ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 222E

COMPUTER ORGANIZATION
PROJECT 2 REPORT

CRN	:	30307
LECTURER	:	Dr. Kadir Özlem

GROUP MEMBERS:

150210032	:	ARİF EREN YOLDAŞ
110190202	:	ANIL ARDA TEVEK

SUMMER 2025

Contents

1 INTRODUCTION

In this project, we designed and implemented a hardwired control unit for the system we developed in project 1. We tweaked the ALU System from the earlier project according to test errors. According to instruction set given in the project we created a CPU system that is able to fetch, decode and execute the given 37 instructions. As a group we did not have a clear division of labor and helped each other where we can.

2 MATERIALS AND METHODS

Modules we have designed and implemented are described below.

2.1 CPU System

CPU System implements the ALU System from the project before with additional control logic.

2.2 Selector Modules

To decode DSTREG/SREG1/SREG2 and RSEL values from the instruction we added two additional modules to the CPU System named RegisterSelector and ARFSelector respectively.

RSEL	REGISTER
00	R1
01	R2
10	R3
11	R4

Figure 1: Regsel Control Inputs

RegisterSelector module decodes 2-bit RSEL from address reference instructions to generate encoded register selection signals for Register File (RF)

```
module RegisterSelector (
    input [1:0] in,
    output reg [3:0] out
    );

    always @(*) begin
        case (in)
            2'b00: out = 4'b1000;
            2'b01: out = 4'b0100;
            2'b10: out = 4'b0010;
            2'b11: out = 4'b0001;
            default: out = 4'b0000;
        endcase
    end
endmodule
```

Figure 2: Register Selector Module

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

Figure 3: Regsel Control Inputs

ARFSelector module decodes the 3-bit DSTREG/SREG1/SREG2 field for register reference instructions. It generates selection signals for RF and ARF (PC, AR, SP)

```verilog
module ARFSelector (
    input [2:0] in,
    output reg [3:0] rf_out,
    output reg [2:0] arf_out
    );

    always @(*) begin
        case (in)
            3'b000: arf_out = 3'b100;    // PC
            3'b001: arf_out = 3'b010;    // SP
            3'b010: arf_out = 3'b001;    // AR
            3'b011: arf_out = 3'b001;    // AR
            3'b100: rf_out = 4'b1000;    // R1
            3'b101: rf_out = 4'b0100;    // R2
            3'b110: rf_out = 4'b0010;    // R3
            3'b111: rf_out = 4'b0001;    // R4
            default: begin
                rf_out = 4'b0000;
                arf_out = 3'b000;
            end
        endcase
    end
endmodule
```

Figure 4: ARF Selector Module

2.3 Fetch and Decode Cycle

At T0 LSB of the Instruction Register (IR) is loaded, and next cycle T1 the MSB of the IR is loaded and Program Counter is incremented.

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

Figure 5: Instruction With Address Reference

OPCODE (6-bit)	**DSTREG (3-bit)**	**SREG1 (3-bit)**	**SREG2 (3-bit)**	**0**

Figure 6: Instruction Without Address Reference

```
always @(*) begin
    if (!Reset) begin
        DisableAll();
        ClearRegisters();
    end
    DisableAll();

    if (T[0]) begin // IR[7:0] <- M[PC]
        ARF_OutDSel = 2'b00; // PC
        Mem_CS = 1'b0;
        Mem_WR = 1'b0;
        IR_Write = 1'b1; // Write Enable
        IR_LH = 1'b0;
        ARF_RegSel = 3'b100; // PC
        ARF_FunSel = 2'b01;  // Increment
    end

    if (T[1]) begin // IR[15:8] <- M[PC]
        ARF_OutDSel = 2'b00;
        Mem_CS = 1'b0;
        Mem_WR = 1'b0;
        IR_Write = 1'b1;  // Write Enable
        IR_LH = 1'b1;
        ARF_RegSel = 3'b100; //PC
        ARF_FunSel = 2'b01; // Increment
    end
```

Figure 7: Verilog Code For Fetch And Decode

2.4 Execute Cycle

Instructions are executed through a series of micro-operations controlled by the timing signals. Below are examples of how some instructions are handled.

