University of Tehran

Faculty of Engineering

School of Electrical and Computer Engineering

Computer Architecture Lap Report

Soroosh Mirzasarvari
810198478

Mohammad Farrahi
810198451

Table of Content

ARM Design

Basic Components

- **ALU:**
  This module is instantiated in the Execution Stage. It is implemented using an always block with combinational dependencies. Making it parallel-case. It also issues certain signals called *z, n, c, and v* for zero, negative, carry out, and overflow respectively. These signals are for checking the condition in the commands and are explained more later. The signals are passed to the Status Register which is explained further in this section. The code for this component is shown in Figure 1.

```
module ALU (
  input [3:0] exe_cmd,
  input c_in,
  input signed [31:0] val1 , val2,
  output reg [32:0] result_alu,
  output n, z, c, v
  );
  always @(*) begin
    case (exe_cmd) //synthesis parallel_case
      4'b0001 /*mov_a*/ : result_alu = val2;
      4'b1001 /*mvn_a*/ : result_alu = ~val2;
      4'b0010 /*add_a*/ : result_alu = val1 + val2;
      4'b0101 /*sbc_a*/ : result_alu = val1 - val2 - 1;
      4'b0110 /*and_a*/ : result_alu = val1 & val2;
      4'b0111 /*orr_a*/ : result_alu = val1 | val2;
      4'b0011 /*adc_a*/ : result_alu = val1 + val2 + c_in;
      4'b0100 /*sub_a*/ : result_alu = val1 - val2;
      4'b1000 /*eor_a*/ : result_alu = val1 ^ val2;
      default:
        result_alu = 33'd0;
    endcase
    end
  assign n = result_alu[31];
  assign z = (result_alu == 33'd0);
  assign c = result_alu[32];
  assign v = result_alu[31] ^ c ;
endmodule
```

Figure 1. ALU code.

- **RAM:**
  This component which is instantiated in the Memory Access Stage is a parameterized RAM. The important note in this component is that **reading is asynchronous whereas writing is synchronous and requires**

one clock. There are *r_en* and *w_en* input signals that are read enable and write enable. The code is shown in Figure 2.

```
module RAM #(parameter word_width = 32, parameter address_width = 5, parameter depth = 8) (
  input clk, rst,
  input w_en,
  input r_en,
  input [address_width-1:0] address,
  input [word_width-1:0] d_in,

  output [word_width-1:0] d_out
);
  reg [word_width-1:0] mem [0:depth-1];
  integer i;
  always @(posedge clk) begin
    if(rst) begin
      for(i = 0; i < depth; i = i+1)
        mem[i] = 0;
    end
    if(w_en)
      mem[address - 1024] = d_in;
  end
  assign d_out = mem[address - 1024];
endmodule
```

Figure 2. RAM code.

- **Register File:**
  This module is used in the Instruction Decode Stage and has 15 registers of 32 bits. **Reading happens asynchronously and does not require any flag whereas writing needs a *write_back_en* flag to write and is synchronous.** The code is shown in Figure 3. Also, one important fact is that **writing happens on the negedge of the clock to avoid structural hazards.**

- **Status Register:**
  This is a simple 32-bit register but only 4 most significant bits are set being the signals explained in the ALU section with a write-enable signal. Reading is again asynchronous. The code is shown in Figure 4.

```
module RegisterFile (
  input clk, rst,
  input [3:0] src1, src2, dest_wb,
  input[31:0] result_wb,
  input write_back_en,
  output [31:0] reg1, reg2
);
  integer counter = 0;
  reg[31:0] reg_file[0:14];

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

  always @(negedge clk, posedge rst) begin
    if (rst) begin
      for(counter=0; counter<15; counter=counter+1)
        reg_file[counter] <= counter;
    end
    else if (write_back_en) reg_file[dest_wb] <= result_wb;
  end

endmodule
```

Figure 3. Register File code.

```
module StatusRegister (
  input wr_en ,n, z, c, v, clk, rst,
  output[3:0] sr
);
  reg[31:0] status_reg;

  assign sr = status_reg[31:28];

  always @(negedge clk) begin
    if(rst == 1)
      status_reg <= 0;
    else if(wr_en == 1) begin
      status_reg[31 /*N*/] <= n;
      status_reg[30 /*Z*/] <= z;
      status_reg[29 /*C*/] <= c;
      status_reg[28 /*V*/] <= v;
    end
  end
endmodule
```

Figure 4. Status Register code.

Instruction Fetch Stage

- **IF_stage module:**

  There are several components here. First, we list all the inputs and outputs of the module. Inputs are as follows:

  ➔ clk: the clock.
  ➔ rst: the reset signal
  ➔ Freeze: the signal to freeze the instruction. This will result in pc staying constant.
  ➔ Branch_taken: a single bit that if 1, the next value of PC will become branch_addr instead of PC + 4.
  ➔ Branch_addr: the address of branch address in instruction memory that PC has to point to.

  The outputs of this module are as below:

  ➔ PC: the final value for the PC register.
  ➔ Instruction: the 32-bit value of the instruction stored in the memory slot in the instruction memory that the PC points to.

  The modules used in this slot are shown in Figure 5. Each module is explained below:

  ➔ PC register: a simple 32-bit register that points to the memory slot for the next instruction in the instruction memory. This register does not have any load-enable signal and with each clock is loaded but the freeze signal will prevent it from loading any new value.
  ➔ Instruction Memory: This is 32-bit memory that is read-only. The read is asynchronous and the implementation is set to be combinational because of synthesizing reasons.

  The code for this module is shown in Figure 6.

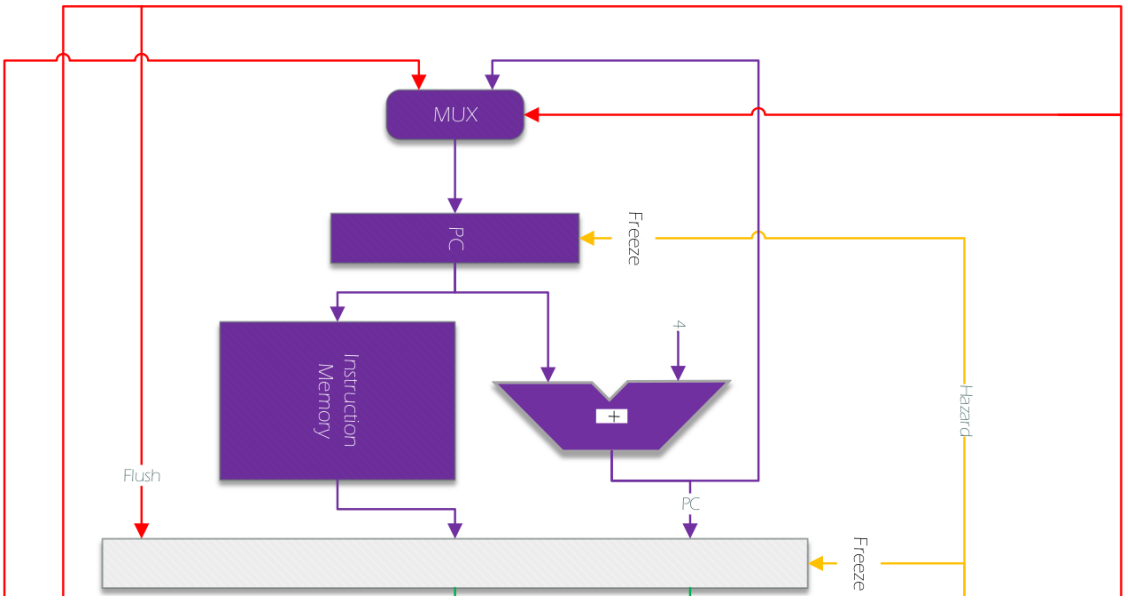

Figure 5. Instruction Fetch Datapath.

```
module IF_stage (
  input clk,
  input rst,
  input freeze,
  input branch_taken,
  input [31:0] branch_addr,

  output [31:0] PC,
  output [31:0] instruction
);
  // internal wires
  wire [31:0] mux_out;
  reg [31:0] pc_reg_out;

  // mux
  assign mux_out = branch_taken ? branch_addr : PC;
  // pc register
  always @(posedge clk, posedge rst) begin
    if (rst)
      pc_reg_out <= 32'd0;
    else if(~freeze)
      pc_reg_out <= mux_out;
  end
  // adder
  assign PC = pc_reg_out + 32'd1;

  InstructionMemory instruction_memory(pc_reg_out, instruction);
endmodule
```

Figure 6. IF_stage module code.

