KOREA UNIVERSITY
Department of
Computer Science and Engineering

# COSE222:Computer Architecture Assignment #2

## Final Report

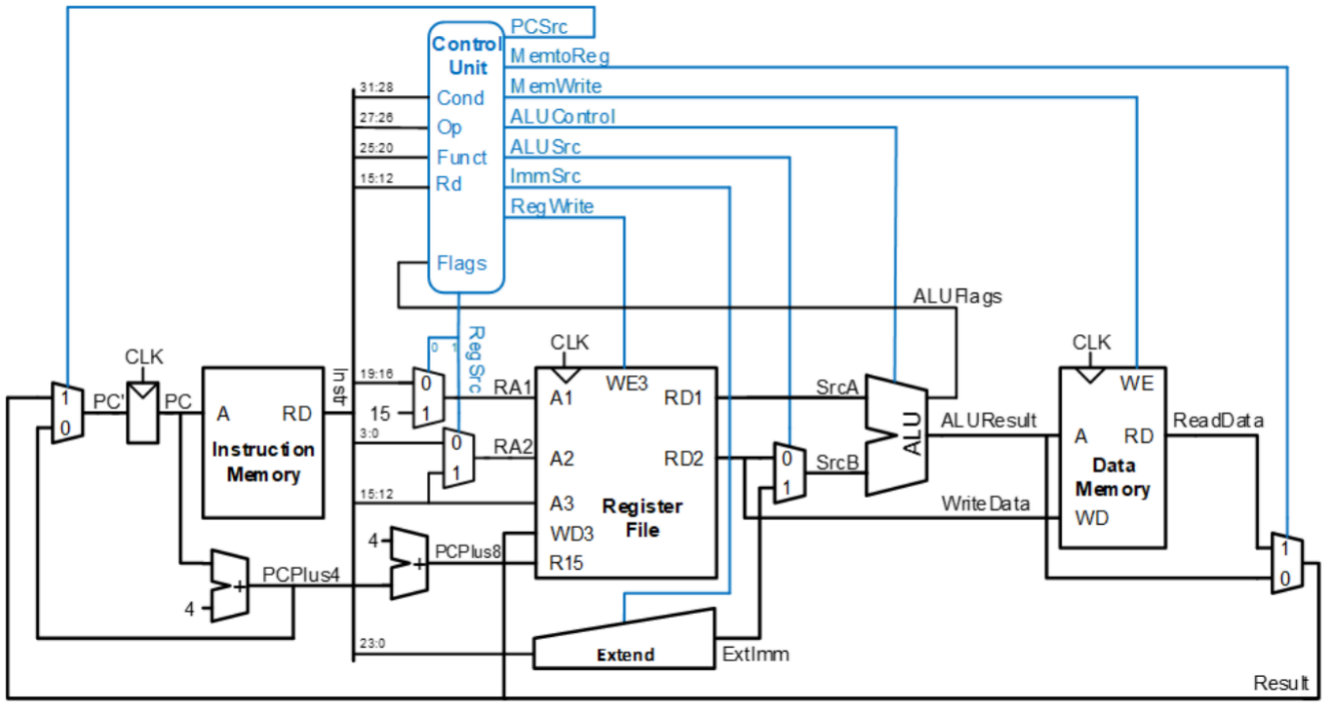
## 32-bit ARM SINGLE CYCLE PROCESSOR

**2021320050** 장오선[2021320050@korea.ac.kr]

Seoul, December 2022

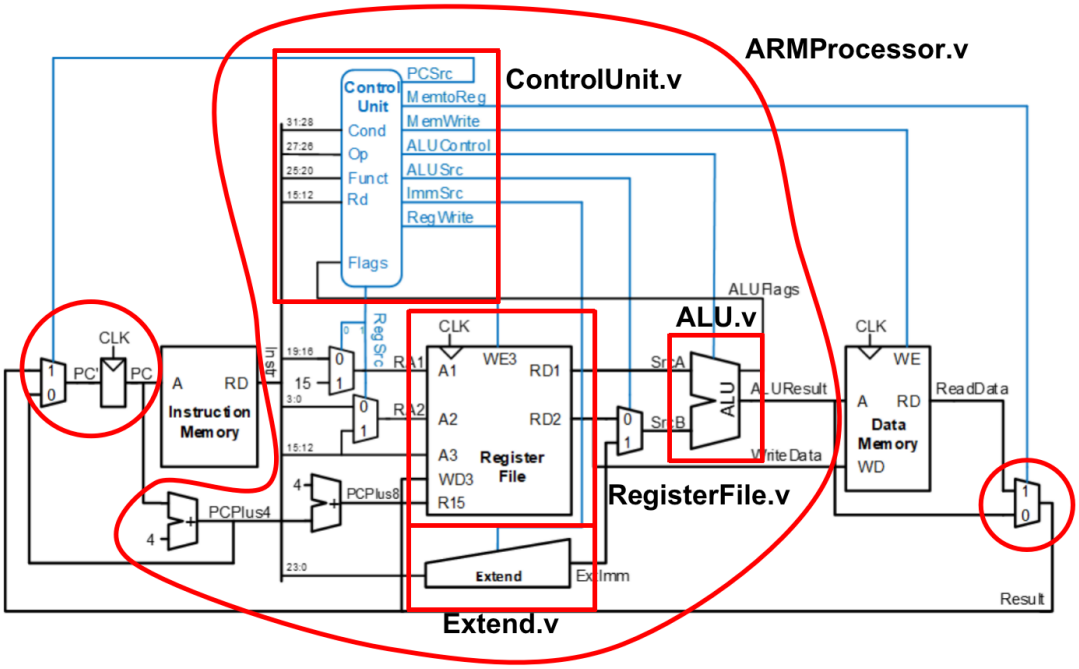
# Introduction

(32-bit ARM single-cycle processor)



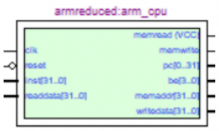

In this report, we will be analyzing the design and implementation of a 32-bit ARM single cycle processor. The ARM processor is a popular choice in the field of embedded systems due to its low power consumption and high performance. The single cycle processor is a simplified version of the ARM processor that executes one instruction per clock cycle. In this report, we will focus on the various modules that make up the single cycle processor and how they work together to execute instructions.

# Module description




1. ARMProcessor

**ARMProcessor module**



- **Input**
  - clk : Clock 90
  - reset : 1 if press “KEY0” button
  - inst[31:0] : current instruction
  - readdata : data from memory
- **Output**
  - pc : instruction address (starting at 0)
  - be[3:0] : $1111_{(2)}$ (default) – don’t change!
  - memaddr[31:0] : data address
  - memread : $1_{(2)}$ (default) – don’t change!
  - memwrite : specify whether memory is to be written
  - writedata[31:0] : write data to memory

The 32-bit ARM single cycle processor is a system designed to execute instructions in a single clock cycle. It is made up of various modules, including the ControlUnit, Decoder, ConditionalLogic, Extend, ALU, and RegisterFile, among others.

The ControlUnit module is responsible for decoding the instructions and controlling the flow of the processor. It receives input from the instruction register and outputs control signals to the other modules. The role of the ControlUnit module is to determine the operation to be performed and to generate the necessary control signals for the other modules.

The Decoder module receives input from the ControlUnit module and decodes the instruction to determine the operation to be performed. It outputs control signals to the ConditionalLogic, Extend, and ALU modules. The role of the Decoder module is to interpret the instruction and generate the necessary control signals for the subsequent stages of the processor.

The ConditionalLogic module receives input from the Decoder module and determines whether the instruction should be executed based on the current condition codes. It outputs control signals to the ALU and RegisterFile modules. The role of the ConditionalLogic module is to evaluate the condition codes and determine if the instruction should be executed.

The Extend module receives input from the Decoder module and extends the immediate value of the instruction to 32 bits. It outputs the extended immediate value to the ALU module. The role of the Extend module is to properly extend the immediate value of the instruction to the correct number of bits.

The ALU module receives input from the Extend and RegisterFile modules and performs the operation specified by the instruction. It outputs the result of the operation and the updated condition codes to the RegisterFile module. The role of the ALU module is to perform the operation specified by the instruction and update the condition codes.

