HW#0 Simulation of a HW-SW Platform

Chun-Jen Tsai
NYCU
09/15/2023

Homework Goal

- In this homework, you will learn how to simulate a HW-SW system using a waveform simulator
- You must know how to trace the execution of a program (Dhrystone) at circuit level
  - Based on your analysis using HW-SW co-simulation, you must optimize the standard library to speed up the program
  - This lab only involves software optimization, no hardware modification is necessary
- This homework is just for practice, there is no deadline

Target Technology of the Aquila SoC

- Here, we use the Aquila processor without the I/D-caches and external DRAM
  - The RTL model of the processor is written in Verilog
    - The model contains 30 files, 10,000+ lines of code
  - Full-system simulation is possible with waveform simulator

- Toolchains used for HW-SW system development:
  - HW: Xilinx Vivado 2023.1 for FPGA circuit design & simulation
  - SW: RISC-V 32-bit GCC 12.2.0

Introduction to AMD/Xilinx EDA Tools

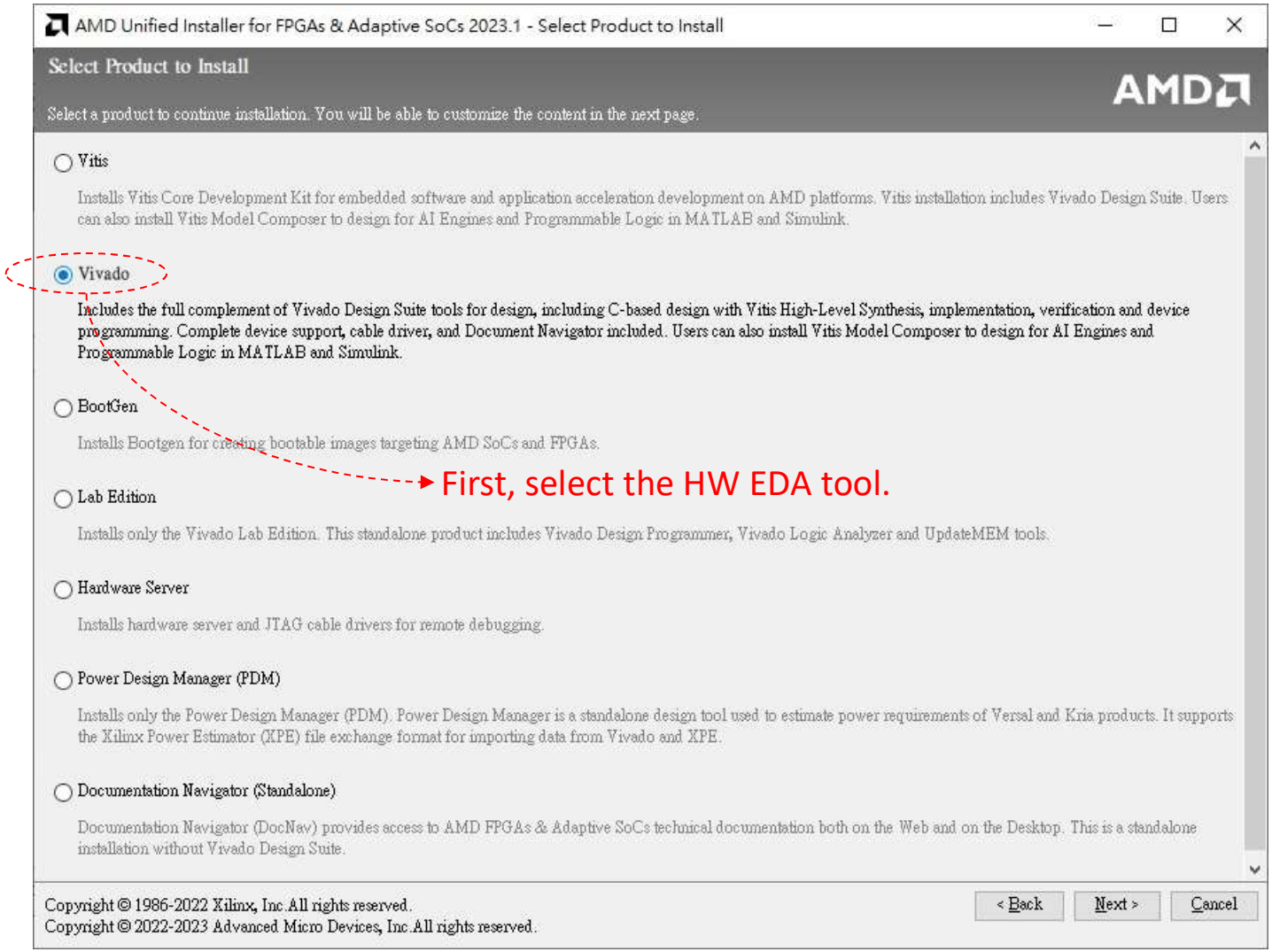
- Vivado (for HW design only)
  - Support traditional HW design flow using Verilog or VHDL
  - Includes circuit synthesizer, simulator, and embedded logic analyzer
- Vitis (for both HW/SW design)
  - Software IDE (only for ARM and Microblaze processors)
  - High-Level Synthesis (HLS) HW design using C/C++
  - Support Xilinx FPGA & AI chips
  - Rely on Vivado for FPGA HW implementation

