EE446 Project Report: Pipelined Processor Architecture with Hazard Unit

Noyan Erdin Kilic

Introduction

This report presents a Verilog implementation of the pipelined processor with a hazard unit and a branch predictor as explained in the project document. Important code blocks, their explanations, instructions that are loaded in the memory and test results are given.

Datapath

The datapath implementation follows the same design in the lecture notes. Figure 1 shows the schematic of the implemented design.

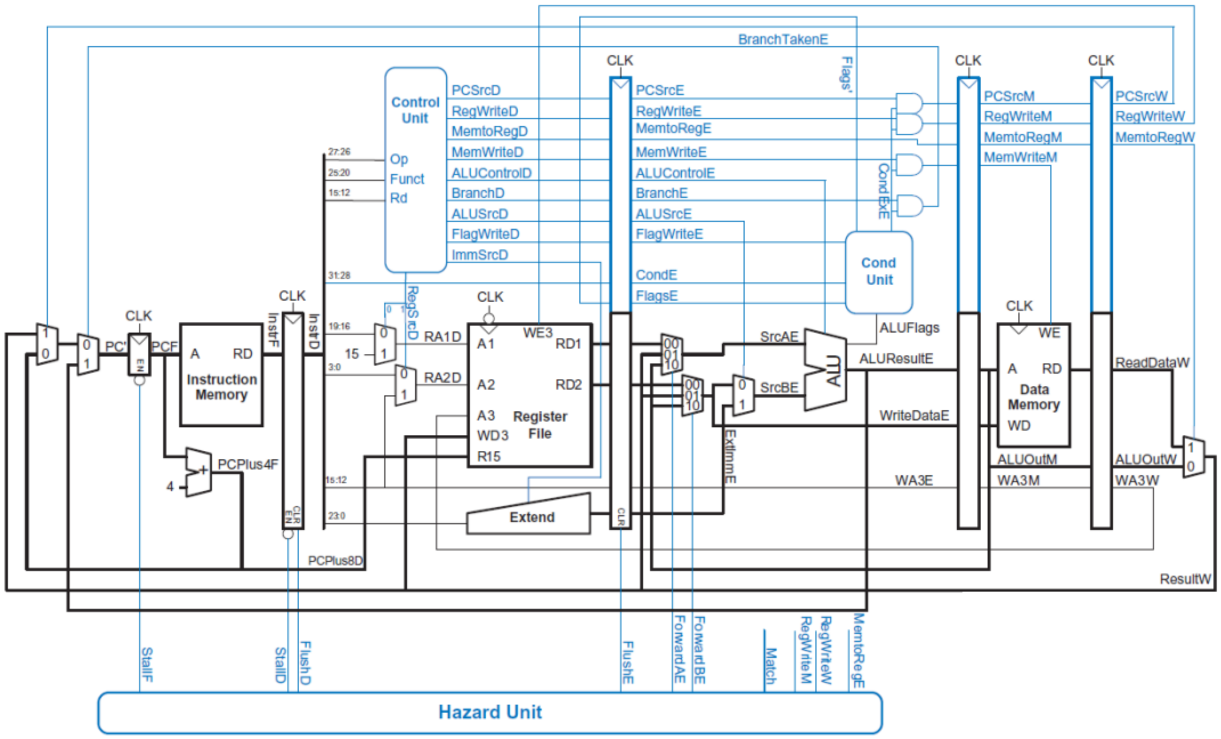
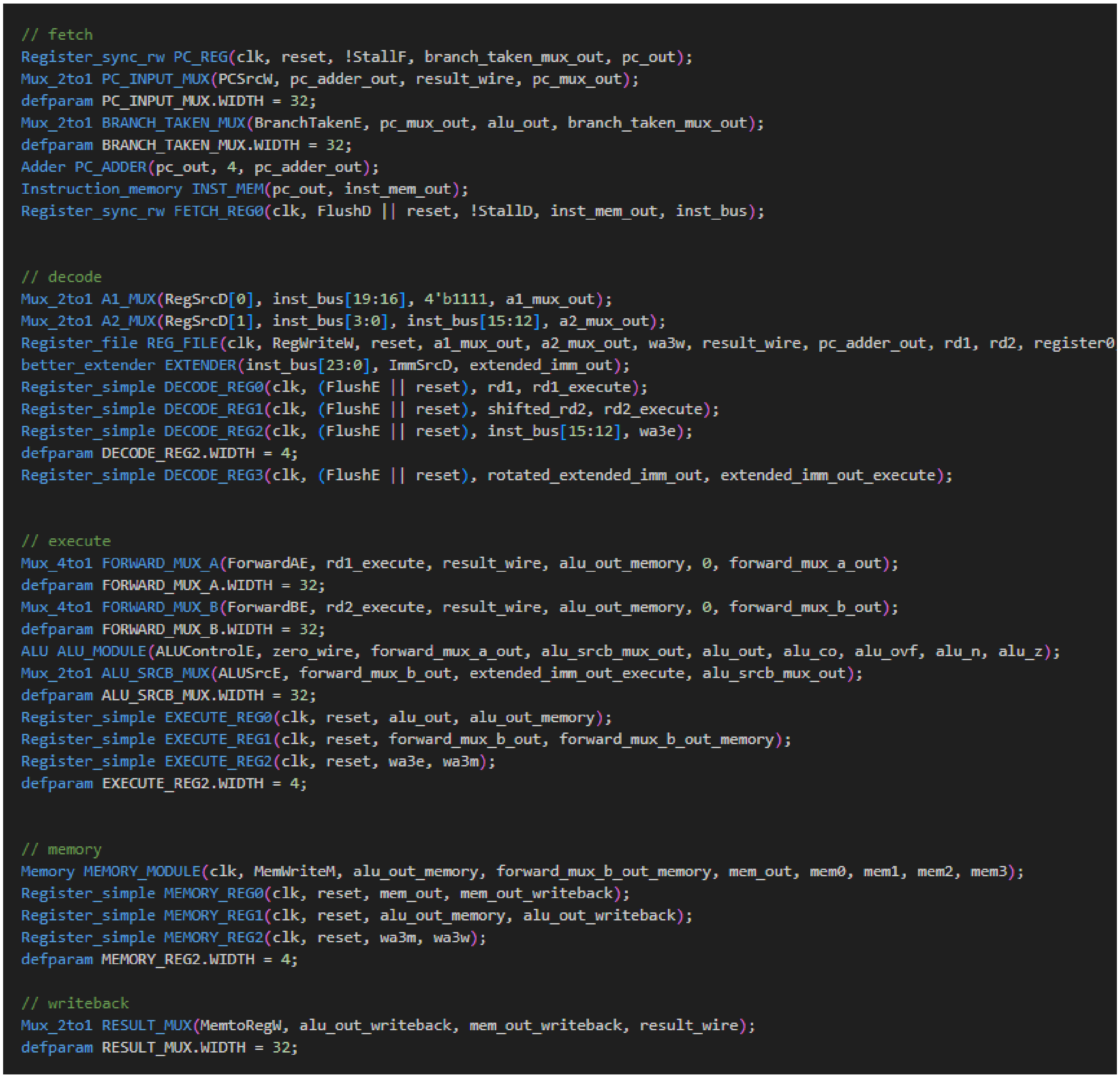


Figure 1: Datapath schematic used for the design.

