ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 222E

COMPUTER ORGANIZATION
PROJECT 2 REPORT

CRNs : 21334

21336

LECTURERs : GÖKHAN İNCE

MUSTAFA ERSEL KAMAŞAK

GROUP MEMBERS:

150210087 : HİLAL KARTAL

150210100 : SELİN YILMAZ

SPRING 2024

Contents

1 INTRODUCTION [10 points]

In this project, Verilog hardware description language and the basic computer we designed earlier are used to implement a hardwired control unit. During this project, this architecture commonly called as Central Processing Unit (CPU). CPU is the part that performs most of the data processing operations in computer.

First, we implemented our *sequence counter* logic. Counter needs to change in every rising edge of the clock until it is *reseted*. Second, we defined the *ALU System* we implemented before in this project. Then, we wrote the *instruction fetch* logic that is done during T_0 and T_1. We took the new instruction from memory in this part. Finally, we wrote the *execution operations*. Instruction fetch and execution operations are done to implement *instruction operations*.

1.1 Task Distribution

We distributed the parts in the following way.

- Sequence Counter $\longrightarrow$ Hilal
- ALU System Definition $\longrightarrow$ Selin
- Instruction Fetch $\longrightarrow$ Hilal
- Execution (type 0) $\longrightarrow$ Hilal, Selin
- Execution (type 1) $\longrightarrow$ Hilal
- Execution (type 2) $\longrightarrow$ Selin

2 MATERIALS AND METHODS [40 points]

2.1 Materials

1. Verilog

2.2 Methods

2.2.1 Sequence Counter

Sequence Counter is defined in a *always* block that is controlled by Clock.

1. It is set to 0 ($T0$) at the end of each instruction.
2. It controls the change of output reg T and a
3. T is a 8 bit register and it keeps the times T0(LSB) to T7(MSB) in each bit.
4. This allows us to check the time whenever we want by writing T[x].
5. Table below shows how T registor works:

7	6	5	4	3	2	1	0	T[x]
0	0	0	0	0	0	0	1	T[0]
0	0	0	0	0	0	1	0	T[1]
0	0	0	0	0	1	0	0	T[2]
0	0	0	0	1	0	0	0	T[3]
0	0	0	1	0	0	0	0	T[4]
0	0	1	0	0	0	0	0	T[5]
0	1	0	0	0	0	0	0	T[6]
1	0	0	0	0	0	0	0	T[7]

Table 1: T[x] Table

2.2.2 ALU System Definition

We used the ALU System, we implemented in the project 1 (Fig. 1), in our CPU.

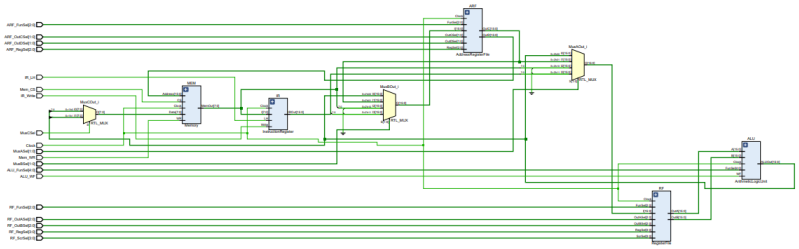


Figure 1: ALU System

We defined all the selectors that belong to ALU system (see Fig. 2) as internal registers. Because we want to control them according to instruction.

