CS220 Lab#2
Adders and Comparators

Mainak Chaudhuri
Indian Institute of Technology Kanpur
(Acknowledgment: Professors Urbi Chatterjee, Debapriya Basu Roy)

Sketch

- Assignment#1: 5-bit adder
  - New learning: how to send two 5-bit numbers as inputs to the synthesized adder (PL) from the processing system (PS)
  - Carries two marks
- Assignment#2: 32-bit comparator
  - New learning: how to efficiently write the Verilog code to connect 32 one-bit comparators
  - Carries two marks

Assignment#1

- General procedure for sending a large input
  - Write the Verilog program of the desired hardware (5-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
  - Run the C program on the PS while synthesizing the desired hardware on the PL
  - The generated inputs are ferried to the PL through the AXI bus when the C program runs

Assignment#1

- Follow the instructions from Lab1 to create a project (name it *Lab2_5bit_adder*) and add the source of a full adder (call it *fulladder.v*)
- After you have saved *fulladder.v*, right-click *Design Sources*, and add the Verilog module for a 5-bit adder (call it *five_bit_adder.v*)
  - This module should have two inputs *a* and *b* which are each five bits wide and two outputs *sum* (five bits) and *carry* (one bit)
    - When specifying input *a*, tick *Bus*, and specify 4 for MSB (this is the position of the most significant bit) and 0 for LSB (least significant bit position)
    - Specify input *b* and output *sum* similarly
    - Output carry is not a bus output (*Bus* not ticked)

Assignment#1

- Open *five_bit_adder.v* by double-clicking it
- Assuming your *fulladder* module is defined as *fulladder(a, b, cin, sum, cout),* edit *five_bit_adder.v* to include the following

```
wire [3:0] cout;
fulladder fa0(a[0], b[0], 1'b0, sum[0], cout[0]);
fulladder fa1(a[1], b[1], cout[0], sum[1], cout[1]);
fulladder fa2(a[2], b[2], cout[1], sum[2], cout[2]);
fulladder fa3(a[3], b[3], cout[2], sum[3], cout[3]);
fulladder fa4(a[4], b[4], cout[3], sum[4], carry);
```

- Save *five_bit_adder.v*

Assignment#1

- At this point, you should write a simulation testbench and simulate the *five_bit_adder* 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 and implementation 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

- Next, we will create the IP block for the five-bit adder module
- From the top horizontal menu bar, click *Tools*, and then click *Create and Package New IP*, click *Next*, select *Create a new AXI4 peripheral*, click *Next*, give any name (don't use hyphen; let's give the name *FiveBitAdderIP*), click *Next*
- *Interface Type* should be *Lite*, *Interface Mode* should be *Slave*

Assignment#1

- 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 we want to collect at PS
  - We just need to send two inputs; so the number of registers should be 2, but minimum is 4; so set it to 4
- Click *Next*, select *Edit IP*, click *Finish*
- These steps create a default IP block
- Now we will add our modules to this IP

Assignment#1

- Right-click *Design Sources* in the left-hand *Sources* pane, click Add Sources, select *Add or create design sources*, click *Next*, click *Add Files*, browse to *five_bit_adder.v* and double-click on it
- Click *Add Files*, browse to *fulladder.v*, double-click on it, click *Finish*
- In the Sources pane on the left, you can see what all we have got in the IP
  - *fulladder.v*, *five_bit_adder.v*, *FiveBitAdderIP_v1_0_S00_AXI.v*, *FiveBitAdderIP_v1_0.v*
    - The last one is the top-level file
    - You can examine/edit each file by double-clicking it

Assignment#1

- We need to edit two of these files to include instantiation of our five-bit adder module
- In the top-level module, we need to emit two outputs *sum* and *carry* which will be connected to the LEDs
- In the top-level module, we don't need to feed any input for our module from the board
  - These will be generated by the PS
- Double-click *FiveBitAdderIP_v1_0_S00_AXI.v* to open it for editing

Assignment#1

- We need to add two output ports to this module
  - Add them as shown below exactly between the two comment lines (first locate the two comment lines; should be line number 17)

```
// Users to add ports here
output wire [4:0] SUM,
output wire CARRY,
// User ports ends
```

  - Remember that these are the first two arguments of this module; we will need to add them to the instantiation later

