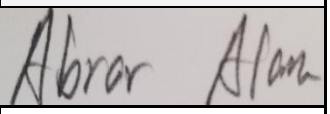
Ryerson University

Department of Electrical, Computer, & Biomedical Engineering
Faculty of Engineering & Architectural Science

Course Title:	Computer Organization and Architecture
Course Number:	COE 608
Semester/Year (e.g.F2016)	Winter 2021
Instructor:	Professor Nagi Mekhiel

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

Submission Date:	April 10, 2021
Due Date:	April 10, 2021

Student LAST Name	Student FIRST Name	Student Number	Section	Signature*
ALAM	ABRAR	500725366	03	

*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.ryerson.ca/senate/current/pol60.pdf

Table of Contents

Objective 2

Design and implementation 2

CPU Reset Circuitry 2

The Complete CPU System 2

Results and Conclusions 4

Part I – The Reset Circuit 4

Part II – The Complete CPU System 5

ADD 5

BNE 7

ANDI 8

JMP 10

ORI 11

Reference 13

Appendix 14

The Reset Circuit 14

Modified Control Unit 15

ADD module that increments the PC 36

CPU Test Simulator 37

Objective

The objective of this final experiment was to finalize a fully functional CPU that was built by connecting already implemented data-path (in lab 4b), and control unit (in lab 5). To achieve this, an additional circuit called **Reset Circuit** was implemented in VHDL. Besides, 3 additional files, which were already given to the students on D2L, were used to simulate the Instruction Memory (IM), and ease the compilation and testing processes.

Design and implementation

CPU Reset Circuitry

To achieve a stable connection between the already implemented Control Unit, and Data-Path, along with stable data surrounding the CPU prior to its operation, a synchronous (Clock driven) **Reset Circuit** was implemented first. **Figure 1** shows the block diagram of the **Reset** Circuit implemented in VHDL.

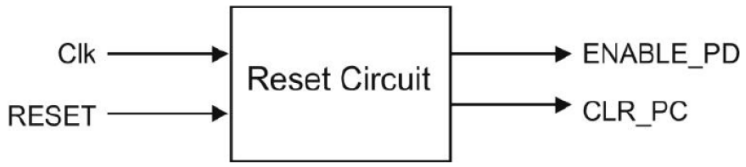


Figure 1: The synchronous Reset Circuit that was implemented in this lab.

The **Reset Circuit** outlined in **figure 1** works as follows: when the **RESET** input is set to high, **ENABLE_PD** goes low, which in turn forces the Control Unit into state T0, and output **CLR_PC** goes high, which clears the Program Counter. It is important to note that clearing the PC will reset the memory location to 0x00000000, which is the starting address of the CPU program. In contrast, when **RESET** goes low, **ENABLE_PD** & **CLR_PC** remains low & high respectively for 4 clock cycles. This is to ensure that the data surrounding the CPU is stabilized before the operation. Besides, this circuit will count and keep track of three clock cycles (T0, T1, and T2).

The Complete CPU System

Now the data-path, and the Control Unit module implemented in previous labs was connected via the VHDL modules provided, namely, “CPU_TEST_Sim.vhdl”, which, in turn relies on the implemented Reset Circuit, another provided system memory VHDL module that simulates the Instruction Memory (M[INST], see see **figure 2**, **figure 3**). No further modifications were made to the previously designed components from past labs, aside from slight name changes. Please note that since the student has already tested the data-path, and the Control Module in past labs, all we need in this lab is to enter each supported CPU instruction according to the format outlined in **Table 1** (taken from the CPU specification document posted on D2L). This table also shows what each instruction is supposed to do (see the **Function** column).

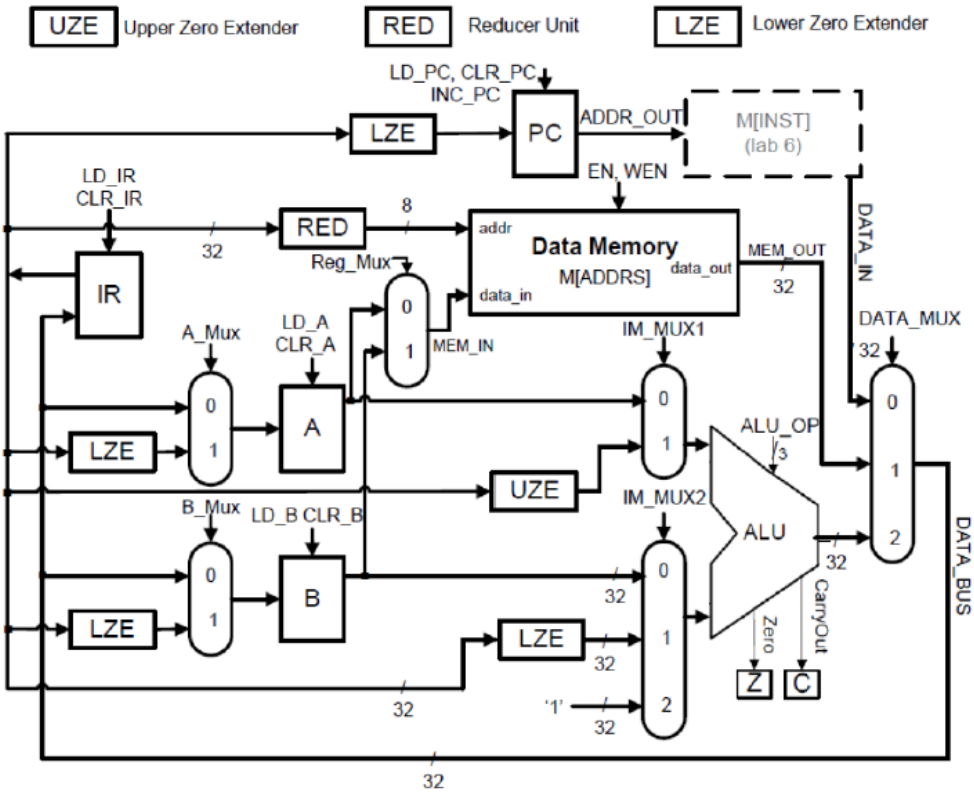


Figure 2: The outlined Instruction memory module which has been simulated in this lab by using the provided "system_memory.vhd" file

Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
0	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
8	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
16	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
24	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
32	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
40	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
48	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
56	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....

Figure 3: The Instruction Memory (M[INST], also seen in figure 2) console generated using the provided "system_memory.mif" file. This is where we typed in various instructions (each block representing individual addresses) for testing.

