

HW#1 Real-time Analysis of a HW-SW Platform



Chun-Jen Tsai
NYCU
9/25/2023

Homework Goal

- ❑ In this homework, you will add profiling hardware to the Aquila core to analyze the program execution behavior
- ❑ The CoreMark benchmark program is used here
 - You will learn how to profile a program on the real platform
 - You will also learn how to use Xilinx Integrated Logic Analyzer (ILA) for real-time debugging
- ❑ Deadline: 10/12, 17:00
 - Upload your source code and a two-page report to E3
 - Only PDF file is allowed for report submission.
 - The TAs will set up a schedule for you to demo your code

Synthesis-Execution Flow

- ❑ To run the HW-SW system on an FPGA, you must follow the steps:
 1. Generate the HW bit file
 2. Power on the Arty board
 3. Use a terminal program to connect to the FPGA via UART
 4. Program the FPGA
 5. Send an ELF program to the FPGA via UART
 6. Wait for the program to execute and print results

Circuit Implementation for FPGA

Synthesize the Aquila SoC

The screenshot displays the Vivado 2023.1 interface for the 'aquila_mpd' project. The left sidebar shows the project hierarchy with 'Generate Bitstream' and 'Open Target' buttons highlighted by red dashed circles. The 'Project Summary' window on the right shows the 'Post-Implementation' results, including a bar chart of resource utilization and a table of power metrics. The 'Design Runs' table at the bottom shows the 'synth_1' run as 'Complete!'.

Project Summary - Post-Implementation

Utilization	Post-Synthesis	Post-Implementation	Power	Sum
LUT	6%			
LUTRAM	1%			
FF	2%			
BRAM	12%			
DSP	2%			
IO	4%			
BUFG	6%			
MMCM	17%			

Power Metrics:

- Total On-Chip Power: 0.223 W
- Junction Temperature: 26.0 °C
- Thermal Margin: 59.0 °C (12.8 W)
- Effective θ_{JA} : 4.6 °C/W
- Power supplied to off-chip devices: 0 W
- Confidence level: Medium

Design Runs Table:

Name	Col	THS	WBSS	TPWS	Total Power	Failed Routes	Methodology
synth_1 (active)	constrs_1	synth_design Complete!					
impl_1	constrs_1	write_bitstream Complete!	6.426	0.000	0.060	0.000	0.000

Click this to generate the FPGA bit file.

Click this to connect to the FPGA board.

Program The FPGA

The screenshot shows the Vivado IDE interface. The top menu bar includes File, Edit, Flow, Tools, Reports, Window, Layout, View, and Help. The top status bar shows 'write_bitstream Complete' with a green checkmark. The left sidebar contains the Flow Navigator with sections: SIMULATION, RTL ANALYSIS, SYNTHESIS, IMPLEMENTATION, and PROGRAM AND DEBUG. The 'PROGRAM AND DEBUG' section is expanded, showing 'Generate Bitstream' and 'Open Hardware Manager'. The 'Open Hardware Manager' button is circled in red and labeled with a red '1'. The main window is titled 'HARDWARE MANAGER - localhost/xilinx_tcf/Digilent/210319B26B20A'. It displays a message: 'There are no debug cores. Program device Refresh device'. The 'Program device' button is circled in red and labeled with a red '2'. Below this is a table with columns 'Name' and 'Status'. The table has two rows: 'localhost (1)' with status 'Connected' and 'xilinx_tcf/Digilent/210319B26B20A' with status 'Open'. Below the table is a 'Properties' section. The bottom window is the 'Tcl Console' showing the following output:

```
INFO: [Labtoolstcl 44-466] Opening hw_target localhost:3121/xilinx_tcf/Digilent/210319B26B20A
set_property PROGRAM_FILE [D:/teaching/my_classnotes/mpd23/homework/aquila_build/aquila_mpd/aquila_mpd.runs/impl_1/soc_top.bit] [get_hw_devices xc7a100t_0]
current_hw_device [get_hw_devices xc7a100t_0]
refresh_hw_device -update_hw_probes false [lindex [get_hw_devices xc7a100t_0] 0]
INFO: [Labtools 27-1434] Device xc7a100t (JTAG device index = 0) is programmed with a design that has no supported debug core(s) in it.
```

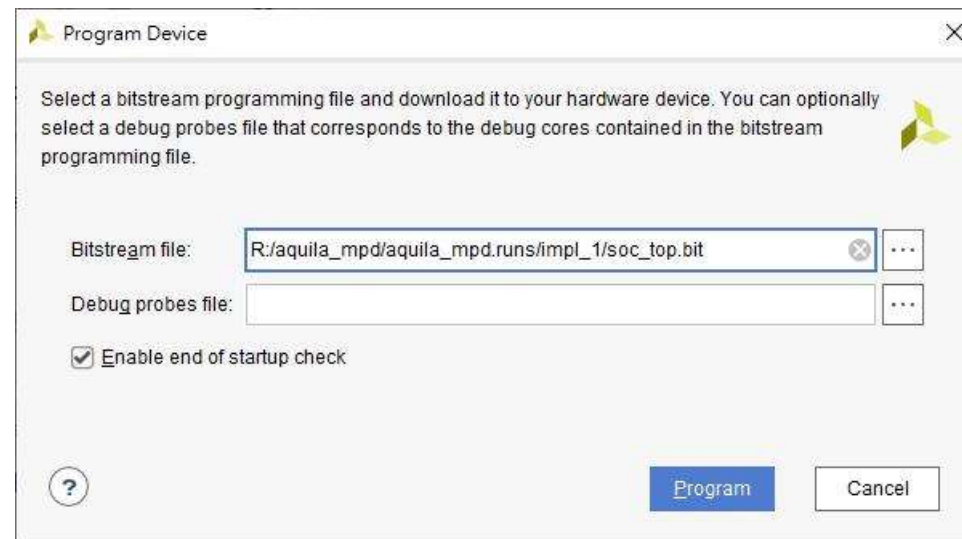
After connecting to the Arty board, click this to program the circuit into the FPGA.

But before you program it, you should have a terminal window connects to the correct UART port first.

Power on the Arty board, and click this to connect to the board. Select "auto connect" in the pop-up dialog box.

Select the Bit File for Programming

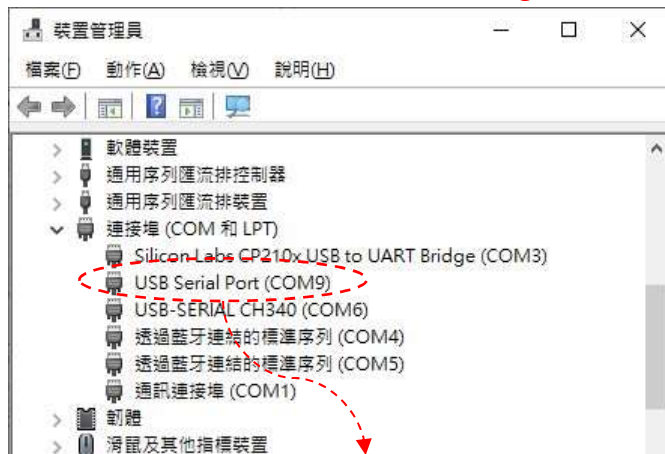
- ❑ The FPGA bit file, `soc_top.bit`, is located under `aquila_mpd/aquila_mpd.runs/impl_1/` after circuit implementation
- ❑ After clicking the “Program device” hyperlink, a file browser will pop-up to let you to load the bit file:



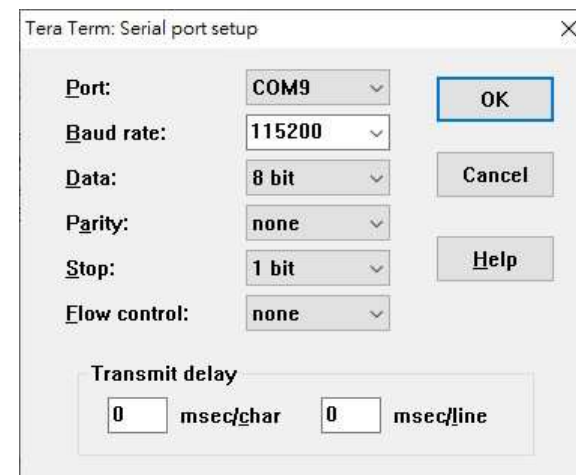
Terminal Settings (1/2)

- ❑ You can use the TeraTerm[†] on Windows, or GTKTerm on Linux as an I/O terminal of the Aquila SoC in FPGA
 - Please do not use “minicom” on Linux!
- ❑ To connect the terminal program to Aquila, you must set the right COM port and UART parameters:

For Windows, check the device manager:



Use the COM port that show up when you plug-in the Arty board. If two ports show up, pick the larger COM number.



