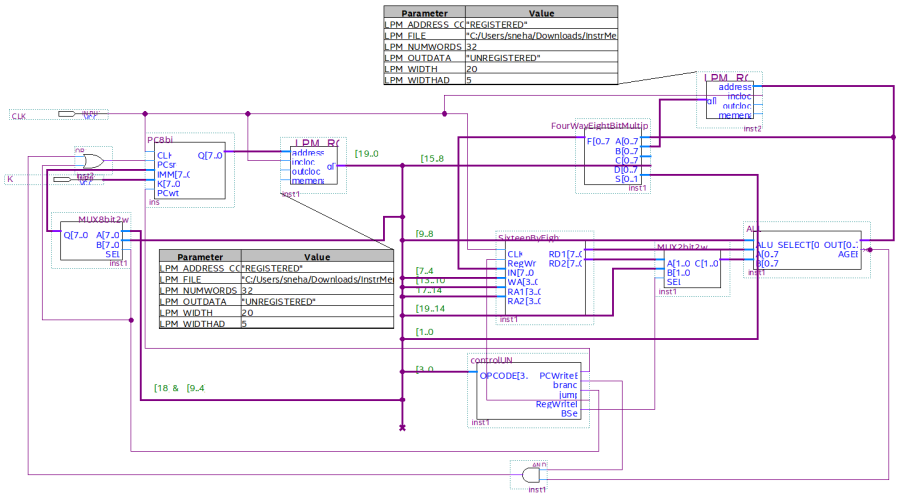

Figure 40: Final Datapath

Therefore, the final datapath can be seen in above figure.

To test whether the CPU is working as intended and executing the instructions correctly, we add output tags to monitor the following values using output pins:

1. The Program Counter
2. The Instruction being read from the Instruction Memory
3. The Data being written into the Register File

The Quartus Prime simulation schematic of the processor is as follows:

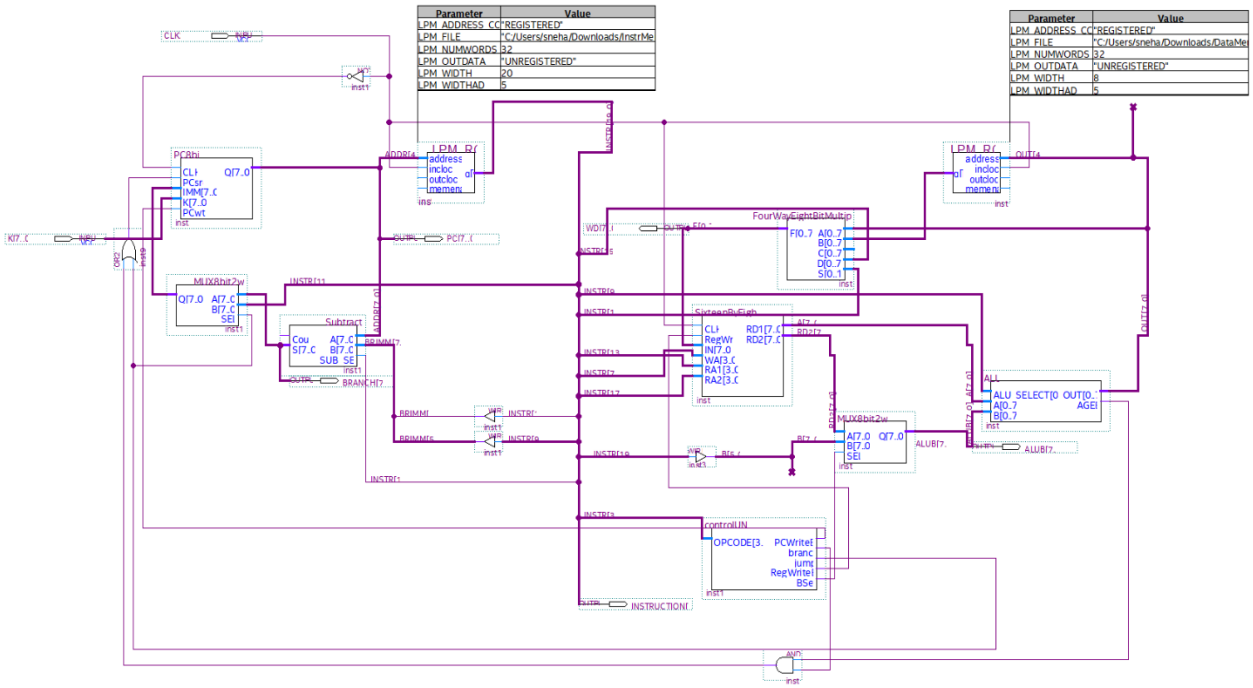

Figure 41: Final Datapath Schematic

4. Results

The designs detailed in the previous section are verified through functional simulations on Quartus Prime Lite. The results are summarized in the following sections.

