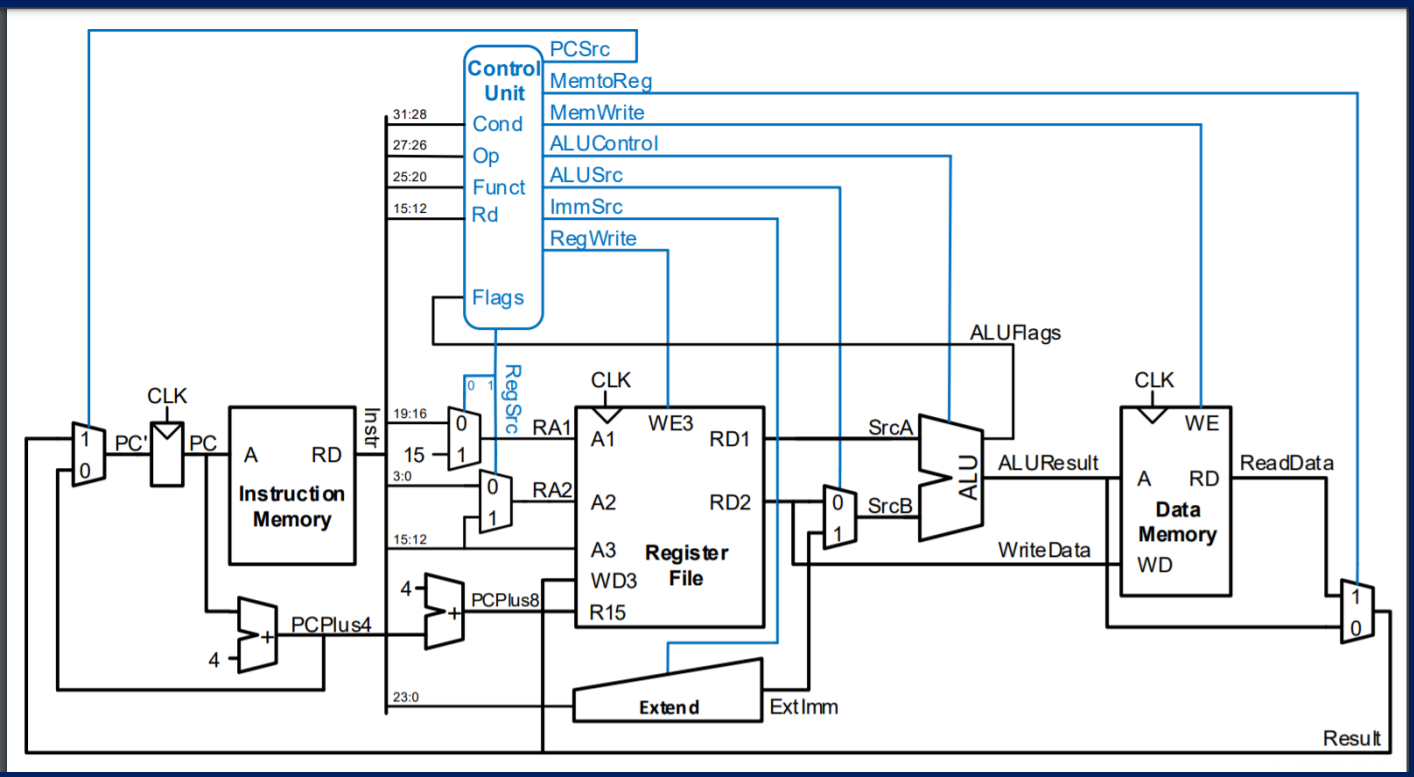
ARM Computer Architecture Lab

Start Oct 1, 2024 | End Jan 22, 2025



ARM Computer Architecture LAB

Report for computer architecture lab in which we developed ARM CPU from scratch.

Table of Contents

01

Instruction Fetch (IF)

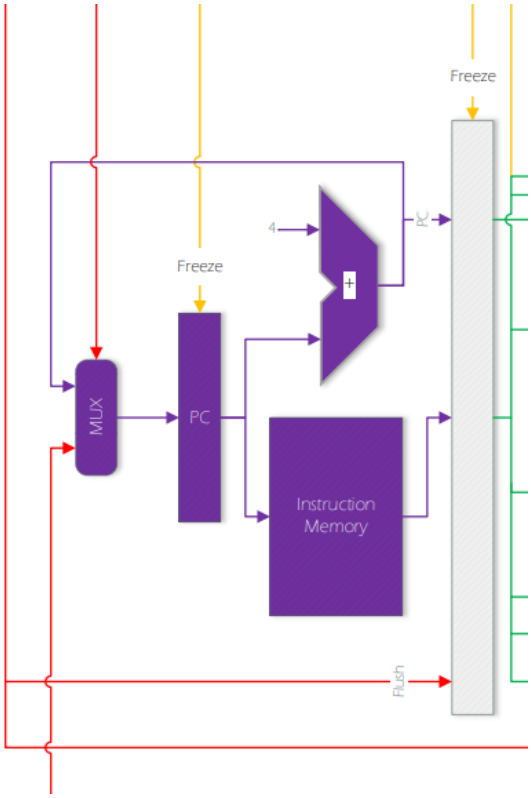


Fig.1 IF of ARM DP

This stage has responsibility to read instructions from memory and send them to ID stage for decoding

Overview

It has four prime modules that handle: location of the last command address that was fetched, “branch” if the branch signal is issued. This module sends the raw instruction to the next stage for further process and execution.

Key features

This module should handle branch and freeze when we have data hazards. In case of a branch we should ignore the hazard signal to prevent stalling the pipeline of the CPU. This feature is handled using code below.

```
register pc(clk , rst, (~freeze) | (branchTaken), PCIn, PCOut);
```

The yellow part is the load signal of PC register which is depicted in fig. 1. This logic ensures that if a branch happens the branched address will be loaded in the PC register.

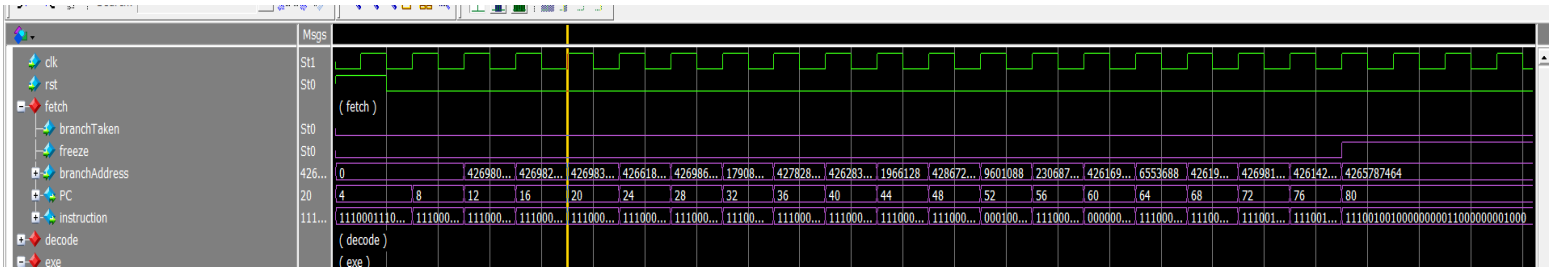


Fig.2. Simulation of IF

‘

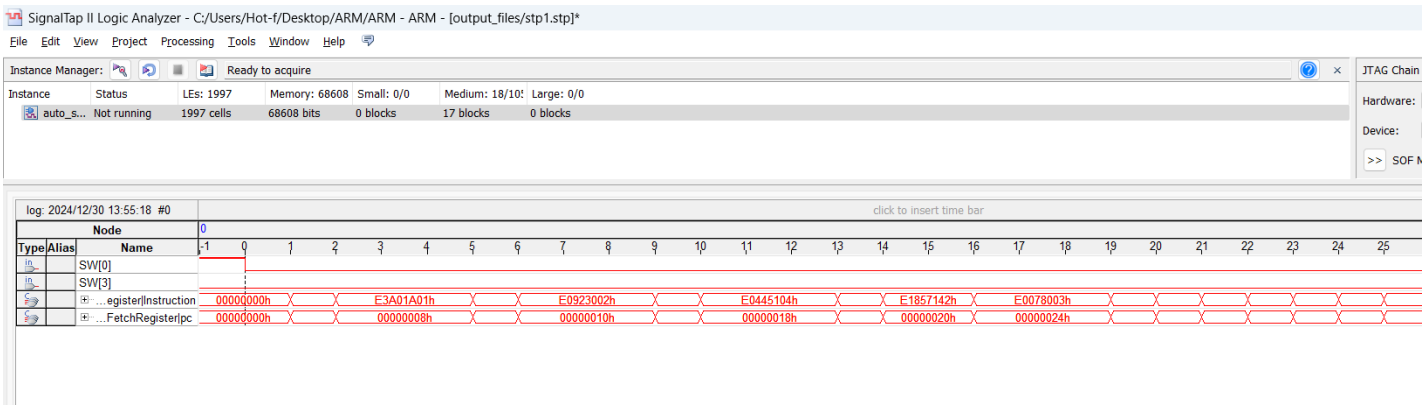


Fig.3. Signal tab of IF

Module-by-module code

There will be a short definition of each module code that is used here.

