# Computer Architecture Lab Report

Amirhosein Yavarikhoo 810199514

February 2023

Faculty of Electrical and Computer Engineering

University of Tehran

# ARM Implementation

The goal is to implement the ARM CPU as described below.

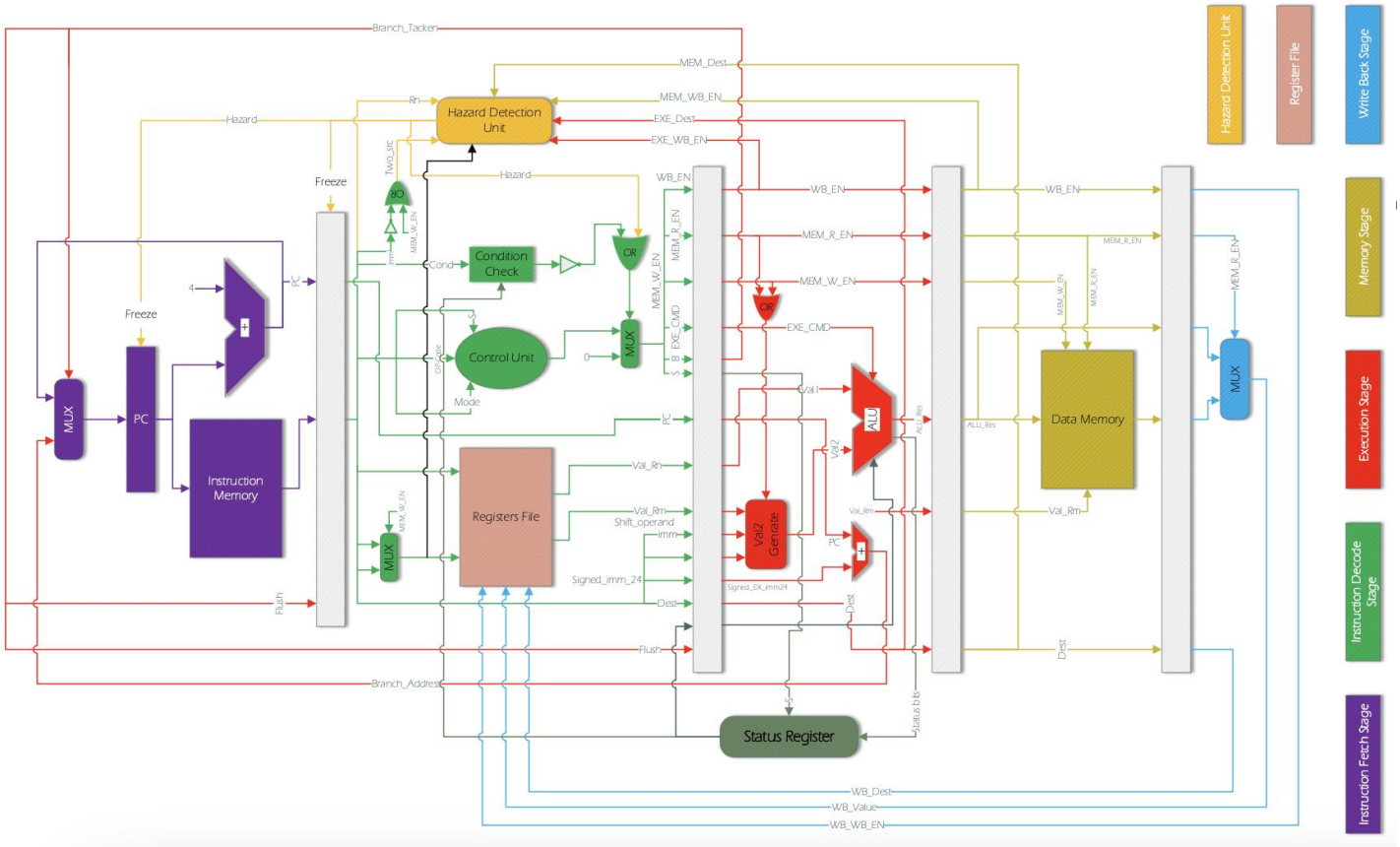



*Figure 1 ARM Schematic*

## Instruction Fetch Stage

To implement the Instruction Fetch Stage, we will first work on the purple sections of Figure 1. This section includes the Program Counter (PC) register, which increments by 4 units each time to fetch the next instruction. Additionally, it contains an Instruction Memory where the processor instructions are stored.

Next, we will implement the remaining four pipeline stages and five processor registers without their contents to observe the movement of the PC.

The implementation of the Instruction Fetch (IF) stage is as follows:

fetch.v

```
module IF_Stage(input clk, rst, freeze, Branch_taken, input [31:0] BranchAddr, output [31:0] PC, instruction);
    wire [31:0] PC_in, PC_in_reg;
    Mux2to1 PC_Mux(.a(PC), .b(BranchAddr), .sel(Branch_taken), .out(PC_in));
    Reg32 PC_Reg (.clk(clk), .rst(rst), .load(~freeze), .data_in(PC_in), .data_out(PC_in_reg));
    Adder PC_Adder(.a(32'd4), .b(PC_in_reg), .out(PC));
    Instruction_Memory Inst_Mem(.address(PC_in_reg), .out(instruction));
endmodule
```

*Figure 2 IF_Stage implementation*

Reg32.v

```
module Reg32 (clk,rst,load,data_in,data_out
);
  input clk;
  input rst;
  input load;
  input [31:0] data_in;
  output reg [31:0] data_out;
  always @(posedge clk) begin
    if (rst) data_out = 32'd0;
    else if (load) data_out = data_in;
  end
endmodule
```

*Figure 3 simple register with asynchronous reset*

Instruction_Memory.v

```
1   module Instruction_Memory (
2       address,
3       out
4   );
5     input [31:0] address;
6     output[31:0] out;
7     reg [7:0] mem[187:0];
8     initial begin
9     $readmemb("Inst.txt",mem);
10    end
11    assign out = {mem[address], mem[address+1], mem[address+2], mem[address+3]};
12  endmodule
```

*Figure4 Memory module description*

First, we will test the IF stage itself. It will be observed that the PC is incremented by 4 regularly, and various instructions are read from the Instruction Memory:

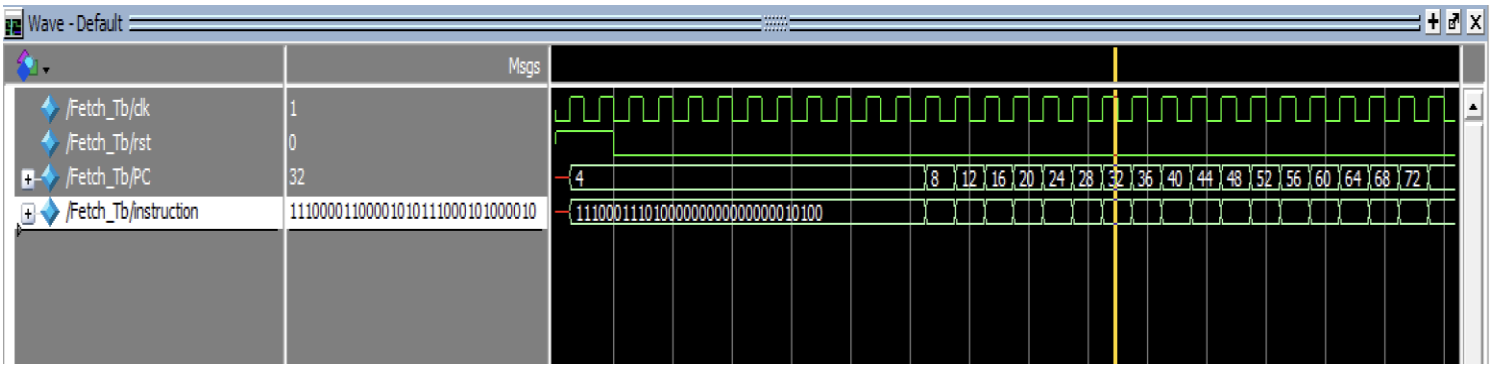



*Figure5 Fetch stage testbench*

IF Register Implementation:

IF_Stage_Reg.v

```
1   module IF_Stage_Reg(input clk, rst, freeze, flush,
2   input[31:0] PC_in, Instruction_in,
3   output reg [31:0] PC, Instruction);
4
5   always @(posedge clk) begin
6       if (rst) begin PC = 32'd0; Instruction = 32'b0; end
7       else if (~freeze && ~flush) begin PC = PC_in; Instruction = Instruction_in; end
8       else if (flush) begin PC = 32'd0; Instruction = 32'b0; end
9   end
10
11  endmodule
```

*Figure 6 Fetch stage register*

Now we test the implementation of various stage registers on board using Signal Tap tool of Quartus.

