Bob Minchin

EEL4713C

Assignment 3

February 13, 2012

INTRODUCTION

The purposes of this assignment were (1) to become further acquainted with the MIPS architecture by going through a bug-finding exercise in processor simulation and (2) to reinforce some of the basic principles covered in class by guiding the student through designing some more basic components that will be needed to build into a microprocessor pipeline later in the semester. A basic adder, a MIPS ALU, an ALU controller, and a register file were assigned, and each was designed and simulated according to the directions.

MIPS BUG SIMULATION ("BUGSPIM")

The purpose of this exercise was to identify five bugs in a MIPS simulation program based on knowledge of how the MIPS architecture operates. Although the directions seemed straightforward, several problems were encountered when trying to complete this part of the assignment. First was my difficulty getting Bugspim to start without the instruction to load it directly into the SPIM directory. Then there was my complete lack of familiarity with how to work within Grid Appliance/Linux. I had to figure out everything from how to edit text to how the file path structure worked. Finally, it took me a long time to figure out that the same lines of code are executed at the beginning of every program and that my attempts to load custom code into the simulator were actually successful.

After overcoming all of these problems, I did not have much time left to actually perform the simulation. In order to determine which instructions created problems, I ran the following instructions numerous times in various different contexts: ADD, ADDI, ADDU, ADDUI, AND, ANDI, BNE, DIV, DIVU, J, JR, LB, LUI, LW, MFHI, MFLO, MULT, MULTU, OR, ORI, SB, SLL, SLT, SLTI, SLTIU, SLTU, SRL, SUB, SUBU, SW, XOR, and XORI. However, I was able to locate only one bug, which was as follows.

The instruction "load upper immediate" (lui) does not work. It is designed to take in a 16-bit number and place it in the upper half of the register. The following code was executed:

```
lui $t0, 65535
```

This should have resulted in $t0 = 0xFFFF0000. Instead, it resulted in $t0 = 0x0000FFFF, which means that the load was accomplished but the shift was not.

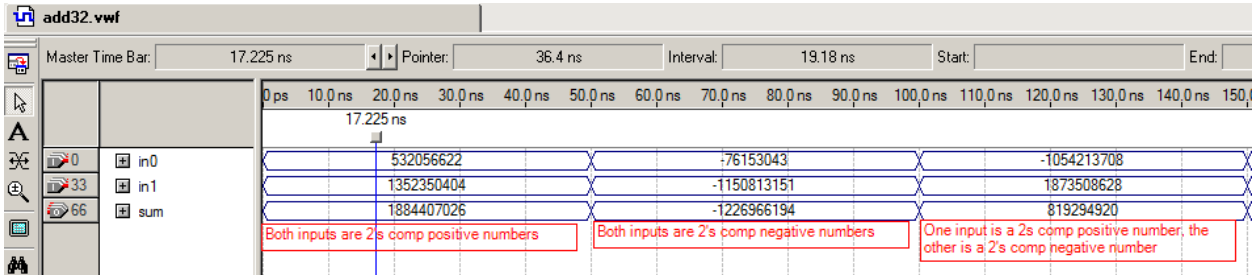
DESIGN AND TESTING

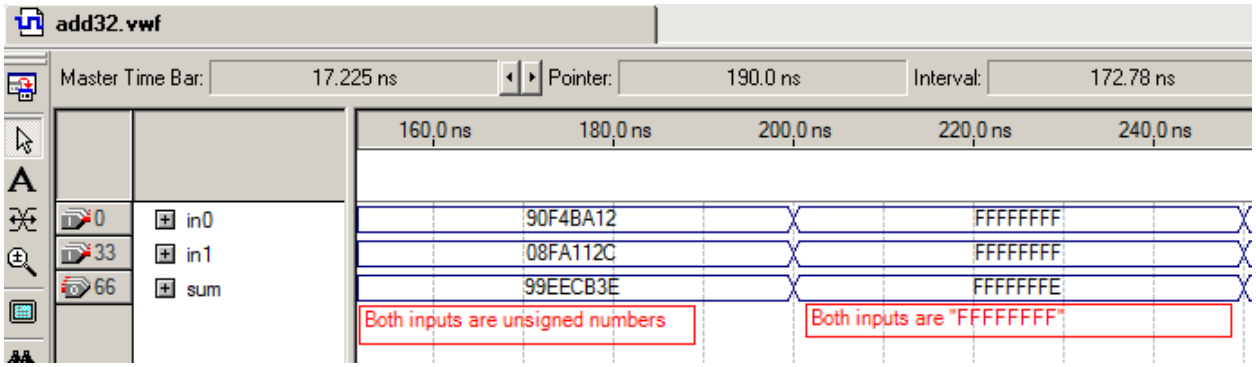
The first device to design was a simple 32-bit adder that was named "add32." It was designed by simply taking two signals and applying an arithmetic add operation, which was accomplished using the following code.

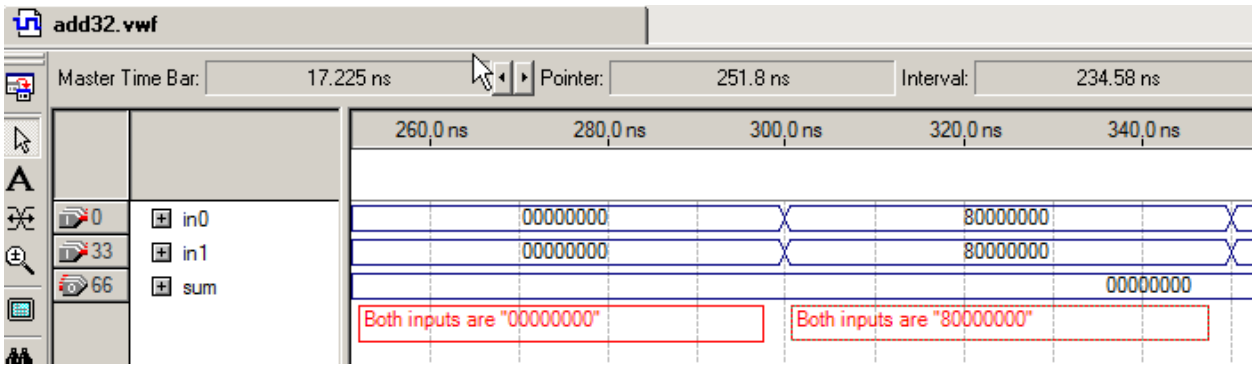
```
-- Bob Minchin
-- EEL 4713C
-- Homework 3
-- Part 3.1: 32-Bit Adder

LIBRARY ieee ;
```

