Toronto Metropolitan University

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

Course Title:	Computer Organization and Architecture
Course Number:	**COE608**
Semester/Year (e.g.F2016)	**W2025**

Instructor:	**Demetres Kostas**

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

Submission Date:	**Apr 2, 2025**
Due Date:	**Apr 9, 2025**

Student LAST Name	Student FIRST Name	Student Number	Section	Signature*
Qureshi	Maheen	[redacted]	[redacted]	[redacted]

*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: https://www.torontomu.ca/content/dam/senate/policies/pol60.pdf

Part I - CPU Reset Circuitry

1. VHDL Implementation

Reset Circuit

```
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      (
9         Reset:        IN STD_LOGIC;
10        Clk:          IN STD_LOGIC;
11        Enable_PD:    OUT STD_LOGIC := '1';
12        Clr_PC:       OUT STD_LOGIC
13     );
14  END reset_circuit;
15
16  ARCHITECTURE Behavior OF reset_circuit IS
17     TYPE clkNum IS (clk0,clk1,clk2,clk3);
18     SIGNAL present_clk: clkNum;
19  BEGIN
20     process(Clk)begin
21        if rising_edge(Clk) then
22           if Reset = '1' then
23              Clr_PC <= '1';
24              Enable_PD <= '0';
25              present_clk <= clk0;
26           elsif present_clk <= clk0 then
27              present_clk <= clk1;
28           elsif present_clk <= clk1 then
29              present_clk <= clk2;
30           elsif present_clk <= clk2 then
31              present_clk <= clk3;
32           elsif present_clk <= clk3 then
33              Clr_PC <= '0';
34              Enable_PD <= '1';
35           end if;
36        end if;
37     end process;
38  END Behavior;
```

Figure 1.0: VHDL code from reset_circuit.vhd

Figure 1.0 Represents the code used to reset the circuit. With the inputs being Reset and Clk, and the outputs being Enable_PD and Clr_PC. Essentially, when the reset is equal to 1, then the PC is reset.

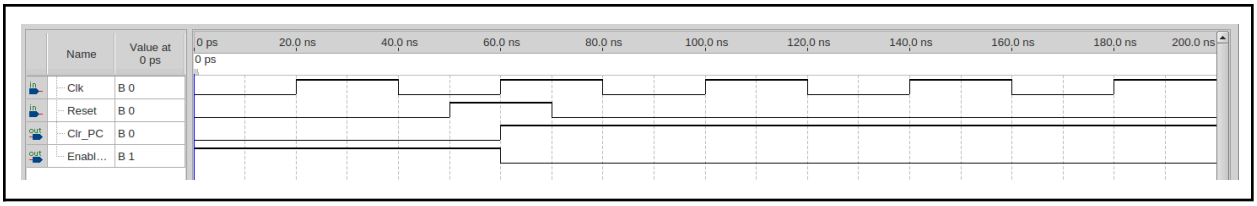


Figure 1.1: Waveform from reset_circuit.vwf

Figure 1.1 This waveform shows how the reset circuit functions. It is visible that until Reset is raised to 1, the Clr_PC is

Upper Zero Value Extender

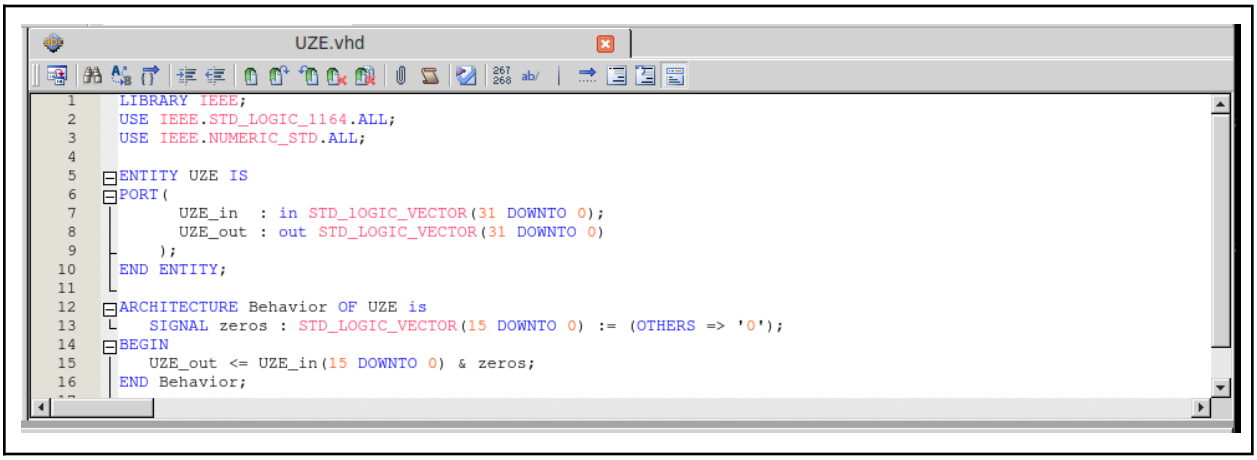


Figure 1.2: VHDL code from UZE.vhd

The UZE.vhd file in Figure 1.2 represents an Upper Zero Value Extender, which takes the lower 16 bits (15 to 0) and stores them as the upper 16 bits (31 to 16), while replacing the rest with zero. This is often used in scenarios that require LSB of the Instruction Register (IR) to be the MSB (31 to 16) of the destination. Within the code, the ports consist of a 32 bit input and output named UZE_IN and UZE_OUT respectively. The architecture is very simple and consists of replacing the upper byte (32 to 16 bits) with the last 16 bits (15 to 0) of the input, and replacing the last 15 to 0 bits of the output UZE_OUT with zeros, which is then output through UZE_OUT.

32-Bit Register

register32.vhd

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

ENTITY register32 IS
PORT(
      d   : IN STD_LOGIC_VECTOR(31 DOWNTO 0); -- input
      ld  : IN STD_LOGIC; -- load/enable
      clr : IN STD_LOGIC; -- async. clear
      clk : IN STD_LOGIC; -- clock
      Q   : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)); -- output
END register32;

ARCHITECTURE Behaviour OF register32 IS
BEGIN
   PROCESS (ld, clr, clk)
   BEGIN
      IF clr = '1' THEN
         Q <= (others => '0');
      ELSIF ((clk'event AND clk = '1') AND (ld  = '1')) THEN
         Q <= d;
      END IF;
   END PROCESS;
END Behaviour;
```

Figure 1.3: VHDL code from register32.vhd

Figure 1.3 depicts the VHDL code for a 32-bit register. The code first establishes the various ports of the register, such as inputs d (data) with 32-bit storage, ld (load/enable), clr (clear), clk (clock) and output Q with 32-bit storage. In terms of functionality of the register, during a rising edge clock (clk = '1') and load (ld = '1'), all 32 bits of data (d) is stored into all 32 bits of the output (Q). The VHDL code also establishes the functionality of the clear input within the register, where its rising edge (clr = '1') clears all 32 bits of output (Q) to zero.

Reducer Unit

RED.vhd

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY RED IS
PORT(
     RED_in   : in STD_LOGIC_VECTOR(31 DOWNTO 0);
     RED_out  : out unsigned(7 DOWNTO 0)
     );
END ENTITY;

ARCHITECTURE Behavior OF RED IS
BEGIN
    RED_out <= unsigned (RED_in(7 DOWNTO 0));
END Behavior;
```

Figure 1.4: VHDL code from RED.vhd

The RED.vhd file in Figure 1.4 represents a Reducer Unit, which is in charge of performing reduction operations on the signal It takes the 32-bit input, and outputs 8-bit. It continues on by essentially extracting the lower 8-bits out of the 32-bit input in the ARCHITECTURE.

4-to-1 Multiplexer

mux4to1.vhd

```
1  LIBRARY IEEE;
2  USE IEEE.STD_LOGIC_1164.ALL;
3
4  ENTITY mux4to1 IS
5     PORT ( s    : in STD_LOGIC_VECTOR(1 DOWNTO 0);
6            X1, X2, X3, X4 : in STD_LOGIC_VECTOR (31 DOWNTO 0);
7            f    : out STD_LOGIC_VECTOR(31 DOWNTO 0));
8  END mux4to1;
9
10 ARCHITECTURE Behavior OF mux4to1 IS
11 BEGIN
12    WITH s SELECT
13       f <= X1 when "00",
14            X2 when "01",
15            X3 when "10",
16            X4 when "11";
17 END Behavior;
```

Figure 1.5: VHDL code from mux4to1.vhd

Figure 1.5 depicts the 4 to 1 Multiplexer included in the data processor. Taking 4 inputs, it assigns certain 2-bit values from 0 to 3 to each of the input values and provides a single output, f.

2-to-1 Multiplexer

mux2to1.vhd

```
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 mux2to1 IS
7     PORT ( s      : IN std_logic;
8            w0, w1 : IN std_logic_vector(31 DOWNTO 0);
9            f      : OUT std_logic_vector(31 DOWNTO 0));
10 END mux2to1;
11
12 ARCHITECTURE Behaviour of mux2to1 IS
13 BEGIN
14    WITH s SELECT
15       f <= w0 when '0',
16            w1 when others;
17 END Behaviour;
```

Figure 1.6: VHDL code from mux2to1.vhd

Figure 1.6 depicts the VHDL code for a 2-to-1 multiplexer that is also used within the program counter below. The role of the multiplexer is to output a specific value of one of the inputs based on a selection. The inputs of the 2-to-1 multiplexer are the selection input (s), input 0 (w0) and input 1 (w1) of the multiplexer and the output function (f). Within the behaviour of the 2-to-1 multiplexer, the function (f) outputs the 32-bit input 0 (w0) when the selector is ‘0’and the 32-bit input 1 (w1) when the selector is ‘1’.

Lower Value Zero Extender

LZE.vhd

```
1  LIBRARY IEEE;
2  USE IEEE.STD_LOGIC_1164.ALL;
3  USE IEEE.NUMERIC_STD.ALL;
4
5  ENTITY LZE IS
6  PORT( LZE_IN   : in STD_LOGIC_VECTOR(31 DOWNTO 0);
7        LZE_OUT  : out STD_LOGIC_VECTOR(31 DOWNTO 0)
8     );
9  END ENTITY;
10
11 ARCHITECTURE Behaviour OF LZE IS
12 SIGNAL zeros: STD_LOGIC_VECTOR(15 DOWNTO 0) := (others => '0');
13 BEGIN
14    LZE_out <= zeros & LZE_in(15 DOWNTO 0);
15 END Behaviour;
```

Figure 1.7: VHDL code from LZE.vhd

The LZE.vhd file in Figure 1.7 represents a Lower Zero Value Extender, which sets the higher bits (31 to 16) to 0 and the lower half (15 to 0) as its original inputs. These are used for immediate operations and loading addresses as witnessed within the LUI (Load Upper Immediate) waveform, as the lower 16 bits of the Instruction Register (IR) are loaded to Register A. It is also used in ADDI (Add Immediate) and ORI (Or Immediate) operations, where the Instruction Register (IR) that holds the lower 16 bits is used as the second operand. Within the code, the ports consist of a 32 bit input and output named LZE_IN and LZE_OUT respectively. The architecture is very simple and consists of replacing the upper byte (32 to 16 bits) with zeros and keeping the last 16 bits (15 to 0) as the input, which is then output through LZE_OUT.

1-Bit Full Adder

fulladd.vhd

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity fulladd is
5  port(
6     Cin, x, y : in std_logic;
7     s, Cout   : out std_logic
8     );
9  end fulladd;
10
11 architecture Behaviour of fulladd is
12 begin
13    s <= x xor y xor Cin;
14    Cout <= (x and y) or (Cin and x) or (Cin and y);
15 end Behaviour;
```

Figure 1.8: VHDL code from fulladd.vhd

Figure 1.8 depicts a 1-Bit Full Adder in VHDL code. The code establishes the functionality of a 1-Bit Full Adder through distinguishing the various ports that compose the component's inputs Cin (Carry in), x and y, as well as outputs s (sum) and Cout (Carry out). The behaviours of outputs s and Cout are defined within the architecture, where s (sum) equates to the xoring of the three inputs x, y and Cin ($x \oplus y \oplus$ Cin). As for Cout, the three inputs are anded and or-ed with one another ((x AND y) OR (Cin AND x) OR (Cin AND y)).

Memory Module Design

data_mem.vhd

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity data_mem is
6  port(
7     clk      : in std_logic;
8     addr     :  in unsigned(7 downto 0);
9     data_in  : in std_logic_vector(31 downto 0);
10    wen      : in std_logic;
11    en       :  in std_logic;
12    data_out : out std_logic_vector(31 downto 0)
13     );
14 end data_mem;
15
16 architecture Behaviour of data_mem is
17    type RAM is array (0 to 255) of std_logic_vector(31 downto 0);
18    signal DATAMEM : RAM;
19 begin
20    process(clk, en, wen)
21    begin
22       if(clk'event and clk='0') then
23          if (en='0') then
24             data_out <= (others => '0');
25          else
26             if (wen = '0') then
27                data_out <= DATAMEM(to_integer(addr));
28             end if;
29             if (wen = '1') then
30                DATAMEM(to_integer(addr)) <= data_in;
31                data_out <= (others => '0');
32             end if;
33          end if;
34       end if;
35    end process;
36 end Behaviour;
```

Figure 1.9: VHDL code from data_mem.vhd

The data_mem.vhd file in Figure 1.9 represents a memory module designed to read and write data within a clock cycle. The various ports within the memory unit assist in carrying out the functionality of the unit. This includes inputs clk (clock), addr (8-bit address input), data_in (32-bit data input), wen (32-bit write enable), en (enable), and output data_out (32-bit data ouput). Within the behaviour of the object, a RAM type array can store the assigned value of data_in when en = 1 and wen = 1 at the assigned value of addr during a falling edge of the input clock. As for reading data, it requires en = 1 and wen = 0, which will return the data stored at the assigned value of addr to data_out during a falling-edge of a clock cycle. When en = 0, regardless of any other inputs or clock edge, there will be no functionality.

Arithmetic Logic Unit (ALU)

alu.vhd

```
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  use ieee.numeric_std.all;
6
7  entity alu is
8  port(
9     a  :  in std_logic_vector(31 downto 0);
10    b  :  in std_logic_vector(31 downto 0);
11    op :  in std_logic_vector(2 downto 0);
12    result : out std_logic_vector(31 downto 0);
13    zero : out std_logic;
14    cout : out std_logic
15    );
16    end alu;
17
18
19    architecture Behavior of alu is
20       component adder32
21          port(
22             Cin   : in std_logic;
23             X,Y   : in std_logic_vector(31 downto 0);
24             S     : out std_logic_vector(31 downto 0);
25             Cout  : out std_logic
26             );
27       end component;
28
29
30       signal result_s: std_logic_vector(31 downto 0):= (others => '0');
31       signal result_add: std_logic_vector(31 downto 0):= (others => '0');
32       signal result_sub: std_logic_vector(31 downto 0):= (others => '0');
33       signal cout_s : std_logic := '0';
34       signal cout_add : std_logic := '0';
35       signal cout_sub : std_logic := '0';
36       signal zero_s : std_logic;
37
38    begin
39       add0 : adder32 port map (op(2),a,b,result_add, cout_add);
40       sub0 : adder32 port map (op(2), a, not b, result_sub, cout_sub);
41
42     process (a, b, op)
43     begin
44        case (op) is
45           when "000" =>
46              result_s <= a and b;
47              cout_s <= '0';
48           when "001" =>
49              result_s <= a or b;
50              cout_s <= '0';
51           when "010" =>
52              result_s <= result_add;
53              cout_s <= cout_add;
54           when "011" =>
55              result_s <= b;
56              cout_s <= '0';
57           when "110" =>
58              result_s <= result_sub;
59              cout_s <= cout_sub;
60           when "100" =>
61              result_s <= a(30 downto 0) & '0';
62              cout_s <= a(31);
63           when "101" =>
64              result_s <= '0' & a(31 downto 1);
65              cout_s <= '0';
66           when others =>
67              result_s <= a;
68              cout_s <= '0';
69        end case;
70
71        case (result_s) is
72           when (others => '0') =>
73              zero_s <= '1';
74           when others =>
75              zero_s <= '0';
76        end case;
77     end process;
78
79     result <= result_s;
80     cout <= cout_s;
81     zero <= zero_s;
82  end Behavior;
```

Figure 1.10: VHDL code from alu.vhd

Figure 3.4 represents the code for the 32-bit ALU. This consists of three main inputs (a, b and op) and three outputs (result, zero and cout). It is necessary to incorporate the previously created 32-bit adder so as to complete calculations for this component. All signals are output using ‘signal result’, and ‘port map’ is utilised to connect all of these input ports to the output ports. It is necessary to be able to handle cases for all possible binary values for op, which is represented in process(a,b,op). Different values result in cout_s giving the correct respective outputs for the ALU. As there are two ports for zero_s, the cases of it being equivalent to ‘0’ or ‘1’ are handled, and the output for cout and result conclude the code.

32-Bit Adder

adder32.vhd

```
library ieee;
use ieee.std_logic_1164.all;

entity adder32 is
port(
    Cin      : in std_logic;
    X,Y      : in std_logic_vector(31 downto 0);
    S     : out std_logic_vector(31 downto 0);
    Cout  : out std_logic
    );
end adder32;

architecture Behaviour of adder32 is
      component adder16
      port(
      Cin    : in std_logic;
      X,Y    : in std_logic_vector(15 downto 0);
      S      : out std_logic_vector(15 downto 0);
      Cout   : out std_logic
      );
      end component;

      signal C : std_logic;
begin
      stage0: adder16 port map (Cin, X(15 downto 0), Y(15 downto 0), S(15 downto 0), C);
      stage1: adder16 port map (C, X(31 downto 16), Y(31 downto 16), S(31 downto 16), Cout);
end Behaviour;
```

Figure 1.11: VHDL code from adder32.vhd

Figure 1.11 depicts a 32-Bit Adder/Subtractor in VHDL code. The code establishes the functionality of a 32-Bit Adder/Subtractor. The code distinguishes the various ports that compose the component's inputs Cin (Carry in), X[31..0] and Y[31..0], as well as outputs S[31..0] (Sum) and Cout (Carry out). The behaviour of the 32-Bit Adder/Subtractor uses the 16-Bit Full Adder in 2 port maps, with specific inputs to represent the 32 bits as established in the code

16-Bit Adder

adder16.vhd

```
library ieee;
use ieee.std_logic_1164.all;

entity adder16 is
port(
    Cin      : in std_logic;
    X,Y      : in std_logic_vector(15 downto 0);
    S     : out std_logic_vector(15 downto 0);
    Cout  : out std_logic
    );
end adder16;

architecture Behaviour of adder16 is
    component adder4
    port(
    Cin      : in std_logic;
    X, Y     : in std_logic_vector(3 downto 0);
    S     : out std_logic_vector(3 downto 0);
    Cout  : out std_logic
    );

    end component;

    signal C : std_logic_vector(1 to 3);
begin
    stage0: adder4 port map (Cin, X(3 downto 0), Y(3 downto 0), S(3 downto 0), C(1)
    );
    stage1: adder4 port map (C(1), X(7 downto 4), Y(7 downto 4), S(7 downto 4), C(2)
    );
    stage2: adder4 port map (C(2), X(11 downto 8), Y(11 downto 8), S(11 downto 8), C(3)
    );
    stage3: adder4 port map (C(3), X(15 downto 12), Y(15 downto 12), S(15 downto 12), Cout);
end Behaviour;
```

Figure 1.12: VHDL code from adder16.vhd

Figure 1.12 depicts a 16-Bit Adder in VHDL code. The code establishes the functionality of a 16-Bit Adder. The code distinguishes the various ports that compose the component's inputs Cin (Carry in), X[15..0] and Y[15..0], as well as outputs S[15..0] (Sum) and Cout (Carry out). The behaviour of the 16-Bit Adder uses the 4-Bit Full Adder in 4 port maps, with specific inputs to represent the 16 bits as established in the code.

4-Bit Adder

adder4.vhd

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity adder4 is
5  port(
6      Cin      : in std_logic;
7      X, Y     : in std_logic_vector(3 downto 0);
8      S        : out std_logic_vector(3 downto 0);
9      Cout     : out std_logic
10     );
11 end adder4;
12
13 architecture Behaviour of adder4 is
14     component fulladd
15     port(
16     Cin, x, y : in std_logic;
17     s, Cout   : out std_logic
18     );
19     end component;
20
21     signal C : std_logic_vector (1 to 3);
22 begin
23     stage0: fulladd port map (Cin, X(0), Y(0), S(0), C(1));
24     stage1: fulladd port map (C(1), X(1), Y(1), S(1), C(2));
25     stage2: fulladd port map (C(2), X(2), Y(2), S(2), C(3));
26     stage3: fulladd port map (C(3), X(3), Y(3), S(3), Cout);
27 end Behaviour;
```

Figure 1.13: VHDL code from adder4.vhd

Figure 1.13 depicts a 4-Bit Adder in VHDL code. The code establishes the functionality of a 4-Bit Adder through distinguishing the various ports that compose the component's inputs Cin (Carry in), X[3..0] and Y[3..0], as well as outputs S[3..0] (Sum) and Cout (Carry out). The behaviour of the 4-Bit Adder uses the Full Adder created in Figure 3.0 in 4 port maps, with specific inputs to represent the 4 bits as established in the code.

Adder

add.vhd

```
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 add IS
7  PORT (A :   IN std_logic_vector(31 DOWNTO 0);
8        B :   OUT std_logic_vector(31 DOWNTO 0)
9        );
10 END add;
11
12 ARCHITECTURE Behaviour OF add IS
13 BEGIN
14 B <= A + 4;
15 END Behaviour;
```

Figure 1.14: VHDL code from add.vhd

Figure 1.14 depicts the VHDL code for an adder that is used for the Program Counter below. The role of the adder is to increment any 32 bit input (A) by 4, and then output it to B. Within the VHDL code add, it establishes the input (A) and output (B), and then its behaviour of setting B to the increment of A by 4.

Program Counter

pc.vhd

