# EEL 3701 – Digital Logic and Computer Systems Lab 6

# Greg Bolling

# UF ID 64911960

## Problem Statement

The goal of the lab is to design, develop, and implement using the G-CPU from Lab 5 and adding an autonomous operation to the program state machine. This adds a program counter and a register to load a new program address and adds instructions for Begin, End and 2’s complement.

## Design

The program was copied from lab 05 and started using that code. The state machine was modified to allow it to operate automatically and not wait for a “start” signal. Also, a few instructions were added for 2’s complement, and a Begin and End instruction were added.

This changed the controller state machine and changed the instruction size of the operation from 3 bits to 4 bit. The ROM was loaded with a program that was asked for in the lab and that is what will be simulated in the lab.

A Load Address state was added because the first address after reset seemed to have a problem with the delay between being presented at the input and loading at the PCLoad command. The delay caused it to run the first instruction a second time but that was because it took two clock cycles to get the address input loaded and then get the value from the ROM and then run the instruction.

A program was written to perform the following:

a = 14, b = 10

c = 3\*a + b

OpCode Dest Operand Instruction

1000 00 0000 (Program Begin)

0000 01 1100 (Load Register A with 1410)

0000 10 1010 (Load Register B with 1010)

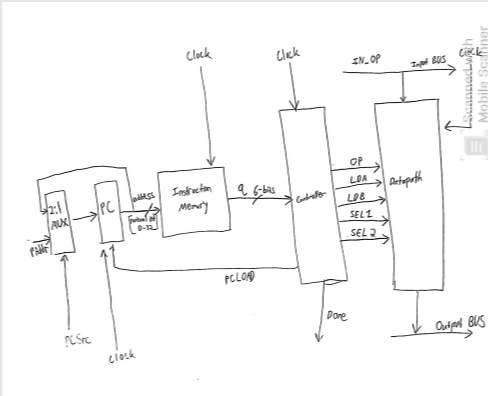
0101 10 0000 (Add A + B and store in B, B has A + B)

0101 10 0000 (Add A + B and store in B, B has 2\*A + B)

0101 10 0000 (Add A + B and store in B, B has 3\*A + B)

1001 00 0000 (Program End)

## Diagrams of Design



A 2:1 mux was added in order to determine whether the address needed to enter into the program counter from the user versus whether the address needed to stay the same. The address loops back around from program counter in order to keep program counter address constant. PCload determines whether Program counter is incremented or not. The instruction memory determines the instructions for the controller based upon the natural integer address coming from the program counter. There is a clock added to every individual module.

I added a PCLoad instruction to the state machine controller. The controller had problems with telling the ALU to grab the value so I had to delay that output one clock cycle so the register could catch the output of the ALU and put that on the Bus\_out.

I had problems with the ROM because it wasn’t working with the add instruction so I converted everything over to std\_logic\_vector and had to figure out a conversion from VHDL to turn that into an integer to give that to the array in the ROM. It works but seems a bit odd. Maybe next time no integers anywhere as that seems to cause problems.

# Two Programs

Two programs were loaded into the ROM at the same time, tmp(nn) is an address decode for the ROM. They were:

-- Op Code Dest Operand Instruction

-- 1000 00 0000 (Program Begin)

tmp(10) := "100000";

-- Op Code Dest Operand Instruction

-- 0000 00 1100 (Load Register A with 0010)

tmp(11) := "000000";

-- Op Code Dest Operand Instruction

-- 0000 01 1010 (Load Register B with 0011)

tmp(12) := "000001";

-- Op Code Dest Operand Instruction

-- 0101 00 0000 (Add A + B and store in A, A has A + B)

tmp(13) := "010100";

-- Op Code Dest Operand Instruction

-- 0101 00 0000 (Add A + B and store in A, A has A + 2\*B)

tmp(14) := "010100";

-- Op Code Dest Operand Instruction

-- 0101 01 0000 (OR A/B and store in B, B has A OR B)

tmp(15) := "010001";

-- Op Code Dest Operand Instruction

-- 1001 00 0000 (Program End)

tmp(16) := "100100";

And also loaded at a later location was:

-- Op Code Dest Operand Instruction

-- 1000 00 0000 (Program Begin)

tmp(21) := "100000";

-- Op Code Dest Operand Instruction

-- 0000 00 1100 (Load Register A with 0010)

tmp(22) := "000000";

-- Op Code Dest Operand Instruction

-- 0000 01 1010 (Load Register B with 0111)

tmp(23) := "000001";

-- Op Code Dest Operand Instruction

-- 0101 01 0000 (Add B - A and store in B)

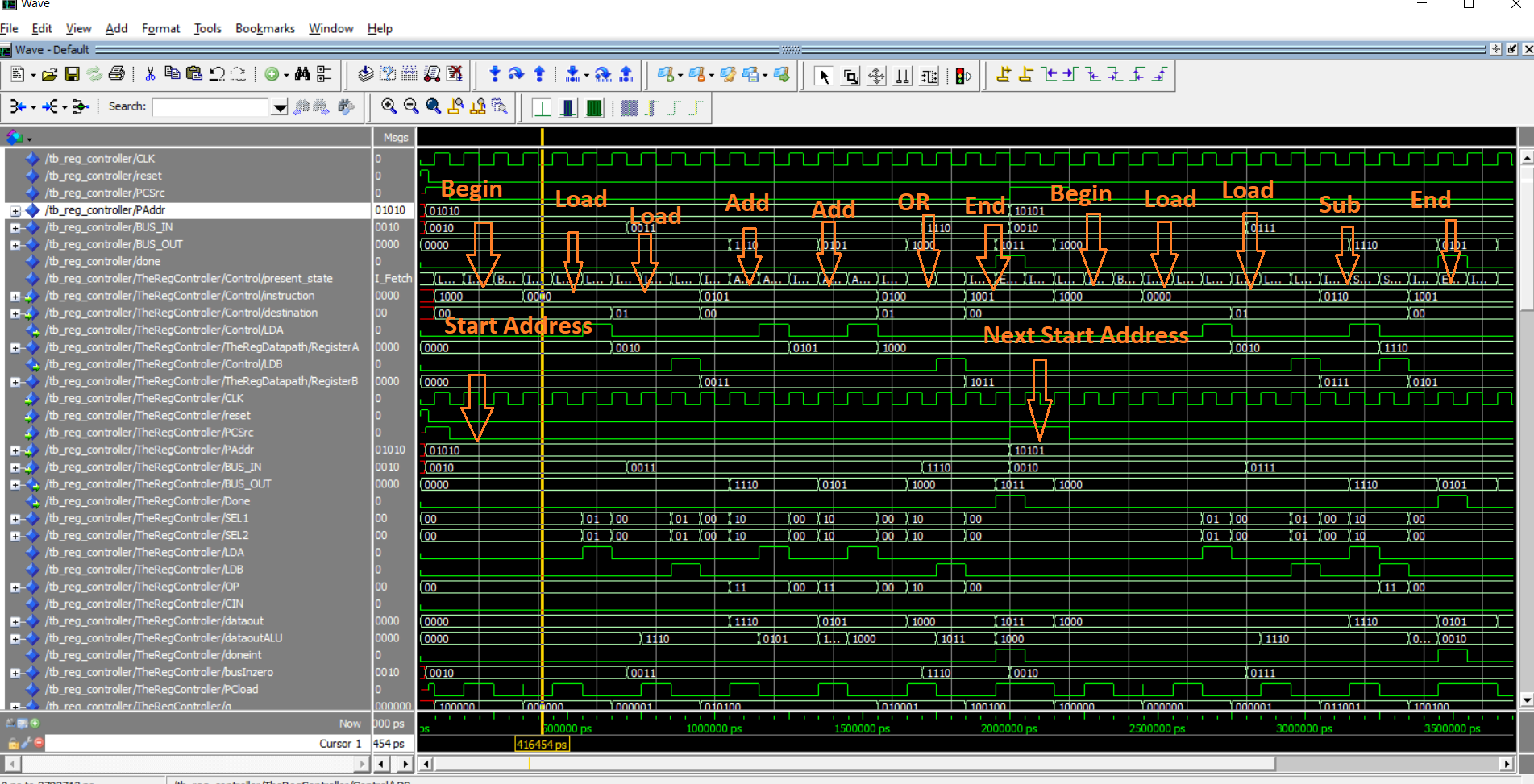
tmp(24) := "011001";