[†] <https://ttssh2.osdn.jp/index.html>

Terminal Settings (2/2)

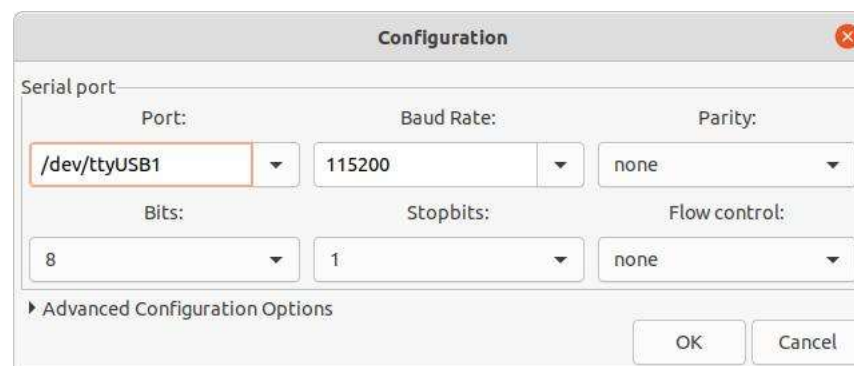
- ❑ For Linux, use “`sudo dmesg`” to see the Arty devices:

```
[2064339.294090] usb 1-7: Product: Digilent USB Device
[2064339.294091] usb 1-7: Manufacturer: Digilent
[2064339.294092] usb 1-7: SerialNumber: 210319A8C7F1
[2064339.297789] ftdi_sio 1-7:1.0: FTDI USB Serial Device converter detected
[2064339.297801] usb 1-7: Detected FT2232H
[2064339.297928] usb 1-7: FTDI USB Serial Device converter now attached to ttyUSB0
[2064339.299920] ftdi_sio 1-7:1.1: FTDI USB Serial Device converter detected
[2064339.299928] usb 1-7: Detected FT2232H
[2064339.300026] usb 1-7: FTDI USB Serial Device converter now attached to ttyUSB1
```

Arty JTAG programming device.

Arty serial port device.

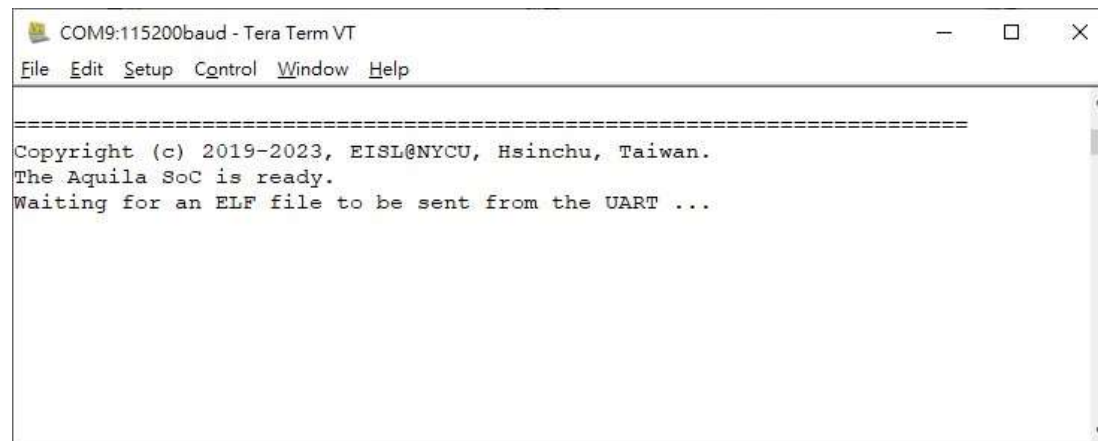
- ❑ GTKTerm configuration:



Run Applications on Real Hardware

Running an Executable on the FPGA

- ❑ Once the FPGA is programmed, a message is shown on the terminal program, waiting for an ELF file to be sent to the Aquila SoC for execution:



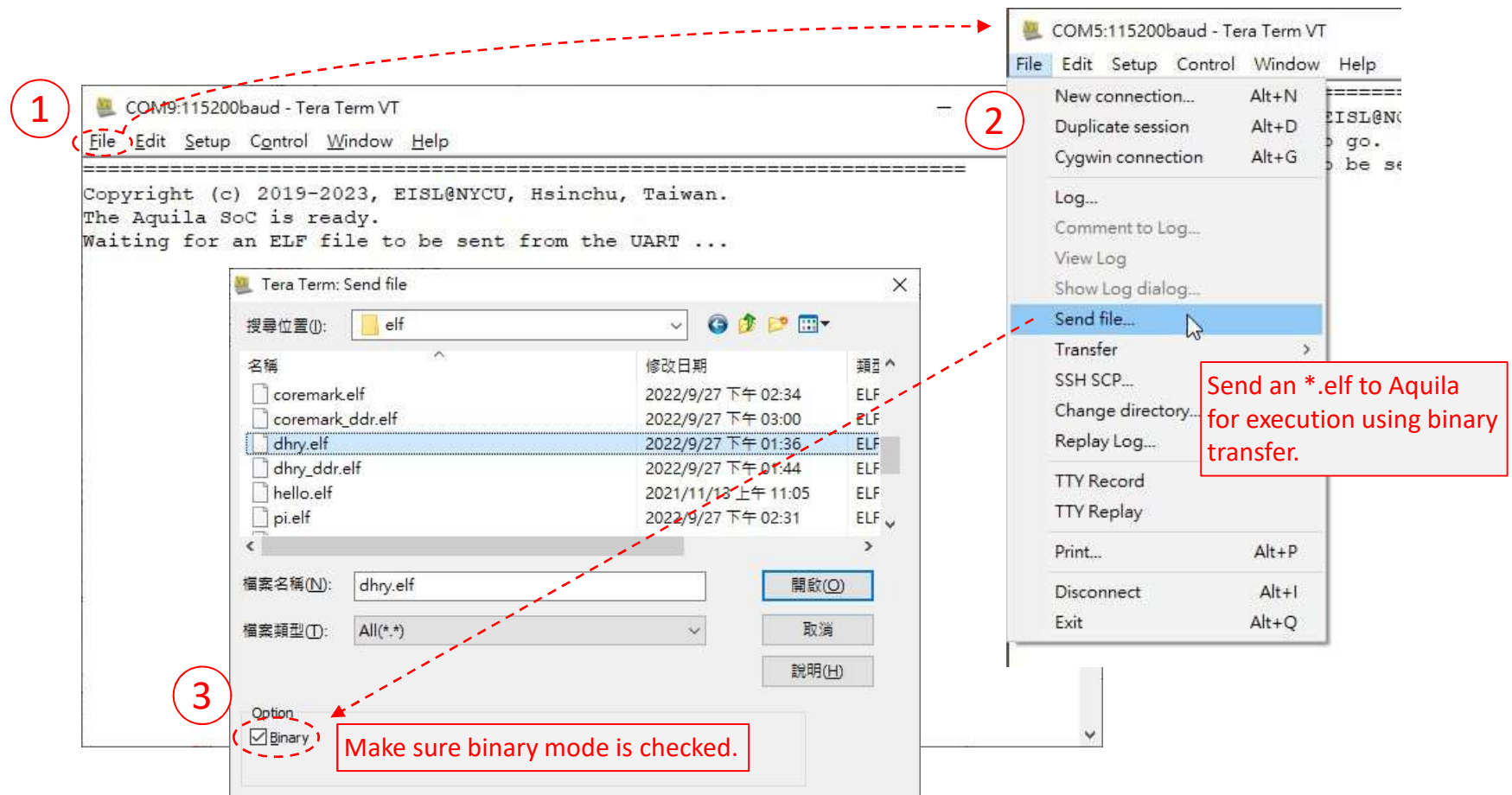
The screenshot shows a terminal window titled "COM9:115200baud - Tera Term VT". The window has a menu bar with "File", "Edit", "Setup", "Control", "Window", and "Help". The terminal text is as follows:

```
=====
Copyright (c) 2019-2023, EISL@NYCU, Hsinchu, Taiwan.
The Aquila SoC is ready.
Waiting for an ELF file to be sent from the UART ...
```