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

ENTITY add32 IS
      --32-bit inputs, output
      PORT (      in0, in1 : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
                  sum :      OUT STD_LOGIC_VECTOR(31 DOWNTO 0) ) ;
END add32 ;

ARCHITECTURE Structure OF add32 IS
BEGIN

      sum <= in0 + in1 ;    --Inputs added, output results

END Structure ;
```

The following are screenshots of the simulation of this device.







The device performed as expected.

The next device was a 32-bit ALU, which eventually will be at the heart of all MIPS operations. The purpose of this device is to perform basic arithmetic and logic operations on numbers in internal registers that will be used to execute more complex operations in the computer architecture.

The code used for the overflow flag was quick and dirty—sort of a "brute force" solution that requires an additional add every time an add or subtract operation is requested. This is an inefficient design that works for our current purposes but will likely need to be replaced by another solution, such as a ripple-carry adder, once timing and efficiency become important.

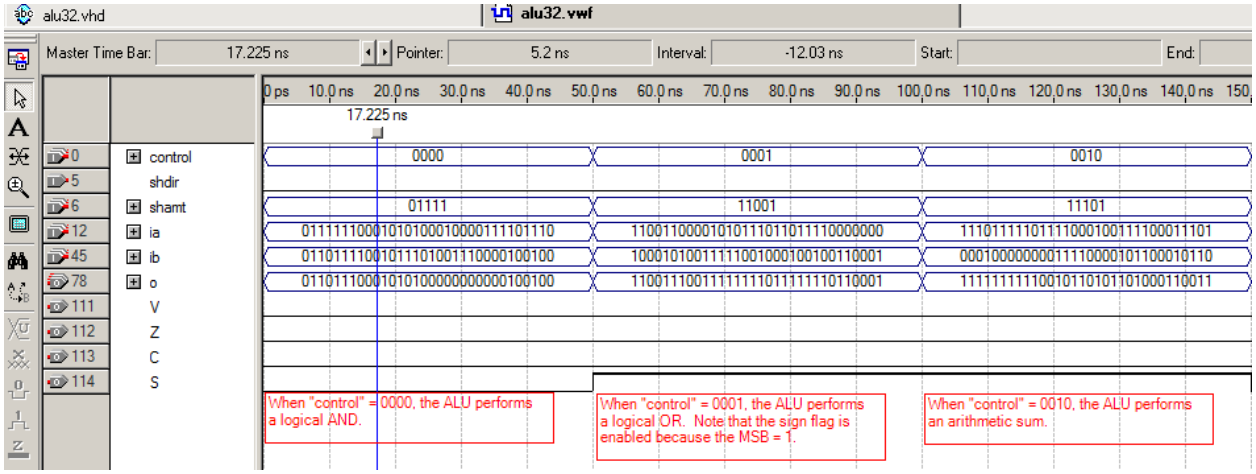
Furthermore, the ALU was implemented as an "all-in-one" device. No modularity was implemented. All functions were performed internally. As the device is refined, more refined methods of implementation may be built in to help facilitate greater efficiency.

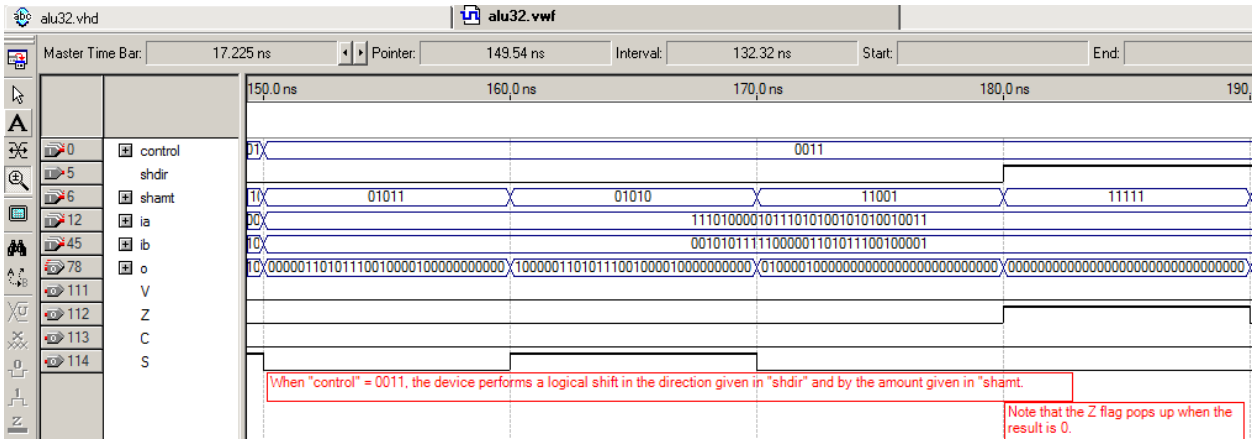
This ALU, "alu32," was accomplished with the following code.

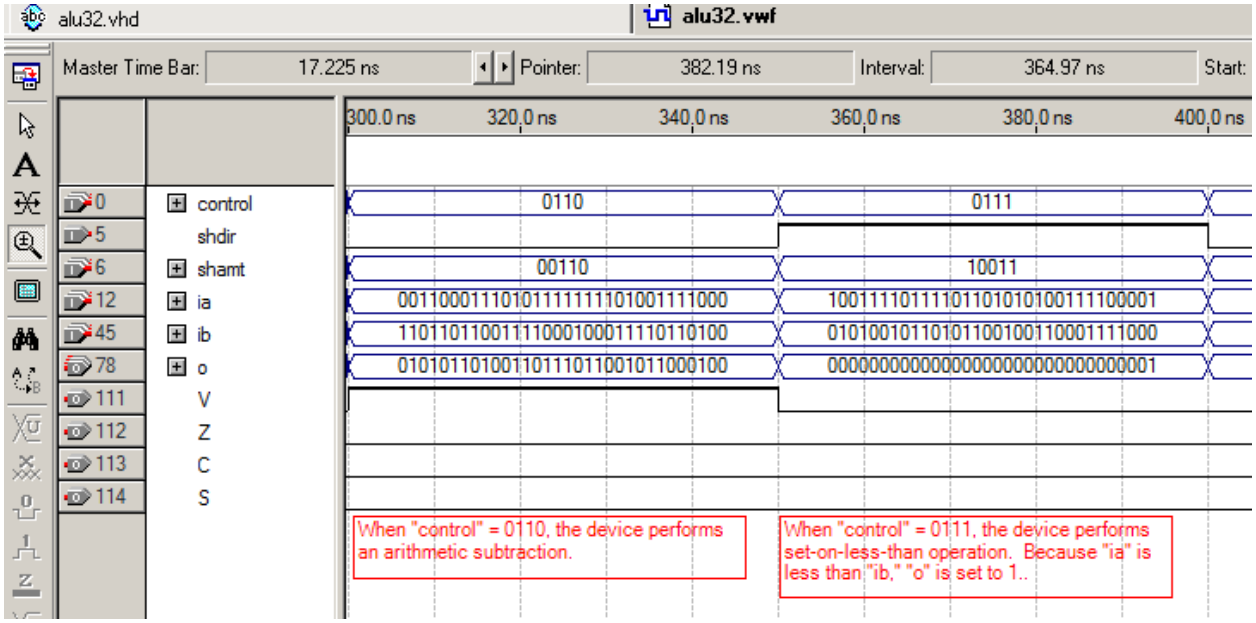
```
-- Bob Minchin
-- EEL 4713C
-- Homework 3
-- Part 3.2: 32-Bit ALU

