



Course Title:	Computer Organization and Architecture
Course Number:	COE608
Semester/Year (e.g. F2017)	W2025

Instructor	Dr. Demetres Kostas & Dr. Vadim Geurkov
TA Name	Yasaman Ahmadiadli

Assignment/Lab Number:	Lab 6
Assignment/Lab Title:	The Complete CPU (Overall Project)

Submission Date:	Apr 5th, 2025
Due Date:	23:59, Apr 10th, 2025

Student LAST Name	Student FIRST Name	Student Number	Section	Signature*
Balar	Om	501165749	05	
Muruganantham	Haran	501166674	10	

*By signing above you attest that you have contributed to this written lab report and confirm that all work you have contributed to this lab report is your own work. Any suspicion of copying or plagiarism in this work will result in an investigation of Academic Misconduct and may result in a "0" on the work, an "F" in the course, or possibly more severe penalties, as well as a Disciplinary Notice on your academic record under the Student Code of Academic Conduct, which can be found online at: <http://www.rverson.ca/senate/current/pol60.pdf>

Table of Contents

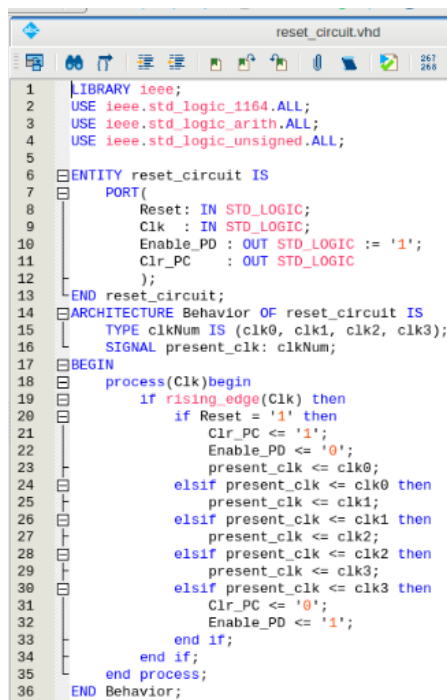
Table of Contents	1
Objective	2
VHDL Code Overview	2
Waveforms and Simulations	8
FPGA Emulation (Bonus)	16
Conclusion	18
References	19

Objective

This lab focuses on integrating all the components developed in previous labs to form a complete CPU system. The main objective is to combine the datapath and control unit, along with a reset circuit and system memory, to create a functional processor capable of executing instructions as specified in the CPU specification document. This lab consists of two parts: implementing the reset circuit and assembling the final CPU system. The reset circuit ensures proper initialization of the CPU, while the final CPU system integrates all subcomponents, including the instruction memory unit, to execute operations correctly. The design will be verified through simulation to confirm that the CPU correctly processes instructions and performs arithmetic, logic, and branching operations.

VHDL Code Overview

First, the reset circuitry was implemented because it is required to create the full CPU system. To implement the reset circuit, a series of if-else statements were used. First the code checks if the Reset signal is high. If so, it will set the clear PC signal to high and set Enable_PD to low. Enable_PD is sent to the enable input of the Control Unit designed in Lab 5. When it is low, the Control Unit will not produce any outputs. If the Reset signal is low then the circuit will wait for 4 clock cycles and then it will set Clear_PC to low and Enable_PD to high. It waits 4 clock cycles to do this because the values in the CPU need to stabilize. Figure 1 below shows the VHDL implementation of this.



```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3  USE ieee.std_logic_arith.ALL;
4  USE ieee.std_logic_unsigned.ALL;
5
6  ENTITY reset_circuit IS
7  PORT(
8      Reset: IN STD_LOGIC;
9      Clk : IN STD_LOGIC;
10     Enable_PD : OUT STD_LOGIC := '1';
11     Clr_PC : OUT STD_LOGIC
12 );
13 END reset_circuit;
14 ARCHITECTURE Behavior OF reset_circuit IS
15     TYPE clkNum IS (clk0, clk1, clk2, clk3);
16     SIGNAL present_clk: clkNum;
17 BEGIN
18     process(Clk)begin
19         if rising_edge(Clk) then
20             if Reset = '1' then
21                 Clr_PC <= '1';
22                 Enable_PD <= '0';
23                 present_clk <= clk0;
24             elsif present_clk <= clk0 then
25                 present_clk <= clk1;
26             elsif present_clk <= clk1 then
27                 present_clk <= clk2;
28             elsif present_clk <= clk2 then
29                 present_clk <= clk3;
30             elsif present_clk <= clk3 then
31                 Clr_PC <= '0';
32                 Enable_PD <= '1';
33             end if;
34         end if;
35     end process;
36 END Behavior;
```

Figure 1. VHDL implementation of the reset circuitry.

To create the CPU system, first a program was written to put together the data path from Lab 4b, the control unit from Lab 5, and the reset circuit from Part I of this lab. It will do this by taking the clock, memory clock, reset signal, and dataIn (instruction) as the input. Then, it will send the dataIn to the instruction register and all of the matching signals are connected between the components (ex. OutIR from the Data_Path is connected to the INST input for the Control Unit). The implementation for this is shown below in Figure 2.

```

1  LIBRARY ieee;
2  USE ieee.std_logic_1164.all;
3  USE ieee.std_logic_arith.all;
4  USE ieee.std_logic_unsigned.all;
5
6  ENTITY cpu1 IS
7  PORT (
8    clk : IN STD_LOGIC;
9    mem_clk : IN STD_LOGIC;
10   rst : IN STD_LOGIC;
11   dataIn : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
12   dataOut : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
13   addrOut : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
14   doutA : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
15   doutB : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
16   doutC : OUT STD_LOGIC;
17   doutZ : OUT STD_LOGIC;
18   doutIR : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
19   doutPC : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
20   wEn : OUT STD_LOGIC;
21   outT : OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
22   wen_mem : OUT STD_LOGIC;
23   en_mem : OUT STD_LOGIC
24 );
25 END cpu1;
26
27 ARCHITECTURE description OF cpu1 IS
28 COMPONENT Data_Path IS
29 PORT (
30   Clk, mClk : IN STD_LOGIC; -- clock Signal
31   --Memory Signals
32   WEN, EN : IN STD_LOGIC;
33   -- Register Control Signals (CLR and LD).
34   Clr_A, Ld_A : IN STD_LOGIC;
35   Clr_B, Ld_B : IN STD_LOGIC;
36   Clr_C, Ld_C : IN STD_LOGIC;
37   Clr_Z, Ld_Z : IN STD_LOGIC;
38   ClrPC, Ld_PC : IN STD_LOGIC;
39   ClrIR, Ld_IR : IN STD_LOGIC;
40   -- Register outputs (Some needed to feed back to control unit. Others pulled out for testing.
41   Out_A : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
42   Out_B : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
43   Out_C : OUT STD_LOGIC;
44   Out_Z : OUT STD_LOGIC;
45   Out_PC : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
46   Out_IR : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
47   -- Special inputs to PC.
48   Inc_PC : IN STD_LOGIC;
49   -- Address and Data Bus signals for debugging.
50   ADDR_OUT : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
51   DATA_IN : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
52   DATA_BUS, MEM_OUT, MEM_IN : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);

