# EEL 3701 – Digital Logic and Computer Systems Post-Lab 5

# Greg Bolling

# UF ID 64911960

## Problem Statement

The goal of the lab is to design, develop, and implement using VHDL/Quartus, a small GCPU that is a four bit ALU with a controller. The controller is a Moore state machine.

## Design

The design of the ALU is simple and is from the template provided for the class. The data path is also started from the class and expanded to included the muxes and an extra output register at the output of the ALU before going to the Data Out.

The Bonus was done by connecting:

Instruction bits 2,1,0 to Inputs from Switch 1,2,3

Destination bits 1,0 to Inputs from Switch 4,5

Bus\_IN bits 2,1,0 to Inputs from Switch 6,7,8

I didn’t have enough switches to connect all the bits so I connected Bus\_IN bit 3 to a ‘0’ inside the VHDL.

Outputs from the BUS\_OUT bits 3,2,1,0 were connected to LED 1,2,3,4

Start is connected to LED 5

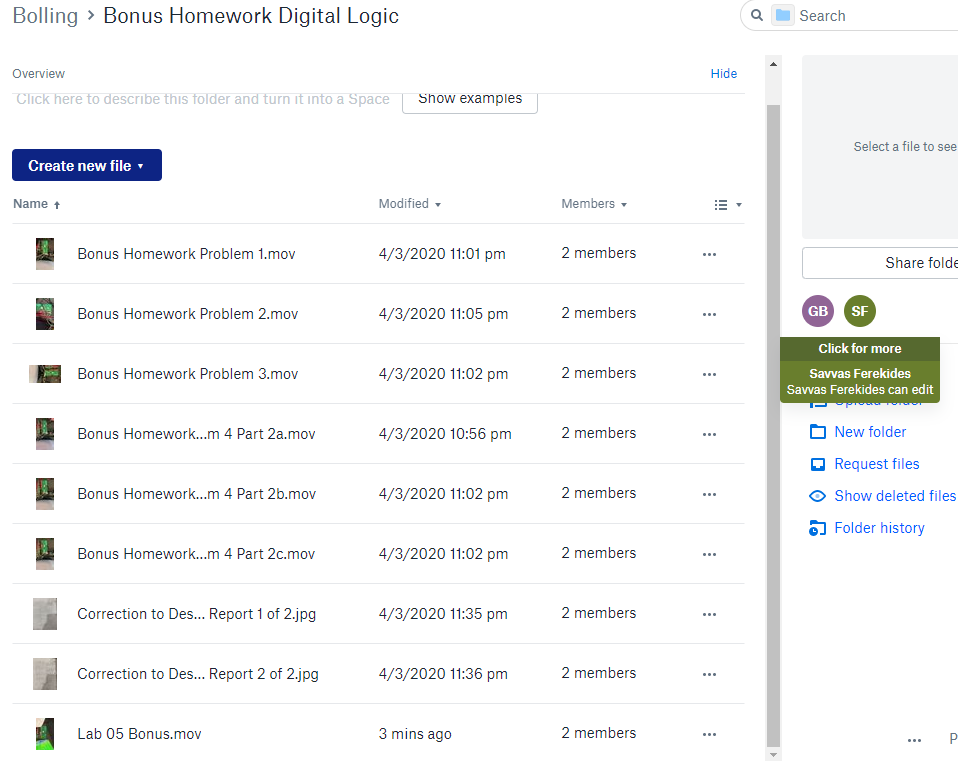
Done is connected to LED 6

Push Button 1 is for Reset

Push Button 2 is for Start.

Also, I used the slow clock.

The bonus video is uploaded to my DropBox under the same folder for the bonus homework and this video is named Lab 05 Bonus.mov.



## Simulation Annotations

Part 1 (ALU)

A separate Testbench was created and this was simulated prior to the entire GCPU being built. A description of that simulation is here. This testbench is called “tb\_reg\_alu”.

A test bench is used in order to simulate and test each individual design of the project with sample inputs in order to determine sample outputs and to see if the individual design was successful. The alu contains add, and, or, subtract operations. The five signals being tested for in the alu testbench are ‘A,B,F,CIN,COUT’. These signals are displayed in green in the screenshots shown below. Each green signal correlates with it’s respective signal name in the grey column to the left of the screenshot. The signal display contains increments of 50 nanoseconds between each vertical grey line located on the black display. The ‘msgs’ numbers indicate the binary results on the display. If the number is not indicated, it is assumed that a high on a signal represents a value of ‘1’, and a low on a signal represents a value of ‘0’.

Signal A contains 4-bit values, each representing a specific sample value operand for A. Each distinctive value lasts for 400 nanoseconds and then changes to the next sample value operand for A. The values given for each part are ‘1001’, ‘0001’, ‘0001’, and ‘1000’ respectively.

