Lab 5

EGCP 381

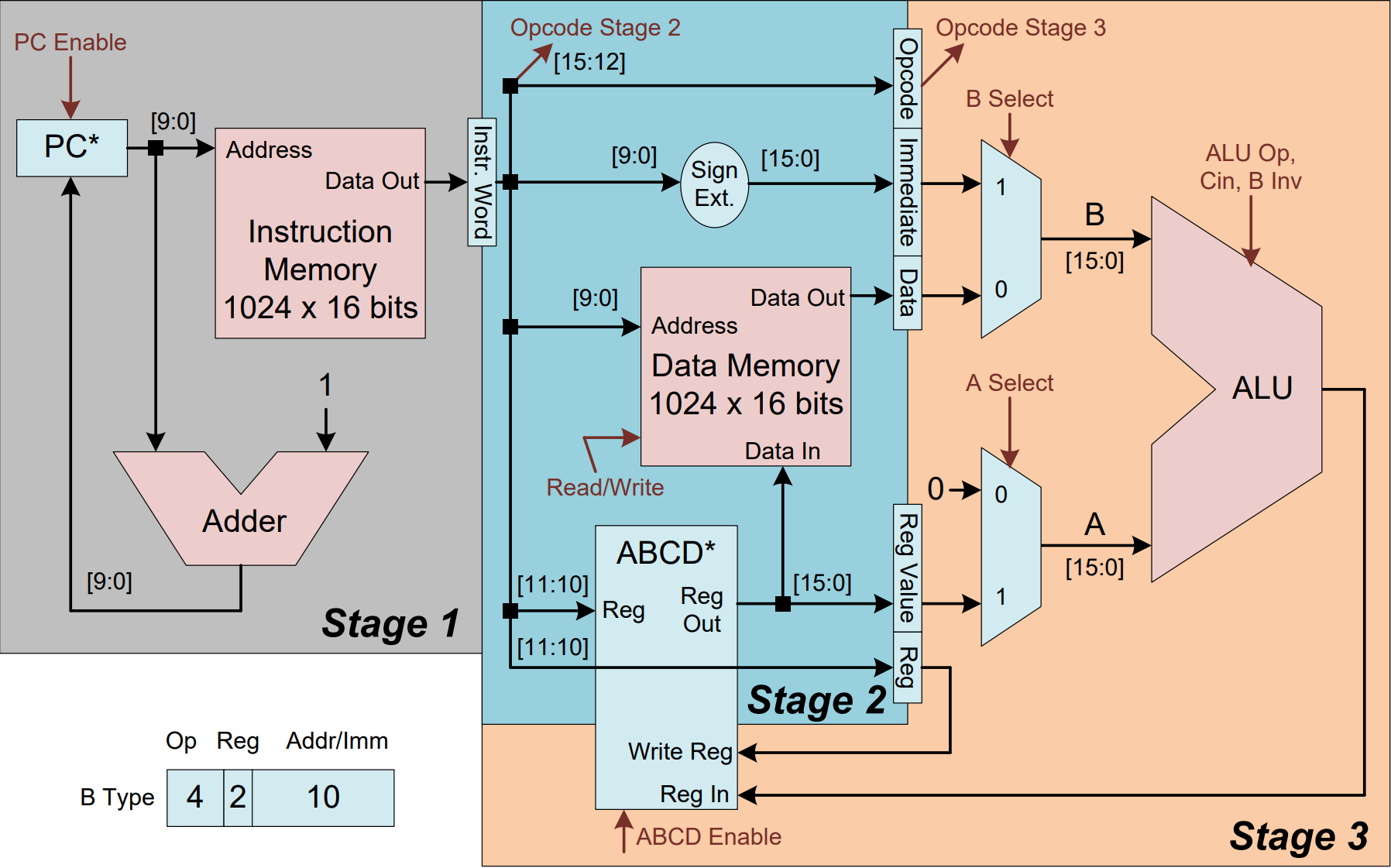
By Levi Randall

Introduction

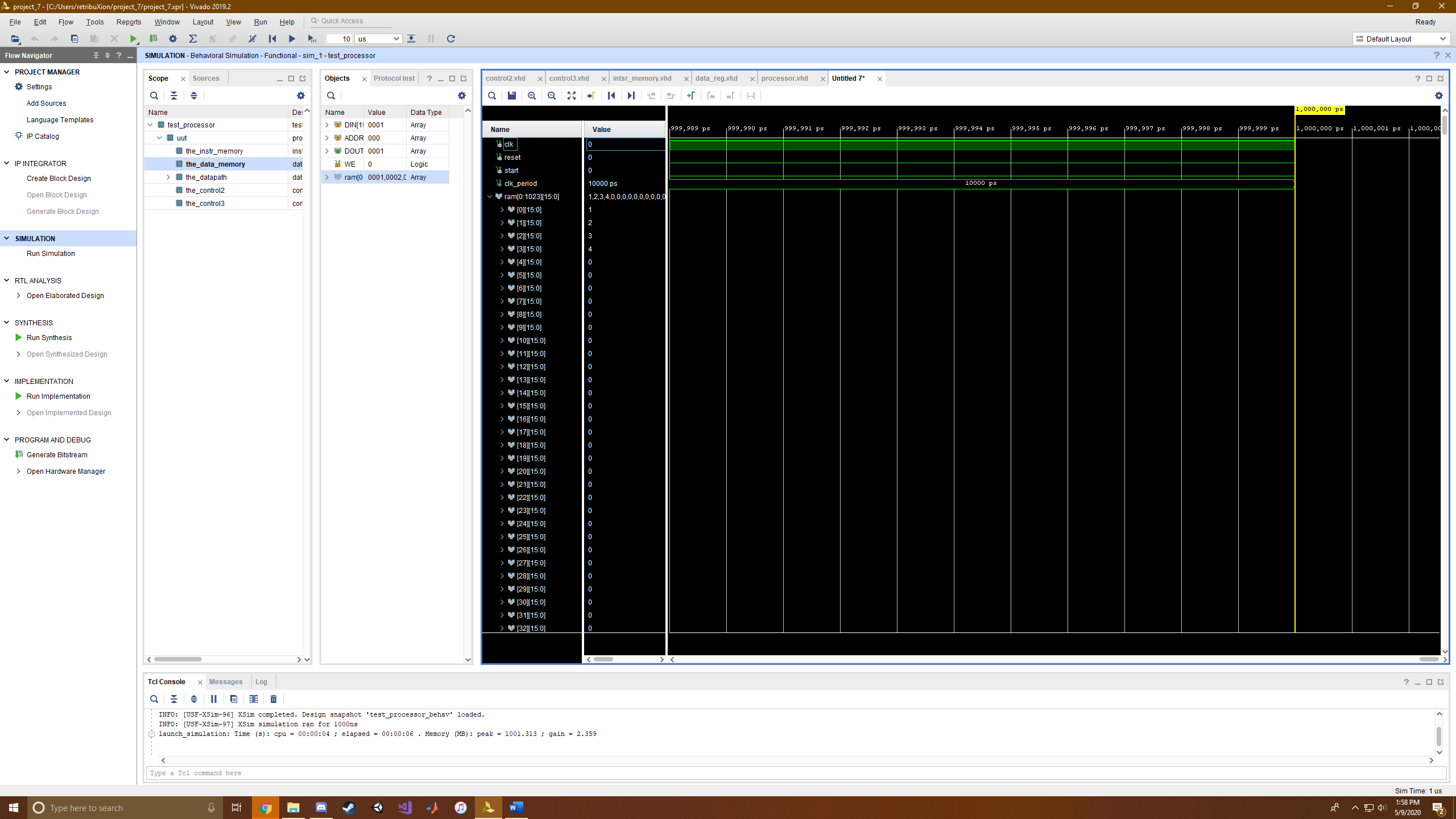
In this lab, we modified our Lab 3 processors to execute instructions in a pipeline. This is done by adding many registers that hold values, including ALU value holding registers, that could store values without needing the slow memory unit. Pipelined processors, can execute fasters much faster, by utilizing all pieces of hardware and reducing downtime of hardware, however, it comes with its own hazards which need to be appropriately addressed.