```
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 pc IS
7  PORT(
8        clr :IN STD_LOGIC;
9        clk :IN STD_LOGIC;
10       ld :IN STD_LOGIC;
11       inc :IN STD_LOGIC;
12       d : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
13       q : OUT STD_LOGIC_VECTOR(31 DOWNTO 0));
14 END pc;
15
16 ARCHITECTURE Behaviour OF pc IS
17    COMPONENT add
18       PORT (
19          A : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
20          B  :  OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
21       );
22    END COMPONENT;
23
24    COMPONENT mux2to1
25       PORT (
26          s : IN STD_LOGIC;
27          w0,w1 : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
28          f : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
29       );
30    END COMPONENT;
31
32    COMPONENT register32
33       PORT(
34          d : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
35          ld : IN STD_LOGIC;
36          clr:  IN STD_LOGIC;
37          clk:  IN STD_LOGIC;
38          Q  :  OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
39       );
40    END COMPONENT;
41
42    SIGNAL add_out : STD_LOGIC_VECTOR(31 DOWNTO 0);
43    SIGNAL mux_out : STD_LOGIC_VECTOR(31 DOWNTO 0);
44    SIGNAL q_out : STD_LOGIC_VECTOR(31 DOWNTO 0);
45 BEGIN
46    add0: add PORT MAP(q_out, add_out);
47    mux0: mux2to1 PORT MAP(inc, d, add_out, mux_out);
48    reg0: register32 PORT MAP (mux_out, ld, clr, clk, q_out);
49    q<= q_out;
50 END Behaviour;
```

Figure 1.15: VHDL code from pc.vhd

Figure 1.15 represents the VHDL code written for a program. Essentially, the program counter holds addresses of instructions, helping keep track of program execution. Executions occur incrementally. The code begins with initializing the necessary input signals including the clock, clear, load, incrementor, and data value. There is a single output representing the address, labeled as (q). By using the aforementioned add, mux2to1 and register32 components, the program counter successfully is able to execute its role. The bottom part of the code consists of the 32-bit counter (represented by 31 DOWNTO 0) to output the add, mux and q outputs. It concludes with using PORT MAP to connect the input and output signals and complete the program counter.

Data Path

data_path.vhd

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY data_path IS
   PORT(
      -- Clock Signal
      Clk, mClk   : in std_logic;

      -- Memory Signals
      WEN, EN     : in std_logic;

      -- Register Control Signals (CLR and LD)
      Clr_A, Ld_A : in std_logic;
      Clr_B, Ld_B : in std_logic;
      Clr_C, Ld_C : in std_logic;
      Clr_Z, Ld_Z : in std_logic;
      ClrPC, Ld_PC : in std_logic;
      ClrIR, Ld_IR : in std_logic;

      -- Register Outputs
      Out_A    : out std_logic_vector(31 downto 0);
      Out_B    : out std_logic_vector(31 downto 0);
      Out_C    : out std_logic;
      Out_Z    : out std_logic;
      Out_PC   : out std_logic_vector(31 downto 0);
      Out_IR   : out std_logic_vector(31 downto 0);

      -- Special Inputs to PC
      Inc_PC   : in std_logic;

      -- Address and Data Bus Signals for debugging
      ADDR_OUT   : out std_logic_vector(31 downto 0);
      DATA_IN    : in std_logic_vector(31 downto 0);
      DATA_BUS,
      MEM_OUT,
      MEM_IN     : out std_logic_vector(31 downto 0);
      MEM_ADDR   : out unsigned(7 downto 0);

      -- Various MUX controls
      DATA_MUX   : in std_logic_vector(1 downto 0);
      REG_MUX    : in std_logic;
      A_MUX,
      B_MUX      : in std_logic;
      IM_MUX1    : in std_logic;
      IM_MUX2    : in std_logic_vector(1 downto 0);

      -- ALU Operations
      ALU_Op     : in std_logic_vector(2 downto 0)
      );
end entity;

architecture Behavior of Data_Path is
   -- Component Instantiations
   -- Data Memory Module
   component data_mem is
      port(
         clk     : in std_logic;
         addr    : in unsigned(7 downto 0);
         data_in: in std_logic_vector(31 downto 0);
         wen     : in std_logic;
         en      : in std_logic;
         data_out: out std_logic_vector(31 downto 0)
```

```
65          );
66      end component;
67
68      component register32 is
69          port(
70              d    : in std_logic_vector(31 downto 0);
71              ld   : in std_logic;
72              clr  : in std_logic;
73              clk  : in std_logic;
74              Q    : out std_logic_vector(31 downto 0)
75              );
76      end component;
77
78      -- Program Counter
79      component pc is
80          port(
81               clr : in std_logic;
82               clk : in std_logic;
83               ld  : in std_logic;
84               inc : in std_logic;
85               d   : in std_logic_vector(31 downto 0);
86               q   : out std_logic_vector(31 downto 0)
87               );
88      end component;
89
90      -- LZE
91      component LZE is
92          port( LZE_in   : in std_logic_vector(31 downto 0);
93                LZE_out  : out std_logic_vector(31 downto 0)
94              );
95      end component;
96
97      -- UZE
98      component UZE is
99          port(
100             UZE_in  : in std_logic_vector(31 downto 0);
101             UZE_out : out std_logic_vector(31 downto 0)
102             );
103     end component;
104
105     -- RED
106     component RED is
107         port(
108             RED_in : in std_logic_vector(31 downto 0);
109             RED_out : out unsigned(7 downto 0)
110             );
111     end component;
112
113     -- Mux2to1
114     component mux2to1 is
115         port(
116              s : in std_logic;
117              w0, w1 : in std_logic_vector(31 downto 0);
118              f : out std_logic_vector(31 downto 0)
119             );
120     end component;
121
122     -- Mux4to1
123     component mux4to1 is
124         port(
125              s : in std_logic_vector(1 downto 0);
126              X1, X2, X3, X4 : in std_logic_vector(31 downto 0);
127              f : out std_logic_vector(31 downto 0)
```

```
127             f : out std_logic_vector(31 downto 0)
128           );
129     end component;
130
131     -- ALU
132     component alu is
133        port(
134           a      : in std_logic_vector(31 downto 0);
135           b      : in std_logic_vector(31 downto 0);
136           op     : in std_logic_vector(2 downto 0);
137           result : out std_Logic_vector(31 downto 0);
138           zero   : out std_logic;
139           cout   : out std_logic
140           );
141     end component;
142
143     -- Signal instantiations
144     signal IR_OUT            : std_logic_vector(31 downto 0);
145     signal data_bus_s        : std_logic_vector(31 downto 0);
146     signal LZE_out_PC        : std_logic_vector(31 downto 0);
147     signal LZE_out_A_Mux     : std_logic_vector(31 downto 0);
148     signal LZE_out_B_Mux     : std_logic_vector(31 downto 0);
149     signal RED_out_Data_Mem  : unsigned(7 downto 0);
150     signal A_Mux_out         : std_logic_vector(31 downto 0);
151     signal B_Mux_out         : std_logic_vector(31 downto 0);
152     signal reg_A_out         : std_logic_vector(31 downto 0);
153     signal reg_B_out         : std_logic_vector(31 downto 0);
154     signal reg_Mux_out       : std_logic_vector(31 downto 0);
155     signal data_mem_out      : std_logic_vector(31 downto 0);
156     signal UZE_IM_MUX1_out   : std_logic_vector(31 downto 0);
157     signal IM_MUX1_out       : std_logic_vector(31 downto 0);
158     signal LZE_IM_MUX2_out   : std_logic_vector(31 downto 0);
159     signal IM_MUX2_out       : std_logic_vector(31 downto 0);
160     signal ALU_out           : std_logic_vector(31 downto 0);
161     signal zero_flag         : std_logic;
162     signal carry_flag        : std_logic;
163     signal temp              : std_logic_vector(30 downto 0) := (others => '0');
164     signal out_pc_sig        : std_logic_vector(31 downto 0);
165
166 begin
167     IR:           register32 port map(
168                   data_bus_s,
169                   Ld_IR,
170                   ClrIR,
171                   Clk,
172                   IR_OUT
173                   );
174
175     LZE_PC:       LZE port map(
176                   IR_OUT,
177                   LZE_out_PC
178                   );
179
180     PC0:          PC port map(
181                   CLRPC,
182                   Clk,
183                   ld_PC,
184                   INC_PC,
185                   LZE_out_PC,
186                   -- ADDR_OUT
187                   out_Pc_sig
188                    );
189
190     LZE_A_Mux: LZE port map(
191                   IR_OUT,
192                   LZE_out_A_Mux
193                   );
194
195     A_Mux0:       mux2to1 port map(
196                   A_MUX,
197                   data_bus_s,
198                   LZE_out_A_Mux,
199                   A_Mux_out
200                   );
201
202     Reg_A:        register32 port map(
203                   A_Mux_out,
204                   Ld_A,
205                   Clr_A,
206                   Clk,
207                   reg_A_out
208                   );
```

```
209
210     LZE_B_Mux:      LZE port map(
211                     IR_OUT,
212                     LZE_out_B_Mux
213                     );
214
215     B_Mux0:         mux2to1 port map(
216                     B_MUX,
217                     data_bus_s,
218                     LZE_out_B_Mux,
219                     B_Mux_out
220                     );
221
222     Reg_B:          register32 port map(
223                     B_Mux_out,
224                     Ld_B,
225                     Clr_B,
226                     Clk,
227                     reg_B_out
228                     );
229
230     Reg_Mux0:       mux2to1 port map(
231                     REG_MUX,
232                     Reg_A_out,
233                     Reg_B_out,
234                     Reg_Mux_out
235                     );
236
237     RED_Data_Mem:   RED port map(
238                     IR_OUT,
239                     RED_out_data_mem
240                     );
241
242     Data_Mem0:      data_mem port map(
243                     mClk,
244                     RED_out_data_mem,
245                     Reg_Mux_out,
246                     WEN,
247                     EN,
248                     data_mem_out
249                     );
250
251     UZE_IM_MUX1:    UZE port map(
252                     IR_OUT,
253                     UZE_IM_MUX1_out
254                     );
255
256     IM_MUX1a:       mux2to1 port map(
257                     IM_MUX1,
258                     reg_A_out,
259                     UZE_IM_MUX1_out,
260                     IM_MUX1_out
261                     );
262
263     LZE_IM_MUX2:    LZE port map(
264                     IR_OUT,
265                     LZE_IM_MUX2_out
266                     );
267
268     IM_MUX2a:       mux4to1 port map(
269                     IM_MUX2,
270                     reg_B_out,
271                     LZE_IM_MUX2_out,
```

```
272                 (temp &'1'),
273                 (others => '0'),
274                 IM_MUX2_out
275                 );
276
277     ALU0:       ALU port map(
278                 IM_MUX1_out,
279                 IM_MUX2_out,
280                 ALU_OP,
281                 ALU_out,
282                 zero_flag ,
283                 carry_flag
284                 );
285
286     DATA_MUX0:  mux4to1 port map(
287                 DATA_MUX,
288                 DATA_IN,
289                 data_mem_out,
290                 ALU_out,
291                 (others => '0'),
292                 data_bus_s
293                 );
294
295     DATA_BUS <= data_bus_s;
296     OUT_A <= reg_A_out;
297     OUT_B <= reg_B_out;
298     OUT_IR <= IR_OUT;
299     ADDR_OUT <= out_pc_sig;
300     OUT_PC <= out_pc_sig;
301
302     MEM_ADDR <= RED_out_Data_Mem;
303     MEM_IN <= Reg_Mux_out;
304     MEM_OUT <= data_mem_out;
305
306 end Behavior;
```

