A Combinational Sign-magnitude to Two's Complement Converter
Coen 313

Adrian PATTERSON
ID: 40048841

March 18, 2020

Date Performed:	March 4, 2020
Instructor:	Professor Le Beux
Section:	UI-X

UNIVERSITÉ Concordia UNIVERSITY

GINA CODY SCHOOL OF ENGINEERING AND COMPUTER SCIENCE

1 Objective

In experiment 3, we implemented a sign-magnitude 2's complement converter VHDL code and synthesized it on the Xilinx Nexys FPGA board. The .vhd code was first written to define the circuit's input/output, along with an .xdc constraint file to define which ports map to which real-life switches and LEDs on the FPGA board. After a successful synthesis, the same VHDL code was then simulated using Modelsim, and it's outputs recorded. Our primary objective in performing this experiment was to familiarize oneself with combinational programming in VHDL. Our secondary objective was to make use of the command line options using vcom options to test the correctness of our code as well as other useful functions. Overall, experiment three was a success in both the FPGA synthesis and the simulation.

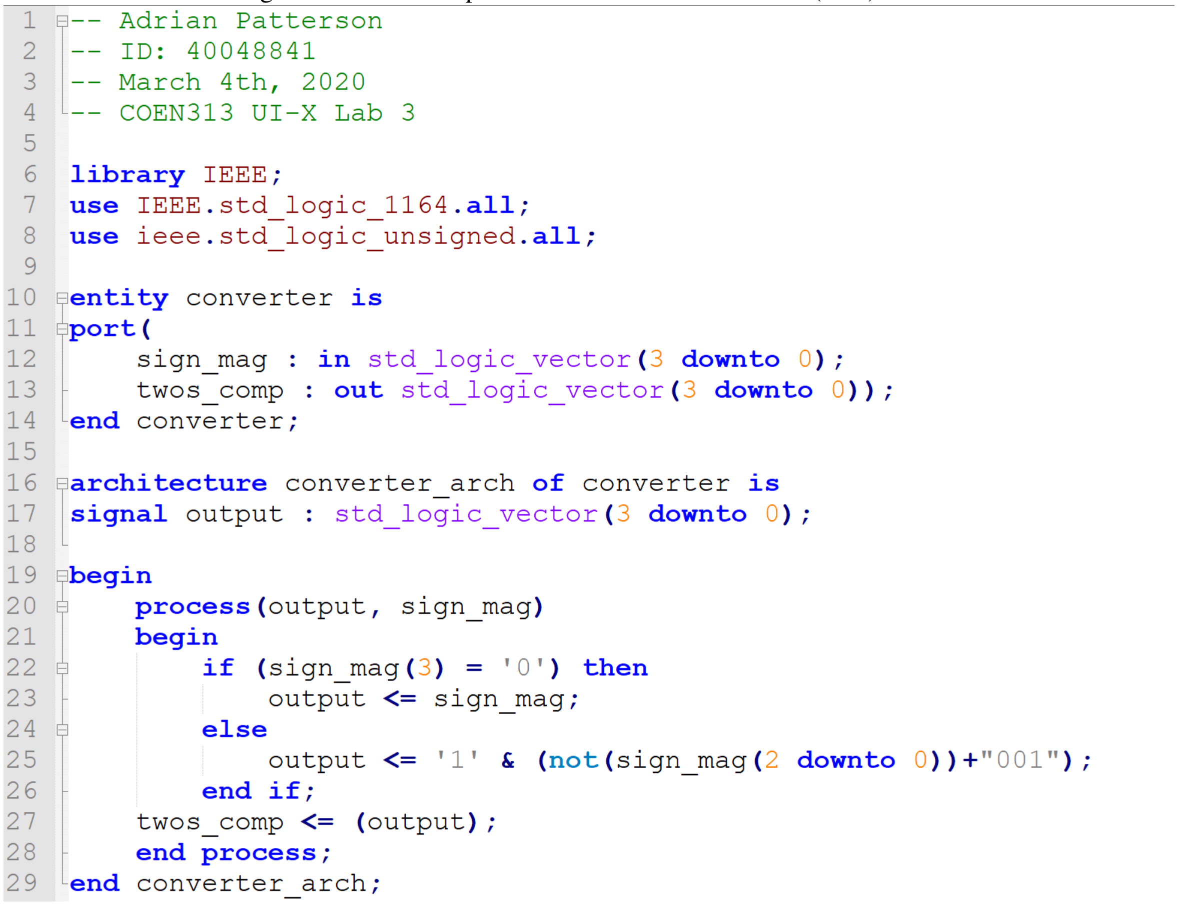
2 Procedure

There were two main procedures involved in this experiment. The first was the synthesis; the second was the simulation.These two portions will be discussed in the following two sections, respectively.

