Laboratory #2: Single-Cycle RISC Processor

Submission Date: March 21st 2025

CEG3156 - Computer Systems Design
Winter 2025

School of Electrical Engineering and Computer Science at the
University of Ottawa

Course Coordinator: Rami Abielmona, PhD, P. Eng
Teaching Assistant: Pavly Saleh

Nida Taj 300239050
August Zhang 300310509
Gavin Gao 300190846

Group 6

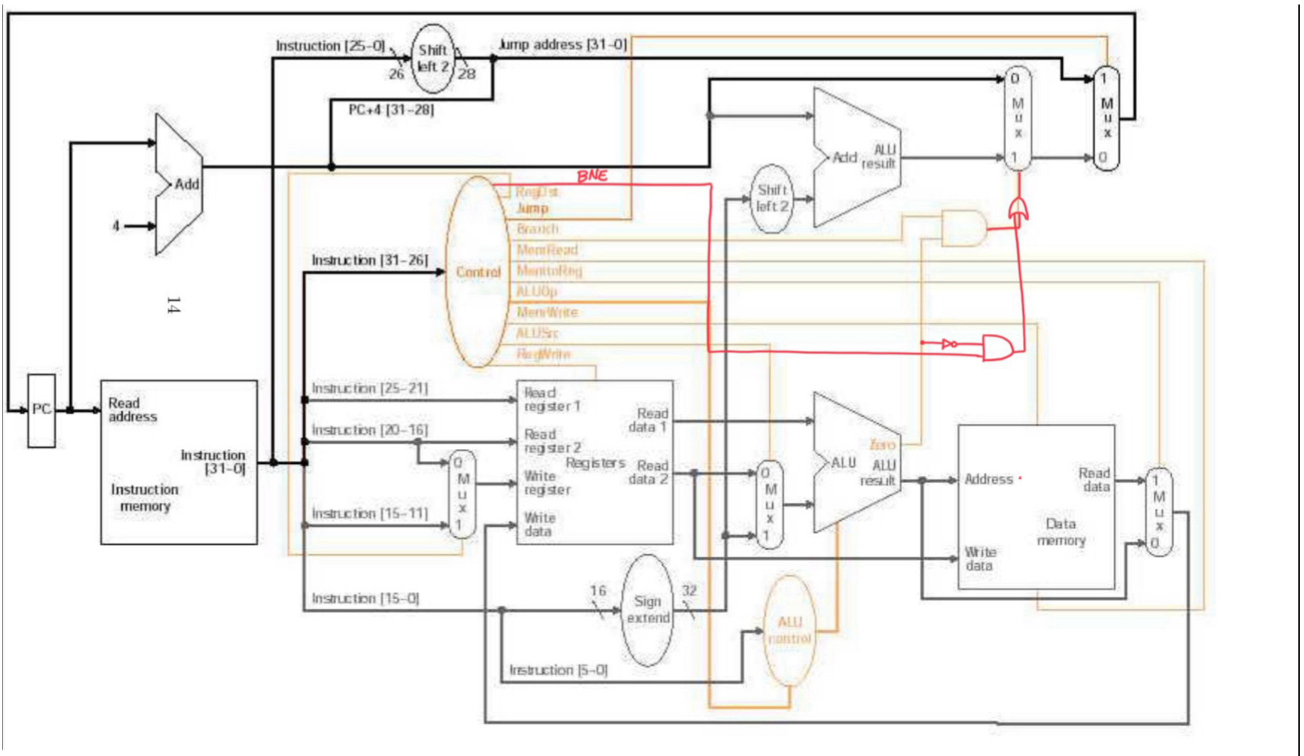
Table of Contents

Theoretical Part

Introduction

The objective of this lab is to be able to successfully design and realize a single-cycle RISC based processor using VHDL. Additionally, the lab requires the successful handling of a branch not equal (BNE) instruction. This concept of designing and implementing a single-cycle processor is particularly important to understand how the different components such as ALUs, file registers, and control units interact with one another to effectively handle instructions. Furthermore, implementing the processor using a RISC based design has practical applications which can be seen in things such as embedded systems; therefore, this lab provides exposure to this fundamental concept often seen in many real-world applications.

Pre – Lab



Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0	BNE
R-format	1	0	0	1	0	0	0	1	0	0
lw	0	1	1	1	1	0	0	0	0	0
sw	X	1	X	0	0	1	0	0	0	0
beq	X	0	X	0	0	0	1	0	1	0
bne	X	0	X	0	0	0	0	0	1	1

Figure 1.1 – Single-cycle processor with additional BNE instruction handling included

Discussion of Problem

The problem to be address over the course of this laboratory is to first modify the single-cycle processor datapath, to be an 8-bit datapath rather than 32-bits which it is currently. Furthermore, the processor despite being an 8-bit datapath should still be able to handle instruction widths of 32-bits. Additionally, the top-level entity of the processor should be able to take as input a ValueSelect signal that is 3-bits wide, which will select and output certain values, such as program counter value, ALU result, read data and write data. The processor should then be able to take as input a benchmark program and output the correct register and memory values.

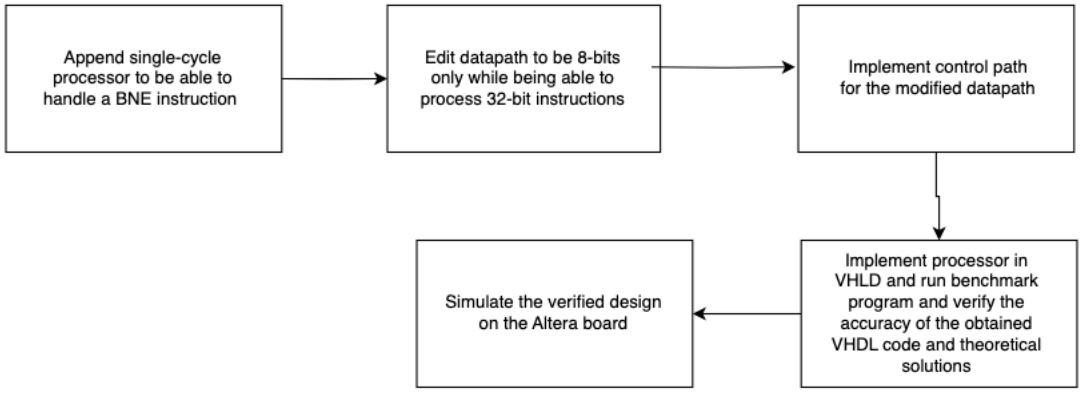

Append single-cycle processor to be able to handle a BNE instruction
Edit datapath to be 8-bits only while being able to process 32-bit instructions
Implement control path for the modified datapath
Implement processor in VHLD and run benchmark program and verify the accuracy of the obtained VHDL code and theoretical solutions
Simulate the verified design on the Altera board

Figure 1.2 – Flowchart representation of proposed solution

Discussion of Algorithmic Solution

The solution for this problem was derived by modifying the single-cycle processor datapath shown in figure 9 of the ceg3156Lab2 document. A control path was also deduced from the revised datapath.

