Bob Minchin

EEL4713C

Assignment 2

January 28, 2012

INTRODUCTION

The purpose of this assignment was to reinforce some of the basic principles covered in class by guiding the student through designing some basic components that will be needed to build into a microprocessor pipeline later in the semester. Three multiplexers, a register, and an extender were assigned, and each was designed and simulated according to the directions.

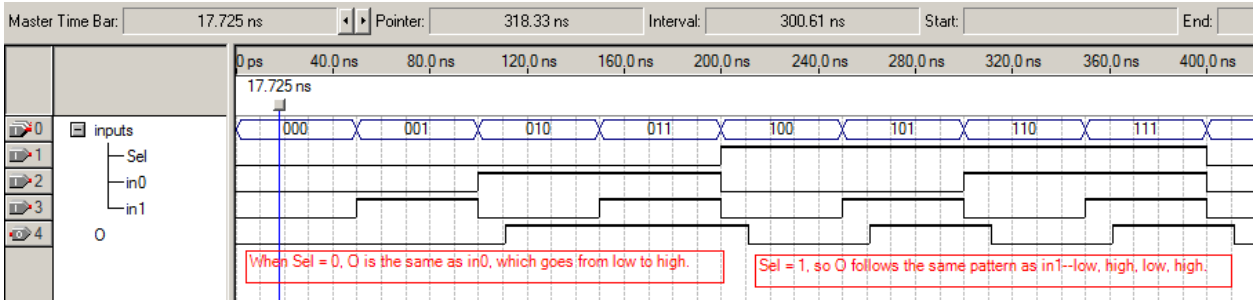
With one exception, which will be detailed in the design section, the EP2C8T144C8 chip was used in all design, compilation, and simulation. As noted in class discussion, the Stratix II would not compile in the Web Edition.

DESIGN AND TESTING

The first device was a simple 1-bit multiplexer that was named "mux1." It was designed to select between two 1-bit input signals. It was accomplished using the following code.

```
-- Bob Minchin
-- EEL 4713C
-- Homework 2
-- Part 3.1: 1-Bit Mux

LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux1 IS
     --1-bit inputs, output
     PORT (     in0, in1, Sel :  IN STD_LOGIC ;
                O :              OUT STD_LOGIC ) ;
END mux1 ;

ARCHITECTURE Structure OF mux1 IS
BEGIN
     O <= in1 WHEN Sel = '1' ELSE in0 ;     --Selects in0 when Sel = 0
END Structure ;                             --and in1 when Sel = 1
```

The following is a screenshot of the simulation of this device.



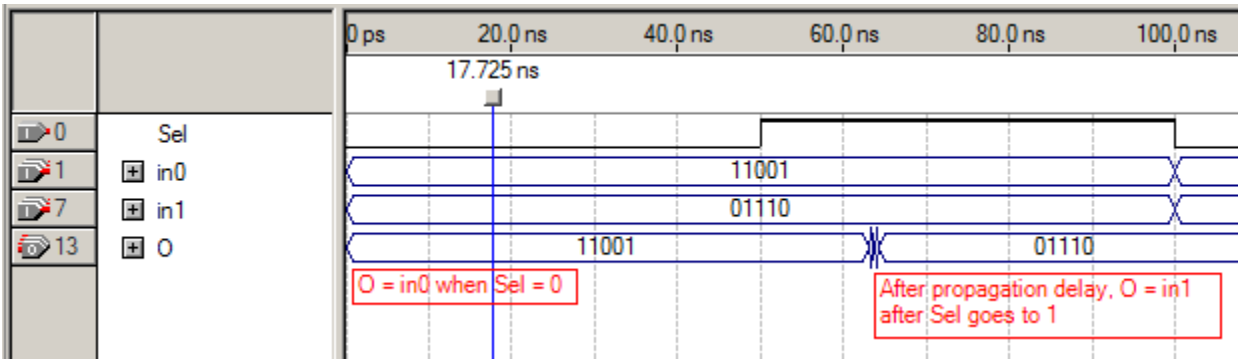
The device performed as expected and had a worst-case propagation delay of 13.172 ns.

