Lab 3:Complex Logic Design with Vivado Logic Analyzer

National Chiao Tung University
Chun-Jen Tsai
04/17/2018

Lab Description

- In this lab, you must design an IP that implements the entire `compute_sad()` loops in hardware
  - The pixel data stored into the IP must be stored inside the IP and reused as much as possible in order to reduce the communication overhead
  - To assist debugging of your circuit, you will learn how to use the Vivado Integrated Logic Analyzer (ILA) IP to intercept and analyze some runtime signals of your design
- Make a demo to your TA on 5/2

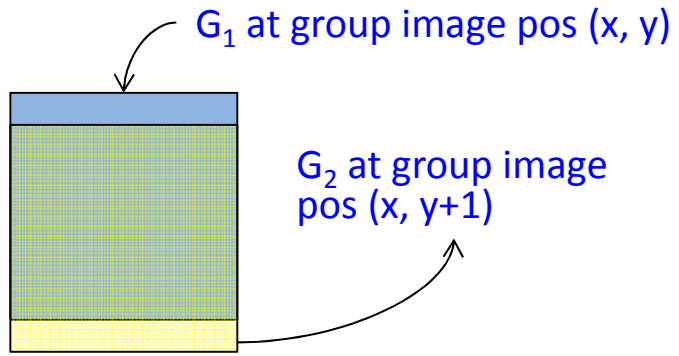
The Old `match()` Function

- During SAD computation, we slide a 32×32 face image through the group image
  - "Face" never changes → copy it once and store it in the HW IP
  - For any two consecutive compute_sad(), the group pixels used are mostly overlapping!

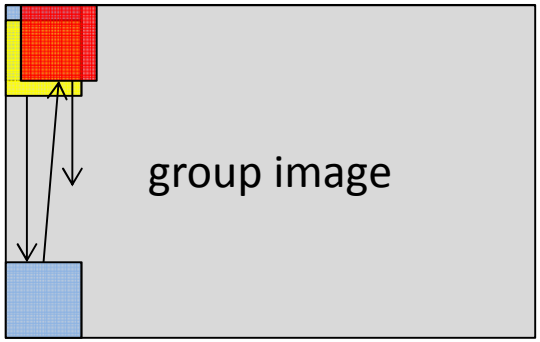
```
int32 match(CImage *group, CImage *face, int *posx, int *posy)
{
  int32  row, col, sad, min_sad;

  min_sad = 256*face->width*face->height;
  for (row = 0; row < group->height-face->height; row++) {
    for (col = 0; col < group->width-face->width; col++) {
      /* trying to compute the matching cost at (col, row) */
      sad = compute_sad(group->pix, group->width, face->pix,
                        face->width, face->height, row, col);
      /* if the matching cost is minimal, record it */
      if (sad <= min_sad), min_sad = sad, *posx = col, *posy = row;
    }
  }
  return min_sad;
}
```

Reusing the Group Pixels in HW IP

- If we perform the block-matching process following a top-down scan path, for most of the compute_sad(), we only need to copy 32 pixels into the HW IP:



G_1 and G_2 only differ by 32 pixels!



Top-down matching path, only the top blocks (in red) require copying of 32×32 pixels into the HW IP

New Behavior for `match()`

- To reduce the number of pixel copies, we can rewrite the `match()` function:
  - The face image must be copied into the HW logic before this new `match()` function can be called

```
int32 match(CImage *group, int *posx, int *posy)
{
  int32  row, col, sad, min_sad;

  min_sad = 256*face->width*face->height;
  for (col = 0; col < group->width-face->width; col++) {
    for (row = 0; row < group->height-face->height; row++) {
      /* trying to compute the matching cost at (col, row) */
      sad = compute_sad(group->pix, group->width, row, col);

      /* if the matching cost is minimal, record it */
      if (sad <= min_sad), min_sad = sad, *posx = col, *posy = row;
    }
  }
  return min_sad;
}
```

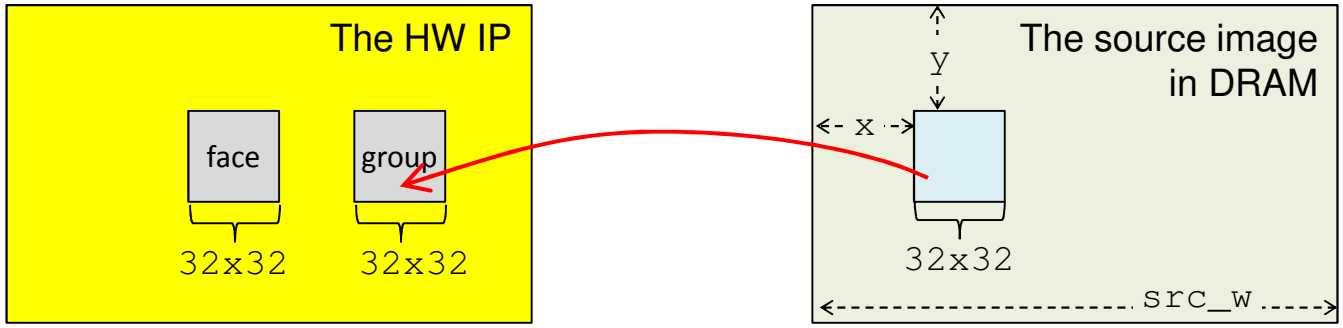
New `compute_sad()`

- The new `compute_sad()` only copy minimal pixels into the HW IP for matching a block:
  - Variables in bold italics fonts are HW registers

```
int32 compute_sad(uint8 *group, int width, int row, int col);
{
  if (!row) {
    for (y = 0; y < 32; y++) {
      /* send 32x32 pixels into the HW IP */
      *reg_bank = y;
      memcpy((void *) group_regs, group+y*width+col, 32);
    }
  } else { /* row != 0 */
    /* send the last row of 32 pixels into the HW IP */
    *reg_bank = (row - 1) % 32;
    memcpy((void *) group_regs, group+(row+31)*width+col, 32);
  }

  *hw_active = 1;
  while (*hw_active)  ; /* busy waiting, not good but faster */
  return *sad_result;
}
```

Image Block Copy Function (1/2)

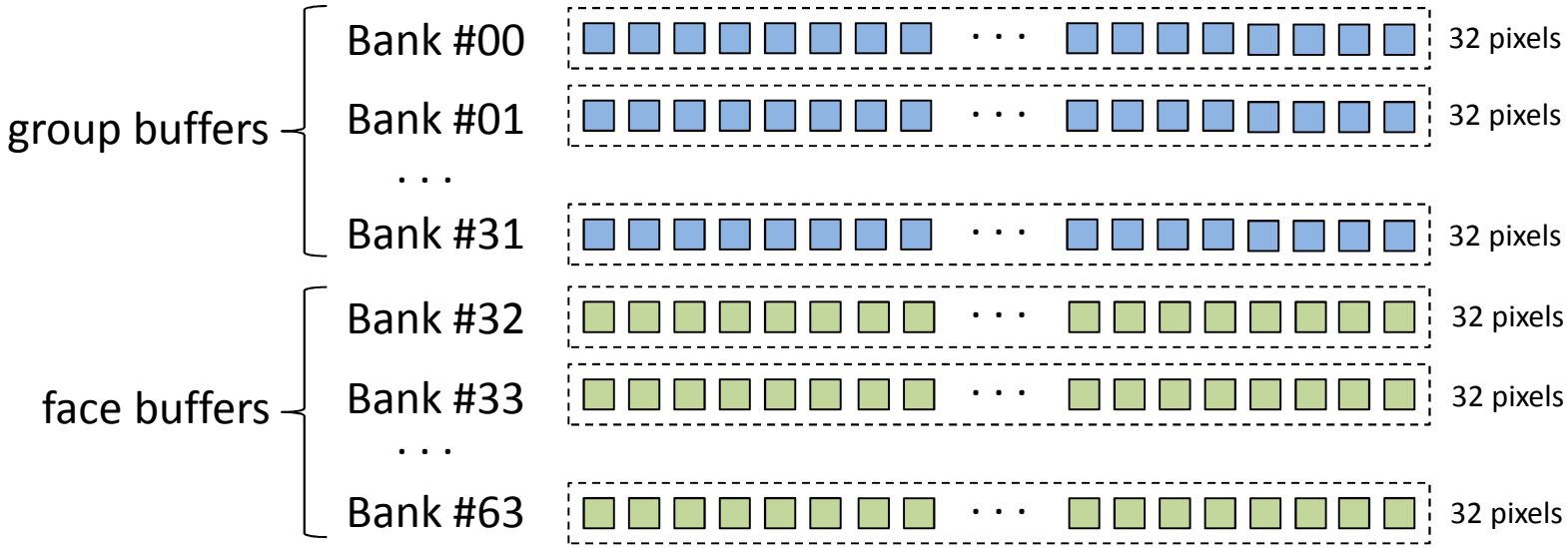
- Copy a rectangular area in a 2D image is a very important function for image/video processing



- However, the default I/O address space of an Vivado IP under AXI4-Lite bus has only 2048 bytes → too small, we need $32 \times 32 \times 2 = 2048$ bytes plus control registers!

Image Block Copy Function (2/2)

- We can define the IP to have 8 32-bit buffer registers, `R0 ~ R7`, and a bank ID register, `reg_bank`.
  - The 32×32 group pixel buffer is divided into 32 banks (#00 ~ #31) and so is the face buffer (#32 ~ #63)
  - Data written into `R0 ~ R7` will be redirected to different banks in group (or face) based on the bank# in `reg_bank`



Hardware IP Internal Buffer Design

- The hardware IP should have two 32×32 pixel buffers to store the group and the face pixels
- You can use registers to implement buffers in this lab, but they are expensive:
  - Wasting the logic cells (LUTs and flip-flops) of the FPGA
  - Making the place-and-route time very long
- As an alternatives, you can use BRAM to create buffers → we will force you to do so in the next Lab
- Note that the 32 banks in the group buffer must form a circular buffer to avoid shifting row pixels! The top of the buffer is at row `(y%32)`, where `y` is the matching window row position in the group image.