2.1 Synthesis

Before performing the lab, two files were created. The first file, named `twos_complement.vhd`, contains the VHDL code which implements a signed 4 bit integer to two's complement converter. The second file, named `twos_complement.xdc`, describes the relationship between port inputs/outputs in the VHDL code and the physical FPGA board switches and LEDs. The two codes displayed below in figures one and two.

Figure 1: Two's Complement Converter VHDL Code (.vhd)



```vhdl
-- Adrian Patterson
-- ID: 40048841
-- March 4th, 2020
-- COEN313 UI-X Lab 3

library IEEE;
use IEEE.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity converter is
port(
    sign_mag : in std_logic_vector(3 downto 0);
    twos_comp : out std_logic_vector(3 downto 0));
end converter;

architecture converter_arch of converter is
signal output : std_logic_vector(3 downto 0);

begin
    process(output, sign_mag)
    begin
        if (sign_mag(3) = '0') then
            output <= sign_mag;
        else
            output <= '1' & (not(sign_mag(2 downto 0))+"001");
        end if;
    twos_comp <= (output);
    end process;
end converter_arch;
```

In the VHDL code above, the entity is first declared. A four-bit input named `sign_mag` is declared along with a four-bit output named `twos_comp`. Then, the architecture for the circuit is described. A signal `output` is used during the conditional assignments, which is then assigned to the output. If the inputs most significant bit is 0, then the output is simply the input. This is because the sign is a non-negative number, meaning its two's complement is identical to its original value. If the most significant big is not 0, then we know our number is negative. To convert it into its two's complement, the following method is used: the three least significant bits of the input are inverted and incremented by one, then a '1' is concatenated to the left of this. This produces the two's complement. This also follows the theory learned in Coen 212 (i.e. invert the number, add one).

Figure 2: Two's Complement Converter Constraint File (.xdc)

```
# Adrian Patterson
# ID: 40048841
# March 4th, 2020
# COEN313 UI-X Lab 3

set_property -dict { PACKAGE_PIN J15 IOSTANDARD LVCMOS33 } [ get_ports { sign_mag[0] } ] ;
set_property -dict { PACKAGE_PIN L16 IOSTANDARD LVCMOS33 } [ get_ports { sign_mag[1] } ] ;
set_property -dict { PACKAGE_PIN M13 IOSTANDARD LVCMOS33 } [ get_ports { sign_mag[2] } ] ;
set_property -dict { PACKAGE_PIN R15 IOSTANDARD LVCMOS33 } [ get_ports { sign_mag[3] } ] ;
set_property -dict { PACKAGE_PIN H17 IOSTANDARD LVCMOS33 } [ get_ports { twos_comp[0] } ] ;
set_property -dict { PACKAGE_PIN K15 IOSTANDARD LVCMOS33 } [ get_ports { twos_comp[1] } ] ;
set_property -dict { PACKAGE_PIN J13 IOSTANDARD LVCMOS33 } [ get_ports { twos_comp[2] } ] ;
set_property -dict { PACKAGE_PIN N14 IOSTANDARD LVCMOS33 } [ get_ports { twos_comp[3] } ] ;
```

As aforementioned, the constraint file defines which ports in the code map to which real life components. As seen in the code above, the four input bits of `sign_mag` are assigned to switches J15, L16, M13, and R15. Then, the four output bits of `twos_comp` are assigned to LEDs H17, K15, J13, and N14. In Vivado, this file is set as the target constraint file to specify that the Xilinx FPGA board will be using these constraints to implemend the VHDL code defined in figure 1.

3 Results

The results to be discussed and elaborated on include the synthesis results and the simulation results. They will be discussed separately in the following two sections.

