###### *CSE 260 – Digital Computers: Organization and Logical Design Jon Turner*

Lab 1 Solution

##### *1/28/2013*

***Part A*** (20 points). Paste the VHDL for your modified calculator below. Highlight your modifications to the code by making them **bold**. Note that the next paragraph is formatted using the ”code style” which uses a fixed-width font and has appropriately spaced tabs. Please always use this paragraph style for your VHDL code. Also, be sure to format your code so that lines do not wrap-around in the lab report. Points will be deducted for code that is badly formatted or difficult to read.

paste your code here

library IEEE;

use ieee.std\_logic\_1164.all;

use ieee.numeric\_std.all;

use ieee.std\_logic\_unsigned.all;

use work.commonDefs.all;

entity calculator is port (

clk: in std\_logic;

**clear, load, add, mode: in std\_logic; -- signals to enable operations**

dIn: in word; -- input data

result: out word;

**error: out std\_logic); -- output result**

end calculator;

architecture a1 of calculator is

signal dReg: word;

**signal dRegResult: word:=(wordsize-1 downto 0 => '0'); --gives an initial value to avoid error at the beginning;**

signal errormsg: std\_logic;

begin

dRegResult<=dReg+dIn;

process (clk) begin

if rising\_edge(clk) then

--sequential process block

if clear = '1' then

dReg <= (wordsize-1 downto 0 => '0');

**errormsg<='0'; --clear error value**

elsif load = '1' then

dReg <= dIn;

elsif add = '1' then

**dReg <= dReg + dIn;**

**if mode = '0' then**

**if (dRegResult< dReg) or (dRegResult< dIn) then**

**errormsg<='1';**

**end if;**

**elsif mode = '1' then**

**if (dRegResult(wordsize-1) /= dReg(wordsize-1)) or (dRegResult(wordsize-1) /= dIn(wordsize-1)) then**

**errormsg<='1';**

**end if;**

**end if;**

end if;

end if;

end process;

result <= dReg;

error <= errormsg;

end a1;

***Part B.*** (10 points) Draw a block diagram of the calculator including your modifications to it. Include a block labeled *ErrorDetector* that represents the logic that determines if an error has occurred. Your diagram need not show how this block is implemented, but it should show all the signals that connect to it. Your diagram should also include a flip flop top hold the error bit.

It’s ok to submit a hand-drawn figure on the printed copy only, but please, make it neat, well-organized and legible.

*paste (or draw) your diagram here*

***Part C.*** (20 points). Modify the *testCalculator* testbench to include additional tests to verify the error detection feature added to the calculator. You must verify that it detects errors for both values of the *mode* input, that error bit is cleared when a *clear* operation is performed and that other operations are blocked when the error bit is set. Highlight the tests you added by making them **bold**.

paste your code here

LIBRARY ieee;

USE ieee.std\_logic\_1164.ALL;

USE ieee.numeric\_std.ALL;

USE ieee.std\_logic\_unsigned.all;

use work.commonDefs.all;

entity testCalculator is

end testCalculator;

architecture a1 of testCalculator is

component calculator port(

clk : in std\_logic;

**clear, load, add, mode : in std\_logic;**

dIn : in word;

result : out word);

end component;

signal clk : std\_logic := '0';

signal clear : std\_logic := '0';

signal load : std\_logic := '0';

signal add : std\_logic := '0';

**signal mode: std\_logic :='0'; --added mode signal**

signal dIn : word := (others=>'0');

signal result : word;

begin

-- create instance of calculator circuit

uut: calculator port map(

clk => clk, clear => clear, load => load, **mode=>mode, --added mode signal**

add => add, dIn => dIn, result => result

);

process begin -- clock process for clk

clk\_loop : loop

clk <= '0'; wait for 10 ns;

clk <= '1'; wait for 10 ns;

end loop clk\_loop;

end process;