The next device was a 5-bit multiplexer, which was accomplished by extending the 1-bit mux from the previous part. The purpose of this 5-bit multiplexer is ostensibly to select from the registers in the MIPS pipeline design. The two source registers and destination register in the coding convention are represented by 5 bits because there are $2^5 = 32$ registers in the MIPS architecture.

This 5-bit mux, "mux5," was accomplished with the following code.

```
-- Bob Minchin
-- EEL 4713C
-- Homework 2
-- Part 3.2: 5-Bit Mux

LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux5 IS
      --5-bit inputs, output
      PORT (     in0, in1 : IN STD_LOGIC_VECTOR(4 DOWNTO 0);
                 Sel :      IN STD_LOGIC ;
                 O :        OUT STD_LOGIC_VECTOR(4 DOWNTO 0) ) ;
END mux5 ;

ARCHITECTURE Structure OF mux5 IS
BEGIN
      O <= in1 WHEN Sel = '1' ELSE in0 ;     --Selects in0 when Sel = 0
END Structure ;                              --and in1 when Sel = 1
```

This is a literal translation of the mux1 code into 5 bits by expanding the std_logic inputs to 5-bit vectors. The following is a screenshot of the simulation.



The device performed as expected and had a worst-case propagation delay of 14.854 ns.

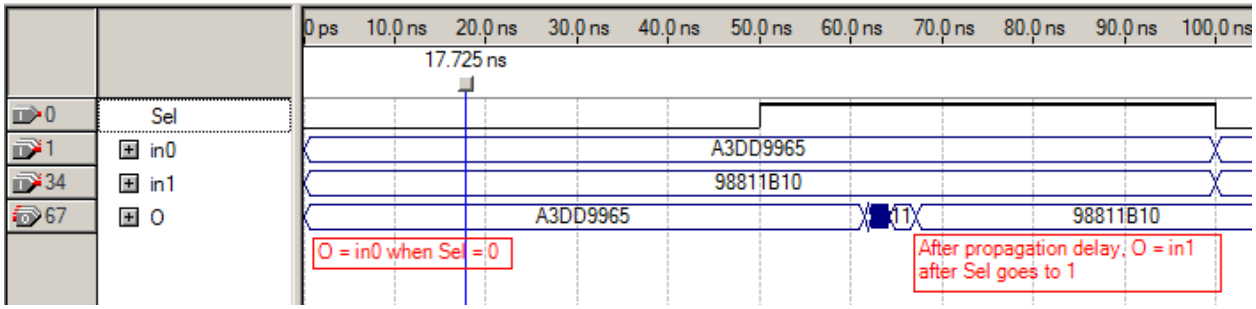
The next part was another extension of the first part, this time to 32 bits. This was called "mux32." This device, however, caused problems with the compilation. Because the inputs were 32 bits apiece and the output also was 32 bits, successful compilation required that the project be built on a device that had at least 32 + 32 + 32 + 1 (select bit) = 97 pins, a number which exceeded the 85 pins available on the EP2C8T144C8. For this reason, a larger chip had to be selected for compilation and simulation of mux32.

Per the suggestion in the class e-mail, the large EP2C70F896C8 device was used only for the purposes of testing in this assignment. For future use, the device will be changed back to EP2C8T144C8 so that it can be used with the other parts in a larger circuit.

The code for mux32 can be seen as follows.

```
-- Bob Minchin
-- EEL 4713C
-- Homework 2
-- Part 3.3: 32-Bit Mux

LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux32 IS
      --32-bit inputs and output
      PORT (     in0, in1 : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
                 Sel :      IN STD_LOGIC ;
                 O :        OUT STD_LOGIC_VECTOR(31 DOWNTO 0) ) ;
END mux32 ;

ARCHITECTURE Structure OF mux32 IS
BEGIN
      O <= in1 WHEN Sel = '1' ELSE in0 ;     --Selects in0 when Sel = 0
END Structure ;                              --and in1 when Sel = 1
```

Again, this is just a vector extension of the original mux1. The simulation can be seen as follows:



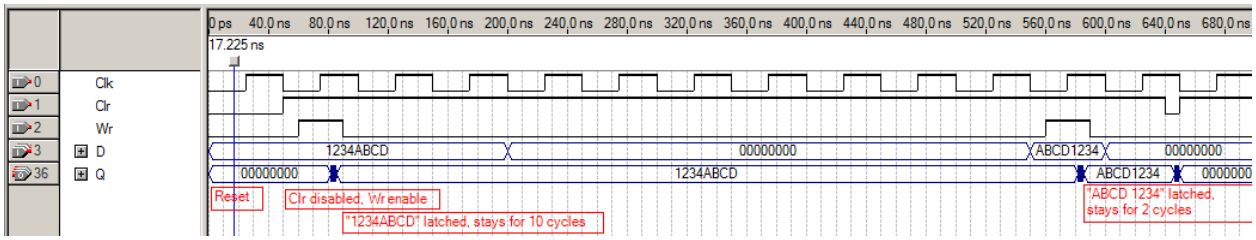
The device performed as expected and had a worst-case propagation delay of 17.665 ns. The more complicated muxes have longer propagation delays because more inputs and outputs means more lookup tables. Larger lookup tables translate to longer latency times.

The next device was something different—a register. It simply needed the capacity to latch in 32 bits of data on an active high clock while featuring an enable and an asynchronous reset. The code to implement this can be seen as follows:

```
-- Bob Minchin
-- EEL 4713C
-- Homework 2
-- Part 3.4: 32-Bit Register
```

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

ENTITY reg32 IS
      --32-bit input, output; 1-bit signal bits
      PORT (     D :              IN STD_LOGIC_VECTOR(31 DOWNTO 0);
                 Clk, Wr, Clr :   IN STD_LOGIC ;
                 Q :              OUT STD_LOGIC_VECTOR(31 DOWNTO 0) ) ;
END reg32 ;

ARCHITECTURE Structure OF reg32 IS
      SIGNAL Q_temp :  STD_LOGIC_VECTOR(31 DOWNTO 0) ;
BEGIN
      --All inputs in process declaration
      PROCESS(D, Clk, Wr, Clr)
            BEGIN
            IF Clr = '0' THEN                        --Asynch reset
                  Q_temp <= x"00000000" ;            --(active low)
            ELSIF (Clk'event and Clk = '1') THEN     --Latch on Clk if Wr
                  IF Wr = '1' THEN                   --enabled
                        Q_temp <= D ;
                  END IF ;
            END IF ;
      END PROCESS ;
      Q <= Q_temp ;
END Structure ;
```

The simulation is as follows.



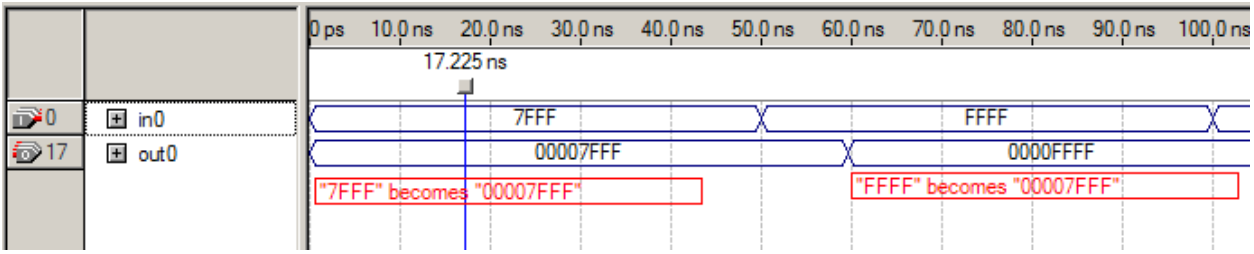
The fastest clock rate to use on this device would probably be related to the worst-case time on the clock-to-output delay, which in this case is 9.94 ns. This clock rate would be 1 / 9.94 ns = 100 MHz.

The next part was a 16-to-32-bit zero extender. This device takes a 16-bit input and outputs its 32-bit, zero-extended equivalent, which can then be used in a 32-bit pipeline. This is useful for unsigned numbers. The code for this part, “zeroext,” can be seen as follows.

