Comp E 475

Digital Systems

HW 9: CPU

Student: Giorgi Solomnishvili

Red ID #: 822164480

12/14/2020

Contents

Task Description

For homework 5, we are asked to write a complete skeleton for ARM instruction decoder module. The module has two inputs: FLAGS and 32-bit driver, called instruction. The instruction represents an ARM assembly instruction. Before executing the instruction, the processor must identify what kind of instruction has been entered. There are 3 types of instructions: data-processing, memory and branch types. Moreover, each of them consists of several subgroups. For example:

- Data-processing type has four different kinds of instruction: immediate, register shifted by value, register shifted by register, and multiplication.
- Memory instruction type has two different kinds of instruction: immediate offset, and register shifted by register offset.
- Branch instruction type has two different kind of instruction: branch only, and branch and link.

For this homework, I was asked to determine the type of entered instruction.

We were instructed to have four outputs. See figure 1.

instr_type, 2 bits, values from 0 to 3
- 1 if given 32 bit instruction is Data Processing instruction
- 2 if it's memory type instruction
- 3 if it's branch instruction
- 0 if not identifiable

data_instr_type, 3 bits, values from 0 to 4
- 1 if given Data Processing instruction is "Immediate" type
- 2 if it's "Register shifted by value" type
- 3 if it's "Register shifted by register" type
- 4 if it's "Multiplication" type
- 0 if not identifiable

- **mem_instr_type**, 2 bits, values from 0 to 2
  - 1 if given Memory instruction type is of Immediate
  - 2 if it's "Register shifted by value" type
  - 0 if it's not identifiable
- **jmp_instr_type**, 2 bits , values from 0 to 2
  - 1 if given Jump instruction type is of Branch only
  - 2 if it's Branch and Link
  - 0 if it's not identifiable

Figure 1

we were asked to write Verilog code for program counter module. The module has three inputs: clk_in, mux_sel, and addr-to-jmp. The module has two outputs: PC_out and R15_out.

```
module program_counter(
    input mux_sel, // selects next instruction address
    input clk_in,
    input [7:0] addr_to_jmp_in, // specifies address wher to jump
    output reg [7:0] PC_out=0, // output of PC
    output [7:0] R15_out // This will be written in register file 15
    );
```

We were asked to write Verilog code for ALU that is able to implement certain data-processing, memory and branch instruction:

Data-processing: AND, XOR, ORR, SUB, RSB, ADD, CMP;

Memory: STR and LDR;

Branch: B.

The ALU has two 32 bit long ports – portA and portB. The ALU has two control signals OP – determining type of instruction, and CMD – determines the instruction. The ALU has one 32 bit long output – Result, and a 4-bit long output flags.

Solution

I_Decoder:

Determining Type of Instruction

The type of instruction is encoded in 27-26 bits of the entered 32-bit long instruction. These two bits are called op. Op for data-processing is 0, for memory type – 1, and for branch type – 2. I have an output, called inst_Type_of_Instruction, which equals to op + 1. Therefore:

- **Type_of_Instruction** is 1 if type is data-processing;
- **Type_of_Instruction** is 2 if type is memory;
- **Type_of_Instruction** is 3 if type is branch;
- **Type_of_Instruction** is 0 if type is unidentified;

Determining Type of Data-Processing Instruction

Moreover, we were asked to determine the type of data-processing instruction. I write an always block to accomplish that task. I have an IF-ELSE statement in the always block.

If op bit is 0, the module determines the type of data-processing instruction. mem_instr_type and jmp_inst_type are assigned with 0.

- **Identifying immediate type**: if 25th bit is 1, the instruction is immediate type.
- **Identifying multiplication type**: if 25:25 bits are 0, and 7:4 bits are 9, the instruction is multiplication.
- **Identifying register shifted by register type**: if 25th and 7th bits are 0s and 4th bit is 1, the instruction is register shifted by register.
- **Identifying register shifted by value type**: if 25th bits is 0 and 4th bit is 0, the instruction is register shifted by register

See figure 2

```
if(instruction[27:26]==0) begin // determine type of data-processing
   mem_instr_type = 0;
   jmp_instr_type = 0;
   if(instruction[25]) begin // i is 1
      Type_of_Data_Processing = 1;    // we have immediate
   end
   else if((instruction[24]==0)&&(instruction[7:4]==9)) begin
      Type_of_Data_Processing = 4;  // multiplication
   end
   else if((instruction[7]==0)&(instruction[4]==1)) begin
      Type_of_Data_Processing = 3;  // register shift by register
   end
   else if(instruction[4]==0) begin
      Type_of_Data_Processing = 2;  // register shifted by value
   end
   else Type_of_Data_Processing=0;
end
```

Figure 2

Determining Type of Memory Instruction

If op bit is 1, the module determines the type of memory instruction. Type_of_data_processing and jmp_inst_type are assigned with 0.

- **Identifying immediate offset type**: the 25th bit is complement of i. The value of i determines if the instruction is immediate offset or not. If the 25th bit is 0, i is 1 and we have immediate offset. Thus, mem_instr_type is 1.
- **Identifying register shifted by register offset type**: if the 25th and the 4th bits are 1, the offset is represented by register.

See Figure 3.

```
else if(instruction[27:26]==1) begin // determine type of memory instruction
   Type_of_Data_Processing = 0;
   jmp_instr_type = 0;
   if(instruction[25]==0) begin // ~i is 0
      mem_instr_type = 1; // offset in imediate
   end
   else if(instruction[4]==1) begin // ~i and inst[4] ==1
      mem_instr_type = 2; // offset in shifted register
   end
   else begin
      mem_instr_type = 0; // unidentified
   end
end
```

Figure 3

Determining Type of Branch Instruction

If op bit is 2, the module determines the type of branch instruction.
Type_of_data_processing and mem_inst_type are assigned with 0.

- **Identifying branch only instruction type**: the 24th bit determines if branch instruction is branch only or branch and link type. If the 24th bit is 0, we have branch only instruction and jmp_inst_type is assigned with 1.
- **Identifying branch and link instruction type**: if the 24th bit is 1, we have branch and link instruction and jmp_inst_type is assigned with 2.

See Figure 4

```
else if(instruction[27:25]==3'b101) begin // determine type of jump instruction
   Type_of_Data_Processing = 0;
   mem_instr_type = 0;
   if(instruction[24]==0) begin // L is 0
      jmp_instr_type = 1; // just branch
   end
   else if(instruction[24]==1) begin // L is 1
      jmp_instr_type = 2; // branch and link
   end
   else begin
      jmp_instr_type = 0;
   end
end
else begin // unidentified
   Type_of_Data_Processing = 0;
   jmp_instr_type = 0;
   mem_instr_type = 0;
end
```

Figure 4

What I have discussed so far is not going to be used by final version of CPU. The above-described code gives us potential to make the final version of CPU more advanced.

The following is going to be used by the final version of CPU.

The final version of I_Decoder unit is going to have 12 outputs. Out of those 10 outputs:

- 3 - flag_reg_write_en, register_file_write_en, memory_write_en are enable;
- 3 - mux_sel_branch_out, register_file_mux_sel, register_file_input_mux are select inputs for multiplexers;
- 2 – data_reg, base_addr_mem_instr indicate which registers are going to be used in memory and data-processing instrucitons;
- 2 - mem_instr_imm_out, branch_imm_out, are immediates;
- 2 – OP_out, CMD_out are control signals for ALU.

ALU Control Signals:

OP determines which type of instruction we have. If op is 0, we have data-processing, else if op is 1, we have memory instruction, else if op is 2 we have branch instruction.

OP is part of input called instruction. OP equals instruction[27:26]. Thus, instruction[27:26] is assigned to OP_out if condition_met is 1. Otherwise, 3 is assigned to OP_out. Condition_met is 1 by default. It becomes 0, if FLAGS do not correspond to cnd part of instruction. In our design, Condition_met is always going to be 1 because we are not going to check FLAGS. The fact that I have condition_met signal enables us to make our CPU more advanced.

CMD_out determines what command needs to be executed by the ALU. It is part of input called instruction.

CMD_out = instruction[24:21].

Enables:

Whenever flag_reg_write_en is asserted, the register for flag is updated by flags generated at ALU. This flag is asserted only if we have data processing type instruction and 20^{th} bit of instruction is 1.

assign flag_reg_write_en = (OP_out==0) & instruction[20]; // update flags if we have dataprocessing inst and S bit of instr is 1

register_file_write_en is asserted whenever we have data processing instruction of load from memory instruction. It enables writing into the register file.

memory_write_en is asserted whenever we have store to memory instruction. This signal enables us to update the data memory.

MUX select signals:

We are going to have 3 multiplexers. One of them multiplexes the input to PC, the second multiplexes the output of the data memory, and the third one multiplexes the input to register file.

mux_sel_branch_out controls which instruction is going to be fetched next. If it is 0, the consecutive instruction will be executed. Otherwise, if it is 1, we have branch instruction and PC + branch immediate instruction is going to be executed.

register_file_mux_sel controls what will be fed to register file. If it is 0, the output of data memory will be fed to register file. Otherwise, the output of ALU will be fed to register file.

register_file_input_mux controls what will be dest_reg instruction[15:12] or 15. If we have branch instruction, the value hold by the register 15 should be sent to ALU. That is why the select signal becomes 0.

Selecting Registers:

Destination register for data-processing instruction is part of signal called instruction. Dest_reg = instruction[15:12].

In case of memory instruction, dest_reg is source whose value should be stored in the memory, or destination which should receive a value taken from data memory.

In case of memory instruction, base_addr_mem_instr stores base address. In case of data processing instruction, base_addr_mem_instr stores one of the operands (sorry for confusing name).

base_addr_mem_instr = instruction[19:16]

Immediates:

We might have two kinds of immediate.

mem_instr_imm_out is offset (memory instruction) or the second operand (data-processing instruction). This signal has 12 bits

branch_imm_out has 24 bits. This is offset from the current PC + 2 to the next instruction.

Next, I generate control signals according to instruction data type.

Data-processing:

The following picture shows code for data processing instruction. Register write enable is asserted unless we have unidentified instruction or CMP instruction.

```
always @(*) begin
case(Type_of_Instruction) // generate signals for functions. Type_of_instruction = OP + 1
1: begin    // if Type_of_Instruction = 1, we have data-processing instruction
      mem_instr_imm_out = instruction[11:0]; // immediate out, The name is mem_instr but it is the same for data-proce
      dest_reg = instruction[15:12]; // dest register
      register_file_input_mux = 1; // dest reg is multiplexed, not 15
      register_file_mux_sel = 1; // ALU output is multiplexed, not output from data memory
      base_addr_mem_instr = instruction[19:16]; // source register

      mux_sel_branch_out = 0; // this 3 lines prevent latches
      branch_imm_out = 0;
      memory_write_en = 0;

      if((OP_out==3)|(CMD_out==4'b1010)) //  cmp or  unidentified command
         register_file_write_en = 0; // do not need to write anything into register files
      else
         register_file_write_en = 1; // enable writing in register files
```

Memory instruction:

The following picture shows code for memory instruction. Lines 199-208 checks if we have store or load memory. If we have LDR, register file write enable is asserted, the output of memory is fed to register file, and memory write enable is cleared.

