Lab 2: Single-Cycle RISC Processor

CEG3156 A – Computer Systems Design
Winter 2021
School of Electrical Engineering and Computer Science
University of Ottawa

Course Coordinator: Dr. Rami Abielmona
Teaching Assistant: Joel Muteba

Punit Daga
300079444

Pavly Saleh
300012936

Experiment Date: February 24th, 2021
Report Due Date: March 24th, 2021

Table of Contents

Introduction

Nowadays, we have a small selection of design styles as applied to processors. One specific design style that we have mainly studied and analyzed is called the Reduced Instruction Set Computer (RISC). RISC is a computer with a small, highly optimized set of instructions, rather than the more specialized set often found in other types of architecture, such as in a complex instruction set computer (CISC). The main distinguishing feature of RISC is that the instruction set is optimized with a large number of registers and a highly regular instruction pipeline, allowing a low number of clock cycles per instruction. One specific example that was studied in detail is the MIPS processor, which was developed in the 1980s. MIPS processors are used in embedded systems such as residential gateways and routers. Originally, MIPS was designed for general-purpose computing. During the 1980s and 1990s, MIPS processors for personal, workstation, and server computers were used by many companies. Historically, video game consoles such as the Nintendo 64, Sony PlayStation, PlayStation 2, and PlayStation Portable used MIPS processors.

Theoretical Part

The lab manual provides the background how to implement the MIPS processor. This MIPS processor includes a couple functionalities such as memory-referencing instructions, arithmetic logic instructions, as well as control flow instructions. As this is a single-cycle processor that we will be implementing in this lab, our implementation will not include all integer operations supported by the MIPS ISA, such as multiply and divide, nor will it support floating-point operations, which are also supported by the MIPS processor. However, the design will be modular and expandable in order to support additional operations if the need arises, such as the branch if not equal instruction for example.

MIPS Instruction Format

MIPS instruction set defines three primary types of instructions, those being R-type, I-type, and J-type instructions. R-type instructions are used in arithmetic and logical nature, such as add, sub, and. A typical instruction example is add $t1, $t2, $t3. In this example, the R-type instruction directs the processor to add the values and contents of registers t1 and t2 together, and store the resulting contents in register t3. I-type instructions are the memory-referencing type, such as load and store word instructions. A typical instruction example is lw $t1 , offset_value($t2). In this example, the I-type instruction directs the processor to add the offset_value to the values and contents of register t2. The result of this will then create the address in the data memory of the word that can be accessed. Since the instruction used here was lw, which stands for load word, the contents of that word in memory are transferred to the register t1. J-type instructions are control-flow in nature, such as branch and jump. A typical instruction example is beg $t1, $t2, 25. In this example, the J-type instruction directs the processor to compare the contents of registers t1 and t2. If they are equal, it performs a branch to a location in memory, 25 instructions below the current PC. Figure 1 below illustrates the internal representation of all the instructions mentioned above.

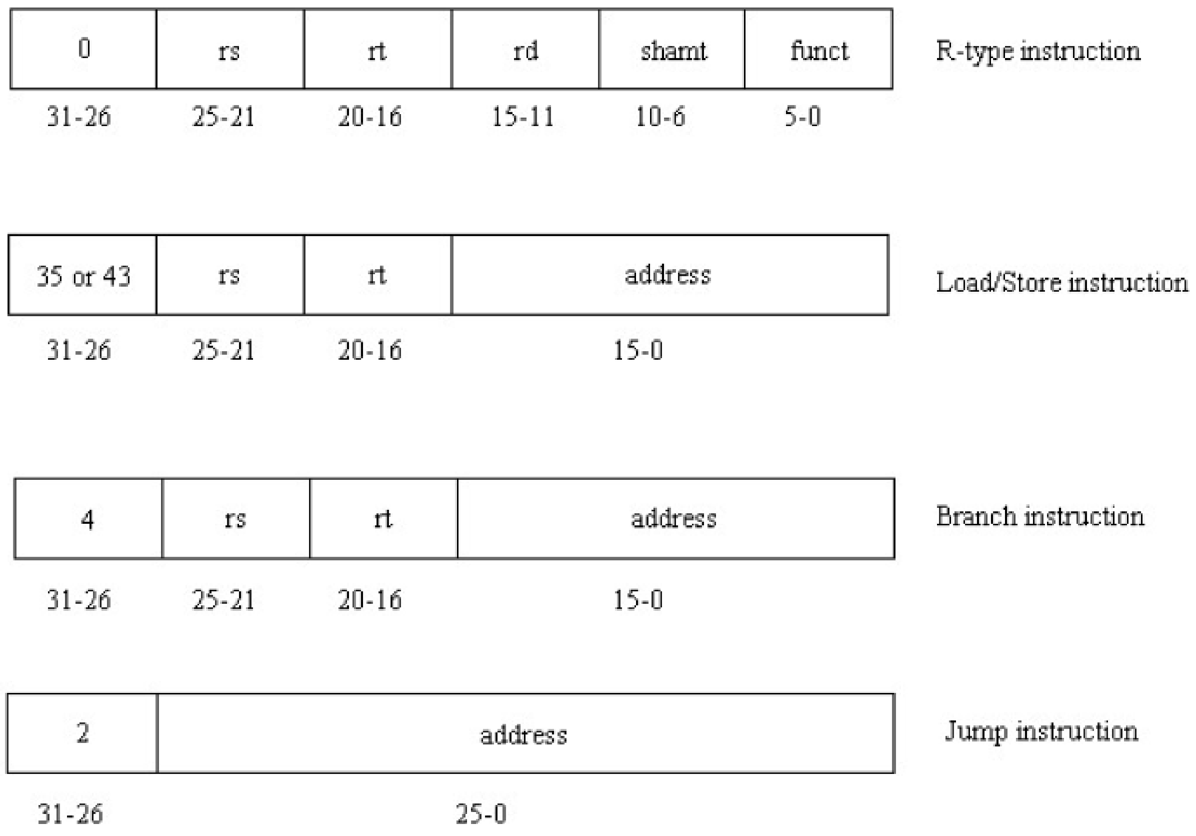


Figure 1: Internal representation of all the instructions Fetch and Increment

Before we begin, we must first ensure that the PC is fetched properly, as well as incremented by 4. This is because our instruction memory is 32-bits wide. The PC is a register that holds the address of the next instruction to be executed and is loadable on every clock cycle. Hence, it is updated at every clock cycle.

R-type Instruction Datapath

In order to get the full datapath for our single-cycle processor, we must first deconstruct and build the datapath for each component of the processor. Starting with the R-type instruction, it is composed of two basic building blocks, those being an ALU and a register file. The register file has two read ports and one write port. This allows three simultaneous accesses to the register file. The ALU consists of a 32-bit MIPS ALU, which is capable of addition, subtraction, AND/OR logical operations, and a comparison operation. The datapath can be seen in Figure 2 below.

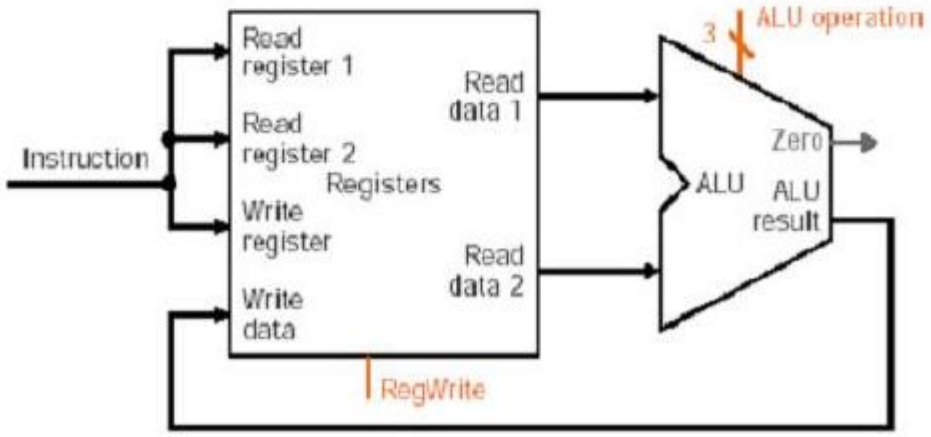


Figure 2: R-type Instruction Datapath

I-type Instruction Datapath

The I-type instruction is composed of four basic building blocks, those being an ALU, a register file, a data memory and a sign-extension unit. The register file is used to decode the register's contents and add those contents to the sign-extended offset value. The output of the ALU, which is the address, is sent to the data memory unit. The datapath can be seen in Figure 3 below.

