

# 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

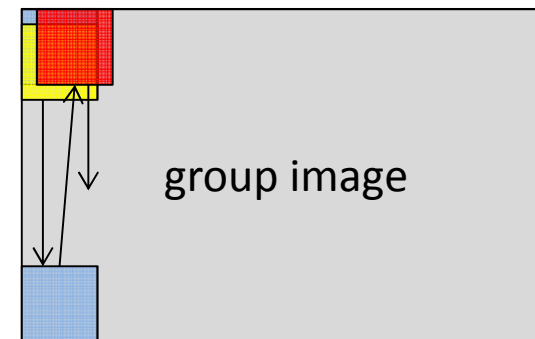
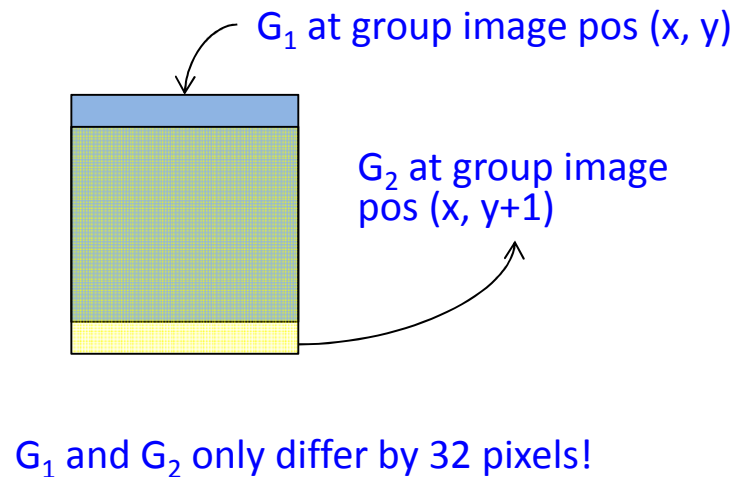
- ❑ During SAD computation, we slide a  $32 \times 32$  face image through the group image
  - “Face” never changes → copy it once and store it in the HW IP
  - For any two consecutive `compute_sad()`, the group pixels used are mostly overlapping!

```
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:



Top-down matching path, only the top blocks (in red) require copying of  $32 \times 32$  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()

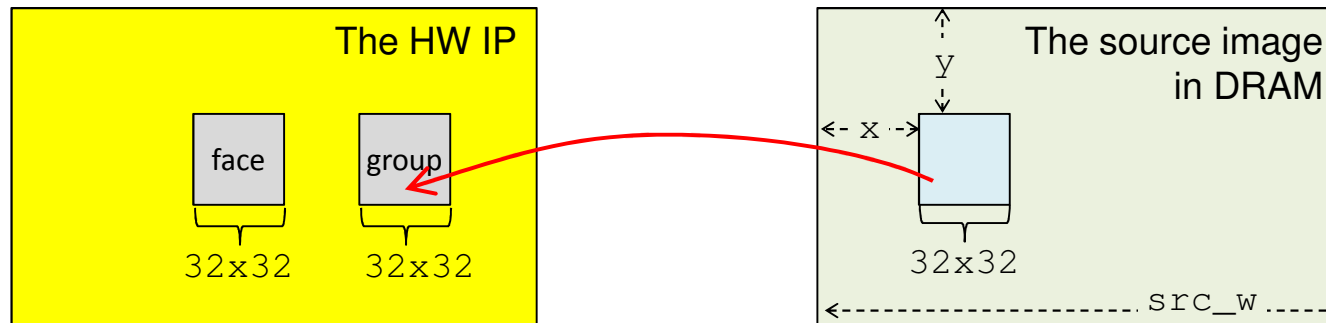
- ❑ The new `compute_sad()` only copy minimal pixels into the HW IP for matching a block:
  - Variables in bold italics fonts are HW registers

```
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)

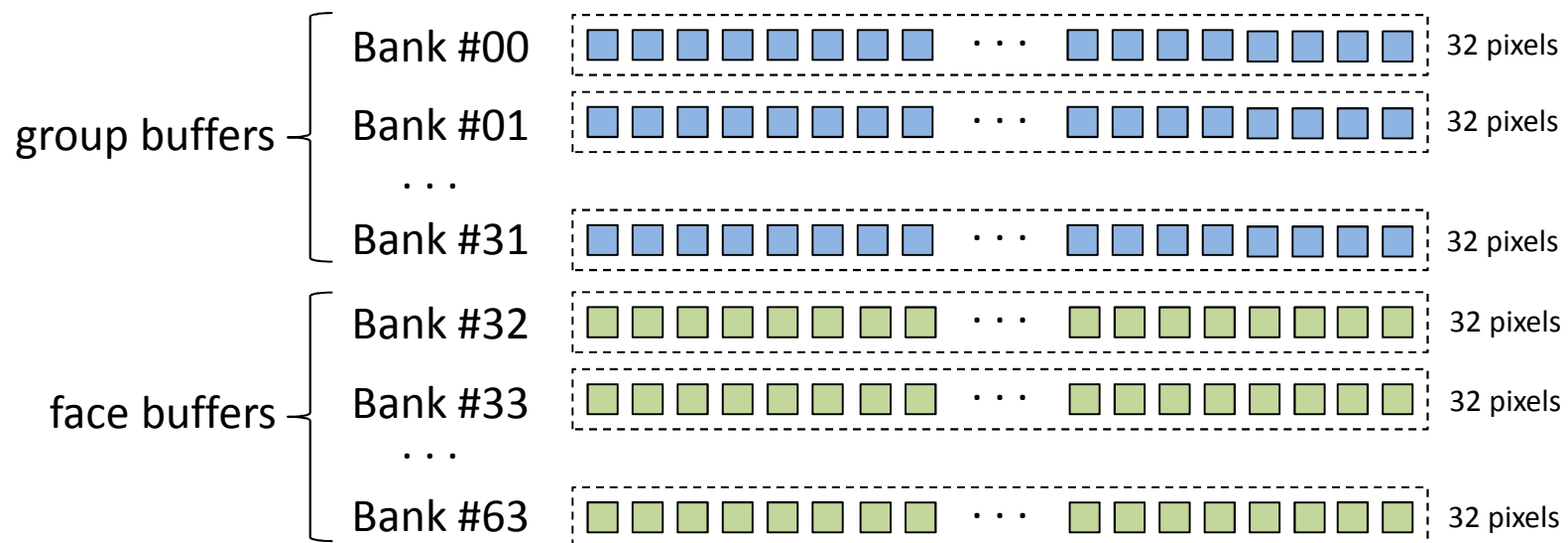
- ❑ Copy a rectangular area in a 2D image is a very important function for image/video processing



- ❑ 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,  $R0 \sim R7$ , and a bank ID register, `reg_bank`.
  - The  $32 \times 32$  group pixel buffer is divided into 32 banks (#00 ~ #31) and so is the face buffer (#32 ~ #63)
  - Data written into  $R0 \sim R7$  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  $32 \times 32$  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:

The screenshot shows the Vivado 2017.4.1 interface with the 'SYNTHESIZED DESIGN - synth\_1 | xc7z020clg484-1 (active)' project. The 'Flow Navigator' on the left highlights 'Report Timing Summary' under the 'SYNTHESIS' section. The 'Netlist' window shows the design hierarchy. The 'Schematic' window shows a block diagram of the design. The 'Timing' window at the bottom displays the 'Design Timing Summary' table.