Instruction memory

This module is a mock of real instruction memory that is implemented using switch-case for simulating the existing program to compare results by adding features e.g. cache. The interface has shown below:

```
module InstructionMemory(   input [31:0] memAddr,
                            output reg [31:0] mem);
```

Register module

This module has Pin, Pout, Load, rst, clk, and flush (RegisterWithFlush) as IO ports. Also this module is parametrized to facilitate reuse. If the load signal is one at the posedge of clk the register value will be the amount of Pin. The module interface is shown below.

```
module register #(parameter WIDTH = 32) (
    input clk,
    input rst,
    input loaden,
    input [WIDTH-1:0] pin,
    output [WIDTH-1:0] pout
);

module registerWithFlush#(parameter WIDTH = 32) (
    input clk,
    input rst, flush,
    input loaden,
    input [WIDTH-1:0] pin,
    output [WIDTH-1:0] pout
);
```

ProgramCounter

This module has the responsibility of incrementing the pc register value by four. The interface is shown below.

```
module ProgramCounter(input [31:0] pcIn , output[31:0] pcOut);
```

mux2to1

This module as everyone knows is policing one of two inputs at the output depending on its select whether be zero it puts first one and if the select is one it puts second one.

```
module Mux2to1( input select,
                input[31:0] in0,
                in1,
                output [31:0] out);
    assign out = (select==1'b1)?in1:in0;
endmodule
```

Input/Output

Output:
As mentioned before this module extracts instructions from memory, increments pc Value By four and sends them to the next stage through the pipe register. So output of this module is two 32 bit values for pc and instruction.
Input:
The module inputs are: branch address, branch signal, and freeze signal. The freeze signal is received in case of hazard and stall pc register and IF-register that belong to pipe registers. Flush signal is to makes value of IF-register zero in case of branch

02

Instruction Decode(ID)

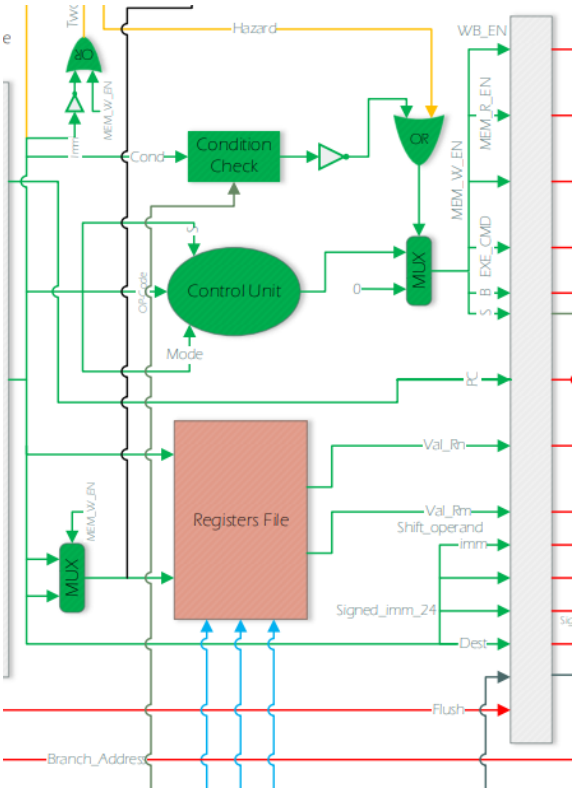


Fig.4 ID of arm dp

This stage decodes the instruction. For this purpose we implemented three main elements alongside some other primitive elements to decode the instruction fetched from the previous layer and send it to the execution phase.

Overview

Instruction decode consists of 3 main modules:
Condition Check, Control unit and Register File, which are discussed below.

Key features

This stage plays a crucial role in handling the hazard signal. All control signals are assigned to zero in case of hazard detection(Note that hazard signal is.

```
assign controlUnitCheck = (~condtionChechOut) | freeze;
```

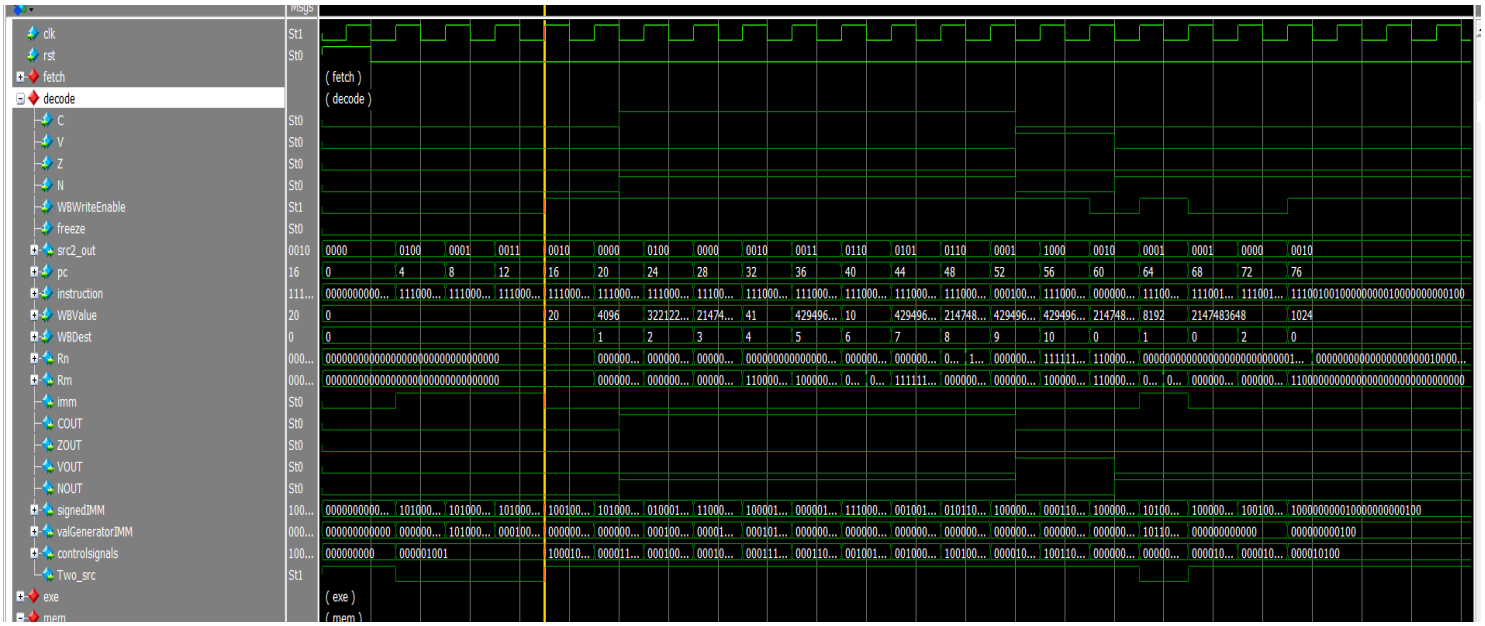


Fig.5. Simulation of ID

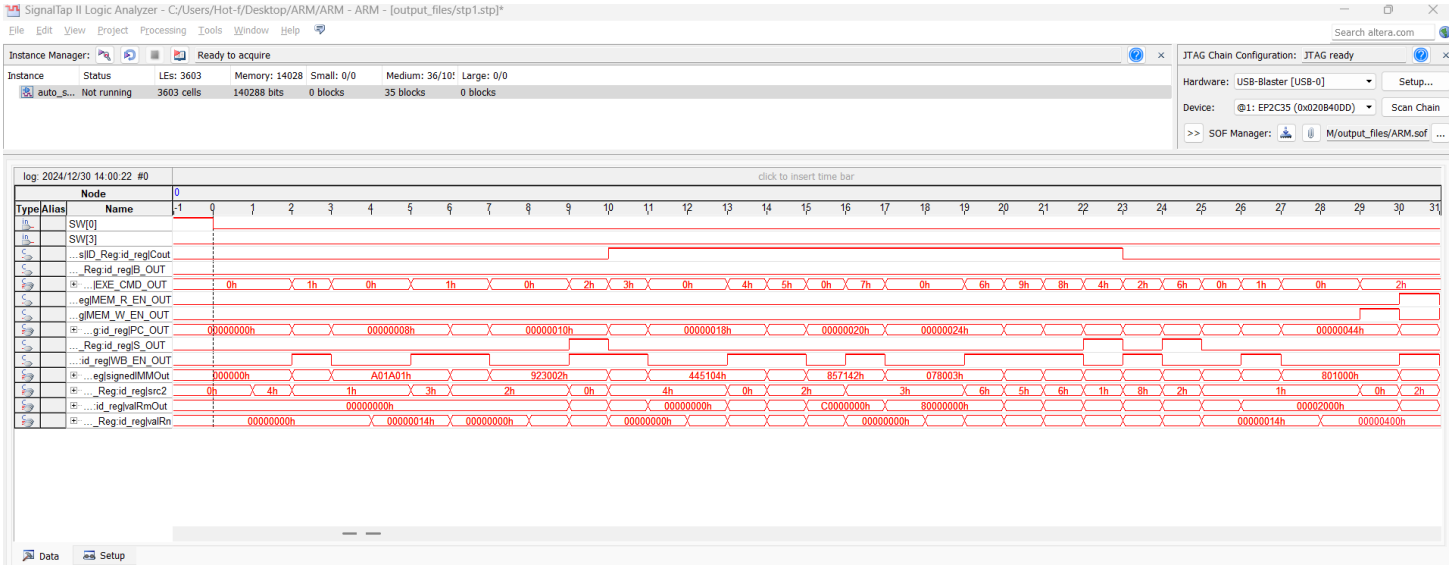


Fig.6. Signal tab of ID

Module-by-module code

Register File

Register File is a simple memory consisting of 16 registers each consisting of 32 bit memory blocks. Read and Write operations are synchronous with clock, but issued on negative edge to avoid hazards relating to WAR(write after read). For read, we implemented to sources (annoted with src1 and src2) which refer to register file addresses and for write a writeBackEn signal is implemented which writes the data in the block corresponding to destWB address.

```
module RegisterFile(input clk, rst,
                    input[3:0] src1, src2, destWB,
                    input[31:0] resultWB,
                    input writeBackEn,
                    output reg [31:0] reg1, reg2);
```

Control Unit

This Unit does a crucial job when it comes to handling the instruction in the ARM architecture. Control signals, including StatusBit (SIn and SOut), EXECommand (which controls the ALU), and memory control signals are generated in the section using the instruction signals acquired from the previous stage. All signals are handled according to the document given in the course description.

```
module ControlUnit( input[3:0] opCodeIn,
                    input[1:0] modeIn,
                    input SIn,
                    output [8:0] out);
```

Condition Check

This module is responsible for checking the condition for conditional instructions using a given condition code and Status bits. A flag is then generated according to condition and given to a 2to1 mux to select between control unit signals and zero. If the conditions are not met or in case of hazard detection, all output signals are mapped to zero to avoid incorrect instruction propagating to the execution stage.

```
module CondtionCheck(input[3:0] cond, input z, c, n, v, output reg flag);
```

Other components

Instruction Decode stage includes a few other components which are not mentioned above, such as an OR gate to detect if the instruction requires two sources or not, this signal is then given to the hazard unit to detect hazard accordingly. We used a multiplexer in src2 input of the register file since instructions such as STR use different bits of the instruction as src2. Also, an OR and a multiplexer were needed to enforce condition check in output registers, which we explained in the previous section.