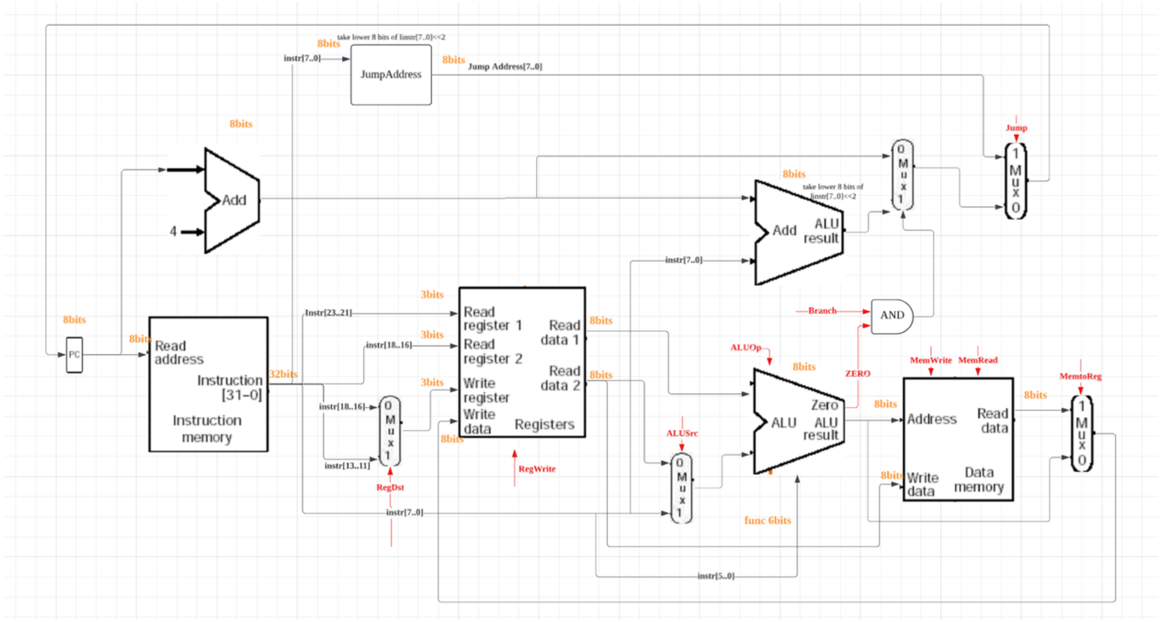


Figure 1.3 – Single-cycle processor revised 8-bit datapath

As can be seen in figure 1.3 above, the datapath has been slightly modified to handle the 32-bit instruction appropriately for an 8-bit datapath. The program counter (PC) only takes as input 8-bits for the address, which is then read and outputs the 32-bit instruction. Previously, before the revision to the datapath, rs took instruction bits 25-21, now it takes bits 23-21. Similarly, rt now takes instr[18-16] and the mux for the write register output still takes rt as an input but rd is now instr[13-11]. The ALUs now all perform 8-bit operations as opposed to 32-bits, and the jump address also only takes instr[7-0] instead of instr[25-0]. The biggest modification to this datapath is the lack of a sign extend component, simply because in this case it is not necessary to sign extend since the datapath can only take 8-bits at a time, but the instruction is 32-bits, therefore the bit lengths are not shorter and therefore don't require the need to be sign extended.

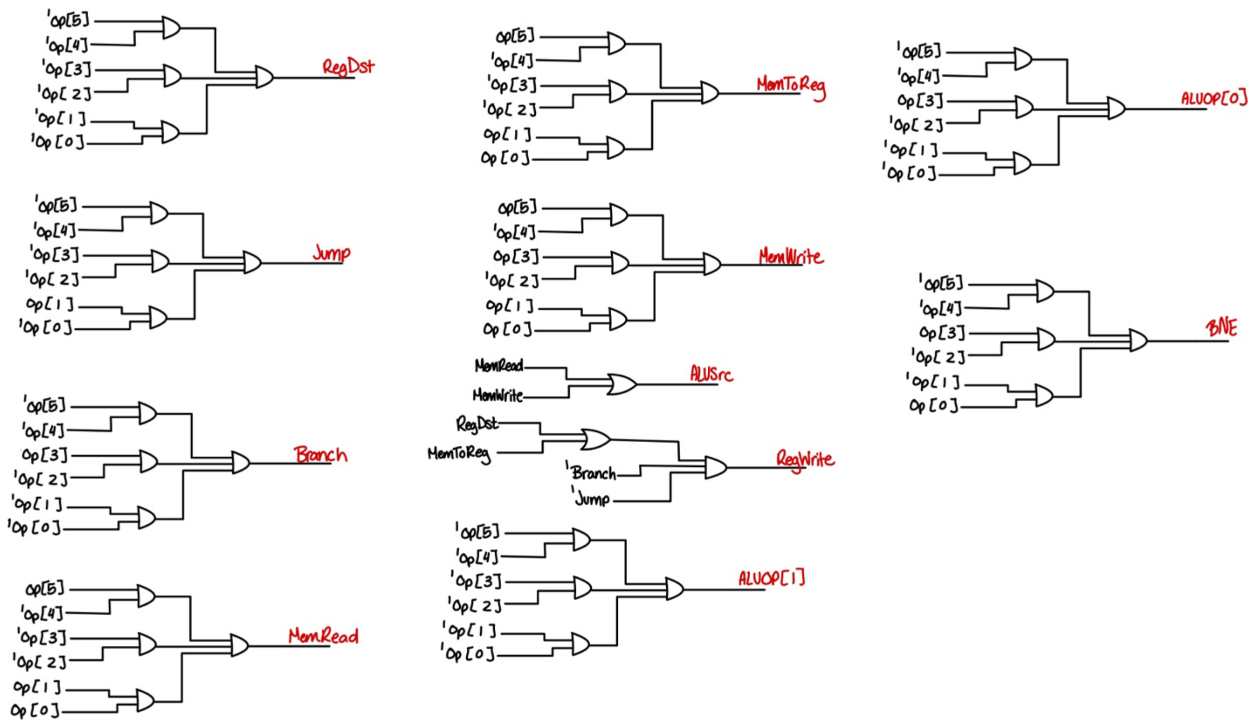


Figure 1.4 – Single-cycle processor control path

Figure 1.4 shows the control logic to be implemented for all the individual control signals in the datapath, that will comprise the control unit. The control signals are derived based on the opcodes, ranging from opcode [5-0], which is equivalent to the funct field mapped to first 6 bits instr[31-26].

Design Part

Discussion of Used Components

PC: fetch current 8bit instruction address, takes input of next 8bit instruction address and updates in next rising clock cycle, sets to 00 when reset signal is on.

FullAdder8bit+4: calculate current instruction address + 4 and pass to Mux for deciding next instruction address.

Instruction Memory: using rom_type, stores total of 256 addresses that contains 32 bits of instruction. In this lab, it contains a list of instructions from the verification part in the lab manual that must be fetched in sequence. Takes 8-bit address input and 32 bit instruction output.

JumpAddress unit:
Calculates the target jump address by shifting the input address by 2 bits to the left.

Central_RegisterFile: using reg-array type, contain 8 registers. read_data1 and read_data2 will constantly read the register contents from address that are determined by the inputs from read_reg1 and read_reg2. Write_data will update the content of the register at the address determined by write_reg when Reg_write is on, in the next rising lock cycle.

ALU: Perform different operations depending on the types of instruction. For lw and sw, performs A+B; for beq, perform A-B; for R type, based on 6bits funct, it performs Add, Subtract, And, OR, SLT(compare if A<B), or no operation in any other situations (output all 0s).

ALU_with_Shift: Calculate PC+4 + Offset address for branch instruction, by shift operand B(offset) to left by 2 bits then add with operand A(PC+4).

Data Memory: using ram type, stores 256 registers that contain address of 8bits data. Initially holds addr 00 = 55 and addr 01 = AA values. The contents of the registers will be updated in the next rising clock cycle.

Mux 3 bits: pass the address from instr[18..16] that will be chosen as write register address in CentralRegisterFile. It's been modified to 3 bits from 8 bits since in this lab only an 8 bit address will be used.

Mux 8 bits: there's several 8bit Muxes responsible for choosing an 8bit address or 8bit ALU result or 8bit value read from data memory.

Control Unit: in a single cycle processor, since all instructions are performed within one cycle, the control unit contains no flipflop but only combinational logic. The unit takes input of 6 bits of instruction first, then decodes to determine which type of instruction will be performed, finally it generates the corresponding control signals based on logic gates.

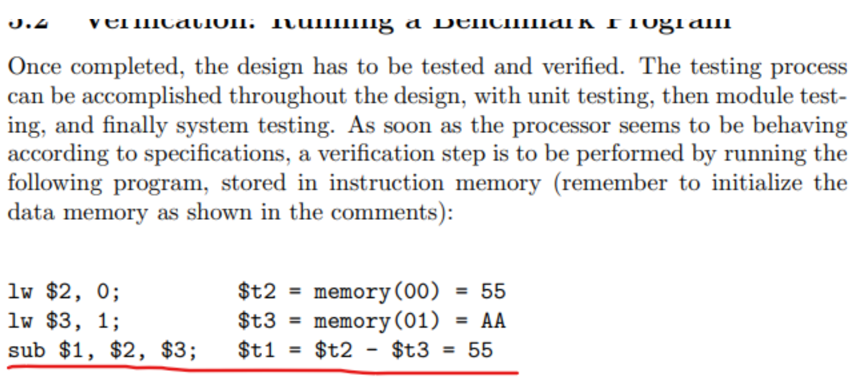
Discussion of Actual Solution

There are several things at the design stage that should be mentioned:

Given the list of instructions from the lab manual, there's an error at instruction line 3: sub $1. $2, $3, if the description on manual is followed, the result should be 55-AA = AB, therefore all the following results in the upcoming instructions will be incorrect; the corrected instruction is Sub $1, $3, $2, the result would be 55, and therefore the following results of the next instructions will match the given answers.

