ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 222E
COMPUTER ORGANIZATION
PROJECT 2 REPORT

CRN : 21334

LECTURER : Doç. Dr. Gökhan İnce

GROUP MEMBERS:

150230739 : MUHAMMED ALİ AÇIKGÖZ

150220032 : YAVUZ SELİM KARA

SPRING 2024

Contents

1 INTRODUCTION [10 points]

The field of computer organization encompasses the study of how computer systems are structured and operate at the hardware level. One fundamental aspect of computer organization is the design and implementation of various components such as registers, register files, arithmetic logic units (ALUs), memory systems, and control units. This project focuses on the design and simulation of a hardwired control unit, building on the components designed in Project 1, using Verilog HDL, a hardware description language commonly used for digital circuit design. The project is divided into three main parts:

1. Designing a main structure for the hardwired control unit system of the structure that have been designed in Project 1.
2. Implementing Fetch-Decode cycles for the given instruction format.
3. Implementing Execute cycle for the given Opcodes.

Each part involves designing and simulating a specific hardware module named "CPUSystem" according to provided specifications. The goal is to use components designed in Project 1 and create a complete hardwired control unit. [1] [2]

1.1 TASK DISTRIBUTION

Throughout the duration of our project, we collaborated effectively, utilizing Discord as our primary communication platform. Our meetings were structured around finding suitable times for both of us, accommodating our schedules to ensure productive work sessions. Over the course of the project, we held approximately 5 to 6 meetings, each lasting between 4 to 5 hours. During these sessions, we handled the problems together and helped each other to understand the missing parts.

Each member had contributed to all parts, including designing the hardwired control unit and implementing the Fetch-Decode-Execute (FDE) cycles. We implemented opcodes for the execution cycle separately in order to make comparisons and apply the version that is logically true and optimized for our system. Furthermore, responsibilities encompassed simulation testing and ensuring compatibility with the overall system architecture. By collectively contributing to the design, simulation, and testing phases, we ensured the successful completion of all tasks, achieving our project objectives effectively.

2 MATERIALS AND METHODS [40 points]

To accomplish the objectives outlined in the project instructions, we utilized Verilog HDL to design and simulate our system. The design process involved the following steps:

1. Understanding the project requirements and specifications outlined in the provided instructions.

2. Implementing the Verilog code for the "CPUSystem" module and FDE cycles, adhering to the defined functionality.

3. Integrating the individual opcodes to create the complete system according to the architecture provided in the project instructions.

4. Simulating the system using the provided simulation file to verify correct functionality under various opcodes.

5. Debugging and refining the design based on simulation results to ensure proper operation.

6. Throughout the design process, we paid close attention to efficient sequence utilization, modularity with functions, and adherence to the given specifications. Additionally, we documented our design decisions and implementation details to provide clarity and facilitate understanding.

2.1 MAIN STRUCTURE DESIGN

Firstly, a module named "CPUSystem" for the main structure is developed with respect to the Project 1 architecture which is given below in Figure 1. (1)

