

HW#0 Simulation of a HW-SW Platform



Chun-Jen Tsai
National Yang Ming Chiao Tung University
09/17/2021

Target Technology of the Aquila SoC

- ❑ The RTL model of the Aquila core is written in Verilog
 - 23 files, 7038 lines of code.
- ❑ Aquila SoC has been verified using two series of FGPAs from Xilinx using the Vivado EDA tool
 - On Kintex XC7K325T, clocked at 100MHz
 - On Artix XC7A35T, clocked at 50MHz

Xilinx EDA Tools

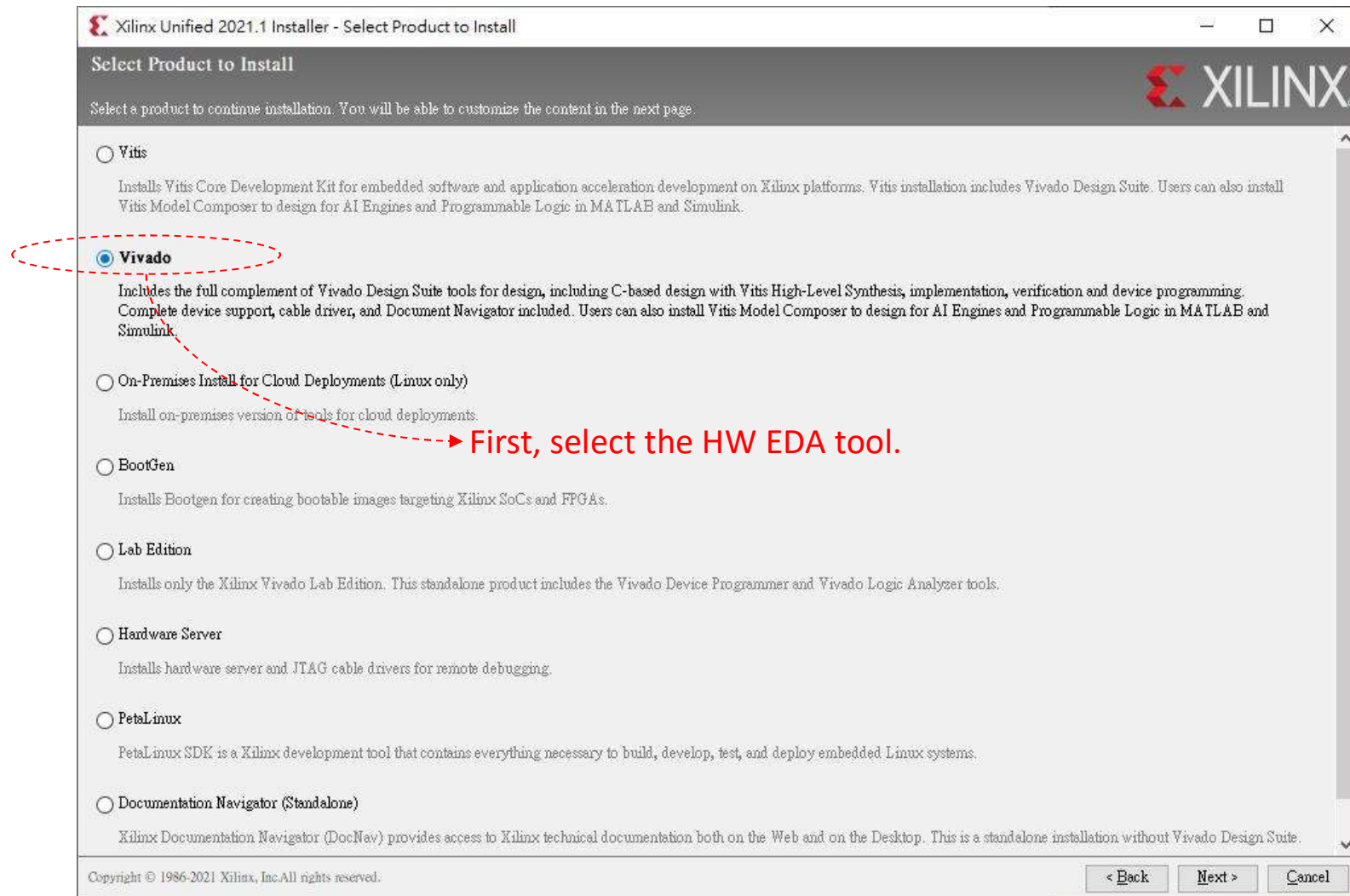
- ❑ Xilinx EDA tools are for SW-HW system design
 - Vitis
 - Software IDE (only for ARM and Microblaze processors)
 - High-Level Synthesis (HLS) HW design using C/C++
 - Support Xilinx FPGA & AI chips
 - Rely on Vivado for FPGA HW implementation
 - Vivado
 - HW design using Verilog or VHDL
- ❑ In this course, we use Vivado for HW design, and command-line GCC for SW design

