ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 222E
COMPUTER ORGANIZATION
PROJECT 1 REPORT

CRN : 21336

LECTURER : Mustafa Ersel Kamaşak

GROUP MEMBERS:

150210061 : METİN ERTEKİN KÜÇÜK

150200059 : MEHMET ALİ BALIKÇI

SPRING 2024

Contents

1 TASK DISTRIBUTION

Metin Ertekin Küçük: Report Preparing, Last 17 Operations
Mehmet Ali Balıkçı: First 17 Operations, Overall Design

2 INTRODUCTION

In this report, we have designed a hardwired control unit for a specific architecture. The structure used for this design is based on Part 4 of Project 1. We will outline the instruction format, types of instructions, opcode fields, register selections, and the symbols for operations along with their descriptions.

The instructions are stored in memory in little-endian order. The instruction register (IR) cannot be filled in one clock cycle due to the RAM's 8-bit output. Therefore, the MSB and LSB of the instruction are loaded in two clock cycles as follows:

- **Clock Cycle 1 (T=0):** Load the LSB of the instruction from memory address A to the LSB of IR.
- **Clock Cycle 2 (T=1):** Load the MSB of the instruction from memory address A+1 to the MSB of IR.
- After the second clock cycle (T=2), the instruction starts to execute.

2.1 Instructions with Address Reference

Instructions with address reference have the following format:

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

2.2 Instructions without Address Reference

Instructions without address reference have the following format:

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

3 Design

Our computer system design consists of several modules, each responsible for specific functions within the system. Let's examine the design details based on the provided Verilog implementation:

3.1 Sequence Counter

The `SequenceCounter` module is responsible for generating sequential timing signals essential for instruction execution. It counts clock cycles and controls the timing of various operations within the system.

3.2 Decoder 3-to-8

The `Decoder3to8` module decodes the timing signals generated by the sequence counter, producing individual time decoder outputs for different stages of the instruction execution. These decoded signals facilitate synchronization and control of operations at each stage.

3.3 CPU System

The `CPUSystem` module serves as the heart of our computer system, orchestrating instruction fetching, decoding, and execution. Let's break down its key components and functionalities:

3.3.1 Instruction Fetch and Decode

During the instruction fetch phase, the system loads the instruction register (IR) with the opcode and operands fetched from memory. This process spans two clock cycles, as per the timing requirements. The opcode is extracted from the IR during the second clock cycle, initiating the instruction decoding process.

3.3.2 Instruction Execution

Upon decoding the opcode, the system determines the type of instruction and executes the corresponding operation. Various control signals are generated based on the opcode and operands, directing the flow of data and control within the system.

3.3.3 Arithmetic Logic Unit (ALU) System

The ALU subsystem performs arithmetic and logic operations required by certain instructions. It receives inputs from the register file (RF) and generates output based on the selected operation. Control signals dictate the ALU function and operand selection, ensuring correct computation according to the instruction being executed.

3.3.4 Register File (RF) and Address Register File (ARF)

The register file stores general-purpose registers used for data manipulation, while the address register file maintains special-purpose registers such as the program counter (PC) and stack pointer (SP). These registers are accessed based on control signals and operand selections determined during instruction execution.

3.3.5 Memory Interface

The system interacts with memory to fetch instructions and access data as required by certain instructions. Control signals regulate memory read and write operations, ensuring proper data retrieval and storage during program execution.

3.4 Overall System Operation

The system operates in a pipelined fashion, with each stage of the instruction execution chain overlapping to maximise throughput. Instructions are fetched, decoded and executed sequentially, with the system maintaining proper timing and synchronisation to ensure correct operation.

3.5 Verification and Testing

To validate the functionality and correctness of our design, extensive simulation and testing were performed. Print statements were added to verify the behavior of individual modules and the system as a whole, ensuring compliance with the specified instruction set architecture and timing requirements.

4 RESULTS

As a result, we observe some outputs as a result of the changes we made to our verilog simulation file and the RAM.MEM file designed to run in it.