-- Op Code Dest Operand Instruction

-- 1001 00 0000 (Program End)

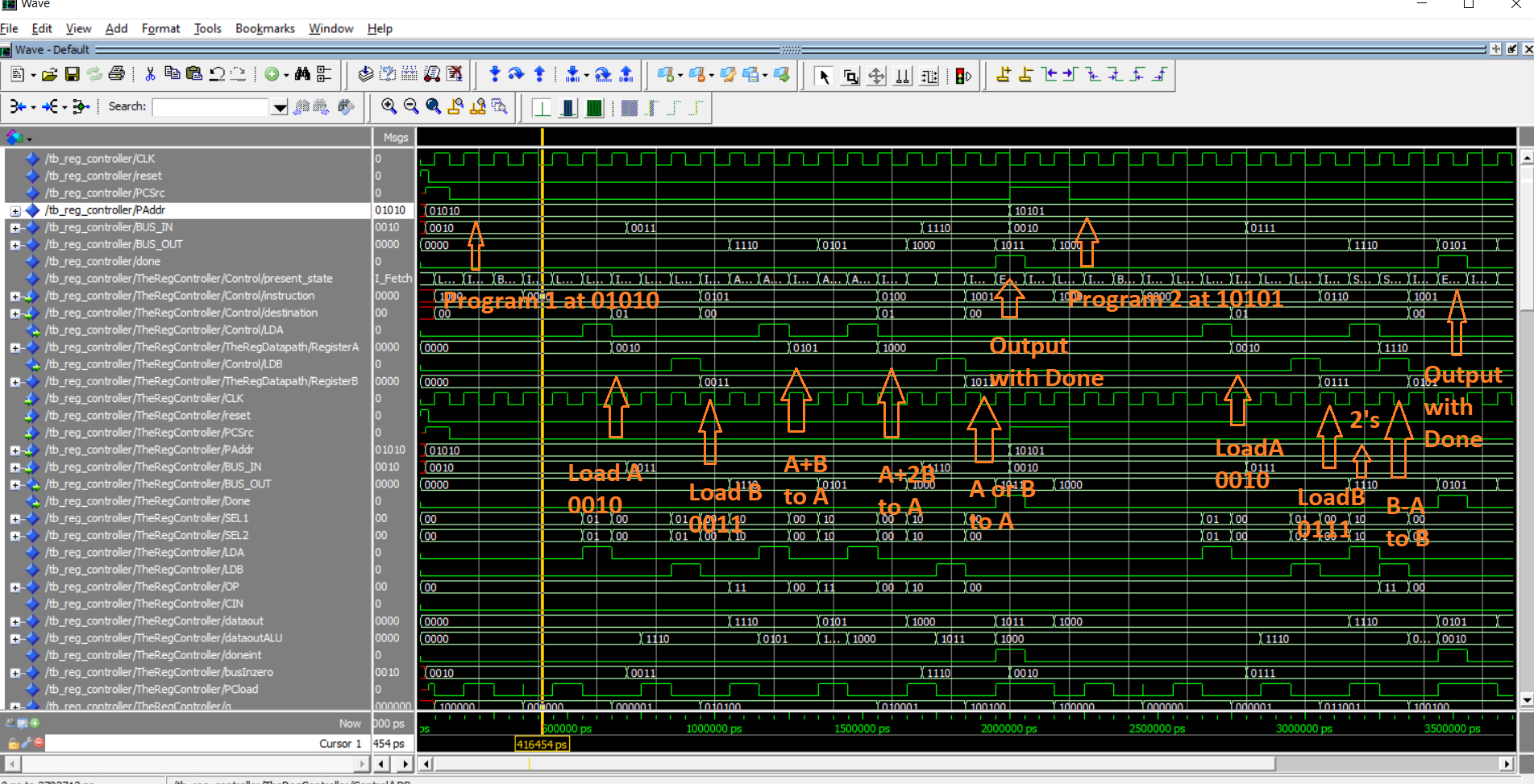
tmp(25) := "100100";