Mnemonic	Function	Instruction Word		
		IR[31..28]	IR[27..16]	IR[15..0]
LDAI	A <= IR[15:0]	0000	X	IMM
LDBI	B <= IR[15:0]	0001	X	IMM
STA	M[ADDRS] <= A, ADDRS <= IR[15:0]	0010	X	ADDRS
STB	M[ADDRS] <= B, ADDRS <= IR[15:0]	0011	X	ADDRS
LDA	A <= M[ADDRS] , ADDRS <= IR[15:0]	1001	X	ADDRS
LDB	B <= M[ADDRS] , ADDRS <= IR[15:0]	1010	X	ADDRS
LUI	A[31:16] <= IR[15:0], A[15:0] <= 0	0100	X	IMM
JMP	PC <= IR[15..0]	0101	X	ADDRS
BEQ	IF(A==B) then PC <= IR[15..0]	0110	X	ADDRS
BNE	IF(A!=B) then PC <= IR[15..0]	1000	X	ADDRS
		IR[31..28]	IR[27..24]	IR[15..0]
ADD	A <= A + B	0111	0000	X
ADDI	A <= A + IR[15..0]	0111	0001	IMM
SUB	A <= A - B	0111	0010	X
INCA	A <= A + 1	0111	0011	X
ROL	A <= A << 1	0111	0100	X
CLRA	A <= 0	0111	0101	X
CLRB	B <= 0	0111	0110	X
CLRC	C <= 0	0111	0111	X
CLRZ	Z <= 0	0111	1000	X
ANDI	A <= A AND IR[15..0]	0111	1001	IMM
TSTZ	If Z = 1 then PC <= PC + 1	0111	1010	X
AND	A <= A AND B	0111	1011	X
TSTC	If C = 1 then PC <= PC + 1	0111	1100	X
ORI	A <= A OR IR[15..0]	0111	1101	IMM
DECA	A <= A - 1	0111	1110	X
ROR	A <= A >> 1	0111	1111	X

Table 1: CPU instruction functionalities, and their formats, which were the basis of the testing and timing simulations of various instructions in this lab.

Results and Conclusions

Part I – The Reset Circuit

The VHDL source code of the Reset Circuit has been presented in the **Appendix** section.

Figure 4 shows the timing simulation waveform of the Reset Circuit. From **figure 6** we observe that the input bit **Reset** goes high and successfully latched upon the rising edge of the clock signal, **Clk**, at around 60 ns. As expected, we see that the output signals, **Clr_PC** goes high at that point, and stays high for about 4 clock cycles, while the **Enable_PD** output goes low, and stays low also for 4 clock cycles. We notice that the delays are too insignificant to hamper the Reset Circuit's normal operations. This was proved by the successful implementation and testing of the CPU (implemented after this Reset Circuit, and this circuit was used to build the connections between the already implemented data-path, and the Control Module).

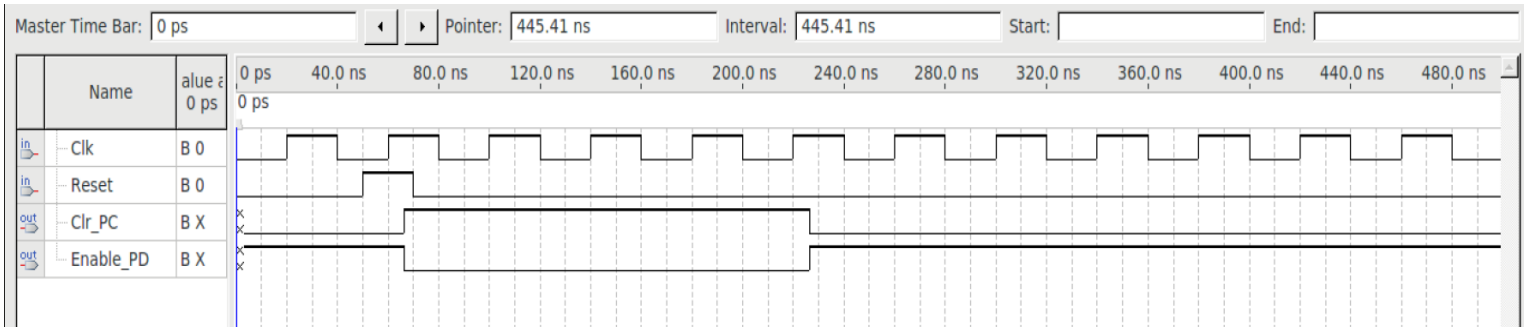


Figure 4: The timing simulation waveform of the Reset Circuit, showing the desired behavior without no significant delays.

Part II – The Complete CPU System

The relevant source codes for the complete CPU have been presented in the **Appendix** section.

Please note that the student has already tested the correctness of **all** instructions with timing waveforms. Therefore, for the sake of brevity, **5** set of instructions of various types have been discussed in this report.

ADD

To test this instruction, we plan to follow the following steps

1. Load the first operand in one of the registers (Register A).
2. Load the second operand in another register (Register B).
3. Place the **ADD** instruction into the Instruction Memory.
4. Read the result from the Register A (according to our design, see **table 1**).

We achieved **step 1**, by placing a **LDAI** (I type, where IMM = 5 = first operand) instruction at the very first address of the Instruction Memory (see **figure 5**, the content of the row 0, column 0). It is important to note that we are following the exact instruction format for **LDAI** outlined in **table 1**.

Then we achieved **step 2**, by placing a **LDBI** (I type, where IMM = 3 = second operand) instruction in the next address (see **figure 5**, the content of the row 0, column 1). Please observe that we are following the exact instruction format for **LDBI** outlined in **table 1**.

Next, we achieved **step 3** by pacing **ADD** instruction (R type) in the address that follows **step 2** (see **figure 5**, the content of the row 0, column 2). Please note that we are also following the exact instruction format for **ADD** outlined in **table 1**.

Finally, we expect to see the result stored in **Register A** being **5+3 = 8** (see **figure 6** waveform, the content of **outA** showing that we have **8** saved in Register A as the result, at around 640 ns).

Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
0	00000005	10000003	70000000	00000000	00000000	00000000	00000000	00000000	.....
8	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
16	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
24	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
32	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
40	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
48	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
56	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....

Figure 5: The series of instructions that was placed in the Instruction Memory to test ADD instruction.