Implementation was done with Verilog HDL, instead of the Schematic Editor in Quartus. The code snippet of the pipelined stages of the datapath can be seen in Figure 2.



```
// fetch
Register_sync_rw PC_REG(clk, reset, !StallF, branch_taken_mux_out, pc_out);
Mux_2to1 PC_INPUT_MUX(PCSrcW, pc_adder_out, result_wire, pc_mux_out);
defparam PC_INPUT_MUX.WIDTH = 32;
Mux_2to1 BRANCH_TAKEN_MUX(BranchTakenE, pc_mux_out, alu_out, branch_taken_mux_out);
defparam BRANCH_TAKEN_MUX.WIDTH = 32;
Adder PC_ADDER(pc_out, 4, pc_adder_out);
Instruction_memory INST_MEM(pc_out, inst_mem_out);
Register_sync_rw FETCH_REG0(clk, FlushD || reset, !StallD, inst_mem_out, inst_bus);


// decode
Mux_2to1 A1_MUX(RegSrcD[0], inst_bus[19:16], 4'b1111, a1_mux_out);
Mux_2to1 A2_MUX(RegSrcD[1], inst_bus[3:0], inst_bus[15:12], a2_mux_out);
Register_file REG_FILE(clk, RegWriteW, reset, a1_mux_out, a2_mux_out, wa3w, result_wire, pc_adder_out, rd1, rd2, register0
better_extender EXTENDER(inst_bus[23:0], ImmSrcD, extended_imm_out);
Register_simple DECODE_REG0(clk, (FlushE || reset), rd1, rd1_execute);
Register_simple DECODE_REG1(clk, (FlushE || reset), shifted_rd2, rd2_execute);
Register_simple DECODE_REG2(clk, (FlushE || reset), inst_bus[15:12], wa3e);
defparam DECODE_REG2.WIDTH = 4;
Register_simple DECODE_REG3(clk, (FlushE || reset), rotated_extended_imm_out, extended_imm_out_execute);


// execute
Mux_4to1 FORWARD_MUX_A(ForwardAE, rd1_execute, result_wire, alu_out_memory, 0, forward_mux_a_out);
defparam FORWARD_MUX_A.WIDTH = 32;
Mux_4to1 FORWARD_MUX_B(ForwardBE, rd2_execute, result_wire, alu_out_memory, 0, forward_mux_b_out);
defparam FORWARD_MUX_B.WIDTH = 32;
ALU ALU_MODULE(ALUControlE, zero_wire, forward_mux_a_out, alu_srcb_mux_out, alu_out, alu_co, alu_ovf, alu_n, alu_z);
Mux_2to1 ALU_SRCB_MUX(ALUSrcE, forward_mux_b_out, extended_imm_out_execute, alu_srcb_mux_out);
defparam ALU_SRCB_MUX.WIDTH = 32;
Register_simple EXECUTE_REG0(clk, reset, alu_out, alu_out_memory);
Register_simple EXECUTE_REG1(clk, reset, forward_mux_b_out, forward_mux_b_out_memory);
Register_simple EXECUTE_REG2(clk, reset, wa3e, wa3m);
defparam EXECUTE_REG2.WIDTH = 4;


// memory
Memory MEMORY_MODULE(clk, MemWriteM, alu_out_memory, forward_mux_b_out_memory, mem_out, mem0, mem1, mem2, mem3);
Register_simple MEMORY_REG0(clk, reset, mem_out, mem_out_writeback);
Register_simple MEMORY_REG1(clk, reset, alu_out_memory, alu_out_writeback);
Register_simple MEMORY_REG2(clk, reset, wa3m, wa3w);
defparam MEMORY_REG2.WIDTH = 4;

// writeback
Mux_2to1 RESULT_MUX(MemtoRegW, alu_out_writeback, mem_out_writeback, result_wire);
defparam RESULT_MUX.WIDTH = 32;
```

Figure 2: Verilog HDL implementation of pipeline stages.

As a modification made on the schematic in Figure 1 is adding the register file inputs RA1D and RA2D to the execute stage registers. These address buses for the registers are propagated to the next stage. This is done in order for the hazard unit to function properly, it needs to have the information of whether there is a match between the registers that are being read at the decode stage and the write address in the memory stage (wa3m). Two signals are used in the hazard unit, namely "Match_1E_M" and "Match_2E_M".

This modification allows for easier implementation of the hazard unit, since it was directly taken from the lecture notes as explained later in the document. The modified schematic of the datapath can be seen in Figure 3.

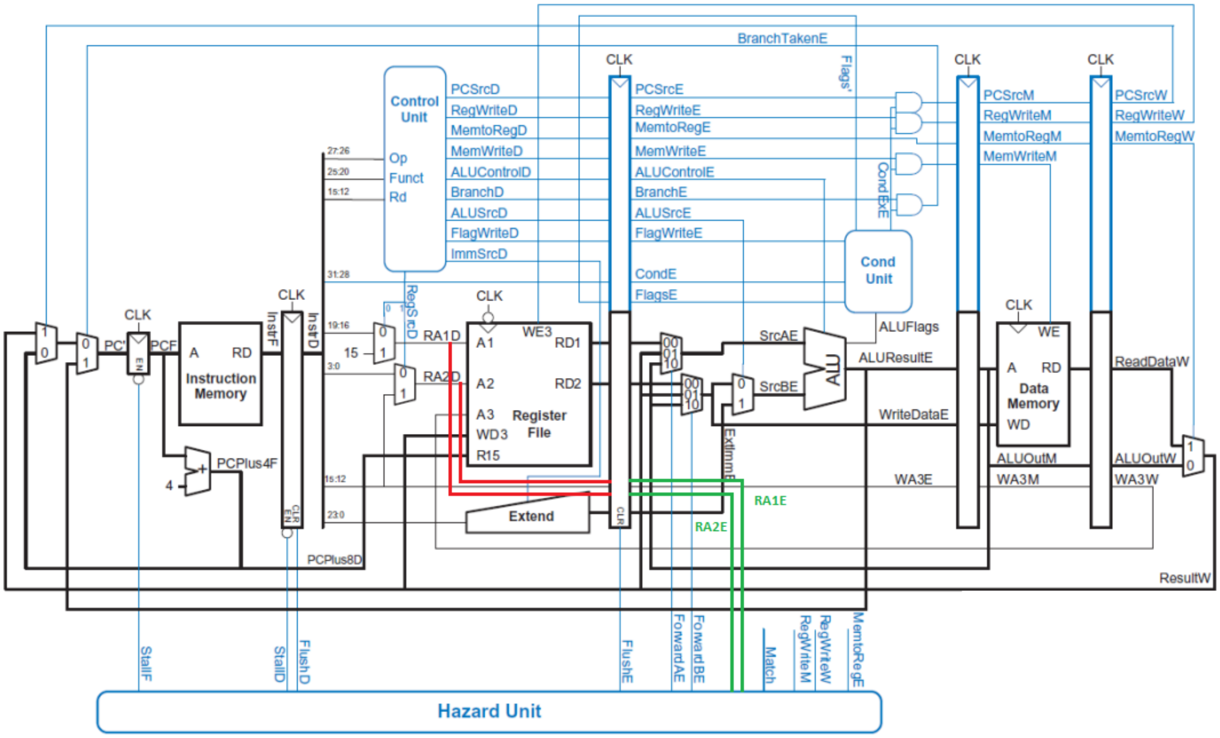


Figure 3: Modified datapath.

In Figure 3, the newly added inputs and outputs of the execute stage register are shown in red and green colors respectively. Naming used for the outputs are RA1E and RA2E, which are consistent with the lecture notes.

Content of the register file registers and data memory are set as output ports for the datapath, for easier demonstration.