**Design Timing Summary**

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 3.833 ns	Worst Hold Slack (WHS): 0.045 ns	Worst Pu
Total Negative Slack (TNS): 0.000 ns	Total Hold Slack (THS): 0.000 ns	Total Pul
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number
Total Number of Endpoints: 3903	Total Number of Endpoints: 3903	Total Nur

A red dashed circle highlights the 'Worst Negative Slack (WNS): 3.833 ns' value. A red arrow points from the text 'Critical path delay = clock period - WNS' to this value.

Timing Summary - timing\_1

网络: FIXED\_IO\_dds\_vrn Type: SIGNAL (I/O) Route status: Internally routed within one site

# Check Utilization Report

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

The screenshot shows the Vivado 2017.4.1 IDE. The 'Flow Navigator' on the left has 'Report Utilization' circled in red. The 'SYNTHESIZED DESIGN' window shows a hierarchy of components. The 'Utilization' tab is active, displaying a table of resource usage.

Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	Bonded IOPADS (130)	BUFCTRL (32)
lab2_i (lab2)	1573	1620	64	0	1
compute_sad_0 (l...	1028	979	64	0	0
processing_syste...	112	0	0	0	1

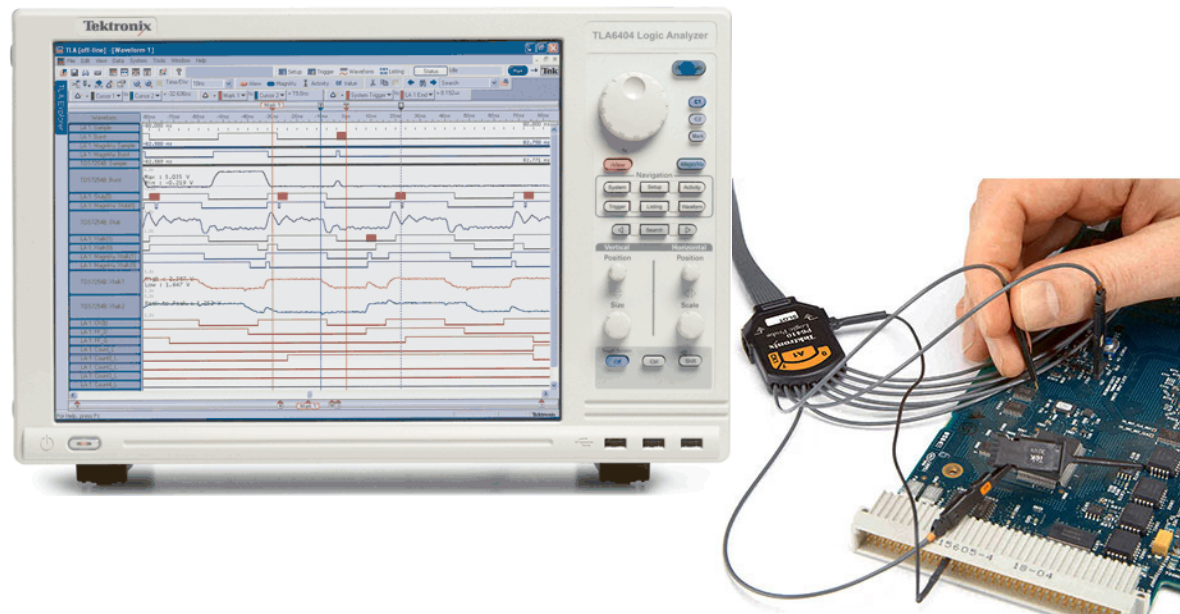
# Suggestions on Your Design Process

---

- ❑ 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

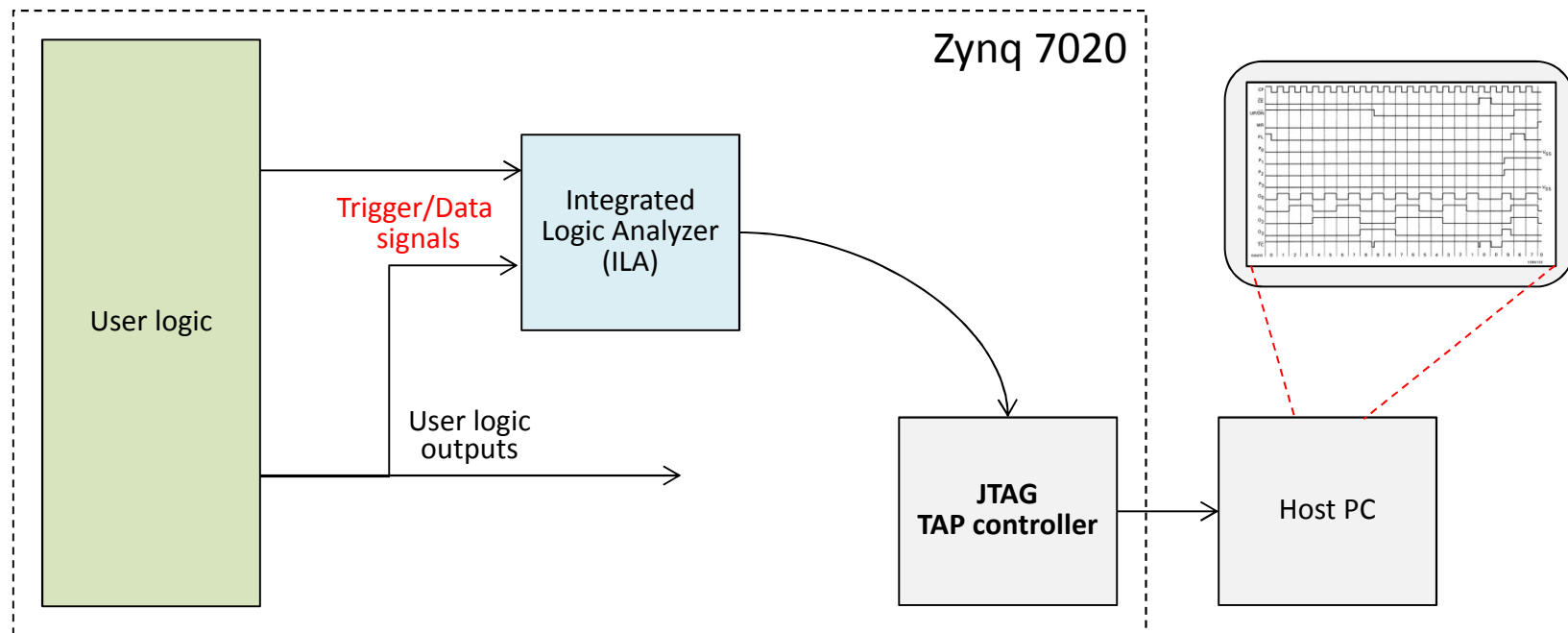
- ❑ Full-system simulations for complex logic and software behaviors would take too much time
- ❑ In the good old days, for real-time debugging of a digital circuit, we must use a logic analyzer device





# 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:

The screenshot displays the Vivado 2017.4.1 interface for a project named 'lab2'. The 'Flow Navigator' on the left shows the 'SYNTHESIS' phase, with 'Open Synthesized Design' highlighted. The 'Netlist' window shows the design hierarchy, with 'abs\_diff\_reg[0]\_58' selected. The 'Schematic' window shows the physical implementation of the design. The 'Debug' window is also visible, showing unassigned debug nets.