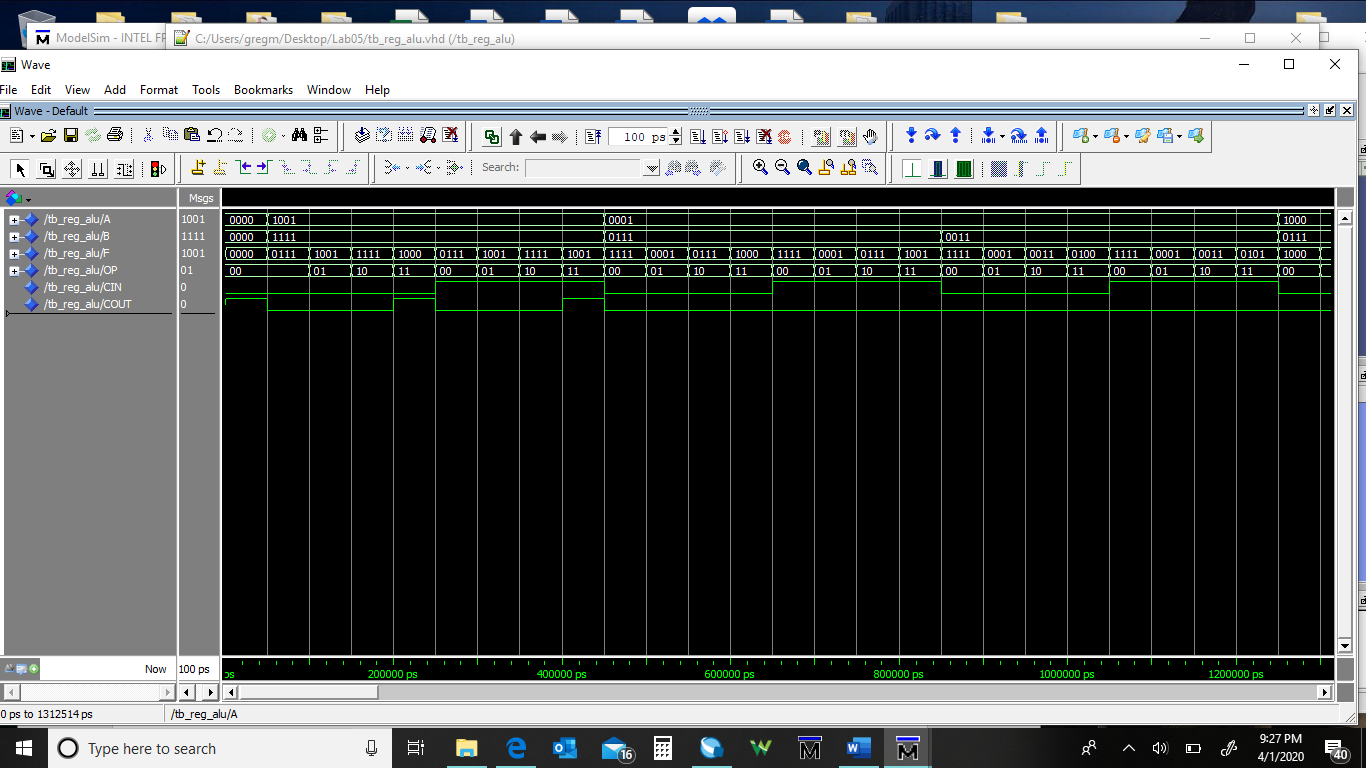
Signal B contains 4-bit values, each representing a specific sample value operand for B. Each distinctive value lasts for 400 nanoseconds and then changes to the next sample value operand for B. The values given for each part are ‘1111’, ‘0111’, ‘0011’, and ‘0111’ respectively.

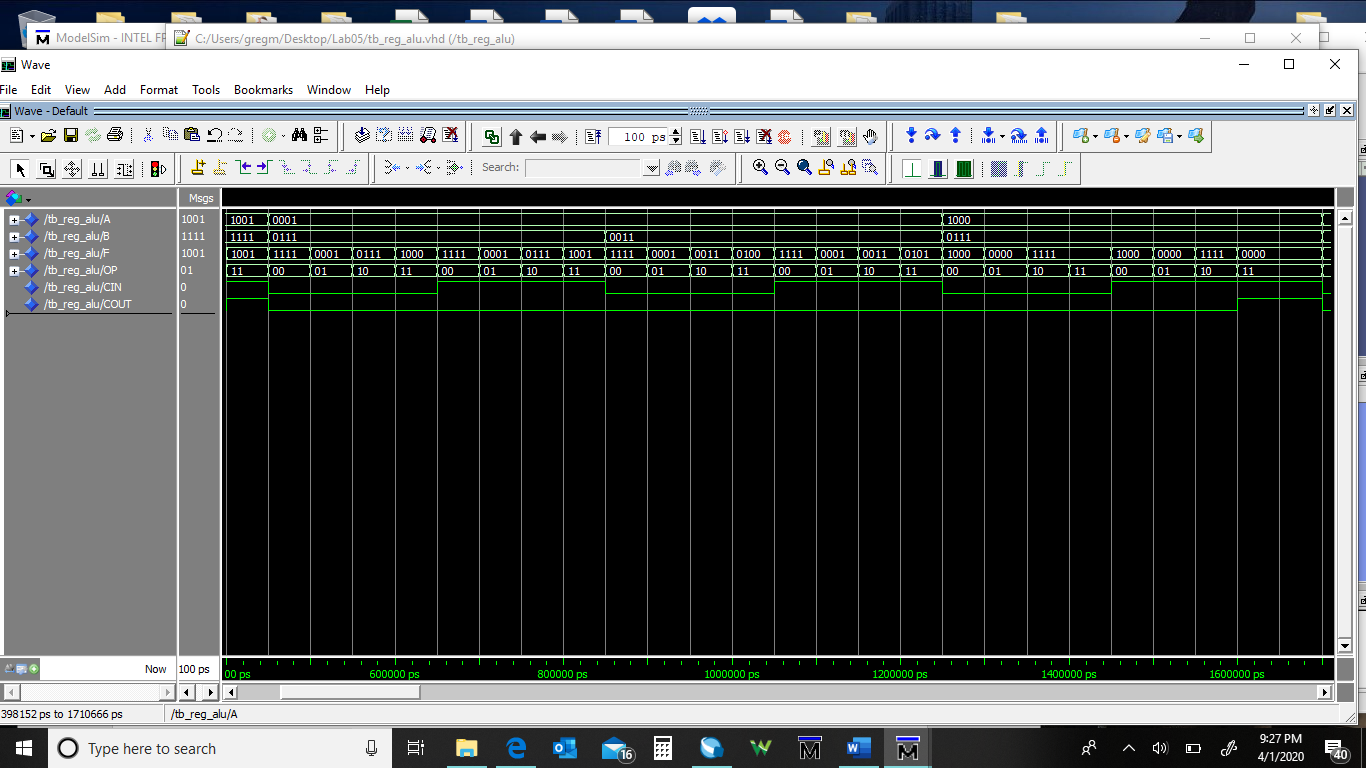
Signal OP contains 2-bit values, each representing a specific operation for values ‘A’ and ‘B’. For each operation, ‘00’ signifies 2’s Complement of A, ‘01’ signifies logical AND for A and B, ‘10’ signifies logical OR for A and B, and ‘11’ signifies addition between A and B (and is dependent upon CIN or carry in). OP increments every 50 nanoseconds in the display, switching between ‘00’, ‘01’, ‘10’, and ‘11’ respectively.