2.4.1 Stack Operations

Stack operations transfer data between memory and registers by referencing stack pointer. POPL/POPH pop 16-bit/32-bit values from the stack, while PSHL/PSHH push values onto the stack.

```
6'b000011: begin // POPL
    if (T[2]) begin
        ARF_RegSel = 3'b010;     // SP
        ARF_FunSel = 2'b01;      // increment
    end
    else if (T[3]) begin
        ARF_RegSel = 3'b010;     // SP
        ARF_FunSel = 2'b01;      // increment
        ARF_OutDSel = 2'b01;     // SP
        Mem_CS = 0;
        Mem_WR = 0;
        DR_E = 1;
        DR_FunSel = 2'b01;       // load
    end
    else if (T[4]) begin
        ARF_OutDSel = 2'b01;     // SP
        Mem_CS = 0;
        Mem_WR = 0;
        DR_E = 1;
        DR_FunSel = 2'b10;
    end
    else if (T[5]) begin  // Rx <- DR
        MuxASel = 2'b10;    // DROut
        RF_RegSel = r_sel;
        RF_FunSel = 3'b101;
        ResetT();
    end
end
```

Figure 8: Verilog Code For POPL

```verilog
            6'b000100: begin // PSHL
                if (T[2]) begin
                    RF_OutASel = {1'b0, RegSel}; // Select Reg
                    MuxDSel = 0;                    // OutA
                    ALU_FunSel = 5'b00000;          // A
                    ALU_WF = 1;
                    MuxCSel = 2'b00;                // LSB

                    ARF_OutDSel = 2'b01;            // SP
                    Mem_CS = 0;
                    Mem_WR = 1;

                    ARF_RegSel = 3'b010;
                    ARF_FunSel = 2'b00; // decrement SP
                end
                if (T[3]) begin
                    RF_OutASel = {1'b0, RegSel}; // Select Reg
                    MuxDSel = 0;                    // OutA
                    ALU_FunSel = 5'b00000;          // A
                    ALU_WF = 1;
                    MuxCSel = 2'b01;                // MSB

                    ARF_OutDSel = 2'b01;            // SP
                    Mem_CS = 0;
                    Mem_WR = 1;

                    ARF_RegSel = 3'b010;
                    ARF_FunSel = 2'b00; // decrement SP

                    ResetT();
                end
            end
```

Figure 9: Verilog Code For PSHL

2.4.2 Branch Operations

Used to modify Program Counter, BRA branches unconditionally and jumps to target address.

```
        6'b000000: begin // BRA
            if (T[2]) begin
                MuxBSel = 2'b11;      // IROut[7:0]
                ARF_RegSel = 3'b100; // PC
                ARF_FunSel = 2'b10;  // Load
                ResetT();
            end
        end
```

Figure 10: Verilog Code For BRA

```
        6'b000001: begin // BNE
            if (T[2]) begin
                if (Z == 0) begin
                    MuxBSel = 2'b11;      // IROut[7:0]
                    ARF_RegSel = 3'b100; // PC
                    ARF_FunSel = 2'b10;  // Load
                end
                ResetT();
            end
        end
```

Figure 11: Verilog Code For BNE

BNE jumps PC to target address if zero flag is 0,

```
        6'b000010: begin // BEQ
            if (T[2]) begin
                if (Z == 1) begin
                    MuxBSel = 2'b11;      // IROut[7:0]
                    ARF_RegSel = 3'b100; // PC
                    ARF_FunSel = 2'b10;  // Load
                end
                ResetT();
            end
        end
```

Figure 12: Verilog Code For BEQ

BEQ jumps PC to target address if zero flag is 1.