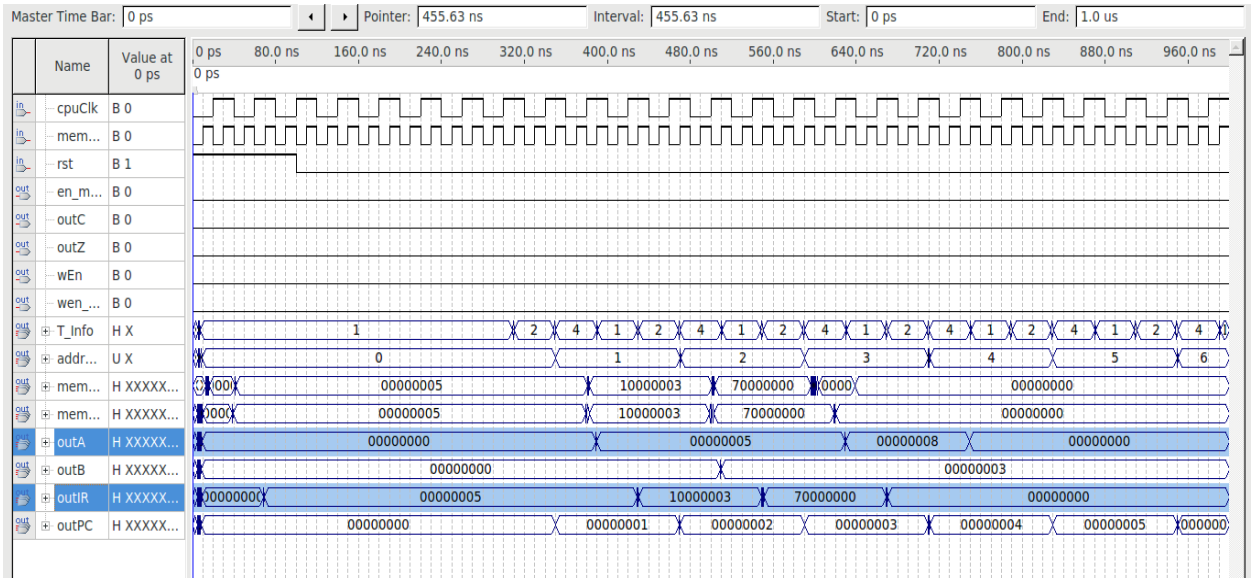


Figure 6: Timing simulation waveform of the ADD instruction

From **figure 6**, we see that we successfully loaded the Register A (**outA**) with the value of the first operand = 5 (around 400 ns). Then, we also load the second operand = 3 in Register B (**outB**) at around 560 ns. Slightly after 630 ns we have the addition result of 5+3 = 8 stored back in Register A (**outA**). Thus, our CPU is performing **ADD** operation as we expect.

BNE

We planned to test the Branch if not Equal (BNE) instruction according to following steps:

1. Load the first operand in one of the registers (Register A).
2. Load the second operand (different from the first operand from **step 1**) in another register (Register B).
3. Place the **BNE** instruction into the Instruction Memory.
4. Read the updated Program Counter (PC) value (according to our design, see **table 1**).

We achieved **step 1**, by placing a **LDAI** (I type, where IMM = $AAAA_{(HEX)}$ = first operand) instruction at the very first address of the Instruction Memory (see **figure 7**, the content of the row 0, column 0). It is important to note that we are following the exact instruction format for **LDAI** outlined in **table 1**.

Then we achieved **step 2**, by placing a **LDBI** (I type, where IMM = $BBBB_{(HEX)}$ = second operand) instruction in the next address (see **figure 7**, the content of the row 0, column 1). Please observe that we are following the exact instruction format for **LDBI** outlined in **table 1**.

Next, we achieved **step 3** by pacing **BNE** instruction in the address that follows **step 2** (see **figure 7**, the content of the row 0, column 2). Please note that we are also following the exact instruction format for **BNE** outlined in **table 1**. Please note that our branch target address will be (**$00F0_{(HEX)}$**) will be appended to the upper 16 bits of the current PC value (which is **$0000_{(HEX)}$**) to form a new PC value, since $AAAA_{(HEX)}$ is **not equal** to $BBBB_{(HEX)}$, according to **table 1**.

Finally, we expect to see the updated PC value in **outPC** being **$000000F0_{(HEX)}$** (see **figure 8** waveform, the content of **outPC** showing that we have **$000000F0_{(HEX)}$** saved in the Program Counter (PC) as the result, at around 650 ns).

Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
0	0000AAAA	1000BBBB	800000F0	00000000	00000000	00000000	00000000	00000000	.....
8	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
16	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
24	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
32	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
40	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
48	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
56	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....

Figure 7: The series of instructions that need to be placed in the Instruction Memory to test BNE instruction.

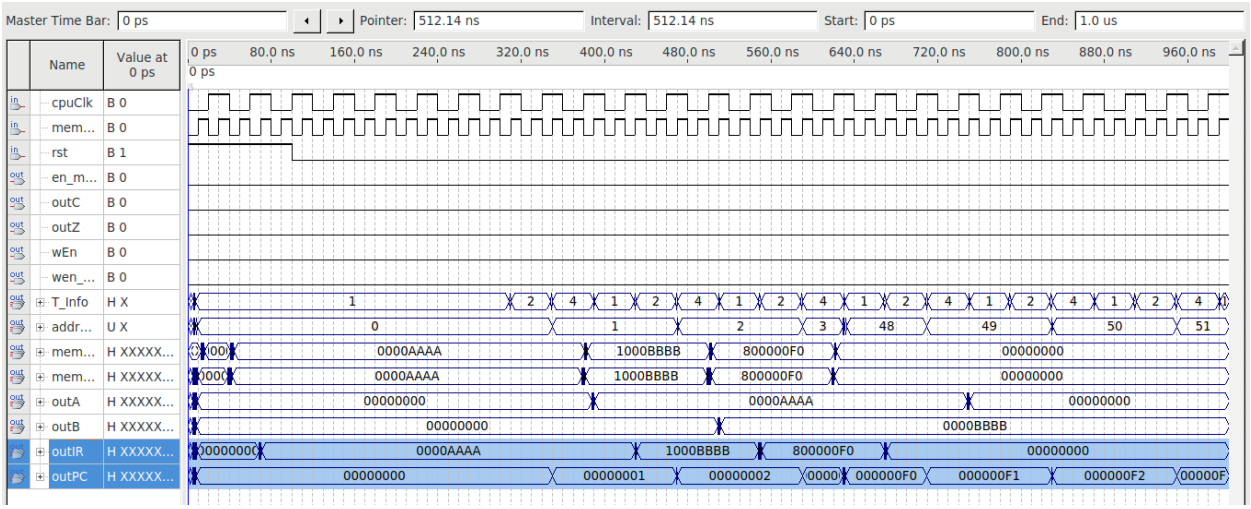


Figure 8: Timing simulation of the BNE instruction.

From **figure 8**, we see that we successfully loaded the Register A (**outA**) with the value of the first operand = $0000AAAA_{(HEX)}$ (around 400 ns). Then, we also load the second operand = $0000BBBB_{(HEX)}$ in Register B (**outB**) at around 560ns. Slightly after 630 ns we see the updated PC value of **$000000F0_{(HEX)}$** in the Program Counter (**out_PC)**. Thus, our CPU is performing **BNE** operation as we expect.

ANDI

The AND Immediate instruction (ANDI) performs bitwise AND between the content already stored in **Register A**, and the given **immediate value** given with the **ANDI** instruction (to comply with **table 1**).

We plan to test the ANDI instruction according to following steps:

1. Load Register A with any desired value
2. Place the **ANDI** instruction into the Instruction Memory (this includes the immediate value, which will be bitwise ANDed with the content already stored in **Register A.**
3. Read the AND result stored in Register A (according to **table 1**).

We achieved **step 1**, by placing a **LDAI** (I type, where IMM = $0006_{(HEX)}$) instruction at the very first address of the Instruction Memory (see **figure 9**, the content of the row 0, column 0). It is important to note that we are following the exact instruction format for **LDAI** outlined in **table 1**.