03

Execution(exe)

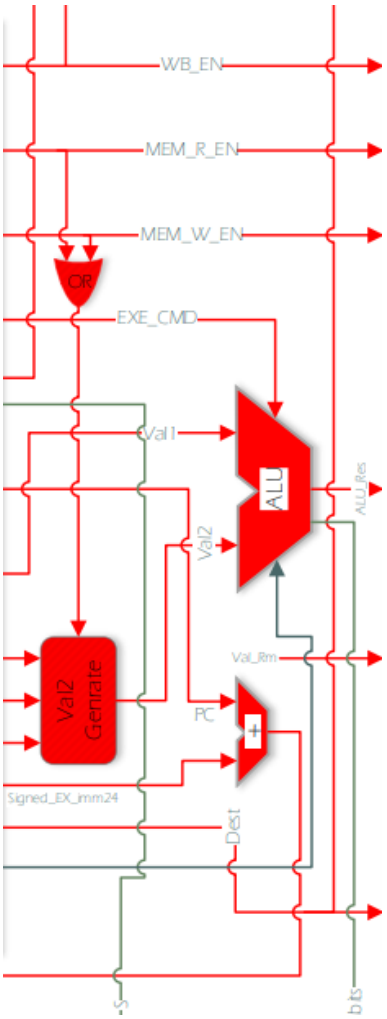


Fig.7 ID of arm dp

This stage has responsibility to Execute instruction, Adding PC by possible branch value.

Overview

One of this module's responsibilities is to do calculations, for example multiplying elements, therefore it needs an arithmetic logic unit (ALU). The second input of ALU will

be determined from val2Genetor module outputs which depend on the instruction type and it could be immediate data, shifted second value , etc.(after adding forward we should add muxes because of adding the forwarded data to this). This stage also calculates the sum of PC value by the possible amount of branch if the condition check determines that branch is happening so this value will be loaded into pc reg.

Key features

The Or that shown in the image has responsibility to control the val2Generator and its select of one of its own muxes .

```
assign memCommand=MEM_R_EN|MEM_W_EN;
```

This assignment handles the mentioned part.

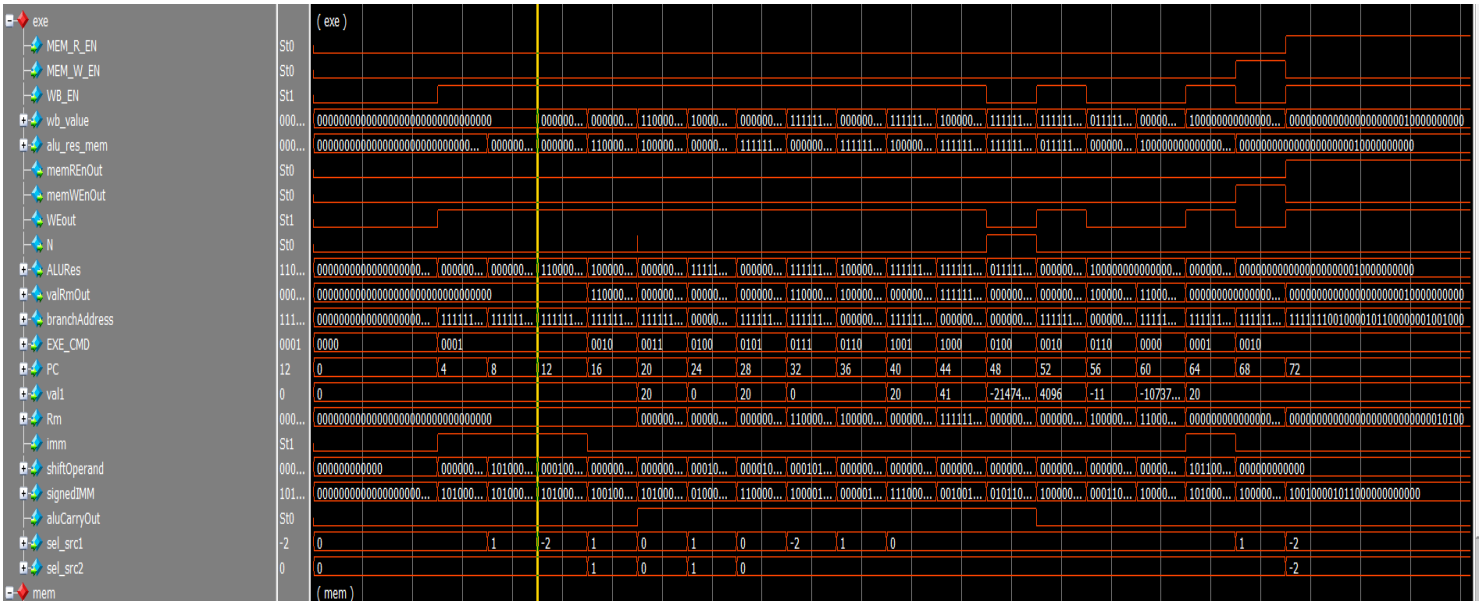


Fig.8. Simulation of ID

Fig.9. Signal tab of EXE

Module-by-module code

There will be a short definition of each module code that is used here.

Arithmetic Logic Unit (ALU)

This module calculates the result of the operation. Output of this module is the result of instruction, it may be written in memory or register file. Also it updates the values of status register.

```
module ALU(input[31:0] val1, val2,
           input[3:0]cmd,
           input carryIn,
           output [31:0] out,
           output reg N,
           output carryOut);
```

Program Counter Adder

This line of behavioral code acts like the adder that illustrated in

```
assign branchAddress=PC+{{6{signedIMM[23]}}, signedIMM, 2'b00};
```

ProgramCounter

This module has the responsibility of incrementing the pc register value by four. The interface is shown below.

```
module ProgramCounter(input [31:0] pcIn , output[31:0] pcOut);
```

Val2Generator

This module generates the second input of ALU because of its unique shifting, the whole code has been brought here. The number may be shifted circular by amount of shiftOprand.

```
module ValGenerator(input [31:0]Rm, input imm, memCommand, input [11:0] shiftOprand ,
output reg [31:0] ALUVal2);
 integer i;
    wire [7:0] immed_8;
    wire[3:0] rotate_imm;
    assign immed_8 = shiftOprand[7:0];
    assign rotate_imm = shiftOprand[11:8];

    always @(*)
    begin
    if (memCommand)
        ALUVal2 = {{20{shiftOprand[11]}}, shiftOprand};
    else
        if(imm)
        begin
           ALUVal2 = {24'b0, immed_8};
              for (i = 0; i < 2 * rotate_imm; i = i + 1) begin
                  ALUVal2 = {ALUVal2[0], ALUVal2[31:1]};
              end
        end
        else
        begin
            case (shiftOprand[6:5])
            2'b00:
            begin
                ALUVal2 = Rm << shiftOprand[11:7];
            end
            2'b01:
            begin
                ALUVal2 = Rm >>shiftOprand[11:7];
            end
            2'b10:
            begin
                ALUVal2 = $signed(Rm) >>> shiftOprand[11:7];
            end
```

```
            2'b11:
            begin
                ALUVal2 = Rm;
                    for (i = 0; i < shiftOprand[11:7]; i = i + 1)
                    begin
                        ALUVal2 = {ALUVal2[0], ALUVal2[31:1]};
                    end
            end
            endcase
        end
    end
endmodule
```

The Overall interface

This is overall interface after adding forward unit

```
module EXE(input clk, rest, MEM_R_EN, MEM_W_EN, WB_EN,
            input[3:0] EXE_CMD,
            input [31:0] PC,
            input [31:0] val1, Rm,
            input imm,
            input[11:0] shiftOperand,
            input[23:0] signedIMM,
            input aluCarryOut,
            input[1:0] sel_src1, sel_src2,
            input [31:0]wb_value,alu_res_mem,
            output memREnOut, memWEnOut, WEout,N,
            output[31:0] ALURes, valRmOut, branchAddress,
            output statusBits);
```

Muxes

These two mux are needed in case of using forwarding and it will be describe in forwarding section

```
Mux3to1 MUX_1(sel_src1,val1,alu_res_mem, wb_value, mux1_out);
    Mux3to1 MUX_2(sel_src2,Rm,alu_res_mem, wb_value, mux2_out);
```

IN-OUTPUTS:

This module drives ALU result, Val_RM, reads, and writes signals (write back enable of register file)

04

Memory

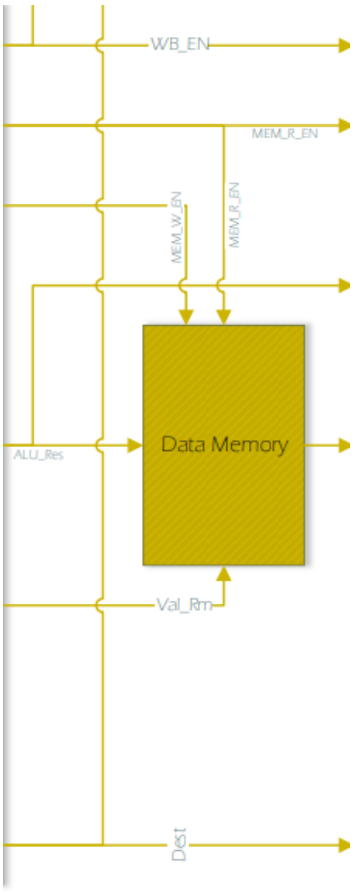


Fig.10. ID of arm dp

At this stage reading from memory is handled.

Overview

As it is obvious the only element of this phase is Data memory.

Key features

The address of data memory is ALU result which is passed through the pipeline register and memory data is Val_Rm. This phase has three implementations for simulating using logic elements of FPGA, using SRAM chip of FPGA, and cache, they will be explained later in this paper.