The RegisterFile module receives input from the ControlUnit, ALU, and DataMemory modules and stores the results of the operations performed by the processor. It also retrieves values from the registers when needed. The role of the RegisterFile module is to store and retrieve values from the registers.

In the 32-bit ARM single cycle processor, the wire setting involves defining the inputs and outputs of the various modules that make up the processor. The wires are used to connect the different modules and pass data between them. In this particular processor, there are various wires that are defined to carry the different data inputs and outputs. These include the ReadAddr1 and ReadAddr2 wires, which are used to carry the register addresses for the first and second operands, respectively. There is also the WriteAddr wire, which is used to carry the register address for the result of the operation.

The wire setting also includes the definition of the Result, PCPlus4, PCBranch, and NextPC wires. The Result wire is used to carry the result of the operation from the ALU module to the register file. The PCPlus4 wire is used to carry the value of the current program counter plus 4, which is used as the next program counter value in the case of a data processing or load/store instruction. The PCBranch wire is used to carry the target address for a branch instruction, which is calculated by adding the current program counter value and the immediate value. The NextPC wire is used to carry the value of the next program counter, which is determined by the value of the PCSrc wire.

```
//TODO: Wire setting
wire [3:0] ReadAddr1;
wire [3:0] ReadAddr2;
wire [3:0] WriteAddr;

wire [31:0] Result;
wire [31:0] PCPlus4;
wire [31:0] PCBranch;
wire [31:0] NextPC;
wire PCSrc;
wire [31:0] SrcB;
wire [3:0] NZCV;
wire [3:0] ALUOp;
wire RegWrite;
wire MemWrite;
wire MemtoReg;
wire ALUSrc;
wire Svalue;
wire [31:0] ReadData1;
wire [31:0] ReadData2;
wire [31:0] ReadData3;
wire [ 1:0] ImmSrc;
wire [31:0] ExtImm;
wire [31:0] ALUResult;
wire [3:0] ALUFlags;
wire [1:0] RegSrc;
```

```
//TODO: Assign values to variables
//assign /*Variable*/ = /*Value*/
assign ReadAddr1 = (RegSrc == 1'b0)? inst[19:16] : 4'd15;
assign ReadAddr2 = (RegSrc == 1'b0)? inst[3:0] : inst[15:12];
assign WriteAddr = inst[15:12];
assign memaddr = ALUResult;
assign memwrite = MemWrite;

assign Result = (MemtoReg == 1'b1)? readdata : ALUResult;
assign SrcB = (ALUSrc == 1'b1)? ExtImm : ReadData2;
assign writedata = ReadData2;

assign PCPlus4 = pctmp + 32'd4;
assign PCBranch = pctmp + ExtImm + 32'd8;
assign NextPC=(PCSrc== 1'b0) ? PCPlus4 : PCBranch;

//assign NZCV = Svalue ? ALUFlags : NZCV;
assign NZCV = Svalue ? ALUFlags : 4'h0;
```

The Assign values to variables part of the code involves assigning values to the various variables that are used in the processor. This includes assigning values to the ReadAddr1 and ReadAddr2 variables, which are used to determine the register addresses for the first and second operands. The WriteAddr variable is also assigned a value, which is used to determine the register address for the result of the operation. The memaddr and memwrite variables are also assigned values, which are used to determine the memory address and write enable signal for memory accesses. The writedata variable is assigned the value of the ReadData2 wire, which carries the second operand value. The Assign values to variables part also includes the assignment of values to the Result, SrcB, and PCPlus4 variables. The Result variable is assigned the value of either the readdata wire or the ALUResult wire, depending on the value of the MemtoReg wire. The SrcB variable is assigned the value of either the ExtImm wire or the ReadData2 wire, depending on the value of the ALUSrc wire. The PCPlus4 variable is assigned the value of the current program counter plus 4. Finally, the NextPC variable is assigned the value of either the PCPlus4 or PCBranch wire, depending on the value of the PCSrc wire.

In conclusion, the 32-bit ARM single cycle processor is a system designed to execute instructions in a single clock cycle. It is made up of various modules, including the ControlUnit, Decoder, ConditionalLogic, Extend, ALU, and RegisterFile, which work together to interpret and execute instructions. The processor is able to perform various operations, including data processing, load and store, and branch instructions, and is able to update the condition codes as needed.