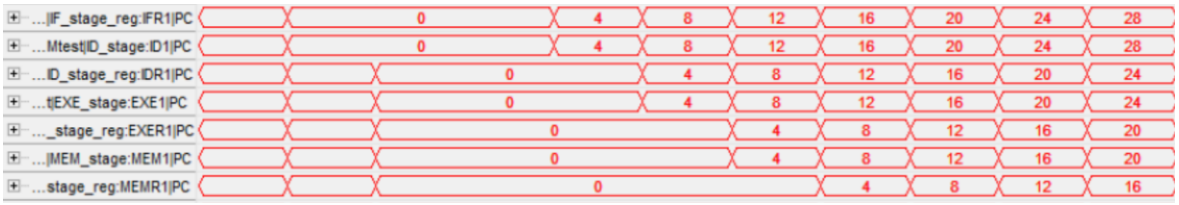



*Figure 7 Signal Tap of Fetch Stage*

## Stage Instruction Decode

In this stage, we need to decode the instructions that we read from memory in the previous stage.

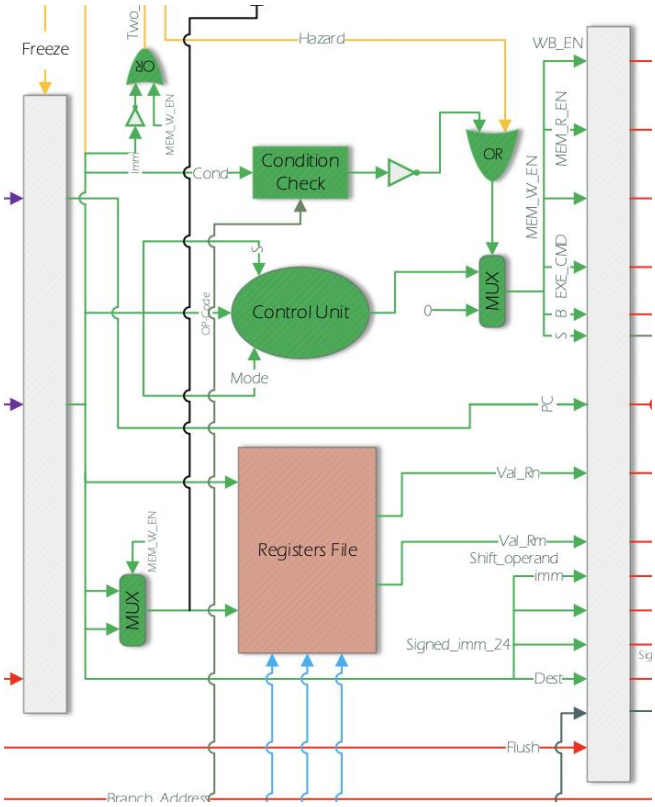



*Figure8 ID_Stage diagram*

To do this, we will first implement the Register File module. We will set the initial values of the registers to be equal to their register numbers. If WriteBackEn is set to one, it will place the input value at the specified address, continuously outputting the values at the src1 and src2 addresses.

```
module Register_File (
    input clk,
    rst,
    input [3:0] src1,
    src2,
    Dest_wb,
    input [31:0] Result_WB,
    input writeBackEn,
    output [31:0] reg1,
    reg2
);

  reg [31:0] reg_file[0:15];
  integer i;

  initial begin
    for (i = 0; i < 16; i = i + 1) reg_file[i] <= i;
  end

  always @(negedge clk) begin
    if (rst) begin
      for (i = 0; i < 16; i = i + 1) begin
        reg_file[i] <= i;
      end
    end else if (writeBackEn) begin
      reg_file[Dest_wb] <= Result_WB;
    end
  end

  assign reg1 = reg_file[src1];
  assign reg2 = reg_file[src2];

endmodule
```

*Figure 9 Register File*

Now, we will implement the Control Unit. This module, by taking the opcode, s, and mode from the instruction, determines the type of instruction concerning reading and writing to memory, etc. It also specifies which instruction should proceed to the next stage, the ALU stage (ExecuteCommand), so that the ALU can perform the corresponding operation. Another output from this unit relates to branch instructions, indicating whether a

branch operation is to be executed or not..

```
module Control_Unit (
    input [1:0] mode,
    input [3:0] Op_code,
    input s,
    output reg [3:0] ExecuteCommand,
    output reg mem_read,
    mem_write,
    WB_Enable,
    output reg B,
    status
);

  always @(s, Op_code, mode) begin  // check
    {WB_Enable, mem_read, mem_write, ExecuteCommand, B, status} = {8'b0, s};
    case (mode)
      2'b00: begin
        case (Op_code)
          4'b1101: {WB_Enable, ExecuteCommand} = 5'b10001;
          4'b1111: {WB_Enable, ExecuteCommand} = 5'b11001;
          4'b0100: {WB_Enable, ExecuteCommand} = 5'b10010;
          4'b0101: {WB_Enable, ExecuteCommand} = 5'b10011;
          4'b0010: {WB_Enable, ExecuteCommand} = 5'b10100;
          4'b0110: {WB_Enable, ExecuteCommand} = 5'b10101;
          4'b0000: {WB_Enable, ExecuteCommand} = 5'b10110;
          4'b1100: {WB_Enable, ExecuteCommand} = 5'b10111;
          4'b0001: {WB_Enable, ExecuteCommand} = 5'b11000;
          4'b1010: {ExecuteCommand} = 4'b0100;
          4'b1000: {ExecuteCommand} = 4'b0110;
        endcase
      end
      2'b01: begin
        ExecuteCommand = 4'b0010;
        case (s)
          1'b1: {WB_Enable, mem_read, status} = {3'b110};
          1'b0: {mem_write, status} = {2'b10};
        endcase
      end
      2'b10: begin
        {B, status} = 2'b10;
      end
    endcase
  end

endmodule
```

*Figure 10 Control Unit*

In the Condition Check module, we generate a single output bit, cond_Check_Out, by receiving the cond bits from the instruction and the status register bits. This output, along with the hazard bit, determines whether the control signals should proceed to the next stage or if all of them should be set to zero.

```
module Condition_Check (
    cond,
    status,
    cond_Check_Out
);

  input [3:0] cond;
  input [3:0] status;
  output cond_Check_Out;
  reg out;
  assign N = status[3];
  assign Z = status[2];
  assign C = status[1];
  assign V = status[0];
  assign cond_Check_Out = out;
  always @(cond, Z, C, N, V) begin
    out = 1'b0;
    case (cond)
      4'b0000: out = Z;
      4'b0001: out = ~Z;
      4'b0010: out = C;
      4'b0011: out = ~C;
      4'b0100: out = N;
      4'b0101: out = ~N;
      4'b0110: out = V;
      4'b0111: out = ~V;
      4'b1000: out = C & ~Z;
      4'b1001: out = ~C & Z;
      4'b1010: out = N ~^ V;
      4'b1011: out = N ^ V;
      4'b1100: out = ~Z & (N ~^ V);
      4'b1101: out = Z & (N ^ V);
      4'b1110: out = 1'b1;

    endcase
  end
endmodule
```

*Figure 11 Condition Check*

ID Register:

ID_STAGE_REG.v

```
module ID_Stage_Reg(
    input clk, rst, flush,
    input WB_EN_IN, MEM_R_EN_IN, MEM_W_EN_IN,
    input B_IN, S_IN,
    input[3:0] EXE_CMD_IN,
    input[31:0] PC_IN,
    input[31:0] Val_Rn_IN, Val_Rm_IN,
    input imm_IN,
    input[11:0] Shift_operand_IN,
    input[23:0] Signed_imm_24_IN,
    input[3:0] Dest_IN,SR,

    output reg WB_EN, MEM_R_EN, MEM_W_EN,
    output reg B, S,
    output reg[3:0] EXE_CMD,
    output reg[31:0] PC,
    output reg[31:0] Val_Rn, Val_Rm,
    output reg imm,
    output reg[11:0] Shift_operand,
    output reg[23:0] Signed_imm_24,
    output reg[3:0] Dest,SR_out
    );

    always @(posedge clk, posedge rst) begin
    if (rst) begin
            WB_EN=1'd0; MEM_R_EN=1'd0; MEM_W_EN =1'd0; B =1'd0; S =1'd0;
            EXE_CMD =4'd0;
            PC =32'd0;
            Val_Rn =32'd0; Val_Rm =32'd0;
            imm =1'd0;
            Shift_operand =12'd0;
            Signed_imm_24 =24'd0;
            Dest=4'd0;
            SR_out = 4'b0000;
        end
    else if (flush) begin
            WB_EN=1'd0; MEM_R_EN=1'd0; MEM_W_EN =1'd0; B =1'd0; S =1'd0;
            EXE_CMD =4'd0;
            PC =32'd0;
            Val_Rn =32'd0; Val_Rm =32'd0;
            imm =1'd0;
            Shift_operand =12'd0;
            Signed_imm_24 =24'd0;
            Dest=4'd0;
            SR_out = 4'b0000;
        end
    else begin
    WB_EN=WB_EN_IN; MEM_R_EN =MEM_R_EN_IN; MEM_W_EN =MEM_W_EN_IN;
    B =B_IN; S =S_IN;
    EXE_CMD =EXE_CMD_IN;
    PC =PC_IN;
    Val_Rn =Val_Rn_IN; Val_Rm =Val_Rm_IN;
    imm =imm_IN;
```

*Figure 12 ID_Stage Register*

Now, we will test the ID and IF stages together using a limited number of instructions:

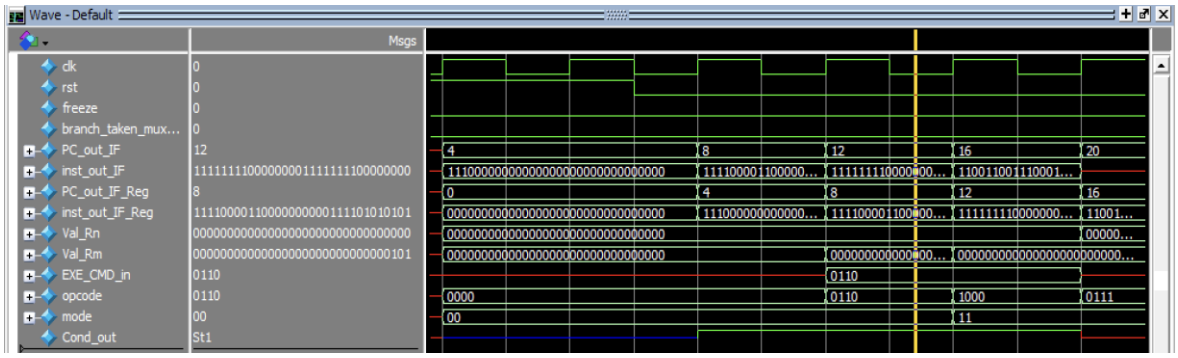



*Figure 13 ID testbench*

Based on the provided inputs, each of the wires mode, exe_cmd, opcode, and cond_out acquires the correct values. Additionally, the PC value was validated at various points.

The image below shows the test of the first 18 instructions as the system completes:

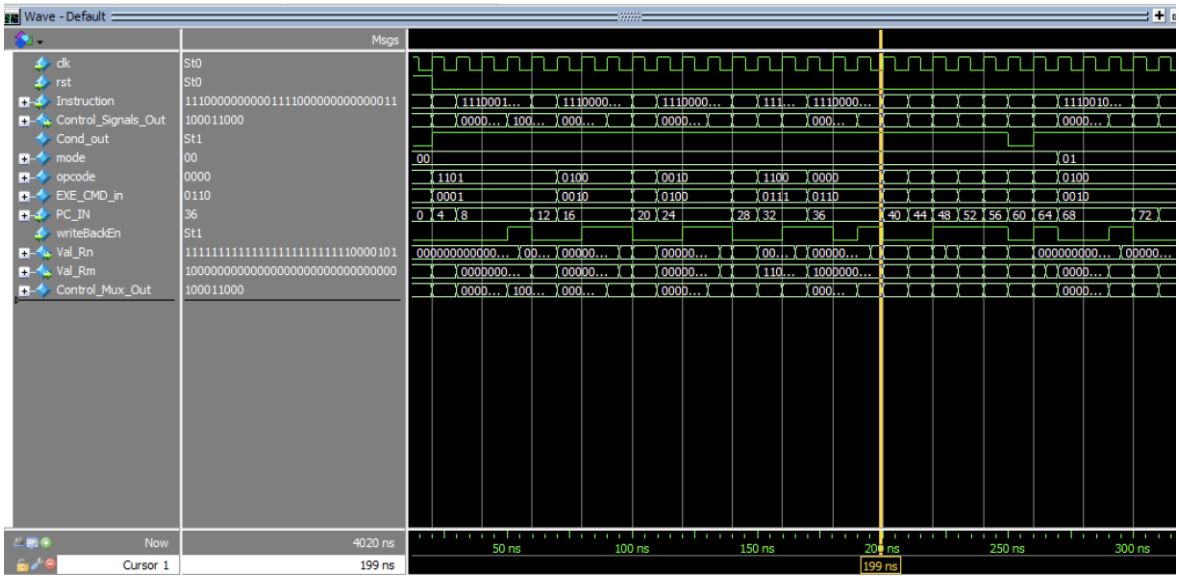
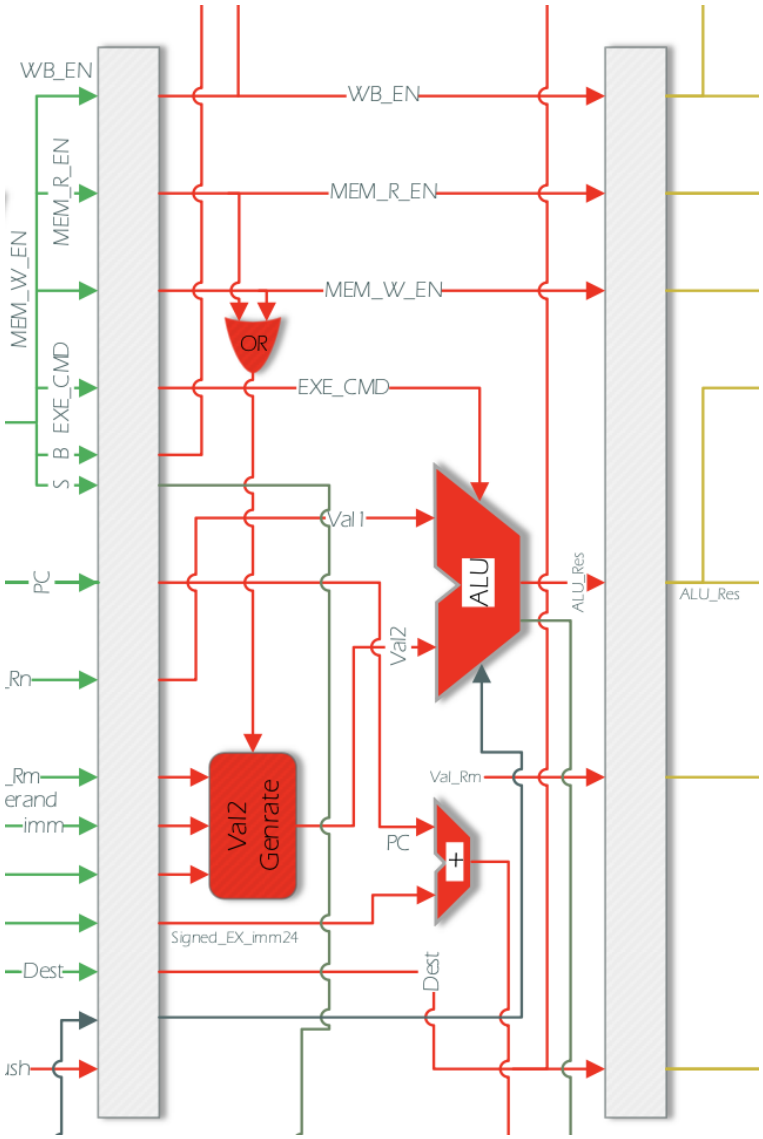



*Figure 14 ID testbench*

# Execution Stage

In this stage, we will perform the calculations for various instructions.