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

ENTITY alu32 IS
      --32-bit inputs, output
      --4-bit control input
      --5-bit ALUOp input, shift amount input
      --1-bit shift direction input, output flags
      PORT (     ia, ib :         IN STD_LOGIC_VECTOR(31 DOWNTO 0) ;
                 control :        IN STD_LOGIC_VECTOR(3 DOWNTO 0) ;
                 shamt :          IN STD_LOGIC_VECTOR(4 DOWNTO 0) ;
                 shdir :          IN STD_LOGIC ;
                 o :              BUFFER STD_LOGIC_VECTOR(31 DOWNTO 0) ;
                 C, Z, S, V :     OUT STD_LOGIC ) ;
END alu32 ;

ARCHITECTURE Structure OF alu32 IS
      SIGNAL temp33 :  STD_LOGIC_VECTOR(32 DOWNTO 0) ;
BEGIN
      PROCESS(ia, ib, control, shamt, shdir, temp33, o)
            VARIABLE temp32 :     unsigned(31 DOWNTO 0) ;
      BEGIN
            C <= '0' ; --initialize flags
            V <= '0' ;
            IF o = x"00000000" THEN
                  Z <= '1' ;
```

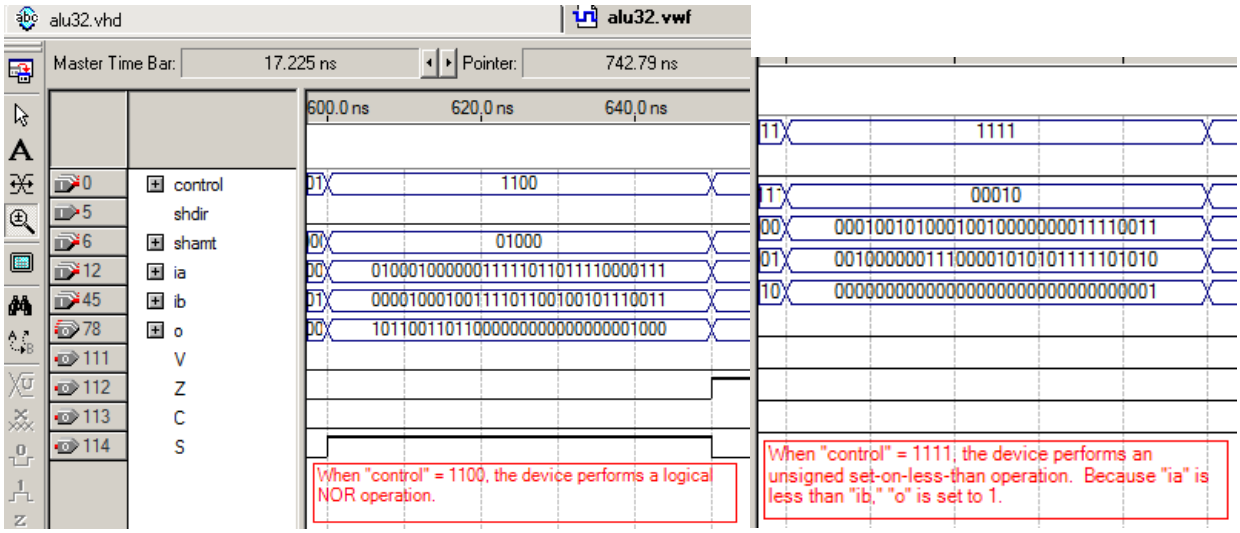
```
            ELSE Z <= '0' ;
            END IF ;
            IF o(31) = '1' THEN
                 S <= '1' ;
            ELSE S <= '0' ;
            END IF ;
            CASE control IS
                 WHEN "0000" =>  --and
                      o <= ia AND ib ;
                 WHEN "0001" =>  --or
                      o <= ia OR ib ;
                 WHEN "0010" =>  --add
                      temp33 <= ('0' & ia) + ('0' & ib) ;
                      temp32 := UNSIGNED(('0' & ia(30 DOWNTO 0))) +
                           UNSIGNED(('0' & ib(30 DOWNTO 0))) ;
                      V <= temp33(32) XOR temp32(31) ;
                      C <= temp33(32) ;
                      o <= temp33(31 DOWNTO 0) ;
                 WHEN "0011" =>  --shift logical
                      IF (shdir = '0') THEN
                           temp32 := SHL(UNSIGNED(ib),
                                UNSIGNED(shamt)) ;
                      ELSE
                           temp32 := SHR(UNSIGNED(ib),
                                UNSIGNED(shamt)) ;
                      END IF ;
                      o <= CONV_STD_LOGIC_VECTOR(temp32, 32) ;
                 WHEN "0110" =>  --subtract
                      temp33 <= ('0' & ia) + (NOT ('1' & ib) ) + 1 ;
                      temp32 := UNSIGNED(('0' & ia(30 DOWNTO 0))) +
                           UNSIGNED(('0' & ib(30 DOWNTO 0))) ;
                      V <= temp33(32) XOR temp32(31) ;
                      C <= temp33(32) ;
                      o <= temp33(31 DOWNTO 0) ;
                 WHEN "0111" =>  --set less than
                      IF (SIGNED(ia) < SIGNED(ib) ) THEN
                           o <= x"00000001" ;
                      ELSE o <= x"00000000" ;
                      END IF ;
                 WHEN "1100" =>  --nor
                      o <= NOT (ia OR ib) ;
                 WHEN "1111" =>  --set less than unsigned
