Lab 4

EGCP 381

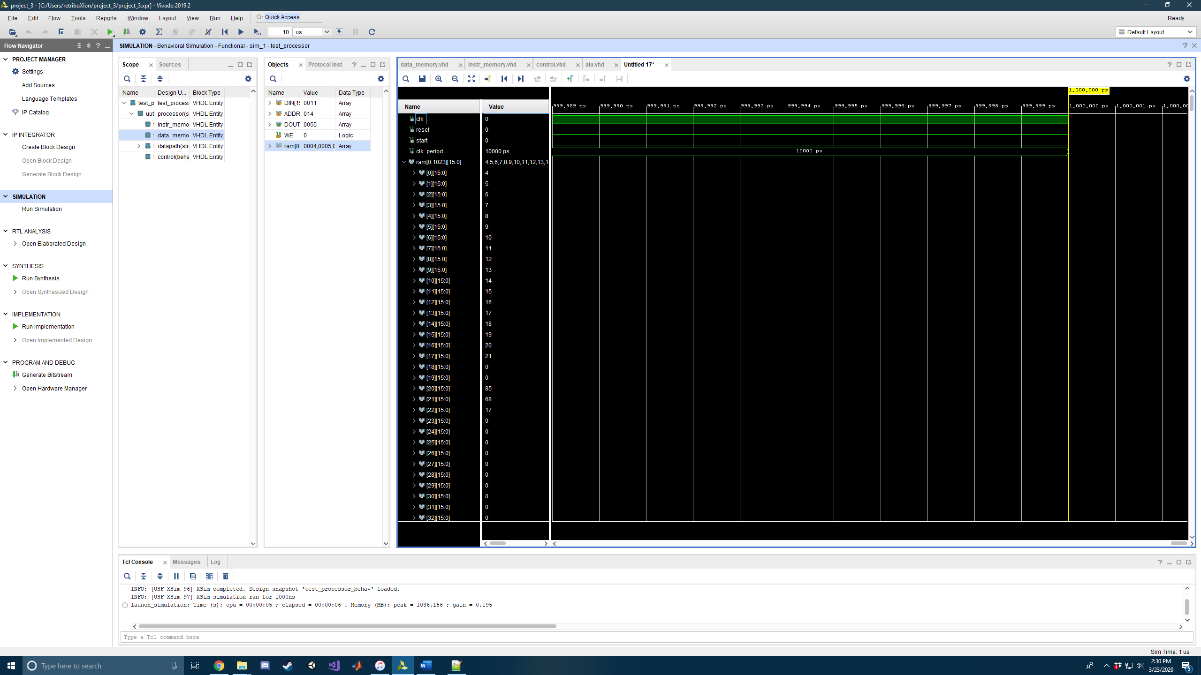
By Levi Randall

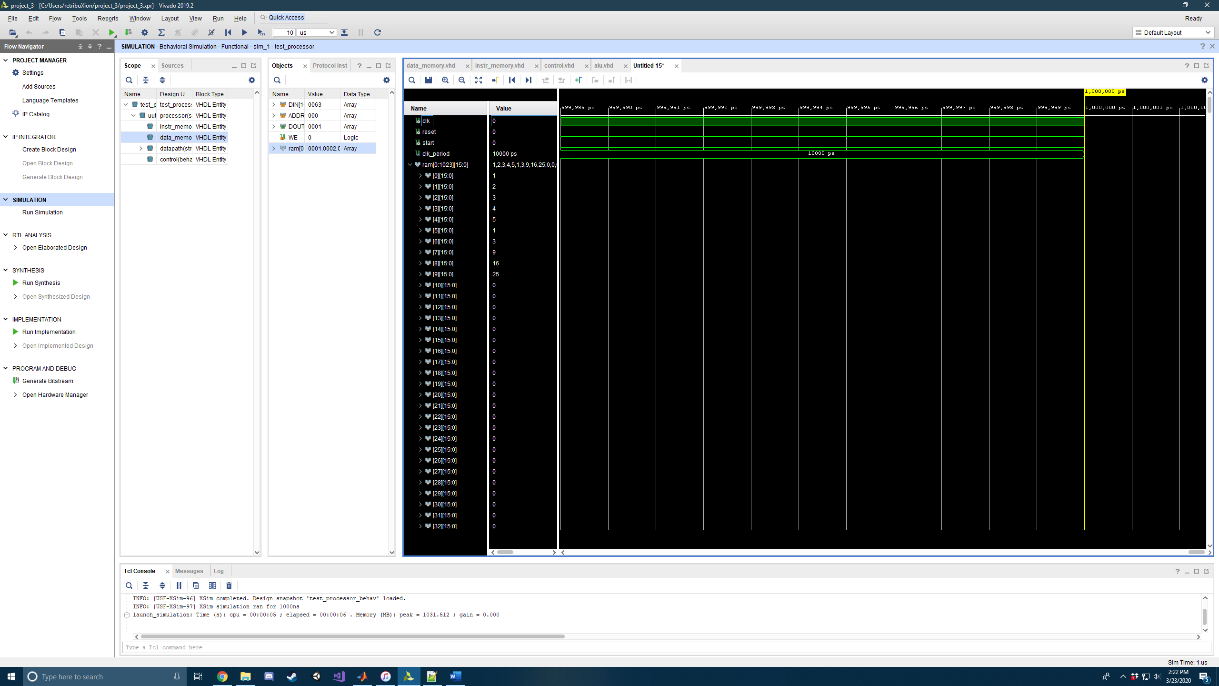
Introduction

Today, we modified the code for the tiny 16 bit processor to reduce the instructions needed to compute the squares of the numbers one through five. The were many ways to accomplish this task with many different approaches. I decided to modify the processor over the data path as it seemed much simpler and realistic than designing a new piece of hardware.

Procedure

I was able to successfully modify the processor in computing a complex task with less instructions. As the data path didn’t change, the original program ran fine, as shown below.

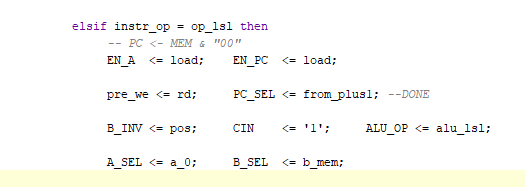


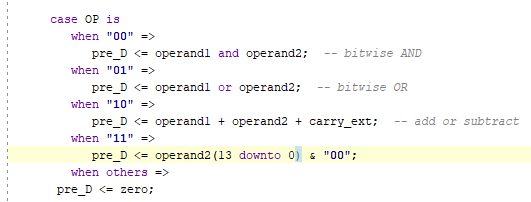
I then computed the sum of squares and was able to compute the sum within 15 instructions down from the pre-altered processor of 20. The picture below shows the wave form and new instruction set compared with the old.



To accomplish this, I added a new opcode that simply shifts the binary word left two bits. I named this code lsl and used the binary string of “111111” to invoke the function. This allowed me to save a few instruction spaces later when computing the higher numbers.