Figure 1.16: VHDL code from data_path.vhd

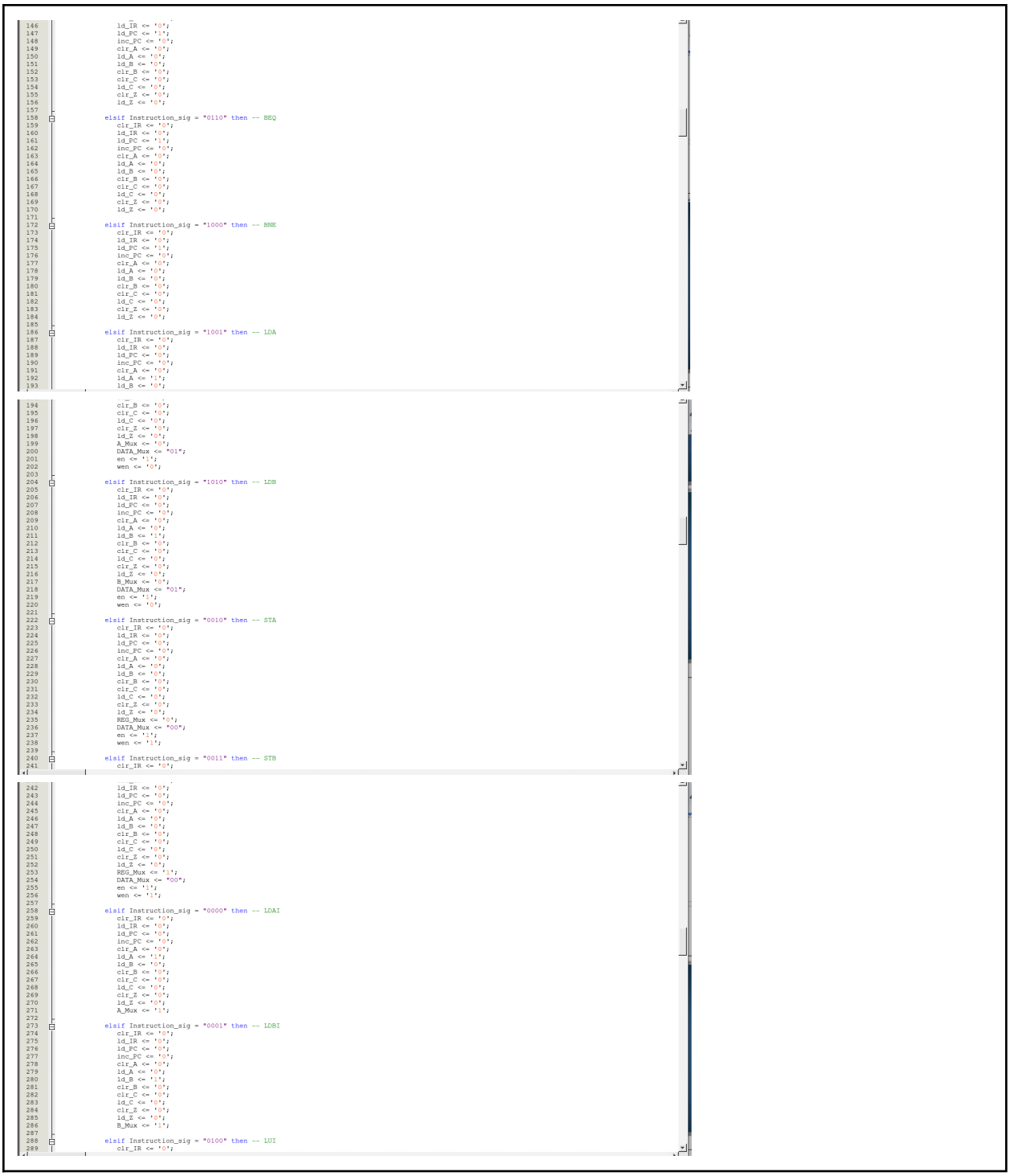
Figure 1.16 Represents all of the aforementioned scripts. It begins with initializing the clock signal, memory signals and register control signals, all inputs for the system. Register Output A follows, with inputs to the PC, Address, Mux controls and ALU operations. Past this, The different parts of the architecture are implemented. This includes the Data Memory module, PC, LZE, UZE, RED, Mux 2-to-1, Mux 4-to-1, and ALU. Signals are installed, and PORT MAP is used to connect and implement all of the signals and calculations, resulting in us getting the final required results.

Control Unit

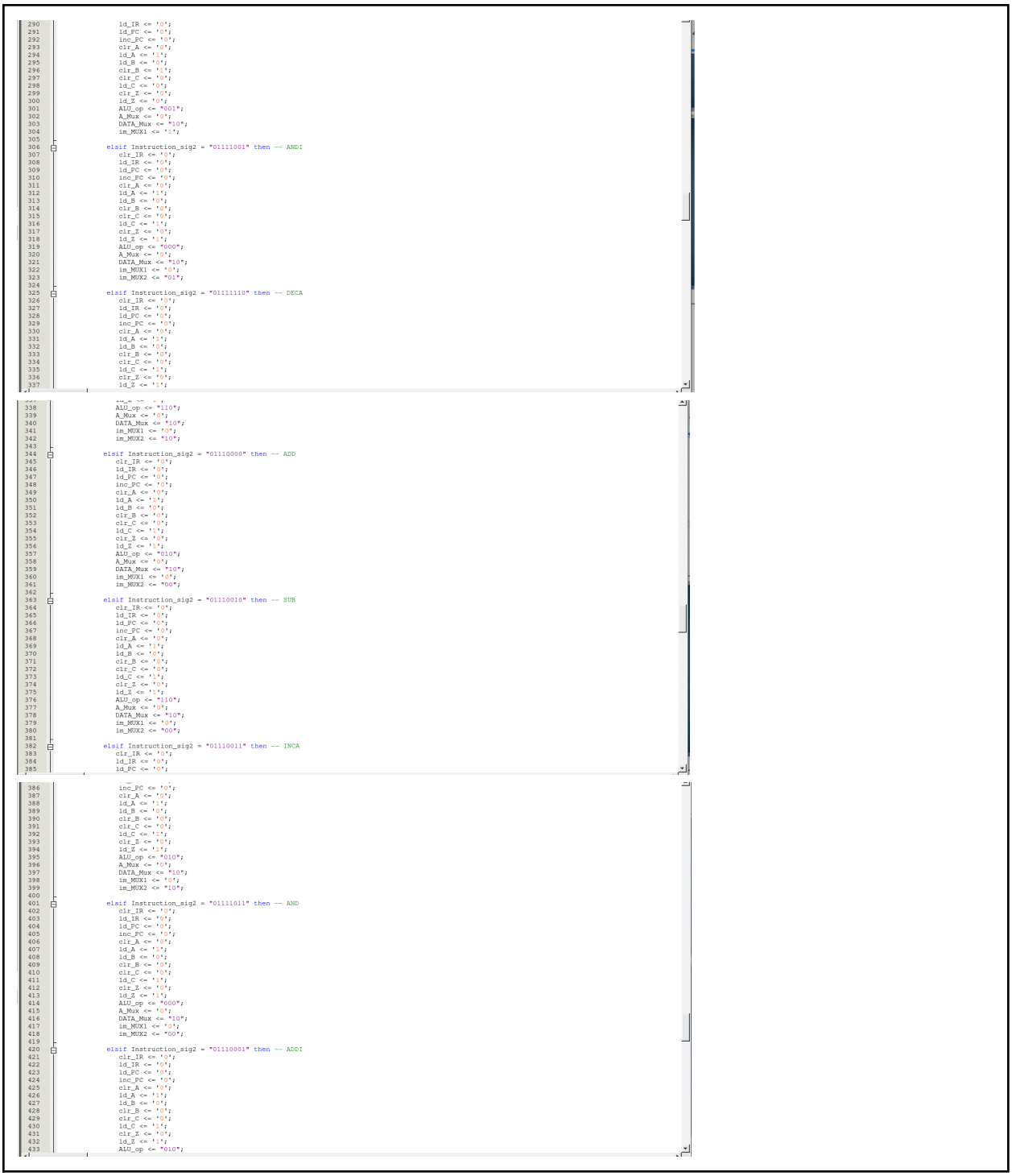
Quartus II 32-bit - /home/student1/r225shar/Desktop/COE608/lab5/lab5 - lab5

control.vhd

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

ENTITY control IS
   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;

ARCHITECTURE description of control IS
   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';
            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
            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_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';
```

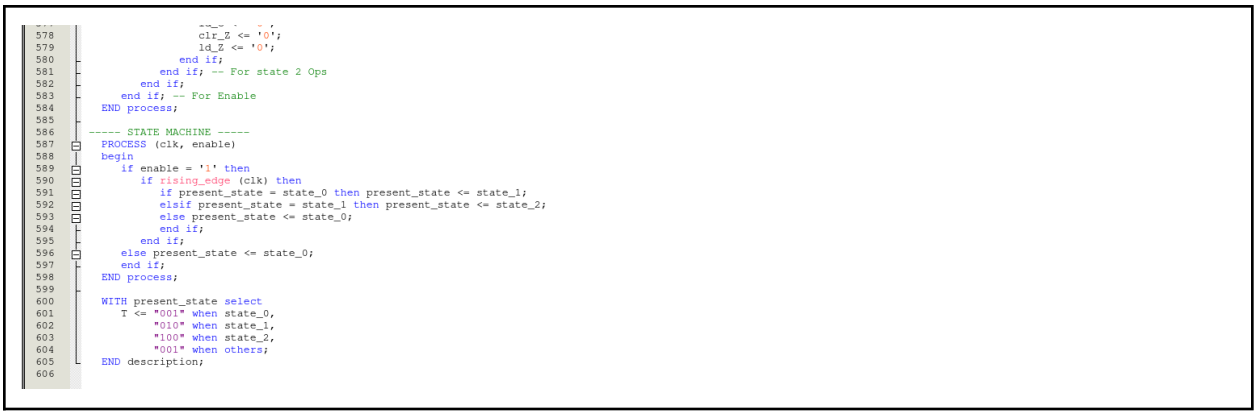


```
            ld_IR <= '0';
            ld_PC <= '1';
            inc_PC <= '0';
            clr_A <= '0';
            ld_A <= '0';
            ld_B <= '0';
            clr_B <= '0';
            clr_C <= '0';
            ld_C <= '0';
            clr_Z <= '0';
            ld_Z <= '0';

        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';
            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';
            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 = "01111011" 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';
```