1. Combinational Logic Elements
    a. Adder/Subtractor
    b. Arithmetic Right Shift
    c. Comparator
    d. Decoder
    e. Control Unit (Instruction Decoder)
    f. Multiplexers
    g. ALU (Arithmetic Logic Unit)
2. State Elements
    a. ROM (Instruction and Data Memory)
    b. Register
    c. Register File

d. Program Counter

4.1 Combinational Logic Elements

4.1.1 Adder/Subtractor

The working of the ***Subtractor*** module is tested by providing the following input values: ***A = 16, B = 4*** and a ***SUB_SEL*** signal that is de-asserted for the first half of the time period and then asserted for the next half. It is expected that the module will perform addition for the first half and produce an output of the value ***S = A + B = 20.*** In the next half, subtraction is performed and an output of ***S = A – B = 12*** is expected.

The simulation waveform can be seen below:

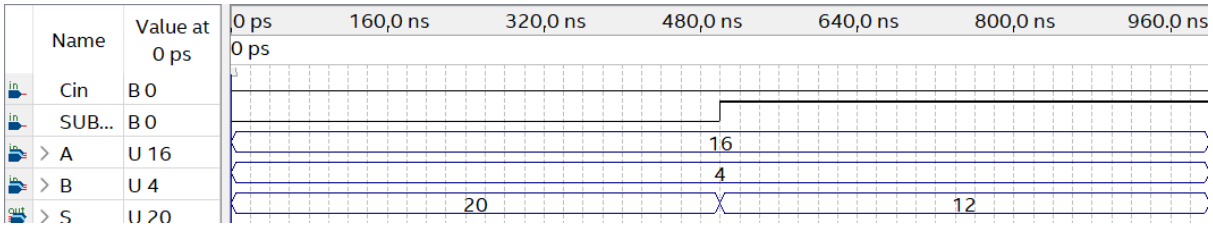

Figure 42: Adder/Subractor Results

Since the functional simulation results correspond with the expected results, we can conclude that the *Subtractor* module works as expected.

4.1.2 Arithmetic Right Shift

The working of the ***RightShift*** module is tested by providing the following input values: ***A = -126***. It is expected that the module will produce an output of ***B = A/2 = -64.***

The functional simulation waveform results can be seen below:

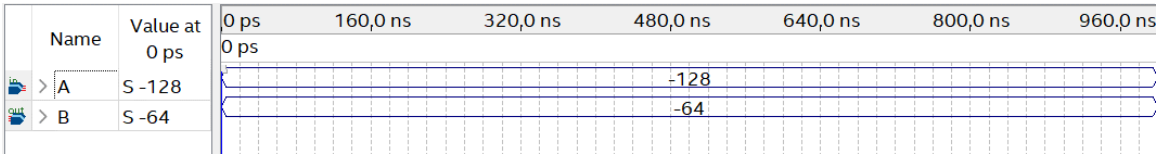

Figure 43: Arithmetic Right Shift Results

Since the functional simulation results correspond with the expected results, we can conclude that the *RightShift* module works as expected.

4.1.3 Comparator

The ***EightBitComparator*** and its sub-components: ***comparator*** and ***FourBitComparator*** are tested by providing varying input signals.

The functional simulation waveform results can be seen below:

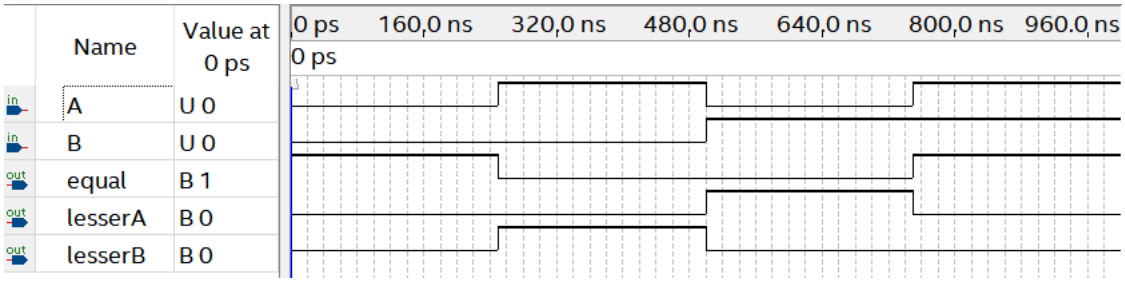

Figure 44: One Bit comparator results

In the *comparator,* it can be seen that when signals ***A & B = 0*** (0 – 250 ns) and ***A & B = 1*** (750ns – 1000ns), the ***equal*** signal is asserted and all other signals are de-asserted since A is equal to B. When signal ***A = 1 & B = 0*** (250ns – 500ns), the ***lesserB*** signal is asserted and all other signals are de-asserted since B is now lesser than A. When signal ***A = 0 & B = 1*** (500ns – 750ns), the ***lesserA*** signal is asserted and all other signals are de-asserted since A is now lesser than B. Since these results correspond with the expected results, we can conclude that the *Comparator* module works as expected.