Signal CIN contains a 1-bit value and allows the user to know when a Carry in will be established for the case of addition between A and B (OP = 11). CIN starts with a carry in of ‘0’ and then oscillates between values of ‘1’ and ‘0’ every 200 nanoseconds on the display.

Signal F contains the outputs of the operations inside the alu based upon signals A,B,OP, and CIN. The outputs of signal F are 4-bits in length, starting at 0 nanoseconds and updating every 50 nanoseconds.

The display begins with all signals being initialized with values of zero for the first 50 nanoseconds of the display. The next 300 nanoseconds we can refer to as ‘test sample 1’ and contain operands A = ‘1001’ and B= ‘1111’. The first 50 nanoseconds of sample 1 utilize 2’s compliment of A because OP=’00’. This operation causes F to output ‘0111’ because the 2’s compliment of A (‘1001’) is ‘0111’. The next 50 nanoseconds produces OP = ‘01’, which is the AND operation for A(1001) and B(1111). The output for this is F = 1001. The next 50 nanoseconds is the OR operation for A(1001) and B(1111). The output for this is F = 1111. The next 50 nanoseconds is the ADD operation for A(1001) and B(1111). Notice the CIN on this display is zero at the moment, therefore there is no carry in for this addition. A(1001) + B(1111) produces a value of F = 1000 with a carry out of 1, and COUT = 1 as shown on the display. The next 200 nanoseconds repeat with a carry in of 1 (CIN = 1) instead of zero meant for the ADD operation. Which is why F= 1001 when A(1001) + B(1111) occurs. The carry out is also a ‘1’ (COUT = 1) at this time. The cycle repeats, with different values for the operands A and B every 400 nanoseconds afterwards.





PART 2 (DataPath)

A separate Testbench was created and this was simulated prior to the entire GCPU being built. A description of that simulation is here.

Similar to that of part 1, the testbench for the Datapath contains a grey column on the left side of the display so as to distinguish the different signals that are presented in the plot. The green horizontal lines stemming from the grey column indicate the corresponding signals. The numbers represented on these signals also range from 1-bit to four bits like part one, only this time some of the signals are highlighted in red and contain U’s in substitution for the bit number representations of 1’s and 0’s. The U’s represent don’t cares, and therefore we do not have to worry about any of the results coming from the red signals, and only have to pay our attention to the green signals. The different signals for the data path are CLK (the clock that runs the datapath), reset (the signal that sets all values to zero), BUS\_IN (the input for the bus), CIN (carry in), SEL1 and SEL2 ( the input for MUX 1 and MUX 2 respectively), LDA and LDB (load A and load B allow us to know which register to load), OP (operation for the alu), COUT (carry out), and BUS\_OUT (the output bus). The backround of the display is black so that the vertical grey lines can be distinguished. These are separated a distance of 50 nanoseconds from each other like before.

The CLK signal is a 1-bit value and starts at ‘0’ for the first 50 nanoseconds, it then oscillates back and forth between ‘1’ and ‘0’ every 50 nanoseconds preceding the first value. The signal on the top of the screen shows the green line moving from low to high or high to low each 50 nanoseconds of time to represent this circumstance.

The reset signal is a 1-bit value and starts at ‘1’ (or ON) for the first 200 nanoseconds, until it switches back off for the rest of the simulation. The second green line from the top depicts a green line in the high position for 200 nanoseconds, and then switches to low for the remainder of the simulation. The reset signal prevents any other signals from displaying output with the exception of the clock.

The BUS\_IN signal is a 4-bit input for the data path and becomes initialized with the variable ‘0011’. Starting at 220 nanoseconds (the extra 20 nanoseconds is used in order to accommodate propagation delay for signals), the input bus holds this value for 100 nanoseconds until it switches to the value 0100 for another hundred nanoseconds. Finally, the input bus reverts to ‘0000’ value until the stimulation finishes. These values are sample values for operands as we will see later on. The reason for the ‘0011’ at the end of the green line for bus input is because the process repeats after the simulation has reached its full length of time.

The CIN signal is off, starting from 220 nanoseconds and continuing until the end of the simulation. There is therefore no carry in represented on the green signal.

The SEL1 signal is a 2-bit value that starts off with the value ‘01’ and proceeds for 200 nanoseconds until its value reverts back to ‘00’ as represented on the fifth green signal from the top on the display.

The SEL2 signal is the same as the SEL1 signal with a 2-bit value that starts off with a value of ‘01’ and proceeds for 200 nanoseconds until its value reverts to ‘00’ as represented on the sixth green signal from the top on the display.

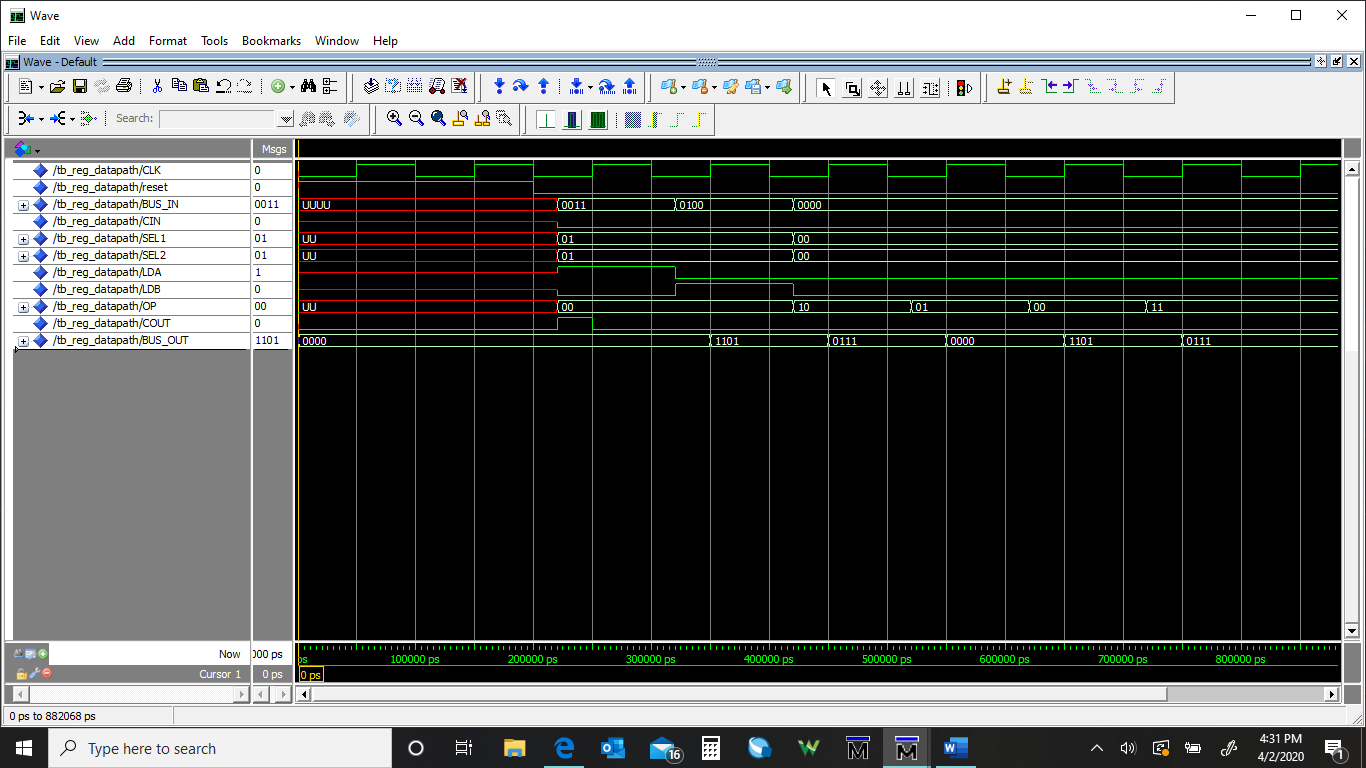
The LDA signal is a 1-bit value that loads in the register A. This is set to a value of ‘1’ for the first 100 nanoseconds that it is initialized until its finished loading in the ‘0011’ value coming in from the input bus. The LDA signal is then ‘0’ (and the green line is low) for the remainder of the simulation.

The LDB signal is also a 1-bit value that loads in register B. The LDB signal is off for the first 100 nanoseconds of the simulation, and then on (the green line is high) for the next 100 nanoseconds in order to load the value ‘0100’ in from the bus input. It is then reverted to the off position for the remainder of the simulation.

The OP signal is a 2 bit operation signal that completes specific operations for the alu. The operand starts at 00 when it is first initialized. This performs 2’s compliment on the A value. It then changes to the value ‘10’ after the first 200 nanoseconds of being initialized. This operation is logical OR, and lasts for 100 nanoseconds until the value is changed to ‘10’ (the logical AND operation). After another 100 nanoseconds, the value changes to ’00’ (2’s compliment of A) and finally to ‘11’ for the remainder of time (ADD operation).

The COUT signal is 1-bit and contains the value of 1 after the first 220 nanoseconds. It then changes to ‘0’ (or low on the green line) after 30 nanoseconds, and lasts for the remainder of the signal (green line).

The BUS\_OUT signal is 4-bits in length and has its first value of ‘1101’ at 350 nanoseconds. This value occurs because The LDA function loads the value of ‘0011’ into register A, and then performs 2’s compliment on A due to the ‘00’ OP operation. After 100 nanoseconds, the next value is ‘0111’. This is because both LDA and LDB have now loaded in ‘0011’ and ‘0100’ values respectively. The operation OP ‘10’ is performed as a logical OR for these two values, and the next rising clock edge causes the result to display. ‘0000’ is the next output as the OP ‘01’ logically AND’s the two values on the next rising clock edge (which occurs in the next 100 nanoseconds). ‘1101’ value then occurs because of OP ‘00’ performing 2’s compliment of A on the next rising clock edge. Finally ‘0111’ occurs and lasts as the remaining value until the end of the simulation because OP ‘11’ is performed as addition between A and B. There are no carry in function or carry out function occurring, so the addition lies within the 4-bit range.



PART 3 (Controller)

This is the simulation of the entire design with everything connected. This simulation takes in the instruction and the destination operation and perform the operation and responds back with a Done signal.

The Clock signal runs from high to low and from low to high once every 50 nanoseconds. The clock signal is displayed on the topmost signal.

The signal reset is the next signal displayed, which shows that reset is set to high for 200 nanoseconds, then switches back off for the rest of the simulation.

The instruction signal is displayed as the third signal from the top of the display, as a 3-bit input signal representing inputs for the states of the controller. The first signal displayed is 000, which loads the registers (register A and B) with input values coming from the input bus. After 600 nanoseconds, the signal changes every 300 nanoseconds to the different values of ‘100’ (OR), ‘011’(And),’101’(Add),’110’(Subtract),’001’(Copy A),’000’(Load register), and finally changes to ‘010’(Copy B).

The destination signal is displayed as the fourth signal from the top of the display. This is represented as a 2-bit input containing values for choosing between register A or register B. The signal begins as ‘00’ so that Register A can be loaded with input values. The signal then changes to ‘01’ after 300 nanoseconds in order to load in Register B with a value. Register A is then loaded again as a default value for ORing, ANDing, ADDing and SUBtracting. The value then changes to ‘01’ in order to Copy A to B, and to load register B with a different value.

The start signal starts as high and lasts for one clock cycle until and then goes back down until the next instruction.

