CS220 Lab#3 Large Adders and Sequential Logic

Mainak Chaudhuri
Indian Institute of Technology Kanpur

(Acknowledgment: Professors Urbi Chatterjee, Debapriya Basu Roy)

Sketch

- Assignment#1: 64-bit adder
  - New learning: how to send outputs generated by the synthesized adder (PL) to the processing system (PS) and print them on computer display
  - Carries two marks
- Assignment#2: Blinking and rippling LEDs
  - New learning: use of a clock signal
  - Carries two marks

Assignment#1

- Procedure for sending outputs from PL to PS
  - Write the Verilog program of the desired hardware (64-bit adder in this case)
  - Connect it to the PS through the Advanced Extensible Interface (AXI) bus
    - Requires generation of an intellectual property (IP) block using the designed hardware
    - Only IP blocks can be connected to the AXI bus
  - Write a C program to generate the required inputs for the desired hardware and collect outputs from the hardware
  - Run the C program on the PS while synthesizing the desired hardware on the PL

Assignment#1

- Follow the instructions from Lab1 to create a project (name it *Lab3_64bit_adder*) and add the following sources
  - *fulladder.v*
    - Takes three bits as input *a*, *b*, *cin* and produces two bits of output *sum*, *cout*
  - *adder32.v*
    - Takes three inputs *a*[31:0], *b*[31:0], *cin* and produces two outputs *sum*[31:0], *cout*
    - Instantiates 32 *fulladder* modules and connects them appropriately in the ripple-carry configuration

Assignment#1

- Follow the instructions from Lab1 to create a project (name it *Lab3_64bit_adder*) and add the following sources (continued)
  - *adder64.v*
    - Takes four inputs *a_low*[31:0], *a_high*[31:0], *b_low*[31:0], *b_high*[31:0] and produces three outputs *sum_low*[31:0], *sum_high*[31:0], *carry*
      - *a_low* stores the least significant 32 bits of *a* and *a_high* stores the most significant 32 bits of *a*
      - *carry* is the output carry bit of the 64-bit adder
    - Instantiates two *adder32* modules and connects them in the ripple-carry configuration
    - PS will send *a_low*, *a_high*, *b_low*, *b_high* to PL and PL will send *sum_low*, *sum_high*, *carry* back to PS

Assignment#1

- At this point, you should write a simulation testbench and simulate the *adder64* module to check its functional correctness
  - This is an optional step and can be skipped if you are confident about the correctness of your module
- You can run synthesis to check that the compiler is able to compile your module without any error
  - This is also an optional step and can be skipped if you are confident that there is no error in your module

Assignment#1

- Create the IP block for the *adder64* module following the steps given in Lab2
  - Let us name it Adder64IP
  - Number of registers should be the number of inputs (each can be up to 32 bits) we want to send from PS and the number of outputs (each can be up to 32 bits) we want to collect at PS
    - We need to send four inputs (*a_low*, *a_high*, *b_low*, *b_high*) and collect three outputs (*sum_low*, *sum_high*, *carry*); so the number of registers should be 7
  - Add all the three .v source files
    - *Adder64IP_v1_0_S00_AXI.v* and *Adder64IP_v1_0.v* will get added automatically

Assignment#1

- No need to add any user-defined ports to the module in *Adder64IP_v1_0_S00_AXI.v*
  - We are not sending any input from the board, nor are we sending any output to LEDs
- Will use *slv_reg0*, *slv_reg1*, *slv_reg2*, *slv_reg3* as the four inputs to the *adder64* module
- Let us call the three outputs of the *adder64* module *sum_hi*, *sum_lo*, *carry*

Assignment#1

- Declare these three signals inside the module in *Adder64IP_v1_0_S00_AXI.v*

```
wire [31:0] sum_lo;
wire [31:0] sum_hi;
wire carry;
```

- Scroll down to the end of the module where you will find the following two line

```
// Add user logic here
// User logic ends
```

  – Between these two comment lines add an instantiation of the *adder64* module with inputs *slv_reg0*, *slv_reg1*, *slv_reg2*, s*lv_reg3* and outputs *sum_hi*, *sum_lo*, *carry*

Assignment#1

- Locate the following three lines in the module (should be around line#411)

```
3'h4   : reg_data_out <= slv_reg4;
3'h5   : reg_data_out <= slv_reg5;
3'h6   : reg_data_out <= slv_reg6;
```

  – In these lines, replace *slv_reg4* by *sum_lo*, *slv_reg5* by *sum_hi*, and *slv_reg6* by *carry*
- Save *Adder64IP_v1_0_S00_AXI.v*
- We don’t need to make any changes to *Adder64IP_v1_0.v*
- Check that the hierarchy shown in the *Design Sources* pane is correct
- Complete packaging the IP (see Lab2 steps)

Assignment#1

- Follow the Lab2 steps to *Create Block Design*
  - In place of the *FiveBitAdderIP*, you should add *Adder64IP*
  - You don't need to create any port this time as we are not sending any input from the board, nor are we connecting any output to the LEDs
- After validating your block design, *Create HDL Wrapper*
- We don't need a constraints file as we are not connecting any input or output of our design to any I/O component of the board