If we have STD, register file write enable is cleared, the output of ALU is fed to register file, and memory write enable is asserted.

```
190   2: begin // if Type_of_Instruction = 2, we have memory instruction
191         base_addr_mem_instr = instruction[19:16]; // base address register number
192         mem_instr_imm_out = instruction[11:0]; // immediate out
193         dest_reg = instruction[15:12];
194
195         mux_sel_branch_out = 0; // next instruction is multiplexed at PC
196         branch_imm_out = 0; // prevents latch
197         register_file_input_mux = 1; // dest reg is multiplexed not 15
198
199         if((instruction[20]==1)&(conditions_met == 1)) begin     // L = 1 -> LDR
200            register_file_mux_sel = 0; // Memory output is multiplexed
201            register_file_write_en = 1; // enable writing into register file
202            memory_write_en = 0; // no need to write into memory
203         end
204         else begin // L = 0 -> STR
205            memory_write_en = 1; // memory write enable
206            register_file_mux_sel = 1; // ALU output is multiplexed
207            register_file_write_en = 0; // disable writing into register file
208         end
209      end
```

Branch instruction:

This code multiplexes PC+2+branch immediate in PC_multiplexer.

```
3: begin    // if Type_of_Instruction = 3, we have branching instruction
        memory_write_en = 0;
        mux_sel_branch_out = conditions_met; // send 1 to PC multiplexer
        branch_imm_out = instruction[23:0]; // branch immediate
        register_file_mux_sel = 1;
        register_file_input_mux = 0; // select 15th register.
        register_file_write_en = 0; // disable writing into register file
        base_addr_mem_instr = 0;
        mem_instr_imm_out = 0;
        dest_reg = 0;
```

Program Counter:

I start writing the code by creating a wire called PC_in. This is the wire connecting mux and PC. The multiplexer is controlled by the signal mux-sel. If mux-sel is 1, PC_in is assigned with addr_to_jmp_in. Otherwise, it is assigned with PC_out+1:

```
assign PC_in = (mux_sel==1) ? addr_to_jmp_in:PC_out+1; // mux that selects the next instruction address
```

At positive edge of the clock, PC_in goes to PC_out:

```
always @(posedge clk_in) begin // PC
   PC_out <= PC_in;
end
```

R15_out is assigned with PC_out+2:

ALU:

At first, I need to distinguish between type of instructions. I created an IF-ELSE statement which checks OP input. If OP is 0, instruction is data-processing, else if OP is 1, the instruction is memory, and if OP is 2 the instruction is branch.

For each of those cases, I created a case statement that checks CMD input. The following table is used to distinguish between different instructions based on CMD:

cmd	Name	Description	Operation
0000	AND Rd, Rn, Src2	Bitwise AND	Rd ← Rn & Src2
0001	EOR Rd, Rn, Src2	Bitwise XOR	Rd ← Rn ^ Src2
0010	SUB Rd, Rn, Src2	Subtract	Rd ← Rn - Src2
0011	RSB Rd, Rn, Src2	Reverse Subtract	Rd ← Src2 - Rn
0100	ADD Rd, Rn, Src2	Add	Rd ← Rn+Src2
0101	ADC Rd, Rn, Src2	Add with Carry	Rd ← Rn+Src2+C
0110	SBC Rd, Rn, Src2	Subtract with Carry	Rd ← Rn - Src2 - $\overline{C}$
0111	RSC Rd, Rn, Src2	Reverse Sub w/ Carry	Rd ← Src2 - Rn - $\overline{C}$
1000 ($S = 1$)	TST Rd, Rn, Src2	Test	Set flags based on Rn & Src2
1001 ($S = 1$)	TEQ Rd, Rn, Src2	Test Equivalence	Set flags based on Rn ^ Src2
1010 ($S = 1$)	CMP Rn, Src2	Compare	Set flags based on Rn - Src2
1011 ($S = 1$)	CMN Rn, Src2	Compare Negative	Set flags based on Rn+Src2
1100	ORR Rd, Rn, Src2	Bitwise OR	Rd ← Rn | Src2

Table 1

Figure 1 shows the case statement for data-processing instructions:

```
always @(*) begin
if(OP==0) begin // data-processing
   case(CMD)
      4'b0000: begin // Bitwise AND
                  Result = portA_in & portB_in;
               end
      4'b0001: begin // Bitwise XOR
                  Result = portA_in ^ portB_in;
               end
      4'b0010: begin //  SUB
                  Result = portA_in - portB_in;
               end
      4'b0011: begin //  RSB
                  Result = portB_in - portA_in;
               end
      4'b0100: begin //  ADD
                  Result = portA_in + portB_in;
               end
      4'b1100: begin // Bitwise ORR
                  Result = portA_in | portB_in;
               end
      4'b1010: begin //  CMP
                  Result = portA_in-portB_in;
               end
      default: begin
                  Result=0;
               end
   endcase
end
```

Figure 1

Figure 2 shows case statement for memory instructions. In memory instructions, if the bit 2 of CMD (U bit) is 1, I need to add offset to base address. Otherwise, I need to subtract offset from base address.

```
else if(OP==1) begin // Memory instruction
   case(CMD[2]) // check U
      0: Result = portA_in-portB_in; // base addr - offset
      1: Result = portA_in+portB_in; // base addr + offset
      default: Result = portA_in+portB_in;
   endcase
end
```

Figure 2

Figure 3 shows implementation of branch instruction. In branch instruction I just need to add portA and portB:

```
end
else if(OP==2) begin // Branching
   Result = portA_in+portB_in;
end
else begin//////
Result=0;
end////////////
```

Figure 3

In addition to this, I was asked to calculate flags after each operation. Flags is 4-bit long output.

- Flag[3] is N bit. It is 1, when result is negative or Result[31] is 1.
- Flag[2] is Z bit. It is 1, when result is zero or Result == 0.
- Flag[1] is C bit. It is 1, when carry is generated from the most significant bit or borrow is required by the most significant bit. If I have addition, C is 1, if the unsigned result is less than unsigned portA or unsigned portB. In case of subtraction, C is 1 if unsigned portA is less than unsigned portB. Other operations cannot set carry flag.
- Flag[0] is Overflow flag. It is 1, when overflow occurs. In case of addition, if the sign of two adders are the same, and the sing of the result is different, overflow flag is asserted. In case of subtraction, if minuend and subtrahend have different sings, and the sign of difference differs from the sign of minuend, overflow flag is asserted.

Only compare, add, sub and rvsb operations can assert carry and overflow flags.

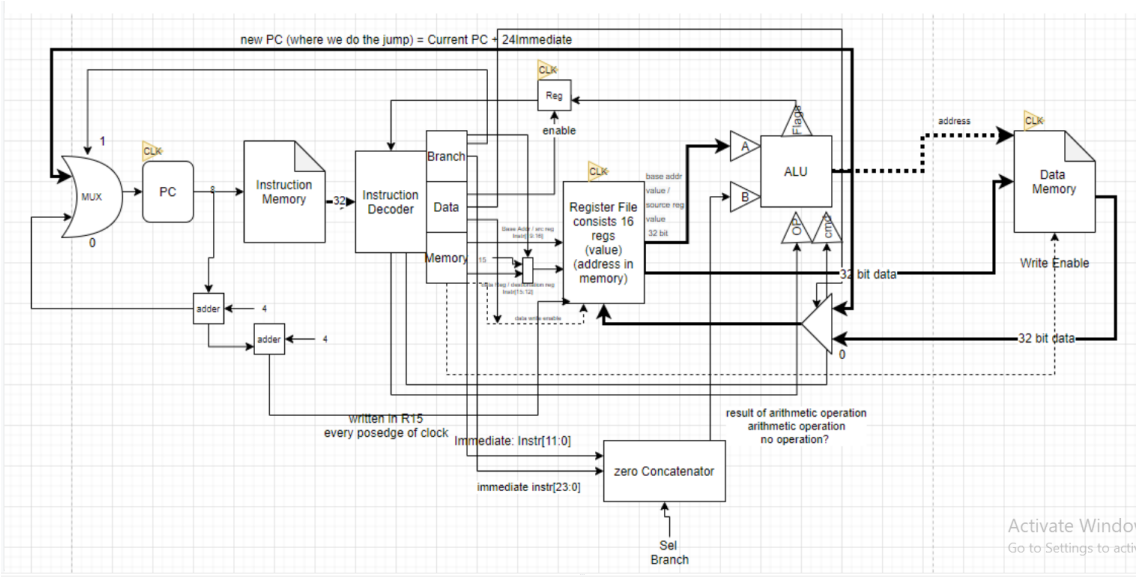
Figure 4 shows implementation of flags:

```
flags_out[3] =(Result[31]==1); // SET/CLEAR N flag
flags_out[2] = (Result==0); // SET/CLEAR z flag
if((CMD==4'b0010)|(CMD==4'b1010)) begin // sub cmp
   flags_out[1] = Result[31];//portA_in < portB_in; // SET/CLEAR c flag
   flags_out[0] = (portA_in[31]!=portB_in[31])&(portA_in[31]!=Result[31]); // SET/CLEAR o flag
end
else if(CMD==4'b0011) begin // rvsub
   flags_out[1] = portA_in > portB_in; // SET/CLEAR c flag
   flags_out[0] = (portA_in[31]!=portB_in[31])&(portB_in[31]!=Result[31]); // SET/CLEAR o flag
end
else if(CMD==4'b0100) begin // add
   flags_out[1] = (portA_in > Result)|(Result < portB_in); // SET/CLEAR c flag
   flags_out[0] = (portA_in[31]==portB_in[31])&(portA_in[31]!=Result[31]); // SET/CLEAR o flag
end
else begin
   flags_out[1] = 0;
   flags_out[0] = 0;
end
end
endmodule
```

Figure 4

CPU:

Next I instantiated all above mentioned modules (alongside with given register file, instr_mem and data_mem) in top module, called CPU, and connect them with each other according to the following block diagram:



Simulation & Verification

I_Decoder:

In order to check the correct functionality of my design, I wrote a testbench and simulated several inputs.

In order to check if Data-processing instruction is correctly identified, I set instruction[27:26] to 0. See figure 5.

As seen on Figure 5, whenever the 25th bit is 1, the instruction is immediate type and type_of_data_processing becomes 1. Whenever, the 27:24 amd 7:4 bits are 0, instruction is register shifted by value and type_of_data_processing is 2. Whenever 27:24 bits are 0s and 7:4 bits are 1001, we have multiplication and type_of_data_processing is 4. Finally, 27:24 and 7:5 bits are 0s and the 4th bit is 1, we have register shifted by register and type_of_data_processing is 3. Since we have data-processing instructions, mem_instr_type and jmp_instr_type are 0s.

