

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

Instructor:	Demetres Kostas
--------------------	-----------------

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

<i>Submission Date:</i>	Apr 2, 2025
<i>Due Date:</i>	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;
39

```

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.

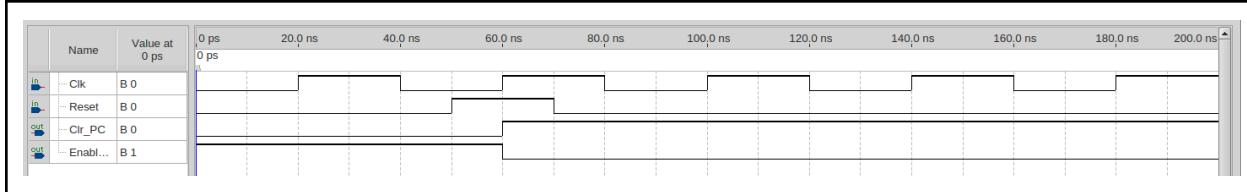


Figure 1.1: Waveform from reset_circuit.vwf

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

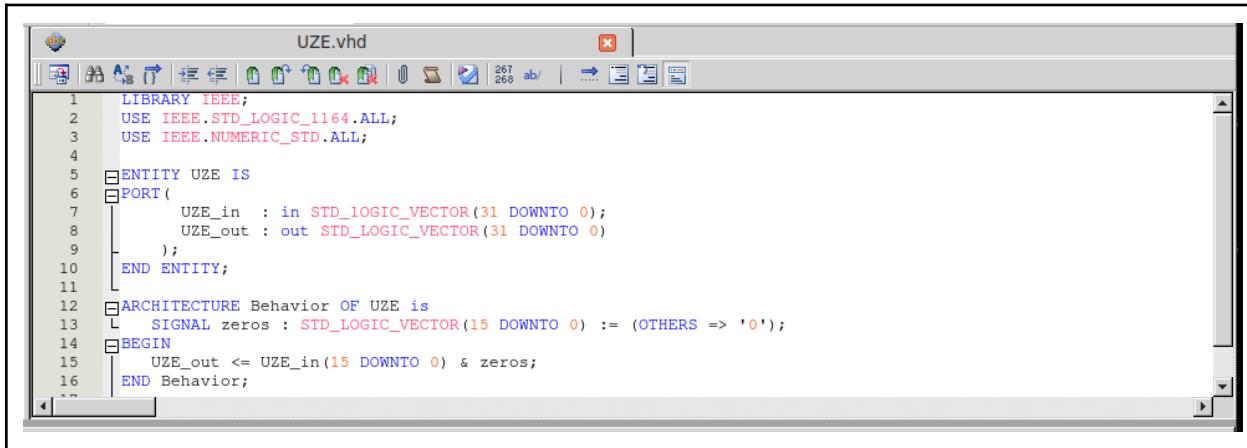
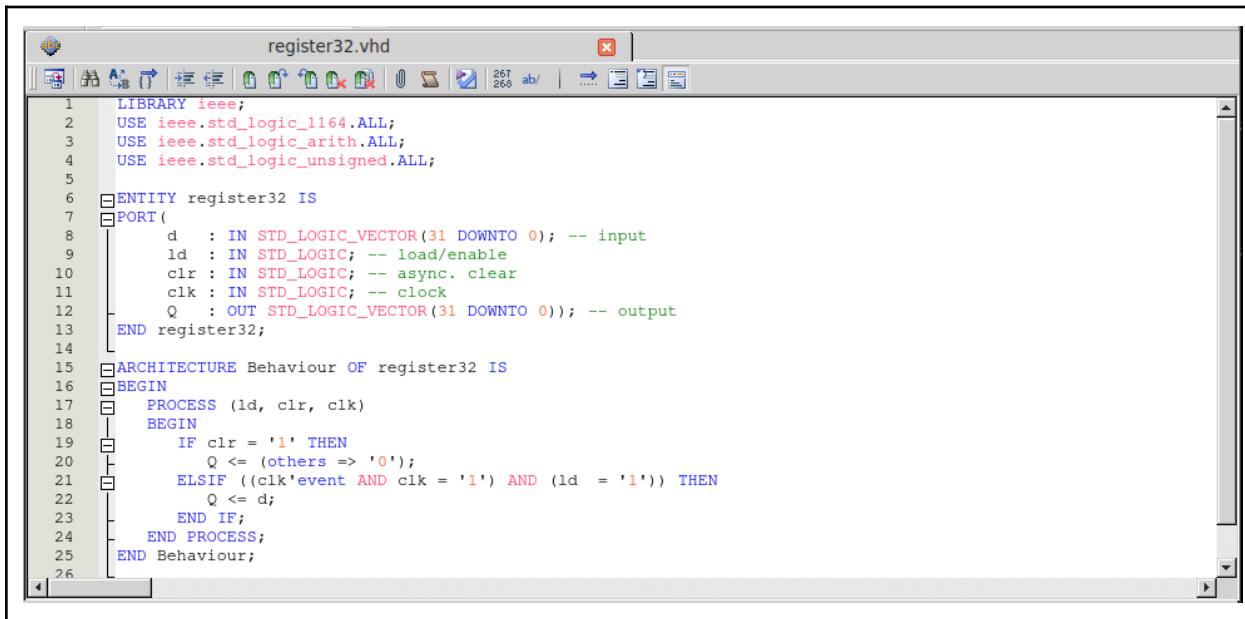


Figure 1.2: VHDL code from UZE.vhd

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



```

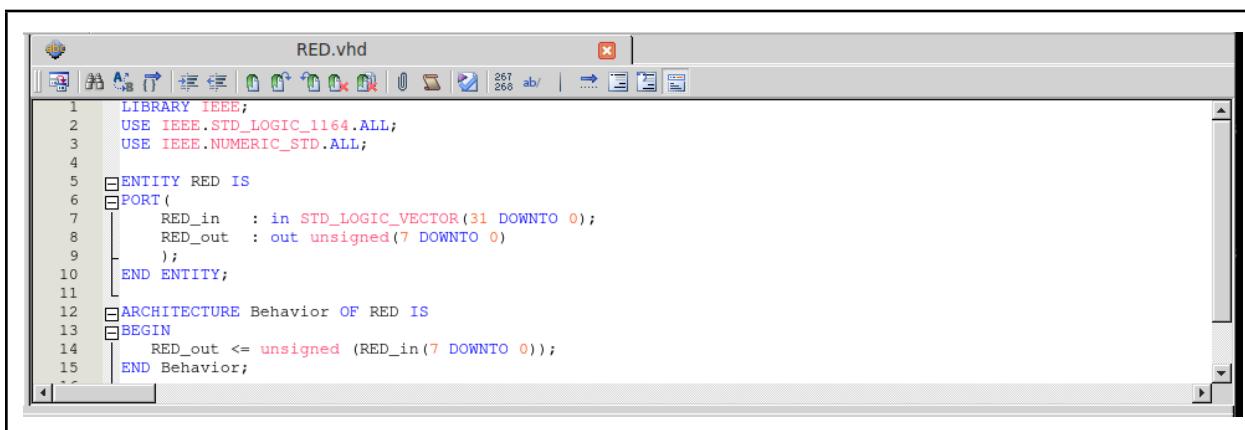
1 LIBRARY ieee;
2 USE ieee.std_logic_1164.ALL;
3 USE ieee.std_logic_arith.ALL;
4 USE ieee.std_logic_unsigned.ALL;
5
6 ENTITY register32 IS
7 PORT(
8     d : IN STD_LOGIC_VECTOR(31 DOWNTO 0); -- input
9     ld : IN STD_LOGIC; -- load-enable
10    clr : IN STD_LOGIC; -- async. clear
11    clk : IN STD_LOGIC; -- clock
12    Q : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)); -- output
13 END register32;
14
15 ARCHITECTURE Behaviour OF register32 IS
16 BEGIN
17 BEGIN
18     PROCESS (ld, clr, clk)
19     BEGIN
20         IF clr = '1' THEN
21             Q <= (others => '0');
22         ELSIF ((clk'event AND clk = '1') AND (ld = '1')) THEN
23             Q <= d;
24         END IF;
25     END PROCESS;
26 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



```

1 LIBRARY IEEE;
2 USE IEEE.STD_LOGIC_1164.ALL;
3 USE IEEE.NUMERIC_STD.ALL;
4
5 ENTITY RED IS
6 PORT(
7     RED_in : in STD_LOGIC_VECTOR(31 DOWNTO 0);
8     RED_out : out unsigned(7 DOWNTO 0)
9 );
10 END ENTITY;
11
12 ARCHITECTURE Behavior OF RED IS
13 BEGIN
14     RED_out <= unsigned (RED_in(7 DOWNTO 0));
15 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

```

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

```

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

```

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

```

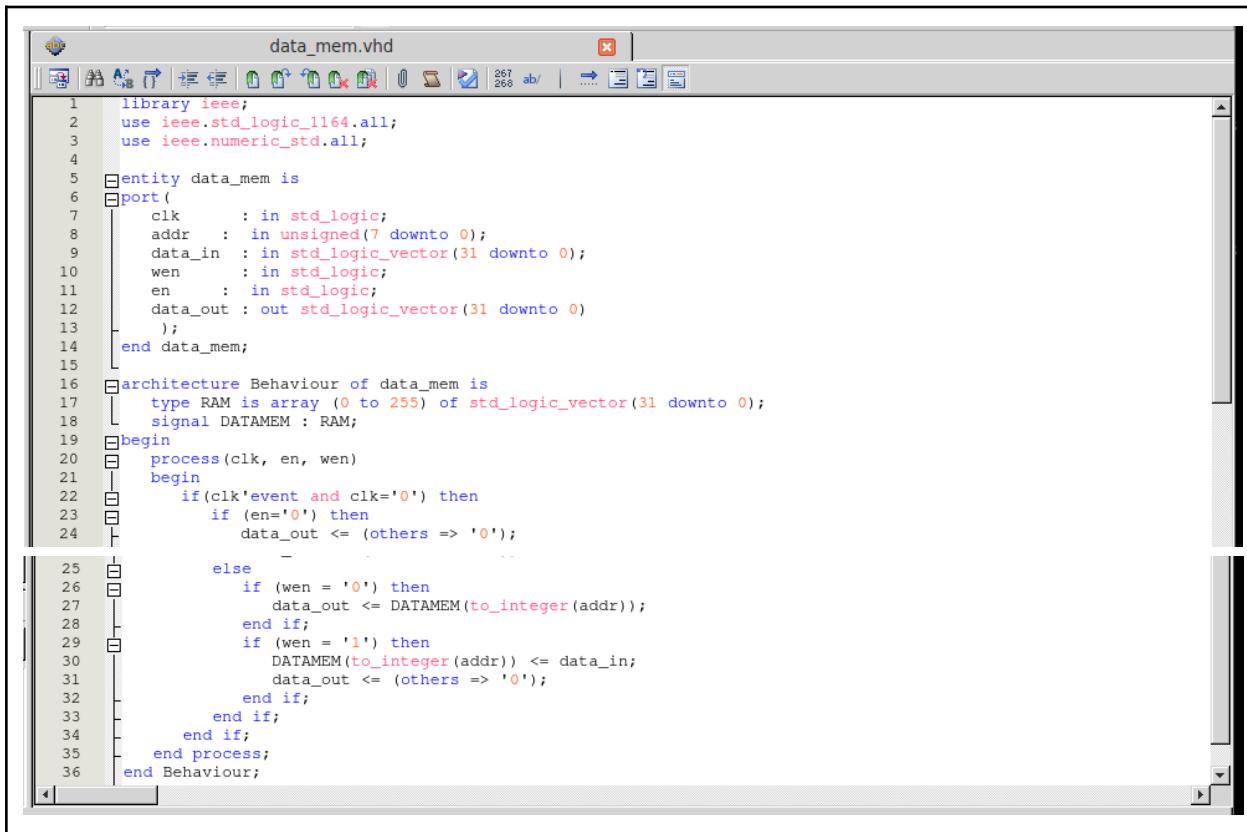
1 library ieee;
2 use ieee.std_logic_1164.all;
3
4 entity fulladd is
5 port(
6   Cin, x, y : in std_logic;
7   s, Cout : out std_logic
8 );
9 end fulladd;
10
11 architecture Behaviour of fulladd is
12 begin
13   s <= x xor y xor Cin;
14   Cout <= (x and y) or (Cin and x) or (Cin and y);
15 end Behaviour;