These two programs were run one after the other as follows:



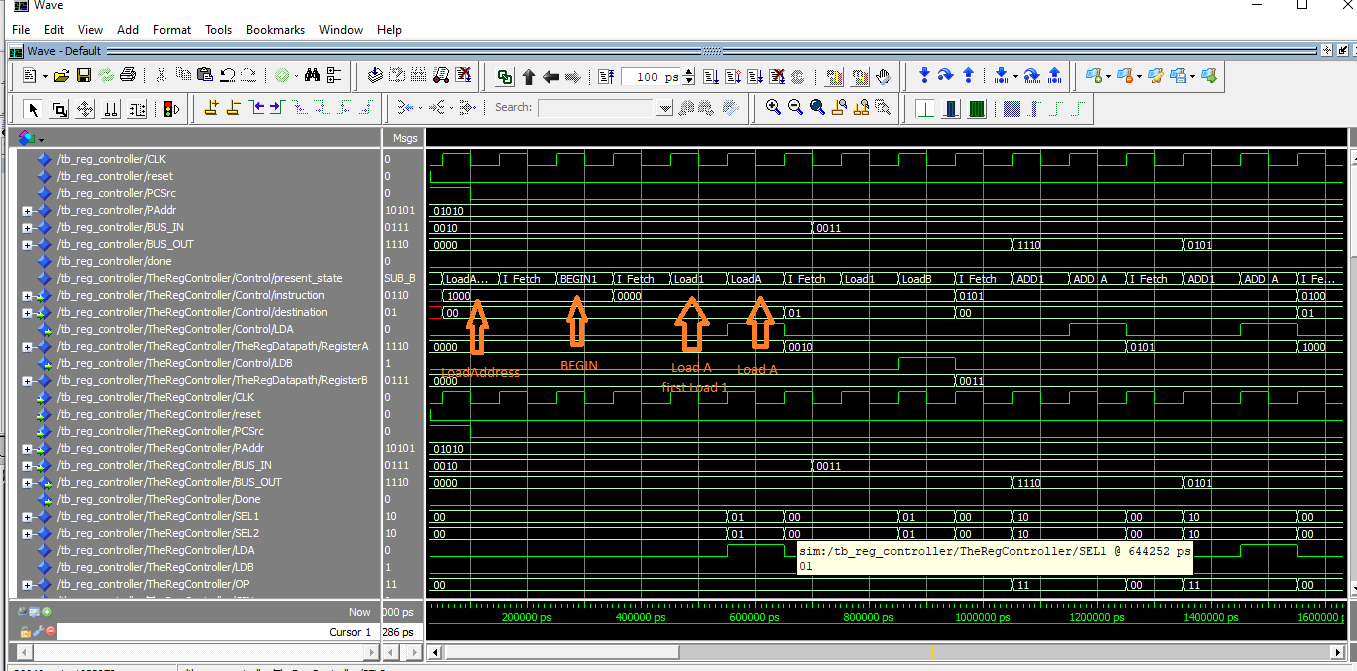
As can be seen, the instructions highlighted in orange were the same ones run form the ROM and at the correct start address.

The Programs that were run are annotated again here with the same sequence:



The Load commands are shown to see that the right destination was selected for the instruction. The command to process the start was with a begin instruction but those were not shown again here because they were shown in the previous screen shot.

Finally, an example of the instructions sequencing through the state machine can be seen in the next diagram. The typical sequence follows the state machine with Begin and then back to IFetch. Then to Load 1 and then Load A and so on.