Install Your Vivado Design Suite

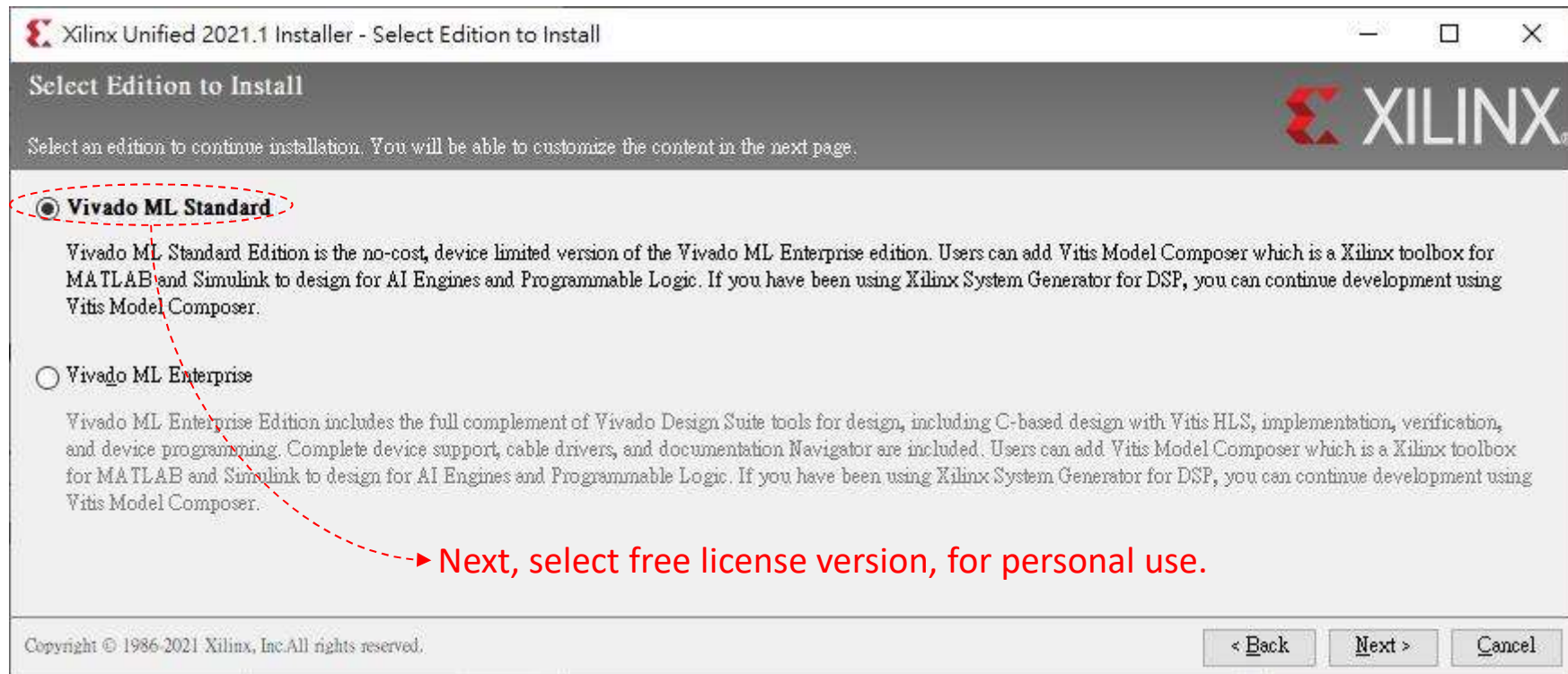
- ❑ You can download the installer for Windows or Linux:
 - <https://www.xilinx.com/support/download.html>
 - MacOS is not supported by Xilinx!