```
578                 clr_Z <= '0';
579                 ld_Z <= '0';
580             end if;
581         end if; -- For state 2 Ops
582       end if;
583     end if; -- For Enable
584   END process;
585
586 ----- STATE MACHINE -----
587   PROCESS (clk, enable)
588   begin
589     if enable = '1' then
590       if rising_edge (clk) then
591         if present_state = state_0 then present_state <= state_1;
592         elsif present_state = state_1 then present_state <= state_2;
593         else present_state <= state_0;
594         end if;
595       end if;
596     else present_state <= state_0;
597     end if;
598   END process;
599
600   WITH present_state select
601     T <= "001" when state_0,
602          "010" when state_1,
603          "100" when state_2,
604          "001" when others;
605 END description;
606
```

Figure 1.17: VHDL code from control.vhd

The Control.vhd file in Figure 1.17 represents the main functionality of the system for the control unit design. Following Lab 4B, the control VHDL file combines the functionality of a state generator, memory signal generator and decoder with the various inputs to create various outputs. The state generator makes use of the enable (en) and clock (clk) to produce the state, which is used in the decoder. The memory signal generator makes use of the clock (clk), memory clock (mclk) and instruction (INST) inputs, to produce outputs wen (write enable) and enable (en). Finally the decoder block makes use of the inputs instruction (inst), status carry (status_c) and status zero (status_z) to produce the various outputs from lab 4b such as load instruction register (LD_IR), clear instruction register (CLR_IR), load register a (LDA), clear register a (CLR_A), … , Multiplexers 1 and 2 and Register Multiplexer (Reg_MUX).

Control Processor Unit 1

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

ENTITY cpu1 IS
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);
dOutA : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
dOutB : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
dOutC : OUT STD_LOGIC;
dOutZ : OUT STD_LOGIC;
dOutIR : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
dOutPC : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
wEn : OUT STD_LOGIC;
outT : OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
wen_mem : OUT STD_LOGIC;
en_mem : OUT STD_LOGIC
);
END cpu1;

ARCHITECTURE description OF cpu1 IS
COMPONENT Data_Path IS
PORT (
Clk, mClk : IN STD_LOGIC; -- clock Signal
--Memory Signals
WEN, EN : IN STD_LOGIC;
-- Register Control Signals (CLR and LD).
Clr_A , Ld_A : IN STD_LOGIC;
Clr_B , Ld_B : IN STD_LOGIC;
Clr_C , Ld_C : IN STD_LOGIC;
Clr_Z , Ld_Z : IN STD_LOGIC;
ClrPC , Ld_PC : IN STD_LOGIC;
ClrIR , Ld_IR : IN STD_LOGIC;
```

```
Out_A : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
Out_B : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
Out_C : OUT STD_LOGIC;
Out_Z : OUT STD_LOGIC;
Out_PC : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
Out_IR : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
-- Special inputs to PC.
Inc_PC : IN STD_LOGIC;
-- Address and Data Bus signals for debugging.
ADDR_OUT : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
DATA_IN : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
DATA_BUS, MEM_OUT, MEM_IN : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
MEM_ADDR : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
-- Various MUX controls.
DATA_Mux : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
REG_Mux : IN STD_LOGIC;
A_MUX, B_MUX : IN STD_LOGIC;
IM_MUX1 : IN STD_LOGIC;
IM_MUX2 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
-- ALU Operations.
ALU_Op : IN STD_LOGIC_VECTOR(2 DOWNTO 0)
);
END COMPONENT;

COMPONENT Control_NEW IS
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;
```

```
    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 COMPONENT;

    COMPONENT reset_circuit IS
    PORT (
    Reset : IN STD_LOGIC;
    Clk : IN STD_LOGIC;
    Enable_PD : OUT STD_LOGIC;
    Clr_PC : OUT STD_LOGIC
    );
    END COMPONENT;

    SIGNAL dp_mux1, dp_clrA, dp_ldA, dp_clrB, dp_ldB, dp_clrC, dp_ldC, dp_clrZ,
    dp_ldZ, memWEN, memEN, dp_muxA, dp_muxB : STD_LOGIC;
    SIGNAL mux_data, reg, enpd, irlc, irld, pinc, pclr, pcld, out0, out1, out7, out6 : STD_LOGIC;
    SIGNAL outIR : STD_LOGIC_VECTOR(31 DOWNTO 0);
    SIGNAL alu : STD_LOGIC_VECTOR(2 DOWNTO 0);
    SIGNAL dp_mux2, dp_muxData : STD_LOGIC_VECTOR(1 DOWNTO 0);

    BEGIN
    dat : Data_Path
    PORT MAP(
    Clk => clk,
    mClk => mem_clk,
    WEN => memWEN,
    EN => memEN,
    Clr_A => dp_clrA,
    Ld_A => dp_ldA, Clr_B => dp_clrB, Ld_B => dp_ldB, Clr_C => dp_clrC, Ld_C
    => dp_ldC, Clr_Z => dp_clrZ, Ld_Z => dp_ldZ,
    ClrPC => pclr, Ld_PC => pcld, ClrIR => irlc, Ld_IR => irld,
    Out_A => dOutA, Out_B => dOutB, Out_C => out0, Out_Z => out1, Out_PC =>
    dOutPC, Out_IR => outIR, Inc_PC => pinc,
    ADDR_OUT => addrOut, DATA_IN => dataIn, DATA_BUS => dataOut, DATA_Mux
    => dp_muxData, REG_Mux => reg, A_MUX => dp_muxA, B_MUX => dp_muxB, IM_MUX1 =>
    dp_mux1, IM_MUX2 => dp_mux2,
    ALU_Op => alu
    );

control_unit : Control_NEW
PORT MAP(
clk => clk, mclk => mem_clk, enable => enpd, statusC => out0, statusZ =>
out1, INST => outIR,

A_Mux => dp_muxA, B_Mux => dp_muxB, IM_MUX1 => dp_mux1, REG_Mux => reg,
IM_MUX2 => dp_mux2, DATA_Mux => dp_muxData,
ALU_op => alu, inc_PC => pinc, ld_PC => pcld, clr_IR => irlc, ld_IR =>
irld,
clr_A => dp_clrA, clr_B => dp_clrB, clr_C => dp_clrC, clr_Z => dp_clrZ,
ld_A => dp_ldA, ld_B => dp_ldB, ld_C => dp_ldC, ld_Z => dp_ldZ,
T => outT, wen => memWEN, en => memEN
);

reset : reset_circuit
PORT MAP(
Reset => rst,
Clk => clk,
Enable_PD => enpd,
Clr_PC => pclr
);
dOutC <= out0;
dOutZ <= out1;
dOutIR <= outIR;
wen_mem <= out7;
en_mem <= out6;

END description;
```

Figure 1.18: VHDL code from cpu1.vhd

Figure 1.18 represents the CPU1.vhd file, which assigns and puts together the architecture of the PC. It consists of the different mux controls, ALU operations, inputs for the PC and the necessary outputs.

CPU Test Simulation

```
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;
```

Figure 1.19: VHDL code from cpu_test_sim.vhd

Figure 1.19 represents the code for the control processor unit testing simulation, which is in VHDL.

System Memory

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY altera_mf;
USE altera_mf.all;

ENTITY system_memory IS
    PORT
    (
        address     : IN STD_LOGIC_VECTOR (5 DOWNTO 0);
        clock     : IN STD_LOGIC  := '1';
        data      : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        wren      : IN STD_LOGIC ;
        q     : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
    );
END system_memory;

ARCHITECTURE SYN OF system_memory IS

    SIGNAL sub_wire0  : STD_LOGIC_VECTOR (31 DOWNTO 0);



    COMPONENT altsyncram
    GENERIC (
        clock_enable_input_a    : STRING;
        clock_enable_output_a       : STRING;
        init_file       : STRING;
        intended_device_family      : STRING;
        lpm_hint    : STRING;
        lpm_type    : STRING;
        numwords_a      : NATURAL;
        operation_mode      : STRING;
        outdata_aclr_a      : STRING;
        outdata_reg_a       : STRING;
        power_up_uninitialized      : STRING;
        widthad_a       : NATURAL;
        width_a     : NATURAL;
        width_byteena_a     : NATURAL
    );
    PORT (
            address_a   : IN STD_LOGIC_VECTOR (5 DOWNTO 0);
            clock0      : IN STD_LOGIC ;
            data_a      : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
            wren_a      : IN STD_LOGIC ;
            q_a     : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
    );
    END COMPONENT;

BEGIN
    q    <= sub_wire0(31 DOWNTO 0);

    altsyncram_component : altsyncram
    GENERIC MAP (
        clock_enable_input_a => "BYPASS",
        clock_enable_output_a => "BYPASS",
        init_file => "system_memory.mif",
        intended_device_family => "Cyclone II",
        lpm_hint => "ENABLE_RUNTIME_MOD=NO",
        lpm_type => "altsyncram",
        numwords_a => 64,
        operation_mode => "SINGLE_PORT",
        outdata_aclr_a => "NONE",
        outdata_reg_a => "CLOCK0",
        power_up_uninitialized => "FALSE",
        widthad_a => 6,
        width_a => 32,
        width_byteena_a => 1
    )
    PORT MAP (
        address_a => address,
        clock0 => clock,
        data_a => data,
        wren_a => wren,
        q_a => sub_wire0
    );

