library ieee;

use ieee.std\_logic\_1164.all;

use ieee.numeric\_std.all;

entity my\_matrix\_5\_v1\_0\_S00\_AXI is

generic (

-- Users to add parameters here

-- User parameters ends

-- Do not modify the parameters beyond this line

-- Width of S\_AXI data bus

C\_S\_AXI\_DATA\_WIDTH : integer := 32;

-- Width of S\_AXI address bus

C\_S\_AXI\_ADDR\_WIDTH : integer := 6

);

port (

-- Users to add ports here

-- User ports ends

-- Do not modify the ports beyond this line

-- Global Clock Signal

S\_AXI\_ACLK : in std\_logic;

-- Global Reset Signal. This Signal is Active LOW

S\_AXI\_ARESETN : in std\_logic;

-- Write address (issued by master, acceped by Slave)

S\_AXI\_AWADDR : in std\_logic\_vector(C\_S\_AXI\_ADDR\_WIDTH-1 downto 0);

-- Write channel Protection type. This signal indicates the

-- privilege and security level of the transaction, and whether

-- the transaction is a data access or an instruction access.

S\_AXI\_AWPROT : in std\_logic\_vector(2 downto 0);

-- Write address valid. This signal indicates that the master signaling

-- valid write address and control information.

S\_AXI\_AWVALID : in std\_logic;

-- Write address ready. This signal indicates that the slave is ready

-- to accept an address and associated control signals.

S\_AXI\_AWREADY : out std\_logic;

-- Write data (issued by master, acceped by Slave)

S\_AXI\_WDATA : in std\_logic\_vector(C\_S\_AXI\_DATA\_WIDTH-1 downto 0);

-- Write strobes. This signal indicates which byte lanes hold

-- valid data. There is one write strobe bit for each eight

-- bits of the write data bus.

S\_AXI\_WSTRB : in std\_logic\_vector((C\_S\_AXI\_DATA\_WIDTH/8)-1 downto 0);

-- Write valid. This signal indicates that valid write

-- data and strobes are available.

S\_AXI\_WVALID : in std\_logic;

-- Write ready. This signal indicates that the slave

-- can accept the write data.

S\_AXI\_WREADY : out std\_logic;

-- Write response. This signal indicates the status

-- of the write transaction.

S\_AXI\_BRESP : out std\_logic\_vector(1 downto 0);

-- Write response valid. This signal indicates that the channel

-- is signaling a valid write response.

S\_AXI\_BVALID : out std\_logic;

-- Response ready. This signal indicates that the master

-- can accept a write response.

S\_AXI\_BREADY : in std\_logic;

-- Read address (issued by master, acceped by Slave)

S\_AXI\_ARADDR : in std\_logic\_vector(C\_S\_AXI\_ADDR\_WIDTH-1 downto 0);

-- Protection type. This signal indicates the privilege

-- and security level of the transaction, and whether the

-- transaction is a data access or an instruction access.

S\_AXI\_ARPROT : in std\_logic\_vector(2 downto 0);

-- Read address valid. This signal indicates that the channel

-- is signaling valid read address and control information.

S\_AXI\_ARVALID : in std\_logic;

-- Read address ready. This signal indicates that the slave is

-- ready to accept an address and associated control signals.

S\_AXI\_ARREADY : out std\_logic;

-- Read data (issued by slave)

S\_AXI\_RDATA : out std\_logic\_vector(C\_S\_AXI\_DATA\_WIDTH-1 downto 0);

-- Read response. This signal indicates the status of the

-- read transfer.

S\_AXI\_RRESP : out std\_logic\_vector(1 downto 0);

-- Read valid. This signal indicates that the channel is

-- signaling the required read data.

S\_AXI\_RVALID : out std\_logic;

-- Read ready. This signal indicates that the master can

-- accept the read data and response information.