Hazard Unit

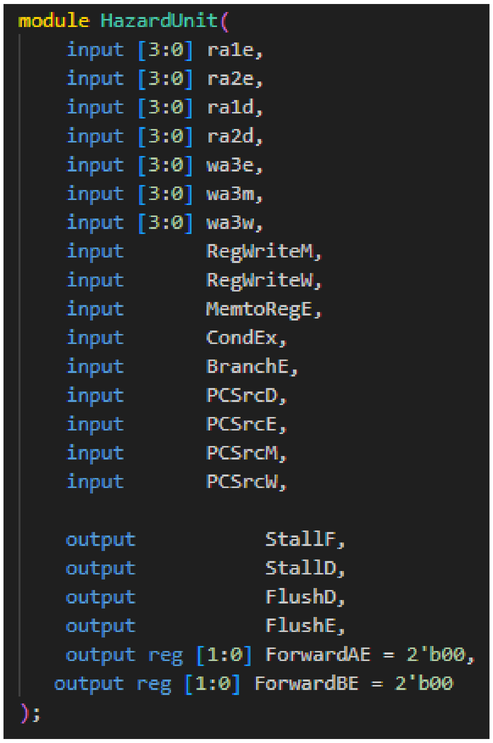
The hazard unit implementation strictly follows the one in the lecture notes. Figure 4 shows the relation between signals outputted from the datapath/controller and the hazard unit outputs. The hazard unit's inputs and outputs are listed in Figure 5.

```
Match_1E_M = (RA1E == WA3M)
Match_1E_W = (RA1E == WA3W)
if (Match_1E_M • RegWriteM)
ForwardAE = 10; // SrcAE = ALUOutM
else if (Match_1E_W • RegWriteW)
ForwardAE = 01; // SrcAE = ResultW
else ForwardAE = 00; // SrcAE from regfile
```

```
Match_2E_M = (RA2E == WA3M)
Match_2E_W = (RA2E == WA3W)
if (Match_2E_M • RegWriteM)
ForwardBE = 10; // SrcBE = ALUOutM
else if (Match_2E_W • RegWriteW)
ForwardBE = 01; // SrcBE = ResultW
else ForwardBE = 00; // SrcBE from regfile
(SrcBE is selected from ExtImmE and regfile
with another MUX)
```

```
Match_12D_E = (RA1D == WA3E) + (RA2D == WA3E)
LDRstall = Match_12D_E • MemtoRegE
BranchTakenE = BranchE • CondEx

PCWrPendingF = PCSrcD + PCSrcE + PCSrcM; Fetch is stalled, Decode is Flushed
StallF = LDRstall + PCWrPendingF; Not asserted during PCSrcW to allow the write
StallD = LDRstall
FlushD = PCWrPendingF + PCSrcW + BranchTakenE; Asserted as long as PC Write is going on
or Branch is taken
FlushE = LDRstall + BranchTakenE;
```

Figure 4: Description of hazard unit signals.

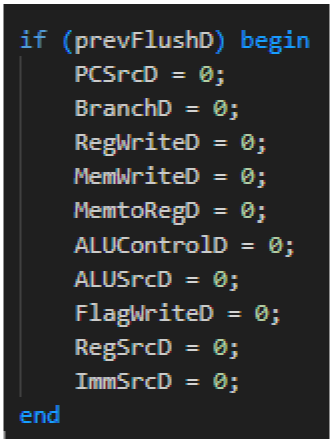


```
module HazardUnit(
    input [3:0] ra1e,
    input [3:0] ra2e,
    input [3:0] ra1d,
    input [3:0] ra2d,
    input [3:0] wa3e,
    input [3:0] wa3m,
    input [3:0] wa3w,
    input       RegWriteM,
    input       RegWriteW,
    input       MemtoRegE,
    input       CondEx,
    input       BranchE,
    input       PCSrcD,
    input       PCSrcE,
    input       PCSrcM,
    input       PCSrcW,

    output           StallF,
    output           StallD,
    output           FlushD,
    output           FlushE,
    output reg [1:0] ForwardAE = 2'b00,
   output reg [1:0] ForwardBE = 2'b00
);
```

Figure 5: Hazard unit input/output list.

One extra modification made to the operation of the hazard unit is flushing of the decode stage register. Normally, only the decode registers are flushed (set to 0). This sets the instruction bus

(INSTD in Figure 3) to 0. The controller treats this instruction as an actual instruction, and the control signals for the “ANDEQ R0, R0, R0, LSL #0” instruction. Also, if the controller treats the flushed decode stage as an actual instruction, the ALU flags could be modified in the next cycle. This behavior is not desired, therefore when the FlushD signal is “1”, all the control signals for the decode stage are set to 0 manually. Figure 6 shows the code for this modification, and it is implemented inside the controller module.



```
if (prevFlushD) begin
    PCSrcD = 0;
    BranchD = 0;
    RegWriteD = 0;
    MemWriteD = 0;
    MemtoRegD = 0;
    ALUControlD = 0;
    ALUSrcD = 0;
    FlagWriteD = 0;
    RegSrcD = 0;
    ImmSrcD = 0;
end
```

Figure 6: Manual flushing of the control signals.

The prevFlushD signal is the FlushD signal in the previous clock cycle. FlushE is not used instead, since in branch instructions, decode and execute stages are flushed together. A separate latch is used for this reason.

Controller

The controller for the designed processor uses ARM32 instruction format. Bits of the instructions are shown in Figure 7.

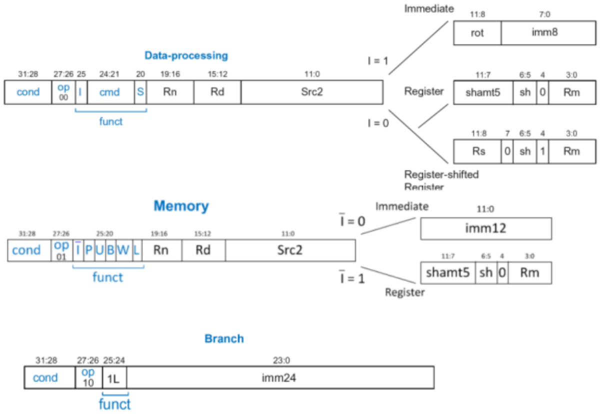
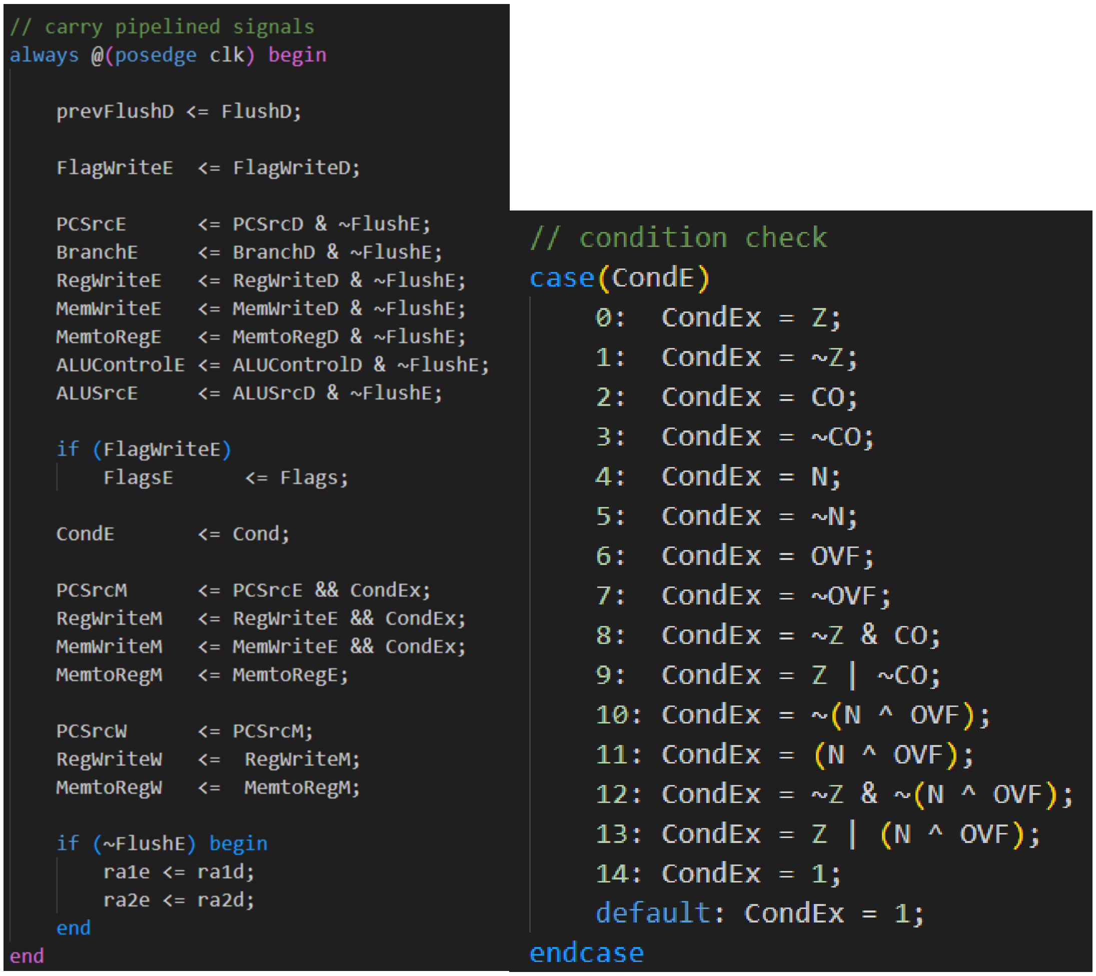


Figure 7: ARM32 instruction format.

Parsing of the instruction is done in the datapath and inputted to the controller. The controller's purpose is to decide on the necessary control signals' values. Based on the opcode of the instruction, ALU flags, condition for the instruction etc. the controller generates the necessary signals.

The propagation of the control signals is done in the controller, instead of the datapath. This can be seen in Figure 8.



```
// carry pipelined signals
always @(posedge clk) begin

    prevFlushD <= FlushD;

    FlagWriteE  <= FlagWriteD;

    PCSrcE      <= PCSrcD & ~FlushE;
    BranchE     <= BranchD & ~FlushE;
    RegWriteE   <= RegWriteD & ~FlushE;
    MemWriteE   <= MemWriteD & ~FlushE;
    MemtoRegE   <= MemtoRegD & ~FlushE;
    ALUControlE <= ALUControlD & ~FlushE;
    ALUSrcE     <= ALUSrcD & ~FlushE;

    if (FlagWriteE)
        FlagsE      <= Flags;

    CondE       <= Cond;

    PCSrcM      <= PCSrcE && CondEx;
    RegWriteM   <= RegWriteE && CondEx;
    MemWriteM   <= MemWriteE && CondEx;
    MemtoRegM   <= MemtoRegE;

    PCSrcW      <= PCSrcM;
    RegWriteW   <=  RegWriteM;
    MemtoRegW   <=  MemtoRegM;

    if (~FlushE) begin
        ra1e <= ra1d;
        ra2e <= ra2d;
    end
end
```

```
// condition check
case(CondE)
    0:  CondEx = Z;
    1:  CondEx = ~Z;
    2:  CondEx = CO;
    3:  CondEx = ~CO;
    4:  CondEx = N;
    5:  CondEx = ~N;
    6:  CondEx = OVF;
    7:  CondEx = ~OVF;
    8:  CondEx = ~Z & CO;
    9:  CondEx = Z | ~CO;
    10: CondEx = ~(N ^ OVF);
    11: CondEx = (N ^ OVF);
    12: CondEx = ~Z & ~(N ^ OVF);
    13: CondEx = Z | (N ^ OVF);
    14: CondEx = 1;
    default: CondEx = 1;
endcase
```

Figures 8 / 9: Propagation of the control signals / Condition check.

The condition check (CondEx signal) is also written in the controller module. Implementation can be seen in Figure 9.

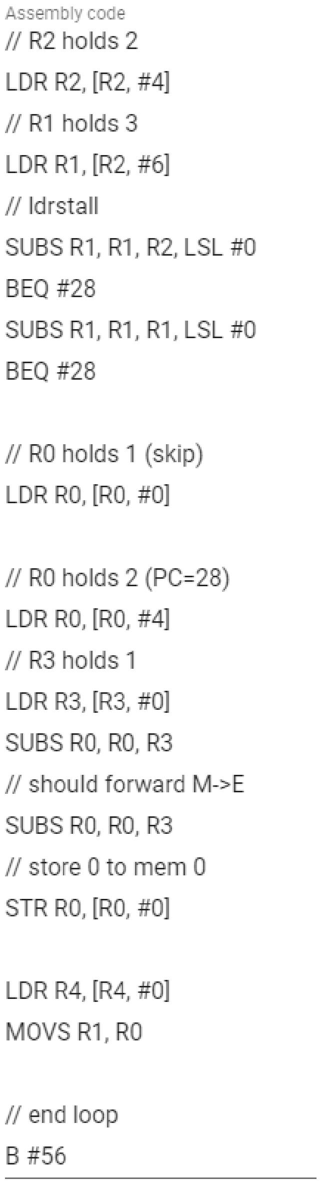
Top-level module

The top-level module simply connects previously explained modules. Inputs and outputs are adjusted for easy demonstration.

Testbench

The test for the top-level module is implemented with the Cocotb Python framework. It provides debug output, prints nearly all of the signals in the processor, and is formatted for easy tracing of the processor's operation.

The instructions buried into the instruction memory of the processor are shown in Figure 10.



Assembly code

```
// R2 holds 2
LDR R2, [R2, #4]
// R1 holds 3
LDR R1, [R2, #6]
// ldrstall
SUBS R1, R1, R2, LSL #0
BEQ #28
SUBS R1, R1, R1, LSL #0
BEQ #28

// R0 holds 1 (skip)
LDR R0, [R0, #0]

// R0 holds 2 (PC=28)
LDR R0, [R0, #4]
// R3 holds 1
LDR R3, [R3, #0]
SUBS R0, R0, R3
// should forward M->E
SUBS R0, R0, R3
// store 0 to mem 0
STR R0, [R0, #0]

LDR R4, [R4, #0]
MOVS R1, R0

// end loop
B #56
```

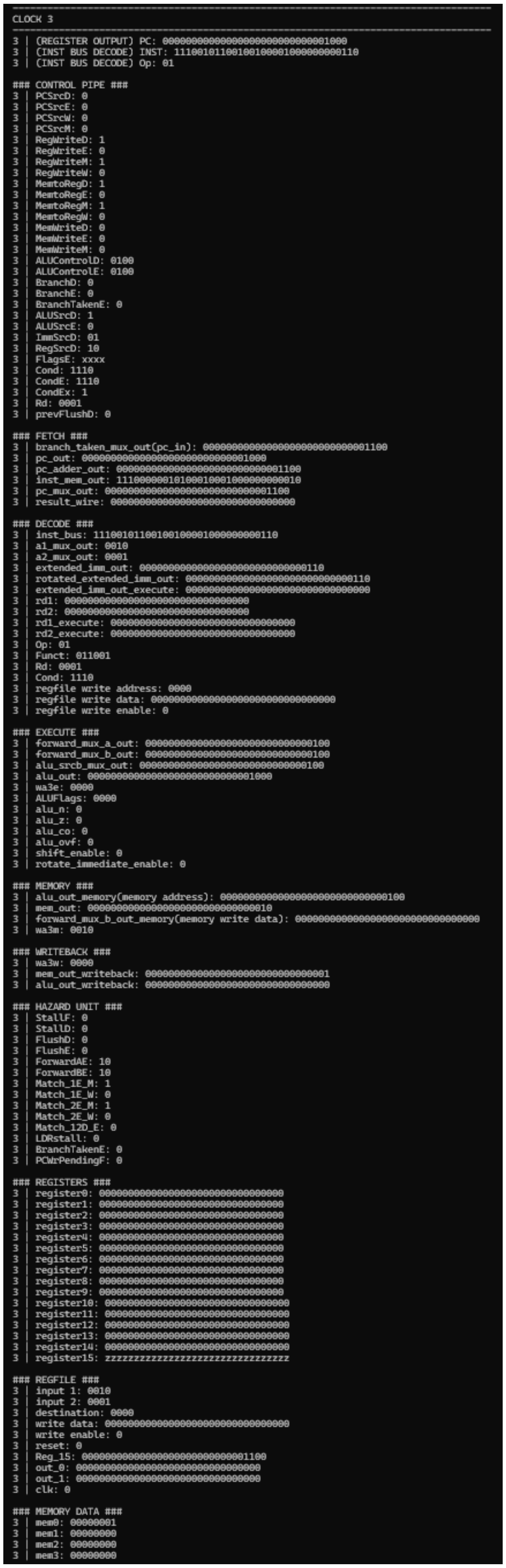
Figure 10: The program for testing the processor.

This small program demonstrates most of the functionality of the processor. Explanations for the operation are given as comments in Figure 10.

On the first LDR instructions, the hazard unit stalls the fetch and decode stages. Later on, in the SUBS instruction the hazard unit intervenes in the operation of the pipeline again. These allow for a complete test for the hazard unit.

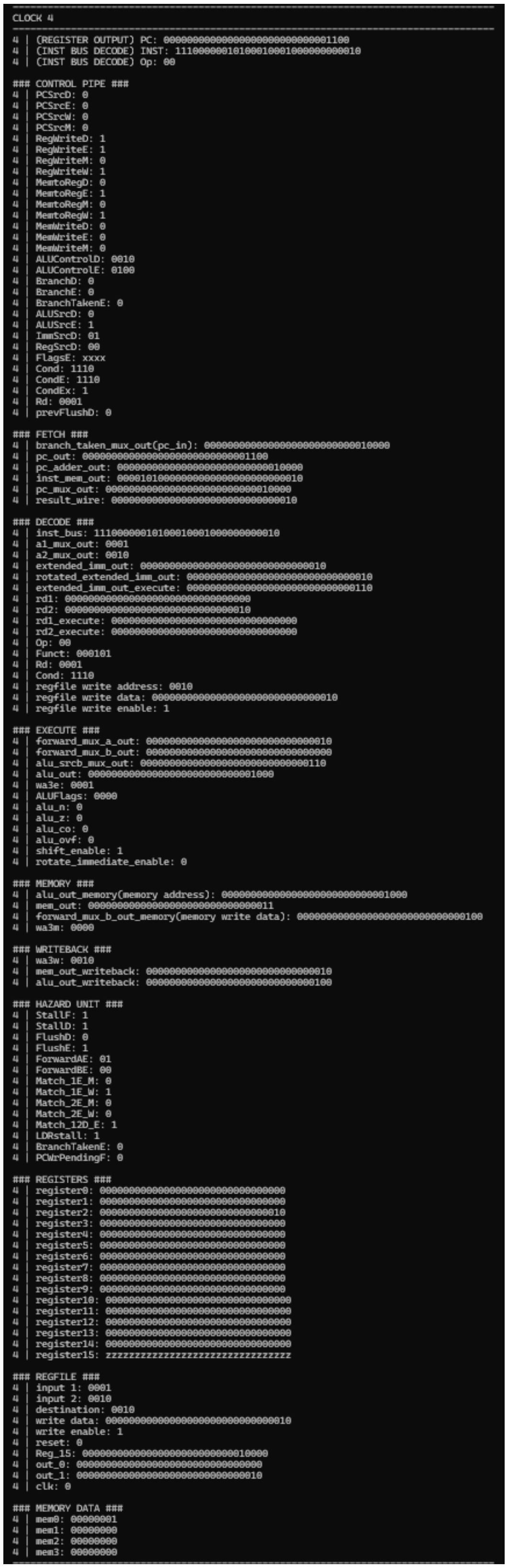
Different instructions (data processing, memory and branch) are used within the program, and also different conditions are paired with those instructions to test the controller. There are no individual unit tests for the datapath components, this is rather a test for the higher level modules (controller, datapath and hazard unit), and it is also an integration test.

A few clock cycles' debug output of the test is shown in Figures 11, 12, 13 and 14. The hazard unit, controller and datapath signals can be traced in these Figures.