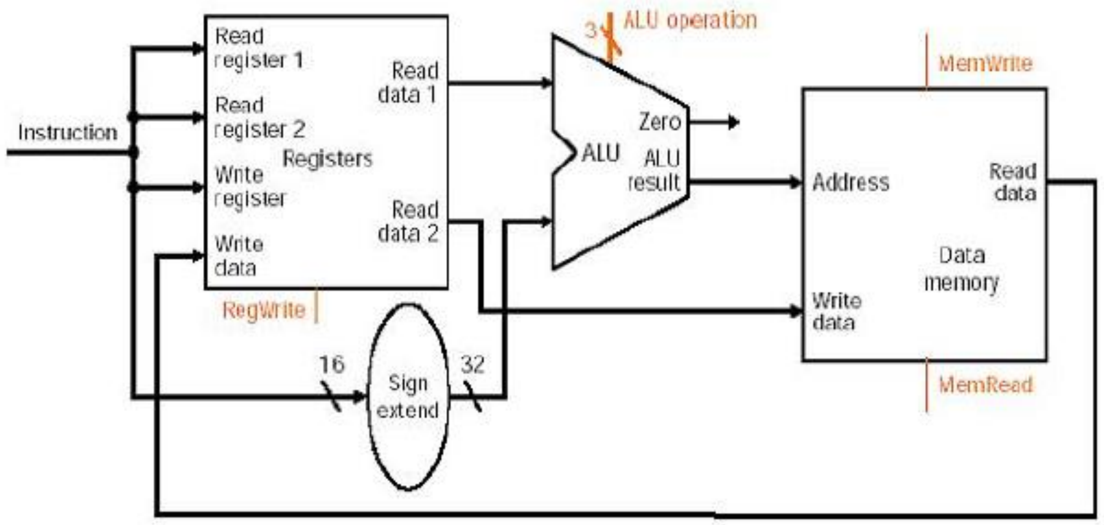


Figure 3: I-type Instruction Datapath

J-type Instruction Datapath

The J-type instruction is composed of three basic building blocks, those being an ALU, a register file, and a sign-extension unit. The register file outputs the contents of the two operands in the instruction. The ALU compares the two numbers. The datapath can be seen in Figure 4 below.

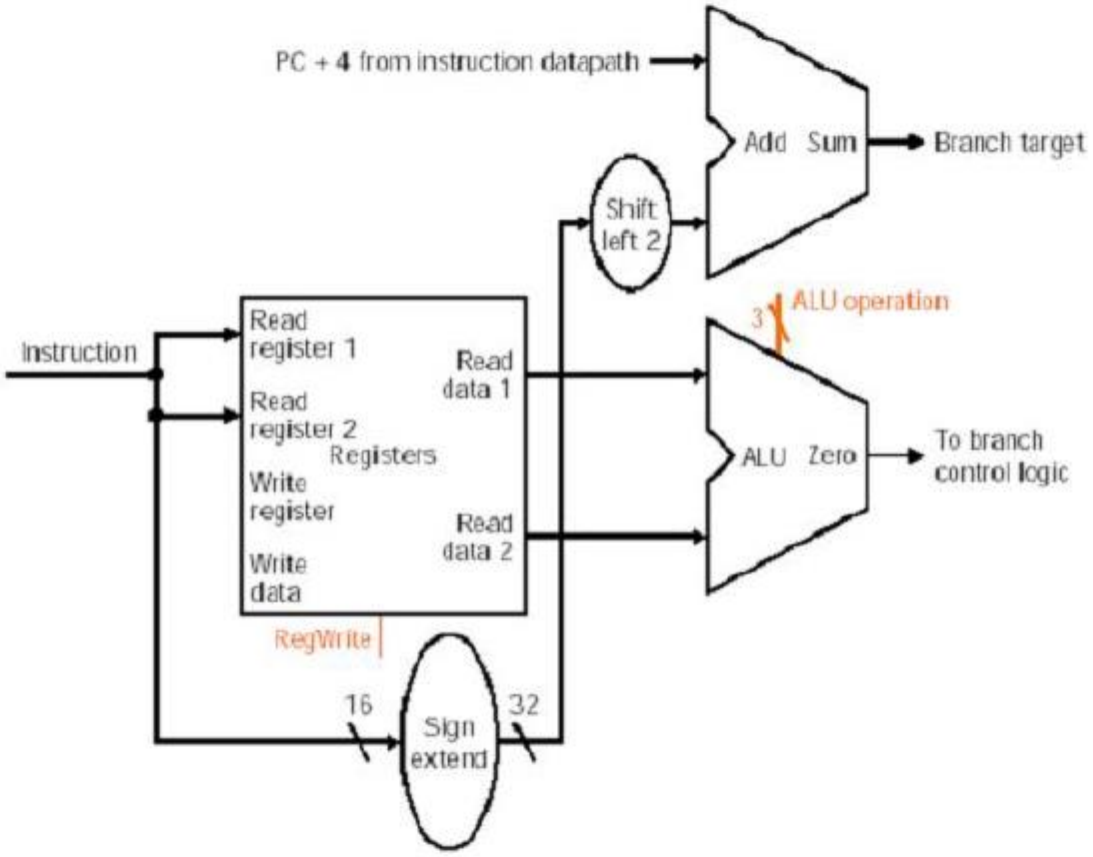


Figure 4: J-type Instruction Datapath

Design Part

Complete Datapath

Now that we have an idea of the datapaths of each instruction type, we can now construct the complete datapath of our processor. By adding the 3 datapaths above together, as well as the addition of some multiplexers and control signals, as well as the ALU control unit and main control unit, we now have built the datapath of our single-cycle RISC processor. The complete datapath can be seen below in Figure 5.

