# COE608 Computer Organization and Architectures
# Winter 2017
# Lab 6: The Complete CPU (Overall Project)

**Due Date: Lab6 Part I & II -Week 12 (During the Lab Session)**
**Bonus -Week 13**

## 1. Overview

In this final lab project, a complete CPU will be implemented whose main components datapath and control have been designed and implemented in the previous labs 4b and 5. Students are to combine the control unit and data-path with a reset circuit (more on this below). When complete, the over-all design will be able to implement the features described in the *CPU specification document*. Students are encouraged to consult this specification document while proceeding to test the CPU. The instruction memory unit (VHDL) and overall CPU testing block diagrams files to complete this lab are provided as follows: 1) The files and specifics for testing and setting up the CPU for simulation can be found in a document located in …/courses/coe608/labs/lab6/* and 2) for bonus marks involving CPU hardware implementation and emulation, the specifications and documentation can be found in …/courses/coe608/labs/bonus/*. The rest of this lab presents the reset circuit needed for the CPU.

## 2. Part I - CPU Reset Circuitry

In order for the CPU developed here to work properly it must incorporate a reset circuit. The block diagram of the reset circuit is illustrated in Figure 1.

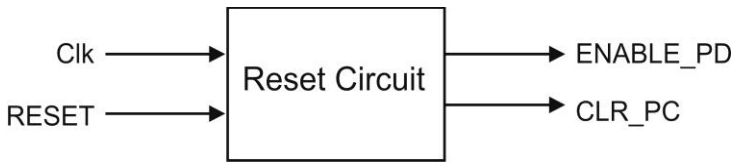


*Figure 1: Reset Circuit*

The reset circuit works as explained here. When RESET signal goes high, ENABLE_PD goes low that forces the control unit into state T0 and CLR_PC goes high, which clears the Program Counter. We know that the CPU program starts in memory at location 0x00000000.

When RESET goes low, ENABLE_PD & CLR_PC remains low & high respectively for 4 clock cycles. This allows the data surrounding the CPU to stabilize before its operation begins. The reset circuit is required to keep track (count) of the three clock cycles (T0, T1, and T2). This reset circuit can either be implemented asynchronously or synchronously. The synchronous waveform is shown in Figure 2.

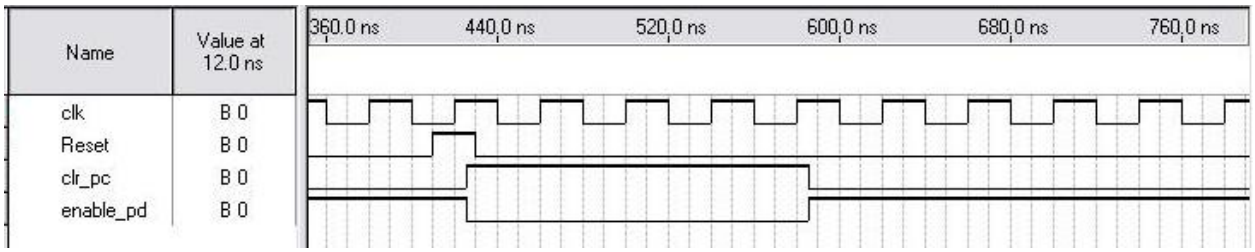


*Figure 2: Reset Circuit Operation*

# 3. VHDL Implementation

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

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

ARCHITECTURE description OF reset_circuit IS BEGIN
-- you fill in what goes here.
END description;
```

Students are free to implement the ARCHITECTURE section however they see fit.

# 4. Part I - What to Hand In

Students must submit the following to obtain full marks for Lab 6, Part I:

- A hard-copy listing of your VHDL source code implementation.
- A hard-copy printout of the timing simulation results for the reset circuit.

The lab instructor will quiz you on both Part I (reset circuit) and Part II (final CPU) for the Lab 6 demo.

# 5. Part II - The Complete CPU System

Once the reset circuit is implemented, all the CPU sub-systems are complete and the CPU can be assembled. The final CPU will consist of one instance of the data-path, the control unit, and the reset circuit. Interconnecting them appropriately is up to the students, however supporting files for VHDL interconnection and setup may be found in the course directory …/courses/coe608/labs/lab6/ in the cpu1 module. To help better understand the expectations and functionality of the final CPU, students are encouraged to refer to the **CPU Testing** document available in the course directory and website. This document will help you to:

- Generate the system's instruction memory unit (MegaCore RAM block (.mif) implementation)
- Assemble a top level working file (with reset circuit, instruction memory, a datapath, and control unit)
- Include and map other supporting files
- Simulate and demo your working CPU for Lab 6
- And optionally emulate the CPU for the Lab 6 Bonus

Students are advised to also refer to the CPU Specification document to ensure that all operations and specifications have been fulfilled by their final CPU.

# 6. Part II - What to Hand in

To obtain full marks for the complete CPU project (i.e. Lab6), students must demonstrate the correct operation of CPU circuit through simulation. This means that students have to demonstrate the timing simulation results that show the CPU correctly loading ALL the instructions and data, performing addition, load upper immediate, etc.

In addition to this, students must submit the VHDL code for their CPU, as well as timing simulation results for the complete CPU. To properly simulate the CPU operation, the *CPU_testing* document should be consulted, and Memory Module files provided should be used. Your lab supervisor will quiz you during the demo for both Parts I and II.

# 7. Bonus Project

To obtain bonus marks, students are required to demonstrate the operation of the processor through emulation on the DE2 boards found in the laboratory. To properly implement the CPU, the **CPU_testing** document should be consulted and system memory, seven-segment, display-unit, decoder, and all the other VHDL file provided in the following course directory should be used.
…/courses/coe608/labs/bonus/*

The following problems are provided as a bonus. Solving them will result in additional marks being added to your course mark.

**Problem 1:**
The processor developed throughout this course includes various branch and jump operations used for conditional statements. Using the built in mnemonics/ instruction set, find a way to implement the following:

```
IF(a == b){
        branch1();
}
else if(a > b){
        branch2();
}
else{
        continue;
}
end IF;
```

**Problem 2:**
Implement the following code using your CPU and its instruction set:

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

To obtain the bonus, the assembly code for both of the problems above must be submitted, and the programs in question must be demonstrated on the DE2 boards in the laboratory.

Ryerson University

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

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

| | |
|---|---|
| **Instructor:** | Dr. Vadim Geurkov |

| | |
|---|---|
| *Assignment/Lab Number:* | Lab 6 |
| *Assignment/Lab Title:* | The Complete CPU |

| | |
|---|---|
| *Submission Date:* | March 31 2025 |
| *Due Date:* | April 4 2025 |

| Student LAST Name | Student FIRST Name | Student Number | Section | Signature* |
|---|---|---|---|---|
| Avella Ordonez | Daniel | 501212214 | 03 | DAO |
| | | | | |
| | | | | |

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

Table of Contents

## Introduction

The goal of Lab 6 was to build and simulate a fully functional Semi-RISC CPU using VHDL. This project involved combining all previously developed modules—datapath, control unit, and reset circuit—into a cohesive system capable of executing a basic instruction set. Through simulation and testing using a series of Memory Initialization Files (.mif), each CPU instruction was validated in Quartus II using waveform outputs. This report presents the finalized VHDL modules, the reset mechanism, and simulation results for each instruction. The .mif values were used to populate instruction memory, and the functionality was confirmed through waveform comparison.

---

## 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  reset : in std_logic;
9  clk : in std_logic;
10 enable_PD : out std_logic :='1';
11 clr_PC : out std_logic
12 );
13 end reset_circuit;
14
15 architecture Behavior of reset_circuit is
16 type clkNum is (clk0, clk1, clk2, clk3);
17 signal present_clk: clkNum;
18 begin
19    process(clk)begin
20       if rising_edge(clk) then
21          if reset = '1' then
22             clr_PC <= '1';
23             enable_PD <= '0';
24             present_clk <= clk0;
25          elsif present_clk <= clk0 then
26             present_clk <= clk1;
27          elsif present_clk <= clk1 then
28             present_clk <= clk2;
29          elsif present_clk <= clk2 then
30             present_clk <= clk3;
31          elsif present_clk <= clk3 then
32             clr_PC <= '0';
33             enable_PD <= '1';
34          end if;
35          end if;
36          end process;
37       end Behavior;
```

Figure 1: reset_circuit.vhd

*This figure shows the VHDL code for the synchronous reset circuit. When the Reset signal is asserted high, the Enable_PD signal is driven low and Clr_PC is driven high, resetting the program counter. After a few clock cycles, the CPU becomes enabled, allowing stable execution.*

