**ISTANBUL TECHNICAL UNIVERSITY**

**COMPUTER ENGINEERING DEPARTMENT**

# BLG 222E
# COMPUTER ORGANIZATION
# PROJECT 2 REPORT

**CRN** : 21335

**LECTURER** : Deniz Turgay Altılar

**GROUP MEMBERS:**

150210053 : NUREFŞAN ALTIN

820210315 : MELİSA GÜLER

**SPRING 2024**

# Contents

# 1 INTRODUCTION [10 points]

In this project, we learned the basics of how to design a computer architecture according to the specifications given during the project. This involved understanding instruction formats, defining opcodes and adjusting register selections, implementing instruction execution logic, testing thoroughly, and documenting the entire process. The instructions are stored in memory in little-endian order and the 8-bit output of RAM. Since our memory is 8 bit and instructions are 16 bit, we designed a mechanism to fetch instructions in two clock cycles, loading the LSB and MSB separately.After fetching, we stored opcodes in 64-bit registers. This implementation has made operations more efficient. Although we lose one clock cycle, it proves to be more helpful in subsequent steps. We have 2 different instruction formats, each have specific fields such as OPCODE, RSEL, ADDRESS, S, DSTREG, SREG1, and SREG2. The table which is given us in project file provided a reference for understanding the purpose of each instruction. Then we determined the steps for each opcode and how to execute them. In implementing the instruction execution logic, we had to carefully consider the timing requirements specified in the project. Overall, the project involved a systematic approach to designing, implementing, testing, and documenting a computer architecture system according to specific requirements.

# 2 MATERIALS AND METHODS [40 points]

In this project, we took our ALU system from the project 1 to the next level by introducing a control unit capable of executing instructions stored in the RAM. Our project began with a thorough analysis of the instructions, followed by the creation of control signals tailored to each instruction.

## 2.1 Counter implementation

Our first task was to establish a timing mechanism within the system. The output of the 8-bit register T is fed into a decoder to generate distinct time intervals labeled T0 to T7. Our mechanism works compatible with reset input and resets itself when (Reset==0), like it is given in the simulation files. We handled this case in the execution block by assigning (8'd1 to the register T) in case of (Reset==0).

## 2.2 Initial begin

To make our system work optimally, we create an initial block at the beginnig of our CPUSystem module. In this block, we clear all of the registers and scratch registers in RF,

and all registers in ARF. We also disabled memory to avoid any unwanted interactions with memory.

## 2.3 FETCHING PART 1

IROut[15:8] — M[PC]

During the first clock cycle (T=0), the initial step in the instruction execution sequence occurs, involving the retrieval of data from RAM into the least significant byte (LSB) of the instruction register (IR). Here we also increment PC to show next part of the current instruction.

## 2.4 FETCHING PART 2

IROut[7:0] — M[PC]

The process continued into the second clock cycle (T=1), where the most significant byte (MSB) was fetched from the next memory address and loaded into the MSB of the IR. PC is incremented again so that it points the next instruction in memory.

## 2.5 DECODING

O[0]-O[33] — IROut[15:10]

Decoding commences from T2 onward, enabling the identification of the instruction to be executed. This decoding process involves mapping the relevant bits from the IR onto a decoder, yielding 34 output bits labeled O0 to O33.

In this cycle, we also fetch the values of RSel, S, DSTREG, SREG1 and SREG2 and assigned them to corresponding registers. In the following commands, we deactivate the mem WR and unable all registers, R1,2,3,4, S1,2,3,4, PC,AR,SP in order to not change them by mistake.

## 2.6 EXECUTION

*Instruction format 1

In this part, we encounter with the problem of dealing with the memory. We read data from memory and write to the memory. Since we are fetching data from memory and writing to memory these operations take more cycle to conclude. In the execution of BRA, BNE, BEQ, MOVH, MOVL, and STRIM operations we used directly the value in address bits of Instruction Register for VALUE, IMMEDIATE and OFFSET. But for the LDRIM, we used the value in the memory, at the location defined in Address bits of Instruction Register.

POP and PUSH operations are performed by assuming that the SP shows the empty location in memory, therefore we first increment and than write for the POP and first decrement and than write for the PUSH.

For all these operations since the memory is 8 bit but the registers are 16 bit, we performed them by incrementing/ decrementing the SP and AR 2 times. By this way we could write and read correct places.