Assignment#1

- Right-click *TopDesign_wrapper* in *Sources* pane, select *Set as Top*, once it is set as top (indicated by three dots in front of it), click *Run Synthesis* from the left hand *Flow Navigator* menu, click *OK*
- Once synthesis completes successfully, select *Run Implementation*, click *OK*, click *OK*
- Once implementation completes successfully, select *Generate Bitstream*, click *OK*, click *OK*

Assignment#1

- Once bitstream generation completes successfully, click *cancel*
- In the top horizontal menu, click *File*, click *Export*, select *Export Hardware*, click *Next*, select *Include bitstream*, click *Next*, note the *XSA* file name (should be *TopDesign_wrapper*) and its path (should be Lab3_64bit_adder), click *Next*, click *Finish*
- This completes generation of the hardware
  - The XSA file stores the hardware (the entire block diagram that we have developed) using some encoding

Assignment#1

- In the top horizontal menu of the right-hand pane, click on *Address Editor*
  - Read the line starting with */Adder64IP_0/S00_AXI*
  - It mentions that the file of AXI registers (denoted *S00_AXI_reg*) starts at address 0x43C00000 and ends at 0x43C0FFFF
    - Numbers with prefix *0x* are hexadecimal numbers
    - Four inputs need to be written to addresses 0x43C00000, 0x43C00004, 0x43C00008, 0x43C0000C
    - Three outputs should be read from addresses 0x43C00010, 0x43C00014, 0x43C00018

Assignment#1

- Following the steps from Lab2, use *Vitis IDE* to prepare the C program that will run on PS
- Edit the C program (helloworld.c) as follows
  - Include three more files

    ```
    #include "xbasic_types.h"
    #include "xparameters.h"
    #include "xil_io.h"
    ```

  - Declare a global pointer variable *base_addr* of type *Xuint32** and initialize it to the address of *slv_reg0*
    - `Xuint32 *base_addr = (Xuint32*)0x43c00000;`

Assignment#1

- Inside the *main*() function, insert the following lines before *cleanup_platform*()

```
print("\n\r");
unsigned a_lo, b_lo, a_hi, b_hi, sum_lo, sum_hi, carry;
a_lo = (1ULL << 32)-2;
a_hi = (1ULL << 32)-1;
b_lo = (1ULL << 32)-3;
b_hi = (1ULL << 32)-1;
Xil_Out32(base_addr, a_lo);
Xil_Out32(base_addr+1, b_lo);
Xil_Out32(base_addr+2, a_hi);
Xil_Out32(base_addr+3, b_hi);
```

  – The order of the last four lines will depend on the order of inputs to your *adder64* module

Assignment#1

- To capture and print the outputs, include the following lines in *main*() after sending the inputs
    - sum_lo = Xil_In32(base_addr+4);
    - sum_hi = Xil_In32(base_addr+5);
    - carry = Xil_In32(base_addr+6);
    - xil_printf("Output: %u %u %u\n\r", carry, sum_hi, sum_lo);
  - The order of the first three lines will depend on the order of outputs in the *adder64* module
  - Note the lower case 'x' in *xil_printf* and upper case 'X' in *Xil_Out32* and *Xil_In32*
- Save the file, build project, and run it following the steps of Lab2

Assignment#1

- Try different inputs, check outputs
- Show your work to a TA to receive credits
- Click *File* in the Vitis IDE top horizontal menu and click *Exit*
- Click *File* in the vivado top horizontal menu and click *Exit*, click *OK*
- Switch OFF the FPGA board and disconnect from computer

Assignment#2

- Create a project with name *Lab3_LED_controller*
- Add *LEDcontroller.v* containing a module *LEDcontroller*
  - Takes three one-bit inputs: *clk*, *blink*, *right*
    - If the *blink* input is 1, all monochrome LEDs should blink once per two seconds (stay ON for one second, stay OFF for one second, stay ON for one second, …)
    - If the *blink* input is 0 and the *right* input is 0, the monochrome LEDs ripple from right to left (LED0 stays ON for one second while others are OFF, LED1 stays ON in the next second while others are OFF, …)
    - If the *blink* input is 0 and the *right* input is 1, the monochrome LEDs ripple from left to right
    - The *right* input has no effect if the *blink* input is 1

Assignment#2

- Create a project with name *Lab3_LED_controller*
- Add *LEDcontroller.v* containing a module *LEDcontroller*
  - Since the module generates signals for the four LEDs, the module should have an array of four signals *LED*[3:0] as output
  - *clk* is the input clock signal
  - Inputs are checked only on rising clock edges
  - We will use a 125 MHz clock signal
    - One second amounts to incrementing a counter up to 125000000 with one increment per clock cycle

Assignment#2

- The module may have the following structure

```
always @ (posedge clk) begin
   counter <= (counter < 125000000) ? counter + 1 :
0;
   if (counter == 125000000) begin
      Do per second events
   end
   Check blink and right and take appropriate actions
end
```

- As done in Assignment#1 of Lab1, create a block diagram of the module and create ports for inputs and outputs
  - Clock port frequency should be set to 125 MHz
- Create HDL wrapper

Assignment#2

- Skip simulation
- Create a constraints file
  - Connect the 125 MHz board clock to the clock port of your design
  - Copy and uncomment the line starting with *create_clock*, change port name to your clock port
  - Connect *blink* and *right* ports to the two switches *SW0* and *SW1*
  - Connect the four outputs to four monochrome LEDs

Assignment#2

- Synthesize, implement, generate bit stream, program FPGA
- Check output
- Show your work to a TA to get credit