- ❑ The installation requires about 50 GiB of disk space, depending on the FPGA devices you selected
 - You must create a Xilinx account before installation
 - Please install the Vivado standard version
 - For this course, we only need Artix-7 FPGA support

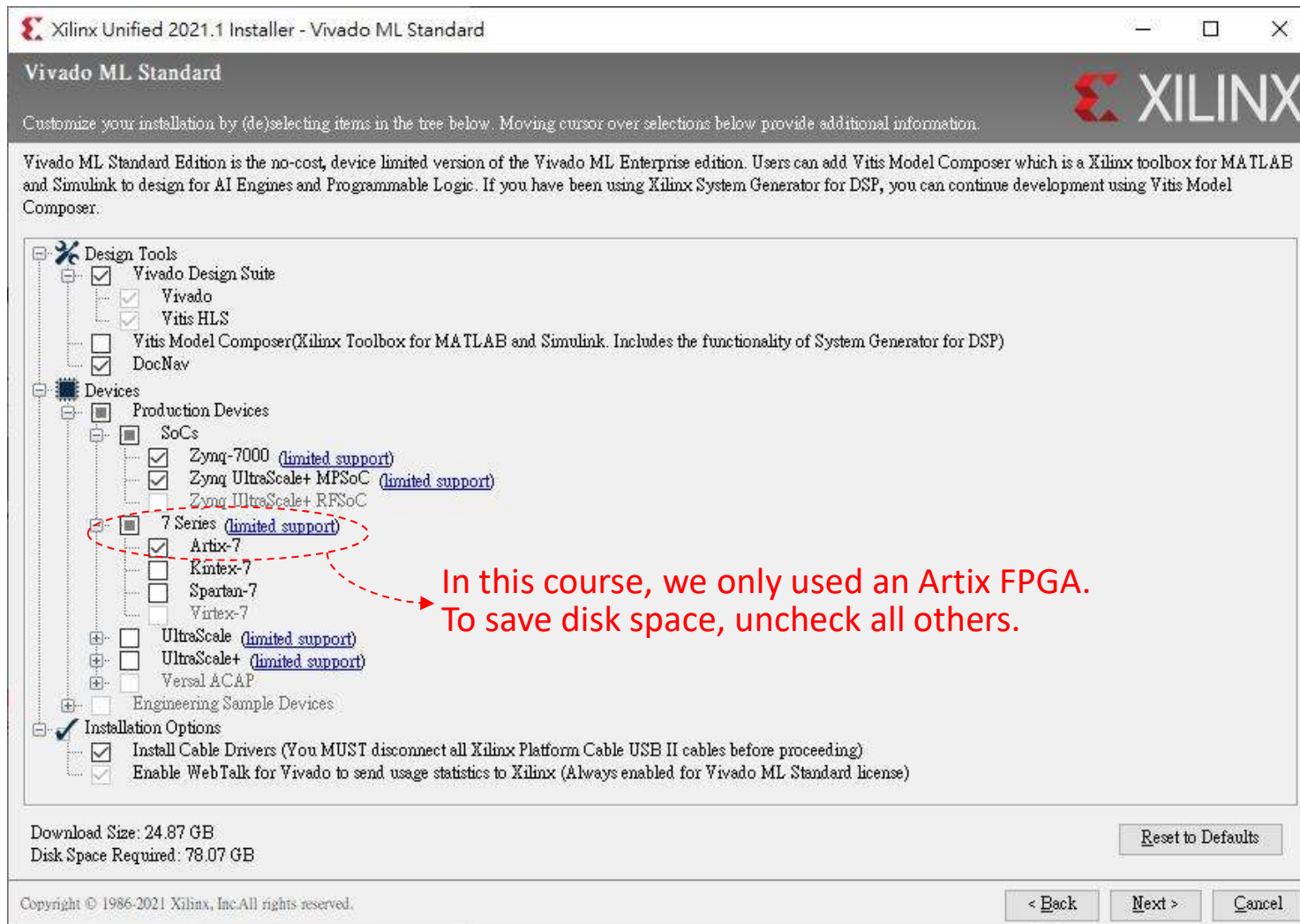
Vivado Installation Guide (1/3)