BUS\_IN is the next signal below the start signal. This signal is a 4-bit signal that produces values for the operands. The first value produced is 0011, which is loaded into register A. Then 1010 is loaded into register B (lasting 300 nanoseconds each). The simulate proceeds to perform the operations as follows:

-- Load Register A with 0011

-- Load Register B with 1010

-- Compute A OR B and store in A

-- Restore A Load Register A with 0011

-- Compute A AND B and store in B

-- Restore B Load Register B with 1010

-- Compute A + B and Store in A

-- Restore A Load Register A with 0011

-- Compute A - B and Store in A

-- Restore A Load Register A with 0011

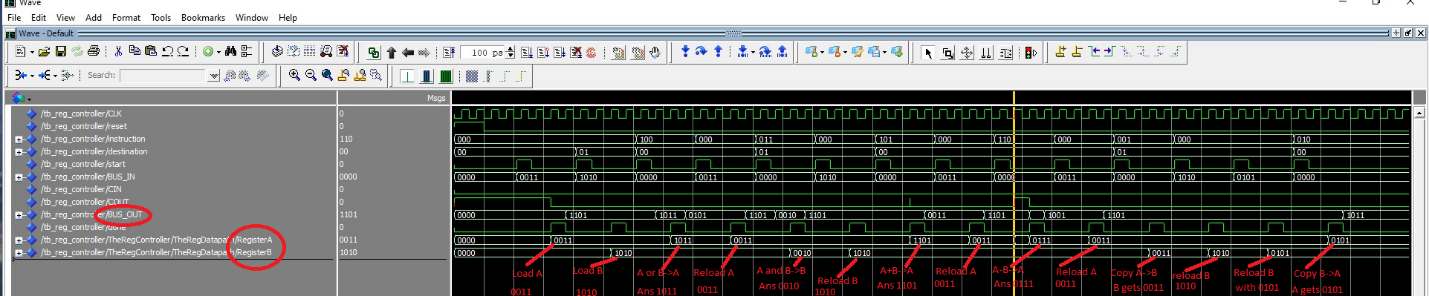
-- Copy A to B

-- Restore B Load Register B with 1010

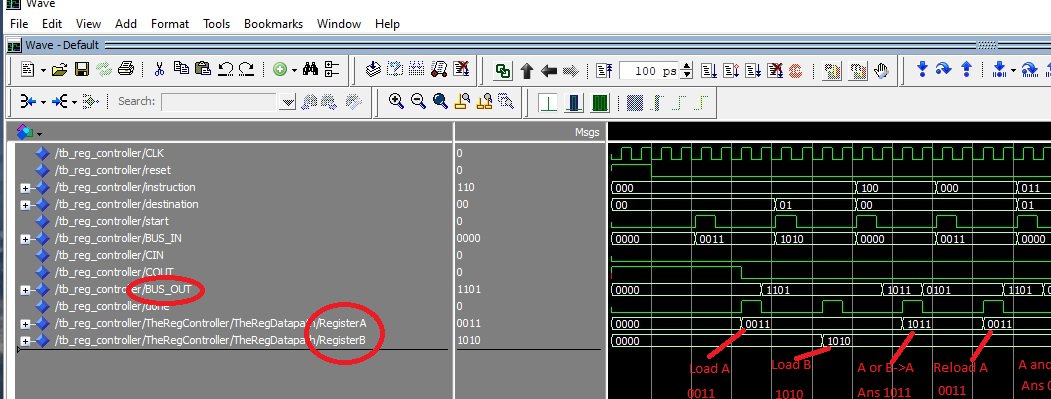
-- Load Register B with 0101

-- Copy B to A

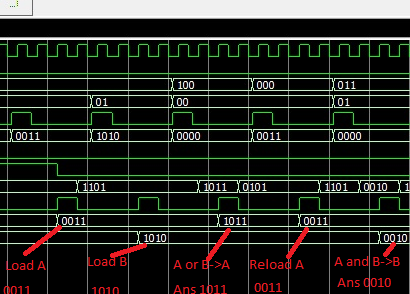
The total simulation is shown below but it is kind of hard to read so there are copies of parts of this after this image to see it better in this file.



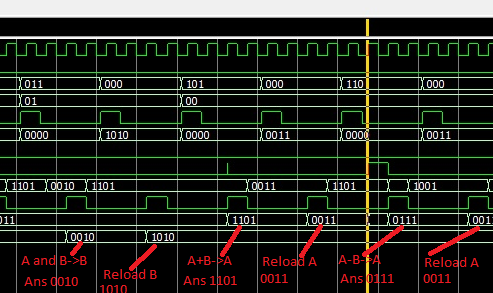
This is the first part of the simulation with annotations.



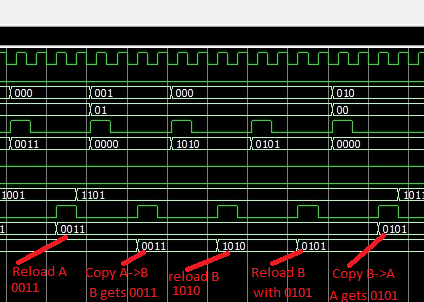
This is the second part of the simulation with annotations.



This is the third part of the simulation with annotations.



This is the fourth part of the simulation with annotations.



Appendix

The Testbench that tested the GCPU is as follows:

LIBRARY ieee;

USE ieee.std\_logic\_1164.ALL;

use IEEE.std\_logic\_arith.all;

use IEEE.STD\_LOGIC\_UNSIGNED.ALL;

ENTITY tb\_reg\_controller IS

END tb\_reg\_controller;

ARCHITECTURE behavior OF tb\_reg\_controller IS

-- Component Declaration for the Unit Under Test (UUT)

COMPONENT GCPU\_REG\_CONTROL

port (

CLK : in std\_logic;

reset : in std\_logic;

instruction : in std\_logic\_vector(2 downto 0);

destination : in std\_logic\_vector(1 downto 0);

start : in std\_logic;

BUS\_IN : in std\_logic\_vector(3 downto 0);

CIN : in std\_logic; -- default to '0'

COUT : out std\_logic;

BUS\_OUT : out std\_logic\_vector(3 downto 0);

done : out std\_logic

);

END COMPONENT;

--Inputs

Signal CLK : std\_logic;

Signal reset : std\_logic;

Signal instruction : std\_logic\_vector(2 downto 0);

Signal destination : std\_logic\_vector(1 downto 0);

Signal start : std\_logic;

Signal BUS\_IN : std\_logic\_vector(3 downto 0);

Signal CIN : std\_logic;

Signal COUT : std\_logic;

Signal BUS\_OUT : std\_logic\_vector(3 downto 0);

Signal done : std\_logic;

BEGIN

TheRegController : GCPU\_REG\_CONTROL

port map (

CLK => CLK ,

reset => reset ,

instruction => instruction,

destination => destination,

start => start ,

BUS\_IN => BUS\_IN ,

CIN => CIN ,

COUT => COUT ,

BUS\_OUT => BUS\_OUT ,

done => done

);

TheClock: process

begin

CLK <= '0';

wait for 50 ns;

CLK <= '1';

wait for 50 ns;

end process;

reset <= '1', '0' after 200 ns;

-- Stimulus Generation process

Stimuli\_Gen: process

begin

start <= '0';

instruction <= "000";

destination <= "00";

BUS\_IN <= "0000";

CIN <= '0';

wait for 120 ns;

-- Load Register A with 0011

wait for 300 ns;

start <= '1';

instruction <= "000";

destination <= "00";

BUS\_IN <= "0011";

CIN <= '0';

wait for 100 ns;

start <= '0';

-- Load Register B with 1010

wait for 300 ns;

start <= '1';

instruction <= "000";

destination <= "01";

BUS\_IN <= "1010";

CIN <= '0';

wait for 100 ns;

start <= '0';

-- Compute A OR B and store in A

wait for 300 ns;

start <= '1';

instruction <= "100";

destination <= "00";

BUS\_IN <= "0000";

CIN <= '0';

wait for 100 ns;

start <= '0';

-- Restore A Load Register A with 0011

wait for 300 ns;

start <= '1';

instruction <= "000";

destination <= "00";

BUS\_IN <= "0011";

CIN <= '0';

wait for 100 ns;

start <= '0';

-- Compute A AND B and store in B

wait for 300 ns;

start <= '1';

instruction <= "011";

destination <= "01";

BUS\_IN <= "0000";

CIN <= '0';

wait for 100 ns;

start <= '0';

-- Restore B Load Register B with 1010

wait for 300 ns;

start <= '1';

instruction <= "000";

destination <= "01";

BUS\_IN <= "1010";

CIN <= '0';

wait for 100 ns;

start <= '0';

-- Compute A + B and Store in A

wait for 300 ns;

start <= '1';

instruction <= "101";

destination <= "00";

BUS\_IN <= "0000";

CIN <= '0';

wait for 100 ns;

start <= '0';

-- Restore A Load Register A with 0011

wait for 300 ns;

start <= '1';

instruction <= "000";

destination <= "00";

BUS\_IN <= "0011";

CIN <= '0';

wait for 100 ns;

start <= '0';

-- Compute A - B and Store in A

wait for 300 ns;

start <= '1';

instruction <= "110";

destination <= "00";

BUS\_IN <= "0000";

CIN <= '0';

wait for 100 ns;

start <= '0';

-- Restore A Load Register A with 0011

wait for 300 ns;

start <= '1';

instruction <= "000";

destination <= "00";

BUS\_IN <= "0011";

CIN <= '0';

wait for 100 ns;

start <= '0';

-- Copy A to B

wait for 300 ns;

start <= '1';

instruction <= "001";

destination <= "01";

BUS\_IN <= "0000";

CIN <= '0';

wait for 100 ns;

start <= '0';

-- Restore B Load Register B with 1010

wait for 300 ns;

start <= '1';

instruction <= "000";

destination <= "01";

BUS\_IN <= "1010";

CIN <= '0';

wait for 100 ns;

start <= '0';

-- Load Register B with 0101

wait for 300 ns;

start <= '1';

instruction <= "000";

destination <= "01";

BUS\_IN <= "0101";

CIN <= '0';

wait for 100 ns;

start <= '0';

-- Copy B to A

wait for 300 ns;

start <= '1';

instruction <= "010";

destination <= "00";

BUS\_IN <= "0000";

CIN <= '0';

wait for 100 ns;

start <= '0';

wait for 1 us;

end process;

END;

The GCPU Controller is:

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.std\_logic\_arith.all;

use IEEE.STD\_LOGIC\_UNSIGNED.ALL;

entity gcpu\_controller is

port (

CLK : in std\_logic;

reset : in std\_logic;

instruction : in std\_logic\_vector(2 downto 0);

destination : in std\_logic\_vector(1 downto 0);

start : in std\_logic;

SEL1 : out std\_logic\_vector(1 downto 0);

SEL2 : out std\_logic\_vector(1 downto 0);

LDA : out std\_logic;

LDB : out std\_logic;

OP : out std\_logic\_vector(1 downto 0);

done : out std\_logic

);

end gcpu\_controller;

architecture Behavioral of gcpu\_controller is

type gcpu\_states is (Waiting, Load1, LoadA, LoadB, CopyA1, CopyA\_A, CopyA\_B, COPYB1, CopyB\_A, COPYB\_B,

AND1, AND\_A, AND\_B, OR1, OR\_A, OR\_B, ADD1, ADD\_A, ADD\_B, SUB1, SUB\_A, SUB\_B);

signal present\_state, next\_state : gcpu\_states;

signal doneint : std\_logic;

begin

DoneOutput : process (CLK, reset) --clock of 2s period 50% duty cycle

begin

if(reset= '1') then

done <= '0';

elsif (CLK='1' and CLK'event) then

done <= doneint;

end if;

end process;

StateMachineNext : process (CLK, reset) --clock of 2s period 50% duty cycle

begin

if(reset= '1') then

present\_state <= Waiting;

elsif (CLK='1' and CLK'event) then

present\_state <= next\_state;

end if;

end process;

StateMachineStates : process (instruction, destination, start, present\_state, next\_state) --clock of 2s period 50% duty cycle

begin

SEL1 <= "00"; -- out std\_logic\_vector(1 downto 0);