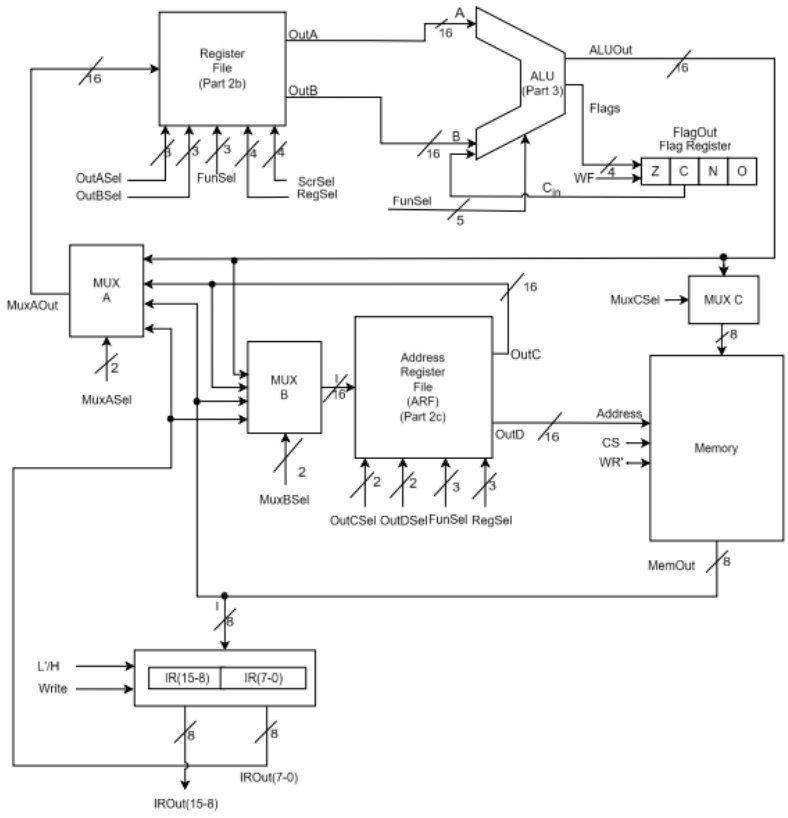


Figure 2

We defined all the selectors we use to control circuit as internal registers. All of the selectors we control during fetch and execution look like following:

Part	Bit	Name
RF	[2:0]	RF_OutBSel
	[2:0]	RF_OutASel
	[2:0]	RF_FunSel
	[3:0]	RF_RegSel
	[3:0]	RF_ScrSel
ALU	[4:0]	ALU_FunSel
		ALU_WF
ARF	[1:0]	ARF_OutCSel
	[1:0]	ARF_OutDSel
	[2:0]	ARF_FunSel
	[2:0]	ARF_RegSel
IR		IR_LH
		IR_Write
Memory		Mem_WR
		Mem_CS
ARF	[1:0]	MuxASel
	[1:0]	MuxBSel
		MuxCSel

Table 2: Internal Registers for ALU

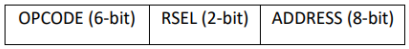
2.2.3 Fetch Instruction

Fetch Operation is defined in a *always* block that is controlled by T[0] and T[1] clocks as the fetch happens at these clock cycles.

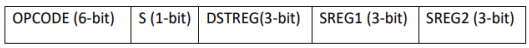
- At T[0], we fetch the LSB of the instruction from the Memory and write it to the LSB of the Instruction register.
- At T[1], first we increment the PC. Then we fetch the MSB of the instruction from the Memory and write it to the MSB of the Instruction register.

2.2.4 Execute Instruction

An operations table is given to us (Table 3). These operations are determined by the opcode in *instruction register*. And we have two types of instruction register: with address and without address (Fig. 3).



(a) IR with address



(b) IR without address

Figure 3

IR's type is determined by looking its opcode. If its opcode definition requires IR with address then it is an IR with address.

In order to make implementation easier, we defined 3 instruction types:

$$\textit{type } 0 \longrightarrow \text{IR with address}$$
$$\textit{type } 1 \longrightarrow \text{IR without address (can be done using ALU)}$$
$$\textit{type } 2 \longrightarrow \text{IR without address}$$

