Department of Electrical & Computer Engineering

ENCS4760 - Computer Architecture

Single Cycle Processor Implementation

Prepared by:

Ahmad Khateeb	1182828
Tareq Shannak	1181404
Anas Nimer	1180180
Ali Malluh	1180256

Instructor: Dr. Aziz Qaroush

Date: June 12, 2022

Abstract

This report discusses the design, and implementation of a *Singe Cycle Processor* according to a given *RISC* instruction set. The design was done by analyzing each instruction, and notice main components it needs. The implementation was done by building the components, then derive the control signals for them. Testing was done for each instruction, and then by applying complete scenarios to the processor and validate the result.

Contents

List of Figures

List of Tables

1 Design Specification

1.1 Motivation

MIPS architecture is a Reduced Instruction Set Computer (RISC) architecture that has been designed by MIPS Technologies, the main architecture of MIPS is MIPS32 which is 32-bit architecture processor. In this project, we aim to implement a similar instruction set, as a single cycle processor. The instruction set has the following properties:

- The instruction size is 24 bits.
- All instructions are conditionally executed.
- There are eight 24-bit general-purpose registers: R0 through R7.
- R0 is hardwired to zero and cannot be written.
- The program counter (PC) is a 24-bit special-purpose register.
- Zero flag is set only by SUB, and CMP instructions.
- Three instruction types (R-type, I-type, and J-type).
- Five addressing modes as in MIPS32 ISA.

1.2 Instruction Formats

The instruction set has the following instruction formats:

1.2.1 R-Type

Cond^2	Op^5	SF^1	Rd^3	Rs^3	Rt^3	Unused^7

Where,

- cond^2: is the condition bits, which determines if the instruction will be executed or not.
- Op^5: is the opcode, which determines the instruction to be exectued.
- SF^1: is the set-flag, if the set-flag was set to 1, then current instruction will affect the Zero-flag. In this project, the set flag was handled for SUB instructions only.
- Rd^3: is the destination register address, which is the register to write on the result of the instruction.
- Rs^3: is the address of first operand register.
- Rt^3: is the address of second operand register.

1.2.2 I-Type

Cond^2	Op^5	SF^1	Rt^3	Rs^3	Immediate^{10}

The instruction format differs from the R-type by,

- Rt^3: is the destination register.
- Rs^3: is the address of first operand register.
- Immediate^{10}: Signed immediate value in two's complement representation.

1.2.3 J-Type

Cond^2	Op^5	Immediate^{17}

1.2.4 Condition Bits

The condition bits presented in each instruction are decoded as shown in Table 1

Table 1: Condition bits decoding

Condition Bits	Meaning
00	Always execute the current instruction. In assembly, there is no change on the instruction mnemonic. For example, ADD R1, R2, R3 is always executed.
01	Execute if equal, i.e., execute the current instruction if the zero-flag bit is set. Otherwise, the current instruction can be treated as a NOP (no operation). This can be reflected in assembly by appending EQ suffix to the instruction mnemonic, such as, ADDEQ, ANDEQ, SUBEQ etc..
10	Execute if not equal, i.e., execute the current instruction if the zero-flag bit is not set. Otherwise, the current instruction can be treated as a NOP (no operation). This can be reflected in assembly by appending NE suffix to the instruction mnemonic, such as, ADDNE, ANDNE, SUBNE etc..
11	Unused

1.3 Instruction Set

Table 2 shows the instructions supported by this instruction set, with their meaning and decoding.

Table 2: Instruction Set

R-Type						
AND	Reg(Rd) = Reg(Rs) & Reg(Rt)	00000	Rs	Rt	Rd	Unused
CAS	Reg(Rd) = max(Reg(Rs), Reg(Rt))	00001	Rs	Rt	Rd	Unused
Lws	Reg(Rd) = M[Reg(Rs) + Reg(Rt)]	00010	Rs	Rt	Rd	Unused
ADD	Reg(Rd) = Reg(Rs) + Reg(Rt)	00011	Rs	Rt	Rd	Unused
SUB	Reg(Rd) = Reg(Rs) − Reg(Rt)	00100	Rs	Rt	Rd	Unused
CMP	Z = Reg(Rs) < Reg(Rt)	00101	Rs	Rt	000	Unused
JR	PC = Reg(Rs)	00110	Rs	000	000	Unused
I-Type						
ANDI	Reg(Rt) = Reg(Rs) & Immediate^{10}	00111	Rs	Rt	Immediate^{10}	
ADDI	Reg(Rt) = Reg(Rs) + Immediate^{10}	01000	Rs	Rt	Immediate^{10}	
LW	Reg(Rt) = M[Reg(Rs) + Immediate^{10}]	01001	Rs	Rt	Immediate^{10}	
SW	M[Reg(Rs) + Immediate^{10}] = Reg(Rt)	01010	Rs	Rt	Immediate^{10}	
BEQ	PC = BTA if Reg(Rs) = Reg(Rt)	01011	Rs	Rt	Immediate^{10}	
J-Type						
J	PC = PC[23:19] $\Vert$ Immediate^{17}	01100	Immediate^{17}			
JAL	R7 = PC + 3, PC = PC[23:19] $\Vert$ Immediate^{17}	01101	Immediate^{17}			
LUI	R1 = Immediate^{17} << 4	01110	Immediate^{17}			