Next, we achieved **step 2** by pacing **ANDI** instruction in the address that follows **step 1** (see **figure 9**, the content of the row 0, column 1). Please note that we are also following the exact instruction format for **ANDI** outlined in **table 1**. Please note that since **$0006_{(HEX)}$ AND $000B_{(HEX)}$ = $0002_{(HEX)}$,** we expect to see the result stored in Register A to be **$00000002_{(HEX)}$** (after converting from 16 bit to 32 bit). This is exactly seen in the timing simulation waveform presented in **figure 10** (the content of **outA** at around 560 ns).

Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
0	00000006	7900000B	00000000	00000000	00000000	00000000	00000000	00000000	....
8	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	....
16	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	....
24	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	....
32	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	....
40	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	....
48	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	....
56	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	....

Figure 9: The series of instructions that need to be placed in the Instruction Memory to test ANDI instruction.

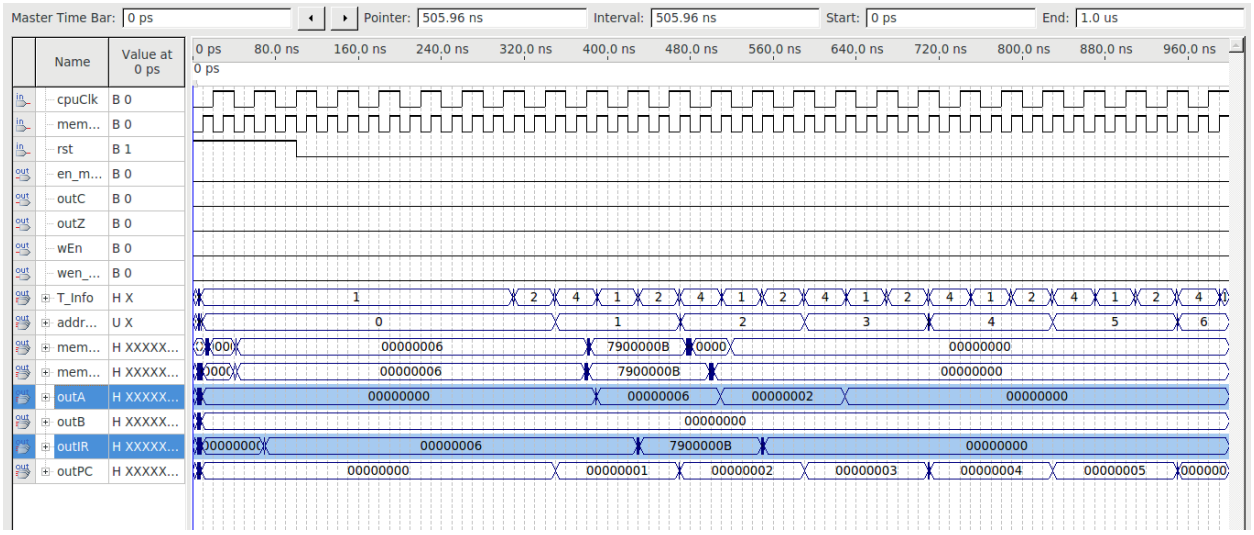


Figure 10: The timing simulation waveform of the ANDI instruction.

From **figure 10**, we see that we successfully loaded the Register A (**outA**) with the initial value = $00000006_{(HEX)}$ (around 400 ns). Then, we place the second instruction into the Instruction Memory (**outIR**) at around 480 ns with the immediate value = $000B_{(HEX)}$. At around 560 ns 630 ns we see the bitwise AND result between the content of Register A and the immediate value (value of **$00000002_{(HEX)}$**) in Register A (**outA)**. Thus, our CPU is performing **ANDI** operation as we expect.

JMP

Jump instruction appends the lower 16 bits of Instruction Register value (that is, the immediate field supplied with JMP instruction stored in the Instruction Memory (IM)) with the current upper 16 bits of the Program Counter (PC) value to form a new PC value to which the program will jump to upon the next rising edge of the clock cycle.

To test JMP instruction we just need to place this instruction in a specific address of the Instruction Memory (IM). The immediate value (IMM) supplied with this instruction will be the relative jump address which will be appended to the higher 16 bit of the current Program Counter (PC) value. In this lab, we placed JMP instruction at the address 0 of the Instruction Memory (row 0, column 0 of the **figure 11**). From **figure 11**, we see expect to see our current PC value to be updated as **0000AAAA$_{(HEX)}$** upon executing this instruction. After this the PC will be incremented as it normally happens. Please note that we are following the instruction format shown in **table 1**.

Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
0	5000AAAA	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
8	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
16	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
24	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
32	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
40	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
48	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
56	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....

Figure 11: The JMP instruction placed in the very first address of the Instruction Memory (IM) to test and simulate.

Figure 12 shows the timing simulation of JMP instruction. We observe that slightly after the Instruction Register's (**outIR**) content being updated to the instruction of JMP at around 80 ns, the Program Counter's (PC) value was updated to be **0000AAAA$_{(HEX)}$** at around 400 ns (the lower 16 bit of the PC = **AAAA$_{(HEX)}$** comes from the immediate value that was supplied with the **JMP** instruction itself.). This proves the correctness of the JMP instruction.

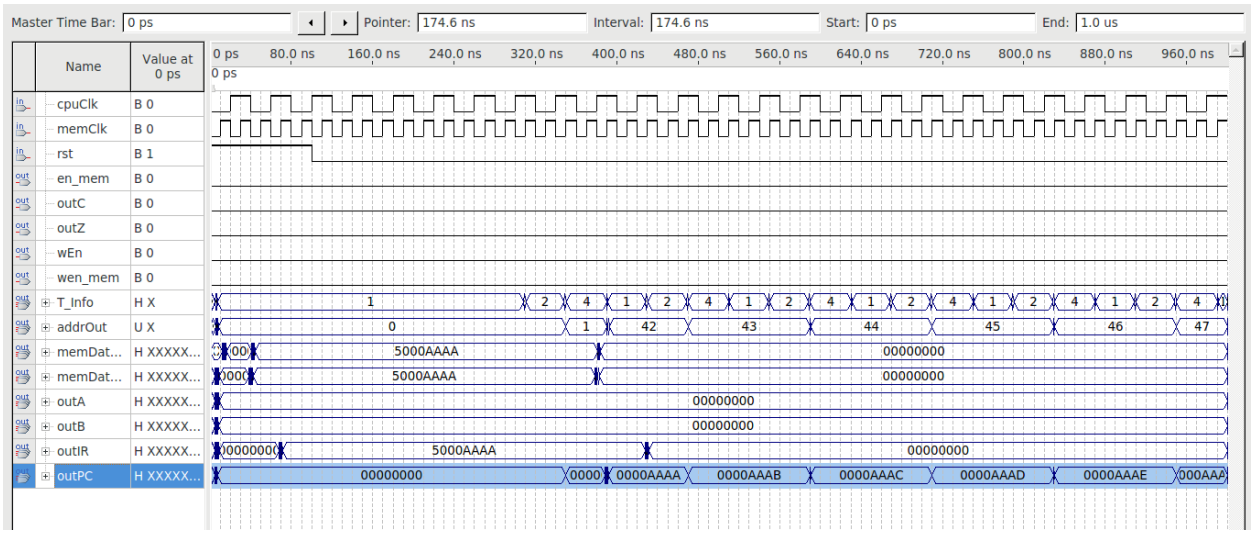


Figure 12: The timing waveform for the JMP instruction.