```

```

53 MEM_ADDR : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
54 -- Various MUX controls.
55 DATA_Mux : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
56 REG_Mux : IN STD_LOGIC;
57 A_MUX, B_MUX : IN STD_LOGIC;
58 IM_MUX1 : IN STD_LOGIC;
59 IM_MUX2 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
60 -- ALU Operations.
61 ALU_Op : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
62 );
63 END COMPONENT;
64
65 COMPONENT Control_NEWIS
66 PORT (
67   clk, mclk : IN STD_LOGIC;
68   enable : IN STD_LOGIC;
69   statusC, statusZ : IN STD_LOGIC;
70   INST : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
71   A_Mux, B_Mux : OUT STD_LOGIC;
72   IM_MUX1, REG_Mux : OUT STD_LOGIC;
73   IM_MUX2, DATA_Mux : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
74   ALU_Op : OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
75   inc_PC, ld_PC : OUT STD_LOGIC;
76   clr_IR : OUT STD_LOGIC;
77   ld_IR : OUT STD_LOGIC;
78   clr_A, clr_B, clr_C, clr_Z : OUT STD_LOGIC;
79   ld_A, ld_B, ld_C, ld_Z : OUT STD_LOGIC;
80   T : OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
81   wen, en : OUT STD_LOGIC;
82 );
83 END COMPONENT;
84
85 COMPONENT reset_circuitIS
86 PORT (
87   Reset : IN STD_LOGIC;
88   clk : IN STD_LOGIC;
89   Enable_PD : OUT STD_LOGIC;
90   Clr_PC : OUT STD_LOGIC;
91 );
92 END COMPONENT;
93
94 SIGNAL dp_mux1, dp_clrA, dp_ldA, dp_clrB, dp_ldB, dp_clrC, dp_ldC, dp_clrZ,
95 dp_ldZ, memWEN, memEN, dp_muxA, dp_muxB : STD_LOGIC;
96 SIGNAL mux_data, reg, enpd, irlc, irlid, pinc, pclr, pcld, out0, out1, out7, out8 : STD_LOGIC;
97 SIGNAL outIR : STD_LOGIC_VECTOR(31 DOWNTO 0);
98 SIGNAL alu : STD_LOGIC_VECTOR(2 DOWNTO 0);
99 SIGNAL dp_mux2, dp_muxData : STD_LOGIC_VECTOR(1 DOWNTO 0);
100
101 BEGIN
102   dat : Data_Path
103 PORT MAP(
104   Clk => clk,

```

```

105 mclk => mem_clk,
106 wen => memWEN,
107 en => memEN,
108 clr_A => dp_clrA,
109 ld_A => dp_ldA, clr_B => dp_clrB, ld_B => dp_ldB, clr_C => dp_clrC, ld_C
110 => dp_ldC, clr_Z => dp_clrZ, ld_Z => dp_ldZ,
111 clr_PC => pclr, ld_PC => pld, clr_IR => irlc, ld_IR => irld,
112 out_A => doutA, out_B => doutB, out_C => out0, out_Z => out1, out_PC =>
113 doutPC, out_IR => outIR, inc_PC => pinc,
114 ADDR_OUT => addrout, DATA_IN => dataIn, DATA_BUS => dataOut, DATA_Mux
115 => dp_muxData, REG_Mux => reg, A_MUX => dp_muxA, B_MUX => dp_muxB, IM_MUX1 =>
116 dp_mux1, IM_MUX2 => dp_mux2,
117 ALU_op => alu
118 );
119
120 control_unit: Control_NEW
121 PORT MAP(
122 clk => clk, mclk => mem_clk, enable => enpd, statusC => out0, statusZ =>
123 out1, INST => outIR,
124
125 A_Mux => dp_muxA, B_Mux => dp_muxB, IM_MUX1 => dp_mux1, REG_Mux => reg,
126 IM_MUX2 => dp_mux2, DATA_Mux => dp_muxData,
127 ALU_op => alu, inc_PC => pinc, ld_PC => pld, clr_IR => irlc, ld_IR =>
128 irld,
129 clr_A => dp_clrA, clr_B => dp_clrB, clr_C => dp_clrC, clr_Z => dp_clrZ,
130 ld_A => dp_ldA, ld_B => dp_ldB, ld_C => dp_ldC, ld_Z => dp_ldZ,
131 T => outT, wen => memWEN, en => memEN
132 );
133
134 reset : reset_circuit
135 PORT MAP(
136 Reset => rst,
137 Clk => clk,
138 Enable_PD => enpd,
139 Clr_PC => pclr
140 );
141 doutC <= out0;
142 doutZ <= out1;
143 doutIR <= outIR;
144 wen_mem <= memWEN;
145 en_mem <= memEN;
146
147 END description;

```

Figure 2. VHDL implementation for connecting the data path, control unit, and the reset circuit.

Next, the instruction memory needs to be integrated into the system. To do so, a system memory was defined. It will take in the address, clock, data, write enable, and it will output the instruction at address once the memory is initialized. The write enable signal will be low for the entire simulation and it will only be high when the memory needs to be initialized. The initialization will take place during compilation or when the MIF file is updated. The address input will go in from the program counter and when write enable is low, the output will be the instruction at the address specified by the program counter. The VHDL implementation of this is shown below in Figure 3.

```

36  LIBRARY ieee;
37  USE ieee.std_logic_1164.all;
38
39  LIBRARY altera_mf;
40  USE altera_mf.all;
41
42  ENTITY system_memory IS
43  PORT
44  (
45      address : IN STD_LOGIC_VECTOR(5 DOWNTO 0);
46      clock    : IN STD_LOGIC := '1';
47      data     : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
48      wren     : IN STD_LOGIC;
49      q        : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
50  );
51  END system_memory
52
53
54  ARCHITECTURE SYN OF system_memory IS
55
56      SIGNAL sub_wire0 : STD_LOGIC_VECTOR(31 DOWNTO 0);
57
58
59
60      COMPONENT altsyncram
61      GENERIC (
62          clock_enable_input_a : STRING;
63          clock_enable_output_a : STRING;
64          init_file : STRING;
65          intended_device_family : STRING;
66          lpm_hint : STRING;
67          lpm_type : STRING;
68          numwords_a : NATURAL;
69          operation_mode : STRING;
70          outdata_aclr_a : STRING;
71          outdata_reg_a : STRING;
72          power_up_uninitialized : STRING;
73          widthad_a : NATURAL;
74          width_a : NATURAL;
75          width_byteena_a : NATURAL
76      );
77      PORT (
78          address_a : IN STD_LOGIC_VECTOR(5 DOWNTO 0);
79          clock0 : IN STD_LOGIC;
80          data_a : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
81          wren_a : IN STD_LOGIC;
82          q_a : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
83      );
84      END COMPONENT;
85
86  BEGIN