S\_AXI\_RREADY : in std\_logic

);

end my\_matrix\_5\_v1\_0\_S00\_AXI;

architecture arch\_imp of my\_matrix\_5\_v1\_0\_S00\_AXI is

-- AXI4LITE signals

signal axi\_awaddr : std\_logic\_vector(C\_S\_AXI\_ADDR\_WIDTH-1 downto 0);

signal axi\_awready : std\_logic;

signal axi\_wready : std\_logic;

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

signal axi\_bvalid : std\_logic;

signal axi\_araddr : std\_logic\_vector(C\_S\_AXI\_ADDR\_WIDTH-1 downto 0);

signal axi\_arready : std\_logic;

signal axi\_rdata : std\_logic\_vector(C\_S\_AXI\_DATA\_WIDTH-1 downto 0);

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

signal axi\_rvalid : std\_logic;

-- Example-specific design signals

-- local parameter for addressing 32 bit / 64 bit C\_S\_AXI\_DATA\_WIDTH

-- ADDR\_LSB is used for addressing 32/64 bit registers/memories

-- ADDR\_LSB = 2 for 32 bits (n downto 2)

-- ADDR\_LSB = 3 for 64 bits (n downto 3)

constant ADDR\_LSB : integer := (C\_S\_AXI\_DATA\_WIDTH/32)+ 1;

constant OPT\_MEM\_ADDR\_BITS : integer := 3;

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

---- Signals for user logic register space example

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

---- Number of Slave Registers 11

signal slv\_reg0 :std\_logic\_vector(C\_S\_AXI\_DATA\_WIDTH-1 downto 0);

signal slv\_reg1 :std\_logic\_vector(C\_S\_AXI\_DATA\_WIDTH-1 downto 0);

signal slv\_reg2 :std\_logic\_vector(C\_S\_AXI\_DATA\_WIDTH-1 downto 0);

signal slv\_reg3 :std\_logic\_vector(C\_S\_AXI\_DATA\_WIDTH-1 downto 0);

signal slv\_reg4 :std\_logic\_vector(C\_S\_AXI\_DATA\_WIDTH-1 downto 0);

signal slv\_reg5 :std\_logic\_vector(C\_S\_AXI\_DATA\_WIDTH-1 downto 0);

signal slv\_reg6 :std\_logic\_vector(C\_S\_AXI\_DATA\_WIDTH-1 downto 0);

signal slv\_reg7 :std\_logic\_vector(C\_S\_AXI\_DATA\_WIDTH-1 downto 0);

signal slv\_reg8 :std\_logic\_vector(C\_S\_AXI\_DATA\_WIDTH-1 downto 0);

signal slv\_reg9 :std\_logic\_vector(C\_S\_AXI\_DATA\_WIDTH-1 downto 0);

signal slv\_reg10 :std\_logic\_vector(C\_S\_AXI\_DATA\_WIDTH-1 downto 0);

signal slv\_reg\_rden : std\_logic;

signal slv\_reg\_wren : std\_logic;

signal reg\_data\_out :std\_logic\_vector(C\_S\_AXI\_DATA\_WIDTH-1 downto 0);

signal byte\_index : integer;

signal aw\_en : std\_logic;

signal val\_c: std\_logic\_vector(31 downto 0);

signal state: std\_logic\_vector(31 downto 0);

signal add\_a: std\_logic\_vector(31 downto 0);

component matrix\_mu\_datapath is

Port (

clk: IN std\_logic;

go: In std\_logic;

addr\_a\_in: IN std\_logic\_vector ( 13 downto 0);

addr\_b\_in: IN std\_logic\_vector ( 13 downto 0);

addr\_c\_in: IN std\_logic\_vector ( 13 downto 0);

val\_a\_in: IN std\_logic\_vector ( 15 downto 0);

val\_b\_in: IN std\_logic\_vector ( 15 downto 0);

val\_c\_out: OUT std\_logic\_vector ( 31 downto 0);

state\_out: OUT std\_logic\_vector (2 downto 0);

w\_e\_a: IN std\_logic;

w\_e\_b: IN std\_logic;

add\_a\_check: OUT std\_logic\_vector ( 13 downto 0)

);

end component;

begin

-- I/O Connections assignments

S\_AXI\_AWREADY <= axi\_awready;

S\_AXI\_WREADY <= axi\_wready;

S\_AXI\_BRESP <= axi\_bresp;