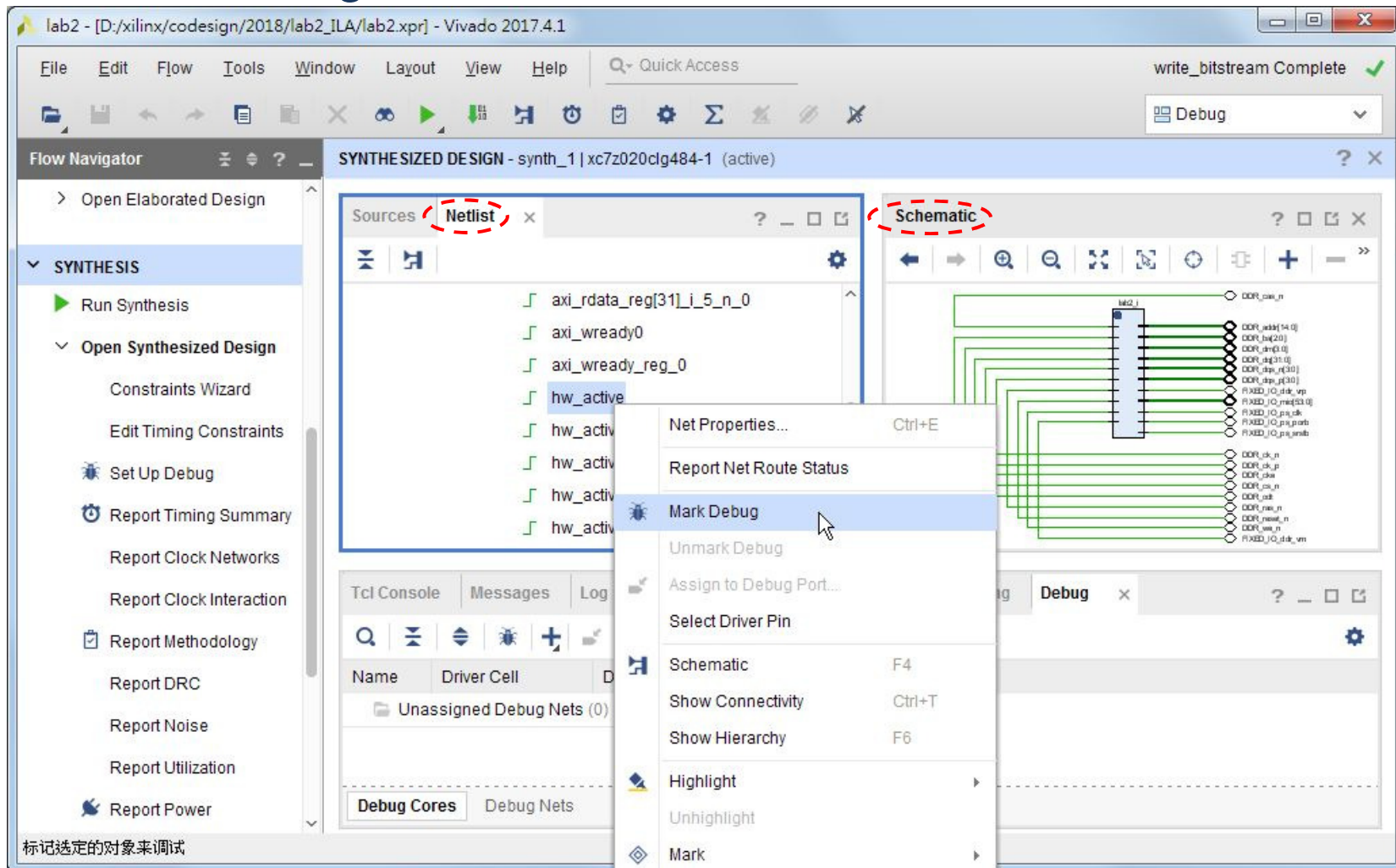
Select signals to be debugged from here!

Click!

This window shows signals to be debugged!

# 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:

The screenshot shows the Vivado 2017.4.1 interface. The 'Set Up Debug' wizard is open, displaying the following text:

**Set Up Debug**

This wizard will guide you through the process of

1. Choosing nets and connecting them to debug cores.
2. Associating a clock domain with each of the nets chosen for debug.
3. Choosing additional features on the debug cores like Data Depth, Advanced Trigger mode and Capture Control.

Note: This setup wizard does not apply to the VIO, IBERT or JTAG-to-AXI-Master debug cores. Please refer to [Vivado Design Suite User Guide: Programming and Debugging \(UG908\)](#) for further instructions on how to use these IPs.

The 'Next >' button is highlighted with a red dashed circle. A red dashed arrow points to the 'Set Up Debug' icon in the Flow Navigator.

Flow Navigator:

- Open Elaborated Design
- SYNTHESIS
  - Run Synthesis
  - Open Synthesized Design
    - Constraints Wizard
    - Edit Timing Constraints
    - Set Up Debug
    - Report Timing Summary
    - Report Clock Networks
    - Report Clock Interaction
    - Report Methodology
    - Report DRC
    - Report Noise
    - Report Utilization
    - Report Power

Netlist:

- s00\_axi\_ws
- sad\_result
- sel0 (5)
- slv\_reg16
- slv\_reg17
- sum\_level
- sum\_level

Tcl Console:

Name	Driver Cell	Driver Pin	Probe Type
lab2_i/compute_sad_0/inst/compute_sad_v1_0_S00_AXI_inst/slv_reg1...	FDRE	Q	
lab2_i/compute_sad_0/inst/compute_sad_v1_0_S00_AXI_inst/s00_axi...	FDRE	Q	