# VHLD Files

# ALU

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;

# PCounter

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.numeric\_std.all;

use IEEE.std\_logic\_arith.all;

use IEEE.STD\_LOGIC\_UNSIGNED.ALL;

entity PCOUNTER is

port

(

reset: in std\_logic;

addr : out std\_logic\_vector(4 downto 0);

PAddr: in std\_logic\_vector(4 downto 0);

PCSrc: in std\_logic;

PCload:in std\_logic;

clk : in std\_logic

);

end entity;

architecture structural of PCOUNTER is

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

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

begin

Mux: process( PCSrc, address, PAddr)

begin

if ((PCSrc = '0')) then

address\_mux <= address + 1;

else

address\_mux <= PAddr;

end if;

end process;

Counter: process( reset, clk)

begin

if (reset = '1') then

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

elsif(rising\_edge(clk)) then

if ((PCload = '1')) then

address <= address\_mux;

end if;

end if;

end process;

addr <= address;

end structural;

# CGPU Controller

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(3 downto 0);

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

C\_in : in std\_logic;

captureALU : out std\_logic;

C\_set : out std\_logic;

C\_reset : out std\_logic;

PCload : out 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 (I\_Fetch, LoadAddress, Load1, LoadA, LoadB, CopyA1, CopyA\_A, CopyA\_B, COPYB1, CopyB\_A, COPYB\_B, BEGIN1, END1,

AND1, AND\_A, AND\_B, OR1, OR\_A, OR\_B, ADD1, ADD\_A, ADD\_B, SUB1, SUB\_A, SUB\_B, TWOCOMP1\_A, TWOCOMP1\_B);

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

signal doneint : std\_logic;

signal captureALUdelay : std\_logic;

begin

done <= doneint;

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

begin

if(reset= '1') then

present\_state <= I\_Fetch;

captureALU <= '0';

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

present\_state <= next\_state;

captureALU <= captureALUdelay;

end if;

end process;

StateMachineStates : process (instruction, destination, present\_state, next\_state, reset) --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;

C\_set <= '0';

C\_reset <= '0';

PCload <= '0';

captureALUdelay<= '0';

next\_state <= present\_state;

case present\_state is

when I\_Fetch =>

PCload <= not C\_in;

if (C\_in = '0') then

next\_state <= LoadAddress;

C\_set <= '1';

else

case (instruction) is

when "0000" => next\_state <= Load1;

when "0001" => next\_state <= CopyA1;

when "0010" => next\_state <= CopyB1;

when "0011" => next\_state <= AND1;

when "0100" => next\_state <= OR1;

when "0101" => next\_state <= ADD1;

when "0110" => next\_state <= SUB1;

when "0111" =>

case (destination) is

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

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

when others => NULL;

end case;

when "1000" => next\_state <= BEGIN1;

PCload <= '1';

when "1001" => next\_state <= END1;

when others => next\_state <= I\_Fetch;

end case;

end if;

when LoadAddress =>

--PCload <= '1';

next\_state <= I\_Fetch;

when Load1 =>

PCload <= '1';

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

LDA <= '1';

next\_state <= I\_Fetch;

captureALUdelay <= '1';

when LoadB =>

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

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

LDB <= '1';

next\_state <= I\_Fetch;

captureALUdelay <= '1';

when CopyA1 =>

PCload <= '1';

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);

LDA <= '1';

next\_state <= I\_Fetch;

captureALUdelay <= '1';

when CopyA\_B =>

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

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

LDB <= '1';

next\_state <= I\_Fetch;

captureALUdelay <= '1';

when CopyB1 =>

PCload <= '1';

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);

LDA <= '1';

next\_state <= I\_Fetch;

captureALUdelay <= '1';

when CopyB\_B =>

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

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

LDB <= '1';

next\_state <= I\_Fetch;

captureALUdelay <= '1';

when AND1 =>

PCload <= '1';

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';

next\_state <= I\_Fetch;

captureALUdelay <= '1';

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';

next\_state <= I\_Fetch;

captureALUdelay <= '1';

when OR1 =>

PCload <= '1';

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';

next\_state <= I\_Fetch;

captureALUdelay <= '1';

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';

next\_state <= I\_Fetch;

captureALUdelay <= '1';

when ADD1 =>

PCload <= '1';

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';

next\_state <= I\_Fetch;

captureALUdelay <= '1';

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';

next\_state <= I\_Fetch;

captureALUdelay <= '1';

when SUB1 =>

PCload <= '1';

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';

next\_state <= I\_Fetch;

captureALUdelay <= '1';