```

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 xorring of the three inputs x, y and Cin ($x \oplus y \oplus \text{Cin}$). 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



```

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 output). 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)

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
entity alu is
port(
    a : in std_logic_vector(31 downto 0);
    b : in std_logic_vector(31 downto 0);
    op : in std_logic_vector(2 downto 0);
    result : out std_logic_vector(31 downto 0);
    zero : out std_logic;
    cout : out std_logic
);
end alu;
architecture Behavior of alu is
component adder32
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 component;
signal result_s: std_logic_vector(31 downto 0):= (others => '0');
signal result_add: std_logic_vector(31 downto 0):= (others => '0');
signal result_sub: std_logic_vector(31 downto 0):= (others => '0');
signal cout_s : std_logic := '0';
signal cout_add : std_logic := '0';
signal cout_sub : std_logic := '0';
signal zero_s : std_logic;
begin
    add0 : adder32 port map (op(2),a,b,result_add, cout_add);
    sub0 : adder32 port map (op(2), a, not b, result_sub, cout_sub);
process (a, b, op)
begin
    case (op) is
        when "000" =>
            result_s <= a and b;
            cout_s <= '0';

        when "001" =>
            result_s <= a or b;
            cout_s <= '0';
        when "010" =>
            result_s <= result_add;
            cout_s <= cout_add;
        when "011" =>
            result_s <= b;
            cout_s <= '0';
        when "110" =>
            result_s <= result_sub;
            cout_s <= cout_sub;
        when "100" =>
            result_s <= a(30 downto 0) & '0';
            cout_s <= a(31);
        when "101" =>
            result_s <= '0' & a(31 downto 1);
            cout_s <= '0';
        when others =>
            result_s <= a;
            cout_s <= '0';
    end case;
    case (result_s) is
        when (others => '0') =>
            zero_s <= '1';
        when others =>
            zero_s <= '0';
    end case;
end process;
result <= result_s;
cout <= cout_s;
zero <= zero_s;
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

```

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;
begin
    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

```

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;
begin
    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

```

library ieee;
use ieee.std_logic_1164.all;
entity adder4 is
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 adder4;
architecture Behaviour of adder4 is
component fulladd
port(
    Cin, x, y : in std_logic;
    s, Cout : out std_logic
);
end component;
signal C : std_logic_vector (1 to 3);
begin
    stage0: fulladd port map (Cin, X(0), Y(0), S(0), C(1));
    stage1: fulladd port map (C(1), X(1), Y(1), S(1), C(2));
    stage2: fulladd port map (C(2), X(2), Y(2), S(2), C(3));
    stage3: fulladd port map (C(3), X(3), Y(3), S(3), Cout);
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

```

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 + 4;
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