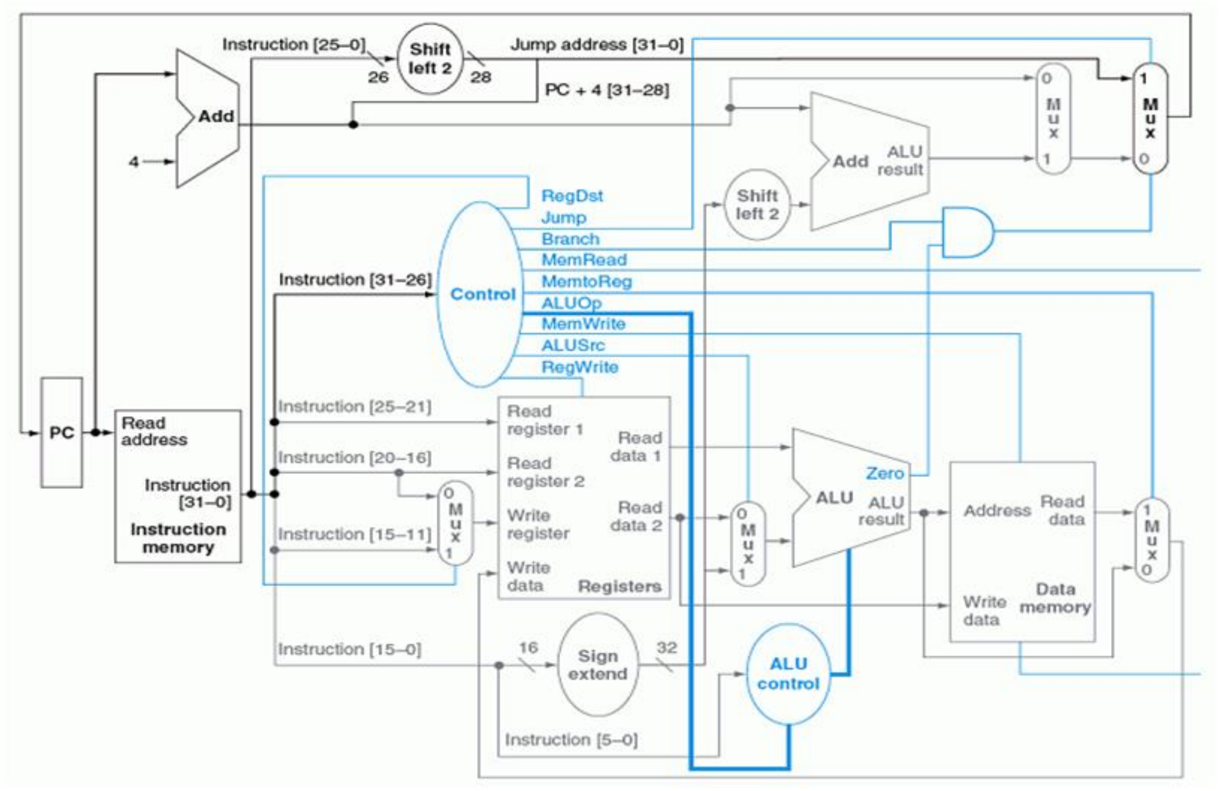


Figure 5: Single-cycle RISC processor datapath

In this single-cycle processor however, all instructions must begin and end in one clock cycle. Also, all values must reach a stable state. In addition, the cycle time is determined by the longest path delay, as we have seen in class. Lastly, the control logic is not systematically defined.

Branch if not equal instruction

As asked of us in the prelab, we have modified the datapath of the single-cycle processor to support a new instruction. That new instruction is the branch if not equal (bne) instruction. The modified datapath can be found below in Figure 6. The two register numbers which are part of the BNE instruction are passed into the Register File which then passes the data to the ALU. If they are equal then the zero flag is set. The instruction decode unit determines whether the branch flag is set based on a whole bunch of control signals which are already present (the aforementioned 0 flag, and a bunch of bits in the instruction opcode). Then the next address for the program counter is calculated based on the ALU on the top which handles adding the offset address to the current address. Figure 7 illustrates the ALUOp signals that the main control produces. ALUop code 11 is not used, thus bne can be defined when ALUOp = 11, then the

ALU control input would be 1110 which would need to also do subtract (same as 0110). Now in the case of bne, we know that ALUop will be 11 and for the PC to be set, the 'zero' signal should be deasserted (meaning they are not equal). So now it should be obvious that we can use the following logic to determine the result of bne: ALUop1 AND ALUop2 AND NOT(‘0’). This will be asserted when PC should be set based on the output of a bne operation. The output should be used to control the same mux as the AND gate that is already in your diagram.

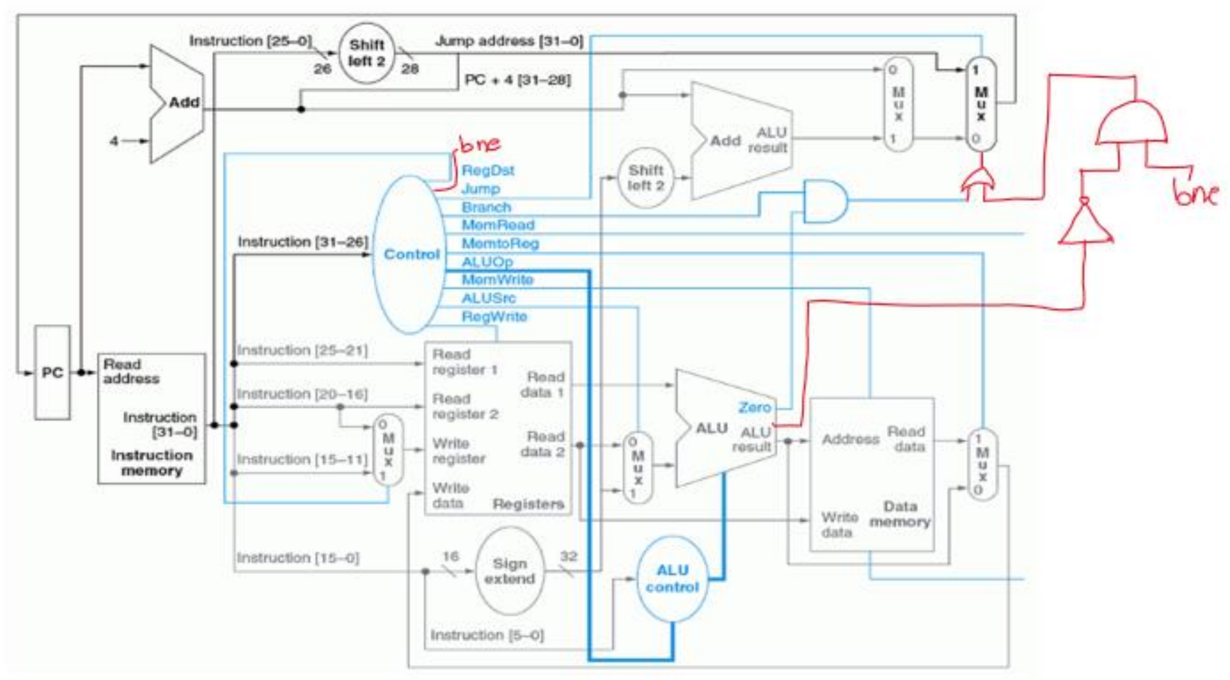
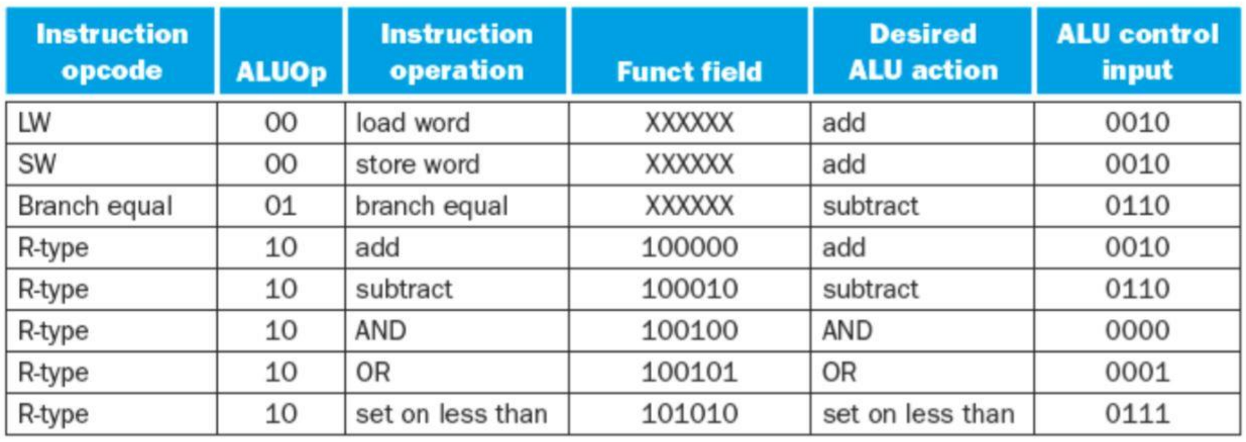


Figure 6: Modifed single-cycle RISC processor datapath to include bne instruction



Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111

Figure 7: ALUOp signals that the main control produces

In order to design our single-cycle processor, we have split it into five parts: the control unit, the data memory, the register file, the ALU and the top file, which links all the parts together.

Control Unit

The control unit is the main component of the processor that sends signals to activate certain components. The 9 different control signals that are sent are determined by decoding the opcode from each instruction. The opcode is defined as the first 6 bits of each instruction. An opcode of 0 identifies the instruction as being R-type, 35 for load word, 43 for store word, 4 for branch and 2 for jump instructions. The VHDL code for the control unit can be found below in Figure 8.