Assignment#1

- Next we need to capture the inputs coming from the PS
  - Recall that we have specified four registers for this purpose when creating the IP
  - These registers appear as *slv_reg0*, *slv_reg1*, *slv_reg2*, and *slv_reg3* in this module
    - Declared in lines 108-111
    - The parameter C_S_AXI_DATA_WIDTH has value 32 (defined in line 12)
    - We will use the first two of these registers to pass *a* and *b*

Assignment#1

- 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 the following instantiation of our five-bit adder module

    five_bit_adder fba (slv_reg0, slv_reg1, SUM, CARRY);

    - Assuming that the five-bit adder module definition is five_bit_adder (a, b, sum, carry)

- Save *FiveBitAdderIP_v1_0_S00_AXI.v*

Assignment#1

- Double-click *FiveBitAdderIP_v1_0.v* for editing
  - Add two output ports as shown below starting at line number 17

```
// Users to add ports here
output wire [4:0] sum,
output wire carry,
// User ports ends
```

Assignment#1

- Scroll down to instantiation of *FiveBitAdderIP_v1_0_S00_AXI* module and add the following two lines after line 51

  .SUM(sum),

  .CARRY(carry),

  – These lines should be added just before the line .S_AXI_ACLK(s00_axi_aclk)

  – Save the file

- We have edited the IP files to (i) pass the inputs to our five bit adder module, (ii) emit outputs from the top module, (iii) instantiate a five-bit adder

Assignment#1

- In the *Sources* pane on the left, you can see the design hierarchy under *Design Sources*
  - *FiveBitAdderIP_v1_0* is the top module (with three dots at the front indicating top module)
  - Next module below is *FiveBitAdderIP_v1_0_S00_AXI_inst*
  - Next module below is *five_bit_adder*
  - Bottom-most module is *fulladder*
- It is always good to verify that the design hierarchy makes sense
  - The hierarchy is decided by which module is instantiated by which module

Assignment#1

- Click on *Package IP – FiveBitAdderIP* from the top menu of the right hand pane
- Click on *File Groups*, click on *Merge changes from File Groups Wizard*
- Click on *Customization Parameters*, click on *Merge changes from Customization Parameters Wizard*
- Click on *Ports and Interfaces*, check that *sum* and *carry* are listed as output ports of proper sizes
- Click on *Review and Package*, click on *Re-Package IP*, click *Yes*

Assignment#1

- We are done building our IP block
- In the left hand *Flow Navigator* menu, click on *Create Block Design*
- Name it something (say, *TopDesign*), click *OK*
  - This will open an empty block design pane
- Click on + in the middle of the pane and select *Zynq7 Processing System* from the drop-down list, double-click on it
  - This is the PS block

Assignment#1

- Click *Run Block Automation*
  - If your machine seems to have hung, terminate *vivado* (this is a known bug of *vivado* on *ubuntu*)
    - Open a new terminal (don't close the existing one)
    - Type the command *ps –U username*
      - Replace *username* by your user name
    - Identify all processes named *vivado* from the list
    - Type command *kill -9 pid*
      - Replace *pid* by the pid of the *vivado* process
      - Type as many *kill* commands as the number of *vivado* processes in the list
    - Type the command *vivado* to relaunch *vivado*
    - Click *Open Project* and browse to *Lab2_5bit_adder.xpr*, double-click it to open

Assignment#1

- Double-click on *TopDesign* in the *Sources* pane
- Click on + in the middle of the pane and select *Zynq7 Processing System* from the drop-down list
  - This is the PS block
- Click *Run Block Automation*, click *OK*
- If your machine did not hang, continue from the next step
- Right-click on *M_AXI_GP0_ACLK* pin, click on *Make Connection*, select *FCLK_CLK0*, click *OK*

Assignment#1

- Right-click in the empty space of the block design pane, click *Add IP*, select and double-click *FiveBitAdderIP* from the drop-down list
- Right-click on *sum* pin (zoom the diagram), click *Create Port*, name it SUM, click *OK*
- Right-click on *carry* pin, click *Create Port*, name it CARRY, click *OK*
- Click *Run Connection Automation*, click *OK*
  - Connects *FiveBitAdderIP* and PS to the AXI bus
- Validate the design by clicking the "tick in a square icon" in the top menu, click *OK*

