ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 222E
COMPUTER ORGANIZATION
PROJECT 2 REPORT

CRN : 21336

LECTURER : Prof. Dr. Mustafa Ersel Kamaşak

GROUP MEMBERS:

150210006 : Yusuf Yıldız (Group Representative)

150210016 : Şafak Özkan Pala

SPRING 2024

Contents

1 INTRODUCTION [10 points]

In this project, our aim is to design and implement a hardwired control unit for a basic computer architecture using Verilog hardware description language within the Vivado platform. Building upon our previous endeavors in simulating fundamental computer functionalities, this project represents a progression towards developing a more intricate system capable of executing a wider range of instructions.

The architecture we're tasked with implementing encompasses instructions stored in memory in little-endian order. This structure necessitates a careful approach to loading instructions into the instruction register (IR) within two clock cycles. The instructions are divided into two types: those with address references and those without. Each instruction type follows a specific format, detailed in the project specifications.

Our control unit's primary function is to oversee the fetching, decoding, and execution of instructions. It must interpret the opcode, determine the operation to be performed, and orchestrate the flow of data between registers and memory accordingly. The control unit serves as the central nervous system of our computer, orchestrating the synchronized interaction of various components to execute instructions accurately and efficiently.

Through this endeavor, we aim to deepen our understanding of digital systems design, particularly in the realm of control units. By translating theoretical concepts into practical implementations, we strive to reinforce our knowledge of computer architecture while honing our skills in Verilog programming and simulation. Ultimately, we endeavor to create a robust and functional control unit that forms the backbone of a rudimentary yet capable computing system.

While testing our design, we utilized the unit testing method by assigning initial values to the necessary registers for each added operation. After understanding the logic of operation with the example outputs provided, we proceeded to test all remaining operations individually and ensured that we obtained the expected results for each. Finally, we tested the provided sample program to observe the sequential execution of our code.

1.1 Task Distribution

All of the design and implementation phases are handled collaboratively by Yusuf Yıldız, and Şafak Özkan Pala. The additional test cases are added, and simulated by the group as a whole, also, debugging was done as a whole group.

2 MATERIALS AND METHODS [40 points]

In this part, we will introduce our implementation details and the solutions that we produced for every task. The main focuses covered in this part are as follows:

- list of control inputs and corresponding functions for your design
- explanations for each constructed part
- task distribution of each group member(given in the first section)
- our understanding of program workflow
- causes, results, and other options for each decision during implementation

2.1 Fetching and Decoding Phase

At the initial stage, the computer starts with PC 0 and memory read mode enabled. All registers are reset, and ALU flags are disabled. The ALU is configured to only output A. Finite state machine logic is used for instruction fetching and decoding. According to this logic, the IR is divided into two parts, and when the last part is loaded, the program enters the execution phase. It stays in that phase until a reset command is received. Each time a reset command is received, the code starts loading instructions again. Decoding occurs in the star block, and based on two different possibilities, decoding is performed. Subsequently, the instruction is used based on the OPCODE. During the reset process, all registers and memory are disabled to prevent conflicts.