S\_AXI\_BVALID <= axi\_bvalid;

S\_AXI\_ARREADY <= axi\_arready;

S\_AXI\_RDATA <= axi\_rdata;

S\_AXI\_RRESP <= axi\_rresp;

S\_AXI\_RVALID <= axi\_rvalid;

-- Implement axi\_awready generation

-- axi\_awready is asserted for one S\_AXI\_ACLK clock cycle when both

-- S\_AXI\_AWVALID and S\_AXI\_WVALID are asserted. axi\_awready is

-- de-asserted when reset is low.

process (S\_AXI\_ACLK)

begin

if rising\_edge(S\_AXI\_ACLK) then

if S\_AXI\_ARESETN = '0' then

axi\_awready <= '0';

aw\_en <= '1';

else

if (axi\_awready = '0' and S\_AXI\_AWVALID = '1' and S\_AXI\_WVALID = '1' and aw\_en = '1') then

-- slave is ready to accept write address when

-- there is a valid write address and write data

-- on the write address and data bus. This design

-- expects no outstanding transactions.

axi\_awready <= '1';

aw\_en <= '0';

elsif (S\_AXI\_BREADY = '1' and axi\_bvalid = '1') then

aw\_en <= '1';

axi\_awready <= '0';

else

axi\_awready <= '0';

end if;

end if;

end if;

end process;

-- Implement axi\_awaddr latching

-- This process is used to latch the address when both

-- S\_AXI\_AWVALID and S\_AXI\_WVALID are valid.

process (S\_AXI\_ACLK)

begin

if rising\_edge(S\_AXI\_ACLK) then

if S\_AXI\_ARESETN = '0' then

axi\_awaddr <= (others => '0');

else

if (axi\_awready = '0' and S\_AXI\_AWVALID = '1' and S\_AXI\_WVALID = '1' and aw\_en = '1') then

-- Write Address latching

axi\_awaddr <= S\_AXI\_AWADDR;

end if;

end if;

end if;

end process;

-- Implement axi\_wready generation

-- axi\_wready is asserted for one S\_AXI\_ACLK clock cycle when both

-- S\_AXI\_AWVALID and S\_AXI\_WVALID are asserted. axi\_wready is

-- de-asserted when reset is low.

process (S\_AXI\_ACLK)

begin

if rising\_edge(S\_AXI\_ACLK) then

if S\_AXI\_ARESETN = '0' then

axi\_wready <= '0';

else

if (axi\_wready = '0' and S\_AXI\_WVALID = '1' and S\_AXI\_AWVALID = '1' and aw\_en = '1') then

-- slave is ready to accept write data when

-- there is a valid write address and write data

-- on the write address and data bus. This design

-- expects no outstanding transactions.

axi\_wready <= '1';

else

axi\_wready <= '0';

end if;

end if;

end if;

end process;

-- Implement memory mapped register select and write logic generation

-- The write data is accepted and written to memory mapped registers when

-- axi\_awready, S\_AXI\_WVALID, axi\_wready and S\_AXI\_WVALID are asserted. Write strobes are used to

-- select byte enables of slave registers while writing.

-- These registers are cleared when reset (active low) is applied.

-- Slave register write enable is asserted when valid address and data are available

-- and the slave is ready to accept the write address and write data.

slv\_reg\_wren <= axi\_wready and S\_AXI\_WVALID and axi\_awready and S\_AXI\_AWVALID ;

process (S\_AXI\_ACLK)

variable loc\_addr :std\_logic\_vector(OPT\_MEM\_ADDR\_BITS downto 0);

begin

if rising\_edge(S\_AXI\_ACLK) then

if S\_AXI\_ARESETN = '0' then

slv\_reg0 <= (others => '0');

slv\_reg1 <= (others => '0');

slv\_reg2 <= (others => '0');

slv\_reg3 <= (others => '0');

slv\_reg4 <= (others => '0');

slv\_reg5 <= (others => '0');

slv\_reg6 <= (others => '0');

slv\_reg7 <= (others => '0');

slv\_reg8 <= (others => '0');

slv\_reg9 <= (others => '0');