2 Components

After analyzing the instruction set, we found that the following components are needed.

2.1 Instruction Memory

The memories in the implementation are separated into two parts, instruction memory, and data memory. This was done to solve some conflicts, such as, one instruction might be fetching the instruction from the memory and the other instruction is loading/storing some data from/to the memory, so in order to obey the isolation principle, they need to be separated into different memory elements.

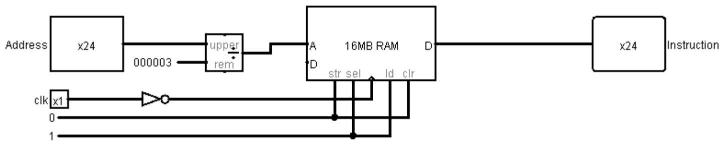


Figure 1: Instruction Memory

Figure 1 shows the implementation of instruction memory using a *Random Access Memory*, where the 24-bit address is divided by 3 to make the memory byte-addressable, which make it easier for compilers and operating systems to deal with addresses.

2.2 Data Memory

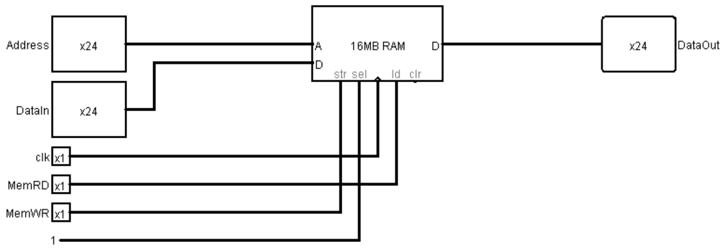


Figure 2: Data Memory

Figure 2 shows the implementation of data memory using a *Random Access Memory*, to support read and write, which is done through and address bus to determine where to write or read, and a data bus to determine the data to write in case of writing to memory.

2.3 Register File

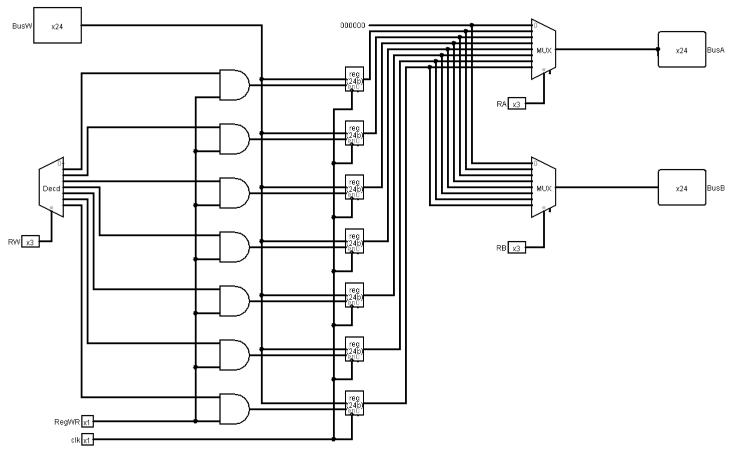


Figure 3: Register File

Figure 3 shows a register file with 8 registers, where register 0 is hardwired to 0 always. To read from register, the address of the register is entered to a multiplexer, which selects the output bus associated with that register. To write to a register, the register address is entered to a decoder which enables writing to that register and the data bus in entered then.

2.4 Extender

The extender was divided into two extenders, one for extending a 10-bit immediate, while the other for extending 17-bit immediate.