Assignment#1

- In the *Sources* pane, right-click *TopDesign*, select *Create HDL Wrapper*, select *Let Vivado manage wrapper and auto-update*
- In the *Sources* pane, right-click on *constrs_1* under *Constraints*, select *Add Sources*, select *Add or create constraints*, click *Next*, click *Create File*, name the file something (say, *Constraints*), click *OK*, click *Finish*
- Click > in front of *constrs_1* to expand and double-click *Constraints.xdc*
  - In this file, we need to connect *SUM* to 5 LEDs and *CARRY* to one LED

Assignment#1

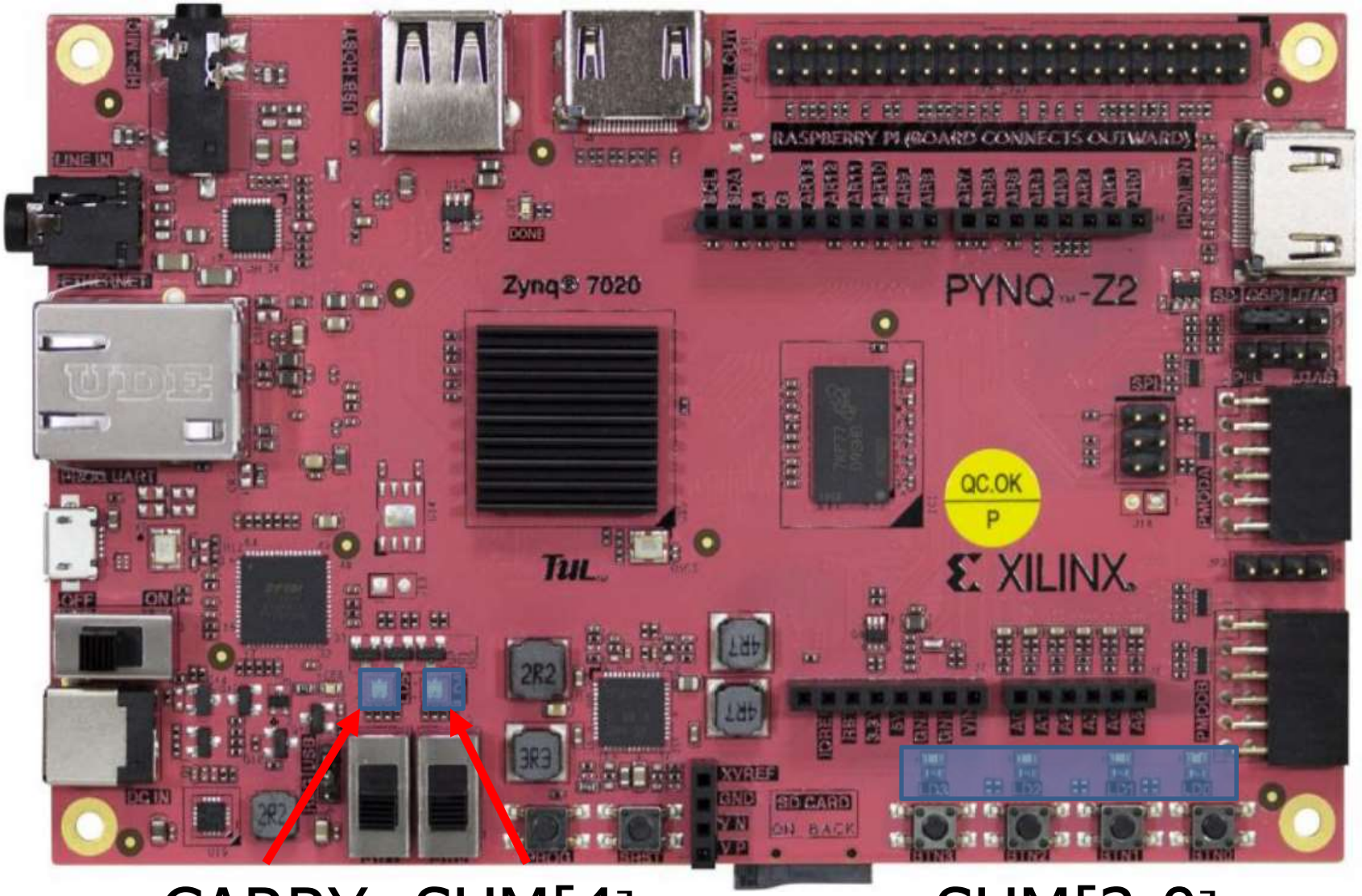
- Include the following lines in *Constraints.xdc* and save the file

```
##RGB LEDs

set_property -dict { PACKAGE_PIN G17   IOSTANDARD LVCMOS33 } [get_ports { SUM[4] }]; #IO_L16P_T2_35 Sch=led4_g
set_property -dict { PACKAGE_PIN L14   IOSTANDARD LVCMOS33 } [get_ports { CARRY }]; #IO_L22P_T3_AD7P_35 Sch=led5_g

##LEDs

set_property -dict { PACKAGE_PIN R14   IOSTANDARD LVCMOS33 } [get_ports { SUM[0] }]; #IO_L6N_T0_VREF_34 Sch=led[0]
set_property -dict { PACKAGE_PIN P14   IOSTANDARD LVCMOS33 } [get_ports { SUM[1] }]; #IO_L6P_T0_34 Sch=led[1]
set_property -dict { PACKAGE_PIN N16   IOSTANDARD LVCMOS33 } [get_ports { SUM[2] }]; #IO_L21N_T3_DQS_AD14N_35 Sch=led[2]
set_property -dict { PACKAGE_PIN M14   IOSTANDARD LVCMOS33 } [get_ports { SUM[3] }]; #IO_L23P_T3_35 Sch=led[3]
```