- **IF_ID_reg:**
  This is a simple register that saves the value of outputs of the IF_stage module for the next clock and the next stage. **From now on, the convention of naming the register between stages A and B is A_B_reg**. So the IF_ID_reg is the register between the IF stage and the ID stage. Note that **this register does not update if the freeze signal is set.** With this technique, we can stall the processor. Also, **the register is reset if the flush signal is set**. With this technique, we can create a bubble in the processor.

Instruction Decode Stage

This stage is where the instruction is decoded and the needed signals for execution and storage are set. Additionally, a figure of datapath is shown in Figure 7.

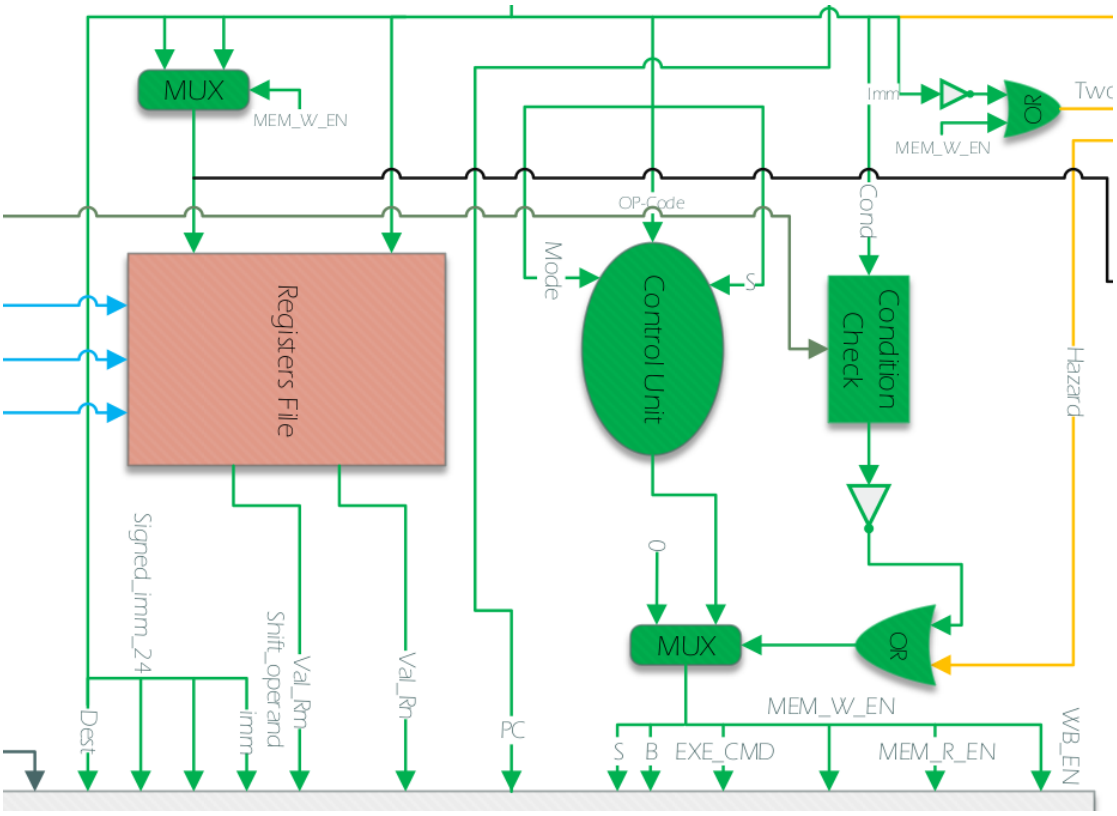

Figure 7. Instruction Decode Stage Datapath.

- **ID_stage module:**
  The inputs are first explained:
  ➔ clk and rst: for the system clock and reset

- ➔ Instruction: the 32-bit instruction from the IF_stage module passed by IF_ID_reg.
- ➔ Write back signals and wires:
  - ◆ wb_en_in: the load-enable signal if set, the result_wb is stored in the register shown by dest_wb.
  - ◆ result_wb: the 32-bit data that is set to be stored in the register file.
  - ◆ dest_wb: the 4-bit address of the register that write-back is executed on.
- ➔ hazard signal: the signal that if set, shows that the Hazard Detection Unit has detected a hazard and now all the controlling signals should be cleared. Further explanation is provided in the Hazard Detection Unit section.
- ➔ sr: a 4-bit wire that contains the signals from Status Register. These signals are used in the Condition Check module where they are evaluated for the condition of the command and if there is any undelivered condition, the controlling signals must again be cleared.

The most important outputs of this part are the control signals from the Control Signal module. As well as some other information that is needed for the next stages. Outputs are explained below:

- ➔ Control signals:
  - ◆ wb_en: if set, data is stored in the register file in the Write-back stage.
  - ◆ mem_write_en: if set, data is stored in the memory.
  - ◆ mem_read_en: if set, data is read from the memory.
  - ◆ b signal: if set, a branch is commanded.
  - ◆ s signal: if set, Status Register is loaded.
  - ◆ exe_command: the opcode for the ALU operation.
  - ◆ imm: if set, it means that the command is immediate. This signal is used in the next stage (Execution stage) in the Val2Generator module which is explained further down the documentation.
- ➔ val_rn: the 32-bit value of the register shown by the Rn part of the command (i.e instruction[19:16]).

- ➔ val_rm: the 32-bit value that can either mean the Rd value or the Rm value.
- ➔ dst_out: instruction[15:12]
- ➔ Val2Generator wires: these are parts of the instruction that depending on the command, might be used in Val2Generator to build the second 32-bit input of the ALU.
  - ◆ signed_imm_24: instruction[23:0]
  - ◆ shift_operand: instruction[11:0]

There are three main components in this module:

- ➔ Register file: connected to the rn and rm signals. It is explained in the previous section.
- ➔ Condition Check: This module inputs all the Status Register signals and decides whether the condition is met or not. As it was explained previously, if the resulting signal is set, the processor creates a bubble. To implement this module, we have used an active-low output signal called cond_check. Also, each condition is checked in a case block and the checking is done based on the signals in the Status Register. The interface for this module is shown in Figure 8. Additionally, an example of checking the condition is provided in Figure 9.

```
module ConditionCheck (
  input[3:0] sr,
  input[3:0] condition,
  output reg cond_check
);
```

Figure 8. ConditionCheck interface.

```
case(condition)
  4'd0 /*eq*/ : begin
    if(sr[2/*Z*/] == 1'b1)
      cond_check = 1'b0;
  end
```

Figure 9. Example of condition check code.

- ➔ Control Unit: This module is the core of the Instruction Decode stage. This module issues the needed signals to properly execute and store the instruction.