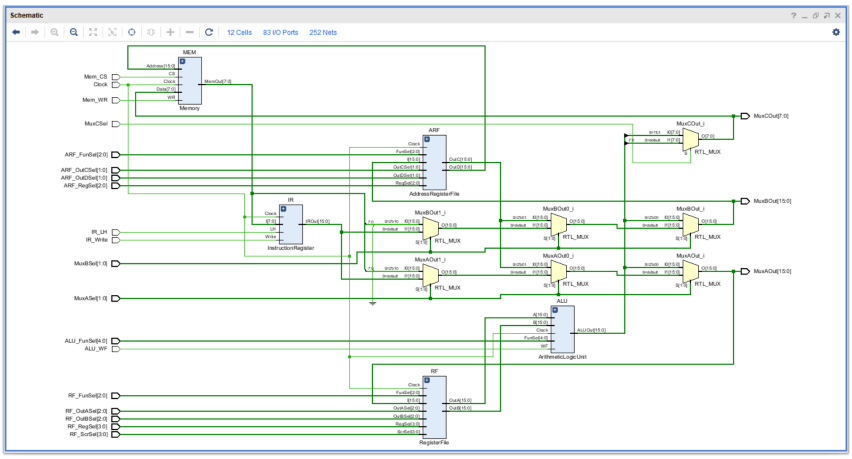
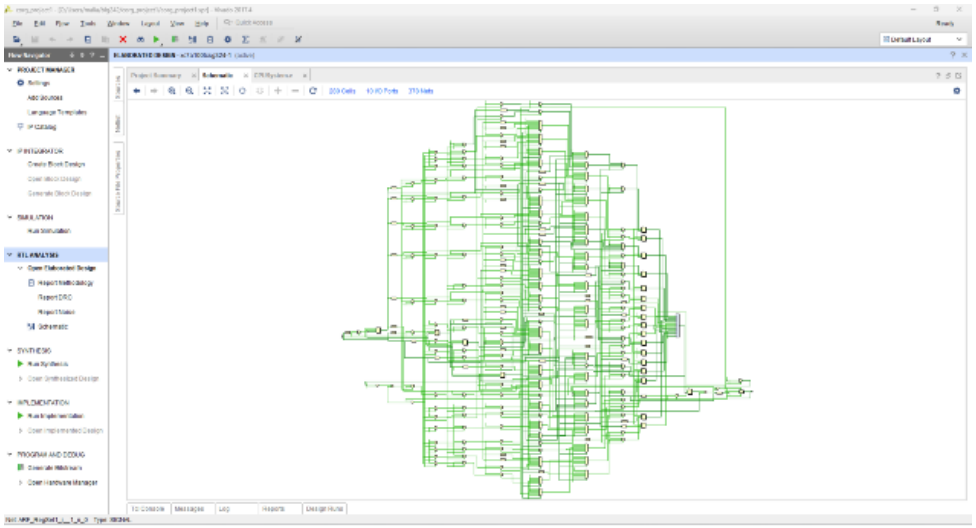


Figure 1: ALUSystem RTL Architecture from Project 1.

By using this structure, a complex module for the hardwired control unit was designed and implemented in the Verilog HDL given in Figure 2 to continue with the FDE cycles. A sequence counter logic is designed by using "localparam" and "case" switches of Verilog

to control clock cycles. The sequence counter used one-hot encoding for the time states given in Figure 3. It updated the counter with each clock cycle. (2) (3)



```
// Registers for Instruction
reg [5:0] Opcode;
reg [7:0] Address;
reg [2:0] DSTREG;
reg [2:0] SREG1;
reg [2:0] SREG2;
reg [1:0] RSel;
reg S;

// Registers for ALUSystem
reg [2:0] RF_OutASel, RF_OutBSel, RF_FunSel;
reg [3:0] RF_RegSel, RF_ScrSel;
reg [4:0] ALU_FunSel;
reg ALU_WF;
reg [1:0] ARF_OutCSel, ARF_OutDSel;
reg [2:0] ARF_FunSel, ARF_RegSel;
reg IR_LH, IR_Write, Mem_WR, Mem_CS;
reg [1:0] MuxASel, MuxBSel;
reg MuxCSel;

wire [15:0] IROut;

ArithmeticLogicUnitSystem _ALUSystem (
    .RF_OutASel(RF_OutASel), .RF_OutBSel(RF_OutBSel), .RF_FunSel(RF_FunSel),
    .RF_RegSel(RF_RegSel), .RF_ScrSel(RF_ScrSel),
    .ALU_FunSel(ALU_FunSel),
    .ALU_WF(ALU_WF),
    .ARF_OutCSel(ARF_OutCSel), .ARF_OutDSel(ARF_OutDSel),
    .ARF_FunSel(ARF_FunSel), .ARF_RegSel(ARF_RegSel),
    .IR_LH(IR_LH), .IR_Write(IR_Write), .Mem_WR(Mem_WR), .Mem_CS(Mem_CS),
    .MuxASel(MuxASel), .MuxBSel(MuxBSel),
    .MuxCSel(MuxCSel),
    .Clock(Clock),
    .IROut(IROut)
    );
```

