ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 222E
COMPUTER ORGANIZATION
PROJECT 2 REPORT

CRN : 21335

LECTURER : Prof. Dr. Deniz Turgay Altılar

GROUP MEMBERS:

150210083 : Kemal Tahir Bıcılıoğlu (Group Representative)

150210097 : Ali Emre Kaya

SPRING 2024

Contents

1 INTRODUCTION [10 points]

In the second project of Computer Organization lecture, we will design a hardwired control unit for the arithmetic logic unit system which already designed in the first project. There are a limited number of operations for CPU system, we will implement 34 different operations for CPU system with using Verilog. The control unit will arrange control inputs which of the ALU system according to selected operation.

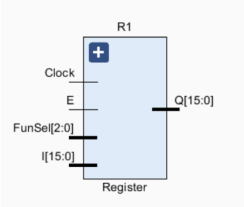
The CPU system is designed to execute a variety of instructions through a sequence of tasks. Each task corresponds to a specific operation, such as logical operation calculation or data transfering, and is implemented through a set of control signals and state transitions. The primary components involved include a sequence counter, a decoder, an arithmetic logic unit (ALU), and various registers. The sequence counter and decoder manage the timing and control flow, while the ALU and registers handle data processing and storage.

Task execution begins with the initialization and setting of control signals to manage data flow and operations across various components. Each task uses control signals to select specific registers, set ALU operations, and manage the sequence counter's progression. Tasks are divided into states according to the sequence counter value, with each state performing specific operations such as loading data into registers, executing ALU operations, or updating the PC. State transitions are managed by incrementing or resetting the sequence counter, enabling sequential execution of multi-step operations.

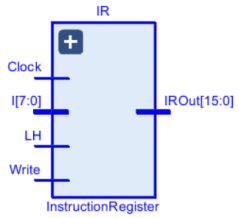
2 MATERIALS AND METHODS [40 points]

2.1 Materials

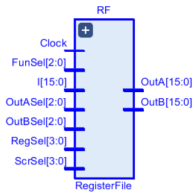
- Verilog
- Register



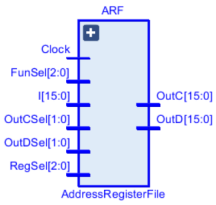
- Instruction Register



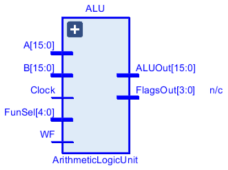
- Register File



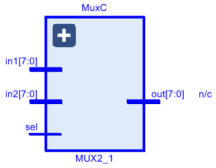
- Address Register File



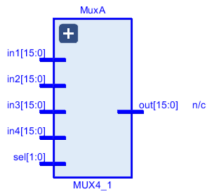
- Arithmetic Logic Unit



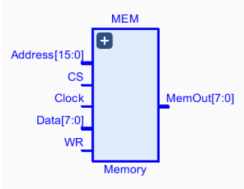
- 2 to 1 Multiplexer



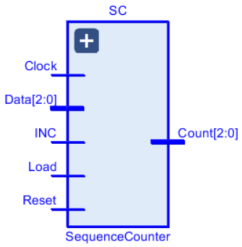
- 4 to 1 Multiplexer



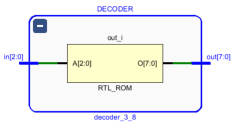
- Memory Unit



- Sequence Counter



- 3 to 8 Decoder



2.2 Methods

The instruction loading process occurs in two clock cycles due to the limitations of the RAM output becuase of the RAM limitation. The RAM used in Project 1 has an 8-bit output, necessitating the two-clock-cycle loading process for the instruction register.

Instructions are stored in little-endian order in memory, meaning that the LSB of the instruction occupies the lower memory address, while the MSB occupies the next consecutive memory address.

8-bit T value is given to us, it will determine the operation to run. It only let to be one bit to one, others will be zero.

Instruction Loading Process:

- During the first clock cycle ($T = 0$), the LSB of the instruction is loaded from memory address A to the LSB of the instruction register (IR), denoted as $IR(7-0)$.
- In the subsequent clock cycle ($T = 1$), the MSB of the instruction is loaded from memory address $A+1$ to the MSB of the instruction register, denoted as $IR(15-8)$.
- Following the second clock cycle ($T = 2$), the instruction is fully loaded into the instruction register and ready for execution.