Procedure

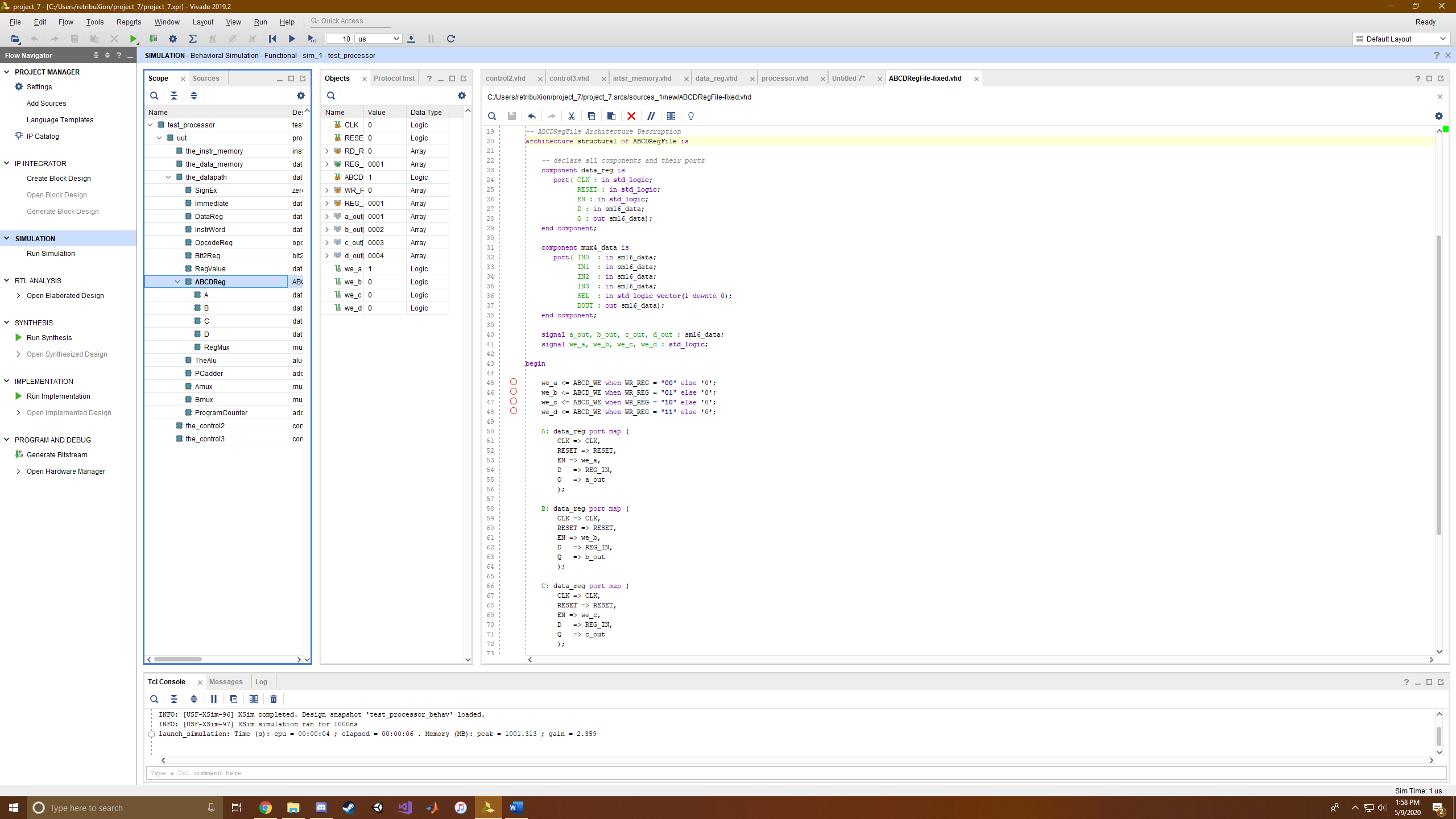
To accomplish this, I modified the Datapath heavily and added two separate control files. I then added many registers on the Datapath and sent their enable signals to a control file. The ABCD register file works as a register that the ALU can call upon as memory. This allows numbers to be stored and then operated on without the need for a memory device.



I then spent a considerable amount of time testing and debugging to create the desired results. I successfully created the desired results and found my design to work properly. All registers carry the correct information and the RAM memory contained the same values after being stored.



Waveform Showing RAM Values



ABCD Registers Showing Reg Values

Hazards

Now to talk about Hazards. This pipeline has a problem with data hazards. An example of this is when a load and store operation are in that order. The load operation takes longer than the store value because it needs to place the value in the register and store does not. This means that the store could potentially change the value before it was loaded. To modify this, I may make the store instruction take the same number of cycles as the load so that store and load go off at the same time.