3.1 Synthesis Results

After the VHDL and constraint files were configured and synthesized using Vivado, the board was tested. The outputs of the table are as follow in table 1. Note that the values for `sign_mag` and `twos_comp` correspond with the switches and the LEDs, respectively.

Table 1: FPGA Test Data

sign_mag	twos_comp
0000	0000
0001	0001
0010	0010
0011	0011
0100	0100
0101	0101
0110	0110
0111	0111
1000	1000
1001	1111
1010	1110
1011	1101
1100	1100
1101	1011
1110	1010
1111	1001

The table above is the output which was expected. For non-negative numbers, the output is the same as the input. For negative inputs (i.e. most significant bit is '1'), the output is the two's complement. Thus, our synthesis and testing of the FPGA board were successful.

3.1.1 Implemented and Elaborated Designs

When using Vivado to implement the VHDL code, the implemented and elaborated design schematics can be obtained. These schematics can help us better understand what exactly was synthesized on the FPGA board. The two figures below show the implemented and elaborated designs, respectively.

Figure 3: Implemented Design Schematic

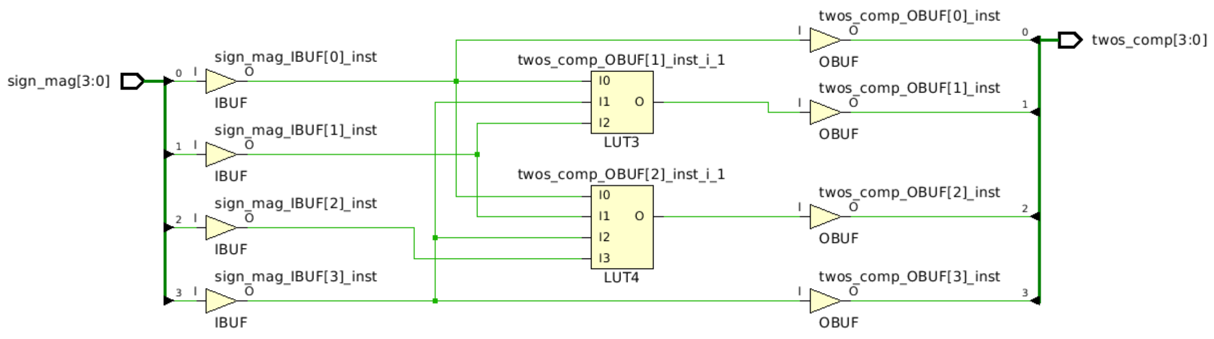
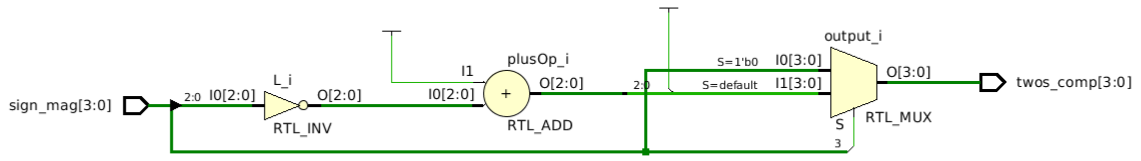


Figure 4: Elaborated Design Schematic



In figure 3, we see the implemented design. This schematic is a lower level physical design of our circuit. We see the input buffers on the left, and the output buffers on the right. These correspond the the input `sign_mag` and the output `twos_comp`. Finally, the two blocks in the center are the logic blocks, where decisions and boolean operations occur.

In figure 4, we see the elaborated design. This design is a bit higher level than the implemented design, as we can see the logic that is synthesized within our circuit. We see a mux that is controlled by the highest order bit, `sign_mag(3)`. If this input is 0, then the output is the original input (as seen on the top selection "I0[3:0]"). If the highest order bit is 0, then the inputs last three bits are inverted (as seen with the hex inverter), 1 is added to it, and a '1' is concatenated to the left, creating the two's complement. Overall, the implemented design helps us understand what is physically synthesized while the elaborated design helps us understand what is logically implemented.

3.1.2 Vivado Log Files

Every time Vivado is used to synthesize an FPGA board, a log file is created. In this log file, the various processes occurring as well as any errors/warnings are recorded. This is useful for a user, as the log file makes debugging easier. If an issue is encountered, one can refer to the log file to see where things went wrong, and which errors/warnings were thrown by Vivado. Since the log file is much too large to be included in one figure/page, snippets will be displayed and discussed below.