- ❑ Under Linux, you can send an ELF file to FPGA with the shell command: `$cat dhry.elf > /dev/ttyUSB1`

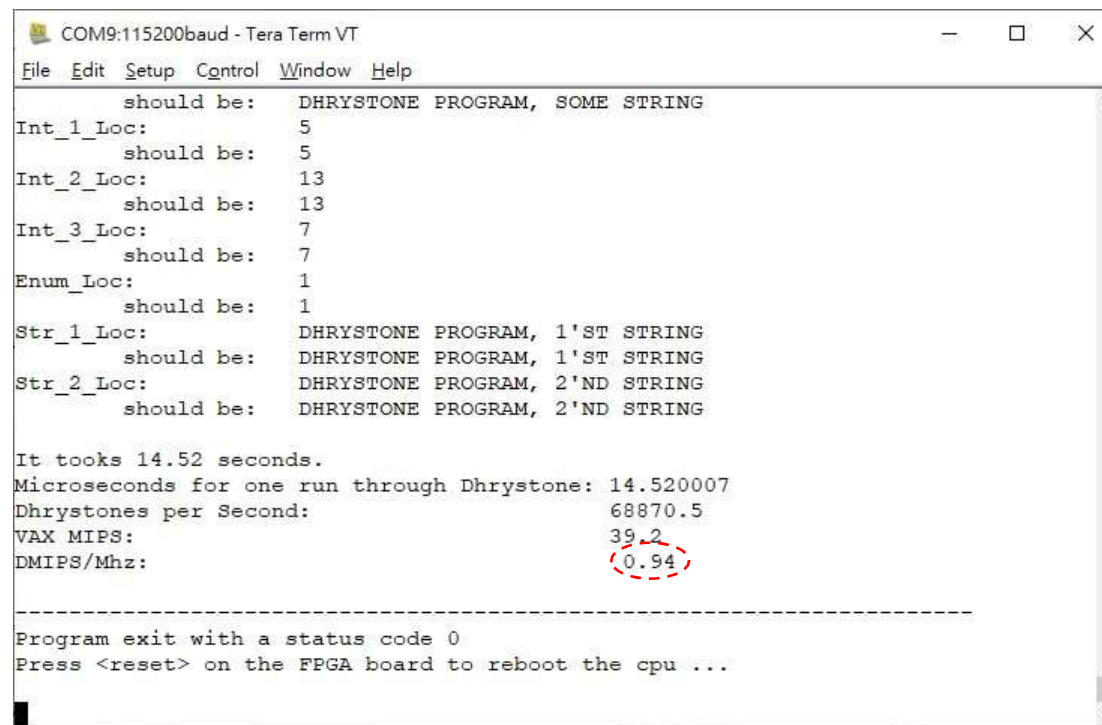
Sending a *.elf to FPGA (Windows)

- ❑ For Windows (TeraTerm), use the “Send file” menu:



Dhrystone Result

- ❑ Send `dhry.elf` to Aquila and you should see:
 - Note: the actual DMIPS number depends on how well you optimize the `ellibc/string.c` functions in HW#0



The screenshot shows a Tera Term VT window titled "COM9:115200baud - Tera Term VT". The window displays the output of the Dhrystone benchmark program. The output includes a list of variables and their values, followed by the execution time and performance metrics. The DMIPS/Mhz value is circled in red.

```
COM9:115200baud - Tera Term VT
File Edit Setup Control Window Help

should be: DHRYSTONE PROGRAM, SOME STRING
Int_1_Loc: 5
should be: 5
Int_2_Loc: 13
should be: 13
Int_3_Loc: 7
should be: 7
Enum_Loc: 1
should be: 1
Str_1_Loc: DHRYSTONE PROGRAM, 1'ST STRING
should be: DHRYSTONE PROGRAM, 1'ST STRING
Str_2_Loc: DHRYSTONE PROGRAM, 2'ND STRING
should be: DHRYSTONE PROGRAM, 2'ND STRING

It tooks 14.52 seconds.
Microseconds for one run through Dhrystone: 14.520007
Dhrystones per Second: 68870.5
VAX MIPS: 39.2
DMIPS/Mhz: 0.94

-----
Program exit with a status code 0
Press <reset> on the FPGA board to reboot the cpu ...
```

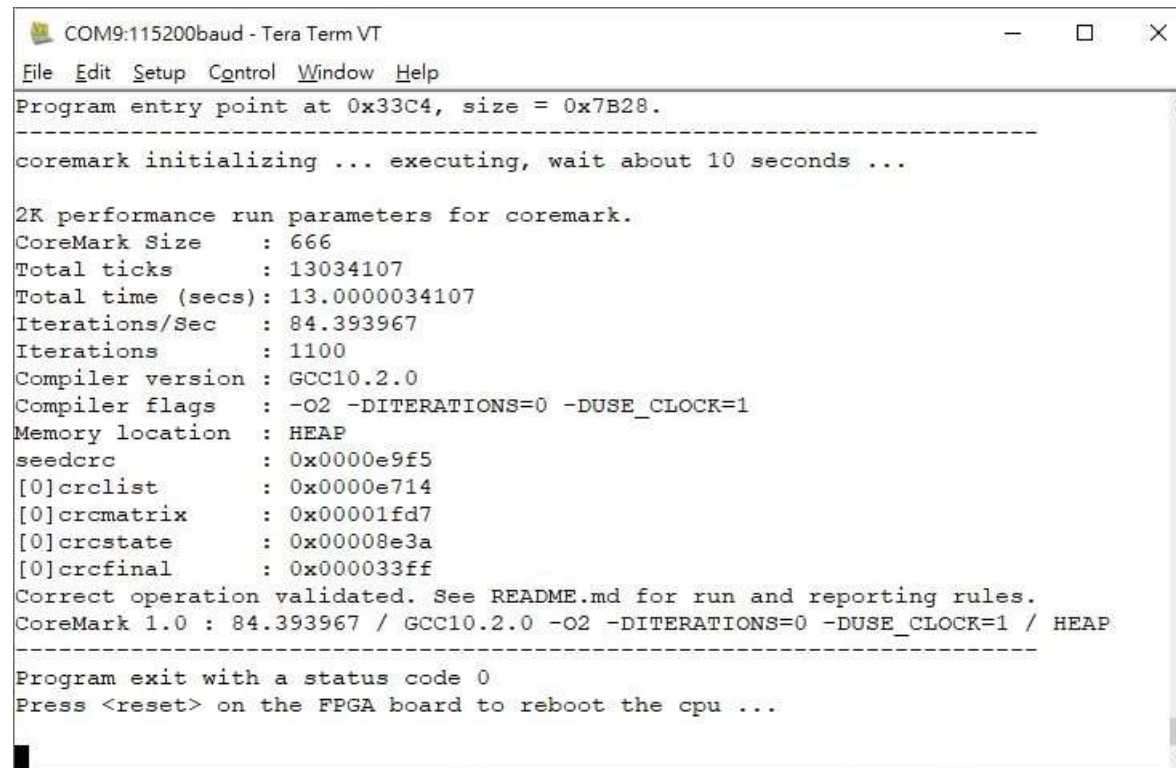
Analyze Aquila Execution on FPGA

- ❑ To analyze the behavior of the Aquila SoC running a program on the real FPGA board, you can use the Integrated Logic Analyzer (ILA)
- ❑ Real-time ILA circuit probing steps:
 - Embed signal probes into your circuit (the Aquila SoC here)
 - Set a trigger condition to capture signal traces to on-chip RAM
 - Perform a post-mortem analysis on a PC afterwards
- ❑ A tutorial on using the ILA is in the Appendix (page 25-).

