ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 222E
COMPUTER ORGANIZATION
PROJECT 2 REPORT

CRN : 21335
21334

LECTURER : Prof. Dr. Deniz Turgay Altılar
Doç. Dr. Gökhan İnce

GROUP MEMBERS:

150220725 : EYLÜL ZEYNEP PINARBAŞI

150210082 : MUHAMMET CANER ASLAN

SPRING 2024

Contents

1 INTRODUCTION

In this project, we have designed a hardware control unit for the given architecture. There are two types of instructions in the control unit. The first type of instructions have the address reference while the other type of instructions do not have the address reference; however, both of them are 16-bit. The control unit we designed in this project communicates with the modules we created for the Project 1. Also, we used clock signals to achieve synchronicity between components.

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 (VALUE defined in ADDRESS bits)
0x21	STRIM	M[AR+OFFSET] ← Rx (**AR is 16-bit register**) (OFFSET defined in ADDRESS bits)

Table 2: RSel table.

RSEL	REGISTER
00	R1
01	R2
10	R3
11	R4

Table 3: DSTREG/SREG1/SREG2 selection table.

DSTREG/SREG1/SREG2	REGISTER
000	PC
001	PC
010	SP
011	AR
100	R1

Figure 1: Provided Opcode, Rsel, DSTREG, SREG1 and SREG2 values

1.1 Task Distribution

For the task distribution, we applied a similar approach to that of Project 1. Since we think that it is crucial to know overall design and how things work, we worked as a team throughout the project including design, implementation and report sections.

2 MATERIALS AND METHODS

2.1 Fetch and Decode

Regarding fetching, it will be done in two clock cycles because an instruction is 16 bits while the memory can only have 8 bits. Since the instructions are stored in the memory in little-endian order (i.e., its least significant bits are stored first), the LSB of instruction is fetched first and then the MSB of instruction is fetched. Also, PC is incremented by one at each part so that we can take the next information stored in the memory. For Decode part, operations below occurred:

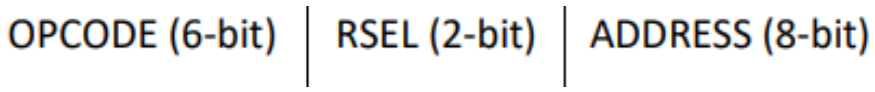

Figure 2: Instruction With an Address Reference

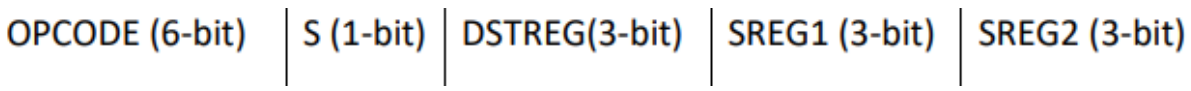

Figure 3: Instruction Without an Address Reference

- If our opcode is not STRIM, STR, LDRIM, or LDR, the address of the Instruction Register (IR [7:0]) is loaded to the Address Register. So, while this condition is satisfied, our **ARF RegSel** (which acts like Enable for Address Register File) enables our Address Register. Otherwise, all the registers in the Address Register are disabled, as well as Register File registers.

- Since there are two instruction formats and we might be unaware of which type of instruction ours fits, Rsel (Register Select for RF), Address, DSTREG (Destination register), SREG1 and SREG2 (Source Registers) are loaded from the information provided from the Instruction Register. S is one bit decider whether we update flags or not.

$$T2 : RSel \leftarrow IROut[9:8]$$

$$T2 : Address \leftarrow IROut[7:0]$$

$$T2 : DSTREG \leftarrow IROut[8:6]$$

$$T2 : SREG1 \leftarrow IROut[5:3]$$

$$T2 : SREG2 \leftarrow IROut[2:0]$$

$$T2 : S \leftarrow IROut[9]$$

2.2 Execution

Since some of the operations share same signals at certain Tx, in order to write less complex and shorter code, we created a table and divided them into some groups. However, we will be discussing each operation in detail.

