# EEL 3701 – Digital Logic and Computer Systems 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.

Problem 1: ALU

The ALU was designed as:

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.std\_LOGIC\_ARITH.ALL;

use IEEE.STD\_LOGIC\_UNSIGNED.ALL;

entity 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 ALU;

architecture Behavioral of ALU is

Signal TEMP: std\_logic\_vector(4 downto 0);

begin

process(A, B, OP)

begin

case OP is

when "00" =>

TEMP(3 downto 0) <= 1+ NOT(A);

when "01" =>

TEMP(3 downto 0) <= A OR B;

when "11" =>

TEMP <= ('0' & A) + ('0' & B) + CIN;

when "10" =>

TEMP(3 downto 0) <= A AND B;

when others =>

NULL;

end case;

end process;

F <= TEMP(3 downto 0);

COUT <= TEMP(4);

end Behavioral;

Problem 2: GCPU Data Path

The GCPU data path was designed as:

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.STD\_LOGIC\_ARITH.ALL;

use IEEE.STD\_LOGIC\_UNSIGNED.ALL;

entity DATAPATH is

Port(

CLK : in std\_logic;

reset : in std\_logic;

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

CIN : in std\_logic;

LDA : in std\_logic;

LDB : in std\_logic;

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

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

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

COUT : out std\_logic;

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

end DATAPATH;

architecture structural of DATAPATH is

component ALU

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\_TO\_REGA:std\_logic\_vector( 3 downto 0);

Signal MUX2\_TO\_REGB:std\_logic\_vector( 3 downto 0);

Signal REGA\_TO\_ALU:std\_logic\_vector( 3 downto 0);

Signal REGB\_TO\_ALU:std\_logic\_vector( 3 downto 0);

Signal ALU\_TO\_REGC:std\_logic\_vector( 3 downto 0);

Signal REGC\_TO\_BUSOUT:std\_logic\_vector( 3 downto 0);

begin

MUX1: process(MUX1\_TO\_REGA,REGA\_TO\_ALU,BUS\_IN,SEL1,REGB\_TO\_ALU,REGC\_TO\_BUSOUT)

begin

case SEL1 is

when "00" =>

MUX1\_TO\_REGA <= REGA\_TO\_ALU;

when "01" =>

MUX1\_TO\_REGA <= BUS\_IN;

when "10" =>

MUX1\_TO\_REGA <= ALU\_TO\_REGC;

when "11" =>

MUX1\_TO\_REGA <= REGB\_TO\_ALU;

end case;

end process;

MUX2: process(MUX2\_TO\_REGB,REGA\_TO\_ALU,BUS\_IN,SEL2,REGB\_TO\_ALU,ALU\_TO\_REGC)

begin

case SEL2 is

when "00" =>

MUX2\_TO\_REGB <= REGB\_TO\_ALU;

when "01" =>

MUX2\_TO\_REGB <= BUS\_IN;

when "10" =>

MUX2\_TO\_REGB <= ALU\_TO\_REGC;

when "11" =>

MUX2\_TO\_REGB <= REGA\_TO\_ALU;

end case;

end process;

REGA: process(CLK, reset, MUX1\_TO\_REGA,REGA\_TO\_ALU, LDA)

begin

if(reset = '1')then

REGA\_TO\_ALU <= (others => '0');

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

REGA\_TO\_ALU <= MUX1\_TO\_REGA;

end if;

end process;

REGB: process(CLK, reset, MUX2\_TO\_REGB, REGB\_TO\_ALU, LDB)

begin

if(reset = '1')then

REGB\_TO\_ALU <= (others => '0');

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

REGB\_TO\_ALU <= MUX2\_TO\_REGB;

end if;

end process;

REGC: process(CLK, reset, ALU\_TO\_REGC, REGC\_TO\_BUSOUT)

begin

if(reset = '1') then

REGC\_TO\_BUSOUT <= (others => '0');

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

REGC\_TO\_BUSOUT <= ALU\_TO\_REGC;

end if;

end process;

ALU\_INST: ALU Port map (

A => REGA\_TO\_ALU,

B => REGB\_TO\_ALU,

F => ALU\_TO\_REGC,

CIN => CIN,

COUT => COUT,

OP => OP);

end structural;

Problem 2 Questions:

1. All computations. The computation must operate on the data in the registers and loaded first.

Since the ALU has a 2’s complement, the input can be loaded and inverted so a NOT is possible for a bit if its one bit in a specific location (can’t be all 1’s or all 0’s and must have lowest bit as a 0 to begin but it could do this in the middle two bits of the four bit ALU.) So not all operations would work on all four bits, but there are a number of operations possible. The registers can then be AND’d, OR’d, added (XOR’d). This means that it could do a lot of logical operations such as NAND, NOR, XOR, XNOR, AND, OR all on the middle two bits given the 2’s complement won’t affect those bits if the upper and lower bit are both 0’s. The ALU can Add, Subtract, 2’s complement. Since it can AND and OR and Invert (using middle two bits) it can really do about any function of two bits. It does not have a shift capability so it cannot shift the bits but it does have a carry. Since the operands must all be in the register, it’s hard to force a carry without loading a third value. Since it can add, it could do multiple adds which is kind of like a multiply and could be made to multiply by 2 or 4 or 3 by repeatedly adding the same value around and around the ALU. The ALU could isolate a single bit by loading 0001, 0010, 0100 or 1000 in A and AND’ing it with B, to isolate a given bit in B. Same could be done to remove a single bit from A by AND’ing it with 0111, 1011, 1101 or 1110 to remove a bit from a register. It can test if a value is close to a limit (greater than) by adding another value such as unknown + 4 and if there is a carry out, clearly the value is greater than 3 (so it can compare two values by testing the carry out bit and using the adding capability.) Using this logic, it can test if a value is negative by adding 1000 to the value and seeing if there is a carry out.

1. A single operand instruction, using the Moore state machine, takes two clock cycles. One to go to the instruction state and a second to decide where to store the output. If this were a Mealy state machine, it might be done in one less clock cycle. If you count the idle clock cycle that has a “start” it takes three clock cycles. Depends if this idle clock cycle is part of the total or part of the lost time.

A two operand instruction first must load the first value into the register. Since there is no need to return to the idle state, it could take two clock cycles to perform the load (ignoring the starting idle state) then two more clock cycles to load the second operand into the second register as the ALU must use values from the registers. Then another clock cycle to load the result from the operation into a register (A, B, or send to the output.) So it would take 5 clock cycles to do the two operand operations (ignoring the extra clock cycle lost in the idle waiting state waiting for the “start” command.) If this extra “start” command was counted, it would take six clock cycles.

Problem 3: GCPU Controller

The GCPU Controller was designed as:

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.STD\_LOGIC\_ARITH.ALL;

use IEEE.STD\_LOGIC\_UNSIGNED.ALL;

entity CONTROL 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;

Done : out std\_logic;

LDA : out std\_logic;

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

LDB : out std\_logic;

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

CIN : out std\_logic;

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

COUT : in std\_logic);

end CONTROL;

Architecture Behavioral of CONTROL is

type GCPU\_STATES is (INITIAL, LOAD1, LOADA, LOADB, AND1, AND1\_A, AND1\_B, OR1, OR1\_A, OR1\_b, ADD1, ADD1\_A, ADD1\_B,

SUB1, SUB1\_STOREA, SUB1\_A, SUB1\_B, COPYA, COPYA\_A, COPYA\_B, COPYB, COPYB\_A, COPYB\_B);

Signal Presentstate, Nextstate : GCPU\_STATES;

begin

PresenttoNextState : process (CLK, reset)

begin

if(reset = '1') then

Presentstate <= INITIAL;

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

Presentstate <= Nextstate;

end if;

end process;

StateMachine : process (INSTRUCTION, DESTINATION, Start, Presentstate, Nextstate)

begin

LDA <= '0';

LDB <= '0';

SEL1 <= "00";

SEL2 <= "00";

OP <= "00";

Done <= '0';

case (Presentstate) is

when INITIAL =>

if (start = '1') then

case (INSTRUCTION) is

when "000" => Nextstate <= LOAD1;

when "001" => Nextstate <= COPYA;

when "010" => Nextstate <= COPYB;

when "011" => Nextstate <= AND1;

when "100" => Nextstate <= OR1;

when "101" => Nextstate <= ADD1;

when "110" => Nextstate <= SUB1;

when others => Nextstate <= Presentstate;

end case;

end if;

when LOAD1 =>

case (DESTINATION) is

when "00"=> Nextstate <= LOADA;

when "01"=> Nextstate <= LOADB;

when others => NULL;

end case;

when LOADA =>

Done <= '1';

LDA <= '1';

SEL1 <= "01";