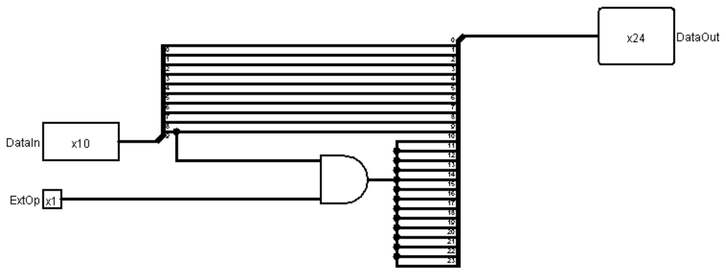


Figure 4: 10-bit extender

Figure 4 shows an extender which takes the 10-bit immediate to extend, and the extending type (signed or unsigned) and outputs a 24-bit extended immediate.

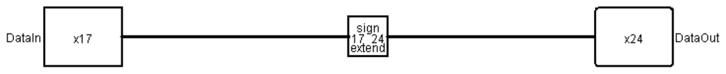


Figure 5: 17-bit extender

Figure 5 shows an extender which takes the 17-bit immediate to extend, and the extending type (signed or unsigned) and outputs a 24-bit extended immediate.

2.5 Arithmetic Logic Unit

Shortly called *ALU*, which is responsible for calculating and doing arithmetic and logical operations.

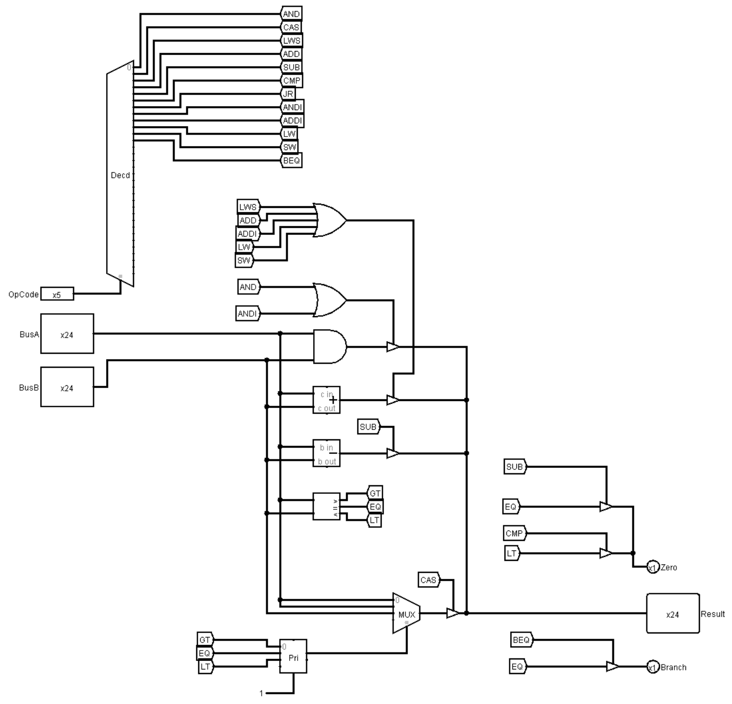


Figure 6: ALU

Figure 6 shows the implementation of the ALU, where all components are working without enable, but the opcode decides which result to go out to the result bus, by using a three state buffer. We notice also that the only CMP and SUB instructions are the only instructions that affect the zero flag. The ALU also, outputs a signal (Branch) that determines whether to go to branch target address or not.

2.6 Control Unit

2.6.1 Truth Table

After building the datapath, the control signals were derived as shown in Table 3

Table 3: Control Signals

Instruction	MemRD	RegWr	ExtOp	ALUSrc	RegDst	RegSrc	MemWR	WBData	PCSrc
AND	0	1	X	0	00	0	0	00	11
CAS	0	1	X	0	00	0	0	00	11
LWS	1	1	X	0	00	0	0	01	11
ADD	0	1	X	0	00	0	0	00	11
SUB	0	1	X	0	00	0	0	00	11
CMP	0	0	X	0	XX	0	0	XX	11
JR	0	0	X	X	XX	0	0	XX	00
ANDI	0	1	0	1	00	0	0	00	11
ADDI	0	1	1	1	00	0	0	00	11
LW	1	1	1	1	00	0	0	01	11
SW	0	0	1	1	XX	1	1	XX	11
BEQ	0	0	1	0	XX	1	0	XX	10
J	0	0	X	X	XX	X	0	XX	01
JAL	0	1	X	X	10	X	0	10	01
LUI	0	1	X	X	01	X	0	11	11

2.6.2 Boolean Expression

By using Table 3 the following boolean expressions were derived

$$\begin{aligned}
MemRD &= LW + LWS \\
RegWR &= \overline{(CMP + JR + SW + BEQ + J)} \\
ExtOp &= \overline{ANDI} \\
ALUSrc &= ANDI + ADDI + LW + SW \\
RegDst &= JAL \mid\mid LUI \\
RegSrc &= BEQ + SW \\
MemWR &= SW \\
WBData &= (LUI + JAL) \mid\mid (LUI + LW + LWS) \\
PCSrc &= \overline{(J + JAL + JR)} \mid\mid \overline{(JR + BEQ)}
\end{aligned}$$

2.6.3 Handling Conditions

To handle the condition bits, a 4x1 multiplexer was used, with inputs $1, Zero, \overline{Zero}, 1$, and the selection bits were the condition bits. The output of the multiplexer is a bit called *Execute*, which was used to determine whether to execute the current instruction or not.

