Kyle Loyka

Lab 3 Report

ECEN 449-503

Due: 15 February 2016

Introduction

The purpose of this lab was to familiarize ourselves with creating a hardware module to use with software on the FPGA board.

Procedure

In this lab, a hardware module was used to compute the product of two unsigned integers. By following the instructions in the lab manual, the custom peripheral was added to PLB. The XPS program generated a Verilog file to describe this hardware block. The block had 3 registers. Two were designated as the input (multiplicand) values, with the third register reserved for output (the product of the two inputs).

C code was written to write data to the input registers and read data from the output register. The code iterated through several combinations of inputs and recorded the outputs.

Results / Q&A

The project functioned as intended.

(A) The purpose of the temporary register was to act as a delay. Without the temporary register, the multiplication block would not be stable.
The output of the multiplication block updates at the positive edge of every clock signal. The multiplication circuit takes a longer time to complete. Without the temporary register, the output of the block is assigned a value before the multiplication circuit has finished. The temporary register allows the multiplication block to meet the physical time constraints of the circuit.

(B) When the product of *slv_reg0* and s*lv_reg1* is larger than 2^{32}, the multiplication block will produce incorrect output. This is because the multiplication block only supports 32 bit unsigned numbers. If the product of two numbers cannot be represented in 32 bits, then the output will be incorrect. This is known as an overflow.
To fix the overflow there are four cases:

I. If the most significant bit of slv_reg0 and slv_reg1 are both 0, then there will not be an overflow.

II. If the most significant bit of slv_reg0 and slv_reg1 are both 1, then there will be an overflow.

III. If one of the most significant bits is 0 and the other is 1, look at output register.

   i. If the most significant bit of the output register is 1, then there will not be an overflow.

   ii. If the most significant bit of the output register is 0, then there will be an overflow.

Using Verilog (assume values of *slv_reg0*, *slv_reg1*, and *slv_reg2* are in reg0,reg1,reg2 respectively):

```
if (reg0[31] == 0 && reg1[31] == 0) // do nothing
if (reg0[31] == 1 && reg1[31] == 1) {
     $display("OVERFLOW");
     reg2 = 0;
}
if ( (reg0[31] == 1 && reg1[31] == 0) ||
     (reg0[31] == 0 && reg1[31] == 1) ) {
     if (reg2[31] == 0) {
          $display("OVERFLOW");
          reg2 = 0;
     }
}
```

