CS220 Lab#1
Introduction to PYNQ-Z2
and
AMD/Xilinx Vivado

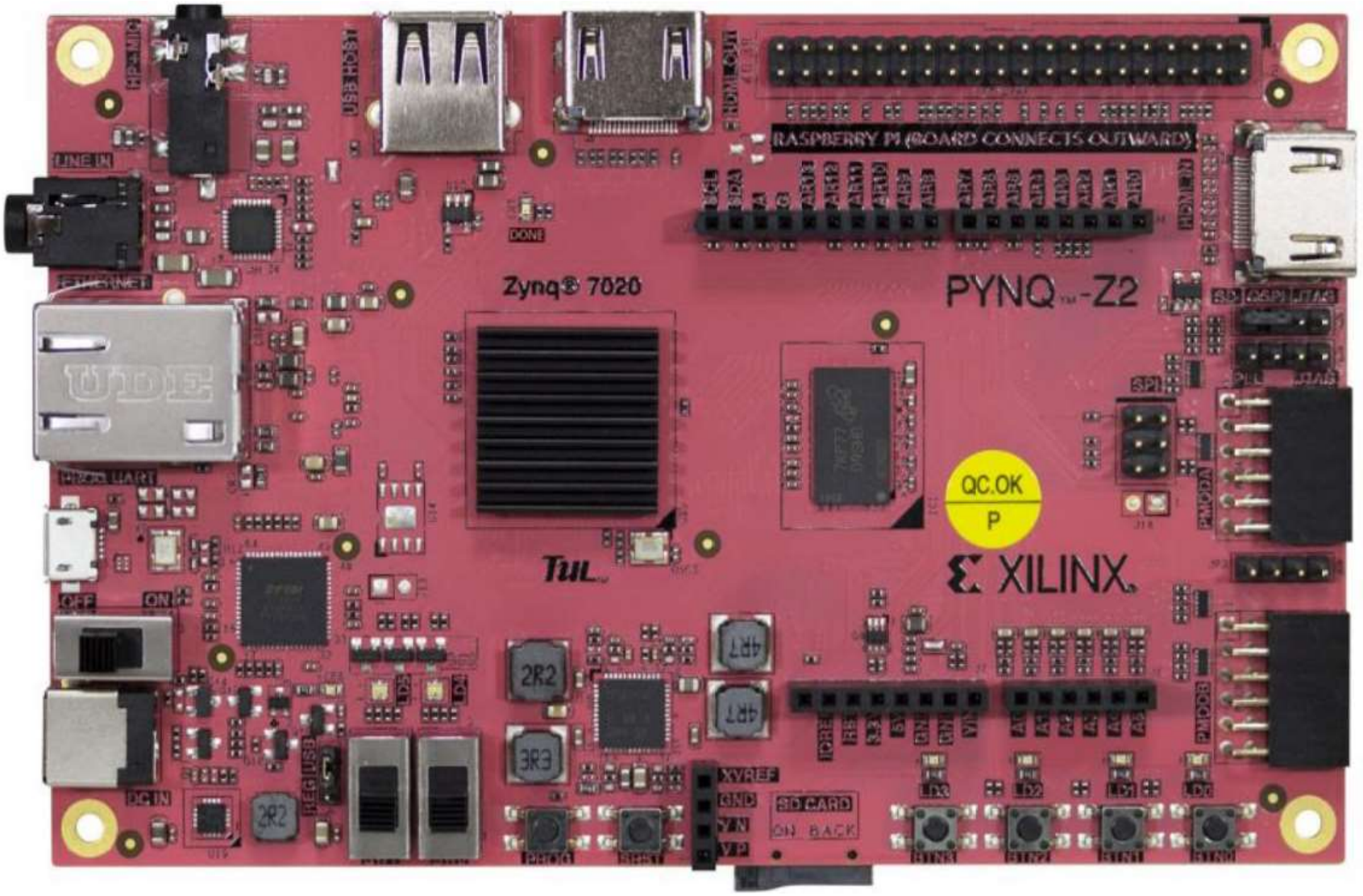
Mainak Chaudhuri
Indian Institute of Technology Kanpur
(Acknowledgment: Professors Urbi Chatterjee, Debapriya Basu Roy)

Sketch

- Brief on PYNQ-Z2 FPGA board
- AMD/Xilinx Vivado
  - Example implementation of a full adder and a two-bit adder

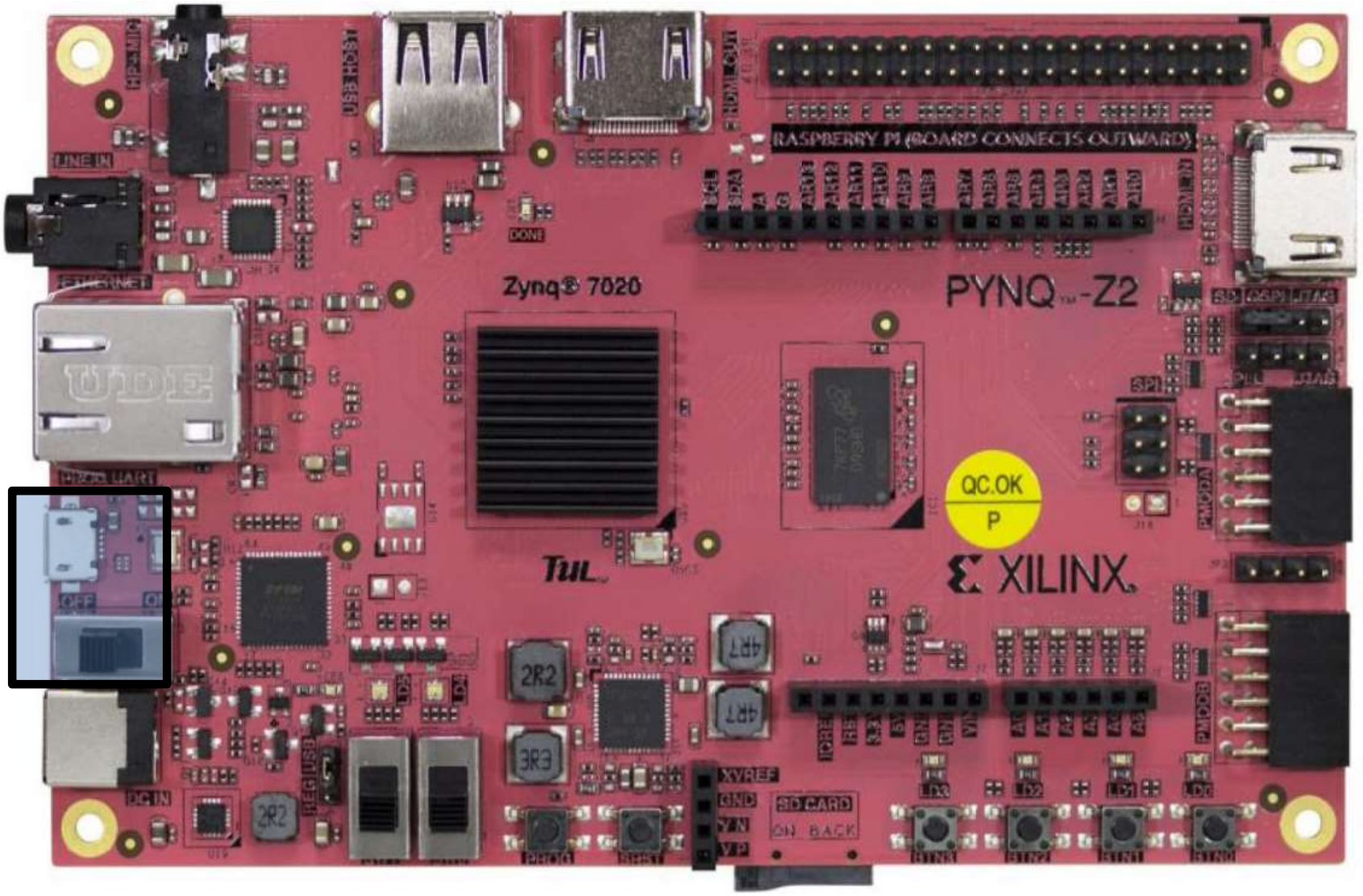
PYNQ-Z2 FPGA Board

- Contains several parts



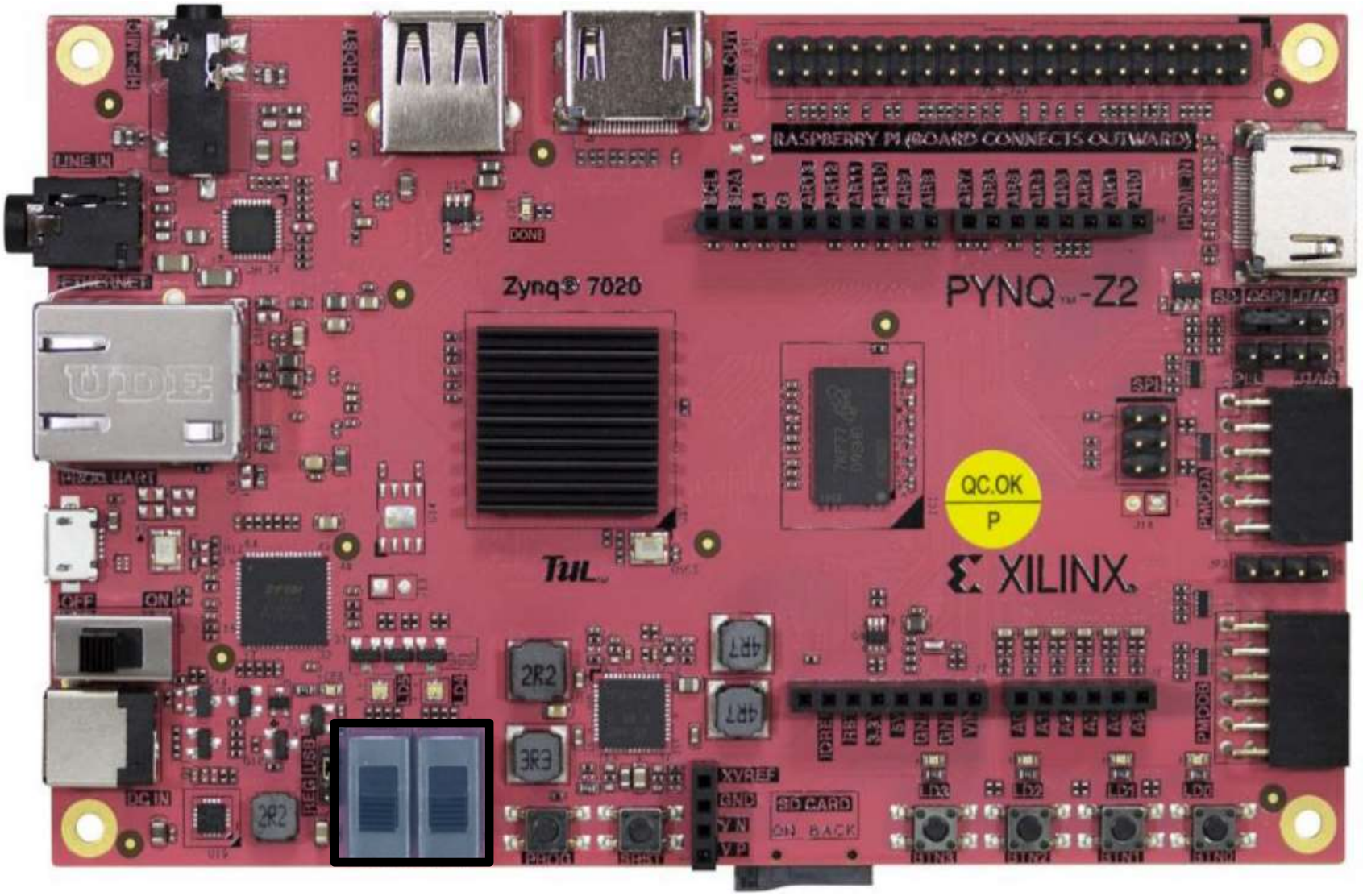
PYNQ-Z2 FPGA Board

- Power connection port and power switch



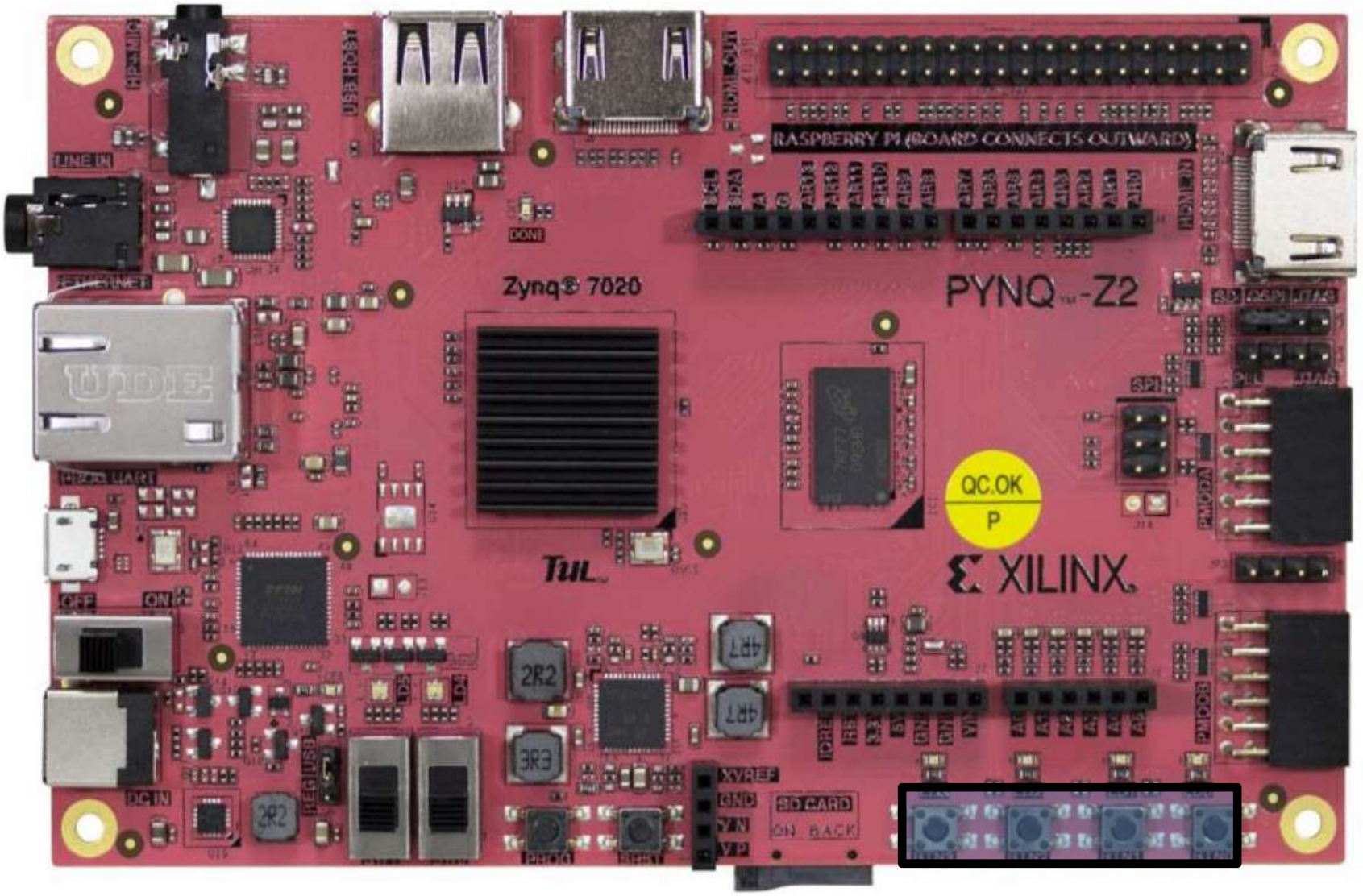
PYNQ-Z2 FPGA Board

- Switches (for giving inputs)



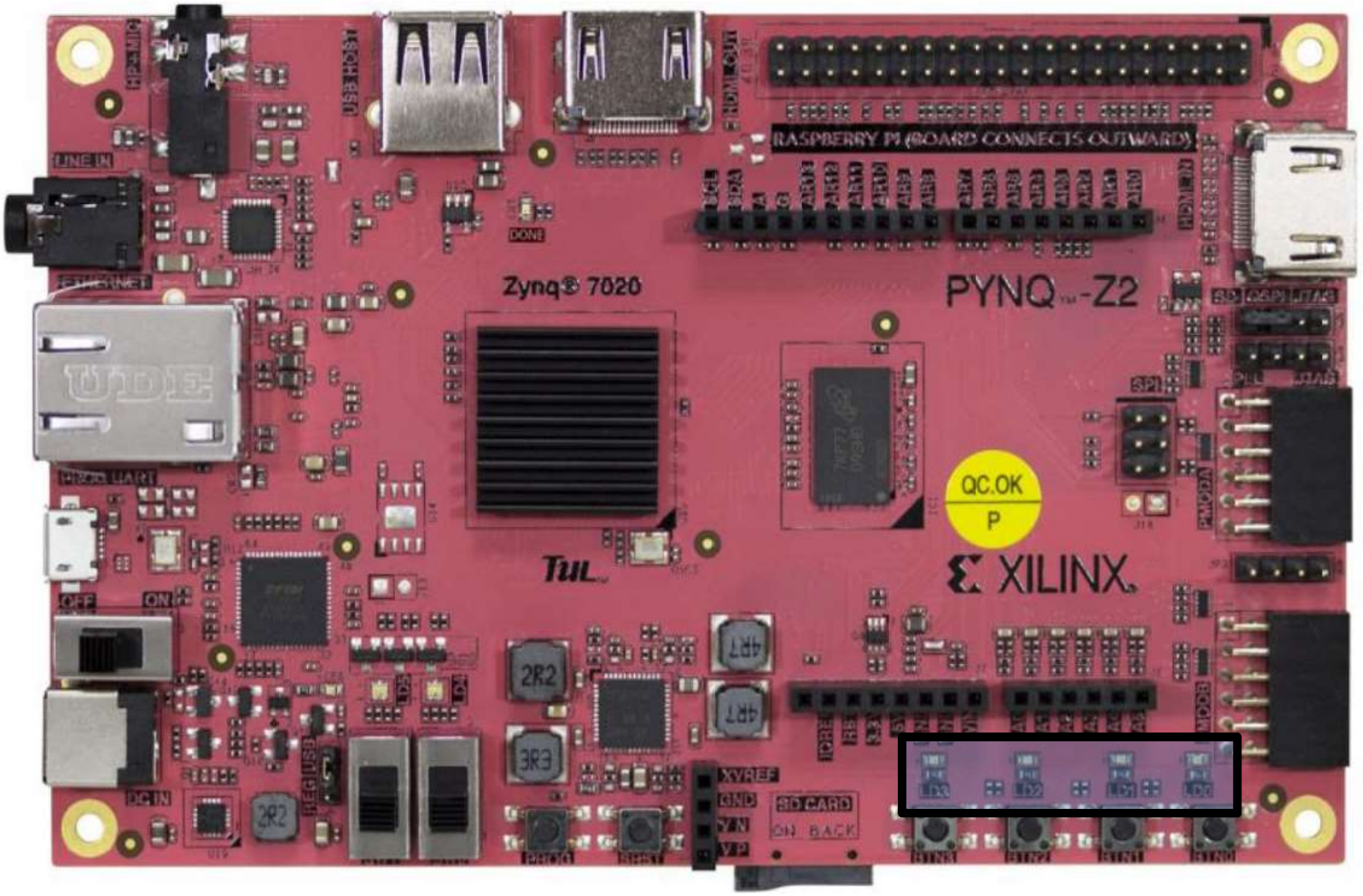
PYNQ-Z2 FPGA Board

- Push buttons (for giving inputs)