OPCODE (HEX)	SYMBOL	DESCRIPTION
0x00	BRA	PC ← PC + VALUE
0x01	BNE	IF Z=0 THEN PC ← PC + VALUE
0x02	BEQ	IF Z=1 THEN PC ← PC + VALUE
0x03	POP	SP ← SP + 1, Rx ← M[SP]
0x04	PSH	M[SP] ← Rx, SP ← SP − 1
0x05	INC	DSTREG ← SREG1 + 1
0x06	DEC	DSTREG ← SREG1 − 1
0x07	LSL	DSTREG ← LSL SREG1
0x08	LSR	DSTREG ← LSR SREG1
0x09	ASR	DSTREG ← ASR SREG1
0x0A	CSL	DSTREG ← CSL SREG1
0x0B	CSR	DSTREG ← CSR SREG1
0x0C	AND	DSTREG ← SREG1 AND SREG2
0x0D	ORR	DSTREG ← SREG1 OR SREG2
0x0E	NOT	DSTREG ← NOT SREG1
0x0F	XOR	DSTREG ← SREG1 XOR SREG2
0x10	NAND	DSTREG ← SREG1 NAND SREG2
0x11	MOVH	DSTREG[15:8] ← IMMEDIATE (8-bit)
0x12	LDR	(16-bit) Rx ← M[AR] (AR is 16-bit register)
0x13	STR	(16-bit) M[AR] ← Rx (AR is 16-bit register)
0x14	MOVL	DSTREG[7:0] ← IMMEDIATE (8-bit)
0x15	ADD	DSTREG ← SREG1 + SREG2
0x16	ADC	DSTREG ← SREG1 + SREG2 + CARRY
0x17	SUB	DSTREG ← SREG1 − SREG2
0x18	MOVS	DSTREG ← SREG1, Flags will change
0x19	ADDS	DSTREG ← SREG1 + SREG2, Flags will change
0x1A	SUBS	DSTREG ← SREG1 − SREG2, Flags will change
0x1B	ANDS	DSTREG ← SREG1 AND SREG2, Flags will change
0x1C	ORRS	DSTREG ← SREG1 OR SREG2, Flags will change
0x1D	XORS	DSTREG ← SREG1 XOR SREG2, Flags will change
0x1E	BX	M[SP] ← PC, PC ← Rx
0x1F	BL	PC ← M[SP]
0x20	LDRIM	Rx ← VALUE
0x21	STRIM	M[AR+OFFSET] ← Rx

Table 3: OPCODEs and their descriptions

1. TYPE 0

   Instructions done at this type could be seen from table below:

OPCODE (HEX)	SYMBOL	DESCRIPTION
0x00	BRA	PC ← PC + VALUE
0x01	BNE	IF Z=0 THEN PC ← PC + VALUE
0x02	BEQ	IF Z=1 THEN PC ← PC + VALUE
0x03	POP	SP ← SP + 1, Rx ← M[SP]
0x04	PSH	M[SP] ← Rx, SP ← SP − 1
0x12	LDR	(16-bit) Rx ← M[AR] (AR is 16-bit register)
0x13	STR	(16-bit) M[AR] ← Rx (AR is 16-bit register)
0x1E	BX	M[SP] ← PC, PC ← Rx
0x1F	BL	PC ← M[SP]
0x20	LDRIM	Rx ← VALUE
0x21	STRIM	M[AR+OFFSET] ← Rx

Table 4: Instruction Type 0

We implemented each instruction separately:

BRA:

- T[2]:
  - PC ← PC + 1
- T[3]:
  - RF_Scr1 ← PC
- T[4]:
  - RF_Scr2 ← M[AR]
  - ALU_Out ← RF_Scr1 + RF_Scr2
  - SC ← Reset

BNE:

- T[2]:
  - PC ← PC + 1
- T[3]:
  - RF_Scr1 ← PC

- T[4]:
  - RF_Scr2 ← M[AR]
  - ALU_Out ← RF_Scr1 + RF_Scr2
  - PC ← ALU_Out
  - SC ← Reset

BEQ:

- T[2]:
  - PC ← PC + 1
- T[3]:
  - RF_Scr1 ← PC
  - AR ← IR
- T[4]:
  - RF_Scr2 ← M[AR]
  - ALU_Out ← RF_Scr1 + RF_Scr2
  - PC ← ALU_Out
  - SC ← Reset