Profiling the Target Benchmark

CoreMark

- ❑ For this homework, we will use CoreMark as the target application:



```
COM9:115200baud - Tera Term VT
File Edit Setup Control Window Help
Program entry point at 0x33C4, size = 0x7B28.
-----
coremark initializing ... executing, wait about 10 seconds ...

2K performance run parameters for coremark.
CoreMark Size      : 666
Total ticks        : 13034107
Total time (secs)  : 13.0000034107
Iterations/Sec     : 84.393967
Iterations         : 1100
Compiler version   : GCC10.2.0
Compiler flags     : -O2 -DITERATIONS=0 -DUSE_CLOCK=1
Memory location    : HEAP
seedcrc            : 0x0000e9f5
[0]crc1list        : 0x0000e714
[0]crcmatrix       : 0x00001fd7
[0]crcstate        : 0x00008e3a
[0]crcfinal        : 0x000033ff
Correct operation validated. See README.md for run and reporting rules.
CoreMark 1.0 : 84.393967 / GCC10.2.0 -O2 -DITERATIONS=0 -DUSE_CLOCK=1 / HEAP
-----
Program exit with a status code 0
Press <reset> on the FPGA board to reboot the cpu ...
```


Profiling a Program for Optimization

- ❑ Often, 80% of the program execution time resides in 20% of the code → **how do you find the hotspots?**
- ❑ An easy way is to use a software profiler to do:
 - Sampling-based hot spot analysis
 - Insert a timer interrupt service routine (ISR) into your program
 - Interrupt the processor at a fixed frequency, say, 100 Hz, and the ISR collects samples of the PC during execution
 - The PC samples are used to estimate the time spent in a function
 - Counter-based call analysis
 - Insert counter code into every function
 - Record the caller and calling frequency

Example: Profiling on PC

- ❑ In Linux, GCC can be used to profile a program:

```
$ gcc -O2 -pg my_prog.c -o my_prof  
$ ./my_prog  
$ gprof ./my_prog gmon.out > profile.txt
```

- A program compiled with the `-pg` flag will have profiling code inserted into the program
- When executed, the profiling code will collect runtime information and store it to the binary file `gmon.out`
- The command `gprof` can convert `gmon.out` to a text file
- Note that if you use WSL, only WSL 2.0 works. For WSL 1.0, `gprof` only provide call analysis, not hotspot analysis
 - Check your WSL version using dos command: `WSL -l -v`

The Profile of CoreMark

- ❑ On Linux PC with gcc 11.4.0, `profile.txt` looks like:

```
Flat profile:
Each sample counts as 0.01 seconds.
%   cumulative   self           self       total
time  seconds    seconds   calls   s/call   s/call   name
26.88    2.29      2.29 105288660    0.00    0.00  core_list_find
23.71    4.31      2.02 104266440    0.00    0.00  core_list_reverse
10.33    5.19      0.88  523376640    0.00    0.00  core_state_transition

. . . . .

          Call graph (explanation follows)
granularity: each sample hit covers 4 byte(s) for 0.12% of 8.52 seconds
index % time    self  children  called  name
[1]   100.0    0.00    8.52           main [1]
           0.00    8.52        6/6    iterate [2]
           0.00    0.00        1/1    core_list_init [23]

. . . . .
```

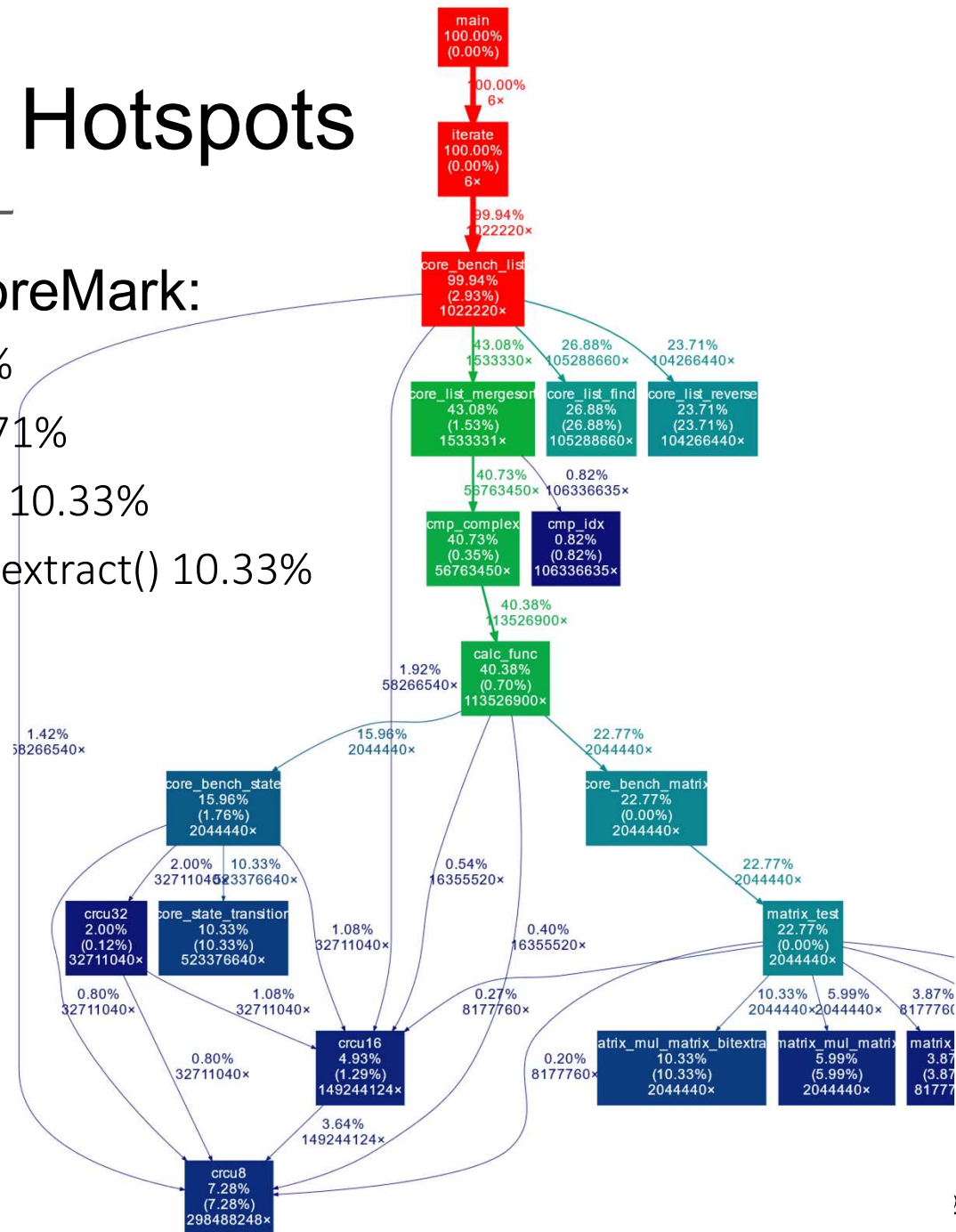
- ❑ You can use `gprof2dot.py` + `graphviz`, to draw a call-graph from `profile.txt`:

```
$ gprof2dot.py profile.txt | dot -Tsvg -o coremark.svg
```

Visualization of Hotspots

□ Top 5 hotspots of CoreMark:

- core_list_find() 26.88%
- core_list_reverse() 23.71%
- core_state_transition() 10.33%
- matrix_mul_matrix_bitextract() 10.33%
- crcu8() 7.28%



Some Comments on using Gprof

- ❑ The result of gprof profiling depends on the gcc version and the processor you use
 - The true top five hotspot for Aquila SoC may not be the same as the ones in the previous slide
- ❑ GCC tends to replace a small function by an inline function, which will mislead the profiling results
 - If a function is inlined, all calls to the function will be replaced by the direct insertion of the function into the caller
 - The GCC compiler flag `-fno-inline-small-functions` should be used in the `Makefile` to avoid inlining