2.2.1 BRA - BNE - BEQ

BRA, BNE and BEW operations all takes place in T3, T4 and T5 and are quite similar However, BNE and BEQ do have one condition to occur. In BNE operation, if ALU Flag Z (Zero) is 0, the operation would occur. Similarly, BEQ operation only occurs if ALU Z is 1.

- BRA:

$$T3 : S1 \leftarrow IR[7:0]$$

$$T4 : S2 \leftarrow PC$$

$$T5 : ALU\_OP,\ PC \leftarrow ALU\_OUT$$

- BNE:

$$T3 : S1 \leftarrow IR[7:0]$$

$$T4 : S2 \leftarrow PC$$

$$T5 : ALU\_OP,\ PC \leftarrow ALU\_OUT$$

- BEQ:

$$T3 : S1 \leftarrow IR[7:0]$$

$$T4 : S2 \leftarrow PC$$

$$T5 : ALU\_OP,\ PC \leftarrow ALU\_OUT$$

2.2.2 POP

POP and PSH operation (which is below) are both associated with stack. Our stack starts from 255 and goes upwards. Pop operation removes the topmost element of the stack and returns it. Little Endian must always lie at the top. Second half resides at SP+1 which means it is closer to the bottom of the stack.

$$T3 : SP \leftarrow SP + 1$$

$$T4 : Rx[15:8] \leftarrow M[SP],\ SP \leftarrow SP + 1$$

$$T5 : Rx[7:0] \leftarrow M[SP]$$

2.2.3 PSH

PSH operation adds a value to the top of the stack. After adding the first half, we decrement SP so that the first half stays at the top.

$$T3 : M[SP] \leftarrow Rx[7:0],\ SP \leftarrow SP - 1$$

$$T4 : M[SP] \leftarrow Rx[15:8],\ SP \leftarrow SP - 1$$

2.2.4 INC - DEC

After handling SREG's value and load it to DSTREG, we increment or decrement it according to the operation.

$$T3 : S1 \leftarrow SREG1$$

$$T4 : ALU\_OP,\ DSTREG \leftarrow ALU\_OUT$$

$$T5 : DSTREG \leftarrow DSTREG + 1$$

$$T5 : DSTREG \leftarrow DSTREG - 1$$

2.2.5 Shift Operations (LSL - LSR - ASR - CSL - CSR)

These shift operations all start with loading the value from the source register which is either a ARF or RF register. Lastly, we take that value which was loaded to S1 and perform the shift operation in ALU. Finally, the value is stored in the destination register.

$$T3 : S1 \leftarrow SREG1$$

$$T4 : ALU\_OP, DSTREG \leftarrow ALU\_OUT$$

ALUOut changes according to the operation.

- For LSL: ALU Funsel is 11011. ALUOut is LSL S1.
- For LSR: ALU Funsel is 11100. ALUOut is LSR S1.
- For ASR: ALU Funsel is 11101. ALUOut is ASR S1.
- For CSL: ALU Funsel is 11110. ALUOut is CSL S1.
- For CSR: ALU Funsel is 11111. ALUOut is CSR S1.

2.2.6 Logic Operations (AND - ORR - NOT - XOR - NAND)

Logic operations do happen similar to shift operations. The only difference is SREG2 which also stores a value for operations to also use it.

$$T3 : S1 \leftarrow SREG1$$

$$T4 : S2 \leftarrow SREG2$$

$$T5 : ALU\_OP, DSTREG \leftarrow ALU\_OUT$$

- For AND: ALU Funsel is 10111. ALUOut is S1 AND S2.
- For ORR: ALU Funsel is 11100. ALUOut is S1 OR S2.
- For NOT: ALU Funsel is 11000. ALUOut is S1'.
- For XOR: ALU Funsel is 11001. ALUOut S1 XOR S2.
- For NAND: ALU Funsel is 11010. ALUOut SI NAND S2.