Installation of Vivado Design Suite

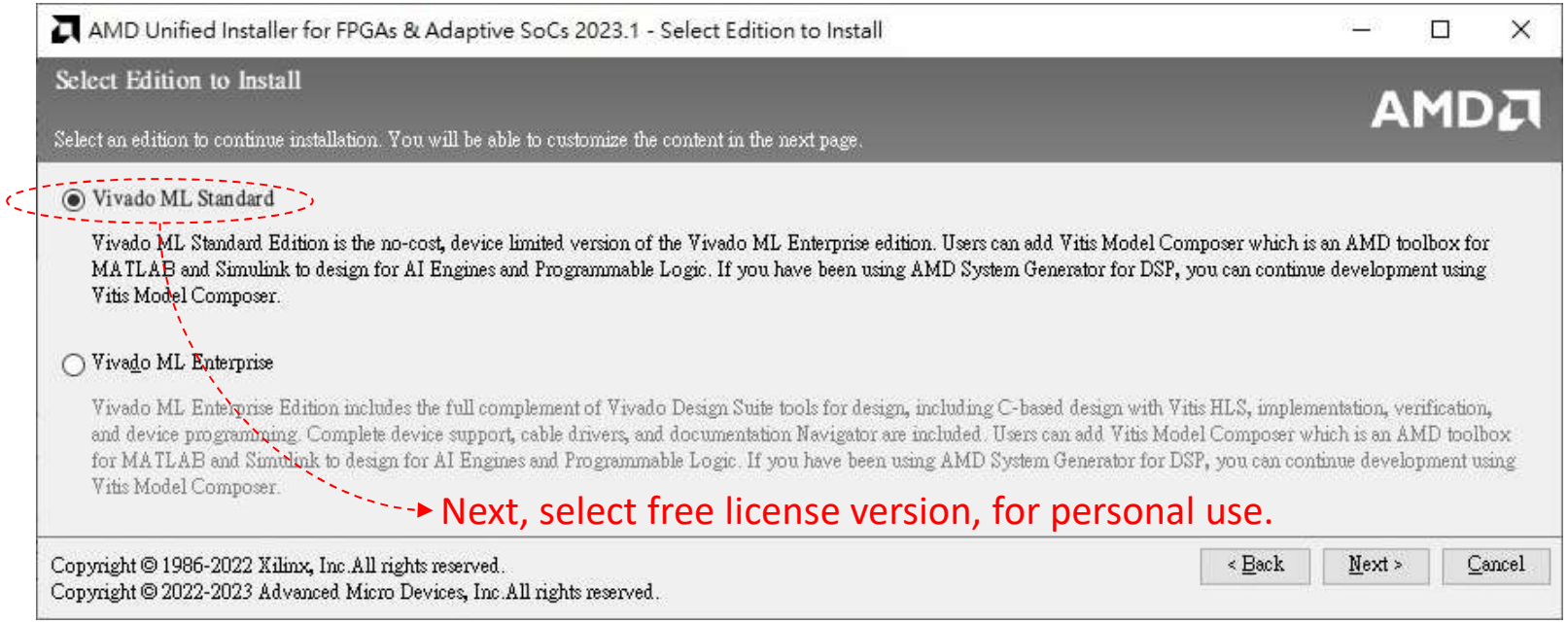
- You can download the installer for Windows or Linux:
  - https://www.xilinx.com/support/download.html
    - Download & install “Vivado ML Edition 2023.1”
  - MacOS is not supported by AMD/Xilinx!