ORI

The OR Immediate instruction (ANDI) performs bitwise OR between the content already stored in **Register A**, and the given **immediate value** supplied with the **ORI** instruction (to comply with **table 1**).

We plan to test the ORI instruction according to following steps:

1. Load Register A with any desired value
2. Place the **ORI** instruction into the Instruction Memory (this includes the immediate value, which will be bitwise ORed with the content already stored in **Register A.**
3. Read the OR result stored in Register A (according to **table 1**).

We achieved **step 1**, by placing a **LDAI** (I type, where IMM = $0006_{(HEX)}$) instruction at the very first address of the Instruction Memory (see **figure 13**, the content of the row 0, column 0). It is important to note that we are following the exact instruction format for **LDAI** outlined in **table 1**.

Next, we achieved **step 2** by pacing **ORI** instruction in the address that follows **step 1** (see **figure 13**, the content of the row 0, column 1). Please note that we are also following the exact instruction format for **ORI** outlined in **table 1**. Please note that since **$0006_{(HEX)}$ OR $000B_{(HEX)}$ = $000F_{(HEX)}$,** we expect to see the result stored in Register A to be **$0000000F_{(HEX)}$** (after converting from 16 bit to 32 bit). This is exactly seen in the timing simulation waveform presented in **figure 14** (the content of **outA** at around 560 ns).

Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
0	00000006	7D00000B	00000000	00000000	00000000	00000000	00000000	00000000	.....
8	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
16	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
24	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
32	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
40	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
48	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
56	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....

Figure 13: The series of instructions that need to be placed in the Instruction Memory to test ORI instruction.

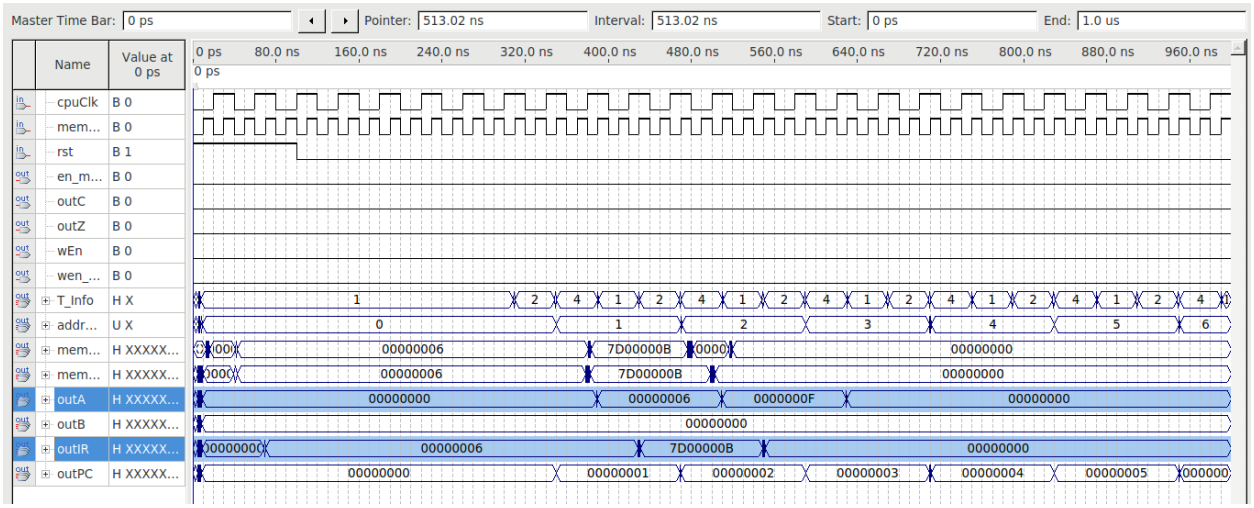


Figure 14: The timing simulation of the ORI instruction

From **figure 14**, we see that we successfully loaded the Register A (**outA**) with the initial value = $00000006_{(HEX)}$ (around 400 ns). Then, we place the second instruction, **ORI** into the Instruction Memory (**outIR**) at around 480 ns with the immediate value = $000B_{(HEX)}$. At around 560 ns 630 ns we see the bitwise OR result between the content of Register A and the immediate value (value of **$0000000F_{(HEX)}$**) in Register A (**outA)**. Thus, our CPU is performing **ORI** operation as we expect.

Reference

1. *COE 608 Lab 6 – The Complete CPU (Overall Project)*. D2L.
2. *COE 608 Lab 6 Tutorial – The Complete CPU (Overall Project)*. D2L.
3. *COE 608 Lab 4b –Data-Path Design*. D2L.
4. *COE 608- CPU Specifications.* D2L.

Appendix

Please note that only the components whose VHDL codes were slightly altered, newly designed have been presented here. Since we already tested and submitted VHDL codes of the other unmodified VHDL components in previous labs, those have not been shown here again. Besides, we are also not showing the already provided system memory files since no change was not made to these VHDL files.

The Reset Circuit

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY reset_circuit IS
    PORT(
        Reset : IN STD_LOGIC;
        Clk : IN STD_LOGIC;
        Enable_PD : OUT STD_LOGIC := '1';
        Clr_PC : OUT STD_LOGIC
    );
END reset_circuit;

ARCHITECTURE Behavior OF reset_circuit IS
    TYPE clkNum IS (clk0, clk1, clk2, clk3);
    SIGNAL present_clk : clkNum;
BEGIN
    process(Clk) begin
        if rising_edge(clk) then
            if Reset = '1' then
                Clr_PC <= '1';
                Enable_PD <= '0';
                present_clk <= clk0;
            elsif present_clk <= clk0 then
                present_clk <= clk1;
            elsif present_clk <= clk1 then
                present_clk <= clk2;
```

```
            elsif present_clk <= clk2 then
                present_clk <= clk3;
            elsif present_clk <= clk3 then
                Clr_PC <= '0';
                Enable_PD <= '1';
            end if;
        end if;
    end process;
END Behavior;
```

Modified Control Unit

```
library ieee;
use ieee.std_logic_1164.ALL;

ENTITY Control_New IS --made change here
    PORT(
        clk, mclk : IN STD_LOGIC;
        enable : IN STD_LOGIC;
        statusC, statusZ : IN STD_LOGIC;
        INST: IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        A_Mux, B_Mux : OUT STD_LOGIC;
        IM_MUX1, REG_Mux : OUT STD_LOGIC;
        IM_MUX2, DATA_Mux : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
        ALU_op : OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
        inc_PC, ld_PC : OUT STD_LOGIC;
        clr_IR: OUT STD_LOGIC;
        ld_IR : OUT STD_LOGIC;
        clr_A,clr_B, clr_C, clr_Z: OUT STD_LOGIC;
        ld_A, ld_B, ld_C, ld_Z : OUT STD_LOGIC;
```

```
        T : OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
        wen, en : OUT STD_LOGIC
    );
END Control_New; --changed here