```

```

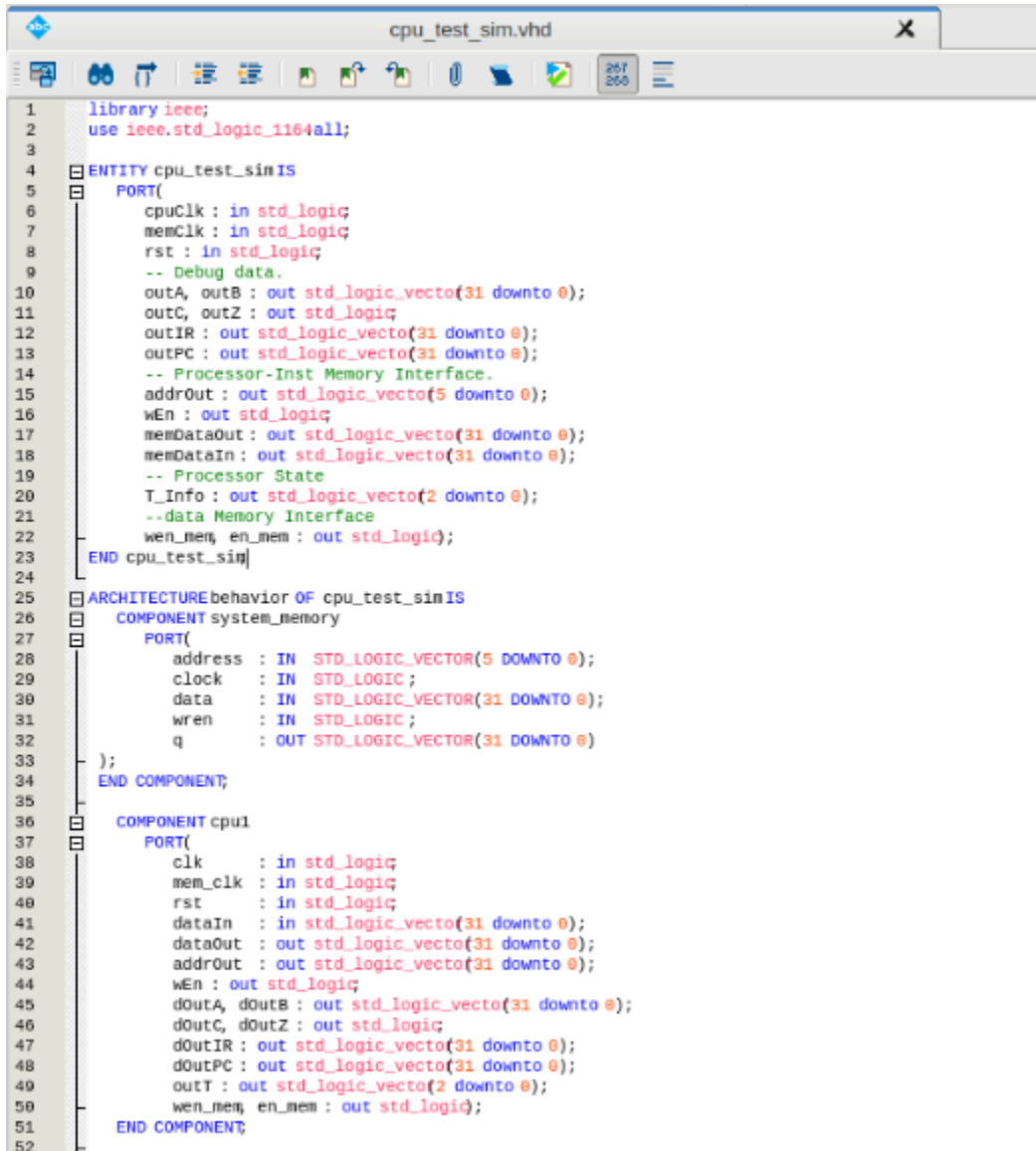
87      q <= sub_wire0(31 DOWNTO 0);
88
89      altsyncram_component: altsyncram
90      GENERIC MAP (
91          clock_enable_input_a => "BYPASS",
92          clock_enable_output_a => "BYPASS",
93          init_file => "system_memory.mif",
94          intended_device_family => "Cyclone II",
95          lpm_hint => "ENABLE_RUNTIME_MOD=NO",
96          lpm_type => "altsyncram",
97          numwords_a => 64,
98          operation_mode => "SINGLE_PORT",
99          outdata_aclr_a => "NONE",
100          outdata_reg_a => "CLOCK0",
101          power_up_uninitialized => "FALSE",
102          widthad_a => 6,
103          width_a => 32,
104          width_byteena_a => 1
105      )
106      PORT MAP (
107          address_a => address,
108          clock0 => clock,
109          data_a => data,
110          wren_a => wren,
111          q_a => sub_wire0
112      );
113
114
115
116  END SYN;

```

Figure 3. VHDL implementation of the system memory.

Finally, the system memory is integrated into the CPU system so that the instructions are loaded and executed automatically. To do so, the system memory component is called and the output from that is sent to the dataIn of cpu1 in Figure 2. To instantiate the memory, the address of the memory is taken from the program counter and the memory clock is used as the clock. Since

none of the instructions will update the instruction memory, wren will always be low and there won't be any data sent to the memory. At the end of the code, addrOut (current address), wEn (write enable for the system memory), memDataOut (output from the instruction memory), and memDataIn (output from the data multiplexer) is updated. The implementation of this is shown below in Figure 4.



```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  ENTITY cpu_test_sim IS
5  PORT
6      cpuClk : in std_logic;
7      memClk : in std_logic;
8      rst : in std_logic;
9      -- Debug data.
10     outA, outB : out std_logic_vector(31 downto 0);
11     outC, outZ : out std_logic;
12     outIR : out std_logic_vector(31 downto 0);
13     outPC : out std_logic_vector(31 downto 0);
14     -- Processor-Inst Memory Interface.
15     addrOut : out std_logic_vector(5 downto 0);
16     wEn : out std_logic;
17     memDataOut : out std_logic_vector(31 downto 0);
18     memDataIn : out std_logic_vector(31 downto 0);
19     -- Processor State
20     T_Info : out std_logic_vector(2 downto 0);
21     --data Memory Interface
22     wen_mem, en_mem : out std_logic;
23 END cpu_test_sim
24
25 ARCHITECTURE behavior OF cpu_test_sim IS
26     COMPONENT system_memory
27     PORT
28         address : IN  STD_LOGIC_VECTOR(5 DOWNTO 0);
29         clock : IN  STD_LOGIC;
30         data : IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
31         wren : IN  STD_LOGIC;
32         q : OUT  STD_LOGIC_VECTOR(31 DOWNTO 0)
33     );
34 END COMPONENT;
35
36     COMPONENT cpu1
37     PORT
38         clk : in std_logic;
39         mem_clk : in std_logic;
40         rst : in std_logic;
41         dataIn : in std_logic_vector(31 downto 0);
42         dataOut : out std_logic_vector(31 downto 0);
43         addrOut : out std_logic_vector(31 downto 0);
44         wEn : out std_logic;
45         doutA, doutB : out std_logic_vector(31 downto 0);
46         doutC, doutZ : out std_logic;
47         doutIR : out std_logic_vector(31 downto 0);
48         doutPC : out std_logic_vector(31 downto 0);
49         outT : out std_logic_vector(2 downto 0);
50         wen_mem, en_mem : out std_logic;
51     END COMPONENT;
52

```



```

53 signal cpu_to_mem std_logic_vector(31 downto 0);
54 signal mem_to_cpu std_logic_vector(31 downto 0);
55 signal add_from_cpu std_logic_vector(31 downto 0);
56 signal wen_from_cpu std_logic;
57
58 BEGIN
59   -- Component instantiations.
60   main_memory: system_memory
61     PORT MAP(
62       address => add_from_cpu(5 downto 0),
63       clock => memClk,
64       data => cpu_to_mem,
65       wren => wen_from_cpu,
66       q => mem_to_cpu
67     );
68   main_processor: cpu1
69     PORT MAP(
70       clk => cpuClk,
71       mem_clk => memClk,
72       rst => rst,
73       dataIn => mem_to_cpu,
74       dataOut => cpu_to_mem,
75       addrOut => add_from_cpu,
76       wEn => wen_from_cpu,
77       doutA => outA,
78       doutB => outB,
79       doutC => outC,
80       doutZ => outZ,
81       doutIR => outIR,
82       doutPC => outPC,
83       outT => T_Info,
84       wen_mem => wen_mem,
85       en_mem => en_mem
86     );
87
88   addrOut <= add_from_cpu(5 downto 0);
89   wEn <= wen_from_cpu;
90   memDataOut <= mem_to_cpu;
91   memDataIn <= cpu_to_mem;
92 END behavior;

```

Figure 4. VHDL implementation for connecting the instruction memory to the CPU system.

Waveforms and Simulations

The code from Figure 1 for the reset circuit is simulated to verify its output. The timing simulation in Figure 5 shows that it works as expected. When reset is 1 (near 60 ns), the clear PC signal will turn on and Enable_PD will turn off. Also, 4 clock cycles after the reset signal turns off (near 220 ns), the clear PC signal will turn off and the Enable_PD signal will turn on.