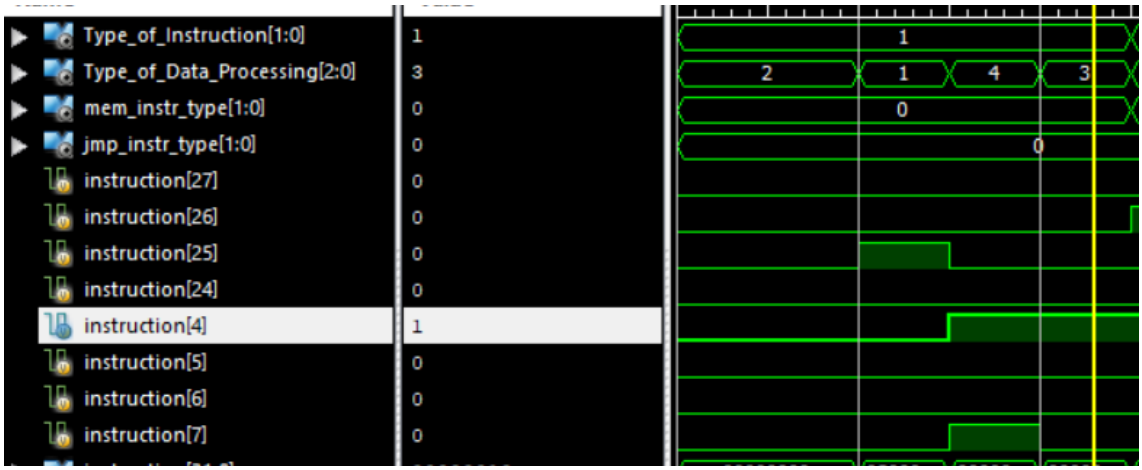


Figure 5

In order to check correct functionality of memory instruction,I set instruction 27:26 bits to 01. See Figure 6.

If the 25th bit is 0, we have immediate offset and mem_instr_type is 1. If the 25th bit is 1, the offset is in a register and mem_instr_type is 2. Since we have memory instruction, type_of_data_processing and jmp_instr_type are 0s.

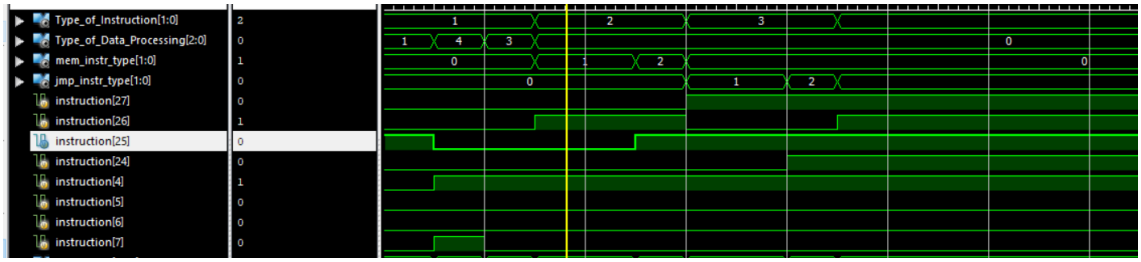


Figure 6

In order to check correct functionality of jump instruction, I set instruction[27:26] to 2. See figure 7. Whenever 24th bit is 1, we have branch and link, and jmp_instr_type is 2. If the 24th bit is 0, we have branch only instruction and jmp_instr_type is 1.

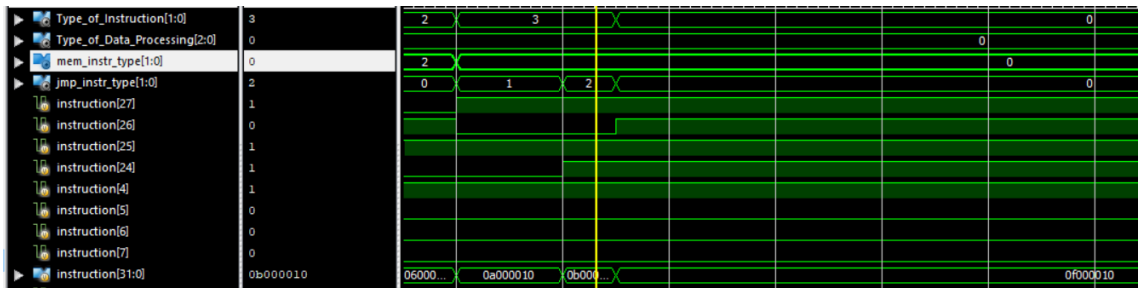


Figure 7

<u>Testing unidentified instruction:</u>

Figure 8 shows testing unidentified instruction. Everything is set to default mode and nothing will be executed by the ALU because OP_out becomes 3.

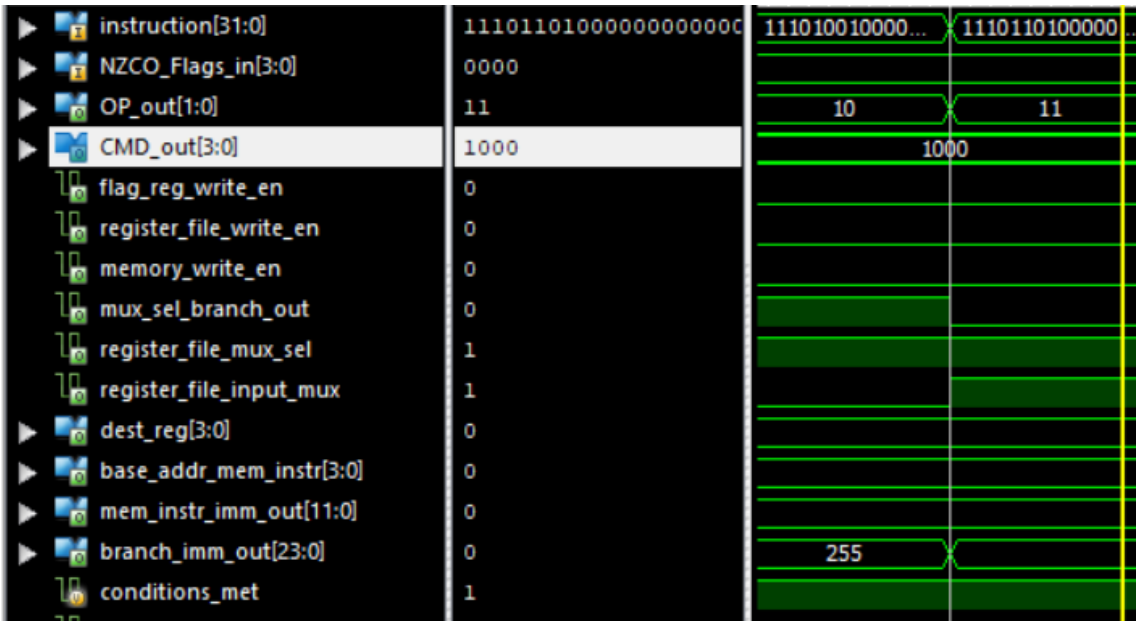


Figure 8

Testing Data-processing:

Figure 9 shows testing the data-processing part. OP_out is 0. If CMD is not for CMP(1010) command, register write enable is asserted. Memory write enable is 0, and other signals are correctly generated.

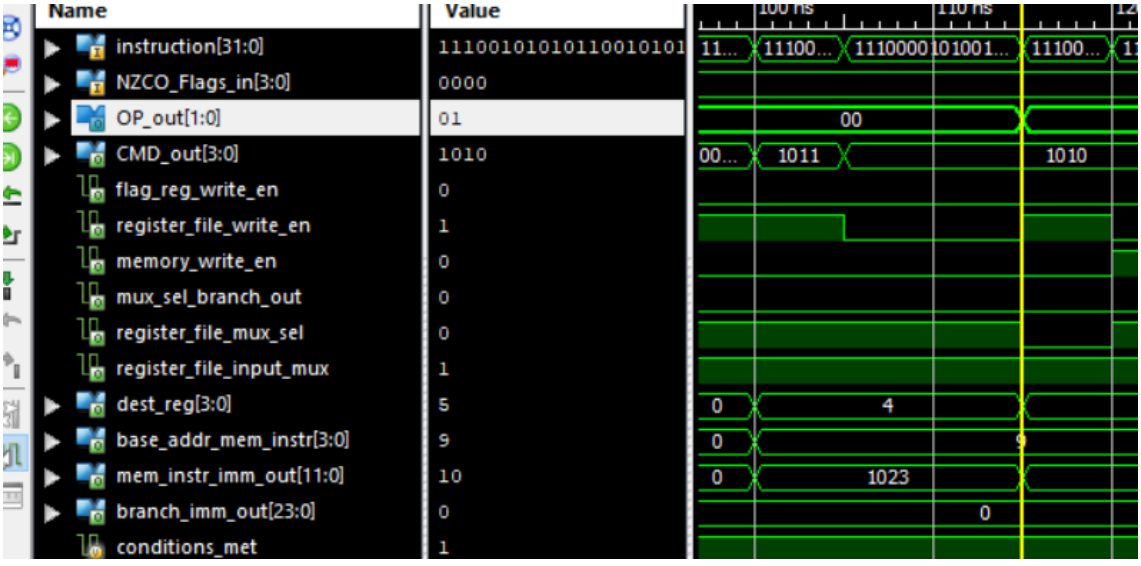


Figure 9

Testing Memory Instruction:

Figure 10 shows testing the memory instructions. At first, we have LDR instruction and register write enable is asserted and memory write enable is cleared. Register file multiplexer multiplexes the output from the data memory. Next, the instruction

becomes STD. Register write enable is cleared, memory write enable is asserted and ALU output is fed to register file.

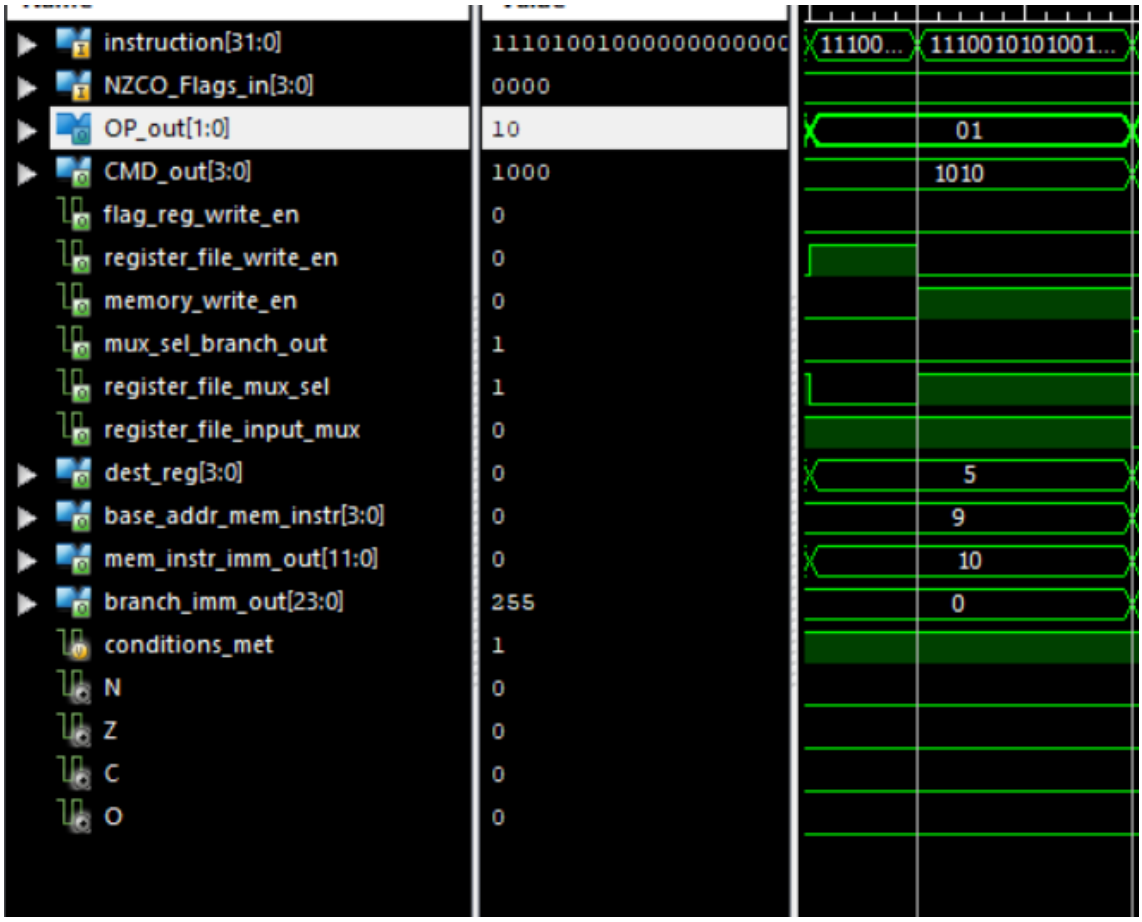


Figure 10

Testing Branch Instruction:

Figure 11 shows testing the branch instruction. The waveforms indicate that PC+2+branch immediate is multiplexed at the input of PC.