HW IP Specification

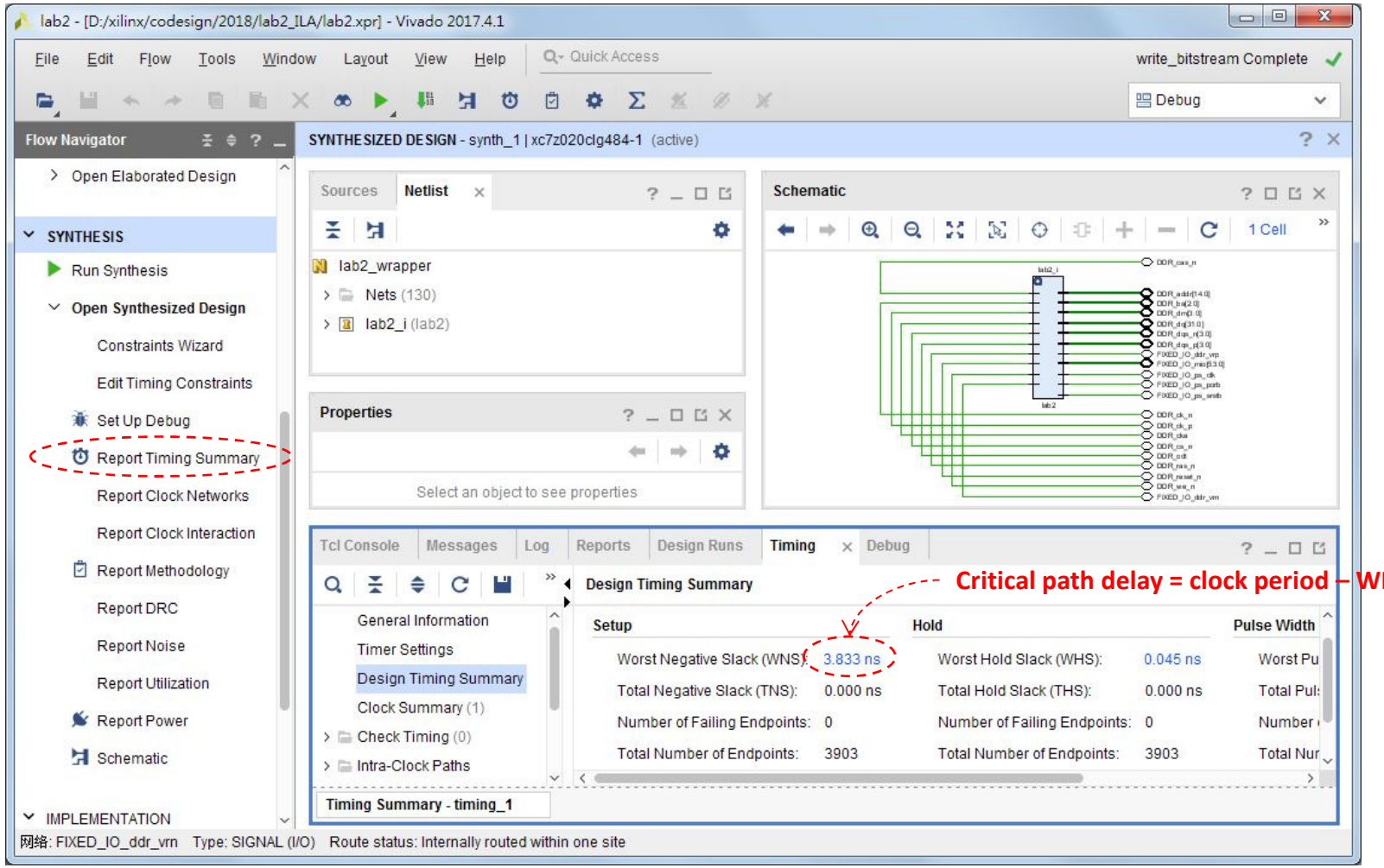
- The HW IP should have the following registers:
  - [i] `R0 ~ R7`: the buffer interface registers
  - [i] `reg_bank`: the bank selection register
  - [i/o] `hw_active`: the HW IP trigger/status register
  - [o] `min_sad`: the minimal SAD value of the motion vector

About The Demo

- For your demo, you should tell the TAs the logic usage and the timing information about your design
  - Logic usage means the numbers of the LUTs, flip-flops, and BRAMs (if you know how to use it) used in your design
  - Timing information is the length of the critical path (in nanoseconds) of your design
- Your grade will be based on the tradeoff between the area usage and the performance of your IP

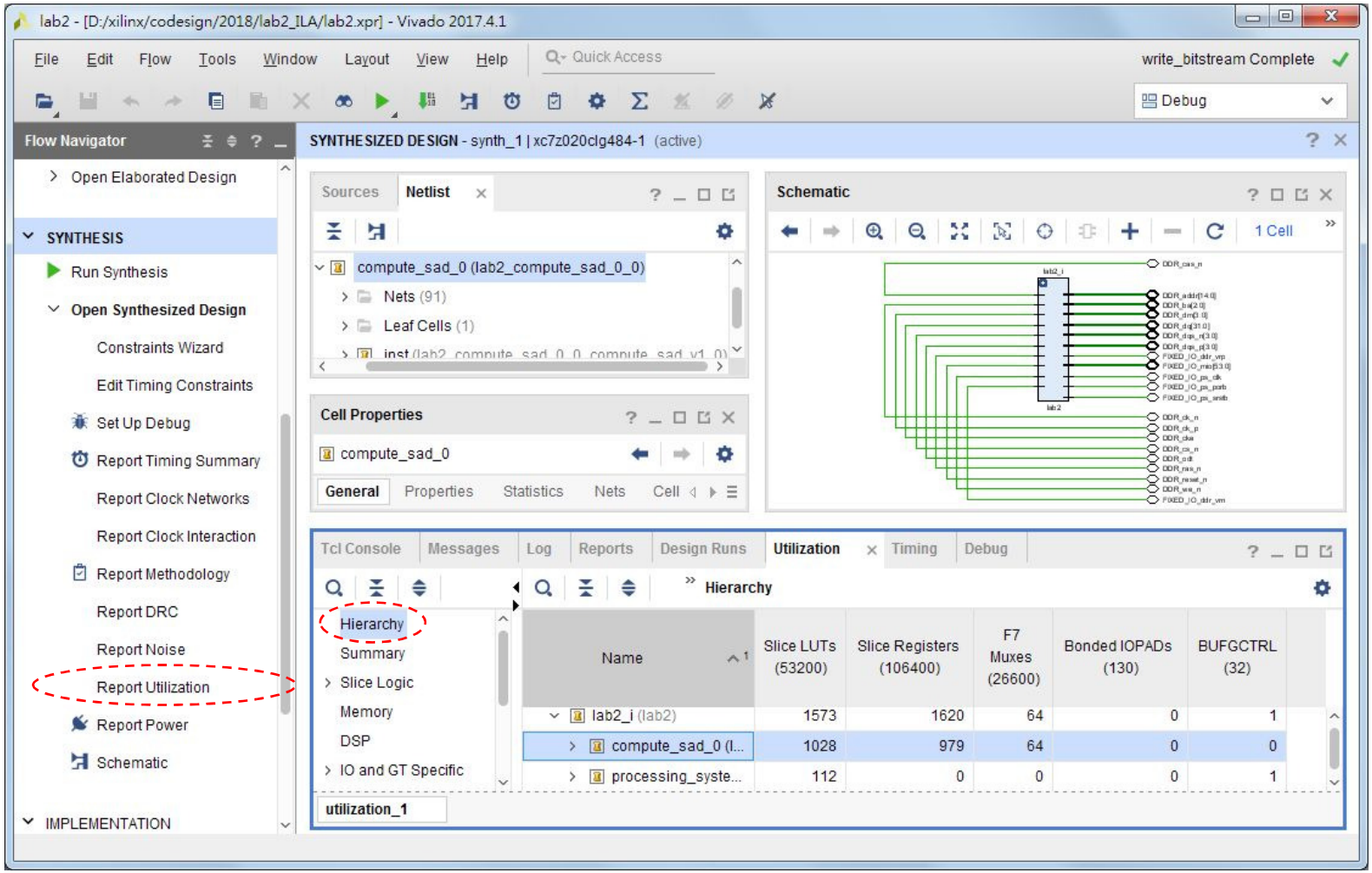
Check Timing Report

- "Open Synthesized Design" in Vivado show you reports:



Check Utilization Report

- Check # of LUTs, Registers, Muxes, DSP's, & BRAMs:

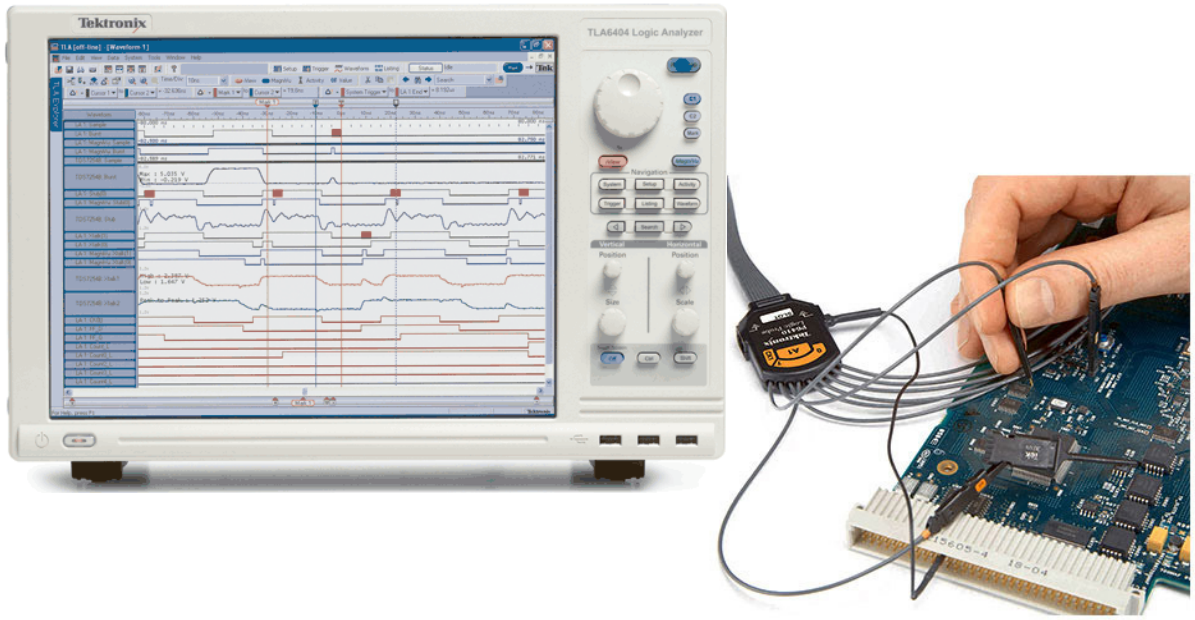


Suggestions on Your Design Process

- Step 1: Create an IP that do 4×4 block matching, and use the full-system simulation flow to verify the correctness of your IP
- Step 2: Scale up the IP in step 1 to do 32×32 block matching, and use the Vivado Logic Analyzer to catch any bugs running on ZedBoard in real-time.