The execute bit, added to the logic of signals (RegWR, MemRD, MemWR, PCSrc) to handle the execution. The modified expressions are as following:

$$\begin{aligned}
MemRD &= (LW + LWS)\ \&\ Execute \\
RegWR &= \overline{(CMP + JR + SW + BEQ + J)}\ \&\ Execute \\
MemWR &= SW\ \&\ Execute \\
PCSrc &= \overline{((J + JAL + JR)\ \&\ Execute)} \mid\mid \overline{((JR + BEQ)\ \&\ Execute)}
\end{aligned}$$

2.6.4 Handling Zero Flag

Since the Zero flag is only affected by CMP and SUBSF instructions, a new control signal was introduced to enable the writing to the zero flag register, this signal was called *ZeroEN* and is expressed as:

$$ZeroEN = CMP + (SF \,\&\, SUB)$$

2.6.5 Implementation

Referring to previous sections, Figure 7 shows the implementation of control unit to output the control signals needed.

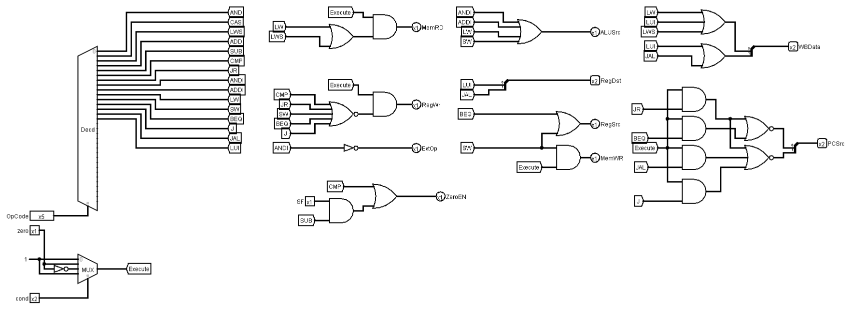


Figure 7: Control Unit

2.7 Complete Datapath

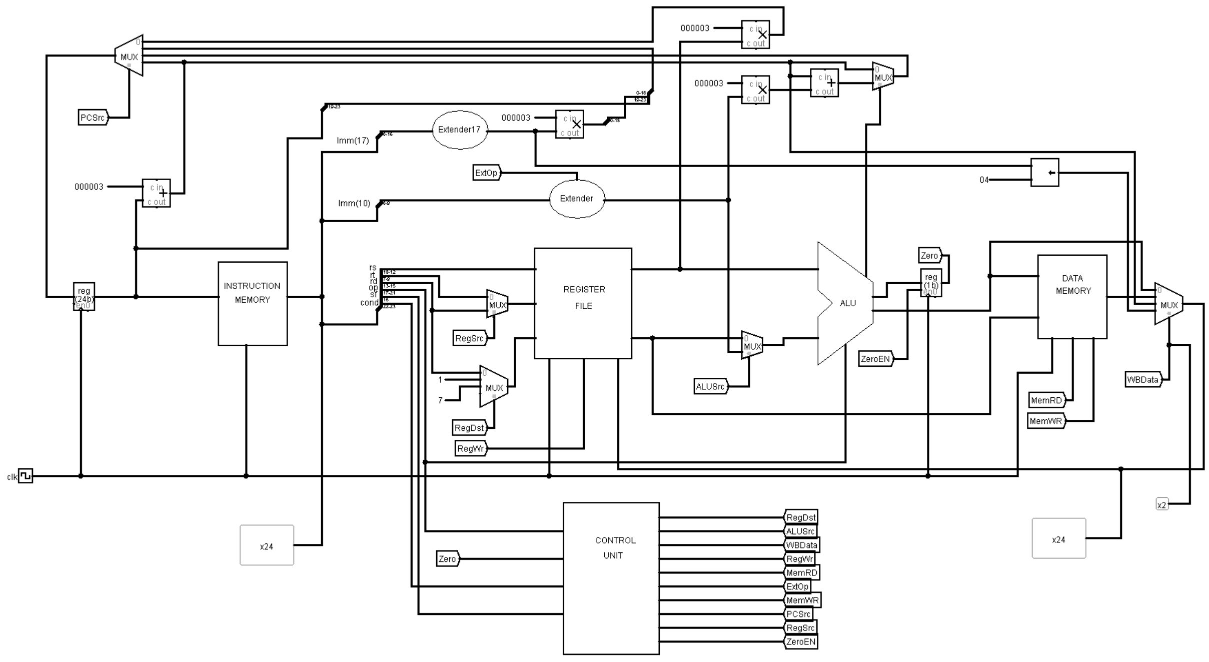


Figure 8: Datapath

Figure 8 shows the datapath, which can be divided into:

2.7.1 Instruction Fetch

The instruction is fetch from instruction memory using the address from the PC. the PC address can be updated to 4 different values:

PCSrc	Next PC
00	PC = Address from JR(Rs)
01	PC = Jump Address
10	PC = Branch Target Address
11	PC = PC + 1

Table 4: PC Sources

2.7.2 Instruction Decode

Here the instruction is decoded, into *Cond, Opcode, SF, Rs, Rt, Rd, Immediate*^{10}, *Immediate*^{17}, then based on the control signals, the registers are read from the Register File. This stage finishes after passing the values of registers, and Immediate into the Execution stage.