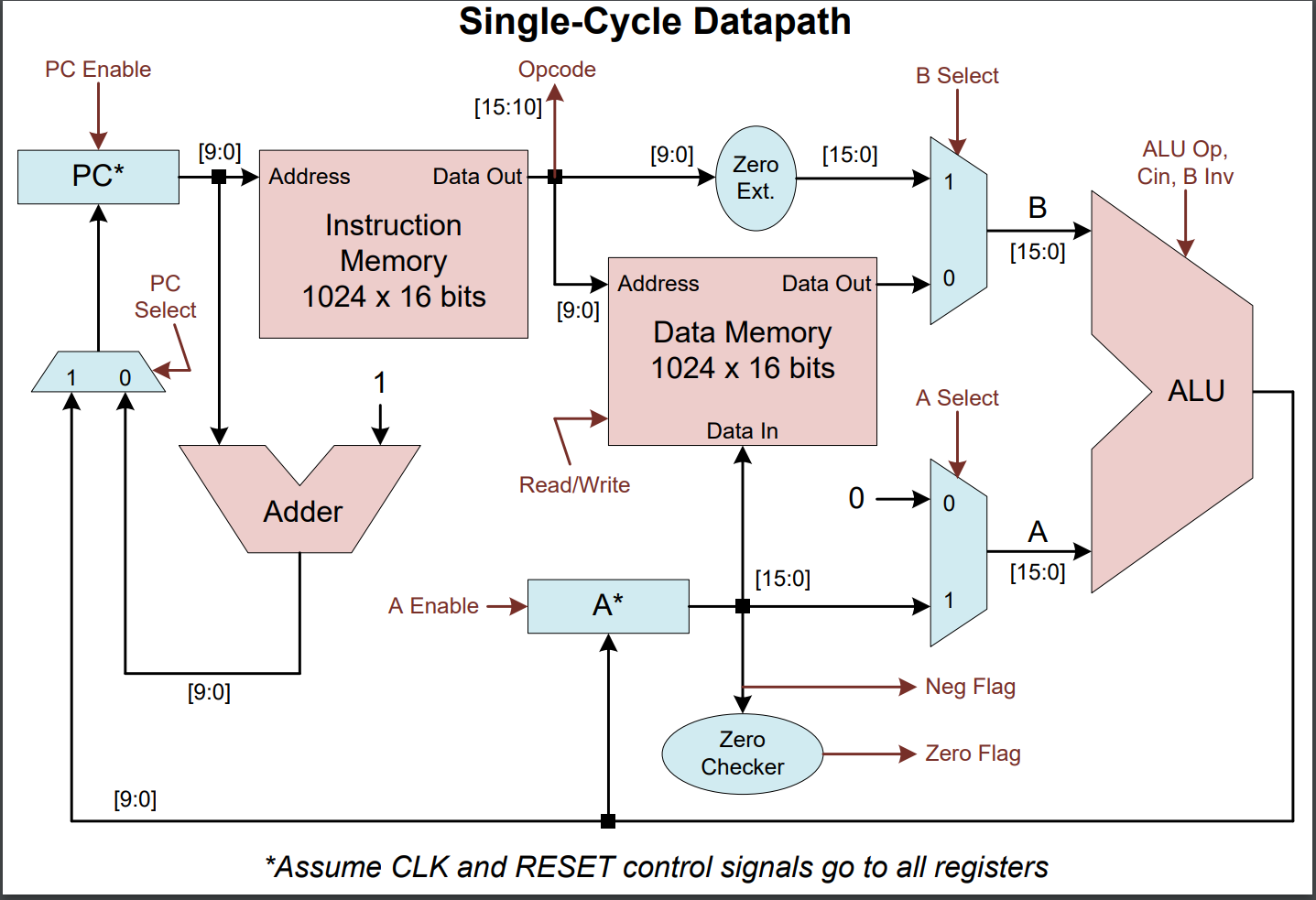
I modified two places to allow the processor to shift bits. First, I went to the control and added all the appropriate signals. I then went to the ALU and added an operation if given the specific “11” code from the OP. To put it simply, the ALU reads the register of whatever value is given and then shifts it two bits.





* **Were the initial program from lab 3 and the squares program easier to code? What are the speedups for each program?**

I felt that after the modifications the new code was much easier to utilize than the older code. The speed up is as follows,



Left Shift Function Added in ALU

Conclusion

Overall, this seemed very difficult, but in practice turned out to be very simple. I had originally planned to design a separate piece of hardware to accomplish this task, however, I realized I could just have the ALU do the shift and would have the same result without overcomplicating things. I had a little bit of trouble with the signals in the control, but worked it out. When originally creating the Lab 3 sums code, I new that multiplication is simply just multiple additions. So, I tackled the problem using that idea, by adding until the square was reached. When solving this lab, I wanted to again use multiplication and found that using a left shift is the same as multiplying and used that to my advantage where I could. The function also only really helped with the bigger numbers. The smaller instructions didn’t benefit from the multiplication as adding was almost as fast. Lastly, if I was to make any other changes, I would create a specific piece of hardware that could change between multiplying by 2, 4, 8, and so on, to better multiply and compute higher numbers. Attached is ALL the code, but I assure you, only the part immediately above this was modified.

References

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

Appendix

library IEEE;

use IEEE.std\_logic\_1164.all;

use IEEE.std\_logic\_signed.all;

use work.sm16\_types.all;

-- alu Entity Description

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

entity alu is

port(

A: in sm16\_data;

B: in sm16\_data;

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

D: out sm16\_data;

CIN,B\_INV: in std\_logic

);

end alu;