                      IF (UNSIGNED(ia) < UNSIGNED(ib) ) THEN
                           o <= x"00000001" ;
                      ELSE o <= x"00000000" ;
                      END IF ;
                 WHEN OTHERS =>
            END CASE ;
      END PROCESS ;
END Structure ;
```

The following are screenshots of the simulation that illustrate each of the ALU operations.









The device performed as expected.

The next part was a controller for this ALU. The job of the controller is to decode the instruction operand into a set of instructions for the ALU. Because there are a number of instructions that require the ALU to perform the same operation, this device simply needs to convert the relevant parts of the machine code into an internal signal that will tell the ALU what it needs to do. Basically, it is a glorified look-up table.

The first step was to create a physical look-up table that included all the MIPS instructions and their corresponding ALU operations. Once the instructions and operations were delineated, three-bit ALU operation codes were assigned to each instruction. If multiple instructions needed the same function performed in the ALU, they were given the same ALU operation code.

The table that was created can be seen below.

Instruction Opcode	ALUop	Instruction Operand	Funct Field	Desired ALU Action	ALU Control Input
R-type	100	add	100000	add	0010
001000	000	add imm		add	0010
001001	000	add imm uns		add	0010
R-type	100	add uns	100001	add	0010
R-type	100	and	100100	and	0000
001100	011	and imm		and	0000
000100	001	branch on equal		subtract	0110
000101	001	branch on not equal		subtract	0110
000010		jump			
000011		jump and link			
R-type	100	jump reg	001000		
100100	000	load byte uns		add	0010

100101	000	load hw uns		add	0010
110000	000	load linked		add	0010
001111	000	load upper imm		add	0010
100011	000	load word		add	0010
R-type	100	nor	100111	nor	1100
R-type	100	or	100101	or	0001
001101	010	or imm		or	0001
R-type	100	set less than	101010	set less than	0111
001010	101	set less than imm		set less than	0111
001011	110	set less than imm uns		set less than uns	1111
R-type	100	set less than uns	101011	set less than uns	1111
R-type	100	shift left log	000000	shift log	0011
R-type	100	shift right log	000010	shift log	0011
101000	000	store byte		add	0010
111000	000	store cond		add	0010
101001	000	store hw		add	0010
101011	000	store word		add	0010
R-type	100	subtract	100010	subtract	0110
R-type	100	subtract uns	100011	subtract	0110

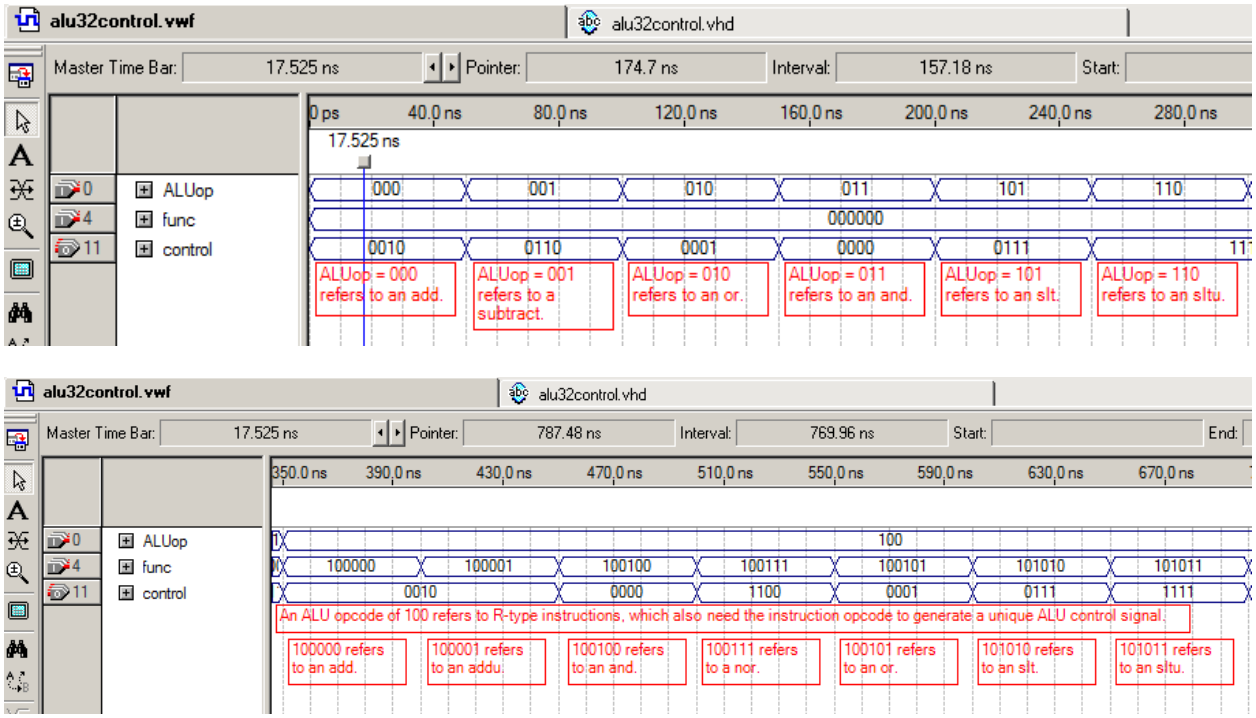
This table was referenced constantly in creating the code for the entity, which was named "alu32control." In all cases except for that of "R-type" instructions, the ALU opcode was all that was needed to determine the ALU control signal. For "R-type" instructions, the instruction opcode was needed as well.

The following code was used to implement "alu32contol."