2.7.3 Execute

This stage involves performing any operations needed by the instruction, using the ALU unit. And the stage finishes after passing values to the Memory stage.

2.7.4 Memory

Reading or writing or nothing can happen in the memory based on the control signals derived above.

2.7.5 Write Back

The write back stage is responsible of writing data back to Register File, the data to be written back can be the result from ALU, or from Memory, or an Immediate in case of LUI instruction, and may be next PC value in case of JAL instruction.

3 Testing

A unit testing was done for each component, and for each instruction. The following test cases are scenarios of multiple instructions executed one after another.

3.1 Test Case 1

The following code was written in high level assembly:

```
ADDI r1, r0, 5              ; r1 <- 5
ADDI r2, r0, 5              ; r2 <- 5
SUBSF r3, r1, r1          ; r3 <- 5 - 5
BEQEQ r1, r2, 1              ; Branch if zero = 1, and r1 = r2
ADDI r4, r0, 4              ; r4 <- 4 if didn't branch
ADDI r5, r0, 5              ; r5 <- 5
```

the corresponding binary instructions are as following:

```
v2.0 raw
102005
104005
096480
562801
108004
10a005
```

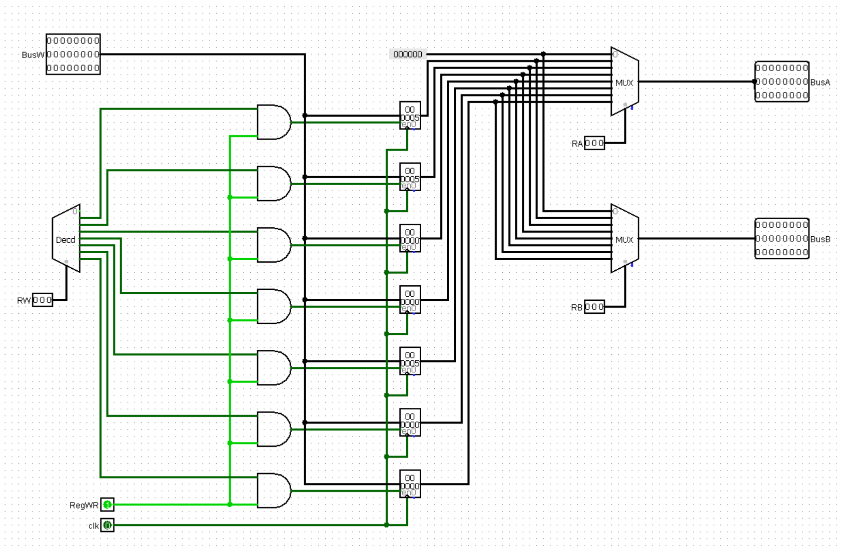


Figure 9: Test 1 Result in Register File

We notice that the value of register 4 didn't get assigned to 4, as expected. And other values of registers were assigned correctly.