END SYN;
```

Figure 1.20: VHDL code from system_memory.vhd

Figure 1.20 represents the code for the system memory. In order to represent memory for different parts of the CPU, the memory file values can be altered.

Part II - The Complete CPU System

CPU Test Simulation Waveform

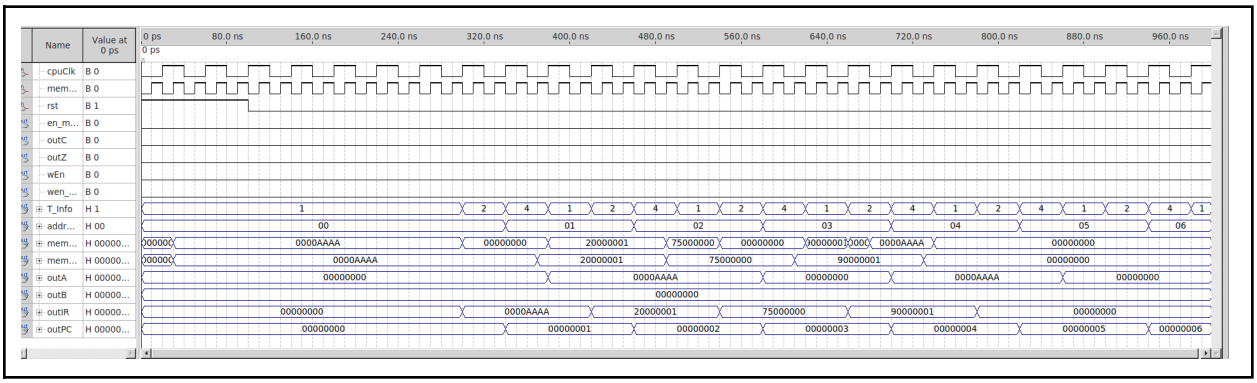


Figure 2.0: Waveform of cpu_test_sim.vwf

Figure 2.0 shows the general waveform for cpu_test_sim, which is for the starting values provided.

LDAI, STA, CLRA, LDA

Addr	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	ASCII
0	0000AAAA	20000001	75000000	90000001	00000000	00000000	00000000	00000000	00000000	00000000	.....
10	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
20	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
30	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
40	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
50	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
60	00000000	00000000	00000000	00000000							....

Figure 2.1.0: system_memory.mif for LDAI, STA, CLRA and LDA

Figure 2.1.0 represents what is stored for the 4 parts of the CPU. We can see that LDAI, which is Load A Immediate, is placing the value AAAA in the memory. For storing, STA we see the value 200000001 is placed. CLRA Which represents clearing A has the value 75, which will function with the CPU once the value reaches that point and LDA which is simply loading A is placed afterwards.

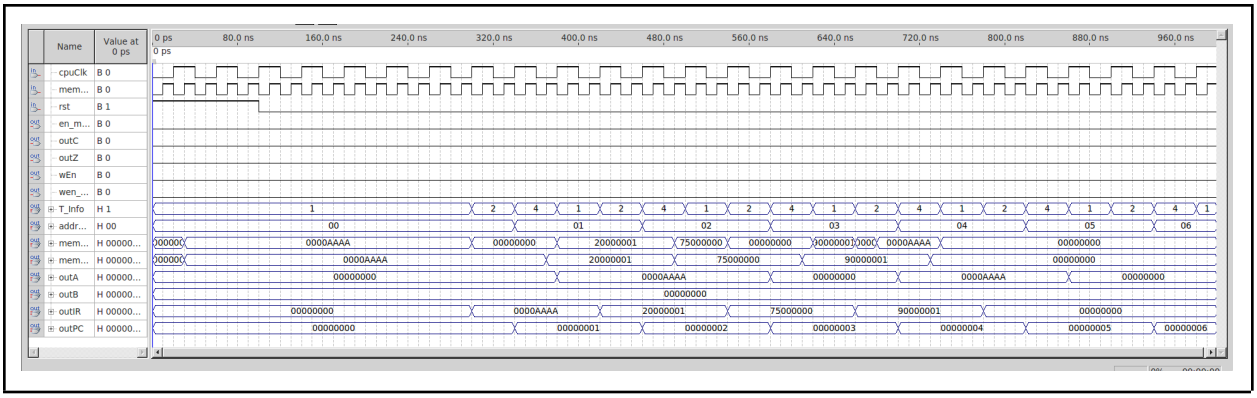


Figure 2.1.1: Waveform for LDAI, STA, CLRA, LDA

Figure 2.1.1 shows the waveform of these values. It is visible in the memory output row that the aforementioned values are all placed here. Our different outputs are representing how the values are manipulated and what we see in the outputs; for example, outA gives 0000AAAA, which is the immediately loaded A value.

LDBI, STB, CLRB, LDB

Addr	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	A
0	1000BBBB	30000001	76000000	A0000001	00000000	00000000	00000000	00000000	00000000	00000000	....
10	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	....
20	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	....
30	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	....
40	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	....
50	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	....
60	00000000	00000000	00000000	00000000							....

Figure 2.2.0: Memory file for LDBI, STB, CLRB, LDB

Figure 2.2.0 represents the values assigned to LDBI, which stands for Load B Immediate, STB, which stands for Store B, CLRB, which stands for Clear B, and LDB, which stands for LDB. It is visible that due to this being a B value, the values are all aligned to it in that way. So, 1000BBBB is the immediate value loaded to B, and all parts of the CPU are related to B.

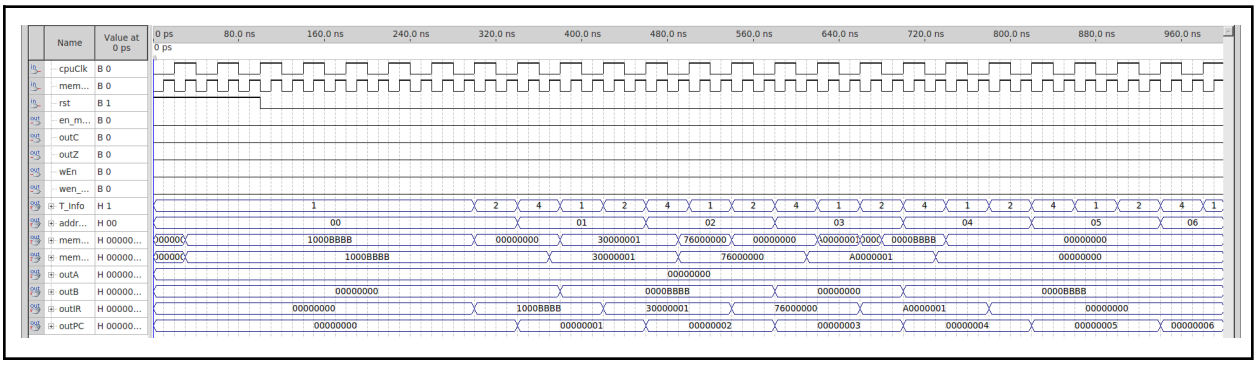


Figure 2.2.1: Waveform ofLDBI, STB, CLRB and LDB implementation

In the waveform visible in Figure 2.2.1, it is visible how the values loaded in the memory are placed. After going through the system, the outA output remains 0, as it is not interacted with. Instead, outB is implemented and has values that result from the B values travelling through.

LUI: Load Upper Immediate

Addr	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9
0	4000AAAA	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
10	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
20	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
30	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
40	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
50	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
60	00000000	00000000	00000000	00000000						

Figure 2.3.0: Memory file of LUI

Figure 2.3.0 Represents the value assigned to LUI, or Load Upper Immediate. The value, 4000AAAA, shows how in the upper 16 bits, a value is loaded. As for the AAAA is a repeating set of values.

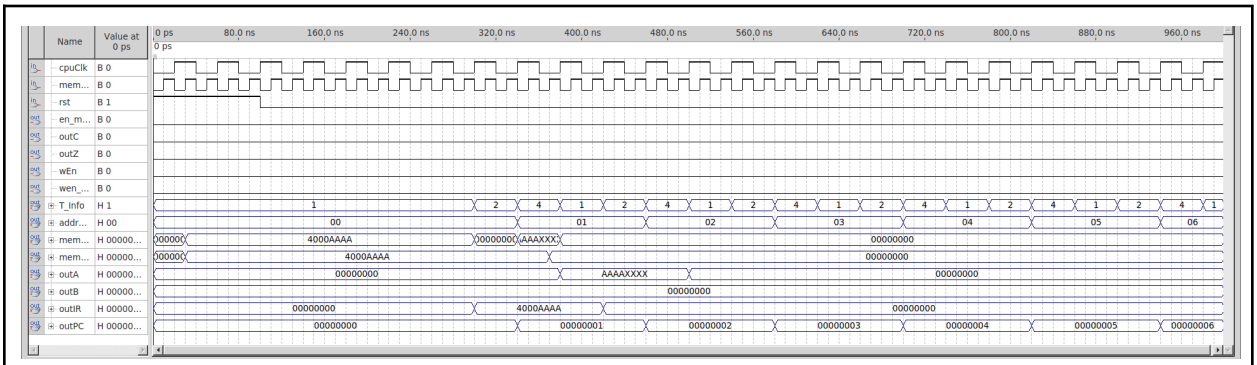


Figure 2.3.1: Waveform of LUI implementation

Figure 2.3.1 shows the implementation of the LUI function. It is visible by looking at the outputs, that in outIR, the value 4000AAAA is loaded immediately after a few moments, and the remaining values are all 0. This shows that it has been successfully implemented.

JMP: Jump

Addr	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	ASCII
0	5000AAAA	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
10	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
20	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
30	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
40	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
50	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
60	00000000	00000000	00000000	00000000							....

Figure 2.4.0: Memory file of JMP

This file represents the value assigned to Jump, which allows going from one place in the memory to another.

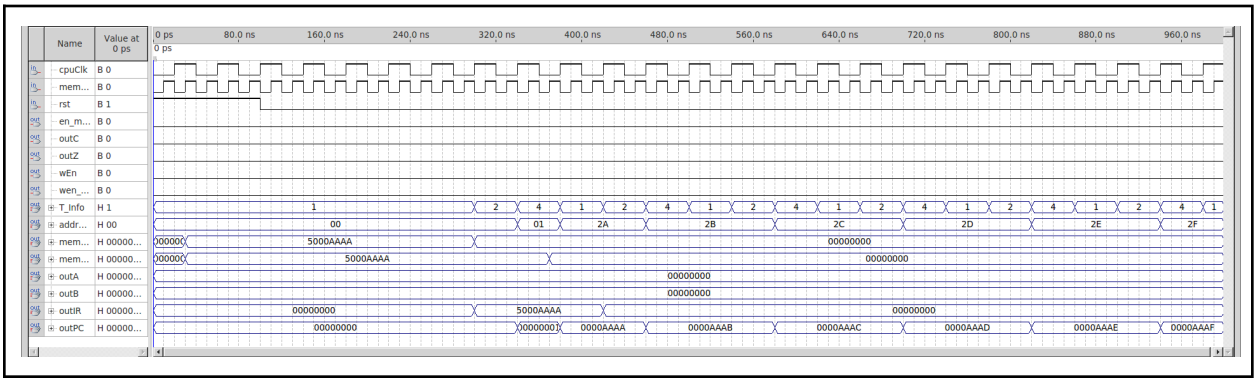


Figure 2.4.1: Waveform of JMP implementation in CPU

Figure 2.4.1 shows how JMP, or Jump is implemented. It is visible by looking at the memory of how the actual jump exists, and the T_info 1 value exists first in a certain timeframe at the beginning. However, in outPC, it is visible that by jumping, the 1 value now exists at a later point of the clock in a different address.

ANDI: AND Immediate

Addr	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	ASCII
0	00000006	7900000B	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
10	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
20	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
30	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
40	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