ARCHITECTURE description OF Control_New IS --Changed here
    TYPE STATETYPE IS (state_0, state_1, state_2);
    SIGNAL present_state: STATETYPE;
    SIGNAL Instruction_sig: STD_LOGIC_VECTOR(3 downto 0);
    SIGNAL Instruction_sig2: STD_LOGIC_VECTOR(7 downto 0);
BEGIN
    Instruction_sig <= INST(31 DOWNTO 28);
    Instruction_sig2 <= INST(31 DOWNTO 24);

----------------------OPERATION DECODER-------------------------
PROCESS(present_state, INST, statusC, statusZ, enable, Instruction_sig, Instruction_sig2)
BEGIN
    if enable = '1' then
        if present_state = state_0 then
            DATA_Mux <= "00"; --Fetch address of next instruction
            clr_IR <= '0';
            ld_IR <= '1';
            ld_PC <= '0';
            inc_PC <= '0';
            clr_A <= '0';
            ld_A <= '0';
            ld_B <= '0';
            clr_B <= '0';
```

```
    clr_C <= '0';
    ld_C <= '0';
    clr_Z <= '0';
    ld_Z <= '0';
    en <= '0';
    wen <= '0';

elsif present_state = state_1 then
    clr_IR <= '0'; --INCREMENT PC COUNTER But how??
    ld_IR <= '0';
    ld_PC <= '1';
    inc_PC <= '1';
    clr_A <= '0';
    ld_A <= '0';
    ld_B <= '0';
    clr_B <= '0';
    clr_C <= '0';
    ld_C <= '0';
    clr_Z <= '0';
    ld_Z <= '0';
    en <= '0';
    wen <= '0';

    if Instruction_sig = "0010" then --STA
        clr_IR <= '0';
        ld_IR <= '0';
        ld_PC <= '1';
        inc_PC <= '1';
```

```
        clr_A <= '0';
        ld_A <= '0';
        ld_B <= '0';
        clr_B <= '0';
        clr_C <= '0';
        ld_C <= '0';
        clr_Z <= '0';
        ld_Z <= '0';
        REG_Mux <= '0';
        DATA_Mux <= "00";
        en <= '1';
        wen <= '1';

    elsif Instruction_sig = "0011" then --STB
        clr_IR <= '0';
        ld_Z <= '0';
        ld_IR <= '0';
        ld_PC <= '1';
        inc_PC <= '1';
        CLR_A <= '0';
        ld_A <= '0';
        ld_B <= '0';
        clr_B <= '0';
        clr_C <= '0';
        ld_C <= '0';
        clr_Z <= '0';
        ld_Z <= '0';
        REG_Mux <= '1';
```

```
    DATA_Mux <= "00";
    en <= '1';
    wen <= '1';

elsif Instruction_sig = "1001" then --LDA
    clr_IR <= '0';
    ld_IR <= '0';
    ld_PC <= '1';
    inc_PC <= '1';
    clr_A <= '0';
    ld_A <= '1';
    ld_B <= '0';
    clr_B <= '0';
    clr_C <= '0';
    ld_C <= '0';
    clr_Z <= '0';
    ld_Z <= '0';
    A_Mux <= '0';
    DATA_Mux <= "01";
    en <= '1';
    wen <= '0';

elsif Instruction_sig = "1010" then --LDB
    clr_IR <= '0';
    ld_IR <= '0';
    ld_PC <= '1';
    inc_PC <= '1';
    clr_A <= '0';
```

```
        ld_A <= '0';
        ld_B <= '1';
        clr_B <= '0';
        clr_C <= '0';
        ld_C <= '0';
        clr_Z <= '0';
        ld_Z <= '0';
        B_Mux <= '0';
        DATA_Mux <= "01";
        en <= '1';
        wen <= '0';
    end if; --END IF FOR LOAD STORE IN STAGE 1

elsif present_state = state_2 then

    if Instruction_sig = "0101" then --JUMP
        clr_IR <= '0';
        ld_IR <= '0';
        ld_PC <= '1';
        inc_PC <= '0';
        clr_A <= '0';
        ld_A <= '0';
        ld_B <= '0';
        clr_B <= '0';
        clr_C <= '0';
        ld_C <= '0';
        clr_Z <= '0';
        ld_Z <= '0';
```

```
elsif Instruction_sig = "0110" then --BEQ
    clr_IR <= '0';
    ld_IR <= '0';
    ld_PC <= '1';
    inc_PC <= '0';
    clr_A <= '0';
    ld_A <= '0';
    ld_B <= '0';
    clr_B <= '0';
    clr_C <= '0';
    ld_C <= '0';
    clr_Z <= '0';
    ld_Z <= '0';

elsif Instruction_sig = "1000" then --BNE
    clr_IR <= '0';
    ld_IR <= '0';
    ld_PC <= '1';
    inc_PC <= '0';
    clr_A <= '0';
    ld_A <= '0';
    ld_B <= '0';
    clr_B <= '0';
    clr_C <= '0';
    ld_C <= '0';
    clr_Z <= '0';
    ld_Z <= '0';
```

```
elsif Instruction_sig = "1001" then --LDA
    clr_IR <= '0';
    ld_IR <= '0';
    ld_PC <= '0';
    inc_PC <= '0';
    clr_A <= '0';
    ld_A <= '1';--just changed here
    ld_B <= '0';
    clr_B <= '0';
    clr_C <= '0';
    ld_C <= '0';
    clr_Z <= '0';
    ld_Z <= '0';
    A_Mux <= '0';
    DATA_Mux <= "01";
    EN <= '1';
    WEN <= '0';

elsif Instruction_sig = "1010" then --LDB
    clr_IR <= '0';
    ld_IR <= '0';
    ld_PC <= '0';
    inc_PC <= '0';
    clr_A <= '0';
    ld_A <= '0';
    ld_B <= '1';--just changed here
    clr_B <= '0';
```

```
    clr_C <= '0';
    ld_C <= '0';
    clr_Z <= '0';
    ld_Z <= '0';
    B_Mux <= '0';
    DATA_Mux <= "01";
    EN <= '1';
    WEN <= '0';

elsif Instruction_sig = "0010" then --STA
    clr_IR <= '0';
    ld_IR <= '0';
    ld_PC <= '0';
    inc_PC <= '0';
    clr_A <= '0';
    ld_A <= '0';
    ld_B <= '0';
    clr_B <= '0';
    clr_C <= '0';
    ld_C <= '0';
    clr_Z <= '0';
    ld_Z <= '0';
    REG_Mux <= '0';
    DATA_Mux <= "00";
    EN <= '1';
    WEN <= '1';

elsif Instruction_sig = "0011" then --STB
```

```
        clr_IR <= '0';
        ld_IR <= '0';
        ld_PC <= '0';
        inc_PC <= '0';
        clr_A <= '0';
        ld_A <= '0';
        ld_B <= '0';
        clr_B <= '0';
        clr_C <= '0';
        ld_C <= '0';
        clr_Z <= '0';
        ld_Z <= '0';
        REG_Mux <= '1';
        DATA_Mux <= "00";
        EN <= '1';
        WEN <= '1';
    elsif Instruction_sig = "0000" then --LDAI
        clr_IR <= '0';
        ld_IR <= '0';
        ld_PC <= '0';
        inc_PC <= '0';
        clr_A <= '0';
        ld_A <= '1';
        ld_B <= '0';
        clr_B <= '0';
        clr_C <= '0';
        ld_C <= '0';
        clr_Z <= '0';
```