Logic operations which have a 'S' at the end of the name (ANDS, ORRS, XORS...) all change the flags of the ALU so if our operation has a S at the end, we enable ALU WF (Flags).

2.2.7 MOVH-MOVL

Value (address which is in IR) is loaded to either high (15:8) or low (7:0) of the Rx which is selected by Rsel.

$$T3 : RX[7:0] \leftarrow IR[7:0]$$

$$T3 : RX[15:8] \leftarrow IR[7:0]$$

2.2.8 LDR

In LDR (Load) operation, we read M[AR] into Rx in T3 and T4. Since our memory only holds 8 bits of data, after loading the second half, AR is decremented so that we can continue to load the other half. Second half must be closer to the bottom.

$$T3 : RX[15:8] \leftarrow M[AR]$$

$$AR \leftarrow AR - 1$$

$$T3 : RX[7:0] \leftarrow M[AR]$$

2.2.9 STR

STR (Store) operation writes data to memory. Since memory output is only 8 bits, 16 bit data is stored in T3 and T4. First, Little Endian (low) is stored. After storing, AR is incremented so that the address which is closer to is loaded with RX[15:8].

$$T3 : M[AR] \leftarrow RX[7:0]$$

$$AR \leftarrow AR + 1$$

$$T3 : M[AR] \leftarrow RX[15:8]$$

2.2.10 MOVS

MOVS operation briefly loads SREG to DSTREG but also changes the flags which means ALU WF is 1 so that flag changing is enabled.

$$T3 : S1 \leftarrow SREG1$$

$$T4 : ALU\_OP, DSTREG \leftarrow ALU\_OUT$$

ALU Funsel is 10000 which means we directly load source register's value into destination register.

2.2.11 BX

BX (which is a short form of Branch and Exchange) lets us to branch to different location. First, the current value of the program counter is being prepared for use in subsequent steps. After storing the data in T5 and T6, SP is decremented by one two times. As the stack grows downward in memory, decrementing the Stack Pointer effectively "reserves" space for the next value to be stored. After the Program Counter is loaded with a value from the Rx, the processor branches to a different location.

$$T3 : S1 \leftarrow PC$$

$$T4 : Nothing$$

$$T5 : M[SP] \leftarrow ALUOUT[15:8]$$

$$SP \leftarrow SP - 1$$

$$T6 : M[SP] \leftarrow ALUOUT[7:0]$$

$$SP \leftarrow SP - 1$$

$$T7 : PC \leftarrow Rx$$

2.2.12 BL

Sp is decremented (as the new values in the stack are stored in decreasing memory addresses) and stored two times (as only 8 bits can be stored at one T.

$$T3 : SP \leftarrow SP - 1$$

$$T4 : PC[15:8] \leftarrow M[SP]$$

$$T5 : SP \leftarrow SP - 1$$

$$T6 : PC[7:0] \leftarrow M[SP]$$

2.2.13 LDRIM

LDRIM operation stores the value which is stored as the address bits in a RF register which is selected by Rsel.

$$T3 : Rx \leftarrow IR[7:0]$$

2.2.14 STRIM

STRIM is the last and the longest operation.

First of all, we store address to S1.

$$T3 : S1 \leftarrow IR[7:0]$$

After storing the value of AR to the S2 (which is set to be not equal to IR[7:0] because we will be needing the old value), we are now ready to perform an addition.

We are also incrementing AR since we need 2 memory spaces for 16 bit data.

$$T4 : AR \leftarrow ALUOut$$

Lastly, we will be writing this sum to M[AR].

$$T5 : M[AR] \leftarrow Rx[7:0]$$

$$AR \leftarrow AR + 1$$

$$T6 : M[AR] \leftarrow Rx[8:15]$$

3 RESULTS

This project was time consuming and long, therefore We needed to implement tests for all the operations. We used a display template which is given below and mostly consisted of simulation file's displays, changed the ram.mem file and calculated the results accordingly. During the project, a lot of mistakes were made as only a small typo could be the reason for the code to fail. Therefore, after comparing the results with the ones we calculated, we used online AI tools to get help for additional tests.