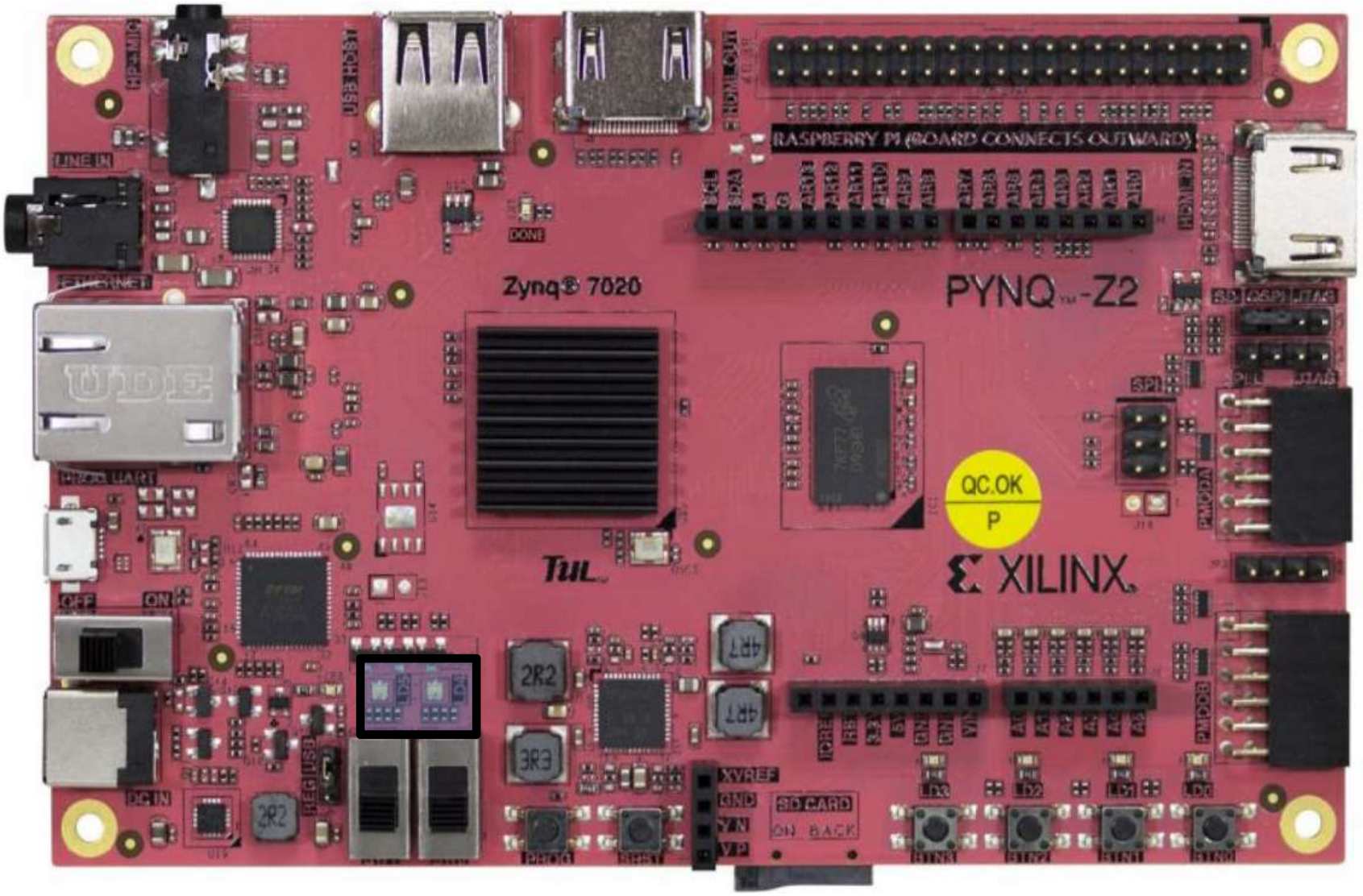
PYNQ-Z2 FPGA Board

- Monochrome LEDs (for seeing outputs)



PYNQ-Z2 FPGA Board

- RGB LEDs (for seeing outputs)



PYNQ-Z2 FPGA Board

- FPGA and processor (Xilinx Zynq Z-7020)

Xilinx Zynq Z-7020

- Formal part name ZYNQ XC7Z020-1CLG400C
- Contains a processor (called processing system) and an FPGA (called programmable logic)
- Processing system (PS)
  - 667 MHz ARM Cortex-A9 processor
  - Has two processing engines (called cores)

Xilinx Zynq Z-7020

- Programmable logic (PL)
  - Xilinx Artix-7 FPGA containing the following
  - A 2D array of configurable logic blocks (CLBs)
  - Each CLB has two logic slices
  - Each of the two logic slices has
    - Four six-input 64-entry look-up table (LUT) function generators (can store any six-input function)
    - Eight flip-flops
  - Each CLB has two cascadeable four-bit adders that can be configured into an eight-bit adder

Xilinx Zynq Z-7020

- Programmable logic (PL)
  - 13300 logic slices
  - 4.9 Mb of block RAM
    - Organized as 140 blocks of 36 Kb RAM
  - 220 DSP slices
    - Each slice has a 48-bit adder and an 18x25 multiplier

Field-programmable gate array

- Two-dimensional array of configurable logic/storage cells (or blocks) interconnected by programmable switches
  - Each cell/block can be programmed to carry out simple combinational functions
  - The interconnection switches can be programmed to decide how the result of one function is input to another function or stored in memory
  - Typically the programming bits are downloaded to the FPGA and the circuit is ready
    - Done in the field as opposed to in the fabrication facility; hence the name FPGA

Field-programmable gate array

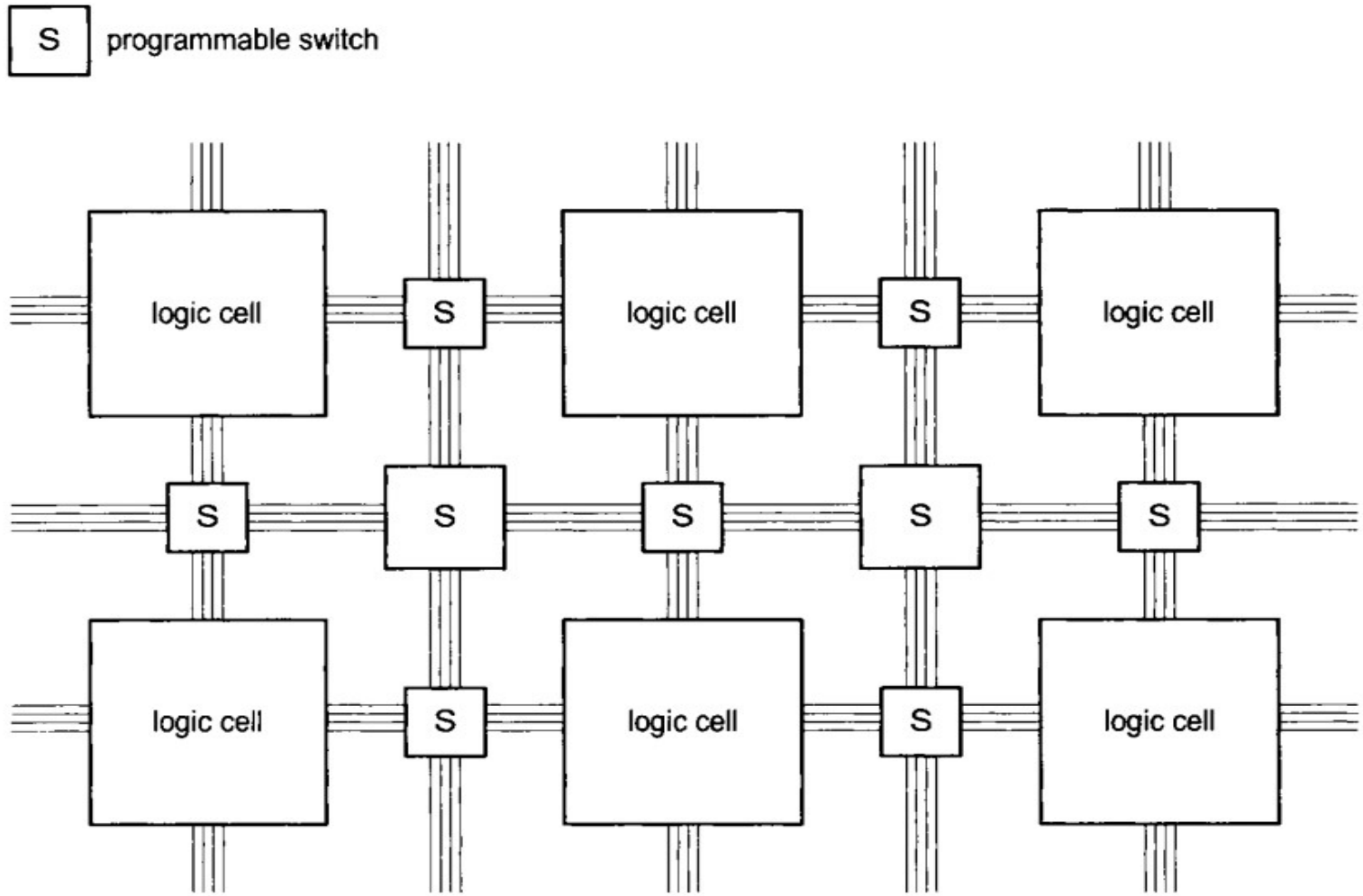


Image source: Chu. FPGA Prototyping by Verilog Examples

AMD/Xilinx Vivado

- Rest of the slides go over multiple examples demonstrating how to use Vivado
  - Can simulate Verilog code
  - Can synthesize hardware on FPGA
    - Synthesizing a hardware means programming the FPGA so that it models the desired hardware
    - A subset of the CLBs, RAM blocks, multipliers, and the switches will participate in implementing the specified hardware
- In this lab, you will do two syntheses
  - Full adder: Verilog to synthesis
  - Two-bit adder: Verilog to synthesis
  - Each synthesis task carries two marks

AMD/Xilinx Vivado

- Log in using your username and password
- Open a terminal by pressing Ctrl+Alt+t
- Create a new directory named *fpga_designs* using the command *mkdir fpga_designs*
- Change directory to fpga_designs using the command *cd fpga_designs*
- Enter the command *source /home/Xilinx/settings64.sh*
- Enter the command *vivado*
  - This will start vivado

AMD/Xilinx Vivado

- Click on *Create Project* and click *Next*
- Enter project name *Lab1A_fulladder*, check that project location ends with *fpga_designs*, tick *Create project subdirectory*, click *Next*
- Select *RTL project* and *Do not specify sources at this time*, click *Next*
- Select *Boards* on upper left corner and select *pynq-z2* by clicking on the *pynq-z2* row, click *Next*
- Review project name and board name, click *Finish*

AMD/Xilinx Vivado

- All project directories have been created; now we will write the Verilog program for the project
- In the *Sources* box on the left, right-click on *Design Sources*, select *Add Sources*
- Select *Add or create design sources*, click *Next*
- Click on *Create File*, enter *fulladder.v* as the name, click *OK*, click *Finish*

AMD/Xilinx Vivado

- Enter *fulladder* as the module name, in the I/O port definitions, enter the following (do not select Bus for any; click + to add rows), click *OK*

  a input, b input, cin input, sum output, cout output (one per row)

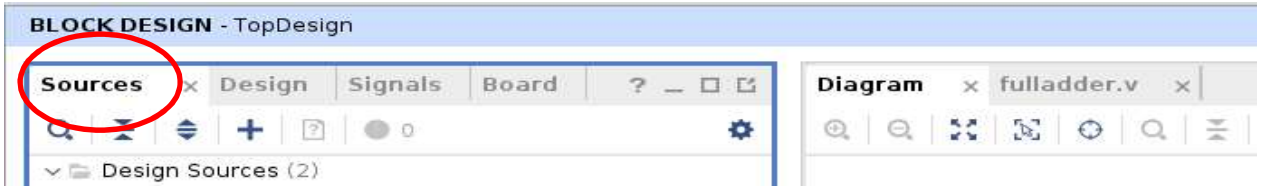
- In the *Sources* box, click on > before *Design Sources*, double-click on *fulladder.v*
  - On the right hand, *fulladder.v* will open with an empty module with all the ports
  - Type the following two lines in the module

```
assign sum = a^b^cin;
assign cout = (a & b) | (b & cin) | (cin & a);
```

AMD/Xilinx Vivado

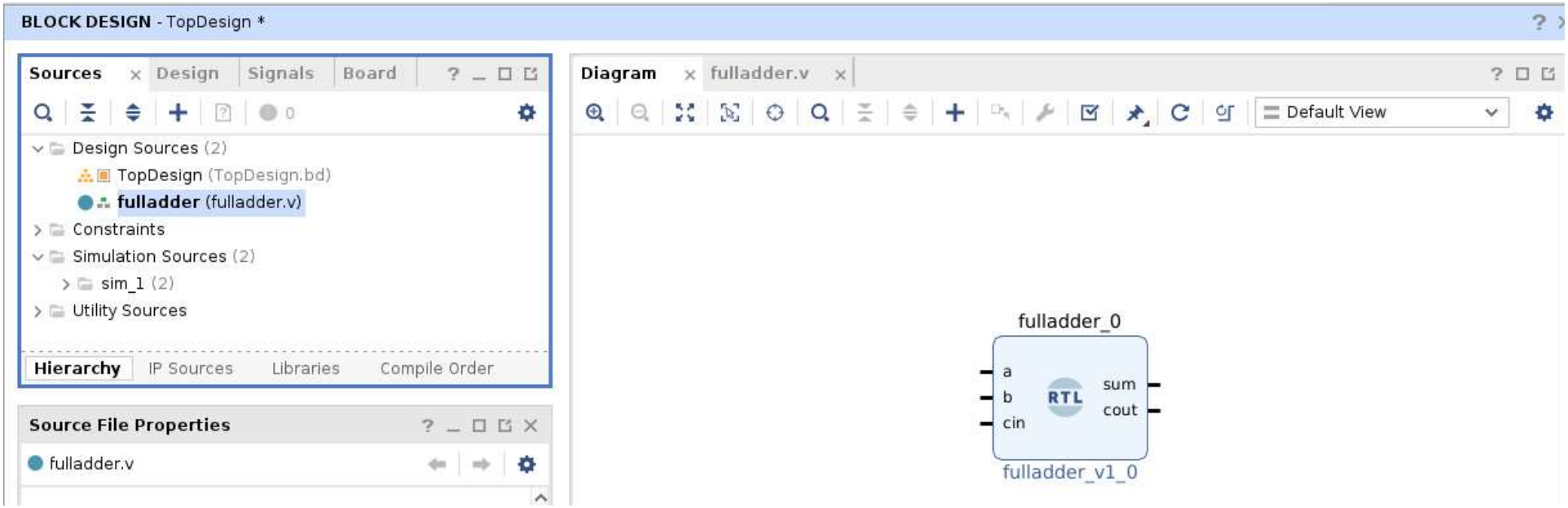
- Save the file by clicking the save icon; a green box will appear in the upper right corner if there is no error

- In the *Flow Navigator* menu on the left, click on *Create Block Design*, enter *TopDesign* as the *Design name*, and click *OK*
- In the *BLOCK DESIGN* box, click *Sources*



AMD/Xilinx Vivado

- Drag and drop *fulladder.v* into the big box titled *Diagram*
  - This will create a block named *fulladder_0*
  - We need to connect the ports to I/O



AMD/Xilinx Vivado

- Right-click on the pin coming out of port *a* and select *Create Port*, enter *Port name* A, don't change anything else, click *OK*
- Do the same for *b*, *cin*, *sum*, *cout* naming the ports B, CIN, SUM, COUT, respectively
- In the *Sources* box, right-click on *TopDesign*, select *Create HDL Wrapper*, select *Let vivado manage wrapper and auto-update*, click *OK*
  - This creates the top-level Verilog module for synthesis

Behavioral simulation in Vivado

- Our design is now complete and we will first do a behavioral simulation of the full adder
  - We have to add a test bench specifying the inputs
- Scroll down in the *Sources* box, right-click on *sim_1* under *Simulation Sources*, select *Add Sources*, select *Add or create simulation sources*, click *Next*
- Click on *Create File*, enter *fulladder_tb* as the file name, click *OK*, click *Finish*
- Since the simulation test bench has no input or output, click *OK*, click *Yes*

Behavioral simulation in Vivado

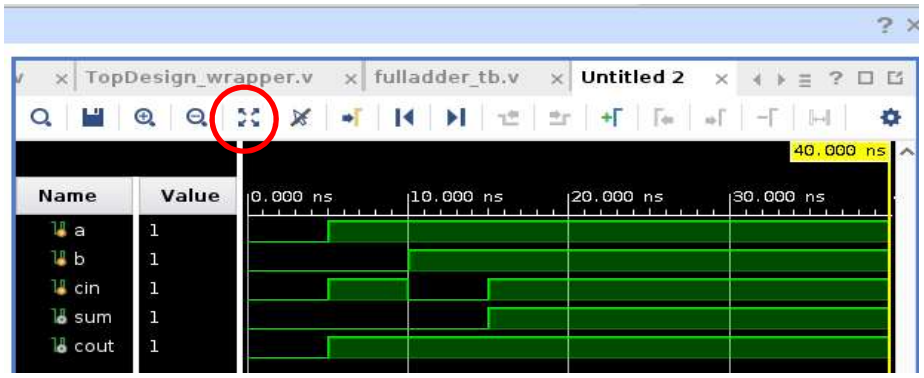
- Double-click on *fulladder_tb.v* after clicking on > before *sim_1*
- Edit the module as shown in the next slide
- Save the module after editing
- Now you should see *fulladder_tb* as the topmost module under *sim_1* in the *Sources* box

Behavioral simulation in Vivado

```
module fulladder_tb;
   reg a, b, cin;
   wire sum, cout;
   TopDesign_wrapper uut(a, b, cin, cout, sum);  // Check the order
   initial begin
      #40
      $finish;
   end
   initial begin
      a=0; b=0; cin=0;
      #5
      a=1; b=0; cin=1;
      #5
      a=1; b=1; cin=0;
      #5
      a=1; b=1; cin=1;
   end
   always @(sum or cout) begin
      $display("<%d> a=%b, b=%b, cin=%b, sum=%b, cout=%b\n", $time,
a, b, cin, sum, cout);
   end
endmodule
```

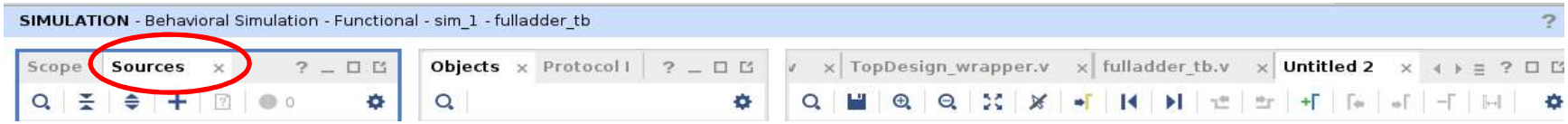
Behavioral simulation in Vivado

- Select *fulladder_tb* from under *sim_1*
- Click on *Run Simulation* in the left hand *Flow Navigator* menu and select *Run Behavioral Simulation*
- If you scroll up in the *Tcl Console* at the bottom, you will see the simulation output
- Click the *Untitled* tab in the rightmost box and click on *Zoom Fit* to see the signal waveforms



Synthesis in Vivado

- Next, we will synthesize our design and map it to the FPGA
  - For this, we need to connect the inputs and outputs of our design to switches/buttons and LEDs in the PYNQ-Z2 board
- Click on *Sources* within the big *Simulation* box



- Click on > before *Constraints*, right-click on *constrs_1*, select *Add Sources*, select *Add or create constraints*, click *Next*

Synthesis in Vivado

- Click *Create File*, enter *Constraints* as the file name, click *OK*, click *Finish*
- Under *constrs_1*, you should see *Constraints.xdc*
- Double-click *Constraints.xdc*
  - Opens an empty file
  - Copy and paste the lines corresponding to two switches, one push button, and two LEDs from the *PYNQ-Z2_v1.0.xdc* file available on course homepage
    - Change these lines as shown in the next slide
    - Save the file by clicking the save icon