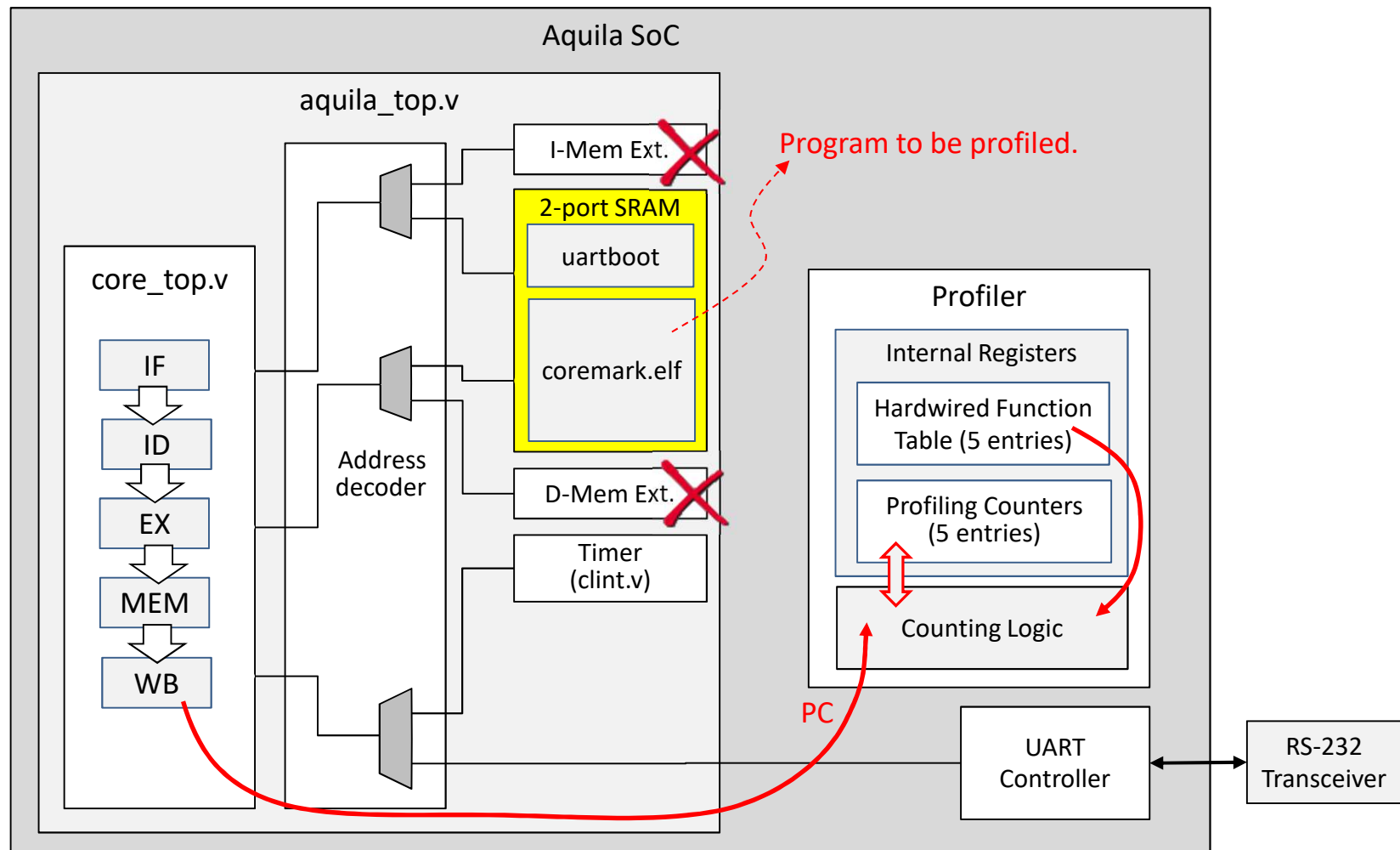
Limitations of Software Profiler

- ❑ May not be available on the target platform
 - You need GCC+Glibc+Linux
 - We use GCC+ellibc, which has no built-in profiling facility
- ❑ Timer interrupts can be obtrusive to the program, especially when the program has its own ISR
 - Storing profiling data in main memory can also be expensive
- ❑ Recursive functions can not be profiled properly
 - Need better granularity
- ❑ Do not differentiate between computations and memory/device accesses

Profiling Using CPU Hardware

- ❑ Since we can modify the hardware of the CPU, can we design a CPU that profile a program during execution automatically?
- ❑ To count the execution cycles of a function, we can design a hardware counter that ticks every time the PC is in a function
 - The starting address and the length of a function is in the *.map file from the compiler
 - We can further analyze that, for a particular function, the ratio of computation cycles versus memory cycles
 - Note: you must take into account the stall cycles

Block Diagram of a CPU with Profiler



Your Homework

- ❑ Add HW code to Aquila such that it can collect the runtime profiling data for the top five hotspots
 - You should also count the total CPU cycles and compute the CPU ratio of each function
 - You should calculate the ratio of computation versus memory cycles for each function

- ❑ Write a 2-page double-column PDF report[†]:
 - Discuss what you have done to collect runtime statistics
 - What have you learned based on the statistics you collected?

[†] A report template (of Microsoft Word format) is available on E3.

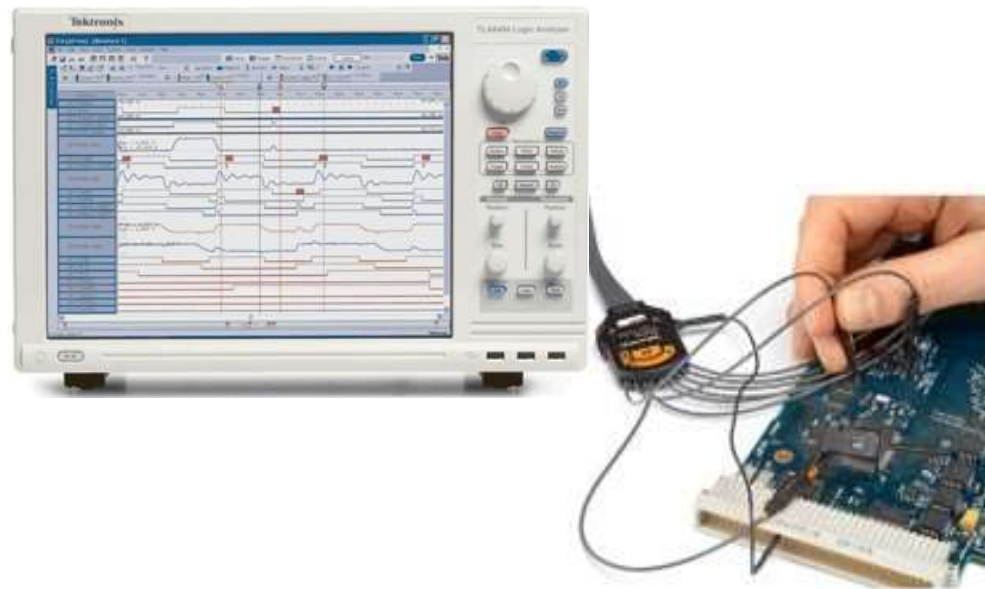
Comments on Report Grading

- ❑ Your report will be graded using the following points:
 - Organization and writing style (25%)
 - The design of your profiling mechanism (25%)
 - Discussions on the profiling results (30%)
 - Comparison of the result to that on a PC
 - What you can say about the computation vs. memory cycles
 - What you can say about the stall cycles
 - Discussions on how to improve Aquila (20%)

Appendix: ILA for Real-Time Circuit Debugging

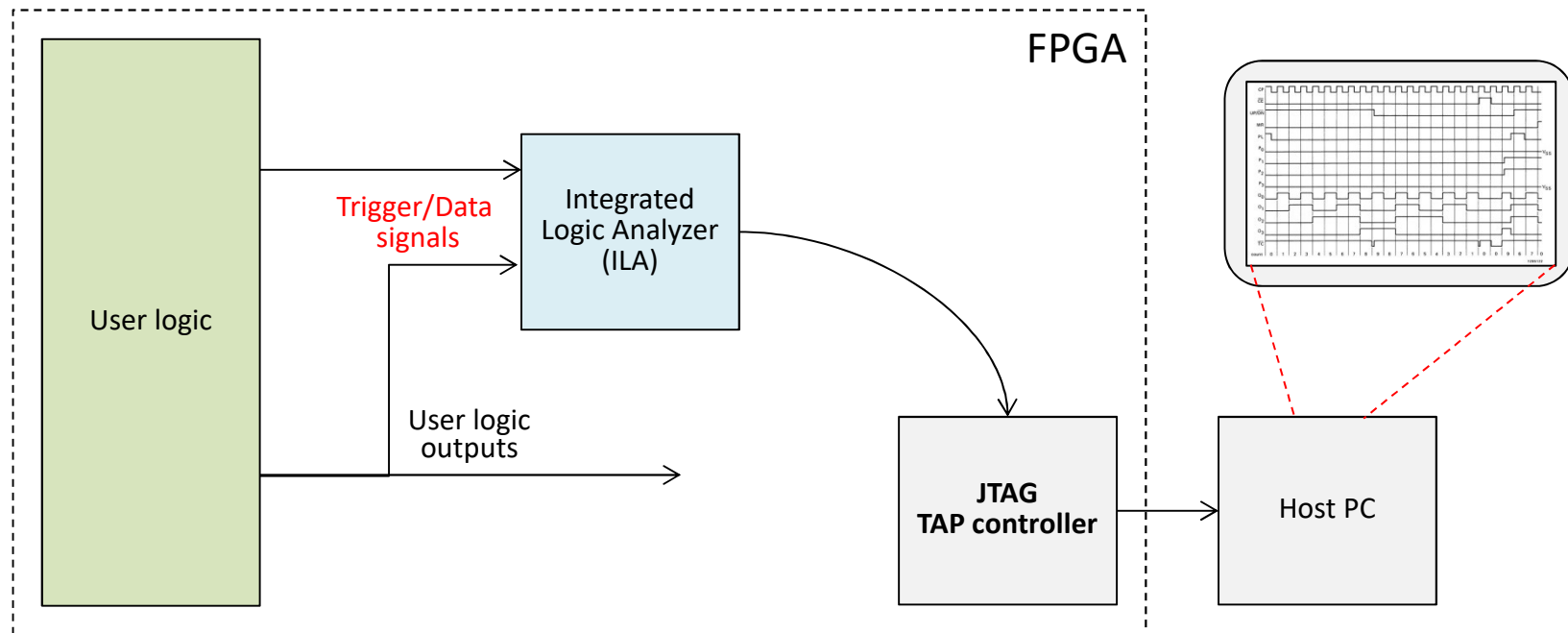
Real-Time Probing Using Vivado

- ❑ Full-system simulations for complex logic and software behaviors would take too much time; and real devices are difficult to simulate
- ❑ In the good old days, for real-time debugging of a digital circuit, we use a logic analyzer for the job