Assignment#1

- We have connected *SUM*[3:0] to the monochrome LEDs, *SUM*[4] to green color of RGB LED4, and *CARRY* to green color of RGB LED5

Assignment#1



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 Lab2_5bit_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

- Now we will develop the software to generate the inputs and run it on PS
- To send the inputs to *slv_reg0* and *slv_reg1*, we need to know the addresses of these two registers
  - Every register connected to the AXI bus has an address

Assignment#1

- In the top horizontal menu of the right-hand pane, click on *Address Editor*
  - Read the line starting with */FiveBitAdderIP_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

Assignment#1

- The input registers are the first two registers in this address range
  - *slv_reg0* starts at address 0x43C00000 and spans four bytes i.e., 0x43C00000, 0x43C00001, 0x43C00002, and 0x43C00003
    - Each address corresponds to a byte (this is called byte-addressable memory)
  - *slv_reg1* starts at address 0x43C00004 and spans four bytes
  - We will use these addresses in our C program to send the inputs

Assignment#1

- In the top horizontal menu, click *Tools*, click *Launch Vitis IDE*
  - This will launch the Vitis IDE, which we will use to prepare our C program, run it on PS, and synthesize our hardware on PL
  - Browse to *Lab2_5bit_adder*, click on the *Create Folder* icon in the upper right-hand corner, create a folder named *vitis_workspace*, click on *Open* in the upper right-hand corner, click *Launch*
- Click *Create Platform Project*
- Give a name (let's use *FiveBitAdderIP_platform*), click *Next*

Assignment#1

- Select *Create a new platform from hardware (XSA)*
- Browse to the *TopDesign_wrapper.xsa* file and double-click it to select, click *Finish*
  - We have now loaded our hardware into the Vitis IDE
- At the top of the leftmost pane, right-click on *FiveBitAdderIP_platform (Out of date),* click *Build Project*
  - If everything goes fine, there will be no error and the "*Out of date*" tag will not be there after *FiveBitAdderIP_platform* in the leftmost pane

Assignment#1

- From the top horizontal menu, click *File*, select *New*, click *Application Project*, click *Next*, select *FiveBitAdderIP_platform* from the list, click *Next*
- Give a name to the application project
  - Let's use the name *FiveBitAdderIP_app*
- Click *Next*, click *Next*
- We will select a simple application program template from the list and edit it
  - Select *Hello World*
  - Click *Finish*

Assignment#1

- In the upper left pane, expand *src* under *FiveBitAdderIP_app*, and double-click *helloworld.c*
- Have a look at the opened program
  - We will edit this program to send two inputs to the addresses of *slv_reg0* and *slv_reg1*
- Add lines to the program to include three more files

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

Assignment#1

- Declare a global pointer variable *base_addr* of type *Xuint32** and initialize it to the address of *slv_reg0*
  - Xuint32 *base_addr = (Xuint32*)0x43c00000;
- Inside the *main*() function, insert the following lines before *cleanup_platform*()

```
print("\n\r");
int a, b;
a = 30;
b = 20;
Xil_Out32(base_addr, a);
Xil_Out32(base_addr+1, b);
```