Synthesis in Vivado

```
##Switches
set_property -dict { PACKAGE_PIN M20   IOSTANDARD LVCMOS33 }
[get_ports { A }]; #IO_L7N_T1_AD2N_35 Sch=sw[0]
set_property -dict { PACKAGE_PIN M19   IOSTANDARD LVCMOS33 }
[get_ports { B }]; #IO_L7P_T1_AD2P_35 Sch=sw[1]
##Buttons
set_property -dict { PACKAGE_PIN D19   IOSTANDARD LVCMOS33 }
[get_ports { CIN }]; #IO_L4P_T0_35 Sch=btn[0]
##LEDs
set_property -dict { PACKAGE_PIN R14   IOSTANDARD LVCMOS33 }
[get_ports { SUM }]; #IO_L6N_T0_VREF_34 Sch=led[0]
set_property -dict { PACKAGE_PIN P14   IOSTANDARD LVCMOS33 }
[get_ports { COUT }]; #IO_L6P_T0_34 Sch=led[1]
```

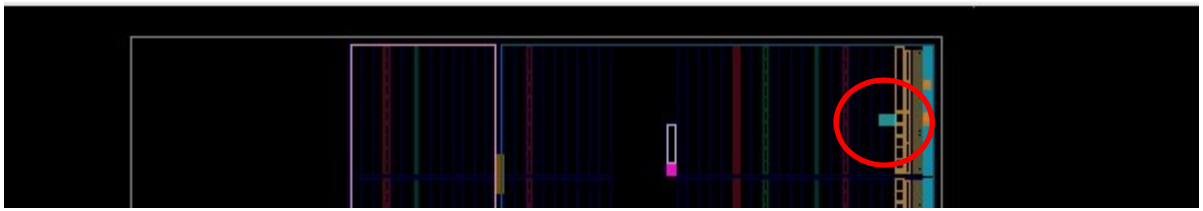
- Remove # from the beginning of the *set_property* lines
- We are connecting the switches to A and B (SW0 and SW1), one push button to CIN (BTN0), and two LEDs to SUM and COUT (LD0 and LD1)

Synthesis in Vivado

- Having connected the inputs A, B, CIN and the outputs SUM and COUT to switches/buttons/LEDs, we are ready to synthesize
- In the *Sources* box, select *TopDesign_wrapper* under *Design Sources*
- Click on *Run Synthesis* in the *Flow Navigator* menu on the left, click *OK*
- Once synthesis completes, when prompted to *Run Implementation*, click *OK*, click *OK*
- When prompted to *Open Implemented Design*, click *OK*

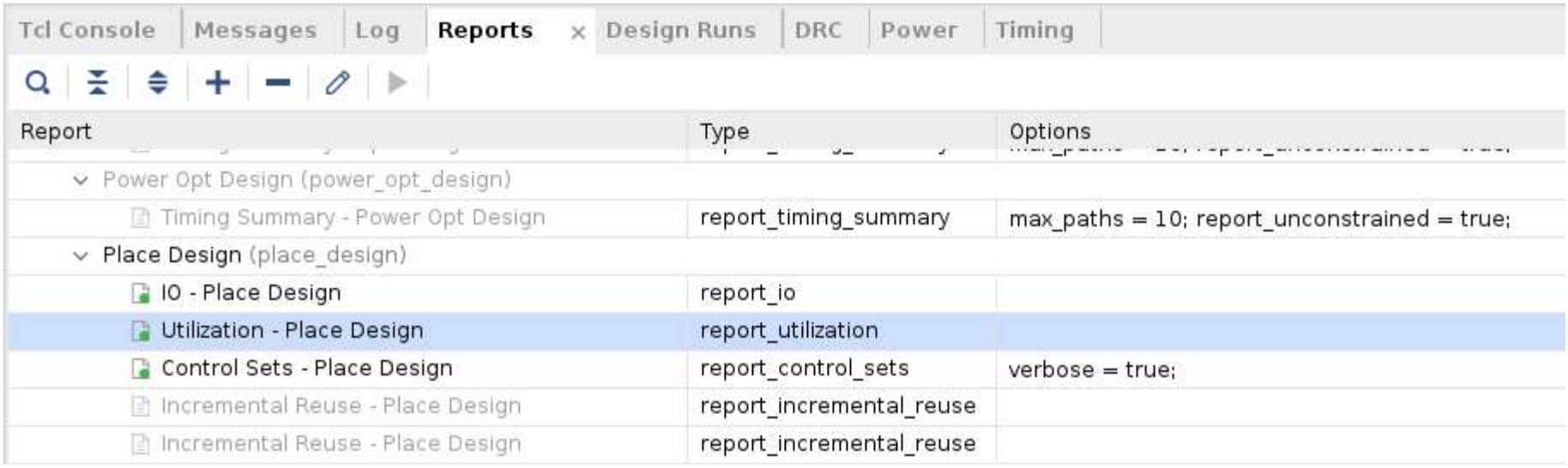
Synthesis in Vivado

- You will see the entire floor of the Zynq Z-7020 chip
- Zoom on the area of cyan color (this area has the synthesized design)
  - Select the area using mouse



Synthesis in Vivado

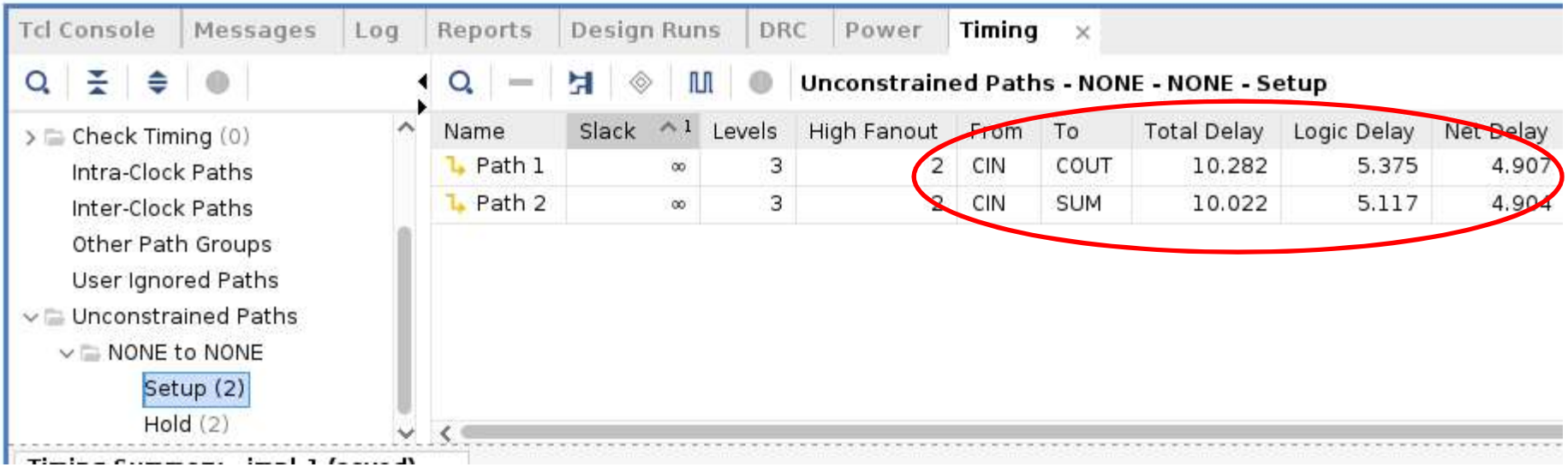
- To see the resource utilization, click on *Reports* in the bottom panel, double-click on the row mentioning *Utilization-Place Design*