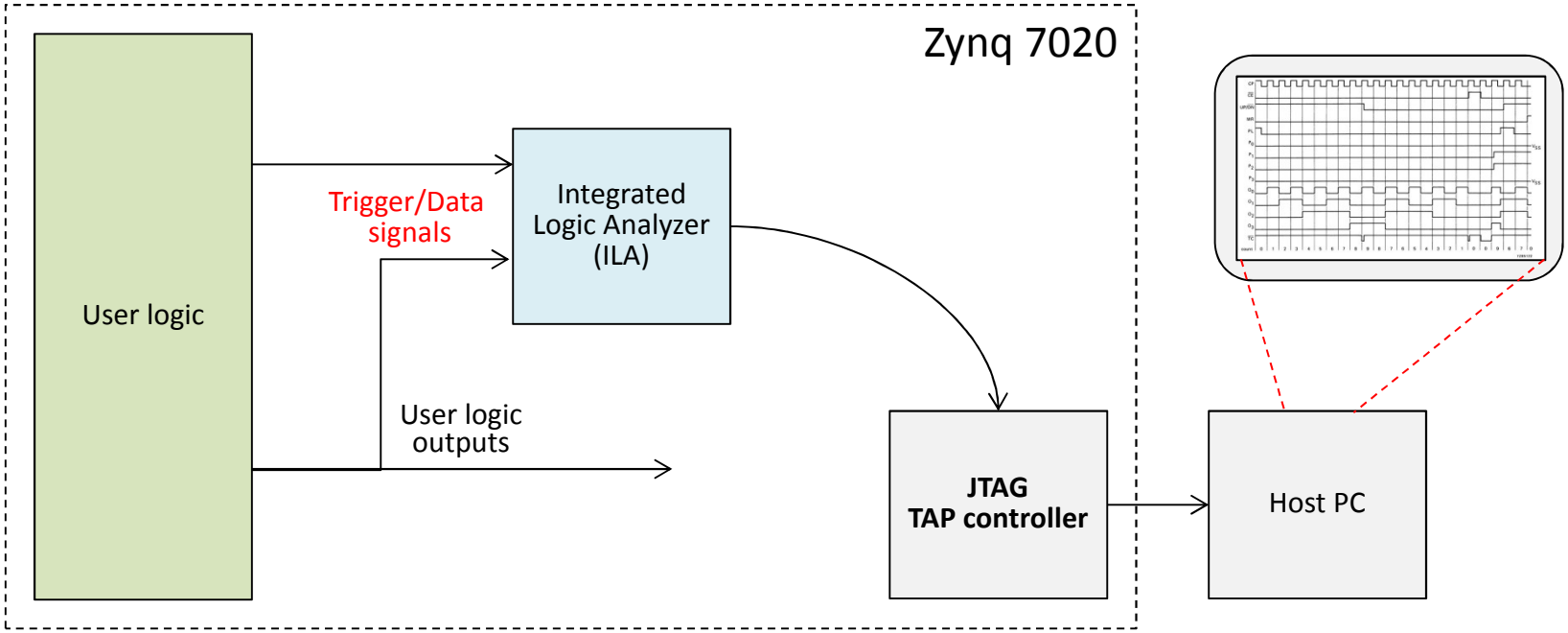
Debugging Your Circuit in Real-Time

- Full-system simulations for complex logic and software behaviors would take too much time
- In the good old days, for real-time debugging of a digital circuit, we must use a logic analyzer device



Vivado Integrated Logic Analyzer

- Vivado Integrated Logic Analyzer (ILA) is an IP that can be integrated into the hardware platform so that some signals in the user IP's can be intercepted and saved in a trace file for analysis



Debug Your Circuit in Real-Time

- To debug your logic in real-time, you must first “mark” your logic signals for debugging using one of the following three methods:
  - Using the “synthesis attribute” syntax in Verilog-2001
  - Using the Vivado GUI IDE
  - Using the TCL command console (we don’t use TCL commands in this course)
- After marking the signals to debug, you must set up the debug wizard before you use the Hardware Manager to capture the signals at runtime

Mark Debug Signals Using Verilog

- In Verilog-2001, you can set the synthesis attributes of a signal, for example:

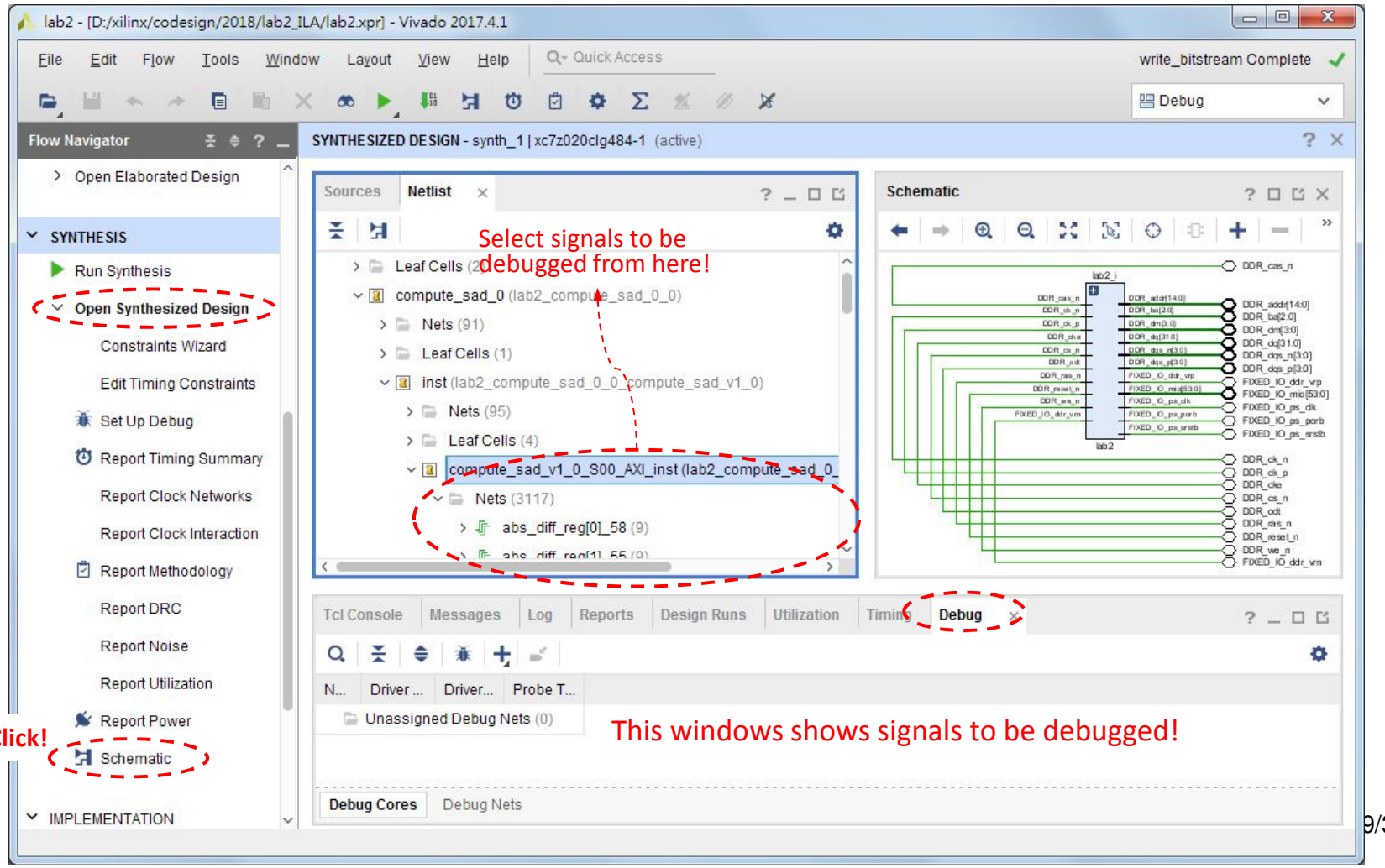
```
(* mark_debug = "true" *) wire my_signal
```

  This will turn on the "debug" attribute of `my_signal`.