Figure 2: CPUSystem RTL Architecture and Main Structure of the Design.

```
// State encoding: One-hot encoding for an 8-state counter
localparam T0 = 8'b0000_0001;
localparam T1 = 8'b0000_0010;
localparam T2 = 8'b0000_0100;
localparam T3 = 8'b0000_1000;
localparam T4 = 8'b0001_0000;
localparam T5 = 8'b0010_0000;
localparam T6 = 8'b0100_0000;
localparam T7 = 8'b1000_0000;

// Sequence Counter
always @(posedge Clock) begin
    if(T == T0) begin
        T <= T1;
    end else if (T == T1) begin
        T <= T2;
    end else if (T == T2) begin
        T <= T3;
    end else if (T == T3) begin
        T <= T4;
    end else if (T == T4) begin
        T <= T5;
    end else if (T == T5) begin
        T <= T6;
    end else if (T == T6) begin
        T <= T7;
    end else if (T == T7) begin
        T <= T0;
    end
end
```

Figure 3: Sequence Counter Design with One-Hot Encoding

2.2 FETCH-DECODE CYCLES DESIGN

Secondly, Fetch and Decode cycles implemented for the given instruction format. The instruction format was demonstrated in Figure 4. There were two types of instructions in the architecture. (4) [2]

There are 2 types of instructions as described below.

1- Instructions with address reference have the format shown in Figure 1.

- The **OPCODE** is a **6-bit field.** (Table 1 is given for opcode definition.)
- The **RSEL** is a **2-bit field.** (Table 2 is given for register selection.)
- The **ADDRESS** is an **8-bit field.**

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

Figure 1: Instructions with an address reference.

2- Instructions without an address reference have the format shown in Figure 2.

- The **OPCODE** is a **6-bit field.** (Table 1 is given for opcode definition.)
- The **S** is a **1-bit field** that specifies whether the flags will change or not.
- The **DSTREG** is a **3-bit field** that specifies the destination register. (Table 3 is given.)
- The **SREG1** is a **3-bit field** that specifies the first source register. (Table 3 is given.)
- The **SREG2** is a **3-bit field** that specifies the second source register. (Table 3 is given.)

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

Figure 2: Instructions without an address reference.

Figure 4: Given Instruction Formats of Two Types

By using the Sequence Counter which was demonstrated in Figure 3 and appropriate

control signals for the components from Project 1 which was demonstrated in Figure 1, a ready-to-execute sequence implemented in T0, T1, and T2 states. At the state T2, instruction is fetched for both types of instructions and used the appropriate registers of the system when it is needed. (3) (1)

2.3 EXECUTE DESIGN

Finally, starting from T2, Execute stage is designed to perform the operations specified by the opcodes given in Figure 5. (5) [2]

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
101	R2
110	R3
111	R4

Figure 5: Given Register Selection Logic

This stage utilizes the control signals and data paths established in the previous cycles (Fetch and Decode) to execute instructions according to their specific functions. The modules from Project 1 which was demonstrated in Figure 1 are excessively used. For the register selection logic, six different functions are designed to determine which register will be used for register for instruction type-1 and for the source registers and the destination register for instruction type-2. The functions are demonstrated in Figure 6, Figure 7, and Figure 8. (1) (6) (7) (8)

```
// Function for appropriate register selection for ARF to be a Source
  function [1:0] ARF_SourceSel;
    input [2:0] in;
    begin
        ARF_SourceSel = in[1:0];
    end
  endfunction

  // Function for appropriate register selection for ARF to be Destination
    function [2:0] ARF_DestSel;
      input [2:0] in;
      begin
          case(in[1:0])
              2'b00: begin
                  ARF_DestSel = 3'b011;
              end
              2'b01: begin
                  ARF_DestSel = 3'b011;
              end
              2'b10: begin
                  ARF_DestSel = 3'b101;
              end
              2'b11: begin
                  ARF_DestSel = 3'b110;
              end
          endcase
      end
    endfunction
```

Figure 6: Functions for ARF to Implement Register Selection Logic

```
// Function for appropriate register selection for RF to be a Source
    function [2:0] RF_SourceSel;
      input [2:0] in;
      begin
          case(in[1:0])
              2'b00: begin
                  RF_SourceSel = 3'b000;
              end
              2'b01: begin
                  RF_SourceSel = 3'b001;
              end
              2'b10: begin
                  RF_SourceSel = 3'b010;
              end
              2'b11: begin
                  RF_SourceSel = 3'b011;
              end
          endcase
      end
    endfunction

// Function for appropriate register selection for RF to be Destination
    function [3:0] RF_DestSel;
      input [2:0] in;
      begin
          case(in[1:0])
              2'b00: begin
                  RF_DestSel = 4'b0111;
              end
              2'b01: begin
                  RF_DestSel = 4'b1011;
              end
              2'b10: begin
                  RF_DestSel = 4'b1101;
              end
              2'b11: begin
                  RF_DestSel = 4'b1110;
              end
          endcase
      end
    endfunction
```

Figure 7: Functions for RF to Implement Register Selection Logic

```
    function [3:0] RF_RSel;
      input [1:0] in;
      begin
          case(in)
              2'b00: begin
                  RF_RSel = 4'b0111;
              end
              2'b01: begin
                  RF_RSel = 4'b1011;
              end
              2'b10: begin
                  RF_RSel = 4'b1101;
              end
              2'b11: begin
                  RF_RSel = 4'b1110;
              end
          endcase
      end
    endfunction

    function [2:0] RF_OutASel_RSel;
      input [1:0] in;
      begin
          case(in)
              2'b00: begin
                  RF_OutASel_RSel = 3'b000;
              end
              2'b01: begin
                  RF_OutASel_RSel = 3'b001;
              end
              2'b10: begin
                  RF_OutASel_RSel = 3'b010;
              end
              2'b11: begin
                  RF_OutASel_RSel = 3'b011;
              end
          endcase
      end
    endfunction
```

Figure 8: Functions for RSel to Implement Register Selection Logic

At the end of each opcode, a reset sequence counter logic is implemented in the next extra clock cycle. Also, reset signals for each used register is set active to prevent any miscalculation. The following details the implementation of the execution logic for each opcode:

- **0x00 - BRA:** The Program Counter (PC) is updated to PC + VALUE, facilitating unconditional branching.
  BRA was implemented in two clock cycles and an extra clock cycle to reset the sequence counter. In the first cycle, Value(IROut) loaded to S1 and in the second cycle, PC loaded to S2 in order to make the needed addition operation in ALU. In the end, the result of ALU (ALUOut) loaded to PC.

- **0x01 - BNE:** If the zero flag (Z) is 0, the PC is updated to PC + VALUE, enabling branching if the previous result was non-zero.
  BNE was implemented in two clock cyles and an extra clock cycle to reset the sequence counter. It had the same operation with BRA with additional ALU Z flag check.

- **0x02 - BEQ:** If the zero flag (Z) is 1, the PC is updated to PC + VALUE, allowing branching if the previous result was zero.
  BEQ had the same operation with BNE but the flag condition is complemented.

- **0x03 - POP:** The Stack Pointer (SP) is incremented by 1, and the value at the memory location pointed to by SP is loaded into the specified register (Rx).
  POP was implemented in three clock cycles and an extra fourth cycle for resetting the sequence counter. In the first cycle, we incremented SP with sending appropriate control signals to Address Register File (ARF) from Project 1. In the second cycle, we loaded SP as the address to the Memory and read the data from Memory, incremented the SP again. Since MemOut from Project 1 is 8-bit, we needed to read the whole memory in two clock cycles.

- **0x04 - PSH:** The value in the specified register (Rx) is stored at the memory location pointed to by SP, and then SP is decremented by 1.
  PSH was implemented with the same logic of 8-bit MemOut but one clock cycle less. Instead of reading, we used a logic for a writing operation.

- **0x05 - INC:** The destination register (DSTREG) is incremented by 1.
  INC was implemented in three clock cycles and an extra fourth cycle for resetting the sequence counter. It was used one source register. In the first cycle, we loaded Source1 to S1 by using our functions which determine the needed source register. In the second clock cycle, we incremented S1 by using its increment function and in the last cycle, we loaded the incremented S1 to the destination register by using our register selection functions demonstrated in Figure 6 and Figure 7.

- **0x06 - DEC:** The destination register (DSTREG) is decremented by 1.

DEC was implemented in the same logic with INC but with a decrement function in S1.