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

ENTITY Control_Unit IS
        PORT (
                i_OP : IN STD_LOGIC_VECTOR (5 DOWNTO 0);
                o_RegDst, o_Jump, O_Branch, o_MemRead : OUT STD_LOGIC;
                o_MemtoReg, o_MemWrite, o_ALUsrc, o_RegWrite : OUT STD_LOGIC;
                o_aluOP : OUT STD_LOGIC_VECTOR (1 DOWNTO 0));
END;

ARCHITECTURE struct OF Control_Unit IS
        SIGNAL int_Rtype, int_lw, int_sw, int_beq, int_jump : STD_LOGIC;
BEGIN
        int_Rtype <= NOT(i_OP(5)) AND NOT(i_OP(4)) AND NOT (i_OP(3)) AND NOT(i_OP(2)) AND NOT (i_OP(1)) AND
NOT(i_OP(0));
        int_lw <= i_OP(5) AND NOT(i_OP(4)) AND NOT(i_OP(3)) AND NOT(i_OP(2)) AND i_OP(1) AND i_OP(0);
        int_sw <= i_OP(5) AND NOT(i_OP(4)) AND i_OP(3) AND NOT(i_OP(2)) AND i_OP(1) AND i_OP(0);
        int_beq <= NOT(i_OP(5)) AND NOT(i_OP(4)) AND NOT (i_OP(3)) AND i_OP(2) AND NOT (i_OP(1)) AND NOT(i_OP(0));
        int_jump <= NOT(i_OP(5)) AND NOT(i_OP(4)) AND NOT (i_OP(3)) AND NOT(i_OP(2)) AND i_OP(1) AND NOT(i_OP(0));

        o_RegDst <= int_Rtype;
        o_jump <= int_jump;
        O_Branch <= int_beq;
        o_MemRead <= int_lw;
        o_MemtoReg <= int_lw;
        o_MemWrite <= int_sw;
        o_ALUsrc <= int_sw OR int_lw;
        o_RegWrite <= int_lw OR int_Rtype;
        o_aluOP(0) <= int_beq;
        o_aluOP(1) <= int_Rtype;
END struct;
```

Figure 8: VHDL implementation of the control unit

Data Memory

The data memory holds values in memory that can be read and written in the processor. It uses the “altsyncram” function from Altera. It has an address length of 8 bits, 28 = 256 addresses, and each memory location is 8 bits wide. The 8 bit address is provided by the ALU result, it uses the same address port for both reading and writing. The write data comes from the Register File’s ReadData2 port. The 8 bit read data is outputted into a mux for choosing which data goes to the register write. Control signals are MemRead and MemWrite, indicating whether the data memory is reading from or writing to memory. The VHDL code for the control unit can be found below in Figure 9.

```
-- megafunction wizard: %RAM: 1-PORT%
-- GENERATION: STANDARD
-- VERSION: WM1.0
-- MODULE: altsyncram

--
============================================
===============
-- File Name: dataMem.vhd
-- Megafunction Name(s):
--                                      altsyncram
--
-- Simulation Library Files(s):
--                                      altera_mf
--
============================================
===============
--
*********************************************
***************
-- THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT
THIS FILE!
--
-- 13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
--
*********************************************
***************
--Copyright (C) 1991-2013 Altera Corporation
--Your use of Altera Corporation's design tools, logic
functions
--and other software and tools, and its AMPP partner
logic
--functions, and any output files from any of the
foregoing
--(including device programming or simulation files),
and any
--associated documentation or information are
expressly subject
--to the terms and conditions of the Altera Program
License
--Subscription Agreement, Altera MegaCore Function
License
--Agreement, or other applicable license agreement,
including,
--without limitation, that your use is for the sole
purpose of
--programming logic devices manufactured by Altera
and sold by
--Altera or its authorized distributors.  Please refer to
the
--applicable agreement for further details.

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

LIBRARY altera_mf;
USE altera_mf.ALL;

ENTITY dataMem IS
          PORT (
                    address : IN
STD_LOGIC_VECTOR (7 DOWNTO 0);
                    clock : IN STD_LOGIC := '1';
                    data : IN STD_LOGIC_VECTOR (7
DOWNTO 0);
                    wren : IN STD_LOGIC;
                    q : OUT STD_LOGIC_VECTOR (7
DOWNTO 0)
          );
END dataMem;
```

```
ARCHITECTURE SYN OF datamem IS

          SIGNAL sub_wire0 : STD_LOGIC_VECTOR (7
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;

          read_during_write_mode_port_a : STRING;
                    widthad_a :
NATURAL;
                    width_a : NATURAL;
                    width_byteena_a :
NATURAL
                    );
                    PORT (
                    address_a : IN
STD_LOGIC_VECTOR (7 DOWNTO 0);
                    clock0 : IN
STD_LOGIC;
                    data_a : IN
STD_LOGIC_VECTOR (7 DOWNTO 0);
                    wren_a : IN
STD_LOGIC;
                    q_a : OUT
STD_LOGIC_VECTOR (7 DOWNTO 0)
                    );
          END COMPONENT;

BEGIN
          q <= sub_wire0(7 DOWNTO 0);

          altsyncram_component : altsyncram
          GENERIC MAP(
                    clock_enable_input_a =>
"BYPASS",
                    clock_enable_output_a =>
"BYPASS",
                    init_file => "RAM.mif",
                    intended_device_family =>
"Cyclone IV E",
                    lpm_hint =>
"ENABLE_RUNTIME_MOD=NO",
                    lpm_type => "altsyncram",
                    numwords_a => 256,
                    operation_mode =>
"SINGLE_PORT",
                    outdata_aclr_a => "NONE",
                    outdata_reg_a => "CLOCK0",
                    power_up_uninitialized =>
"FALSE",
                    read_during_write_mode_port_a
=> "DONT_CARE",
                    widthad_a => 8,
                    width_a => 8,
                    width_byteena_a => 1
          )
          PORT MAP(
                    address_a => address,
                    clock0 => clock,
                    data_a => data,
                    wren_a => wren,
                    q_a => sub_wire0
          );