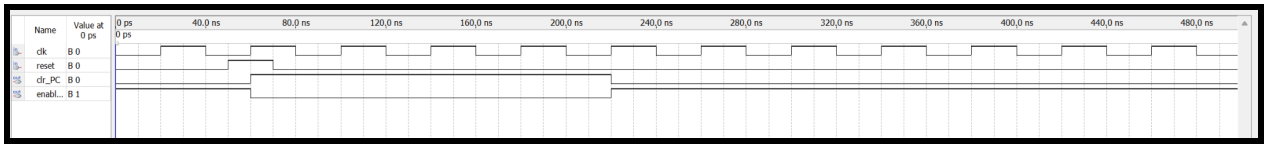


Figure 2: Reset Circuit Simulation Waveform

*The waveform confirms proper operation: Enable_PD is low and Clr_PC is high during the first 4 cycles after reset is asserted, and the CPU begins execution once the reset phase completes.*

---

## Core VHDL Modules

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity Control_New is
5  port(
6  clk, mclk : in std_logic;
7  enable : in std_logic;
8  statusC, statusZ : in std_logic;
9  INST : in std_logic_vector(31 downto 0);
10 A_Mux, B_Mux : out std_logic;
11 IM_MUX1, REG_MUX : out std_logic;
12 IM_MUX2, DATA_Mux : out std_logic_vector(1 downto 0);
13 ALU_op : out std_logic_vector(2 downto 0);
14 inc_PC, ld_PC : out std_logic;
15 clr_IR, ld_IR : out std_logic;
16 clr_A, clr_B, clr_C, clr_Z : out std_logic;
17 ld_A, ld_B, ld_C, ld_Z : out std_logic;
18 T : out std_logic_vector(2 downto 0);
19 wen, en : out std_logic
20 );
21 end Control_New;
22
23 architecture description of Control_New is
24     type STATETYPE is (state_0, state_1, state_2);
25     signal present_state : STATETYPE;
26     signal Instruction_sig : std_logic_vector(3 downto 0);
27     signal Instruction_sig2 : std_logic_vector(7 downto 0);
28 begin
29     Instruction_sig <= INST(31 downto 28);
30     Instruction_sig2 <= INST(31 downto 24);
```

Figure 3: Control_New.vhd

*This file contains the finite state machine (FSM) logic that drives control signals for the datapath. It interprets instructions stored in IR, managing transitions through the T0–T3 states and issuing the necessary control signals for memory, register loading, and ALU operations.*

---

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

ENTITY cpu1 IS
PORT (
clk : IN STD_LOGIC;
mem_clk : IN STD_LOGIC;
rst : IN STD_LOGIC;
dataIn : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
dataOut : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
addrOut : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
dOutA : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
dOutB : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
dOutC : OUT STD_LOGIC;
dOutZ : OUT STD_LOGIC;
dOutIR : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
dOutPC : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
wEn : OUT STD_LOGIC;
outT : OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
wen_mem : OUT STD_LOGIC;
en_mem : OUT STD_LOGIC
);
END cpu1;

ARCHITECTURE description OF cpu1 IS
COMPONENT data_path IS
PORT (
    Clk, mClk : IN STD_LOGIC;  -- Clock Signals
    WEN, EN : IN STD_LOGIC;    -- Memory Signals
    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;
    Clr_PC , Ld_PC : IN STD_LOGIC;     -- □ MATCHES data_path ENTITY
    Clr_IR , Ld_IR : IN STD_LOGIC;     -- □ MATCHES data_path ENTITY
    Inc_PC : IN STD_LOGIC;
    ADDR_OUT : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
    DATA_IN : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
    DATA_BUS, MEM_OUT, MEM_IN : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
    MEM_ADDR : OUT UNSIGNED(7 DOWNTO 0);  -- □ MATCH your data_path type
    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);
```

```
  COMPONENT cpu1
     PORT(
         clk      : in std_logic;
         mem_clk  : in std_logic;
         rst      : in std_logic;
         dataIn   : in std_logic_vector(31 downto 0);
         dataOut  : out std_logic_vector(31 downto 0);
         addrOut  : out std_logic_vector(31 downto 0);
         wEn : out std_logic;
         dOutA, dOutB : out std_logic_vector(31 downto 0);
         dOutC, dOutZ : out std_logic;
         dOutIR : out std_logic_vector(31 downto 0);
         dOutPC : out std_logic_vector(31 downto 0);
         outT : out std_logic_vector(2 downto 0);
         wen_mem, en_mem : out std_logic);
  END COMPONENT;

  signal cpu_to_mem: std_logic_vector(31 downto 0);
  signal mem_to_cpu: std_logic_vector(31 downto 0);
  signal add_from_cpu: std_logic_vector(31 downto 0);
  signal wen_from_cpu: std_logic;

BEGIN
 -- Component instantiations.
  main_memory : system_memory
     PORT MAP(
         address => add_from_cpu(5 downto 0),
         clock => memClk,
         data => cpu_to_mem,
         wren => wen_from_cpu,
         q => mem_to_cpu
      );
  main_processor : cpu1
     PORT MAP(
      clk => cpuClk,
      mem_clk => memClk,
      rst => rst,
      dataIn => mem_to_cpu,
      dataOut => cpu_to_mem,
      addrOut => add_from_cpu,
      wEn => wen_from_cpu,
      dOutA => outA,
      dOutB => outB,
      dOutC => outC,
      dOutZ => outZ,
      dOutIR => outIR,
```

```
      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 4: cpu1.vhd

*This file instantiates and wires together the datapath, control unit, and reset circuit. It maps internal signals and interfaces with the testbench and instruction memory, acting as the main processor entity.*

```vhdl
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;
```

Figure 5: CPU_TEST_Sim.vhd

*This testbench-style top-level file connects cpu1 and the system_memory. It provides external clocks and data ports to simulate the system.*

```
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 6: system_memory.vhd

*This memory unit connects to cpu1 and uses .mif files to preload instructions for simulation. Each test modifies this memory with specific opcodes to verify CPU behavior.*

---

```
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); -- Input data (32 bits)
    B : OUT STD_LOGIC_VECTOR(31 DOWNTO 0) -- Output (A + 4)
);
END add;

ARCHITECTURE Behavior OF add IS
BEGIN
    B <= A + 1; -- Add 4 to the input A and output the result to B
END Behavior;
```

Figure 7: add.vhd (Updated)

*Modified to add by 1, this version helps test increment operations and is used in simulations involving INCA, ADDI, and other arithmetic instructions.*

---

## Instruction Simulations and MIF Files

Each of the following instructions was tested by updating the .mif file with corresponding opcodes and running functional simulation. The results were observed using Quartus II's waveform viewer.

### LDAI, STA, CLRA, LDA

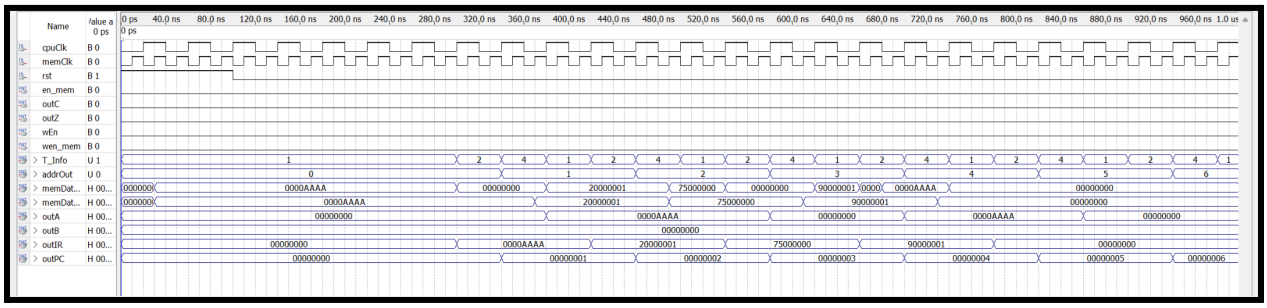


Figure 8: Waveform Simulation

*Confirms LDAI loads immediate into register A, STA stores it into memory, CLRA clears A, and LDA reloads the stored value.*

```
WIDTH=32;
DEPTH=64;

ADDRESS_RADIX=UNS;
DATA_RADIX=HEX;

CONTENT BEGIN
    0   : 0000AAAA;
    1   : 20000001;
    2   : 75000000;
    3   : 90000001;
    [4..63] : 00000000;
END;
```

Figure 9: MIF Explanation

*The instruction memory includes four key opcodes at addresses 0 through 3. These drive register and memory interaction patterns.*