slv\_reg10 <= (others => '0');

else

loc\_addr := axi\_awaddr(ADDR\_LSB + OPT\_MEM\_ADDR\_BITS downto ADDR\_LSB);

if (slv\_reg\_wren = '1') then

case loc\_addr is

when b"0000" =>

for byte\_index in 0 to (C\_S\_AXI\_DATA\_WIDTH/8-1) loop

if ( S\_AXI\_WSTRB(byte\_index) = '1' ) then

-- Respective byte enables are asserted as per write strobes

-- slave registor 0

slv\_reg0(byte\_index\*8+7 downto byte\_index\*8) <= S\_AXI\_WDATA(byte\_index\*8+7 downto byte\_index\*8);

end if;

end loop;

when b"0001" =>

for byte\_index in 0 to (C\_S\_AXI\_DATA\_WIDTH/8-1) loop

if ( S\_AXI\_WSTRB(byte\_index) = '1' ) then

-- Respective byte enables are asserted as per write strobes

-- slave registor 1

slv\_reg1(byte\_index\*8+7 downto byte\_index\*8) <= S\_AXI\_WDATA(byte\_index\*8+7 downto byte\_index\*8);

end if;

end loop;

when b"0010" =>

for byte\_index in 0 to (C\_S\_AXI\_DATA\_WIDTH/8-1) loop

if ( S\_AXI\_WSTRB(byte\_index) = '1' ) then

-- Respective byte enables are asserted as per write strobes

-- slave registor 2

slv\_reg2(byte\_index\*8+7 downto byte\_index\*8) <= S\_AXI\_WDATA(byte\_index\*8+7 downto byte\_index\*8);

end if;

end loop;

when b"0011" =>

for byte\_index in 0 to (C\_S\_AXI\_DATA\_WIDTH/8-1) loop

if ( S\_AXI\_WSTRB(byte\_index) = '1' ) then

-- Respective byte enables are asserted as per write strobes

-- slave registor 3

slv\_reg3(byte\_index\*8+7 downto byte\_index\*8) <= S\_AXI\_WDATA(byte\_index\*8+7 downto byte\_index\*8);

end if;

end loop;

when b"0100" =>

for byte\_index in 0 to (C\_S\_AXI\_DATA\_WIDTH/8-1) loop

if ( S\_AXI\_WSTRB(byte\_index) = '1' ) then

-- Respective byte enables are asserted as per write strobes

-- slave registor 4

slv\_reg4(byte\_index\*8+7 downto byte\_index\*8) <= S\_AXI\_WDATA(byte\_index\*8+7 downto byte\_index\*8);

end if;

end loop;

when b"0101" =>

for byte\_index in 0 to (C\_S\_AXI\_DATA\_WIDTH/8-1) loop

if ( S\_AXI\_WSTRB(byte\_index) = '1' ) then

-- Respective byte enables are asserted as per write strobes

-- slave registor 5

slv\_reg5(byte\_index\*8+7 downto byte\_index\*8) <= S\_AXI\_WDATA(byte\_index\*8+7 downto byte\_index\*8);

end if;

end loop;

when b"0110" =>

for byte\_index in 0 to (C\_S\_AXI\_DATA\_WIDTH/8-1) loop

if ( S\_AXI\_WSTRB(byte\_index) = '1' ) then

-- Respective byte enables are asserted as per write strobes

-- slave registor 6

slv\_reg6(byte\_index\*8+7 downto byte\_index\*8) <= S\_AXI\_WDATA(byte\_index\*8+7 downto byte\_index\*8);

end if;

end loop;

when b"0111" =>

for byte\_index in 0 to (C\_S\_AXI\_DATA\_WIDTH/8-1) loop

if ( S\_AXI\_WSTRB(byte\_index) = '1' ) then

-- Respective byte enables are asserted as per write strobes

-- slave registor 7

slv\_reg7(byte\_index\*8+7 downto byte\_index\*8) <= S\_AXI\_WDATA(byte\_index\*8+7 downto byte\_index\*8);

end if;

end loop;