## 2. RegisterFile

- **RegisterFile Module**
  - Register 14 ($14) – Link Register (LR)
    - This register holds the address of the next instruction after a Branch a nd Link instruction
    - $14 = PC + 4
  - Register 15 ($15)
    - It can be used in most instructions as a pointer to the instruction whic h is two instructions after the instruction being executed
    - $15 = PC + 8

- **PC (It is different with Register 15($15))**
  - PC (next instruction) = PC (current instruction) + 4

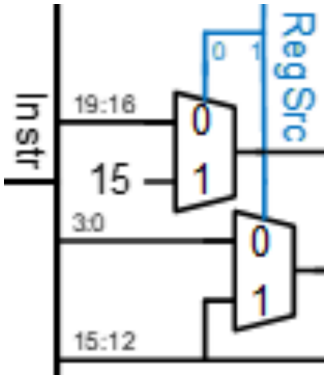
In the single-cycle processor, the Register Module works like a memory module that stores the values of registers in the processor. The role of the register file module is to read and write to these registers as directed by the processor. It also stores the program counter (pc) value, which is used to keep track of the current instruction being executed.

To implement this, the register file module uses a clocked always block to write to the registers. The block is triggered on the negative edge of the clock signal, and it first checks if the reset input is active. If it is, all the registers are reset to their default value of 0. If the reset input is not active, the block checks if the write enable (we) input is active. If it is, the block writes the data specified by the wdata input to the register specified by the waddr input, unless the RegSrc[0] input is active, in which case the value of the pc plus 4 is written to register 14. If the we input is not active, the block updates the pc value with the pcin input, unless the waddr input is all 1s and the RegSrc[0] input is not active.

Another clocked always block is used to read from the registers. This block is triggered on the positive edge of the clock signal, and it operates in a similar manner to the write block. If the reset input is active, the output data signals data1, data2, and data3 are all set to 0. If the reset input is not active, the block reads the values of the registers specified by the addr1, addr2, and addr3 inputs and assigns them to data1, data2, and data3, respectively. If any of these addresses are equal to 15, the corresponding output data is set to the value of the pc plus 8. The RegSrc[1] input is used as a multiplexer to select either the value of the register specified by the addr2 input or the value of the register specified by the waddr input for the data2 output.

The implementation result of this role of the register file module is a module that can read and write to the processor's registers and store the pc value. It can also reset all the registers to their default value and update the pc value as needed, with additional functionality to write to register 14 and multiplex the data2 output.

## 3. ControlUnit



In the single-cycle processor, the ControlUnit module works like a decoder which decodes the instructions that are given to the processor. The role of the control unit module is to determine the actions that need to be performed based on the instruction that is given, and to generate the necessary control signals for the other modules in the processor. In order to implement this role, I have implemented a decoder and a conditional logic module in the ControlUnit module.

The Decoder module in this single-cycle processor is responsible for decoding the instructions that are given to the processor and generating the necessary control signals for the other modules in the processor. The module takes in two inputs: the opcode (op) and the funct field of the instruction. Based on the values of these inputs, the module generates several output control signals: MemtoReg, ALUSrc, ImmSrc, RegSrc, ALUOp, and Svalue.

The MemtoReg control signal determines whether the result of the instruction should be written to the memory or to a register. The ALUSrc control signal determines whether the second operand of the ALU should come from a register or from an immediate value. The ImmSrc control signal determines the source of the immediate value for the instruction. The RegSrc control signal determines the source of the register operand for the instruction. The ALUOp control signal determines the operation that should be performed by the ALU. The Svalue control signal is used to distinguish between load (LDR) and store (STR) instructions.