3.2 Test Case 2

The following code was written in high level assembly:

```
ADDI r2, r0, 5   ; 5 Iterations
BEQ r1, r2, 3    ; Break condition
```

```
ADDI r3, r3, 10 ; Loop body
ADDI r1, r1, 1        ; Increment
J 1                                 ; Return to start of loop
ADDI r4, r0, 3        ; To check if executed correctly
```

the corresponding binary instructions are as following:

```
v2.0 raw
104005
162803
106c0a
102401
180001
108003
```

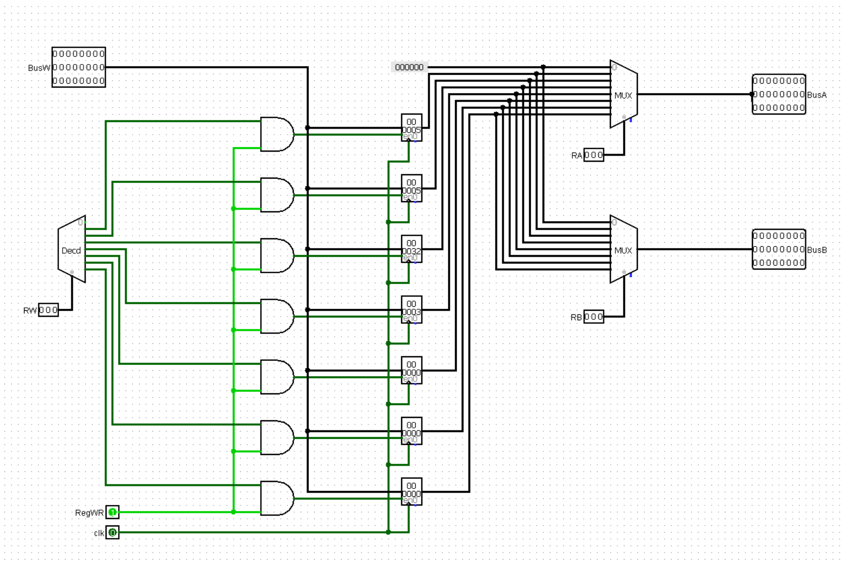


Figure 10: Test 2 Result in Register File

We notice that, register 3 was assigned 50, which means that the loop executed 5 times. And register 4 was assigned 3 which means the loop ended correctly.

3.3 Test Case 3

The following code was written in high level assembly:

```
ADDI r1, r0, 10                ; r1 <- 10
ADDI r2, r0, 15         ; r2 <- 15
CAS r3, r1, r2                ; r3 <- max(10, 15)
ADD r4, r1, r3                ; r4 <- 10 + max(10, 15)
SUB r5, r2, r1                ; r4 <- 15 - 10
AND r6, r1, r2                ; r6 <- 10 & 15
SUBSF r2, r3, r2        ; r2 <- 15 - 15
ADDINE r2, r0, 20         ; r2 <- 20 (if z = 0)
LUIEQ 7                                 ; r1 <- 7 * 16 (if z = 1)
SW r1, r2, 2                  ; M[0 + 2] <- 7 * 16
```

the corresponding binary instructions are as following:

```
v2.0 raw
10200a
10400f
026500
068580
08a880
00c500
094d00
904014
5c0007
142802
```

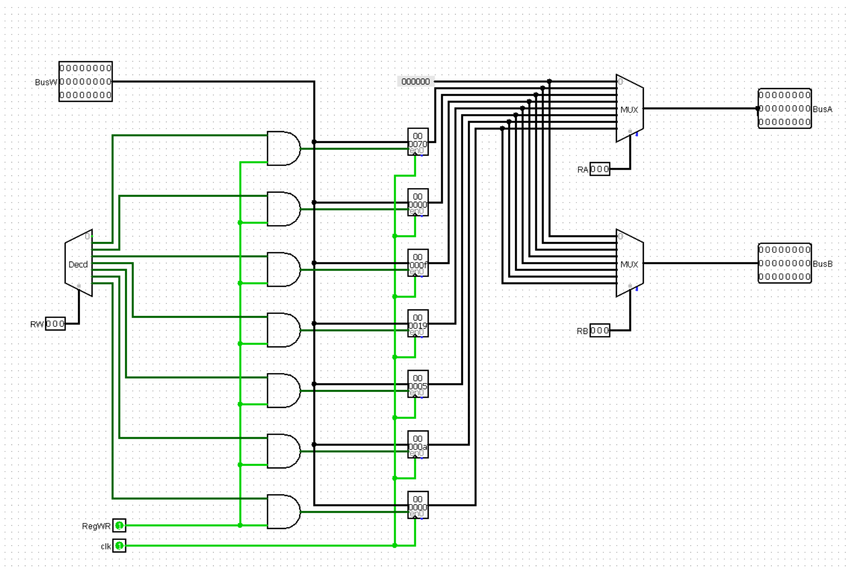
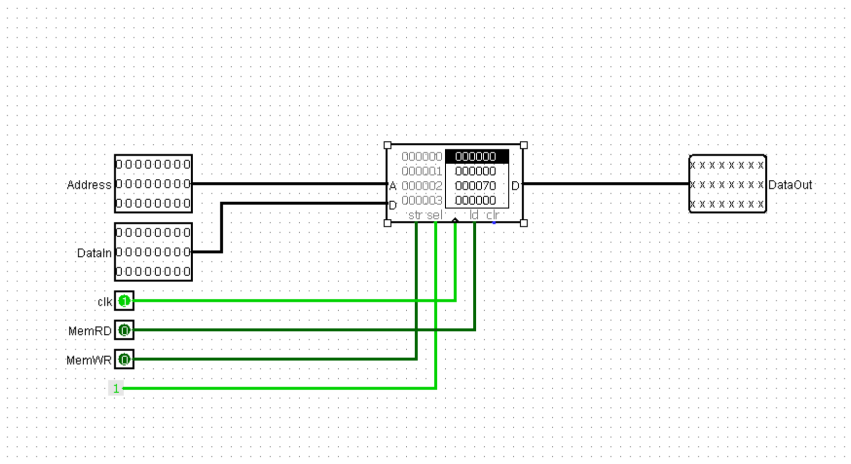