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

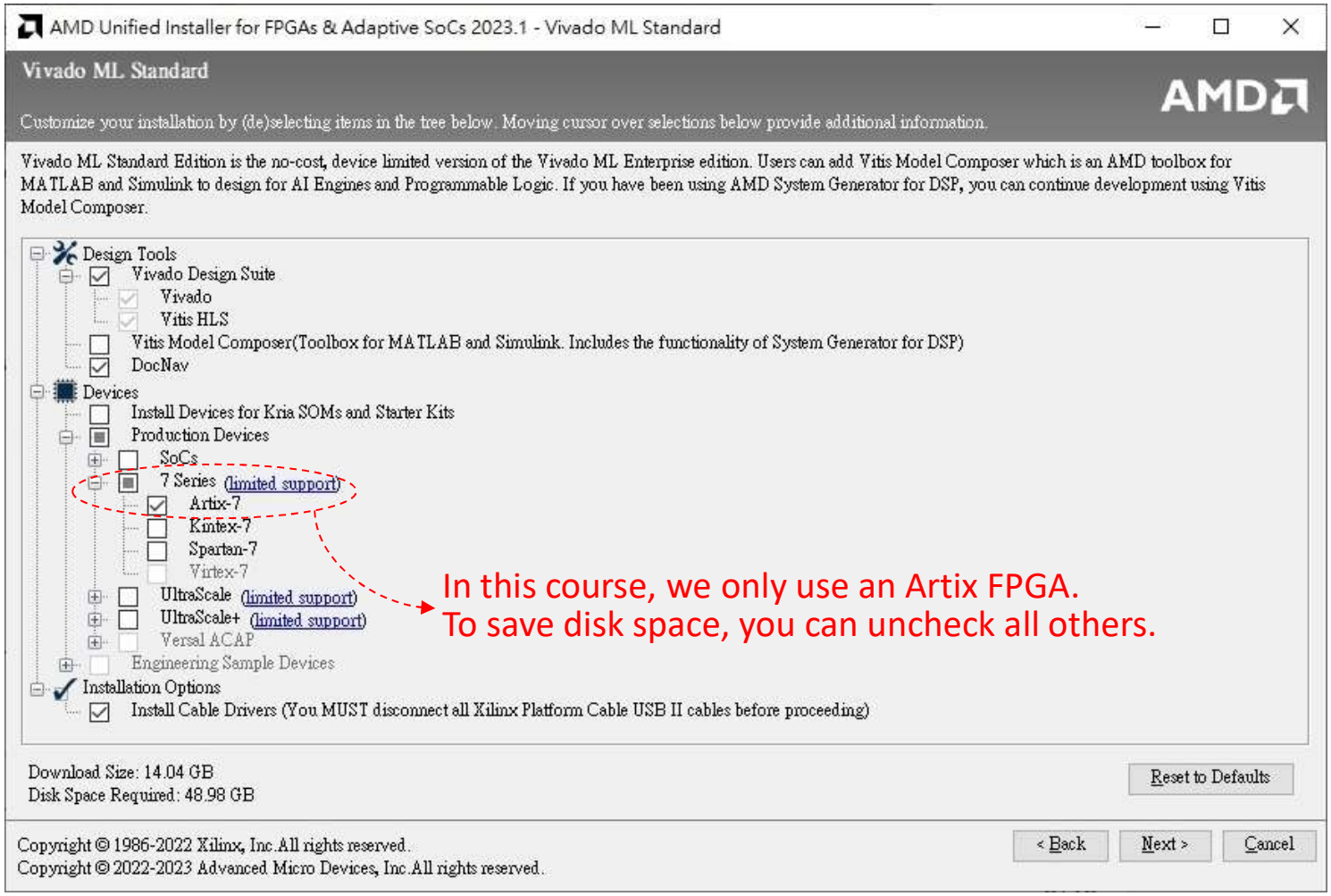
Vivado Installation Guide (1/3)



Vivado Installation Guide (2/3)



Vivado Installation Guide (3/3)



In this course, we only use an Artix FPGA.
To save disk space, you can uncheck all others.

Installation of Arty Board Definitions

- Install the Arty board file:
  - Go to https://github.com/Digilent/vivado-boards, download the directory arty-a7-100/*

  - Make a directory Digilent/ under <INST_DIR>/Vivado/2023.1/data/xhub/boards/XilinxBoardStore/boards/
  - Put arty-a7-100/* under Digilent/

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

Creation of an Aquila SoC Workspace

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

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

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

```
C:\<INST_DIR>\Vivado\2023.1\bin\vivado.bat -mode batch -source build.tcl
```

Or, if you use Linux, under bash prompt:

```
<INST_DIR>/Vivado/2023.1/bin/vivado -mode batch -source build.tcl
```

The generated workspace will be in ./`aquila_mpd/`.

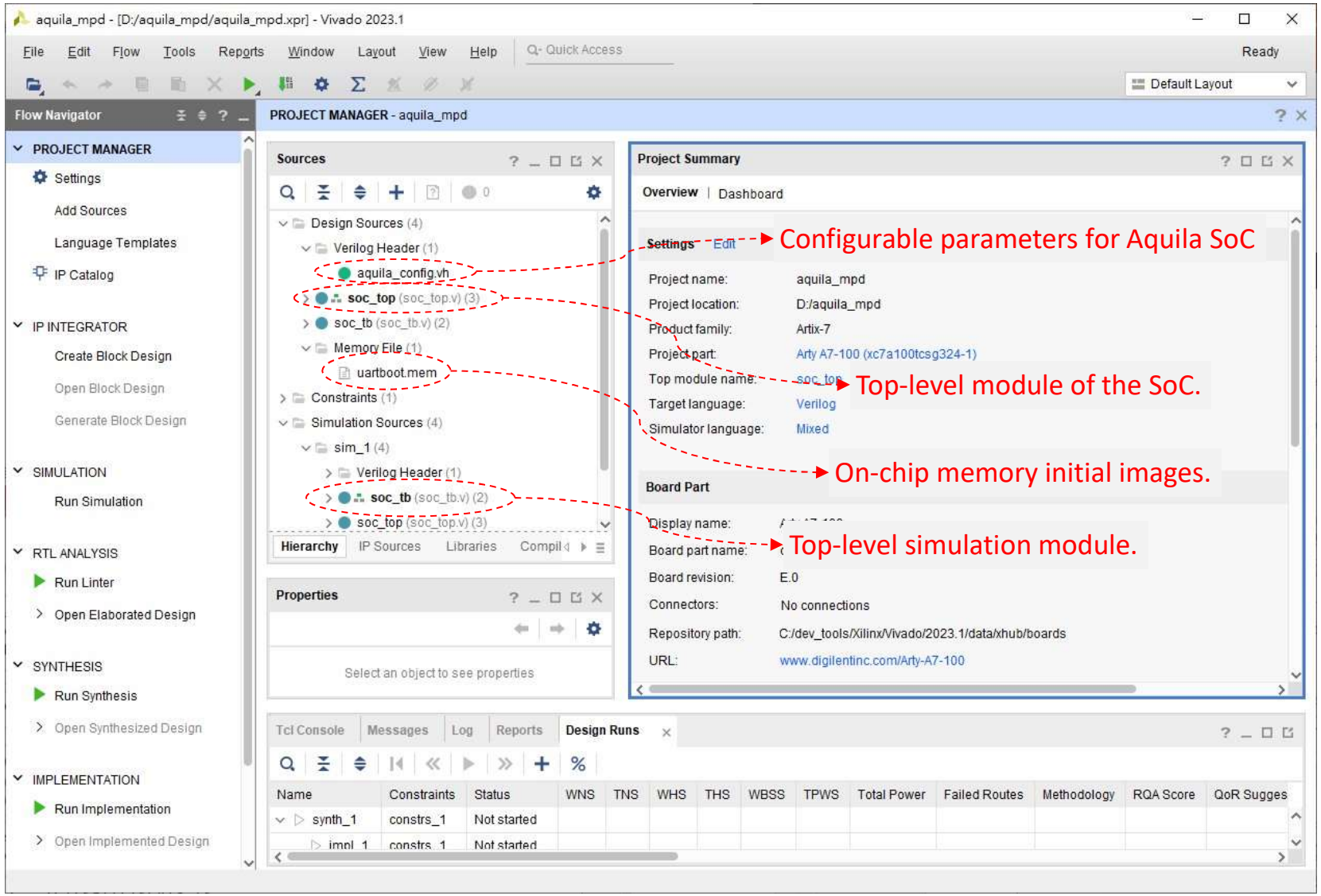
Open the Workspace

- Under Windows, just double-click the file `aquila_mpd.xpr`

- Under Linux bash, you can type the command:

```
<INST_DIR>/Vivado/2023.1/bin/vivado aquila_mpd.xpr
```

Overview of the Aquila Workspace



Installation of the SW Toolchain

- You can build the latest GCC under Linux (or WSL†):
  - First, clone the source of the compiler:

```
$ git clone https://github.com/riscv-collab/riscv-gnu-toolchain.git
```

  - Build the Newlib version with the configuration parameters:

```
$ ./configure --prefix=/opt/riscv --with-arch=rv32ima_zicsr_zifencei --with-abi=ilp32
$ sudo make -j16
```

  - After installation, you should set the shell variables:

```
$ export PATH=$PATH:/opt/riscv/bin