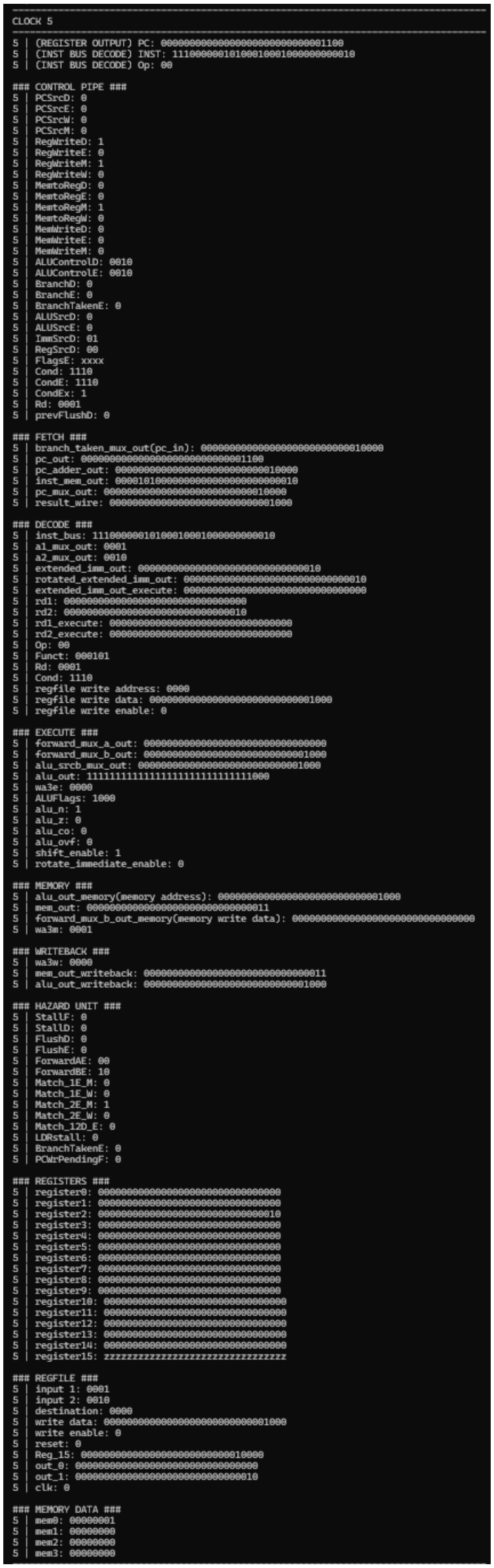
```
================================================================================
CLOCK 3
================================================================================
3 | (REGISTER OUTPUT) PC: 00000000000000000000000000001000
3 | (INST BUS DECODE) INST: 11100101100100100001000000000110
3 | (INST BUS DECODE) Op: 01

### CONTROL PIPE ###
3 | PCSrcD: 0
3 | PCSrcE: 0
3 | PCSrcW: 0
3 | PCSrcM: 0
3 | RegWriteD: 1
3 | RegWriteE: 0
3 | RegWriteM: 1
3 | RegWriteW: 0
3 | MemtoRegD: 1
3 | MemtoRegE: 0
3 | MemtoRegM: 1
3 | MemtoRegW: 0
3 | MemWriteD: 0
3 | MemWriteE: 0
3 | MemWriteM: 0
3 | ALUControlD: 0100
3 | ALUControlE: 0100
3 | BranchD: 0
3 | BranchE: 0
3 | BranchTakenE: 0
3 | ALUSrcD: 1
3 | ALUSrcE: 0
3 | ImmSrcD: 01
3 | RegSrcD: 10
3 | FlagsE: xxxx
3 | Cond: 1110
3 | CondE: 1110
3 | CondEx: 1
3 | Rd: 0001
3 | prevFlushD: 0

### FETCH ###
3 | branch_taken_mux_out(pc_in): 00000000000000000000000000001100
3 | pc_out: 00000000000000000000000000001000
3 | pc_adder_out: 00000000000000000000000000001100
3 | inst_mem_out: 11100000010100010001000000000010
3 | pc_mux_out: 00000000000000000000000000001100
3 | result_wire: 00000000000000000000000000000000

### DECODE ###
3 | inst_bus: 11100101100100100001000000000110
3 | a1_mux_out: 0010
3 | a2_mux_out: 0001
3 | extended_imm_out: 00000000000000000000000000000110
3 | rotated_extended_imm_out: 00000000000000000000000000000110
3 | extended_imm_out_execute: 00000000000000000000000000000000
3 | rd1: 00000000000000000000000000000000
3 | rd2: 00000000000000000000000000000000
3 | rd1_execute: 00000000000000000000000000000000
3 | rd2_execute: 00000000000000000000000000000000
3 | Op: 01
3 | Funct: 011001
3 | Rd: 0001
3 | Cond: 1110
3 | regfile write address: 0000
3 | regfile write data: 00000000000000000000000000000000
3 | regfile write enable: 0

### EXECUTE ###
3 | forward_mux_a_out: 00000000000000000000000000000100
3 | forward_mux_b_out: 00000000000000000000000000000100
3 | alu_srcb_mux_out: 00000000000000000000000000000100
3 | alu_out: 00000000000000000000000000001000
3 | wa3e: 0000
3 | ALUFlags: 0000
3 | alu_n: 0
3 | alu_z: 0
3 | alu_co: 0
3 | alu_ovf: 0
3 | shift_enable: 0
3 | rotate_immediate_enable: 0

### MEMORY ###
3 | alu_out_memory(memory address): 00000000000000000000000000000100
3 | mem_out: 00000000000000000000000000000010
3 | forward_mux_b_out_memory(memory write data): 00000000000000000000000000000000
3 | wa3m: 0010

### WRITEBACK ###
3 | wa3w: 0000
3 | mem_out_writeback: 00000000000000000000000000000001
3 | alu_out_writeback: 00000000000000000000000000000000

### HAZARD UNIT ###
3 | StallF: 0
3 | StallD: 0
3 | FlushD: 0
3 | FlushE: 0
3 | ForwardAE: 10
3 | ForwardBE: 10
3 | Match_1E_M: 1
3 | Match_1E_W: 0
3 | Match_2E_M: 1
3 | Match_2E_W: 0
3 | Match_12D_E: 0
3 | LDRstall: 0
3 | BranchTakenE: 0
3 | PCWrPendingF: 0

### REGISTERS ###
3 | register0: 00000000000000000000000000000000
3 | register1: 00000000000000000000000000000000
3 | register2: 00000000000000000000000000000000
3 | register3: 00000000000000000000000000000000
3 | register4: 00000000000000000000000000000000
3 | register5: 00000000000000000000000000000000
3 | register6: 00000000000000000000000000000000
3 | register7: 00000000000000000000000000000000
3 | register8: 00000000000000000000000000000000
3 | register9: 00000000000000000000000000000000
3 | register10: 00000000000000000000000000000000
3 | register11: 00000000000000000000000000000000
3 | register12: 00000000000000000000000000000000
3 | register13: 00000000000000000000000000000000
3 | register14: 00000000000000000000000000000000
3 | register15: zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz

### REGFILE ###
3 | input 1: 0010
3 | input 2: 0001
3 | destination: 0000
3 | write data: 00000000000000000000000000000000
3 | write enable: 0
3 | reset: 0
3 | Reg_15: 00000000000000000000000000001100
3 | out_0: 00000000000000000000000000000000
3 | out_1: 00000000000000000000000000000000
3 | clk: 0

### MEMORY DATA ###
3 | mem0: 00000001
3 | mem1: 00000000
3 | mem2: 00000000
3 | mem3: 00000000
```

Figure 11: 3rd clock cycle of the test.



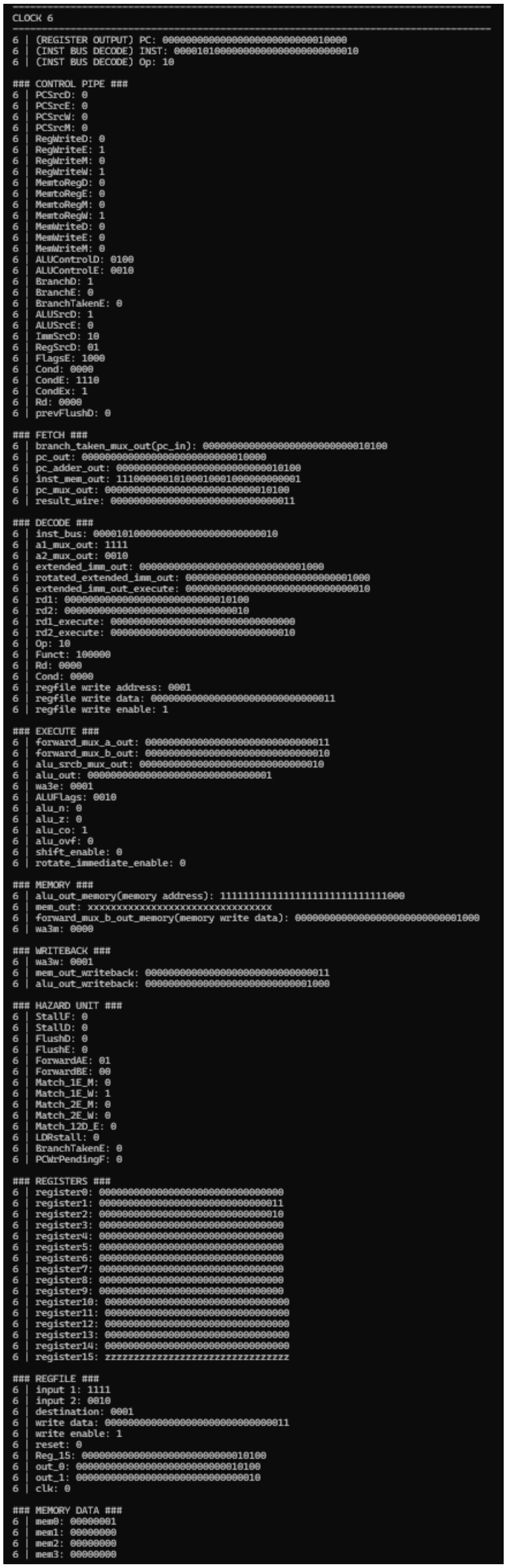
```
--------------------------------------------------------------------------------
CLOCK 4
--------------------------------------------------------------------------------
4 | (REGISTER OUTPUT) PC: 00000000000000000000000000001100
4 | (INST BUS DECODE) INST: 11100000010100010001000000000010
4 | (INST BUS DECODE) Op: 00

### CONTROL PIPE ###
4 | PCSrcD: 0
4 | PCSrcE: 0
4 | PCSrcW: 0
4 | PCSrcM: 0
4 | RegWriteD: 1
4 | RegWriteE: 1
4 | RegWriteM: 0
4 | RegWriteW: 1
4 | MemtoRegD: 0
4 | MemtoRegE: 1
4 | MemtoRegM: 0
4 | MemtoRegW: 1
4 | MemWriteD: 0
4 | MemWriteE: 0
4 | MemWriteM: 0
4 | ALUControlD: 0010
4 | ALUControlE: 0100
4 | BranchD: 0
4 | BranchE: 0
4 | BranchTakenE: 0
4 | ALUSrcD: 0
4 | ALUSrcE: 1
4 | ImmSrcD: 01
4 | RegSrcD: 00
4 | FlagsE: xxxx
4 | Cond: 1110
4 | CondE: 1110
4 | CondEx: 1
4 | Rd: 0001
4 | prevFlushD: 0

### FETCH ###
4 | branch_taken_mux_out(pc_in): 00000000000000000000000000010000
4 | pc_out: 00000000000000000000000000001100
4 | pc_adder_out: 00000000000000000000000000010000
4 | inst_mem_out: 00001010000000000000000000000010
4 | pc_mux_out: 00000000000000000000000000010000
4 | result_wire: 00000000000000000000000000000010

### DECODE ###
4 | inst_bus: 11100000010100010001000000000010
4 | a1_mux_out: 0001
4 | a2_mux_out: 0010
4 | extended_imm_out: 00000000000000000000000000000010
4 | rotated_extended_imm_out: 00000000000000000000000000000010
4 | extended_imm_out_execute: 00000000000000000000000000000110
4 | rd1: 00000000000000000000000000000000
4 | rd2: 00000000000000000000000000000010
4 | rd1_execute: 00000000000000000000000000000000
4 | rd2_execute: 00000000000000000000000000000000
4 | Op: 00
4 | Funct: 000101
4 | Rd: 0001
4 | Cond: 1110
4 | regfile write address: 0010
4 | regfile write data: 00000000000000000000000000000010
4 | regfile write enable: 1

### EXECUTE ###
4 | forward_mux_a_out: 00000000000000000000000000000010
4 | forward_mux_b_out: 00000000000000000000000000000000
4 | alu_srcb_mux_out: 00000000000000000000000000000110
4 | alu_out: 00000000000000000000000000001000
4 | wa3e: 0001
4 | ALUFlags: 0000
4 | alu_n: 0
4 | alu_z: 0
4 | alu_co: 0
4 | alu_ovf: 0
4 | shift_enable: 1
4 | rotate_immediate_enable: 0

### MEMORY ###
4 | alu_out_memory(memory address): 00000000000000000000000000001000
4 | mem_out: 00000000000000000000000000000011
4 | forward_mux_b_out_memory(memory write data): 00000000000000000000000000000100
4 | wa3m: 0000

### WRITEBACK ###
4 | wa3w: 0010
4 | mem_out_writeback: 00000000000000000000000000000010
4 | alu_out_writeback: 00000000000000000000000000000100

### HAZARD UNIT ###
4 | StallF: 1
4 | StallD: 1
4 | FlushD: 0
4 | FlushE: 1
4 | ForwardAE: 01
4 | ForwardBE: 00
4 | Match_1E_M: 0
4 | Match_1E_W: 1
4 | Match_2E_M: 0
4 | Match_2E_W: 0
4 | Match_12D_E: 1
4 | LDRstall: 1
4 | BranchTakenE: 0
4 | PCWrPendingF: 0

### REGISTERS ###
4 | register0: 00000000000000000000000000000000
4 | register1: 00000000000000000000000000000000
4 | register2: 00000000000000000000000000000010
4 | register3: 00000000000000000000000000000000
4 | register4: 00000000000000000000000000000000
4 | register5: 00000000000000000000000000000000
4 | register6: 00000000000000000000000000000000
4 | register7: 00000000000000000000000000000000
4 | register8: 00000000000000000000000000000000
4 | register9: 00000000000000000000000000000000
4 | register10: 00000000000000000000000000000000
4 | register11: 00000000000000000000000000000000
4 | register12: 00000000000000000000000000000000
4 | register13: 00000000000000000000000000000000
4 | register14: 00000000000000000000000000000000
4 | register15: zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz

### REGFILE ###
4 | input 1: 0001
4 | input 2: 0010
4 | destination: 0010
4 | write data: 00000000000000000000000000000010
4 | write enable: 1
4 | reset: 0
4 | Reg_15: 00000000000000000000000000010000
4 | out_0: 00000000000000000000000000000000
4 | out_1: 00000000000000000000000000000010
4 | clk: 0

### MEMORY DATA ###
4 | mem0: 00000001
4 | mem1: 00000000
4 | mem2: 00000000
4 | mem3: 00000000
--------------------------------------------------------------------------------
```

Figure 12: 4th clock cycle of the test.



```
--------------------------------------------------------------------------------
CLOCK 5
--------------------------------------------------------------------------------
5 | (REGISTER OUTPUT) PC: 00000000000000000000000000001100
5 | (INST BUS DECODE) INST: 11100000010100010001000000000010
5 | (INST BUS DECODE) Op: 00

### CONTROL PIPE ###
5 | PCSrcD: 0
5 | PCSrcE: 0
5 | PCSrcW: 0
5 | PCSrcM: 0
5 | RegWriteD: 1
5 | RegWriteE: 0
5 | RegWriteM: 1
5 | RegWriteW: 0
5 | MemtoRegD: 0
5 | MemtoRegE: 0
5 | MemtoRegM: 1
5 | MemtoRegW: 0
5 | MemWriteD: 0
5 | MemWriteE: 0
5 | MemWriteM: 0
5 | ALUControlD: 0010
5 | ALUControlE: 0010
5 | BranchD: 0
5 | BranchE: 0
5 | BranchTakenE: 0
5 | ALUSrcD: 0
5 | ALUSrcE: 0
5 | ImmSrcD: 01
5 | RegSrcD: 00
5 | FlagsE: xxxx
5 | Cond: 1110
5 | CondE: 1110
5 | CondEx: 1
5 | Rd: 0001
5 | prevFlushD: 0

### FETCH ###
5 | branch_taken_mux_out(pc_in): 00000000000000000000000000010000
5 | pc_out: 00000000000000000000000000001100
5 | pc_adder_out: 00000000000000000000000000010000
5 | inst_mem_out: 00001010000000000000000000000010
5 | pc_mux_out: 00000000000000000000000000010000
5 | result_wire: 00000000000000000000000000001000

### DECODE ###
5 | inst_bus: 11100000010100010001000000000010
5 | a1_mux_out: 0001
5 | a2_mux_out: 0010
5 | extended_imm_out: 00000000000000000000000000000010
5 | rotated_extended_imm_out: 00000000000000000000000000000010
5 | extended_imm_out_execute: 00000000000000000000000000000000
5 | rd1: 00000000000000000000000000000000
5 | rd2: 00000000000000000000000000000010
5 | rd1_execute: 00000000000000000000000000000000
5 | rd2_execute: 00000000000000000000000000000000
5 | Op: 00
5 | Funct: 000101
5 | Rd: 0001
5 | Cond: 1110
5 | regfile write address: 0000
5 | regfile write data: 00000000000000000000000000001000
5 | regfile write enable: 0

### EXECUTE ###
5 | forward_mux_a_out: 00000000000000000000000000000000
5 | forward_mux_b_out: 00000000000000000000000000001000
5 | alu_srcb_mux_out: 00000000000000000000000000001000
5 | alu_out: 11111111111111111111111111111000
5 | wa3e: 0000
5 | ALUFlags: 1000
5 | alu_n: 1
5 | alu_z: 0
5 | alu_co: 0
5 | alu_ovf: 0
5 | shift_enable: 1
5 | rotate_immediate_enable: 0

### MEMORY ###
5 | alu_out_memory(memory address): 00000000000000000000000000001000
5 | mem_out: 00000000000000000000000000000011
5 | forward_mux_b_out_memory(memory write data): 00000000000000000000000000000000
5 | wa3m: 0001

### WRITEBACK ###
5 | wa3w: 0000
5 | mem_out_writeback: 00000000000000000000000000000011
5 | alu_out_writeback: 00000000000000000000000000001000

### HAZARD UNIT ###
5 | StallF: 0
5 | StallD: 0
5 | FlushD: 0
5 | FlushE: 0
5 | ForwardAE: 00
5 | ForwardBE: 10
5 | Match_1E_M: 0
5 | Match_1E_W: 0
5 | Match_2E_M: 1
5 | Match_2E_W: 0
5 | Match_12D_E: 0
5 | LDRstall: 0
5 | BranchTakenE: 0
5 | PCWrPendingF: 0

### REGISTERS ###
5 | register0: 00000000000000000000000000000000
5 | register1: 00000000000000000000000000000000
5 | register2: 00000000000000000000000000000010
5 | register3: 00000000000000000000000000000000
5 | register4: 00000000000000000000000000000000
5 | register5: 00000000000000000000000000000000
5 | register6: 00000000000000000000000000000000
5 | register7: 00000000000000000000000000000000
5 | register8: 00000000000000000000000000000000
5 | register9: 00000000000000000000000000000000
5 | register10: 00000000000000000000000000000000
5 | register11: 00000000000000000000000000000000
5 | register12: 00000000000000000000000000000000
5 | register13: 00000000000000000000000000000000
5 | register14: 00000000000000000000000000000000
5 | register15: zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz

### REGFILE ###
5 | input 1: 0001
5 | input 2: 0010
5 | destination: 0000
5 | write data: 00000000000000000000000000001000
5 | write enable: 0
5 | reset: 0
5 | Reg_15: 00000000000000000000000000010000
5 | out_0: 00000000000000000000000000000000
5 | out_1: 00000000000000000000000000000010
5 | clk: 0

### MEMORY DATA ###
5 | mem0: 00000001
5 | mem1: 00000000
5 | mem2: 00000000
5 | mem3: 00000000
```