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 “mark” the signals for debugging with one of the 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 here)
- ❑ After marking the signals, you must set up the debug wizard so that ILA can capture the signals at runtime
- ❑ Do not mark the system clock. The waveform viewer has tick markers.

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, **unless the signal is void**
 - Vivado will insert an ILA IP into the synthesized design to monitor and capture these signals at runtime

Mark Debug Signals Using GUI

- ❑ To debug a circuit, open the synthesized design:

The screenshot shows the Vivado 2022.1 GUI with the following components:

- Flow Navigator:** The 'SYNTHESIS' section is expanded, and 'Open Synthesized Design' is highlighted with a red dashed circle.
- Netlist:** The 'data_sel_r' net is selected, and its context menu is open. The 'Mark Debug' option is highlighted with a red dashed circle.
- Bus Net Properties:** The 'General' tab is selected, showing the 'data_sel_r' net.
- Project Summary:** The 'Device' tab is selected, showing the 'core_top.v' device.
- Debug Window:** The 'Debug' window is open, showing a list of unassigned debug nets. The 'Debug' button in the menu bar is also highlighted with a red dashed circle.

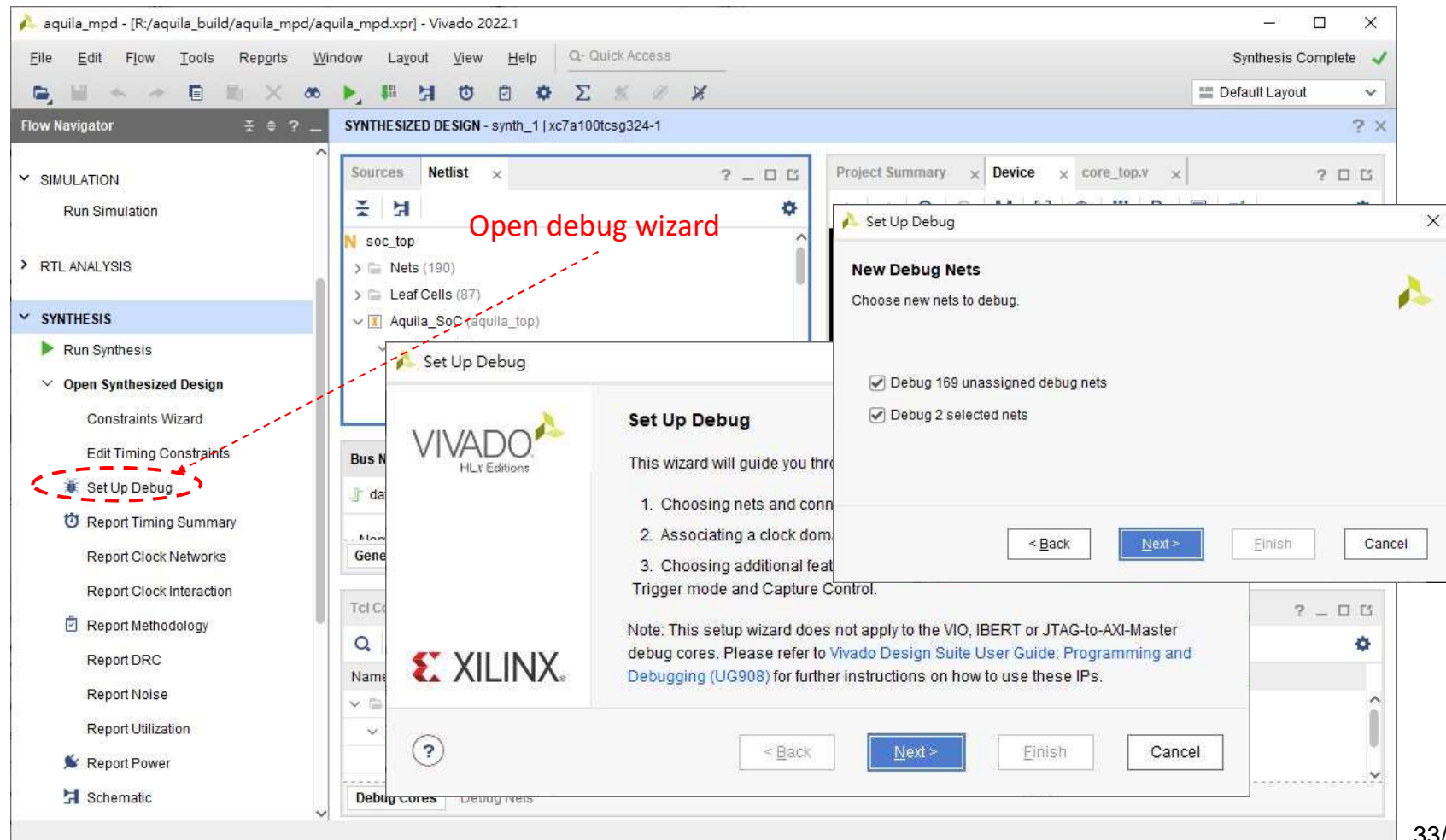
Annotations in the image:

- "You can also select signals to be debugged from here!" (pointing to the context menu)
- "open debug window from the menu bar" (pointing to the Debug button)
- "This windows shows signals to be debugged!" (pointing to the Debug window)

Name	Driver Cell	Driver Pin	Probe Type
Unassigned Debug Nets (169)			
Aquila_SoC/RISCV_CORE0/code_addr_o (19)	FDRE	Q	
Aquila_SoC/RISCV_CORE0/code_addr_o[2]	FDRE	Q	