Given the last three instructions to be executed, ideally at the cycle where instruction j 11 with address 36 in decimal (0x24) is fetched, the PC in next cycle will updates address to 44 in decimal (0x2C in hexadecimal) in instruction memory, performing beq $1, $1, -44, updates PC to 00 and restarts the instruction list, skipping instruction beq $1, $2, -8;

if beq$1, $2, -8 instruction is assigned to address 44, it will cause an infinite loop between j 11 and beq $1, $2, -8, since beq $1, $2, -8 will update PC in next cycle 8 bytes backwards, which means (44 – 8 = 36, 0x24 in hex) going back to the address where j 11 is at. Therefore, the address of beq $1, $2, -8 should be assigned before or after beq $1, $1, -44, that is, address 40 in decimal (0x28 in hex) or 48 in decimal (0x30 in hex).

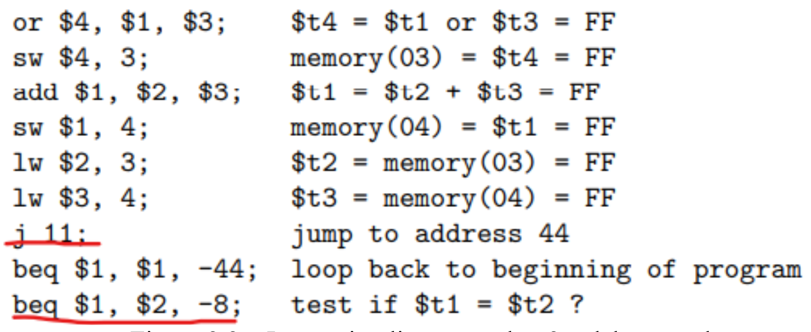


3.2 Verification: Running a Benchmark Program

Once completed, the design has to be tested and verified. The testing process can be accomplished throughout the design, with unit testing, then module testing, and finally system testing. As soon as the processor seems to be behaving according to specifications, a verification step is to be performed by running the following program, stored in instruction memory (remember to initialize the data memory as shown in the comments):

```
lw $2, 0;          $t2 = memory(00) = 55
lw $3, 1;          $t3 = memory(01) = AA
sub $1, $2, $3;    $t1 = $t2 - $t3 = 55
```

Figure 2.1 – Instruction list screenshot 1 at lab manual



```
or $4, $1, $3;     $t4 = $t1 or $t3 = FF
sw $4, 3;          memory(03) = $t4 = FF
add $1, $2, $3;    $t1 = $t2 + $t3 = FF
sw $1, 4;          memory(04) = $t1 = FF
lw $2, 3;          $t2 = memory(03) = FF
lw $3, 4;          $t3 = memory(04) = FF
j 11;              jump to address 44
beq $1, $1, -44;   loop back to beginning of program
beq $1, $2, -8;    test if $t1 = $t2 ?
```

Figure 2.2 – Instruction list screenshot 2 at lab manual

Discussion of Tool

In this lab, the tools used were the Quartus II software and the Altera DE2 Board. The Quartus II software was used to code the VHDL and design the hardware implemented in the amended single-cycle processor datapath above. The verification and simulation of the VHDL code was performed using the Altera DE2 board. More specifically, the switches on the board were used to set values for the ValueSelect multiplexer. The outputs were simulated on the LEDS of the board.

Discussion of Challenging Problems

The majority of the lab was developed successfully, with minimal issues. However, the major challenge encountered during this lab was being able to demonstrate the final solution on the board. When attempting to synthesize the design onto the Altera board, the MuxOut output was displaying incorrect results on the BCD 7 segment display on the board. The simulations confirmed that the VHDL code was functioning correctly without any errors, therefore the error was determined to not be in the logic. To troubleshoot the error, the clock was checked, to determine why the correct changes in the outputs was not occurring, however this was also later determined to not be an issue. Unfortunately, this issue was not successfully debugged over the course of this lab, resulting in an unsuccessful board demonstration.