2.2 Functional(Task) Programming and Benefits

We utilized tasks to improve coding practices and work more systematically while writing code. These tasks helped us optimize the use of register codes we frequently used and reduce repetition in our code. For example, as seen in the image below, you can see two of othe tasks we wrote to select a register. One is to select a register from RF or ARF and then perform an operation on that register. Other is for selection of scratch regisers. This process contributed significantly to reducing the length of the code since these tasks occurred in almost all operations.

```
task SetREGSel;
    input [2:0] REGSEL;
    input [2:0] FunSel;
    case (REGSEL)
        3'b000: begin // PC
            ARF_RegSel = 3'b011;
            ARF_FunSel = FunSel;
        end
        3'b001: begin // PC
            ARF_RegSel = 3'b011;
            ARF_FunSel = FunSel;
        end
        3'b010: begin // SP
            ARF_RegSel = 3'b110;
            ARF_FunSel = FunSel;
        end
        3'b011: begin // AR
            ARF_RegSel = 3'b101;
            ARF_FunSel = FunSel;
        end
        3'b100: begin  // R1
            RF_RegSel = 4'b0111;
            RF_FunSel = FunSel;
        end
        3'b101: begin  // R2
            RF_RegSel = 4'b1011;
            RF_FunSel = FunSel;
        end
        3'b110: begin  // R3
            RF_RegSel = 4'b1101;
            RF_FunSel = FunSel;
        end
        3'b111: begin  // R4
            RF_RegSel = 4'b1110;
            RF_FunSel = FunSel;
        end
    endcase
endtask
```

```
task SetScratchSel;
    input [1:0] SSEL_addr;
    input [2:0] FunSel;

    begin
    case (SSEL_addr)
        2'b00: begin // S1
            RF_ScrSel = 4'b0111;
            RF_FunSel = FunSel;
         end
        2'b01: begin // S2
            RF_ScrSel = 4'b1011;
            RF_FunSel = FunSel;
        end
        2'b10: begin // S3
            RF_ScrSel = 4'b1101;
            RF_FunSel = FunSel;
        end
        2'b11: begin // S4
            RF_ScrSel = 4'b1110;
            RF_FunSel = FunSel;
        end
    endcase
    end
endtask
```

Figure 1: Tasks

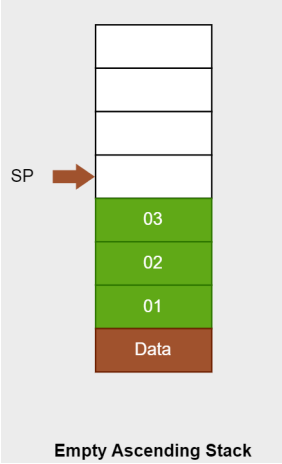
2.3 Grouping Similar Operations in a Case

Most operations in the CPU are based on the same logic. We often manipulate data using specific registers and then transfer it to another designated register. These data manipulations frequently occur within the ALU. Therefore, the operations differ mainly

in the ALU functions. Due to this pattern, when writing operations with similar register usage, we ensure that only the ALU functions differ, while the rest of the operations remain the same. This approach has made our code clearer and shorter. Sometimes, due to the need for different paths for registers to move from source to destination, we overcome this challenge by using if-else statements within common operations.

2.4 Memory Manipulations and ARF Registers

- **AR:** Adress register is a pointer for CPU to access a data from the memory. It will take 8-bit memory and do not increment during this process. To change the pointed memory, AR is must be changed by using register operations of the CPU.
- **SP:** In stack pointer implementation Empty Ascending Stack model is used. You can see the models in this link. Initially SP is pointing the memory address 255 which is the bottom of the memory. And moves upward with every addition to stack. Stack stores 16-bit data therefore it will go upwards by 2 memory cell. CPU system automatically moves SP upwards to point free memory. You can see used model in the below.



- **PC:** Program counter is the pointer of the instructions. It starts initially with 0 and increases with every operations except BNE and BEQ operations which causes for branching and change the value of the PC. After instruction is fetched from the memory it points to the next instruction.

2.5 Handling Operations

1. **Branch and Compare Instructions, BRA, BNE, BEQ (Opcode: 6'h00, 6'h01, 6'h02):**

   These opcodes are used for conditional branching based on the Zero flag (Z).

`T = 4 (0x04):`

- Set the Multiplexer A Select (`MuxASel`) to `2'b11` to select the Instruction Register output (`IROut[7:0]`).
- Enable only Scratch register S1 by setting Scratch Register Select (`SetScratchSel`) to `2'b00` and Register Select (`REGSEL`) for all scratch registers to `3'b111`.
- Set the ALU Function Select (`ALU_FunSel`) to `5'b10100` to perform addition (`ALUOut = A + B`) on the register values.
- Set the Register File Output A Select (`RF_OutASel`) to `3'b100` to select the lower 8-bits of register A as input to the ALU.
- Set the Register File Output B Select (`RF_OutBSel`) to `3'b101` to select the lower 8-bits of register B as input to the ALU.