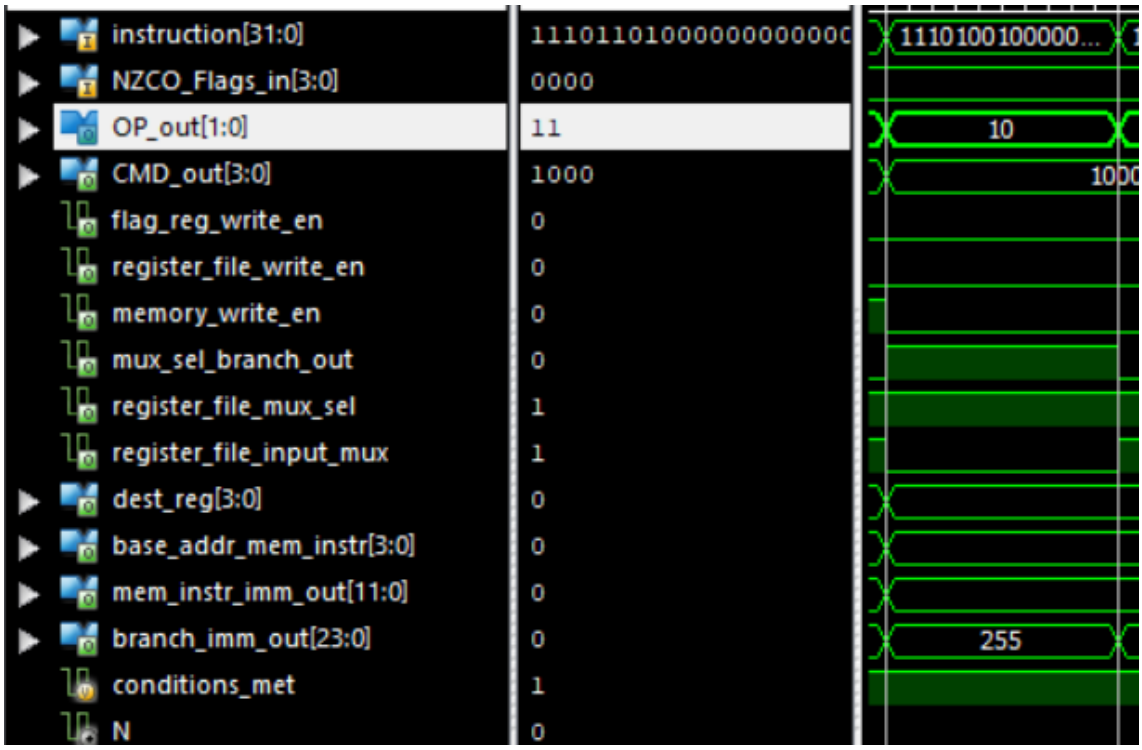


Figure 11

Testing flag_reg_write_en:

Figure 12 shows that flag register is updated if we have data processing instruction and instruction[20] is 1.

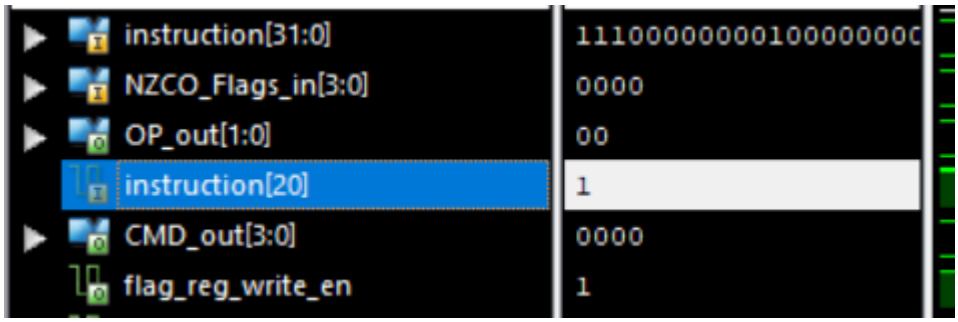


Figure 12

Program Counter:

In order to test my design, I wrote a testbench. Figure 2 shows that if mux-sel is 0, PC-out starts at 0 and increases by 1 at every positive edge of the clock. At the same time, R15_out is always more than PC_out by 2.

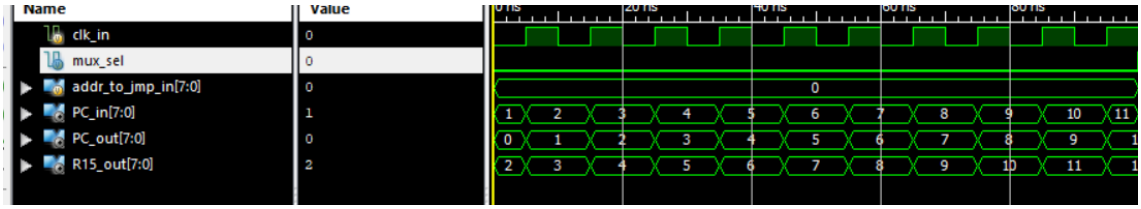


Figure 2

Figure 3 shows that when mux-sel becomes 1, PC_in becomes addr_to_jmp_in and is assigned to PC_out at the next positive edge of the clock.

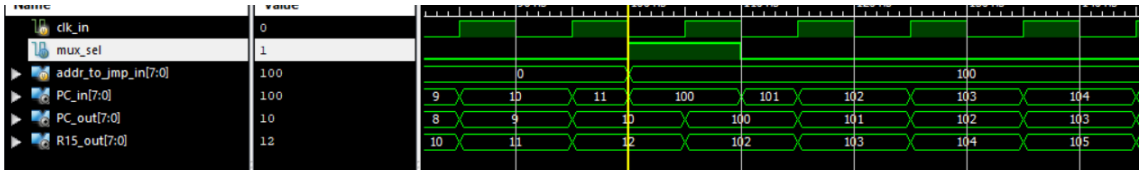


Figure 3

ALU:

In order to test my design, I wrote a testbench.

I tested each operation.

Testing Data-Processing

AND:

Figure 5 shows that AND operation is correctly executed, and it can set N and F flags:

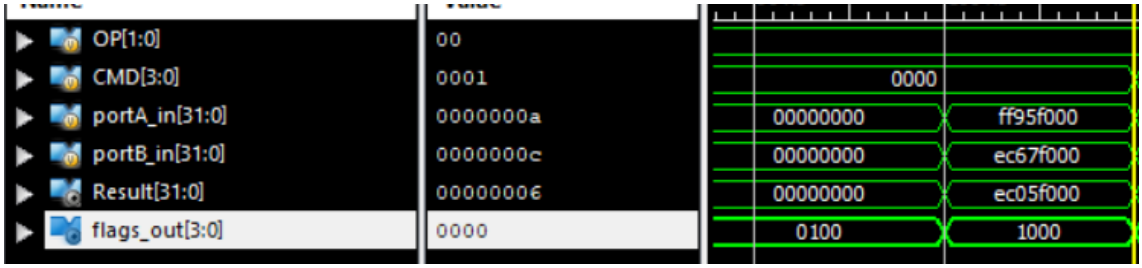


Figure 5

XOR:

Figure 6 shows that XOR operation is correctly executed, and it can set N and F flags:

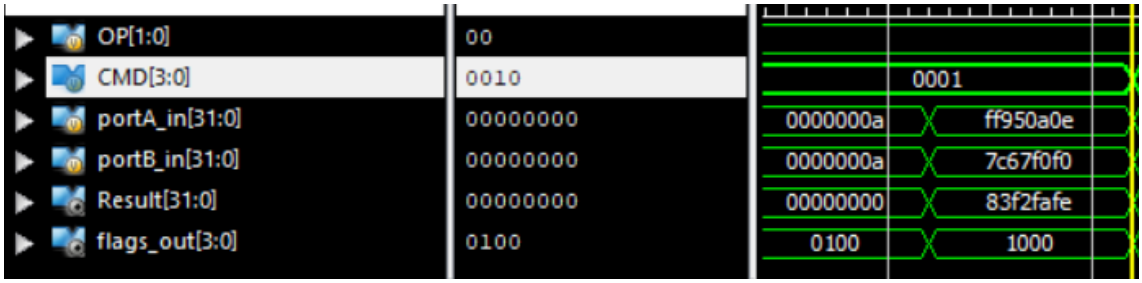


Figure 6

SUB:

Figure 7 shows that SUB operation is correctly executed, and it can set all flags correctly:

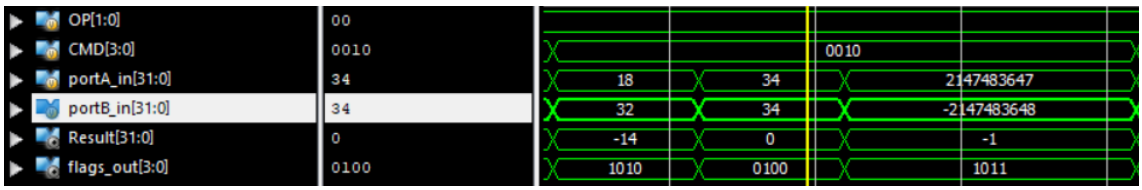


Figure 7

RVSUB:

Figure 8 shows that RVSUB operation is correctly executed, and it can set all flags correctly:

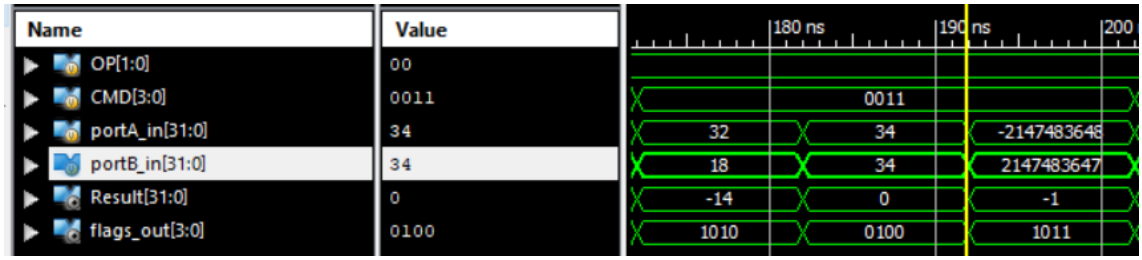


Figure 8

ADD:

Figure 9 shows that ADD operation is correctly executed, and it can set all flags correctly:

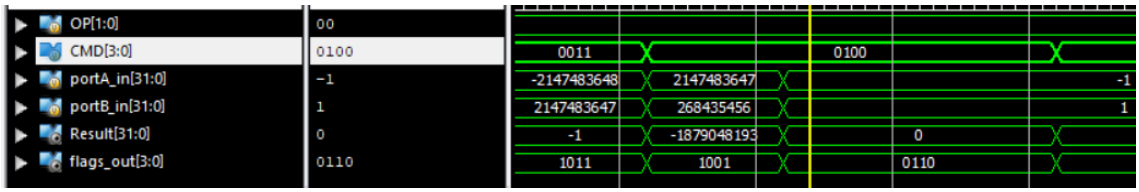


Figure 9

OR:

Figure 10 shows that OR operation is correctly executed, and it can set all flags correctly:

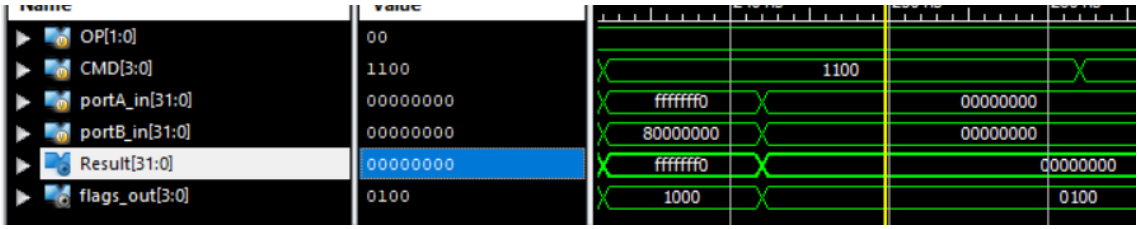


Figure 10

CMP:

Figure 11 shows that OR operation is correctly executed, and it can set all flags correctly:

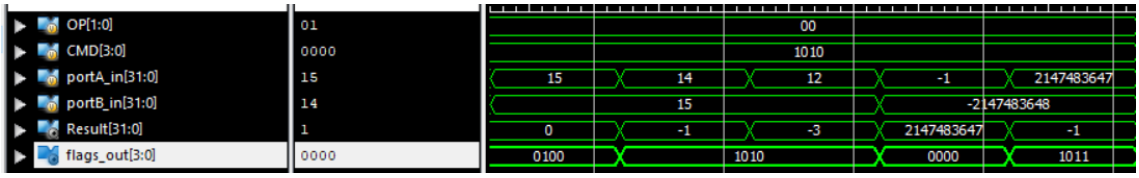


Figure 11

Testing Memory And Branch Instruction

Memory:

I set OP to 1. If CMD[2] is 0, address is portA – portB. Otherwise, I add them.

Branch:

I set OP to 2. Result is portA + portB.

See figure 12.

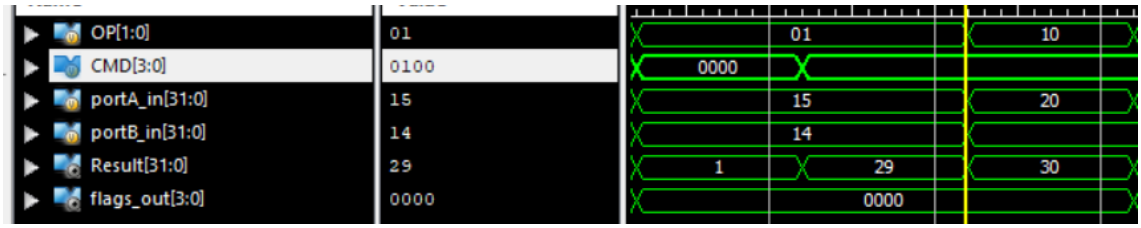
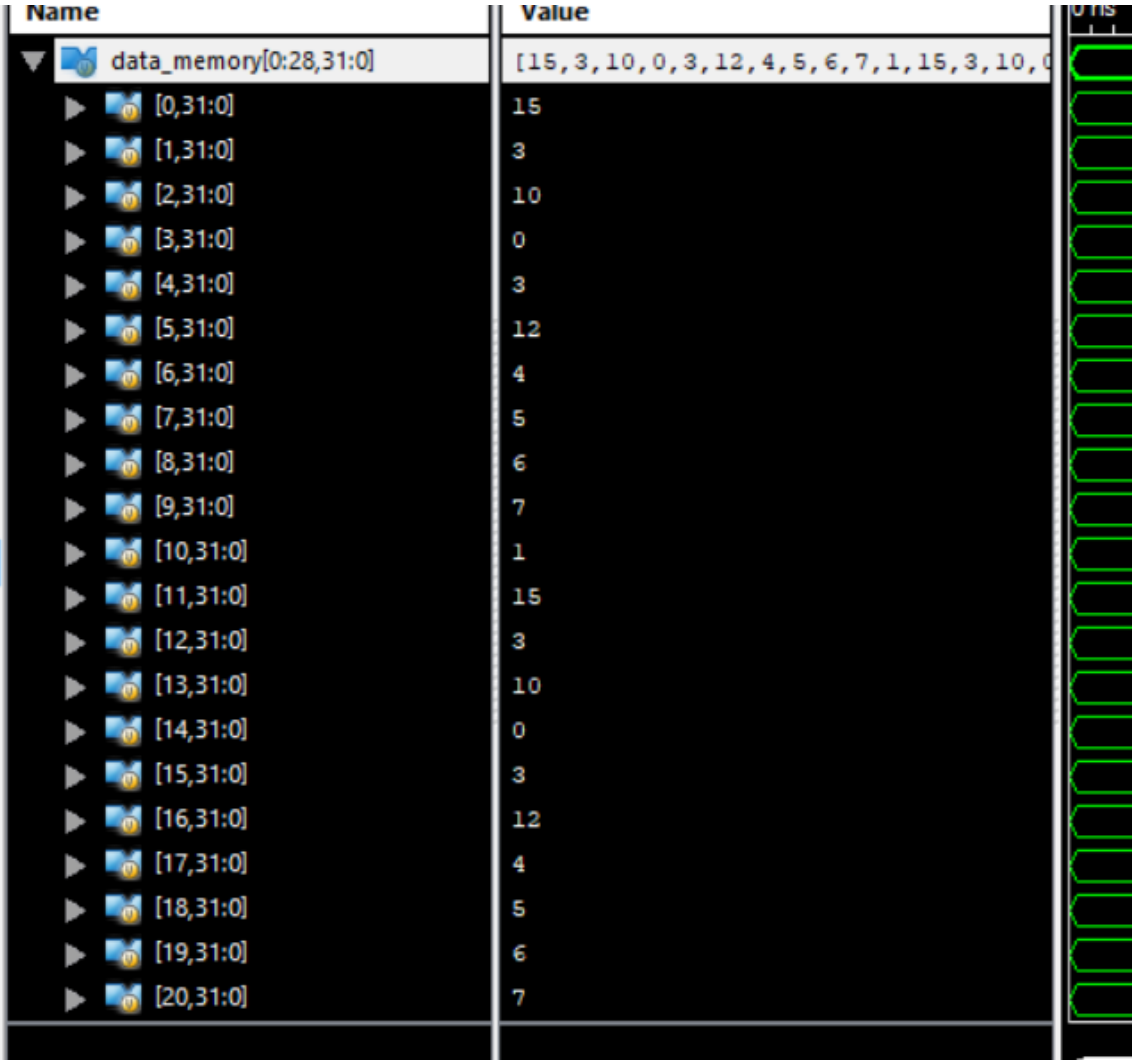
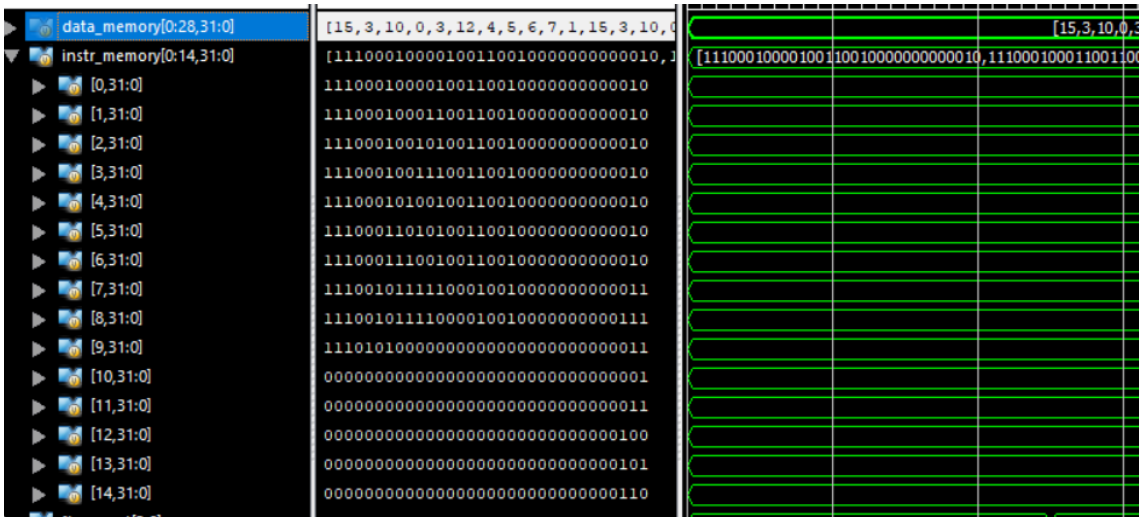


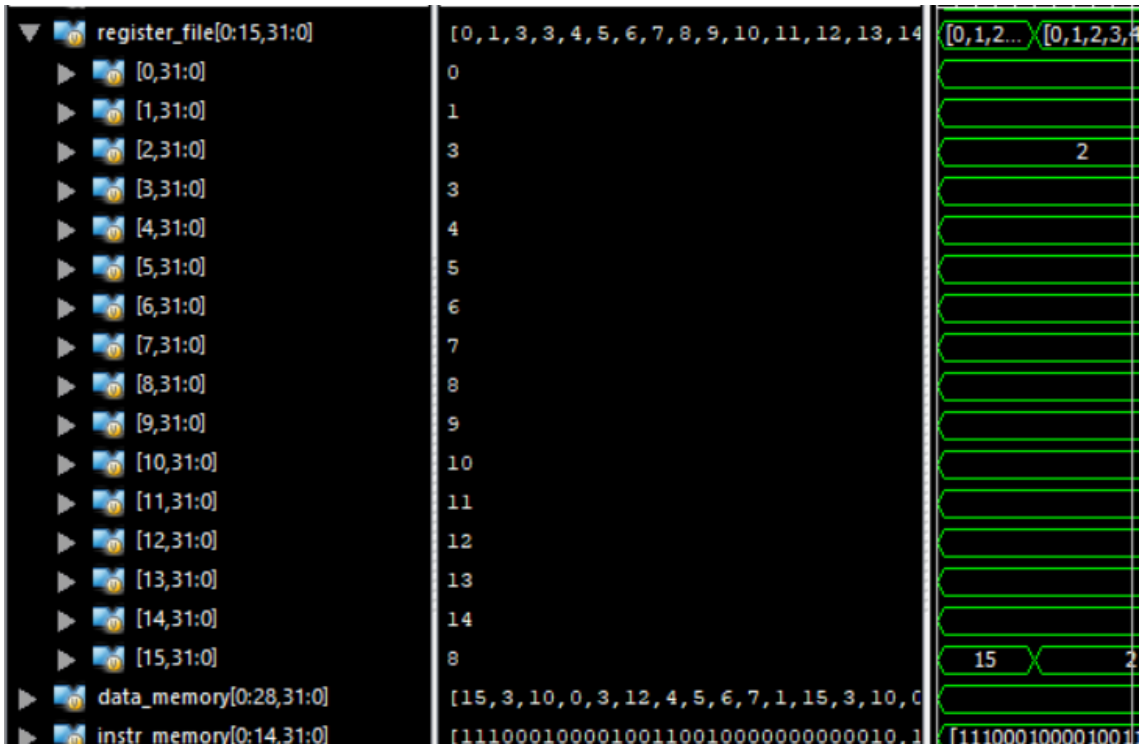
Figure 12

CPU:

In order to test CPU, I initialized register file, data and instruction memories with the following values:


Name
Value
data_memory[0:28,31:0]
[15,3,10,0,3,12,4,5,6,7,1,15,3,10,
[0,31:0]
15
[1,31:0]
3
[2,31:0]
10
[3,31:0]
0
[4,31:0]
3
[5,31:0]
12
[6,31:0]
4
[7,31:0]
5
[8,31:0]
6
[9,31:0]
7
[10,31:0]
1
[11,31:0]
15
[12,31:0]
3
[13,31:0]
10
[14,31:0]
0
[15,31:0]
3
[16,31:0]
12
[17,31:0]
4
[18,31:0]
5
[19,31:0]
6
[20,31:0]
7

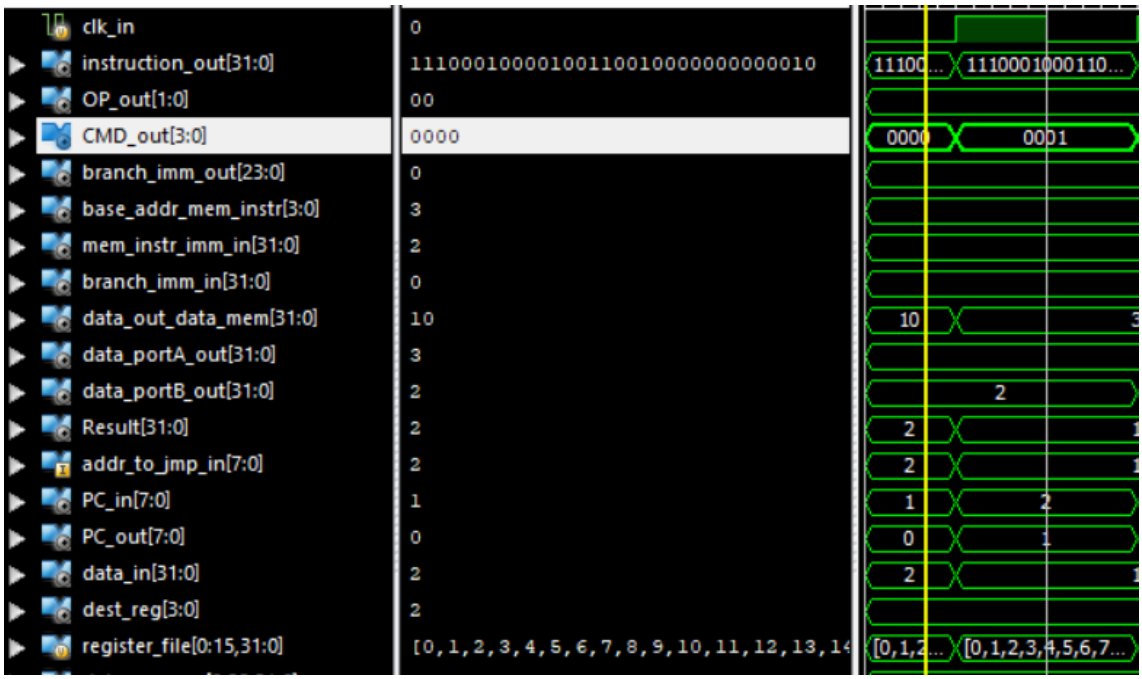

data_memory[0:28,31:0]
[15,3,10,0,3,12,4,5,6,7,1,15,3,10,
instr_memory[0:14,31:0]
[0,31:0]
[1,31:0]
[2,31:0]
[3,31:0]
[4,31:0]
[5,31:0]
[6,31:0]
[7,31:0]
[8,31:0]
[9,31:0]
[10,31:0]
[11,31:0]
[12,31:0]
[13,31:0]
[14,31:0]



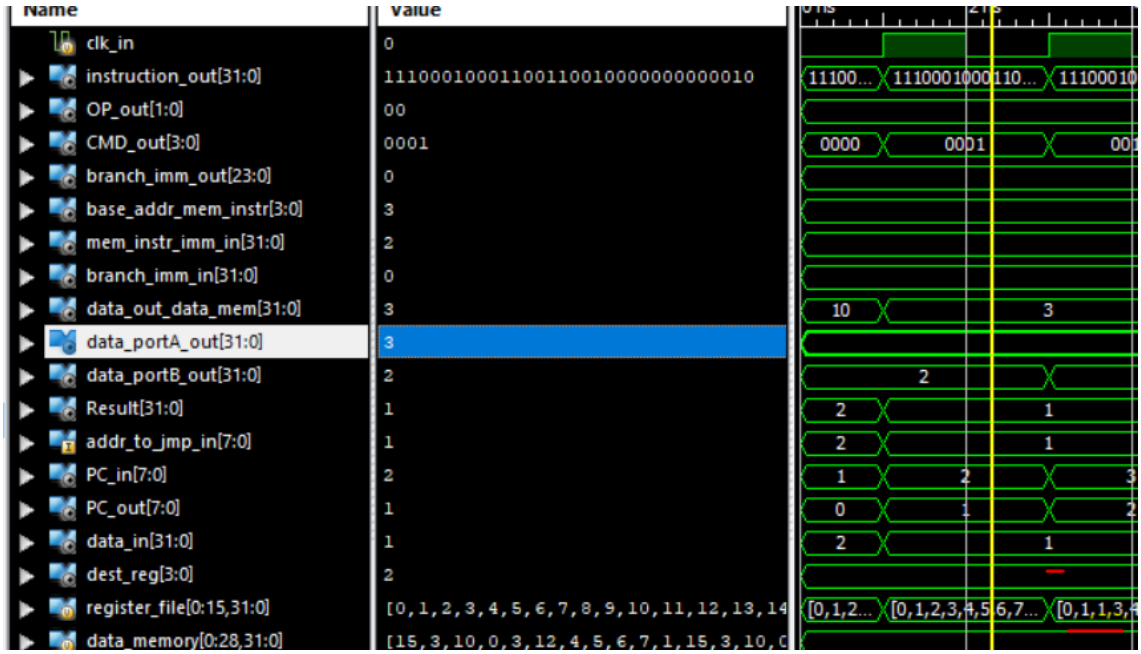
I generated the clock and test the following commands consecutively:

AND, XOR, SUB, RVB, ADD, CMP, OR, LDR,STR, B.

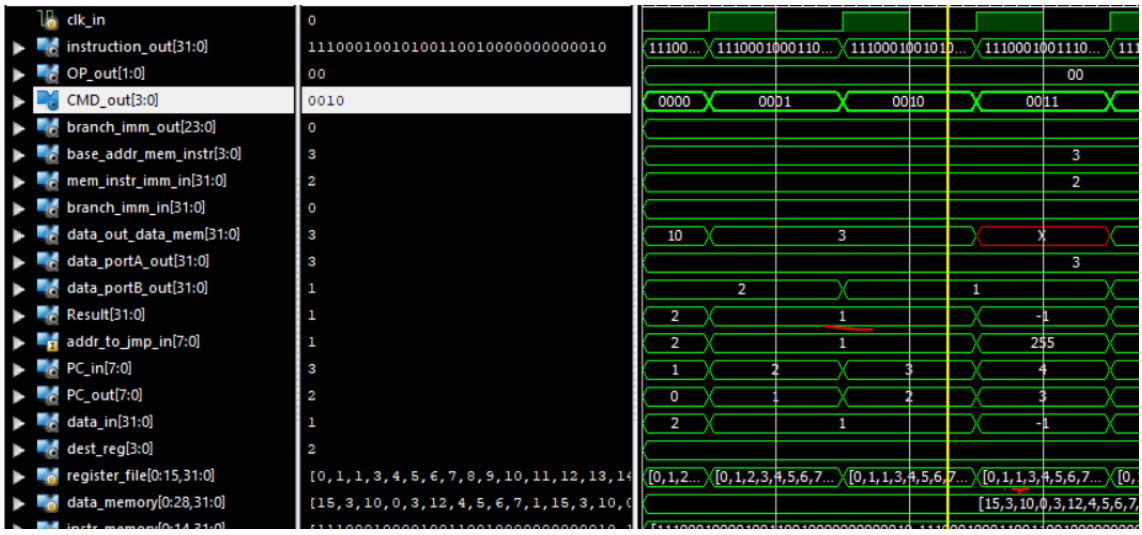
AND: mem_instr_imm_in is the second operand and is 2. The first operand is in register number 3 and its value is 3. 3&2=2. Thus Result is 2 and it is stored in Register[2].



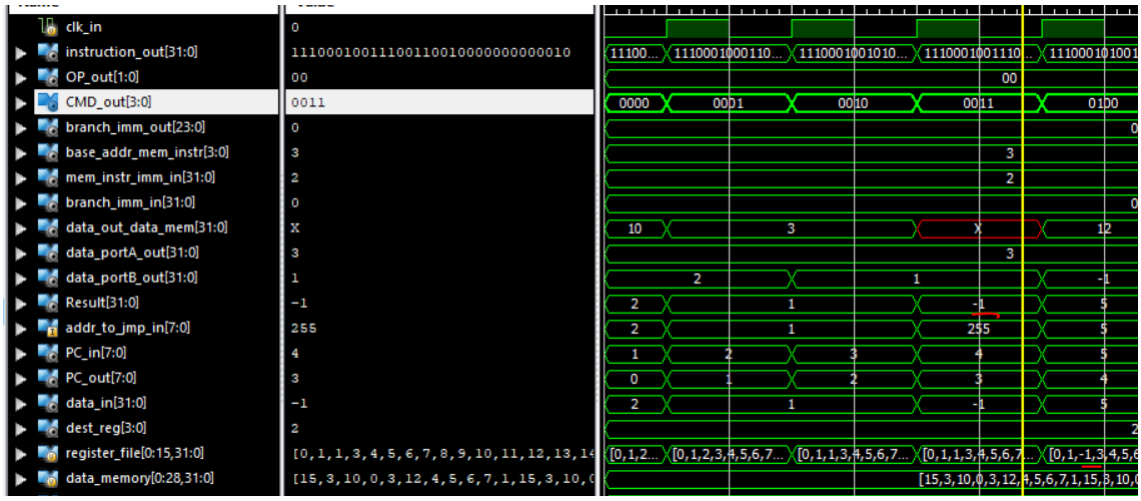
XOR: mem_instr_imm_in is the second operand and is 2. The first operand is in register number 3 and its value is 3. 3^2=1. Thus Result is 1 and it is stored in Register[2].