- The report shows the number of LUTs, flip-flops, etc. used to synthesize the design

Synthesis in Vivado

- Before programming the FPGA, let us examine the critical path delay of the design
  - Select *Timing* from bottom panel, select *Setup* from under *NONE to NONE* under *Unconstrained Paths*



Synthesis in Vivado

- Depending on the quality of synthesis in your case, the delays may vary
  - In my case, *cin* to *cout* delay (in ns) is slightly higher than *cin* to *sum* delay
    - The total delay is also divided into logic and wire/net delays

Timing simulation in Vivado

- Behavioral simulation assumes that the computation takes no time
- Timing simulation incorporates the path delays obtained from synthesis
- Let us do a timing simulation of the implemented design
  - Select *fulladder_tb* from under *Simulation Sources* → *sim_1* in the *Sources* box, click *Run Simulation* from the *Flow Navigator* menu on the left, select *Run Post-implementation Timing Simulation*

Timing simulation in Vivado

- Timing simulation output is visible in the Tcl Console and the waveforms are visible in the Untitled tab (which you should zoom)
  - Notice that several outputs are incorrect
  - This is because the input is changed within 5 time units, which is much smaller than the critical path delay of more than 10 time units
- Change #5 to #15 in *fulladder_tb.v* and simulate for 80 time units i.e., change #40 to #80 in the first *initial* block, save the file, click *Run Simulation*, select *Run Post-implementation Timing Simulation*, click *OK* when asked to relaunch simulation
  - Now all simulation results are correct

Execution on FPGA

- Now, as the last step, we will run our design on the FPGA
- Click *Generate Bitstream* in the *Flow Navigator* menu on the left, click *OK*
- Once Bitstream generation completes, select *Open Hardware Manager*, click *OK*
- At the top, you will see the message *No hardware target is open*
- Connect the power connection port of the FPGA board to your computer using the USB cable, switch it on (red LED should glow), click *Open target*, select *Auto Connect*

Execution on FPGA

- In the Hardware box (upper left), select *Xilinx_tcf/Xilinx/1234-tulA*, click *Program device*, click *Program*, a green LED on the FPGA board glows on successful programming
- Input A and B through the two switches and holding down the rightmost push button (BTN0) will set CIN to 1
- Check output in the rightmost two LEDs
  - SUM is the rightmost LED and COUT is the next LED

Exiting Vivado

- Keep the FPGA board on and connected
- From the top menu, click *File*, select *Exit*, click *OK*
- Switch off the FPGA board and disconnect
- To get credits for the first assignment, you should show the following to a TA
  - Behavioral simulation output
  - Critical path delay report
  - Post-implementation timing simulation output with old and new testbenches
  - Execution on FPGA with all eight possible inputs

Assignment#2: Two-bit adder

- Assignment#2
  - Synthesize a two-bit adder on the FPGA
    - Inputs A0, A1 should come from two switches
    - Inputs B0, B1 should come from two push buttons
    - Connect CIN of the first full adder to another push button for a third optional input
    - Outputs S0 should appear in the rightmost LED, S1 in the next LED, and COUT in the next LED
  - Do behavioral simulation to check functional correctness
  - Study and optimize critical path delay
  - Do post-implementation timing simulation with and without critical path optimization

Assignment#2: Two-bit adder

- Follow the same procedure
  - Name the project as *Lab1B_twobit_adder*
  - Add fulladder.v source
  - Will not write the Verilog program for a two-bit adder; instead, will create the block diagram of it by connecting two full adders in ripple-carry configuration
    - After editing and saving *fulladder.v*, when creating the block design, drag and drop *fulladder.v* twice
    - This will create two instances of the *fulladder* module
    - Right-click on *cout* of *fulladder_0*, select *Make Connection*, select *cin* of *fulladder_1*
    - Create ports A0, A1, B0, B1, CIN, S0, S1, COUT

Assignment#2: Two-bit adder

- Create HDL wrapper of TopDesign (assuming the name of the block design is TopDesign)
- Create a simulation testbench with the following schedule

```
CIN = 0;
A1 = 0; A0 = 0; B1 = 0; B0 = 0;
#5
A1 = 1; A0 = 0; B1 = 0; B0 = 1;
#5
A1 = 1; A0 = 1;
#5
B1 = 1; B0 = 1;
#5
$finish;
```

Assignment#2: Two-bit adder

- Do behavioral simulation
- Create *Constraints.xdc* as per the connections of A0, A1, B0, B1, CIN, S0, S1, COUT with the switches, buttons, and LEDs
- Synthesize, implement, generate bit stream, execute on FPGA
- Check resource report and timing report
  - Click Reports from top horizontal menu bar
  - Find the critical path delay (mine is around 11)
- Do post-implementation timing simulation
  - See whether the results are correct

Optimizing the critical path

- To force the compiler optimize the critical path, we need to put a constraint on the maximum allowed critical path delay
  - This is done by inserting the following line at the end of *Constraints.xdc*

    set_max_delay -datapath_only -from [all_inputs] -to [all_outputs] 8.000
  - Save *Constraints.xdc*
  - Relaunch synthesis and then implementation

Optimizing the critical path

- On completion, select *View Reports*, click *OK*
- At the top, you will see the message: *Implemented design is out-of-date*... Click on *Reload*
- Select *Timing* in the bottom panel, from the left hand menu click on *Setup* under *Other Path Groups*
  - Note that the slack is negative and the critical path delay has overshot the constraint by that amount
- Change the max delay to 9.7 in *Constraints.xdc* and relaunch synthesis and implementation
  - Adjust the delay if the slack is still negative

Optimizing the critical path

- Once a positive slack is obtained and the slack is small enough, redo post-implementation timing simulation
  - Adjust the delays in the testbench to be an integer more than the max delay
- To get credits, show all your work to a TA