Figure 5: Log Snippet 1

```
#-----------------------------------------------------------
# Vivado v2018.2 (64-bit)
# SW Build 2258646 on Thu Jun 14 20:02:38 MDT 2018
# IP Build 2256618 on Thu Jun 14 22:10:49 MDT 2018
# Start of session at: Wed Mar  4 14:58:33 2020
# Process ID: 4541
# Current directory: /nfs/home/a/a_atter/COEN313/LAB3
# Command line: vivado
# Log file: /nfs/home/a/a_atter/COEN313/LAB3/vivado.log
# Journal file: /nfs/home/a/a_atter/COEN313/LAB3/vivado.jou
#-----------------------------------------------------------
start_gui
create_project Lab3 /nfs/home/a/a_atter/COEN313/LAB3/Lab3 -part xc7a100tcsg324-1
INFO: [IP_Flow 19-234] Refreshing IP repositories
INFO: [IP_Flow 19-1704] No user IP repositories specified
INFO: [IP_Flow 19-2313] Loaded Vivado IP repository '/CMC/tools/xilinx/Vivado_2018.2/Vivado/2018.2/data/ip'.
create_project: Time (s): cpu = 00:00:08 ; elapsed = 00:00:07 . Memory (MB): peak = 6263.906 ; gain = 11.012 ; free physical = 12539 ; free virtual = 24135
set_property target_language VHDL [current_project]
set_property simulator_language VHDL [current_project]
add_files -norecurse /nfs/home/a/a_atter/COEN313/LAB3/twos_complement.vhd
import_files -force -norecurse
INFO: [filemgmt 20-348] Importing the appropriate files for fileset: 'sources_1'
import_files -fileset constrs_1 -force -norecurse /nfs/home/a/a_atter/COEN313/LAB3/twos_complement.xdc
update_compile_order -fileset sources_1
set_property target_constrs_file /nfs/home/a/a_atter/COEN313/LAB3/Lab3/Lab3.srcs/constrs_1/imports/LAB3/twos_complement.xdc [current_fileset -constrset]
launch_runs synth_1 -jobs 2
[Wed Mar  4 15:01:46 2020] Launched synth_1...
Run output will be captured here: /nfs/home/a/a_atter/COEN313/LAB3/Lab3/Lab3.runs/synth_1/runme.log
launch_runs impl_1 -jobs 2
[Wed Mar  4 15:03:51 2020] Launched impl_1...
Run output will be captured here: /nfs/home/a/a_atter/COEN313/LAB3/Lab3/Lab3.runs/impl_1/runme.log
launch_runs impl_1 -to_step write_bitstream -jobs 2
[Wed Mar  4 15:05:43 2020] Launched impl_1...
Run output will be captured here: /nfs/home/a/a_atter/COEN313/LAB3/Lab3/Lab3.runs/impl_1/runme.log
open_hw
connect_hw_server
INFO: [Labtools 27-2285] Connecting to hw_server url TCP:localhost:3121
INFO: [Labtools 27-2222] Launching hw_server...
INFO: [Labtools 27-2221] Launch Output:
```

Figure 5 above is the start of the log file. Here, base parameters are established for our synthesis. Such parameters include target language, different file paths for the simulation, etc.