- **0x07 - LSL:** The value in the source register (SREG1) is left-shifted, and the result is stored in the destination register (DSTREG).
  LSL was implemented in the same logic with INC but instead of using the function in Register File (RF), using ALU for logic shift left operation so it implemented in two clock cycles and extra cycle to reset the sequence counter.

- **0x08 - LSR:** The value in the source register (SREG1) is right-shifted logically, and the result is stored in the destination register (DSTREG).
  LSR was implemented in the same logic with LSL but with logic shift right operation in ALU.

- **0x09 - ASR:** The value in the source register (SREG1) is right-shifted arithmetically, and the result is stored in the destination register (DSTREG).
  ASR was implemented in the same logic with LSL but with arithmetic shift right operation in ALU.

- **0x0A - CSL:** The value in the source register (SREG1) is circularly shifted left, and the result is stored in the destination register (DSTREG).
  CSL was implemented in the same logic with LSL but with circulat shift left operation in ALU.

- **0x0B - CSR:** The value in the source register (SREG1) is circularly shifted right, and the result is stored in the destination register (DSTREG).
  CSL was implemented in the same logic with LSL but with circulat shift right operation in ALU.

- **0x0C - AND:** A bitwise AND operation is performed between the values in SREG1 and SREG2, and the result is stored in DSTREG.
  AND was implemented in three clock cycles and an extra fourth cycle for resetting the sequence counter. It was used two source registers. In the first cycle, we loaded Source1 to S1 by using our functions which determine the needed source register and in the second clock cycle, we loaded Source2 to S2 by using our functions which determine the needed source register. In the last cycle, we loaded the S1 and S2 to ALU and select ALU function to ADD operation. In the same cycle, we loaded the ALUOut to the destination register by using our register selection functions demonstrated in Figure 6 and Figure 7.

- **0x0D - ORR:** A bitwise OR operation is performed between the values in SREG1 and SREG2, and the result is stored in DSTREG.
  ORR was implemented in the same logic with AND but with OR operation in ALU.

- **0x0E - NOT:** A bitwise NOT operation is performed on the value in SREG1, and the result is stored in DSTREG.
  NOT was implemented in the same logic with LSL but with complement operation in ALU.

- **0x0F - XOR:** A bitwise XOR operation is performed between the values in SREG1 and SREG2, and the result is stored in DSTREG.
  XOR was implemented in the same logic with AND but with XOR operation in ALU.

- **0x10 - NAND:** A bitwise NAND operation is performed between the values in SREG1 and SREG2, and the result is stored in DSTREG.
  NAND was implemented in the same logic with AND but with NAND operation in ALU.

- **0x11 - MOVH:** The Immediate value which is address is loaded into the upper 8 bits of DSTREG.
  MOVH was implemented in one clock cycle as expected and an extra clock cycle to reset the sequence counter. In the cycle, Immediate (IR[7:]) was loaded to destination register's 8 MSB which was selected with our function demonstrated in Figure 8.

- **0x12 - LDR:** The value at the memory location addressed by the Address Register (AR) is loaded into the specified register (Rx).
  LDR was implemented in two clock cycles and an extra clock cycle to reset the sequence counter. In the first cycle, AR is loaded to Memory for reading operation and 8-bit MemOut logic that have been explained before in other opcodes used. In the second cycle, data read from Memory to register which was chosen with our function demonstrated in Figure 8.

- **0x13 - STR:** The value in the specified register (Rx) is stored at the memory location addressed by AR.
  STR was implemented in the same logic with LDR but using write logic for Memory instead of read.

- **0x14 - MOVL:** The immediate value is loaded into the lower 8 bits of DSTREG.
  MOVL was implemented in the same logic with MOVH but write in 8 LSB.

- **0x15 - ADD:** The values in SREG1 and SREG2 are added, and the result is stored in DSTREG.
  ADD was implemented in the same logic with AND but with addition operation in ALU.

- **0x16 - ADC:** The values in SREG1 and SREG2 are added along with the carry bit, and the result is stored in DSTREG.
  ADC was implemented in the same logic with AND but with addition with carry operation in ALU.

- **0x17 - SUB:** The value in SREG2 is subtracted from the value in SREG1, and the result is stored in DSTREG.
  SUB was implemented in the same logic with AND but with substraction operation in ALU.