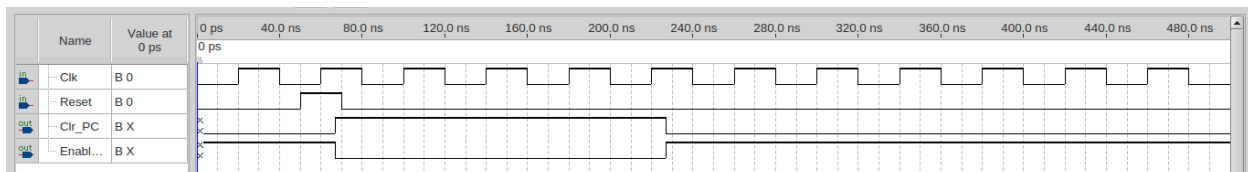


Figure 5. Timing simulation of the reset circuit.

Next, the code in Figure 4 was simulated to ensure that the full CPU works properly. This simulation should execute the instructions in system_memory.mif properly and move to the next instruction automatically. Figures 6 to 20 so the simulation results for all of the instructions the

CPU can execute. The first 100 ns of every simulation below don't do anything because the reset signal is high. Then, the signals will in the CPU stabilize from 100 ns to 260ns (4 clock cycles).

In the simulation below, first LDAI will execute at the end of T2. It loads 0000AAAA into register A as expected near 380 ns. Next, the value in register A will be stored into the data memory at address 1. This doesn't change any registers so all values remain the same except for the en and wen values which become 1 when the value is being stored. After that, the value in register A is cleared. Since clear is asynchronous, it will clear the value in register A as soon as clr_A is set to 1 near 580 ns. Finally, to ensure the STA instruction works, the value at address 1 in the data memory is loaded into register A. This happens at the end of T1 (near 700 ns). At this point, 0000AAAA is shown in register A and this means that the STA and LDA instructions work.

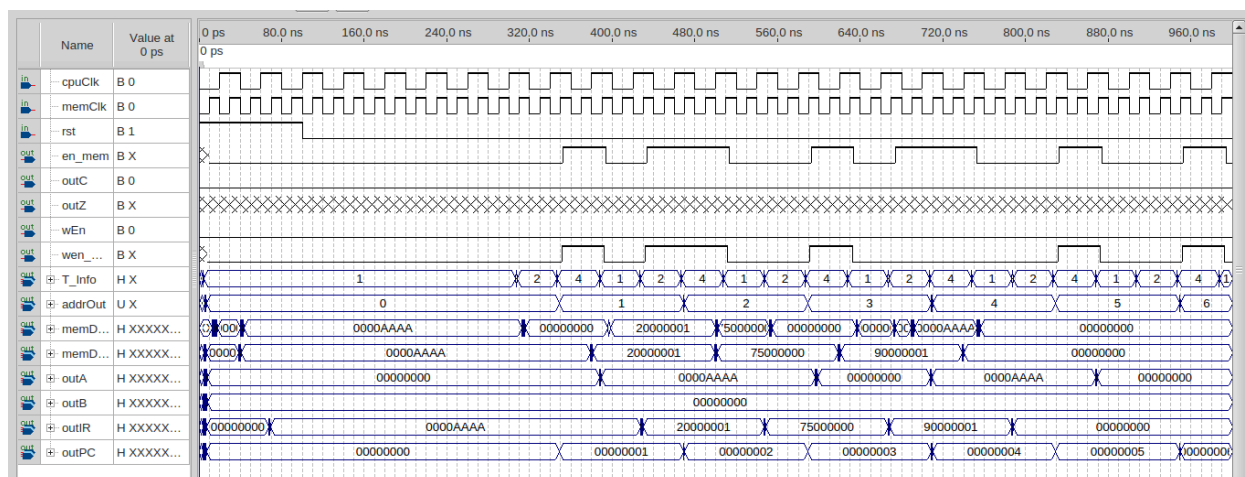


Figure 6. Timing simulation of the CPU test for LDAI, STA, CLRA, and LDA.

In the simulation below, the results are almost identical to those in Figure 6 except register B is used instead of register A and 0000BBBB is loaded instead of 0000AAAA. This is expected because the instructions that are executed are analogous to those executed above.

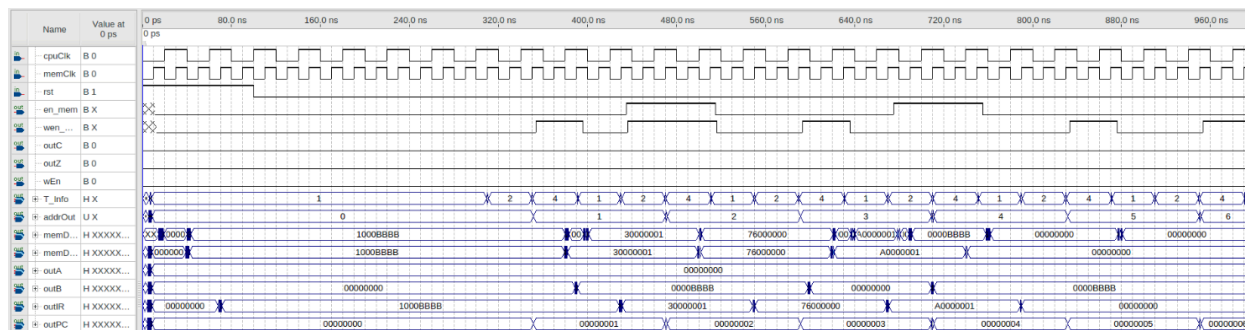


Figure 7. Timing simulation of the CPU test for LDBI, STB, CLRB, and LDB.

The simulation below executes the LUI instruction. It will load the immediate value into the upper 16 bits of register A. The instruction will be executed at the end of T2 near 380 ns. It

executes as expected because the upper 16 bits of register A are equal to the immediate value.

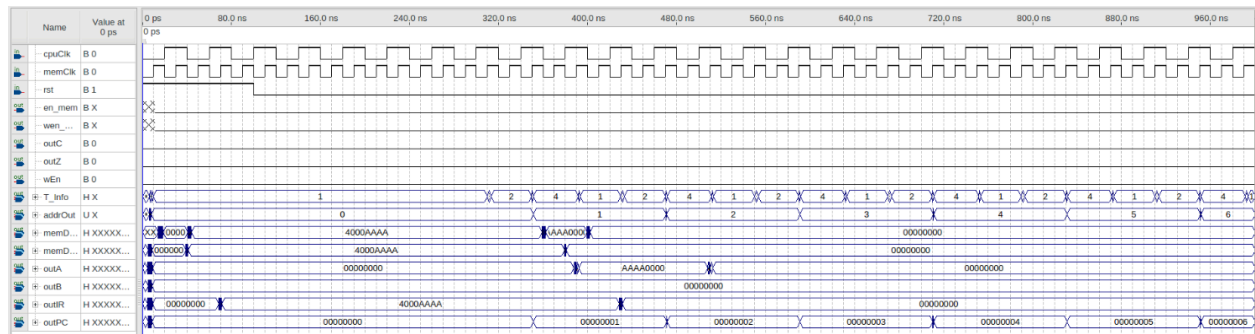


Figure 8. Timing simulation of the CPU test for LUI.

In the simulation below, the JMP instruction will be executed. Once this is executed, the PC value should change to the immediate value. At the end of T2, the value changes to 0000AAAA and then it starts counting up from there which shows that the code works as expected.