-- alu Architecture Description

architecture rtl of alu is

signal pre\_D : sm16\_data;

begin

ArithProcess: process(A,B,CIN,B\_INV,OP)

variable operand1, operand2 : sm16\_data;

variable carry\_ext : std\_logic\_vector(1 downto 0);

constant zero : sm16\_data := "0000000000000000";

begin

carry\_ext := '0' & CIN; -- needed to make CIN positive

operand1 := A;

if B\_INV = '0' then

operand2 := B;

else

operand2 := not B; -- needed for subtract

end if;

case OP is

when "00" =>

pre\_D <= operand1 and operand2; -- bitwise AND

when "01" =>

pre\_D <= operand1 or operand2; -- bitwise OR

when "10" =>

pre\_D <= operand1 + operand2 + carry\_ext; -- add or subtract

when "11" =>

pre\_D <= operand2(13 downto 0) & "00";

when others =>

pre\_D <= zero;

assert false

report "Illegal ALU operation" severity error;

end case;

end process ArithProcess;

D <= pre\_D;

end rtl;

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 control is

port( CLK : in std\_logic;

RESET : in std\_logic;

START : in std\_logic;

WE : 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;

PC\_SEL : out std\_logic;

EN\_A : out std\_logic;

EN\_PC : out std\_logic;

Z\_FLAG : in std\_logic;

N\_FLAG : in std\_logic;

INSTR\_OP : in sm16\_opcode);

end control;

-- control Architecture Description

architecture behavorial of control 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";

constant alu\_lsl : std\_logic\_vector(1 downto 0) := "11";

-- 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';

-- pc select

constant from\_plus1 : std\_logic := '0';

constant from\_alu : std\_logic := '1';

-- register load control

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

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

-- data memory write enable control

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

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

-- b invert control

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

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

-- op codes

constant op\_add : sm16\_opcode := "000000";

constant op\_sub : sm16\_opcode := "000001";

constant op\_load : sm16\_opcode := "000010";

constant op\_store : sm16\_opcode := "000011";

constant op\_addi : sm16\_opcode := "000100";

constant op\_seti : sm16\_opcode := "000101";

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

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

constant op\_lsl : sm16\_opcode := "111111";

-- 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

WE <= pre\_we and (not CLK);

-- 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, INSTR\_OP, Z\_FLAG, N\_FLAG )

begin

case state is

-- Stopped is the stopped state; wait for start

when stopped =>

if( START /= '1' ) then

-- issue nop

EN\_A <= hold; EN\_PC <= hold;

pre\_we <= rd; PC\_SEL <= from\_plus1;

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

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

next\_state <= stopped;

else

EN\_A <= hold; EN\_PC <= hold;

pre\_we <= rd; PC\_SEL <= from\_plus1;

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

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

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

end if;

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

when running =>

if instr\_op = op\_add then

-- A <- A + Mem

EN\_A <= load; EN\_PC <= load;

pre\_we <= rd; PC\_SEL <= from\_plus1;

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

A\_SEL <= a\_a; B\_SEL <= b\_mem;

next\_state <= running;

elsif instr\_op = op\_sub then

-- A <- A - Mem

EN\_A <= load; EN\_PC <= load;

pre\_we <= rd; PC\_SEL <= from\_plus1; -- READ OR WRITE / INCREMENT PROGRAM COUNTER OR JUMP

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

A\_SEL <= a\_a; B\_SEL <= b\_mem;

next\_state <= running;

elsif instr\_op = op\_seti then

-- A <- 0 + Immediate

EN\_A <= load; EN\_PC <= load;

pre\_we <= rd; PC\_SEL <= from\_plus1;

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

A\_SEL <= a\_0; B\_SEL <= b\_imm;

next\_state <= running;

elsif instr\_op = op\_jump then

-- PC <- 0 + Immediate

EN\_A <= hold; EN\_PC <= load;

pre\_we <= rd; PC\_SEL <= from\_alu;

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

A\_SEL <= a\_0; B\_SEL <= b\_imm;

next\_state <= running;

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

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

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

elsif instr\_op = op\_load then

-- PC <- 0 + Immediate

EN\_A <= load; EN\_PC <= load;

pre\_we <= rd; PC\_SEL <= from\_plus1; --DONE

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

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

next\_state <= running;

elsif instr\_op = op\_store then

-- PC <- 0 + Immediate

EN\_A <= hold; EN\_PC <= load;

pre\_we <= wr; PC\_SEL <= from\_plus1; --DONE

B\_INV <= pos; CIN <= '1'; ALU\_OP <= alu\_add;