`T = 8 (0x08):`

- Enable only Scratch register S2 and set its value by setting SetScratchSel to 2'b01 and Register Select for S2 to 3'b010.
- Set the ARF Output Control Select (`ARF_OutCSel`) to `2'b00` to select the Program Counter (PC) output.
- Set the Multiplexer A Select (`MuxASel`) to `2'b01` to select the output of the `ARF_OutCSel` (PC value) as input to the ALU.

`T = 16 (0x10):`

- Set the Multiplexer B Select (`MuxBSel`) to `2'b00` to select the ALU output for the branching operation.
- Enable the PC register by setting the ARF Register Select (`ARF_RegSel`) to `3'b011`.
- Set the ARF Function Select (`ARF_FunSel`) to `3'b010` to load the ALU output into the PC.
- Reset ResetAgain (instruction execution continues) to allow the branch to take effect if the condition is met.

2. **POP Instruction (Opcode: 6'h03)**

   This opcode is used to pop a value from the stack and store it in a register.

   `T = 4 (0x04):`

- Set the Register Select (REGSEL) for the Stack Pointer (SP) to increment its value by 1. This indicates that the SP will point to the next free location on the stack after the pop operation.

`T = 8 (0x08):`

- Set the ARF Output Data Select (ARF_OutDSel) to 2'b11, indicating that the SP is pointing to memory.
- Set the Memory Write (Mem_WR) to 0, signifying a read operation from memory.
- Enable the Memory Chip Select (Mem_CS), allowing communication with memory.
- Set the Multiplexer A Select (MuxASel) to 2'b10, selecting the Memory output to be loaded into the ALU.
- Set the lower 4 bits of the Register Select (REGSEL) for the destination register (based on the RSEL bits in the instruction) to enable loading the lower 4-bits. This means the lower 4-bits of the value stored at the memory location pointed to by SP will be loaded into the lower 4-bits of the destination register.

`T = 16 (0x10):`

- Set the upper 4 bits of the Register Select (REGSEL) for the destination register to enable loading the higher 4-bits. This allows the entire 16-bit value from the stack to be loaded into the destination register.
- Set the ARF Register Select (ARF_RegSel) to 3'b111, disabling the automatic increment of SP again. Since the top element has been popped, the SP doesn't need to be incremented further as the cell is now free.
- Reset ResetAgain (instruction execution continues). This ensures the pipeline keeps processing instructions.

3. **PSH Instruction (Opcode: 6'h04)**

   Opcode to push a value onto the stack.

   `ALU_FunSel:` ALU Function Select is set to 5 bits 10000, indicating that the ALU output is simply a register value.

   `T = 4 (0x04):`

   - Register File Output A Select is set to select the lower 8 bits of the register specified by the instruction.

- Multiplexer Control Select of MuxCSel is set to select the most significant bit (MSB) of the register value for the first push operation.
- Memory Chip Select is enabled to allow communication with memory.
- Memory Write is enabled, indicating a write operation to memory.
- ARF_OutDSel Output Data Select is set to indicate that the Stack Pointer (SP) is currently pointing to memory.
- SetREGSel(3'b010,3'b000), Register Select for the Stack Pointer (SP) is decremented by 1, updating it to point to the next free location on the stack.

`T = 8 (0x08):`

- MuxCSel, Multiplexer Control Select is set to select the least significant bit (LSB) of the register value for the second push operation.
- ResetAgain is set to 0, ensuring the pipeline continues processing instructions.

4. **ALU Instructions**

   In this section, 12 instructions are grouped into case operations because they have similar logic. In the following the distinguishing features of these operations will be explained and then the T cycles grouped according to DSTREGs and SREGs will be given.

   - **AND (Opcode: 6'h0C):**
     - ALU Function Select (ALU_FunSel) is set to 5 bits 10111.
   - **ORR (Opcode: 6'h0D):**
     - ALU Function Select (ALU_FunSel) is set to 5 bits 11000.
   - **XOR (Opcode: 6'h0F):**
     - ALU Function Select (ALU_FunSel) is set to 5 bits 11001.
   - **NAND (Opcode: 6'h10):**
     - ALU Function Select (ALU_FunSel) is set to 5 bits 11010.
   - **ADD (Opcode: 6'h15):**
     - ALU Function Select (ALU_FunSel) is set to 5 bits 10100.
   - **ADD Carry (Opcode: 6'h16):**
     - ALU Function Select (ALU_FunSel) is set to 5 bits 10101.
   - **SUB (Opcode: 6'h17):**

  - ALU Function Select (ALU_FunSel) is set to 5 bits 10110.
- **ADD with Flag (Opcode: 6'h19):**
  - ALU Function Select (ALU_FunSel) is set to 5 bits 10100.
  - ALU_WF is set based on a certain condition.
- **SUB with Flag (Opcode: 6'h1A):**
  - ALU Function Select (ALU_FunSel) is set to 5 bits 10110.
  - ALU_WF is set based on a certain condition.
- **AND with Flag (Opcode: 6'h1B):**
  - ALU Function Select (ALU_FunSel) is set to 5 bits 10111.
  - ALU_WF is set based on a certain condition.
- **OR with Flag (Opcode: 6'h1C):**
  - ALU Function Select (ALU_FunSel) is set to 5 bits 11000.
  - ALU_WF is set based on a certain condition.
- **XOR with Flag (Opcode: 6'h1D):**
  - ALU Function Select (ALU_FunSel) is set to 5 bits 11001.
  - ALU_WF is set based on a certain condition.

If both sources are from ARF:

- `T = 4 (0x04):`
  - ARF Output Control Select (ARF_OutCSel) is set based on the first operand.
  - Multiplexer A Select (MuxASel) is set.
  - Scratch Register Select is set.
  - Register File Output A Select (RF_OutASel) is set.
- `T = 8 (0x08):`
  - ARF Output Control Select (ARF_OutCSel) is set based on the second operand.
  - Multiplexer A Select (MuxASel) is set.
  - Scratch Register Select is set.
  - Register File Output B Select (RF_OutBSel) is set.
- `T = 16 (0x10):`

- Scratch Registers are disabled.
- Multiplexer B Select (MuxBSel) and Multiplexer A Select (MuxASel) are set.
- Register Select for Destination Register (SetREGSel) is set.
- ResetAgain is set to 0.

Both Sources from Register File

- `T = 4 (0x04):`
  - Register File Output A Select (RF_OutASel) and Register File Output B Select (RF_OutBSel) are set according to specific register addresses.
  - Multiplexer B Select (MuxBSel) and Multiplexer A Select (MuxASel) are configured.
  - Destination Register Selection (SetREGSel) is adjusted based on specific register addresses.
  - ResetAgain is set to 0.

One Operand from RF, One Operand from ARF

- `T = 4 (0x04):`
  - If the first operand is from ARF:
    - ARF Output Control Select (ARF_OutCSel) is determined based on the second operand's register address.
    - Multiplexer A Select (MuxASel) is configured.
    - Scratch Register Selection (SetScratchSel) is adjusted.
    - Register File Output A Select (RF_OutASel) is set based on the first operand's register address.
    - Register File Output B Select (RF_OutBSel) is set based on the second operand's register address.
  - If the first operand is from RF:
    - ARF Output Control Select (ARF_OutCSel) is determined based on the first operand's register address.
    - Multiplexer A Select (MuxASel) is configured.
    - Scratch Register Selection (SetScratchSel) is adjusted.

        * Register File Output B Select (RF_OutBSel) is set based on the first operand's register address.
        * Register File Output A Select (RF_OutASel) is set based on the second operand's register address.