*Instruction format 2

In this part we deal with 3 different registers, SREG1,SREG2 and DSTREG. We noticed that, there is 2 different kind of operations, the ones with one operand and the others with 2 operand. While implementing these operations, we also see that, only the FunSel of the ALU will change according the type of the operation so we performed these grouped types with the help of 'or' and execute them together.

In the first case, we used only SREG1 and DSTREG. Then we divided this case into 4 different cases since these two can be in RF or ARF. These 4 four cases:

SREG1 DSTREG

RF, RF

RF, ARF

ARF, RF

ARF,ARF

In our 2nd case, we used SREG1, SREG2 and DSTREG. We divided this part into 8 different cases according to the relationship between them. These 8 cases:

SREG1 SREG2 DSTREG

RF, RF, RF

RF, RF, ARF

RF, ARF, RF

RF, ARF, ARF

ARF, RF, RF

ARF, RF, ARF

ARF, ARF, RF

ARF, ARF, ARF

After realizing that increment/decrement operations are performed after assigning to DSTREG, we performed increment and decrement operations separately.

In order to change flags, we utilize WF and S's value. We make the WF enable when any operation comes with the requirement of a flag change.

In the end, after each operations we put a reset part to prepare the system for the next instruction. We clear the registers and the ALUOut, and also we reset the clock here.