2.4.3 Arithmetic Operations

These are the operations that utilize the ALU to perform arithmetic operations (INC, DEC, ADD, ADC, SUB) and update destination register with the result. For example ADD operation implementation is given below.

```
        6'b010101: begin // ADD
            if (T[2]) begin
                if (SrcReg1[2]) begin   // RF registers
                    RF_OutASel = {1'b0, SrcReg1[1:0]};
                    MuxDSel = 0;
                    ALU_FunSel = 5'b10000;
                end
                else begin   // ARF registers
                    ARF_OutCSel = SrcReg1[1:0];
                    MuxDSel = 1;
                    ALU_FunSel = 5'b00000;
                end

                ALU_WF = 1;
                MuxASel = 2'b00;
                RF_ScrSel = 4'b1000;     // S1
                RF_FunSel = 3'b010;      // load
            end
```

Figure 13: Verilog Code For ADD

2.4.4 Logic Operations

Logic operations (AND, ORR, XOR, NAND, NOT) between two source registers or one source and one destination register to store the result are performed using the ALU. For example AND operation implementation is given below.

```
        6'b010001: begin // AND
            if (T[2]) begin   // S1 <- SREG1
                if (SrcReg1[2]) begin   // RF registers
                    RF_OutASel = {1'b0, SrcReg1[1:0]};
                    MuxDSel = 0;
                    ALU_FunSel = 5'b10000;
                end
                else begin   // ARF registers
                    ARF_OutCSel = SrcReg1[1:0];
                    MuxDSel = 1;
                    ALU_FunSel = 5'b00000;
                end

                ALU_WF = 1;
                MuxASel = 2'b00;
                RF_ScrSel = 4'b1000;    // S1
                RF_FunSel = 3'b010;     // load
            end
```

Figure 14: Verilog Code For AND

2.4.5 Shift Operations

Utilizing the ALU operations such as logical left/right shift, arithmetic left/right shift and circular shifts are performed (LSL, LSR, ASR, CSL, CSR). For example, Logic Shift Left implementation is provided below.

```
        6'b001011: begin // LSL
            if (T[2]) begin
                if (SrcReg1[2]) begin   // RF registers
                    RF_OutASel = {1'b0, SrcReg1[1:0]};
                    MuxDSel = 0;
                    ALU_FunSel = 5'b11011;  // 32-bit
                end
                else begin   // ARF registers
                    ARF_OutCSel = SrcReg1[1:0];
                    MuxDSel = 1;
                    ALU_FunSel = 5'b01011;  // 16-bit
                end

                ALU_WF = 1;
                MuxASel = 2'b00;
                RF_ScrSel = 4'b1000;    // S1
                RF_FunSel = 3'b010;     // load
            end
```

Figure 15: Verilog Code For Logic Shift Left

2.4.6 Data Movement Operations

These instructions (MOV, MOVL, MOVSH) move data between registers. For example, MOV instruction implementation is provided below.

```
        6'b011000: begin // MOV
            if (T[2]) begin
                if (SrcReg1[2]) begin   // RF registers
                    RF_OutASel = {1'b0, SrcReg1[1:0]};
                    MuxDSel = 0;
                    ALU_FunSel = 5'b10000;  // 32-bit
                end
                else begin   // ARF registers
                    ARF_OutCSel = SrcReg1[1:0];
                    MuxDSel = 1;
                    ALU_FunSel = 5'b00000;  // 16-bit
                end

                if (DestReg[2]) begin
                    MuxASel = 2'b00;   // ALUOut
                    RF_RegSel = rf_dest;
                    RF_FunSel = 3'b010;  // load
                end
                else begin
                    MuxBSel = 2'b00;  // ALUOut
                    ARF_RegSel = arf_dest;
                    ARF_FunSel = 2'b10;  // load
                end
                ResetT();
            end
        end
```

Figure 16: Verilog Code For MOV

2.4.7 Memory Access Instructions

Instructions (LDARL, LDARH, STAR, LDAL, LDAH, STA) for handling data transfer between registers and memory. For example, implementation for LDARL instruction is provided below.

```
            6'b011011: begin // LDARL
                if (T[2]) begin
                    ARF_RegSel = 3'b001;  // AR
                    ARF_FunSel = 2'b01;   // increment
                    ARF_OutDSel = 2'b10;   // AR
                    Mem_CS = 0;
                    Mem_WR = 0;
                    DR_E = 1;
                    DR_FunSel = 2'b01;

                end
```

Figure 17: Verilog Code For LDARL

3 RESULTS

Simulation results for CPUSystem are provided in this section.