END SYN;
```

```
--
==================================================
========
-- CNX file retrieval info
--
==================================================
========
-- Retrieval info: PRIVATE: ADDRESSSTALL_A NUMERIC "0"
-- Retrieval info: PRIVATE: AclrAddr NUMERIC "0"
-- Retrieval info: PRIVATE: AclrByte NUMERIC "0"
-- Retrieval info: PRIVATE: AclrData NUMERIC "0"
-- Retrieval info: PRIVATE: AclrOutput NUMERIC "0"
-- Retrieval info: PRIVATE: BYTE_ENABLE NUMERIC "0"
-- Retrieval info: PRIVATE: BYTE_SIZE NUMERIC "8"
-- Retrieval info: PRIVATE: BlankMemory NUMERIC "0"
-- Retrieval info: PRIVATE: CLOCK_ENABLE_INPUT_A NUMERIC
"0"
-- Retrieval info: PRIVATE: CLOCK_ENABLE_OUTPUT_A
NUMERIC "0"
-- Retrieval info: PRIVATE: Clken NUMERIC "0"
-- Retrieval info: PRIVATE: DataBusSeparated NUMERIC "1"
-- Retrieval info: PRIVATE: IMPLEMENT_IN_LES NUMERIC "0"
-- Retrieval info: PRIVATE: INIT_FILE_LAYOUT STRING "PORT_A"
-- Retrieval info: PRIVATE: INIT_TO_SIM_X NUMERIC "0"
-- Retrieval info: PRIVATE: INTENDED_DEVICE_FAMILY STRING
"Cyclone IV E"
-- Retrieval info: PRIVATE: JTAG_ENABLED NUMERIC "0"
-- Retrieval info: PRIVATE: JTAG_ID STRING "NONE"
-- Retrieval info: PRIVATE: MAXIMUM_DEPTH NUMERIC "0"
-- Retrieval info: PRIVATE: MIFfilename STRING "RAM.mif"
-- Retrieval info: PRIVATE: NUMWORDS_A NUMERIC "256"
-- Retrieval info: PRIVATE: RAM_BLOCK_TYPE NUMERIC "0"
-- Retrieval info: PRIVATE:
READ_DURING_WRITE_MODE_PORT_A NUMERIC "2"
-- Retrieval info: PRIVATE: RegAddr NUMERIC "1"
-- Retrieval info: PRIVATE: RegData NUMERIC "1"
-- Retrieval info: PRIVATE: RegOutput NUMERIC "1"
-- Retrieval info: PRIVATE: SYNTH_WRAPPER_GEN_POSTFIX
STRING "0"
-- Retrieval info: PRIVATE: SingleClock NUMERIC "1"
-- Retrieval info: PRIVATE: UseDQRAM NUMERIC "1"
-- Retrieval info: PRIVATE: WRCONTROL_ACLR_A NUMERIC "0"
-- Retrieval info: PRIVATE: WidthAddr NUMERIC "8"
-- Retrieval info: PRIVATE: WidthData NUMERIC "8"
-- Retrieval info: PRIVATE: rden NUMERIC "0"
-- Retrieval info: LIBRARY: altera_mf
altera_mf.altera_mf_components.all
-- Retrieval info: CONSTANT: CLOCK_ENABLE_INPUT_A STRING
"BYPASS"
-- Retrieval info: CONSTANT: CLOCK_ENABLE_OUTPUT_A
STRING "BYPASS"
-- Retrieval info: CONSTANT: INIT_FILE STRING "RAM.mif"
-- Retrieval info: CONSTANT: INTENDED_DEVICE_FAMILY
STRING "Cyclone IV E"
-- Retrieval info: CONSTANT: LPM_HINT STRING
"ENABLE_RUNTIME_MOD=NO"
-- Retrieval info: CONSTANT: LPM_TYPE STRING "altsyncram"
-- Retrieval info: CONSTANT: NUMWORDS_A NUMERIC "256"
-- Retrieval info: CONSTANT: OPERATION_MODE STRING
"SINGLE_PORT"
-- Retrieval info: CONSTANT: OUTDATA_ACLR_A STRING "NONE"
-- Retrieval info: CONSTANT: OUTDATA_REG_A STRING
"CLOCK0"
-- Retrieval info: CONSTANT: POWER_UP_UNINITIALIZED
STRING "FALSE"
-- Retrieval info: CONSTANT:
READ_DURING_WRITE_MODE_PORT_A STRING "DONT_CARE"
-- Retrieval info: CONSTANT: WIDTHAD_A NUMERIC "8"
-- Retrieval info: CONSTANT: WIDTH_A NUMERIC "8"
-- Retrieval info: CONSTANT: WIDTH_BYTEENA_A NUMERIC "1"
-- Retrieval info: USED_PORT: address 0 0 8 0 INPUT NODEFVAL
"address[7..0]"
-- Retrieval info: USED_PORT: clock 0 0 0 0 INPUT VCC "clock"
-- Retrieval info: USED_PORT: data 0 0 8 0 INPUT NODEFVAL
"data[7..0]"
-- Retrieval info: USED_PORT: q 0 0 8 0 OUTPUT NODEFVAL
"q[7..0]"
-- Retrieval info: USED_PORT: wren 0 0 0 0 INPUT NODEFVAL
"wren"
-- Retrieval info: CONNECT: @address_a 0 0 8 0 address 0 0 8 0
-- Retrieval info: CONNECT: @clock0 0 0 0 0 clock 0 0 0 0
-- Retrieval info: CONNECT: @data_a 0 0 8 0 data 0 0 8 0
-- Retrieval info: CONNECT: @wren_a 0 0 0 0 wren 0 0 0 0
-- Retrieval info: CONNECT: q 0 0 8 0 @q_a 0 0 8 0
-- Retrieval info: GEN_FILE: TYPE_NORMAL dataMem.vhd TRUE
-- Retrieval info: GEN_FILE: TYPE_NORMAL dataMem.inc FALSE
-- Retrieval info: GEN_FILE: TYPE_NORMAL dataMem.cmp TRUE
-- Retrieval info: GEN_FILE: TYPE_NORMAL dataMem.bsf FALSE
-- Retrieval info: GEN_FILE: TYPE_NORMAL dataMem_inst.vhd
FALSE
-- Retrieval info: LIB_FILE: altera_mf
```

Figure 9: VHDL implementation of the data memory

Register File

The register file was built as a hub of 8 registers. To write to a register in the register file, one had to update the value of the WriteReg signal. Each one of these registers had an enable signal that would become active depending on the value of WriteReg so that only the register specified was being written to. Two registers could be read from the register file. The date from any register was read by specifying the correct register using the ReadData signal and then setting the value of the output of the register to a signal. The VHDL code for the register file can be found below in Figure 10.

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

ENTITY RegisterFile_8x8 IS
        PORT (
                in_Read1 : IN
STD_LOGIC_VECTOR (2 DOWNTO 0);
                in_Read2 : IN
STD_LOGIC_VECTOR (2 DOWNTO 0);
                in_Write_sel : IN
STD_LOGIC_VECTOR (2 DOWNTO 0);
                in_Write_Data : IN
STD_LOGIC_VECTOR (7 DOWNTO 0);
                in_Write_en : IN STD_LOGIC;
                in_clk, in_resetbar : IN
STD_LOGIC;
                o_Data1 : OUT
STD_LOGIC_VECTOR (7 DOWNTO 0);
                o_Data2 : OUT
STD_LOGIC_VECTOR (7 DOWNTO 0));
END;

ARCHITECTURE struct OF RegisterFile_8x8 IS
        SIGNAL int_Write_en, int_Decoder_out :
STD_LOGIC_VECTOR(7 DOWNTO 0);
        SIGNAL int_ReadData1, int_ReadData2 :
STD_LOGIC_VECTOR(7 DOWNTO 0);
        SIGNAL int_Reg0_out, int_Reg1_out,
int_Reg2_out : STD_LOGIC_VECTOR(7 DOWNTO 0);
        SIGNAL int_Reg3_out, int_Reg4_out,
int_Reg5_out : STD_LOGIC_VECTOR(7 DOWNTO 0);
        SIGNAL int_Reg6_out, int_Reg7_out :
STD_LOGIC_VECTOR(7 DOWNTO 0);

        COMPONENT Register_8bit IS
                PORT (
                        in_Input : IN
STD_LOGIC_VECTOR (7 DOWNTO 0);
                        in_clk, in_en,
in_resetbar : IN STD_LOGIC;
                        o_Output : OUT
STD_LOGIC_VECTOR (7 DOWNTO 0));
        END COMPONENT;

        COMPONENT Decoder_3x8 IS
                PORT (
                        i_Input : IN
STD_LOGIC_VECTOR (2 DOWNTO 0);
                        o_Output : OUT
STD_LOGIC_VECTOR (7 DOWNTO 0));
        END COMPONENT;

        COMPONENT MUX_8x1_8bit IS
                PORT (
                        i_SEL : IN
STD_LOGIC_VECTOR(2 DOWNTO 0);
                        i_p0 : IN
STD_LOGIC_VECTOR(7 DOWNTO 0);
                        i_p1 : IN
STD_LOGIC_VECTOR(7 DOWNTO 0);
                        i_p2 : IN
STD_LOGIC_VECTOR(7 DOWNTO 0);
                        i_p3 : IN
STD_LOGIC_VECTOR(7 DOWNTO 0);
                        i_p4 : IN
STD_LOGIC_VECTOR(7 DOWNTO 0);
                        i_p5 : IN
STD_LOGIC_VECTOR(7 DOWNTO 0);
                        i_p6 : IN
STD_LOGIC_VECTOR(7 DOWNTO 0);
                        i_p7 : IN
STD_LOGIC_VECTOR(7 DOWNTO 0);
                        o_MUX : OUT
STD_LOGIC_VECTOR(7 DOWNTO 0));
        END COMPONENT;
```

```
BEGIN
        int_Write_en(0) <= in_Write_en AND
int_Decoder_out(0);
        int_Write_en(1) <= in_Write_en AND
int_Decoder_out(1);
        int_Write_en(2) <= in_Write_en AND
int_Decoder_out(2);
        int_Write_en(3) <= in_Write_en AND
int_Decoder_out(3);
        int_Write_en(4) <= in_Write_en AND
int_Decoder_out(4);
        int_Write_en(5) <= in_Write_en AND
int_Decoder_out(5);
        int_Write_en(6) <= in_Write_en AND
int_Decoder_out(6);
        int_Write_en(7) <= in_Write_en AND
int_Decoder_out(7);

        Decoder : Decoder_3x8
        PORT MAP(
                i_Input => in_Write_sel,
                o_Output => int_Decoder_out);

        Register0 : Register_8bit
        PORT MAP(
                in_Input => in_Write_Data,
                in_clk => in_clk,
                in_en => int_Write_en(0),
                in_resetbar => in_resetbar,
                o_Output => int_Reg0_out);

        Register1 : Register_8bit
        PORT MAP(
                in_Input => in_Write_Data,
                in_clk => in_clk,
                in_en => int_Write_en(1),
                in_resetbar => in_resetbar,
                o_Output => int_Reg1_out);

        Register2 : Register_8bit
        PORT MAP(
                in_Input => in_Write_Data,
                in_clk => in_clk,
                in_en => int_Write_en(2),
                in_resetbar => in_resetbar,
                o_Output => int_Reg2_out);

        Register3 : Register_8bit
        PORT MAP(
                in_Input => in_Write_Data,
                in_clk => in_clk,
                in_en => int_Write_en(3),
                in_resetbar => in_resetbar,
                o_Output => int_Reg3_out);

        Register4 : Register_8bit
        PORT MAP(
                in_Input => in_Write_Data,
                in_clk => in_clk,
                in_en => int_Write_en(4),
                in_resetbar => in_resetbar,
                o_Output => int_Reg4_out);

        Register5 : Register_8bit
        PORT MAP(
                in_Input => in_Write_Data,
                in_clk => in_clk,
                in_en => int_Write_en(5),
                in_resetbar => in_resetbar,
                o_Output => int_Reg5_out);

        Register6 : Register_8bit
        PORT MAP(
                in_Input => in_Write_Data,
                in_clk => in_clk,
                in_en => int_Write_en(6),
                in_resetbar => in_resetbar,
                o_Output => int_Reg6_out);

        Register7 : Register_8bit
        PORT MAP(
                in_Input => in_Write_Data,
                in_clk => in_clk,
                in_en => int_Write_en(7),
                in_resetbar => in_resetbar,
                o_Output => int_Reg7_out);
```

```
Read_mux1 : MUX_8x1_8bit
        PORT MAP(
                i_SEL => in_Read1,
                i_p0 => int_Reg0_out,
                i_p1 => int_Reg1_out,
                i_p2 => int_Reg2_out,
                i_p3 => int_Reg3_out,
                i_p4 => int_Reg4_out,
                i_p5 => int_Reg5_out,
                i_p6 => int_Reg6_out,
                i_p7 => int_Reg7_out,
                o_MUX => int_ReadData1);

        Read_mux2 : MUX_8x1_8bit
        PORT MAP(
                i_SEL => in_Read2,
                i_p0 => int_Reg0_out,
                i_p1 => int_Reg1_out,
                i_p2 => int_Reg2_out,
                i_p3 => int_Reg3_out,
                i_p4 => int_Reg4_out,
                i_p5 => int_Reg5_out,
                i_p6 => int_Reg6_out,
                i_p7 => int_Reg7_out,
                o_MUX => int_ReadData2);

        o_Data1 <= int_ReadData1;
        o_Data2 <= int_ReadData2;
END struct;
```