## LUI

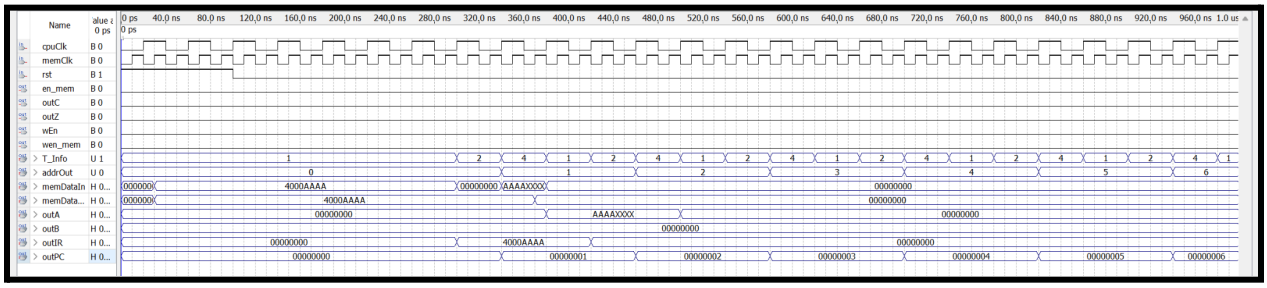


Figure 10: Waveform Simulation

*LUI loads an upper immediate value into register A. The waveform shows a correct load into A without affecting the lower bits.*

```
WIDTH=32;
DEPTH=64;

ADDRESS_RADIX=UNS;
DATA_RADIX=HEX;

CONTENT BEGIN
    0   : 4000AAAA;  -- LUI instruction
    [1..63] : 00000000;
END;
```

Figure 11: MIF Explanation

*The opcode at address 0 sets the upper 16 bits of A, validating LUI functionality.*

---

## LDBI, STB, CLRB, LDB

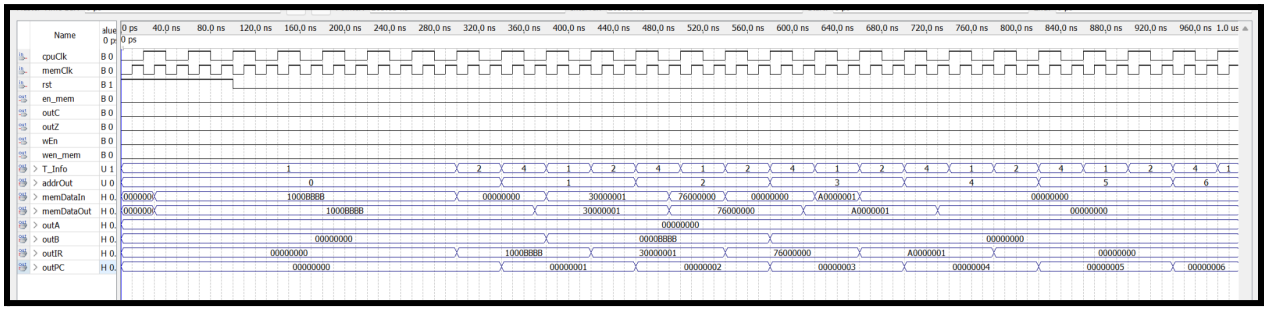


Figure 12: Waveform Simulation

*Immediate value is loaded into B, stored to memory, cleared, and then reloaded from memory.*

```
WIDTH=32;
DEPTH=64;

ADDRESS_RADIX=UNS;
DATA_RADIX=HEX;

CONTENT BEGIN
    0   : 1000BBBB;  -- LDBI instruction
    1   : 30000001;  -- STB instruction
    2   : 76000000;  -- CLRB instruction
    3   : A0000001;  -- LDB instruction
    [4..63] : 00000000;
END;
```

Figure 13: MIF Explanation

*Memory addresses 0–3 are loaded with opcodes representing the four instructions above.*

---

**JMP**

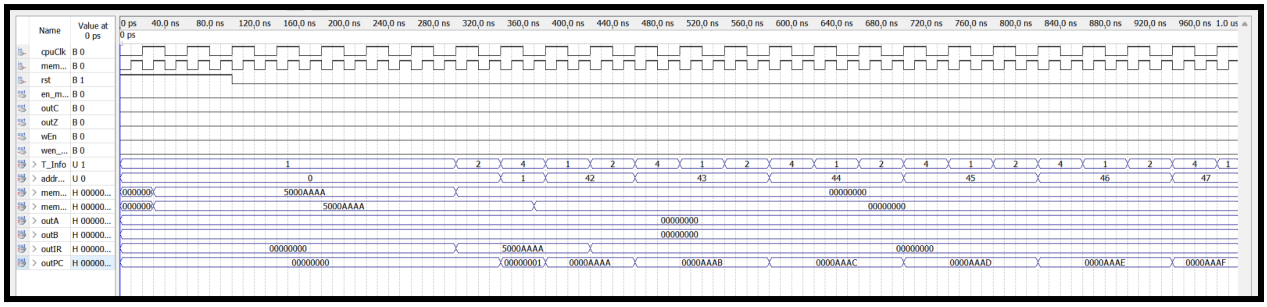


Figure 14: Waveform Simulation

*Confirms the Program Counter jumps to a new instruction address when JMP is executed.*

```
WIDTH=32;
DEPTH=64;

ADDRESS_RADIX=UNS;
DATA_RADIX=HEX;

CONTENT BEGIN
    0   : 5000AAAA;  -- JMP instruction
    [1..63] : 00000000;
END;
```

Figure 15: MIF Explanation

*Memory address 0 contains a JMP opcode that modifies the PC directly.*

---

## ANDI

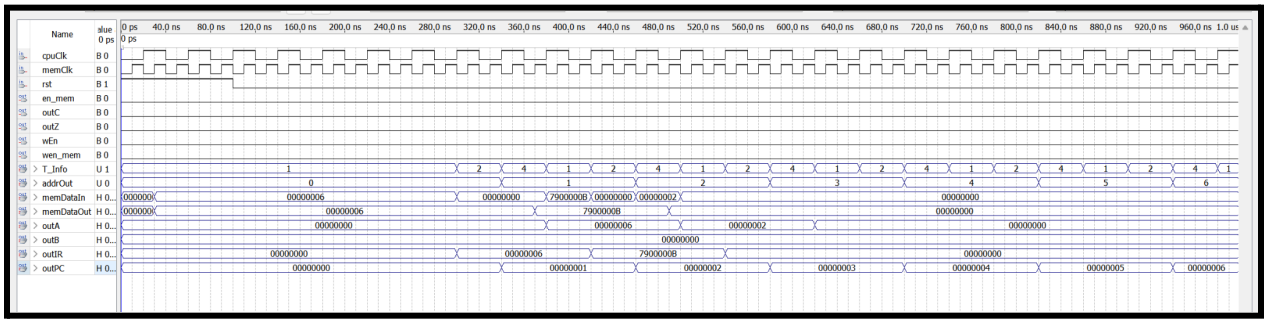


Figure 16: Waveform Simulation

*ANDI performs a logical AND between A and immediate, storing result in C. The waveform shows changes in C with the Zero flag set appropriately.*

```
WIDTH=32;
DEPTH=64;

ADDRESS_RADIX=UNS;
DATA_RADIX=HEX;

CONTENT BEGIN
    0   : 00000006;  -- Load initial value (e.g., CLRA or a base op)
    1   : 7900000B;  -- ANDI operation with immediate value 0x0000000B
    [2..63] : 00000000;
END;
```

Figure 17: MIF Explanation

*Memory includes immediate load and ANDI opcode with bitwise result testing.*

---

## ADDI

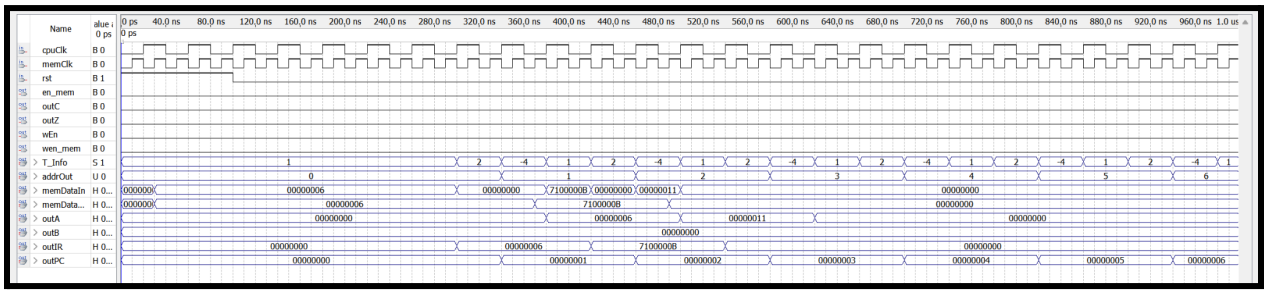


Figure 18: Waveform Simulation

*ADDI performs immediate addition and updates the result in C. The waveform shows carry/zero status and updated output.*

```
WIDTH=32;
DEPTH=64;

ADDRESS_RADIX=UNS;
DATA_RADIX=HEX;

CONTENT BEGIN
    0   : 00000006;  -- Initialize (e.g., CLRA or similar setup)
    1   : 7100000B;  -- ADDI operation with immediate value 0x0000000B
    [2..63] : 00000000;
END;
```

Figure 19: MIF Explanation

*Test verifies signed/unsigned immediate handling through two opcodes.*

---

**ORI**

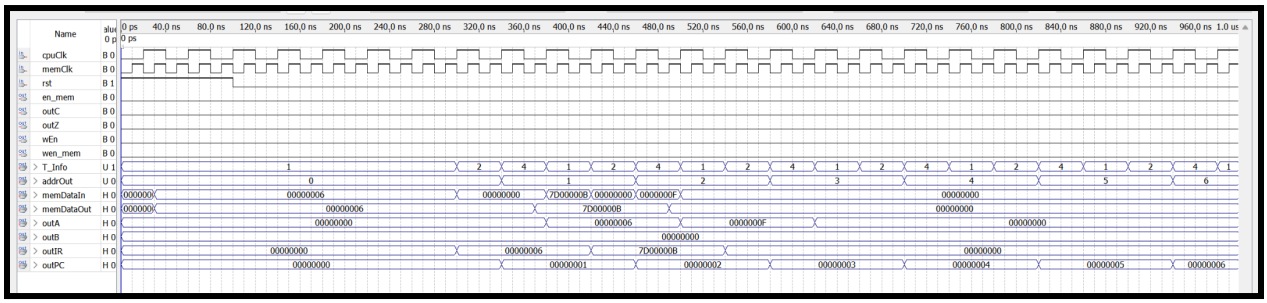


Figure 20: Waveform Simulation

*Performs bitwise OR with an immediate value. Waveform shows OR result populating C.*

```
WIDTH=32;
DEPTH=64;

ADDRESS_RADIX=UNS;
DATA_RADIX=HEX;

CONTENT BEGIN
    0   : 00000006;  -- Possibly CLRA or setup
    1   : 7D00000B;  -- ORI operation with immediate value 0x0000000B
    [2..63] : 00000000;
END;
```

Figure 21: MIF Explanation

*Memory holds immediate and ORI instruction opcodes.*

---

## ADD

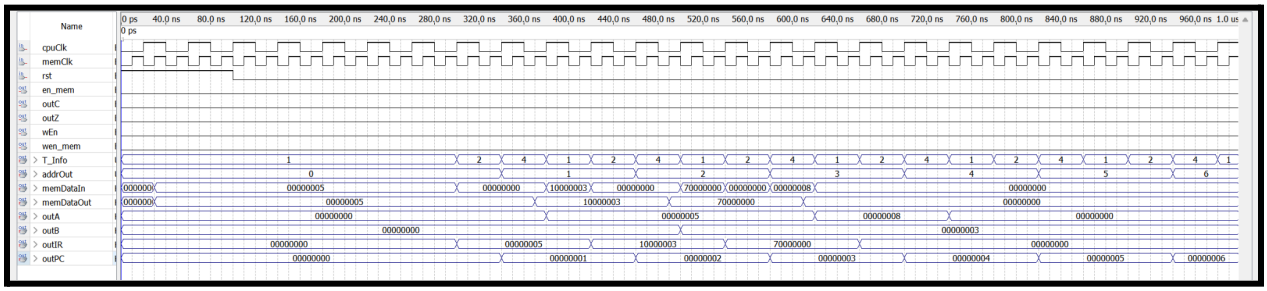


Figure 22: Waveform Simulation

*Standard addition between registers A and B. Result is shown on output C.*

```
WIDTH=32;
DEPTH=64;

ADDRESS_RADIX=UNS;
DATA_RADIX=HEX;

CONTENT BEGIN
    0   : 00000005;  -- CLRA or load immediate value into register A
    1   : 10000003;  -- LDBI instruction to load value into register B
    2   : 70000000;  -- ADD instruction
    [3..63] : 00000000;
END;
```

Figure 23: MIF Explanation

*Tested with preload of values in A and B, with ALU ADD opcode issued.*

---

## SUB

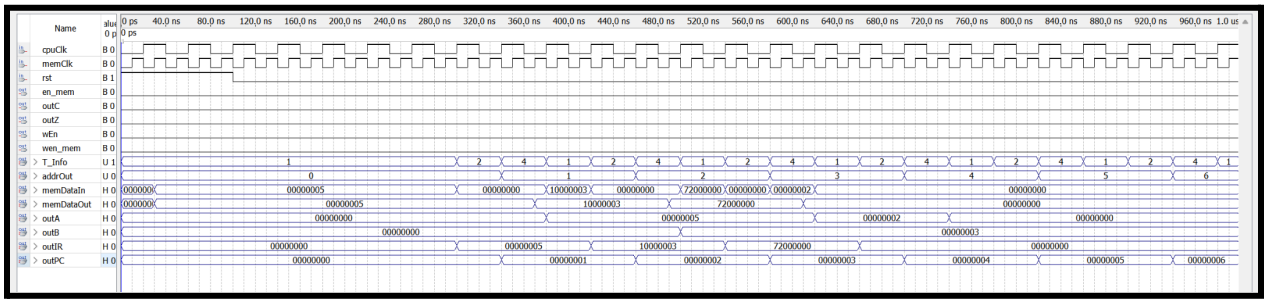


Figure 24: Waveform Simulation

*Subtracts B from A and shows result in C with corresponding zero/carry status.*

```
WIDTH=32;
DEPTH=64;

ADDRESS_RADIX=UNS;
DATA_RADIX=HEX;

CONTENT BEGIN
    0   : 00000005;  -- CLRA or load immediate into A
    1   : 10000003;  -- LDBI to load value into B
    2   : 72000000;  -- SUB instruction
    [3..63] : 00000000;
END;
```

Figure 25: MIF Explanation

*Program loads values and performs a subtract via ALU.*

---

## DECA

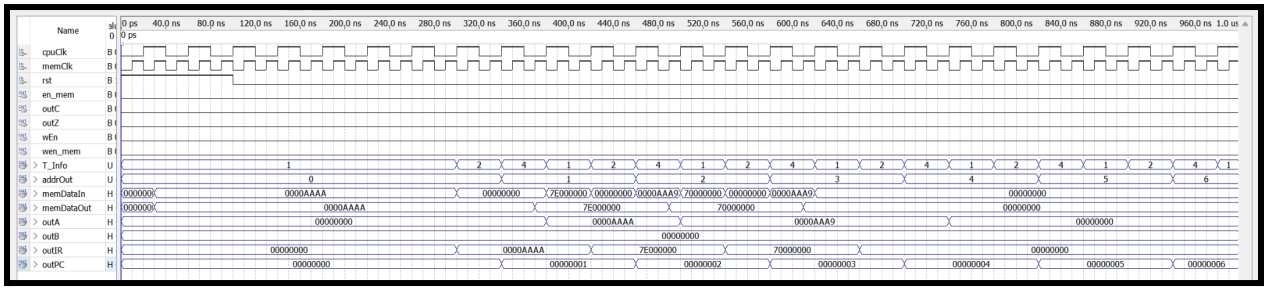


Figure 26: Waveform Simulation

*Decrements A and shows updated result.*

```
WIDTH=32;
DEPTH=64;

ADDRESS_RADIX=UNS;
DATA_RADIX=HEX;

CONTENT BEGIN
    0   : 0000AAAA;  -- Load immediate value into register A
    1   : 7E000000;  -- DECA instruction
    2   : 70000000;  -- ADD instruction to verify result (can be viewed as NOP here)
    [3..63] : 00000000;
END;
```