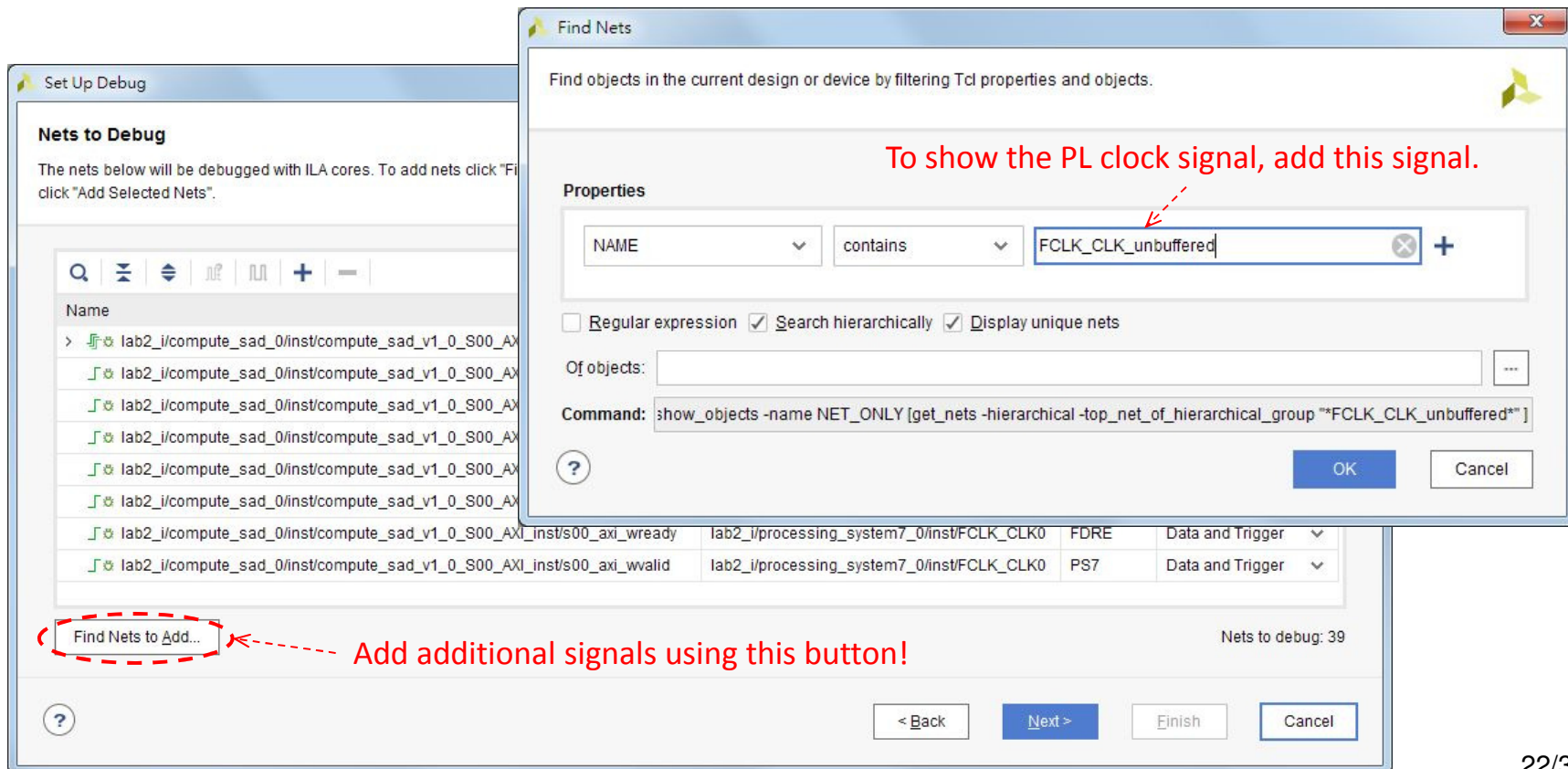
Debug Cores: Debug Nets

启动向导去选择信号走线和连接它们到调试内核



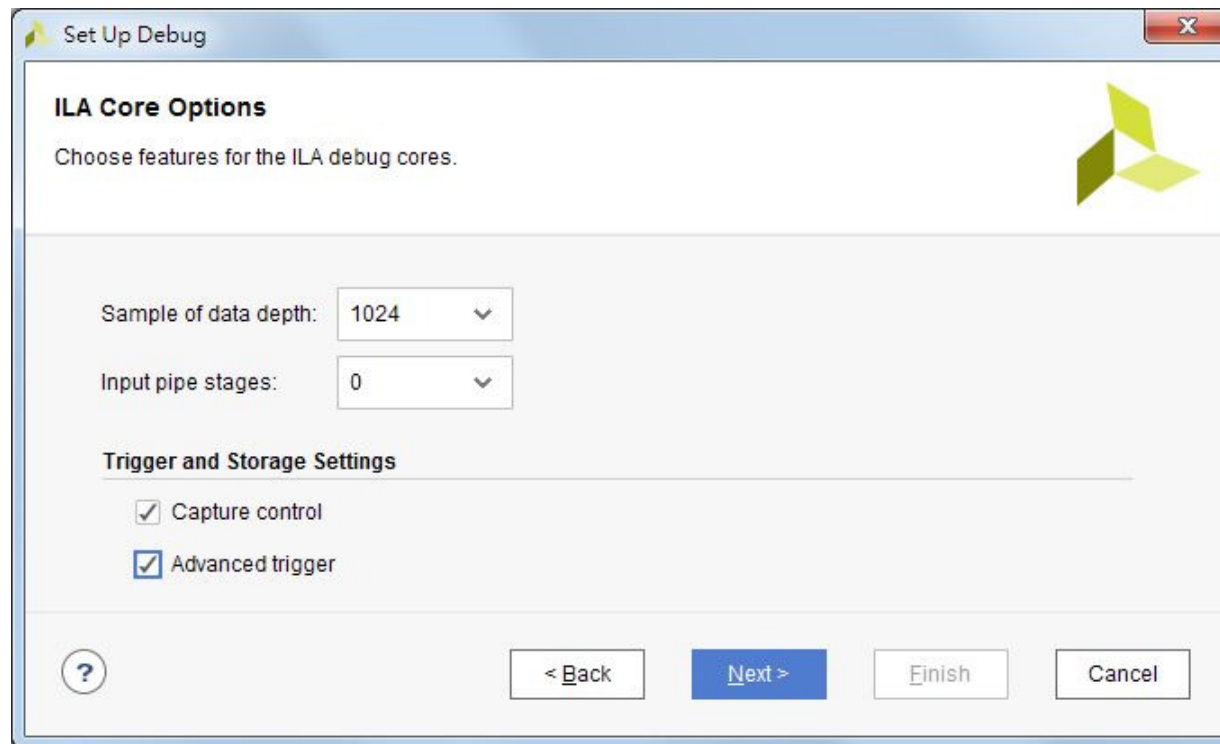
# Check Nets to Be Debugged Again

- ❑ 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!



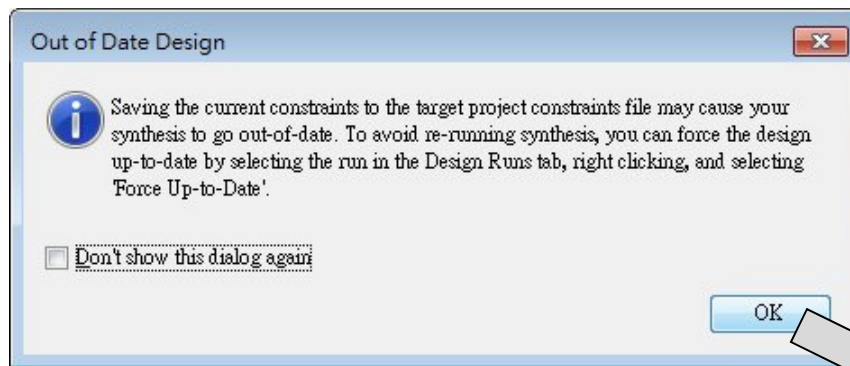
# Modify Trigger Options

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

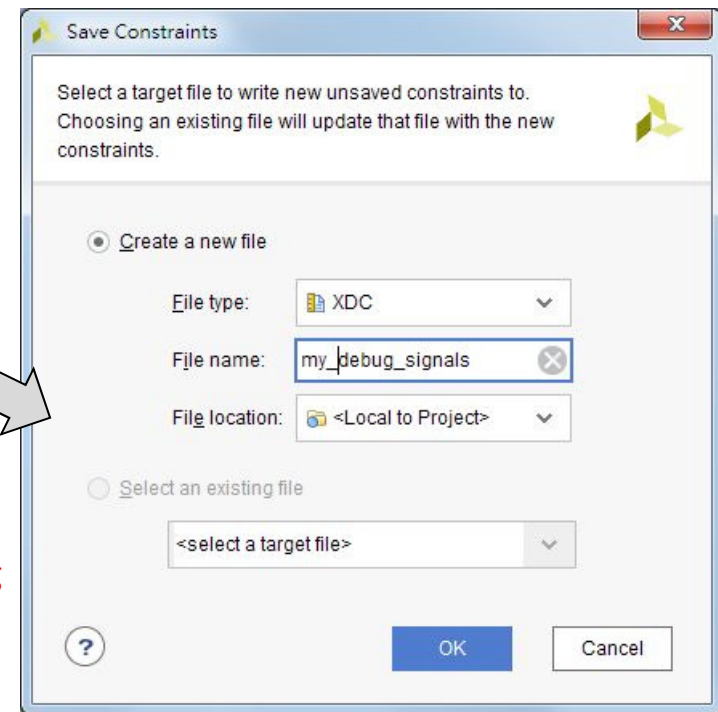


# Save the New Debug Constraints

- ❑ 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.