Vivado Installation Guide (2/3)



Vivado Installation Guide (3/3)



Installation of Arty Board Definitions

❑ Install the Arty board file:

- Go to <https://github.com/Digilent/vivado-boards>, download the directory arty-a7-35/*



- Make a directory Digilent/ under C:/<INST_DIR>/Vivado/2021.1/data/xhub/boards/XilinxBoardStore/boards/
- Put arty-a7-35/* under Digilent/

† <INST_DIR> is the directory of your Vivado installation.

Creation of an Aquila SoC Workspace

- ❑ Download `aquila_mpd_build.zip` from E3
 - It contains a TCL script that generates the Aquila workspace

```
aquila_mpd_build -- src/  
                  |  
                  +-- build.tcl
```

- ❑ Unzip the package to a local directory, type the following command under a Windows command prompt:

```
C:\<INST_DIR>\Vivado\2021.1\bin\vivado.bat -mode batch -source build.tcl
```

If you use the Linux bash prompt:

```
<INST_DIR>/Vivado/2021.1/bin/vivado -mode batch -source build.tcl
```

The generated workspace will be in `aquila_mpd/`.

Open the Workspace

- ❑ Under Windows, just double-click the file `aquila_mpd.xpr`
- ❑ Under Linux bash, just type the command:

```
<INST_DIR>/Vivado/2021.1/bin/vivado aquila_mpd.xpr
```

Overview of the Aquila Workspace

Click this to run simulation.

The screenshot shows the Vivado 2021.1 interface for the 'aquila_mpd' project. The left sidebar contains the 'PROJECT MANAGER' and 'SIMULATION' sections. The 'SIMULATION' section has a 'Run Simulation' button circled in red. The 'PROJECT MANAGER' section shows the 'Sources' pane with a tree view of the project files. The 'soc_top (soc_top.v) (3)' file is circled in red. The 'bootrom.mem' file is also circled in red. The 'aquila_tb (aquila_tb.v) (2)' file is circled in red. The 'Project Summary' pane on the right shows the 'Overview' tab with project details. The 'Design Runs' pane at the bottom shows a table of design runs.

Annotations:

- Top-level module of the SoC. (points to `soc_top (soc_top.v)`)
- Boot code of the SoC. (points to `bootrom.mem`)
- Top-level simulation module. (points to `aquila_tb (aquila_tb.v)`)

Project Summary:

- Project name: aquila_mpd
- Project location: R:/aquila_mpd_build/aquila_mpd
- Product family: Artix-7
- Project part: Arty A7-35 (xc7a35ticsg324-1L)
- Top module name: soc_top
- Target language: Verilog
- Simulator language: Mixed

Design Runs:

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAM	URAM	DSP	Start
synth_1	constrs_1	Not started													
impl_1	constrs_1	Not started													

System Memory Map of the SoC

- ❑ Aquila follows the “memory-mapped system model”
 - Every **slave** circuit block has an address space assigned to it!
- ❑ The memory map of the Aquila SoC on Arty A7-35:

Address Range	Circuit Block	Access Type
0x0000 0000 ~ 0x0000 FFFF	On-chip memory (64KB) that stores the boot code.	Uncached
0x8000 0000 ~ 0x8FFF FFFF	DDR3 DRAM (256MB).	Cached
0xC000 0000 ~ 0xCFFF FFFF	I/O device space: UART and SD card.	Uncached
0xF000 0000 ~ 0xFFFF FFFF	System device space: CLINT (timer).	Uncached