```
    ld_Z <= '0';
    A_Mux <= '1';
elsif Instruction_sig = "0001" then --LDBI
    clr_IR <= '0';
    ld_IR <= '0';
    ld_PC <= '0';
    inc_PC <= '0';
    clr_A <= '0';
    ld_A <= '0';
    ld_B <= '1';
    clr_B <= '0';
    clr_C <= '0';
    ld_C <= '0';
    clr_Z <= '0';
    ld_Z <= '0';
    B_Mux <= '1';

elsif Instruction_sig = "0100" then --LUI
    clr_IR <= '0';
    ld_IR <= '0';
    ld_PC <= '0';
    inc_PC <= '0';
    clr_A <= '0';
    ld_A <= '1';
    ld_B <= '0';
    clr_B <= '1';
    clr_C <= '0';
    ld_C <= '0';
```

```
        clr_Z <= '0';
        ld_Z <= '0';
        ALU_op <= "001";
        A_Mux <= '0';
        DATA_Mux <= "10";
        IM_MUX1 <= '1';
    elsif Instruction_sig2 = "01111001" then --ANDI
        clr_IR <= '0';
        ld_IR <= '0';
        ld_PC <= '0';
        inc_PC <= '0';
        clr_A <= '0';
        ld_A <= '1';
        ld_B <= '0';
        clr_B <= '0';
        clr_C <= '0';
        ld_C <= '1';
        clr_Z <= '0';
        ld_Z <= '1';
        ALU_op <= "000";
        A_Mux <= '0';
        DATA_Mux <= "10";
        IM_MUX1 <= '0';
        IM_MUX2 <= "01";
    elsif Instruction_sig2 = "01111110" then --DECA
        clr_IR <= '0';
        ld_IR <= '0';
        ld_PC <= '0';
```

```
        inc_PC <= '0';
        clr_A <= '0';
        ld_A <= '1';
        ld_B <= '0';
        clr_B <= '0';
        clr_C <= '0';
        ld_C <= '1';
        clr_Z <= '0';
        ld_Z <= '1';
        ALU_op <= "110";
        A_Mux <= '0';
        DATA_Mux <= "10";
        IM_MUX1 <= '0';
        IM_MUX2 <= "10";
    elsif Instruction_sig2 = "01110000" then --ADD
        clr_IR <= '0';
        ld_IR <= '0';
        ld_PC <= '0';
        inc_PC <= '0';
        clr_A <= '0';
        ld_A <= '1';
        ld_B <= '0';
        clr_B <= '0';
        clr_C <= '0';
        ld_C <= '1';
        clr_Z <= '0';
        ld_Z <= '1';
        ALU_op <= "010";
```

```
        A_Mux <= '0';
        DATA_Mux <= "10";
        IM_MUX1 <= '0';
        IM_MUX2 <= "00";
    elsif Instruction_sig2 = "01110010" then --SUB
        clr_IR <= '0';
        ld_IR <= '0';
        ld_PC <= '0';
        inc_PC <= '0';
        clr_A <= '0';
        ld_A <= '1';
        ld_B <= '0';
        clr_B <= '0';
        clr_C <= '0';
        ld_C <= '1';
        clr_Z <= '0';
        ld_Z <= '1';
        ALU_op <= "110";
        A_Mux <= '0';
        DATA_Mux <= "10";
        IM_MUX1 <= '0';
        IM_MUX2 <= "00";
    elsif Instruction_sig2 = "01110011" then --INCA
        clr_IR <= '0';
        ld_IR <= '0';
        ld_PC <= '0';
        inc_PC <= '0';
        clr_A <= '0';
```

```
        ld_A <= '1';
        ld_B <= '0';
        clr_B <= '0';
        clr_C <= '0';
        ld_C <= '1';
        clr_Z <= '0';
        ld_Z <= '1';
        ALU_op <= "010";
        A_Mux <= '0';
        DATA_Mux <= "10";
        IM_MUX1 <= '0';
        IM_MUX2 <= "10";
    elsif Instruction_sig2 = "01111011" then --AND
        clr_IR <= '0';
        ld_IR <= '0';
        ld_PC <= '0';
        inc_PC <= '0';
        clr_A <= '0';
        ld_A <= '1';
        ld_B <= '0';
        clr_B <= '0';
        clr_C <= '0';
        ld_C <= '1';
        clr_Z <= '0';
        ld_Z <= '1';
        ALU_op <= "000";
        A_Mux <= '0';
        DATA_Mux <= "10";
```

```
        IM_MUX1 <= '0';
        IM_MUX2 <= "00";
    elsif Instruction_sig2 = "01110001" then --ADDI
        clr_IR <= '0';
        ld_IR <= '0';
        ld_PC <= '0';
        inc_PC <= '0';
        clr_A <= '0';
        ld_A <= '1';
        ld_B <= '0';
        clr_B <= '0';
        clr_C <= '0';
        ld_C <= '1';
        clr_Z <= '0';
        ld_Z <= '1';
        ALU_op <= "010";
        A_Mux <= '0';
        DATA_Mux <= "10";
        IM_MUX1 <= '0';
        IM_MUX2 <= "01";
    elsif Instruction_sig2 = "01111101" then --ORI
        clr_IR <= '0';
        ld_IR <= '0';
        ld_PC <= '0';
        inc_PC <= '0';
        clr_A <= '0';
        ld_A <= '1';
        ld_B <= '0';
```

```
        clr_B <= '0';
        clr_C <= '0';
        ld_C <= '1';
        clr_Z <= '0';
        ld_Z <= '1';
        ALU_op <= "001";
        A_Mux <= '0';
        DATA_Mux <= "10";
        IM_MUX1 <= '0';
        IM_MUX2 <= "01";
    elsif Instruction_sig2 = "01110100" then --ROL
        clr_IR <= '0';
        ld_IR <= '0';
        ld_PC <= '0';
        inc_PC <= '0';
        clr_A <= '0';
        ld_A <= '1';
        ld_B <= '0';
        clr_B <= '0';
        clr_C <= '0';
        ld_C <= '1';
        clr_Z <= '0';
        ld_Z <= '1';
        ALU_op <= "100";
        A_Mux <= '0';
        DATA_Mux <= "10";
        IM_MUX1 <= '0';
    elsif Instruction_sig2 = "01111111" then --ROR
```