POP:

- T[2]:
  - PC ← PC + 1
  - SP ← SP + 1
  - RF_Rx[7:0] ← M[SP]
- T[3]:
  - SP ← SP + 1
  - RF_Rx[15:8] ← M[SP]
  - SC ← Reset

PSH:

- T[2]:
  - PC ← PC + 1
  - M[SP] ← RF_Rx[7:0]
- T[3]:
  - AR ← SP

- T[4]:
  - AR ← AR + 1
  - M[AR] ← RF_Rx[15:8]
- T[5]:
  - SP ← SP - 1
  - SC ← Reset

LDR:

- T[2]:
  - PC ← PC + 1
  - RF_Rx[7:0] ← M[AR]
- T[3]:
  - AR ← AR + 1
  - RF_Rx[15:8] ← M[AR]
  - SC ← Reset

STR:

- T[2]:
  - PC ← PC + 1
  - M[AR] ← RF_Rx[7:0]
- T[3]:
  - AR ← AR + 1
  - M[AR] ← RF_Rx[15:8]
  - SC ← Reset

BX:

- T[2]:
  - PC ← PC + 1
  - M[SP] ← PC
- T[3]:
  - SP ← SP + 1
  - M[SP] ← PC
- T[4]:
  - RF_Rx ← PC

- SC ← Reset

BL:

- T[2]:
  - PC ← PC + 1
  - RF_Scr1[7:0] ← M[SP]
- T[3]:
  - SP ← SP + 1
  - RF_Scr1[15:8] ← M[SP]
- T[4]:
  - PC ← ALU_Out
  - SC ← Reset

LDRIM:

- T[2]:
  - PC ← PC + 1
- T[3]:
  - AR ← IR
  - RF_Rx ← M[AR]
  - SC ← Reset

STRIM:

- T[2]:
  - PC ← PC + 1
- T[3]:
  - RF_Scr1 ← AR
- T[4]:
  - RF_Scr2 ← M[IR]
  - ALU_Out ← Scr1 + Scr2
- T[5]:
  - AR ← ALU_Out
  - M[AR] ← RF_Rx
- T[6]:
  - AR ← AR + 1

- AR ← ALU_Out
- M[AR] ← RF_Rx
- SC ← Reset

2. TYPE 1

   Instructions done at this type could be seen from table below:

OPCODE (HEX)	**SYMBOL**	**DESCRIPTION**
0x07	LSL	DSTREG ← LSL SREG1
0x08	LSR	DSTREG ← LSR SREG1
0x09	ASR	DSTREG ← ASR SREG1
0x0A	CSL	DSTREG ← CSL SREG1
0x0B	CSR	DSTREG ← CSR SREG1
0x0C	AND	DSTREG ← SREG1 AND SREG2
0x0D	ORR	DSTREG ← SREG1 OR SREG2
0x0E	NOT	DSTREG ← NOT SREG1
0x0F	XOR	DSTREG ← SREG1 XOR SREG2
0x10	NAND	DSTREG ← SREG1 NAND SREG2
0x15	ADD	DSTREG ← SREG1 + SREG2
0x16	ADC	DSTREG ← SREG1 + SREG2 + CARRY
0x17	SUB	DSTREG ← SREG1 − SREG2
0x18	MOVS	DSTREG ← SREG1, Flags will change
0x19	ADDS	DSTREG ← SREG1 + SREG2, Flags will change
0x1A	SUBS	DSTREG ← SREG1 − SREG2, Flags will change
0x1B	ANDS	DSTREG ← SREG1 AND SREG2, Flags will change
0x1C	ORRS	DSTREG ← SREG1 OR SREG2, Flags will change
0x1D	XORS	DSTREG ← SREG1 XOR SREG2, Flags will change

Table 5: Instruction Type 1

- At T=2:
  - We increment PC.
  - We send AR to the Scratch Register 1.
- At T=3:
  - We send PC to the Scratch Register 2.
- At T=4:
  - We send SP to the Scratch Register 3.