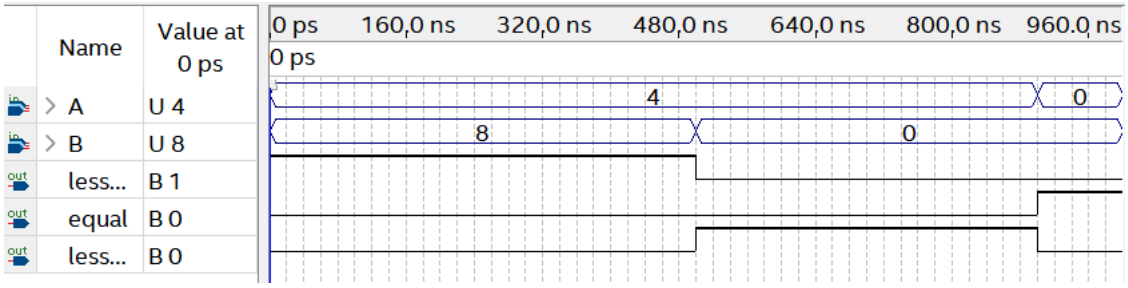

Figure 45: Four Bit comparator results

Similarly, in the *FourBitComparator,* it can be seen that when ***A = 4 & B = 8*** (0 – 500ns), the ***lesserA*** signal is asserted and all other signals are de-asserted since A is lesser than B. When signal ***A = 4 & B = 0*** (500ns – 900ns), the ***lesserB*** signal is asserted and all other signals are de-asserted since B is lesser than A . When signal ***A = 0 & B = 0*** (900ns – 100ns), the ***equal*** signal is asserted and all other signals are de-asserted since A is now equal to B. Since these results correspond with the expected results, we can conclude that the *FourBitComparator* module works as expected.

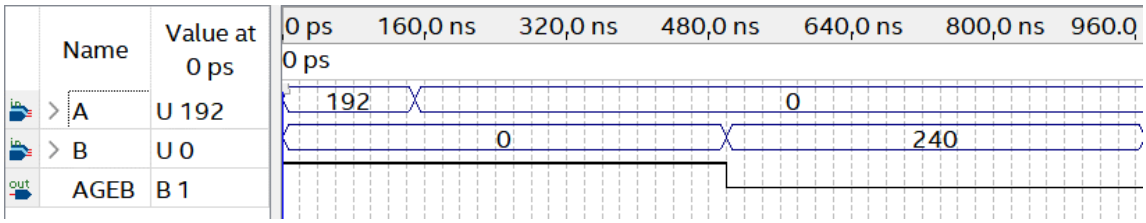

Figure 46: Eight Bit Comparator results

Finally, in the *EightBitComparator,* it can be seen that when ***A = 192 & B = 0*** (0 – 150ns), the ***AGEB*** signal is asserted as A is greater than B. When signal ***A = 0 & B = 0*** (150ns – 500ns), the ***AGEB*** signal is asserted since A is equal to B. When signal ***A = 0 & B = 240*** (500ns – 1000ns), the ***AGEB*** signal is de-asserted since A is now lesser than B. Since these results correspond

with the expected results, we can conclude that the *EightBitComparator* module works as expected.

4.1.4 – Decoder

The ***FourToSixteenDecoder*** and its sub-components ***decoder*** are tested by providing an input signal that begins with value 0 and increments by 1 until the value $n-1$ is reached. Here, n represents the number of outputs of the decoder module.

Assuming that the outputs are numbered from $0\ to\ n-1$, the decoder is expected to assert the $(n-1)^{th}$ output and de-assert all other outputs when the value $n-1$ is supplied at the input.

The functional simulation waveform results can be seen below:

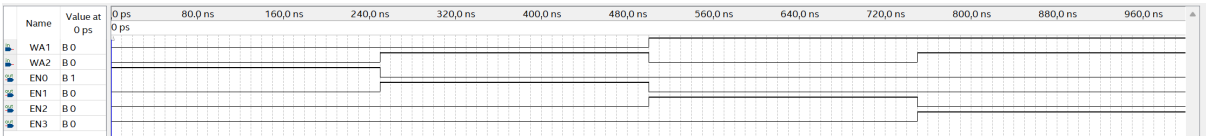

Figure 47: Two to Four Decoder results

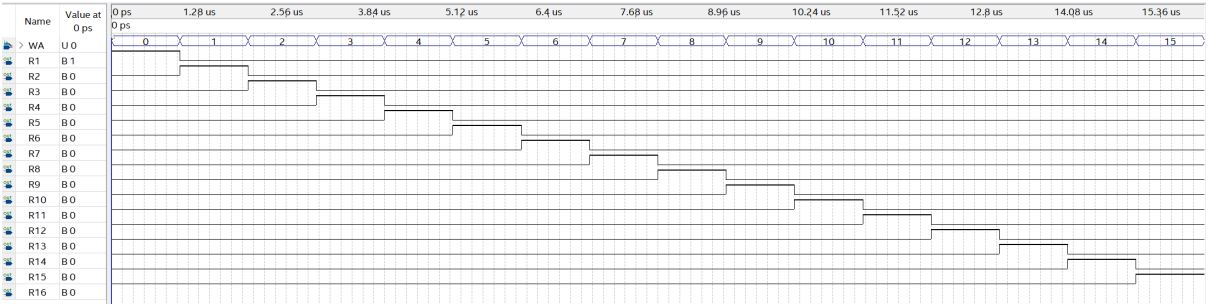

Figure 48: Four to Sixteen Decoder results

Since these results correspond with the expected results, we can conclude that the *FourToSixteenDecoder* and the *Decoder* module work as expected.

4.1.5 Control Unit

To test the functionality of the *ControlUnit,* the opcodes of all 7 instructions in the ISA are provided as inputs. The expected outputs of the module are summarised in table 17. The functional simulation waveform result is seen below:

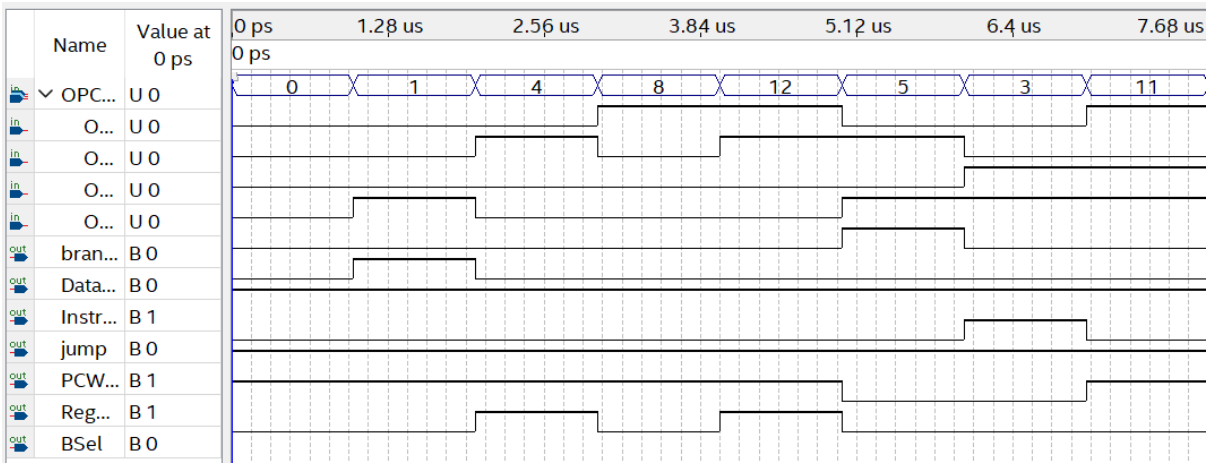

Figure 49: Control Unit Results

Since the results are in line with the expected results, we can conclude that the *ControlUnit* works as expected.

4.1.6 Multiplexers

The ***SixteenWay8BitMUX*** and its sub-component ***SixteenWayOneBitMultiplexer*** are tested in this section.

For the *SixteenWayOneBitMultiplexer,* the select signal ***S*** is configured to start at 0 and increase by the value 1 after each $1\mu s$ until the value 15 is reached. The signal for each input is defined such that it is asserted for exactly $1\mu s$, corresponding to the time when the select signal refers to it. For example, ***D0 = 1*** between 0 and $1\mu s$ and the select signal is also equal to zero here. Then ***D1 = 1*** when ***S = 1, D2 = 1*** when ***S = 2*** and so on. It is expected that the output ***Q*** will be permanently asserted.

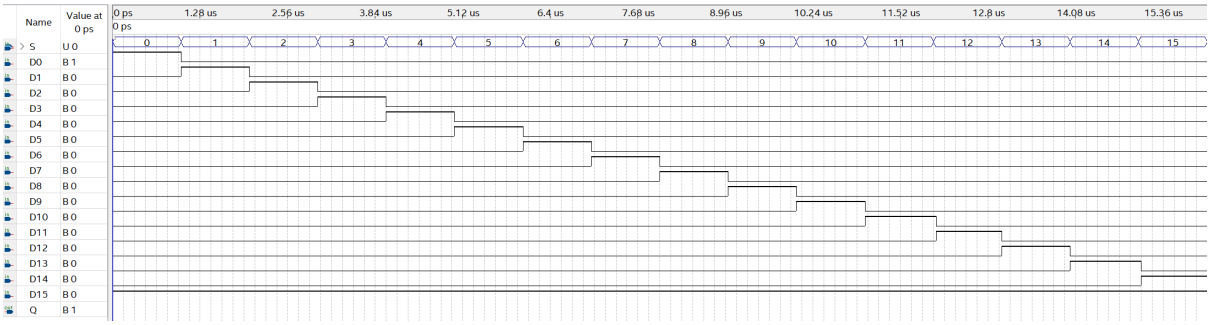

Figure 50: Sixteen Way One Bit MUX results

Since the results match the expected results, we can conclude that the *Sixteen-Way-One-Bit Multiplexer* works as intended.