Also for control hazards, the pipeline circuit will have a potential issue if a branch is taken because the next instruction coming afterwards will be executed regardless of the branch decision. This at minimum needs a branch delay slot or could be solved by finding the results of the branch in an earlier stage.

Conclusion

Hands down this has been the hardest lab so far. I spent upwards of 12 hours on this lab and that is no exaggeration. I spent so much time trying to debug the code and figure out how the ABCD register thing worked. I had encountered many problems with the code. Firstly, I accidently deleted the WE bit for the data memory which ruined everything for a while. I got confused with how to actually pipeline the process then emailed you in regards to how to do it. You replied that the best approach was two separate control files, each pertaining to their own stage. So, I coded that and it worked. The only complications I had were related to the ABCD\_EN signal and which control file it should be in. Also, I found that if the ABCD\_EN signal was not within the beginning start statements, then it would not write to my registers. Now, most notably, the biggest issue I had: In the instruction memory, we are given sample code, however, the section that states “others => “0000000000000000” ruins the end result of your code. It continually adds register A to itself making you believe that the register is faulty in some way. This confused me for about three hours and to fix this all I needed was to change the others from an add to an addi so it would add 0 to itself, or simply “others => “0000000000000000” to “others => “0100000000000000,” within instruction memory. In the end, it was a very hard lab, but it tested me and my VHDL considerably and I am happy I finished it.

References

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

Appendix

Processor

library IEEE;

use IEEE.std\_logic\_1164.all;

use work.sm16\_types.all;

-- processor Entity Description

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

entity processor is

port( CLK : in std\_logic;

RESET : in std\_logic;

START : in std\_logic -- signals to run the processor

);

end processor;

-- processor Architecture Description

architecture structural of processor is

-- declare all components and their ports

component 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;

EN\_PC : in std\_logic;

-- ALU Multiplexer Select Signals

A\_SEL : in std\_logic;

B\_SEL : 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);

end component;

component instr\_memory is

port( DIN : in sm16\_data;

ADDR: in sm16\_address;

DOUT: out sm16\_data;

WE: in std\_logic);

end component;

component data\_memory is

port( DIN : in sm16\_data;

ADDR : in sm16\_address;

DOUT : out sm16\_data;

WE : in std\_logic);

end component;

component control2 is

port( CLK : in std\_logic;

RESET : in std\_logic;

START : in std\_logic;

WE : out std\_logic;

EN\_PC : out std\_logic;

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

-- Enable Signals

EN\_INSTR : out std\_logic;

EN\_IMM : out std\_logic;

EN\_OP : out std\_logic;

EN\_DATA : out std\_logic;

EN\_RV : out std\_logic;

EN\_B2R : out std\_logic);

end component;

component control3 is

port( CLK : in std\_logic;

RESET : in std\_logic;

START : in std\_logic;

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

EN\_ABCD : out std\_logic;

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

B\_INV : out std\_logic;

CIN : out std\_logic;

A\_SEL : out std\_logic;

B\_SEL : out std\_logic);

end component;

signal DataAddress\_Connect, PCAddress\_Connect : sm16\_address;

signal Data\_IntoMem\_Connect : sm16\_data;

signal DataOut\_OutofMem\_Connect, Instruction\_Connect : sm16\_data;

signal ReadWrite\_Connect : std\_logic;

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

signal Binv\_Connect : std\_logic;

signal Cin\_Connect : std\_logic;

signal ASel\_Connect : std\_logic;

signal BSel\_Connect : std\_logic;

signal EnPC\_Connect : std\_logic;

signal EnInstr\_Connect : std\_logic;

signal EnImm\_Connect : std\_logic;

signal EnOP\_Connect : std\_logic;

signal EnData\_Connect : std\_logic;

signal EnRV\_Connect : std\_logic;

signal EnB2R\_Connect : std\_logic;

signal EnABCD\_connect : std\_logic;