Figure 10: VHDL implementation of the register file

Instruction Memory

The instruction memory holds the instructions for the processor to fetch, decode, and execute. It uses the “LPM_ROM” function from Altera. It has an address length of 8 bits, 28 = 256 addresses, and each memory location is 32 bits wide. The 32-bit instruction output is fetched from the 8 bit address pointed to by the PC. The VHDL code for the instruction memory can be found below in Figure 11.

```
-- megafunction wizard: %ROM: 1-PORT%
-- GENERATION: STANDARD
-- VERSION: WM1.0
-- MODULE: altsyncram

--
============================================
==============
-- File Name: instrMem.vhd
-- Megafunction Name(s):
--                                    altsyncram
--
-- Simulation Library Files(s):
--                                    altera_mf
--
============================================
==============
--
*********************************************
**************
-- THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT
THIS FILE!
--
-- 13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
--
*********************************************
**************
--Copyright (C) 1991-2013 Altera Corporation
--Your use of Altera Corporation's design tools, logic
functions
--and other software and tools, and its AMPP partner
logic
--functions, and any output files from any of the
foregoing
--(including device programming or simulation files),
and any
--associated documentation or information are
expressly subject
--to the terms and conditions of the Altera Program
License
--Subscription Agreement, Altera MegaCore Function
License
--Agreement, or other applicable license agreement,
including,
--without limitation, that your use is for the sole
purpose of
--programming logic devices manufactured by Altera
and sold by
--Altera or its authorized distributors.  Please refer to
the
--applicable agreement for further details.

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

LIBRARY altera_mf;
USE altera_mf.ALL;

ENTITY instrMem IS
        PORT (
                address : IN
STD_LOGIC_VECTOR (7 DOWNTO 0);
                clock : IN STD_LOGIC := '1';
                q : OUT STD_LOGIC_VECTOR
(31 DOWNTO 0)
        );
END instrMem;
```

```
ARCHITECTURE SYN OF instrmem IS

        SIGNAL sub_wire0 : STD_LOGIC_VECTOR (31
DOWNTO 0);

        COMPONENT altsyncram
                GENERIC (
                        address_aclr_a :
STRING;

        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;

                        widthad_a :
NATURAL;

                        width_a : NATURAL;
                        width_byteena_a :
NATURAL
                );
                PORT (
                        address_a : IN
STD_LOGIC_VECTOR (7 DOWNTO 0);

                        clock0 : 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(
                address_aclr_a => "NONE",
                clock_enable_input_a =>
"BYPASS",
                clock_enable_output_a =>
"BYPASS",
                init_file => "ROM2.mif",
                intended_device_family =>
"Cyclone IV E",
                lpm_hint =>
"ENABLE_RUNTIME_MOD=NO",
                lpm_type => "altsyncram",
                numwords_a => 256,
                operation_mode => "ROM",
                outdata_aclr_a => "NONE",
                outdata_reg_a => "CLOCK0",
                widthad_a => 8,
                width_a => 32,
                width_byteena_a => 1
        )
        PORT MAP(
                address_a => address,
                clock0 => clock,
                q_a => sub_wire0
        );

END SYN;
```

```
--
==================================================
========
-- CNX file retrieval info
--
==================================================
========
-- Retrieval info: PRIVATE: ADDRESSSTALL_A NUMERIC "0"
-- Retrieval info: PRIVATE: AclrAddr NUMERIC "0"
-- Retrieval info: PRIVATE: AclrByte NUMERIC "0"
-- Retrieval info: PRIVATE: AclrOutput NUMERIC "0"
-- Retrieval info: PRIVATE: BYTE_ENABLE NUMERIC "0"
-- Retrieval info: PRIVATE: BYTE_SIZE NUMERIC "8"
-- Retrieval info: PRIVATE: BlankMemory NUMERIC "0"
-- Retrieval info: PRIVATE: CLOCK_ENABLE_INPUT_A NUMERIC
"0"
-- Retrieval info: PRIVATE: CLOCK_ENABLE_OUTPUT_A
NUMERIC "0"
-- Retrieval info: PRIVATE: Clken NUMERIC "0"
-- Retrieval info: PRIVATE: IMPLEMENT_IN_LES NUMERIC "0"
-- Retrieval info: PRIVATE: INIT_FILE_LAYOUT STRING "PORT_A"
-- Retrieval info: PRIVATE: INIT_TO_SIM_X NUMERIC "0"
-- Retrieval info: PRIVATE: INTENDED_DEVICE_FAMILY STRING
"Cyclone IV E"
-- Retrieval info: PRIVATE: JTAG_ENABLED NUMERIC "0"
-- Retrieval info: PRIVATE: JTAG_ID STRING "NONE"
-- Retrieval info: PRIVATE: MAXIMUM_DEPTH NUMERIC "0"
-- Retrieval info: PRIVATE: MIFfilename STRING "ROM.mif"
-- Retrieval info: PRIVATE: NUMWORDS_A NUMERIC "256"
-- Retrieval info: PRIVATE: RAM_BLOCK_TYPE NUMERIC "0"
-- Retrieval info: PRIVATE: RegAddr NUMERIC "1"
-- Retrieval info: PRIVATE: RegOutput NUMERIC "1"
-- Retrieval info: PRIVATE: SYNTH_WRAPPER_GEN_POSTFIX
STRING "0"
-- Retrieval info: PRIVATE: SingleClock NUMERIC "1"
-- Retrieval info: PRIVATE: UseDQRAM NUMERIC "0"
-- Retrieval info: PRIVATE: WidthAddr NUMERIC "8"
-- Retrieval info: PRIVATE: WidthData NUMERIC "32"
-- Retrieval info: PRIVATE: rden NUMERIC "0"
-- Retrieval info: LIBRARY: altera_mf
altera_mf.altera_mf_components.all
-- Retrieval info: CONSTANT: ADDRESS_ACLR_A STRING "NONE"
-- Retrieval info: CONSTANT: CLOCK_ENABLE_INPUT_A STRING
"BYPASS"
-- Retrieval info: CONSTANT: CLOCK_ENABLE_OUTPUT_A
STRING "BYPASS"
-- Retrieval info: CONSTANT: INIT_FILE STRING "ROM.mif"
-- Retrieval info: CONSTANT: INTENDED_DEVICE_FAMILY
STRING "Cyclone IV E"
-- Retrieval info: CONSTANT: LPM_HINT STRING
"ENABLE_RUNTIME_MOD=NO"
-- Retrieval info: CONSTANT: LPM_TYPE STRING "altsyncram"
-- Retrieval info: CONSTANT: NUMWORDS_A NUMERIC "256"
-- Retrieval info: CONSTANT: OPERATION_MODE STRING "ROM"
-- Retrieval info: CONSTANT: OUTDATA_ACLR_A STRING "NONE"
-- Retrieval info: CONSTANT: OUTDATA_REG_A STRING
"CLOCK0"
-- Retrieval info: CONSTANT: WIDTHAD_A NUMERIC "8"
-- Retrieval info: CONSTANT: WIDTH_A NUMERIC "32"
-- Retrieval info: CONSTANT: WIDTH_BYTEENA_A NUMERIC "1"
-- Retrieval info: USED_PORT: address 0 0 8 0 INPUT NODEFVAL
"address[7..0]"
-- Retrieval info: USED_PORT: clock 0 0 0 0 INPUT VCC "clock"
-- Retrieval info: USED_PORT: q 0 0 32 0 OUTPUT NODEFVAL
"q[31..0]"
-- Retrieval info: CONNECT: @address_a 0 0 8 0 address 0 0 8 0
-- Retrieval info: CONNECT: @clock0 0 0 0 0 clock 0 0 0 0
-- Retrieval info: CONNECT: q 0 0 32 0 @q_a 0 0 32 0
-- Retrieval info: GEN_FILE: TYPE_NORMAL instrMem.vhd TRUE
-- Retrieval info: GEN_FILE: TYPE_NORMAL instrMem.inc FALSE
-- Retrieval info: GEN_FILE: TYPE_NORMAL instrMem.cmp TRUE
-- Retrieval info: GEN_FILE: TYPE_NORMAL instrMem.bsf FALSE
-- Retrieval info: GEN_FILE: TYPE_NORMAL instrMem_inst.vhd
FALSE
-- Retrieval info: LIB_FILE: altera_mf
```