Similarly, it can be seen from the following waveform that the output ***Q*** is equal to the input whose position is written into the select signal ***S.*** For example, when ***S = 0, Q = A*** and when ***S = 1, Q = B*** and so on.

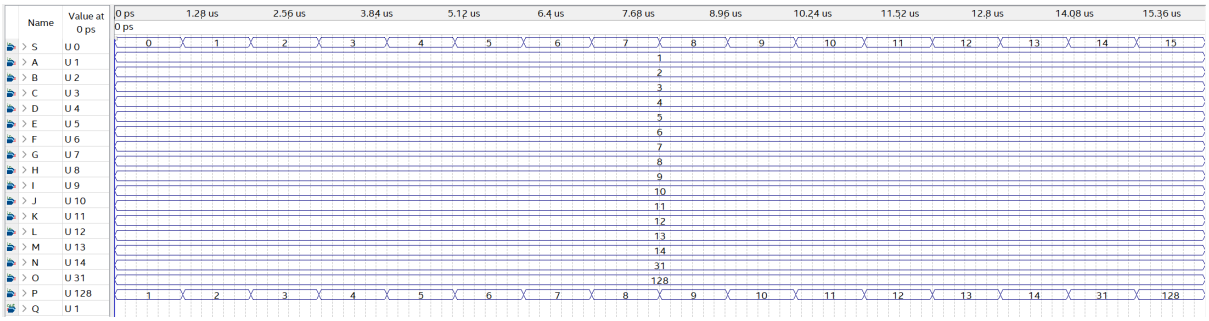

Figure 51: Sixteen Way Eight Bit MUX Results

Since these results correspond with the expected results, we can conclude that the *SixteenWay8BitMUX* works as intended.

4.1.7 ALU

The ***ALU*** produces the value of three different arithmetic/logic operations depending on the control signal ***ALU_SEL*** provided. The A greater than equal to B comparison's value is always made available, regardless of the ***ALU_SEL*** signal provided.

The results of the functional simulation can be seen below:

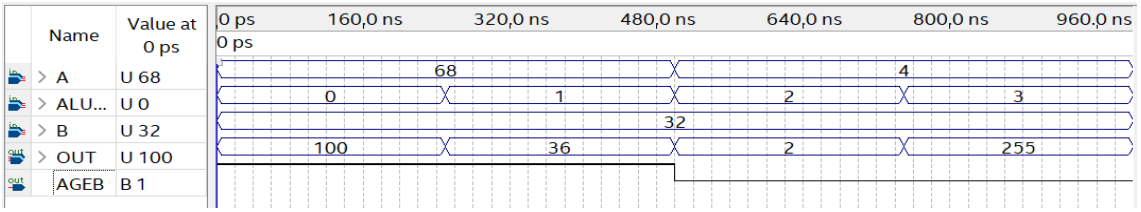

Figure 52: ALU Resutls

From the simulation results it is observed that ***OUT = 100 = A + B*** when ***ALU_SEL = 0.*** When ***ALU_SEL = 1, OUT = 32 = A – B.*** Also, when ***ALU_SEL = 2, OUT = 2 = A/2.*** Since these results comply with the expected results, we can conclude that the *ALU* works as intended.

4.2 State Elements

4.2.1 Register

The register is expected to store the value ***D*** on a rising edge of the clock when the enable signal ***EN*** is asserted. The value stored in the register is read at the port ***C.***

The simulation waveform results for this module can be seen below:

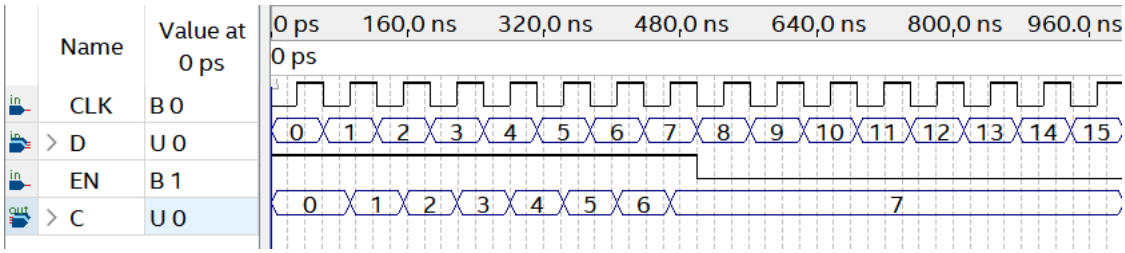

Figure 53: Register Resutls

From the waveform, it is evident that on the first rising edge of the clock, the value 0 is stored in the register since EN is high. This value is also read at port C. On the next rising edge however, the value 1 is stored in the register and this value is read at port C. This behaviour continues until the 500ns mark after which the enable signal is turned off. After this point, the value of the register is never updated regardless of the changes in the input port D and the value read at port C remains constant.