$ export RISCV=/opt/riscv
```

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

Sample Software

- The sample software source tree for HW#0:

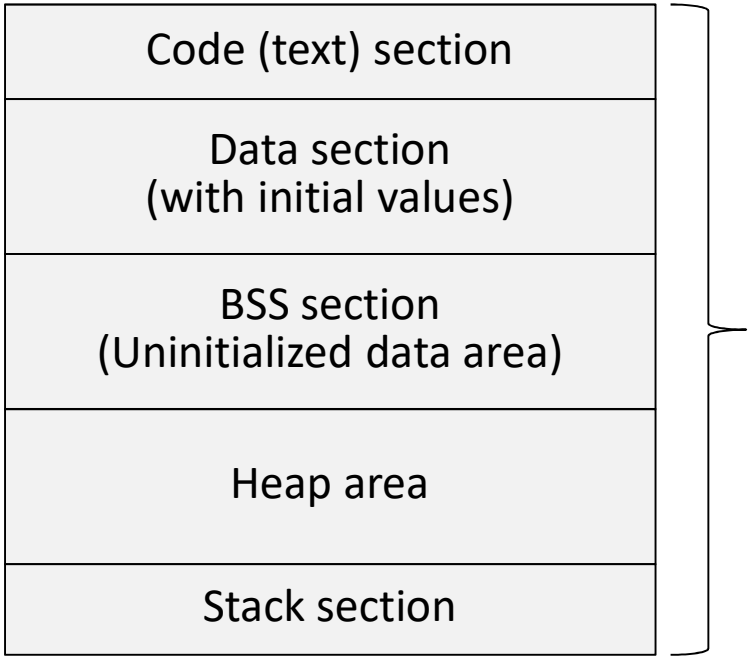
```
aquila_sw -+- uartboot/  → A boot code that loads ELF files
           |
           +- elibc/     → A small "unoptimized" C library
           |
           +- CoreMark/  → The CoreMark benchmark program
           |
           +- Dhrystone/ → The Dhrystone benchmark program
```

- 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 Makefile!

Runtime Memory MAP

- A typical runtime memory map:



These sections do not have to occupy contiguous memory areas.

The POSIX `*.elf` executable file format allows a non-contiguous memory layout.

The system program, loader, will parse and load the program into the correct memory areas.

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

Linker Script Example

- For the initial RAM image, the linker script is as follows:

```
__stack_size = 0x800;
__heap_size  = 0x5000;
__heap_start = __stack_top + __heap_size;

MEMORY
{
    code_ram   (rx!rw) : ORIGIN = 0x00000000, LENGTH = 0x5000
    data_ram   (rw!x)  : ORIGIN = 0x00005000, LENGTH = 0x4000
}

ENTRY(crt0)

SECTIONS
{
    .text :
    {
        libelibc.a(.text)
        *(.text)
    } > code_ram

    .data :
    {
        *(.data)
        *(.bss)
        *(.rodata*)
    } > data_ram

    .stack : ALIGN(0x10)
    {
        . += __stack_size;
        __stack_top = .;
    } > data_ram
}
```

The compiler will read the linker script and generate machine codes accordingly.

Program Binary File Formats

- We use two different program binary file formats:
  - `*.mem` – used for the initialization of the on-chip memory
  - `*.elf` – the standard UNIX Executable and Linkable Format

- To load and run an ELF file, the on-chip memory must be initialized with the `uartboot.mem`.
  - `uartboot` will do the following things
    - Print an prompt in the terminal window
    - Asks you to transfer an `*.elf` program from the host PC through the UART on to the Arty FPGA board for execution
    - Put the ELF file sections into the proper memory areas

Simulation Using Vivado Simulator

- In Aquila workspace, there are two top-level modules:
  - `soc_top.v` – for circuit synthesis
  - `soc_tb.v` – for circuit simulation
- In simulation mode, the UART controller, `uart.v`, reads an ELF file from disk and "simulates" the transfer of the file from the UART input port
  - The ELF file path/name is defined in `aquila_config.h`

Run Behavioral Simulation

aquila_mpd - [D:/aquila_mpd/aquila_mpd.xpr] - Vivado 2023.1

File Edit Flow Tools Reports Window Layout View Help Quick Access Ready

Default Layout

Flow Navigator

PROJECT MANAGER - aquila_mpd

PROJECT MANAGER
- Settings
- Add Sources
- Language Templates
- IP Catalog

IP INTEGRATOR
- Create Block Design
- Generate Block Design

SIMULATION
- Run Simulation

RTL ANALYSIS
- Run Linter
- Open Elaborate

SYNTHESIS
- Run Synthesis
- Open Synthesized Des

Click this to run simulation.

Run Behavioral Simulation
Run Post-Synthesis Functional Simulation
Run Post-Synthesis Timing Simulation
Run Post-Implementation Functional Simulation
Run Post-Implementation Timing Simulation

Sources

Design Sources (4)
- Verilog Header (1)
  - aquila_config.vh
- soc_top (soc_top.v) (3)
- soc_tb (soc_tb.v) (2)
- Memory File (1)
  - uartboot.mem
- Constraints (1)
- Simulation Sources (5)

Hierarchy IP Sources Libraries Compil

Properties

Project Summary | aquila_config.vh | soc_tb.v

D:/aquila_mpd/aquila_mpd.srcs/sources_1/imports/src/core_rtl/aquila_config.vh

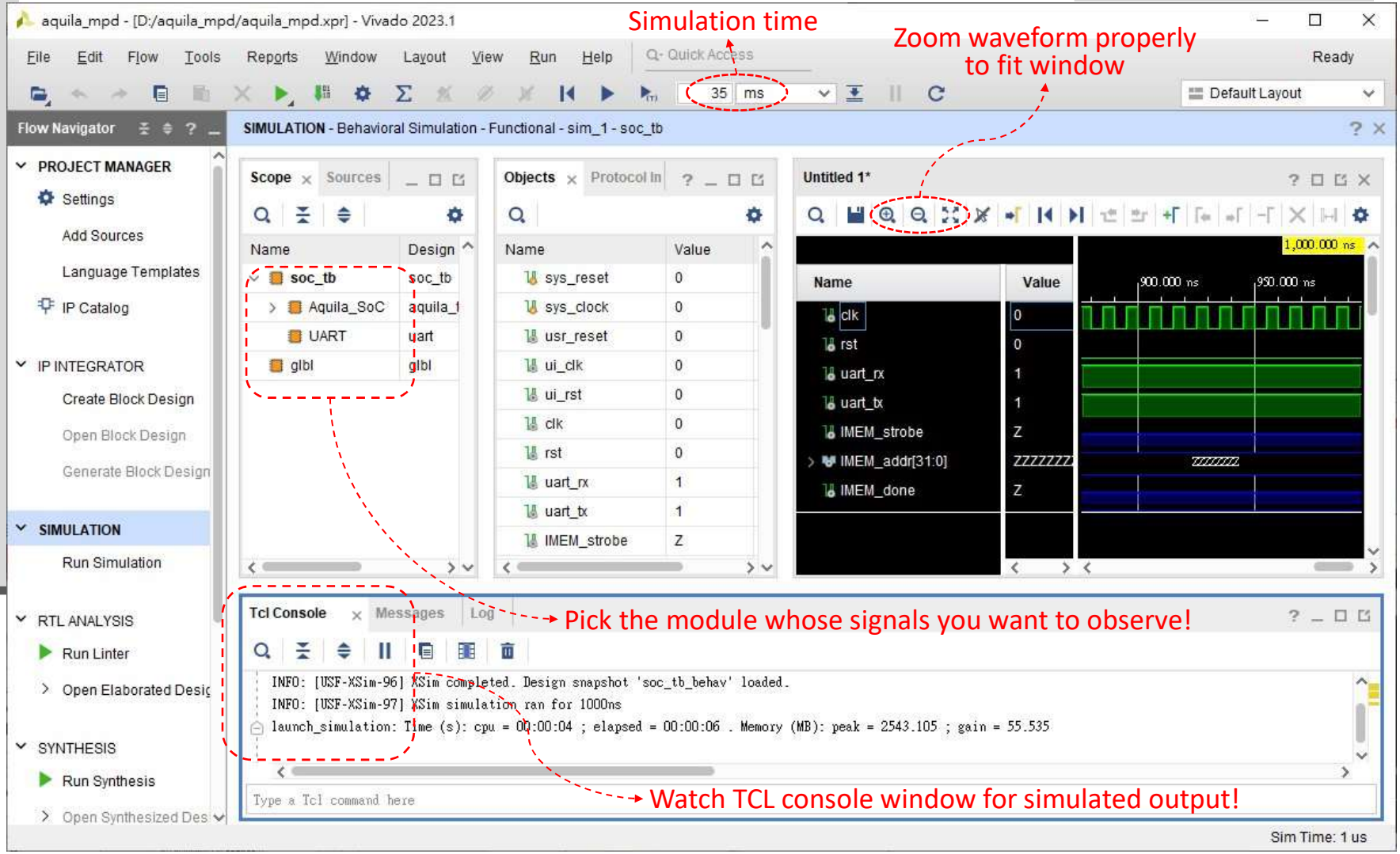
```
82
83  // Branch Prediction
84  `define ENABLE_BRANCH_PREDICTION
85
86  // Atomic Unit
87  `define ENABLE_ATOMIC_UNIT
88
89  // Muldiv Integer multiplier
90  `define ENABLE_FAST_MULTIPLY
91
92  // Fetch
93  `define NOP 32'h00000013
94
95  // SIM_FNAME defines the RISC-V program path of an ELF file for simul
96  `define SIM_FNAME "D:/7dhry.elf"
97
98
```

ELF program to be executed!

Design Runs

			WNS	TNS	WHS	THS	WBSS	TPWS	Total Power	Failed Routes	Methodology	RQA Score	QoR Sug
synth_1	constrs_1	Not started											
impl_1	constrs_1	Not started											