- `T = 8 (0x08):`
  - Multiplexer B Select (MuxBSel) and Multiplexer A Select (MuxASel) are configured.
  - Destination Register Selection (SetREGSel) is adjusted.
  - ResetAgain is set to 0.

5. **INC/DEC Instructions (Opcode: 6'h05, 6'h06):**

- `T = 4 (0x04):`
  - If the primary source is from ARF:
    * ARF Output Control Select (ARF_OutCSel) is determined based on the primary source's register address.
    * Multiplexer B Select (MuxBSel) and Multiplexer A Select (MuxASel) are configured.
  - If the primary source is from RF:
    * Register File Output A Select (RF_OutASel) is determined based on the primary source's register address.
    * ALU Function Select (ALU_FunSel) is set to assign S1 value to 16-bit A.
    * Multiplexer B Select (MuxBSel) and Multiplexer A Select (MuxASel) are configured.
  - Destination Register Selection (SetREGSel) is adjusted based on the destination register address.

- `T = 8 (0x08):`
  - Depending on the OPCODE (for 6'h05: DSTREG = SREG1 + 1, for 6'h06: DSTREG = SREG1 - 1), the destination register selection is made.
  - ResetAgain is set to 0.

6. **Shift and Bitwise Operations (Opcodes: 6'h07, 6'h08, 6'h09, 6'h0A, 6'h0B, 6'h0E, 6'h18):**

   In this section, as above for ALU operations, 7 operations are combined in a case structure since they have similar logic. What distinguishes these operations from the

others and from ALU is that they are realized with only 1 source and 1 destination register. These operations are differentiated according to ALU_FunSel and ALU_WF changes.

- The ALU operation is determined based on the opcode:
  - Opcode 6'h07 corresponds to logical shift left (LSL).
  - Opcode 6'h08 corresponds to logical shift right (LSR).
  - Opcode 6'h09 corresponds to arithmetic shift right (ASR).
  - Opcode 6'h0A corresponds to circular shift left (CSL).
  - Opcode 6'h0B corresponds to circular shift right (CSR).
  - Opcode 6'h0E corresponds to bitwise NOT operation.
  - Opcode 6'h18 corresponds to ALU operation specified by IROut[9] with ALU_WF set.

If the source is from ARF

- `T = 4 (0x04):`
  - ARF Output Control Select (ARF_OutCSel) is determined based on the source register address.
  - Multiplexer A Select (MuxASel) is set to 2'b01, and Scratch Register Selection (SetScratchSel) is configured accordingly.
  - Register File Output A Select (RF_OutASel) is determined based on the source register address.
- `T = 8 (0x08):`
  - Multiplexer B Select (MuxBSel) and Multiplexer A Select (MuxASel) are both set to 2'b00.
  - Destination Register Selection (SetREGSel) is configured based on the destination register address.
  - ResetAgain is set to 0, and all scratch registers are disabled.

If the source is from RF

  - `T = 4 (0x04):`
    - Register File Output A Select (RF_OutASel) is determined based on the source register address.
    - Multiplexer B Select (MuxBSel) and Multiplexer A Select (MuxASel) are both set to 2'b00.

- Destination Register Selection (SetREGSel) is configured based on the destination register address.
- ResetAgain is set to 0.

7. **MOVH and MOVL Instructions (Opcodes: 6'h11, 6'h14):**
   - 6'h11 corresponds to the MOVH instruction, where the upper 8 bits of the destination register (DSTREG) are loaded with an immediate 8-bit value.
   - 6'h14 corresponds to the MOVL instruction, where the lower 8 bits of the destination register (DSTREG) are loaded with an immediate 8-bit value.
   - `T = 4 (0x04):`
     - Depending on the opcode, the destination register selection (DREGSEL) is set to select the appropriate register, and the immediate value is written to the upper or lower 8 bits of the register.
     - Multiplexer A Select (MuxASel) is set to 2'b11 to select the immediate value for writing to the destination register.
     - ResetAgain is set to 0 to ensure the pipeline continues processing instructions.