```
-- Bob Minchin
-- EEL 4713C
-- Homework 3
-- Part 3.3: 32-Bit ALU Controller

LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY alu32control IS
      --6-bit instruction opcode input
      --5-bit ALU opcode input
      --4-bit ALU control signal output
      PORT (     func :     IN STD_LOGIC_VECTOR(5 DOWNTO 0) ;
                 ALUop :    IN STD_LOGIC_VECTOR(2 DOWNTO 0) ;
                 control :  OUT STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
END alu32control ;

ARCHITECTURE Structure OF alu32control IS
BEGIN
```

```
	PROCESS(func, ALUop)
	BEGIN
		CASE ALUop IS
			WHEN "100" =>	--r-type
				CASE func IS
					WHEN "100000" => --add
						control <= "0010" ;
					WHEN "100001" => --add unsigned
						control <= "0010" ;
					WHEN "100100" => --and
						control <= "0000" ;
					WHEN "100111" => --nor
						control <= "1100" ;
					WHEN "100101" => --or
						control <= "0001" ;
					WHEN "101010" => --set less than
						control <= "0111" ;
					WHEN "101011" => --set less than unsigned
						control <= "1111" ;
					WHEN "000000" => --shift left logical
						control <= "0011" ;
					WHEN "000010" => --shift right logical
						control <= "0011" ;
					WHEN "100010" => --subtract
						control <= "0110" ;
					WHEN "100011" => --subtract unsigned
						control <= "0110" ;
					WHEN OTHERS =>
				END CASE ;
			WHEN "000" =>	--add immediate
				control <= "0010" ;
			WHEN "011" =>	--and immediate
				control <= "0000" ;
			WHEN "001" =>	--branch
				control <= "0110" ;
			WHEN "010" =>	--or immediate
				control <= "0001" ;
			WHEN "101" =>	--set less than
				control <= "0111" ;
			WHEN "110" =>	--set less than unsigned
				control <= "1111" ;
			WHEN OTHERS =>
		END CASE ;
	END PROCESS ;
END Structure ;
```

The simulation waveforms can be seen as follows.



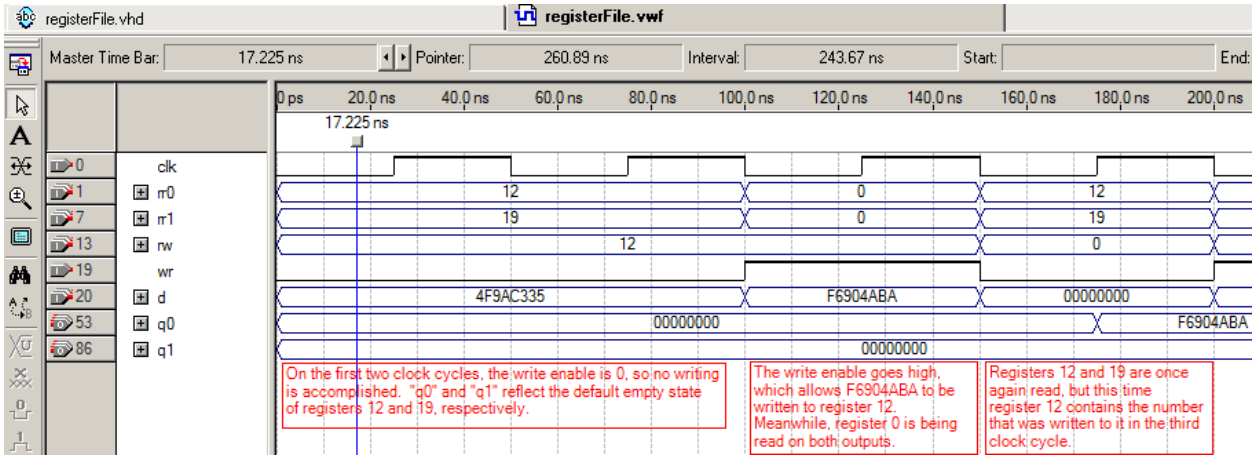
The device performed as expected.

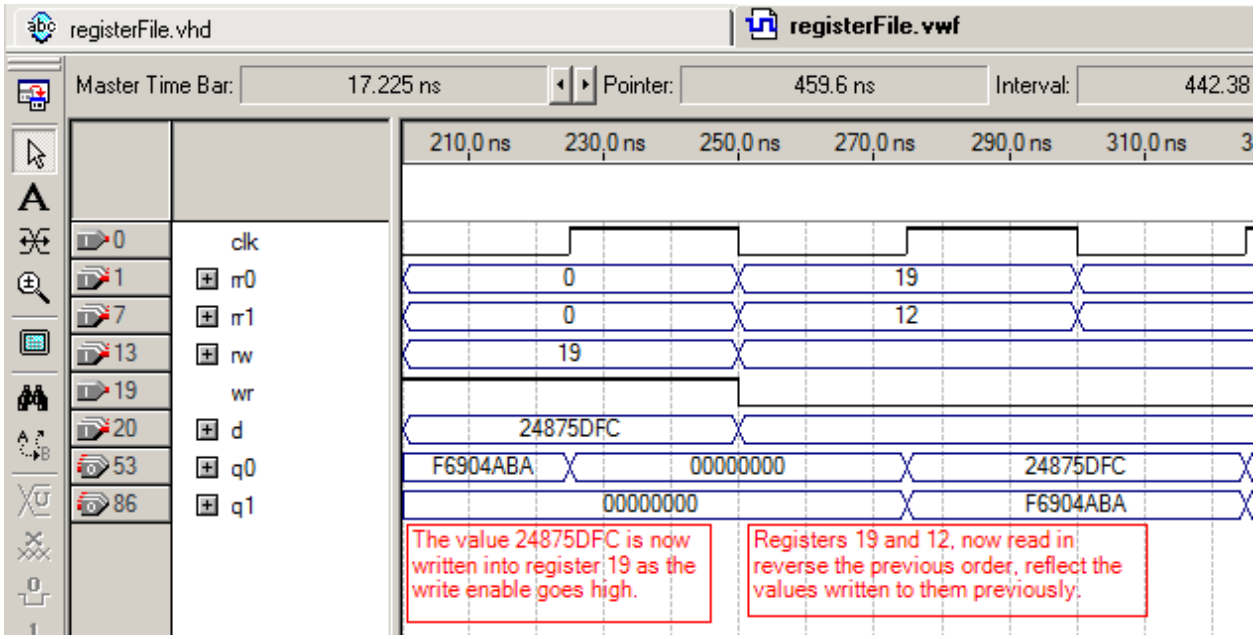
The final device for this project was a register file called “registerFile.”. The register file contains all 32 internal registers for the MIPS microprocessor. Each register in the register file is multiplexed at the input and the output to determine which muxes are written to and/or read from on any given operation. Two registers can be read from at once, while only one register can be written to. This allows register-register operations, where both inputs come simultaneously from separate registers, and the output is written to a third. Because the zero register cannot be written to, no control logic was created for its modification. Any attempt to write over its default value of zero will be ignored.

The device was implemented using the 32-bit register module, “reg32,” created in Assignment 2. The device was multiplexed internally, and the control logic for the write enables also was handled internally. Thus, the only modular component used was “reg32.” This was not an inefficient implementation in terms of architecture, but whether the coding would have gone faster had I used a 2-D array is subject to debate. But I’ll leave that one to the philosophers.

One other thing worth noting is that in its current form, the device performs on a purely synchronous basis. In future use, the rising-clock synchronization of the register file outputs will probably be replaced by an asynchronous realization that changes the output as soon as the read register control input changes.