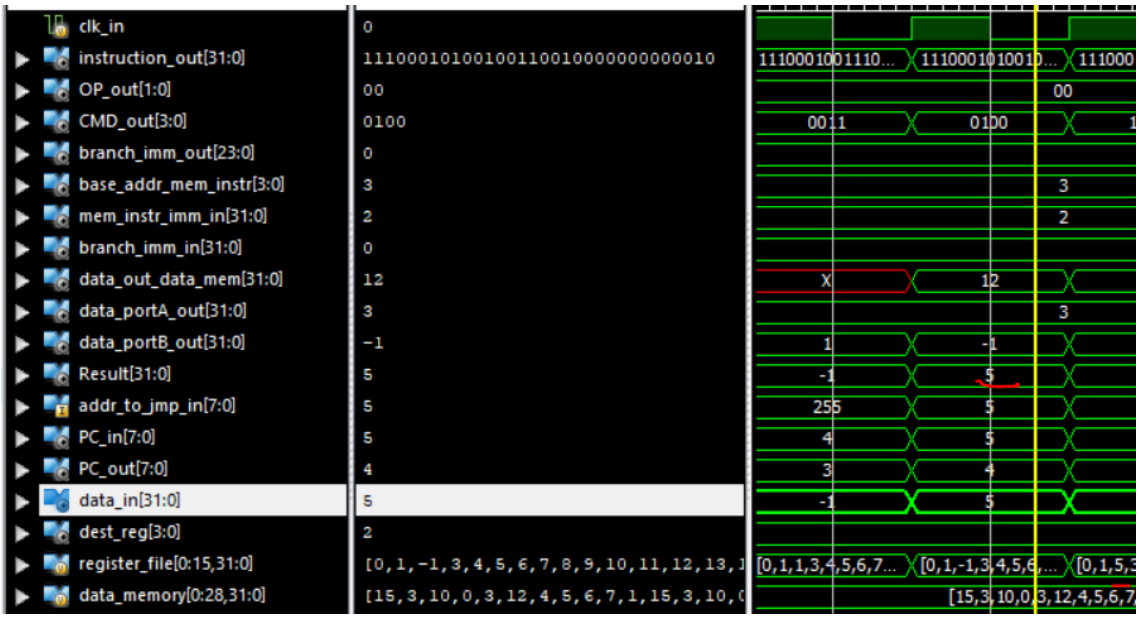
SUB: mem_instr_imm_in is the second operand and is 2. The first operand is in register number 3 and its value is 3. 3-2=1. Thus Result is 1 and it is stored in Register[2].



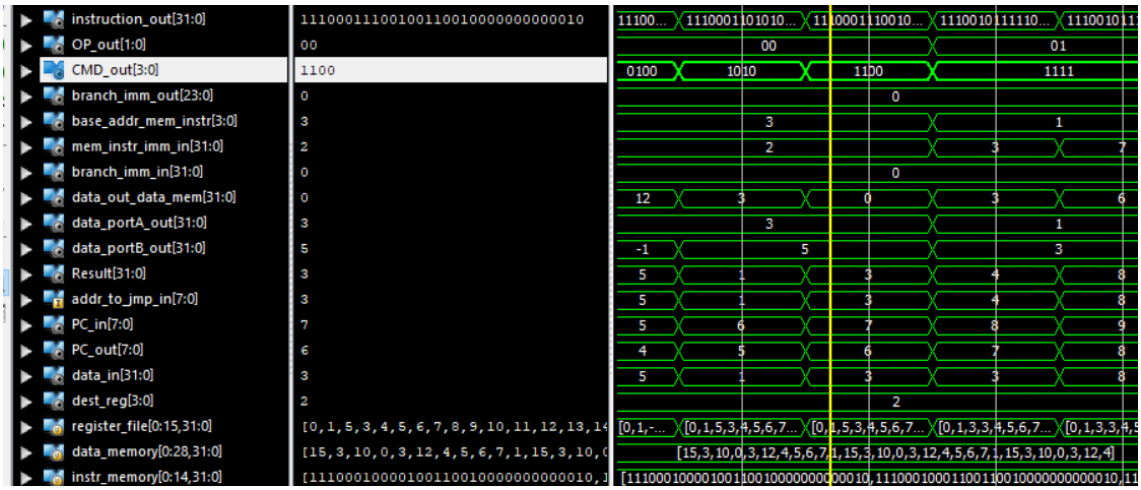
RVB: mem_instr_imm_in is the second operand and is 2. The first operand is in register number 3 and its value is 3. 2-3=-1. Thus Result is -1 and it is stored in Register[2]. Nevermind the red X on the simulation. It indicates that data memory is out of range. Since I am not trying to access data memory or store anything in it, it does not matter. No harm done.



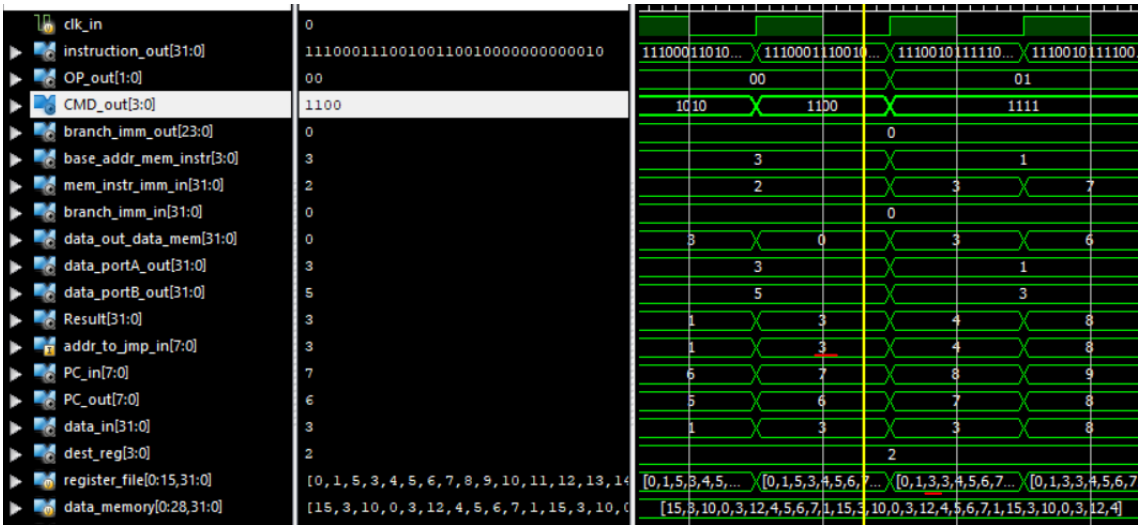
ADD: mem_instr_imm_in is the second operand and is 2. The first operand is in register number 3 and its value is 3. 3+2=5. Thus Result is 5 and it is stored in Register[2].



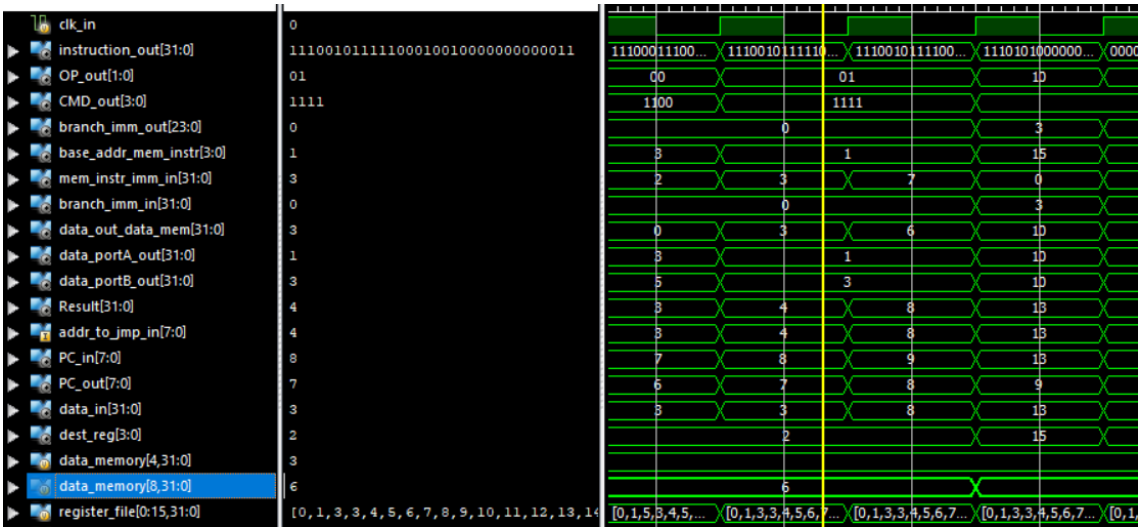
CMP: I have already show that compare works in testing the ALU. Now I show that CMP does not cause the update of data memory or registers: CMD 1010 is CMP. Only 15th register PC is updated



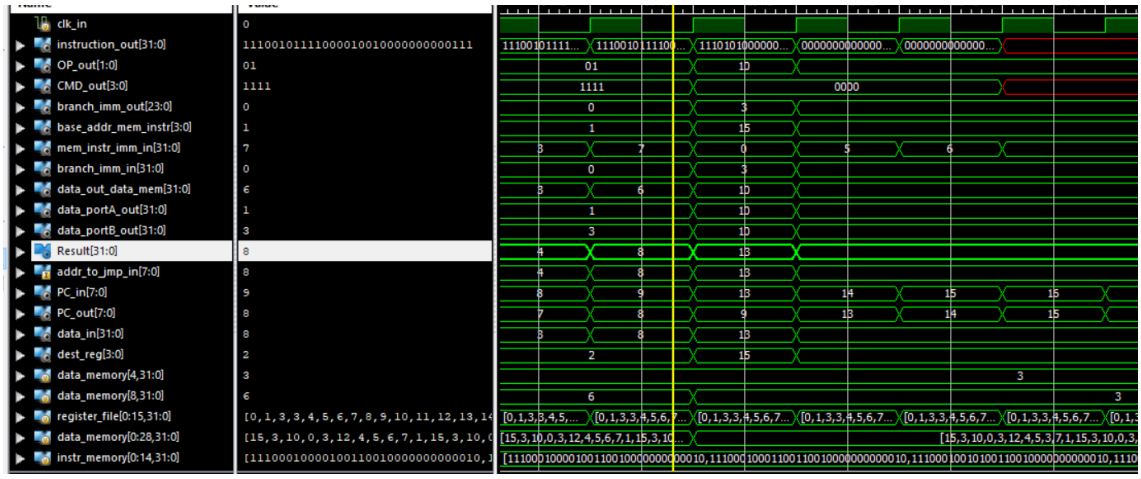
OR: : mem_instr_imm_in is the second operand and is 2. The first operand is in register number 3 and its value is 3. 3|2=3. Thus Result is 3 and it is stored in Register[2].