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';

next\_state <= I\_Fetch;

captureALUdelay <= '1';

when TWOCOMP1\_A =>

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

SEL1 <= "10";

SEL2 <= "10";

OP <= "00";

next\_state <= I\_Fetch;

captureALUdelay <= '1';

when TWOCOMP1\_B =>

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

SEL1 <= "10";

SEL2 <= "10";

OP <= "00";

next\_state <= I\_Fetch;

captureALUdelay <= '1';

when BEGIN1 =>

C\_set <= '1';

next\_state <= I\_Fetch;

captureALUdelay <= '1';

--PCload <= '1';

when END1 =>

doneint <= '1';

C\_reset <= '1';

next\_state <= I\_Fetch;

captureALUdelay <= '1';

PCload <= '1';

when others => next\_state <= I\_Fetch;

end case;

end process;

end Behavioral;

# ROM

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.numeric\_std.all;

entity ROM\_32\_6 is

port

(

addr : in std\_logic\_vector(4 downto 0);

clk : in std\_logic;

q : out std\_logic\_vector(5 downto 0)

);

end entity;

architecture rtl of ROM\_32\_6 is

-- Build a 2-D array type for the RoM

subtype word\_t is std\_logic\_vector(5 downto 0);

type memory\_t is array(31 downto 0) of word\_t;

function init\_rom

return memory\_t is

variable tmp : memory\_t := (others => (others => '0'));

begin

tmp(0) := "100000"; -- begin

tmp(1) := "100100"; -- end

tmp(2) := "100000"; -- begin

tmp(3) := "100100"; -- end

tmp(4) := "100000"; -- begin

tmp(5) := "100100"; -- end

tmp(6) := "100000"; -- begin

tmp(7) := "100100"; -- end

tmp(8) := "100000"; -- begin

tmp(9) := "100100"; -- end

-- Op Code Dest Operand Instruction

-- 1000 00 0000 (Program Begin)

tmp(10) := "100000";

-- Op Code Dest Operand Instruction

-- 0000 00 1100 (Load Register A with 0010)

tmp(11) := "000000";

-- Op Code Dest Operand Instruction

-- 0000 01 1010 (Load Register B with 0011)

tmp(12) := "000001";

-- Op Code Dest Operand Instruction

-- 0101 00 0000 (Add A + B and store in A, A has A + B)

tmp(13) := "010100";

-- Op Code Dest Operand Instruction

-- 0101 00 0000 (Add A + B and store in A, A has A + 2\*B)

tmp(14) := "010100";

-- Op Code Dest Operand Instruction

-- 0101 01 0000 (OR A/B and store in B, B has A OR B)

tmp(15) := "010001";

-- Op Code Dest Operand Instruction

-- 1001 00 0000 (Program End)

tmp(16) := "100100";

tmp(17) := "100000"; -- begin

tmp(18) := "100100"; -- end

tmp(19) := "100000"; -- begin

tmp(20) := "100100"; -- end

-- Op Code Dest Operand Instruction

-- 1000 00 0000 (Program Begin)

tmp(21) := "100000";

-- Op Code Dest Operand Instruction

-- 0000 00 1100 (Load Register A with 0010)

tmp(22) := "000000";

-- Op Code Dest Operand Instruction

-- 0000 01 1010 (Load Register B with 0111)

tmp(23) := "000001";

-- Op Code Dest Operand Instruction

-- 0101 01 0000 (Add B - A and store in B)

tmp(24) := "011001";

-- Op Code Dest Operand Instruction

-- 1001 00 0000 (Program End)

tmp(25) := "100100";

tmp(26) := "100100"; -- end

tmp(27) := "100000"; -- begin

tmp(28) := "100100"; -- end

tmp(29) := "100000"; -- begin

tmp(30) := "100100"; -- end

tmp(31) := "100100"; -- end

-- Declare the ROM signal and specify a default value

return tmp;

end init\_rom;

signal rom : memory\_t := init\_rom;

begin

process(clk)

begin

if(rising\_edge(clk)) then

q <= rom(to\_integer(unsigned(addr)));

end if;

end process;

end rtl;

# Reg\_ALU

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;

# Reg Datapath

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;

# Lab 06

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.std\_logic\_arith.all;

use IEEE.STD\_LOGIC\_UNSIGNED.ALL;

use ieee.numeric\_std.all;

entity Lab06 is

Port (

CLK: in std\_logic;

reset: in std\_logic;

PCSrc: in std\_logic;

PAddr: in std\_logic\_vector(4 downto 0);

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

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

Done: out std\_logic);

end Lab06;

architecture structural of Lab06 is

COMPONENT REG\_DATAPATH

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 COMPONENT;

COMPONENT ROM\_32\_6

port(

addr : std\_logic\_vector(4 downto 0);

clk : in std\_logic;

q : out std\_logic\_vector(5 downto 0)

);

end COMPONENT;

COMPONENT PCOUNTER

port(

reset: in std\_logic;

addr : out std\_logic\_vector(4 downto 0);

PAddr: in std\_logic\_vector(4 downto 0);

PCSrc: in std\_logic;

PCload:in std\_logic;

clk : in std\_logic

);

end COMPONENT;

COMPONENT gcpu\_controller

port (

CLK : in std\_logic;

reset : in std\_logic;

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

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

C\_in : in std\_logic;

captureALU : out std\_logic;

PCload : out std\_logic;

C\_set : out std\_logic;

C\_reset : out 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 COMPONENT;

--Inputs

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);

signal CIN : std\_logic;

signal dataout : std\_logic\_vector(3 downto 0);

signal dataoutALU : std\_logic\_vector(3 downto 0);

signal doneint : std\_logic;

signal captureALU : std\_logic;

signal busInzero : std\_logic\_vector(3 downto 0);

signal PCload: std\_logic;

signal q : std\_logic\_vector(5 downto 0);

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

signal instruction : std\_logic\_vector(3 downto 0);

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

signal C\_set : std\_logic;

signal C\_reset : std\_logic;

signal Creg : std\_logic;

begin

CIN <= '0';

DataPrint : process(clk, reset)

begin

if(reset = '1') then

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

Creg <= '0';

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

if (captureALU = '1') then

dataout <= dataoutALU;

end if;

if (C\_set = '1') then

Creg <= '1';

elsif (C\_reset = '1') then

Creg <= '0';

end if;

end if;

end process;

done <= doneint;

BUS\_OUT <= dataout;

busInzero <= BUS\_IN;

TheRegDatapath : REG\_DATAPATH

port map (

CLK => clk , -- : in std\_logic;

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

BUS\_IN => busInzero , -- : in std\_logic\_vector(3 downto 0);

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

SEL1 => SEL1 , -- : in std\_logic\_vector(1 downto 0);

SEL2 => SEL2 , -- : in std\_logic\_vector(1 downto 0);

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

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

OP => OP , -- : in std\_logic\_vector(1 downto 0);

BUS\_OUT => dataoutALU -- : out std\_logic\_vector(3 downto 0)

);

Control: gcpu\_controller

port map (

CLK => clk , -- : in std\_logic;

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

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

destination => destination , -- : in std\_logic\_vector(1 downto 0);

C\_in => Creg , -- : in std\_logic;

captureALU => captureALU , -- : out std\_logic;

C\_set => C\_set , -- : out std\_logic;

C\_reset => C\_reset , -- : out std\_logic;

SEL1 => SEL1 , -- : out std\_logic\_vector(1 downto 0);

SEL2 => SEL2 , -- : out std\_logic\_vector(1 downto 0);

LDA => LDA , -- : out std\_logic;

LDB => LDB , -- : out std\_logic;

PCload => PCload,

OP => OP , -- : out std\_logic\_vector(1 downto 0);

done => doneint -- : out std\_logic

);

Rom : ROM\_32\_6

port map (

addr => addr,

clk => clk,

q => q );

instruction <= q(5 downto 2);

destination <= q(1 downto 0);

ProgramCounter : PCOUNTER

port map (

reset => reset,

addr => addr,

PAddr => PAddr,

PCSrc => PCSrc,

PCload => PCload,

clk => clk );

end structural;