```

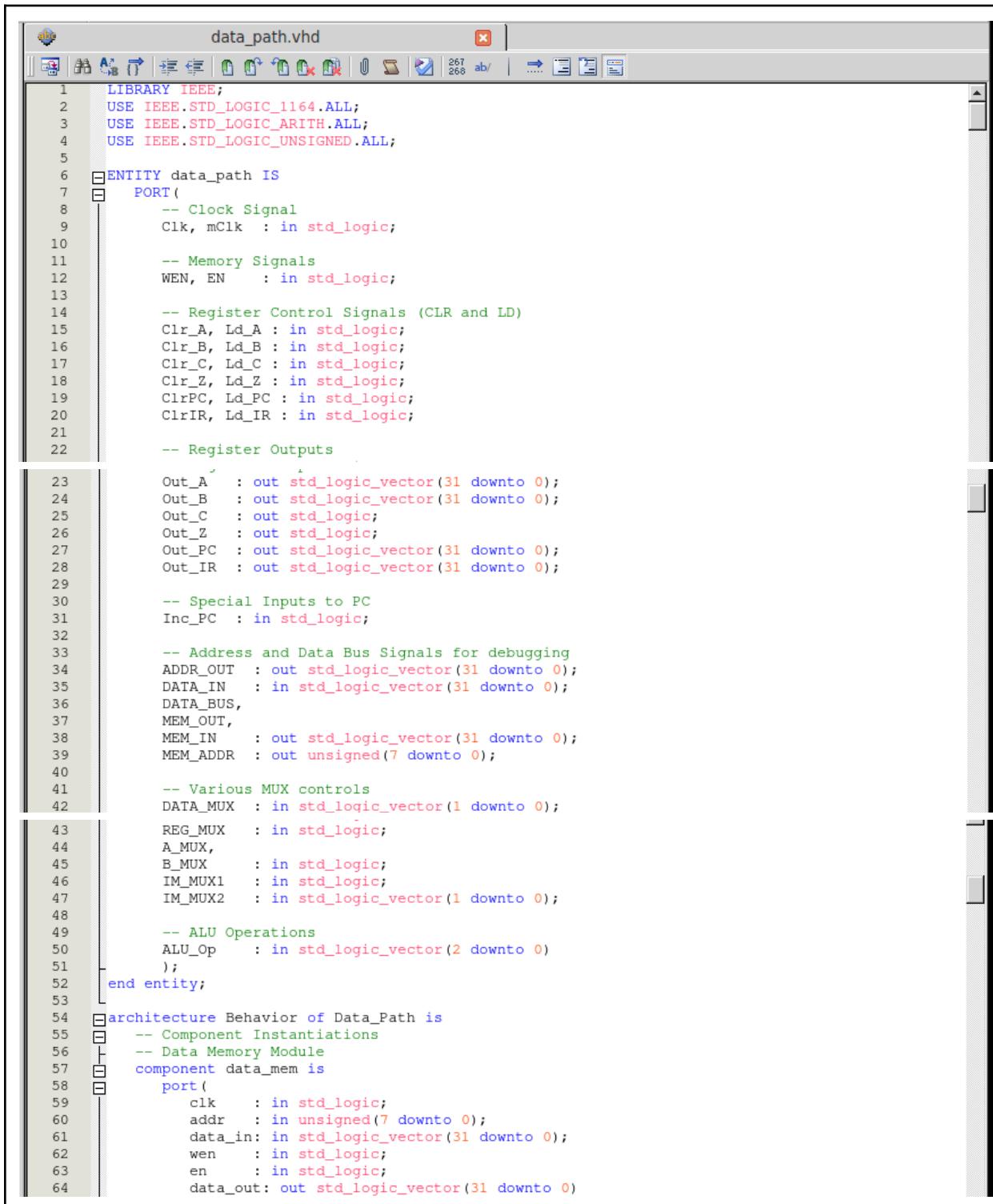
1 LIBRARY ieee;
2 USE ieee.std_logic_1164.ALL;
3 USE ieee.std_logic_arith.ALL;
4 USE ieee.std_logic_unsigned.ALL;
5
6 ENTITY pc IS
7 PORT(
8     clr :IN STD_LOGIC;
9     clk :IN STD_LOGIC;
10    ld :IN STD_LOGIC;
11    inc :IN STD_LOGIC;
12    d : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
13    q : OUT STD_LOGIC_VECTOR(31 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



```

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 data_path IS
7 PORT(
8     -- Clock Signal
9     Clk, mClk : in std_logic;
10
11    -- Memory Signals
12    WEN, EN   : in std_logic;
13
14    -- Register Control Signals (CLR and LD)
15    Clr_A, Ld_A : in std_logic;
16    Clr_B, Ld_B : in std_logic;
17    Clr_C, Ld_C : in std_logic;
18    Clr_Z, Ld_Z : in std_logic;
19    ClrPC, Ld_PC : in std_logic;
20    ClrIR, Ld_IR : in std_logic;
21
22    -- Register Outputs
23    Out_A    : out std_logic_vector(31 downto 0);
24    Out_B    : out std_logic_vector(31 downto 0);
25    Out_C    : out std_logic;
26    Out_Z    : out std_logic;
27    Out_PC   : out std_logic_vector(31 downto 0);
28    Out_IR   : out std_logic_vector(31 downto 0);
29
30    -- Special Inputs to PC
31    Inc_PC   : in std_logic;
32
33    -- Address and Data Bus Signals for debugging
34    ADDR_OUT : out std_logic_vector(31 downto 0);
35    DATA_IN  : in std_logic_vector(31 downto 0);
36    DATA_BUS,
37    MEM_OUT,
38    MEM_IN   : out std_logic_vector(31 downto 0);
39    MEM_ADDR  : out unsigned(7 downto 0);
40
41    -- Various MUX controls
42    DATA_MUX  : in std_logic_vector(1 downto 0);
43    REG_MUX   : in std_logic;
44    A_MUX,
45    B_MUX    : in std_logic;
46    IM_MUX1  : in std_logic;
47    IM_MUX2  : in std_logic_vector(1 downto 0);
48
49    -- ALU Operations
50    ALU_Op   : in std_logic_vector(2 downto 0)
51 );
52 end entity;
53
54 architecture Behavior of Data_Path is
55 begin
56    -- Component Instantiations
57    -- Data Memory Module
58    component data_mem is
59        port(
60            clk      : in std_logic;
61            addr    : in unsigned(7 downto 0);
62            data_in : in std_logic_vector(31 downto 0);
63            wen     : in std_logic;
64            en      : in std_logic;
65            data_out: out std_logic_vector(31 downto 0)
66        );
67
68    end component;
69
70    data_mem1: data_mem
71        port map(
72            clk      => Clk,
73            addr    => MEM_ADDR,
74            data_in => Out_Z,
75            wen     => WEN,
76            en      => EN,
77            data_out=> Out_Z
78        );
79
80    process is
81    begin
82        wait on Inc_PC;
83        if Inc_PC = '1' then
84            PC := PC + 1;
85        end if;
86    end process;
87
88    process is
89    begin
90        wait on Out_Z;
91        if Out_Z = '1' then
92            PC := PC + 1;
93        end if;
94    end process;
95
96    process is
97    begin
98        wait on Out_Z;
99        if Out_Z = '1' then
100            PC := PC + 1;
101        end if;
102    end process;
103
104    process is
105    begin
106        wait on Out_Z;
107        if Out_Z = '1' then
108            PC := PC + 1;
109        end if;
110    end process;
111
112    process is
113    begin
114        wait on Out_Z;
115        if Out_Z = '1' then
116            PC := PC + 1;
117        end if;
118    end process;
119
120    process is
121    begin
122        wait on Out_Z;
123        if Out_Z = '1' then
124            PC := PC + 1;
125        end if;
126    end process;
127
128    process is
129    begin
130        wait on Out_Z;
131        if Out_Z = '1' then
132            PC := PC + 1;
133        end if;
134    end process;
135
136    process is
137    begin
138        wait on Out_Z;
139        if Out_Z = '1' then
140            PC := PC + 1;
141        end if;
142    end process;
143
144    process is
145    begin
146        wait on Out_Z;
147        if Out_Z = '1' then
148            PC := PC + 1;
149        end if;
150    end process;
151
152    process is
153    begin
154        wait on Out_Z;
155        if Out_Z = '1' then
156            PC := PC + 1;
157        end if;
158    end process;
159
160    process is
161    begin
162        wait on Out_Z;
163        if Out_Z = '1' then
164            PC := PC + 1;
165        end if;
166    end process;
167
168    process is
169    begin
170        wait on Out_Z;
171        if Out_Z = '1' then
172            PC := PC + 1;
173        end if;
174    end process;
175
176    process is
177    begin
178        wait on Out_Z;
179        if Out_Z = '1' then
180            PC := PC + 1;
181        end if;
182    end process;
183
184    process is
185    begin
186        wait on Out_Z;
187        if Out_Z = '1' then
188            PC := PC + 1;
189        end if;
190    end process;
191
192    process is
193    begin
194        wait on Out_Z;
195        if Out_Z = '1' then
196            PC := PC + 1;
197        end if;
198    end process;
199
200    process is
201    begin
202        wait on Out_Z;
203        if Out_Z = '1' then
204            PC := PC + 1;
205        end if;
206    end process;
207
208    process is
209    begin
210        wait on Out_Z;
211        if Out_Z = '1' then
212            PC := PC + 1;
213        end if;
214    end process;
215
216    process is
217    begin
218        wait on Out_Z;
219        if Out_Z = '1' then
220            PC := PC + 1;
221        end if;
222    end process;
223
224    process is
225    begin
226        wait on Out_Z;
227        if Out_Z = '1' then
228            PC := PC + 1;
229        end if;
230    end process;
231
232    process is
233    begin
234        wait on Out_Z;
235        if Out_Z = '1' then
236            PC := PC + 1;
237        end if;
238    end process;
239
240    process is
241    begin
242        wait on Out_Z;
243        if Out_Z = '1' then
244            PC := PC + 1;
245        end if;
246    end process;
247
248    process is
249    begin
250        wait on Out_Z;
251        if Out_Z = '1' then
252            PC := PC + 1;
253        end if;
254    end process;
255
256    process is
257    begin
258        wait on Out_Z;
259        if Out_Z = '1' then
260            PC := PC + 1;
261        end if;
262    end process;
263
264    process is
265    begin
266        wait on Out_Z;
267        if Out_Z = '1' then
268            PC := PC + 1;
269        end if;
270    end process;
271
272    process is
273    begin
274        wait on Out_Z;
275        if Out_Z = '1' then
276            PC := PC + 1;
277        end if;
278    end process;
279
280    process is
281    begin
282        wait on Out_Z;
283        if Out_Z = '1' then
284            PC := PC + 1;
285        end if;
286    end process;
287
288    process is
289    begin
290        wait on Out_Z;
291        if Out_Z = '1' then
292            PC := PC + 1;
293        end if;
294    end process;
295
296    process is
297    begin
298        wait on Out_Z;
299        if Out_Z = '1' then
300            PC := PC + 1;
301        end if;
302    end process;
303
304    process is
305    begin
306        wait on Out_Z;
307        if Out_Z = '1' then
308            PC := PC + 1;
309        end if;
310    end process;
311
312    process is
313    begin
314        wait on Out_Z;
315        if Out_Z = '1' then
316            PC := PC + 1;
317        end if;
318    end process;
319
320    process is
321    begin
322        wait on Out_Z;
323        if Out_Z = '1' then
324            PC := PC + 1;
325        end if;
326    end process;
327
328    process is
329    begin
330        wait on Out_Z;
331        if Out_Z = '1' then
332            PC := PC + 1;
333        end if;
334    end process;
335
336    process is
337    begin
338        wait on Out_Z;
339        if Out_Z = '1' then
340            PC := PC + 1;
341        end if;
342    end process;
343
344    process is
345    begin
346        wait on Out_Z;
347        if Out_Z = '1' then
348            PC := PC + 1;
349        end if;
350    end process;
351
352    process is
353    begin
354        wait on Out_Z;
355        if Out_Z = '1' then
356            PC := PC + 1;
357        end if;
358    end process;
359
360    process is
361    begin
362        wait on Out_Z;
363        if Out_Z = '1' then
364            PC := PC + 1;
365        end if;
366    end process;
367
368    process is
369    begin
370        wait on Out_Z;
371        if Out_Z = '1' then
372            PC := PC + 1;
373        end if;
374    end process;
375
376    process is
377    begin
378        wait on Out_Z;
379        if Out_Z = '1' then
380            PC := PC + 1;
381        end if;
382    end process;
383
384    process is
385    begin
386        wait on Out_Z;
387        if Out_Z = '1' then
388            PC := PC + 1;
389        end if;
390    end process;
391
392    process is
393    begin
394        wait on Out_Z;
395        if Out_Z = '1' then
396            PC := PC + 1;
397        end if;
398    end process;
399
400    process is
401    begin
402        wait on Out_Z;
403        if Out_Z = '1' then
404            PC := PC + 1;
405        end if;
406    end process;
407
408    process is
409    begin
410        wait on Out_Z;
411        if Out_Z = '1' then
412            PC := PC + 1;
413        end if;
414    end process;
415
416    process is
417    begin
418        wait on Out_Z;
419        if Out_Z = '1' then
420            PC := PC + 1;
421        end if;
422    end process;
423
424    process is
425    begin
426        wait on Out_Z;
427        if Out_Z = '1' then
428            PC := PC + 1;
429        end if;
430    end process;
431
432    process is
433    begin
434        wait on Out_Z;
435        if Out_Z = '1' then
436            PC := PC + 1;
437        end if;
438    end process;
439
440    process is
441    begin
442        wait on Out_Z;
443        if Out_Z = '1' then
444            PC := PC + 1;
445        end if;
446    end process;
447
448    process is
449    begin
450        wait on Out_Z;
451        if Out_Z = '1' then
452            PC := PC + 1;
453        end if;
454    end process;
455
456    process is
457    begin
458        wait on Out_Z;
459        if Out_Z = '1' then
460            PC := PC + 1;
461        end if;
462    end process;
463
464    process is
465    begin
466        wait on Out_Z;
467        if Out_Z = '1' then
468            PC := PC + 1;
469        end if;
470    end process;
471
472    process is
473    begin
474        wait on Out_Z;
475        if Out_Z = '1' then
476            PC := PC + 1;
477        end if;
478    end process;
479
480    process is
481    begin
482        wait on Out_Z;
483        if Out_Z = '1' then
484            PC := PC + 1;
485        end if;
486    end process;
487
488    process is
489    begin
490        wait on Out_Z;
491        if Out_Z = '1' then
492            PC := PC + 1;
493        end if;
494    end process;
495
496    process is
497    begin
498        wait on Out_Z;
499        if Out_Z = '1' then
500            PC := PC + 1;
501        end if;
502    end process;
503
504    process is
505    begin
506        wait on Out_Z;
507        if Out_Z = '1' then
508            PC := PC + 1;
509        end if;
510    end process;
511
512    process is
513    begin
514        wait on Out_Z;
515        if Out_Z = '1' then
516            PC := PC + 1;
517        end if;
518    end process;
519
520    process is
521    begin
522        wait on Out_Z;
523        if Out_Z = '1' then
524            PC := PC + 1;
525        end if;
526    end process;
527
528    process is
529    begin
530        wait on Out_Z;
531        if Out_Z = '1' then
532            PC := PC + 1;
533        end if;
534    end process;
535
536    process is
537    begin
538        wait on Out_Z;
539        if Out_Z = '1' then
540            PC := PC + 1;
541        end if;
542    end process;
543
544    process is
545    begin
546        wait on Out_Z;
547        if Out_Z = '1' then
548            PC := PC + 1;
549        end if;
550    end process;
551
552    process is
553    begin
554        wait on Out_Z;
555        if Out_Z = '1' then
556            PC := PC + 1;
557        end if;
558    end process;
559
560    process is
561    begin
562        wait on Out_Z;
563        if Out_Z = '1' then
564            PC := PC + 1;
565        end if;
566    end process;
567
568    process is
569    begin
570        wait on Out_Z;
571        if Out_Z = '1' then
572            PC := PC + 1;
573        end if;
574    end process;
575
576    process is
577    begin
578        wait on Out_Z;
579        if Out_Z = '1' then
580            PC := PC + 1;
581        end if;
582    end process;
583
584    process is
585    begin
586        wait on Out_Z;
587        if Out_Z = '1' then
588            PC := PC + 1;
589        end if;
590    end process;
591
592    process is
593    begin
594        wait on Out_Z;
595        if Out_Z = '1' then
596            PC := PC + 1;
597        end if;
598    end process;
599
600    process is
601    begin
602        wait on Out_Z;
603        if Out_Z = '1' then
604            PC := PC + 1;
605        end if;
606    end process;
607
608    process is
609    begin
610        wait on Out_Z;
611        if Out_Z = '1' then
612            PC := PC + 1;
613        end if;
614    end process;
615
616    process is
617    begin
618        wait on Out_Z;
619        if Out_Z = '1' then
620            PC := PC + 1;
621        end if;
622    end process;
623
624    process is
625    begin
626        wait on Out_Z;
627        if Out_Z = '1' then
628            PC := PC + 1;
629        end if;
630    end process;
631
632    process is
633    begin
634        wait on Out_Z;
635        if Out_Z = '1' then
636            PC := PC + 1;