Nextstate <= INITIAL;

when LOADB =>

Done <= '1';

LDB <= '1';

SEL2 <= "01";

Nextstate <= INITIAL;

when ADD1 =>

case (DESTINATION) is

when "00"=>

Nextstate <= ADD1\_A;

when "01"=>

Nextstate <= ADD1\_B;

when others => NULL;

end case;

when ADD1\_A =>

LDA <= '1';

OP <= "11";

SEL1 <= "10";

Done <= '1';

Nextstate <= INITIAL;

when ADD1\_B =>

LDB <= '1';

OP <= "11";

SEL2 <= "10";

Done <= '1';

Nextstate <= INITIAL;

when OR1 =>

case (DESTINATION) is

when "00"=>

Nextstate <= OR1\_A;

when "01"=>

Nextstate <= OR1\_B;

when others => NULL;

end case;

when OR1\_A =>

LDA <= '1';

OP <= "10";

SEL1 <= "10";

Done <= '1';

Nextstate <= INITIAL;

when OR1\_B =>

LDB <= '1';

OP <= "10";

SEL2 <= "10";

Done <= '1';

Nextstate <= INITIAL;

when AND1 =>

case (DESTINATION) is

when "00"=>

LDA <= '1';

OP <= "01";

SEL1 <= "10";

Done <= '1';

Nextstate <= AND1\_A;

when "01"=>

Nextstate <= AND1\_B;

when others => NULL;

end case;

when AND1\_A =>

LDA <= '1';

OP <= "01";

SEL1 <= "10";

Done <= '1';

Nextstate <= INITIAL;

when AND1\_B =>

LDB <= '1';

OP <= "01";

SEL2 <= "10";

Done <= '1';

Nextstate <= INITIAL;

when SUB1 =>

LDA <= '1'; -- Load register A with operand

SEL1 <= "01";

SEL2 <= "01";

OP <= "00";

Nextstate <= SUB1\_STOREA;

when SUB1\_STOREA =>

LDA <= '1'; -- Load register A and do 2's complement

SEL1 <= "10";

SEL2 <= "10";

OP <= "00";

case (DESTINATION) is

when "00"=>

Nextstate <= SUB1\_A;

when "01"=>

Nextstate <= SUB1\_B;

when others => NULL;

end case;

when SUB1\_A =>

LDA <= '1';

OP <= "11";

SEL1 <= "10";

Done <= '1';

Nextstate <= INITIAL;

when SUB1\_B =>

LDB <= '1';

OP <= "11";

SEL2 <= "10";

Done <= '1';

Nextstate <= INITIAL;

when COPYA =>

case (DESTINATION) is

when "00"=>

Nextstate <= COPYA\_A;

when "01"=>

Nextstate <= COPYA\_B;

when others => NULL;

end case;

when COPYA\_A =>

LDA <= '1';

OP <= "11";

SEL1 <= "00";

Done <= '1';

Nextstate <= INITIAL;

when COPYA\_B =>

LDB <= '1';

OP <= "11";

SEL2 <= "00";

Done <= '1';

Nextstate <= INITIAL;

when COPYB =>

case (DESTINATION) is

when "00"=>

Nextstate <= COPYB\_A;

when "01"=>

Nextstate <= COPYB\_B;

when others => NULL;

end case;

when COPYB\_A =>

LDA <= '1';

OP <= "11";

SEL1 <= "11";

Done <= '1';

Nextstate <= INITIAL;

when COPYB\_B =>

LDB <= '1';

OP <= "11";

SEL2 <= "11";

Done <= '1';

Nextstate <= INITIAL;

when others => Nextstate <= INITIAL;

end case;

end process;

end Behavioral;

The main entity for LAB05 was designed as:

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.std\_logic\_arith.all;

use IEEE.STD\_LOGIC\_UNSIGNED.ALL;

entity Lab05 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;

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

CIN: in std\_logic;

COUT: out std\_logic;

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

Done: out std\_logic);

end Lab05;

architecture structural of Lab05 is

COMPONENT DATAPATH

port (

CLK : in std\_logic;

reset : in std\_logic;

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

CIN : in std\_logic;

LDA : in std\_logic;

LDB : in std\_logic;

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

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

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

COUT : out std\_logic;

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

END COMPONENT;

COMPONENT 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;

Done: out std\_logic;

LDA : out std\_logic;

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

LDB: out std\_logic;

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

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

end COMPONENT;

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

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

signal LDA : std\_logic;

signal LDB : std\_logic;