Real Implementation

Shown simulation/synthesis results

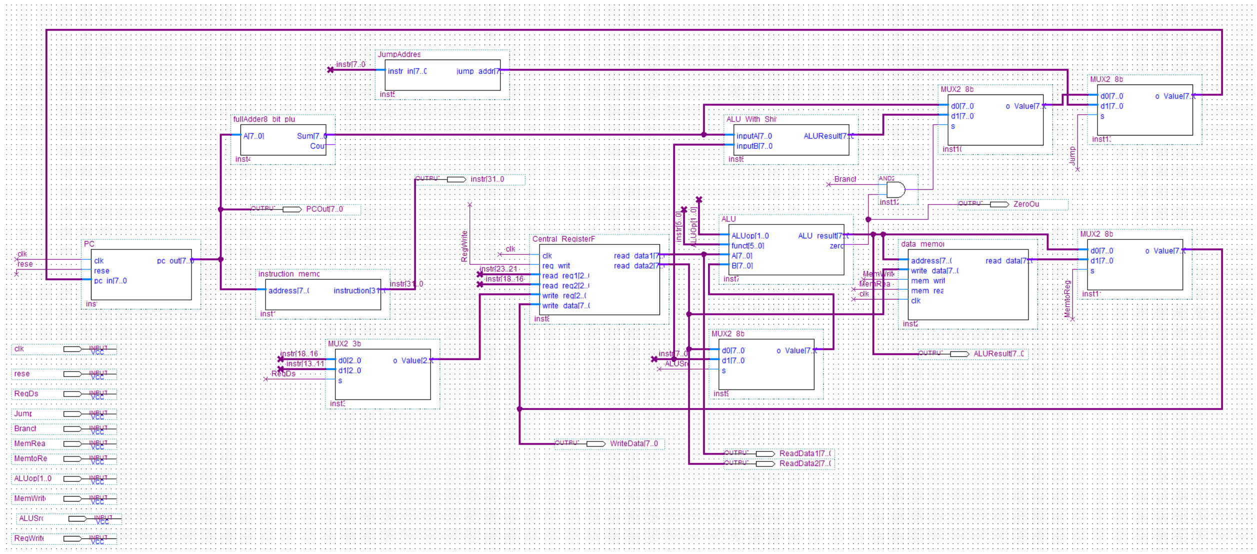


Figure 3.1 – Quartus II Data Path implementation

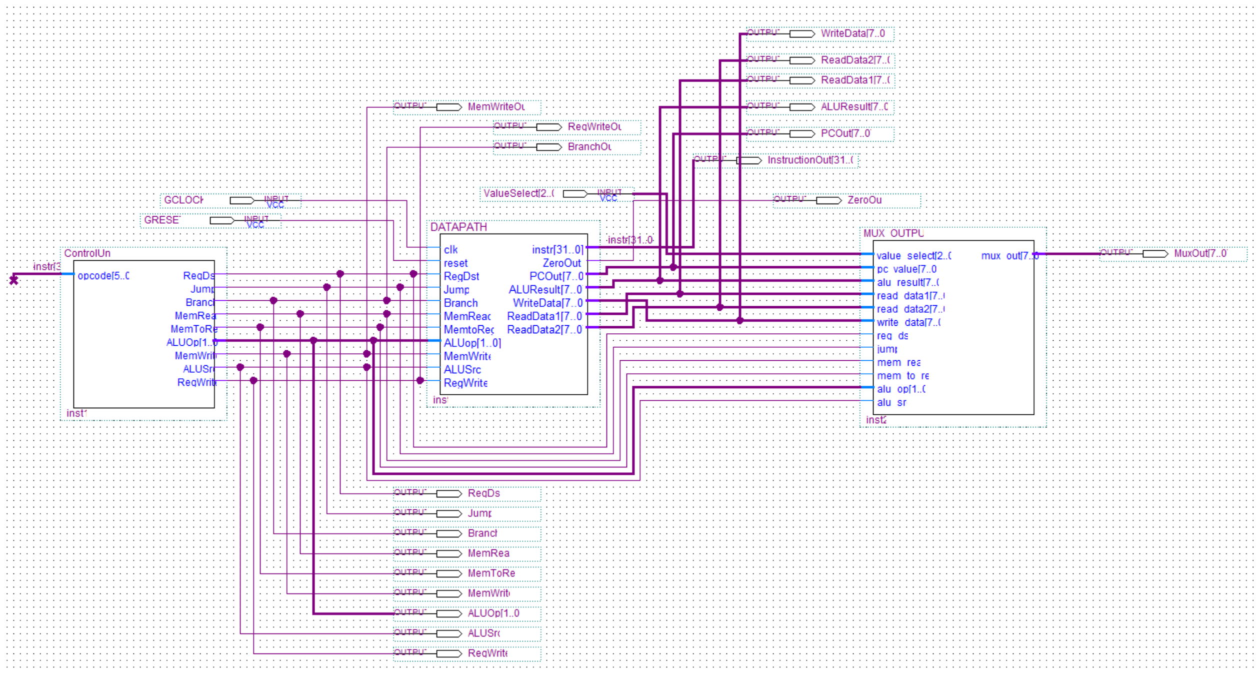


Figure 3.2 – Quartus II Top-Level Processor implementation

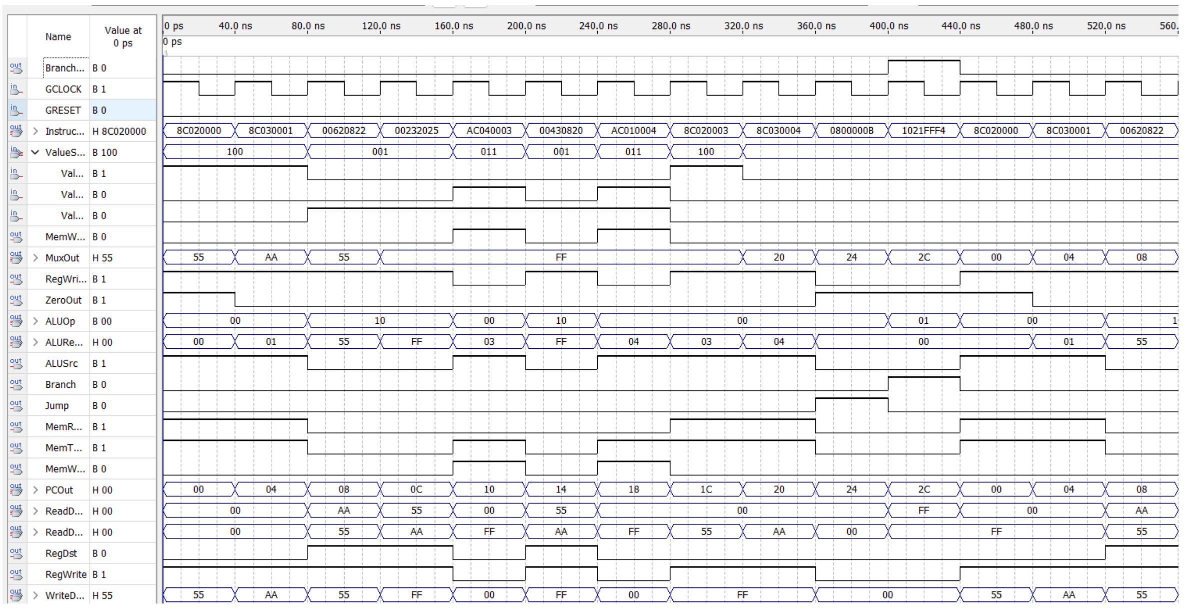


Figure 3.3 – Waveform Simulation of 32-bit instruction/8-bit Datapath single-cycle processor

Figure 3.1 shows the successful implementation of the revised design of the datapath derived in the previous steps. As can be seen the muxout outputs, corresponding to the correct register and memory variable that should be outputted based on the benchmark code provided. When 100 is selected by the mux, values 55 and AA are written into the registers t2 and t3 as expected in the first two clock cycles. When 001 is selected the ALU result is outputted with 55 for the first sub operation. The rest of the output is seen as FF for the read data signal (011) and the following ALU operations, which is also in accordance with the provided benchmark code. Finally, 24,44, and 48 are shown as the mux outputs, which correspond to the PC values at that point in execution.

Verification

Over the course of this lab, the successful synthesis of a datapath, with the required changes was achieved. A control path following the datapath was also successfully achieved, and the VHDL code was successfully implemented corresponding to the datapath and control path, combined into a top-level entity. The waveform simulations were also successfully outputting the correct values for all the used components; however, the design could not be successfully verified in this lab on the Altera DE2 board.

Discussion

Fortunately, the design of the datapath above successfully solved the problem at hand, and did not cause any discrepancies between the design and the actual implementation in VHDL. However, there were some challenges faced during the synthesis of the code onto the hardware display. The values displayed on the 7-segment decoder were not correct, and therefore the design was not successfully verified on the board.

Conclusion

In conclusion, this lab provided a deeper understanding of the design and functionality of a single-cycle processor. The planned datapath and control path were successfully converted into VHDL code, and a successful waveform simulation was generated with the correct input and output values. Unfortunately, the simulated code could not be synthesized onto the Altera DE2 board within the scope of this lab.

Appendix A

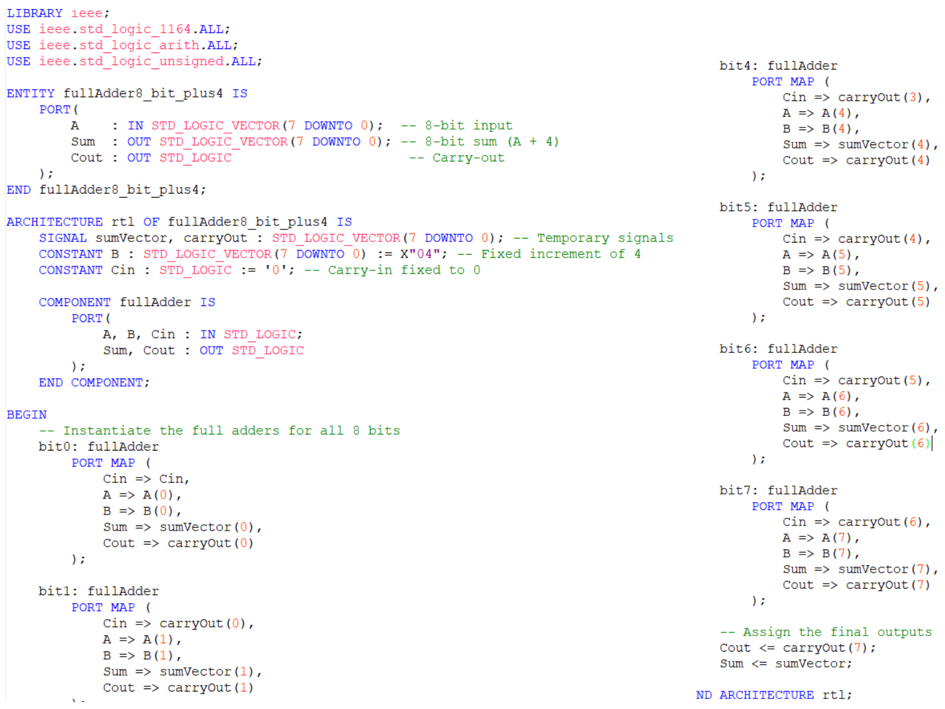
A-1 PC unit VHDL

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

ENTITY PC IS
    PORT (
        clk      : IN std_logic;  -- Clock signal
        reset    : IN std_logic;  -- Asynchronous reset
        pc_in    : IN std_logic_vector(7 DOWNTO 0);  -- Input address (from external logic)
        pc_out   : OUT std_logic_vector(7 DOWNTO 0)  -- Current PC Address
    );
END PC;

ARCHITECTURE structural OF PC IS
    COMPONENT DFF IS
        PORT (
            d      : IN std_logic_vector(7 DOWNTO 0);
            clk    : IN std_logic;
            reset  : IN std_logic;
            q      : OUT std_logic_vector(7 DOWNTO 0)
        );
    END COMPONENT;

    COMPONENT MUX2 IS
        PORT (
            a      : IN std_logic_vector(7 DOWNTO 0);
            b      : IN std_logic_vector(7 DOWNTO 0);
            sel    : IN std_logic;
            y      : OUT std_logic_vector(7 DOWNTO 0)
        );
    END COMPONENT;

    SIGNAL pc_reg : std_logic_vector(7 DOWNTO 0);
    SIGNAL next_pc : std_logic_vector(7 DOWNTO 0);

BEGIN
    -- Instantiate D Flip-Flop for PC Register
    PC_Reg: DFF PORT MAP (
        d => next_pc,
        clk => clk,
        reset => reset,
        q => pc_reg
    );

    -- Direct connection for next PC
    next_pc <= pc_in;