- In Vivado, if your logic has signals with the debug attribute enabled, then:
  - The signals will not be "optimized-out" by the logic synthesizer
  - Vivado will insert an ILA IP into the synthesized design to monitor and capture these signals at runtime

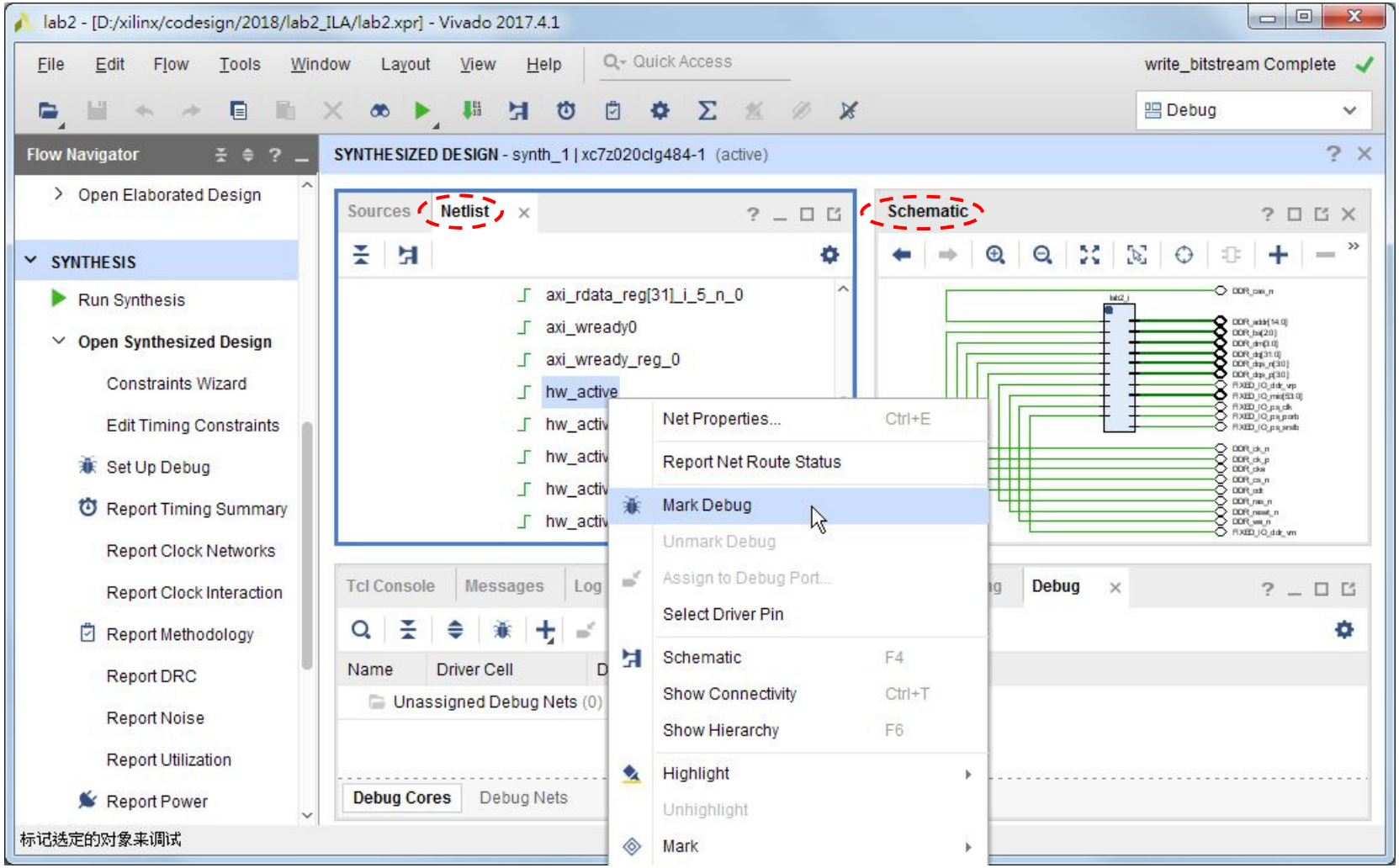
Mark Debug Signals Using GUI (1/2)

- Using Lab2 as an example, open the synthesized design:



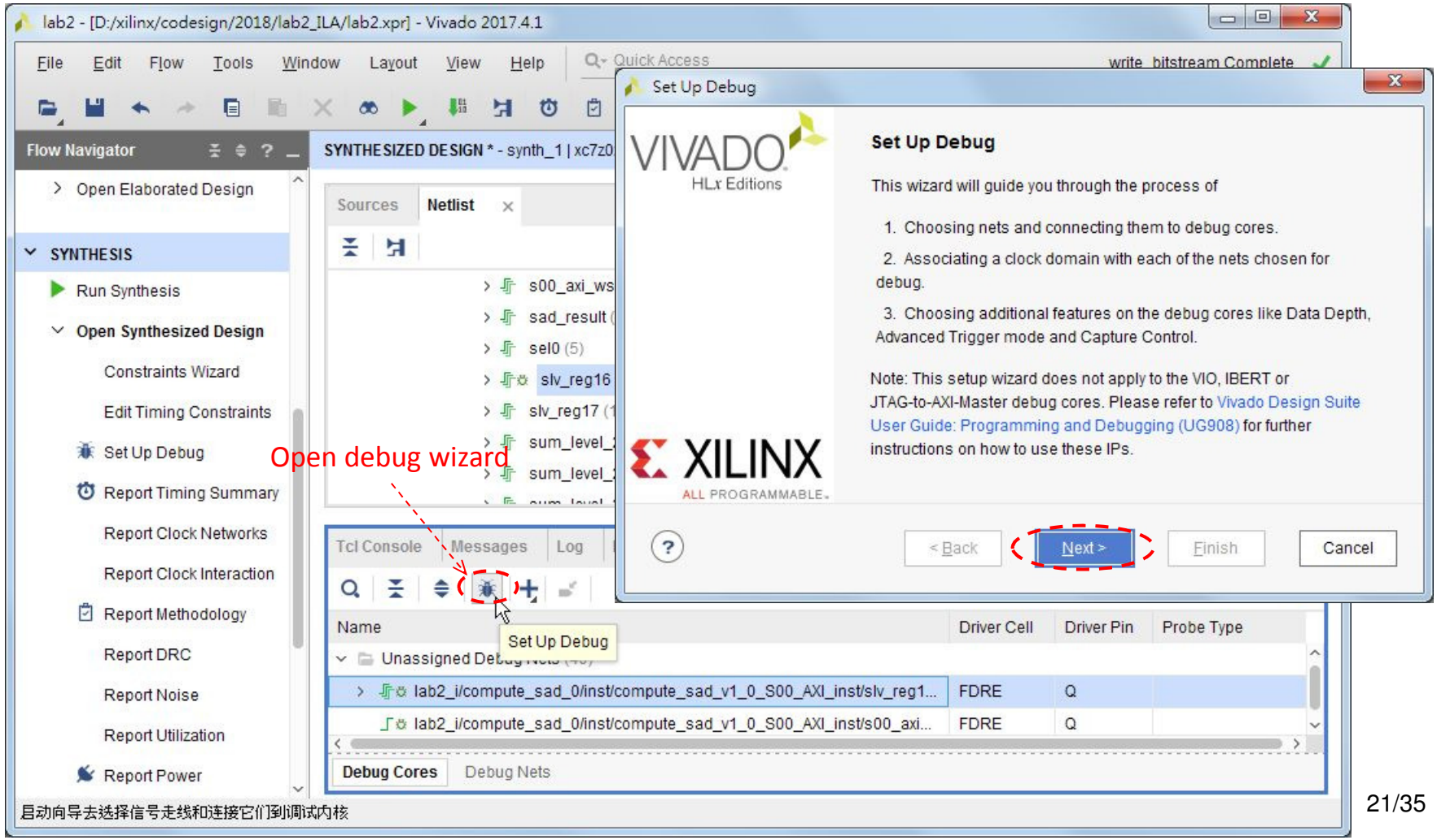
Mark Debug Signals Using GUI (2/2)

- Mark the signal in the “Netlist” or “Schematic” windows:



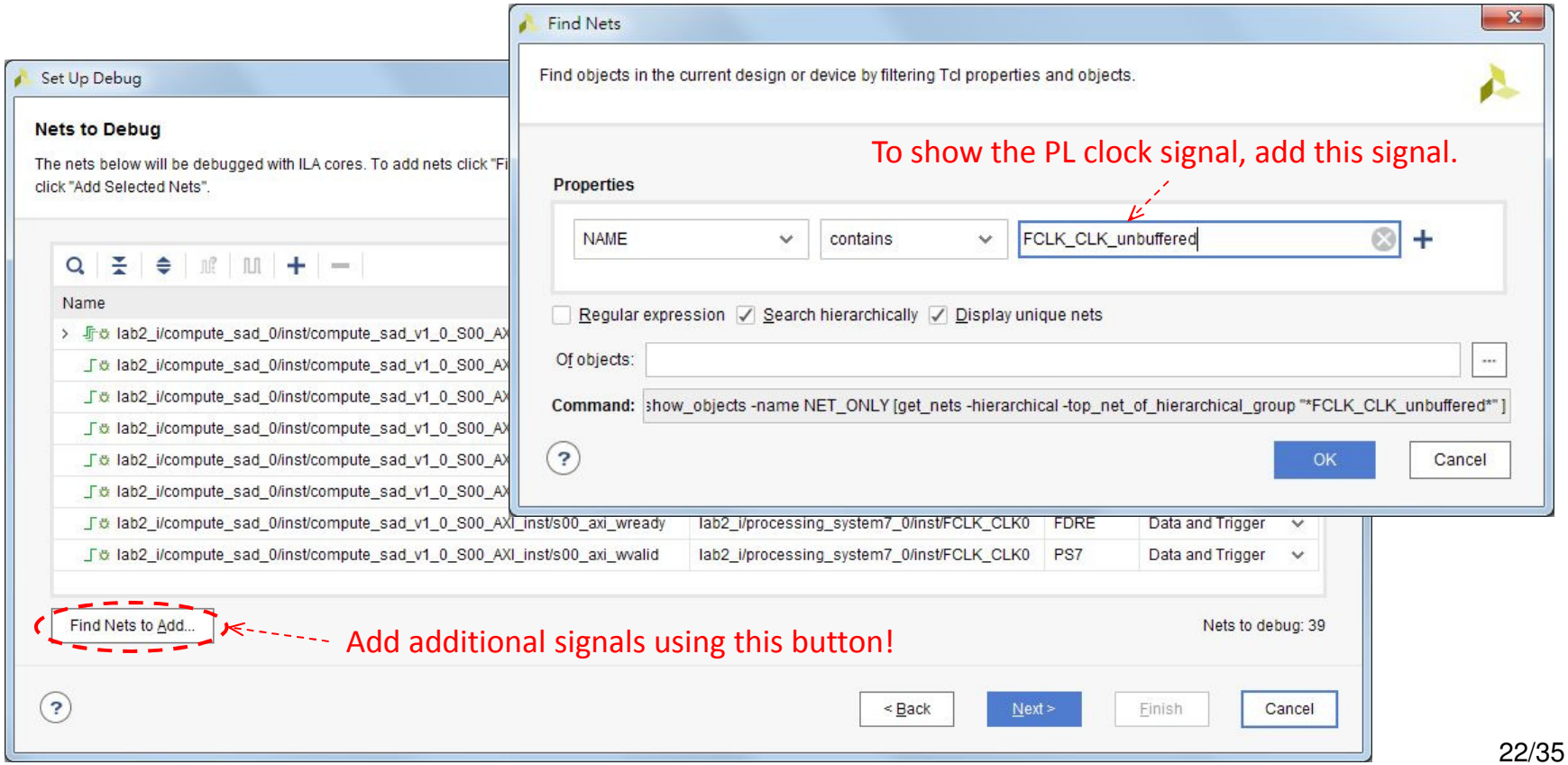
Set Up the Debug Wizard

- Open the “Set Up Debug” wizard:



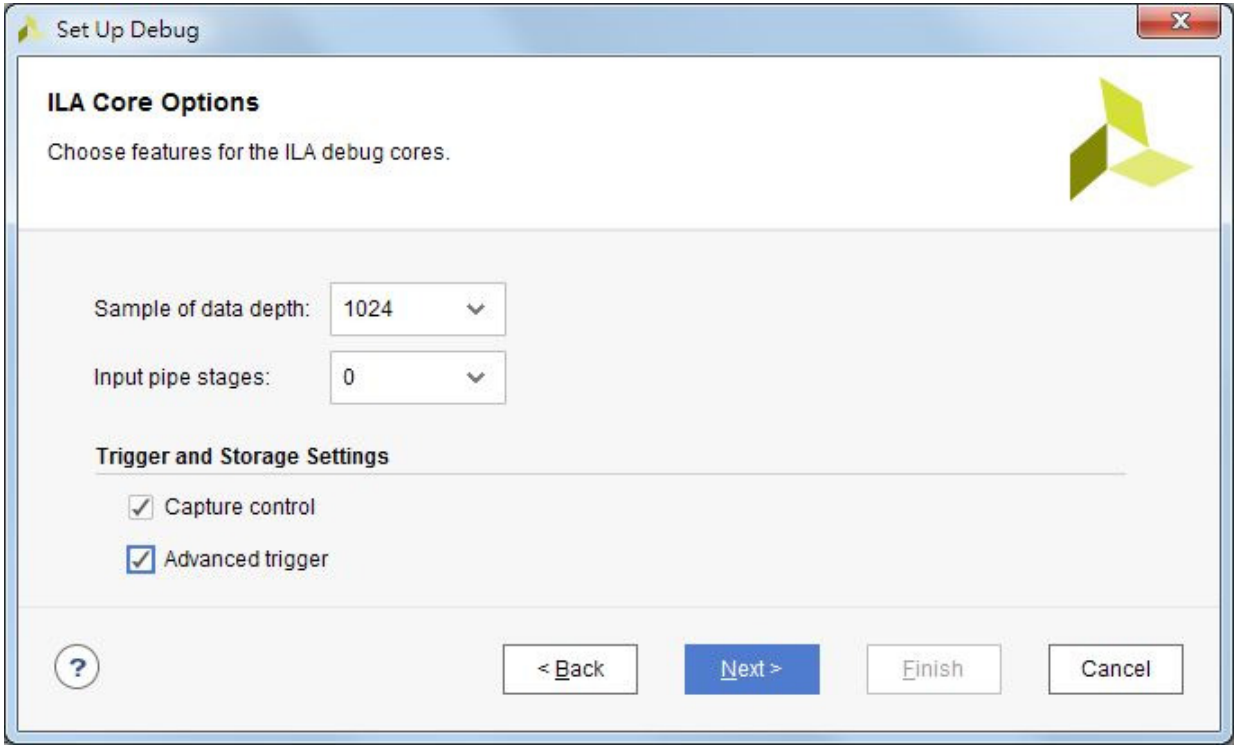
Check Nets to Be Debugged Again

- You can add any missing signals in this dialog box
  - Note: some signals in your Verilog code may be missing due to the optimization process of the logic synthesizer!