```
-- Bob Minchin
-- EEL 4713C
-- Homework 2
-- Part 3.5: 16-Bit Zero Extender
```

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

ENTITY zeroext IS
      --16-bit input, 32-bit output
      PORT (     in0 : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
                 out0 :     OUT STD_LOGIC_VECTOR(31 DOWNTO 0) ) ;
END zeroext ;

ARCHITECTURE Structure OF zeroext IS
BEGIN
      out0(15 DOWNTO 0) <= in0 ;       --16 LSB = in0
      out0(31 DOWNTO 16) <= x"0000" ;  --16 MSB = “0000000000000000”
END Structure ;
```

The simulation is as follows.



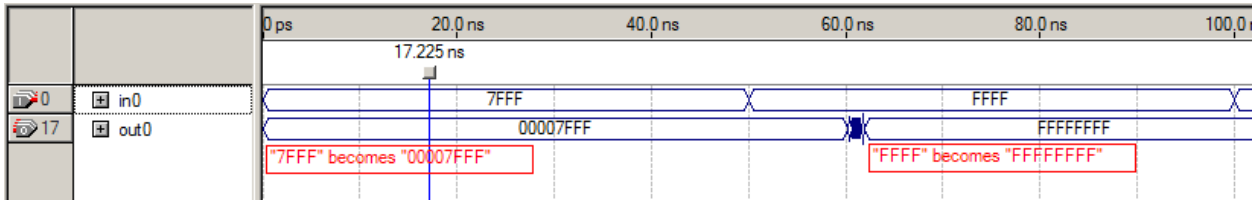
The device functioned as expected.

The next part was the same idea, only modified to accommodate signed numbers. This new part, the sign extender, takes a 16-bit input and extends its sign bit (bit 15) into all 16 most significant bits of its 32-bit representation. The code to implement this part, “signext,” is as follows.

```
-- Bob Minchin
-- EEL 4713C
-- Homework 2
-- Part 3.6: 16-Bit Sign Extender

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

ENTITY signext IS
      --16-bit input, 32-bit output
      PORT (     in0 : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
                 out0 :     OUT STD_LOGIC_VECTOR(31 DOWNTO 0) ) ;
END signext ;

ARCHITECTURE Structure OF signext IS
BEGIN
      out0(15 DOWNTO 0) <= in0 ;                        --16 LSB = in0
```

```
      out0(31 DOWNTO 16) <= (others => in0(15) ) ;      --16 MSB = sign
END Structure ;
```

The simulation follows.



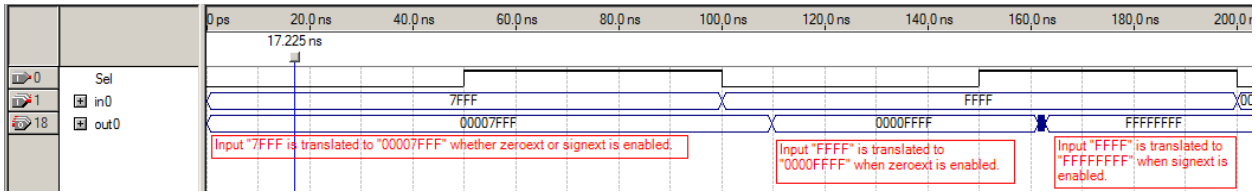
The device performs as expected.