8. **LDR Instruction (Opcode: 6'h12):** LDR is a 16-bit load instruction where the value from memory at the address specified by the address register (AR) is loaded into a register (Rx).
   - `T = 4 (0x04):`
     - ARF Output Data Select (ARF_OutDSel) is set to 2'b10, indicating that the address register (AR) is pointing to memory.
     - Memory Write (Mem_WR) is set to 0 to perform a read operation from memory.
     - Memory Chip Select (Mem_CS) is enabled to allow communication with memory.
     - Multiplexer A Select (MuxASel) is set to 2'b10 to select the memory output for loading into the register.
     - The destination register selection (DREGSEL) is set to select the destination register (Rx).
     - ResetAgain is set to 0 to ensure the pipeline continues processing instructions.

9. **STR Instruction (Opcode: 6'h13):**

   STR is a 16-bit store instruction where the value from a register (Rx) is stored into memory at the address specified by the address register (AR).

   - `T = 4 (0x04):`
     - Register File Output A Select (RF_OutASel) is set to select the contents of register Rx for the ALU input A.
     - ARF Output Data Select (ARF_OutDSel) is set to 2'b10, indicating that the address register (AR) is pointing to memory.
     - Multiplexer Control Select (MuxCSel) is set to 1'b0 to select the LSB (bits [7:0]) of the ALU output.
     - Memory Write (Mem_WR) is enabled (set to 1) to perform a write operation to memory.
     - Memory Chip Select (Mem_CS) is enabled (set to 0) to allow communication with memory.
     - The address register (AR) is incremented by setting its selection to increment.
     - ResetAgain is set to 0 to ensure the pipeline continues processing instructions.

10. **BX Instruction (Opcode: 6'h1E):**

    BX is an instruction that loads the program counter (PC) with the value from a specified register (Rx), and simultaneously stores the current PC value into memory at the address pointed to by the stack pointer (SP).

    - `T = 4 (0x04):`
      - Register File Output A Select (RF_OutASel) is set to select the S1.
      - ARF Output Data Select (ARF_OutDSel) is set to 2'b11, indicating that the stack pointer (SP) is pointing to memory.
      - Scratch Register Selection (SetScratchSel) is set to enable only S2 and load.
      - ARF Output Control Select (ARF_OutCSel) is set to 2'b00, selecting the program counter (PC) as the output.
      - Multiplexer Control Select (MuxASel) is set to 2'b01 to select the output of the ARF.
    - `T = 8 (0x08):`

- Memory Chip Select (Mem_CS) is enabled (set to 0) to allow communication with memory.
- Memory Write (Mem_WR) is enabled (set to 1) to perform a write operation to memory.
- Multiplexer Control Select (MuxCSel) is set to 1'b1 to select the MSB of the register value.
- The stack pointer (SP) is decremented by setting its selection to decrement.

- `T = 16 (0x10):`
  - Multiplexer Control Select (MuxCSel) is set to 1'b0 to select the LSB of the register value.

- `T = 32 (0x20):`
  - Memory Chip Select (Mem_CS) is disabled (set to 1) to halt communication with memory.
  - Register File Output A Select (RF_OutASel) is set to select the contents of register Rx for the ALU input A.
  - Multiplexer Control Select (MuxBSel) is set to 2'b00.
  - The program counter (PC) is loaded with the value from register Rx.
  - ResetAgain is set to 0 to ensure the pipeline continues processing instructions.

11. **BL Instruction (Opcode: 6'h1F):**

    BL is a branch with link instruction that loads the program counter (PC) with the value from memory at the address pointed to by the stack pointer (SP), and simultaneously increments the stack pointer.

    - `T = 4 (0x04):`
      - The stack pointer (SP) is incremented to prepare for reading the address from memory.
    - `T = 8 (0x08):`
      - ARF Output Data Select (ARF_OutDSel) is set to 2'b11, indicating that the stack pointer (SP) is pointing to memory.
      - Memory Write (Mem_WR) is disabled (set to 0) to perform a read operation from memory.
      - Memory Chip Select (Mem_CS) is enabled (set to 0) to allow communication with memory.