Figure 6: Log Snippet 2

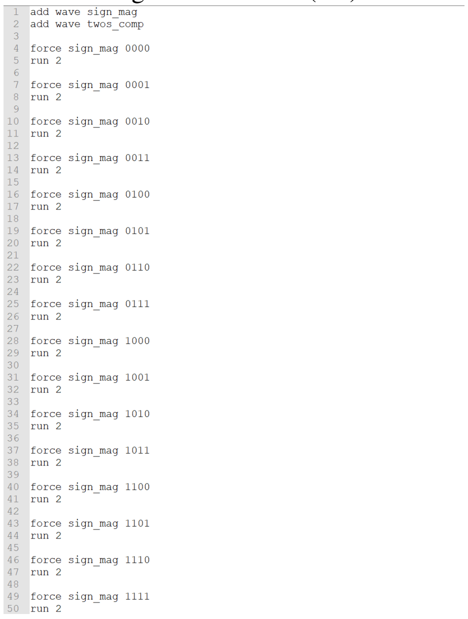
```
106 RTL Elaboration Complete:  : Time (s): cpu = 00:00:13 ; elapsed = 00:00:34 . Memory (MB): peak = 7419.090 ; gain = 498.145 ; free physical = 11436 ; free virtual = 23135
107 6 Infos, 0 Warnings, 0 Critical Warnings and 0 Errors encountered.
108 synth_design completed successfully
109 synth_design: Time (s): cpu = 00:00:13 ; elapsed = 00:00:34 . Memory (MB): peak = 7419.090 ; gain = 498.145 ; free physical = 11436 ; free virtual = 23135
110 close_design
111 open_run impl_1
112 INFO: [Netlist 29-17] Analyzing 4 Unisim elements for replacement
113 INFO: [Netlist 29-28] Unisim Transformation completed in 0 CPU seconds
114 INFO: [Project 1-479] Netlist was created with Vivado 2018.2
115 INFO: [Project 1-570] Preparing netlist for logic optimization
116 INFO: [Timing 38-478] Restoring timing data from binary archive.
117 INFO: [Timing 38-479] Binary timing data restore complete.
118 INFO: [Project 1-856] Restoring constraints from binary archive.
119 INFO: [Project 1-853] Binary constraint restore complete.
120 Reading XDEF placement.
121 Reading placer database...
122 Reading XDEF routing.
123 Read XDEF File: Time (s): cpu = 00:00:00.01 ; elapsed = 00:00:00.01 . Memory (MB): peak = 7735.238 ; gain = 0.000 ; free physical = 11047 ; free virtual = 22748
124 Restored from archive | CPU: 0.010000 secs | Memory: 0.023514 MB |
125 Finished XDEF File Restore: Time (s): cpu = 00:00:00.01 ; elapsed = 00:00:00.01 . Memory (MB): peak = 7735.238 ; gain = 0.000 ; free physical = 11047 ; free virtual = 2274
126 INFO: [Project 1-111] Unisim Transformation Summary:
127 No Unisim elements were transformed.
128
129 open_run: Time (s): cpu = 00:00:13 ; elapsed = 00:00:36 . Memory (MB): peak = 7890.941 ; gain = 471.844 ; free physical = 10989 ; free virtual = 22691
130 WARNING: [Timing 38-313] There are no user specified timing constraints. Timing constraints are needed for proper timing analysis.
131 exit
132 INFO: [Common 17-206] Exiting Vivado at Wed Mar  4 15:10:46 2020...
```

Figure 6 shows the end of the log file. At this point, the RTL elaboration, constraints, and other simulation/synthesis processes have completed. On line 130, there is a warning. The warning states that ”There are no user specified timing constraints. Timing constraints are needed for proper timing analysis.” This warning is telling us that we have not specified time constraints, and a timing analysis is thus not possible. This is because the VHDL code written uses a sensitivity list, rather than timing constraints.

3.2 Simulation

The code above in figure 1 was then simulated using Modelsim. For testing inputs, a DO file was written to force all the possible 4-bit inputs. Figure 7 shows the contents of the DO file that was used to simulate all possible inputs.

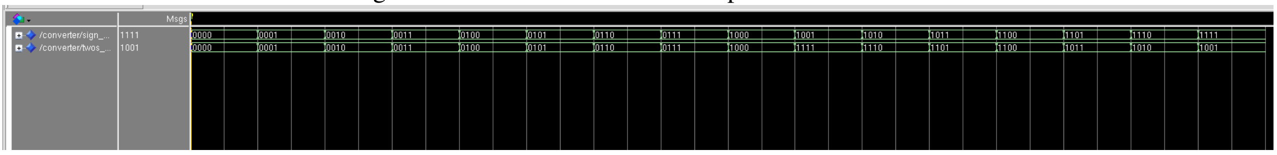
Figure 7: DO File (.do)



```
add wave sign_mag
add wave twos_comp

force sign_mag 0000
run 2

force sign_mag 0001
run 2

force sign_mag 0010
run 2

force sign_mag 0011
run 2

force sign_mag 0100
run 2

force sign_mag 0101
run 2

force sign_mag 0110
run 2

force sign_mag 0111
run 2

force sign_mag 1000
run 2

force sign_mag 1001
run 2

force sign_mag 1010
run 2

force sign_mag 1011
run 2

force sign_mag 1100
run 2

force sign_mag 1101
run 2

force sign_mag 1110
run 2

force sign_mag 1111
run 2
```

After running this file for the 2's complement VHDL code, the waveform output is as follows in figure 8.