637        end if;
638    end process;
639
640    process is
641    begin
642        wait on Out_Z;
643        if Out_Z = '1' then
644            PC := PC + 1;
645        end if;
646    end process;
647
648    process is
649    begin
650        wait on Out_Z;
651        if Out_Z = '1' then
652            PC := PC + 1;
653        end if;
654    end process;
655
656    process is
657    begin
658        wait on Out_Z;
659        if Out_Z = '1' then
660            PC := PC + 1;
661        end if;
662    end process;
663
664    process is
665    begin
666        wait on Out_Z;
667        if Out_Z = '1' then
668            PC := PC + 1;
669        end if;
670    end process;
671
672    process is
673    begin
674        wait on Out_Z;
675        if Out_Z = '1' then
676            PC := PC + 1;
677        end if;
678    end process;
679
680    process is
681    begin
682        wait on Out_Z;
683        if Out_Z = '1' then
684            PC := PC + 1;
685        end if;
686    end process;
687
688    process is
689    begin
690        wait on Out_Z;
691        if Out_Z = '1' then
692            PC := PC + 1;
693        end if;
694    end process;
695
696    process is
697    begin
698        wait on Out_Z;
699        if Out_Z = '1' then
700            PC := PC + 1;
701        end if;
702    end process;
703
704    process is
705    begin
706        wait on Out_Z;
707        if Out_Z = '1' then
708            PC := PC + 1;
709        end if;
710    end process;
711
712    process is
713    begin
714        wait on Out_Z;
715        if Out_Z = '1' then
716            PC := PC + 1;
717        end if;
718    end process;
719
720    process is
721    begin
722        wait on Out_Z;
723        if Out_Z = '1' then
724            PC := PC + 1;
725        end if;
726    end process;
727
728    process is
729    begin
730        wait on Out_Z;
731        if Out_Z = '1' then
732            PC := PC + 1;
733        end if;
734    end process;
735
736    process is
737    begin
738        wait on Out_Z;
739        if Out_Z = '1' then
740            PC := PC + 1;
741        end if;
742    end process;
743
744    process is
745    begin
746        wait on Out_Z;
747        if Out_Z = '1' then
748            PC := PC + 1;
749        end if;
750    end process;
751
752    process is
753    begin
754        wait on Out_Z;
755        if Out_Z = '1' then
756            PC := PC + 1;
757        end if;
758    end process;
759
760    process is
761    begin
762        wait on Out_Z;
763        if Out_Z = '1' then
764            PC := PC + 1;
765        end if;
766    end process;
767
768    process is
769    begin
770        wait on Out_Z;
771        if Out_Z = '1' then
772            PC := PC + 1;
773        end if;
774    end process;
775
776    process is
777    begin
778        wait on Out_Z;
779        if Out_Z = '1' then
780            PC := PC + 1;
781        end if;
782    end process;
783
784    process is
785    begin
786        wait on Out_Z;
787        if Out_Z = '1' then
788            PC := PC + 1;
789        end if;
790    end process;
791
792    process is
793    begin
794        wait on Out_Z;
795        if Out_Z = '1' then
796            PC := PC + 1;
797        end if;
798    end process;
799
800    process is
801    begin
802        wait on Out_Z;
803        if Out_Z = '1' then
804            PC := PC + 1;
805        end if;
806    end process;
807
808    process is
809    begin
810        wait on Out_Z;
811        if Out_Z = '1' then
812            PC := PC + 1;
813        end if;
814    end process;
815
816    process is
817    begin
818        wait on Out_Z;
819        if Out_Z = '1' then
820            PC := PC + 1;
821        end if;
822    end process;
823
824    process is
825    begin
826        wait on Out_Z;
827        if Out_Z = '1' then
828            PC := PC + 1;
829        end if;
830    end process;
831
832    process is
833    begin
834        wait on Out_Z;
835        if Out_Z = '1' then
836            PC := PC + 1;
837        end if;
838    end process;
839
840    process is
841    begin
842        wait on Out_Z;
843        if Out_Z = '1' then
844            PC := PC + 1;
845        end if;
846    end process;
847
848    process is
849    begin
850        wait on Out_Z;
851        if Out_Z = '1' then
852            PC := PC + 1;
853        end if;
854    end process;
855
856    process is
857    begin
858        wait on Out_Z;
859        if Out_Z = '1' then
860            PC := PC + 1;
861        end if;
862    end process;
863
864    process is
865    begin
866        wait on Out_Z;
867        if Out_Z = '1' then
868            PC := PC + 1;
869        end if;
870    end process;
871
872    process is
873    begin
874        wait on Out_Z;
875        if Out_Z = '1' then
876            PC := PC + 1;
877        end if;
878    end process;
879
880    process is
881    begin
882        wait on Out_Z;
883        if Out_Z = '1' then
884            PC := PC + 1;
885        end if;
886    end process;
887
888    process is
889    begin
890        wait on Out_Z;
891        if Out_Z = '1' then
892            PC := PC + 1;
893        end if;
894    end process;
895
896    process is
897    begin
898        wait on Out_Z;
899        if Out_Z = '1' then
900            PC := PC + 1;
901        end if;
902    end process;
903
904    process is
905    begin
906        wait on Out_Z;
907        if Out_Z = '1' then
908            PC := PC + 1;
909        end if;
910    end process;
911
912    process is
913    begin
914        wait on Out_Z;
915        if Out_Z = '1' then
916            PC := PC + 1;
917        end if;
918    end process;
919
920    process is
921    begin
922        wait on Out_Z;
923        if Out_Z = '1' then
924            PC := PC + 1;
925        end if;
926    end process;
927
928    process is
929    begin
930        wait on Out_Z;
931        if Out_Z = '1' then
932            PC := PC + 1;
933        end if;
934    end process;
935
936    process is
937    begin
938        wait on Out_Z;
939        if Out_Z = '1' then
940            PC := PC + 1;
941        end if;
942    end process;
943
944    process is
945    begin
946        wait on Out_Z;
947        if Out_Z = '1' then
948            PC := PC + 1;
949        end if;
950    end process;
951
952    process is
953    begin
954        wait on Out_Z;
955        if Out_Z = '1' then
956            PC := PC + 1;
957        end if;
958    end process;
959
960    process is
961    begin
962        wait on Out_Z;
963        if Out_Z = '1' then
964            PC := PC + 1;
965        end if;
966    end process;
967
968    process is
969    begin
970        wait on Out_Z;
971        if Out_Z = '1' then
972            PC := PC + 1;
973        end if;
974    end process;
975
976    process is
977    begin
978        wait on Out_Z;
979        if Out_Z = '1' then
980            PC := PC + 1;
981        end if;
982    end process;
983
984    process is
985    begin
986        wait on Out_Z;
987        if Out_Z = '1' then
988            PC := PC + 1;
989        end if;
990    end process;
991
992    process is
993    begin
994        wait on Out_Z;
995        if Out_Z = '1' then
996            PC := PC + 1;
997        end if;
998    end process;
999
1000 process is
1001 begin
1002     wait on Out_Z;
1003     if Out_Z = '1' then
1004         PC := PC + 1;
1005     end if;
1006 end process;
1007
1008 process is
1009 begin
1010     wait on Out_Z;
1011     if Out_Z = '1' then
1012         PC := PC + 1;
1013     end if;
1014 end process;
1015
1016 process is
1017 begin
1018     wait on Out_Z;
1019     if Out_Z = '1' then
1020         PC := PC + 1;
1021     end if;
1022 end process;
1023
1024 process is
1025 begin
1026     wait on Out_Z;
1027     if Out_Z = '1' then
1028         PC := PC + 1;
1029     end if;
1030 end process;
1031
1032 process is
1033 begin
1034     wait on Out_Z;
1035     if Out_Z = '1' then
1036         PC := PC + 1;
1037     end if;
1038 end process;
1039
1040 process is
1041 begin
1042     wait on Out_Z;
1043     if Out_Z = '1' then
1044         PC := PC + 1;
1045     end if;
1046 end process;
1047
1048 process is
1049 begin
1050     wait on Out_Z;
1051     if Out_Z = '1' then
1052         PC := PC + 1;
1053     end if;
1054 end process;
1055
1056 process is
1057 begin
1058     wait on Out_Z;
1059     if Out_Z = '1' then
1060         PC := PC + 1;
1061     end if;
1062 end process;
1063
1064 process is
1065 begin
1066     wait on Out_Z;
1067     if Out_Z = '1' then
1068         PC := PC + 1;
1069     end if;
1070 end process;
1071
1072 process is
1073 begin
1074     wait on Out_Z;
1075     if Out_Z = '1' then
1076         PC := PC + 1;
1077     end if;
1078 end process;
1079
1080 process is
1081 begin
1082     wait on Out_Z;
1083     if Out_Z = '1' then
1084         PC := PC + 1;
1085     end if;
1086 end process;
1087
1088 process is
1089 begin
1090     wait on Out_Z;
1091     if Out_Z = '1' then
1092         PC := PC + 1;
1093     end if;
1094 end process;
1095
1096 process is
1097 begin
1098     wait on Out_Z;
1099     if Out_Z = '1' then
1100         PC := PC + 1;
1101     end if;
1102 end process;
1103
1104 process is
1105 begin
1106     wait on Out_Z;
1107     if Out_Z = '1' then
1108         PC := PC + 1;
1109     end if;
1110 end process;
1111
1112 process is
1113 begin
1114     wait on Out_Z;
1115     if Out_Z = '1' then
1116         PC := PC + 1;
1117     end if;
1118 end process;
1119
1120 process is
1121 begin
1122     wait on Out_Z;
1123     if Out_Z = '1' then
1124         PC := PC + 1;
1125     end if;
1126 end process;
1127
1128 process is
1129 begin
1130     wait on Out_Z;
1131     if Out_Z = '1' then
1132         PC := PC + 1;
1133     end if;
1134 end process;
1135
1136 process is
1137 begin
1138     wait on Out_Z;
1139     if Out_Z = '1' then
1140         PC := PC + 1;
1141     end if;
1142 end process;
1143
1144 process is
1145 begin
1146     wait on Out_Z;
1147     if Out_Z = '1' then
1148         PC := PC + 1;
1149     end if;
1150 end process;
1151
1152 process is
1153 begin
1154     wait on Out_Z;
1155     if Out_Z = '1' then
1156         PC := PC + 1;
1157     end if;
1158 end process;
1159
1160 process is
1161 begin
1162     wait on Out_Z;
1163     if Out_Z = '1' then
1164         PC := PC + 1;
1165     end if;
1166 end process;
1167
1168 process is
1169 begin
1170     wait on Out_Z;
1171     if Out_Z = '1' then
1172         PC := PC + 1;
1173     end if;
1174 end process;
1175
1176 process is
1177 begin
1178     wait on Out_Z;
1179     if Out_Z = '1' then
1180         PC := PC + 1;
1181     end if;
1182 end process;
1183
1184 process is
1185 begin
1186     wait on Out_Z;
1187     if Out_Z = '1' then
1188         PC := PC + 1;
1189     end if;
1190 end process;
1191
1192 process is
1193 begin
1194     wait on Out_Z;
1195     if Out_Z = '1' then
1196         PC := PC + 1;
1197     end if;
1198 end process;
1199
1200 process is
1201 begin
1202     wait on Out_Z;
1203     if Out_Z = '1' then
1204         PC := PC + 1;
1205     end if;
1206 end process;
1207
1208 process is
1209 begin
1210     wait on Out_Z;
1211     if Out_Z = '1' then
1212         PC := PC + 1;
1213     end if;
1214 end process;
1215
1216 process is
1217 begin
1218     wait on Out_Z;
1219     if Out_Z = '1' then
1220         PC := PC + 1;
1221     end if;
1222 end process;
1223
1224 process is
1225 begin
1226     wait on Out_Z;
1227     if Out_Z = '1' then
1228         PC := PC + 1;
1229     end if;
1230 end process;
1231
1232 process is
1233 begin
1234     wait on Out_Z;
1235     if Out_Z = '1' then
1236         PC := PC + 1;
1237     end if;
1238 end process;
1239
1240 process is
1241 begin
1242     wait on Out_Z;
1243     if Out_Z = '1' then
1244         PC := PC + 1;
1245     end if;
1246 end process;
1247
1248 process is
1249 begin
1250     wait on Out_Z;
1251     if Out_Z = '1' then
1252         PC := PC + 1;
1253     end if;
1254 end process;
1255
1256 process is
1257 begin
1258     wait on Out_Z;
1259     if Out_Z = '1' then
1260         PC := PC + 1;
1261     end if;
1262 end process;
1263
1264 process is
1265 begin
1266     wait on Out_Z;
1267     if Out_Z = '1' then
1268         PC := PC + 1;
1269     end if;
1270 end process;
1271
1272 process is
1273 begin
1274     wait on Out_Z;
1275     if Out_Z = '1' then
1276         PC := PC + 1;
1277     end if;
1278 end process;
1279
1280 process is
1281 begin
1282     wait on Out_Z;
1283     if Out_Z = '1' then
1284         PC := PC + 1;
1285     end if;
1286 end process;
1287
1288 process is
1289 begin
1290     wait on Out_Z;
1291     if Out_Z = '1' then
1292         PC := PC + 1;
1293     end if;
1294 end process;
1295
1296 process is
1297 begin
1298     wait on Out_Z;
1299     if Out_Z = '1' then
1300         PC := PC + 1;
1301     end if;
1302 end process;
1303
1304 process is
1305 begin
1306     wait on Out_Z;
1307     if Out_Z = '1' then
1308         PC := PC + 1;
1309     end if;
1310 end process;
1311
1312 process is
1313 begin
1314     wait on Out_Z;
1315     if Out_Z = '1' then
1316         PC := PC + 1;
1317     end if;
1318 end process;
1319
1320 process is
1321 begin
1322     wait on Out_Z;
1323     if Out_Z = '1' then
1324         PC := PC + 1;
1325     end if;
1326 end process;
1327
1328 process is
1329 begin
1330     wait on Out_Z;
1331     if Out_Z = '1' then
1332         PC := PC + 1;
1333     end if;
1334 end process;
1335
1336 process is
1337 begin
1338     wait on Out_Z;
1339     if Out_Z = '1' then
1340         PC := PC + 1;
1341     end if;
1342 end process;
1343
1344 process is
1345 begin
1346     wait on Out_Z;
1347     if Out_Z = '1' then
1348         PC := PC + 1;
1349     end if;
1350 end process;
1351
1352 process is
1353 begin
1354     wait on
```