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

signal Op3\_Connect : std\_logic\_vector (3 downto 0);

begin

the\_instr\_memory: instr\_memory port map (

DIN => "0000000000000000",

ADDR => PCAddress\_Connect,

DOUT => Instruction\_Connect,

WE => '0' -- always read

);

the\_data\_memory: data\_memory port map (

DIN => Data\_IntoMem\_Connect,

ADDR => DataAddress\_Connect,

DOUT => DataOut\_OutofMem\_Connect,

WE => ReadWrite\_Connect

);

the\_datapath: datapath port map (

CLK => CLK,

RESET => RESET,

DATA\_IN => Data\_IntoMem\_Connect,

DATA\_OUT => DataOut\_OutofMem\_Connect,

DATA\_ADDR => DataAddress\_Connect,

INSTR\_OUT => Instruction\_Connect,

INSTR\_ADDR => PCAddress\_Connect,

ALU\_OP => ALUOp\_Connect,

B\_INV => Binv\_Connect,

CIN => Cin\_Connect,

A\_SEL => ASel\_Connect,

B\_SEL => BSel\_Connect,

EN\_PC => EnPC\_Connect,

--I added these

--OP Signals

OP2 => OP2\_Connect,

OP3 => Op3\_Connect,

-- Enable Signals

EN\_INSTR => EnInstr\_Connect,

EN\_IMM => EnImm\_Connect,

EN\_OP => EnOP\_Connect,

EN\_DATA => EnData\_Connect,

EN\_RV => EnRV\_Connect,

EN\_B2R => EnB2R\_Connect,

EN\_ABCD => EnABCD\_connect

);

the\_control2: control2 port map (

CLK => CLK,

RESET => RESET,

START => START,

WE => ReadWrite\_Connect,

EN\_PC => EnPC\_Connect,

OP2 => OP2\_Connect,

EN\_INSTR => EnInstr\_Connect,

EN\_IMM => EnImm\_Connect,

EN\_OP => EnOP\_Connect,

EN\_DATA => EnData\_Connect,

EN\_RV => EnRV\_Connect,

EN\_B2R => EnB2R\_Connect

);

the\_control3: control3 port map (

CLK => CLK,

RESET => RESET,

START => START,

ALU\_OP => ALUOp\_Connect,

B\_INV => Binv\_Connect,

CIN => Cin\_Connect,

A\_SEL => ASel\_Connect,

B\_SEL => BSel\_Connect,

OP3 => Op3\_Connect,

EN\_ABCD => EnABCD\_connect

);

end structural;

Control 2

library ieee;

use ieee.std\_logic\_1164.all;

use work.sm16\_types.all;

-- control Entity Description

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

entity control2 is

port( CLK : in std\_logic;

RESET : in std\_logic;

START : in std\_logic;

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

-- Enable Signals

WE : out std\_logic;

EN\_PC : out std\_logic;

EN\_INSTR : out std\_logic;

EN\_IMM : out std\_logic;

EN\_OP : out std\_logic;

EN\_DATA : out std\_logic;

EN\_RV : out std\_logic;

EN\_B2R : out std\_logic);

end control2;

-- control Architecture Description

architecture behavorial of control2 is

-- op codes

constant op\_add : sm16\_opcode := "0000";

constant op\_sub : sm16\_opcode := "0001";

constant op\_load : sm16\_opcode := "0010";

constant op\_store : sm16\_opcode := "0011";

constant op\_addi : sm16\_opcode := "0100";

constant op\_seti : sm16\_opcode := "0101";

-- constant op\_jump : sm16\_opcode := "000110";

-- constant op\_jz : sm16\_opcode := "000111";

-- definitions of the states the control can be in

type states is (stopped, running); -- single cycle now, so only one running state

signal state, next\_state : states := stopped;

-- internal write enable, ungated by the clock

signal pre\_we : std\_logic;

begin

-- write enable is gated when the clock is low

-- process to state register

state\_reg: process( CLK, RESET )

begin

if( RESET = '1' ) then

state <= stopped;

elsif( rising\_edge(CLK) ) then

state <= next\_state;

end if;

end process state\_reg;

-- ############################################ --

-- process to define next state transitions and output signals

next\_state\_and\_output: process(state, START, OP2 )

begin

case state is

-- Stopped is the stopped state; wait for start

when stopped =>

if( START /= '1' ) then

-- issue nop

-- Enable Signals

EN\_INSTR <= '1'; EN\_IMM <= '1'; EN\_OP <= '1'; EN\_DATA <= '1';

EN\_RV <= '1'; EN\_B2R <= '1'; EN\_PC <='1';

WE <= '0';

next\_state <= stopped;

else

-- Enable Signals

EN\_INSTR <= '1'; EN\_IMM <= '1'; EN\_OP <= '1'; EN\_DATA <= '1';