3.1 CPUSystem Simulation

The resulting wavefrom from CPUSystem simulation is provided below.

3.1.1 Simulation Waveform

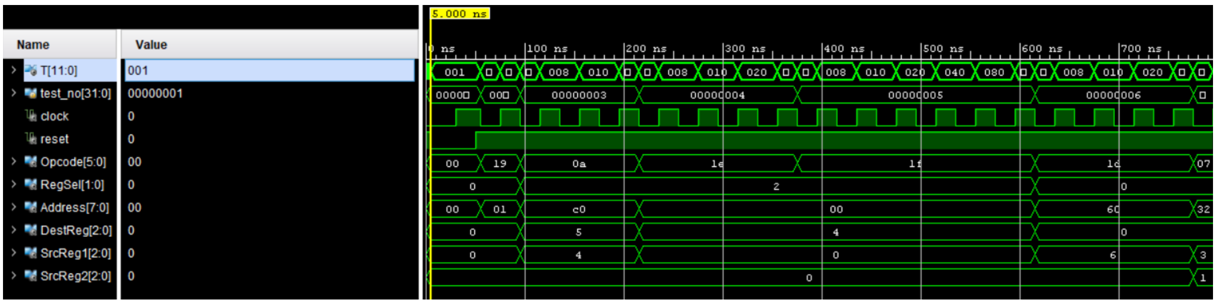

Figure 18: Simulation Waveform

3.1.2 Example Instruction Waveforms

Below are closer looks at some of the instructions executed during simulation.