For simplicity, we have ignored the fetch and decode cycles and for BNE operation, we showed first the non-working and then the working states:

4.1 BNE operation

```
Output Values:
T:   4
Address Register File: PC:     2, AR:     0, SP:     0
Instruction Register :  1064
Register File Registers: R1:     0, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: 1, N: 1, C: 1, O: 1
ALU Result: ALUOut:     x
```

```
Output Values:
T:   1
Address Register File: PC:     2, AR:     0, SP:     0
Instruction Register :  1064
Register File Registers: R1:     0, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: 1, N: 1, C: 1, O: 1
ALU Result: ALUOut:     x
```

The above simulation results shows that if Z is different from zero, the PC + VALUE operations does not run and returns back to the cycle 1.

```
Output Values:
T:   4
Address Register File: PC:     2, AR:     0, SP:     0
Instruction Register :  1064
Register File Registers: R1:     0, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: 0, N: 0, C: 0, O: 0
ALU Result: ALUOut:     x
```

```
Output Values:
T:   8
Address Register File: PC:     2, AR:     0, SP:     0
Instruction Register :  1064
Register File Registers: R1:     0, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:    40, S2:     0, S3:     0, S4:     0
ALU Flags: Z: 0, N: 0, C: 0, O: 0
ALU Result: ALUOut:    40
```

Figure 1: IF Z=0 THEN PC ← PC + VALUE

The above simulation results shows that if Z is equal to zero, the PC + VALUE operations starts to run and increases ALUOut.

4.2 LSL Operation

```
Output Values:
T:   4
Address Register File: PC:     2, AR:     x, SP:     x
Instruction Register :  7520
Register File Registers: R1:    10, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:    10
```

```
Output Values:
T:   8
Address Register File: PC:     2, AR:     x, SP:     x
Instruction Register :  7520
Register File Registers: R1:    10, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:    20
```

```
Output Values:
T:  16
Address Register File: PC:     2, AR:     x, SP:     x
Instruction Register :  7520
Register File Registers: R1:    10, R2:    20, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:    20
```

```
Output Values:
T:   1
Address Register File: PC:     2, AR:     x, SP:     x
Instruction Register :  7520
Register File Registers: R1:    10, R2:    20, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:    20
```

Figure 2: DSTREG ← LSL SREG1

Above operation makes logical shift left for SREG1 and assigns it to the DSTREG

4.3 AND Operation

```
Output Values:
T:   4
Address Register File: PC:     2, AR:     0, SP:     0
Instruction Register : 12640
Register File Registers: R1:    10, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: 0, N: 0, C: 0, O: 0
ALU Result: ALUOut:     x
OpCode: 0c
```

```
Output Values:
T:   8
Address Register File: PC:     2, AR:     0, SP:     0
Instruction Register : 12640
Register File Registers: R1:    10, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: 0, N: 0, C: 0, O: 0
ALU Result: ALUOut:     x
OpCode: 0c
```

```
Output Values:
T:  16
Address Register File: PC:     2, AR:     0, SP:     0
Instruction Register : 12640
Register File Registers: R1:    10, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     2, S3:     0, S4:     0
ALU Flags: Z: 0, N: 0, C: 0, O: 0
ALU Result: ALUOut:     2
OpCode: 0c
```

```
Output Values:
T:  32
Address Register File: PC:     2, AR:     0, SP:     0
Instruction Register : 12640
Register File Registers: R1:    10, R2:     2, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     2, S3:     0, S4:     0
ALU Flags: Z: 0, N: 0, C: 0, O: 0
ALU Result: ALUOut:     2
OpCode: 0c
```

Figure 3: DSTREG ← SREG1 AND SREG2

We tested this operation by assigning 60 to the first line and 31 to the second line in the MEM file. Then the operation gave the correct result. Here, too, we skipped the fetch and decode parts for simple demonstration.

4.4 ADD Operation

```
Output Values:
T:   4
Address Register File: PC:     2, AR:     0, SP:     0
Instruction Register : 21605
Register File Registers: R1:    10, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: 0, N: 0, C: 0, O: 0
ALU Result: ALUOut:     x
OpCode: 15
```

```
Output Values:
T:   8
Address Register File: PC:     2, AR:     0, SP:     0
Instruction Register : 21605
Register File Registers: R1:    10, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: 0, N: 0, C: 0, O: 0
ALU Result: ALUOut:     x
OpCode: 15
```

```
Output Values:
T:  16
Address Register File: PC:     2, AR:     0, SP:     0
Instruction Register : 21605
Register File Registers: R1:    10, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: 0, N: 0, C: 0, O: 0
ALU Result: ALUOut:    10
OpCode: 15
```

```
Output Values:
T:   1
Address Register File: PC:    10, AR:     0, SP:     0
Instruction Register : 21605
Register File Registers: R1:    10, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: 0, N: 0, C: 0, O: 0
ALU Result: ALUOut:    10
OpCode: 15
```

Figure 4: DSTREG ← SREG1 + SREG2

We assigned our assigned values to the RAM.MEM file as first line 65, second line 54. So that means we are summing R1 and R2 and assigning it to the PC. And the figures shows that our operation done correctly.