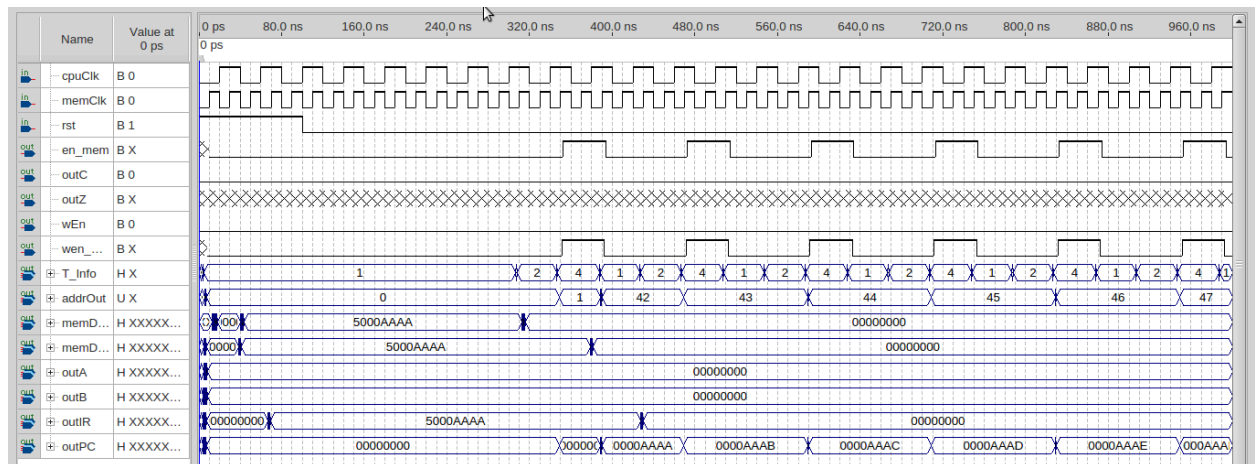


Figure 9. Timing simulation of the CPU test for JMP.

For the ANDI operation, the value in register A should be changed to A AND the immediate value. To verify this code, first 00000006 is loaded into register A using LDAI and then the ANDI operation is performed. It will do 00000006 & 0000000B and this should result in 00000002. At around 500ns, the value in register A changes to 00000002 which shows that the CPU can execute this instruction correctly.

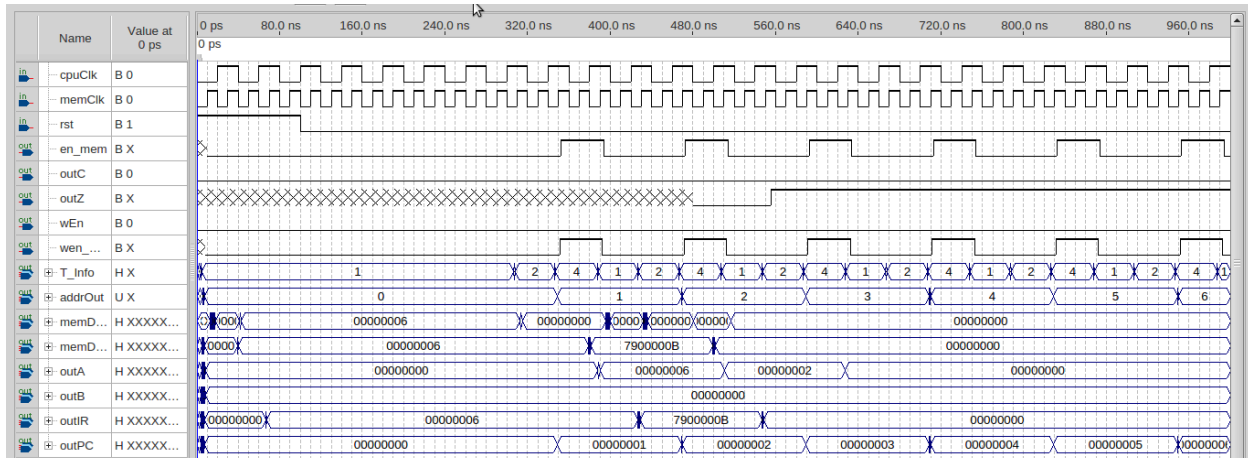


Figure 10. Timing simulation of the CPU test for ANDI.

The ADDI instruction will add the value in register A with the immediate value and load the sum back into register A. First 00000006 is loaded into register A using the LDAI operation. Then, the ADDI operation is executed. It will do $00000006 + 0000000B$ which should result in 00000011. The value in register A changes at T2 of the ADDI operation to 00000011 which shows that the CPU executes the ADDI operation correctly.

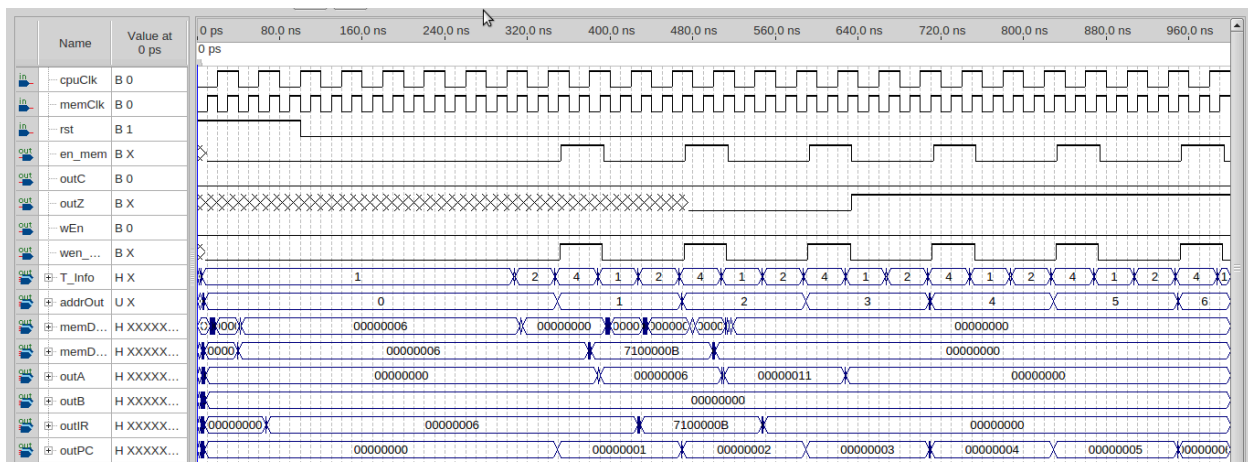


Figure 11. Timing simulation of the CPU test for ADDI.

This simulation is almost the same as the ANDI operation in Figure 10 except it will perform an OR operation instead of an AND operation. Doing 00000006 OR 0000000B will give 0000000F. This value is loaded into register A after T2 of the ORI instruction near 500 ns. This shows that the CPU executes the ORI instruction correctly.

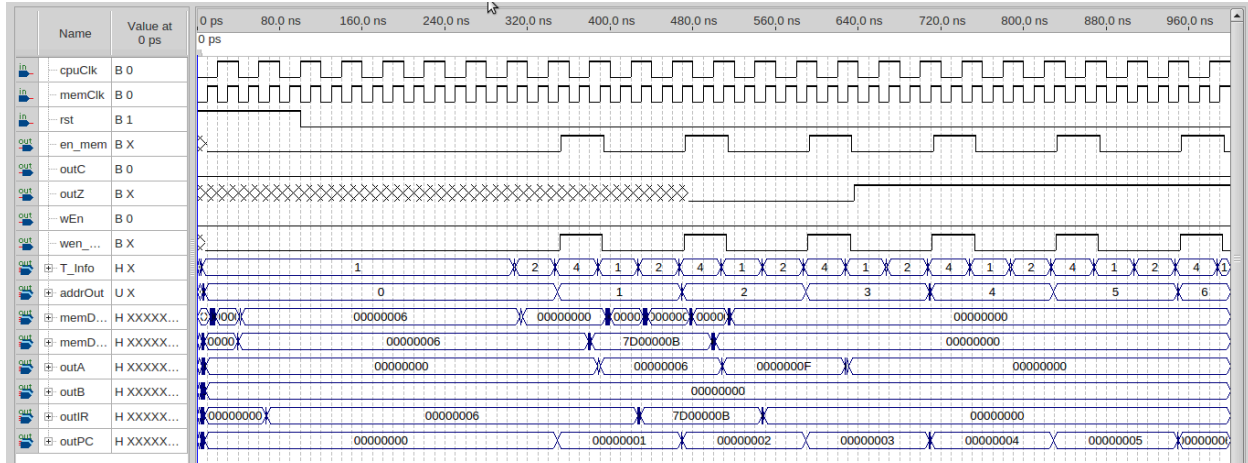


Figure 12. Timing simulation of the CPU test for ORI.