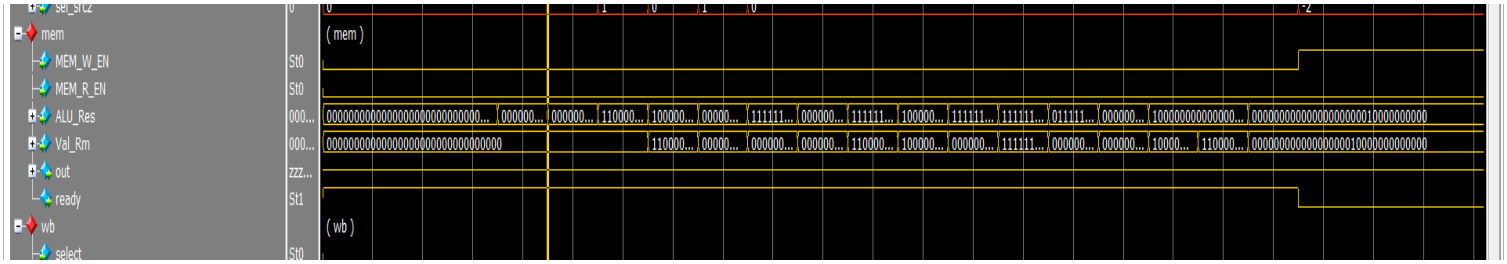


Fig.11. Simulation of ID

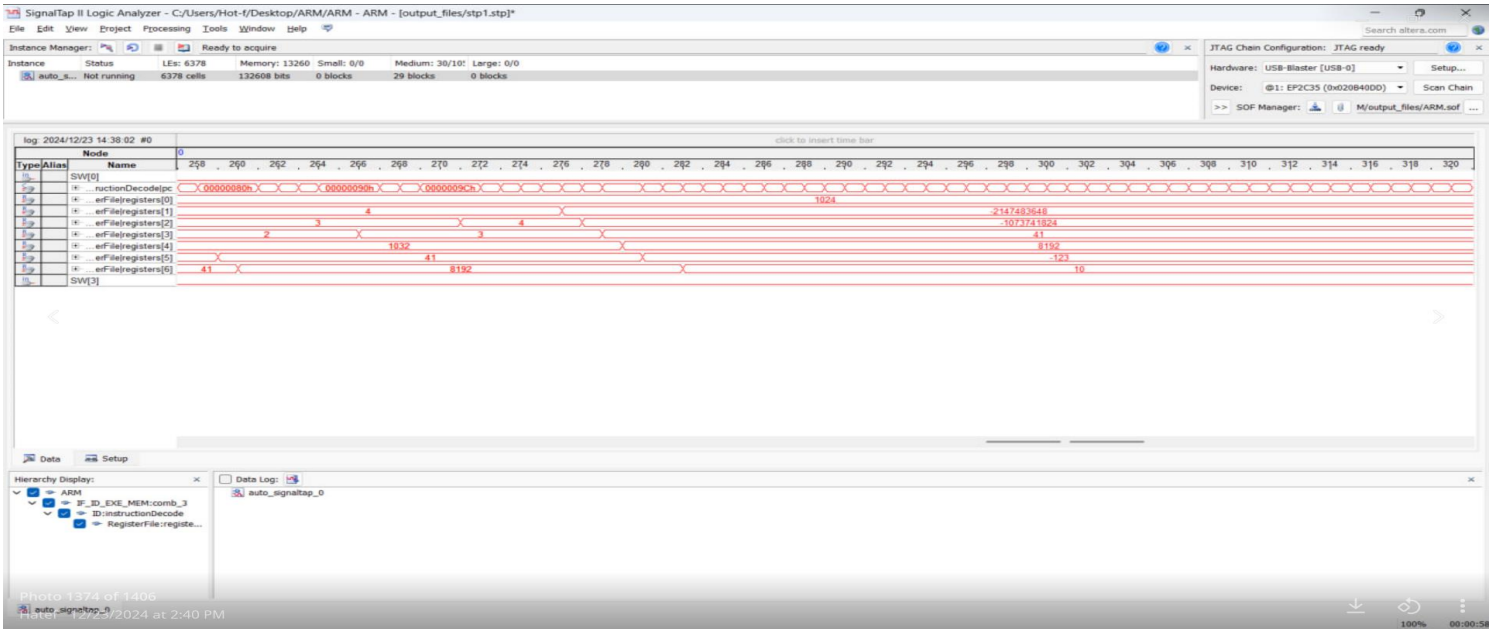


Fig.12. Signal tab of Mem

Module-by-module code

There will be a short definition of each module code that is used here.

Data Memory

Nothing to say about it, mem is meming

```
module DataMemory( input clk , rst , MEM_W_EN, MEM_R_EN,
                    input[31:0] ALU_Res, Val_Rm,
                    output [31:0] out,
                    output ready , output [15:0]SRAM_DQ, output [17:0] SRAM_ADDR,
                    output SRAM_UB_N, SRAM_LB_N, SRAM_CE_N, SRAM_OE_N,
                    output SRAM_WE_N);
```

The last 7 outputs are related to SRAM.

05

Write Back

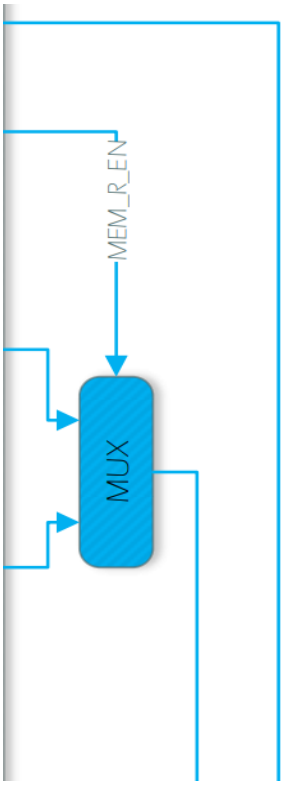


Fig.13 ID of arm dp

This little mux is to choose which data is to write on the register file.

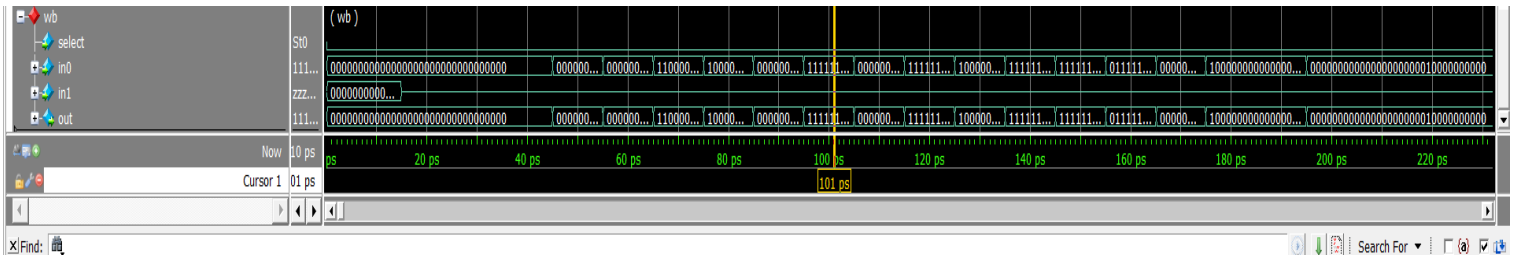


Fig.15. Simulation of write back

06

Hazard Unit

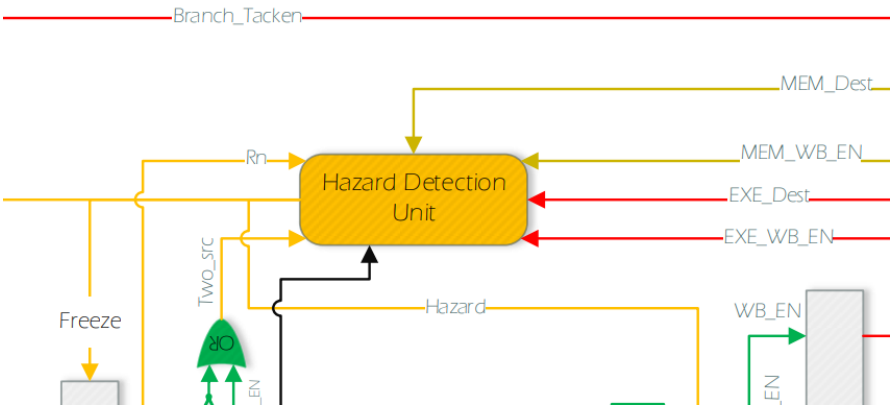


Fig.14 Hazard in DF

This unit detects Read after Write hazards, which could happen in four conditions before implementing Forwarding but reduced to only 2 conditions afterwards.

Overview

This is not a stage but a simple unit that detects RAW hazards. Hazard unit operates in 2 modes:

1. If the forward unit is present in the pipeline, it checks the read operation to not occur immediately after a write operation since memory has a read and write delay which results in wrong output when these two operation happen simultaneously.
2. In case we don't have a forwarding unit, incorrect write operation can occur when either of MEM or EXE operations need to write in a destination which is read in later instructions, therefore we need to include not only EXE destination (which is the only consideration when forwarding is present) but MEM destination as well when deciding to output a freeze signal.