Figure 27: MIF Explanation

*Instruction sequence loads value, decrements, and stores.*

---

## INCA

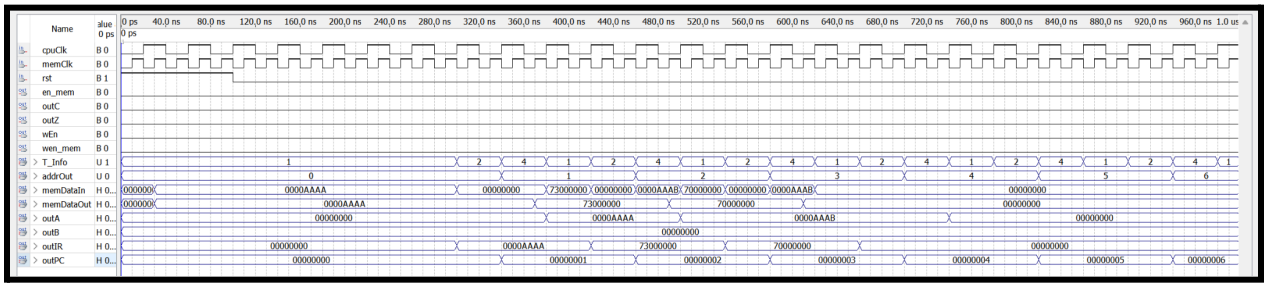


Figure 28: Waveform Simulation

*Increments A. ALU output is verified along with zero flag.*

```
WIDTH=32;
DEPTH=64;

ADDRESS_RADIX=UNS;
DATA_RADIX=HEX;

CONTENT BEGIN
    0   : 0000AAAA;  -- Load immediate value into register A
    1   : 73000000;  -- INCA instruction
    2   : 70000000;  -- ADD instruction (used here to observe changes or as NOP)
    [3..63] : 00000000;
END;
```

Figure 29: MIF Explanation

*Value is loaded and incremented through INCA opcode.*

---

## ROL

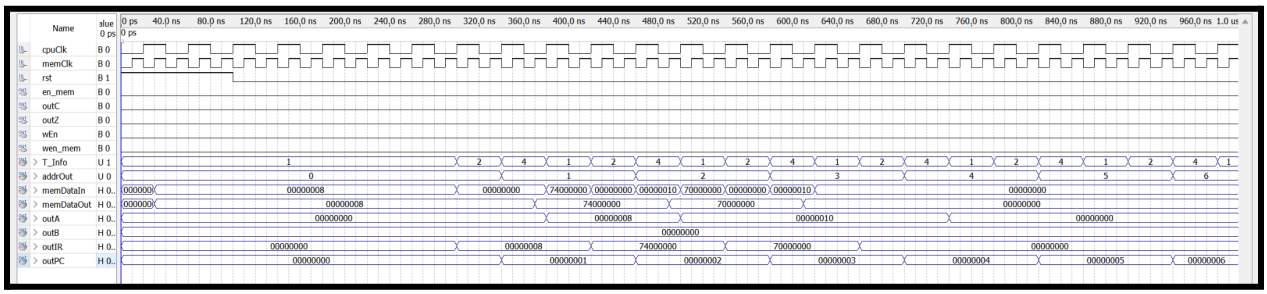


Figure 30: Waveform Simulation

*Performs rotate-left operation on register A.*

```
WIDTH=32;
DEPTH=64;

ADDRESS_RADIX=UNS;
DATA_RADIX=HEX;

CONTENT BEGIN
    0   : 00000008;  -- Load immediate value into register A
    1   : 74000000;  -- ROL instruction
    2   : 70000000;  -- ADD instruction (or NOP for observing result)
    [3..63] : 00000000;
END;
```

Figure 31: MIF Explanation

*Two opcodes simulate ROL on predefined value.*

---

## ROR

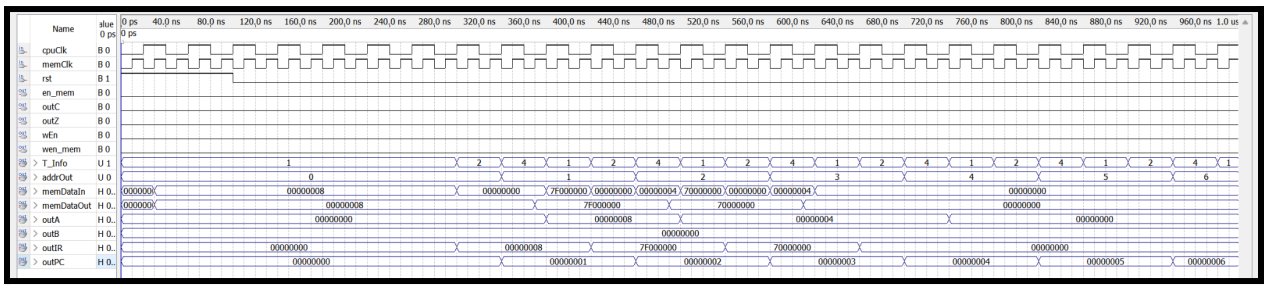


Figure 32: Waveform Simulation

*Performs rotate-right operation on register A.*

```
WIDTH=32;
DEPTH=64;

ADDRESS_RADIX=UNS;
DATA_RADIX=HEX;

CONTENT BEGIN
    0   : 00000008;  -- Load immediate value into register A
    1   : 7F000000;  -- ROR instruction
    2   : 70000000;  -- ADD instruction (or NOP to view result)
    [3..63] : 00000000;
END;
```

Figure 33: MIF Explanation

*Instruction memory loaded to show ROR effect on bit positions.*

---

## BEQ

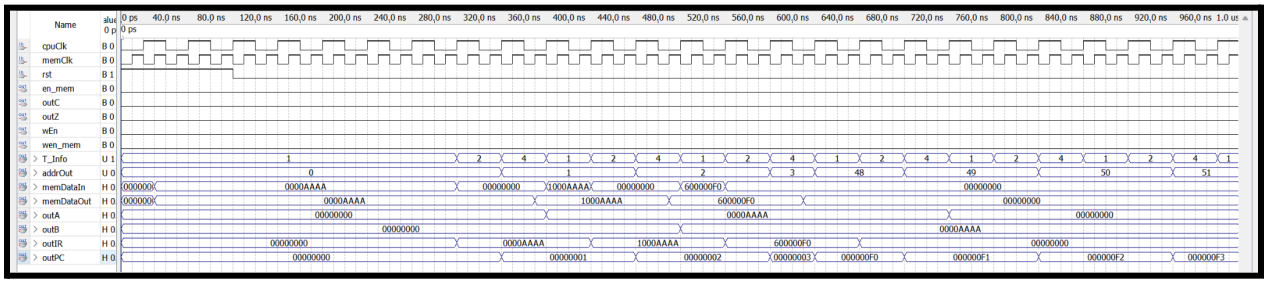


Figure 34: Waveform Simulation

*Simulates branch-if-equal using flags from ALU. Jump is taken based on Z flag.*

```
WIDTH=32;
DEPTH=64;

ADDRESS_RADIX=UNS;
DATA_RADIX=HEX;

CONTENT BEGIN
    0   : 0000AAAA;  -- LDAI (Load Immediate into A)
    1   : 1000AAAA;  -- LDBI (Load Immediate into B)
    2   : 600000F0;  -- BEQ to address offset (usually means jump if A == B)
    [3..63] : 00000000;
END;
```

Figure 35: MIF Explanation

*Memory programmed to test conditional branching. BEQ occurs based on equality check.*

---

## BNE Instruction Simulation

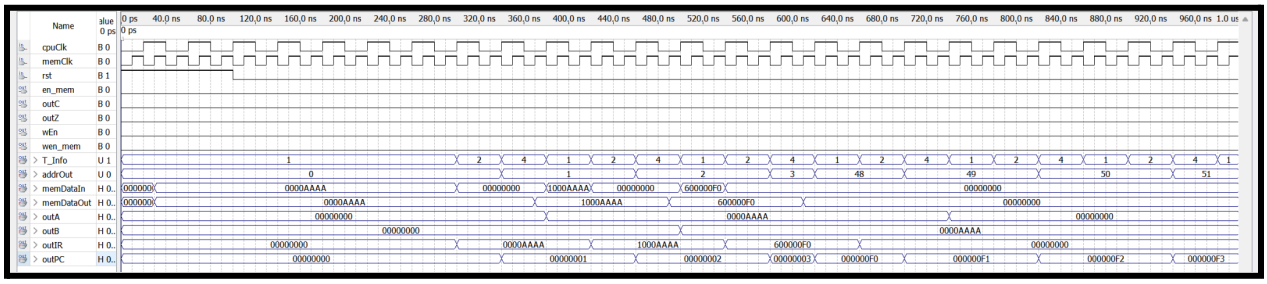


Figure 36: Waveform for BNE Execution

*This waveform illustrates the behavior of the BNE (Branch if Not Equal) instruction. Register A is loaded with AAAA, and Register B with BBBB. Since the two values differ, the CPU correctly performs a branch to the address F0, as shown by the updated program counter (PC) in the waveform.*

```
WIDTH=32;
DEPTH=64;

ADDRESS_RADIX=UNS;
DATA_RADIX=HEX;

CONTENT BEGIN
    0   : 0000AAAA;  -- LDAI (Load Immediate into A)
    1   : 1000AAAA;  -- LDBI (Load Immediate into B)
    2   : 600000F0;  -- BEQ to address offset (usually means jump if A == B)
    [3..63] : 00000000;
END;
```

Figure 37: MIF Setup for BNE Instruction

*This MIF setup initializes the instruction memory to test BNE. It loads values into A and B, followed by the branch instruction. The remaining memory is filled with zeros.*

---

## VHDL Implementation of CPU

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

entity Control is
port(
clk, mclk : in std_logic;
enable : in std_logic;
statusC, statusZ : in std_logic;
INST : in std_logic_vector(31 downto 0);
A_Mux, B_Mux : out std_logic;
IM_MUX1, REG_MUX : out std_logic;
IM_MUX2, DATA_Mux : out std_logic_vector(1 downto 0);
ALU_op : out std_logic_vector(2 downto 0);
inc_PC, ld_PC : out std_logic;
clr_IR, ld_IR : out std_logic;
clr_A, clr_B, clr_C, clr_Z : out std_logic;
ld_A, ld_B, ld_C, ld_Z : out std_logic;
T : out std_logic_vector(2 downto 0);
wen, en : out std_logic
);
end Control;

architecture description of Control is
    type STATETYPE is (state_0, state_1, state_2);
    signal present_state : STATETYPE;
    signal Instruction_sig : std_logic_vector(3 downto 0);
    signal Instruction_sig2 : std_logic_vector(7 downto 0);
begin
    Instruction_sig <= INST(31 downto 28);
    Instruction_sig2 <= INST(31 downto 24);

-----OPERATION DECODER-----
process (present_state, INST, statusC, statusZ, enable, Instruction_sig, Instruction_sig2)
begin
   if enable = '1' then
      if present_state = state_0 then
         DATA_Mux <= "00"; --Fetch address of the next instruction_sig
         clr_IR<='0';
         ld_IR<='1';
         ld_PC<='0';
         inc_PC<='0';
         clr_A<='0';
         ld_A<='0';
         ld_B<='0';
         clr_B<='0';
         clr_C<='0';
         ld_C<='0';
         clr_Z<='0';
```

```
         elsif Instruction_sig2 = "01111100" then --TSTC
         if(statusC = '1') then
         clr_IR<='0';
         ld_IR<='0';
         ld_PC<='1';
         inc_PC<='1';
         clr_A<='0';
         ld_A<='0';
         ld_B<='0';
         clr_B<='0';
         clr_C<='0';
         ld_C<='0';
         clr_Z<='0';
         ld_Z<='0';
         end if;
         end if;
         end if;
         end if;
         end process;

-----STATE MACHINE-----
process (clk, enable)
begin
   if enable = '1' then
   if rising_edge (clk) then
   if present_state = state_0 then present_state <= state_1;
   elsif present_state = state_1 then present_state <= state_2;
   else present_state <= state_0;
   end if;
   end if;
   else present_state <= state_0;
   end if;
   end process;

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

Figure 1: VHDL Code for Control Unit

*The Control.vhd module defines the logic for the CPU control unit, ensuring correct sequencing of fetch, decode, and execute phases. The FSM transitions through states T0, T1, and T2, setting control signals accordingly. The operation decoder determines execution steps based on the instruction opcode, and the memory signal generator enables correct read/write operations. This implementation ensures that all CPU components function in harmony, allowing seamless execution of instructions.*

---

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

-- 4-bit Adder
ENTITY adder4 IS
    PORT (
        Cin  : IN STD_LOGIC;  -- Carry-in
        X, Y : IN STD_LOGIC_VECTOR(3 DOWNTO 0);  -- 4-bit inputs
        S    : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);  -- 4-bit sum output
        Cout : OUT STD_LOGIC  -- Carry-out
    );
END adder4;

ARCHITECTURE behavior 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);  -- Internal carry signals

BEGIN
    -- Instantiate four 1-bit full adders to form a 4-bit ripple-carry add
    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 behavior;

```

Figure 3. Adder4.vhd

*A 4-bit adder module that forms part of the ALU's arithmetic capabilities.*

---

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

-- 16-bit Adder
ENTITY adder16 IS
    PORT (
        Cin  : IN STD_LOGIC;  -- Carry-in
        X, Y : IN STD_LOGIC_VECTOR(15 DOWNTO 0);  -- 16-bit inputs
        S    : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);  -- 16-bit sum output
        Cout : OUT STD_LOGIC  -- Carry-out
    );
END adder16;

ARCHITECTURE behavior OF adder16 IS
    COMPONENT adder4
        PORT (
            Cin  : IN STD_LOGIC;
            X, Y : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
            S    : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
            Cout : OUT STD_LOGIC
        );
    END COMPONENT;

    SIGNAL C : STD_LOGIC_VECTOR(1 TO 3);  -- Internal carry signals

BEGIN
    -- Instantiate four 4-bit adders to form a 16-bit ripple-carry adder
    stage0: adder4 PORT MAP(Cin, X(3 DOWNTO 0),   Y(3 DOWNTO 0),   S(3 DOW
    stage1: adder4 PORT MAP(C(1), X(7 DOWNTO 4),  Y(7 DOWNTO 4),  S(7 DOWN
    stage2: adder4 PORT MAP(C(2), X(11 DOWNTO 8), Y(11 DOWNTO 8), S(11 DOW
    stage3: adder4 PORT MAP(C(3), X(15 DOWNTO 12), Y(15 DOWNTO 12), S(15 I

END behavior;

```

Figure 4. Adder16.vhd.

*A 16-bit adder that enables larger arithmetic operations within the CPU.*

---

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

-- 32-bit Adder
ENTITY adder32 IS
    PORT (
        Cin  : IN STD_LOGIC;  -- Carry-in
        X, Y : IN STD_LOGIC_VECTOR(31 DOWNTO 0);  -- 32-bit inputs
        S    : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);  -- 32-bit sum output
        Cout : OUT STD_LOGIC  -- Carry-out
    );
END adder32;

ARCHITECTURE behavior OF adder32 IS
    COMPONENT adder16
        PORT (
            Cin  : IN STD_LOGIC;
            X, Y : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
            S    : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
            Cout : OUT STD_LOGIC
        );
    END COMPONENT;

    SIGNAL C : STD_LOGIC;  -- Internal carry signal

BEGIN
    -- Instantiate two 16-bit adders to form a 32-bit ripple-carry adder
    stage0: adder16 PORT MAP(Cin, X(15 DOWNTO 0),   Y(15 DOWNTO 0),   S(15
    stage1: adder16 PORT MAP(C,   X(31 DOWNTO 16),  Y(31 DOWNTO 16),  S(31

END behavior;
```

Figure 5. Adder32.vhd

*A 32-bit adder responsible for performing full-width arithmetic computations.*

---

```vhdl
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;

-- ALU Entity
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);  -- □ Fixed missing semic
        result: OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
        zero  : OUT STD_LOGIC;  -- □ Fixed `std_login` to `std_logic`
        cout  : OUT STD_LOGIC   -- □ Fixed `out std_logic_vector` to `out
    );
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, cout_add, cout_sub : STD_LOGIC := '0';
    SIGNAL zero_s : STD_LOGIC;

BEGIN
    -- Instantiate Adders
    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);

    -- ALU Operation Logic
    PROCESS (a, b, op)
    BEGIN
        CASE op IS
            WHEN "000" =>  -- AND
                result_s <= a AND b;
                cout_s <= '0';
            WHEN "001" =>  -- OR
                result_s <= a OR b;
                cout_s <= '0';
            WHEN "010" =>  -- ADD
                result_s <= result_add;
                cout_s <= cout_add;
            WHEN "011" =>  -- Pass B
                result_s <= b;
                cout_s <= '0';
            WHEN "110" =>  -- SUB
                result_s <= result_sub;
                cout_s <= cout_sub;
            WHEN "100" =>  -- Shift Left Logical (SLL 1)
                result_s <= a(30 DOWNTO 0) & '0';  -- Correct syntax
                cout_s <= a(31);
            WHEN "101" =>  -- Shift Right Logical (SRL 1)
                result_s <= '0' & a(31 DOWNTO 1);  -- Correct syntax
                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;

    -- Assign Outputs
    result <= result_s;
    cout <= cout_s;
    zero <= zero_s;

END behavior;
```

Figure 6. Alu.vhd

*The Arithmetic Logic Unit (ALU) processes arithmetic and logical operations based on control signals.*

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

ENTITY data_path IS
    PORT (
        -- Clock Signals
        Clk, mClk : IN STD_LOGIC;

        -- Memory Signals
        WEN, EN : IN STD_LOGIC;

        -- Register Control Signals (Clr and Load)
        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;
        Clr_PC, Ld_PC : IN STD_LOGIC;
        Clr_IR, Ld_IR : IN STD_LOGIC;

        -- 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 DOWN
        MEM_ADDR : OUT UNSIGNED(7 DOWNTO 0);

        -- Various MUX Controls
        DATA_MUX : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        REG_MUX, A_MUX, B_MUX, 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);

        -- Register Outputs
        Out_A, Out_B : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
        Out_C, Out_Z : OUT STD_LOGIC;
        Out_PC, Out_IR : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
    );
END data_path;

ARCHITECTURE Behavior OF data_path IS
```

```
    -- Component Instantiations
    COMPONENT data_mem IS
        PORT (
            clk : IN STD_LOGIC;
            addr : IN UNSIGNED(7 DOWNTO 0);
            data_in : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
            wen, en : IN STD_LOGIC;
            data_out : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
        );
    END COMPONENT;

    COMPONENT register32 IS
        PORT (
            d : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
            ld, clr, clk : IN STD_LOGIC;
            q : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
        );
    END COMPONENT;

    COMPONENT pc IS
        PORT (
            clr, clk, ld, inc : IN STD_LOGIC;
            d : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
            q : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
        );
    END COMPONENT;

    COMPONENT LZE IS
        PORT (
            LZE_in : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
            LZE_out : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
        );
    END COMPONENT;

    COMPONENT UZE IS
        PORT (
            UZE_in : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
            UZE_out : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
        );
    END COMPONENT;

    COMPONENT RED IS
        PORT (
            RED_in : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
            RED_out : OUT UNSIGNED(7 DOWNTO 0)
        );
    END COMPONENT;
```

```
    COMPONENT mux2to1 IS
        PORT (
            s : IN STD_LOGIC;
            w0, w1 : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
            f : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
        );
    END COMPONENT;

    COMPONENT mux4to1 IS
        PORT (
            s : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
            X1, X2, X3, X4 : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
            f : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
        );
    END COMPONENT;

    COMPONENT alu IS
        PORT (
            a, 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, cout : OUT STD_LOGIC
        );
    END COMPONENT;

    -- Signal Declarations
    SIGNAL IR_OUT, data_bus_s : STD_LOGIC_VECTOR(31 DOWNTO 0);
    SIGNAL LZE_out_PC, LZE_out_A_Mux, LZE_out_B_Mux : STD_LOGIC_VECTOR(31
    SIGNAL RED_out_Data_Mem : UNSIGNED(7 DOWNTO 0);
    SIGNAL A_Mux_out, B_Mux_out, reg_A_out, reg_B_out : STD_LOGIC_VECTOR(3
    SIGNAL reg_Mux_out, data_mem_out : STD_LOGIC_VECTOR(31 DOWNTO 0);
    SIGNAL UZE_IM_MUX1_out, IM_MUX1_out : STD_LOGIC_VECTOR(31 DOWNTO 0);
    SIGNAL LZE_IM_MUX2_out, IM_MUX2_out, ALU_out : STD_LOGIC_VECTOR(31 DOW
    SIGNAL zero_flag, carry_flag : STD_LOGIC;
    SIGNAL temp : STD_LOGIC_VECTOR(30 DOWNTO 0) := (OTHERS => '0');
    SIGNAL out_pc_sig : STD_LOGIC_VECTOR(31 DOWNTO 0);

BEGIN

    -- Register and Memory Connections
    IR: register32 PORT MAP(data_bus_s, Ld_IR, Clr_IR, Clk, IR_OUT);
    LZE_PC: LZE PORT MAP(IR_OUT, LZE_out_PC);
    PC0: pc PORT MAP(Clr_PC, Clk, Ld_PC, Inc_PC, LZE_out_PC, out_pc_sig);

    -- A MUX & Register A
    LZE_A_Mux: LZE PORT MAP(IR_OUT, LZE_out_A_Mux);
    A_Mux0: mux2to1 PORT MAP(A_MUX, data_bus_s, LZE_out_A_Mux, A_Mux_out);
```

```
    LZE_PC: LZE PORT MAP(IR_OUT, LZE_out_PC);
    PC0: pc PORT MAP(Clr_PC, Clk, Ld_PC, Inc_PC, LZE_out_PC, out_pc_sig);

    -- A MUX & Register A
    LZE_A_Mux: LZE PORT MAP(IR_OUT, LZE_out_A_Mux);
    A_Mux0: mux2to1 PORT MAP(A_MUX, data_bus_s, LZE_out_A_Mux, A_Mux_out);
    Reg_A: register32 PORT MAP(A_Mux_out, Ld_A, Clr_A, Clk, reg_A_out);

    -- B MUX & Register B
    LZE_B_Mux: LZE PORT MAP(IR_OUT, LZE_out_B_Mux);
    B_Mux0: mux2to1 PORT MAP(B_MUX, data_bus_s, LZE_out_B_Mux, B_Mux_out);
    Reg_B: register32 PORT MAP(B_Mux_out, Ld_B, Clr_B, Clk, reg_B_out);

    -- Register Multiplexer
    Reg_Mux0: mux2to1 PORT MAP(REG_MUX, reg_A_out, reg_B_out, reg_Mux_out)

    -- Data Memory
    RED_Data_Mem: RED PORT MAP(IR_OUT, RED_out_Data_Mem);
    Data_Mem0: data_mem PORT MAP(mClk, RED_out_Data_Mem, reg_Mux_out, WEN,

    -- Immediate MUX 1
    UZE_IM_MUX1: UZE PORT MAP(IR_OUT, UZE_IM_MUX1_out);
    IM_MUX1a: mux2to1 PORT MAP(IM_MUX1, reg_A_out, UZE_IM_MUX1_out, IM_MUX

    -- Immediate MUX 2
    LZE_IM_MUX2: LZE PORT MAP(IR_OUT, LZE_IM_MUX2_out);
    IM_MUX2a: mux4to1 PORT MAP(IM_MUX2, reg_B_out, LZE_IM_MUX2_out, (temp

    -- ALU
    ALU0: alu PORT MAP(IM_MUX1_out, IM_MUX2_out, ALU_OP, ALU_out, zero_fla

    -- Data MUX
    DATA_MUX0: mux4to1 PORT MAP(DATA_MUX, DATA_IN, data_mem_out, ALU_out,

    -- Assign Outputs
    DATA_BUS <= data_bus_s;
    OUT_A <= reg_A_out;
    OUT_B <= reg_B_out;
    OUT_IR <= IR_OUT;
    ADDR_OUT <= out_pc_sig;
    OUT_PC <= out_pc_sig;
    MEM_ADDR <= RED_out_Data_Mem;
    MEM_IN <= reg_Mux_out;
    MEM_OUT <= data_mem_out;

END Behavior;
```

Figure 7: Data_path.vhd

*Defines the data path, including registers, ALU, and multiplexers, ensuring correct data movement.*

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

-- Data Memory Module Entity
ENTITY data_mem IS
    PORT (
        clk      : IN std_logic;  -- Clock signal
        addr     : IN unsigned(7 DOWNTO 0);  -- 8-bit memory address (256 locations)
        data_in  : IN std_logic_vector(31 DOWNTO 0);  -- 32-bit input data
        wen      : IN std_logic;  -- Write enable (1 = write, 0 = read)
        en       : IN std_logic;  -- Enable signal (1 = active, 0 = inactive)
        data_out : OUT std_logic_vector(31 DOWNTO 0)  -- 32-bit output data
    );
END data_mem;

ARCHITECTURE Behavior OF data_mem IS
    -- Declare memory as an array of 256 words, each 32-bits wide
    TYPE RAM IS ARRAY (0 TO 255) OF std_logic_vector(31 DOWNTO 0);
    SIGNAL DATAMEM : RAM := (OTHERS => (OTHERS => '0'));  -- Initialize memory to zeros

BEGIN
    PROCESS (clk)
    BEGIN
        -- On the falling edge of the clock, perform read/write operations
        IF (clk'EVENT AND clk = '0') THEN
            IF (en = '1') THEN  -- Only operate when enabled
                IF (wen = '1') THEN
                    -- Write operation: Store data_in at address addr
                    DATAMEM(to_integer(addr)) <= data_in;
                    data_out <= (OTHERS => '0');  -- Clear output after write
                ELSIF (wen = '0') THEN
                    -- Read operation: Output the value stored at address addr
                    data_out <= DATAMEM(to_integer(addr));
                END IF;
            ELSE
                -- When enable is low, output zeros
                data_out <= (OTHERS => '0');
            END IF;
        END IF;
    END PROCESS;