Figure 11: Test 3 Result in Register File



```
000000 000000
000001 000000
000002 000070
000003 000000
```

Figure 12: Test 3 Result in Data Memory

We notice that register r1 has took the value 10, register r2 didn't take the value 20 because the instruction ADDINE didn't execute, and the memory at 2 got assigned of 70, which means the instructions executed correctly.

3.4 Test Case 4

The following code was written in high level assembly:

```
ADDI r2, r0, 2          ; r2 <- 2
LW r1, r2, 3          ; r1 <- M[2 + 3]
LW r2, r0, 1          ; r2 <- M[0 + 1]
CMP r1, r2                  ; Zero <- (r1 < r2)
JALEQ 6                          ; Jump to 6 if (r1 < r2)
JALNE 7                          ; Jump to 7 if (r1 >= r2)
ADDI r3, r3, 6          ; r3 <- r3 + 6
ADDI r3, r3, 7          ; r3 <- r3 + 7
```

the corresponding binary instructions are as following:

```
v2.0 raw
104002
122803
124001
0a0500
5a0006
9a0007
106c06
106c07
```

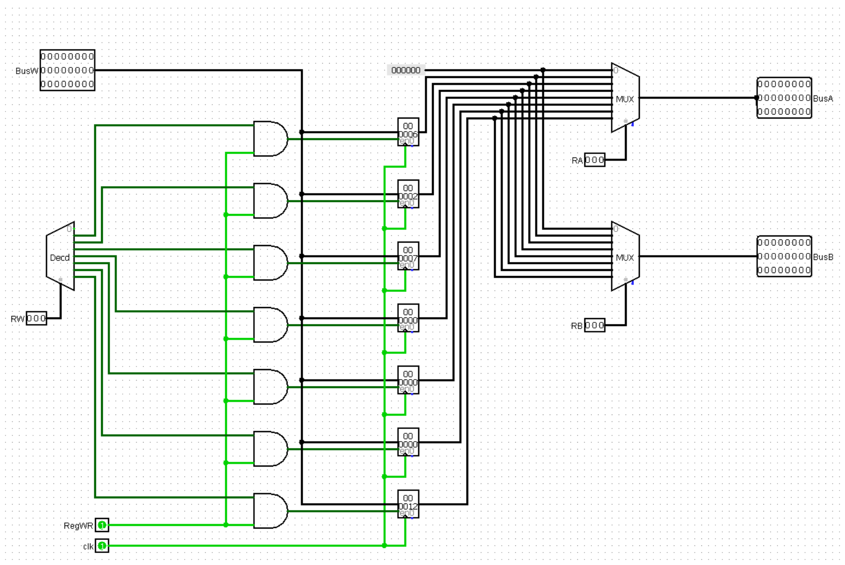
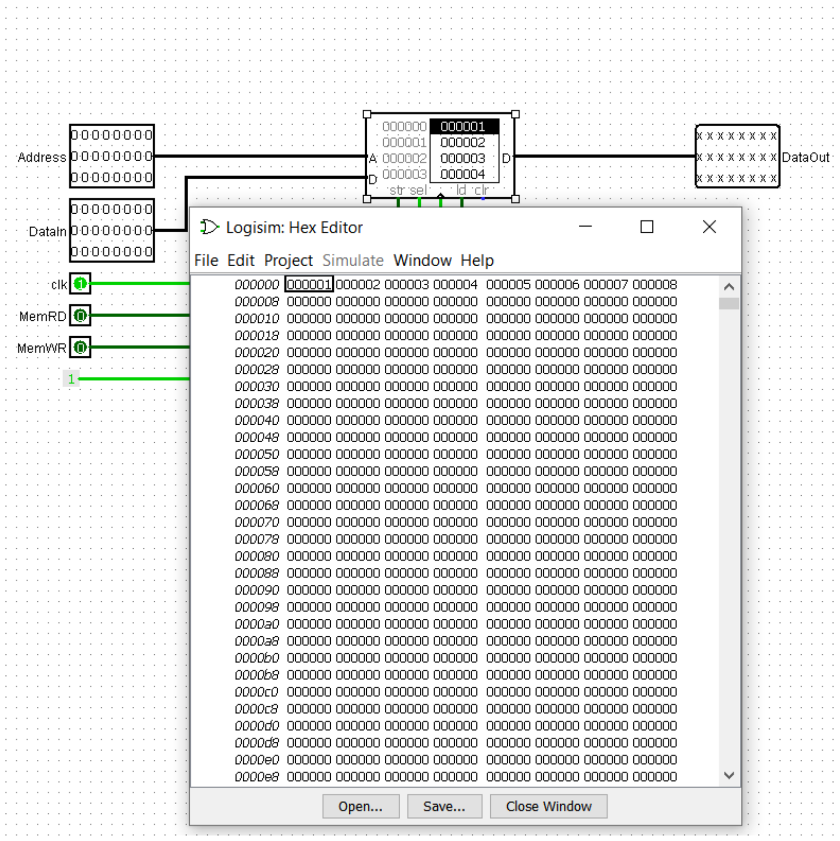