Key features

Detecting Hazards is an important task in implementing the architecture. This unit ensures data consistency across different stages and sends the freeze signal to stop the ID register, PC register and change IF output to zero to create a bubble and stop the incorrect instruction from propagating through the pipeline.

```
module Hazard(  input MEM_WB_EN, EXE_WB_EN, Two_src,
                input[3:0] MEM_Dest, EXE_Dest,
                input[3:0] Rn, Rm,
                input EXE_MEM_R_EN,forward_en,
                output reg freeze);
```

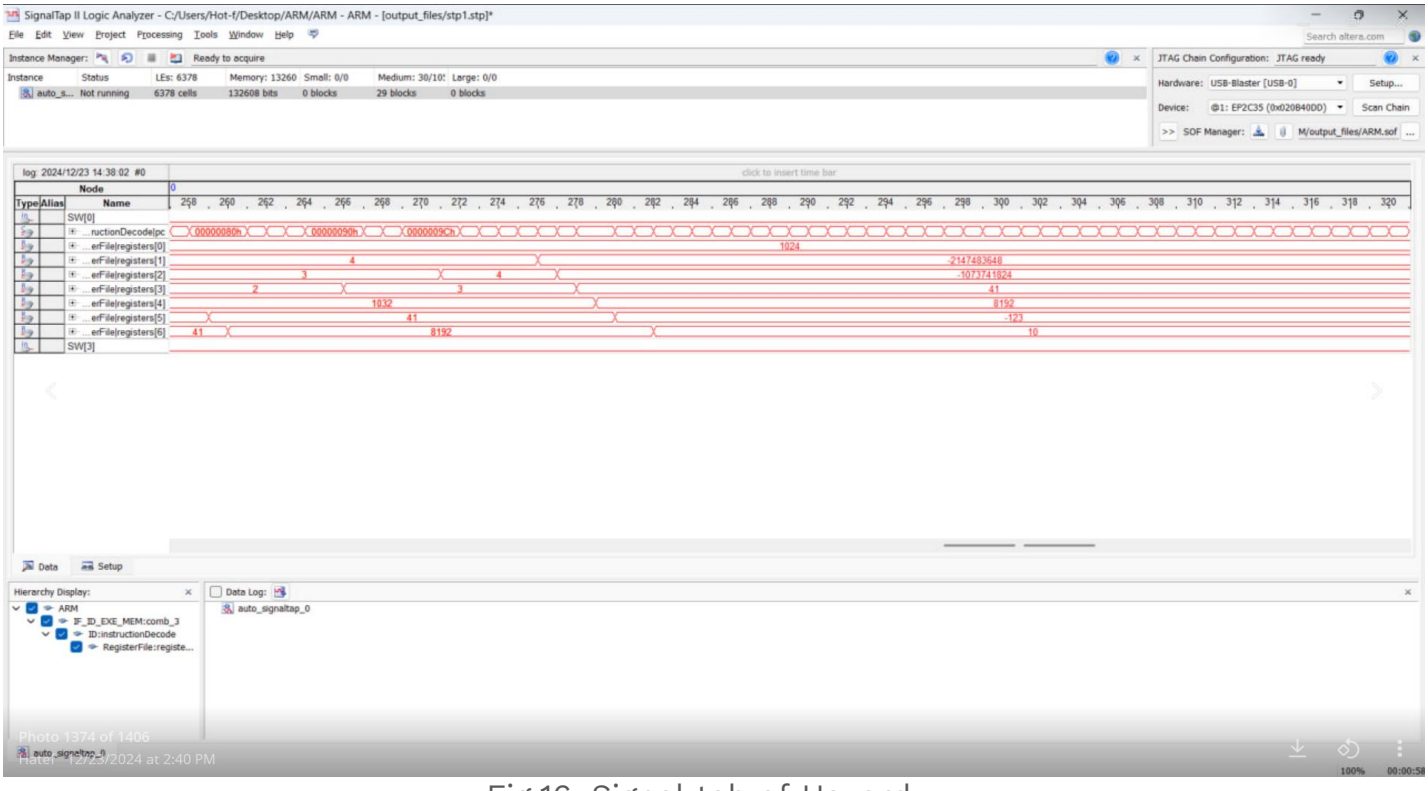


Fig.16. Signal tab of Hazard

07

Forwarding Unit

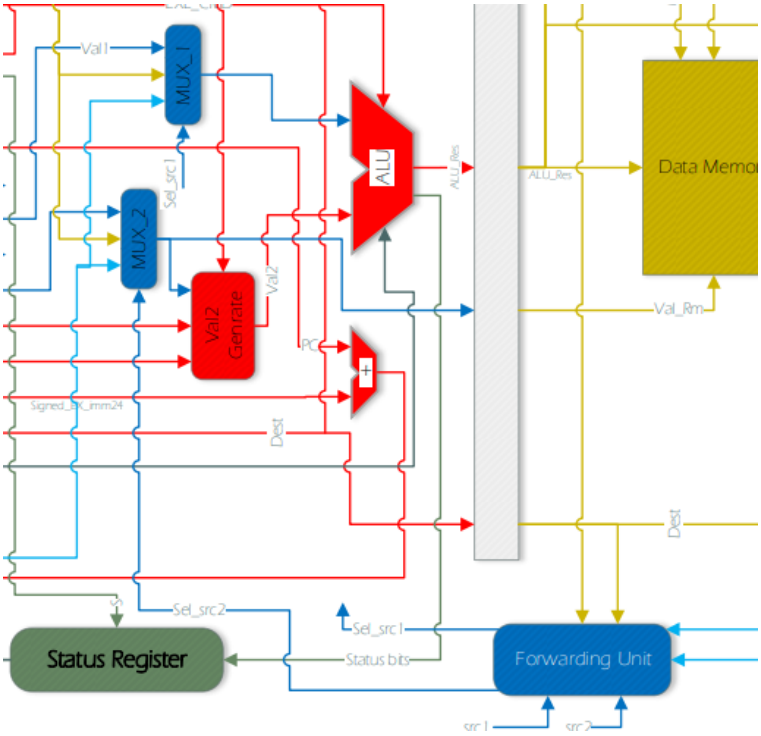


Fig.17 Forward in DF

In this part of the experiment, we added a Forwarding unit to decrease the amount of total waiting cycles for results to be propagated through the pipeline.

Overview

As discussed in the hazard unit section, one of the problematic parts of implementing the pipeline was detecting hazards and freezing the processor to wait for valid value to be written in the registers. By adding the forwarding unit, we took advantage of the valid values in the pipeline and instead of waiting for an I/O operation, we forwarded the values

directly into the Execution stage, reducing the number of hazards. The Hazard unit also underwent some changes to adapt this new design, which was explained in detail in the previous section. The Forwarding unit forwards data from Memory and WriteBack stages to Execution whenever the source of an instruction is present in those stages.

Key features

This unit has a simple structure but plays a crucial role in implementing an efficient pipeline. One of 2 types of hazards (explained in Hazard Unit) have been eliminated by this unit and only type of hazard remaining is when an instruction requires a memory output which is not ready since the memory block has read delay and pipeline must be stalled until memory operation is done. Another change that has been made while implementing the design on board was decreasing the clock speed to overcome the clock cycle issue which occurred as a result of overhead added to stages by the Forwarding unit. This issue was handled simply by adding a frequency divider on the clock signal. As shown below, adding a Forwarding unit resulted in a 1.4 speed up in overall architecture.

```
module ForwardingUnit(  input[3:0] src1, src2,
                        input[3:0] mem_Dest, wb_Dest,
                        input mem_WB_EN, wb_WB_EN,
                        output reg [1:0] sel_src1, sel_src2);
```