50	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
60	00000000	00000000	00000000	00000000							....

Figure 2.5.0: Memory File of AND Immediate

Figure 2.5.0 represents the value for ANDI, which covers two spaces. One that has values only in the last 4 bits, and one that covers the first 8 bits and then the last 4.

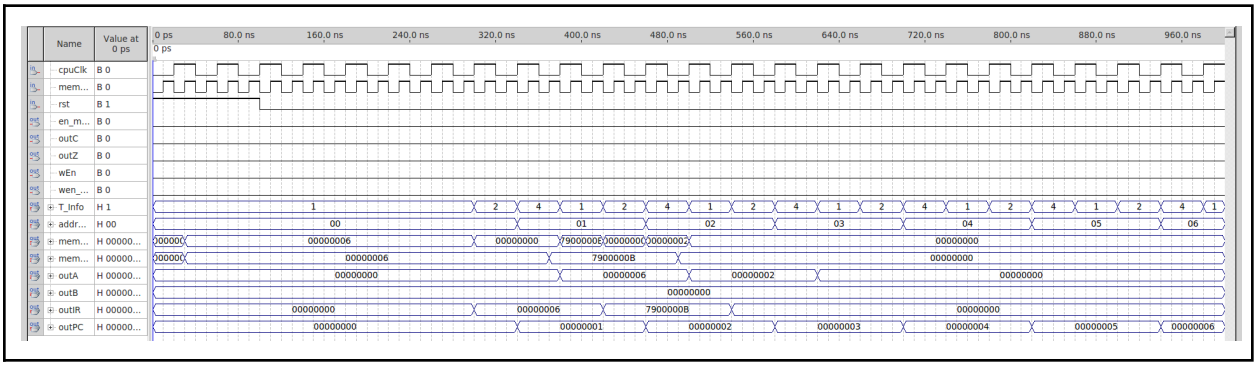


Figure 2.5.1: Waveform of ANDI CPU implementation

In Figure 2.5.1, it is visible that the two values allow for producing the AND of two values that are inserted.

ADDI: Add Immediate

Addr	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	ASCII
0	00000006	7100000B	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
10	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
20	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
30	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
40	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
50	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
60	00000000	00000000	00000000	00000000							....

Figure 2.6.0: Memory File of ADDI Add Immediate

This file shows how the value exists and is immediately added to in memory with another value.

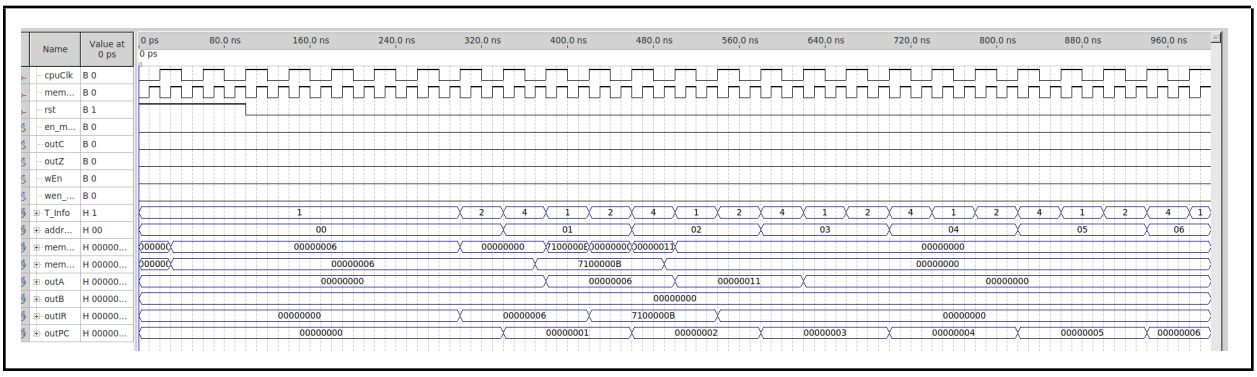


Figure 2.6.1: Waveform of ADDI Add Immediate

Figure 2.6.1 represents how given an input value, it can be added to a constant value as soon as it enters a certain time frame. If the memory row is observed, which is the actual ADDI implementation, the final output shows how the value has been added to.

ORI: Or Immediate

Addr	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	ASCII
0	00000006	7D00000B	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
10	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
20	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
30	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
40	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
50	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
60	00000000	00000000	00000000	00000000							....

Figure 2.7.0: Memory Value For ORI Or Immediate

The memory values for Or Immediate consist of two, one being the actual OR function (represented as 0006) and the other being the immediate value to be added.

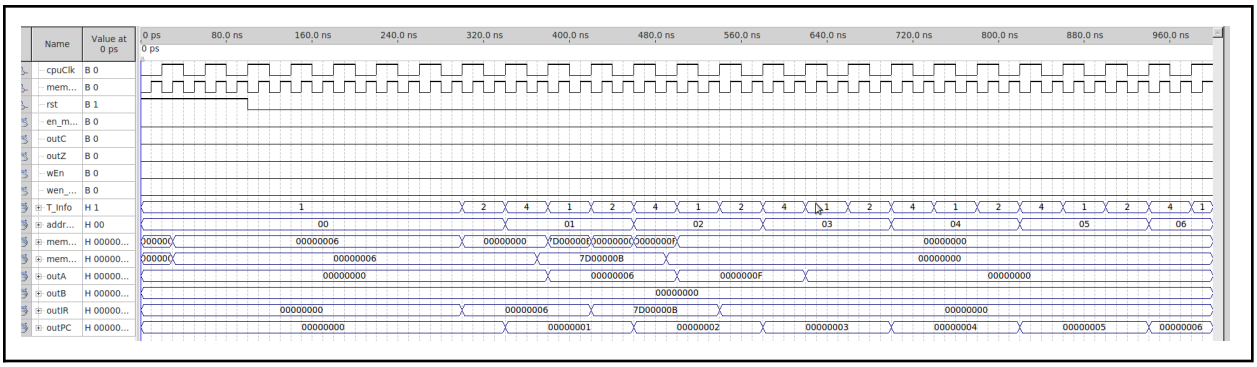


Figure 2.7.1: Waveform of ORI OR Immediate CPU Implementation

Figure 2.7.1 Shows the actual CPU Implementation of ORI. The memory holds the actual OR function, which is visibly output in the outA row. The value is successfully OR'd with another value.

ADD: Addition

Addr	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	ASCII
0	00000005	10000003	70000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
10	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
20	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
30	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
40	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
50	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
60	00000000	00000000	00000000	00000000							....

Figure 2.8.0: Memory File of ADD (Addition)

This memory file represents the addition function, which is represented by the space under +2. The other two values visible are there since in order to add, it is necessary for two values to exist.

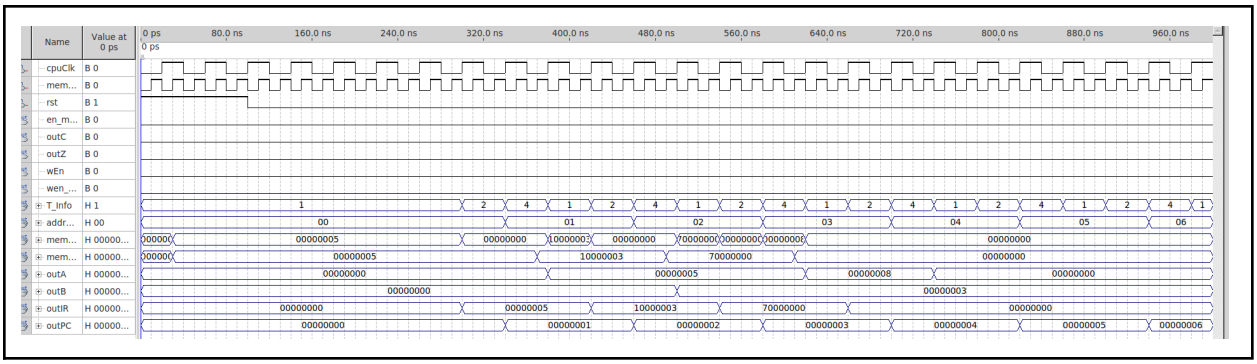


Figure 2.8.1: Waveform of ADD

Figure 2.8.1 represents the waveform that consists of ADD. It is visible that in memory, we have our two values that are to be added. Implemented by the add function, in outIR we can see that the two values are added to provide the final value.

SUB: Subtract

Addr	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	ASCII
0	00000005	10000003	72000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
10	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
20	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
30	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
40	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
50	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
60	00000000	00000000	00000000	00000000							....

Figure 2.9.0: Memory File for Subtract (SUB)

This memory file is similar to ADD, with the main difference being that the SUB is now implemented, having a value of 72 instead of 70. The values move on to be subtracted.

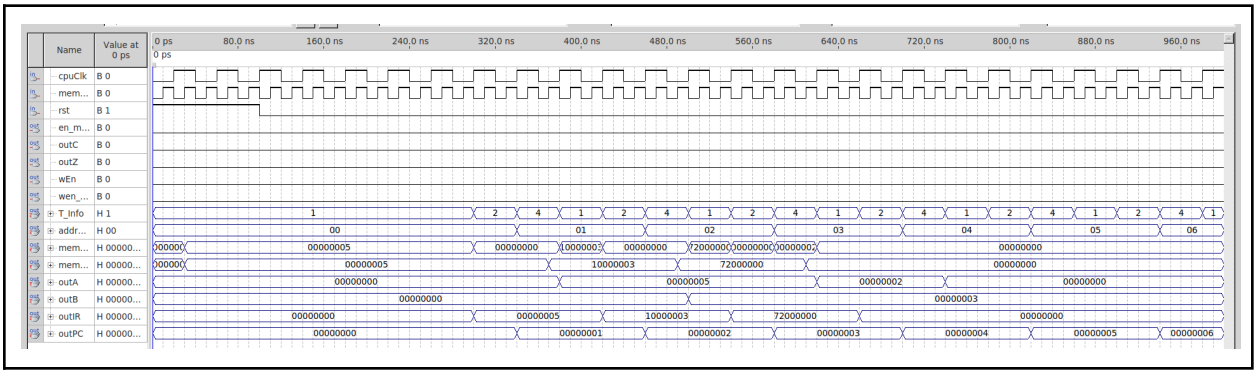


Figure 2.9.1: Waveform of SUB implementation on CPU

Figure 2.9.1 Shows the waveform of the SUB implementation on the CPU. It is once again visible to see how the values are implemented through the outA section, where we can see that 5 and 3 are subtracted to give us 2.

DECA: Decrement A

Addr	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	ASCII
0	0000AAAA	7E000000	70000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
10	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
20	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
30	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
40	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
50	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
60	00000000	00000000	00000000	00000000							....

Figure 2.10.0: Memory File of DECA (Decrement A)

This memory file shows the Decrementing of A. +0 represents the A possible A values (AAAA), and goes on to decrement the value by 1.

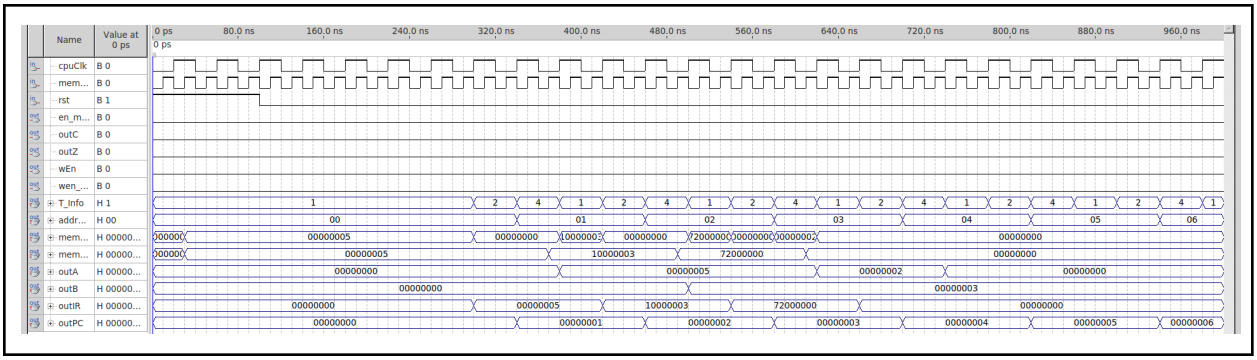


Figure 2.10.1: Waveform of DECA implementation on CPU

Figure 2.10.1 represents the decrementing of the values, whose output is visible in outA. With the initial value existing, the final output value is now one less, which shows that it has successfully been decremented.

INCA: Increment A