The Decoder module uses a case statement to decode the instructions based on the values of the opcode and funct fields. For data processing instructions (opcode = 00), the module sets the MemtoReg control signal to 0, the ALUOp control signal to the value of the funct[4:1] field, the Svalue control signal to the value of the funct[0] field, the ImmSrc control signal to 00 (unsigned 8-bit immediate value), the ALUSrc control signal to the value of the funct[5] field, and the RegSrc control signal to 00. For load/store instructions (opcode = 01), the module sets the MemtoReg control signal to the value of the funct[0] field, the ALUOp control signal to either add (0100) or subtract (0010) based on the value of the funct[3] field, the Svalue control signal to the value of the funct[0] field, the ImmSrc control signal to 01 (unsigned 12-bit immediate offset), the ALUSrc control signal to the value of the funct[5] field, and the RegSrc control signal to 00. For branch instructions (opcode = 10), the module sets the MemtoReg control signal to 0, the ALUOp control signal to 0100, the Svalue control signal to 0, the ImmSrc control signal to 10 (2's complement 24-bit offset), the ALUSrc control signal to 1, and the RegSrc control signal to 01. For all other instructions, the module sets all control signals to default values.

Overall, the Decoder module in this single-cycle processor plays a crucial role in the correct execution of instructions by generating the necessary control signals for the other modules in the processor based on the values of the opcode and funct fields of the instruction.

```
//TODO: Write code for decoder
/*
(1) Data processing instruction
(2) Load, Store instruction
(3) Branch instruction
*/

/*
-----------------------------------
   Data processing (Page 10/60  in arm_inst_set_manual2.pdf)
   funct[5]:Immediate Operand.        0 = source operand is a register.  1 = Unsigned 8 bit immediate value
   funct[0]:Set condition codes.      0 = do not alter condition codes.  1 = set condition codes.

    (From slides. Single-Cycle ARM Processor Design 2nd Term Project Single-cycle Processor)
    RegSrc[0] = "0" : ADD, SUB, CMP, MOV, LDR, STR
        Using Register[inst(19:16)]
    RegSrc[0] = "1" : B, BL
        Using Register[15] for calculating branch target address

-----------------------------------
   LDR/STR (Page 26/60  in arm_inst_set_manual2.pdf)
   funct[5]:Immediate Operand.   0 = source operand is a register.  1 = Unsigned 12 bit immediate offset
   funct[3]:Up/Down bit.         0 = down; subtract offset from base.   1 = up; add offset to base.

   funct[0]:Load/Store bit.      0 = Store to memory. 1 = Load from memory.
            Svalue is the bit which is used for distinguishing LDR and STR instruction.
-----------------------------------
   BL (Page 8/60  in arm_inst_set_manual2.pdf)
   2's complement 24 bit offset
*/
```

The ConditionalLogic module in this single-cycle processor is responsible for generating the control signals for the processor based on the values of the opcode, funct field, and condition code (cond) of the instruction, as well as the value of the Zero signal. The module takes in four inputs: the opcode (op), the funct field, the condition code, and the Zero signal. Based on these inputs, the module generates three output control signals: PCSrc, RegWrite, and MemWrite.

The PCSrc control signal determines whether the program counter (PC) should be updated based on the result of the instruction or if it should be updated based on a branch instruction. The RegWrite control signal determines whether the result of the instruction should be written to a register. The MemWrite control signal determines whether the result of the instruction should be written to memory.

The ConditionalLogic module first generates a cond_true signal based on the value of the condition code and the Zero signal. The cond_true signal is 1 if the condition code is 0 and the Zero signal is 1, or if the condition code is 1 and the Zero signal is 0. For all other values of the condition code, the cond_true signal is 0.

The module then uses a case statement to generate the control signals based on the values of the opcode and funct fields. For data processing instructions (opcode = 00), the module sets the PCSrc control signal to 0, the MemWrite control signal to 0, and the RegWrite control signal to the cond_true signal ANDed with the result of the comparison of the funct[4:1] field to 1010 (except for the CMP instruction). For load/store instructions (opcode = 01), the module sets the PCSrc control signal to 0, the RegWrite control signal to the cond_true signal ANDed with the value of the funct[0] field (load instruction), and the MemWrite control signal to the cond_true signal ANDed with the negation of the funct[0] field (store instruction). For branch instructions (opcode = 10), the module sets the PCSrc control signal to the cond_true signal, the RegWrite

control signal to 0, and the MemWrite control signal to 0. For all other instructions, the module sets all control signals to 0.

Overall, the ConditionalLogic module in this single-cycle processor plays a crucial role in the correct execution of instructions by generating the necessary control signals for the processor based on the values of the opcode, funct field, condition code, and Zero signal. It ensures that the instructions are only executed if the specified condition is met, and generates the appropriate control signals for updating the PC, writing to registers, and writing to memory.

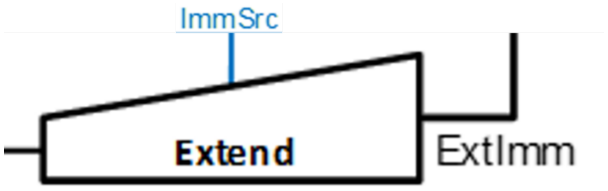
```
    reg cond_true;
    always @(*)
    begin
        case(cond)
        4'b0000: cond_true = Zero;
        4'b0001: cond_true = ~Zero;
        default: cond_true = 1'b0;
        endcase
    end

    always @(*)
    begin
        case(op)
        2'b00: begin // Data processing instruction
            PCSrc = 1'b0;
            MemWrite = 1'b0;
            RegWrite = cond_true & (funct[4:1] != 4'b1010); // Except CMP
        end
        2'b01: begin  // Load, Store instruction
            PCSrc = 1'b0;
            RegWrite = cond_true & funct[0]; // Load.
            MemWrite = cond_true & (~funct[0]); // Store.
        end
        2'b10: begin  // Branch instruction
            PCSrc = cond_true;
            RegWrite = 1'b0;
            MemWrite = 1'b0;
        end
        default: begin
            PCSrc = 1'b0;
            RegWrite = 1'b0;
            MemWrite = 1'b0;
        end
        endcase
    end

endmodule
```

The implementation result of this role of the control unit module is a fully functional decoder and conditional logic module which are able to decode the instructions that are given to the processor and generate the necessary control signals for the other modules in the processor. This allows the processor to correctly execute the instructions that are given to it, and to perform the necessary actions based on the instruction that is given.

## 4. Extend




The Extend module in this single-cycle processor is responsible for extending the immediate value of an instruction to the full 32-bit width of the processor. The module takes in two inputs:

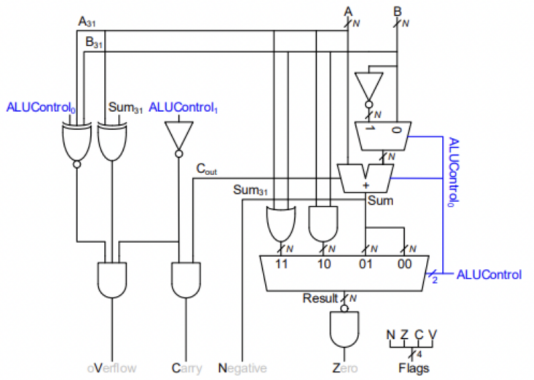
the immediate value (in) and the immediate source signal (ImmSrc). Based on these inputs, the module generates a single output signal: the extended immediate value (ExtImm).

The role of the Extend module is to ensure that the immediate value of an instruction is properly sign or zero extended to the full width of the processor, based on the value of the ImmSrc signal. The ImmSrc signal indicates the type of immediate value being passed to the module, and can take on one of three values: 00, 01, or 10.

To implement the Extend module, a case statement is used to distinguish between the different values of the ImmSrc signal. For ImmSrc = 00 (zero extended-imm8), the Extend module zero extends the 8-bit immediate value to a 32-bit value by concatenating 24 0s to the least significant 8 bits of the in signal. For ImmSrc = 01 (zero extended-imm12), the Extend module zero extends the 12-bit immediate value to a 32-bit value by concatenating 20 0s to the least significant 12 bits of the in signal. For ImmSrc = 10 (sign-extended-imm24, shifting left two bits), the Extend module sign extends the 24-bit immediate value to a 32-bit value by concatenating the 6 most significant bits of the in signal to the left of the in signal, shifting the in signal two bits to the left, and concatenating two 0s to the right of the in signal.

The implementation result of the Extend module is a 32-bit extended immediate value that is properly sign or zero extended based on the value of the ImmSrc signal. This extended immediate value is used by the processor to correctly execute instructions that utilize an immediate value.

## 5. ALU




In the single-cycle processor, the ALU module is responsible for performing various arithmetic and logical operations on the input operands, and outputting the result of the operation as well as the status flags. The role of the ALU module is to execute the instructions specified by the control unit and provide the necessary data for other modules in the processor.