Figure 11: VHDL implementation of the instruction memory

ALU

The ALU was required to perform a number of operations depending on the value of ALU control. Logical AND, OR and SET ON LESS THAN were supported as well as arithmetic Add and Subtract operations. A CLA unit was incorporated in the ALU to improve efficiency and reduce gate delays. The VHDL code for the ALU can be found below in Figure 12.

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

ENTITY unitALU_8bit IS
	PORT (
		A, B : IN
STD_LOGIC_VECTOR (7 DOWNTO 0);
		Op : IN
STD_LOGIC_VECTOR (2 DOWNTO 0);
		Output : OUT
STD_LOGIC_VECTOR (7 DOWNTO 0);
		Zero, Ovr : OUT
STD_LOGIC
	);
END;

ARCHITECTURE struct OF unitALU_8bit
IS
	SIGNAL int_OR, int_AND,
int_CLA, int_Out : STD_LOGIC_VECTOR
(7 DOWNTO 0);
	SIGNAL int_SLT : STD_LOGIC;

	COMPONENT aluCLA_8bit IS
		PORT (
			A, B : IN
STD_LOGIC_VECTOR (7 DOWNTO 0);
			Cin : IN
STD_LOGIC;
			Sum : OUT
STD_LOGIC_VECTOR (7 DOWNTO 0);
			Cout, Ovr :
OUT STD_LOGIC
		);
	END COMPONENT;

	COMPONENT
sgnComparator_8bit IS
		PORT (
			A, B : IN
STD_LOGIC_VECTOR (7 DOWNTO 0);
			Ls, Eq, Gr :
OUT STD_LOGIC
		);
	END COMPONENT;

	COMPONENT MUX_4x1_8bit IS
		PORT (
			input0,
input1, input2, input3 : IN
STD_LOGIC_VECTOR (7 DOWNTO 0);
			SEL : IN
STD_LOGIC_VECTOR (1 DOWNTO 0);
			output : OUT
STD_LOGIC_VECTOR (7 DOWNTO 0)
		);
	END COMPONENT;
```

```
BEGIN
	MUX : MUX_4x1_8bit
	PORT MAP(
		input0 => int_OR,
		input1 => int_AND,
		input2 => int_CLA,
		input3(7 DOWNTO 1)
=> "0000000",
		input3(0) => int_SLT,
		SEL => Op (1 DOWNTO
0),
		output => int_Out
	);

	SLT : sgnComparator_8bit
	PORT MAP(
		A => A,
		B => B,
		Ls => int_SLT,
		Eq => OPEN,
		Gr => OPEN
	);

	CLA : aluCLA_8bit
	PORT MAP(
		A => A,
		B => B,
		Cin => Op(2),
		Sum => int_CLA,
		Cout => OPEN,
		Ovr => Ovr
	);