Conclusion
This lab provided a strong foundation for mixing Verilog with C code. Knowledge and understanding was gained as to how hardware blocks can be incorporated into C level code.

```
// Kyle Loyka
// ECEN 449
// Lab 3
// lab3.c

#include<xparameters.h>
#include<multiply.h>


int main(){

unsigned int input1 = 0;
unsigned int input2 = 0;
unsigned int output = 0;

/* calculating product of various input values */
for ( ; input1 < 17; input1++){
      // put inputs into registers
      MULTIPLY_mWriteSlaveReg0(XPAR_MULTIPLY_0_BASEADDR, 0, input1);
      MULTIPLY_mWriteSlaveReg1(XPAR_MULTIPLY_0_BASEADDR, 0, 16-input1);

      // read output from hardware
      output = MULTIPLY_mReadSlaveReg2(XPAR_MULTIPLY_0_BASEADDR, 0);

      xil_printf("%d x %d = %d\n\r", input1,16-input1,output);

}

input1 = 0;

/* calculating squares */
for ( ; input1 < 17; input1++){
      // put inputs into registers
      MULTIPLY_mWriteSlaveReg0(XPAR_MULTIPLY_0_BASEADDR, 0, input1);
      MULTIPLY_mWriteSlaveReg1(XPAR_MULTIPLY_0_BASEADDR, 0, input1);

      // read output from hardware
      output = MULTIPLY_mReadSlaveReg2(XPAR_MULTIPLY_0_BASEADDR, 0);

      xil_printf("%d x %d = %d\n\r", input1,input1,output);
}
return 0;
}
```

```
module user_logic
(
  // -- ADD USER PORTS BELOW THIS LINE ---------------
  // --USER ports added here
  // -- ADD USER PORTS ABOVE THIS LINE ---------------

  // -- DO NOT EDIT BELOW THIS LINE ------------------
  // -- Bus protocol ports, do not add to or delete
  Bus2IP_Clk,                    // Bus to IP clock
  Bus2IP_Reset,                  // Bus to IP reset
  Bus2IP_Data,                   // Bus to IP data bus
  Bus2IP_BE,                     // Bus to IP byte enables
  Bus2IP_RdCE,                   // Bus to IP read chip enable
  Bus2IP_WrCE,                   // Bus to IP write chip enable
  IP2Bus_Data,                   // IP to Bus data bus
  IP2Bus_RdAck,                  // IP to Bus read transfer
acknowledgement
  IP2Bus_WrAck,                  // IP to Bus write transfer
acknowledgement
  IP2Bus_Error                   // IP to Bus error response
  // -- DO NOT EDIT ABOVE THIS LINE ------------------
); // user_logic

// -- ADD USER PARAMETERS BELOW THIS LINE ------------
// --USER parameters added here
// -- ADD USER PARAMETERS ABOVE THIS LINE ------------

// -- DO NOT EDIT BELOW THIS LINE --------------------
// -- Bus protocol parameters, do not add to or delete
parameter C_SLV_DWIDTH                   = 32;
parameter C_NUM_REG                      = 3;
// -- DO NOT EDIT ABOVE THIS LINE --------------------

// -- ADD USER PORTS BELOW THIS LINE -----------------
// --USER ports added here
// -- ADD USER PORTS ABOVE THIS LINE -----------------

// -- DO NOT EDIT BELOW THIS LINE --------------------
// -- Bus protocol ports, do not add to or delete
input                                     Bus2IP_Clk;
input                                     Bus2IP_Reset;
input      [0 : C_SLV_DWIDTH-1]           Bus2IP_Data;
input      [0 : C_SLV_DWIDTH/8-1]         Bus2IP_BE;
input      [0 : C_NUM_REG-1]              Bus2IP_RdCE;
input      [0 : C_NUM_REG-1]              Bus2IP_WrCE;
output     [0 : C_SLV_DWIDTH-1]           IP2Bus_Data;
output                                    IP2Bus_RdAck;
output                                    IP2Bus_WrAck;
output                                    IP2Bus_Error;
// -- DO NOT EDIT ABOVE THIS LINE --------------------
```

```
//----------------------------------------------------------------------
--------
// Implementation
//----------------------------------------------------------------------
--------

  // --USER nets declarations added here, as needed for user logic

  // Nets for user logic slave model s/w accessible register example
  reg        [0 : C_SLV_DWIDTH-1]           slv_reg0;
  reg        [0 : C_SLV_DWIDTH-1]           slv_reg1;
  reg        [0 : C_SLV_DWIDTH-1]           slv_reg2;
  wire       [0 : 2]                        slv_reg_write_sel;
  wire       [0 : 2]                        slv_reg_read_sel;
  reg        [0 : C_SLV_DWIDTH-1]           slv_ip2bus_data;
  wire                                      slv_read_ack;
  wire                                      slv_write_ack;
  integer                                   byte_index, bit_index;