| 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) |

Figure 1: OPCODE field and symbols for operations and their descriptions

# 3 RESULTS [15 points]

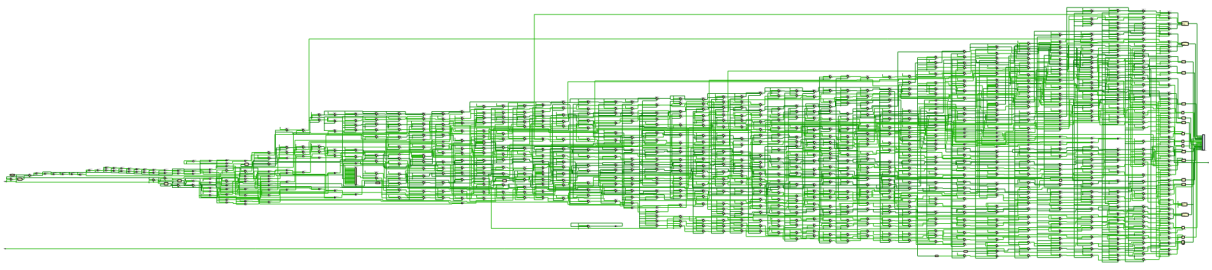

Figure 2: Schematic for Control Unit Module


Figure 3: XOR operation while IR value is equal to 3DA5

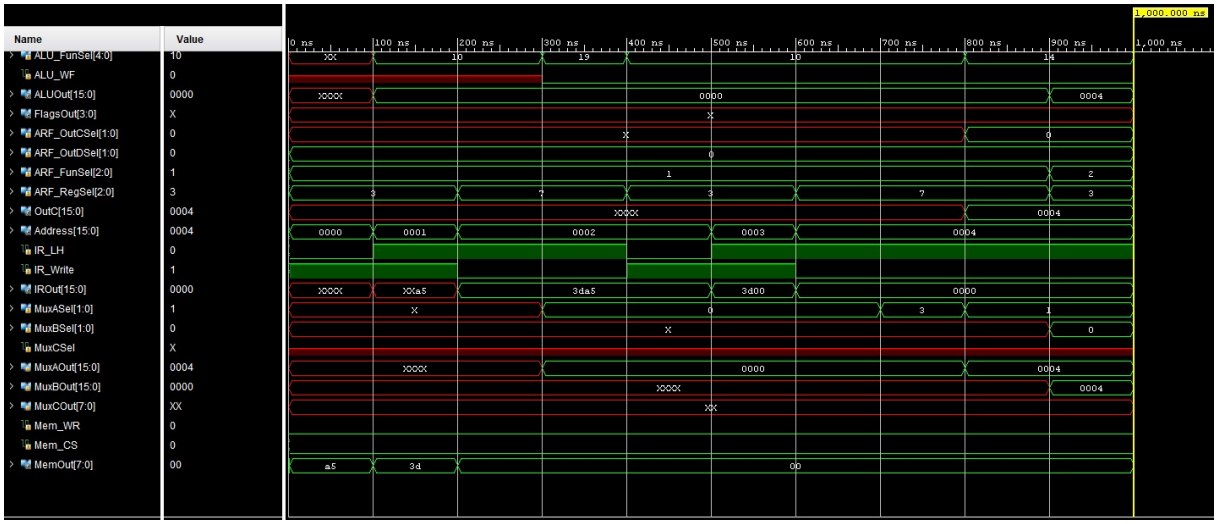


Figure 4: XOR operation while IR value is equal to 3DA5

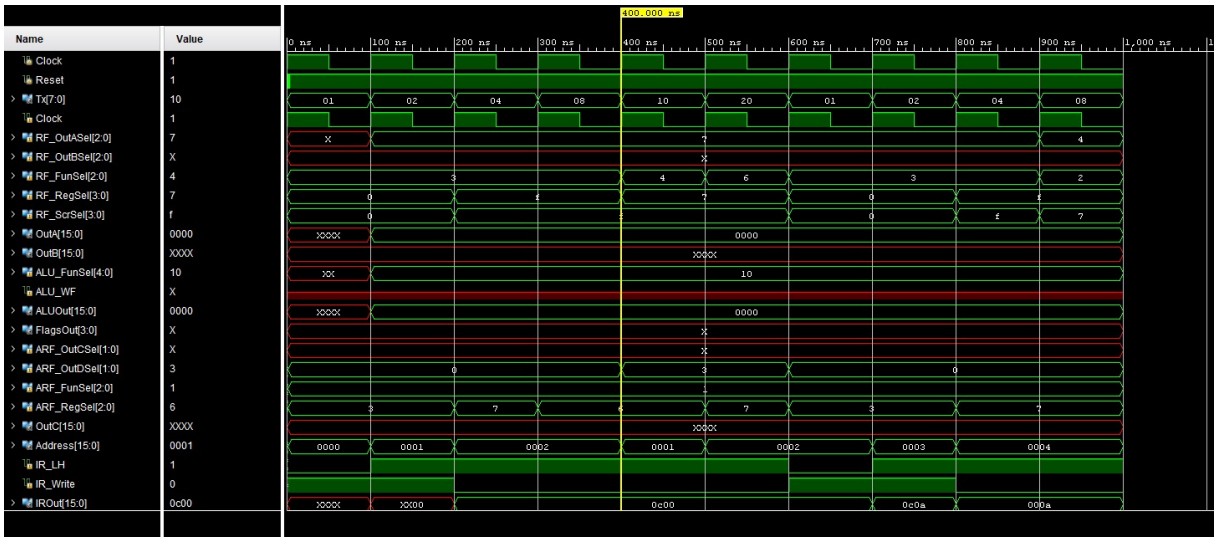


Figure 5: POP operation while IR value is equal to 0C00

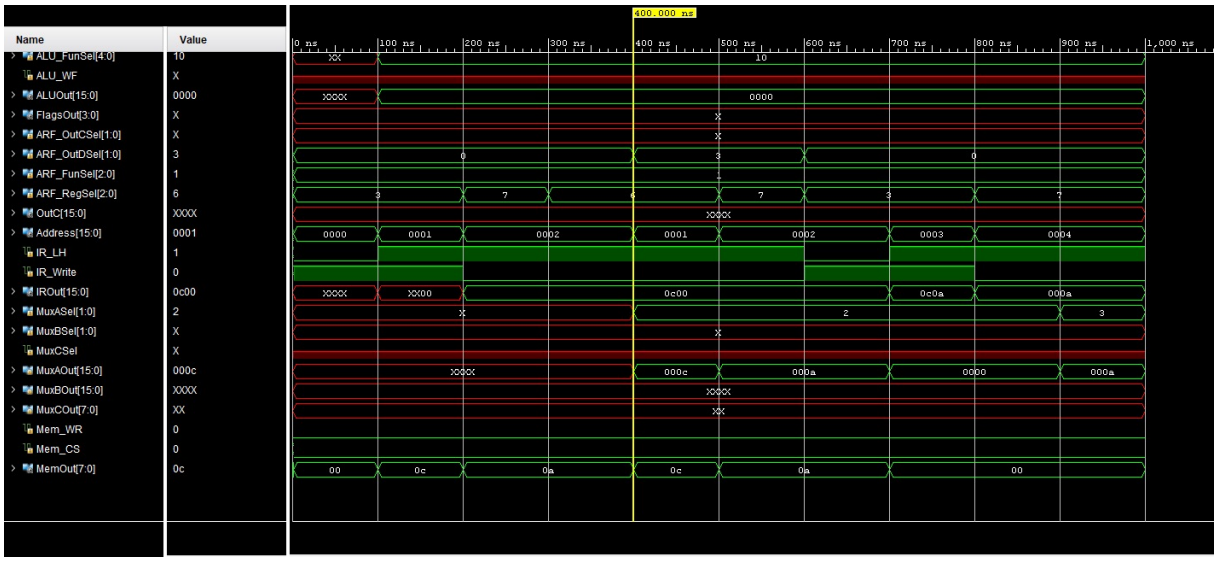


Figure 6: POP operation while IR value is equal to 0C00

# 4 DISCUSSION [25 points]

## 4.1 PART 1(Instructions with an address reference)

The processor uses little-endian byte ordering, meaning the least significant byte (LSB) comes first in memory. The instruction register (IR) is 16 bits wide, but the memory can only output 8 bits at a time. To overcome this limitation, instructions are fetched in two clock cycles: In the first cycle (T=0), the LSB of the instruction is loaded from memory address to the LSB of IR (IR(7-0)).In the second cycle (T=1), the MSB of the instruction is loaded from memory address to the MSB of IR (IR(15-8)). In Instructions with address

reference part, these instructions include an opcode (6 bits) that defines the operation to be performed. They also have an RSEL field (2 bits) that selects a register based on the values in Table. Finally, the included ADDRESS field (8 bits) that specifies the memory location for data operand or direct value of the operand.

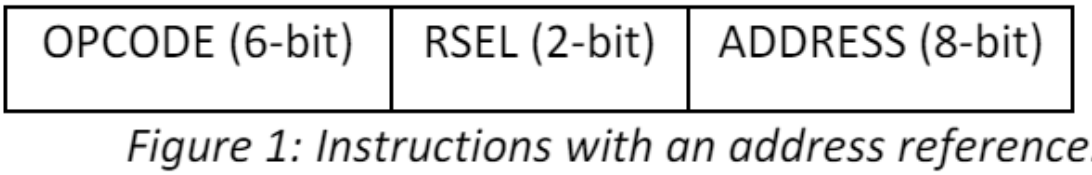


*Figure 1: Instructions with an address reference.*

Figure 7: : Instructions with an address reference

### 4.2 PART 2 (Instructions without an address reference)

In the second part of the project, we designed the operations without memory. These instructions operate only on data stored in the processor's registers. In this design, we used a 6-bit opcode, a 1-bit S, 3 bit SREG1 and SREG2 and a 3-bit DSTREG. The opcode remains the same as with address-reference instructions, specifying the operation the processor needs to perform (e.g., addition, subtraction, logical operations). S provides flag functionality, allowing