**tb : process begin -- test inputs, mode=0 by default**

clear <= '1'; load <= '1'; add <= '1'; dIn <= x"ffff"; wait for 20 ns;

clear <= '0'; load <= '1'; add <= '0'; dIn <= x"ffff"; wait for 20 ns;

clear <= '0'; load <= '1'; add <= '1'; dIn <= x"ffff"; wait for 20 ns;

clear <= '0'; load <= '0'; add <= '1'; dIn <= x"0001"; wait for 20 ns;

clear <= '0'; load <= '0'; add <= '1'; dIn <= x"0002"; wait for 20 ns;

clear <= '0'; load <= '0'; add <= '1'; dIn <= x"0003"; wait for 20 ns;

clear <= '0'; load <= '0'; add <= '1'; dIn <= x"0100"; wait for 20 ns;

clear <= '0'; load <= '0'; add <= '1'; dIn <= x"0200"; wait for 20 ns;

clear <= '0'; load <= '0'; add <= '1'; dIn <= x"0300"; wait for 20 ns;

**--added tests below, mode 0**

**clear <= '1'; load <= '0'; add <= '1'; mode<='0'; dIn <= x"0000"; wait for 20 ns;**

**clear <= '0'; load <= '1'; add <= '1'; mode<='0'; dIn <= x"ffff"; wait for 20 ns;**

**clear <= '0'; load <= '0'; add <= '1'; mode<='0'; dIn <= x"0001"; wait for 20 ns;**

**clear <= '0'; load <= '0'; add <= '1'; mode<='0'; dIn <= x"0002"; wait for 20 ns;**

**clear <= '1'; load <= '0'; add <= '1'; mode<='0'; dIn <= x"0000"; wait for 20 ns;**

**--addde test for mode 1**

**clear <= '1'; load <= '0'; add <= '1'; mode<='1'; dIn <= x"0000"; wait for 20 ns;**

**clear <= '0'; load <= '1'; add <= '1'; mode<='1'; dIn <= x"7fff"; wait for 20 ns; --x(7fff)=2^16-1, which is the biggerst number represntable by 16 bit signed.**