- Multiplexer Control Select (MuxBSel) is set to 2'b10 to select the memory output.
- The program counter (PC) is enabled, and only the low part is loaded from memory.

- `T = 16 (0x10):`
  - The stack pointer (SP) is incremented again to prepare for loading the high part of the address from memory.
- `T = 32 (0x20):`
  - The program counter (PC) is enabled, and the high part is loaded from memory.
  - ResetAgain is set to 0 to ensure the pipeline continues processing instructions.

12. **LDRIM Instruction (Opcode: 6'h20):**

    LDRIM is a load immediate instruction that loads the value defined in the ADDRESS bits into the specified register (Rx).

    - `T = 4 (0x04):`
      - The instruction register (IR) value is selected and loaded into the register specified by the 8-bit address low bits.
      - ResetAgain is set to 0 to ensure the pipeline continues processing instructions.

13. **STRIM Instruction (Opcode: 6'h21):**

    STRIM is a store immediate instruction that stores the value in register Rx to the memory location pointed to by AR plus the OFFSET defined in the ADDRESS bits.

    - `T = 4 (0x04):`
      - The ALU performs addition (A + B), where A is the value from the instruction register (IR) and B is the sign-extended immediate OFFSET.
      - S1 is loaded with the OFFSET value first, and then S2 is loaded with AR.
      - ARF_OutCSel selects AR to be used as an operand in the ALU operation.
    - `T = 8 (0x08):`
      - MuxASel selects AR as the input for the ALU operation.
      - The scratch register is loaded with AR to prepare for the next cycle.

- `T = 16 (0x10):`
  - MuxBSel selects the ALU output as the input for AR, and sets it with AR + OFFSET.
  - The scratch register is enabled to ensure the register file (RF) operation is disabled.
- `T = 32 (0x20):`
  - The ALU output is selected as A, with register Rx selected as the operand in A.
  - RF_RegSel is disabled to prevent any further RF operations.
  - ARF_OutDSel is set to point to memory for memory operation.
  - MuxCSel selects the LSB for memory access.
  - Mem_CS is enabled, and Mem_WR is set for writing to memory.
  - AR is incremented.
- `T = 64 (0x40):`
  - MuxCSel selects the MSB of ALUOut for memory access.
  - ResetAgain is set to 0 to continue the instruction execution cycle.

14. **Preserving T Cycle**

    At the end, the T value is set to $T = T + T$ because the example outputs showed that T is not incrementing by 1, instead we are expected to logical shift left the T in each Clock, in other words, we are inrementing it by itself.

3 RESULTS [15 points]

Unit tests were conducted for all operations, and no issues were encountered. Upon comparison with the provided example outputs, it was observed that the outputs were only one clock cycle longer. Since it was stated on the message board that clock cycles were not obligatory, no changes were made due to not interfere to the code structure. Subsequently, a memory was designed based on the given example basic program and tested using it. Following the unit tests, the sequential instructions were successfully completed. An example output for BRA is given below with all of its time sequence.

```
Output Values:
T:   1
Address Register File: PC:     0, AR:     x, SP:     x
Instruction Register :     x
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: x, C: x, N: x, O: x
ALU Result: ALUOut:     0
```

```
Output Values:
T:   2
Address Register File: PC:     1, AR:     x, SP:     x
Instruction Register :     X
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: x, C: x, N: x, O: x
ALU Result: ALUOut:     0
```

```
Output Values:
T:   4
Address Register File: PC:     2, AR:     x, SP:     x
Instruction Register :    40
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: x, C: x, N: x, O: x
ALU Result: ALUOut:     0
```

```
Output Values:
T:   8
Address Register File: PC:     2, AR:     x, SP:     x
Instruction Register :    40
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: x, C: x, N: x, O: x
ALU Result: ALUOut:     0
```

```
Output Values:
T:  16
Address Register File: PC:     2, AR:     x, SP:     x
Instruction Register :    40
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: x, C: x, N: x, O: x
ALU Result: ALUOut:    40
```

```
Output Values:
T:  32
Address Register File: PC:     2, AR:     x, SP:     x
Instruction Register :    40
Register File Registers: R1:     0, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:    40, S2:     2, S3:     0, S4:     0
ALU Flags: Z: x, C: x, N: x, O: x
ALU Result: ALUOut:    42
```

```
Output Values:
T:   1
Address Register File: PC:    42, AR:     x, SP:     x
Instruction Register :    40
Register File Registers: R1:     0, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:    40, S2:     2, S3:     0, S4:     0
ALU Flags: Z: x, C: x, N: x, O: x
ALU Result: ALUOut:    42
```