- We check the bit 9 of the instruction (S) to decide whether Flags will change or not.
- We check the bits 5 to 3 and 2 to 0 of instruction to decide which Source Registers ($SREG1$ *and* $SREG2$) are used for the operation. $SREG1$ is taken from OutA and $SREG2$ is taken from OutB.
- We check the bits 15 to 10 of instruction to decide which instruction to execute.

- At T=5:
  - We load the result of the operation to the Destination Register ($DSTREG$) defined in the instruction bits $8to6$
  - Lastly, we reset the Sequence Counter to 0 as the instruction is executed.

3. TYPE 2

   Instructions done at this type could be seen from table below:

OPCODE (HEX)	**SYMBOL**	**DESCRIPTION**
0x05	INC	DSTREG ← SREG1 + 1
0x06	DEC	DSTREG ← SREG1 - 1
0x11	MOVH	DSTREG[15:8] ← IMMEDIATE
0x14	MOVL	DSTREG[7:0] ← IMMEDIATE

Table 6: Instruction Type 2

DSTREG/SREG1	**REGISTER**
000	PC
001	PC
010	SP
011	AR
100	R1
101	R2
110	R3
111	R4

Table 7: Selection Table

We check which one to choose using case method. So we have four options (for INC and DEC):

$$\text{ARF} \longleftarrow \text{ARF}$$
$$\text{ARF} \longleftarrow \text{RF}$$
$$\text{RF} \longleftarrow \text{ARF}$$
$$\text{RF} \longleftarrow \text{RF}$$

We implemented each instruction separately:

INC:

- ARF to ARF
  - T[2]:
    - PC $\longleftarrow$PC + 1
  - T[3]:
    - Increment SREG1 (ARF_FunSel = increment)
    - Send to ARF (MuxBSel = ARFOut)
  - T[4]:
    - Write on DSTREG (ARF_FunSel = Load)
    - Reset the Sequence Counter
- RF to ARF
  - T[2]:
    - PC $\longleftarrow$ PC + 1
    - Increment SREG1 (RF_FunSel = increment)
    - Send it to ALU (ALU_FunSel = A, ALU_WF = RSEL)
    - Send ALU to ARF (MuxBSel = ALUOut)
  - T[3]:
    - Write on DSTREG
    - Reset the Sequence Counter
- ARF to RF
  - T[2]:
    - PC $\longleftarrow$ PC + 1
  - T[3]:
    - Increment SREG1
    - Send it to RF (MuxASel = ARFOut)
    - Write on DSTREG

        * Reset Sequence Counter

- RF to Rf
  - T[2]:
    * PC ⟵ PC + 1
    * Increment SREG1
    * Send it to ALU
    * Send it to RF
  - T[3]:
    * Write it on DSTREG
    * Reset the Sequence Counter

DEC:

- ARF to ARF
  - T[2]:
    * PC ⟵ PC + 1
  - T[3]:
    * Decrement SREG1 (ARF_FunSel = decrement)
    * Send it to ARF (MuxBSel = ARFOut)
  - T[4]:
    * Write on DSTREG
    * Reset Sequence Counter
- RF to ARF
  - T[2]:
    * PC ⟵ PC + 1
    * Decrement SREG1 (RF_FunSel = decrement)
    * Send it to ALU (ALU_FunSel = A, ALU_WF = RSEL)
    * Send ALU to ARF (MuxBSel = ALUOut)
  - T[3]:
    * Write on DSTREG
    * Reset the Sequence Counter
- ARF to RF
  - T[2]:
    * PC ⟵ PC + 1

  - T[3]:
    * Decrement SREG1
    * Send it to RF (MuxASel = ARFOut)
    * Write on DSTREG
    * Reset Sequence Counter