In Figure 13, the ADD operation is executed. This operation is expected to sum the value in register A and B and store the sum in register A. To test this operation, first LDAI is used to load 00000005 into register A and LDBI is used to load 00000003 into register B. Then the ADD operation is performed and this should do $00000005 + 00000003$. Therefore, 00000008 should be in register A after the instruction execution stage. Near 620 ns, the value in register A changes to 00000008 which shows that the CPU can execute the ADD instruction.

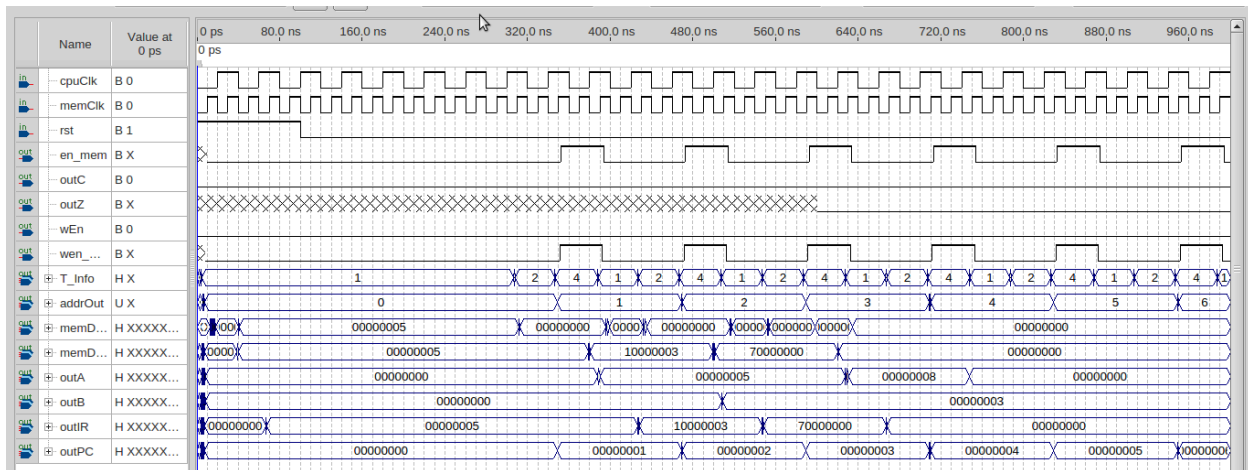


Figure 13. Timing simulation of the CPU test for ADD.

In Figure 14, the SUB operation is executed. This operation is expected to find the difference between the values in register A and B and store the difference in register A. To test this operation, first LDAI is used to load 00000005 into register A and LDBI is used to load 00000003 into register B. Then the SUB operation is performed and this should do $00000005 - 00000003$. Therefore, 00000002 should be in register A after the instruction execution stage. Near 620 ns, the value in register A changes to 00000002 which shows that the CPU can execute the SUB instruction.

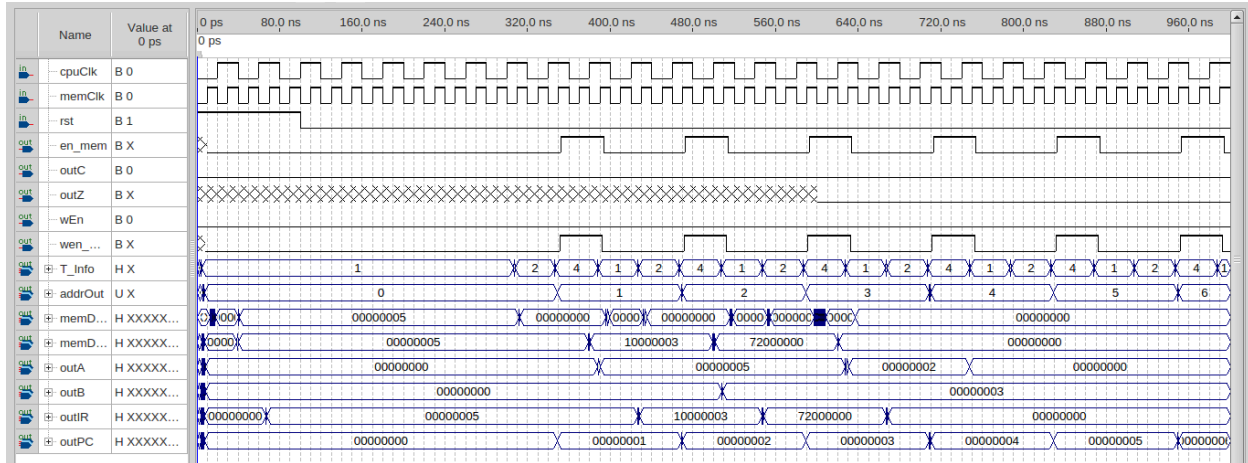


Figure 14. Timing simulation of the CPU test for SUB.

In Figure 15, the DECA operation is executed. This operation is expected to reduce the value in register A by 1. To test this operation, first LDAI is used to load 0000AAAA. Then the DECA operation is performed and this should do $0000AAAA - 00000001$. Therefore, 0000AAA9 should be in register A after the instruction execution stage. Near 500ns, the value in register A changes to 0000AAA9 which shows that the CPU can execute the DECA instruction.

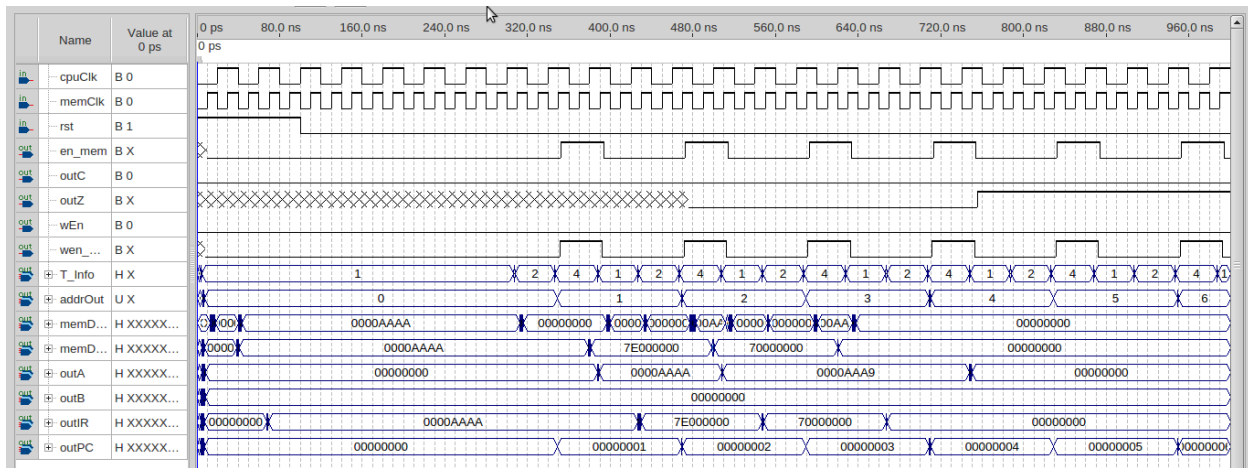


Figure 15. Timing simulation of the CPU test for DECA.

In Figure 16, the INCA operation is executed. This operation is expected to reduce the value in register A by 1. To test this operation, first LDAI is used to load 0000AAAA. Then the INCA operation is performed and this should do $0000AAAA + 00000001$. Therefore, 0000AAAB should be in register A after the instruction execution stage. Near 500ns, the value in register A changes to 0000AAAB which shows that the CPU can execute the INCA instruction.