*Execution Stage diagram 15 Figure*

First, we will proceed to create the ALU module.

```
module ALU ( input [31:0] Val1,Val2, input [3:0] EXE_CMD, input [3:0] status_bits, output reg [31:0] out, output [3:0] status_bits_out
);
  parameter[3:0] MOV = 4'b0001, MVN = 4'b1001, ADD = 4'b0010, ADC = 4'b0011, SUB = 4'b0100, SBC = 4'b0101, AND = 4'b0110
    , ORR = 4'b0111, EOR = 4'b1000, CMP = 4'b0100, TST = 4'b0110, LDR = 4'b0010, STR = 4'b0010;
  wire n, z, c, v;
  assign n = status_bits[3];
  assign z = status_bits[2];
  assign c = status_bits[1];
  assign v = status_bits[0];
  wire n_out, z_out, v_out;
  reg c_out;
  always @(EXE_CMD, Val1, Val2, status_bits) begin
    c_out = c;
    case (EXE_CMD)
      MOV: out = Val2;
      MVN: out = ~Val2;
      ADD: begin
        {c_out, out} = Val1 + Val2;
      end
      ADC: begin
        {c_out, out} = Val1 + Val2 + c;
      end
      SUB: begin
        {c_out, out} = Val1 - Val2;
      end
      SBC: begin
        {c_out, out} = Val1 - Val2 - !c;
      end
      AND: out = Val1 & Val2;
      ORR: out = Val1 | Val2;
      EOR: out = Val1 ^ Val2;
      CMP: begin
        {c_out, out} = Val1 - Val2;
      end
      TST: out = Val1 & Val2;
      LDR: begin
        {c_out, out} = Val1 + Val2;
      end
      STR: begin
        {c_out, out} = Val1 + Val2;
      end
      default: out = 32'd0;
    endcase
  end
  assign n_out = out[31];
  assign z_out = ~(|out);
  assign v_out = ((EXE_CMD == 4'b0010) | (EXE_CMD == 4'b0011))?
   (out[31] & ~Val1[31] & ~Val2[31]) | (~out[31] & Val1[31] & Val2[31])
    :((EXE_CMD == 4'b0100) | (EXE_CMD == 4'b0101))?
   (out[31] & ~Val1[31] & Val2[31]) | (~out[31] & Val1[31] & ~Val2[31])
    : 1'b0;
  assign status_bits_out = {n_out, z_out, c_out, v_out};
endmodule
```

*Figure 16 ALU Module*

In this module, based on the EXE_CMD received from the ID stage, the ALU performs an operation defined in a switch-case structure. After executing the operation, the corresponding output is generated. Additionally, depending on the type of operation and operands, the n, v, c, and z bits are set and sent to the status register.

Next, we will implement the val2generator module. This module is used to determine the second input for the ALU. If the instruction is of type 32-bit immediate, the immediate

value is determined based on the shift operand. If the instruction is an immediate shift, we shift the ValRm.

To perform the shift operations, we need a barrel shifter capable of executing the specified shifts. The code for the barrel shifter, along with the val2generator, is provided below.

Val2_Generator.v

```
module Val2_Generate(input [31:0] Val_Rm,input MEM_OP,imm,input [11:0] shift_operand, output [31:0] out );
wire [31:0] imm_out;
wire [31:0] imm_shift,imm_rotate;
wire [40:0] imm_long;
assign imm_long = {shift_operand[7:0],24'd0,shift_operand[7:0]}>>{shift_operand[11:8],1'b0};
assign imm_out = imm_long[31:0];
//imm_creator imm_gen (shift_operand[11:8],shift_operand[7:0],imm_out);
barrel_shifter rotate_right (Val_Rm,shift_operand[11:7],imm_rotate);
assign out = MEM_OP?{20'd0,shift_operand}:
             imm?imm_out:
             ((imm==1'b0)&&(~shift_operand[4]))?imm_shift:Val_Rm;
assign imm_shift = (shift_operand[6:5]==2'b00)?Val_Rm<<shift_operand[11:7]:
                   (shift_operand[6:5]==2'b01)?Val_Rm>>shift_operand[11:7]:
                   (shift_operand[6:5]==2'b10)?Val_Rm>>>shift_operand[11:7]:
                   (shift_operand[6:5]==2'b11)?imm_rotate:32'd0;
endmodule
```

*Figure 17 Val2_generator*

We will connect the modules together and place them in the EXE stage:

EXE_STAGE.v

```
module EXE_Stage(
    input clk,
    input [3:0] EXE_CMD,
    input MEM_R_EN, MEM_W_EN,
    input [31:0] PC,
    input [31:0] Val_Rn, Val_Rm,
    input imm,
    input [11:0] Shift_operand,
    input [23:0] Signed_imm_24,
    input [3:0] SR,

    output [31:0] ALU_result, Br_addr,
    output [3:0] status
);
    wire mem_op;
    wire [31:0] Val2;
    assign mem_op = MEM_R_EN|MEM_W_EN;
    ALU alu1(.Val1(Val_Rn), .Val2(Val2), .EXE_CMD(EXE_CMD), .status_bits(SR), .out(ALU_result), .status_bits_out(status));
    Val2_Generate Val2gen (.Val_Rm(Val_Rm),.MEM_OP(mem_op),.imm(imm),.shift_operand(Shift_operand),.out(Val2));
    Adder PC_EXE_ADDER (.a(PC),.b({{{8{Signed_imm_24[23]}}, Signed_imm_24}}<<2),.out(Br_addr));
    //assign Br_addr = PC + Signed_imm_24;

endmodule
```

*Figure 18 Execution stage module*

EXE Register:

EXE_REG.v

```
module EXE_Stage_Reg(
    input clk, rst, WB_en_in, MEM_R_EN_in, MEM_W_EN_in,
    input[31:0] ALU_result_in, ST_val_in,
    input[3:0] Dest_in,
    output reg WB_en, MEM_R_EN, MEM_W_EN,
    output reg[31:0] ALU_result, ST_val,
    output reg[3:0] Dest
    );

    always @(posedge clk, posedge rst) begin
        if (rst) begin
            WB_en=1'd0; MEM_R_EN=1'd0; MEM_W_EN =1'd0;
            ALU_result =32'd0; ST_val =32'd0;
            Dest=4'd0;
            end
        else begin
            WB_en=WB_en_in; MEM_R_EN=MEM_R_EN_in; MEM_W_EN =MEM_W_EN_in;
            ALU_result =ALU_result_in; ST_val =ST_val_in;
            Dest=Dest_in;
            end
    end

endmodule
```

*Figure 19 Execution Stage Register*

EXE Testbench:

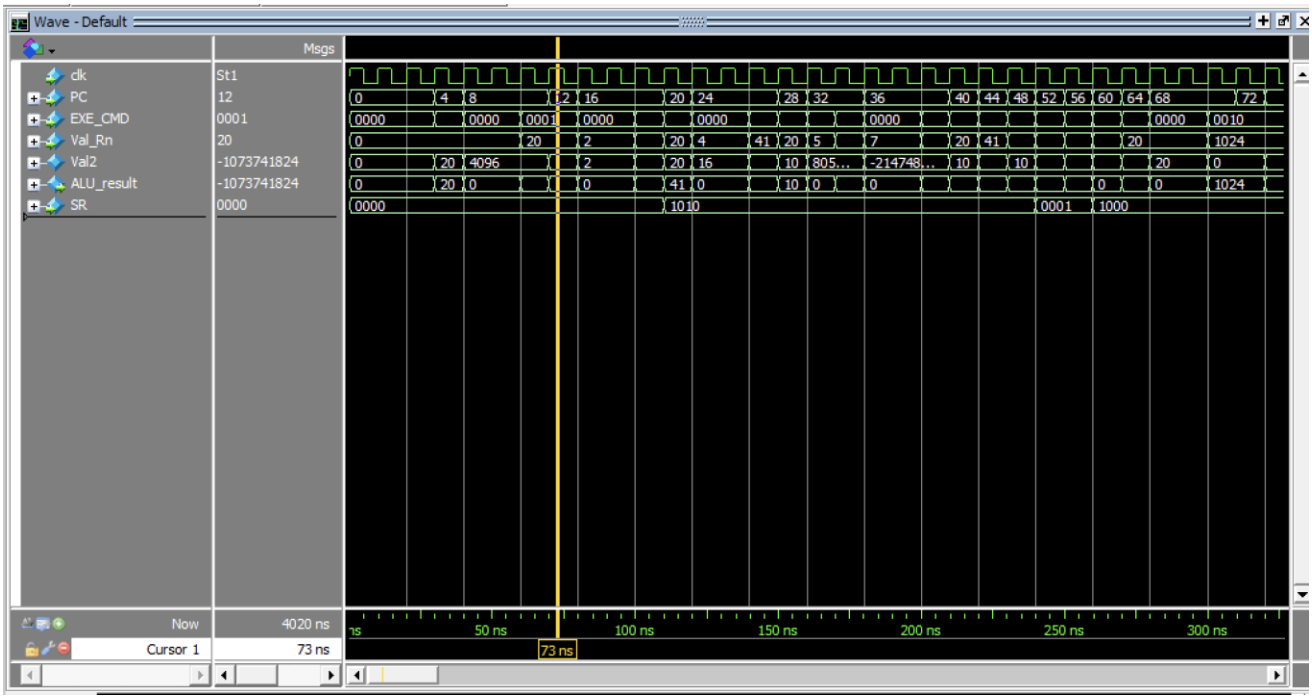



*Figure 20 Exe Stage*

# Stage Memory:

Now we add a memory module to this system.

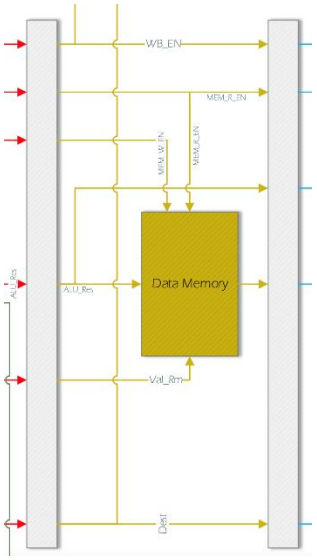



*Figure 21 Memory Stage*

The memory consists of 64 locations, each 32 bits wide, and based on the input signals, it performs read or write operations. We subtract 1024 from the input address and then right-shift it twice to determine the address where the desired data is located in memory.

Memory.v

```
module Memory (
    clk,
    MEM_R_EN,
    MEM_W_EN,
    address,
    d_in,
    d_out
);
  input clk, MEM_R_EN, MEM_W_EN;
  input [31:0] address;
  input [31:0] d_in;
  output [31:0] d_out;

  reg  [31:0] mem [0:64];
  wire [31:0] adr;

  always @(posedge clk) if (MEM_W_EN) {mem[(address-32'd1024)>>2]} <= d_in;

  assign d_out = (MEM_R_EN) ? mem[(address-32'd1024)>>2] : 32'd0;
endmodule
```

*Figure 22 Memory Module*

MEM_reg.v

```
module MEM_Stage_Reg(
    input clk,rst,
    WB_en_in,MEM_R_en_in,
    input [31:0] ALU_result_in,Mem_read_value_in,
    input [3:0] Dest_in,
    output reg WB_en,MEM_R_en,
    output reg [31:0] ALU_result,Mem_read_value,
    output reg [3:0] Dest
);

   always @(posedge clk, posedge rst) begin
        if (rst) begin
            WB_en=1'd0; MEM_R_en=1'd0;
            ALU_result =32'd0; Mem_read_value =32'd0;
            Dest=4'd0;
            end
        else begin
            WB_en=WB_en_in; MEM_R_en=MEM_R_en_in;
            ALU_result =ALU_result_in; Mem_read_value =Mem_read_value_in;
            Dest=Dest_in;
            end
    end

endmodule
```

*Figure 23 Memory Stage Register*

## Writeback Stage:

This stage includes a multiplexer (MUX) that determines which data to send to the register file based on the value of MEM_R_EN.

WB_STAGE.v

```
1   module WB_STAGE(input [31:0] ALU_result,MEM_result,input MEM_R_en,output [31:0] out);
2     assign out = MEM_R_en?MEM_result:ALU_result;
3   endmodule
```

*Figure 24 Writeback Stage*

## Status Register:

This register updates its state on the falling edge of the clock (clk) when S is equal to 1. The bits that are transferred include the flags for zero, negative, overflow, and carry.

status_register.v

```
1   module status_register(input clk, rst, s, input [3:0] in, output reg[3:0] out);
2       always@(negedge clk,posedge rst) begin
3           if(rst)
4               out = 4'd0;
5           else begin
6               if(s)   out = in;
7           end
8       end
9   endmodule
```

*Figure 25 Status Register*

## Hazard Detection Unit:

This module identifies data dependencies. In this module, we aim to detect read-after-write hazards, and if such a hazard is observed, we stall the pipeline. For example, in two consecutive instructions, if the second instruction depends on the result of the first instruction, we must wait until the first instruction completes before proceeding.

```
module hazard_Detection_Unit (src1,src2,Exe_Dest,Exe_WB_EN,Mem_Dest,Mem_WB_EN,Two_src,hazard_Detected);

input Two_src, Exe_WB_EN, Mem_WB_EN;
input [3:0] src1, src2, Exe_Dest, Mem_Dest;
output hazard_Detected;

 assign hazard_Detected = ((src1 == Exe_Dest) && (Exe_WB_EN == 1'b1)) ||
    ((src1 == Mem_Dest) && (Mem_WB_EN == 1'b1)) ||
    ((src2 == Exe_Dest) && (Exe_WB_EN == 1'b1) && (Two_src == 1'b1)) ||
    ((src2 == Mem_Dest) && (Mem_WB_EN == 1'b1) && (Two_src == 1'b1));

endmodule
```