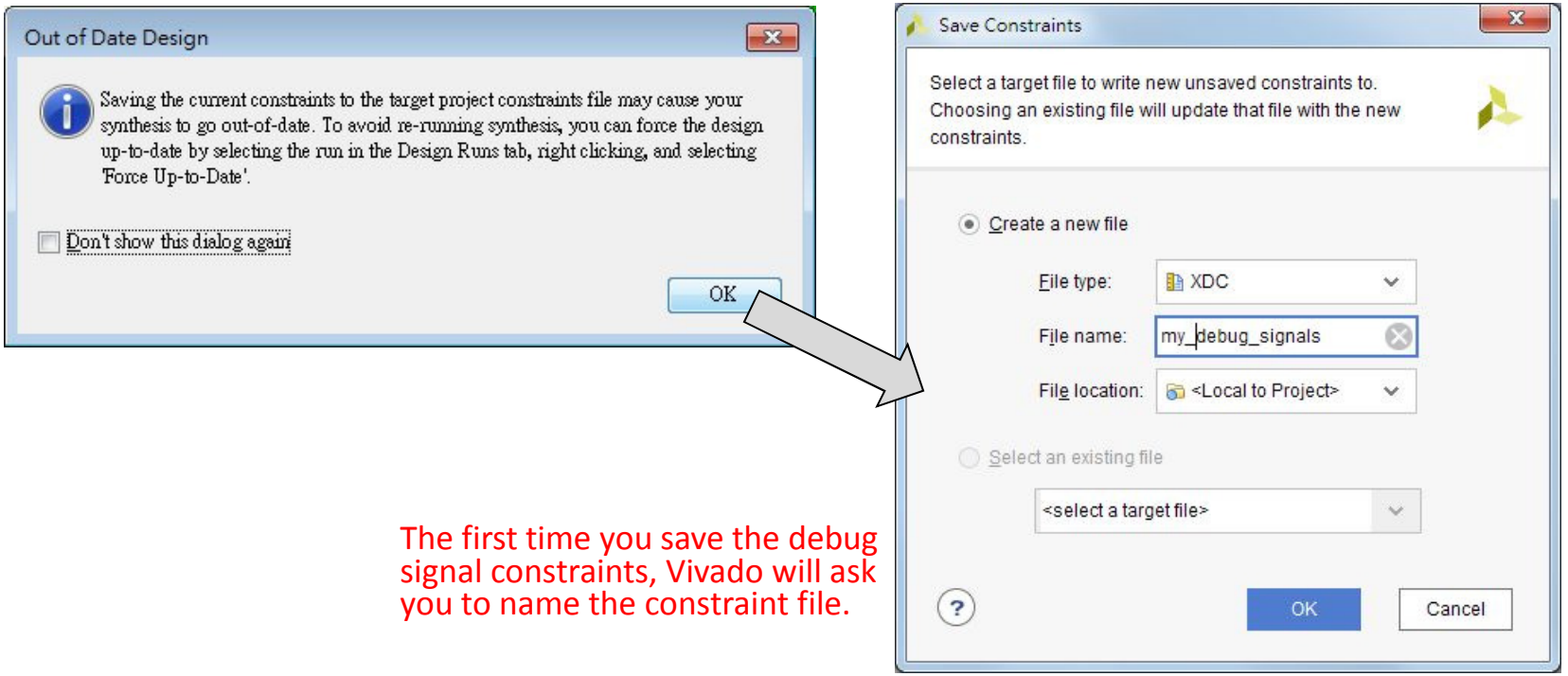
Modify Trigger Options

- You can check both the “Capture control” and the “Advanced trigger” boxes



Save the New Debug Constraints

- Select the menu items: File → Save Constraints

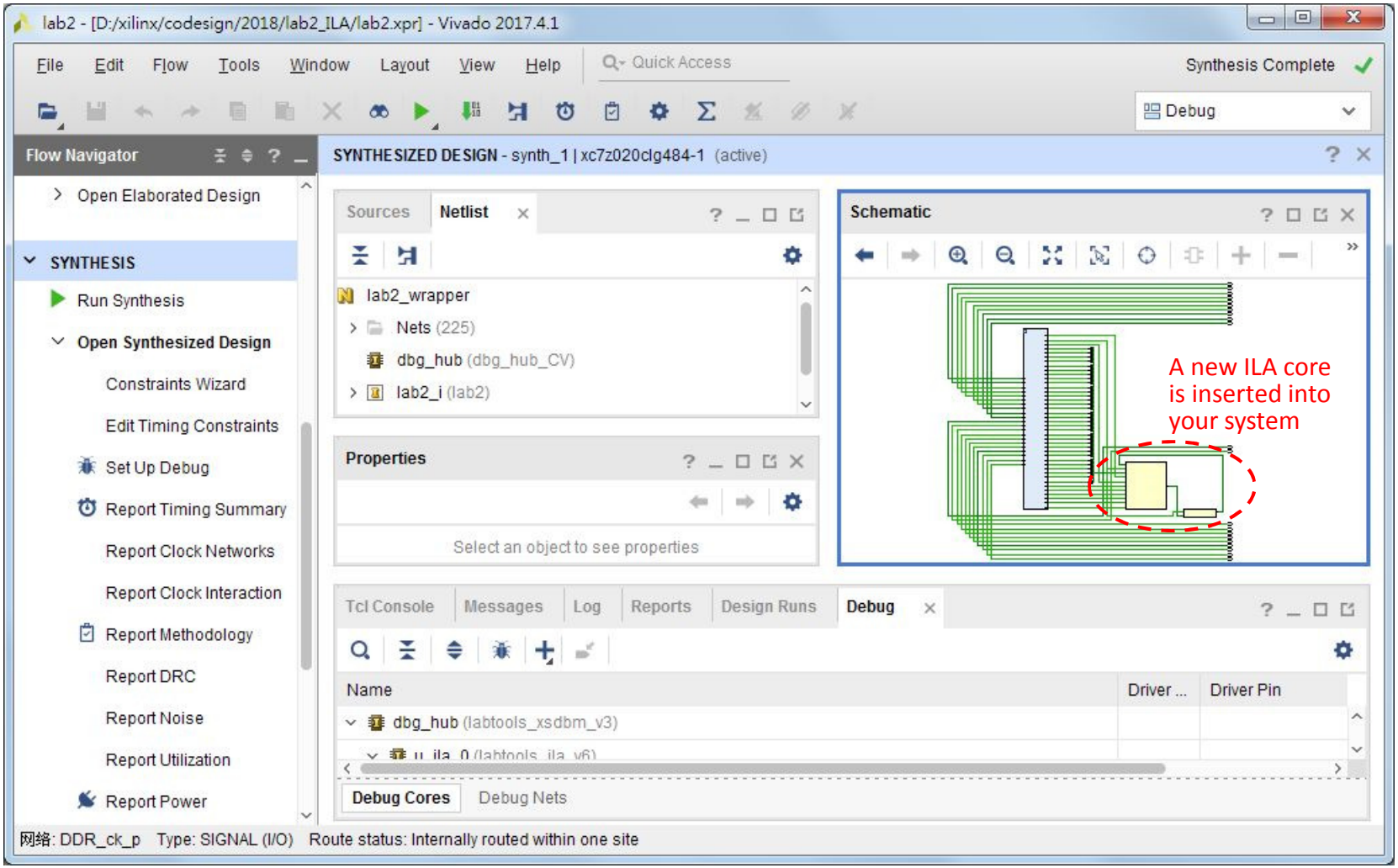


The first time you save the debug signal constraints, Vivado will ask you to name the constraint file.

- Re-synthesize the design to show the debug core

An ILA Debug Core Has Been Added

- Your hardware platform now has an extra ILA IP:

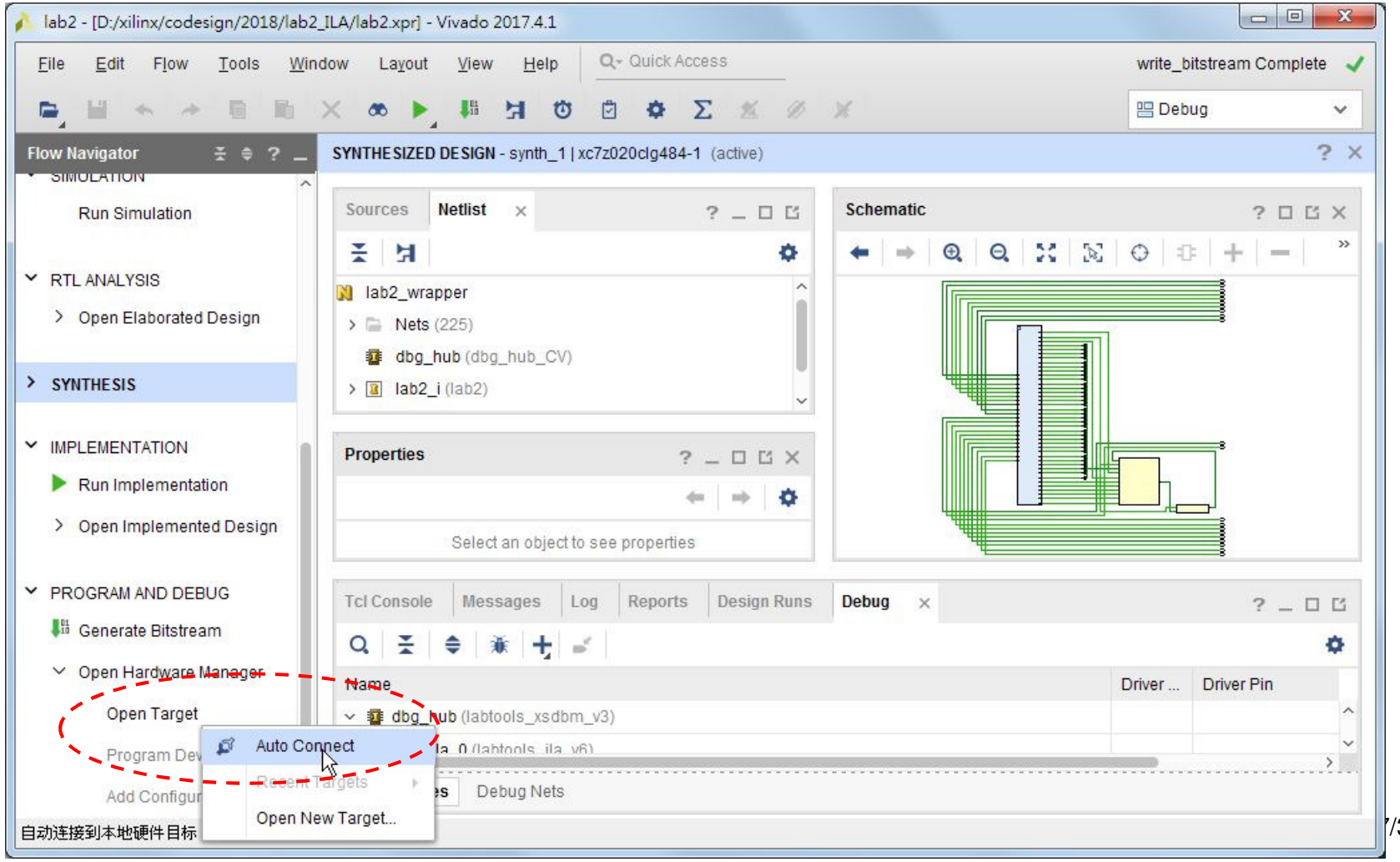


Generate Bitstream

- Now, you can click “Generate Bitstream”
- Export the new hardware to the SDK and launch it
- In order to capture the debug signals at runtime, we must use the Hardware Manager to configure the FPGA and connects to the ILA logic
  - Do not use the SDK to configure the FPGA

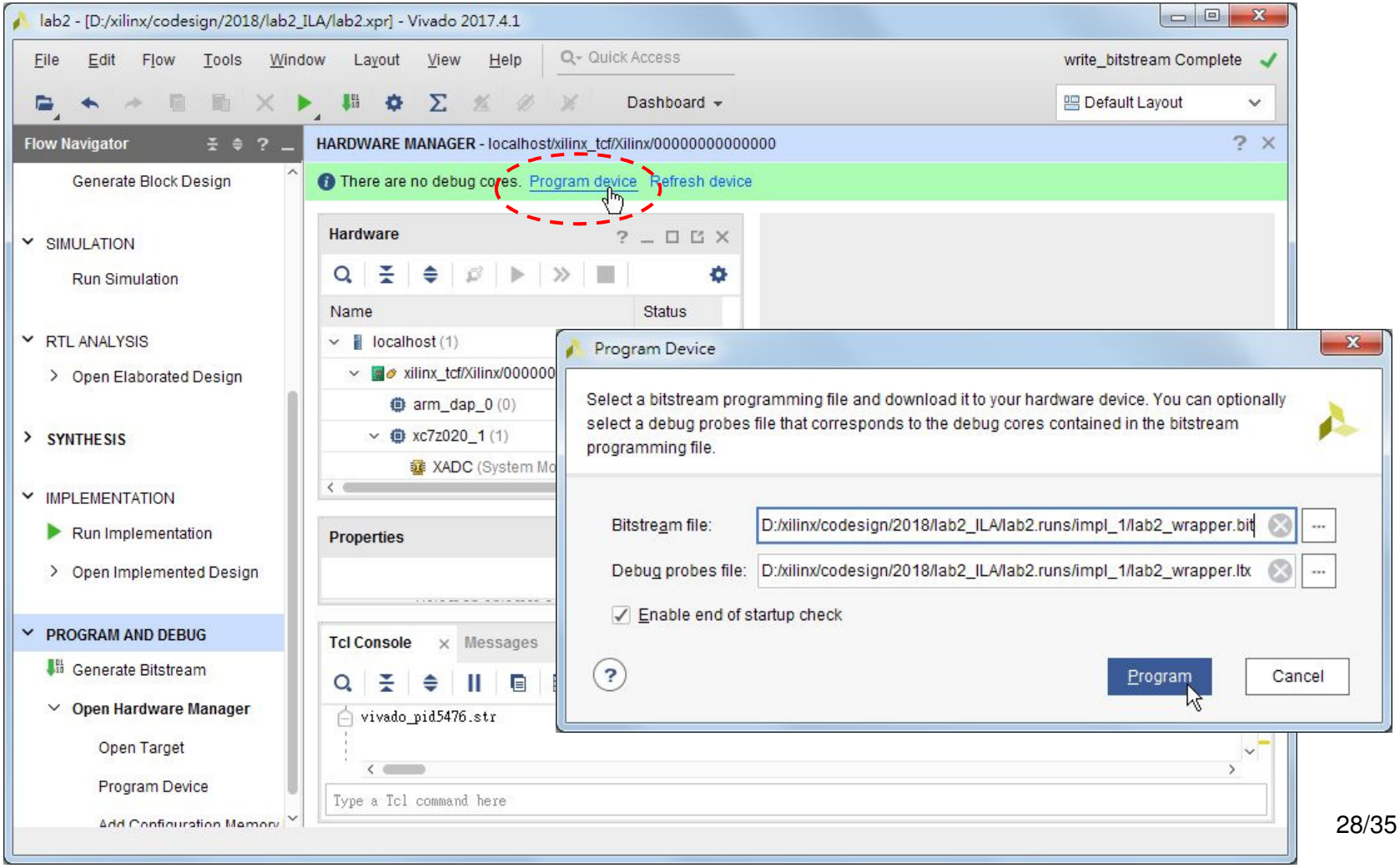
Open Hardware Manager

- Click the “Hardware Manager” and “Open Target”