Since this behaviour matches the expected results, we can conclude that the *reg8Bit* module works as intended.

4.2.2 Register File

The ***SixteenByEightBitRF*** is verified using functional simulation as seen below:

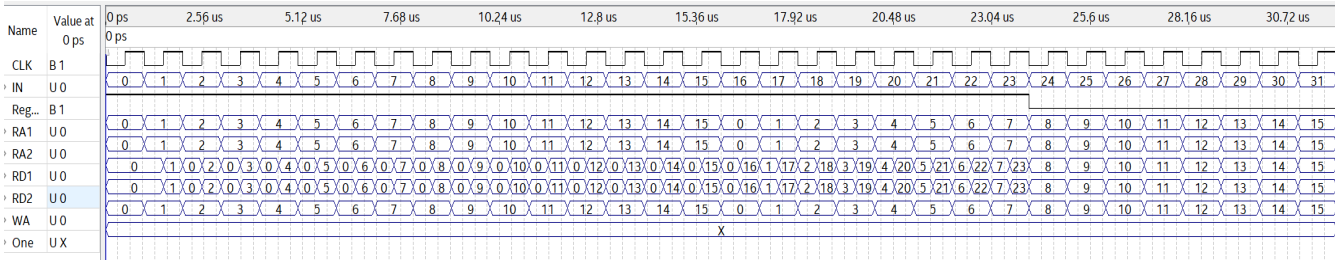

Figure 54: Register File resulst

When the ***RegWrite*** signal is set to high, the value ***IN*** is written to the register addressed by ***WA.*** This behaviour is observed from $0\mu s - 23\mu s$ where the values written into the register are also read using the addresses ***RA1, RA2*** at ***RD1, RD2.*** When the RegWrite signal is de-asserted, the values are no longer written into the registers and the value read at the read ports remain constant.

Since these results comply with the expected results, we can conclude that the *SixteenByEightBitRF* works as intended.

4.2.3 Program Counter

The eight-bit program counter ***PC8bit*** is verified using the functional simulation as seen below:

Figure 11

It is observed that when the ***PCSrc = 0, PCWte = 1*** between $0ns - 250ns$ & $500ns - 700ns$ the PC is incremented by the value ***K = 4.*** When ***PCSrc = 1, PCWte = 1*** between $250ns - 500ns$ the PC is set to the immediate value ***IMM*** provided. When ***PCWte = 0*** between $700\ ns - 1000\ ns$, the PC value is not updated and no value is read.

Since these results correspond to the expected results, we can conclude that the Program Counter works as expected.

The CPU

The schematic developed is tested via functional simulation on Quartus Prime Lite to ensure that the CPU is capable of performing a binary search.

For this test, the parameters are as follows:

- The length of the sorted array on which the CPU will perform the search operation is 32.
- The array contains the following values: 1, 2, 4, 5, 10, 14, 20, 32, 33, 61, 71, 84, 93, 110, 113, 119, 121, 136, 138, 139, 151, 155, 188, 191, 193, 196, 204, 220, 244, 245, 248, 252
- The search target = 20
- The expected result = address of the element 20 = 6

Furthermore, the functionality is verified by observing the Program Counter, the corresponding instructions and the Data written into the register file on relevant instructions.It can be seen that the CPU executes instruction 20 only when the search target has been found or the loop condition is no longer satisfied (i.e., search target does not exist in the array). Thus, by observing the Data being written into the register file when instruction 20 is executed, we can obtain the address of the search target in the sorted array.

To perform the binary search operation correctly, the CPU is expected to execute the instructions in the following order:

Table 29: Instruction Execution Order

Rising Edge No.	**PC**	**Instruction Encoding**	**Instruction**	**Description**
1	0	11	lui a0, 0	$a0 = 0$
2	1	8219	lui a1, 32	$a1 = 32$
3	2	5163	lui a2, 20	$a2 = 20$
4	3	251	lui x0, 0	$x0 = 0$
5	4	15412	addi t0, x0, 0	$t0 = L = 0$
6	5	17732	addi t1, a1, -1	$t1 = R = 31$
7	6	32088	addi a3, x0, -1	$a3 = result = -1$
8	7	53344	add t3, t1, t0	$t3 = R + L = 31$
9	8	23148	srai t3, t3, 1	$t3 = \frac{t3}{2} = 15$
10	9	6257	lw t4, 0(t3)	$t4 = A[M] = 119$
11	10	39989	bgeu t4, a2, elseif	$119 > 20 => branch$
12	13	116789	bgeu a2,t4, else	$20 < 119 => do\ not\ branch$
13	14	22582	addi t1, t3, -1	$t1 = t3 - 1 = 14$
14	15	291	j endif	$jump$
15	18	577717	bgeu t1, t0, loop	$14 > 0 => branch$
16	7	53344	add t3, t1, t0	$t3 = R + L = 14$
17	8	23148	srai t3, t3, 1	$t3 = \frac{t3}{2} = 7$
18	9	6257	lw t4, 0(t3)	$t4 = A[M] = 33$
19	10	39989	bgeu t4, a2, elseif	$33 > 20 => branch$
20	13	116789	bgeu a2,t4, else	$33 < 119 => do\ not\ branch$
21	14	22582	addi t1, t3, -1	$t1 = t3 - 1 = 6$
22	15	291	j endif	$jump$

23	18	577717	bgeu t1, t0, loop	$14 > 0 => branch$
24	7	53344	add t3, t1, t0	$t3 = R + L = 6$
25	8	23148	srai t3, t3, 1	$t3 = \frac{t3}{2} = 3$
26	9	6257	lw t4, 0(t3)	$t4 = A[M] = 5$
27	10	39989	bgeu t4, a2, elseif	$5 < 20 => do\ not\ branch$
28	11	22580	addi t0, t3, 1	$t0 = t3 + 1 = 4$
29	12	291	j endif	$jump$
30	18	577717	bgeu t1, t0, loop	$6 > 4 => branch$
31	7	53344	add t3, t1, t0	$t3 = R + L = 10$
32	8	23148	srai t3, t3, 1	$t3 = \frac{t3}{2} = 5$
33	9	6257	lw t4, 0(t3)	$t4 = A[M] = 14$
34	10	39989	bgeu t4, a2, elseif	$14 < 20 => do\ not\ branch$
35	11	22580	addi t0, t3, 1	$t0 = t3 + 1 = 6$
36	12	291	j endif	$jump$
37	18	577717	bgeu t1, t0, loop	$6 = 6 => branch$
38	7	53344	add t3, t1, t0	$t3 = R + L = 12$
39	8	23148	srai t3, t3, 1	$t3 = \frac{t3}{2} = 6$
40	9	6257	lw t4, 0(t3)	$t4 = A[M] = 20$
41	10	39989	bgeu t4, a2, elseif	$20 = 20 => branch$
42	13	116789	bgeu a2,t4, else	$20 = 20 => branch$
43	16	251984	add a3, x0, t3	$a3 = t3 + 0 = 6$
44	17	323	j end	$jump$
45	20	97280	add a0, x0, a3	$a0 = a3 + 0 = 6$