```
        clr_IR <= '0';
        ld_IR <= '0';
        ld_PC <= '0';
        inc_PC <= '0';
        clr_A <= '0';
        ld_A <= '1';
        ld_B <= '0';
        clr_B <= '0';
        clr_C <= '0';
        ld_C <= '1';
        clr_Z <= '0';
        ld_Z <= '1';
        ALU_op <= "101";
        A_Mux <= '0';
        DATA_Mux <= "10";
        IM_MUX1 <= '0';
    elsif Instruction_sig2 = "01110101" then --CLR_A
        clr_IR <= '0';
        ld_IR <= '0';
        ld_PC <= '0';
        inc_PC <= '0';
        clr_A <= '1';
        ld_A <= '0';
        ld_B <= '0';
        clr_B <= '0';
        clr_C <= '0';
        ld_C <= '0';
        clr_Z <= '0';
```

```
        ld_Z <= '0';
    elsif Instruction_sig2 = "01110110" then --CLR_B
        clr_IR <= '0';
        ld_IR <= '0';
        ld_PC <= '0';
        inc_PC <= '0';
        clr_A <= '0';
        ld_A <= '0';
        ld_B <= '0';
        clr_B <= '1';
        clr_C <= '0';
        ld_C <= '0';
        clr_Z <= '0';
        ld_Z <= '0';
    elsif Instruction_sig2 = "01110111" then --CLR_C
        clr_IR <= '0';
        ld_IR <= '0';
        ld_PC <= '0';
        inc_PC <= '0';
        clr_A <= '0';
        ld_A <= '0';
        ld_B <= '0';
        clr_B <= '0';
        clr_C <= '1';
        ld_C <= '0';
        clr_Z <= '0';
        ld_Z <= '0';
    elsif Instruction_sig2 = "01111000" then --CLR_z
```

```
        clr_IR <= '0';
        ld_IR <= '0';
        ld_PC <= '0';
        inc_PC <= '0';
        clr_A <= '0';
        ld_A <= '0';
        ld_B <= '0';
        clr_B <= '0';
        clr_C <= '0';
        ld_C <= '0';
        clr_Z <= '1';
        ld_Z <= '0';
    elsif Instruction_sig2 = "01111010" then --TSTZ
        if(statusZ = '1') then
            clr_IR <= '0';--INCREMENT PC COUNTER
            ld_IR <= '0';
            ld_PC <= '1';
            inc_PC <= '1';
            clr_A <= '0';
            ld_A <= '0';
            ld_B <= '0';
            clr_B <= '0';
            clr_C <= '0';
            ld_C <= '0';
            clr_Z <= '0';
            ld_Z <= '0';
        end if;
    elsif Instruction_sig2 = "01111100" then --TSTC
```

```
                if(statusC = '1') then
                    clr_IR <= '0';--INCREMENT PC COUNTER
                    ld_IR <= '0';
                    ld_PC <= '1';
                    inc_PC <= '1';
                    clr_A <= '0';
                    ld_A <= '0';
                    ld_B <= '0';
                    clr_B <= '0';
                    clr_C <= '0';
                    ld_C <= '0';
                    clr_Z <= '0';
                    ld_Z <= '0';
                end if;
            end if; --For stage 2 Ops
        end if;
    end if;--For Enable
END PROCESS;
---------------STATE MACHINE----------------
PROCESS (clk, enable)
begin
    if enable = '1' then
        if rising_edge (clk) then
            if present_state = state_0 then present_state <= state_1;
            elsif present_state = state_1 then present_state <= state_2;
            else present_state <= state_0;
            end if;
        end if;
```

```
    else present_state <= state_0;
    end if;
END PROCESS;

WITH present_state select
T <= "001" when state_0,
     "010" when state_1,
     "100" when state_2,
     "001" when others;
END description;
```

ADD module that increments the PC

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;

entity add is
port(A : in std_logic_vector (31 downto 0);
    B : out std_logic_vector(31 downto 0)
);
end add;

architecture Behaviour of add is
begin
B <= A + 1;--changed here
end Behaviour;
```

CPU Test Simulator

```
library ieee;


use ieee.std_logic_1164.all;


ENTITY CPU_TEST_Sim IS


        PORT(


                cpuClk : in std_logic;


                memClk : in std_logic;


                rst : in std_logic;


                -- Debug data.


                outA, outB : out std_logic_vector(31 downto 0);


                outC, outZ : out std_logic;


                outIR : out std_logic_vector(31 downto 0);


                outPC : out std_logic_vector(31 downto 0);


                -- Processor-Inst Memory Interface.


                addrOut : out std_logic_vector(5 downto 0);
```

```
		wEn : out std_logic;

		memDataOut : out std_logic_vector(31 downto 0);

		memDataIn : out std_logic_vector(31 downto 0);

		-- Processor State

		T_Info : out std_logic_vector(2 downto 0);

		--data Memory Interface

		wen_mem, en_mem : out std_logic);

END CPU_TEST_Sim;



ARCHITECTURE behavior OF CPU_TEST_Sim IS

	COMPONENT system_memory

		PORT(

			address		: IN	STD_LOGIC_VECTOR (5 DOWNTO 0);

			clock	: IN	STD_LOGIC ;
```

```
                data            : IN    STD_LOGIC_VECTOR (31 DOWNTO 0);

                wren            : IN    STD_LOGIC ;

                q                       : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)

);

END COMPONENT;



        COMPONENT cpu1

                PORT(

                        clk             : in std_logic;

                        mem_clk         : in std_logic;

                        rst             : in std_logic;

                        dataIn  : in std_logic_vector(31 downto 0);

                        dataOut         : out std_logic_vector(31 downto 0);

                        addrOut         : out std_logic_vector(31 downto 0);
```

```
                wEn : out std_logic;

                dOutA, dOutB : out std_logic_vector(31 downto 0);

                dOutC, dOutZ : out std_logic;

                dOutIR : out std_logic_vector(31 downto 0);

                dOutPC : out std_logic_vector(31 downto 0);

                outT : out std_logic_vector(2 downto 0);

                wen_mem, en_mem : out std_logic);

END COMPONENT;




signal cpu_to_mem: std_logic_vector(31 downto 0);

signal mem_to_cpu: std_logic_vector(31 downto 0);

signal add_from_cpu: std_logic_vector(31 downto 0);

signal wen_from_cpu: std_logic;
```

```
BEGIN

 -- Component instantiations.

       main_memory : system_memory

               PORT MAP(

                       address => add_from_cpu(5 downto 0),

                       clock => memClk,

                       data => cpu_to_mem,

                       wren => wen_from_cpu,

                       q => mem_to_cpu

                );

       main_processor : cpu1

               PORT MAP(

                clk => cpuClk,

                mem_clk => memClk,
```

```
rst => rst,

dataIn => mem_to_cpu,

dataOut => cpu_to_mem,

addrOut => add_from_cpu,

wEn => wen_from_cpu,

dOutA => outA,

dOutB => outB,

dOutC => outC,

dOutZ => outZ,

dOutIR => outIR,

dOutPC => outPC,

outT => T_Info,

wen_mem => wen_mem,

en_mem => en_mem
```

```
            );




        addrOut <= add_from_cpu(5 downto 0);


        wEn <= wen_from_cpu;


        memDataOut <= mem_to_cpu;


        memDataIn <= cpu_to_mem;


END behavior;
```