EN\_RV <= '1'; EN\_B2R <= '1'; EN\_PC <='1';

WE <= '0';

next\_state <= running; -- go to fetch state

end if;

-- In running state, each instruciton has its own control signals

when running =>

if OP2 = op\_add then

-- A <- A + Mem

EN\_INSTR <= '1'; EN\_IMM <= '1'; EN\_OP <= '1'; EN\_DATA <= '1';

EN\_RV <= '1'; EN\_B2R <= '1'; EN\_PC <='1';

WE <= '0';

next\_state <= running;

elsif OP2 = op\_sub then

-- A <- A - Mem

-- Enable Signals

EN\_INSTR <= '1'; EN\_IMM <= '1'; EN\_OP <= '1'; EN\_DATA <= '1';

EN\_RV <= '1'; EN\_B2R <= '1'; EN\_PC <='1';

WE <= '0';

next\_state <= running;

elsif OP2 = op\_seti then

-- A <- 0 + Immediate

-- Enable Signals

EN\_INSTR <= '1'; EN\_IMM <= '1'; EN\_OP <= '1'; EN\_DATA <= '0';

EN\_RV <= '0'; EN\_B2R <= '1'; EN\_PC <='1';

WE <= '0';

next\_state <= running;

elsif OP2= op\_load then

-- PC <- 0 + Immediate

-- Enable Signals

EN\_INSTR <= '1'; EN\_IMM <= '1'; EN\_OP <= '1'; EN\_DATA <= '1';

EN\_RV <= '1'; EN\_B2R <= '1'; EN\_PC <='1';

WE <= '0';

next\_state <= running;

elsif OP2 = op\_store then

-- PC <- 0 + Immediate

-- Enable Signals

EN\_INSTR <= '1'; EN\_IMM <= '0'; EN\_OP <= '1'; EN\_DATA <= '0';

EN\_RV <= '0'; EN\_B2R <= '0'; EN\_PC <='1';

WE <= '1';

next\_state <= running;

elsif OP2 = op\_addi then

-- PC <- 0 + Immediate

-- Enable Signals

EN\_INSTR <= '1'; EN\_IMM <= '1'; EN\_OP <= '1'; EN\_DATA <= '1';

EN\_RV <= '1'; EN\_B2R <= '1'; EN\_PC <='1';

WE <= '0';

next\_state <= running;

end if;

when others => -- unknown state

-- should never get here, but if it does, set PC<=0 and stop

EN\_PC <='1';

next\_state <= stopped;

end case;

end process next\_state\_and\_output;

end behavorial;

Control 3

library ieee;

use ieee.std\_logic\_1164.all;

use work.sm16\_types.all;

-- control Entity Description

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

entity control3 is

port( CLK : in std\_logic;

RESET : in std\_logic;

START : in std\_logic;

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

EN\_ABCD : out std\_logic;

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

B\_INV : out std\_logic;

CIN : out std\_logic;

A\_SEL : out std\_logic;

B\_SEL : out std\_logic);

end control3;

-- control Architecture Description

architecture behavorial of control3 is

-- control signal values

-- alu operations

constant alu\_nop : std\_logic\_vector(1 downto 0) := "00";

constant alu\_and : std\_logic\_vector(1 downto 0) := "00";

constant alu\_or : std\_logic\_vector(1 downto 0) := "01";

constant alu\_add : std\_logic\_vector(1 downto 0) := "10";

-- a select control

constant a\_0 : std\_logic := '0';

constant a\_a : std\_logic := '1';

-- b select control

constant b\_mem : std\_logic := '0';

constant b\_imm : std\_logic := '1';

-- register load control

constant hold : std\_logic := '0';

constant load : std\_logic := '1';

-- b invert control

constant pos : std\_logic := '0';

constant inv : std\_logic := '1';

-- op codes

constant op\_add : sm16\_opcode := "0000";

constant op\_sub : sm16\_opcode := "0001";

constant op\_load : sm16\_opcode := "0010";

constant op\_store : sm16\_opcode := "0011";

constant op\_addi : sm16\_opcode := "0100";

constant op\_seti : sm16\_opcode := "0101";

-- constant op\_jump : sm16\_opcode := "000110";

-- constant op\_jz : sm16\_opcode := "000111";