- **0x18 - MOVS:** The value in SREG1 is moved to DSTREG, with flags updated based on the result. Logic is same with NOT with flags updated but without the complement operation.

- **0x19 - ADDS:** The values in SREG1 and SREG2 are added, with flags updated based on the result, and the result is stored in DSTREG. Logic is same with ADD but with flags updated.

- **0x1A - SUBS:** The value in SREG2 is subtracted from the value in SREG1, with flags updated based on the result, and the result is stored in DSTREG. Logic is same with SUB but with flags updated.

- **0x1B - ANDS:** A bitwise AND operation is performed between the values in SREG1 and SREG2, with flags updated based on the result, and the result is stored in DSTREG. Logic is same with AND but with flags updated.

- **0x1C - ORRS:** A bitwise OR operation is performed between the values in SREG1 and SREG2, with flags updated based on the result, and the result is stored in DSTREG. Logic is same with ORR but with flags updated.

- **0x1D - XORS:** A bitwise XOR operation is performed between the values in SREG1 and SREG2, with flags updated based on the result, and the result is stored in DSTREG. Logic is same with XOR but with flags updated.

- **0x1E - BX:** The current PC is stored at the memory location pointed to by SP, and the PC is updated with the value in the specified register (Rx).

BX was implemented in three clock cycles and an extra clock cycle to reset the sequence counter. In the first cycle, the current program counter (PC) is stored at the address pointed to by the stack pointer (SP), and the stack pointer is decremented. The address bus is set to the stack pointer's value, and the PC is loaded into a temporary register. The multiplexer selects the PC value, and the register file is set to write the PC to memory. Memory write is enabled, chip select is active, and the stack pointer is decremented. In the second cycle, the continuation of storing the PC into memory is executed. The address bus remains set to the stack pointer's value, and the PC is still in the temporary register. The multiplexer selection remains unchanged, and the register file continues to load the PC into memory. However, the multiplexer output selection changes to handle the lower byte. Memory write and chip select remain active, and the stack pointer decrement continues as before. In the third cycle, the program counter is updated to the value of the register specified by Rx. The register file output selection is set to the appropriate register, and the ALU passes this register value. The multiplexer is set to pass this value directly, and the address register file is set to load the new program counter value.

- **0x1F - BL:** The PC is updated with the value at the memory location pointed to by SP.

  BL was implemented in three clock cycles and an extra clock cycle to reset sequence counter. In the first clock cycle, the sequence begins with preparing the stack pointer (SP) for the next operation. In the second clock cycle, the system reads the memory location pointed to by SP and starts preparing to load this value into the Program Counter (PC). In the third clock cycle, the higher byte of the memory location pointed to by SP is written to the PC.

- **0x20 - LDRIM:** The immediate value is loaded directly into the specified register (Rx).

  LDRIM was implemented in one clock cycle and an extra clock cycle to reset the sequence counter. In the cycle, Value (defined in Address) is loaded into register which was selected with our function demonstrated in Figure 8.

- **0x21 - STRIM:** The value in the specified register (Rx) is stored at the memory location addressed by AR plus an offset.

  STRIM was implemented in five clock cycles and an extra clock cycle to reset the sequence counter. In the first cycle, the offset from the instruction register (IR) is loaded into the register file's source register S1. The control signals set during this cycle include MuxASel to select the offset, RF_ScrSel to load S1 , and RF_FunSel to extend the load operation. In the second clock cycle, the address calculation takes

place. The address register (AR) output is selected, and the offset previously loaded into S1 is combined with the contents of the AR. The control signals ensure that MuxASel selects the AR, RF_ScrSel loads S2, and RF_FunSel loads S2. The ALU is configured to add S1 and S2, and the result is loaded back into the AR. In the third cycle, the lower byte of the data from the specified register (Rx) is prepared to be written to memory. The AR is used to set the memory address, and the ALU operation passes Rx directly. The memory control signals are set to enable writing. Additionally, the AR is incremented to prepare for the next write operation. In the fourth clock cycle, the higher byte of Rx is written to memory. Similar to the third cycle, the AR is used to set the memory address, and the ALU passes Rx directly. However, this time the control signal for MuxCSel is set to write the most significant byte. The memory write enable and chip select signals remain active. In the fifth clock cycle, the AR value recorded in previous cycles is prepared for the next potential operation. The control signals set include RF_OutASel to select the recorded AR value, ALU_FunSel to pass it through the ALU, and MuxBSel to select the appropriate bus. The AR is then loaded with this value again.