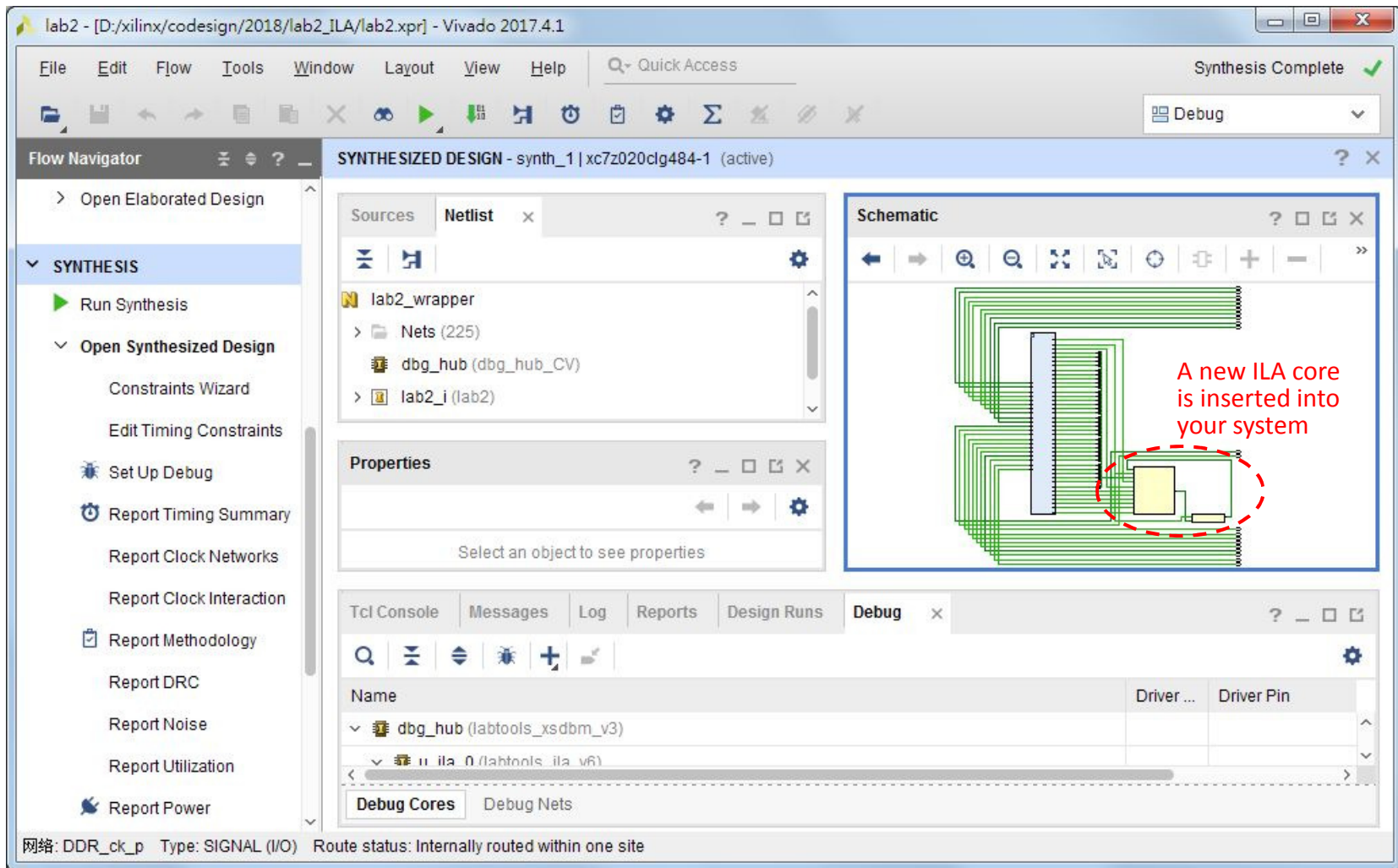


- ❑ Re-synthesize the design to show the debug core



# An ILA Debug Core Has Been Added

- ❑ Your hardware platform now has an extra ILA IP:



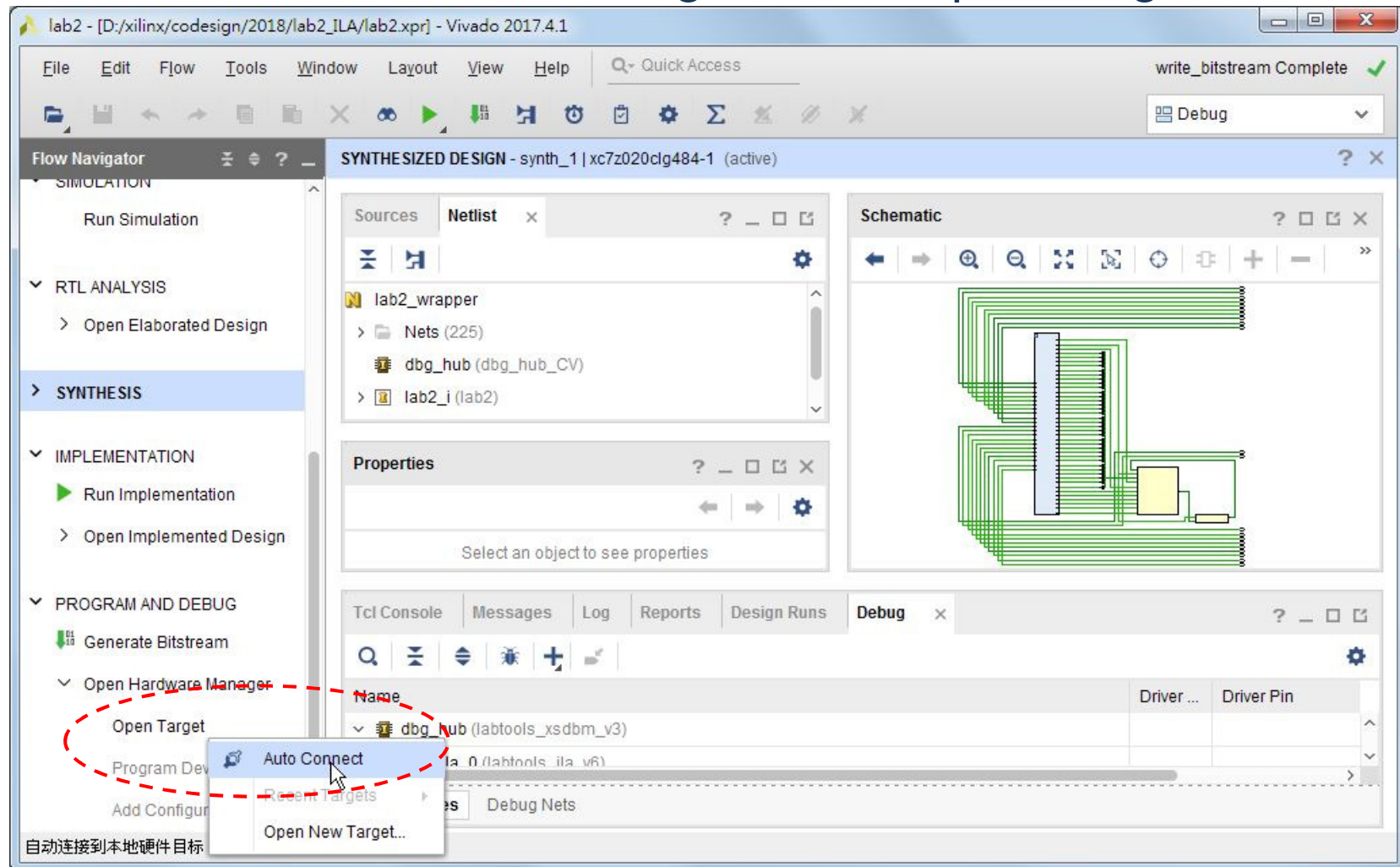
# Generate Bitstream

---

- ❑ 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

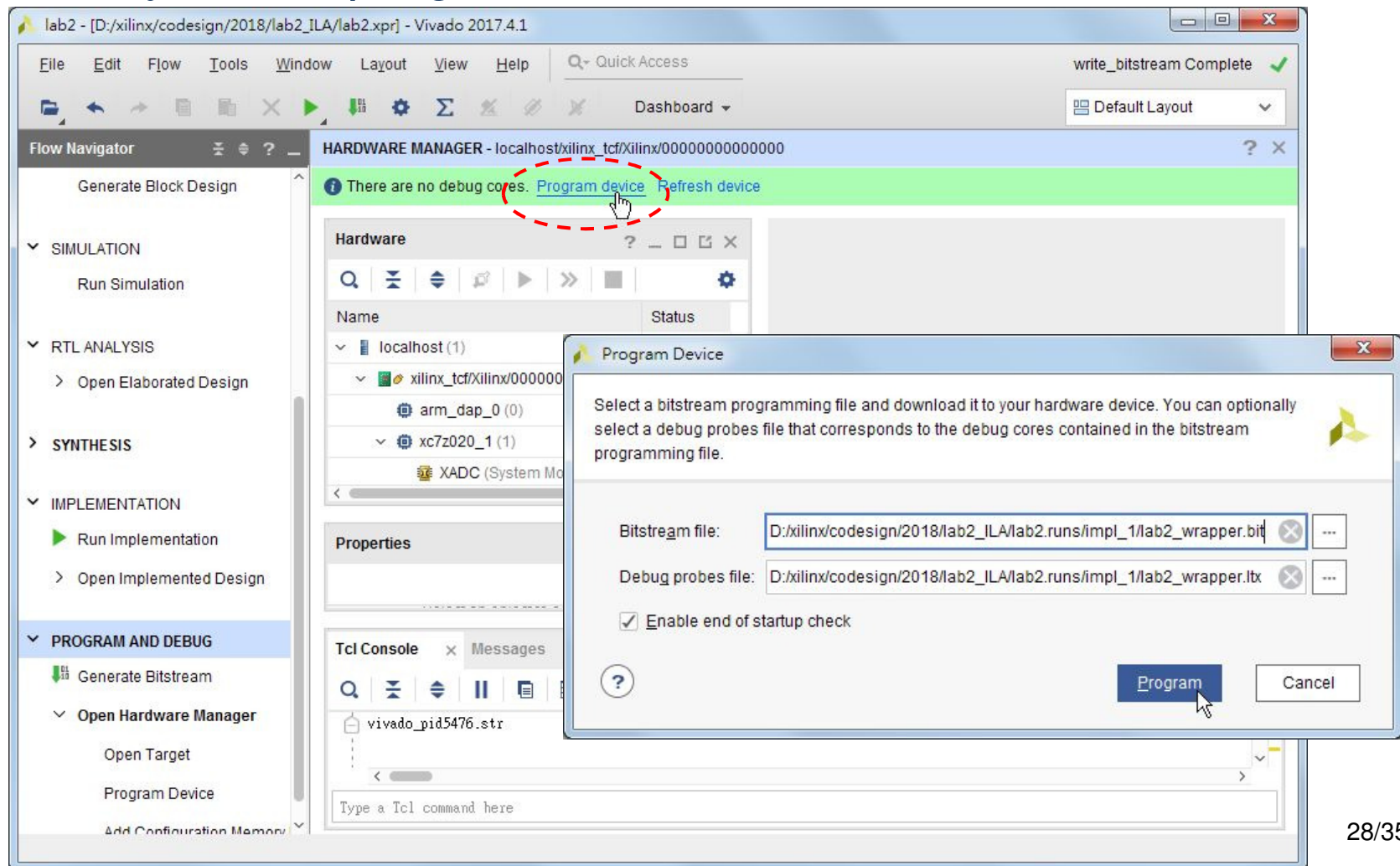
# Open Hardware Manager

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



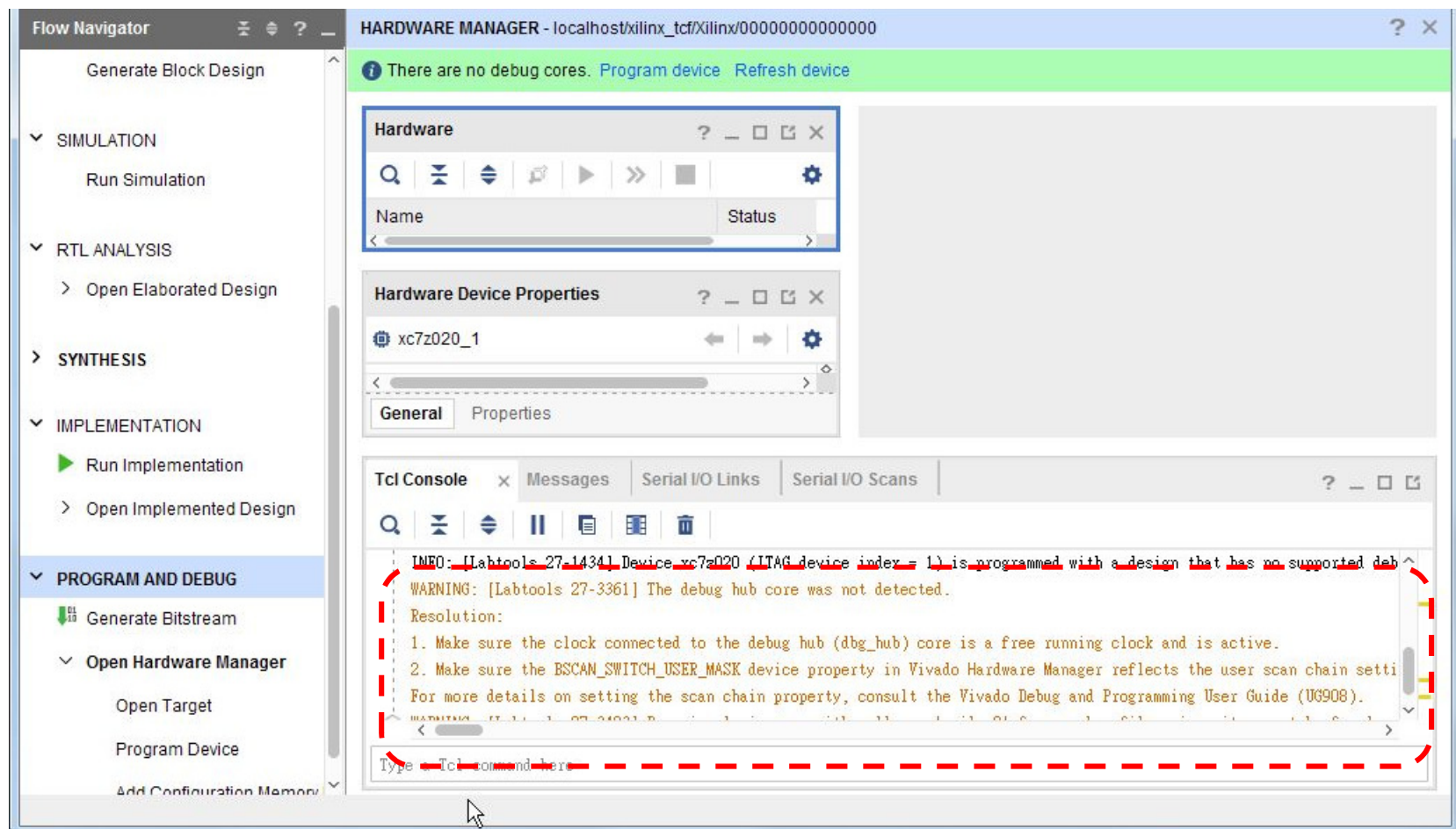
# Program the FPGA

- ❑ Now you can program the FPGA:



# The ILA Logic is Missing?

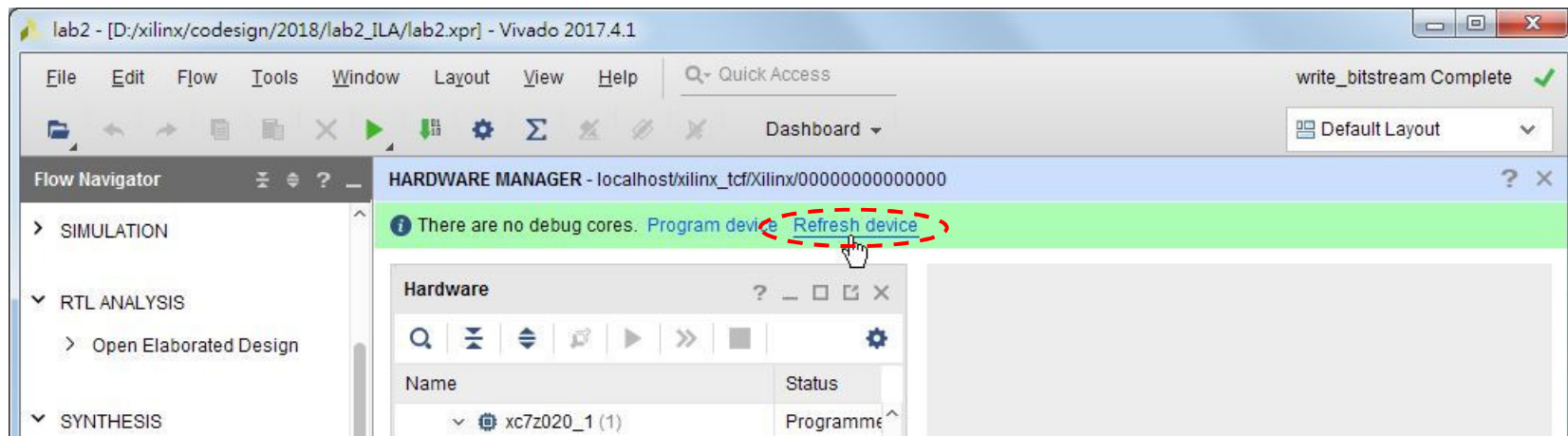
- ❑ After you programmed the FPGA, the HW Manager displays a warning: the debug core is not found!





# 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

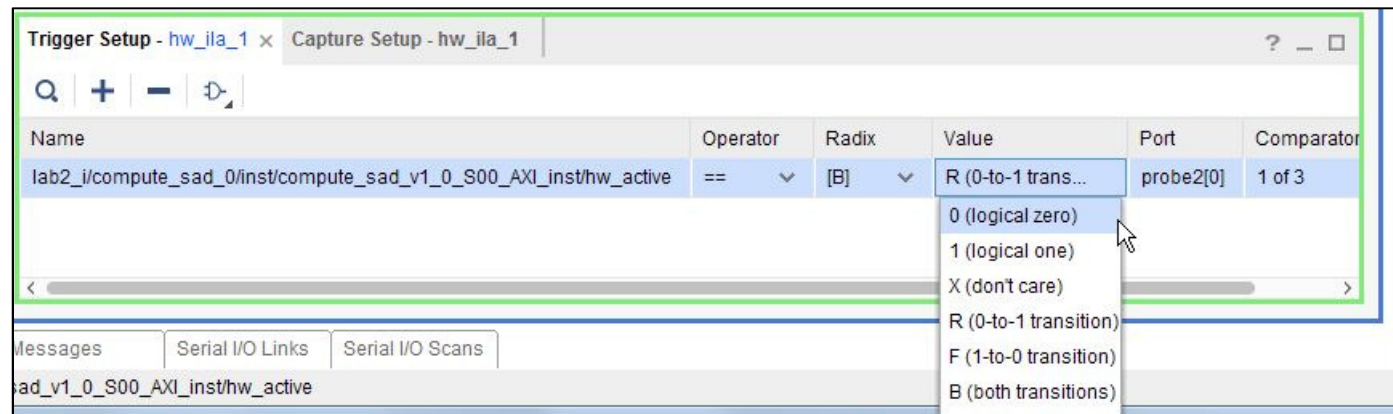
The screenshot displays the Vivado 2017.4.1 interface with the Hardware Manager open. The left sidebar shows the Project Manager with the 'PROGRAM AND DEBUG' section selected. The main area is divided into several panes:

- Hardware Manager:** Shows a list of hardware components. The 'hw\_ila\_1 (u\_ila\_0)' component is highlighted with a red dashed circle and labeled 'The ILA core'.
- Hardware Device Properties:** Shows properties for the 'xc7z020\_1' device.
- ILA configuration window:** A red arrow points to the 'hw\_ila\_1' pane, which contains a table of signals to monitor and set triggers on. The table has columns for 'Name' and 'Value'. The signals listed are:
  - lab2\_i/compute\_s...slv\_reg16[31:0]
  - lab2\_i/processin...unbuffered[0:0]
  - lab2\_i/compute\_s...inst/hw\_active
  - lab2\_i/compute\_...00\_axi\_arready
  - lab2\_i/compute\_s...00\_axi\_arvalid
- Runtime waveforms:** A red arrow points to the 'Waveform - hw\_ila\_1' pane, which displays a waveform for the selected signals.
- Trigger setup window:** A red arrow points to the 'Trigger Setup - hw\_ila\_1' pane, which shows the 'Core status' as 'Idle' and the 'Capture status' as 'Window 1 of 1'.