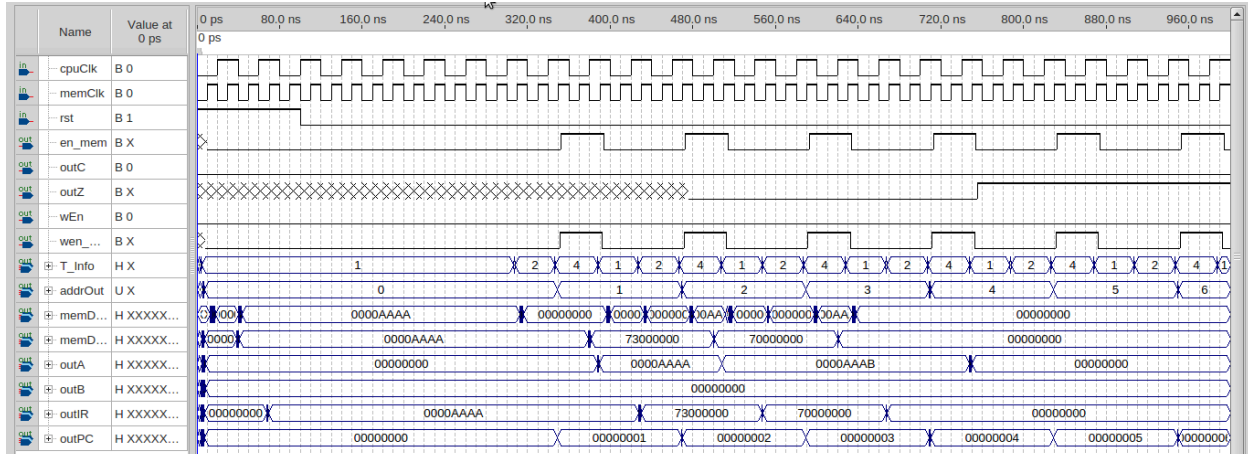


Figure 16. Timing simulation of the CPU test for INCA.

In Figure 17, the ROL operation is executed. This operation is expected to shift the bits in register A to the left. To test this operation, first LDAI is used to load 00000008. Then the INCA operation is performed and this should move the bits one index to the left and set the LSB to zero. Therefore, 00000010 should be in register A after the instruction execution stage. Near 500ns, the value in register A changes to 00000010 which shows that the CPU can execute the ROL instruction.

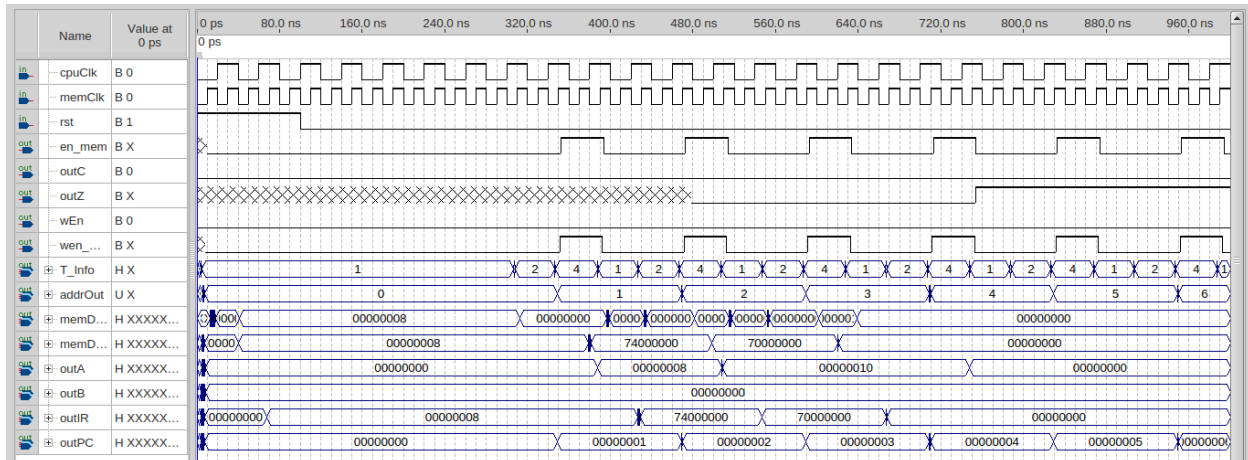


Figure 17. Timing simulation of the CPU test for ROL.

In Figure 18, the ROR operation is executed. This operation is expected to shift the bits in register A to the right. To test this operation, first LDAI is used to load 00000008. Then the ROR operation is performed and this should move the bits one index to the right and set the MSB to zero. Therefore, 00000004 should be in register A after the instruction execution stage. Near 500ns, the value in register A changes to 00000004 which shows that the CPU can execute the ROR instruction.

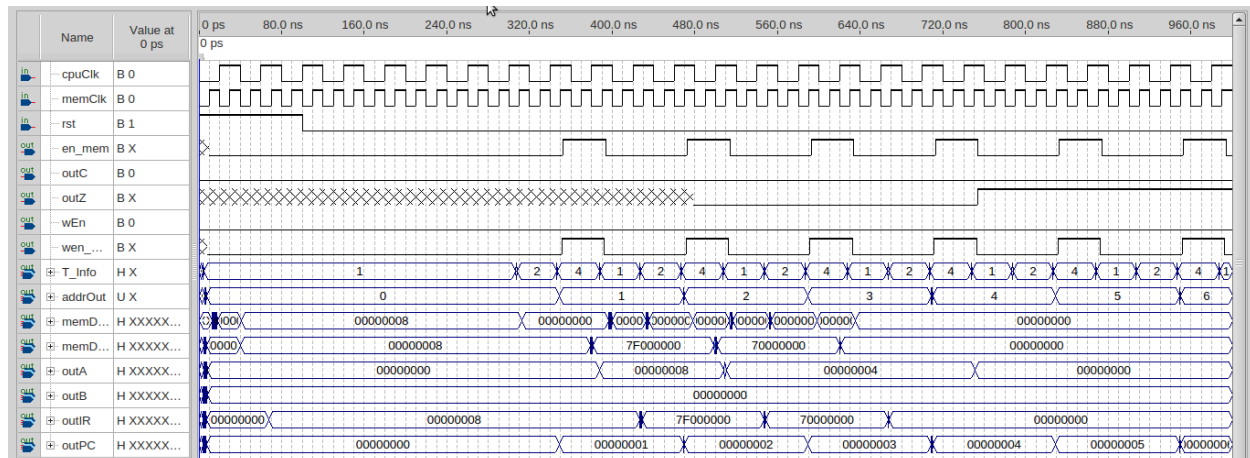


Figure 18. Timing simulation of the CPU test for ROR.

In Figure 19, the BEQ operation is executed. This operation is expected to branch to a memory address if the values of register A and B match. To test this operation, first LDAI is used to load 0000AAAA into register A and LDBI is used to load 0000AAAA into register B. Then the BEQ operation is performed and this should compare the values of register A and B and since they match it jumps to the set memory address. Therefore, 000000F0 should be in the PC counter register after the instruction execution stage. Near 620ns, the value in the PC counter register changes to 000000F0 which shows that the CPU can execute the BEQ instruction.

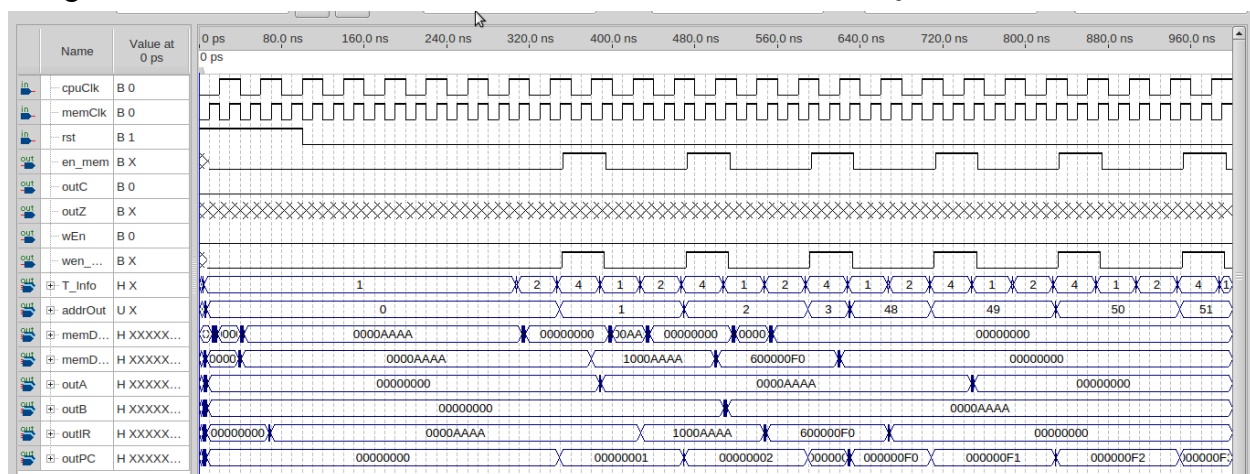


Figure 19. Timing simulation of the CPU test for BEQ.