COE608: Computer Org. & Architecture > Lab 6 > The Complete CPU (Overall Project)

```
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)
```

COE608: Computer Org. & Architecture > Lab 6 > The Complete CPU (Overall Project)

```
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
148 signal LZE_out_A_Mux   : std_logic_vector(31 downto 0);
149 signal LZE_out_B_Mux   : std_logic_vector(31 downto 0);
150 signal RED_out_Data_Mem : unsigned(7 downto 0);
151 signal A_Mux_out        : std_logic_vector(31 downto 0);
152 signal B_Mux_out        : std_logic_vector(31 downto 0);
153 signal reg_A_out        : std_logic_vector(31 downto 0);
154 signal reg_B_out        : std_logic_vector(31 downto 0);
155 signal reg_Mux_out      : std_logic_vector(31 downto 0);
156 signal data_mem_out    : std_logic_vector(31 downto 0);
157 signal UZB_IM_MUX1_out  : std_logic_vector(31 downto 0);
158 signal IM_MUX1_out      : std_logic_vector(31 downto 0);
159 signal LZE_IM_MUX2_out  : std_logic_vector(31 downto 0);
160 signal IM_MUX2_out      : std_logic_vector(31 downto 0);
161 signal ALU_out          : std_logic_vector(31 downto 0);
162 signal zero_flag        : std_logic;
163 signal carry_flag       : std_logic;
164 signal temp             : std_logic_vector(30 downto 0) := (others => '0');
165 signal out_pc_sig       : std_logic_vector(31 downto 0);
166
167 begin
168     IR:      register32 port map(
169         . .
170         data_bus_s,
171         Ld_IR,
172         ClrIR,
173         Clk,
174         IR_OUT
175     );
176
177     LZE_PC:   LZE port map(
178         IR_OUT,
179         LZE_out_PC
180     );
181
182     PC0:      PC port map(
183         CLR_PPC,
184         Clk,
185         ld_PPC,
186         INC_PPC,
187         LZE_out_PPC,
188         -- ADDR_OUT
189         out_Pc_sig
190     );
191
192     LZE_A_Mux: LZE port map(
193         IR_OUT,
194         LZE_out_A_Mux
195     );
196
197     A_Mux0:    mux2to1 port map(
198         A_MUX,
199         data_bus_s,
200         LZE_out_A_Mux,
201         A_Mux_out
202     );
203
204     Reg_A:     register32 port map(
205         A_Mux_out,
206         Ld_A,
207         Clr_A,
208         Clk,
209         reg_A_out
210     );
```

```
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: mux4tol 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

File Edit View Insert Tools Window Help

control.vhd

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

entity control is
  port(
    cik, mclk : IN STD_LOGIC;
    enable : IN STD_LOGIC;
    status, status1 : IN STD_LOGIC;
    en : IN STD_LOGIC;
    A_Mux, B_Mux : OUT STD_LOGIC;
    IM_MUX1, REG_Mux : OUT STD_LOGIC;
    ALU_op : OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
    inc_FC, id_FC : OUT STD_LOGIC;
    inc_Z, id_Z : OUT STD_LOGIC;
    id_IR : OUT STD_LOGIC;
    cir_A, cir_B, cir_C, cir_Z : OUT STD_LOGIC;
    B_MUX : OUT STD_LOGIC;
    T : OUT STD_LOGIC_VECTOR(5 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, status, status1, enable, Instruction_sig, Instruction_sig2)
  BEGIN
    if enable = '1' then
      if present_state = state_0 then
        DATA_Mux <= "000"; -- Fetch Address of Next Instruction
        cir_IR <= '0';
        id_IR <= '0';
        inc_FC <= '0';
        cir_A <= '0';
        id_A <= '0';
        id_B <= '0';
        cir_B <= '0';
        cir_C <= '0';
        id_C <= '0';
        id_Z <= '0';
        cir_Z <= '0';
        id_Z <= '0';
        en <= '0';
        wen <= '0';

        cir_Z <= '0';
        id_Z <= '0';
        wen <= '0';

        cir_IR <= '0'; -- INCREMENT PC COUNTER
        id_IR <= '0';
        id_FC <= '1';
        inc_FC <= '1';
        cir_A <= '0';
        id_A <= '0';
        id_B <= '0';
        cir_B <= '0';
        cir_C <= '0';
        id_C <= '0';
        id_Z <= '0';
        cir_Z <= '0';
        id_Z <= '0';
        en <= '1';
        wen <= '1';

        if Instruction.sig = "0010" then -- STA
          cir_IR <= '0';
          id_IR <= '0';
          id_FC <= '1';
          inc_FC <= '1';
          cir_A <= '0';
          id_A <= '0';
          id_B <= '0';
          cir_B <= '0';
          cir_C <= '0';
          id_C <= '0';

          cir_Z <= '0';
          id_Z <= '0';
          wen <= '1';

          REG_Mux <= '1';
          DATA_Mux <= "000";
          en <= '1';
          wen <= '1';

        elsif Instruction.sig = "0011" then -- STB
          cir_IR <= '0';
          id_IR <= '0';
          id_FC <= '1';
          inc_FC <= '1';
          cir_A <= '0';
          id_A <= '0';
          id_B <= '0';
          cir_B <= '0';
          cir_C <= '0';
          id_C <= '0';

          cir_Z <= '0';
          id_Z <= '0';
          wen <= '1';

          en <= '1';
          wen <= '1';

        elsif Instruction.sig = "0101" then -- LDA
          cir_IR <= '0';
          id_IR <= '0';
          id_FC <= '1';
          inc_FC <= '1';
          cir_A <= '0';
          id_A <= '0';
          id_B <= '0';
          cir_B <= '0';
          cir_C <= '0';
          id_C <= '0';

          cir_Z <= '0';
          id_Z <= '0';
          B_MUX <= '0';
          DATA_Mux <= "001";
          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
            cir_IR <= '0';

          end if;
        end if;
      end if;
    end process;
  end if;
end control;
```

COE608: Computer Org. & Architecture > Lab 6 > The Complete CPU (Overall Project)

```

146      id_IR <= '0';
147      id_PC <= '1';
148      id_FC <= '1';
149      cir_IR <= '0';
150      id_A <= '0';
151      id_B <= '0';
152      cir_B <= '0';
153      cir_C <= '0';
154      id_C <= '0';
155      cir_Z <= '0';
156      id_Z <= '0';
157
158      elsif instruction_sig = "0110" then -- BEQ
159          cir_IR <= '0';
160          id_A <= '1';
161          id_PC <= '1';
162          inc_FC <= '0';
163          cir_A <= '1';
164          id_A <= '0';
165          id_B <= '0';
166          cir_B <= '0';
167          id_C <= '0';
168          cir_C <= '0';
169          id_Z <= '0';
170          id_Z <= '0';
171
172      elsif instruction_sig = "1000" then -- BNE
173          cir_IR <= '0';
174          id_A <= '0';
175          id_B <= '1';
176          id_FC <= '1';
177          cir_A <= '0';
178          id_A <= '1';
179          id_B <= '0';
180          cir_B <= '0';
181          cir_C <= '0';
182          id_C <= '0';
183          cir_Z <= '0';
184          id_Z <= '0';
185
186      elsif instruction_sig = "1001" then -- LDA
187          cir_IR <= '0';
188          id_A <= '0';
189          id_FC <= '0';
190          inc_FC <= '0';
191          cir_A <= '1';
192          id_A <= '1';
193          id_B <= '0';
194          cir_B <= '0';
195          id_C <= '0';
196          id_D <= '0';
197          id_E <= '0';
198          id_F <= '0';
199          id_Mux <= '0';
200          DATA_MUX <= "011";
201          en <= '1';
202          wen <= '0';
203
204      elsif instruction_sig = "1010" then -- LDH
205          cir_IR <= '0';
206          id_A <= '0';
207          id_B <= '1';
208          inc_FC <= '0';
209          cir_A <= '0';
210          id_A <= '1';
211          id_B <= '0';
212          cir_B <= '0';
213          id_C <= '0';
214          id_D <= '0';
215          cir_Z <= '0';
216          id_Z <= '0';
217          B_Mux <= '0';
218          DATA_MUX <= "01";
219          en <= '1';
220          wen <= '0';
221
222      elsif instruction_sig = "0010" then -- STA
223          cir_IR <= '0';
224          id_A <= '0';
225          id_B <= '1';
226          inc_FC <= '0';
227          cir_A <= '0';
228          id_A <= '1';
229          id_B <= '0';
230          cir_B <= '0';
231          id_C <= '0';
232          id_D <= '0';
233          id_E <= '0';
234          id_F <= '0';
235          REG_Mux <= '0';
236          DATA_MUX <= "000";
237          en <= '1';
238          wen <= '1';
239
240      elsif instruction_sig = "0011" then -- STB
241          cir_IR <= '0';
242
243          id_IR <= '0';
244          id_FC <= '0';
245          inc_FC <= '0';
246          cir_A <= '0';
247          id_A <= '0';
248          id_B <= '0';
249          cir_B <= '0';
250          id_C <= '0';
251          id_D <= '0';
252          id_E <= '0';
253          REG_Mux <= '1';
254          DATA_MUX <= "000";
255          en <= '1';
256          wen <= '1';
257
258      elsif instruction_sig = "0000" then -- LDAI
259          cir_IR <= '0';
260          id_A <= '0';
261          id_FC <= '0';
262          inc_FC <= '0';
263          cir_A <= '0';
264          id_A <= '1';
265          id_B <= '0';
266          cir_B <= '0';
267          id_C <= '0';
268          id_D <= '0';
269          cir_Z <= '0';
270          id_E <= '0';
271          A_Mux <= '1';
272
273      elsif instruction_sig = "0001" then -- LDHI
274          cir_IR <= '0';
275          id_A <= '0';
276          id_B <= '1';
277          inc_FC <= '0';
278          cir_A <= '0';
279          id_A <= '1';
280          id_B <= '0';
281          cir_B <= '0';
282          id_C <= '0';
283          id_D <= '0';
284          cir_Z <= '0';
285          id_E <= '0';
286          B_Mux <= '1';
287
288      elsif instruction_sig = "0100" then -- LUI
289          cir_IR <= '0';

```

COE608: Computer Org. & Architecture > Lab 6 > The Complete CPU (Overall Project)

```

280      id_IR <= '0';
281      id_PC <= '0';
282      inc_PC <= '0';
283      cir_A <= '0';
284      cir_B <= '0';
285      id_B <= '0';
286      cir_B <= '1';
287      cir_C <= '0';
288      id_C <= '0';
289      cir_Z <= '0';
290      id_Z <= '0';
291      ALU_op <= "001";
292      A_MUX <= '0';
293      DATA_MUX <= "000";
294      im_MUX1 <= '0';
295      im_MUX2 <= '0';
296
297      elsif Instruction_sig2 = "01111001" then -- ANDI
298
299          cir_IR <= '0';
300          id_IR <= '0';
301          id_PC <= '0';
302          inc_PC <= '0';
303          cir_A <= '0';
304          id_A <= '1';
305          cir_B <= '0';
306          cir_B <= '0';
307          cir_C <= '0';
308          id_B <= '1';
309          cir_Z <= '0';
310          id_Z <= '1';
311          cir_A <= '0';
312          id_A <= '1';
313          cir_B <= '0';
314          cir_B <= '0';
315          cir_C <= '0';
316          id_B <= '1';
317          cir_Z <= '0';
318          id_Z <= '1';
319          ALU_op <= "0000";
320          A_MUX <= '0';
321          DATA_MUX <= "10";
322          im_MUX1 <= '1';
323          im_MUX2 <= '01';
324
325
326      elsif Instruction_sig2 = "01111110" then -- DECA
327
328          cir_IR <= '0';
329          id_IR <= '0';
330          inc_PC <= '0';
331          cir_A <= '0';
332          id_A <= '0';
333          cir_B <= '0';
334          cir_C <= '0';
335          id_B <= '0';
336          cir_Z <= '0';
337          id_Z <= '1';
338
339          ALU_op <= "110";
340          A_MUX <= '0';
341          DATA_MUX <= "10";
342          im_MUX1 <= '0';
343          im_MUX2 <= '10';
344
345
346      elsif Instruction_sig2 = "01111000" then -- ADD
347
348          cir_IR <= '0';
349          id_IR <= '0';
350          id_PC <= '0';
351          inc_PC <= '0';
352          cir_A <= '0';
353          id_A <= '1';
354          cir_B <= '0';
355          cir_C <= '0';
356          id_B <= '0';
357          cir_Z <= '0';
358          id_Z <= '1';
359          ALU_op <= "110";
360          A_MUX <= '0';
361          DATA_MUX <= "10";
362          im_MUX1 <= '0';
363          im_MUX2 <= '10';
364
365
366      elsif Instruction.sig2 = "01110010" then -- SUB
367
368          cir_IR <= '0';
369          id_IR <= '0';
370          id_PC <= '0';
371          inc_PC <= '0';
372          cir_A <= '0';
373          id_A <= '1';
374          cir_B <= '0';
375          cir_C <= '0';
376          id_B <= '1';
377          cir_Z <= '0';
378          id_Z <= '1';
379          ALU_op <= "110";
380          A_MUX <= '0';
381          DATA_MUX <= "10";
382          im_MUX1 <= '0';
383          im_MUX2 <= '00';
384
385
386      elsif Instruction.sig2 = "01110011" then -- INCA
387
388          cir_IR <= '0';
389          id_IR <= '0';
390          id_PC <= '0';
391          inc_PC <= '0';
392          cir_A <= '0';
393          id_A <= '1';
394          cir_B <= '0';
395          cir_C <= '0';
396          id_B <= '0';
397          cir_Z <= '0';
398          id_Z <= '1';
399          im_MUX1 <= '0';
400
401          ALU_op <= "010";
402          A_MUX <= '0';
403          DATA_MUX <= "00";
404          im_MUX2 <= '0';
405
406
407      elsif Instruction.sig2 = "01111011" then -- AND
408
409          cir_IR <= '0';
410          id_IR <= '0';
411          id_PC <= '0';
412          inc_PC <= '0';
413          cir_A <= '0';
414          id_A <= '1';
415          cir_B <= '0';
416          cir_C <= '0';
417          id_B <= '0';
418          cir_Z <= '0';
419          id_Z <= '1';
420
421          ALU_op <= "000";
422          A_MUX <= '0';
423          DATA_MUX <= "00";
424          im_MUX1 <= '0';
425          im_MUX2 <= '00';
426
427
428      elsif Instruction.sig2 = "01110001" then -- ADDI
429
430          cir_IR <= '0';
431          id_IR <= '0';
432          id_PC <= '0';
433          inc_PC <= '0';
434          cir_A <= '0';
435          id_A <= '1';
436          cir_B <= '0';
437          cir_C <= '0';
438          id_B <= '0';
439          cir_Z <= '0';
440          id_Z <= '1';
441          cir_A <= '0';
442          id_A <= '1';
443          cir_B <= '0';
444          cir_C <= '0';
445          id_B <= '0';
446          cir_Z <= '0';
447          id_Z <= '1';
448          ALU_op <= "010";
449
450
451

```

COE608: Computer Org. & Architecture > Lab 6 > The Complete CPU (Overall Project)

```

436      ... ...
437      A_Mux <= '0';
438      DATA_Mux <= "10";
439      id_A <= '0';
440      in_MUX2 <= "01";
441
442      elsif Instruction_sig2 = "01111101" then -- ORI
443          clr_IR <= '0';
444          id_IR <= '0';
445          id_A <= '1';
446          inc_PC <= '0';
447          clr_A <= '0';
448          id_A <= '0';
449          id_B <= '0';
450          clr_B <= '0';
451          id_B <= '1';
452          inc_PC <= '1';
453          A_Mux <= '01';
454          DATA_Mux <= "10";
455          id_C <= '0';
456          in_MUX2 <= "01";
457
458      elsif Instruction_sig2 = "01110100" then -- ROL
459          clr_IR <= '0';
460          id_IR <= '0';
461          id_A <= '1';
462          inc_PC <= '0';
463          clr_A <= '0';
464          id_A <= '1';
465          id_B <= '0';
466          clr_B <= '0';
467          id_B <= '1';
468          id_C <= '1';
469          clr_Z <= '0';
470          id_C <= '0';
471          A_Mux <= "100";
472          DATA_Mux <= '01';
473          id_MUX1 <= '0';
474          id_MUX2 <= '0';
475
476      elsif Instruction_sig2 = "01111111" then -- ROR
477          clr_IR <= '0';
478          id_IR <= '0';
479          id_A <= '1';
480          inc_PC <= '0';
481          clr_A <= '0';
482
483          ... ...
484          id_B <= '0';
485          clr_B <= '0';
486          id_C <= '0';
487          id_C <= '1';
488          clr_Z <= '0';
489          id_Z <= '1';
490          A_Mux <= "101";
491          DATA_Mux <= "10";
492          id_MUX1 <= '1';
493          id_MUX2 <= '0';
494
495      elsif Instruction_sig2 = "01110101" then -- CLR_A
496          clr_IR <= '0';
497          id_IR <= '0';
498          id_FC <= '0';
499          id_A <= '1';
500          clr_A <= '1';
501          id_A <= '0';
502          id_B <= '0';
503          clr_B <= '0';
504          id_B <= '1';
505          id_Z <= '0';
506          id_Z <= '0';
507
508      elsif Instruction_sig2 = "01110110" then -- CLR_B
509          clr_IR <= '0';
510          id_IR <= '0';
511          id_FC <= '0';
512          inc_PC <= '0';
513          clr_A <= '0';
514          id_A <= '0';
515          id_B <= '0';
516          clr_B <= '1';
517          id_B <= '0';
518          id_C <= '0';
519          id_C <= '0';
520          id_Z <= '0';
521
522      elsif Instruction_sig2 = "01110111" then -- CLR_C
523          clr_IR <= '0';
524          id_IR <= '0';
525          id_FC <= '0';
526          id_A <= '0';
527          clr_A <= '0';
528          id_A <= '0';
529          id_B <= '0';
530
531          ... ...
532          id_B <= '0';
533          id_C <= '0';
534          id_D <= '0';
535
536      elsif Instruction_sig2 = "01110111" then -- CLR_Z
537          clr_IR <= '0';
538          id_IR <= '0';
539          id_FC <= '0';
540          inc_PC <= '0';
541          id_A <= '0';
542          id_A <= '1';
543          id_B <= '0';
544          id_B <= '0';
545          id_C <= '0';
546          id_C <= '1';
547          id_D <= '0';
548          id_D <= '1';
549          id_Z <= '0';
550
551      elsif Instruction_sig2 = "01111010" then -- TSTZ
552          if (status < '1') then
553              clr_IR <= '0'; -- INCREMENT PC COUNTER
554              id_IR <= '0';
555              id_FC <= '0';
556              inc_PC <= '1';
557              clr_A <= '0';
558              id_A <= '0';
559              id_B <= '0';
560              id_B <= '1';
561              id_C <= '0';
562              id_C <= '1';
563              id_D <= '0';
564              id_D <= '0';
565          end if;
566
567      elsif Instruction_sig2 = "01111100" then -- TSTC
568          if (status < '1') then
569              clr_IR <= '0'; -- INCREMENT PC COUNTER
570              id_IR <= '0';
571              id_FC <= '0';
572              inc_PC <= '1';
573              clr_A <= '0';
574              id_A <= '0';
575              id_B <= '0';
576              id_C <= '0';
577              id_C <= '1';

```

```

578      clr_Z <= '0';
579      ld_Z <= '0';
580    end if;
581  end if; -- For state 2 Ops
582  end if; -- For Enable
583 END process;
584
585 ----- STATE MACHINE -----
586 PROCESS (clk, enable)
587 begin
588   if enable = '1' then
589     if rising_edge (clk) then
590       if present_state = state_0 then present_state <= state_1;
591     elsif present_state = state_1 then present_state <= state_2;
592     else present_state <= state_0;
593   end if;
594   end if;
595   else present_state <= state_0;
596   end if;
597 END process;
598
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 cpul 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 cpul;

ARCHITECTURE description OF cpul 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, 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 cpul.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 cpul
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;
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 : cpul
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

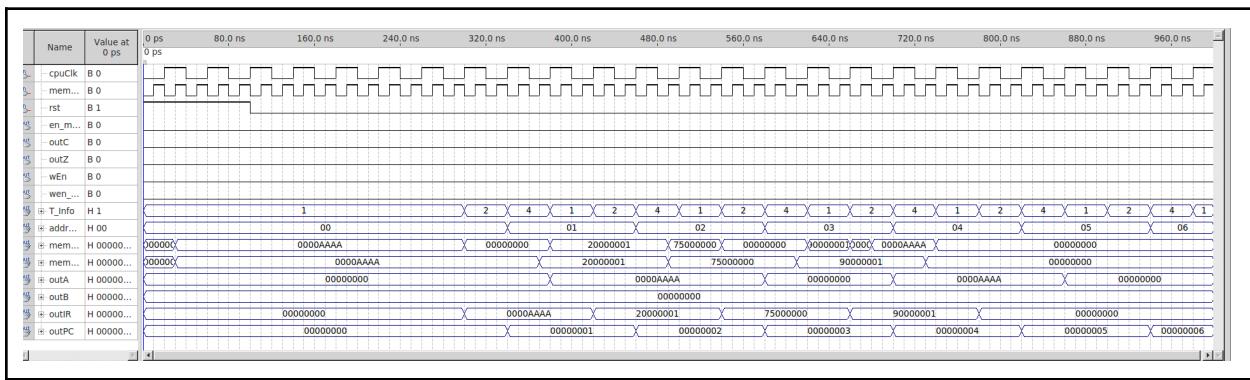


Figure 2.0: Waveform of `cpu_test_sim.vwf`

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.

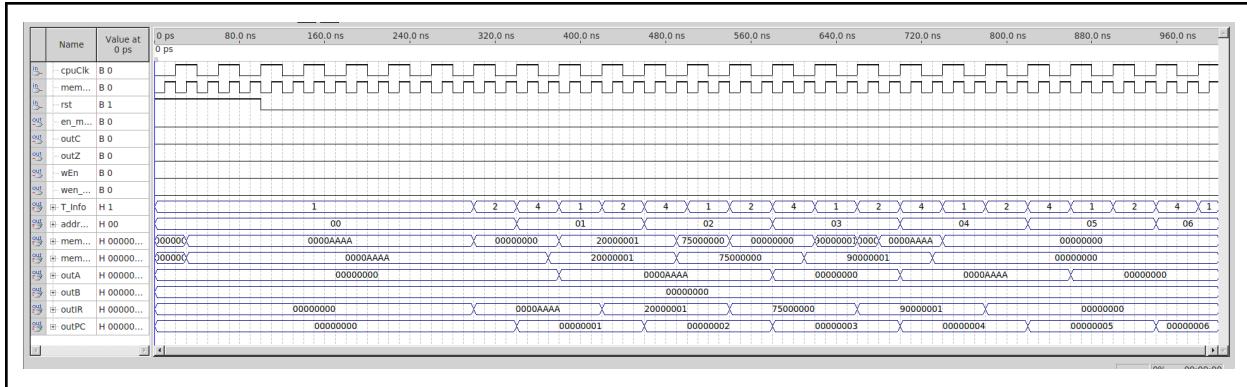


Figure 2.1.1: Waveform for LDAI, STA, CLRA, LDA

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.

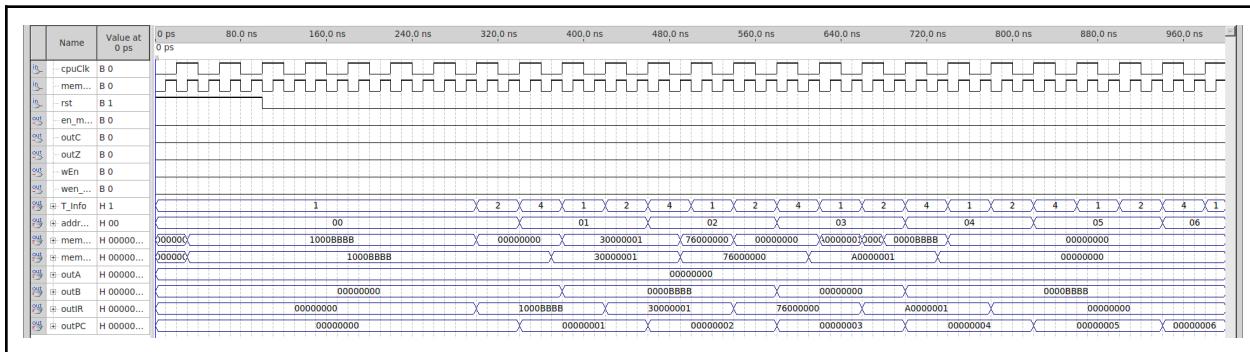


Figure 2.2.1: Waveform ofLDBI, STB, CLRB and LDB implementation

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	00000000	00000000	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.

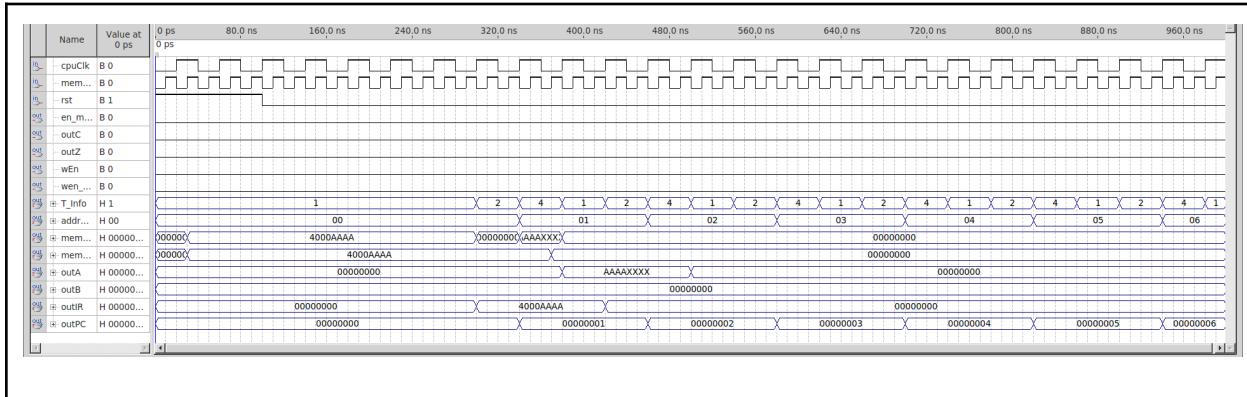


Figure 2.3.1: Waveform of LUI implementation

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	00000000	00000000	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.

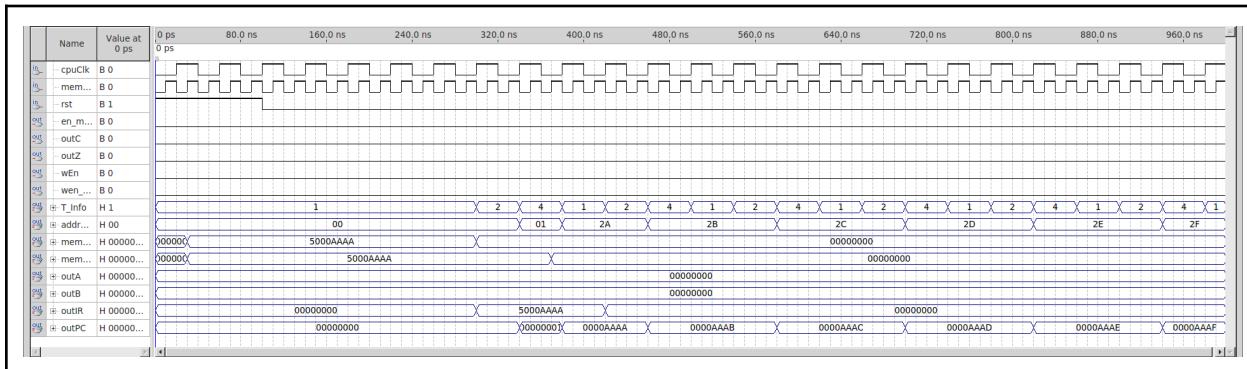


Figure 2.4.1: Waveform of JMP implementation in CPU

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	00000000	00000000	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.

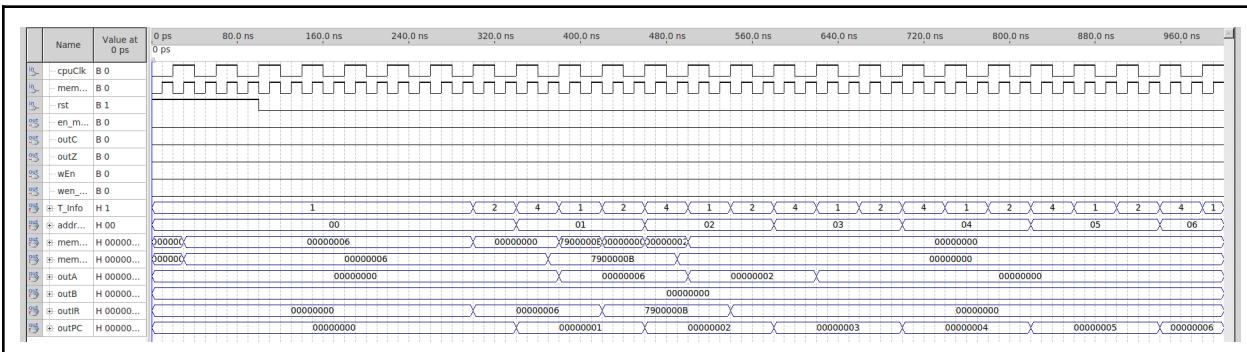


Figure 2.5.1: Waveform of ANDI CPU implementation