  - The last two lines send out the values of *a* and *b* to the addresses of *slv_reg0* and *slv_reg1*

Assignment#1

- Save the file by clicking the save icon in the top menu
- In the upper leftmost pane, right-click on *FiveBitAdderIP_app_system*, click *Build Project*
  - Should compile without any error
- Open a new ubuntu terminal by pressing Ctrl+Alt+t
- Type the command *gtkterm* in the terminal
  - A new terminal should pop up
  - We will use this terminal to communicate with PS

Assignment#1

- Connect the FPGA board to the computer using one of the USB ports and switch it ON
- Click on the *Configuration* tab of the *gtk* terminal
- Enter */dev/ttyUSB1* as the port, *Baud Rate* should be 115200, do not change other fields, click *OK*
  - If you have connected the FPGA board and switched it ON, the selected port should connect without any error message

Assignment#1

- Go back to Vitis IDE, in the upper leftmost pane right-click *FiveBitAdderIP_app_system*, select *Run As*, click *Launch Hardware*
  - Vitis will program the FPGA
  - Once it is programmed, you should see the output of addition (30+20) in the LEDs
- Change the C program to give some other input combination
  - Inputs can be non-negative and less than 32
  - Remember to save, build the project, and reprogram the FPGA as outlined above
- Show your outputs to a TA to get credit

Assignment#1

- 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

- In this assignment, we will use a *for* loop to instantiate an array of a module and connect these instantiations to build a bigger module
- As an example, consider designing a 128-bit ripple-carry adder using 128 full-adder modules
  - Let us assume that the full-adder module is defined as *fulladder(a, b, cin, sum, cout)*

Assignment#2

- The following module defines a 128-bit adder

```
module adder128 (a, b, sum, carry);
   input [127:0] a;
   input [127:0] b;
   output [127:0] sum;
   output carry;
   wire [127:0] c;
   genvar i;
   fulladder fa0 (a[0], b[0], 1'b0, sum[0], c[0]);
   generate for (i=1; i<128; i=i+1) begin: ripple
      fulladder fa (a[i], b[i], c[i-1], sum[i], c[i]);
   end
   endgenerate
   assign carry = c[127];
endmodule
```

Assignment#2

- Few notes
  - *genvar* is the type of the variable *i* that is used to instantiate the full adders in a loop
  - *ripple* is the name of the generate for loop
    - Any name can be used
- We will use *generate for* loops to connect one-bit comparators to design a 32-bit comparator in this assignment

Assignment#2

- Create a project with name *Lab2_32bit_comparator*
- Add the sources for one-bit comparator and 32-bit comparator
  - The one-bit comparator should have five inputs (*a*, *b*, *less_prev*, *greater_prev*, *equal_prev*) and three outputs (*less*, *greater*, *equal*)
    - The idea is to connect (*less*, *greater*, *equal*) of bit *n* to (*less_prev*, *greater_prev*, *equal_prev*) of bit *n* – 1
    - The cleanest way of implementing this module is to work out the Boolean formulae for *less*, *greater*, *equal* and use assign statements to write them

Assignment#2

- Connect 32 1-bit comparators using a generate for loop
  - This module should have two 32-bit inputs *a* and *b* and three one-bit outputs *less*, *greater*, *equal*
- Write a short testbench to simulate the 32-bit comparator and check its behavioral correctness

Assignment#2

- Create IP for 32-bit comparator with outputs *less*, *greater*, *equal* which will be connected to three LEDs
- Create block diagram
  - Remember to creates ports *LESS*, *GREATER*, *EQUAL*
- Create HDL wrapper
- Create constraint file
  - Connect *LESS*, *GREATER*, *EQUAL* to rightmost three monochrome LEDs (*led*[0], *led*[1], *led*[2])
- Set the HDP wrapper as top module, synthesize, implement, generate bitstream, export hardware

Assignment#2

- Develop a C program using Vitis IDE
  - Make sure to declare *a* and *b* as *unsigned int*
  - Try different values of *a* and *b* and observe output in the LEDs
    - Try values such as *a* = (1ULL << 32) – 1 and *b* = (1ULL << 32) – 2
    - Try more values
  - Show the outputs to a TA