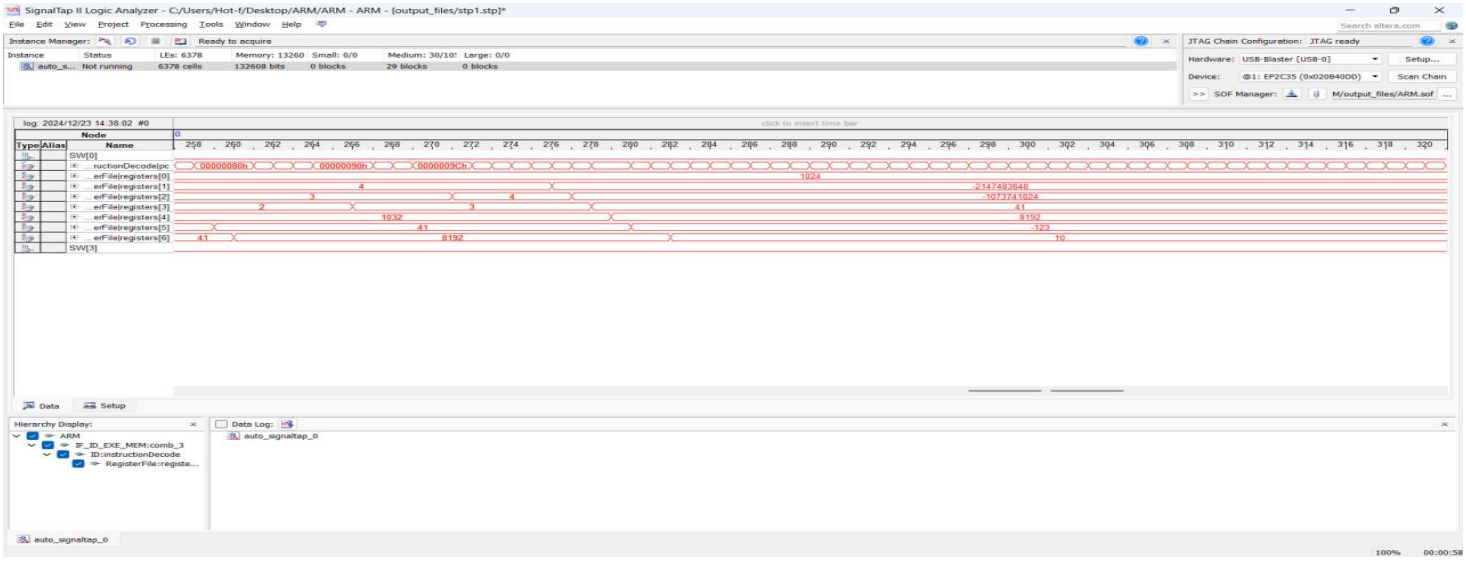


Fig.18. Without forwarding

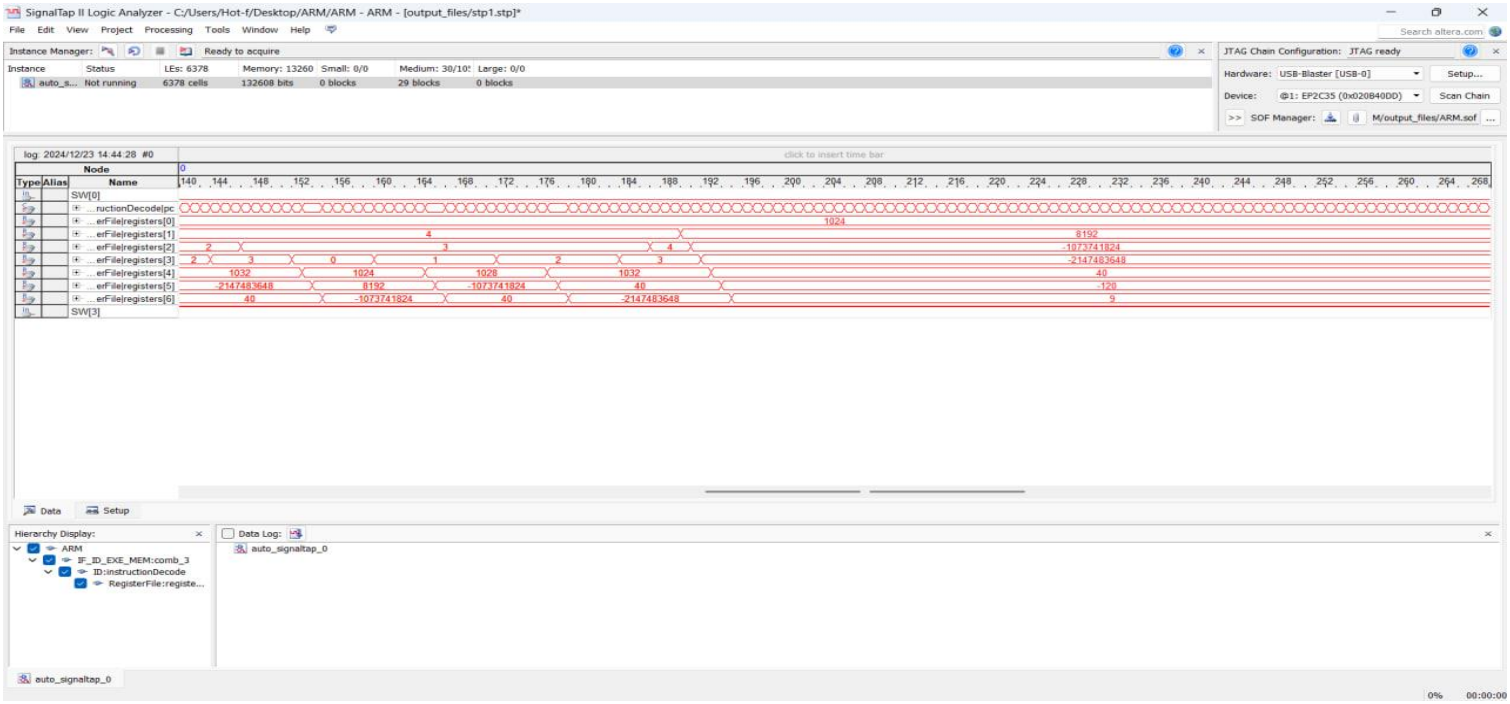


Fig.6. With forwarding, the process could have been completed sooner. However, due to the higher combinational delay, it was unable to compute the final number. Therefore, we need to add a frequency divider.

After adding frequency divider:

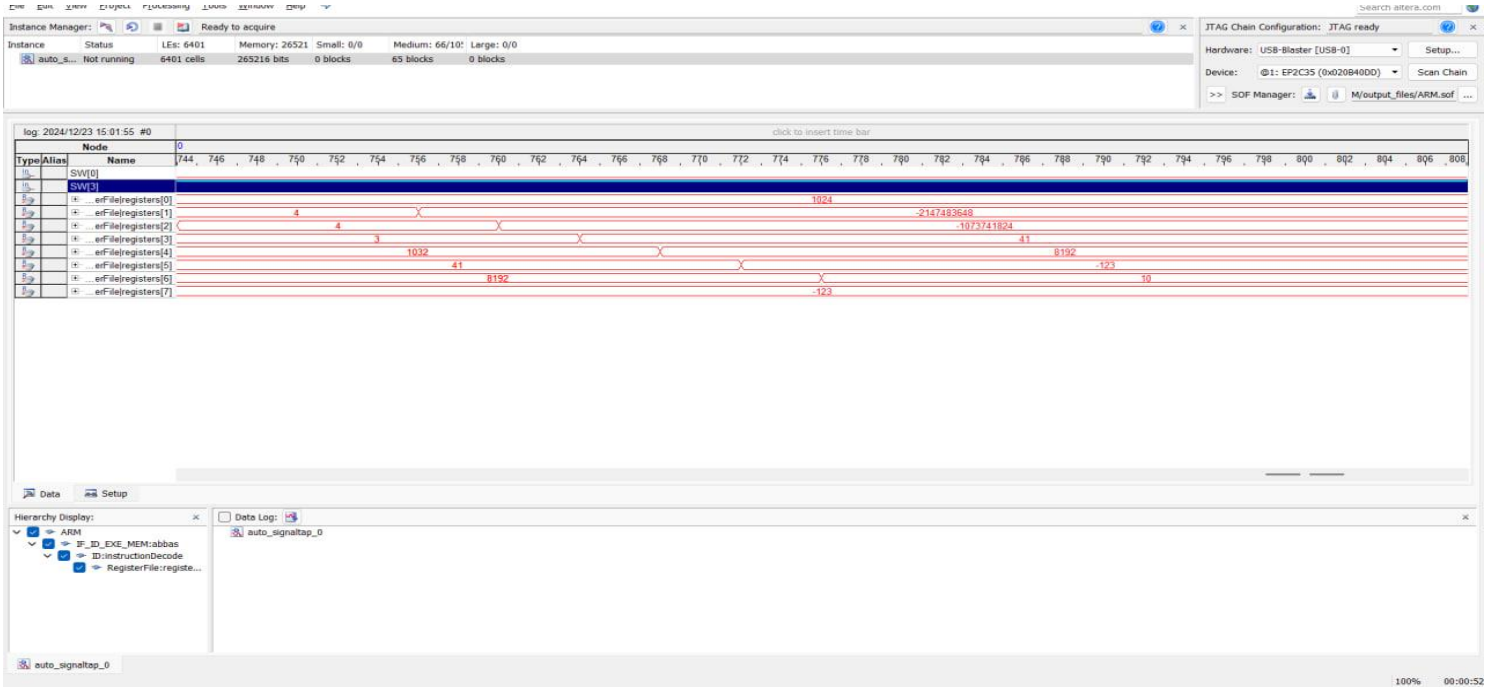


fig.19. Forwarding with adding frequency divider

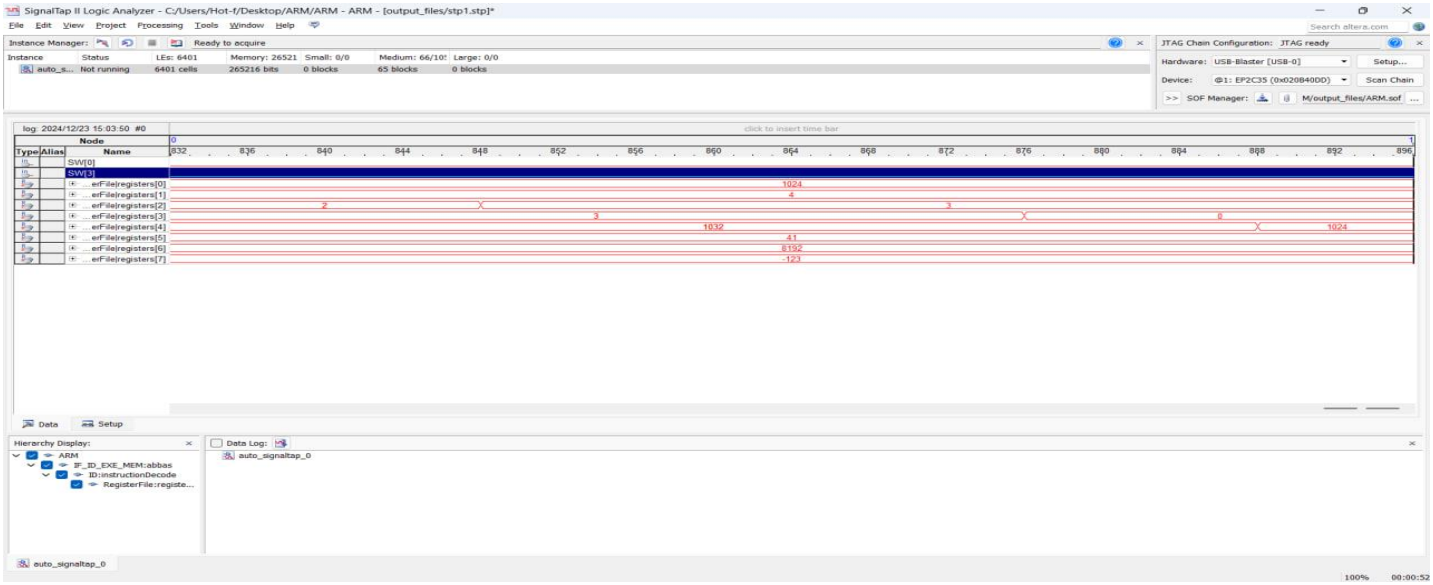


Fig .20. Without forwarding using frequency divider (it hasnt finish yet)

So by adding forward unit we have 282/194= 1.4 speed up

07

SRAM

Using SRAM chip of fpga instead of its logic element to synthesizing DataMemory stage more realistically

Overview

In this part, we use an actual memory chip that is slower but has a larger capacity. This chip is the SRAM chip of the Altera Cyclone-II FPGA, provided by the lab. The chip has an input/output bus called **SRAM_DQ**, which can transfer 16-bit data either into or out of the memory. Therefore, reading or writing a word requires more than one clock cycle. Due to synchronous read operations, the address must be set and held to read the first two bytes (one word) in the second clock cycle. Consequently, six clock cycles are needed to perform a complete read or write operation. (While fewer states could be used for implementation, six clock cycles are chosen to facilitate the integration of a cache later). The code will clarify the concept better.

Key features

In case of reading data from sram we need a controller to handle responsibility of handshaking with the chip.

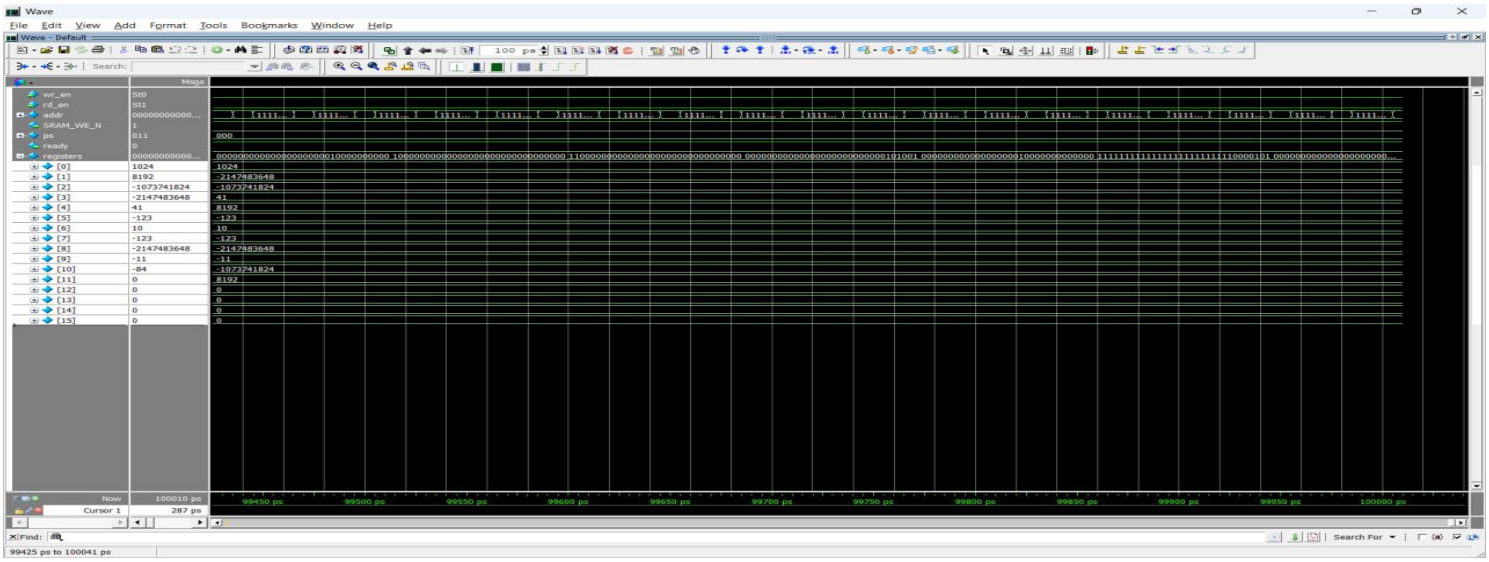


Fig.21. Simulation of SRAM

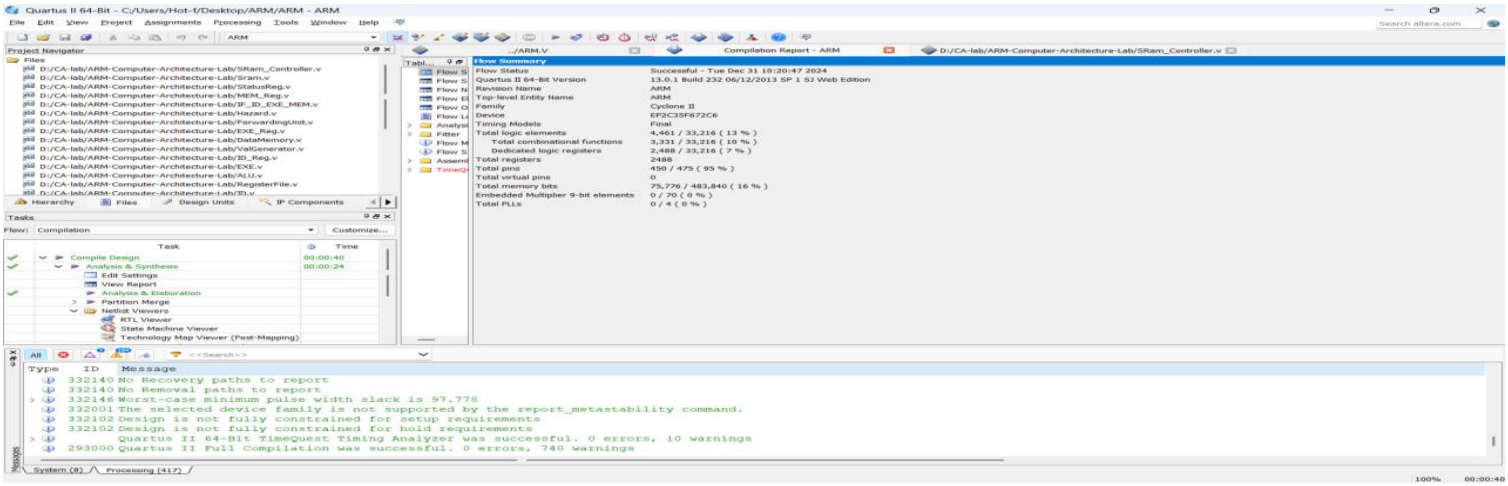


Fig .22. Compile output of sram(it used less logic element)

Fig.23. Signal tab of Sram

Module-by-module code

There will be a short definition of each module code that is used here.

SRAM controller

In this section, the `always` statement that handles `SRAM_ADDR` and related signals is introduced first.

```
 always @(ps, rd_en, wr_en)
begin
    ready = 1'b1;
    SRAM_WE_N = 1'b1;
    SRAM_DQ_Reg = 16'd0;
    SRAM_ADDR = 18'd0;
    case (ps)
    3'b000:
        ready = idel_ready;
    3'b001:
    begin
        ready = 1'b0;
        SRAM_WE_N = ~wr_en;
```

```
        if (wr_en == 1'b1) begin
            SRAM_ADDR = addr;
            SRAM_DQ_Reg = writeData[15:0];
        end else begin
            SRAM_ADDR = addr;
        end
    end
    3'b010:
    begin
        SRAM_WE_N = ~wr_en;
        ready = 1'b0 ;
        if (wr_en == 1'b1) begin
            SRAM_ADDR = addr + 18'd1;
            SRAM_DQ_Reg = writeData[31:16];
        end else begin
            SRAM_ADDR = addr;
            readData[15:0] = SRAM_DQ;
        end
    end
    3'b011:
    begin
        SRAM_WE_N = 1'b1;
        ready = 1'b0;
        if (wr_en == 1'b0) begin
            SRAM_ADDR = addr + 18'd1;
            // readData[31:16] = SRAM_DQ;
        end
    end
    3'b100:begin
        SRAM_WE_N = 1'b1;
        ready = 1'b0;
         if (wr_en == 1'b0) begin
            SRAM_ADDR = addr + 18'd1;
            readData[31:16] = SRAM_DQ;
        end
    end
    3'b101:
    begin
        SRAM_WE_N = 1'b1;
        ready = 1'b1;
    end
    endcase
end
```

This is a little different from the code that has been brought in lab description but this is true, because SRAM has synchronous read with clock.
The interface of module:

```
module SramController(input clk, rst, wr_en, rd_en,
                      input [31:0] ALU_Res, writeData,
                      output reg [31:0] readData,
                      output reg ready,
                      inout [15:0] SRAM_DQ,
                      output reg [17:0] SRAM_ADDR,
                      output reg SRAM_WE_N,
                      output SRAM_UB_N, SRAM_LB_N, SRAM_CE_N, SRAM_OE_N);
```

It is said that the memory address of our program starts from 1024 and our cpu needs words, so this is how the address is generated.

```
 assign temp = ALU_Res-32'd1024;
 assign addr={temp[18:2],1'b0};
```

08

Cache

In the final part of the lab, a cache was implemented. This cache is a two-way, 64-bit cache that can store four words per line. It uses LRU (Least Recently Used) replacement for temporal locality, and stores two adjacent words in 64 bits to optimize spatial locality.

Overview

To implement the cache, we use the logic elements of the FPGA. Unlike real caches, this cache provides results in a single clock cycle. However, it still demonstrates how caches can improve CPU efficiency. We implemented this section slightly differently from the original description. Specifically, we use a write-through policy, writing data to both the memory and the cache simultaneously.

Key features

In this part, the SRAM controller is modified to handle two addresses: one for reading (now two words are read) and one for writing. This will be explained in the code section.