In Figure 20, the BNE operation is executed. This operation is expected to branch to a memory address if the values of register A and B do not match. To test this operation, first LDAI is used to load 0000AAAA into register A and LDBI is used to load 0000BBBB into register B. Then the BNE operation is performed and this should compare the values of register A and B and since they do not match it jumps to the set memory address. Therefore, 000000F0 should be in the PC counter register after the instruction execution stage. Near 620ns, the value in the PC

counter register changes to 000000F0 which shows that the CPU can execute the BNE instruction.

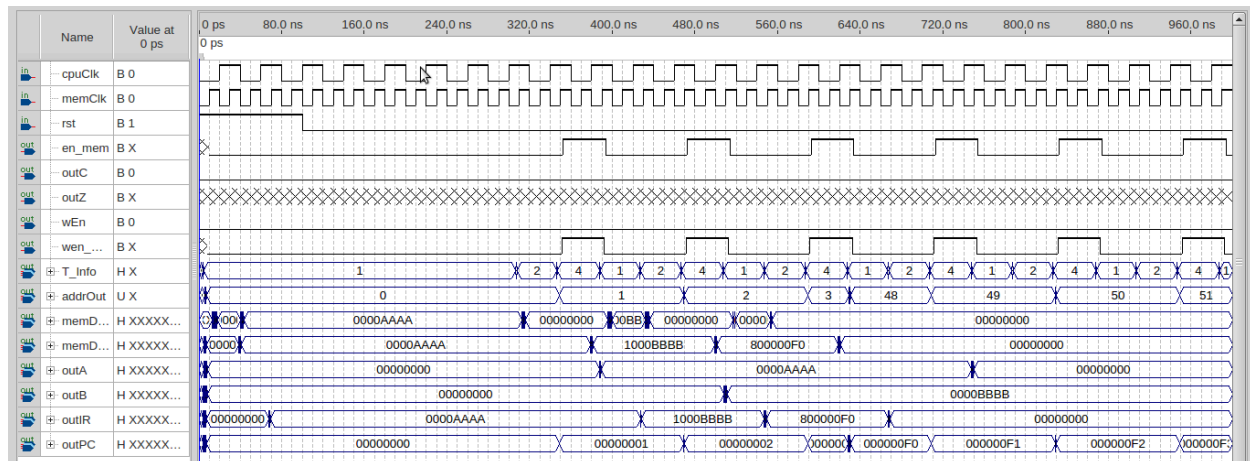


Figure 20. Timing simulation of the CPU test for BNE.

FPGA Emulation (Bonus)

The instruction set for this CPU is used to execute the following instructions:

1.

```
If (a == b) {
    branch1();
}
Else if (a > b) {
    branch2();
}
Else {
    Continue;
}
```

2.

```
a = 1
for (i = 1; i < 6; i++) {
    a = a * 2i;
}
```

For the first problem, the BEQ instruction was used to determine if a is equal to b. If it is then it will branch to branch1. Otherwise, the program continues to check if a is greater than b. Since there isn't a BGT instruction, first a SUB operation was used. If a is greater than b then the result of SUB should be positive otherwise it should be negative. To check if the result of the SUB was positive, an AND operation was performed between register A (result of the SUB) and 80000000. For negative values, the AND should result in a nonzero value because the MSB of a

negative value is 1. For positive values, the AND operation will result in 0 because the MSB of a positive value is 0. The result of this AND operation is stored in register A. Then, register B is cleared and a BEQ operation is performed so the program branches if register A has a 0 (only possible when $a > b$). If it does not branch then it means $a < b$ so the program continues. The assembly code for this is shown below.

```
LDAI a // Load a into register A
LDBI b // Load b into register B

BEQ branch1 // Branches if A and B are equal

SUB
LUI 8000 // Make MSB of register B 1
AND // Check if MSB of register A is 1 or 0
CLRB // Make B = 0
BEQ branch2 // Branches if a > b

// Continues if a < b

branch1: // Branch 1 continues here

branch2: // Branch 2 continues here
```

For the second problem, 1 was loaded into register A and 0 will be loaded into register B. Register B will keep the value of the loop counter. Since there isn't any multiply operation for this CPU, during every loop, a was rotated left i times. To do so, a nested loop is used which goes from 0 to i (value in register B). Since there are only 2 registers, the store and load operations are used to use the same register multiple times for different values. The assembly code for this is shown below.

```
LDAI 0001 // Load 1 into register A
LDBI 0000 // Load 0 into register B

STA 1 // Store value of register A into the data memory at address 1

// The next 5 lines will increment B
LOOP: STB 2 // Start the loop and store register B into the data memory at address 2
LDA 2 // Load it into register A
INCA // Increment A by 1
STA 2 // Store the incremented value into the data memory at address 2
LDB 2 // Load the value at address 2 into B

LDAI 0006 // Set the end of the loop in register A
```

```
BEQ END // If B = A then the loop executed 5 times and the loop condition isn't satisfied

LOOP_2: LDAI 0000 // Start the nested loop and store 0 in register A (A is the loop counter)
BEQ LOOP // If the nest loop condition is false (A = B) then branch to the main loop
STA 0 // Store the nested loop counter at address 0

LDA 1 // Load the value of A into register A
ROL // Multiply by 2
STA 1 // Store the value of register A at address 1

LDA 0 // Load back the nested loop counter
INCA // Increment the nested loop counter
JMP LOOP_2 // Go back to the start of the nested loop

END: LDA 1 // Load the value of A into register A
```

The assembly code above was converted to instructions that can be understood by the CPU and the instructions are written to the system_memory.mif file. Next, the BSF file was compiled and the Quartus Programmer was used to program the FPGA board. KEY0 was used to increment the program counter and every click will move the CPU to the next stage. The value of register A and B are shown on the LCD and the PC value was shown on the seven segment display. By emulating the programs on the FPGA board, the assembly code above was verified.

Conclusion

In conclusion, this lab successfully implemented and tested a fully functional CPU by integrating the datapath, control unit, reset circuit, and system memory. The reset circuit ensured proper initialization, while the final CPU system, including the instruction memory unit, demonstrated correct instruction execution through simulation. The results verified that the CPU correctly performed various operations, including arithmetic, logic, and branching, while correctly loading instructions from memory. Additionally, the bonus question was successfully completed and pushed to FPGA, demonstrating the implementation of branching logic and loop execution within the CPU.

References

Toronto Metropolitan University. *CPU_Specification*. (2017). COE 608: Computer Organization and Architecture. <https://www.ee.torontomu.ca/~courses/coe608/lab-index.html>

Toronto Metropolitan University. *CPU Testing for Lab-6*. (2017). COE 608: Computer Organization and Architecture. <https://www.ee.torontomu.ca/~courses/coe608/lab-index.html>

Toronto Metropolitan University. *The Complete CPU (Overall Project)*. (2017). COE 608: Computer Organization and Architecture. <https://www.ee.torontomu.ca/~courses/coe608/lab-index.html>