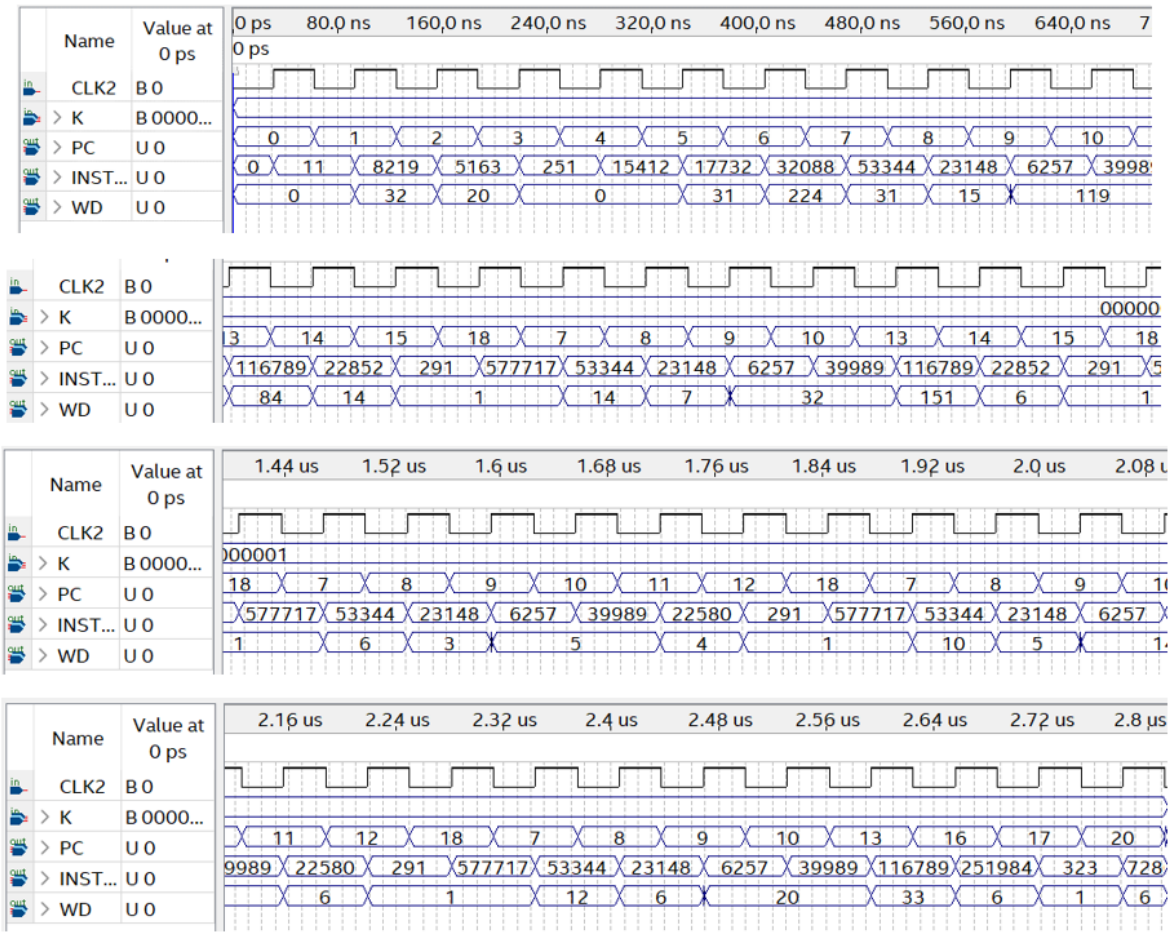

Figure 54: Datapath Simulation Results

At instruction 20, the value ‘6’ is written into the register a0. This means that the search target 20 is found at index 6 of the array. From further observation of the waveform, it can also be seen that the order of execution of the instructions complies with the expected order in table 29. Therefore, we can conclude that the CPU successfully performs the binary search function.

5 Discussion and Conclusion

In this project, an Instruction Set Architecture was developed to perform a binary search operation. A CPU capable of executing the instructions defined in the ISA was successfully developed, simulated and verified.

A major hurdle in ensuring that the CPU successfully executed the instructions was timing. When the datapath was simulated with a single clock signal, it was observed that on the first

clock cycle (rising edge) of the clock, the program counter was incremented and in the next clock cycle, the instruction was fetched and decoded. This caused issues with the branch and jump instructions as the program counter had already incremented to point to the next instruction before the branch and jump instructions are executed, causing the CPU to jump to incorrect places in the code as seen below.

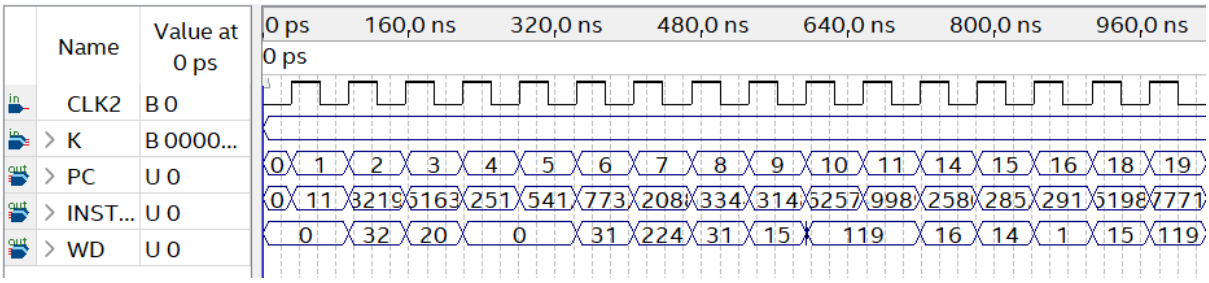

Figure 55: Incorrect Instruction Order Example.

To avoid this issue, the CPU needed to be forced to fetch, decode and execute the instruction before the program counter is incremented. To do so, the clock signal is sent through a not gate before being fed into the program counter. This sends a clock signal that is to the Program counter, ensuring that the counter is incremented only after the instruction is executed .

However, this means that the instruction memory and program counter run on different clock signals. To avoid that, the following alternative was attempted:
A 20-bit register was added after the instruction memory to store the instruction fetched from it. This register was disabled when branch or jump function was being executed to prevent the next instruction from being incorrectly executed. However, this does not solve the problem as the register further causes delays of an extra clock cycle. Therefore, two clock signals were implemented.

6 References

[1] Binary search algorithm, https://en.wikipedia.org/wiki/Binary_search_algorithm, [online], Accessed 2022-09-14.

[2] Harvard Architecture, https://en.wikipedia.org/wiki/Harvard_architecture, [online], Accessed 2022-10-10

[3] Instruction Cycle, https://en.wikipedia.org/wiki/Instruction_cycle, [online], Accessed 2022-10-10

7 Appendix