Figure 13: Test 4 Result in Register File



```
000000  000001
000001  000002
000002  000003
000003  000004
```

```
000000 000001 000002 000003 000004  000005 000006 000007 000008
000008 000000 000000 000000 000000  000000 000000 000000 000000
000010 000000 000000 000000 000000  000000 000000 000000 000000
000018 000000 000000 000000 000000  000000 000000 000000 000000
000020 000000 000000 000000 000000  000000 000000 000000 000000
000028 000000 000000 000000 000000  000000 000000 000000 000000
000030 000000 000000 000000 000000  000000 000000 000000 000000
000038 000000 000000 000000 000000  000000 000000 000000 000000
000040 000000 000000 000000 000000  000000 000000 000000 000000
000048 000000 000000 000000 000000  000000 000000 000000 000000
000050 000000 000000 000000 000000  000000 000000 000000 000000
000058 000000 000000 000000 000000  000000 000000 000000 000000
000060 000000 000000 000000 000000  000000 000000 000000 000000
000068 000000 000000 000000 000000  000000 000000 000000 000000
000070 000000 000000 000000 000000  000000 000000 000000 000000
000078 000000 000000 000000 000000  000000 000000 000000 000000
000080 000000 000000 000000 000000  000000 000000 000000 000000
000088 000000 000000 000000 000000  000000 000000 000000 000000
000090 000000 000000 000000 000000  000000 000000 000000 000000
000098 000000 000000 000000 000000  000000 000000 000000 000000
0000a0 000000 000000 000000 000000  000000 000000 000000 000000
0000a8 000000 000000 000000 000000  000000 000000 000000 000000
0000b0 000000 000000 000000 000000  000000 000000 000000 000000
0000b8 000000 000000 000000 000000  000000 000000 000000 000000
0000c0 000000 000000 000000 000000  000000 000000 000000 000000
0000c8 000000 000000 000000 000000  000000 000000 000000 000000
0000d0 000000 000000 000000 000000  000000 000000 000000 000000
0000d8 000000 000000 000000 000000  000000 000000 000000 000000
0000e0 000000 000000 000000 000000  000000 000000 000000 000000
0000e8 000000 000000 000000 000000  000000 000000 000000 000000
```

Figure 14: Test 4 Result in Data Memory

We notice that r1 > r2 so JALNE was executed and the value of register 3 took a final value of 7 which means ADDI r3, r3, 6 didn't execute.

4 Conclusion

The design of processor includes multiple steps that must be precise and correct, to avoid conflicts or edge cases.

Testing must be made for all components and for all instructions, then a scenarios that manipulate data, and branches must be executed to test the performance and correctness of the design.