Installation of the SW Toolchain

- ❑ We use the 32-bit RISC-V GCC 8.2 toolchain (generic ELF/Newlib version) for SW design
- ❑ You can download a pre-compiled toolchain for Linux[†] from the following link:
https://www.cs.nctu.edu.tw/~cjtsai/riscv32_gcc.tgz
 - Just unzip it under `/opt` of your Linux system and add `/opt/riscv/bin` to your command path.
- ❑ Or, you can follow the instructions and build the tools:
<https://github.com/riscv/riscv-gnu-toolchain>

[†] If you use Windows, you can install WSL to use the toolchain.

Sample Software

- ❑ The sample software source tree for this course:

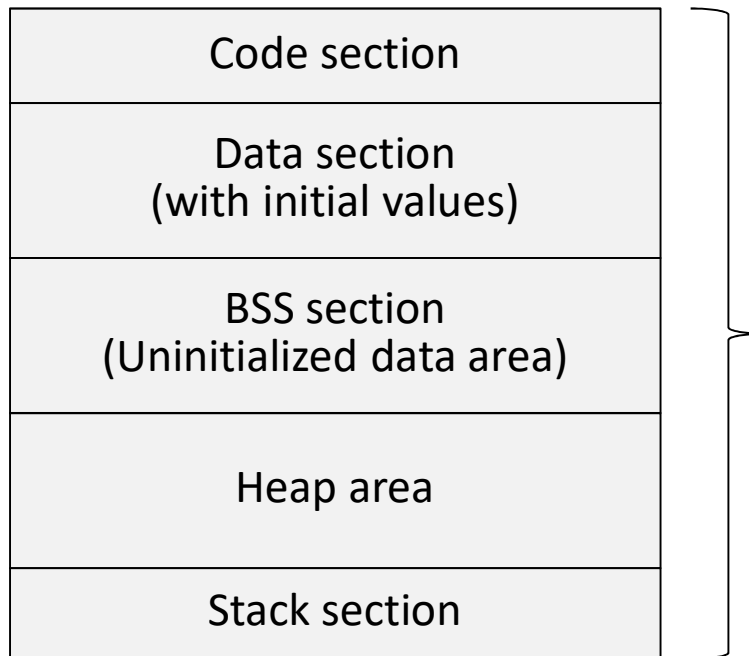
```
aquila_sw +- uartboot/    → A boot code for on-chip memory
           |
           +- elibc/       → A small C library
           |
           +- Dhrystone/   → The Dhrystone benchmark
```

- ❑ Download `aquila_sw.tgz` from E3. Unpack the file under your Linux system. You can build the software by simply typing “make” in each source directory.

Please read and understand the Makefiles!

Runtime Memory MAP

- ❑ A typical runtime memory map:



These sections do not have to occupy contiguous memory areas.

A normal POSIX *.elf executable file format allows a non-contiguous memory layout.

However, the *.ebf format used in lab 1 ~ lab 3 assumes a contiguous memory layout.

- ❑ A linker script (*.ld) can be used to control the memory layout of an executable file

Behavior Simulation Using Vivado Sim

- ❑ In the Aquila workspace, there are two top-level modules:
 - `soc_top.v` – for circuit synthesis
 - `aquila_tb.v` – for circuit simulation, provides a simulated clock and the reset logic.
- ❑ The normal ROM image of the Aquila SoC uses a real UART device to read an executable such as the `dhry.ebf` for execution
- ❑ For simulation, the `uart.v` module does not support simulated I/O properly!

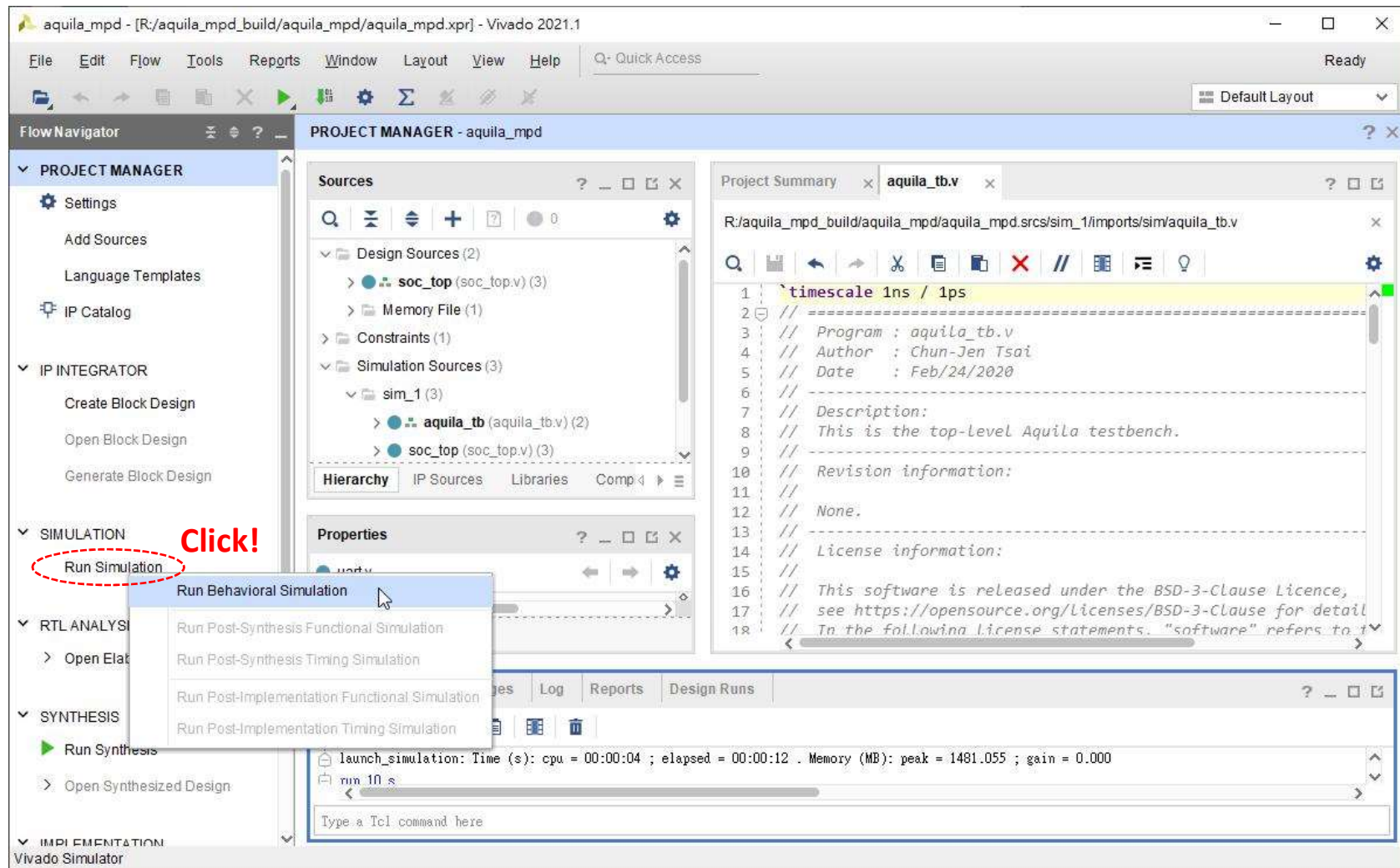
Boot ROM for Simulation

- ❑ We can compile the Dhrystone program into the ROM image for simulation
 - The original Dhrystone program uses the floating point library to compute DMIPS/Mhz. This will make it more difficult to trace the program using a waveform simulator

- ❑ For DMIPS calculation, you can modify it so that:
 - Only integer computation are used[†]
 - Disable `printf()`
 - The DMIPS/MHz value (scaled to an integer) can be stored in a register so that the simulator can show its value

[†] That is, you can use scaled fixed-point arithmetic to compute the DMIPS/MHz.

Run the Simulation



Vivado Simulator Window

Simulation time

Zoom waveform to fit window

Pick the module whose signals you want to observe!