Addr	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	ASCII
0	0000AAAA	73000000	70000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
10	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
20	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
30	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
40	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
50	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
60	00000000	00000000	00000000	00000000							....

Figure 2.11.0: Memory File of Increment A (INCA)

The main difference between DECA and INCA lies in the +1 space, which changes the second 4 bits to allow for incrementing by one (or adding one). As it is still A that is being incremented, the +0 value remains as AAAA for the last 16 bits.

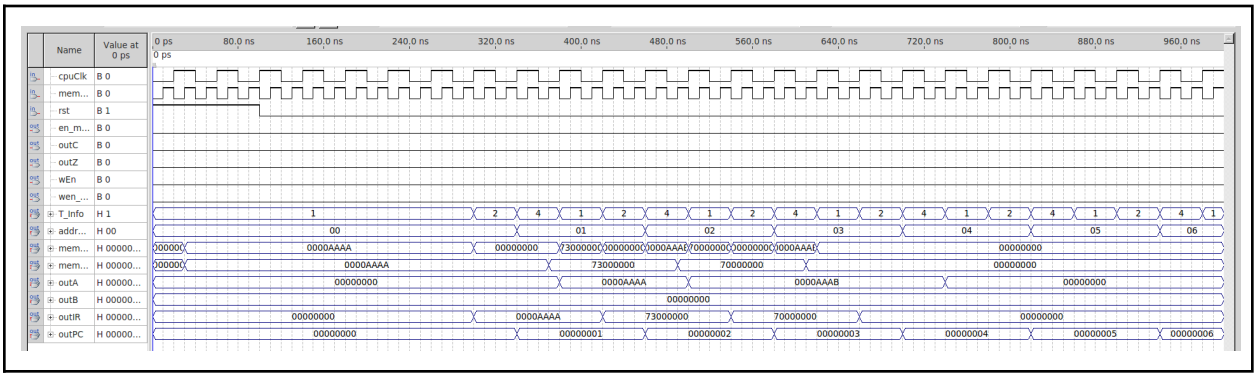


Figure 2.11.1: Waveform of Increment A (INCA)

Figure 2.11.1 Shows the incrementing of the value loaded into A. What begins as AAAA can be seen in outA as changing to AAAB, which shows that the last 4 bits have been incremented and changed the value from A to the next one, which is B.

ROL: Rotate Left

Addr	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	ASCII
0	00000008	74000000	70000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
10	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
20	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
30	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
40	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
50	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
60	00000000	00000000	00000000	00000000							....

Figure 2.12.0: Memory File for Rotate Left (ROL)

The memory file for ROL allows for shifting the bits in the existing value to the left by one.

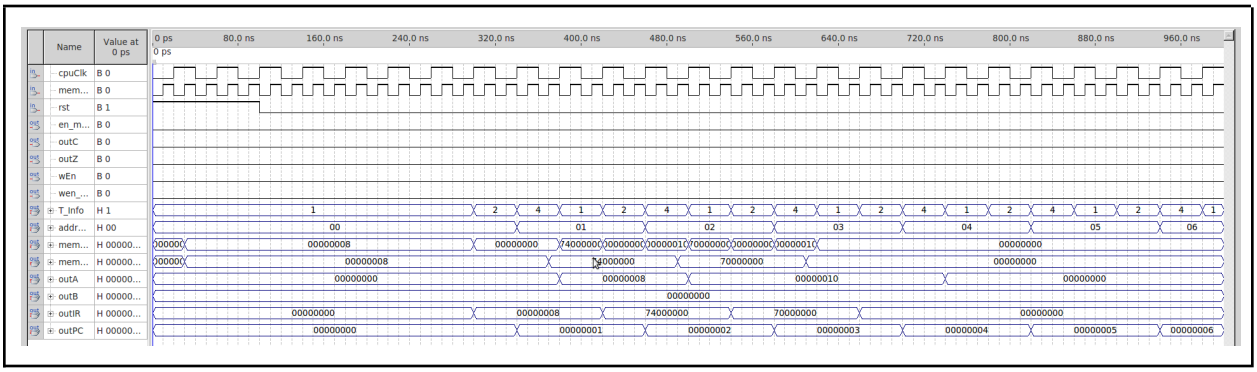


Figure 2.12.1: Waveform of Rotate Left ROL

Figure 2.12.1 represents how the values implemented are shifted to the left. If the outA section is observed, it is visible how the value has gone from 0001 to 0010, showing that it has successfully been shifted.

ROR: Rotate Right

Addr	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	ASCII
0	00000008	7F000000	70000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
10	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
20	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
30	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
40	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
50	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
60	00000000	00000000	00000000	00000000							....

Figure 2.13.0: Memory File for Rotate Right (ROR)

The memory file for ROR represents how the implemented value would be shifted right by one bit.

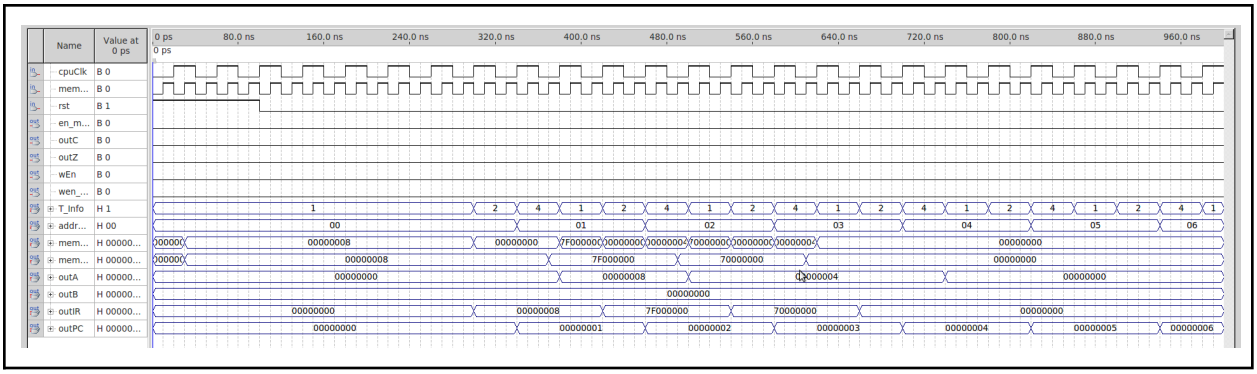


Figure 2.13.1: Waveform of Rotate Right (ROR) Implemented on the CPU

In this waveform, it is important to observe the rows for memory and for outA. With the initial given values, it is shifted one bit towards the right, representing it with an output of 0004 for the last 4 bits.

BEQ: Branch If Equal

Addr	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	ASCII
0	0000AAAA	1000AAAA	600000F0	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
10	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
20	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
30	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
40	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
50	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
60	00000000	00000000	00000000	00000000							....

Figure 2.14.0: Memory File for Branch If Equal (BEQ)

For this section, we are focusing on AAAA. There are two steps for this process. First, two values are compared to each other and determined if they are equal or not. If they are indeed equal, branching is then applied.

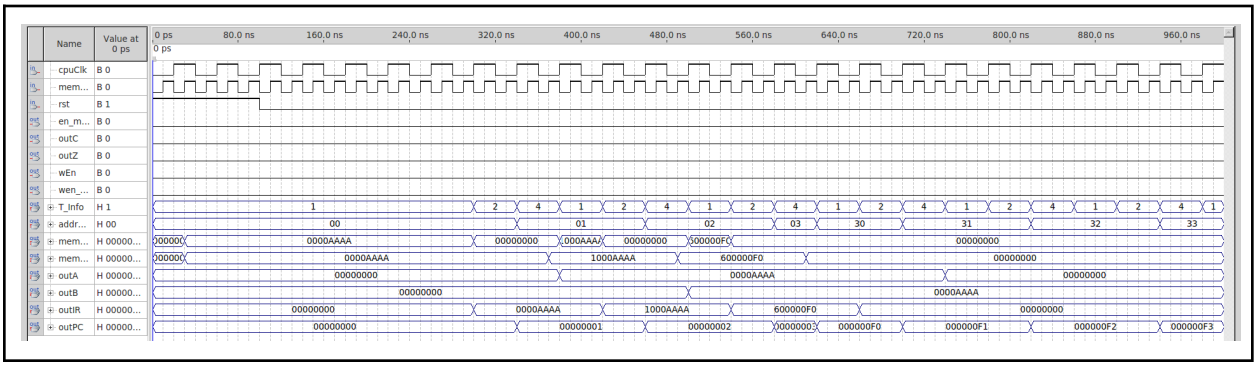


Figure 2.14.1: Waveform of Branch If Equal (BEQ)

In this waveform, we can see that the two values that we would be looking at are in AAAA. The system has AAAA followed by 1000AAAA; since we want to know if it's equal, we can compare outA and outB. At a certain point in the time, it is visible that both have 0000AAAA, which is specifically where it is confirmed they are equal. In that specific space, we can see that outIR changes to 6000000F0, and outPC also changes, showing that it is successfully branched.

BNE: Branch If Not Equal

Addr	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	ASCII
0	0000AAAA	1000BBBB	800000F0	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
10	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
20	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
30	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
40	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
50	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	.....
60	00000000	00000000	00000000	00000000							....

Figure 2.15.0: Memory File of Branch If Not Equal (BNE)

This memory file is representing how branching would occur if it is not equal. We are comparing the values in +0, and in the case that it is not equal it would branch, represented by 1000BBBB.

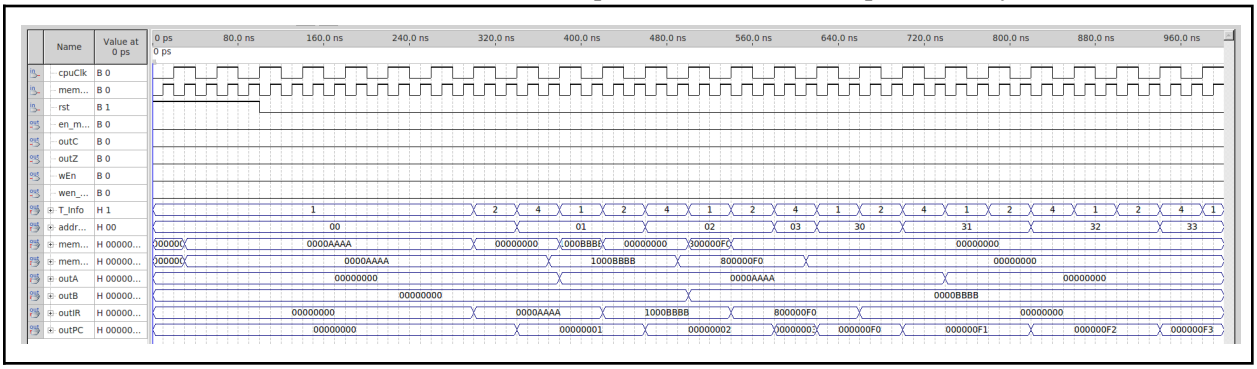


Figure 2.15.1: Waveform of Branch If Not Equal (BNE)

In this waveform, it is visible how as the timing goes onwards, the comparisons are made. We can see here that for outA and outB, for a good part of the beginning they are first both 00000000. However, after a certain point, we see that outA has 0000AAAA, while outB has 0000BBBB. Here, we are implementing branch if not equal, which is then done and visible in both outIR and outPC, as the values go from incrementing in outPC by one, to F0, showing that a branch has been done.