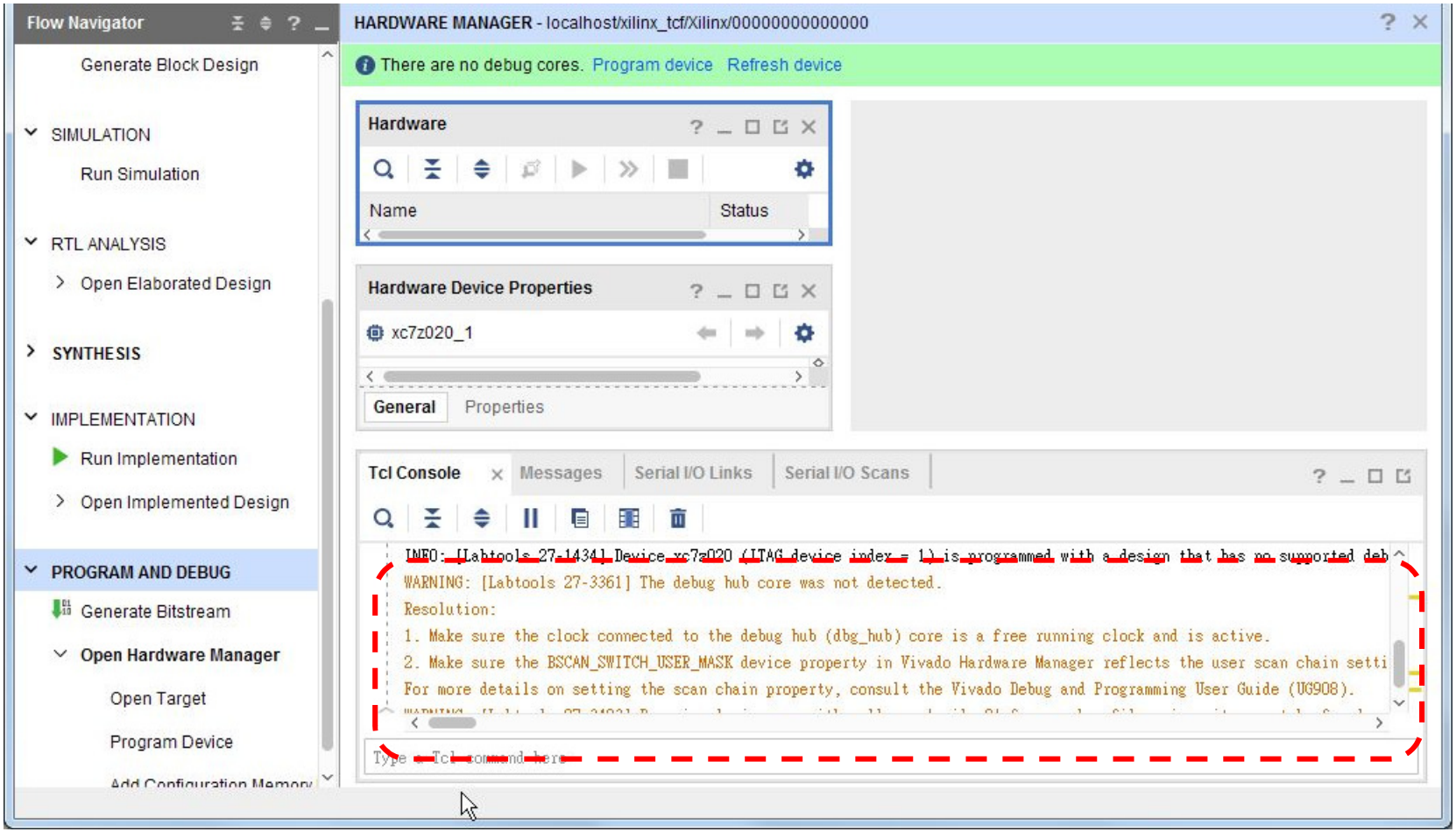
Program the FPGA

- Now you can program the FPGA:



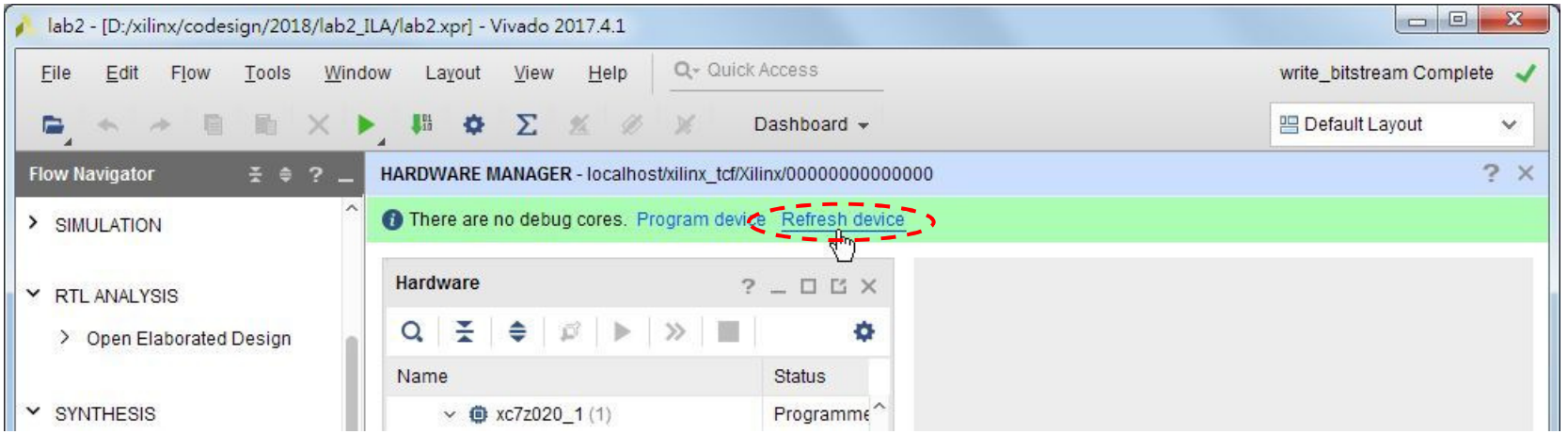
The ILA Logic is Missing?

- After you programmed the FPGA, the HW Manager displays a warning: the debug core is not found!

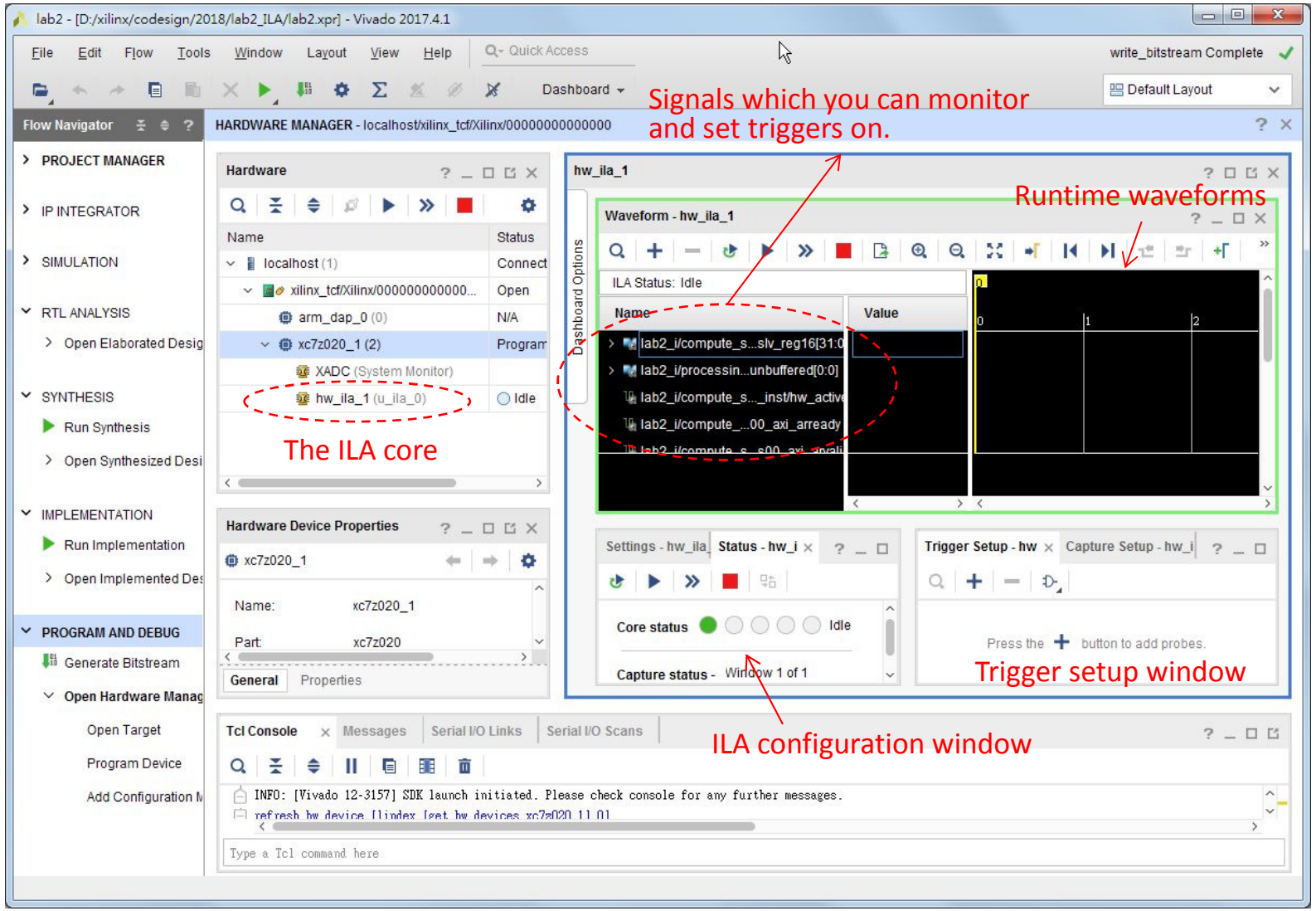


Fixing the Clock Issue

- The problem is that PS7 has not yet been initialized, and the ILA core need the clock signal from the PS7 clock module by default
- Use the SDK to run any program on the ZedBoard, then come back to the HW Manager in Vivado and click "Refresh device" to connect to the ILA
  - Note again: Don't configure the FPGA from the SDK!