While implementing project 2, we realized the mistakes we made from the previous project. Since project 1 was crucial for this project, the mistakes would of course shown themselves among some cases.

First, we added this line to the register module: initial Q = 0; Another problems occurred during the checking the operation. LSR operation was wrong due to a logical error. This problem is fixed.

In order to prevent possible mistakes, we decided to disable all registers at the end of every T so that they would be only enabled if they are needed and won't be left enabled.

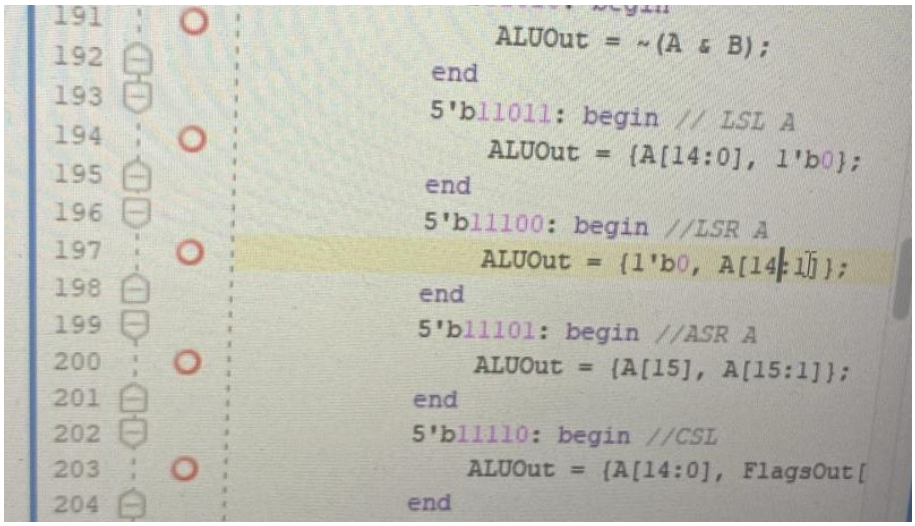

Figure 4: Modified function

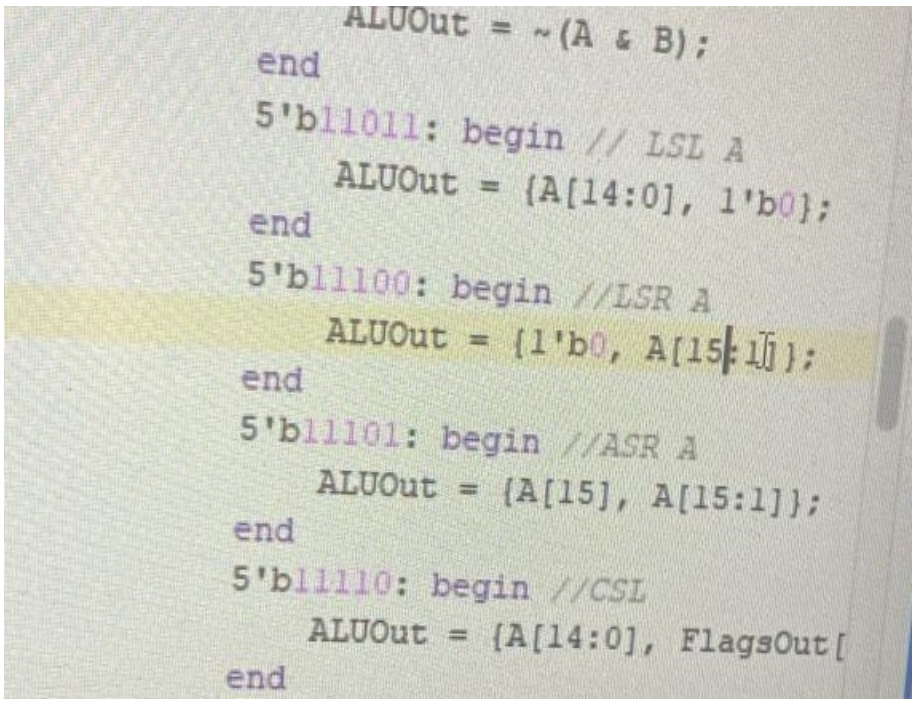

Figure 5: Previous function

4 DISCUSSION

In this project, we designed and implemented a control unit and established the necessary connections between the ALU system we previously implemented. What control unit does is basically to tell ALU which operation it will apply on the operand and what is the address of the operand. In the first two clock cycles (i.e., T = 0 and T = 1), the LSB and MSB of the instruction is loaded from corresponding addresses to the LSB and MSB

```
Output Values:
OpCode: 34
T:   1
Address Register File: PC:    44, AR:    36, SP:    38
Instruction Register : 53796
Register File Registers: R1:    37, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:    22, S2:    22, S3:     0, S4:     0
ALU Flags: Z: 0, N: 0, C: 0, O: 0
Rsel: 10, S: 1, DSTREG: 0000, SReg1: 0100, SReg2: 0100
Output Values:
T:   1