4.5 STRIM Operation

```
Output Values:
T:   4
Address Register File: PC:     2, AR:    10, SP:     0
Instruction Register : 33800
Register File Registers: R1:    10, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: 0, N: 0, C: 0, O: 0
ALU Result: ALUOut:     x
OpCode: 100001
```

```
Output Values:
T:   8
Address Register File: PC:     2, AR:    10, SP:     0
Instruction Register : 33800
Register File Registers: R1:    10, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     8, S2:     0, S3:     0, S4:     0
ALU Flags: Z: 0, N: 0, C: 0, O: 0
ALU Result: ALUOut:     8
OpCode: 100001
```

```
Output Values:
T:  16
Address Register File: PC:     2, AR:    10, SP:     0
Instruction Register : 33800
Register File Registers: R1:    10, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     8, S2:    10, S3:     0, S4:     0
ALU Flags: Z: 0, N: 0, C: 0, O: 0
ALU Result: ALUOut:    18
OpCode: 100001
```

```
Output Values:
T:  32
Address Register File: PC:     2, AR:    18, SP:     0
Instruction Register : 33800
Register File Registers: R1:    10, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     8, S2:    10, S3:     0, S4:     0
ALU Flags: Z: 0, N: 0, C: 0, O: 0
ALU Result: ALUOut:    10
OpCode: 100001
```

```
Output Values:
T:  64
Address Register File: PC:     2, AR:    19, SP:     0
Instruction Register : 33800
Register File Registers: R1:    10, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: 0, N: 0, C: 0, O: 0
ALU Result: ALUOut:    10
OpCode: 100001
```

Figure 5: M[AR+OFFSET] ← Rx

In the above figure, AR and offset value are summed by using Arithmetic Logic Unit and given as output to Address register and address register writes R1 to this value

corresponding to the address in memory, then AR increases by one and writes again.

4.6 MOVH

```
Output Values:
T:   4
Address Register File: PC:     2, AR:     0, SP:     0
Instruction Register : 17731
Register File Registers: R1:    10, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: 0, N: 0, C: 0, O: 0
ALU Result: ALUOut:     x
OpCode: 010001
```

```
Output Values:
T:   8
Address Register File: PC:     2, AR:     0, SP:     0
Instruction Register : 17731
Register File Registers: R1:    10, R2:    67, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: 0, N: 0, C: 0, O: 0
ALU Result: ALUOut:     x
OpCode: 010001
```

Figure 6: DSTREG[15:8] ← IMMEDIATE (8-bit)

Above operation used 4543 hexadecimal representation and takes Instruction Register's Least Significant 8 bits and assigns them to the R2 register according to our hexadecimal Instruction. So this operation took 2 clock cycle except fetch and decode operation.

5 Discussion

We have showed some operations that designed and implemented in our project. Several notable aspects emerged for discussion throughout the development process:

5.1 Instruction Set

The instruction set provided was the foundation of our project. Understanding its format, encoding and operation was crucial to designing the system architecture and implementing the instruction execution logic in Verilog. We discussed the considerations of simplicity versus functionality in the instruction set design, taking into account factors such as instruction complexity, and ease of decoding.

5.2 Timing Requirements

The main challenge was to implement the fetch and decode steps correctly. Implementing two clock cycle loads of the instruction register (IR) required careful synchronisation and memory access handling. Through iterative testing and refinement, we ensured that our Verilog implementation met the specified timing constraints and that the instructions executed smoothly.

5.3 Verification and Testing

Verifying the correctness and functionality of our system was a critical aspect of the project. We employed simulation techniques to validate the behavior of the Verilog implementation against expected outcomes. This involved testing various instruction sequences, edge cases, and boundary conditions to verify proper operation and identify any potential issues or bugs.

5.4 Learning Outcomes

The overall result of our project was a valuable insight into the principles of computer organisation and design. It deepened our understanding of hardware implementation techniques. Through hands-on experience with Verilog and simulation tools, we gained practical skills that can be applied to real-world hardware development projects. In addition, the collaborative nature of the project developed teamwork, communication and problem-solving skills that are essential for success in the field of computer engineering.

6 CONCLUSION

In this project, we successfully designed and implemented a computer system based on the provided instruction set architecture. Although there were challenges, our Verilog implementation effectively executes instructions according to timing requirements.

This project has improved our understanding of computer organisation and design principles, provided us with practical experience and a basis for future projects in hardware system development. It also took us a long time to make sense of the complex instructions, but in the end we learned how a simple computer works and how to simulate it in a simple way.