**clear <= '0'; load <= '0'; add <= '1'; mode<='1'; dIn <= x"0001"; wait for 20 ns;**

**clear <= '0'; load <= '0'; add <= '1'; mode<='1'; dIn <= x"0002"; wait for 20 ns;**

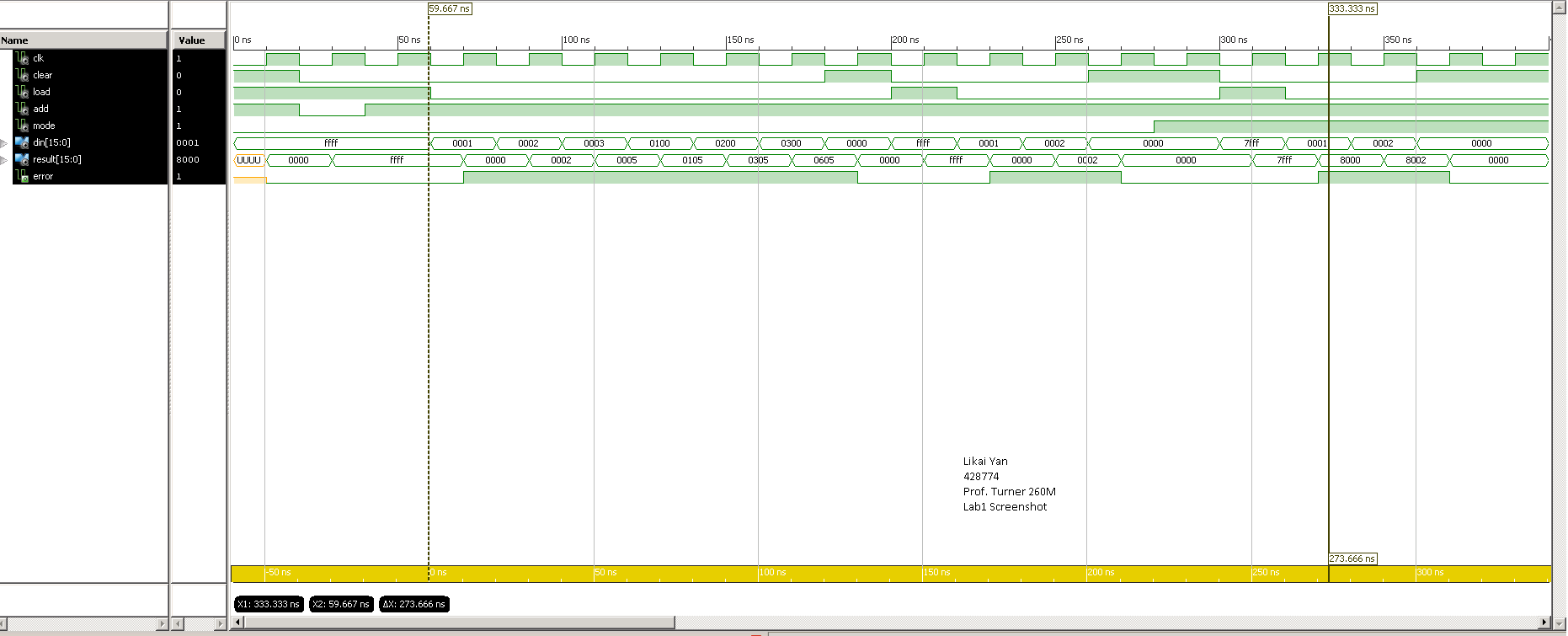
**clear <= '1'; load <= '0'; add <= '1'; mode<='1'; dIn <= x"0000"; wait for 20 ns;**

wait for 20 ns;

assert (false) report "Simulation ended normally." severity failure;

end process;

end a1;

Run the simulator using the *testCalculator* testbench and paste a screenshot of the waveform window below. Set the radix to hexadecimal for the *dIn* and *result* signals. Note that by default, the simulator displays signals in white on a black background. This works poorly when printed. You can change this using the Preferences item in the Edit menu of the waveform window. Please make the waveform area white and make the signal colors and all text black. You should only have to do this one time. After that, your preferences will be remembered. If your screenshot extends over a longer time period than 400 ns, split it into two screenshots and paste them one after the other. That is, first paste a screenshot that covers the first 400 ns of your simulation run, then paste a second screenshot for the next 400 ns. Make sure that all text is clearly legible on the printed copy. If not reduce the time period covered by each screenshot until it is legible.

*paste your screenshot here*

**see left**

Find the place in your simulation where *mode*=0 and *error* goes high for the first time. What are the values of the two operands at this moment? Explain why these input values should trigger an arithmetic error.

**Place:230.00ns**

**Values of operands: x(ffff) and x(0001), or 65535 and 1 in unsigned decimal.**

**They should trigger arithmetic error because the result of adding x(ffff) and x(0001) is x(0000), which is essentially a modular addition. Such addition behave exactly like normal addition unless the result is too big to represent by four hex digits, which is the case for our error test as 4 hex digits can’t represent unsigned 65536 in decimal.**

Find the place in your simulation where *mode*=1 and *error* goes high for the first time. What are the values of the two operands at this moment? Explain why these input values should trigger an arithmetic error.