The final part was a multiplexed extender that would allow the controller to determine whether a sign extension or a zero extension should be performed on a number. Such a device is useful because it allows for signed numbers while allowing the full use of all 16 bits of an unsigned number, which is not possible when a sign extension is applied. It was decided that the best way to accomplish this would be to attach the outputs of zeroext and signext to mux32 using port maps. The code to accomplish this part, "extender," is as follows.

```
-- Bob Minchin
-- EEL 4713C
-- Homework 2
-- Part 3.7: 16-Bit Extender

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

ENTITY extender IS
      --16-bit input, 32-bit output, 1-bit control signal
      PORT (      in0 :       IN STD_LOGIC_VECTOR(15 DOWNTO 0);
                  Sel :       IN STD_LOGIC ;
                  out0 :      OUT STD_LOGIC_VECTOR(31 DOWNTO 0) ) ;
END extender ;

ARCHITECTURE Structure OF extender IS

      --Internal signals
      SIGNAL signout : STD_LOGIC_VECTOR(31 DOWNTO 0) ;
      SIGNAL zeroout : STD_LOGIC_VECTOR(31 DOWNTO 0) ;

      --Declaration of included components
      COMPONENT mux32 IS
            PORT (      in0, in1 : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
                        Sel :       IN STD_LOGIC ;
                        O :         OUT STD_LOGIC_VECTOR(31 DOWNTO 0) ) ;
      END COMPONENT ;
```

```
	COMPONENT zeroext IS
		PORT (	in0 : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
			out0 :	OUT STD_LOGIC_VECTOR(31 DOWNTO 0) ) ;
	END COMPONENT ;

	COMPONENT signext IS
		PORT (	in0 : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
			out0 :	OUT STD_LOGIC_VECTOR(31 DOWNTO 0) ) ;
	END COMPONENT ;

BEGIN

	--Component connections
	Mux : mux32 PORT MAP (
		zeroout, signout, Sel, out0 ) ;

	ZeroExtend : zeroext PORT MAP (
		in0, zeroout ) ;

	SignExtend : signext PORT MAP (
		in0, signout ) ;

END Structure ;
```

The simulation can be seen as follows.



The device performs as expected.

APPENDIX: Textbook Problems

2.11.4) The first is an R-type instruction because it contains information for specific registers rd and shamt. The second is an I-type instruction because it contains a constant.

2.11.5) The first is an add instruction. It adds the contents of R1 and R2 and stores the result in R3.

```
add $v1, $at, $v0
```

The second is a store instruction. It stores the word located in R16 in the memory location one word after the one pointed to by the contents of R5.

```
sw $a1, 4($s0)
```

2.11.6) 000000 00001 00010 00011 00000 100000

101011 01000 00101 0000000000000100

2.14.1) a)

```
srl $t1, $t0, 5
andi $t1, $t1, 0x0001ffff
```

b)

```
sll $t1, $t0, 9
andi $t1, $t1, 0xffff8000
```

2.14.2) a)

```
andi $t1, $t0, 0x0000000f
```

b)

```
sll $t1, $t0, 14
andi $t1, $t1, 0x0003C000
```

2.14.3) a)

```
srl $t1, $t0, 28
```

b)

```
srl $t1, $t1, 14
andi $t1, $t1, 0x0001C000
```

2.21.1) a) For the call to main, the stack moves back by 4 addresses. For the call to leaf_function, the stack moves back an additional 4 addresses.

b) Same as above, except this time the value 100 is stored at 0x10008000 under the label my_global.

2.21.2) a)

```
main:   addi $sp, $sp, -4
        sw $ra, 0($sp)
        addi $a0, $0, 1
```

```
                jal leaf_function
                lw $ra, 0($sp)
                addi $sp, $sp, 4
                jr $ra
leaf_function:  addi $sp, $sp, -4
                sw $ra, 0($sp)
                addi $t0, $a0, 1
                addi $t1, $0, 5
                beq $t0, $t1, done
                add $a0, $t0, $0
                jal leaf_function
done:           add $v0, $t0, $0
                lw $ra, 0($sp)
                addi $sp, $sp, 4
                jr $ra
```

b) This assumes that my_global has already been attached to $s0.

```
main:           addi $sp, $sp, -4
                sw $ra, 0($sp)
                addi $a0, $0, 10
                addi $t0, $0, 20
                lw $a1, 0($s0)
                jal my_function
                add $t2, $v0, $0
                lw $ra, 0($sp)
                addi $sp, $sp, 4
```

```
                jr $ra
my_function:    addi $sp, $sp, -4
                sw $ra, 0($sp)
                sub $v0, $a0, $a1
                lw $ra, 0($sp)
                addi $sp, $sp, 4
                jr $ra
```