SEL2 <= "00"; -- out std\_logic\_vector(1 downto 0);

LDA <= '0'; -- out std\_logic;

LDB <= '0'; -- out std\_logic;

OP <= "00"; -- out std\_logic\_vector(1 downto 0);

doneint <= '0'; -- out std\_logic;

case present\_state is

when Waiting =>

if (start = '1') then

case (instruction) is

when "000" => next\_state <= Load1;

when "001" => next\_state <= CopyA1;

when "010" => next\_state <= CopyB1;

when "011" => next\_state <= AND1;

when "100" => next\_state <= OR1;

when "101" => next\_state <= ADD1;

when "110" => next\_state <= SUB1;

when others => next\_state <= Waiting;

end case;

end if;

when Load1 =>

case (destination) is

when "00" => next\_state <= LoadA;

when "01" => next\_state <= LoadB;

when others => NULL;

end case;

when LoadA =>

SEL1 <= "01"; -- select the Bus In

SEL2 <= "01"; -- select the Bus In

doneint <= '1';

LDA <= '1';

next\_state <= Waiting;

when LoadB =>

SEL1 <= "01"; -- select the Bus In

SEL2 <= "01"; -- select the Bus In

doneint <= '1';

LDB <= '1';

next\_state <= Waiting;

when CopyA1 =>

SEL1 <= "00"; -- out std\_logic\_vector(1 downto 0);

SEL2 <= "00"; -- out std\_logic\_vector(1 downto 0);

case (destination) is

when "00" => next\_state <= CopyA\_A;

when "01" => next\_state <= CopyA\_B;

when others => NULL;

end case;

when CopyA\_A =>

SEL1 <= "00"; -- out std\_logic\_vector(1 downto 0);

SEL2 <= "00"; -- out std\_logic\_vector(1 downto 0);

doneint <= '1';

LDA <= '1';

next\_state <= Waiting;

when CopyA\_B =>

SEL1 <= "00"; -- out std\_logic\_vector(1 downto 0);

SEL2 <= "00"; -- out std\_logic\_vector(1 downto 0);

doneint <= '1';

LDB <= '1';

next\_state <= Waiting;

when CopyB1 =>

SEL1 <= "00"; -- out std\_logic\_vector(1 downto 0);

SEL2 <= "00"; -- out std\_logic\_vector(1 downto 0);

case (destination) is

when "00" => next\_state <= CopyB\_A;

when "01" => next\_state <= CopyB\_B;

when others => NULL;

end case;

when CopyB\_A =>

SEL1 <= "11"; -- out std\_logic\_vector(1 downto 0);

SEL2 <= "11"; -- out std\_logic\_vector(1 downto 0);

doneint <= '1';

LDA <= '1';

next\_state <= Waiting;

when CopyB\_B =>

SEL1 <= "11"; -- out std\_logic\_vector(1 downto 0);

SEL2 <= "11"; -- out std\_logic\_vector(1 downto 0);

doneint <= '1';

LDB <= '1';

next\_state <= Waiting;

when AND1 =>

OP <= "01"; -- AND operator

SEL1 <= "00"; -- out std\_logic\_vector(1 downto 0);

SEL2 <= "00"; -- out std\_logic\_vector(1 downto 0);

case (destination) is

when "00" => next\_state <= AND\_A;

when "01" => next\_state <= AND\_B;

when others => NULL;

end case;

when AND\_A =>

OP <= "01"; -- AND operator

SEL1 <= "10"; -- out std\_logic\_vector(1 downto 0);

SEL2 <= "10"; -- out std\_logic\_vector(1 downto 0);

LDA <= '1';

doneint <= '1';

next\_state <= Waiting;

when AND\_B =>

OP <= "01"; -- AND operator

SEL1 <= "10"; -- out std\_logic\_vector(1 downto 0);

SEL2 <= "10"; -- out std\_logic\_vector(1 downto 0);

LDB <= '1';

doneint <= '1';

next\_state <= Waiting;

when OR1 =>

OP <= "10"; -- OR operator

SEL1 <= "10"; -- out std\_logic\_vector(1 downto 0);

SEL2 <= "10"; -- out std\_logic\_vector(1 downto 0);

case (destination) is

when "00" => next\_state <= OR\_A;

when "01" => next\_state <= OR\_B;

when others => NULL;

end case;

when OR\_A =>

OP <= "10"; -- OR operator

SEL1 <= "10"; -- out std\_logic\_vector(1 downto 0);

SEL2 <= "10"; -- out std\_logic\_vector(1 downto 0);

LDA <= '1';

doneint <= '1';

next\_state <= Waiting;

when OR\_B =>

OP <= "10"; -- OR operator

SEL1 <= "10"; -- out std\_logic\_vector(1 downto 0);

SEL2 <= "10"; -- out std\_logic\_vector(1 downto 0);

LDB <= '1';

doneint <= '1';

next\_state <= Waiting;

when ADD1 =>

OP <= "11"; -- ADD operator

SEL1 <= "10"; -- out std\_logic\_vector(1 downto 0);

SEL2 <= "10"; -- out std\_logic\_vector(1 downto 0);

case (destination) is

when "00" => next\_state <= ADD\_A;

when "01" => next\_state <= ADD\_B;

when others => NULL;

end case;

when ADD\_A =>

OP <= "11"; -- ADD operator

SEL1 <= "10"; -- out std\_logic\_vector(1 downto 0);

SEL2 <= "10"; -- out std\_logic\_vector(1 downto 0);

LDA <= '1';

doneint <= '1';

next\_state <= Waiting;

when ADD\_B =>

OP <= "11"; -- ADD operator

SEL1 <= "10"; -- out std\_logic\_vector(1 downto 0);

SEL2 <= "10"; -- out std\_logic\_vector(1 downto 0);

LDB <= '1';

doneint <= '1';

next\_state <= Waiting;

when SUB1 =>

OP <= "00"; -- Generate the 2's complement of A

SEL1 <= "10"; -- out std\_logic\_vector(1 downto 0);