    -- Output the PC value
    pc_out <= pc_reg;
END structural;
```

A-2 Instruction Memory unit VHDL

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

ENTITY instruction_memory IS
    PORT (
        address   : IN  std_logic_vector(7 DOWNTO 0); -- 8-bit address for 256 locations
        instruction : OUT std_logic_vector(31 DOWNTO 0)
    );
END instruction_memory;

ARCHITECTURE rtl OF instruction_memory IS
    TYPE rom_type IS ARRAY (0 TO 255) OF std_logic_vector(31 DOWNTO 0);

    CONSTANT rom : rom_type := (
        -- Address 00 (0x00): lw $2, 0($0) ; Load value from memory(00) -> $t2 = 55
        0  => X"8C020000",
        -- Address 04 (0x04): lw $3, 1($0) ; Load value from memory(01) -> $t3 = AA
        4  => X"8C030001",
        -- Address 08 (0x08): sub $1, $3, $2 ; $t1 = $t3 - $t2 = AA - 55 = 55 (Fixed)
        8  => X"00620822",
        -- Address 12 (0x0C): or $4, $1, $3 ; $t4 = $t1 OR $t3 = 55 OR AA = FF
        12 => X"00232025",
        -- Address 16 (0x10): sw $4, 3($0) ; Store $t4 (FF) into memory(03)
        16 => X"AC040003",
        -- Address 20 (0x14): add $1, $2, $3 ; $t1 = $t2 + $t3 = 55 + AA = FF
        20 => X"00430820",
        -- Address 24 (0x18): sw $1, 4($0) ; Store $t1 (FF) into memory(04)
        24 => X"AC010004",
        -- Address 28 (0x1C): lw $2, 3($0) ; Load memory(03) -> $t2 = FF
        28 => X"8C020003",
        -- Address 32 (0x20): lw $3, 4($0) ; Load memory(04) -> $t3 = FF
        32 => X"8C030004",
        -- Address 36 (0x24): j 11 ; Jump to address 44 (0x2C)
        36 => X"0800000B",
        -- Address 40 (0x28): beq $1, $2, -8 ; Test if $t1 == $t2 (FF == FF?)
        40 => X"1022FFFD",
        -- Address 44 (0x2C):  beq $1, $1, -44 ; Loop back to beginning if $t1 == $t1
        44 => X"1021FFF4",

        -- Fill remaining with NOP (00000000)
        OTHERS => X"00000000"
    );
BEGIN
    instruction <= rom(CONV_INTEGER(address));
END rtl;
```

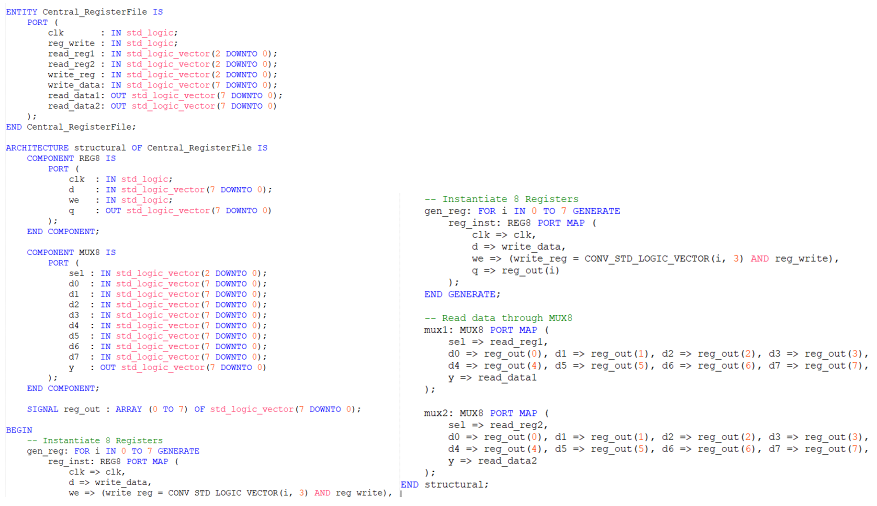
A-3 FullAdder8bits+4 unit VHDL



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

ENTITY fullAdder8_bit_plus4 IS
    PORT(
        A    : IN STD_LOGIC_VECTOR(7 DOWNTO 0);  -- 8-bit input
        Sum  : OUT STD_LOGIC_VECTOR(7 DOWNTO 0); -- 8-bit sum (A + 4)
        Cout : OUT STD_LOGIC                      -- Carry-out
    );
END fullAdder8_bit_plus4;

ARCHITECTURE rtl OF fullAdder8_bit_plus4 IS
    SIGNAL sumVector, carryOut : STD_LOGIC_VECTOR(7 DOWNTO 0); -- Temporary signals
    CONSTANT B : STD_LOGIC_VECTOR(7 DOWNTO 0) := X"04"; -- Fixed increment of 4
    CONSTANT Cin : STD_LOGIC := '0'; -- Carry-in fixed to 0

    COMPONENT fullAdder IS
        PORT(
            A, B, Cin : IN STD_LOGIC;
            Sum, Cout : OUT STD_LOGIC
        );
    END COMPONENT;

BEGIN
    -- Instantiate the full adders for all 8 bits
    bit0: fullAdder
        PORT MAP (
            Cin => Cin,
            A => A(0),
            B => B(0),
            Sum => sumVector(0),
            Cout => carryOut(0)
        );

    bit1: fullAdder
        PORT MAP (
            Cin => carryOut(0),
            A => A(1),
            B => B(1),
            Sum => sumVector(1),
            Cout => carryOut(1)
```

```vhdl
    bit4: fullAdder
        PORT MAP (
            Cin => carryOut(3),
            A => A(4),
            B => B(4),
            Sum => sumVector(4),
            Cout => carryOut(4)
        );

    bit5: fullAdder
        PORT MAP (
            Cin => carryOut(4),
            A => A(5),
            B => B(5),
            Sum => sumVector(5),
            Cout => carryOut(5)
        );

    bit6: fullAdder
        PORT MAP (
            Cin => carryOut(5),
            A => A(6),
            B => B(6),
            Sum => sumVector(6),
            Cout => carryOut(6)
        );

    bit7: fullAdder
        PORT MAP (
            Cin => carryOut(6),
            A => A(7),
            B => B(7),
            Sum => sumVector(7),
            Cout => carryOut(7)
        );

    -- Assign the final outputs
    Cout <= carryOut(7);
    Sum <= sumVector;

ND ARCHITECTURE rtl;
```

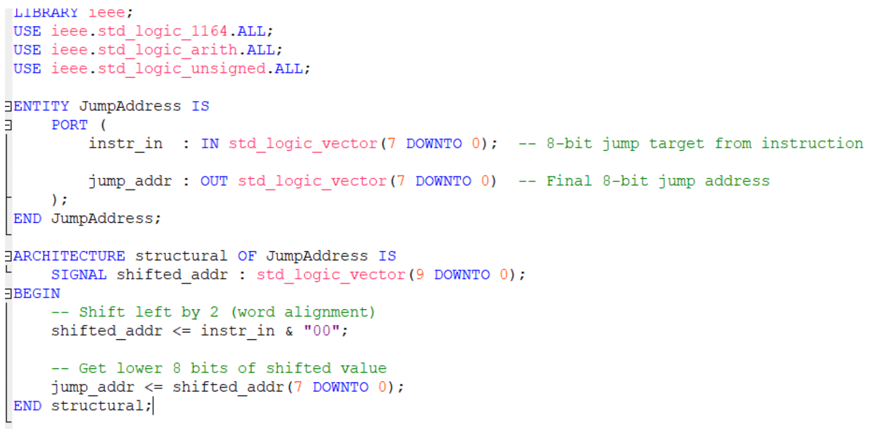
A-4 Central_RegisterFile unit VHDL



```vhdl
ENTITY Central_RegisterFile IS
    PORT (
        clk       : IN std_logic;
        reg_write : IN std_logic;
        read_reg1 : IN std_logic_vector(2 DOWNTO 0);
        read_reg2 : IN std_logic_vector(2 DOWNTO 0);
        write_reg : IN std_logic_vector(2 DOWNTO 0);
        write_data: IN std_logic_vector(7 DOWNTO 0);
        read_data1: OUT std_logic_vector(7 DOWNTO 0);
        read_data2: OUT std_logic_vector(7 DOWNTO 0)
    );
END Central_RegisterFile;

ARCHITECTURE structural OF Central_RegisterFile IS
    COMPONENT REG8 IS
        PORT (
            clk  : IN std_logic;
            d    : IN std_logic_vector(7 DOWNTO 0);
            we   : IN std_logic;
            q    : OUT std_logic_vector(7 DOWNTO 0)
        );
    END COMPONENT;