-- definitions of the states the control can be in

type states is (stopped, running); -- single cycle now, so only one running state

signal state, next\_state : states := stopped;

begin

-- process to state register

state\_reg: process( CLK, RESET )

begin

if( RESET = '1' ) then

state <= stopped;

elsif( rising\_edge(CLK) ) then

state <= next\_state;

end if;

end process state\_reg;

-- ############################################ --

-- process to define next state transitions and output signals

next\_state\_and\_output: process(state, START, OP3 )

begin

case state is

-- Stopped is the stopped state; wait for start

when stopped =>

if( START /= '1' ) then

-- issue nop

B\_INV <= pos; CIN <= '0'; ALU\_OP <= alu\_nop;

A\_SEL <= a\_0; B\_SEL <= b\_mem; EN\_ABCD <= '0';

next\_state <= stopped;

else

B\_INV <= pos; CIN <= '0'; ALU\_OP <= alu\_and;

A\_SEL <= a\_0; B\_SEL <= b\_mem; EN\_ABCD <= '0';

next\_state <= running; -- go to fetch state

end if;

-- In running state, each instruciton has its own control signals

when running =>

if OP3 = op\_add then

-- A <- A + Mem

B\_INV <= pos; CIN <= '0'; ALU\_OP <= alu\_add;

A\_SEL <= a\_a; B\_SEL <= b\_mem; EN\_ABCD <= '1';

next\_state <= running;

elsif OP3 = op\_sub then

-- A <- A - Mem

B\_INV <= inv; CIN <= '1'; ALU\_OP <= alu\_add; --ADD TOGETHER

A\_SEL <= a\_a; B\_SEL <= b\_mem; EN\_ABCD <= '1';

next\_state <= running;

elsif OP3 = op\_seti then

-- A <- 0 + Immediate

B\_INV <= pos; CIN <= '0'; ALU\_OP <= alu\_add;

A\_SEL <= a\_0; B\_SEL <= b\_imm; EN\_ABCD <= '1';

next\_state <= running;

elsif OP3 = op\_load then

-- PC <- 0 + Immediate

B\_INV <= pos; CIN <= '0'; ALU\_OP <= alu\_add; --DONE

EN\_ABCD <= '1';

A\_SEL <= a\_0; B\_SEL <= b\_mem; --DONE

next\_state <= running;

elsif OP3 = op\_store then

-- PC <- 0 + Immediate

B\_INV <= pos; CIN <= '0'; ALU\_OP <= alu\_nop;

A\_SEL <= a\_a; B\_SEL <= b\_imm; EN\_ABCD <= '0';

next\_state <= running;

elsif OP3 = op\_addi then

-- PC <- 0 + Immediate

B\_INV <= pos; CIN <= '0'; ALU\_OP <= alu\_add;

A\_SEL <= a\_a; B\_SEL <= b\_imm; EN\_ABCD <= '1';

next\_state <= running;

end if;

when others => -- unknown state

-- should never get here, but if it does, set PC<=0 and stop

B\_INV <= pos; CIN <= '0'; ALU\_OP <= alu\_and;

A\_SEL <= a\_0; B\_SEL <= b\_mem; EN\_ABCD <= '0';

next\_state <= stopped;

end case;

end process next\_state\_and\_output;

end behavorial;

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

-- store 0011

signal ram : rammemory := ("0101000000000001", -- 0: seti A 1

"0101010000000010", -- 1: seti B 2

"0101100000000011", -- 2: seti C 3

"0101110000000100", -- 3: seti D 4

"0011000000000000", -- 4: store A 0

"0011010000000001", -- 5: store B 1

"0011100000000010", -- 6: store C 2

"0011110000000011", -- 7: store D 3

others => "0100000000000000");

begin

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

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

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

);

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 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 : sm16\_data;

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

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

begin

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

);

InstrWord:data\_reg port map(

CLK => CLK,

RESET => RESET,

EN => EN\_INSTR,

D => INSTR\_OUT,

Q => instrwrdout

);

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

);

ProgramCounter: address\_reg port map (

CLK => CLK,

RESET => RESET,

EN => EN\_PC,

D => pc\_in,

Q => pc\_out

);

OP2 <= instrwrdout(15 downto 12);

DATA\_IN <= abcdregout;

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

INSTR\_ADDR <= pc\_out;

end structural;