SEL2 <= "10"; -- out std\_logic\_vector(1 downto 0);

LDA <= '1'; -- Load A with 2's Complement of A

case (destination) is

when "00" => next\_state <= SUB\_A;

when "01" => next\_state <= SUB\_B;

when others => NULL;

end case;

when SUB\_A =>

OP <= "11"; -- ADD operator

SEL1 <= "10"; -- out std\_logic\_vector(1 downto 0);

SEL2 <= "10"; -- out std\_logic\_vector(1 downto 0);

LDA <= '1';

doneint <= '1';

next\_state <= Waiting;

when SUB\_B =>

OP <= "11"; -- ADD operator

SEL1 <= "10"; -- out std\_logic\_vector(1 downto 0);

SEL2 <= "10"; -- out std\_logic\_vector(1 downto 0);

LDB <= '1';

doneint <= '1';

next\_state <= Waiting;

when others => next\_state <= Waiting;

end case;

end process;

end Behavioral;

The Reg Datapath is:

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.std\_logic\_arith.all;

use IEEE.STD\_LOGIC\_UNSIGNED.ALL;

entity REG\_DATAPATH is

Port (

CLK : in std\_logic;

reset : in std\_logic;

BUS\_IN : in std\_logic\_vector(3 downto 0);

CIN : in std\_logic;

SEL1 : in std\_logic\_vector(1 downto 0);

SEL2 : in std\_logic\_vector(1 downto 0);

LDA : in std\_logic;

LDB : in std\_logic;

OP : in std\_logic\_vector(1 downto 0);

COUT : out std\_logic;

BUS\_OUT : out std\_logic\_vector(3 downto 0)

);

end REG\_DATAPATH;

architecture structural of REG\_DATAPATH is

component REG\_ALU

--- Register ALU declaration. Implemented in a separated projet. Problem 1

port (

A : in std\_logic\_vector(3 downto 0);

B : in std\_logic\_vector(3 downto 0);

CIN : in std\_logic;

OP : in std\_logic\_vector(1 downto 0);

COUT : out std\_logic;

F : out std\_logic\_vector(3 downto 0)

);

end component;

signal Mux1\_RegA : std\_logic\_vector(3 downto 0); -- Mux 1 to register A

signal Mux2\_RegB : std\_logic\_vector(3 downto 0); -- Mux 2 to register B

signal RegisterA : std\_logic\_vector(3 downto 0); -- Register A

signal RegisterB : std\_logic\_vector(3 downto 0); -- Register B

signal RegisterC : std\_logic\_vector(3 downto 0); -- Register B

signal ALUOutput : std\_logic\_vector(3 downto 0); -- ALU Output

-- --ALU\_OUT to mux1

--- ALU\_OUT also goes to bus and back to multiplexers

begin

Mux1: process(SEL1, RegisterA, BUS\_IN, ALUOutput, RegisterB)

begin

case SEL1 is

when "00" =>

Mux1\_RegA <= RegisterA;

when "01" =>

Mux1\_RegA <= BUS\_IN;

when "10" =>

Mux1\_RegA <= ALUOutput;

when "11" =>

Mux1\_RegA <= RegisterB;

when others =>

Mux1\_RegA <= (others => '0');

end case;

end process;

Mux2: process(SEL1, RegisterA, BUS\_IN, ALUOutput, RegisterB)

begin

case SEL1 is

when "00" =>

Mux2\_RegB <= RegisterA;

when "01" =>

Mux2\_RegB <= BUS\_IN;

when "10" =>

Mux2\_RegB <= ALUOutput;

when "11" =>

Mux2\_RegB <= RegisterB;

when others =>

Mux2\_RegB <= (others => '0');

end case;

end process;

RegA: process(CLK, reset)

begin

if (reset = '1') then

RegisterA <= (others => '0');

elsif (CLK'event and CLK='1') then

if (LDA = '1') then

RegisterA <= Mux1\_RegA;

end if;

end if;

end process;

RegB : process(CLK, reset)

begin

if (reset = '1') then

RegisterB <= (others => '0');

elsif (CLK'event and CLK='1') then

if (LDB = '1') then

RegisterB <= Mux2\_RegB;

end if;

end if;

end process;

RegC: process(CLK, reset)

begin

if (reset = '1') then

RegisterC <= (others => '0');

elsif (CLK'event and CLK='1') then

RegisterC <= ALUOutput;

end if;

end process;

TheALU : REG\_ALU

port map (

A => RegisterA, -- : in std\_logic\_vector(3 downto 0);

B => RegisterB, -- : in std\_logic\_vector(3 downto 0);

CIN => CIN, -- : in std\_logic;

COUT => COUT,

OP => OP,

F => ALUOutput -- : out std\_logic\_vector(3 downto 0)

);

-- BUS\_OUT <= ALUOutput;

BUS\_OUT <= RegisterC;

end structural;

The Reg ALU is:

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.std\_logic\_arith.all;

use IEEE.STD\_LOGIC\_UNSIGNED.ALL;

entity reg\_alu is

port (

A : in std\_logic\_vector(3 downto 0);

B : in std\_logic\_vector(3 downto 0);

CIN : in std\_logic;

OP : in std\_logic\_vector(1 downto 0);

COUT : out std\_logic;

F : out std\_logic\_vector(3 downto 0));

end reg\_alu;

architecture Behavioral of reg\_alu is

signal ALUOut : std\_logic\_vector(4 downto 0);

begin

process(A, B, OP)

begin

case OP is

when "00" =>

ALUOut <= '0' & NOT(A) + 1; -- 2s Complement of A

when "01" =>

ALUOut <= '0' & (A AND B); -- A AND B, this differs from template because instructions are different

when "10" =>

ALUOut <= '0' & (A OR B); -- A OR B, this differs from template because instructions are different

when "11" =>

ALUOut <= ('0' & A) + ('0'& B) + CIN; -- Bitwise A+B when "011"

when others =>

ALUOut <= (others =>'0');

end case;

end process;

F <= ALUOut(3 downto 0);

COUT <= ALUOut(4);

end Behavioral;