    COMPONENT MUX8 IS
        PORT (
            sel : IN std_logic_vector(2 DOWNTO 0);
            d0  : IN std_logic_vector(7 DOWNTO 0);
            d1  : IN std_logic_vector(7 DOWNTO 0);
            d2  : IN std_logic_vector(7 DOWNTO 0);
            d3  : IN std_logic_vector(7 DOWNTO 0);
            d4  : IN std_logic_vector(7 DOWNTO 0);
            d5  : IN std_logic_vector(7 DOWNTO 0);
            d6  : IN std_logic_vector(7 DOWNTO 0);
            d7  : IN std_logic_vector(7 DOWNTO 0);
            y   : OUT std_logic_vector(7 DOWNTO 0)
        );
    END COMPONENT;

    SIGNAL reg_out : ARRAY (0 TO 7) OF std_logic_vector(7 DOWNTO 0);

BEGIN
    -- Instantiate 8 Registers
    gen_reg: FOR i IN 0 TO 7 GENERATE
        reg_inst: REG8 PORT MAP (
            clk => clk,
            d => write_data,
            we => (write_reg = CONV_STD_LOGIC_VECTOR(i, 3) AND reg_write),
```

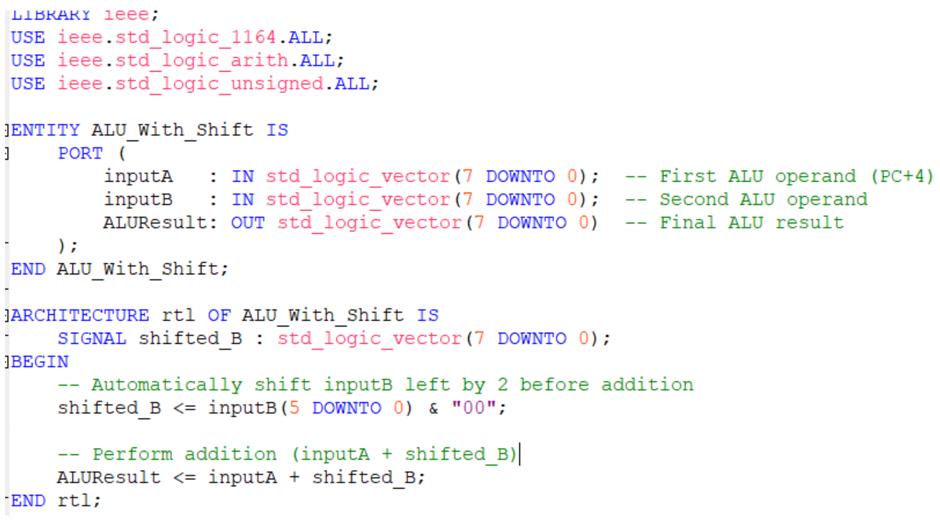
```vhdl
    -- Instantiate 8 Registers
    gen_reg: FOR i IN 0 TO 7 GENERATE
        reg_inst: REG8 PORT MAP (
            clk => clk,
            d => write_data,
            we => (write_reg = CONV_STD_LOGIC_VECTOR(i, 3) AND reg_write),
            q => reg_out(i)
        );
    END GENERATE;

    -- Read data through MUX8
    mux1: MUX8 PORT MAP (
        sel => read_reg1,
        d0 => reg_out(0), d1 => reg_out(1), d2 => reg_out(2), d3 => reg_out(3),
        d4 => reg_out(4), d5 => reg_out(5), d6 => reg_out(6), d7 => reg_out(7),
        y => read_data1
    );

    mux2: MUX8 PORT MAP (
        sel => read_reg2,
        d0 => reg_out(0), d1 => reg_out(1), d2 => reg_out(2), d3 => reg_out(3),
        d4 => reg_out(4), d5 => reg_out(5), d6 => reg_out(6), d7 => reg_out(7),
        y => read_data2
    );
END structural;
```

A-5 Jump Address unit VHDL



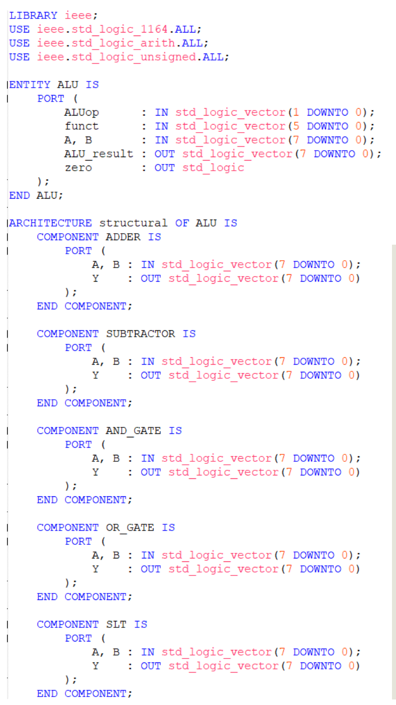
```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY JumpAddress IS
    PORT (
        instr_in  : IN std_logic_vector(7 DOWNTO 0);  -- 8-bit jump target from instruction

        jump_addr : OUT std_logic_vector(7 DOWNTO 0)  -- Final 8-bit jump address
    );
END JumpAddress;

ARCHITECTURE structural OF JumpAddress IS
    SIGNAL shifted_addr : std_logic_vector(9 DOWNTO 0);
BEGIN
    -- Shift left by 2 (word alignment)
    shifted_addr <= instr_in & "00";

    -- Get lower 8 bits of shifted value
    jump_addr <= shifted_addr(7 DOWNTO 0);
END structural;
```

A-6 ALU_with_shift unit VHDL

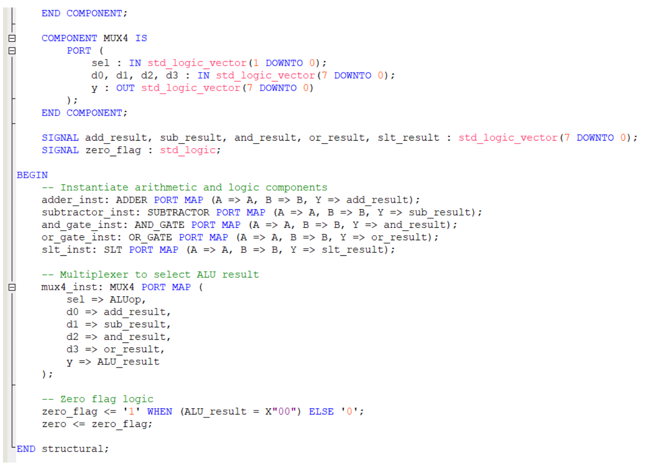


```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY ALU_With_Shift IS
    PORT (
        inputA   : IN std_logic_vector(7 DOWNTO 0);  -- First ALU operand (PC+4)
        inputB   : IN std_logic_vector(7 DOWNTO 0);  -- Second ALU operand
        ALUResult: OUT std_logic_vector(7 DOWNTO 0)  -- Final ALU result
    );
END ALU_With_Shift;

ARCHITECTURE rtl OF ALU_With_Shift IS
    SIGNAL shifted_B : std_logic_vector(7 DOWNTO 0);
BEGIN
    -- Automatically shift inputB left by 2 before addition
    shifted_B <= inputB(5 DOWNTO 0) & "00";

    -- Perform addition (inputA + shifted_B)
    ALUResult <= inputA + shifted_B;
END rtl;
```

A-7 ALU unit VHDL

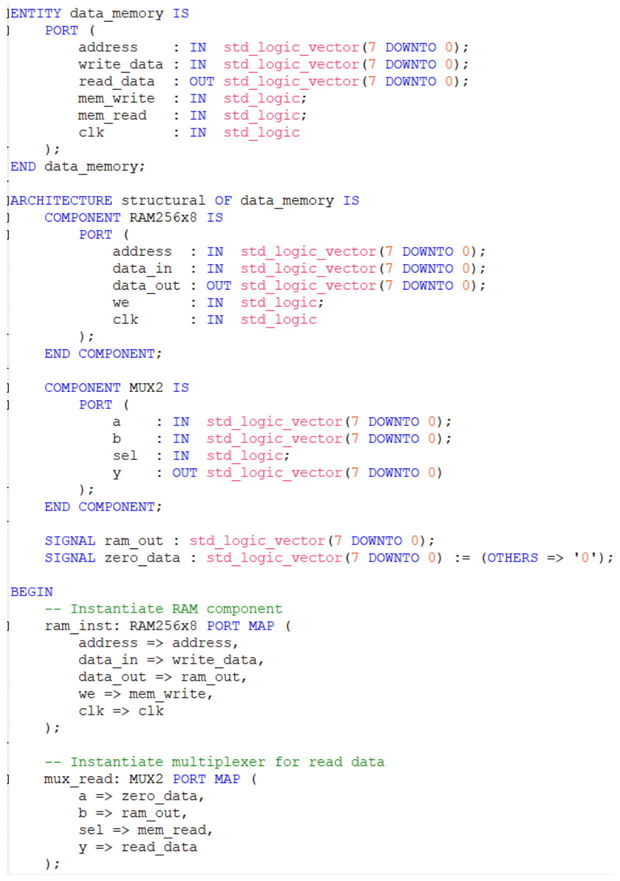