when b"1000" =>

for byte\_index in 0 to (C\_S\_AXI\_DATA\_WIDTH/8-1) loop

if ( S\_AXI\_WSTRB(byte\_index) = '1' ) then

-- Respective byte enables are asserted as per write strobes

-- slave registor 8

slv\_reg8(byte\_index\*8+7 downto byte\_index\*8) <= S\_AXI\_WDATA(byte\_index\*8+7 downto byte\_index\*8);

end if;

end loop;

when b"1001" =>

for byte\_index in 0 to (C\_S\_AXI\_DATA\_WIDTH/8-1) loop

if ( S\_AXI\_WSTRB(byte\_index) = '1' ) then

-- Respective byte enables are asserted as per write strobes

-- slave registor 9

slv\_reg9(byte\_index\*8+7 downto byte\_index\*8) <= S\_AXI\_WDATA(byte\_index\*8+7 downto byte\_index\*8);

end if;

end loop;

when b"1010" =>

for byte\_index in 0 to (C\_S\_AXI\_DATA\_WIDTH/8-1) loop

if ( S\_AXI\_WSTRB(byte\_index) = '1' ) then

-- Respective byte enables are asserted as per write strobes

-- slave registor 10

slv\_reg10(byte\_index\*8+7 downto byte\_index\*8) <= S\_AXI\_WDATA(byte\_index\*8+7 downto byte\_index\*8);

end if;

end loop;

when others =>

slv\_reg0 <= slv\_reg0;

slv\_reg1 <= slv\_reg1;

slv\_reg2 <= slv\_reg2;

slv\_reg3 <= slv\_reg3;

slv\_reg4 <= slv\_reg4;

slv\_reg5 <= slv\_reg5;

slv\_reg6 <= slv\_reg6;

slv\_reg7 <= slv\_reg7;

slv\_reg8 <= slv\_reg8;

slv\_reg9 <= slv\_reg9;

slv\_reg10 <= slv\_reg10;

end case;

end if;

end if;

end if;

end process;

-- Implement write response logic generation

-- The write response and response valid signals are asserted by the slave

-- when axi\_wready, S\_AXI\_WVALID, axi\_wready and S\_AXI\_WVALID are asserted.

-- This marks the acceptance of address and indicates the status of

-- write transaction.

process (S\_AXI\_ACLK)

begin

if rising\_edge(S\_AXI\_ACLK) then

if S\_AXI\_ARESETN = '0' then

axi\_bvalid <= '0';

axi\_bresp <= "00"; --need to work more on the responses

else

if (axi\_awready = '1' and S\_AXI\_AWVALID = '1' and axi\_wready = '1' and S\_AXI\_WVALID = '1' and axi\_bvalid = '0' ) then

axi\_bvalid <= '1';

axi\_bresp <= "00";

elsif (S\_AXI\_BREADY = '1' and axi\_bvalid = '1') then --check if bready is asserted while bvalid is high)

axi\_bvalid <= '0'; -- (there is a possibility that bready is always asserted high)

end if;

end if;

end if;

end process;

-- Implement axi\_arready generation

-- axi\_arready is asserted for one S\_AXI\_ACLK clock cycle when

-- S\_AXI\_ARVALID is asserted. axi\_awready is

-- de-asserted when reset (active low) is asserted.

-- The read address is also latched when S\_AXI\_ARVALID is

-- asserted. axi\_araddr is reset to zero on reset assertion.

process (S\_AXI\_ACLK)

begin

if rising\_edge(S\_AXI\_ACLK) then

if S\_AXI\_ARESETN = '0' then

axi\_arready <= '0';

axi\_araddr <= (others => '1');

else

if (axi\_arready = '0' and S\_AXI\_ARVALID = '1') then

-- indicates that the slave has acceped the valid read address

axi\_arready <= '1';

-- Read Address latching

axi\_araddr <= S\_AXI\_ARADDR;

else

axi\_arready <= '0';

end if;

end if;

end if;

end process;