*Figure 26 Hazard Unit*

## ARM Testbench:

We will connect all parts of the processor together and test it

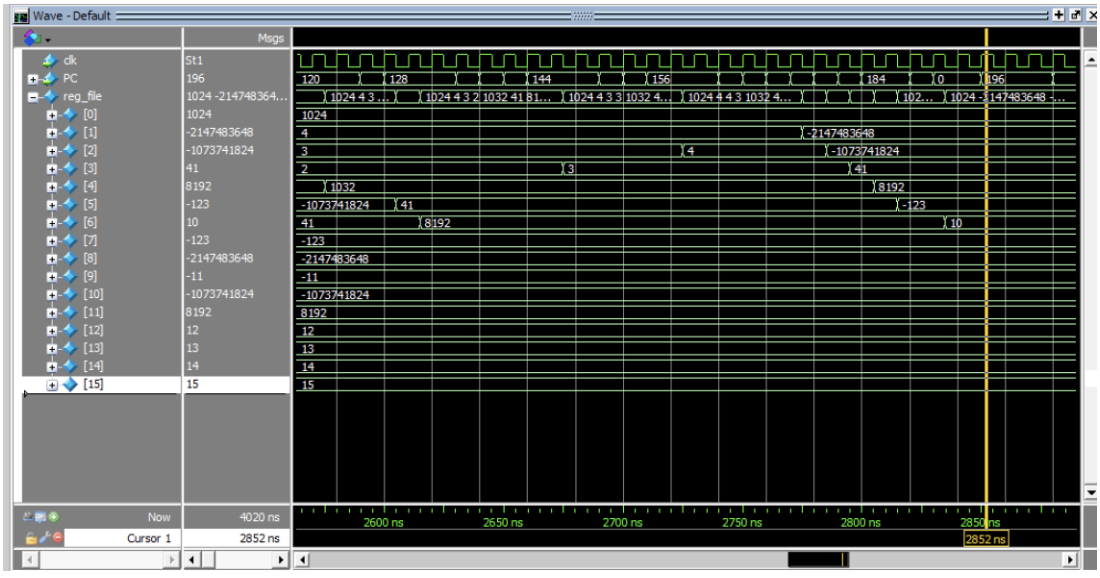
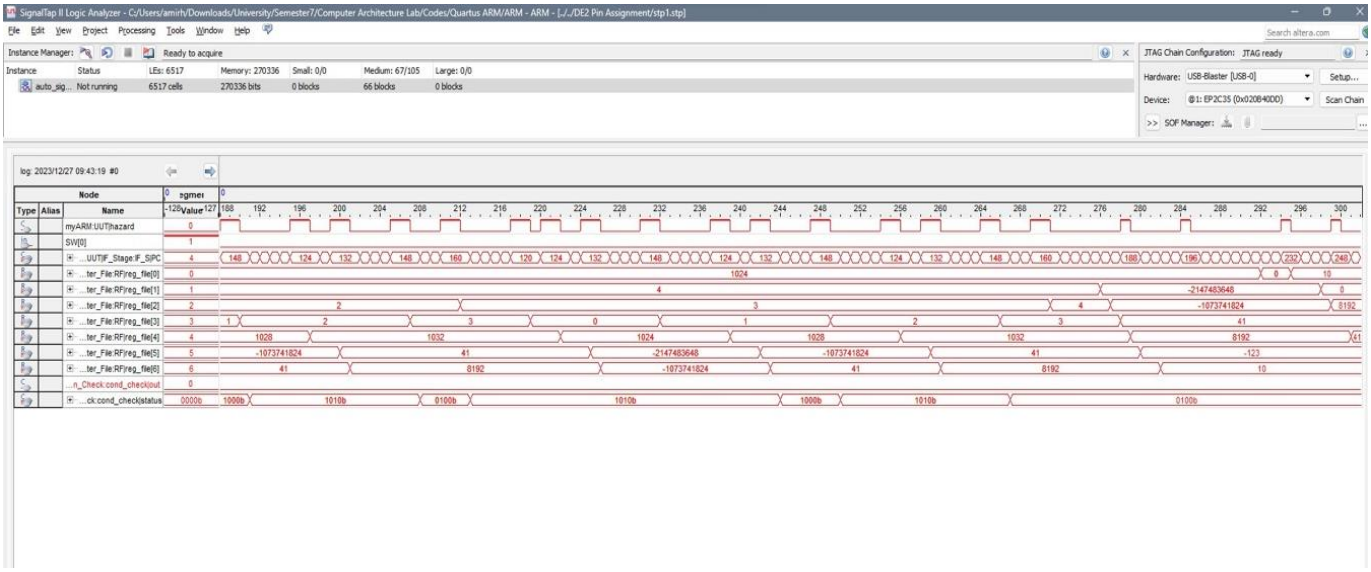



*Figure 27 Bubble sort testbench*

It is observed that the numbers have been correctly organized in the register file. (Bubble sort algorithm)

Signal Tap testbench:



*Signal Tap result28 Figure*

ARM synthesis report:

| Flow Summary | |
|---|---|
| Flow Status | Successful - Wed Dec 27 09:42:56 2023 |
| Quartus II 64-Bit Version | 13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition |
| Revision Name | ARM |
| Top-level Entity Name | ARM |
| Family | Cyclone II |
| Device | EP2C35F672C6 |
| Timing Models | Final |
| Total logic elements | 7,526 / 33,216 ( 23 % ) |
| Total combinational functions | 4,034 / 33,216 ( 12 % ) |
| Dedicated logic registers | 5,791 / 33,216 ( 17 % ) |
| Total registers | 5791 |
| Total pins | 418 / 475 ( 88 % ) |
| Total virtual pins | 0 |
| Total memory bits | 272,416 / 483,840 ( 56 % ) |
| Embedded Multiplier 9-bit elements | 0 / 70 ( 0 % ) |
| Total PLLs | 0 / 4 ( 0 % ) |

*Figure 29 Synthesis report*

# Forwarding

In the previous section, we completed the structure of the ARM processor. Due to the pipeline structure, the presence of numerous data dependencies often caused the hazard detection unit to stall the pipeline in many stages. To prevent this, we need to deliver the required data to the EXE unit earlier to avoid hazards and utilize updated data.

There are two scenarios that can create hazards and consequently stall the pipeline:

- The previous instruction is in the Memory stage, while the current instruction is in the EXE stage.
- The previous instruction is in the Write Back stage, while the current instruction is in the EXE stage.

It is important to note that in both scenarios, data dependency must exist, meaning that the instruction ahead in the pipeline affects one of the data inputs of the instruction in the EXE stage.

To address this issue, we will add a Forwarding Unit to the circuit. We will also include a fw_en input to control the activation or deactivation of this feature. The structure of this component is as follows:

```
Ln#
 1  module Forwarding_Unit (
 2      input forwarding_en,
 3      input [3:0] src1,
 4      input [3:0] src2,
 5      input WB_EN_Mem,
 6      WB_EN_WB,
 7      input [3:0] Dest_Mem,
 8      Dest_WB,
 9      output reg [1:0] sel_src1,
10      output reg [1:0] sel_src2
11  );
12    always @(src1, src2, WB_EN_Mem, WB_EN_WB, Dest_Mem, Dest_WB, forwarding_en) begin
13      if (src1 == Dest_Mem && WB_EN_Mem == 1'b1 && forwarding_en == 1'b1) sel_src1 = 2'b01;
14      else if (src1 == Dest_WB && WB_EN_WB == 1'b1 && forwarding_en == 1'b1) sel_src1 = 2'b10;
15      else sel_src1 = 2'b00;
16
17      if (src2 == Dest_Mem && WB_EN_Mem == 1'b1 && forwarding_en == 1'b1) sel_src2 = 2'b01;
18      else if (src2 == Dest_WB && WB_EN_WB == 1'b1 && forwarding_en == 1'b1) sel_src2 = 2'b10;
19      else sel_src2 = 2'b00;
20
21    end
22  endmodule
23
```

*Figure30 Forwarding unit*

The second and third inputs represent the inputs obtained from the Decode stage during ID. The mem_WB_EN input indicates whether the instruction in the Memory stage is about to update a register in the register file. The WB_wb_en input shows whether the instruction currently in the Write Back stage will update the register file.

The outputs sel_src1 and sel_src2 indicate the control inputs for the multiplexers of the ALU inputs, which control the data fed into this unit. The state 00 is the default for both multiplexers.

If the first input matches the destination register of the instruction in the Memory stage and mem_WB_en is active, it means that the instruction in the Memory stage is going to update the specified register. If forward_en is active, the updated value will replace the output from the ID stage to be sent to the ALU.

Similarly, if the first input matches the destination register of the instruction in the Write Back stage and WB_wb_en is active, it means that the instruction in the Write Back stage is going to update the register used in the instruction in the EXE stage. If FW_en is active, the

updated value will replace the output from the ID stage to be sent to the ALU. The same logic applies to the second input of the ALU.