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	00000000	00000000	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.

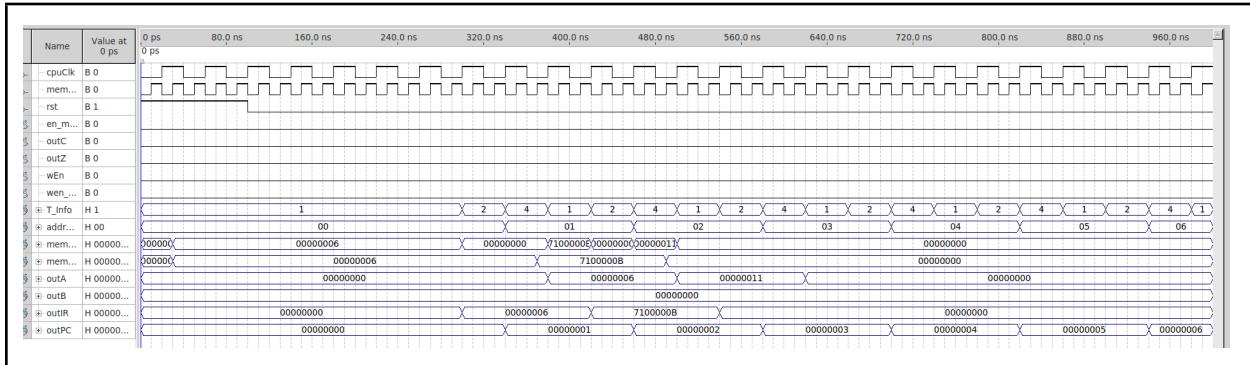


Figure 2.6.1: Waveform of ADDI Add Immediate

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	00000000	00000000	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.

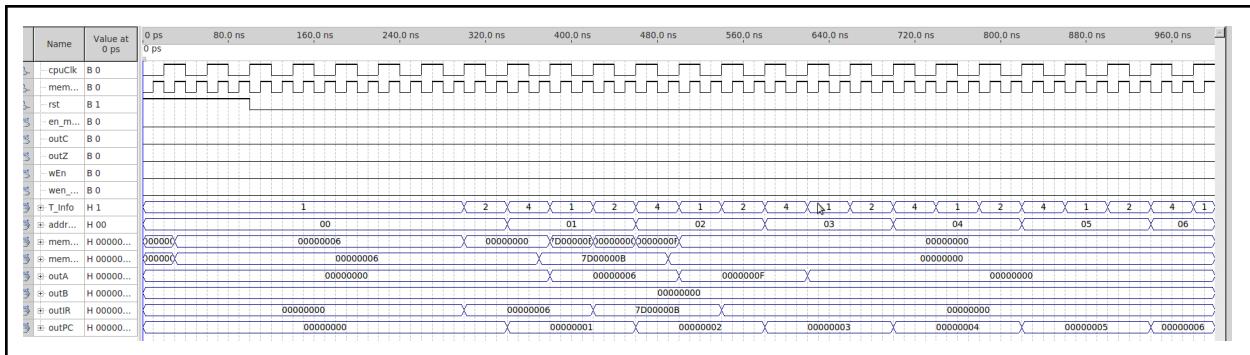


Figure 2.7.1: Waveform of ORI OR Immediate CPU Implementation

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	00000000	00000000	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.

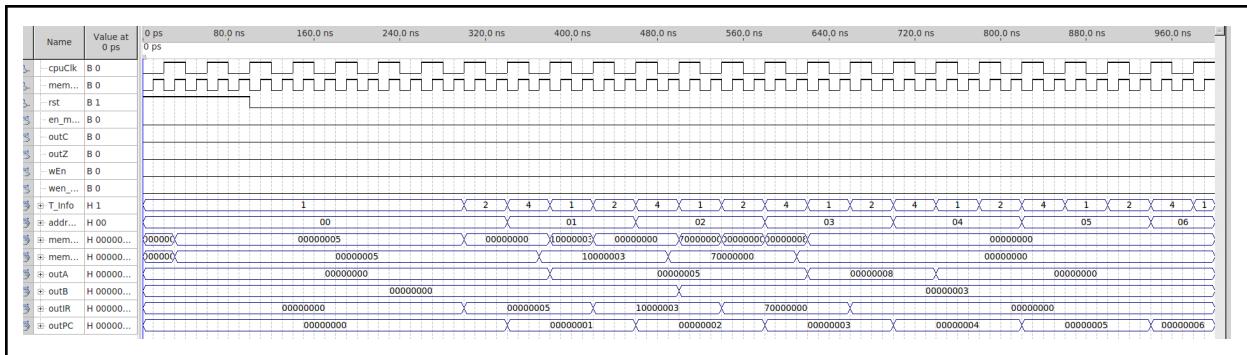


Figure 2.8.1: Waveform of ADD

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	00000000	00000000	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.

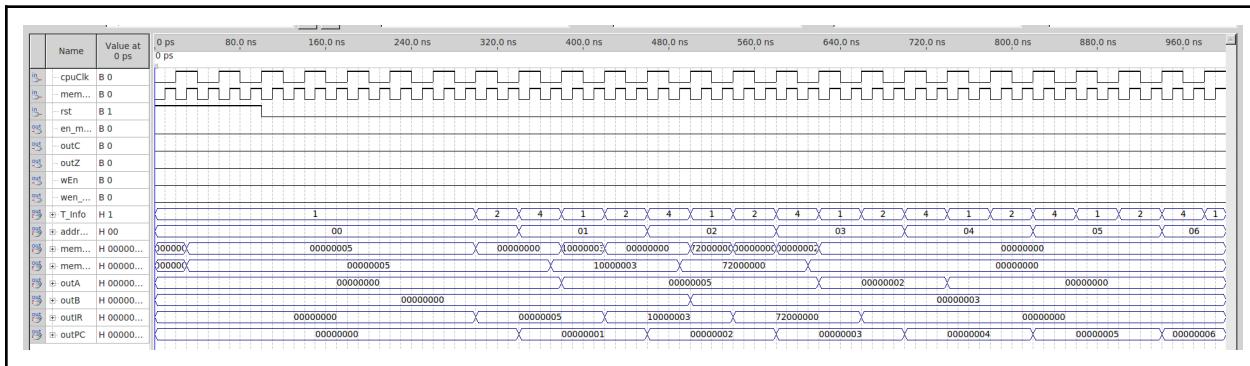


Figure 2.9.1: Waveform of SUB implementation on CPU

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.

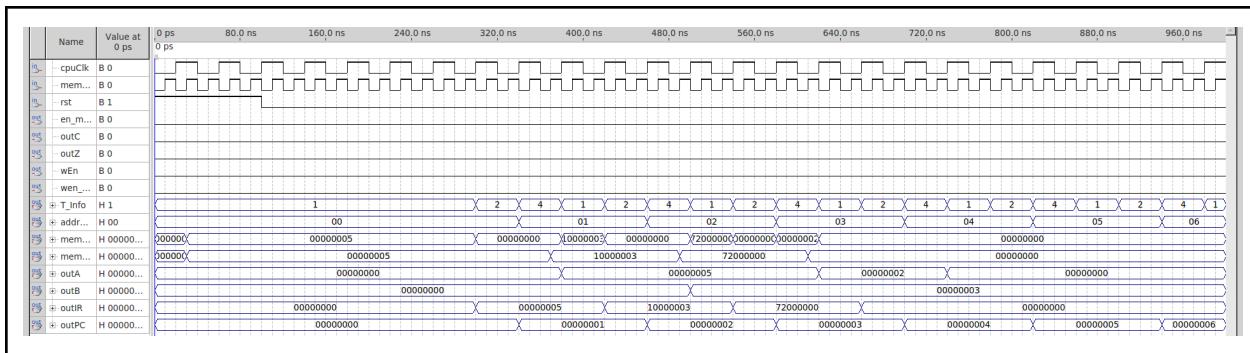


Figure 2.10.1: Waveform of DECA implementation on CPU

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.

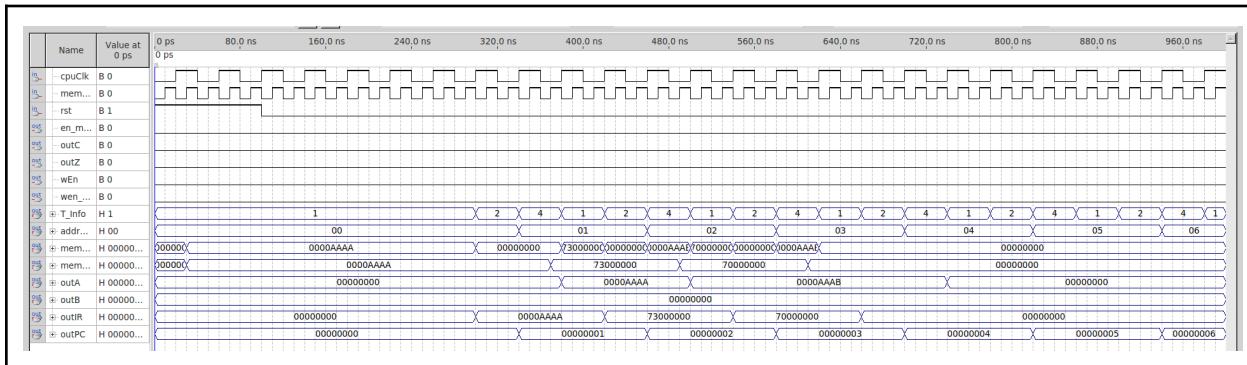


Figure 2.11.1: Waveform of Increment A (INCA)

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	00000000	00000000	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.

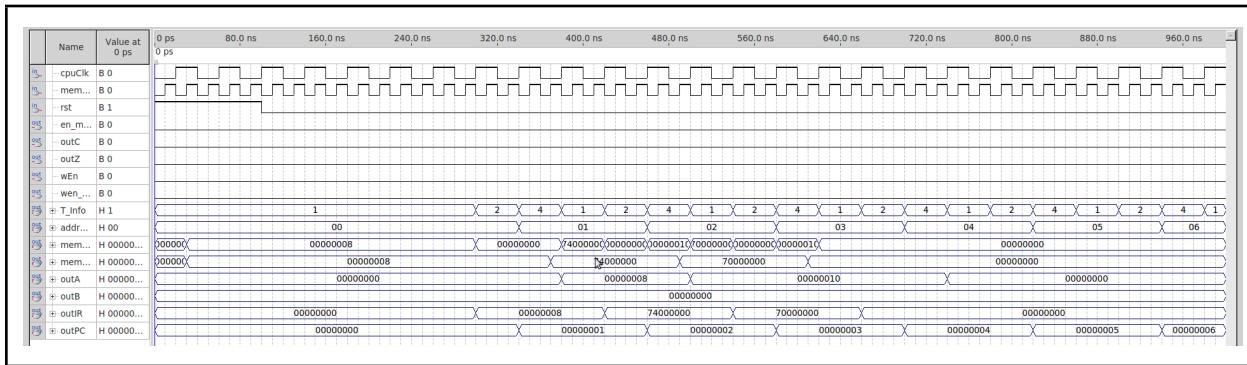


Figure 2.12.1: Waveform of Rotate Left ROL

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 00000001 to 00000000, 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	00000000	00000000	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.

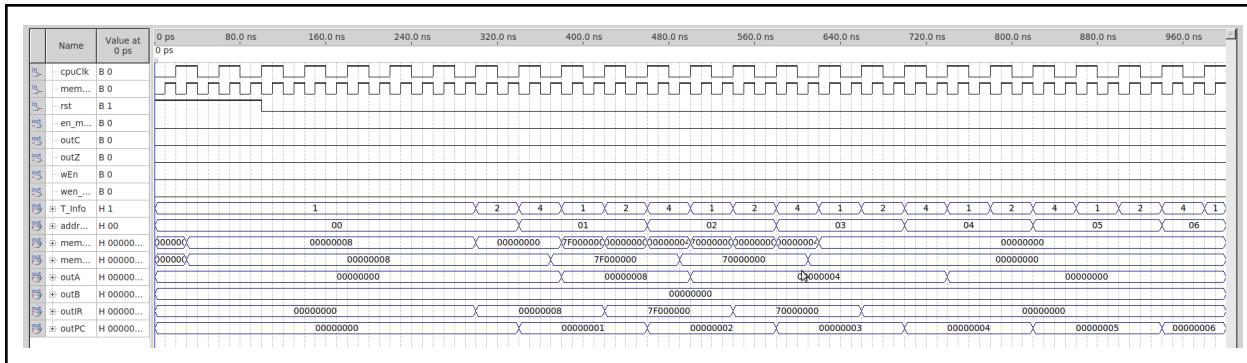


Figure 2.13.1: Waveform of Rotate Right (ROR) Implemented on the CPU

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.

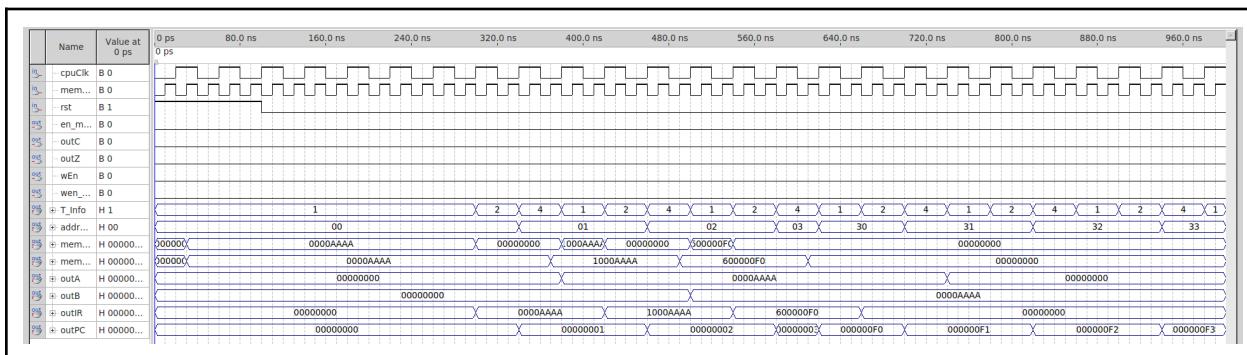


Figure 2.14.1: Waveform of Branch If Equal (BEQ)

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 600000F0, 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.

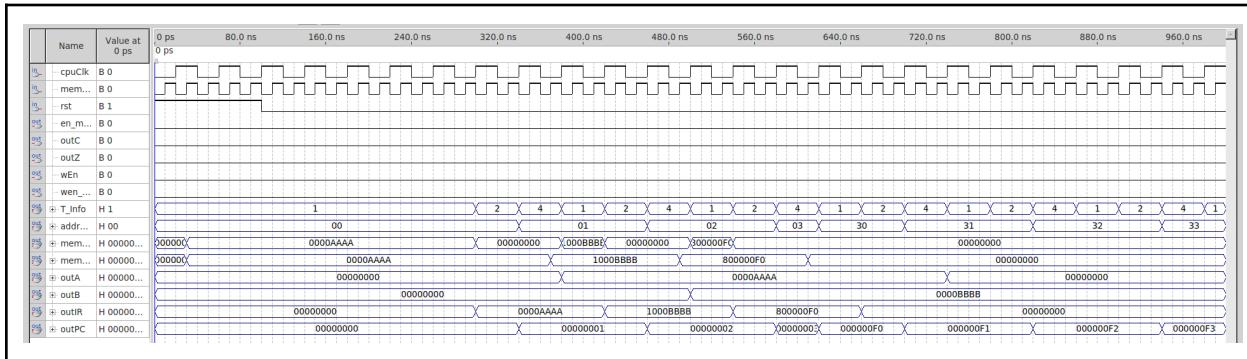


Figure 2.15.1: Waveform of Branch If Not Equal (BNE)

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.