Due to its length, the code for “registerFile” can be found in Appendix B. The simulation waveforms are as follows.





The device performs as expected.

APPENDIX A: Textbook Problems (From the **UN**revised Edition)

4.2.1) a) From the components in Figure 4.2, this instruction will use the PC, the instruction memory, the register file (both read ports and the write port), and the ALU.

b) Again from Figure 4.2, this instruction will use the PC, the instruction memory, the register file (one read port and the write port), and the mux between the instruction memory and the ALU.

4.2.2) a) This instruction will need a third register read port and a third ALU input port.

b) This instruction will need a shifting module, which probably should be placed in the ALU.

4.2.3) a) This instruction will need an additional ALU control signal that would correspond to a 3-input addition.

b) This instruction will need a ALU new control signal as well.

4.8.1) a) This bit is used only for shift amount (R-type) or constant/address value (I-type). One way would be to load a known value into a register and write it to address 0x110000000. If this value appears instead in address 0x100000000, then the bit is stuck at zero.

b) This signal is used only when loading data from memory. If it is stuck at zero, then the ouput of the ALU will be read instead of the memory. So one solution would be to write a certain number to memory and then attempt to load a register from that location in memory, referenced by a lower memory value with an offset, which must be different from the value in memory. The output of the ALU will be the offset, so if the register is loaded with this value after execution, then the signal is stuck on zero.

4.8.2) a) The same test as above could be used (using the memory values in opposite roles), or an ORI with 0000 and the zero register could be executed. If bit 7 of the result is 1, then the signal is stuck at 1.

b) The test above can't be used because there would be no way to know whether anything read from memory was already there or not. However, if this signal is stuck at 1, then it will not be possible to read from the ALU output. Any simple ALU operation would not assert this signal, so if a register's value does not reflect the output of the ALU, then the signal is stuck.

4.8.3) a) Yes. There is always more than one way to skin a cat when it comes to constants and offsets. This would require some more shifting around and some creative use of registers, but you could make it work.

b) No. If a register can't read from the ALU output, then nothing useful can be done.

APPENDIX B: Code for “registerFile”

```
-- Bob Minchin
-- EEL 4713C
-- Homework 3
-- Part 3.4: 32-Bit, 32-Register File

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

ENTITY registerFile IS
      --32-bit input, output
      --5-bit mux signals
      --1-bit clock, write enable
      PORT (      rr0, rr1, rw :    IN STD_LOGIC_VECTOR(4 DOWNTO 0) ;
                  d :               IN STD_LOGIC_VECTOR(31 DOWNTO 0) ;
                  clk, wr :         IN STD_LOGIC ;
                  q0, q1 :          OUT STD_LOGIC_VECTOR(31 DOWNTO 0) ) ;
END registerFile ;