A\_SEL <= a\_a; B\_SEL <= b\_mem;

next\_state <= running;

elsif instr\_op = op\_addi then

-- PC <- 0 + Immediate

EN\_A <= load; EN\_PC <= load;

pre\_we <= rd; PC\_SEL <= from\_plus1; --DONE

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

A\_SEL <= a\_a; B\_SEL <= b\_imm;

next\_state <= running;

elsif instr\_op = op\_lsl then

-- PC <- MEM & "00"

EN\_A <= load; EN\_PC <= load;

pre\_we <= rd; PC\_SEL <= from\_plus1; --DONE

B\_INV <= pos; CIN <= '1'; ALU\_OP <= alu\_lsl;

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

next\_state <= running;

elsif instr\_op = op\_jz then

-- Because the zero flag comes directly from the A register through the

-- zero checker component (not from the ALU), the control signals do not

-- affect the outcome of the check. Therefore, both conditions of the

-- jump can evaluated in the one cycle for the instruction.

if z\_flag = '1' then

-- successful jump

-- PC <- 0 + Immediate

EN\_A <= hold; EN\_PC <= load;

pre\_we <= rd; PC\_SEL <= from\_alu;

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

A\_SEL <= a\_0; B\_SEL <= b\_imm;

next\_state <= running;

else

-- unsuccessful jump

-- PC <- PC + 1 (as normal)

EN\_A <= hold; EN\_PC <= load;

pre\_we <= rd; PC\_SEL <= from\_plus1;

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

A\_SEL <= a\_0; B\_SEL <= b\_imm;

next\_state <= running;

end if;

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

else -- unknown opcode

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

EN\_A <= hold; EN\_PC <= load;

pre\_we <= rd; PC\_SEL <= from\_plus1;

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

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

next\_state <= stopped;

end if;

when others => -- unknown state

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

EN\_A <= hold; EN\_PC <= load;

pre\_we <= rd; PC\_SEL <= from\_plus1;

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

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

next\_state <= stopped;

end case;

end process next\_state\_and\_output;

end behavorial;

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 4 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 000000 addi 000100

-- sub 000001 seti 000101

-- load 000010 jump 000110

-- store 000011 jz 000111

-- lsl 111111

signal ram : rammemory := ("0001000000000001", -- 0:

"0000110000000101", -- 1:

"0001000000000010", -- 2:

"0000110000000110", -- 3:

"0000100000000010", -- 4:

"0000000000000010", -- 5:

"0000000000000010", -- 6:

"0000110000000111", -- 7:

"0000100000000011", -- 8:

"1111110000000011", -- 9:

"0000110000001000", -- 10:

"0000100000000100", -- 11:

"1111110000000100", -- 12:

"0000000000000100", -- 13:

"0000110000001001", -- 14:

"0000000000000000", -- 15:

"0000000000000000", -- 16:

"0000000000000000", -- 17:

"0000000000000000", -- 18:

"0000000000000000", -- 19:

"0000000000000000", -- 20:

others => "0000000000000000");

begin

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

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

end behavioral;

library IEEE;

use IEEE.std\_logic\_1164.all;

use IEEE.numeric\_std.all;

use work.sm16\_types.all;

-- data\_memory Entity Description

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

entity data\_memory is

port( DIN : in sm16\_data;

ADDR : in sm16\_address;

DOUT : out sm16\_data;

WE : in std\_logic);

end data\_memory;

-- data\_memory Architecture Description

architecture behavioral of data\_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 data -------

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

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

signal ram : rammemory := ("0000000000000001", -- 0: array[0]=1

"0000000000000010", -- 1: array[1]=2

"0000000000000011", -- 2: array[2]=3

"0000000000000100", -- 3: array[3]=4

"0000000000000101", -- 4: array[4]=5

"0000000000000000", -- 5: array[5]=0

"0000000000000000", -- 6: array[6]=0

"0000000000000000", -- 7: array[7]=0

"0000000000000000", -- 8: array[8]=0

"0000000000000000", -- 9: array[9]=0

"0000000000000000", -- 10: array[10]=0

"0000000000000000", -- 11: array[11]=0

"0000000000000000", -- 12: array[12]=0

"0000000000000000", -- 13: array[13]=0

"0000000000000000", -- 14: array[14]=0

"0000000000000000", -- 15: array[15]=0

"0000000000000000", -- 16: array[16]=0

"0000000000000000", -- 17: array[17]=0

others => "0000000000000000");

begin

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

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

end behavioral;