Figure 13: 5th clock cycle of the test.



```
--------------------------------------------------------------------------------
CLOCK 6
--------------------------------------------------------------------------------
6 | (REGISTER OUTPUT) PC: 00000000000000000000000000010000
6 | (INST BUS DECODE) INST: 00001010000000000000000000000010
6 | (INST BUS DECODE) Op: 10

### CONTROL PIPE ###
6 | PCSrcD: 0
6 | PCSrcE: 0
6 | PCSrcW: 0
6 | PCSrcM: 0
6 | RegWriteD: 0
6 | RegWriteE: 1
6 | RegWriteM: 0
6 | RegWriteW: 1
6 | MemtoRegD: 0
6 | MemtoRegE: 0
6 | MemtoRegM: 0
6 | MemtoRegW: 1
6 | MemWriteD: 0
6 | MemWriteE: 0
6 | MemWriteM: 0
6 | ALUControlD: 0100
6 | ALUControlE: 0010
6 | BranchD: 1
6 | BranchE: 0
6 | BranchTakenE: 0
6 | ALUSrcD: 1
6 | ALUSrcE: 0
6 | ImmSrcD: 10
6 | RegSrcD: 01
6 | FlagsE: 1000
6 | Cond: 0000
6 | CondE: 1110
6 | CondEx: 1
6 | Rd: 0000
6 | prevFlushD: 0

### FETCH ###
6 | branch_taken_mux_out(pc_in): 00000000000000000000000000010100
6 | pc_out: 00000000000000000000000000010000
6 | pc_adder_out: 00000000000000000000000000010100
6 | inst_mem_out: 11100000010100010001000000000001
6 | pc_mux_out: 00000000000000000000000000010100
6 | result_wire: 00000000000000000000000000000011

### DECODE ###
6 | inst_bus: 00001010000000000000000000000010
6 | a1_mux_out: 1111
6 | a2_mux_out: 0010
6 | extended_imm_out: 00000000000000000000000000001000
6 | rotated_extended_imm_out: 00000000000000000000000000001000
6 | extended_imm_out_execute: 00000000000000000000000000000010
6 | rd1: 00000000000000000000000000010100
6 | rd2: 00000000000000000000000000000010
6 | rd1_execute: 00000000000000000000000000000000
6 | rd2_execute: 00000000000000000000000000000010
6 | Op: 10
6 | Funct: 100000
6 | Rd: 0000
6 | Cond: 0000
6 | regfile write address: 0001
6 | regfile write data: 00000000000000000000000000000011
6 | regfile write enable: 1

### EXECUTE ###
6 | forward_mux_a_out: 00000000000000000000000000000011
6 | forward_mux_b_out: 00000000000000000000000000000010
6 | alu_srcb_mux_out: 00000000000000000000000000000010
6 | alu_out: 00000000000000000000000000000001
6 | wa3e: 0001
6 | ALUFlags: 0010
6 | alu_n: 0
6 | alu_z: 0
6 | alu_co: 1
6 | alu_ovf: 0
6 | shift_enable: 0
6 | rotate_immediate_enable: 0

### MEMORY ###
6 | alu_out_memory(memory address): 11111111111111111111111111111000
6 | mem_out: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
6 | forward_mux_b_out_memory(memory write data): 00000000000000000000000000001000
6 | wa3m: 0000

### WRITEBACK ###
6 | wa3w: 0001
6 | mem_out_writeback: 00000000000000000000000000000011
6 | alu_out_writeback: 00000000000000000000000000001000

### HAZARD UNIT ###
6 | StallF: 0
6 | StallD: 0
6 | FlushD: 0
6 | FlushE: 0
6 | ForwardAE: 01
6 | ForwardBE: 00
6 | Match_1E_M: 0
6 | Match_1E_W: 1
6 | Match_2E_M: 0
6 | Match_2E_W: 0
6 | Match_12D_E: 0
6 | LDRstall: 0
6 | BranchTakenE: 0
6 | PCWrPendingF: 0

### REGISTERS ###
6 | register0: 00000000000000000000000000000000
6 | register1: 00000000000000000000000000000011
6 | register2: 00000000000000000000000000000010
6 | register3: 00000000000000000000000000000000
6 | register4: 00000000000000000000000000000000
6 | register5: 00000000000000000000000000000000
6 | register6: 00000000000000000000000000000000
6 | register7: 00000000000000000000000000000000
6 | register8: 00000000000000000000000000000000
6 | register9: 00000000000000000000000000000000
6 | register10: 00000000000000000000000000000000
6 | register11: 00000000000000000000000000000000
6 | register12: 00000000000000000000000000000000
6 | register13: 00000000000000000000000000000000
6 | register14: 00000000000000000000000000000000
6 | register15: zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz

### REGFILE ###
6 | input 1: 1111
6 | input 2: 0010
6 | destination: 0001
6 | write data: 00000000000000000000000000000011
6 | write enable: 1
6 | reset: 0
6 | Reg_15: 00000000000000000000000000010100
6 | out_0: 00000000000000000000000000010100
6 | out_1: 00000000000000000000000000000010
6 | clk: 0

### MEMORY DATA ###
6 | mem0: 00000001
6 | mem1: 00000000
6 | mem2: 00000000
6 | mem3: 00000000
```

Figure 14: 6th clock cycle of the test.

Conclusion

The implemented processor design, its sub-modules and the test results are presented in this report. The branch predictor as explained in the project document is not implemented.

The Verilog HDL code is ready to be uploaded to the FPGA, and its input/output ports are arranged for easy demonstration.