- RF to Rf
  - T[2]:
    * PC $\longleftarrow$ PC + 1
    * Decrement SREG1
    * Send it to ALU
    * Send it to RF
  - T[3]:
    * Write it on DSTREG
    * Reset the Sequence Counter

After finishing everything, we learnt that DSTREG for MOVH and MOVL only include registers in RF (R_x). And it must be chosen by RSEL (Fig. 3a). IMMEDIATE (IR's adress bits) is written on registers in RF. The table in order to decide DSTREG (for MOVH and MOVL) is:

DSTREG	**REGISTER**
00	R1
01	R2
10	R3
11	R4

Table 8: Selection Table

MOVH:

- Write to RF
  - T[2]:
    * PC $\longleftarrow$ PC + 1
    * Clear DSTREG (RF_FunSel = clear)
  - T[3]:
    * Write IR on DSTREG high (MuxASel = IROut, RF_FunSel = write high)

        * Reset the Sequence Counter

MOVL:

- Write to RF
  - T[3]:
    * PC $\longleftarrow$ PC + 1
    * Write low IR on DSTREG (MuxASel = IROut, RF_FunSel = clear and write low)
    * Reset the Sequence Counter

3 RESULTS [15 points]

Result for BRA instruction:

1. From Figure 4, it can be seen that T[x] starts from T[0] and increases at each cycle.(To understand the workings of T[x] please refer to subsection 2.2.1).

2. It can also be seen that PC's is set to it's initial value 0,

```
Output Values:
T:  1
Address Register File: PC:    x, AR:    x, SP:    x
Instruction Register :    x
Register File Registers: R1:    x, R2:    x, R3:    x, R4:    x
Register File Scratch Registers: S1:    x, S2:    x, S3:    x, S4:    x
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:    x


Output Values:
T:  2
Address Register File: PC:    0, AR:    x, SP:    x
Instruction Register :    x
Register File Registers: R1:    x, R2:    x, R3:    x, R4:    x
Register File Scratch Registers: S1:    x, S2:    x, S3:    x, S4:    x
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:    x


Output Values:
T:  4
Address Register File: PC:    0, AR:    x, SP:    x
Instruction Register :    x
Register File Registers: R1:    x, R2:    x, R3:    x, R4:    x
Register File Scratch Registers: S1:    x, S2:    x, S3:    x, S4:    x
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:    x
```

Figure 4: T[0], T[1] and T[2]

1. From Figure 5, it can be seen that T[x] still increases at each cycle. However, at T[5] it is set to 0 as the execution is over.(To understand how BRA is executed please refer to list 1).

```
Output Values:
T:  8
Address Register File: PC:    1, AR:    x, SP:    x
Instruction Register :    X
Register File Registers: R1:    x, R2:    x, R3:    x, R4:    x
Register File Scratch Registers: S1:    x, S2:    x, S3:    x, S4:    x
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:    0


Output Values:
T:  16
Address Register File: PC:    1, AR:    X, SP:    x
Instruction Register :    X
Register File Registers: R1:    x, R2:    x, R3:    x, R4:    x
Register File Scratch Registers: S1:    1, S2:    x, S3:    x, S4:    x
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:    0


Output Values:
T:  0
Address Register File: PC:    1, AR:    x, SP:    x
Instruction Register :    x
Register File Registers: R1:    x, R2:    x, R3:    x, R4:    x
Register File Scratch Registers: S1:    X, S2:    x, S3:    x, S4:    x
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:    0
```

Figure 5: T[3], T[4] and T[0](reset)

4 DISCUSSION [25 points]

4.1 SEQUENCE COUNTER

Our Sequence Counter is a register made up of 7 bits. Each bit represents each clock cycle. Starting from T[0] it can go up to T[7]. We use the sequence counter to keep up with with where we are at each instruction as they are made up of different clock cycles.

4.2 FETCH

Fetch operation is done in two clock cycles T[0] and T[1] (also T[2]) as the instruction is a 16 bit information but we can only read from memory 8 bits per clock cycle.