Red annotations highlight key features: 'Signals which you can monitor and set triggers on.' points to the signal list; 'Runtime waveforms' points to the waveform display; 'The ILA core' points to the hardware component; and 'Trigger setup window' points to the trigger configuration pane.

# 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:

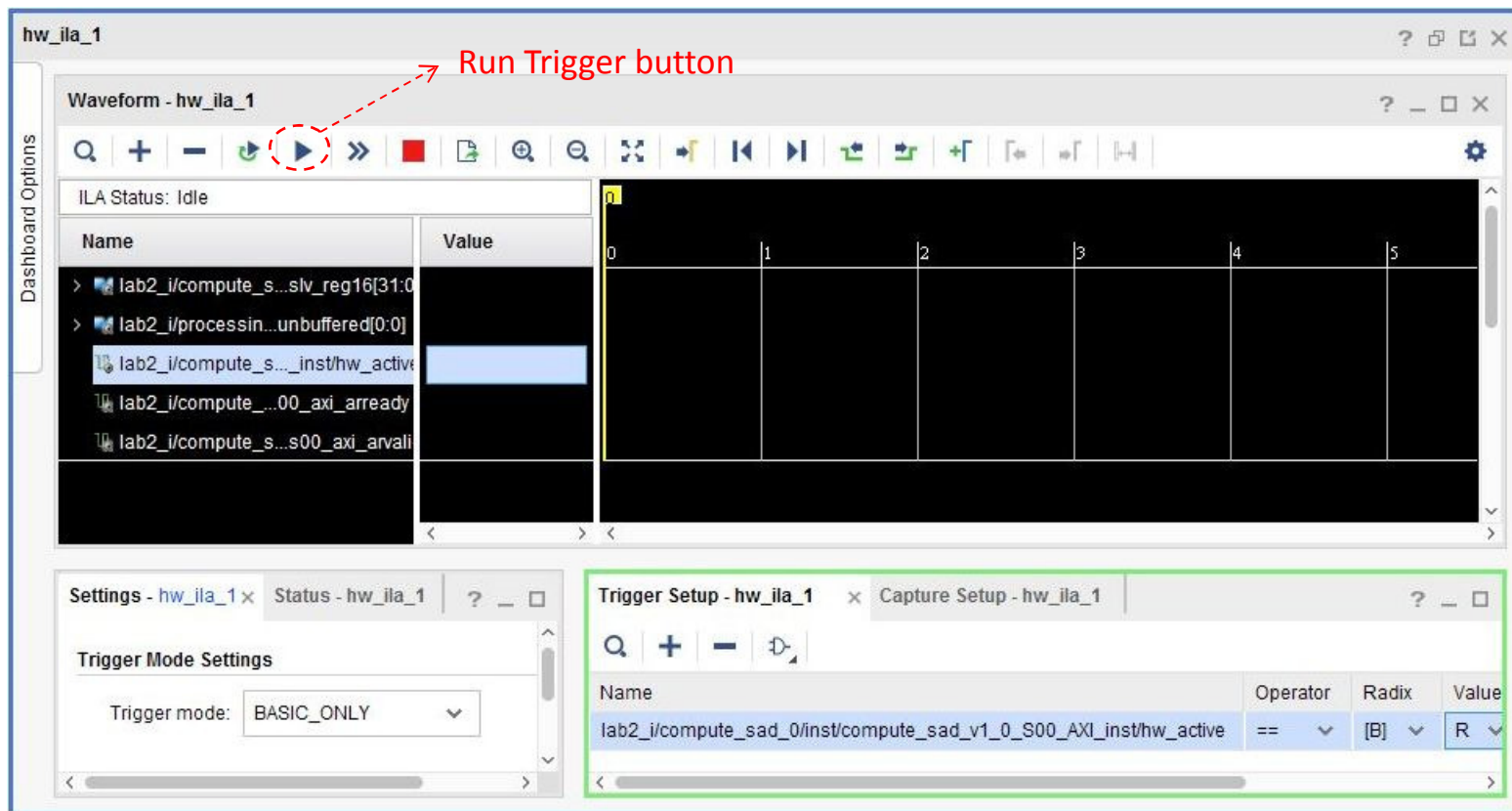


- ❑ Whenever `find_face_rtos` calls the `compute_sad` logic, it will trigger the ILA to capture the signals



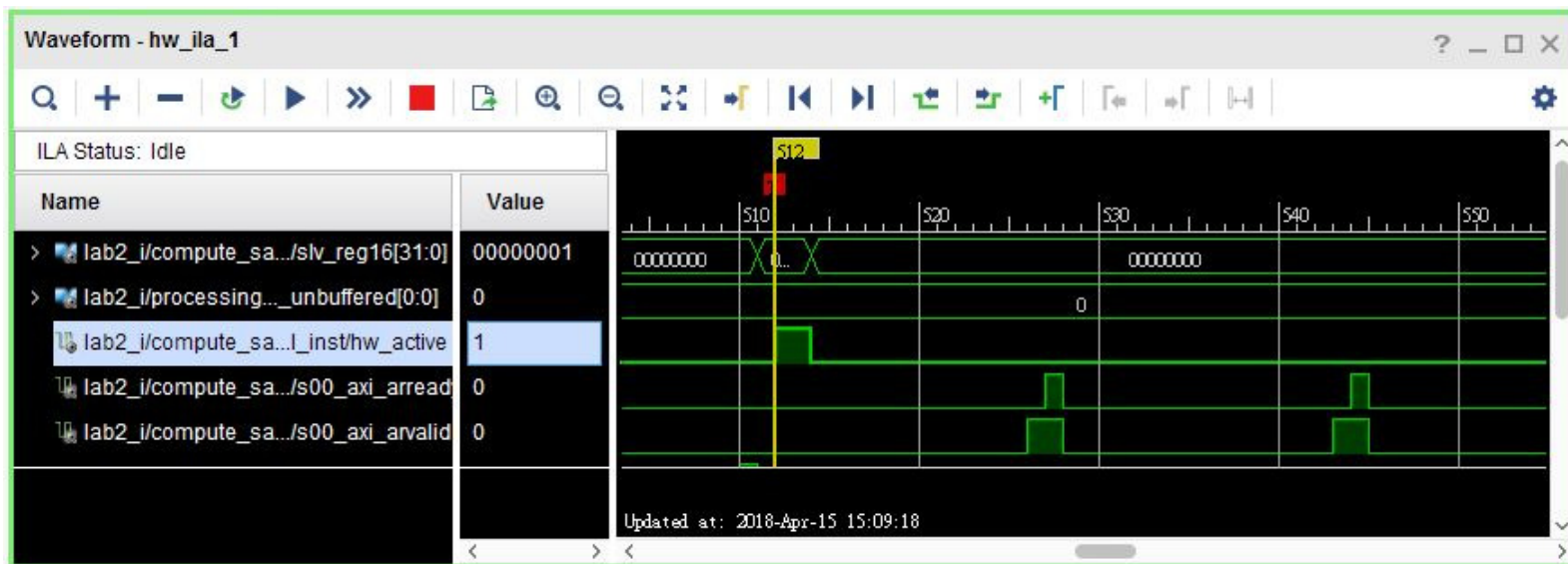
# Capturing the Signals

- ❑ 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.