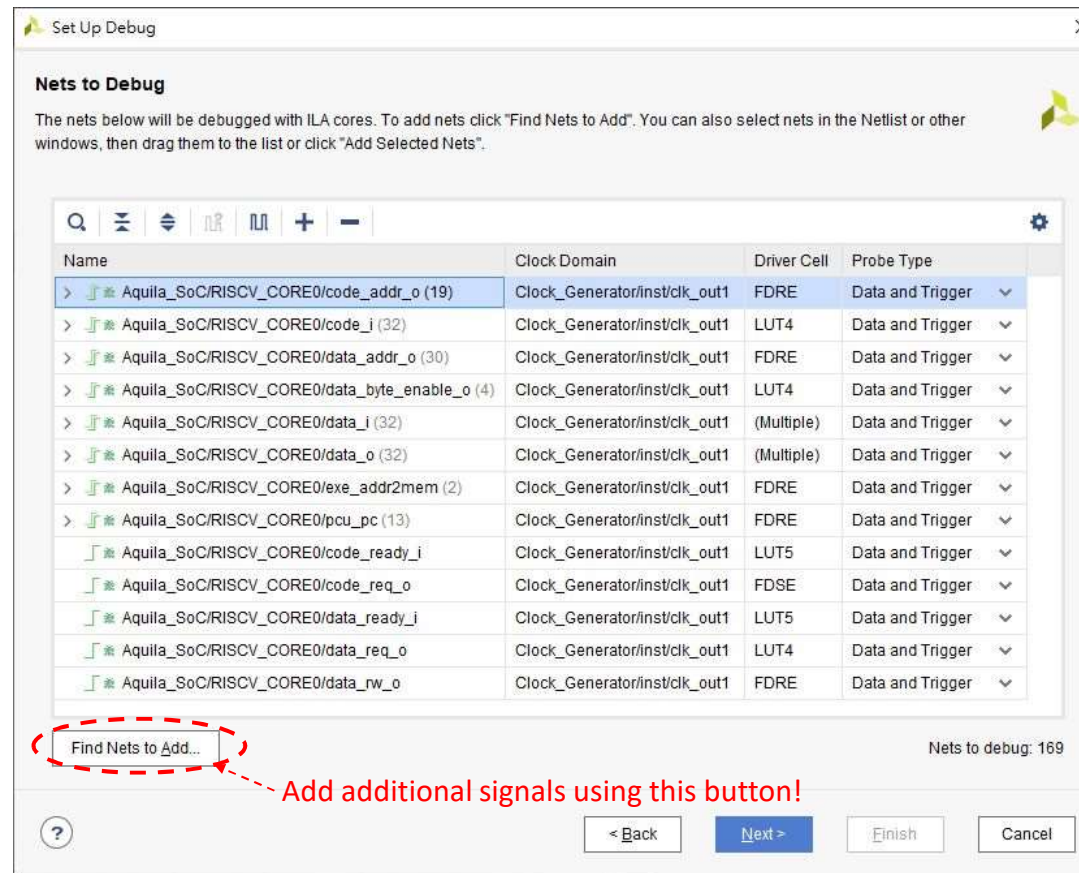
Set Up the Debug Wizard

- ❑ Open the “Set Up Debug” wizard:



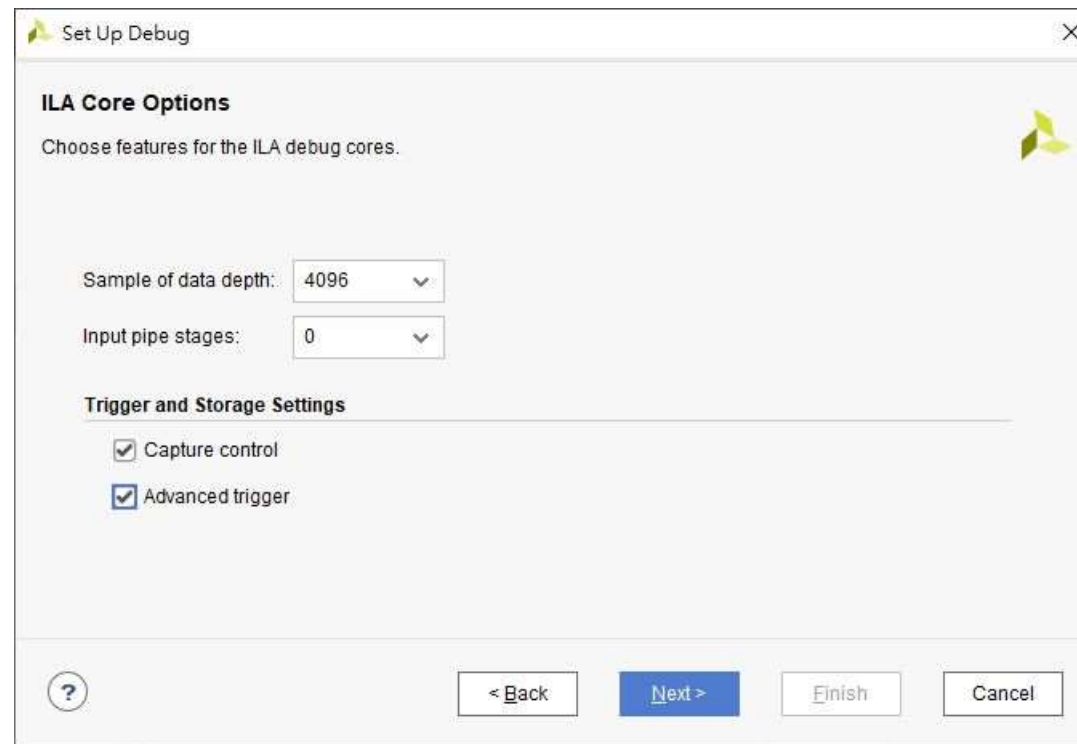
Double-Check Nets to Be Debugged

- ❑ You can add any missing signals in this dialog box
 - Some signals in your Verilog code may be optimized out!



Modify Trigger Options

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



Set Up Debug

ILA Core Options

Choose features for the ILA debug cores.