The screenshot displays the Vivado Simulator Window for a behavioral simulation. The top toolbar features a simulation time dropdown set to 10 us. The main window is divided into three panes: Flow Navigator on the left, Objects in the center, and a waveform viewer on the right. The Flow Navigator shows a hierarchy of modules, with 'RISCV_CORE0' selected. The Objects pane lists various signals like clk_i, rst_i, and data_i. The waveform viewer displays a timing diagram for selected signals, with a zoom button circled. The Tcl Console at the bottom shows simulation logs.

Name	Value	Data Type
clk_i	0	Logic
rst_i	0	Logic
stall_i	0	Logic
init_pc_addr	00000000	Array
code_i[31:0]	XXXXXXXX	Array
code_read	1	Logic
code_addr	000000a4	Array
code_req_1	1	Logic
data_i[31:0]	XXXXXXXX	Array
data_ready	X	Logic
data_of31	00000000	Array

Tcl Console

```
INFO: [USF-XSim-96] XSim completed. Design snapshot 'aquila_tb_behav' loaded.  
INFO: [USF-XSim-97] XSim simulation ran for 1000ns  
launch_simulation: Time (s): cpu = 00:00:02 ; elapsed = 00:00:05 . Memory (MB): peak = 1481.055 ; gain = 0.000
```

Sim Time: 1 us

Tracing the Assembly Code

- ❑ After you make the ROM image, there should be an
* `.objdump` file that contains the assembly code of the
compiled program

- ❑ To understand the assembly code, you need to know:
 - The instruction set architecture (ISA)
 - The Application Binary Interface (ABI) defined by the CPU
designer

Sample Assembly Code

❑ The assembly code of the bootrom file :

Aquila execution
begins here!

```
dhry.out:      file format elf32-littleriscv
```

```
Disassembly of section .text:
```

```
00000000 <crt0>:
```

```
0:      ff010113      addi   sp,sp,-16
4:      00112623      sw     ra,12(sp)
8:      000062b7      lui    t0,0x6
c:      3a22ac23      sw     sp,952(t0) # 63b8 <sp_store>
10:     000062b7      lui    t0,0x6
14:     38c2a103      lw     sp,908(t0) # 638c <stack_top>
18:     4c5020ef      jal    ra,2cdc <main>
1c:     000062b7      lui    t0,0x6
20:     3b82a103      lw     sp,952(t0) # 63b8 <sp_store>
24:     00c12083      lw     ra,12(sp)
28:     01010113      addi   sp,sp,16
2c:     00008067      ret
```

```
00000030 <Proc_2>:
```

```
30:     000097b7      lui    a5,0x9
34:     ae87c703      lbu    a4,-1304(a5) # 8ae8 <Ch_1_Glob>
```

```
. . . .
```

Showing Register Values

- ❑ Note that the registers of Aquila is declared in the file `reg_file.v`.
- ❑ There are 32 registers $x[0] \sim x[31]$ for a RISC-V processor. Each register also has an assembly alias
 - For example, $x[1]$ is the return address register `ra`, $x[2]$ is the stack pointer `sp`, etc.
- ❑ You can open the `reg_file` module in the simulator, and drag the register signals to the watch window to see their waveforms

Storing a Debug Value in Registers

- ❑ You can use inline assembly to store a variable into a register so the simulator can display its value:
 - Similar to debugging a program using `printf()`!
 - Here, we assume the local variable is stored in a register by the compiler. This would be the case for the first few locals.

```
int var1 = 123;

void variable_to_reg()
{
    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, %0, 0" : "=r"(var2));
}
```

Your Homework

- ❑ Go through the entire HW-SW source code
 - Browse the HW source code, understand the top-level organization of the simulated SoC (trace `aquila_top.v`)
- ❑ Trace the boot code, uartboot and the application code, Dhrystone. Modify the source code of Dhrystone and the Makefile such that you can build a bootrom.mem that contains the Dhrystone program for simulation
- ❑ Simulate the execution and determine the DMIPS/Mhz of the Aquila
 - **NOTE: the number will NOT be around 0.9 DMIPS/MHz**
- ❑ You do not have to write a report for this homework!