**Place: 330.00ns**

**Values of operands: x(7fff) and x(0001), or 32767 and 1 in signed decimal.**

**They should trigger arithmetic error because adding x(7fff) and x(0001) results in x(8000), which, in 2s complement representation’s perspective, is equivalent to 32767+1=-32768. However, that is not what we expect in normal addition, because the result should be bigger than either of the operand with the same sign. The structure of positive and negative numbers’ organization determined that we need to set such case as an error.**

***Part D.*** (10 points) Paste a copy of the modified version of top below.

paste your code here

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.STD\_LOGIC\_ARITH.ALL;

use IEEE.STD\_LOGIC\_UNSIGNED.ALL;

use work.commonDefs.all;

entity top is port(

clk: in std\_logic;

-- S3 board buttons, knob, switches and LEDs

btn: in buttons;

knob: in knobSigs;

swt: in switches;

led: out leds;

-- signals for controlling LCD display

lcd: out lcdSigs);

end top;

architecture a1 of top is

component calculator port (

clk: in std\_logic;

clear, load, add, mode: in std\_logic;

din : in word;

result: out word;

**error: out std\_logic); --add error output**

end component;

component binaryInMod port(

clk: in std\_logic;

btn: in buttons;

knob: in knobSigs;

resetOut: out std\_logic;

dBtn: out std\_logic\_vector(3 downto 1);

pulse: out std\_logic\_vector(3 downto 1);

inBits: out word);

end component;

**component binaryOutMod port(**

**clk, reset, error: in std\_logic;**

**topRow, botRow: in word;**

**lcd: out lcdSigs);**

end component binaryOutMod;

**signal reset, clear, load, add, mode, error: std\_logic;**

signal dBtn, pulse: std\_logic\_vector(3 downto 1);

signal inBits, outBits: word;

begin

-- connect the sub-components

**imod: binaryInMod port map(clk,btn,knob,reset,dBtn,pulse,inBits);**

**calc: calculator port map(clk,clear,load,add,mode,inBits,outBits,error);**

**omod: binaryOutMod port map(clk,reset,error,inBits,outBits,lcd);**

-- define internal control signals

clear <= dBtn(1) or reset;

load <= pulse(2);

add <= pulse(3);