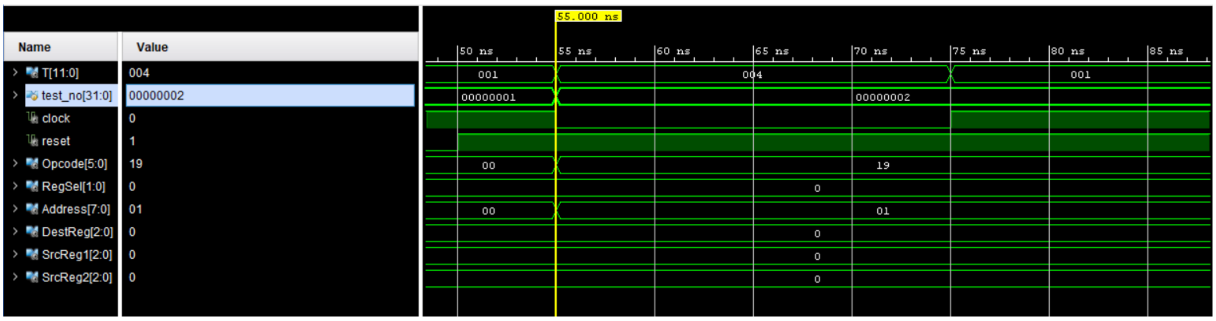

Figure 19: MOVL Instruction Waveform

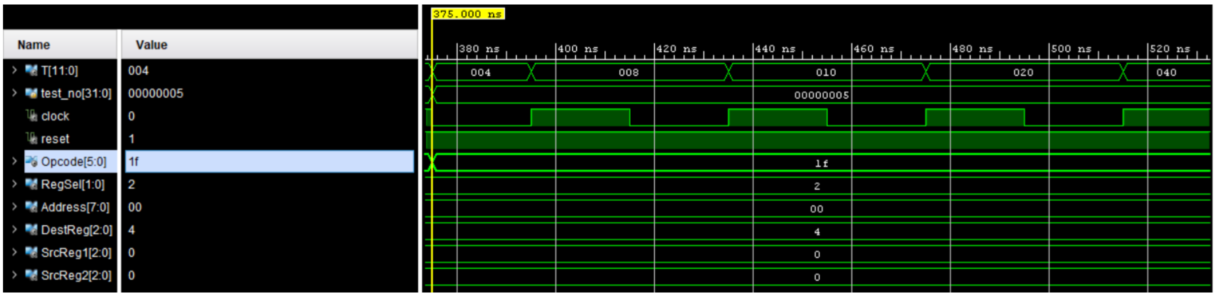

Figure 20: LDAH Instruction Waveform

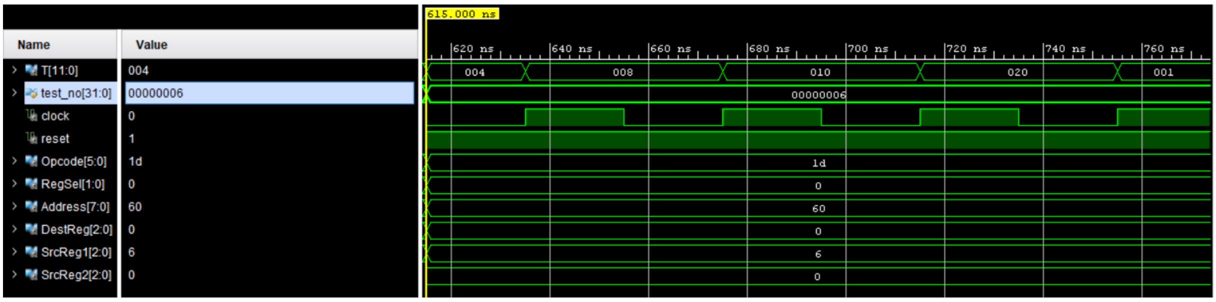

Figure 21: STAR Instruction Waveform

3.2 CPUSystemSimulation Factorial

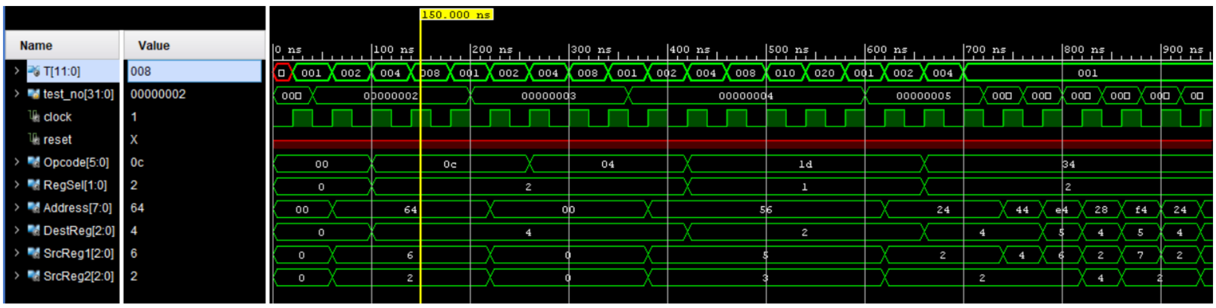

Figure 22: Factorial Simulation Waveform

4 DISCUSSION

In this project, we developed a hardwired CPU control unit that utilizes the ALU we created in the earlier project this term. We started by analogizing the instruction table to figure out control signals needed to perform the operations and figure out the timings necessary.

We then started to implement the modules necessary for the control operations of the system. We continued with the implementation of fetch and decode cycle. Lastly, we implemented the micro-operations of opcode cases for each instruction.

Finally, we tested our design and implementation using the simulations and tests. And tried to debug the failed cases to complete our project.

5 CONCLUSION

We have faced difficulties in debugging without the appropriate debugging tools available, or we were not aware of such tools. And we have developed a deeper appreciation for debugging tools in software developement as a result. We were a bit confused by the endiannes of memory and registers which caused struggles in the developement phase. But we have developed a deeper understanding of CPU design and implementation by completing this project.

REFERENCES