Two different instruction format is given for this project; with address reference and without address reference.

OPCODE(6-bit)	RSEL(2-bit)	ADDRESS(8-bit)

Table 1: Instructions with an address reference

OPCODE(6-bit)	S(1-bit)	DSTREG(3-bit)	SREG1(3-bit)	SREG2(3-bit)

Table 2: Instructions without an address reference

The registers to be used are selected based on the output of the values which are $RSEL$, $DSTREG$, $SREG1$ and $SREG2$ in the tables below. If instruction which has address reference were chosen, $RSEL$ will be used. If instruction which has not address reference were chosen, $DSTREG$, $SREG1$ and $SREG2$ will be used.

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

Table 3: RSel table

DSTREG/SREG1/SREG2	**REGISTER**
000	PC
001	PC
010	SP
011	AR
100	R1
101	R2
110	R3
111	R4

Table 4: DSTREG/SREG1/SREG2 selection table

All 34 operations are given below. Highlighted with yellows represent the address reference instructions, highlighted with blues represent the instructions without an address reference.

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]

In the project, we did not have proper simulation files to use while debugging so we wrote some of the connections to the simulation file to see them in the waveform diagram. In the implementation, we have a ResetAll() task that initializes all the components as 0 or disabled. We utilized the assignments of the ResetAll() task to give legitimate values that could be used whether the implementation is correct or not. The reason why we added a wire to the simulation as REACHED is that sometimes we had trouble seeing correct results from the wavefrom and suspected that it was not entering the correct code block. Adding the REACHED value to the current operation, we ensured that we were doing the correct decoding.

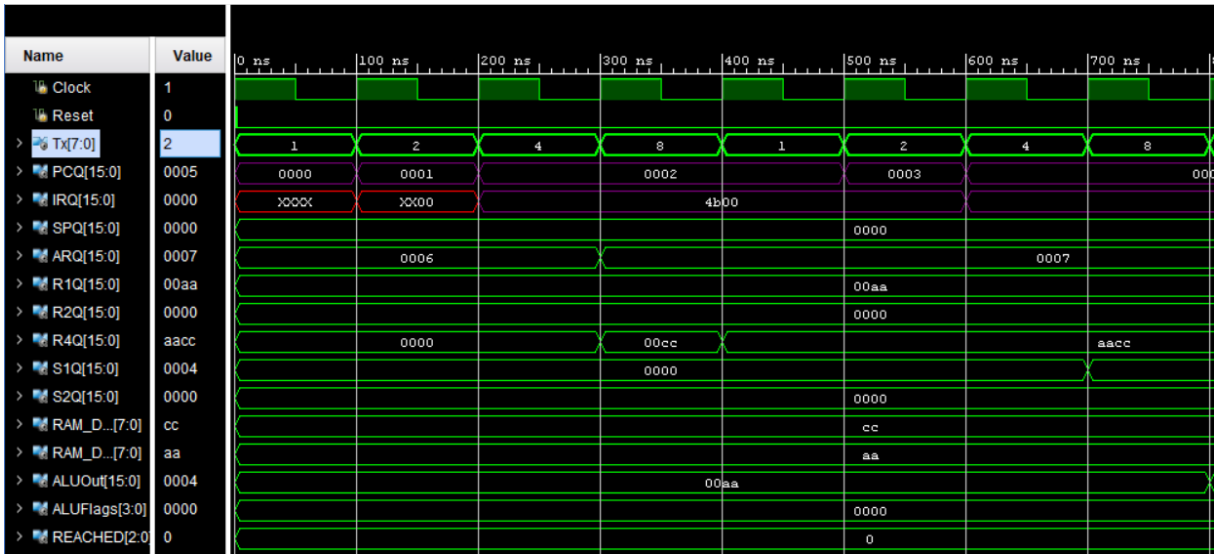


Figure 2: LDR Instruction

Initial values for debugging the LDR Instruction are:
PC = 0, it will read the first line of the memory and second line of the memory in the next clock cycle, Memory: LSB 00000000 MSB 01001011 meaning that we need to write 00 4B to the memory, 010010 is the corresponding operation code for LDR, 11 is the R4 for the regsel value, and the LSB is 0 because it is not used.

$$\text{AR.Q} = 0006,\ \text{R4} = 0003,\ \text{MEM}[7.\ \text{line}] = \text{cc},\ \text{MEM}[8.\ \text{line}] = \text{aa}.$$