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

begin

TheRegDatapath : DATAPATH

port map (

CLK => CLK,

reset => reset,

BUS\_IN => BUS\_IN,

CIN => CIN,

SEL1 => SEL1,

SEL2 => SEL2,

LDA => LDA,

LDB => LDB,

OP => OP,

COUT => COUT,

BUS\_OUT => BUS\_OUT);

Controller: CONTROL

port map (

CLK => CLK,

reset => reset,

instruction => instruction,

destination => destination,

Start => Start,

SEL1 => SEL1,

SEL2 => SEL2,

LDA => LDA,

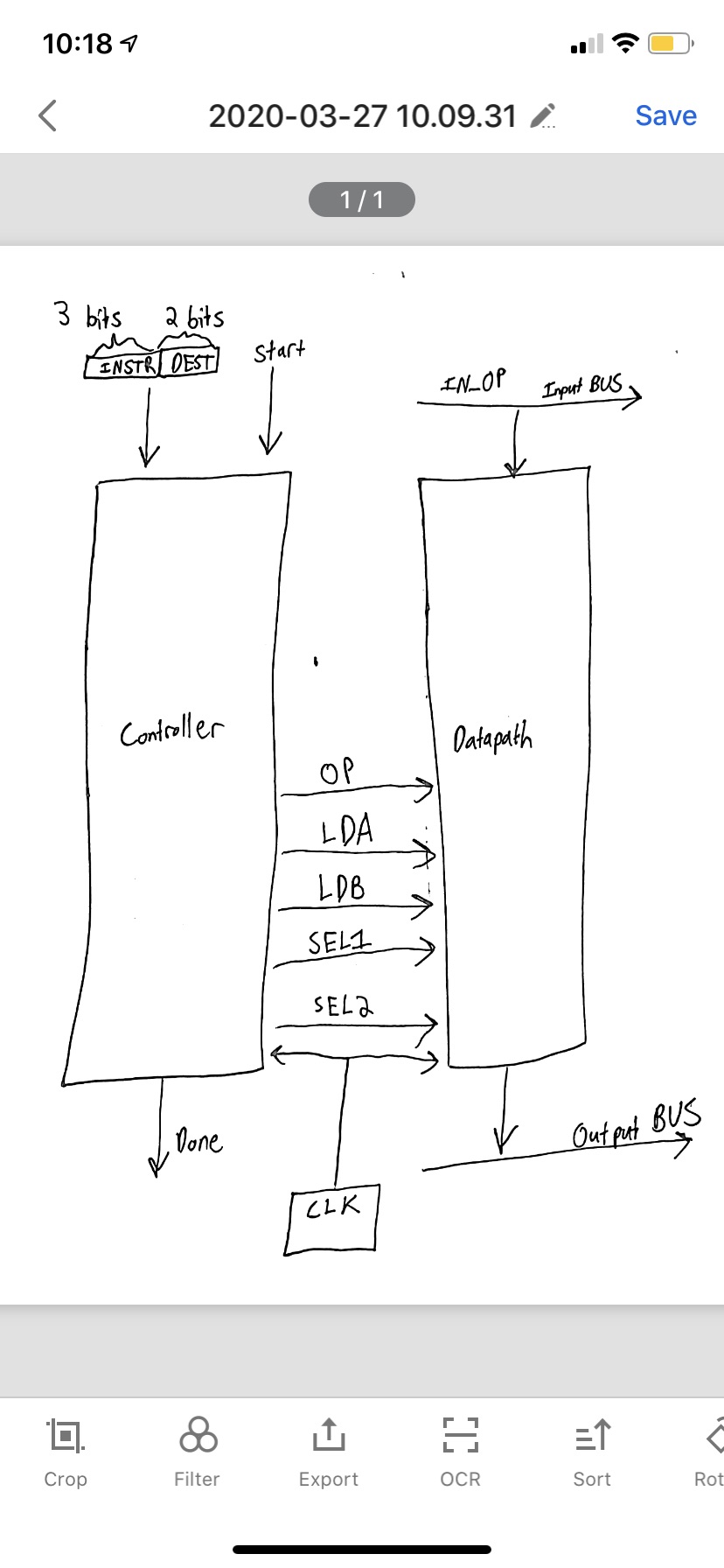
LDB => LDB,

OP => OP,

Done => Done );

end structural;

The diagram of the controller is:



The diagram for the Moore state machine is:

