Lab 6

EGCP 381

By Levi Randall

Introduction

Our last lab was a choice. We were given several options of how to improve our 16-bit small processor. Although many options were given, I chose to support jumps within my program. I did this by adding a customized register that modified the previous instruction word register and the PC register. This register sends out a signal when detecting the jump Opcode instruction from instruction memory. It then sends a signal, which removes the last instruction in the PC register, and tells the PC to jump to whatever number in code is given.

Procedure

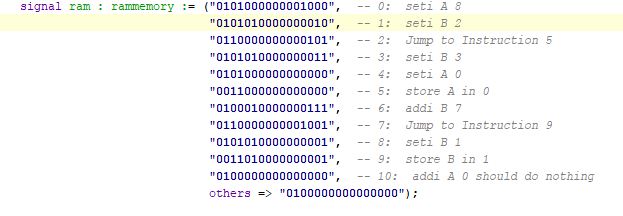
First, I needed to create my custom device that would judge what signal was coming in and send a signal out if it was a jump. I basically created a register with an if statement and an additional output. This if statement watches for the jump opcode and then sends a special signal upon seeing it. The signal it sends is the address to be jumped to with a sort of enable bit in front of it that tells the pc register that the data coming in is an address and not just garbage or zeros. This cancels out the next line of code that should be coming and immediately changes it to the next instruction at the jump coordinates, eliminating the branch delay slot.

A screenshot of a cell phone

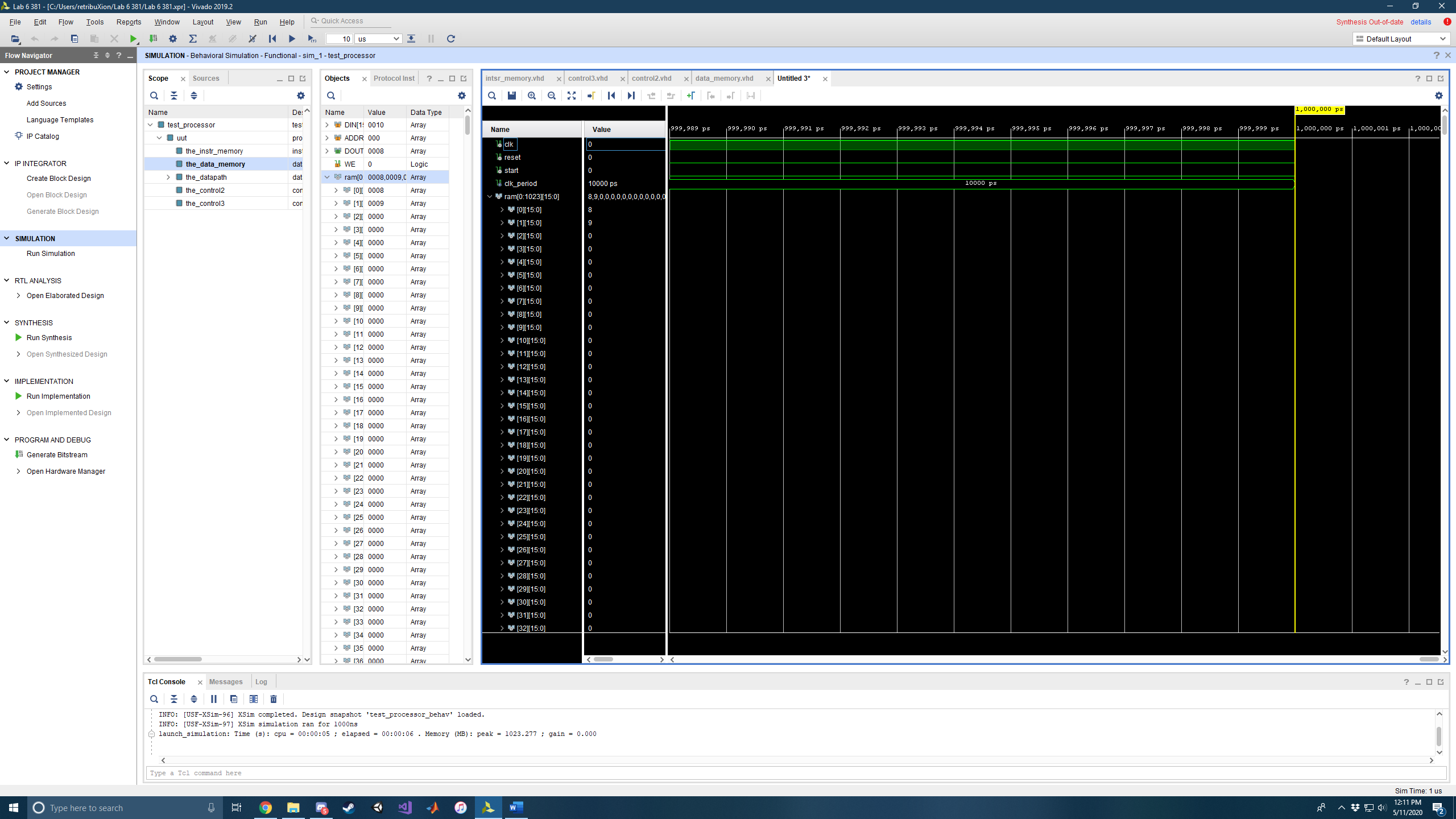
Description automatically generated

A modified datapath with my improvement is shown above.

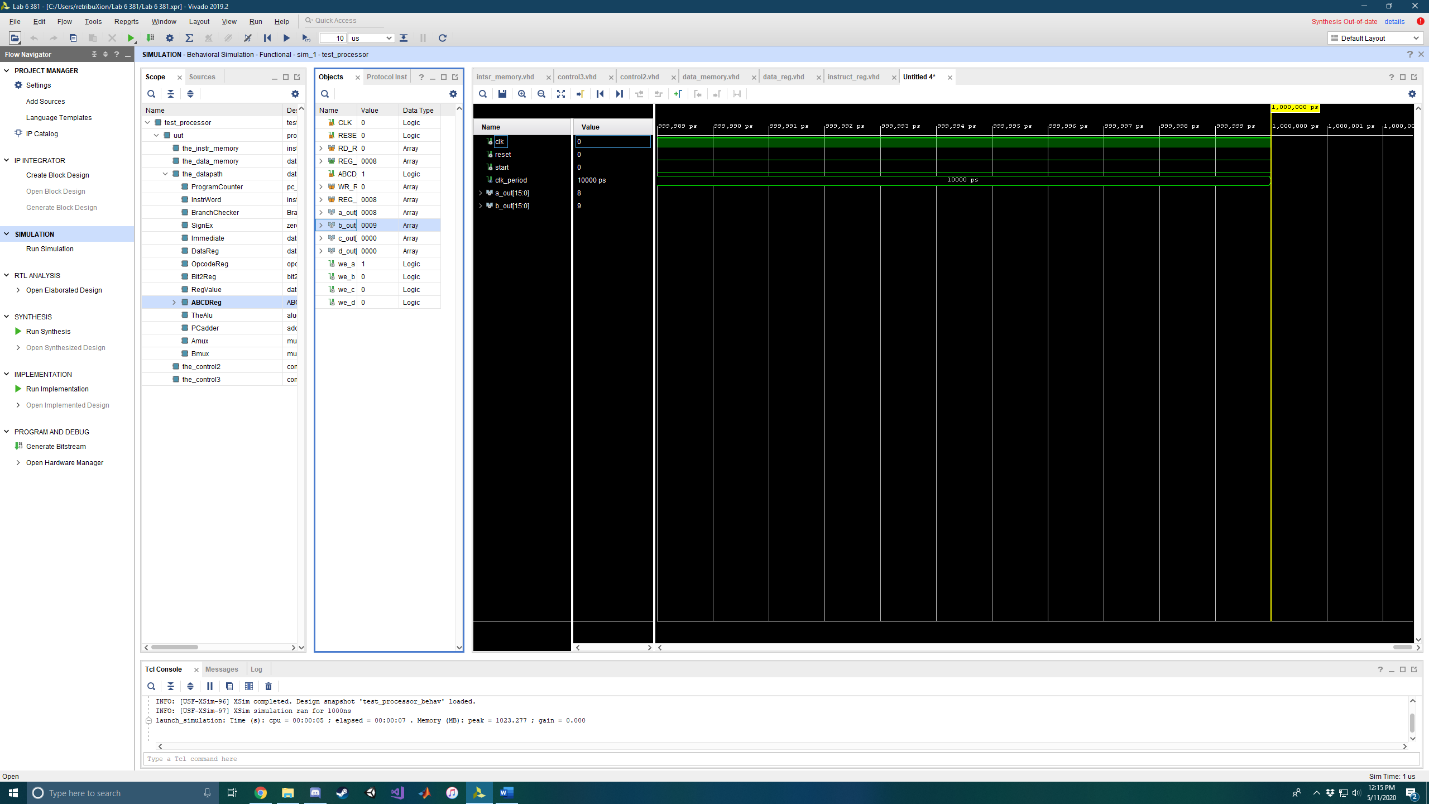
Results



I used this instruction code to test my newly modified processor. Albeit a little complicated, the result of this code should simply be storing the first two numbers A=8 and B=2+7 and store the result in RAM in the first two blocks of bits.



And that is exactly what is stored within DATA memory.



Also, these values are retained within the registers and aren’t lost, since the branch delay is eliminated.

Conclusion

Overall, this lab wasn’t nearly as challenging as the first lab. I only barely modified my data path to support jumps and was able to complete all the tasks asked for. The most difficult piece of this lab was realizing I needed to have the signal for the pc to jump to be sent before the rising edge of the clock so the next instruction didn’t get sent along the pipeline. Aside from that, I had a good time doing this lab because after lab 5, I felt ready for anything.

References

I used James Samawi as a reference and helper with debugging.

Appendix

PC\_REG

library IEEE;

use IEEE.std\_logic\_1164.all;

use work.sm16\_types.all;

-- address\_reg Entity Description

-- Adapted from Dr. Michael Crocker's Spring 2013 CSCE 385 Lab 4 at Pacific Lutheran University

entity pc\_reg is

port( CLK : in std\_logic;

RESET : in std\_logic;

EN : in std\_logic;

D : in sm16\_address;

branch : in std\_logic\_vector(10 downto 0);

Q : out sm16\_address);

end pc\_reg;

-- address\_reg Architecture Description

architecture behavioral of pc\_reg is

begin

RegisterProcess: process( CLK, RESET )

begin

if( RESET = '1' ) then

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

elsif( EN = '1' ) then

if (branch(10) = '1') then

Q <= branch(9 downto 0);

elsif (rising\_edge(CLK)) then

Q <= D;

end if;

end if;

end process RegisterProcess;

end behavioral;

BranchChecker

library IEEE;

use IEEE.std\_logic\_1164.all;

use work.sm16\_types.all;

-- data\_reg Entity Description

-- Adapted from Dr. Michael Crocker's Spring 2013 CSCE 385 Lab 4 at Pacific Lutheran University

entity BranchEval is

port( CLK : in std\_logic;

RESET : in std\_logic;

EN : in std\_logic;

D : in sm16\_data;

branch : out std\_logic\_vector(10 downto 0);

Q : out sm16\_data);

end BranchEval;

-- data\_reg Architecture Description

architecture behavioral of BranchEval is

begin

RegisterProcess: process( CLK, RESET )

begin

if( RESET = '1' ) then

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

elsif( rising\_edge(CLK) ) then

if( EN = '1' ) then

if(D(15 downto 12) = "0110") then

branch(10) <= '1';

branch(9 downto 0) <= D(9 downto 0);

else

branch <= "00000000000";

Q <= D;

end if;

end if;

end if;

end process RegisterProcess;

end behavioral;

Datapath

library IEEE;

use IEEE.std\_logic\_1164.all;

use work.sm16\_types.all;

-- datapath Entity Description

-- Adapted from Dr. Michael Crocker's Spring 2013 CSCE 385 Lab 4 at Pacific Lutheran University

entity datapath is

port( CLK : in std\_logic;

RESET : in std\_logic;

-- I/O with Data Memory

DATA\_IN : out sm16\_data;

DATA\_OUT : in sm16\_data;

DATA\_ADDR : out sm16\_address;

-- I/O with Instruction Memory

INSTR\_OUT : in sm16\_data;

INSTR\_ADDR : out sm16\_address;

-- Control Signals to the ALU

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

B\_INV : in std\_logic;

CIN : in std\_logic;

-- ALU Multiplexer Select Signals

A\_SEL : in std\_logic;

B\_SEL : in std\_logic;

-- Enable Signals for all registers

EN\_PC : in std\_logic;

---------- I ADDED THESE

--OP Signals

OP2 : out std\_logic\_vector (3 downto 0);

OP3 : out std\_logic\_vector (3 downto 0);

-- Enable Signals

EN\_INSTR : in std\_logic;

EN\_IMM : in std\_logic;

EN\_OP : in std\_logic;

EN\_DATA : in std\_logic;

EN\_RV : in std\_logic;

EN\_B2R : in std\_logic;

EN\_ABCD : in std\_logic;

EN\_BRNCH : in std\_logic

);

end datapath;

-- datapath Architecture Description

architecture structural of datapath is

-- declare all components and their ports

component address\_reg is

port( CLK : in std\_logic;

RESET : in std\_logic;

EN : in std\_logic;

D : in sm16\_address;

Q : out sm16\_address);

end component;

component pc\_reg is

port( CLK : in std\_logic;

RESET : in std\_logic;

EN : in std\_logic;

D : in sm16\_address;

branch : in std\_logic\_vector(10 downto 0);

Q : out sm16\_address);

end component;

component BranchEval is

port( CLK : in std\_logic;

RESET : in std\_logic;

EN : in std\_logic;

D : in sm16\_data;

branch : out std\_logic\_vector(10 downto 0);

Q : out sm16\_data);

end component;

component instr\_reg is

port( CLK : in std\_logic;

RESET : in std\_logic;

EN : in std\_logic;

branch : in std\_logic\_vector (10 downto 0);

D : in sm16\_data;

Q : out sm16\_data);

end component;

component data\_reg is

port( CLK : in std\_logic;

RESET : in std\_logic;

EN : in std\_logic;

D : in sm16\_data;

Q : out sm16\_data);

end component;

component alu is

port( A : in sm16\_data;

B : in sm16\_data;

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

D : out sm16\_data;

CIN : in std\_logic;

B\_INV : in std\_logic);

end component;

component adder is

port( A : in sm16\_address;

B : in sm16\_address;

D : out sm16\_address);

end component;

component mux2\_addr is

port( IN0 : in sm16\_address;

IN1 : in sm16\_address;

SEL : in std\_logic;

DOUT : out sm16\_address);

end component;

component mux2\_data is

port( IN0 : in sm16\_data;

IN1 : in sm16\_data;

SEL : in std\_logic;

DOUT : out sm16\_data);

end component;

component zero\_extend is

port(

A: in sm16\_address;

Z: out sm16\_data

);

end component;

----------------------------------------------------

component opcode\_reg is

port( CLK : in std\_logic;

RESET : in std\_logic;

EN : in std\_logic;

D : in sm16\_opcode;

Q : out sm16\_opcode);

end component;

component bit2\_reg is

port( CLK : in std\_logic;

RESET : in std\_logic;

EN : in std\_logic;

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

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

end component;

component ABCDRegFile is

port( CLK : in std\_logic;

RESET : in std\_logic;

RD\_REG : in std\_logic\_vector(1 downto 0); -- Which register to read and output

REG\_OUT : out sm16\_data;

ABCD\_WE : in std\_logic; -- Write enable signal

WR\_REG : in std\_logic\_vector(1 downto 0); -- Which register to write to

REG\_IN : in sm16\_data);

end component;

signal zero\_16 : sm16\_data := "0000000000000000";

signal alu\_a, alu\_b, alu\_out : sm16\_data;

signal pc\_out, pc\_in : sm16\_address;

signal a\_out, immediate\_zero\_extend\_out, brnch2instr : sm16\_data;

signal instrwrdout, signexout,abcdregout, dataregout : std\_logic\_vector (15 downto 0); -- I added this

signal bit2regout : std\_logic\_vector (1 downto 0);

signal pcjump :std\_logic\_vector(10 downto 0);

begin

ProgramCounter: pc\_reg port map (

CLK => CLK,

RESET => RESET,

EN => EN\_PC,

branch => pcjump,

D => pc\_in,

Q => pc\_out

);

--InstrWord: instr\_reg port map(

-- CLK => CLK,

-- RESET => RESET,

-- branch => pcjump,

-- EN => EN\_INSTR,

-- D => INSTR\_OUT,

-- Q => brnch2instr

-- );

BranchChecker: BranchEval port map(

CLK => CLK,

RESET =>RESET,

EN => EN\_BRNCH,

D=> INSTR\_OUT,

--D => brnch2instr,

branch => pcjump,

Q => instrwrdout

);

SignEx: zero\_extend port map(

A=> instrwrdout(9 downto 0),

Z=> signexout

);

Immediate:data\_reg port map(

CLK => CLK,

RESET => RESET,

EN => EN\_IMM,

D => signexout,

Q => immediate\_zero\_extend\_out

);

DataReg:data\_reg port map(

CLK => CLK,

RESET => RESET,

EN => EN\_DATA,

D => DATA\_OUT,

Q => dataregout

);

OpcodeReg:opcode\_reg port map(

CLK => CLK,

RESET => RESET,

EN => EN\_OP,

D => instrwrdout(15 downto 12),

Q => OP3

);

Bit2Reg:bit2\_reg port map(

CLK => CLK,

RESET => RESET,

EN => EN\_B2R,

D => instrwrdout(11 downto 10),

Q => bit2regout

);

RegValue:data\_reg port map(

CLK => CLK,

RESET => RESET,

EN => EN\_RV,

D => abcdregout,

Q => a\_out

);

ABCDReg:ABCDRegFile port map(

CLK => CLK,

RESET => RESET,

RD\_REG => instrwrdout(11 downto 10),

REG\_OUT => abcdregout,

ABCD\_WE => EN\_ABCD,

WR\_REG => bit2regout,

REG\_IN => alu\_out

);

TheAlu: alu port map (

A => alu\_a,

B => alu\_b,

OP => ALU\_OP,

D => alu\_out,

CIN => CIN,

B\_INV => B\_INV

);

PCadder: adder port map (

A => pc\_out,

B => "0000000001",

D => pc\_in

);

Amux: mux2\_data port map (

IN0 => zero\_16, -- 00

IN1 => a\_out, -- 01

SEL => A\_SEL,

DOUT => alu\_a

);

Bmux: mux2\_data port map (

IN0 => dataregout, -- 00

IN1 => immediate\_zero\_extend\_out, -- 01

SEL => B\_SEL,

DOUT => alu\_b

);

OP2 <= instrwrdout(15 downto 12);

DATA\_IN <= abcdregout;

DATA\_ADDR <= instrwrdout(9 downto 0);

INSTR\_ADDR <= pc\_out;

end structural;

Instruction Memory

library IEEE;

use IEEE.std\_logic\_1164.all;

use IEEE.numeric\_std.all;

use work.sm16\_types.all;

-- instr\_memory Entity Description

-- Adapted from Dr. Michael Crocker's Spring 2013 CSCE 385 Lab 6 at Pacific Lutheran University

entity instr\_memory is

port( DIN : in sm16\_data;

ADDR : in sm16\_address;

DOUT : out sm16\_data;

WE : in std\_logic);

end instr\_memory;

-- instr\_memory Architecture Description

architecture behavioral of instr\_memory is subtype ramword is bit\_vector(15 downto 0);

type rammemory is array (0 to 1023) of ramword;

----------------------------------------------

----------------------------------------------

---- This is where you put your program -----

----------------------------------------------

----------------------------------------------

-- add 0000 addi 0100

-- sub 0001 seti 0101

-- load 0010 Jump 0110

-- store 0011

signal ram : rammemory := ("0101000000001000", -- 0: seti A 8

"0101010000000010", -- 1: seti B 2

"0110000000000101", -- 2: Jump to Instruction 5

"0101010000000011", -- 3: seti B 3

"0101000000000000", -- 4: seti A 0

"0011000000000000", -- 5: store A in 0

"0100010000000111", -- 6: addi B 7

"0110000000001001", -- 7: Jump to Instruction 9

"0101010000000001", -- 8: seti B 1

"0011010000000001", -- 9: store B in 1

-- "0100000000000000", -- 10: addi A 0 should do nothing

others => "0100000000000000");

begin

DOUT <= to\_stdlogicvector(ram(to\_integer(unsigned(ADDR))));

ram(to\_integer(unsigned(ADDR))) <= to\_bitvector(DIN) when WE = '1';

end behavioral;