The ControlUnti module itself is a kind of wrapper on ControlUntiCore. Also, the responsibility of creating a bubble by clearing all the signals is devoted to this module.
ControlUntiCore is again implemented with a combinational always block that based on the opcode of the instruction and other signals (e.g immediate or s_in) sets a subset of signals. An example is provided in Figure 10. The 0100 upcode is devoted to ADD command in ARM. Also, the interface is provided in Figure 11.

```
4'b0100 : begin
  cmd_exe = 2 ;
  s_out = s_in ;
  wb_en = 1;
end
```

Figure 10. Example of ContolUnitCore setting control flags.

```
module ControlUnitCore (
  input [1:0] mode,
  input [3:0] opcode,
  input s_in,

  output reg [3:0] cmd_exe,
  output reg mem_r_en,
  output reg mem_w_en,
  output reg wb_en,
  output reg b_out,
  output reg s_out
);
```

Figure 11. ControlUnitCore interface.

- **ID_EXE_reg:**
  Simple register like the previous stage. Flush clears all parts of the register.

Execution Stage

In this stage of the pipeline, the result of ALU is calculated and stored in the EXE_MEM_reg. Also to make the second input of the ALU, a module called Val2Generator is designed. A figure of datapath is shown in Figure 12.

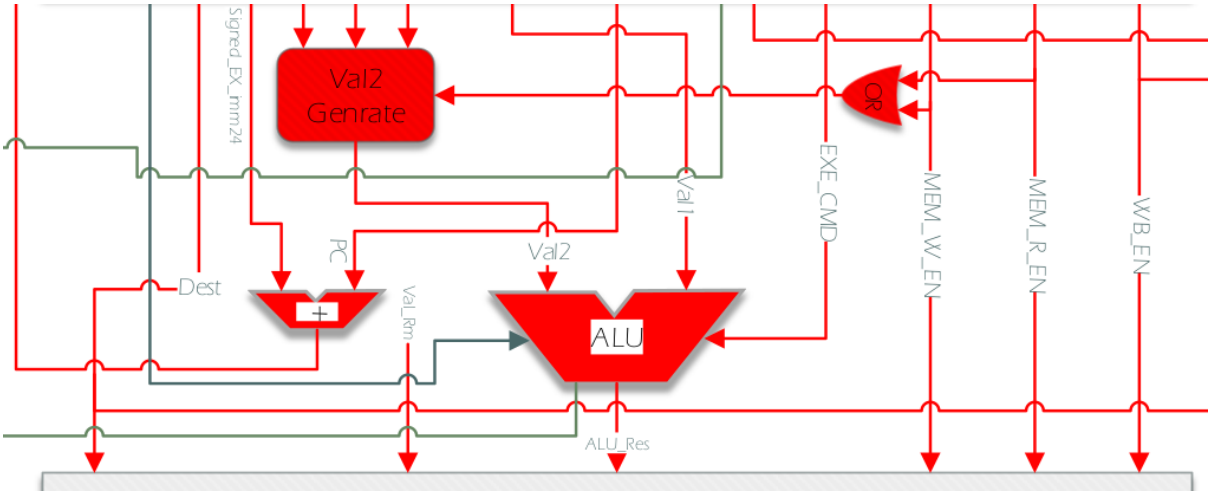

Figure 12. Execution Stage Datapath.

- **EXE_stage module:**

  It is important to know that most of the inputs in this part are only transferred to the next stages. The inputs are explained below:

  ➔ clk and rst: for the system clock and reset.

  ➔ Control signals that are issued in the next stage and also all the information needed like the 12-bit shipt_operand.

  For the outputs, the most important one is the result of ALU. All of them are explained below:

  ➔ alu_result: a 32-bit wire that has the asynchronous result of the ALU.

  ➔ The memory access flags are propagated from the previous stage. (mem_read_en and mem_write_en).

  ➔ wb_en flag from the previous stage.

  ➔ 32-bit rn_value and the 4-bit dest for the write-back stage.

  The more important part of this stage is the different modules it has. Note that **to reduce wiring overhead, the Status register is instantiated in this module too**:

  ➔ ALU: this module has been explained in the previous section called Basic Components. It is important to note that **the result is**

provided asynchronously and non of the operations need more than one clock length of time to execute.

➔ Val2Generator: This module gets all the wires needed and provides a 32-bit output that is used as the second input of the ALU. It creates the value based on the imm signal and also a signal called or_out which essentially shows whether we have memory access or not (i.e it ors the two enable signals of memory). There are different outputs for the second value:

- If or_out is set: it means we have either STR or LDR commands and the second value for ALU must be the last 12 bit of command (in the form of 12-bit shift_operand wire). The 12 bits are then sign-extended to be 32 bits.
- Else if imm is set: it means that the command is 32-bit immediate and the output is calculated based on a rotational shift explained in the 01-ARM documentation provided in the LAB class.
- Else if imm is clear and the 5th bit is also clear: it means that the command is the immediate shift and based on the shift value (shift_operand[6:5]) the type of shift is decided. And then the shift is applied on the 32-bit Rm value as much as the shift_immd value (shift_operand[11:7]).

The code for this part of the switch is shown in Figure 13.

```
assign rotate_imm_mul2 = (shift_operand[11:8] << 1);
always @(*) begin
  temp_data = 64'd0;
  shift_temp = 64'd0;
  val2 = 32'd0;
  if(or_out == 1) begin
    val2 = {{20{shift_operand[11]}},shift_operand};
  end
  else if(imm == 1) begin
    val2 = {{24{1'b0}},shift_operand[7:0]};
    temp_data[63:32] = val2;
    shift_temp = temp_data >> rotate_imm_mul2;
    val2 = (shift_temp[63:32] | shift_temp[31:0]);
  end
  else if(imm == 0 && shift_operand[4] == 0) begin
    case(shift_operand[6:5])
      2'b00: val2 = val_rm << (shift_operand[11:7]);
      2'b01: val2 = val_rm >> (shift_operand[11:7]);
      2'b10: val2 = val_rm >>> (shift_operand[11:7]);
      2'b11: begin
        temp_data[63:32] = val_rm;
        shift_temp = temp_data >> shift_operand[11:7];
        val2 = (shift_temp[63:32] | shift_temp[31:0]);
      end
    endcase
  end
end
```

Figure 13. Val2Generator code.

- **EXE_MEM_reg:**
  A simple register that stores the outputs explained in the Exe_stage module for further use down the pipeline stages.

Memory Access Stage

This stage contains the actual RAM explained above and there is a register called MEM_WB_reg that stores the result of reading memory and also the result of ALU. Datapath is shown in Figure 14.

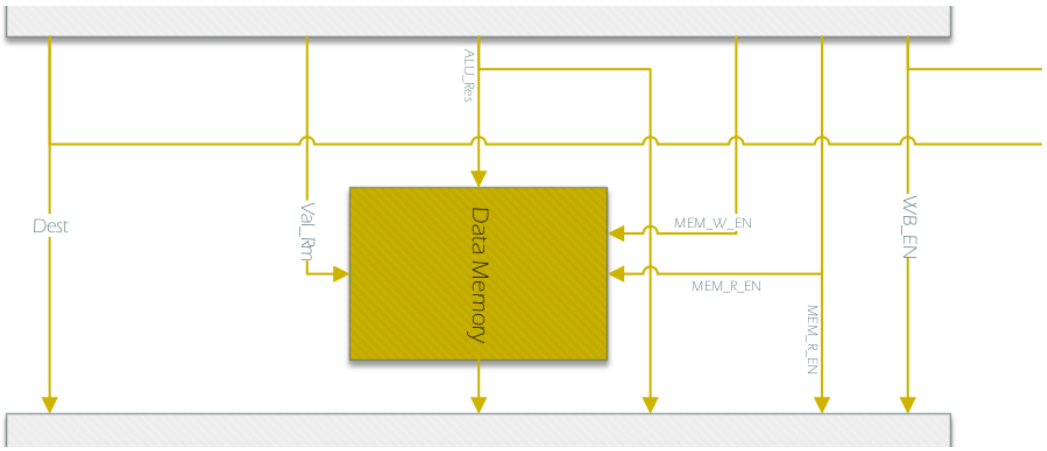

Figure 14. Memory Access Stage Datapath.

Writeback Stage

This stage contains a multiplexer that has to decide whether to use the ALU result or the output of memory as the value to store in the destination register. To write to the register file, the signals are passed to the ID_stage module to write to the register file. Datapath is shown in Figure 15.

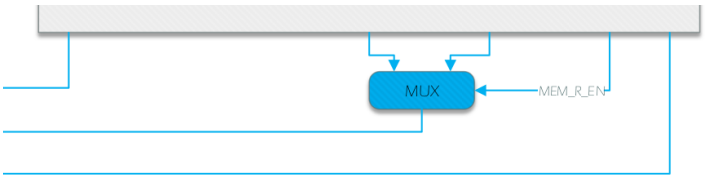

Figure 15. Writeback Stage Datapath.

Hazard Detection Unit

As discussed in the 01_ARM_Comment_III documentation, since ARM is an in-order processor, the only hazard that needs to be detected is the read-after-write hazard which is a type of data hazard.
This hazard happens when a command changes the value of a certain register and in the **next two preceding commands,** the value of the same register is used. To give an example, we use these three commands:

```
ADD   R2, R1, R0
OR    R3, R1, R2
```

AND R4, R1, R2

In this example, R2 is loaded in the first command and then its value is used in the second and third commands. Since we use a pipelined processor, the second and third commands access the register file in the ID stage sooner than the first command access loads the register file in the WB stage. To detect this hazard we check the controlling signals in the different stages. The interface of the HazardDetectionUnit module is shown in Figure 16. The only output of this module is the freeze signal that if set, means that processor needs to stall for one clock.

```
module HazardDetectionUnit(
  input[3:0] rn_ID,
  input[3:0] src2_ID,
  input[3:0] dst_exe,
  input[3:0] dst_memmory,
  input two_src_ID,
  input memory_read_en_exe,
  input wb_en_memory,
  input wb_en_exe,
  input forwarding_en,

  output reg freeze
);
```

Figure 16. HazaardDetectionUnit interface.

To explain how this module is implemented, the 4-bit address of certain registers in different stages are checked. Different cases when the freeze signal is set are listed below:

RAW data hazard list

1. if the memory read enable flag in the EXE stage (memory_read_en_exe) is set, and the destination register in the EXE stage is the same as the Rn register in the ID stage, the freeze signal is set.
2. If memory_read_en_exe is set and the command is one of the commands that involve two registers (this is checked with the two_src_ID flag), then if the destination register in the EXE stage is the same as the Rm register in the ID stage, the freeze signal is set.
3. If the wb_enable flag in the EXE stage is set and rn_ID is the same as dst_memory.

4. If the wb_enable flag in the EXE stage is set and rn_ID is the same as dst_exe.
5. If two_src_ID is set as well as wb_en_exe and the values of src2_ID and dst_exe are the same.
6. If wb_en_memory is set and the values of src2_ID and dst_exe are the same.

An implementation of the explained module is shown in Figure 17.

```
always @(*) begin
  freeze = 1'b0;
  if (memory_read_en_exe == 1'b1)begin
    if (dst_exe == rn_ID)
      freeze = 1'b1;

    else if (two_src_ID == 1'b1 && dst_exe==src2_ID) begin
      freeze = 1'b1;
    end
  end
    if(wb_en_exe == 1'b1) begin
      if(rn_ID == dst_exe)
        freeze = 1'b1;
    end
    if(wb_en_memory == 1'b1) begin
      if(rn_ID == dst_memmory)
        freeze = 1'b1;
    end
    if(two_src_ID == 1'b1) begin
      if(wb_en_exe == 1'b1) begin
        if(src2_ID == dst_exe)
          freeze = 1'b1;
      end
    end
    if(wb_en_memory == 1'b1) begin
        if(src2_ID == dst_memmory)
          freeze = 1'b1;
    end
end
```

Figure 17. HazaradDetectionUnit always block to issue the freeze flag.

Forwarding

When a read-after-write hazard is detected, sometimes there is no need to stall the execution. In other words, instead of using the 32-bit output of the register file in the EXE stage, we can use the previous command's MEM stage's ALU result or two previous commands' WB stage's ALU result. In this case, we can "forward" the data to the next stages and prevent some stalls caused by hazards. Note that in the occasions of RAW data hazard listed in the previous section,

the first two possibilities cannot be solved using the forwarding method. To achieve that, several changes need to take place in the system:

Changes in the EXE_stage module

As explained above, either of the operands in ALU's input can be fed not only via the ID_stage's register file but also from the ALU result in the MEM stage or the writeback value in the WB stage. Thus, a multiplexer is designed to decide between the different feeds for the Val2Generator input and ALU's first operand input. The modified datapath is shown in Figure 18. Where the dark blue input of multiplexers is the old value of the previous stage. The golden line is the ALU result in the MEM stage's register (EXE_MEM_reg). Finally, the light blue wire is fed from the writeback value in the WB stage. The code for each of these multiplexers is shown in Figure 19 and Figure 20.

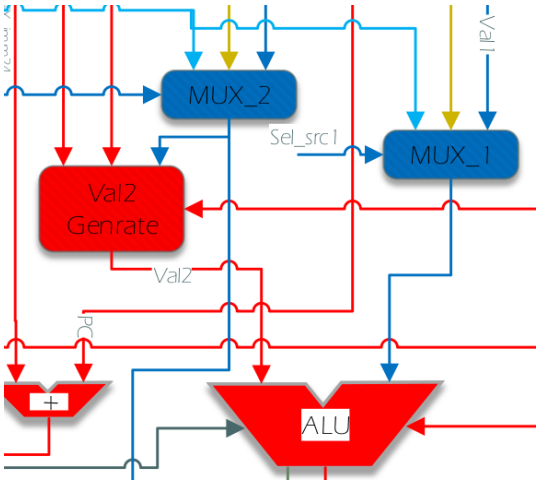

Figure 18. Changed Datapath due to Forwarding.

```
assign val1 = src1_mux_out;
always @(*) begin // src_1_mux
  if(forwarding_en == 0)
    src1_mux_out = val_rn_ID_EXE;
  else begin
    if (select_src_1 == 0)
      src1_mux_out = val_rn_ID_EXE;
    else if (select_src_1 == 1)
      src1_mux_out = alu_result_mem;
    else if (select_src_1 == 2)
      src1_mux_out = data_wb_wb;
    else
      src1_mux_out = val_rn_ID_EXE;
  end
end
```

Figure 19. Forwarding mux_1.

```
always @(*) begin // src_2_mux
  if(forwarding_en == 0)
    src2_mux_out = val_rm_ID_EXE;
  else begin
    if (select_src_2 == 0)
      src2_mux_out = val_rm_ID_EXE;
    else if (select_src_2 == 1)
      src2_mux_out = alu_result_mem;
    else if (select_src_2 == 2)
      src2_mux_out = data_wb_wb;
    else
      src2_mux_out = val_rm_ID_EXE;
  end
end
```

Figure 20. Forwarding mux_2.

Also, some inputs had to be added to the MEM_stage module which is shown in Figure 21. There are two 2-bit signals called select_src_1 and select_src_2 that are **the select signals for the multiplexers and are issued from the Fowrading Unit**. The Forwarding Unit is the next component that we discuss.

```
input[31:0] alu_result_mem,
input[31:0] data_wb_wb,
input[1:0] select_src_1,
input[1:0] select_src_2,
input forwarding_en,
```

Figure 21. Added inputs to EXE_stage.

Creating Forwarding Unit

This component inputs some of the control signals of different stages of the system and issues the two 2-bit multiplexers selects in the EXE stage. This module essentially decides if the current command in the EXE stage forms any RAW hazard with one or two previous commands that are in the MEM stage and WB stage respectively. The interface of the module is shown in Figure 22. By checking the wb_en signal in the MEM stage and WB stage, it can decide and issue the signals.

1. Both wb_en_mem and wb_en_wback are clear: There is no RAW hazard and thus both selects are set to 0 meaning the EXE_stage works as normal.

```
module ForwardingUnit (
  input[3:0] dst_memmory,
  input[3:0] dst_wb,
  input[3:0] src1,
  input[3:0] src2,
  input wb_en_memory,
  input wb_en_wback,

  output reg[1:0] select_src_1 = 2'd0,
  output reg[1:0] select_src_2 = 2'd0
);
```

Figure 22. ForwardingUnit interface.

2. Only wb_en_wback is set: The data should be fed from the writeback value and thus, based on the comparison between dst_wb and src1 and dst_wb and src2, one of select_src_1 or select_src_2 is set to 2.

3. Only wb_en_mem is set: The data should be fed from the ALU result in MEM_stage and similar to the previous case, one of the two select signals are set to 1.

4. Both wb_en_mem and wb_en_wback are set: This means that the current command has dependencies with both one and two previous commands. And thus, 4 cases can happen and the select signals are set accordingly.

The code for different cases of the Fowrading Unit is shown in Figure 23 to Figure 26.

```
always @(*) begin
  select_src_1 = 2'd0;
  select_src_2 = 2'd0;
  case({wb_en_memory, wb_en_wback})
    2'b00 : begin
      select_src_1 = 2'd0;
      select_src_2 = 2'd0;
    end
```

Figure 23. signals are 00.

```
    2'b01 : begin
      if(dst_wb == src1)
        select_src_1 = 2'd2;
      if(dst_wb == src2)
        select_src_2 = 2'd2;
    end
```

Figure 24. signals are 01.

```
    2'b10 : begin
      if(dst_memmory == src1)
        select_src_1 = 2'd1;
      if(dst_memmory == src2)
        select_src_2 = 2'd1;
    end
```

Figure 25. signals are 10.

```
    2'b11 : begin
      if(dst_memmory == src1)
        select_src_1 = 2'd1;
      else if(dst_wb == src1)
        select_src_1 = 2'd2;
      if(dst_memmory == src2)
        select_src_2 = 2'd1;
      else if(dst_wb == src2)
        select_src_2 = 2'd2;
    end
```

Figure 26. signals are 11.

Changes in the HazaradDetectionUnit module

As explained above, after forwarding is enabled, only the first RAW hazards need to stall the processor and thus in this module we check if forwarding is enabled, we don't check for other RAW hazards. The code is shown in Figure 27.

```
always @(*) begin
  freeze = 1'b0;
  if (memory_read_en_exe == 1'b1)begin
    if (dst_exe == rn_ID)
      freeze = 1'b1;

    else if (two_src_ID == 1'b1 && dst_exe==src2_ID) begin
      freeze = 1'b1;
    end
  end
  else if(forwarding_en == 1'b0) begin
    if(wb_en_exe == 1'b1) begin
      if(rn_ID == dst_exe)
        freeze = 1'b1;
    end
    if(wb_en_memory == 1'b1) begin
      if(rn_ID == dst_memmory)
        freeze = 1'b1;
    end
    if(two_src_ID == 1'b1) begin
      if(wb_en_exe == 1'b1) begin
        if(src2_ID == dst_exe)
          freeze = 1'b1;
      end
    end
    if(wb_en_memory == 1'b1) begin
        if(src2_ID == dst_memmory)
          freeze = 1'b1;
    end
    end
end
```

Figure 27. Changes in the HazardDetectionUnit.

SRAM

In the real world, the memory of the FPGA board is very limited and solutions to this shortage have to be provided. In this section, we have used the SRAM component embedded in the FPGA board as the external memory. The datapath of the MEM stage changes and instead of the native RAM component, the SRAMController module is replaced that interfaces with the SRAM in the FPGA board. An illustration of the changed data path of the MEM stage is provided in Figure 28.

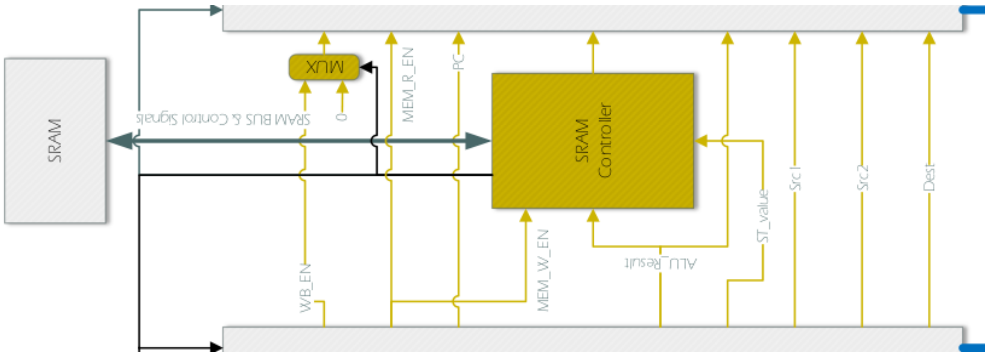

Figure 28. Changed Datapath for SRAM.

Creating SRAM Controller

This module acts as an interface to the actual SRAM component. Since SRAM is slower, 6 cycles are needed for the component to read or write data. The SRAMController module provides the needed state machine to work with SRAM.

Also, since SRAM works with 16-bit data, SRAMController has to provide needed buffering and concatenation. To do so, two 16-bit registers called regL and regH are instantiated for the first and second 16 bits of the input and output accordingly. These registers have load_enable signals and write synchronously and read asynchronously.

A 16-bit data wire is provided that while writing, feeds the SRAM_DQ bus of SRAM. While reading, the bus is in high impedance mode by the controller. The state machine is shown in Figure 29. As is shown in the state machine's figure, the top row of the machine devotes to writing to SRAM (as wr_en and rd_en are active-low) and the bottom part devotes to reading from SRAM.

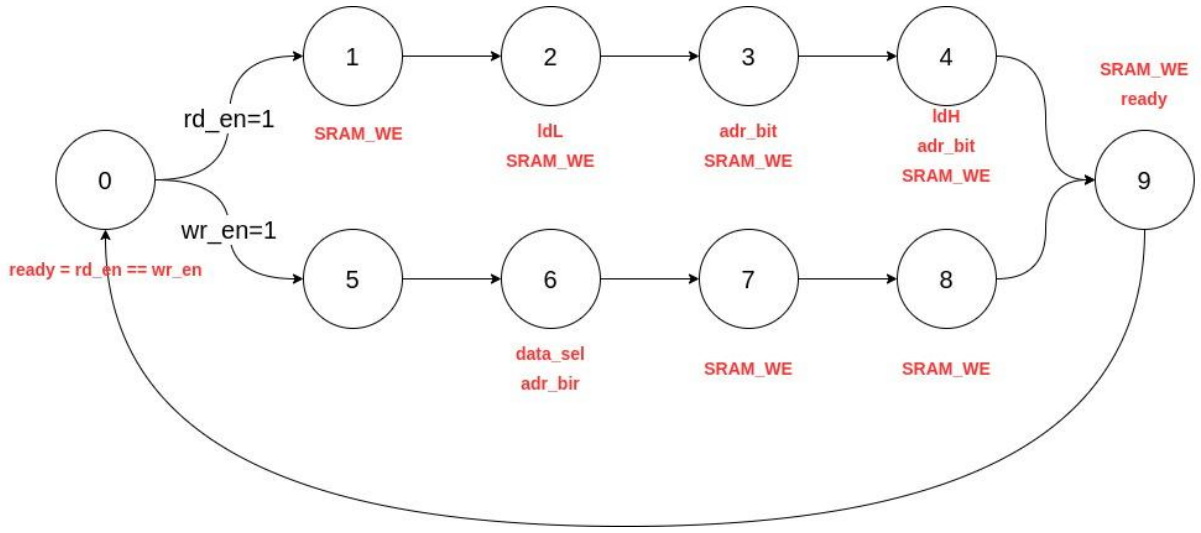

Figure 29. SRAMController State Machine.

The code for this module essentially implements the state machine. An example of one state of the machine is shown in Figure 30. Also, the output result of memory is the concatenation of regL and regH that is outputted via the read_data port.

```
4'b0001:begin
  nstate = 4'b0010;
  adr_bit=1'b0;
  ldL=1'b0;
  ldH=1'b0;
  data_sel= 1'b0;
  SRAM_WE = 1'b1;
end
```

Figure 30. Example of code in SRAMController's state machine.

Caching

To resolve the slow speed caused by the 6 clocks needed for SRAM, we use a cache. The replacement policy for this caching system is Least Recently Used (LRU). To implement caching, a CachingController is designed that acts as a wrapper over the previous SRAMController.

Creating CacheController Module

The interface of this module is similar to SRAMController in the previous section. A figure is provided in Figure 31.

```
module CacheController (
  input clk,
  input rst,

  input wr_en,
  input rd_en,
  input[31:0] address,
  input[31:0] writeData,
  output[31:0] readData,
  output ready,

  inout[15:0] SRAM_DQ,
  output[17:0] SRAM_ADDR,
  output SRAM_UB_N,
  output SRAM_LB_N,
  output SRAM_WE_N,
  output SRAM_CE_N,
  output SRAM_OE_N
);
```

Figure 31. CahceController Interface.

Also, the SRAMController is instantiated in this module. The wiring is the same and is illustrated in Figure 32.

```
SramController sram_controller (
  .clk(clk),
  .rst(rst),

  .wr_en(wr_en),
  .rd_en(rd_en & ~hit),
  .address(address),
  .write_data(writeData),
  .read_data(sram_read_data),
  .ready(sram_ready),

  .SRAM_DQ(SRAM_DQ),
  .SRAM_ADDR(SRAM_ADDR),
  .SRAM_UB_N(SRAM_UB_N),
  .SRAM_LB_N(SRAM_LB_N),
  .SRAM_WE_N(SRAM_WE_N),
  .SRAM_CE_N(SRAM_CE_N),
  .SRAM_OE_N(SRAM_OE_N)
);
```

Figure 32. SramController Instantiation.

Reading Data

As it is mentioned in the description, the caching grid is 2-way Set Associates. Each way is a 64-bit grid and thus, we have 4 blocks of 32-bit-word length (dataA0, dataB0, dataA1, and dataB1). Also for each way, each block has a 10-bit tag wire for matching the block (tag0 and tag1 in the code) and also 1 bit for validation (v0 and v1). To apply the replacement policy, 1 bit is considered too (LRU). A figure of the said registers in the code is provided in Figure 33.

```
reg [0:63] LRU, v0, v1;
reg [9:0] tag0 [0:63];
reg [9:0] tag1 [0:63];
reg [31:0] data0A [0:63];
reg [31:0] data0B [0:63];
reg [31:0] data1A [0:63];
reg [31:0] data1B [0:63];
```

Figure 33 Cache Data Parts.

To implement the process of retrieving the data from the cache (and not the main memory) and check for data hits, several wire signals are designed. For each way, first, we match the tag part of the instruction (adr[18:9]) with the tag part of how the index of instruction (adr[8:3]) matches. If matched, and also the valid bit of the entry is set, a hit happens and the data is fed from the cache and the LRU bit is set in the always block. This process happens for both ways of the cache and is illustrated in Figure 34. Since each row in a way contains 2 32-bit words, to distinguish between the two, offset bits are considered.

```
wire hit, hit0, hit1, cmp1, cmp0;
assign cmp0 = (tag0[temp_adr[8:3]] == temp_adr[18:9]);
assign hit0 = cmp0 && v0[temp_adr[8:3]];
assign cmp1 = (tag1[temp_adr[8:3]] == temp_adr[18:9]);
assign hit1 = cmp1 && v1[temp_adr[8:3]];
assign hit = hit0 | hit1;
assign data0 = temp_adr[2] ? data0B[temp_adr[8:3]] : data0A[temp_adr[8:3]];
assign data1 = temp_adr[2] ? data1B[temp_adr[8:3]] : data1A[temp_adr[8:3]];
assign dataOut = hit0 ? data0 : data1;
```

Figure 34 Cache Hit Computation.

If there is a miss, the CacheController module will essentially retrieve the needed data from the memory and also updates the tables. To do so, the rd_en of SRAMController is set and after the ready signal is issued, the data is fed from the SRAMController's output. The code is illustrated in Figure 35.

```
assign readData = hit ? dataOut : temp_adr[2] ? sram_read_data[63:32] : sram_read_data[31:0];
```

Figure 35 Cach Output System.

To update the cache table, several occasions might happen:

1. If there is a hit, only the RLU bit is changed to point the corresponding way. The code is shown in Figure 36

```
else if (hit & rd_en) begin
  LRU[temp_adr[8:3]] <= hit0 ? 1'b0 : 1'b1;
end
```

Figure 36 If there is a hit.

2. If there is a miss, to update the cache after the data is retrieved from the memory, the entry that is replaced by the new data is decided from the LRU values and valid values. If there is an entry that is not valid (valid

bit is clear), the new data is overwritten in that slot. If both slots are not valid, the first way is replaced. The code is shown in Figure 37.

```
else if(v0[temp_adr[8:3]]==0 && v1[temp_adr[8:3]]==1)begin
  LRU[temp_adr[8:3]] <= 1'b0;
  data0A[temp_adr[8:3]] <= sram_read_data[31:0];
  data0B[temp_adr[8:3]] <= sram_read_data[63:32];
  tag0[temp_adr[8:3]] <= temp_adr[18:9];
  v0[temp_adr[8:3]]<=1'b1;
end
else if(v0[temp_adr[8:3]]==1 && v1[temp_adr[8:3]]==0)begin
  LRU[temp_adr[8:3]] <= 1'b1;
  data1A[temp_adr[8:3]] <= sram_read_data[31:0];
  data1B[temp_adr[8:3]] <= sram_read_data[63:32];
  tag1[temp_adr[8:3]] <= temp_adr[18:9];
  v1[temp_adr[8:3]]<=1'b1;
end
else begin
  LRU[temp_adr[8:3]] <= 1'b0;
  data0A[temp_adr[8:3]] <=  sram_read_data[31:0];
  data0B[temp_adr[8:3]] <= sram_read_data[63:32];
  tag0[temp_adr[8:3]] <= temp_adr[18:9];
  v0[temp_adr[8:3]]<= 1'b1;
end
```

Figure 37 if there is an invalid section.

3. If there is a miss and both corresponding ways have valid bits set, we decide based on LRU value to replace the entry that was least recently used. The code is shown in Figure 38.

```
else if(~hit & rd_en & sram_ready)begin
  if(v0[temp_adr[8:3]]==1 && v1[temp_adr[8:3]]==1)begin
    if(LRU[temp_adr[8:3]])begin
      LRU[temp_adr[8:3]] <= 1'b0;
      data0A[temp_adr[8:3]] <= sram_read_data[31:0];
      data0B[temp_adr[8:3]] <= sram_read_data[63:32];
      tag0[temp_adr[8:3]] <= temp_adr[18:9];
    end
    else begin
      LRU[temp_adr[8:3]] <= 1'b1;
      data1A[temp_adr[8:3]] <= sram_read_data[31:0];
      data1B[temp_adr[8:3]] <= sram_read_data[63:32];
      tag1[temp_adr[8:3]] <= temp_adr[18:9];
    end
  end
```

Figure 38 if all sections are valid.

Writing Data

To write data, data is directly sent to the memory (as our caching system is write-through) and the table is not fundamentally updated. The only update that might occur in the cache is to clear the valid bit of the corresponding entry that is written over in the memory. The code is shown in Figure 39.

```
else if(wr_en & hit & sram_ready)
  if(hit0)
    v0[temp_adr[8:3]] <= 0;
  else
    v1[temp_adr[8:3]] <= 0;
```

Figure 39 writing to memory via cache.

LAB Results

Simulation Results

The clock is 10ps and in all parts of the simulation, the rst signal is cleared in t = 200ps.

Phase 1: ARM

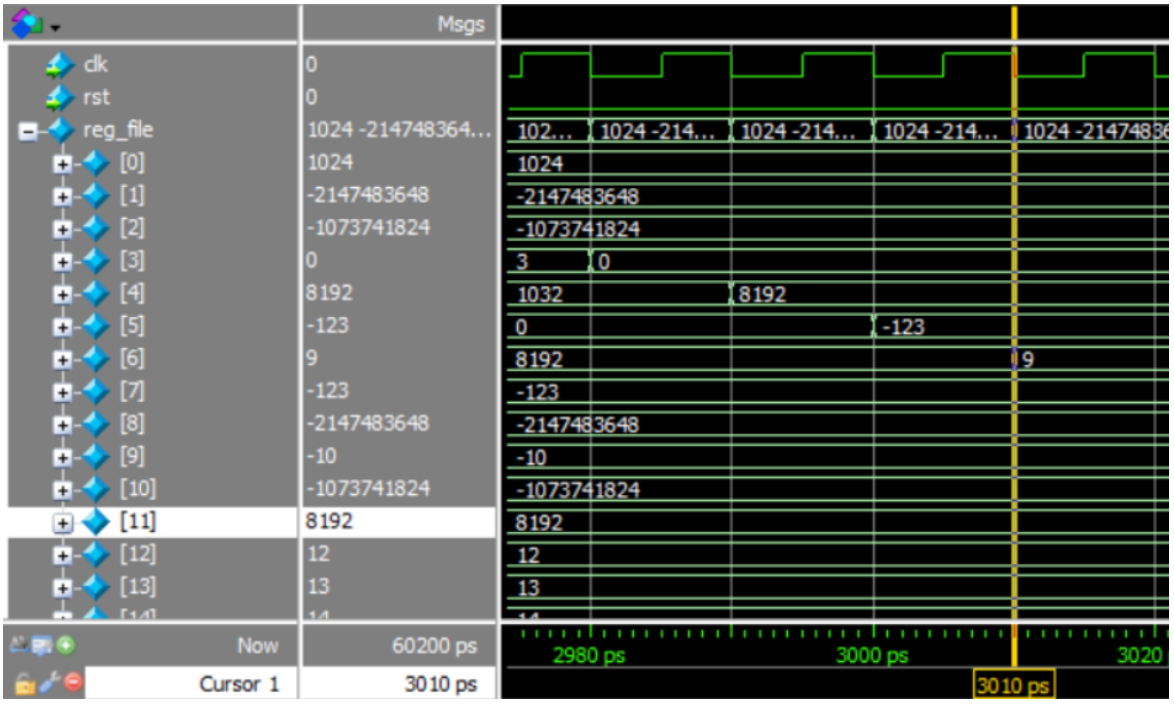

Figure 40.

As it is seen in the simulation, the process ends in t = 3010ps and all results are sorted.

Phase 2: Forwarding

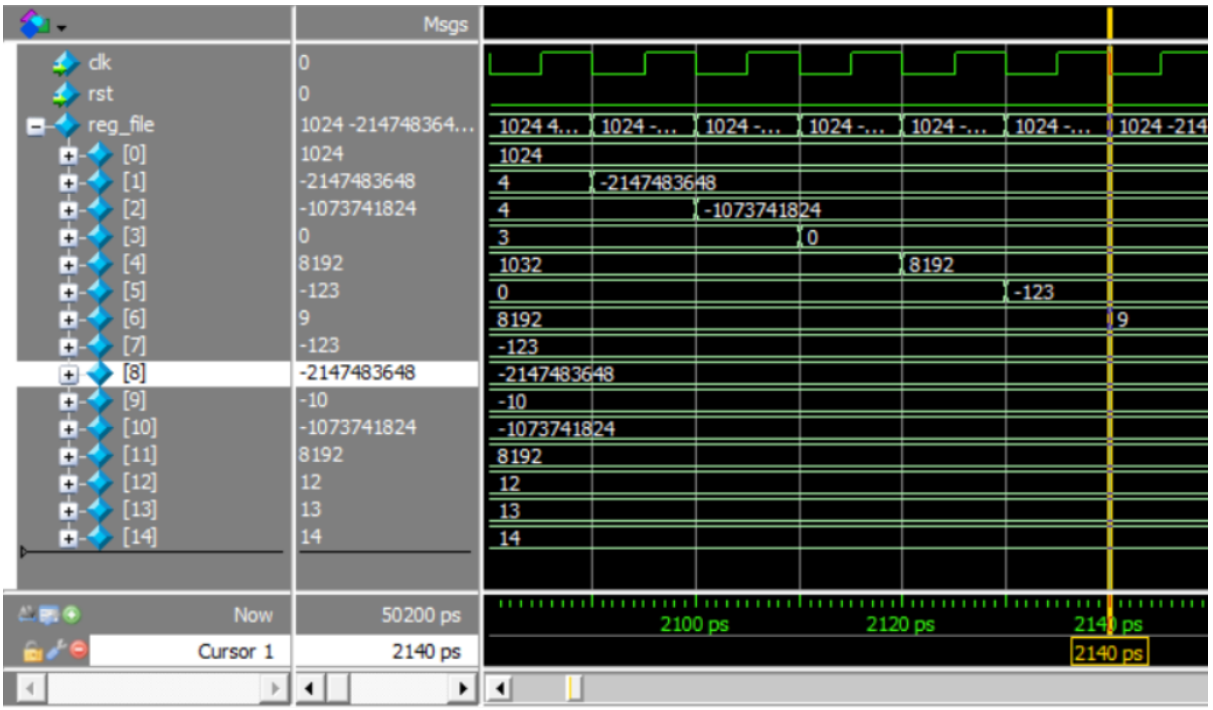

Figure 41.

As it is seen in the simulation, the process ends in t = 2140ps and all results are sorted. The time taken is considerably shorter than in the previous section.

Phase 3: SRAM

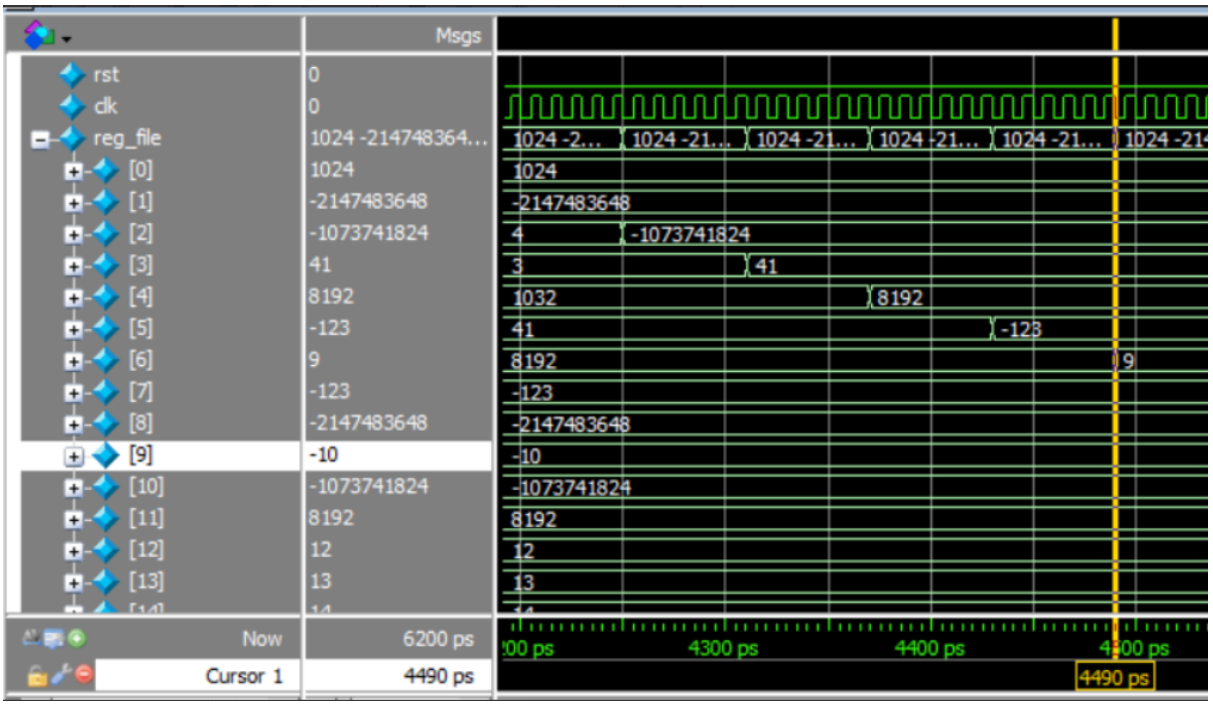

Figure 42.

As it is seen in the simulation, the process ends in t = 4490ps and all results are sorted. The time taken is considerably longer than in the previous section. This is since each memory access takes 6 clocks instead of 1 clock to be executed properly.

Phase 4: Caching

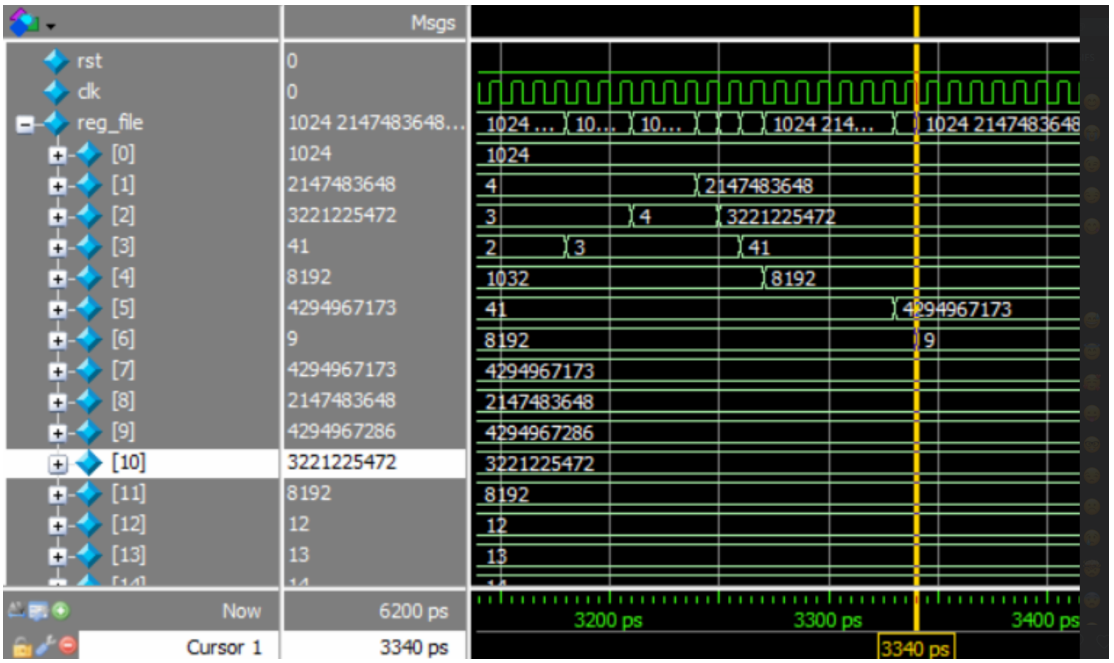

Figure 43.

As it is seen in the simulation, the process ends in t = 3340ps and all results are sorted. The time taken is shorter than SRAM alone and it means that caching is effective.

Synthesize Results

Phase 1: ARM

Top-level Entity Name	ARMCPU
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	4,498 / 33,216 (14 %)
Total combinational functions	4,292 / 33,216 (13 %)
Dedicated logic registers	2,127 / 33,216 (6 %)
Total registers	2127
Total pins	418 / 475 (88 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Figure 44.

Phase 2: Forwarding

Revision Name	ARMCPU
Top-level Entity Name	ARMCPU
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	4,608 / 33,216 (14 %)
Total combinational functions	4,476 / 33,216 (13 %)
Dedicated logic registers	2,131 / 33,216 (6 %)
Total registers	2131
Total pins	418 / 475 (88 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Figure 45.

Phase 3: SRAM

Revision Name	ARMCPU
Top-level Entity Name	ARMCPU
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	2,235 / 33,216 (7 %)
Total combinational functions	2,066 / 33,216 (6 %)
Dedicated logic registers	797 / 33,216 (2 %)
Total registers	797
Total pins	418 / 475 (88 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Figure 46.

Phase 4: Caching

Revision Name	ARMCPU
Top-level Entity Name	ARMCPU
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	18,797 / 33,216 (57 %)
Total combinational functions	18,558 / 33,216 (56 %)
Dedicated logic registers	10,493 / 33,216 (32 %)
Total registers	10493
Total pins	418 / 475 (88 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Figure 47.

Calculations

We know that clock takes 10ps and rst is cleared in t = 200ps. Also, there are 47 instructions after the execution is done.
We have used the following formula to calculate CPI:

$$CPI = \frac{number\ of\ cycles}{number\ of\ instructions}$$

Phase 1: ARM

It took $3010 - 200 = 2810ps$ to execute 47 instructions. Thus we have:

$$cycles_{arm} = \frac{2810}{10} = 281cycles$$

And thus:

$$CPI_{arm} = \frac{281}{47} = 5.97$$

Phase 2: Forwarding

It took $2140 - 200 = 1940ps$ to execute 47 instructions. Thus we have:

$$cycles_{forwarding} = \frac{1940}{10} = 194cycles$$

And thus:

$$CPI_{forwarding} = \frac{194}{47} = 4.13$$

This means that by using the forwarding technique, we increase in our performance:

$$\frac{cycles_{arm}}{cycles_{forwarding}} = \frac{281}{194} = 1.44 = 44\%\ increase\ in\ performance$$

Phase 3: SRAM

It took $4490 - 200 = 1940ps$ to execute 47 instructions. Thus we have:

$$cycles_{SRAM} = \frac{4290}{10} = 429cycles$$

And thus:

$$CPI_{SRAM} = \frac{429}{47} = 9.13$$

This means that by using SRAM, the performance decreased compared to the previous section:

$$\frac{cycles_{forwarding}}{cycles_{SRAM}} = \frac{194}{429} = 0.45 = 54\% \; decrease \; in \; performance$$

Phase 4: Caching

It took $3340 - 200 = 3140ps$ to execute 47 instructions. Thus we have:

$$cycles_{caching} = \frac{3140}{10} = 314cycles$$

And thus:

$$CPI_{caching} = \frac{314}{47} = 6.68$$

This means that by using caching, the performance increased compared to the previous section:

$$\frac{cycles_{SRAM}}{cycles_{caching}} = \frac{429}{314} = 1.36 = 36\% \; increase \; in \; performance$$