We are going to observe that in the first and the second clock signals, IR will be the instruction to be executed, 4b00. When Tx = 4, control signals will be set to write memory's 7.line to Rx = R4' LSB, MEM[7. line] = cc. When Tx = 4, control signals will be set to write memory's 7.line to Rx = R4' MSB, MEM[8. line] = aa. After the execution is finished, clock should be reseted so that it will turn back to the fetch and

decode. We can observe that Rx = R4 is now aacc meaning that we correctly implemented LDR instruction. R4 < − MEM[AR]

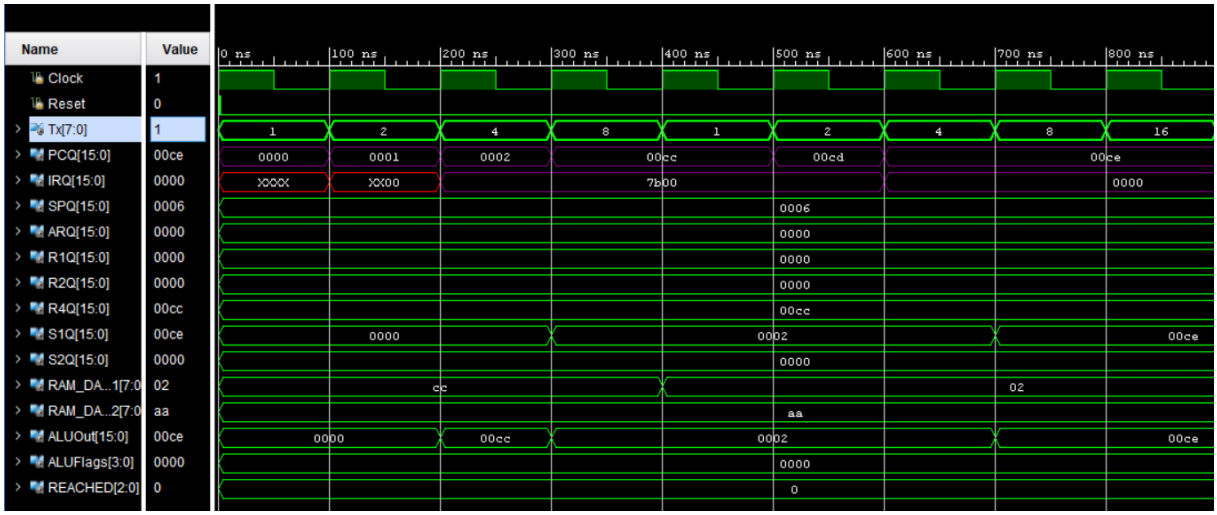


Figure 3: BX Instruction

Memory: 00—4B, SP.Q = 0006, R4 = 00cc, MEM[7. line] < − PC, PC < − R4

Figure 4: AND Instruction

Memory: E0—30, SREG1 = R1, SREG2 = PC,
DESTREG = AR => AR = R1 AND PC

Figure 5: MOVH Instruction

Memory: E0—44, AR[15:8] = IMMEDIATE

4 DISCUSSION [25 points]

In the beginning of the project, we analyse the CPU simulation file and we determine necessary values of the CPU system modules which are Clock, Reset and 8-bit T. After that we initialized the Sequence Counter, 3 to 8 Decoder and Arithmetic Logic Unit System. decoder will decode T and select which task will execute. The sequence counter (SC) tracks the current step within an instruction's execution cycle, advancing through defined states based on control signals (SC_Inc and SC_Reset). The decoder translates the current count from the sequence counter into specific control signals, enabling precise execution of each task by generating control signals for different CPU components based on the current step. The **ResetAll** task's purpose is disable all the components before the simulation start, it does components 0 or disable, its only called in the beginning of the program. In each clock cycle, some of the components are enabled. After they are used,they need to be turned into the disabled state so that we do not modify registers that are not supposed to change, to provide this process we build **DisableUnits** task.

We firstly did the instructions with address reference, for 11 different address register operations, our operation implementations can be explained briefly as follows:

BRA Operation performs an unconditional branch operation, updating the Program Counter (PC) to a new address computed as `PC + immediate value from instruction`.

BNE Operation performs a conditional branch if not equal (BNE). It checks the zero flag (Z) in the ALU flags, and if the flag is not set (indicating that the result of the

previous operation was not zero), it performs the branch operation by calling `OP_BRA`. If the zero flag is set, it resets the sequence counter.