Address Register File: PC:    44, AR:    36, SP:    38
Instruction Register : 53796
Register File Registers: R1:    37, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:    22, S2:    22, S3:     0, S4:     0
ALU Flags: Z: 0, N: 0, C: 0, O: 0
ALU Result: ALUOut:    44
```

Figure 6: Outputs

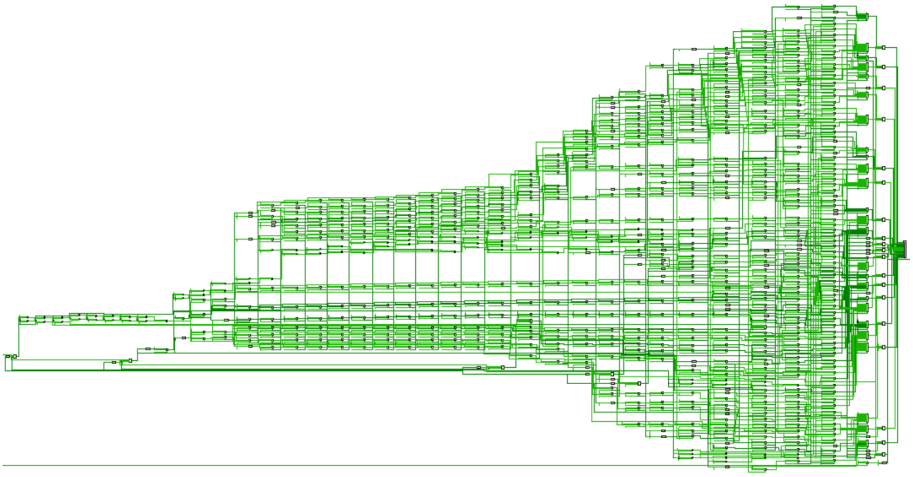

Figure 7: RTL design of the project

of IR respectively. What we have had done up to this point was fetching and decoding. After the third clock cycle (i.e., T = 2), the instruction starts to execute based on the OPCODE which is located at the first 6 bits of Instruction. After executing each operation the sequence counter is set to zero so that the system can read the next instruction from the memory.

5 CONCLUSION

Even though project was hard a bit, it showed us the other side of computers. There were two main difficulties we have faced throughout the project. The first one was limited time. Since this project is assigned at the peak time of the term in terms of midterms and homeworks, we barely spare time for it. Even though the deadline is postponed one week thanks to objections, this project also affected other homeworks, quizes, and

midterms. The other biggest obstacle was the fact that TA was answering the questions in the message board in Ninova E-Learning System too late. This caused us to wait at the same point for a while. Apart from these, we think the things we have learnt through this project will help us in the final exam and overall design of a computer a lot.