END Behavior;

```

Figure 8: Data_mem.vhd

*Implements the memory storage unit, supporting read and write operations.*

---

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

-- 1-bit Full Adder
ENTITY fulladd IS
    PORT (
        Cin, x, y : IN STD_LOGIC;  -- Inputs: Carry-in, x, and y
        s, Cout   : OUT STD_LOGIC   -- Outputs: Sum (lowercase s) and Carr
    );
END fulladd;

ARCHITECTURE behavior OF fulladd IS
BEGIN
    -- Sum computation using XOR gates
    s <= x XOR y XOR Cin;

    -- Carry-out computation using OR-AND logic
    Cout <= (x AND y) OR (Cin AND x) OR (Cin AND y);
END behavior;

```

Figure 9. Fulladd.vhd

*A full-adder module utilized within arithmetic operations.*

---

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY LZE IS
    PORT(
        LZE_in  : IN std_logic_vector(31 DOWNTO 0);  -- Input signal
        LZE_out : OUT std_logic_vector(31 DOWNTO 0)  -- Output signal
    );
END ENTITY LZE;

ARCHITECTURE Behavior OF LZE IS
    SIGNAL zeros: std_logic_vector(15 DOWNTO 0) := (OTHERS => '0');  -- 16
BEGIN
    -- Concatenates 16 zeros with the lower 16 bits of LZE_in
    LZE_out <= zeros & LZE_in(15 DOWNTO 0);
END Behavior;

```

Figure 10. LZE.vhd

Logical Zero Extension module extends immediate values for operations requiring sign extension.

---

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY mux2to1 IS
PORT(
    s  : IN STD_LOGIC;                -- Select signal
    w0 : IN STD_LOGIC_VECTOR(31 DOWNTO 0);  -- Input 0
    w1 : IN STD_LOGIC_VECTOR(31 DOWNTO 0);  -- Input 1
    f  : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)  -- Output
);
END mux2to1;

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
    WITH s SELECT
        f <= w0 WHEN '0',
             w1 WHEN OTHERS;
END Behavior;

```

Figure 11. Mux2to1.vhd

*A two-input multiplexer used for selecting data sources.*

---

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY mux4to1 IS
    PORT (
        s  : IN std_logic_vector(1 DOWNTO 0);  -- 2-bit selector
        X1, X2, X3, X4 : IN std_logic_vector(31 DOWNTO 0);  -- 4 input dat
        f  : OUT std_logic_vector(31 DOWNTO 0)  -- Output line
    );
END ENTITY mux4to1;

ARCHITECTURE Behavior OF mux4to1 IS
BEGIN
    -- Selects one of the four inputs based on selector 's'
    WITH s SELECT
        f <= X1 WHEN "00",
             X2 WHEN "01",
             X3 WHEN "10",
             X4 WHEN "11";
END Behavior;

```

Figure 12. Mux4to1.vhd

*A four-input multiplexer used for complex data routing.*

---

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

ENTITY pc IS
PORT(
    clr : IN STD_LOGIC;                         -- Clear signal
    clk : IN STD_LOGIC;                         -- Clock signal
    ld  : IN STD_LOGIC;                         -- Load/Enable signal
    inc : IN STD_LOGIC;                         -- Increment signal
    d   : IN STD_LOGIC_VECTOR(31 DOWNTO 0);     -- Data input (PC value)
    q   : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)     -- Output (PC value)
);
END pc;

ARCHITECTURE Behavior OF pc IS

    COMPONENT add
    PORT(
        A  : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        B  : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
    );
    END COMPONENT;

    COMPONENT mux2to1
    PORT(
        s  : IN STD_LOGIC;
        w0 : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        w1 : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        f  : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
    );
    END COMPONENT;

    COMPONENT register32
    PORT(
        d   : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        ld  : IN STD_LOGIC;
        clr : IN STD_LOGIC;
        clk : IN STD_LOGIC;
        Q   : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
    );
    END COMPONENT;

    SIGNAL add_out : STD_LOGIC_VECTOR(31 DOWNTO 0);
    SIGNAL mux_out : STD_LOGIC_VECTOR(31 DOWNTO 0);
    SIGNAL q_out : STD_LOGIC_VECTOR(31 DOWNTO 0);
```

```vhdl
    COMPONENT add
    PORT(
        A  : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        B  : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
    );
    END COMPONENT;

    COMPONENT mux2to1
    PORT(
        s  : IN STD_LOGIC;
        w0 : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        w1 : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        f  : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
    );
    END COMPONENT;

    COMPONENT register32
    PORT(
        d   : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        ld  : IN STD_LOGIC;
        clr : IN STD_LOGIC;
        clk : IN STD_LOGIC;
        Q   : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
    );
    END COMPONENT;

    SIGNAL add_out : STD_LOGIC_VECTOR(31 DOWNTO 0);
    SIGNAL mux_out : STD_LOGIC_VECTOR(31 DOWNTO 0);
    SIGNAL q_out : STD_LOGIC_VECTOR(31 DOWNTO 0);

BEGIN

    -- Add block to increment the PC by 4
    add0 : add PORT MAP (q_out, add_out);

    -- 2-to-1 Multiplexer to select between PC+4 and the data input `d`
    mux0 : mux2to1 PORT MAP (inc, d, add_out, mux_out);

    -- 32-bit Register to hold the PC value
    reg0 : register32 PORT MAP (mux_out, ld, clr, clk, q_out);

    -- Output assignment
    q <= q_out;

END Behavior;
```

Figure 13. Pc.vhd

*The Program Counter module responsible for maintaining instruction execution order.*

---

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY RED IS
    PORT (
        RED_in  : IN std_logic_vector(31 DOWNTO 0);  -- 32-bit input
        RED_out : OUT unsigned(7 DOWNTO 0)  -- 8-bit output
    );
END ENTITY RED;

ARCHITECTURE Behavior OF RED IS
BEGIN
    -- Extracts the lower 8 bits of RED_in and converts them to an unsigne
    RED_out <= unsigned(RED_in(7 DOWNTO 0));
END Behavior;

```

Figure 14. RED.vhd

*Extracts address and control signals from instructions.*

---

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

ENTITY register32 IS
PORT(
    d   : IN STD_LOGIC_VECTOR(31 DOWNTO 0); -- Input data (32 bits)
    ld  : IN STD_LOGIC;                     -- Load/Enable signal
    clr : IN STD_LOGIC;                     -- Clear signal
    clk : IN STD_LOGIC;                     -- Clock signal
    Q   : OUT STD_LOGIC_VECTOR(31 DOWNTO 0) -- Output data (32 bits)
);
END register32;

ARCHITECTURE Behavior OF register32 IS
BEGIN
    PROCESS(ld, clr, clk)
    BEGIN
        IF clr = '1' THEN
            Q <= (OTHERS => '0');  -- Clear the register to 0
        ELSIF (clk'event AND clk = '1' AND ld = '1') THEN
            Q <= d;  -- Load the value of `d` into the register on rising
        END IF;
    END PROCESS;
END Behavior;

```

Figure 15. Register32.vhd

*Defines 32-bit registers used for storing operands and computation results.*

---

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY UZE IS
    PORT(
        UZE_in  : IN std_logic_vector(31 DOWNTO 0);  -- Input signal
        UZE_out : OUT std_logic_vector(31 DOWNTO 0)  -- Output signal
    );
END ENTITY UZE;

ARCHITECTURE Behavior OF UZE IS
    SIGNAL zeros: std_logic_vector(15 DOWNTO 0) := (OTHERS => '0');  -- 16
BEGIN
    -- Concatenates lower 16 bits of UZE_in with 16 zeros
    UZE_out <= UZE_in(15 DOWNTO 0) & zeros;
END Behavior;

```

Figure 16. UZE.vhd

*Upper Zero Extension module for handling upper immediate values*

---

**Conclusion**

This lab demonstrated the full implementation of a Semi-RISC CPU in VHDL. By integrating a datapath, control unit, and reset circuit, the system could interpret and execute 16+ instructions. Simulations confirmed accurate behavior via .mif-based instruction loading and waveform verification. Each opcode's behavior—arithmetic, logic, memory, and branching—was successfully emulated. This lab served as a culmination of concepts learned throughout the course and provided a foundation for more complex CPU architectures and digital systems.

# References

[1] Geurkov, V. (2017). *COE608: Computer Organization and Architecture Lab Manual* (Winter 2017). Toronto Metropolitan University.

[2] IEEE. (2008). *IEEE Standard VHDL Language Reference Manual*. IEEE.

[3] "VHDL Documentation and Tutorials." (n.d.). Retrieved January 2025, from https://www.vhdl.org