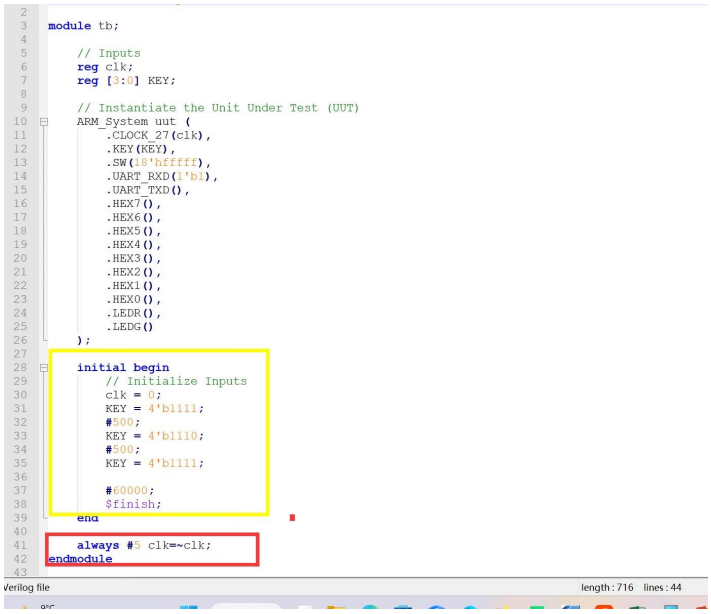
To implement the ALU module, I have used a case statement to specify the different operations to be performed based on the value of the ALUOp input. The ALUOp input is a 4-bit signal that specifies the operation to be performed by the ALU. The possible values for ALUOp are ADD, SUB, CMP, and MOV.

The ADD operation is performed by adding the values of SrcA and SrcB and storing the result in the ALUResult output. The SUB operation is performed by subtracting the value of SrcB from SrcA and storing the result in the ALUResult output. The CMP operation is performed by subtracting the value of SrcB from SrcA, but the result is not stored in the ALUResult output. Instead, the ALUFlags output is set to indicate the status of the result. The MOV operation simply copies the value of SrcB to the ALUResult output. In the default case, the ALUResult output is set to 0 and the ALUFlags output is set to 0.

The implementation result of this role of the ALU module is a module that is able to perform a variety of arithmetic and logical operations on the input operands and output the result and status flags as needed. This allows the single-cycle processor to execute the instructions specified by the control unit and provide the necessary data for other modules in the processor.

## Simulation

Testbench:



```
module tb;

    // Inputs
    reg clk;
    reg [3:0] KEY;

    // Instantiate the Unit Under Test (UUT)
    ARM_System uut (
        .CLOCK_27(clk),
        .KEY(KEY),
        .SW(18'hfffff),
        .UART_RXD(1'b1),
        .UART_TXD(),
        .HEX7(),
        .HEX6(),
        .HEX5(),
        .HEX4(),
        .HEX3(),
        .HEX2(),
        .HEX1(),
        .HEX0(),
        .LEDR(),
        .LEDG()
    );

    initial begin
        // Initialize Inputs
        clk = 0;
        KEY = 4'b1111;
        #500;
        KEY = 4'b1110;
        #500;
        KEY = 4'b1111;

        #60000;
        $finish;
    end

    always #5 clk=~clk;
endmodule
```

Verilog file length : 716 lines : 44

A clock is being added to the FPGA in the simulation environment.
A reset button is being simulated by writing the appropriate values to the ports of the instantiated ARM_System module in the testbench.
The clock signal is simulated by flipping its value every 5 nanoseconds, creating a continuous clock. The reset button is simulated by starting with a value of 1111, then changing it to 1110 after 500 nanoseconds, and then releasing it.

In the actual hardware setup, a crystal oscillator would provide the clock signal for the ARM_System processor. If this is not available in the simulation environment, the clock can be simulated using the "always #5 clk=~clk" code. This clock signal is used by the processor to execute instructions, including fetching, decoding, and executing.
The value of the program counter (pc) is divided by 4 to get the index in the assignment1 memory-initialized file (mif) and the instructions on the waveform are consistent with this.