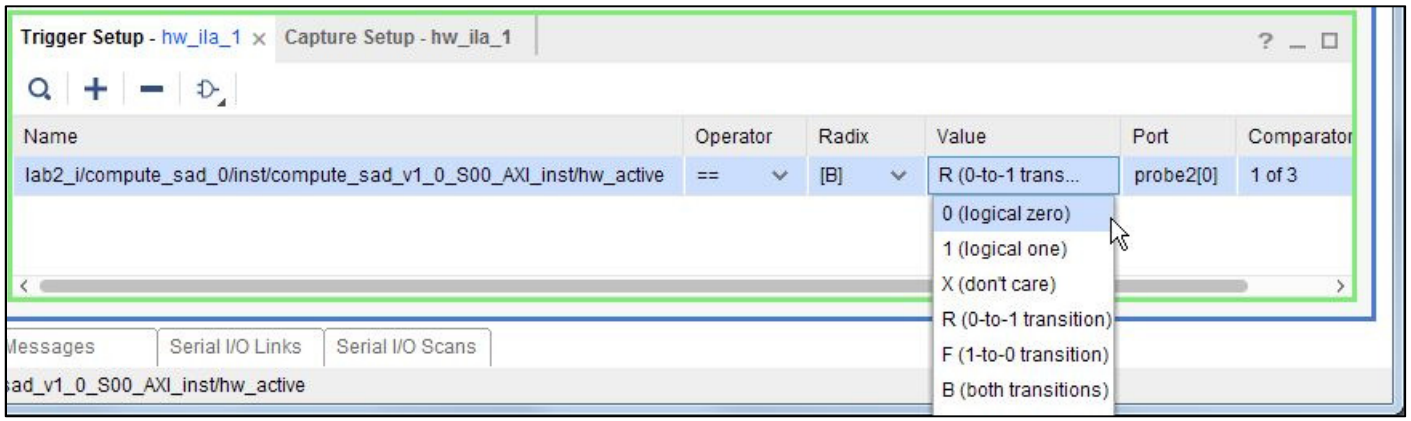


The Hardware Manager with ILA View



Setting a Trigger

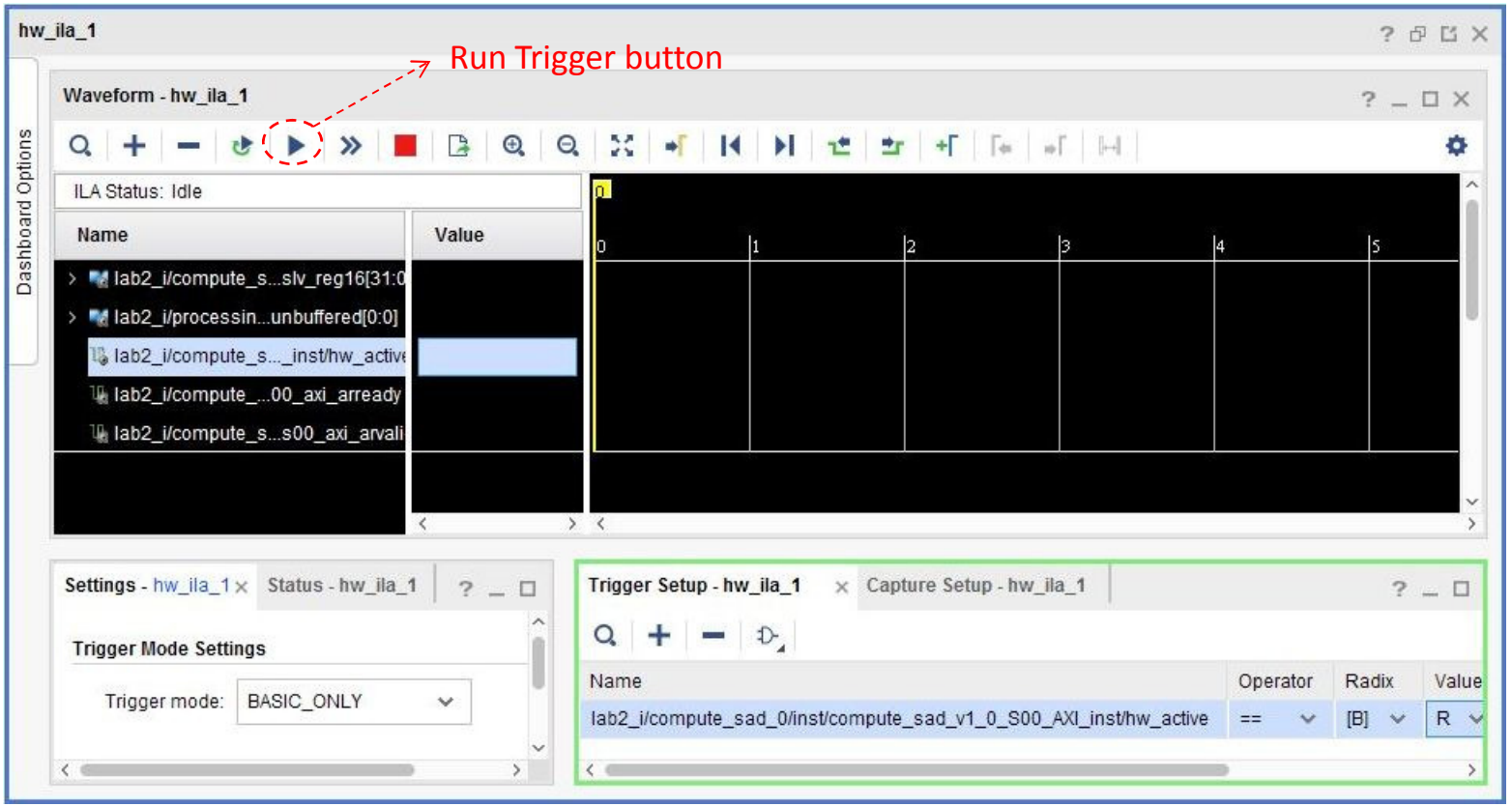
- A trigger is a signal condition that tells the ILA to begin capturing waveforms; Drag a signal from “Waveform” window to “Trigger Setup” window to use it as a trigger
- Enter the trigger condition:



- Whenever `find_face_rtos` calls the `compute_sad` logic, it will trigger the ILA to capture the signals

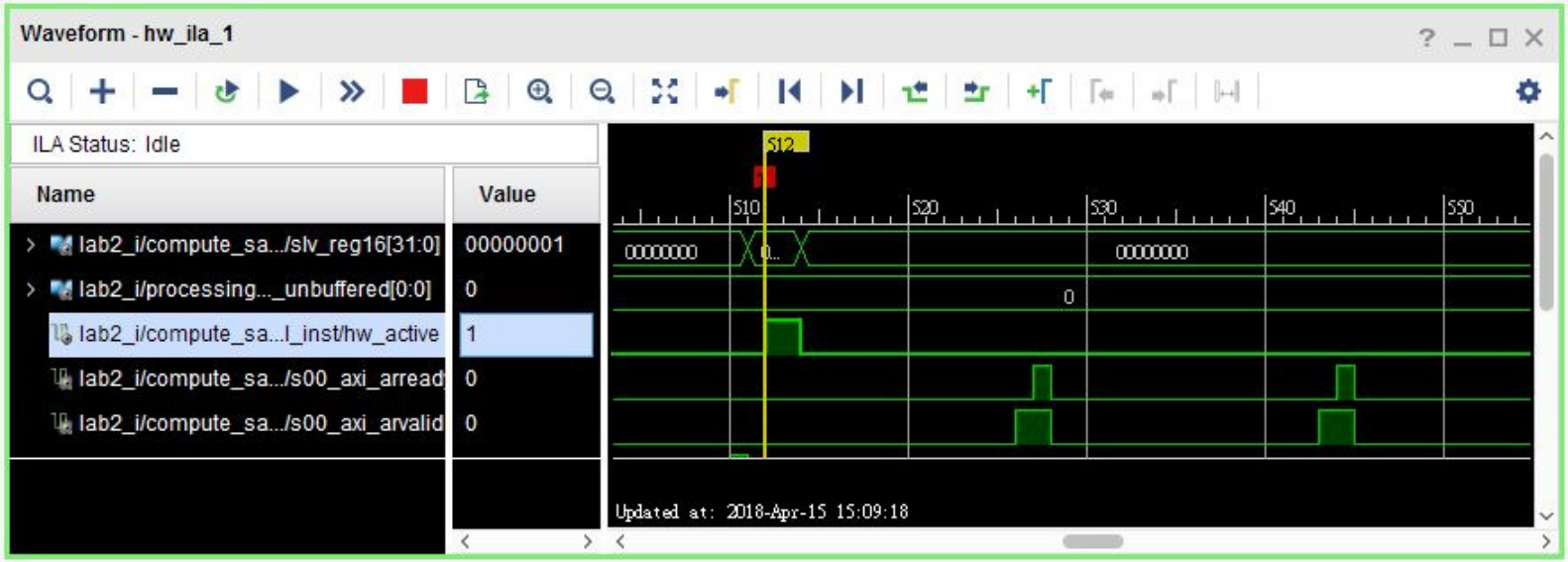
Capturing the Signals

- Now, you can hit the “Run Trigger” button, then go back to the SDK to run the `find_face_rtos` program



Analyze the Captured Waveform

- Now you can analyze the waveforms for bugs:



References

- Xilinx, *Vivado Design Suite Tutorial: Programming and Debugging*, UG936 (v2017.4) Dec. 20, 2017.