-- Implement axi\_arvalid generation

-- axi\_rvalid is asserted for one S\_AXI\_ACLK clock cycle when both

-- S\_AXI\_ARVALID and axi\_arready are asserted. The slave registers

-- data are available on the axi\_rdata bus at this instance. The

-- assertion of axi\_rvalid marks the validity of read data on the

-- bus and axi\_rresp indicates the status of read transaction.axi\_rvalid

-- is deasserted on reset (active low). axi\_rresp and axi\_rdata are

-- cleared to zero on reset (active low).

process (S\_AXI\_ACLK)

begin

if rising\_edge(S\_AXI\_ACLK) then

if S\_AXI\_ARESETN = '0' then

axi\_rvalid <= '0';

axi\_rresp <= "00";

else

if (axi\_arready = '1' and S\_AXI\_ARVALID = '1' and axi\_rvalid = '0') then

-- Valid read data is available at the read data bus

axi\_rvalid <= '1';

axi\_rresp <= "00"; -- 'OKAY' response

elsif (axi\_rvalid = '1' and S\_AXI\_RREADY = '1') then

-- Read data is accepted by the master

axi\_rvalid <= '0';

end if;

end if;

end if;

end process;

-- Implement memory mapped register select and read logic generation

-- Slave register read enable is asserted when valid address is available

-- and the slave is ready to accept the read address.

slv\_reg\_rden <= axi\_arready and S\_AXI\_ARVALID and (not axi\_rvalid) ;

process (slv\_reg0, slv\_reg1, slv\_reg2, slv\_reg3, slv\_reg4, slv\_reg5, slv\_reg6, slv\_reg7, val\_c, state, add\_a, axi\_araddr, S\_AXI\_ARESETN, slv\_reg\_rden)

variable loc\_addr :std\_logic\_vector(OPT\_MEM\_ADDR\_BITS downto 0);

begin

-- Address decoding for reading registers

loc\_addr := axi\_araddr(ADDR\_LSB + OPT\_MEM\_ADDR\_BITS downto ADDR\_LSB);

case loc\_addr is

when b"0000" =>

reg\_data\_out <= slv\_reg0;

when b"0001" =>

reg\_data\_out <= slv\_reg1;

when b"0010" =>

reg\_data\_out <= slv\_reg2;

when b"0011" =>

reg\_data\_out <= slv\_reg3;

when b"0100" =>

reg\_data\_out <= slv\_reg4;

when b"0101" =>

reg\_data\_out <= slv\_reg5;

when b"0110" =>

reg\_data\_out <= slv\_reg6;

when b"0111" =>

reg\_data\_out <= slv\_reg7;

when b"1000" =>

reg\_data\_out <= val\_c;

when b"1001" =>

reg\_data\_out <= state;

when b"1010" =>

reg\_data\_out <= add\_a;

when others =>

reg\_data\_out <= (others => '0');

end case;

end process;

-- Output register or memory read data

process( S\_AXI\_ACLK ) is

begin

if (rising\_edge (S\_AXI\_ACLK)) then

if ( S\_AXI\_ARESETN = '0' ) then

axi\_rdata <= (others => '0');

else

if (slv\_reg\_rden = '1') then

-- When there is a valid read address (S\_AXI\_ARVALID) with

-- acceptance of read address by the slave (axi\_arready),

-- output the read dada

-- Read address mux

axi\_rdata <= reg\_data\_out; -- register read data

end if;

end if;

end if;

end process;

-- Add user logic here

state(31 downto 3) <= (others => '0');

add\_a(31 downto 14) <= (others =>'0');

matrix\_mul: matrix\_mu\_datapath

Port MAP(

clk => S\_AXI\_ACLK,

go => slv\_reg0(0),

w\_e\_a => slv\_reg1(0),

w\_e\_b => slv\_reg2(0),

addr\_a\_in => slv\_reg3(13 downto 0),

addr\_b\_in => slv\_reg4(13 downto 0),

addr\_c\_in => slv\_reg5(13 downto 0),

val\_a\_in => slv\_reg6(15 downto 0),

val\_b\_in => slv\_reg7(15 downto 0),

val\_c\_out => val\_c(31 downto 0),

state\_out => state(2 downto 0),

add\_a\_check => add\_a(13 downto 0)

);

-- User logic ends

end arch\_imp;