```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY ALU IS
    PORT (
        ALUop      : IN std_logic_vector(1 DOWNTO 0);
        funct      : IN std_logic_vector(5 DOWNTO 0);
        A, B       : IN std_logic_vector(7 DOWNTO 0);
        ALU_result : OUT std_logic_vector(7 DOWNTO 0);
        zero       : OUT std_logic
    );
END ALU;

ARCHITECTURE structural OF ALU IS
    COMPONENT ADDER IS
        PORT (
            A, B : IN std_logic_vector(7 DOWNTO 0);
            Y    : OUT std_logic_vector(7 DOWNTO 0)
        );
    END COMPONENT;

    COMPONENT SUBTRACTOR IS
        PORT (
            A, B : IN std_logic_vector(7 DOWNTO 0);
            Y    : OUT std_logic_vector(7 DOWNTO 0)
        );
    END COMPONENT;

    COMPONENT AND_GATE IS
        PORT (
            A, B : IN std_logic_vector(7 DOWNTO 0);
            Y    : OUT std_logic_vector(7 DOWNTO 0)
        );
    END COMPONENT;

    COMPONENT OR_GATE IS
        PORT (
            A, B : IN std_logic_vector(7 DOWNTO 0);
            Y    : OUT std_logic_vector(7 DOWNTO 0)
        );
    END COMPONENT;

    COMPONENT SLT IS
        PORT (
            A, B : IN std_logic_vector(7 DOWNTO 0);
            Y    : OUT std_logic_vector(7 DOWNTO 0)
        );
    END COMPONENT;
```



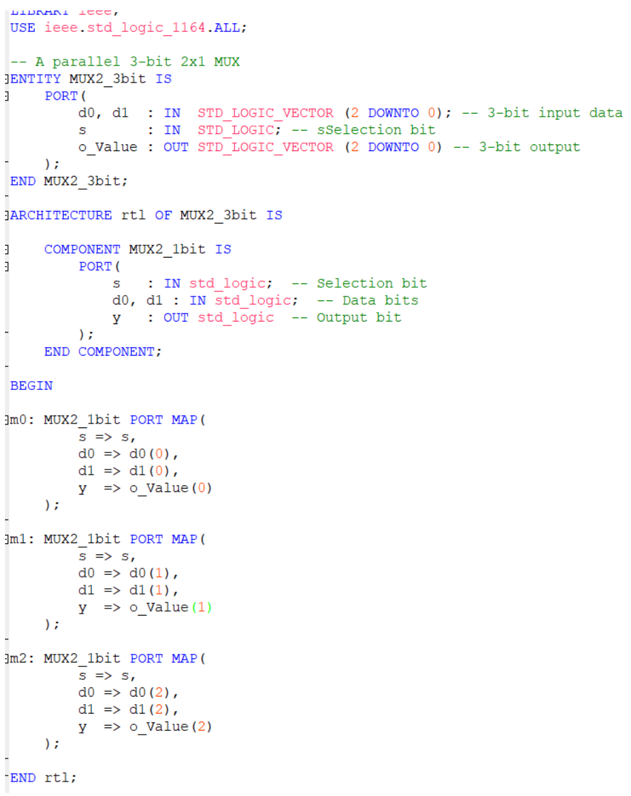
```
    END COMPONENT;

    COMPONENT MUX4 IS
        PORT (
            sel : IN std_logic_vector(1 DOWNTO 0);
            d0, d1, d2, d3 : IN std_logic_vector(7 DOWNTO 0);
            y : OUT std_logic_vector(7 DOWNTO 0)
        );
    END COMPONENT;

    SIGNAL add_result, sub_result, and_result, or_result, slt_result : std_logic_vector(7 DOWNTO 0);
    SIGNAL zero_flag : std_logic;

BEGIN
    -- Instantiate arithmetic and logic components
    adder_inst: ADDER PORT MAP (A => A, B => B, Y => add_result);
    subtractor_inst: SUBTRACTOR PORT MAP (A => A, B => B, Y => sub_result);
    and_gate_inst: AND_GATE PORT MAP (A => A, B => B, Y => and_result);
    or_gate_inst: OR_GATE PORT MAP (A => A, B => B, Y => or_result);
    slt_inst: SLT PORT MAP (A => A, B => B, Y => slt_result);

    -- Multiplexer to select ALU result
    mux4_inst: MUX4 PORT MAP (
        sel => ALUop,
        d0 => add_result,
        d1 => sub_result,
        d2 => and_result,
        d3 => or_result,
        y => ALU_result
    );

    -- Zero flag logic
    zero_flag <= '1' WHEN (ALU_result = X"00") ELSE '0';
    zero <= zero_flag;

END structural;
```

A-8 Data Memory unit VHDL



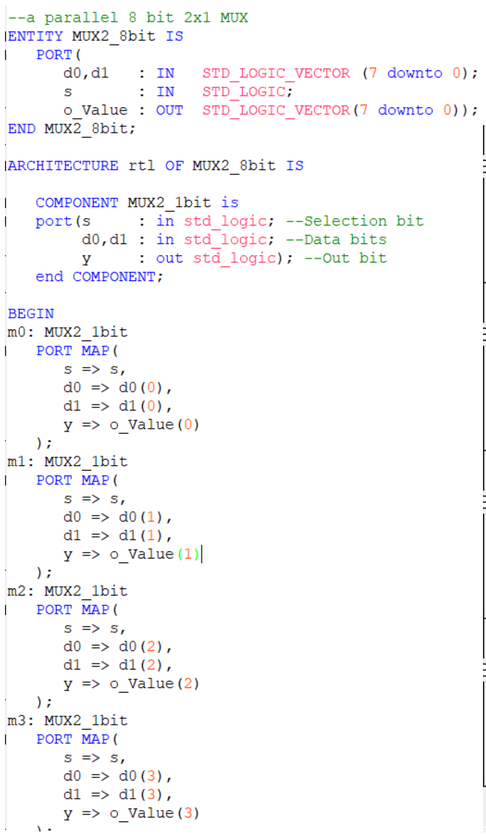
```
ENTITY data_memory IS
    PORT (
        address    : IN  std_logic_vector(7 DOWNTO 0);
        write_data : IN  std_logic_vector(7 DOWNTO 0);
        read_data  : OUT std_logic_vector(7 DOWNTO 0);
        mem_write  : IN  std_logic;
        mem_read   : IN  std_logic;
        clk        : IN  std_logic
    );
END data_memory;

ARCHITECTURE structural OF data_memory IS
    COMPONENT RAM256x8 IS
        PORT (
            address  : IN  std_logic_vector(7 DOWNTO 0);
            data_in  : IN  std_logic_vector(7 DOWNTO 0);
            data_out : OUT std_logic_vector(7 DOWNTO 0);
            we       : IN  std_logic;
            clk      : IN  std_logic
        );
    END COMPONENT;

    COMPONENT MUX2 IS
        PORT (
            a    : IN  std_logic_vector(7 DOWNTO 0);
            b    : IN  std_logic_vector(7 DOWNTO 0);
            sel  : IN  std_logic;
            y    : OUT std_logic_vector(7 DOWNTO 0)
        );
    END COMPONENT;

    SIGNAL ram_out : std_logic_vector(7 DOWNTO 0);
    SIGNAL zero_data : std_logic_vector(7 DOWNTO 0) := (OTHERS => '0');

BEGIN
    -- Instantiate RAM component
    ram_inst: RAM256x8 PORT MAP (
        address => address,
        data_in => write_data,
        data_out => ram_out,
        we => mem_write,
        clk => clk
    );