ARCHITECTURE Structure OF registerFile IS
      --Temporary signals between output of registers and output of device
      --Arrays would have been better
      SIGNAL out00, out01, out02, out03, out04, out05, out06, out07, out08,
            out09, out10, out11, out12, out13, out14, out15, out16, out17,
            out18, out19, out20, out21, out22, out23, out24, out25, out26,
            out27, out28, out29, out30, out31 : STD_LOGIC_VECTOR(31 DOWNTO
                  0) ;
      --Individual write enables for the 32 muxes
     SIGNAL wr01, wr02, wr03, wr04, wr05, wr06, wr07, wr08, wr09, wr10, wr11,
            wr12, wr13, wr14, wr15, wr16, wr17, wr18, wr19, wr20, wr21, wr22,
            wr23, wr24, wr25, wr26, wr27, wr28, wr29, wr30, wr31  :
STD_LOGIC ;
      --“reg32” component from Assignment 2
      COMPONENT reg32 IS
            PORT (      D :               IN STD_LOGIC_VECTOR(31 DOWNTO 0);
                        Clk, Wr, Clr :    IN STD_LOGIC ;
                        Q :               OUT STD_LOGIC_VECTOR(31 DOWNTO
                              0) ) ;
      END COMPONENT ;
BEGIN
      PROCESS(rr0, clk)
      BEGIN
            --Control logic for output 1
            IF clk'EVENT AND clk = '1' THEN
                  CASE rr0 IS
                        WHEN "00000" => q0 <= out00 ;
                        WHEN "00001" => q0 <= out01 ;
                        WHEN "00010" => q0 <= out02 ;
                        WHEN "00011" => q0 <= out03 ;
                        WHEN "00100" => q0 <= out04 ;
                        WHEN "00101" => q0 <= out05 ;
                        WHEN "00110" => q0 <= out06 ;
                        WHEN "00111" => q0 <= out07 ;
                        WHEN "01000" => q0 <= out08 ;
```

```
                  WHEN "01001" => q0 <= out09 ;
                  WHEN "01010" => q0 <= out10 ;
                  WHEN "01011" => q0 <= out11 ;
                  WHEN "01100" => q0 <= out12 ;
                  WHEN "01101" => q0 <= out13 ;
                  WHEN "01110" => q0 <= out14 ;
                  WHEN "01111" => q0 <= out15 ;
                  WHEN "10000" => q0 <= out16 ;
                  WHEN "10001" => q0 <= out17 ;
                  WHEN "10010" => q0 <= out18 ;
                  WHEN "10011" => q0 <= out19 ;
                  WHEN "10100" => q0 <= out20 ;
                  WHEN "10101" => q0 <= out21 ;
                  WHEN "10110" => q0 <= out22 ;
                  WHEN "10111" => q0 <= out23 ;
                  WHEN "11000" => q0 <= out24 ;
                  WHEN "11001" => q0 <= out25 ;
                  WHEN "11010" => q0 <= out26 ;
                  WHEN "11011" => q0 <= out27 ;
                  WHEN "11100" => q0 <= out28 ;
                  WHEN "11101" => q0 <= out29 ;
                  WHEN "11110" => q0 <= out30 ;
                  WHEN "11111" => q0 <= out31 ;
                  WHEN OTHERS =>
            END CASE ;
      END IF ;
END PROCESS ;
PROCESS(rr1, clk)
BEGIN
      --Control logic for output 2
      IF clk'EVENT AND clk = '1' THEN
            CASE rr1 IS
                  WHEN "00000" => q1 <= out00 ;
                  WHEN "00001" => q1 <= out01 ;
                  WHEN "00010" => q1 <= out02 ;
                  WHEN "00011" => q1 <= out03 ;
                  WHEN "00100" => q1 <= out04 ;
                  WHEN "00101" => q1 <= out05 ;
                  WHEN "00110" => q1 <= out06 ;
                  WHEN "00111" => q1 <= out07 ;
                  WHEN "01000" => q1 <= out08 ;
                  WHEN "01001" => q1 <= out09 ;
                  WHEN "01010" => q1 <= out10 ;
                  WHEN "01011" => q1 <= out11 ;
                  WHEN "01100" => q1 <= out12 ;
                  WHEN "01101" => q1 <= out13 ;
                  WHEN "01110" => q1 <= out14 ;
                  WHEN "01111" => q1 <= out15 ;
                  WHEN "10000" => q1 <= out16 ;
                  WHEN "10001" => q1 <= out17 ;
                  WHEN "10010" => q1 <= out18 ;
                  WHEN "10011" => q1 <= out19 ;
                  WHEN "10100" => q1 <= out20 ;
                  WHEN "10101" => q1 <= out21 ;
                  WHEN "10110" => q1 <= out22 ;
                  WHEN "10111" => q1 <= out23 ;
                  WHEN "11000" => q1 <= out24 ;
```

```
                  WHEN "11001" => q1 <= out25 ;
                  WHEN "11010" => q1 <= out26 ;
                  WHEN "11011" => q1 <= out27 ;
                  WHEN "11100" => q1 <= out28 ;
                  WHEN "11101" => q1 <= out29 ;
                  WHEN "11110" => q1 <= out30 ;
                  WHEN "11111" => q1 <= out31 ;
                  WHEN OTHERS =>
            END CASE ;
      END IF ;
END PROCESS ;
PROCESS(rw, wr)
BEGIN
      --all write enable initialized to 0
      wr01 <= '0' ;
      wr02 <= '0' ;
      wr03 <= '0' ;
      wr04 <= '0' ;
      wr05 <= '0' ;
      wr06 <= '0' ;
      wr07 <= '0' ;
      wr08 <= '0' ;
      wr09 <= '0' ;
      wr10 <= '0' ;
      wr11 <= '0' ;
      wr12 <= '0' ;
      wr13 <= '0' ;
      wr14 <= '0' ;
      wr15 <= '0' ;
      wr16 <= '0' ;
      wr17 <= '0' ;
      wr18 <= '0' ;
      wr19 <= '0' ;
      wr20 <= '0' ;
      wr21 <= '0' ;
      wr22 <= '0' ;
      wr23 <= '0' ;
      wr24 <= '0' ;
      wr25 <= '0' ;
      wr26 <= '0' ;
      wr27 <= '0' ;
      wr28 <= '0' ;
      wr29 <= '0' ;
      wr30 <= '0' ;
      wr31 <= '0' ;
      CASE rw IS
            --control logic for write enables
            WHEN "00001" =>
                  IF wr = '1' THEN wr01 <= '1' ;
                  END IF ;
            WHEN "00010" =>
                  IF wr = '1' THEN wr02 <= '1' ;
                  END IF ;
            WHEN "00011" =>
                  IF wr = '1' THEN wr03 <= '1' ;
                  END IF ;
            WHEN "00100" =>
```