Sample of data depth: 4096

Input pipe stages: 0

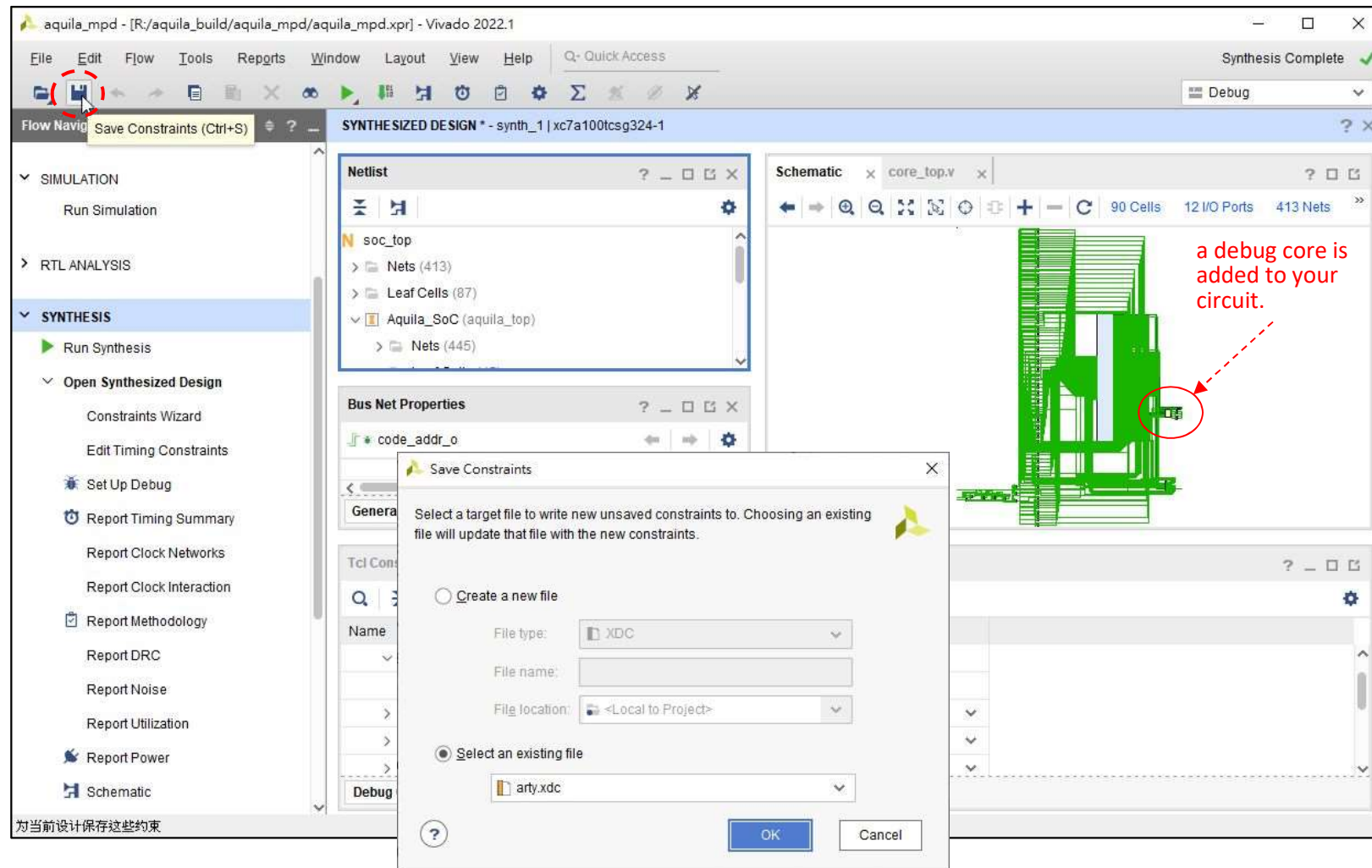
Trigger and Storage Settings

☒ Capture control

☒ Advanced trigger

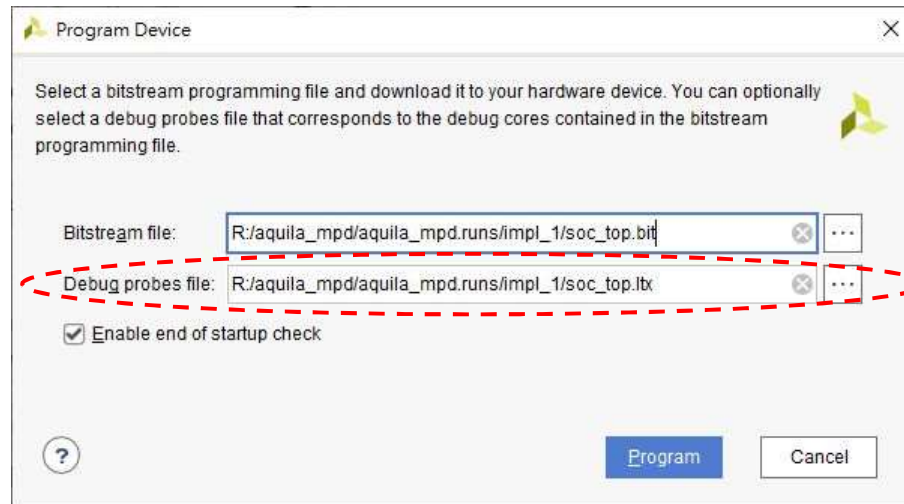
? < Back Next > Finish Cancel

Save the New Debug Constraints



Program the FPGA for ILA

- ❑ After generating the bit file, we can program the FPGA
- ❑ Once you hit the “program device” menu item, you will see that an extra ILA configuration file is selected:



The Hardware Manager with ILA View

The screenshot displays the Vivado 2022.1 Hardware Manager interface. The top menu bar includes File, Edit, Flow, Tools, Reports, Window, Layout, View, and Help. The main window is titled "HARDWARE MANAGER - localhost:xc7a100t_0/Digilent/210319B26B02A".

On the left, the Flow Navigator shows the project structure, including "Set Up Design", "Report Timing", "Report Constraints", "Report Metrics", "Report Design", "Report Utilization", "Report Power", and "Schematic". The "IMPLEMENTATION" section is expanded, showing "Run Implementation", "Open Implementation", and "Generate Bits". The "PROGRAM AND DEPLOY" section is also expanded, showing "Open Hardware", "Open Target", "Program", and "Add Configuration".

The central pane shows the "Hardware" tab, which lists the hardware components and their status:

Name	Status
localhost (1)	Connected
xc7a100t_0 (2)	Programmed

Below this, the "Hardware Device Properties" for "xc7a100t_0" are shown:

Name	Part	ID code
xc7a100t_0	xc7a100t	13631093

The right pane shows the "Waveform - hw_ila_1" window, which displays the runtime waveform. The waveform is a digital signal with a value of 0000. The "ILA Status" is "Idle". The "Core status" is "Idle". The "Capture status" is "Window 1 of 1".

Red annotations highlight key features:

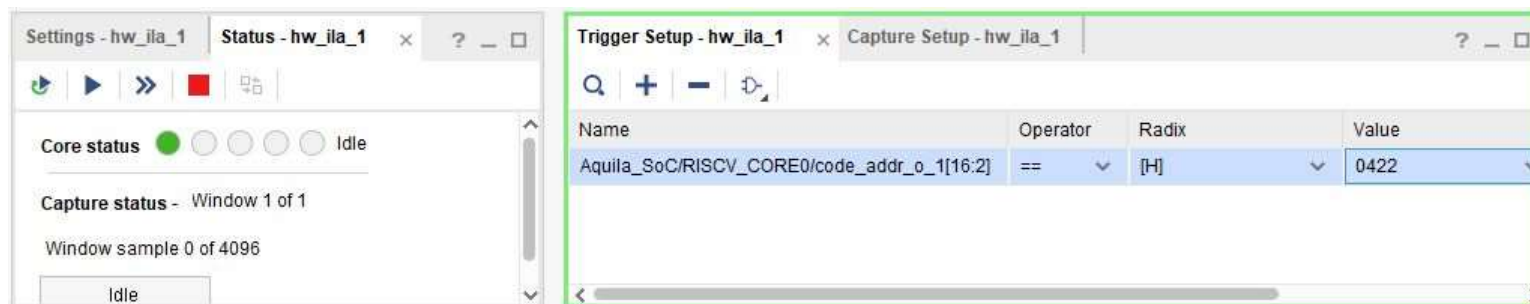
- "Each clock domain has one ILA tab." points to the "hw_ila_1" tab.
- "Runtime waveform window." points to the waveform display area.
- "Trigger setup window" points to the "Trigger Setup - hw_ila_1" window.
- "ILA configuration window" points to the "Settings - hw_ila_1" window.
- "Signals which you can capture" points to the "Aquila_S..._1[16:2]" and "Aquila_So..._1[31:28]" signals in the "Dashboard Options" table.

The bottom pane shows the "Tcl Console" with the following commands and output:

```
program_hw_devices [get_hw_devices xc7a100t_0]
INFO: ILAtools 27-31641 End of startup status: HIGH
```


Setting a Trigger

- ❑ A trigger is a signal condition that tells the ILA to begin capturing waveforms
- ❑ Set the trigger condition
 - The instruction address is `code_addr_o[31:0]` in `core_top.v`
 - If the `main()` is at `0x1088`, we trigger the ILA when `code_addr_o[16:2]` equals `0x422`.



Capturing the Signals

- ❑ Now, you can activate the ILA, then send a program from the UART to FPGA to capture the waveform

The screenshot shows the Vivado IDE interface with the following components:

- Flow Navigator:** Shows the project hierarchy with sections like Settings, Add Sources, Language Te, IP Catalog, IP INTEGRATOR, SIMULATION, RTL ANALYSIS, and SYNTHESIS.
- HARDWARE MANAGER:** Displays the hardware configuration for the target device (localhost/xilinx_tcf/Digilent/210319B26B02A).
- Waveform - hw_ila_1:** Shows the ILA status as 'Idle' and a table of signals and their values.
- Trigger Setup - hw_ila_1:** Shows the trigger condition: 'Aquila_SoC/RISCV_CORE0/code_addr_o_1[16:2] == [H] 0422'.

Name	Value
Aquila_S..._1[16:2]	0000
Aquila_So...o[31:28]	0
Aquila So..._I[31:0]	00000000

Core status: ● Idle

Analyze the Captured Waveform

aquila_mpd - [R:/aquila_build/aquila_mpd/aquila_mpd.xpr] - Vivado 2022.1

File Edit Flow Tools Reports Window Layout View Help Q- Quick Access write_bitstream Complete ✓

Dashboard ▾ Default Layout ▾

Flow Navigator

HARDWARE MANAGER - localhost/xilinx_tcf/Digilent/210319B26B02A

core_top.v x hw_ila_1 x

Waveform - hw_ila_1

ILA Status: Idle

Name	Value
Aquila_SoC/RISC_V...de_addr_o_1[16:2]	0422
Aquila_SoC/RISC_V...de_addr_o[31:28]	0
Aquila_SoC/RISC_V...CORE0/code_i[31:0]	95c2a103
Aquila_SoC/RISC_V...yte_enable_o[3:0]	3
Aquila_SoC/RISC_V...data_addr_o[31:2]	3fff72b
Aquila_SoC/RISC_V...RE0/pcu_pc_1[1:0]	0
Aquila_SoC/RISC_V...RE0/pcu_pc_0[27:1]	000

Updated at: 2022-Oct-05 18:28:10

Settings - hw_ila_1 Status - hw_ila_1

Core status: Idle

Capture status - Window 1 of 1

Trigger Setup - hw_ila_1 Capture Setup - hw_ila_1

Name	Operator	Radix	Value
Aquila_SoC/RISC_V...CORE0/code_addr_o_1[16:2]	==	[H]	0422

Tcl Console Messages Serial I/O Links Serial I/O Scans

Store a Trigger Value in Registers

- ❑ In ILA, you can see waveforms of registers easily, but not so easy to check C variables
- ❑ You can use inline assembly code to store a variable into a register so the ILA can display its value:
 - Can be used to **set a trigger point!**
 - Note, the local variable is assumed to be stored in a register

```
int var1 = 123;

void main()
{
    int var2 = 456;

    /* Save a global variable to register t1 */
    asm volatile ("lui t0, %hi(var1)");
    asm volatile ("lw  t1, %lo(var1)(t0)");

    /* Save a local variable to register t1 */
    asm volatile ("addi t1, %1, 0" : "=r"(var2) : "r"(var2));
}
```