BEQ Operation performs a conditional branch if equal (BEQ). It checks the zero flag (Z) in the ALU flags, and if the flag is set (indicating that the result of the previous operation was zero), it performs the branch operation by calling `OP_BRA`. If the zero flag is not set, it resets the sequence counter.

POP Operation performs a pop operation, which involves reading a value from the stack and storing it in a specified register.

PSH Operation performs a push operation, where a value from a register is pushed onto the stack. The value is split into two parts: the least significant byte (LSB) and the most significant byte (MSB). First, the LSB is written to memory at the stack pointer location, then the stack pointer is incremented, and finally, the MSB is written to the new stack pointer location.

LDR Operation performs a load operation, where a value from memory is loaded into a specified register. The value is read from memory at the address stored in the address register (AR) and then stored into the register. The address register is incremented to prepare for the next memory read operation.

STR Operation performs a store operation, where a value from a specified register is stored into memory. The value is split into two parts: the least significant byte (LSB) and the most significant byte (MSB). First, the LSB is written to memory at the address stored in the address register (AR), then the address register is incremented, and finally, the MSB is written to the new address register location.

BX Operation performs a branch and exchange operation. It first saves the current Program Counter (PC) value onto the stack and then updates the PC to the value stored in a specified register.

BL Operation performs a branch with link operation. It loads the PC with the value stored at the stack pointer location.

LDRIM Operation performs a load immediate operation. It loads a register with a value from memory at an address specified by an immediate value in the instruction register.

STRIM Operation performs a store immediate operation. It stores a value from a register into memory at an address computed as the sum of an immediate value and a value from another register.

After we handle all the address register instructions, we control all of them with writing our own simulations files. We manipulate **RAM** file and control the outputs, the values was as expected, so we started to handle the instructions without an address reference.

For the rest of operations, which are the instructions without address reference, out implementation logic as follows,

All Type 2 instructions will be executed in one clock cycle when the source registers (`SREG1` and `SREG2`) and destination register (`DESTREG`) are known. To generalize Type 2 instructions for all combinations of source and destination registers, the approach suggests writing `SREG1` and `SREG2` to registers `S1` and `S2`, respectively, over two clock cycles. Subsequently, the values from `S1` and `S2` will be passed to the ALU for computation. Finally, the result from the ALU will be written to the destination register (**DESTREG**). To implement this logic, we add extra 3 task whichs are **Type2InstructionSREG**, **Type2InstructionDESTREG** and **Type2InstructionTask**. Their purposes are as follows;

Type2InstructionSREG manages the control signals for Type 2 instructions involving register operations, dynamically selecting registers and setting control signals based on the input conditions provided by SREG and move_T.

Type2InstructionDESTREG manages the control signals necessary for writing the output of the ALU to the specified destination register (DESTREG). It selects the appropriate registers and sets control signals based on the input conditions provided by DESTREG.

Type2InstructionTask handles the execution of Type 2 instructions by decoding the instruction opcode and setting the appropriate control signals for different clock cycles (T2, T3, and T4). It calls specific tasks (Type2InstructionSREG and Type2InstructionDESTREG) to manage specific functionalities related to register operations and ALU operations.

This modification simplifies the handling of Type 2 instructions by generalizing the process for all combinations of source and destination registers. By standardizing the process, it reduces the need to write different code blocks for each combination of source and destination registers for each instruction. The code becomes easier to understand and maintain, as the process is consistent across all Type 2 instructions.

Overall, the CPU system's logic focuses on managing control signals, sequencing through instruction execution steps, and performing data operations using the ALU and registers. Each operation is implemented to ensure correct execution of operations, leveraging the sequence counter for precise control flow and the decoder for state-specific signal generation. This modular and systematic approach facilitated the execution of complex instructions and efficient CPU operation.

5 CONCLUSION [10 points]

We faced difficulties because simulation file is not given properly, there are lots of case to control but simulation is include only little of them. To handle that we prepare our own simulations and we made sure to implement all operations work correctly. It improved our understandable and knowledge on clock's function.

In this project, we successfully designed and implemented a hardwired control unit for the arithmetic logic unit (ALU) system, as part of our Computer Organization course. Through this endeavor, we gained a deeper understanding of hardwired control unit and sequential logic circuit design.

REFERENCES