Figure 8: Modelsim Waveform Output



```
                      Msgs
/converter/sign_...   1111   0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
/converter/twos_...   1001   0000 0001 0010 0011 0100 0101 0110 0111 1000 1111 1110 1101 1100 1011 1010 1001
```

Therefore, the simulation provides the same results as does the synthesis testing. Once again, this confirms the successful functionality of our code and of its implementation.

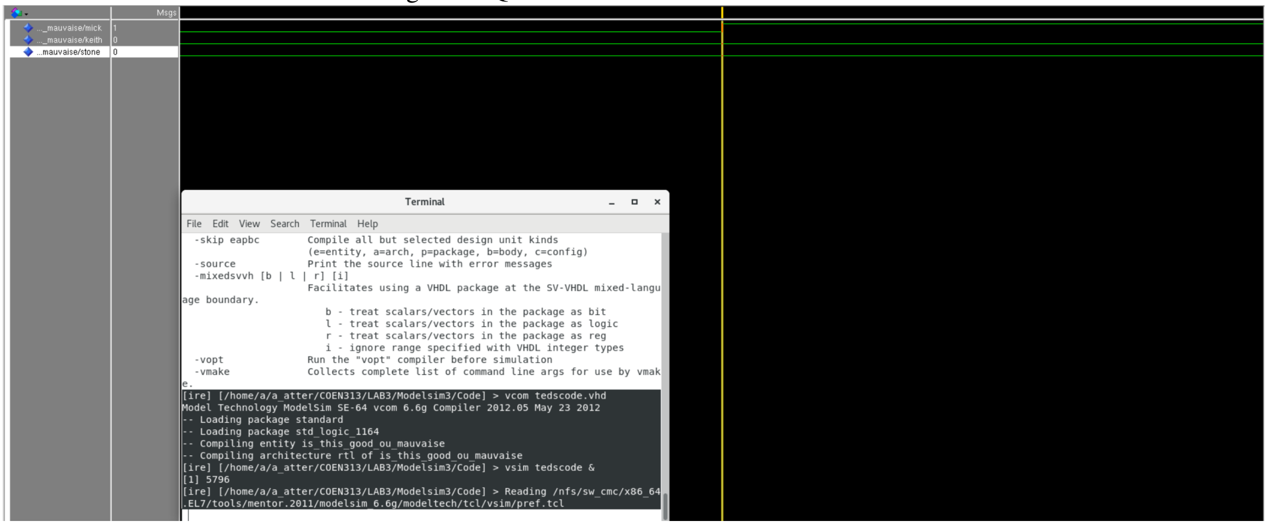
4 Questions

1. What will result (during synthesis) if a signal appears on both sides of the signal assignment operator (<=) within a combinational VHDL process such as:

```
library IEEE;
use IEEE.std_logic_1164.all;

entity is_this_good_ou_mauvaise is
        port( mick : in std_logic;
                keith : out std_logic);
end;

architecture rtl of is_this_good_ou_mauvaise is
signal stone : std_logic;
begin
        process(mick)
        begin
                stone <= mick and stone;
        end process ;
        keith <= stone ; -- tout le monde sais que Keith == stone
end ;
```

If a signal were to appear on both sides of a signal assignment operator, a latch would occur. This means that the value for "stone" would never change. Since stone is initialized as 0, the value for `mick and stone` would always return 0 for any value of mick. For example, the figure below shows the simulation output for forcing 0 then forcing 1 for mick.