4 DISCUSSION [25 points]

In this project, we have implemented a control unit in order to connect it to the ALU system we implemented before and let it run the system. Control unit decides which of the switches will be on and off according to the sequence counter in it and gives the switches as output to ALU system. Then, ALU system takes the outputs as its inputs and behaves according to the decisions of control unit. Control unit has the ability to choose which operation in the system is going to happen depending on the given instruction, which is taken from the memory. After figuring out what to do with operations, it was easier to implement. In first two cycles of sequence counter, fetch occurs. Then, one cycle is reserved for the decoding operation. After fetch and decode cycles, execution of the OPCODE starts and after execution ends, sequence counter is reset in the next cycle. Therefore, the system is able to start to get the next instruction from the memory.

Throughout this project, our goal was to design and implement a hardwired control unit using Verilog within the Vivado platform for a basic computer architecture. Our main aim was to create a control unit capable of managing the fetching, decoding, and execution of instructions stored in memory. We encountered challenges, especially in efficiently loading instructions into the instruction register (IR) within the required two clock cycles due to the little-endian order of instructions in memory.

To overcome these challenges, we took a meticulous approach, carefully planning our control unit's implementation to ensure timely and accurate execution of instructions. This involved translating theoretical concepts into practical Verilog code and refining our design iteratively to enhance performance and functionality. Through simulations

and individual operation tests, we validated our implementation and pinpointed areas for improvement.

This project deepened our understanding of digital systems design, particularly in control unit architecture. We learned to interpret opcodes, determine corresponding operations, and manage data flow between registers and memory effectively. Despite the hurdles, our persistence and teamwork led to the development of a robust control unit, serving as the core of a basic computing system.

In summary, this project provided valuable insights into control unit architecture and strengthened our Verilog programming and simulation skills. It was a rewarding learning experience, and we look forward to applying these skills in future projects to further explore and advance in the field of digital systems design.

5 CONCLUSION [10 points]

Throughout this project, we encountered several challenges that tested our understanding and skills in digital systems design. One of the main difficulties we faced was ensuring the efficient loading of instructions into the instruction register (IR) within the required two clock cycles, especially considering the little-endian order of instructions stored in memory. This necessitated careful planning and meticulous attention to detail in the implementation of our control unit.

However, through perseverance and iterative refinement of our design, we were able to overcome these challenges and successfully develop a robust control unit capable of fetching, decoding, and executing instructions accurately. This process deepened our understanding of control unit architecture and Verilog programming, allowing us to apply theoretical concepts in a practical setting.

Moreover, this project provided valuable insights into the complexities of coordinating the interaction between various components within a computer architecture. By orchestrating the flow of data between registers and memory, our control unit served as the central nervous system of our computing system, enabling seamless execution of instructions.

In conclusion, this project was a valuable learning experience that reinforced our knowledge of digital systems design and enhanced our proficiency in Verilog programming. By tackling real-world challenges and testing our design through rigorous unit testing, we gained confidence in our abilities and developed a deeper appreciation for the intricacies of computer architecture. Moving forward, we are excited to apply these newfound skills to future projects and continue our journey of exploration and discovery in the field of digital systems design.