**mode<=swt(0);**

-- connect a few input and output bits to leds

**led(7 downto 2) <= ("000000");**

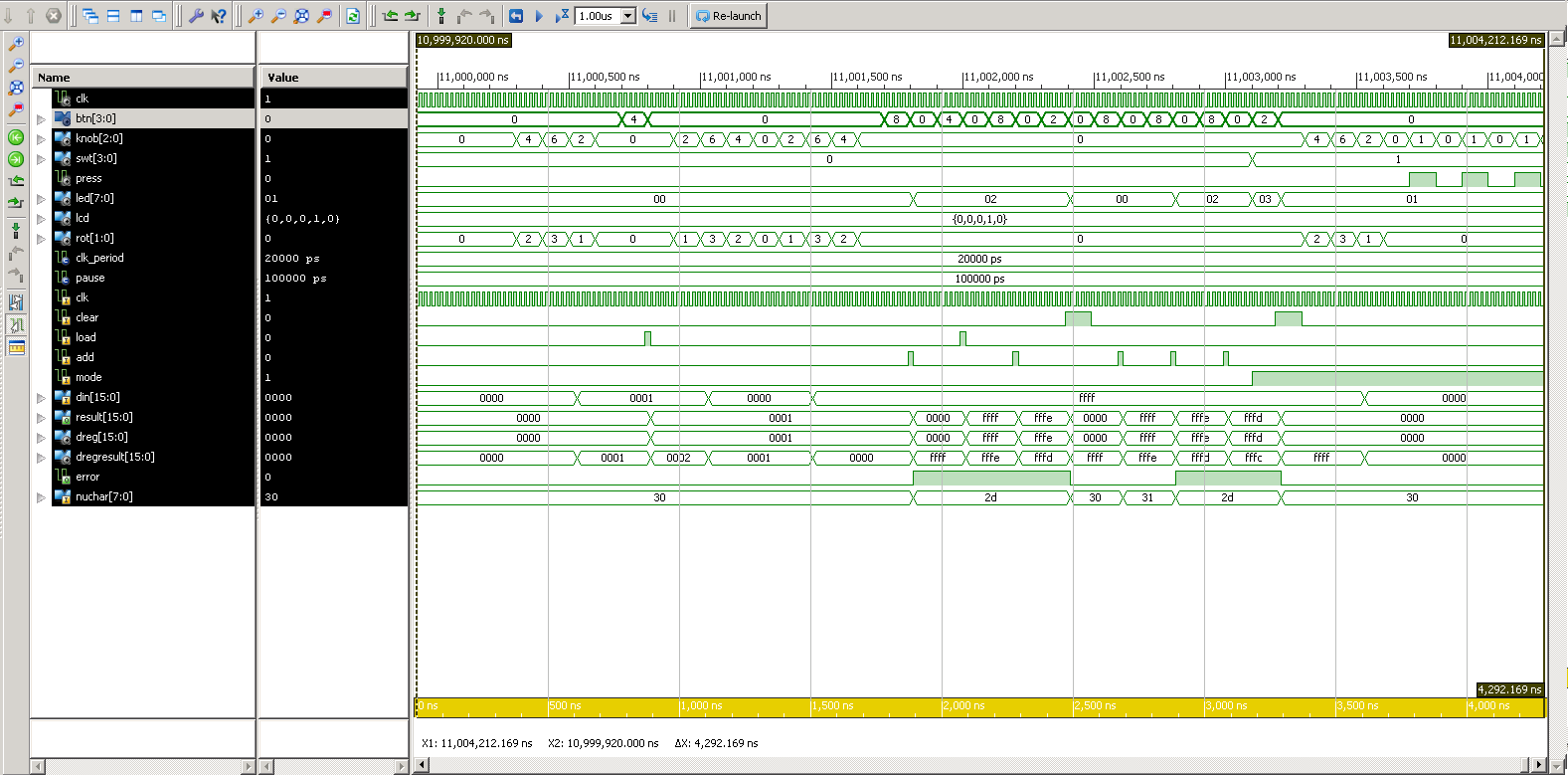
**led(0)<=mode;**

**led(1)<=error;**

end a1;

***Part E***. (10 points) Draw a block diagram of your top level circuit below. It should include the input module, output module and calculator and all signals connecting them. Label all signals with the names used by the top module.

*paste (or draw) your diagram here*

*****Part F***. (15 points) Run the simulator using the provided *testTop* testbench. Do not modify the testbench. When you run the simulation, you will notice that all the interesting stuff happens near the very end of the simulation. Focus on this part when you are checking to make sure your circuit works correctly. Add signals to the waveform display to help you verify that the circuit works correctly. Specifically, be sure to include all of the calculator’s internal signals and the *nuChar* signal in the output module. Organize the signals so that related signals are grouped together and use dividers to label different groups of signals. Paste a screenshot of your simulation output showing the time period from 11,001 microseconds to 11,003 microseconds. Make sure that all text is clearly legible on the printed output.

*paste your screenshot here*

**See left.**

What are the operand values for the first addition operation in this time period? What is the result of this addition? Is it correct? Is an error detected at this point?