1. At T[0], we fetch the LSB of the instruction.
2. At T[1], we fetch the MSB of the instruction. Increment PC for T[0].
3. At T[2], we also increment PC again this time for T[1]. It is a event left from fetch. We have to do it at the T[2] because increment happens before any sort of read or load. So, to read the proper addresses at T[0] and T[1], we do the PC increments at the following steps.

4.3 INSTRUCTION - TYPE 0

In this section, we implemented each instruction individually and noticed some similarities among them:

BRA, BNE, BEQ
POP, PSH
LDR, STR
BX, BL
LDRIM, STRIM

Identifying these similarities helped us during implementation, as we used the implementation of the first instruction as a template for the others.

4.4 INSTRUCTION - TYPE 1

For this type of instructions we decided to first bring the ARF registers(PC, SP, AR) information to the Scratch registers in the RF. This cost us three clock cycles (We come to T[4]) and all executions start from T[4] as a result.

At T[4] according to the SREG1 and SREG2 we decide to input A and B of the ALU. According to the opcode we also decide the ALU operation(ALUFunSel).

At T[5], we load the Output of ALU to the destination register specified in DSTREG part of the instruction.

4.5 INSTRUCTION - TYPE 2

This instruction type actually encompasses two types (due to a misunderstanding in the assignment):

INC, DEC
MOVH, MOVL

For INC and DEC, we checked the 8^{th} and 5^{th} bits of IR. The 8^{th} bit determined the destination module (ARF if 0, RF if 1), and the 5^{th} bit determined the source module. Using the IR's 4^{th} and 3^{rd} bits, we selected the source register with *RegSel* and sent it as an output with *OutSel*. Using the IR's 7^{th} and 6^{th} bits, we selected the destination register with *RegSel*.

For MOVH and MOVL, we sent IR directly to RF. By checking the 9^{th} and 8^{th} bits of the IR, we determined the destination register and selected it with *RegSel*. We then wrote to it by choosing the appropriate *FunSel*.

5 CONCLUSION [10 points]

5.1 SEQUENCE COUNTER

At first we have tried to implement the Sequence Counter with a different module. However, we were unable to connect it to the CPUSystem. So, implemented it inside the CPUSystem. The module for the Sequence Counter still could be seen at the top of the file *CPUSystem.v* it is commented out.

5.2 MEMORY READ

While doing any sort of read or write to memory (mostly in instruction type 0), we did not notice at the beginning that Memory was holding each word at two consecutive addresses. So, at the beginning we wrote the code by just reading and writing the LSB. Luckily, we were able to notice and fix that mistake in a short time.

5.3 INSTRUCTION - TYPE 0

In addition to encountering issues with memory read and write operations, we also struggled with understanding certain expressions, particularly "VALUE" and "OFFSET."

5.4 INSTRUCTION - TYPE 1

For this type of instructions we are aware that we are losing some clock cycles(Some instructions may just use RF registers for their SREG1 and SREG2, but we still carry the ARF registers). However, for the sake of a order of code we decided that it would be the best course of action.

5.5 INSTRUCTION - TYPE 2

During our review of the MOVH and MOVL instructions, we realized that we had misunderstood the use of the DSTREG. Upon checking the message board in Ninova, we promptly corrected this error. However, without someone bringing it to our attention, we might not have noticed it.

5.6 EXECUTION - RESULTS

In the lessons, we learned that if we do the increment and send to output operations at the same clock cycle, incremented value would go to the output. So our code is designed to fit what we learned.(PC explanation could be seen from section 4.2). However, while

trying to run the simulation, we have noticed that doing these operations at the same clock cycle would result in sending the value to the output first and then the incrementation (Meaning at the fetch, we should have done the PC incrementations at the both could cycles.) Which means we had to change the entire code to fit this new information. However there was no such time left to do this so we left our code as it is.

[1]

REFERENCES

[1] M. Morris Mano. *Computer System Architecture.* Prentice Hall, 1993.