us to control the execution status of operations. S = 0: When set to 0, the execution of the command does not affect the processor's status flags (such as zero flag, carry flag). This allows for faster execution without changing flags that may not be relevant to the current operation. S = 1: When set to 1, the execution of the command will update the processor's status flags depending on the result of the operation. This is important for conditional branching or operations based on flag values. DSTREG identifies the destination register where the operation result will be stored. SREGs these fields specify the source registers that hold the data operands for the operation.

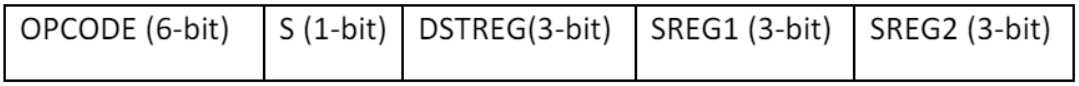


Figure 8: Instructions without an address reference.

## 5 CONCLUSION [10 points]

The most difficult part of the assignment was to find the stages of the given opcodes with the help of the diagram and to convert them into bit operations. The variety and

complexity of instruction formats and operations posed a significant challenge. Understanding the purpose and behavior of each instruction, as well as implementing them accurately, required meticulous attention to detail and thorough testing. It required careful consideration and planning to ensure correct interpretation and handling of instruction bytes. for this we also used the tables given in the previous project. at the same time, although there was confusion from time to time while calculating the clock cycles, we finally managed to get the desired outputs.Also, we experienced delays in clock cycle calculations from time to time and we had difficulty in simulating this situation while finding a solution.Throughout the project, the provided tables served as invaluable references for defining opcode meanings, register selections, and other relevant information. This aided in writing clear and concise code for instruction execution. After the first project, we were more inclined to use Verilog and we think we have a better grasp of the concept in general. Despite these challenges, we emerged with a wealth of experience in hardware description language (Verilog) and hardware system design.

# REFERENCES