The execution logic ensures that each operation is carried out efficiently, utilizing the sequence counter and control signals to manage the various states and transitions required for proper execution of each instruction. This stage is critical for the functionality of the CPU, as it directly impacts the performance and correctness of the operations performed.

3 RESULTS [15 points]

The results of our project include the design and simulation of each opcode as well as the hardwired control unit system. Opcodes demonstrated correct functionality and responded appropriately to control signals and memory. Simulation results confirmed the proper operation of the designed system under various test scenarios. An example test result for the opcode BRA is given below:

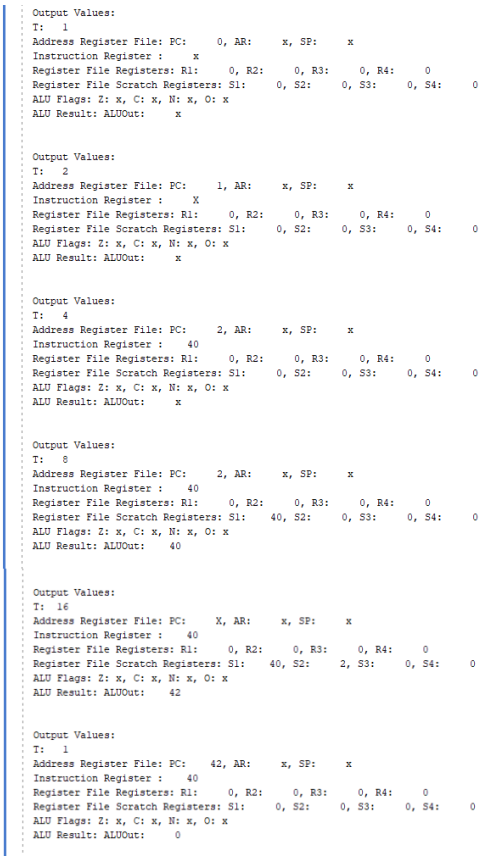


Figure 9: Results for the opcode BRA.

We have tried to pass all the test cases from the simulation files and execute the system successfully.

4 DISCUSSION [25 points]

The completion of this project marks a significant milestone in our understanding and application of computer organization principles. Throughout the development process, several key considerations and challenges emerged, each contributing to the refinement and optimization of our hardware design.

A pivotal decision during the execution logic for each opcode demanded meticulous attention to detail, particularly in managing control signals and data paths. The comprehensive analysis of each operation's requirements enabled us to devise efficient assignments and optimize sequence utilization, contributing to the overall performance and reliability of the system.

One notable challenge encountered during the project was the interpretation and application of ambiguous or incomplete instructions. In such instances, extensive research and consultation were necessary to make informed design decisions and ensure compatibility with project objectives. This iterative process not only enhanced our problem-solving

skills but also underscored the importance of clear and concise specifications in engineering projects.

Simulation testing played a crucial role in validating the functionality and correctness of our designs. Through exhaustive testing scenarios and corner cases, we verified the robustness and reliability of our hardware implementation, mitigating the risk of potential errors or inconsistencies.

Looking ahead, this project serves as a foundation for further exploration and experimentation in computer organization and digital circuit design. The knowledge and insights gained from this experience will undoubtedly inform future endeavors in hardware development and system architecture.

5 CONCLUSION [10 points]

In conclusion, our project demonstrates a design and simulation of a hardwired control unit for a computer organization using Verilog HDL in Vivado environment. By systematically following the provided instructions and applying sound design principles, we were able to create an efficient and reliable basic computer.

Through simulation testing, we validated the functionality of our control unit design and ensured that it meets the specified requirements.

Overall, this project provided valuable hands-on experience in digital circuit design and computer organization principles, enhancing our understanding of hardware implementation in computer systems.

REFERENCES

[1] ITU. Blg 222e project 1.pdf, 2024.

[2] ITU. Blg 222e project 2.pdf, 2024.