In addition to the above changes, the Hazard Detection Unit will also undergo modifications. Given that the forwarding unit resolves some hazards, we will need to check for fewer hazards.

```
  assign hazard_wof = ((src1 == Exe_Dest) && (Exe_WB_EN == 1'b1)) ||
      ((src1 == Mem_Dest) && (Mem_WB_EN == 1'b1)) ||
      ((src2 == Exe_Dest) && (Exe_WB_EN == 1'b1) && (Two_src == 1'b1)) ||
      ((src2 == Mem_Dest) && (Mem_WB_EN == 1'b1) && (Two_src == 1'b1));

  assign hazard_f = ((EXE_MEM_R_EN) && ((src1 == Exe_Dest) || (src2 == Exe_Dest)));
  assign hazard_Detected = (fw_en) ? hazard_f : hazard_wof;

endmodule
```

*Figure 31 New hazard module*

Signal Tap testbench result is as follows:

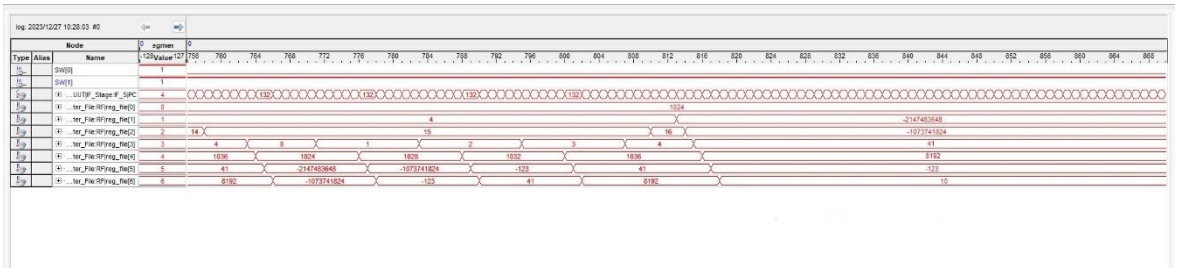


*Figure 32 Signal Tap with forwarding*

The result is the same as before proving the functionality of circuit is correct.

Without the Forwarding Unit, it took 567 clock cycles for the instructions to complete.

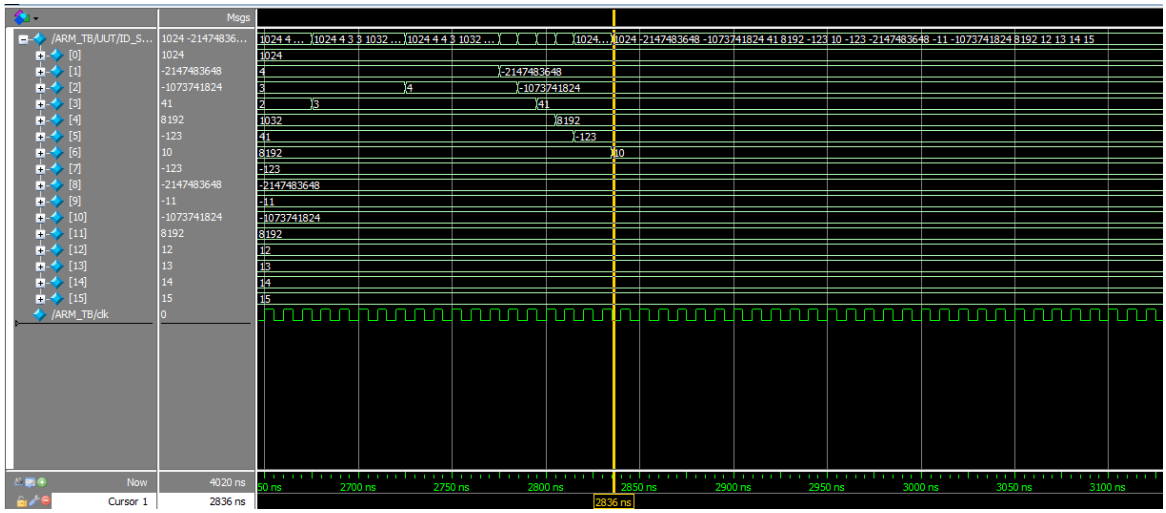



*Figure 33 Simulation with forwarding*

With the addition of the Forwarding Unit, it now takes 391 clock cycles for the processor to complete the instructions. As a result, this improvement increases the system's efficiency by 46%.

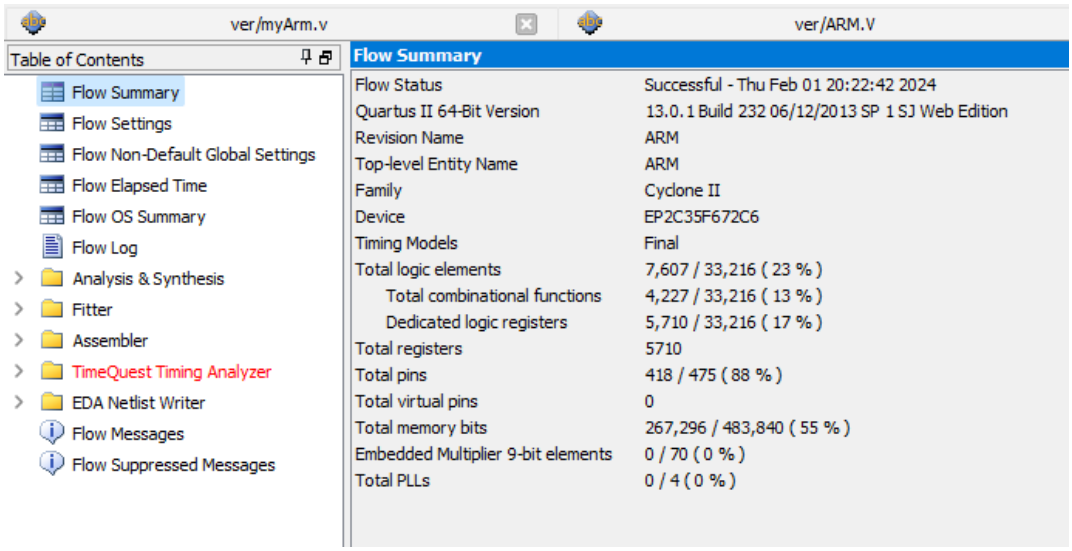



*Figure 34 synthesis report with forwarding*

# SRAM

In this section, we will design the SRAM section. Since the number of Logical Elements in an FPGA is limited, the boards allocate a portion to the memory unit. To access this memory, we need to design a controller that can correctly read the information stored in the SRAM and accurately write data to it. The figure below illustrates the controller unit of an SRAM:

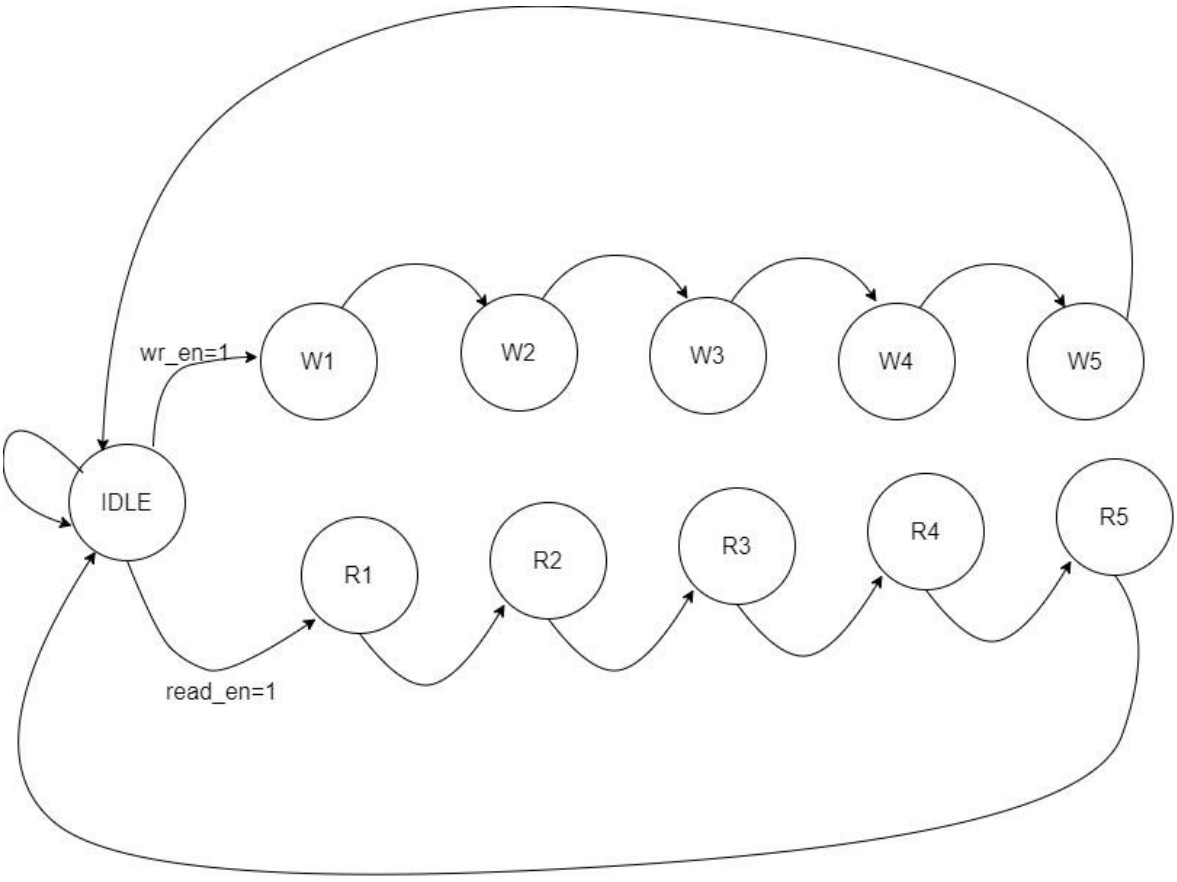



*Figure 35 Sram controller*

If the instruction is a write operation, we have the following states:

- **W1**: A 17-bit address is provided to the SRAM, with the last bit set to 0. The signal SRAM_WE_N, which is active low, is set to 0. At this address, 16 low-value bits of data are written.

- **W2**: A 17-bit address is given to the SRAM, with the last bit set to 1. The SRAM_WE_N signal remains at 0. At this address, 16 high-value bits of data are written.
- **W3**: In this state, the writing operation is completed. The next two states are merely for timing to ensure the writing and reading operations are synchronized and do not perform any actions.

If the instruction is a read operation, we have the following states:

- **R1**: In this state, a 17-bit input address is provided along with a 0 at the end as the address for the SRAM.
- **R2**: In this state, we retain the previous address and read the input data. This data consists of the 16 low-value bits of the desired 32-bit data, which is stored in the read_data register within the controller.
- **R3**: In this state, we provide the address of the next memory location to the SRAM to read the 16 high-value bits of the desired data.
- **R4**: In this state, we simply wait for one clock cycle to ensure that the input data is stored correctly.
- **R5**: In this state, we read the 16 high-value bits of the desired data, completing the reading operation.

```
1   module Sram_Controller (
2       input clk,
3       input rst,
4
5       input wr_en,
6       input rd_en,
7       input [31:0] address,
8       input [31:0] writeData,
9
10      output reg [31:0] readData,
11
12      output ready,
13
14      inout [15:0] SRAM_DQ,
15      output reg [17:0] SRAM_ADDR,
16      output SRAM_UB_N,
17      output SRAM_LB_N,
18      output reg SRAM_WE_N,
19      output SRAM_CE_N,
20      output SRAM_OE_N
21  );
```

*Figure 36 Sram controller interface*

The following images illustrate the design of the SRAM controller:

```
22      reg [3:0] ns, ps;
23      assign SRAM_UB_N = 1'b0;
24      assign SRAM_LB_N = 1'b0;
25      assign SRAM_CE_N = 1'b0;
26      assign SRAM_OE_N = 1'b0;
27      localparam [3:0] idle = 0, W1 = 1, W2 = 2, W3 = 3, W4 = 4, W5 = 5,
28                       R1 = 6, R2 = 7, R3 = 8, R4 = 9, R5 = 10,R6=11;
29      always @(ps, wr_en, rd_en) begin
30        ns = idle;
31        case (ps)
32          idle: begin
33            if (wr_en == 1'b1) ns = W1;
34            else if (rd_en == 1'b1) ns = R1;
35            else ns = idle;
36          end
37
38          W1: ns = W2;
39          W2: ns = W3;
40          W3: ns = W4;
41          W4: ns = W5;
42          W5: ns = idle;
43
44          R1: ns = R2;
45          R2: ns = R3;
46          R3: ns = R4;
47          R4: ns = R5;
48          R5: ns = R6;
49          R6: ns = idle;
50
51        endcase
52
53      end
```

*Figure 37 Sram controller code*

```
55      always @(ps) begin
56        SRAM_WE_N = 1'b1;
57
58        case (ps)
59
60          W1: begin
61            SRAM_ADDR = {address[18:2], 1'b0};
62            SRAM_WE_N = 1'b0;
63          end
64          W2: begin
65            SRAM_ADDR = {address[18:2], 1'b1};
66            SRAM_WE_N = 1'b0;
67          end
68          W3: begin
69            SRAM_WE_N = 1'b1;
70          end
71          R1: begin
72            SRAM_ADDR = {address[18:2], 1'b0};
73          end
74          R2: begin
75            SRAM_ADDR = {address[18:2], 1'b0};
76            readData[15:0] = SRAM_DQ;
77          end
78          R3: begin
79            SRAM_ADDR = {address[18:2], 1'b1};
80          end
81          R5: begin
82            SRAM_ADDR = {address[18:2], 1'b1};
83            readData[31:16] = SRAM_DQ;
84          end
85          default: begin
86            SRAM_WE_N = 1'b1;
87          end
88        endcase
89      end
90
91      always @(posedge clk, posedge rst) begin
92        if (rst) ps <= idle;
93        else ps <= ns;
94      end
95      assign SRAM_DQ = (ps == W1) ? writeData[15:0] : (ps == W2) ? writeData[31:16] : 16'bz;
96      assign ready   = (ns == idle) ? 1'b1 : 1'b0;
97    endmodule
```

*Figure 38 Sram controller code*

In the next section, we will replace the memory with the controller unit and test the design on the board:

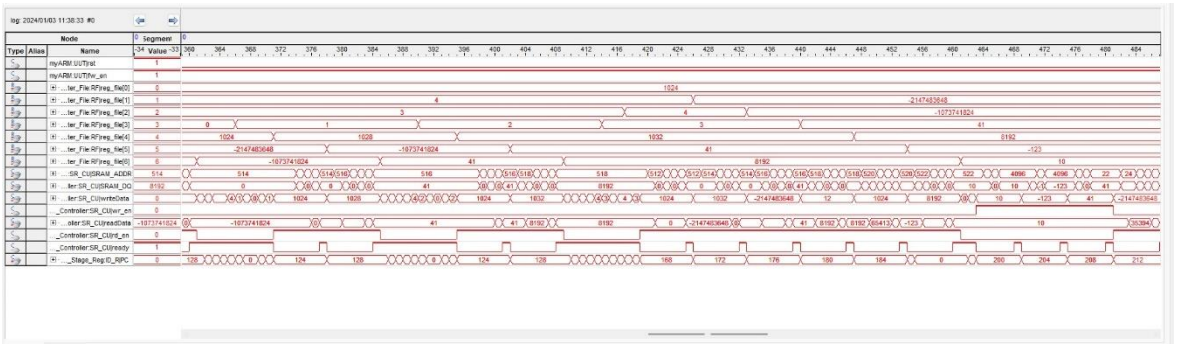


*Figure 39 Signal tap testbench*