**Operand: x(ffff) from dIn, x(0001) from dReg. Or 65535, 1, in unsigned decimal respectively.**

**Result: x(0000), or 0 in decimal.**

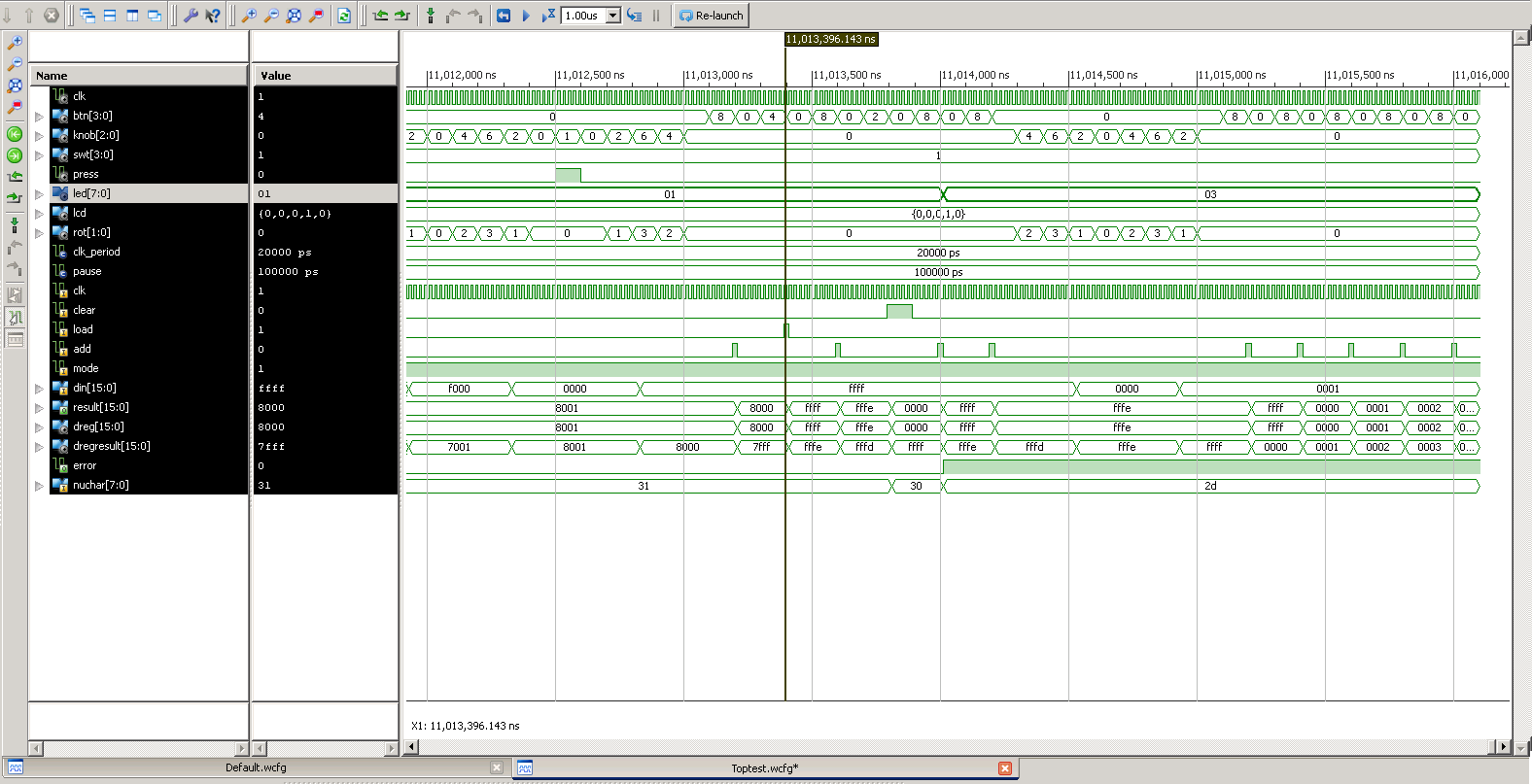
**It is not correct because we should expect 65536, not 0. Error detected with error signal high and nuchar set as -.**

What is the value of *nuChar* at this point? Is it what you expect? Are the LEDs correct?

**The value of nuChar is set as x(2d), which is dash in ASCII. LEDs are also showing the error has occurred by showing x(02), which means lighting up led(1), the error indicator.**

Paste another screenshot below, this time covering the time period from 11,012 microseconds to 11,015 microseconds.

*paste your screenshot here*



What are the operands for the first addition operation in this time range? What result is produced? Is an error detected at this point?

**Operands: x(ffff) from dIn and x(8001) from dReg. In signed decimal, they are -1 and -32767, respectively.**

**Result: x(8000), or -32768 in signed decimal.**

**No error detected at this point. This addition is consistent with our normal addition because we will get -32768 by adding (-1) and (-32767).**