  // --USER logic implementation added here

      /* LAB 3 code added here */

      reg [0: C_SLV_DWIDTH-1] tmp_reg;
      always @(posedge Bus2IP_Clk) begin
           // Reset conditions
           if (Bus2IP_Reset == 1) begin
                slv_reg2 <= 0;
                tmp_reg <= 0;
                end
           // multiplication logic
           else begin
                tmp_reg <= slv_reg0*slv_reg1;    // multiply inputs
                slv_reg2 <= tmp_reg;       // output result to
register
                end
           end
  // ------------------------------------------------------
  // Example code to read/write user logic slave model s/w accessible
registers
  //
  // Note:
  // The example code presented here is to show you one way of
reading/writing
  // software accessible registers implemented in the user logic slave
model.
  // Each bit of the Bus2IP_WrCE/Bus2IP_RdCE signals is configured to
correspond
  // to one software accessible register by the top level template.
For example,
  // if you have four 32 bit software accessible registers in the user
logic,
  // you are basically operating on the following memory mapped
registers:
  //
  //    Bus2IP_WrCE/Bus2IP_RdCE   Memory Mapped Register
  //                     "1000"   C_BASEADDR + 0x0
  //                     "0100"   C_BASEADDR + 0x4
  //                     "0010"   C_BASEADDR + 0x8
  //                     "0001"   C_BASEADDR + 0xC
  //
  // ------------------------------------------------------

  assign
    slv_reg_write_sel = Bus2IP_WrCE[0:2],
    slv_reg_read_sel  = Bus2IP_RdCE[0:2],
    slv_write_ack     = Bus2IP_WrCE[0] || Bus2IP_WrCE[1] ||
Bus2IP_WrCE[2],
    slv_read_ack      = Bus2IP_RdCE[0] || Bus2IP_RdCE[1] ||
Bus2IP_RdCE[2];

  // implement slave model register(s)
  always @( posedge Bus2IP_Clk )
    begin: SLAVE_REG_WRITE_PROC

      if ( Bus2IP_Reset == 1 )
        begin
          slv_reg0 <= 0;
          slv_reg1 <= 0;
//      slv_reg2 <= 0;
        end
      else
        case ( slv_reg_write_sel )
          3'b100 :
            for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1;
byte_index = byte_index+1 )
              if ( Bus2IP_BE[byte_index] == 1 )
                for ( bit_index = byte_index*8; bit_index <=
byte_index*8+7; bit_index = bit_index+1 )
                  slv_reg0[bit_index] <= Bus2IP_Data[bit_index];
          3'b010 :
            for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1;
byte_index = byte_index+1 )
              if ( Bus2IP_BE[byte_index] == 1 )
                for ( bit_index = byte_index*8; bit_index <=
byte_index*8+7; bit_index = bit_index+1 )
                  slv_reg1[bit_index] <= Bus2IP_Data[bit_index];
//        3'b001 :
//          for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1;
byte_index = byte_index+1 )
//            if ( Bus2IP_BE[byte_index] == 1 )
//              for ( bit_index = byte_index*8; bit_index <=
byte_index*8+7; bit_index = bit_index+1 )
//                slv_reg2[bit_index] <= Bus2IP_Data[bit_index];
```

```
          default : ;
        endcase

    end // SLAVE_REG_WRITE_PROC

  // implement slave model register read mux
  always @( slv_reg_read_sel or slv_reg0 or slv_reg1 or slv_reg2 )
    begin: SLAVE_REG_READ_PROC

      case ( slv_reg_read_sel )
        3'b100 : slv_ip2bus_data <= slv_reg0;
        3'b010 : slv_ip2bus_data <= slv_reg1;
        3'b001 : slv_ip2bus_data <= slv_reg2;
        default : slv_ip2bus_data <= 0;
      endcase

    end // SLAVE_REG_READ_PROC

  // ------------------------------------------------------------
  // Example code to drive IP to Bus signals
  // ------------------------------------------------------------

  assign IP2Bus_Data    = slv_ip2bus_data;
  assign IP2Bus_WrAck   = slv_write_ack;
  assign IP2Bus_RdAck   = slv_read_ack;
  assign IP2Bus_Error   = 0;

endmodule
```

<u>Kermit Output</u>

```
0 x 16 = 0
1 x 15 = 15
2 x 14 = 28
3 x 13 = 39
4 x 12 = 48
5 x 11 = 55
6 x 10 = 60
7 x 9 = 63
8 x 8 = 64
9 x 7 = 63
10 x 6 = 60
11 x 5 = 55
12 x 4 = 48
13 x 3 = 39
14 x 2 = 28
15 x 1 = 15
16 x 0 = 0
1 x 1 = 1
2 x 2 = 4
3 x 3 = 9
4 x 4 = 16
5 x 5 = 25
6 x 6 = 36
7 x 7 = 49
8 x 8 = 64
9 x 9 = 81
10 x 10 = 100
11 x 11 = 121
12 x 12 = 144
13 x 13 = 169
14 x 14 = 196
15 x 15 = 225
16 x 16 = 256
```