Figure 9: Question 1 Simulation



```
  -skip eapbc         Compile all but selected design unit kinds
                      (e=entity, a=arch, p=package, b=body, c=config)
  -source             Print the source line with error messages
  -mixedsvvh [b | l | r] [i]
                      Facilitates using a VHDL package at the SV-VHDL mixed-langu
age boundary.
                         b - treat scalars/vectors in the package as bit
                         l - treat scalars/vectors in the package as logic
                         r - treat scalars/vectors in the package as reg
                         i - ignore range specified with VHDL integer types
  -vopt               Run the "vopt" compiler before simulation
  -vmake              Collects complete list of command line args for use by vmak
e.
[ire] [/home/a/a_atter/COEN313/LAB3/Modelsim3/Code] > vcom tedscode.vhd
Model Technology ModelSim SE-64 vcom 6.6g Compiler 2012.05 May 23 2012
-- Loading package standard
-- Loading package std_logic_1164
-- Compiling entity is_this_good_ou_mauvaise
-- Compiling architecture rtl of is_this_good_ou_mauvaise
[ire] [/home/a/a_atter/COEN313/LAB3/Modelsim3/Code] > vsim tedscode &
[1] 5796
[ire] [/home/a/a_atter/COEN313/LAB3/Modelsim3/Code] > Reading /nfs/sw_cmc/x86_64
.EL7/tools/mentor.2011/modelsim_6.6g/modeltech/tcl/vsim/pref.tcl
```

As one can see, the value for the signal `stone` and for the output `keith` remains 0, regardless of the input for mick.

2. What will happen during simulation if a signal is read from within a combinational process but does not appear in the process sensitivity list?

If the signal changes within the process, then the signal won't actually get the value. Since the signal wouldn't be in the process sensitivity list, it would not be passed back as a parameter, thus not updating its value.

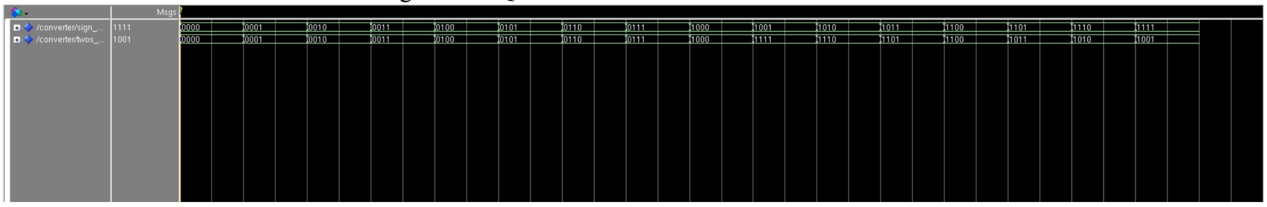
3. If you made use of variable in your combinational process, rewrite the VHDL code such that the process makes use of only signals. If you originally made use of only signals, rewrite your VHDL code such that it makes use of variable(s). Simulate your new VHDL code to show that it gives the same simulation results. You do not have to re-synthesize. Comment on the salient differences between the code which uses only signals and the code which makes use of a variable.

My code rewritten without using signals and only with variables is as follows:

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

entity converter is
        port(
                        sign_mag : in std_logic_vector(3 downto 0);
                        twos_comp : out std_logic_vector(3 downto 0));
end converter;

architecture converter_arch of converter is

begin
process(sign_mag)
        variable var: std_logic_vector(3 downto 0);
        begin
                if (sign_mag(3) = '0') then
                        var := sign_mag;
                else
                        var := '1' & (not(sign_mag(2 downto 0))+"001");
                end if;
        twos_comp <= var;
        end process;
end converter_arch;
```

This code was simulated on Modelsim using the exact same DO file as seen in figure 7. The model sim results for this code are as follow in the following figure:

Figure 10: Question 3 Simulation



When compared with figure 8, they are the exact same. Thus, using only signals or using no signals will produce the same output.

5 Conclusion

In conclusion, experiment 3 helped to further understand the implementation of VHDL code through synthesis and simulation. Specifically, we investigated the use of VHDL combinational processes to implement combinational logic. Additionally, we familiarized ourselves with the different command line options of vcom, and how we can use this to help verify our design. In this experiment, a VHDL code (.vhd) and a constraint file (.xdc) were first written to synthesize a sign-magnitude two's complement converter. This design was first tested physically on the Xilinx board, then tested virtually on Modelsim. For both tests, the VHDL code written performed as expected, thus the experiment was a success.