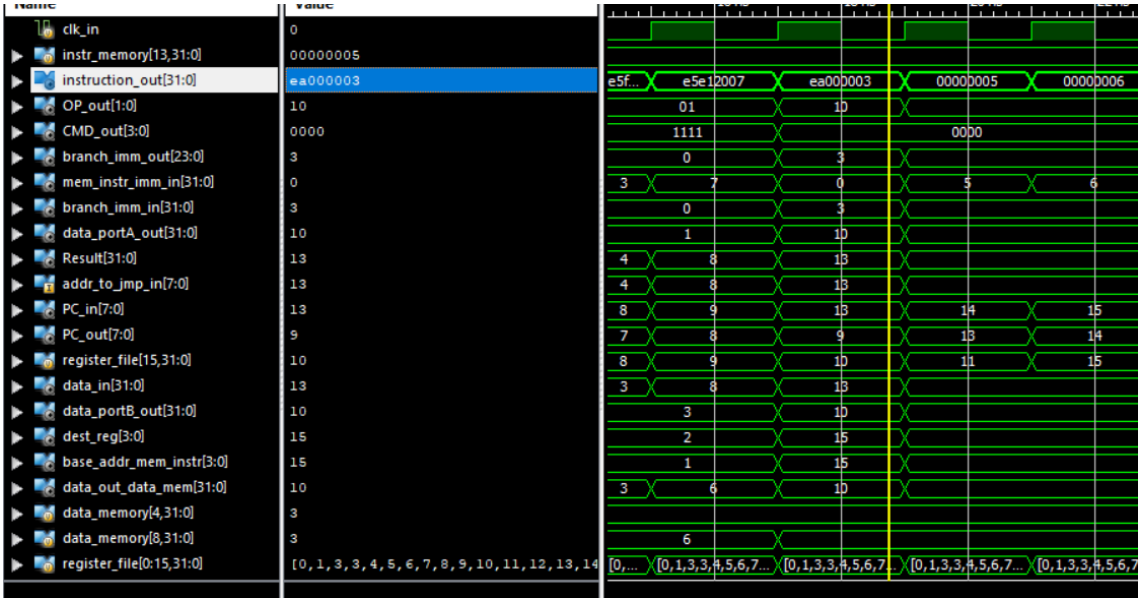
LDR: Base address is stored in register[1] and is 1. Offset is stored (mem_instr_imm) is 3. The address is 4 (Result). At data_mem[4], the value is 3. Destination register (dest_reg) is 2. Register[2] becomes 3.



STR: Base address is stored in register[1] and is 1. Offset is stored (mem_instr_imm) is 7. The address is 8 (Result). Source register is register[2] again which stores 3. At posedge, data_memory[8] becomes 3.



B: Immeidate is 3. The Value stored in Register[15] was 10. I wanted to jump at instruction 13. As you can see, addr_to_jmp became 13. An during the following posedge, the 13th instruction was fetched.



Conclusion

This is the final module microprocessor. While writing this module, I was supposed to use git and save my progress to repository. I learned a lot while writing CPU.

```
SOLO@DESKTOP-RCP9PH6 MINGW64 ~/Desktop/san diego/The Fourht Year/The Fall Semest
er/COMPE - 475/HW 4/Instruction_Decoding (master)
$ git log --oneline
6462d99 (HEAD -> master) Finished I_Decoder
14b2a9b HW 5 PDF
e295df2 modify .gitignore
5d98255 HW 5 testbench module
84653fd HW 5 .v module
5c80be1 Start writing pdf for HW 5
2bd9019 Start writing pdf for HW 5
2f34531 HW 5 - Detecting type-of instruction and the subgroup of each type
40638c3 ADD Detectection of type of memory instruction
238dcef Start edditing HW 4, developing instruction decoder
1164833 HW 4 .v module
8480c82 HW 4 PDF
7ff556e Add pdf
b152575 Make the design combinational circuit
28bb365 ignore docx files
957286b Modified to satisfy requirements
56f4e07 Synthesis report
00bde56 simulation - correctly detect type of instruction and type of data proce
ssing
28fdd1b Synthesis report for fixed bug design
a9b6434 Fix the bug for detecting multiplication data processing
82de4a2 Modify git ignore
82f10bf Synthesis report for sequential design
:...skipping...
6462d99 (HEAD -> master) Finished I_Decoder
14b2a9b HW 5 PDF
e295df2 modify .gitignore
5d98255 HW 5 testbench module
84653fd HW 5 .v module
5c80be1 Start writing pdf for HW 5
2bd9019 Start writing pdf for HW 5
2f34531 HW 5 - Detecting type-of instruction and the subgroup of each type
40638c3 ADD Detectection of type of memory instruction
238dcef Start edditing HW 4, developing instruction decoder
1164833 HW 4 .v module
8480c82 HW 4 PDF
7ff556e Add pdf
b152575 Make the design combinational circuit
28bb365 ignore docx files
957286b Modified to satisfy requirements
56f4e07 Synthesis report
00bde56 simulation - correctly detect type of instruction and type of data processing
28fdd1b Synthesis report for fixed bug design
a9b6434 Fix the bug for detecting multiplication data processing
82de4a2 Modify git ignore
82f10bf Synthesis report for sequential design
9b1c04e Determine type of instruction
514ec6b Initialize Git
~
```

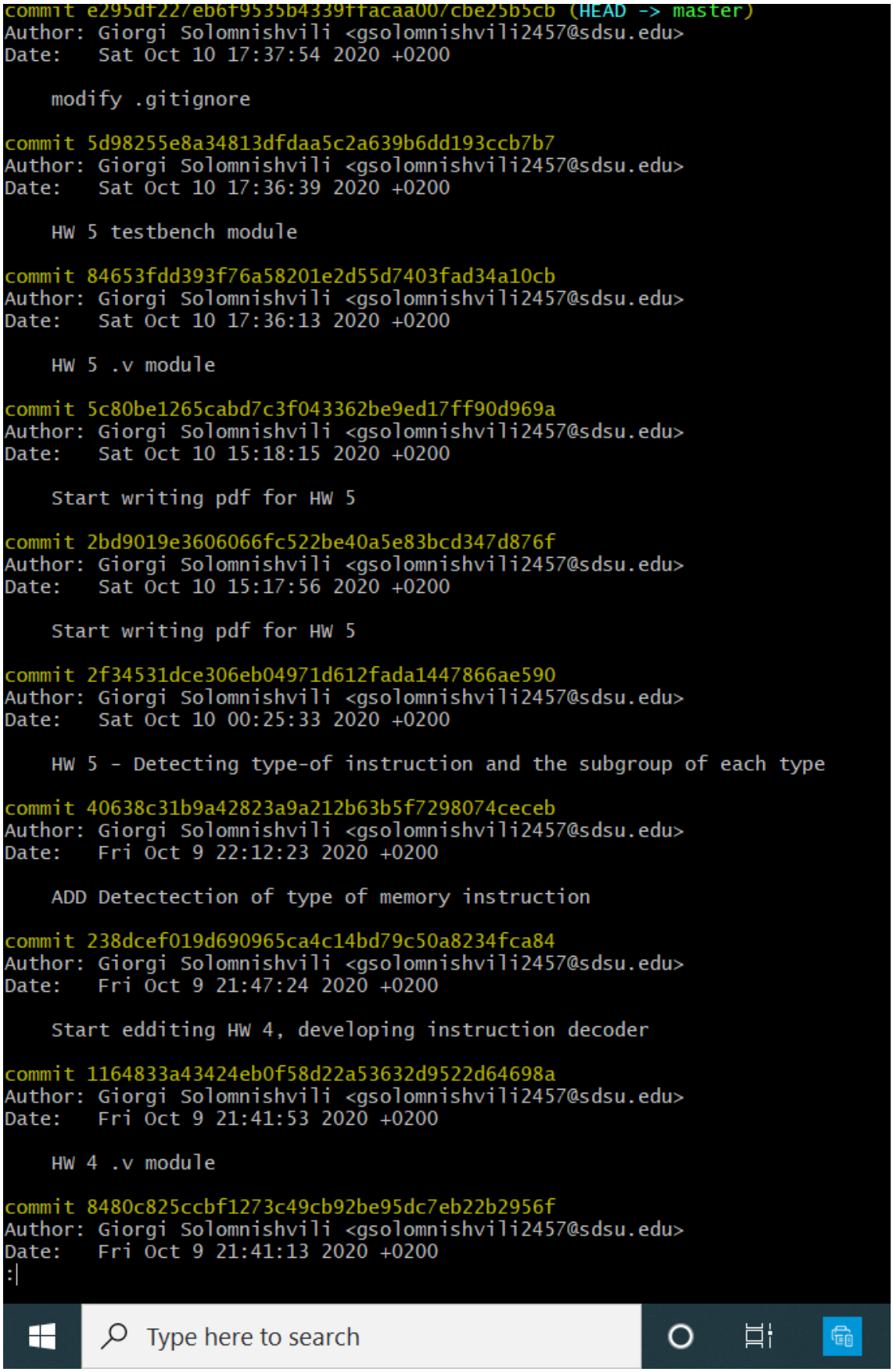

commit e295df227eb6f9535b4339ffacaa007cbe25b5cb (HEAD -> master)
Author: Giorgi Solomnishvili <gsolomnishvili2457@sdsu.edu>
Date: Sat Oct 10 17:37:54 2020 +0200
modify .gitignore
commit 5d98255e8a34813dfdaa5c2a639b6dd193ccb7b7
Author: Giorgi Solomnishvili <gsolomnishvili2457@sdsu.edu>
Date: Sat Oct 10 17:36:39 2020 +0200
HW 5 testbench module
commit 84653fdd393f76a58201e2d55d7403fad34a10cb
Author: Giorgi Solomnishvili <gsolomnishvili2457@sdsu.edu>
Date: Sat Oct 10 17:36:13 2020 +0200
HW 5 .v module
commit 5c80be1265cabd7c3f043362be9ed17ff90d969a
Author: Giorgi Solomnishvili <gsolomnishvili2457@sdsu.edu>
Date: Sat Oct 10 15:18:15 2020 +0200
Start writing pdf for HW 5
commit 2bd9019e3606066fc522be40a5e83bcd347d876f
Author: Giorgi Solomnishvili <gsolomnishvili2457@sdsu.edu>
Date: Sat Oct 10 15:17:56 2020 +0200
Start writing pdf for HW 5
commit 2f34531dce306eb04971d612fada1447866ae590
Author: Giorgi Solomnishvili <gsolomnishvili2457@sdsu.edu>
Date: Sat Oct 10 00:25:33 2020 +0200
HW 5 - Detecting type-of instruction and the subgroup of each type
commit 40638c31b9a42823a9a212b63b5f7298074ceceb
Author: Giorgi Solomnishvili <gsolomnishvili2457@sdsu.edu>
Date: Fri Oct 9 22:12:23 2020 +0200
ADD Detectection of type of memory instruction
commit 238dcef019d690965ca4c14bd79c50a8234fca84
Author: Giorgi Solomnishvili <gsolomnishvili2457@sdsu.edu>
Date: Fri Oct 9 21:47:24 2020 +0200
Start edditing HW 4, developing instruction decoder
commit 1164833a43424eb0f58d22a53632d9522d64698a
Author: Giorgi Solomnishvili <gsolomnishvili2457@sdsu.edu>
Date: Fri Oct 9 21:41:53 2020 +0200
HW 4 .v module
commit 8480c825ccbf1273c49cb92be95dc7eb22b2956f
Author: Giorgi Solomnishvili <gsolomnishvili2457@sdsu.edu>
Date: Fri Oct 9 21:41:13 2020 +0200
:
Type here to search

Figure 8