	int_OR <= A OR B;
	int_AND <= A AND B;
	Zero <= (NOT (int_Out(7) OR
int_Out(6) OR int_Out(5) OR int_Out(4)
OR int_Out(3) OR int_Out(2) OR
int_Out(1) OR int_Out(0)));
	Output <= int_Out;
END struct;
```

Figure 12: VHDL implementation of the ALU

Top File

This file outputs the contents of the single cycle processor. It outputs 32-bit Instructions, and Control signals; ZeroOut, BranchOut, MemWriteOut, RegWriteOut. There is also an 8x1 MUX controlled by inputs, which output different data from processor. This top-level block instantiates its own control and datapaths. This top file (single-cycle processor) was implemented in VHDL as seen in Figure 14.

This ALU unit was simulated using Quartus' built in simulation, the results of which can be seen in Figure 13 From Figure 13, it can be seen that we were able to successfully create a single-cycle RISC-type processor in VHDL.

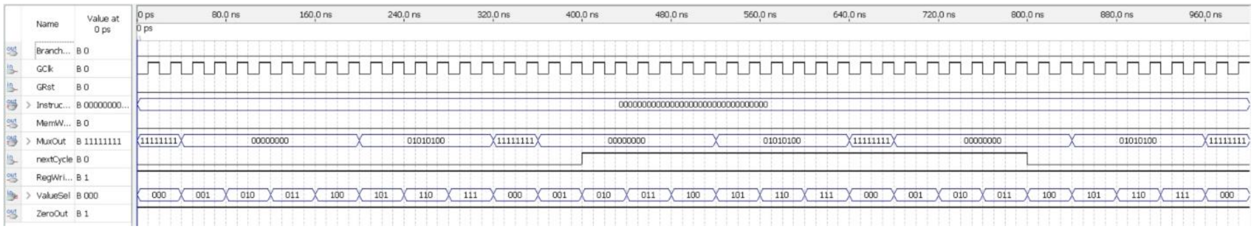


Figure 13: Simulation results of the complete ALU from Quartus

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

ENTITY Top IS
		PORT (
				ValueSel : IN
STD_LOGIC_VECTOR (2 DOWNTO 0);
				nextCycle, GClk, GRst : IN
STD_LOGIC;
				InstructionOut : OUT
STD_LOGIC_VECTOR (31 DOWNTO 0);
				MuxOut : OUT
STD_LOGIC_VECTOR (7 DOWNTO 0);
				BranchOut, ZeroOut,
MemWriteOut, RegWriteOut : OUT STD_LOGIC
		);
END;

ARCHITECTURE struct OF Top IS
		SIGNAL int_Instruct :
STD_LOGIC_VECTOR (31 DOWNTO 0);
		SIGNAL int_PC, int_aluResult, int_Data1,
int_Data2, int_WriteData, int_Other :
STD_LOGIC_VECTOR (7 DOWNTO 0);
		SIGNAL int_ALUOp : STD_LOGIC_VECTOR
(1 DOWNTO 0);
		SIGNAL int_Branch, int_Zero,
int_MemWrite, int_RegWrite, int_Jump,
int_RegDst, int_MemtoReg, int_ALUSrc :
STD_LOGIC;
		SIGNAL int_nextCycle : STD_LOGIC;

		COMPONENT processor_8bit IS
				PORT (
						nextCycle, GClk,
GRst : IN STD_LOGIC;
						Instruct : OUT
STD_LOGIC_VECTOR (31 DOWNTO 0);
						PC, aluResult,
Data1, Data2, WriteData : OUT STD_LOGIC_VECTOR
(7 DOWNTO 0);
						ALUOp : OUT
STD_LOGIC_VECTOR (1 DOWNTO 0);
						Branch, Zero,
MemWrite, RegWrite, Jump, RegDst, MemtoReg,
ALUSrc : OUT STD_LOGIC
				);
		END COMPONENT;

		COMPONENT MUX_8x1_8bit IS
				PORT (
						i_SEL : IN
STD_LOGIC_VECTOR(2 DOWNTO 0);
						i_p0 : IN
STD_LOGIC_VECTOR(7 DOWNTO 0);
						i_p1 : IN
STD_LOGIC_VECTOR(7 DOWNTO 0);
						i_p2 : IN
STD_LOGIC_VECTOR(7 DOWNTO 0);
						i_p3 : IN
STD_LOGIC_VECTOR(7 DOWNTO 0);
						i_p4 : IN
STD_LOGIC_VECTOR(7 DOWNTO 0);
						i_p5 : IN
STD_LOGIC_VECTOR(7 DOWNTO 0);
						i_p6 : IN
STD_LOGIC_VECTOR(7 DOWNTO 0);
						i_p7 : IN
STD_LOGIC_VECTOR(7 DOWNTO 0);
						o_MUX : OUT
STD_LOGIC_VECTOR(7 DOWNTO 0));
		END COMPONENT;
```

```
BEGIN
		int_Other(7) <= '0';
		int_Other(6) <= int_RegDst;
		int_Other(5) <= int_Jump;
		int_Other(4) <= NOT
int_MemWrite;
		int_Other(3) <= int_MemtoReg;
		int_Other(2 DOWNTO 1) <=
int_ALUOp;
		int_Other(0) <= int_ALUSrc;

		PROC : processor_8bit
		PORT MAP(
				nextCycle => nextCycle,
				GClk => GClk,
				GRst => GRst,
				Instruct => int_Instruct,
				PC => int_PC,
				aluResult =>
int_aluResult,
				Data1 => int_Data1,
				Data2 => int_Data2,
				WriteData =>
int_WriteData,
				ALUOp => int_ALUOp,
				Branch => int_Branch,
				Zero => int_Zero,
				MemWrite =>
int_MemWrite,
				RegWrite =>
int_RegWrite,
				Jump => int_Jump,
				RegDst => int_RegDst,
				MemtoReg =>
int_MemtoReg,
				ALUSrc => int_ALUSrc
		);

		MUX : MUX_8x1_8bit
		PORT MAP(
				i_SEL => ValueSel,
				i_p0 => int_PC,
				i_p1 => int_aluResult,
				i_p2 => int_Data1,
				i_p3 => int_Data2,
				i_p4 => int_WriteData,
				i_p5 => int_Other,
				i_p6 => int_Other,
				i_p7 => int_Other,
				o_MUX => MUXOut
		);

		int_nextCycle <= NOT nextCycle;
		InstructionOut <= int_Instruct;
		BranchOut <= int_Branch;
		ZeroOut <= int_Zero;
		MemWriteOut <= int_MemWrite;
		RegWriteOut <= int_RegWrite;
END struct;
```

Figure 14: VHDL implementation of the top level entity

Real Implementation

In order to demonstrate that our single-cycle processor works properly it was loaded onto an Altera DE-2 board. To do this we ran a benchmark program. Once completed, the design has to be tested and veried. The testing process can be accomplished throughout the design, with unit testing, then module testing, and finally system testing. A wrapper was also created. The wrapper includes a block to slow down the clock for it to be seen. These two blocks were found on the course website. The wrapper is also used to assign the pins to those on the board. These pin assignments of the board can be found in Table 1. Table 2 shows the Output Multiplexer Selection

Table 1: Input/Output Specification

Port Type	**Name**	**Description**
Input	Gclock	Global clock needed to synchronize the circuitry
Input	GReset	Global reset needed to bring the internals to known states
Input	ValueSelect[2..0]	Selector for MuxOut[7..0]
Output	MuxOut[7..0]	Multiplexer output controlled by ValueSelect[2..0]
Output	InstructionOut[31..0]	The current instruction being executed
Output	BranchOut	The branch control signal
Output	ZeroOut	The zero status signal
Output	MemWriteOut	The memory write control signal
Output	RegWriteOut	The register write control signal

Table 2: Output Multiplexer Selection

ValueSelect[2..0]	**MuxOut[7..0]**	**Description**
000	PC[7..0]	The program counter value
001	ALUResult[7..0]	The results of the current ALU operation
010	ReadData1[7..0]	The read data 1 port of the register file
011	ReadData2[7..0]	The read data 2 port of the register file
100	WriteData[7..0]	The write data port of the register file
Other	['0', RegDst, Jump, MemRead, MemtoReg, AluOp[1..0], AluSrc]	The remaining control information

Discussion

Over the course of the lab, we encountered few major issues. Issues encountered were mainly compilation errors in Modelsim, or small bugs in the design. These could be fixed either by following the error messages provided or by looking through the simulation waves.

As we now had experience using ASM methodology, designing and using structural VHDL, as well simulating using Modelsim, performing this lab went by fairly smoothly with no major issues arising. We were both a bit rusty in using Quartus however. The program continuously kept freezing and displaying not responding error, which was in part due to a weak internet connection from my end. The simulation ran as expected. However, while doing the demonstration to the TA, we encountered some problems assigning the pins, which took away some crucial time. The demo was half successful, as the program was finally loaded onto the Altera board. However, there seemed to be another error (with the pin assignment), that caused the board to stay on a single output and not change or reset as expected.

We were able to successfully create a single-cycle processor. We were able to efficiently accomplish the objectives of the lab, such as designing, realizing, and testing a single-cycle RISC processor. In addition, we were able to demonstrate a complete understanding for RISC processors and instruction set architectures. The five steps of ASM method, as well as coding in RTL and structural VHDL proved to be of utmost importance for the design of the single-cycle RISC processor. This was done by completing the ASM steps, as well as designing, realizing, and testing the synchronous digital circuits of the datapath and control logic. Lastly, we demonstrated more or less a complete understanding for circuit realization using ASM and VHDL. This can be proven by the fact that the simulations yielded the expected outputs. Furthermore, problems encountered were successfully overcome. Our design fit the parameters required and worked as expected. Thus, the results recorded were identical to those of the theoretical predicted ones. No errors were encountered in our design as all the elements worked as expected. Overall, this lab proved to be a good experience as we were able to learn and apply new methods, such as designing and building a single-cycle RISC-type processor in VHDL. With that, we are now able to continue applying these newly acquired skills to upcoming labs.