    -- Instantiate multiplexer for read data
    mux_read: MUX2 PORT MAP (
        a => zero_data,
        b => ram_out,
        sel => mem_read,
        y => read_data
    );
```

A-9 MUX 3bits unit VHDL

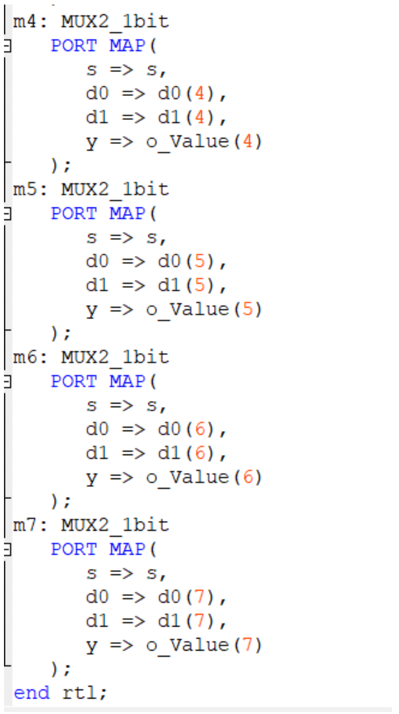


```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

-- A parallel 3-bit 2x1 MUX
ENTITY MUX2_3bit IS
    PORT(
        d0, d1   : IN  STD_LOGIC_VECTOR (2 DOWNTO 0); -- 3-bit input data
        s        : IN  STD_LOGIC; -- sSelection bit
        o_Value : OUT STD_LOGIC_VECTOR (2 DOWNTO 0) -- 3-bit output
    );
END MUX2_3bit;

ARCHITECTURE rtl OF MUX2_3bit IS

    COMPONENT MUX2_1bit IS
        PORT(
            s    : IN std_logic;  -- Selection bit
            d0, d1 : IN std_logic;  -- Data bits
            y    : OUT std_logic  -- Output bit
        );
    END COMPONENT;

BEGIN

m0: MUX2_1bit PORT MAP(
        s => s,
        d0 => d0(0),
        d1 => d1(0),
        y  => o_Value(0)
    );

m1: MUX2_1bit PORT MAP(
        s => s,
        d0 => d0(1),
        d1 => d1(1),
        y  => o_Value(1)
    );

m2: MUX2_1bit PORT MAP(
        s => s,
        d0 => d0(2),
        d1 => d1(2),
        y  => o_Value(2)
    );

END rtl;
```

A-10 MUX 8bits unit VHDL

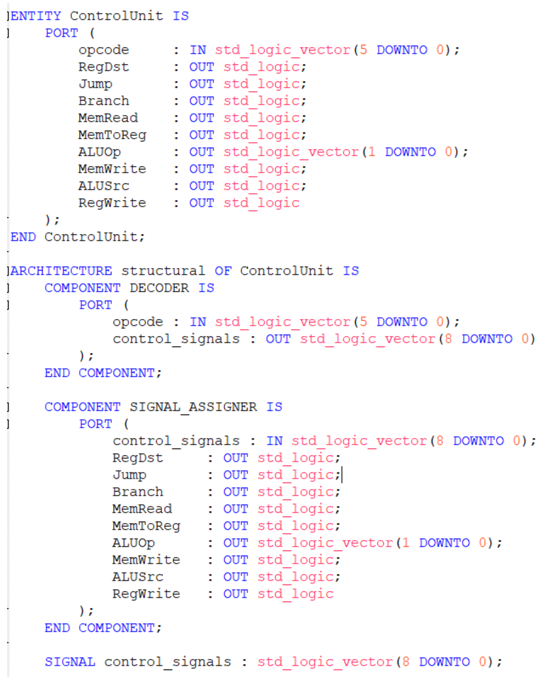


```
--a parallel 8 bit 2x1 MUX
ENTITY MUX2_8bit IS
    PORT(
        d0,d1    : IN   STD_LOGIC_VECTOR (7 downto 0);
        s        : IN   STD_LOGIC;
        o_Value  : OUT  STD_LOGIC_VECTOR(7 downto 0));
END MUX2_8bit;

ARCHITECTURE rtl OF MUX2_8bit IS

    COMPONENT MUX2_1bit is
    port(s      : in std_logic; --Selection bit
          d0,d1 : in std_logic; --Data bits
          y     : out std_logic); --Out bit
    end COMPONENT;

BEGIN
m0: MUX2_1bit
    PORT MAP(
        s => s,
        d0 => d0(0),
        d1 => d1(0),
        y => o_Value(0)
    );
m1: MUX2_1bit
    PORT MAP(
        s => s,
        d0 => d0(1),
        d1 => d1(1),
        y => o_Value(1)
    );
m2: MUX2_1bit
    PORT MAP(
        s => s,
        d0 => d0(2),
        d1 => d1(2),
        y => o_Value(2)
    );
m3: MUX2_1bit
    PORT MAP(
        s => s,
        d0 => d0(3),
        d1 => d1(3),
        y => o_Value(3)
    );
```

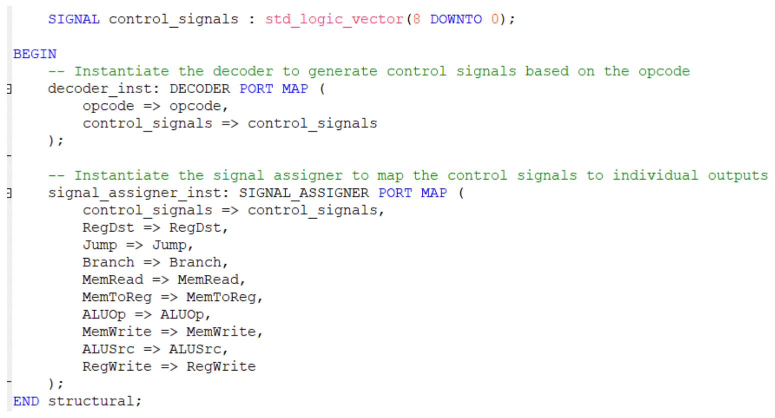


```
m4: MUX2_1bit
    PORT MAP(
        s => s,
        d0 => d0(4),
        d1 => d1(4),
        y => o_Value(4)
    );
m5: MUX2_1bit
    PORT MAP(
        s => s,
        d0 => d0(5),
        d1 => d1(5),
        y => o_Value(5)
    );
m6: MUX2_1bit
    PORT MAP(
        s => s,
        d0 => d0(6),
        d1 => d1(6),
        y => o_Value(6)
    );
m7: MUX2_1bit
    PORT MAP(
        s => s,
        d0 => d0(7),
        d1 => d1(7),
        y => o_Value(7)
    );
end rtl;
```

A-11 Control unit VHDL



```
ENTITY ControlUnit IS
    PORT (
        opcode      : IN std_logic_vector(5 DOWNTO 0);
        RegDst      : OUT std_logic;
        Jump        : OUT std_logic;
        Branch      : OUT std_logic;
        MemRead     : OUT std_logic;
        MemToReg    : OUT std_logic;
        ALUOp       : OUT std_logic_vector(1 DOWNTO 0);
        MemWrite    : OUT std_logic;
        ALUSrc      : OUT std_logic;
        RegWrite    : OUT std_logic
    );
END ControlUnit;

ARCHITECTURE structural OF ControlUnit IS
    COMPONENT DECODER IS
        PORT (
            opcode : IN std_logic_vector(5 DOWNTO 0);
            control_signals : OUT std_logic_vector(8 DOWNTO 0)
        );
    END COMPONENT;

    COMPONENT SIGNAL_ASSIGNER IS
        PORT (
            control_signals : IN std_logic_vector(8 DOWNTO 0);
            RegDst     : OUT std_logic;
            Jump       : OUT std_logic;
            Branch     : OUT std_logic;
            MemRead    : OUT std_logic;
            MemToReg   : OUT std_logic;
            ALUOp      : OUT std_logic_vector(1 DOWNTO 0);
            MemWrite   : OUT std_logic;
            ALUSrc     : OUT std_logic;
            RegWrite   : OUT std_logic
        );
    END COMPONENT;

    SIGNAL control_signals : std_logic_vector(8 DOWNTO 0);
```



```
    SIGNAL control_signals : std_logic_vector(8 DOWNTO 0);

BEGIN
    -- Instantiate the decoder to generate control signals based on the opcode
    decoder_inst: DECODER PORT MAP (
        opcode => opcode,
        control_signals => control_signals
    );

    -- Instantiate the signal assigner to map the control signals to individual outputs
    signal_assigner_inst: SIGNAL_ASSIGNER PORT MAP (
        control_signals => control_signals,
        RegDst => RegDst,
        Jump => Jump,
        Branch => Branch,
        MemRead => MemRead,
        MemToReg => MemToReg,
        ALUOp => ALUOp,
        MemWrite => MemWrite,
        ALUSrc => ALUSrc,
        RegWrite => RegWrite
    );
END structural;
```