```
      IF wr = '1' THEN wr04 <= '1' ;
      END IF ;
WHEN "00101" =>
      IF wr = '1' THEN wr05 <= '1' ;
      END IF ;
WHEN "00110" =>
      IF wr = '1' THEN wr06 <= '1' ;
      END IF ;
WHEN "00111" =>
      IF wr = '1' THEN wr07 <= '1' ;
      END IF ;
WHEN "01000" =>
      IF wr = '1' THEN wr08 <= '1' ;
      END IF ;
WHEN "01001" =>
      IF wr = '1' THEN wr09 <= '1' ;
      END IF ;
WHEN "01010" =>
      IF wr = '1' THEN wr10 <= '1' ;
      END IF ;
WHEN "01011" =>
      IF wr = '1' THEN wr11 <= '1' ;
      END IF ;
WHEN "01100" =>
      IF wr = '1' THEN wr12 <= '1' ;
      END IF ;
WHEN "01101" =>
      IF wr = '1' THEN wr13 <= '1' ;
      END IF ;
WHEN "01110" =>
      IF wr = '1' THEN wr14 <= '1' ;
      END IF ;
WHEN "01111" =>
      IF wr = '1' THEN wr15 <= '1' ;
      END IF ;
WHEN "10000" =>
      IF wr = '1' THEN wr16 <= '1' ;
      END IF ;
WHEN "10001" =>
      IF wr = '1' THEN wr17 <= '1' ;
      END IF ;
WHEN "10010" =>
      IF wr = '1' THEN wr18 <= '1' ;
      END IF ;
WHEN "10011" =>
      IF wr = '1' THEN wr19 <= '1' ;
      END IF ;
WHEN "10100" =>
      IF wr = '1' THEN wr20 <= '1' ;
      END IF ;
WHEN "10101" =>
      IF wr = '1' THEN wr21 <= '1' ;
      END IF ;
WHEN "10110" =>
      IF wr = '1' THEN wr22 <= '1' ;
      END IF ;
WHEN "10111" =>
```

```
                IF wr = '1' THEN wr23 <= '1' ;
                END IF ;
            WHEN "11000" =>
                IF wr = '1' THEN wr24 <= '1' ;
                END IF ;
            WHEN "11001" =>
                IF wr = '1' THEN wr25 <= '1' ;
                END IF ;
            WHEN "11010" =>
                IF wr = '1' THEN wr26 <= '1' ;
                END IF ;
            WHEN "11011" =>
                IF wr = '1' THEN wr27 <= '1' ;
                END IF ;
            WHEN "11100" =>
                IF wr = '1' THEN wr28 <= '1' ;
                END IF ;
            WHEN "11101" =>
                IF wr = '1' THEN wr29 <= '1' ;
                END IF ;
            WHEN "11110" =>
                IF wr = '1' THEN wr30 <= '1' ;
                END IF ;
            WHEN "11111" =>
                IF wr = '1' THEN wr31 <= '1' ;
                END IF ;
            WHEN OTHERS =>
      END CASE ;
END PROCESS ;

Reg00 :     reg32 PORT MAP (d, clk, '0', '1', out00) ;
Reg01 :     reg32 PORT MAP (d, clk, wr01, '1', out01) ;
Reg02 :     reg32 PORT MAP (d, clk, wr02, '1', out02) ;
Reg03 :     reg32 PORT MAP (d, clk, wr03, '1', out03) ;
Reg04 :     reg32 PORT MAP (d, clk, wr04, '1', out04) ;
Reg05 :     reg32 PORT MAP (d, clk, wr05, '1', out05) ;
Reg06 :     reg32 PORT MAP (d, clk, wr06, '1', out06) ;
Reg07 :     reg32 PORT MAP (d, clk, wr07, '1', out07) ;
Reg08 :     reg32 PORT MAP (d, clk, wr08, '1', out08) ;
Reg09 :     reg32 PORT MAP (d, clk, wr09, '1', out09) ;
Reg10 :     reg32 PORT MAP (d, clk, wr10, '1', out10) ;
Reg11 :     reg32 PORT MAP (d, clk, wr11, '1', out11) ;
Reg12 :     reg32 PORT MAP (d, clk, wr12, '1', out12) ;
Reg13 :     reg32 PORT MAP (d, clk, wr13, '1', out13) ;
Reg14 :     reg32 PORT MAP (d, clk, wr14, '1', out14) ;
Reg15 :     reg32 PORT MAP (d, clk, wr15, '1', out15) ;
Reg16 :     reg32 PORT MAP (d, clk, wr16, '1', out16) ;
Reg17 :     reg32 PORT MAP (d, clk, wr17, '1', out17) ;
Reg18 :     reg32 PORT MAP (d, clk, wr18, '1', out18) ;
Reg19 :     reg32 PORT MAP (d, clk, wr19, '1', out19) ;
Reg20 :     reg32 PORT MAP (d, clk, wr20, '1', out20) ;
Reg21 :     reg32 PORT MAP (d, clk, wr21, '1', out21) ;
Reg22 :     reg32 PORT MAP (d, clk, wr22, '1', out22) ;
Reg23 :     reg32 PORT MAP (d, clk, wr23, '1', out23) ;
Reg24 :     reg32 PORT MAP (d, clk, wr24, '1', out24) ;
Reg25 :     reg32 PORT MAP (d, clk, wr25, '1', out25) ;
Reg26 :     reg32 PORT MAP (d, clk, wr26, '1', out26) ;
```

```
      Reg27 :     reg32 PORT MAP (d, clk, wr27, '1', out27) ;
      Reg28 :     reg32 PORT MAP (d, clk, wr28, '1', out28) ;
      Reg29 :     reg32 PORT MAP (d, clk, wr29, '1', out29) ;
      Reg30 :     reg32 PORT MAP (d, clk, wr30, '1', out30) ;
      Reg31 :     reg32 PORT MAP (d, clk, wr31, '1', out31) ;

END STRUCTURE ;
```