Fig.24.cache on the board

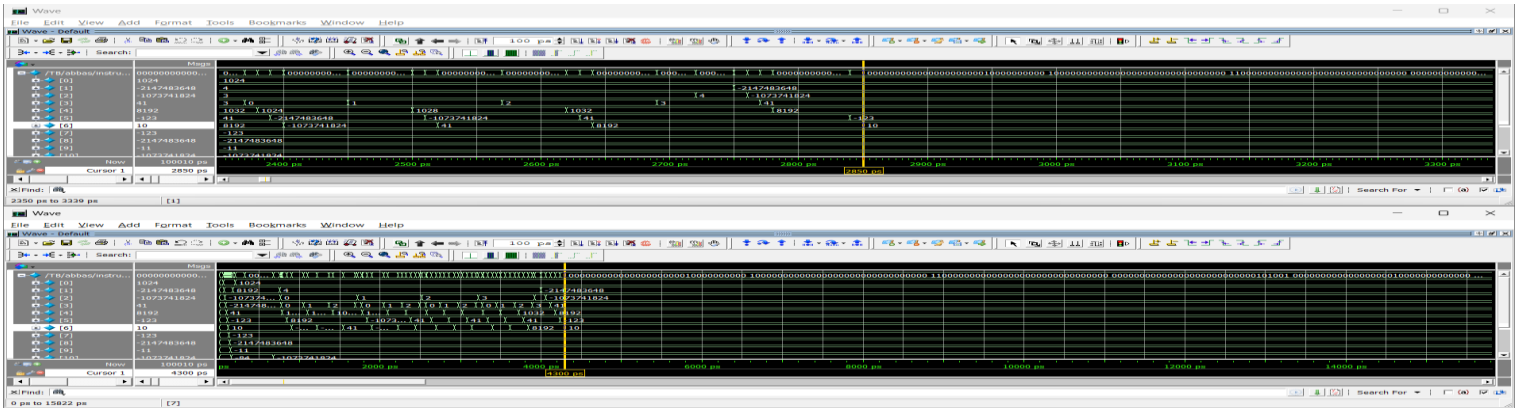


Fig. 25. Comparison of cache and without cache

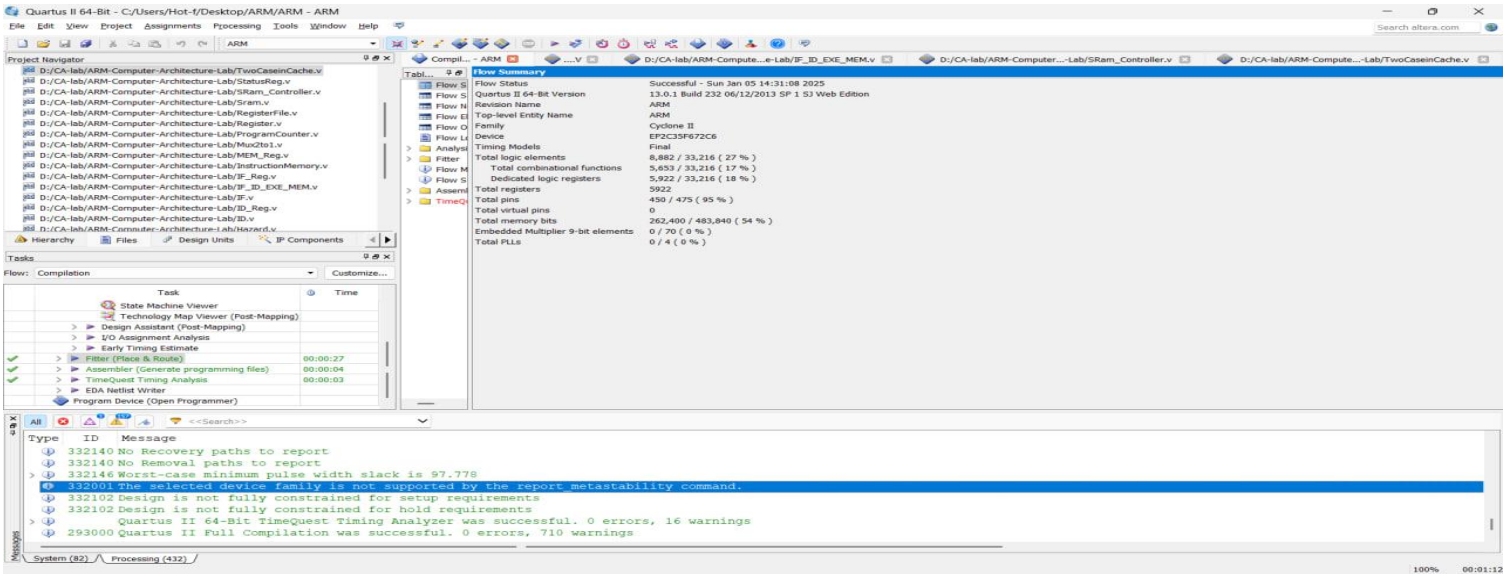


Fig .26. Compile result of cache state

So by adding cache we have 4300/2850= 1.5 speed up

Module-by-module code

There will be a short definition of each module code that is used here.

Cache controller

Because of complexion the whole code of cache section has been brought:

```
module TwoCaseinCache(
    input clk, rst, rd_en_in, wr_en_in, sram_ready_in,
    input [31:0] adr_in, wdata_in,
    input [63:0] sram_read_data_in,
    output sram_wr_en_out, ready_out, sram_rd_en_out,
    output [31:0] rdata_out
);

wire [2:0] offset;
wire [5:0] index;
wire [9:0] tag;

assign offset = adr_in[2:0];
assign index = adr_in[8:3];
assign tag = adr_in[18:9];

reg [31:0] way0_f[0:63];
reg [31:0] way0_s[0:63];
```

```
reg [31:0] way1_f[0:63];
reg [31:0] way1_s[0:63];
reg [9:0] way0_tag[0:63];
reg [9:0] way1_tag[0:63];

reg [63:0] index_lru;
wire [31:0] data_way0, data_way1;
wire [9:0] tag_way0, tag_way1;
wire valid_way0, valid_way1;

assign data_way0 = (offset[2] == 1'b0) ? way0_f[index] : way0_s[index];
assign data_way1 = (offset[2] == 1'b0) ? way1_f[index] : way1_s[index];
assign tag_way0 = way0_tag[index];
assign tag_way1 = way1_tag[index];

reg [63:0] way0_valid;
reg [63:0] way1_valid;
assign valid_way0 = way0_valid[index];
assign valid_way1 = way1_valid[index];

wire hit;
wire hit_way0, hit_way1;

assign hit_way0 = (tag_way0 == tag && valid_way0 == 1'b1);
assign hit_way1 = (tag_way1 == tag && valid_way1 == 1'b1);
assign hit = hit_way0 | hit_way1;

wire [31:0] data;
wire [31:0] read_data_q;

assign data = hit_way0 ? data_way0 : hit_way1 ? data_way1 : 32'bz;

assign read_data_q = hit ? data : sram_ready_in ? (offset[2] == 1'b0 ?
sram_read_data_in[31:0] : sram_read_data_in[63:32]) : 32'bz;

assign rdata_out = (rd_en_in == 1'b1) ? read_data_q : 32'bz;
assign ready_out = sram_ready_in;

assign sram_rd_en_out = ~hit & rd_en_in;
assign sram_wr_en_out = wr_en_in;

reg [31:0] read_cnt, hit_cnt;

always @(posedge clk or posedge rst)
begin
    if (rst) begin
        way0_valid <= 64'd0;
        way1_valid <= 64'd0;
        index_lru <= 64'd0;
```

```
        read_cnt <= 32'b0;
        hit_cnt <= 32'b0;
    end
    else begin
        if (rd_en_in) begin //5
            if (hit) begin //
                index_lru[index] <= hit_way1;
                hit_cnt <= hit_cnt + 1;
                read_cnt <= read_cnt + 1;
            end //
            else begin
                if (sram_ready_in) begin //1
                    read_cnt <= read_cnt + 1;
                    if (index_lru[index] == 1'b1) begin//2
                        index_lru[index] <= 1'b0;
                        {way0_s[index], way0_f[index]} <= sram_read_data_in;
                        way0_valid[index] <= 1'b1;
                        way0_tag[index] <= tag;
                    end//2
                    else begin//3
                        index_lru[index] <= 1'b1;
                        {way1_s[index], way1_f[index]} <= sram_read_data_in;
                        way1_valid[index] <= 1'b1;
                        way1_tag[index] <= tag;
                    end//3
                end//1
            end
        end//5
        if (wr_en_in) begin//8
            if (hit_way0) begin//6
                index_lru[index] <= 1'b0;
                if (offset[2] == 1'b0)
                    way0_f[index] <= wdata_in;
                else
                    way0_s[index] <= wdata_in;
            end //6
            else if (hit_way1) begin//7
                index_lru[index] <= 1'b1;
                if (offset[2] == 1'b0)
                    way1_f[index] <= wdata_in;
                else
                    way1_s[index] <= wdata_in;
            end//7
        end //8
    end //9
end

endmodule
```

This code is stateless and waits until the memory is read in case of a cache miss. If it is a hit, it uses the tag to find the data in one of the ways and, with the offset, locates it within the 64-bit block. The `read_cnt` and `hit_cnt` act like program counters. As mentioned earlier, the code modifies the data in the case of a store-word instruction.

At the end, I will present the memory stage hierarchy. As you can see, the SRAM controller communicates exclusively through the cache controller.

```
module DataMemory( input clk , rst , MEM_W_EN, MEM_R_EN,
                   input[31:0] ALU_Res, Val_Rm,
                   output [31:0] out,
                   output ready , output [15:0]SRAM_DQ, output [17:0] SRAM_ADDR,
                   output SRAM_UB_N, SRAM_LB_N, SRAM_CE_N, SRAM_OE_N,
                   output SRAM_WE_N);
    wire [63:0] sram_data_out;
    wire sram_ready , SramWriteEn, SramReadEn;
    TwoCaseinCache cache(clk, rst, MEM_R_EN,MEM_W_EN,
sram_ready,ALU_Res,Val_Rm,sram_data_out,SramWriteEn,ready , SramReadEn,out);



    SramController sram_ctrl(clk, rst, SramWriteEn, SramReadEn,
                        ALU_Res, Val_Rm,
                        sram_data_out,
                        sram_ready,
                        SRAM_DQ,
                        SRAM_ADDR,
                        SRAM_WE_N,
                        SRAM_UB_N, SRAM_LB_N, SRAM_CE_N, SRAM_OE_N);

endmodule
```

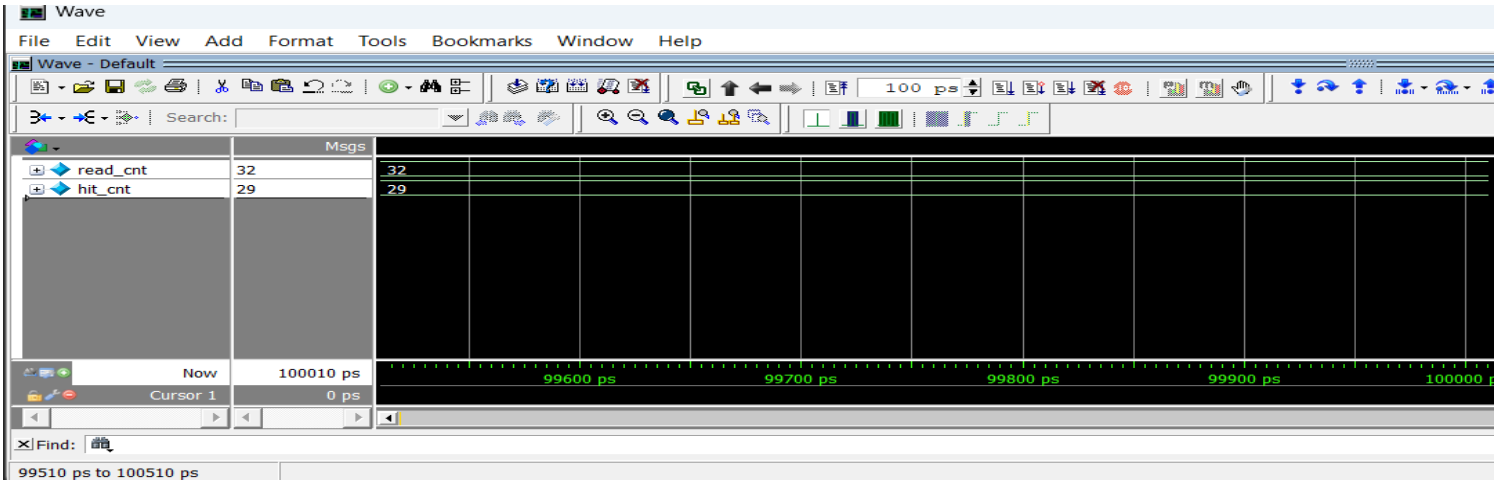


Fig.27 read_cnt and hit count it has 0.9 hit rate

Related Links:

GitHub Repository

Hatef

Arian