Vivado Simulator Window

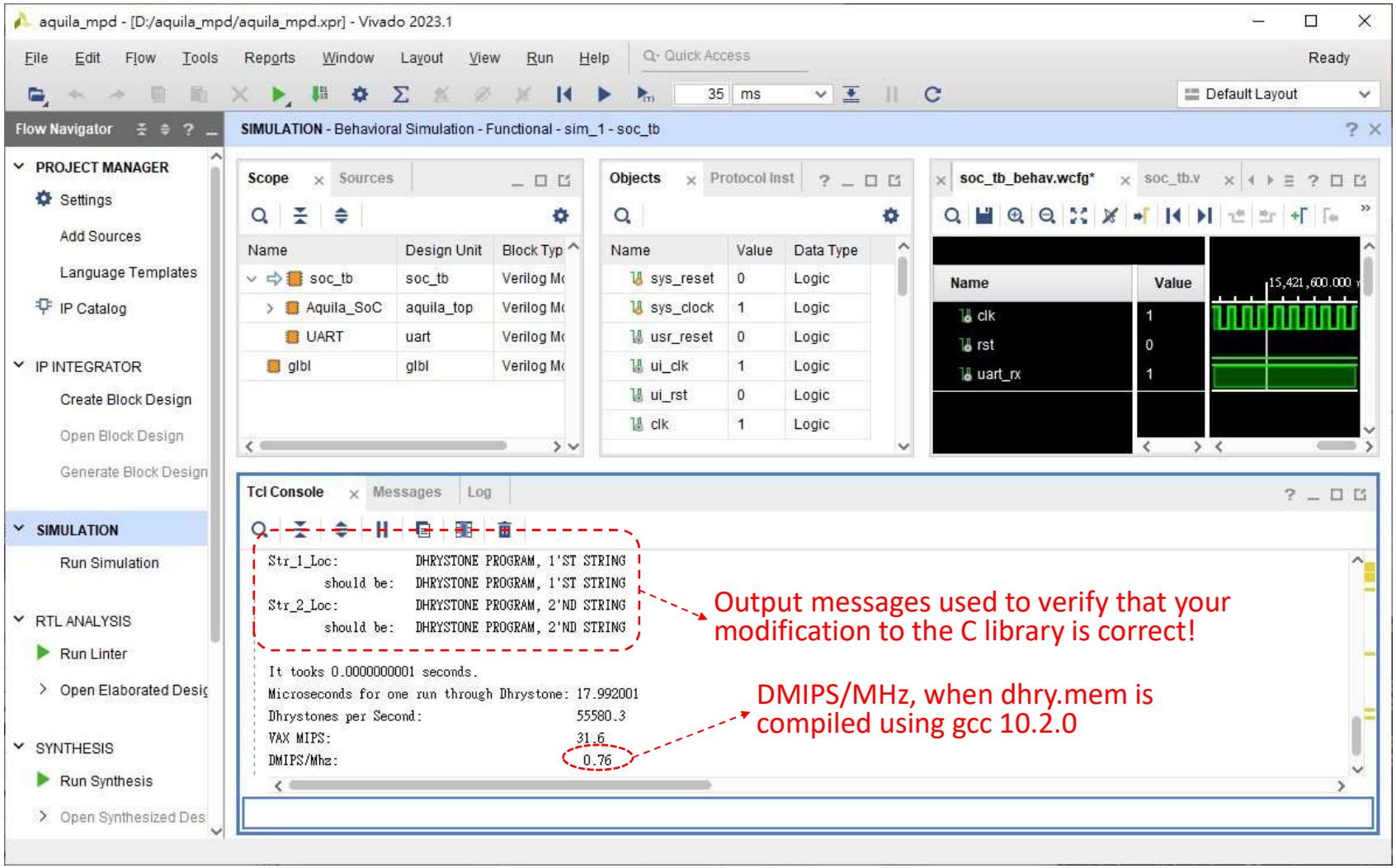


On Simulation of `printf()`

- Note that the testbench and `uart.v` also support the simulation of output from the C library `printf()`
  - At circuit level, `printf()` sends ASCIIs to the `uart` module
  - In simulation mode, the `uart` module will sent the ASCIIs to the “Tcl Console” of Vivado

- There is a trap in `uart.v` such that when the ASCII code `0x03` is printed, the simulation will terminate

Simulated Results



Tracing the Execution of a Function

- One of the reasons that the DMIPS of Aquila is a bit low because the C library (`elibc`) is not optimized!

- For example, you can trace the execution of the `strcpy()` at circuit level, and analyze why the processor cannot execute the function efficiently
  - The `dhry.map` tells you the start address of the function
  - `strcpy()` begins at `0x00002110` in my build (gcc 12.2.0),
  - In the simulator, search for the code address `0x2110`, and start tracing the waveform
    - pay attention to the stall cycles for load/store instruction execution

Cross-Referencing the Assembly

- After making the `dhry.elf` program, there is a text file, `dhry.objdump`, that contains the assembly code of the program
  - This file is helpful for you to analyze the waveform

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

Sample Assembly Code

- The assembly code of `dhry.objdump`：

Aquila execution begins here!

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

Disassembly of section .text:

                                . . . .
00001378 <crt0>:
    1378:         ff010113          add      sp,sp,-16
    137c:         00112623          sw       ra,12(sp)
    1380:         0000a2b7          lui      t0,0xa
    1384:         1422a623          sw       sp,332(t0) # a14c <sp_store>
    1388:         0000a2b7          lui      t0,0xa
    138c:         0f02a103          lw       sp,240(t0) # a0f0 <stack_top>
    1390:         5a9030ef          jal      5138 <main>
    1394:         0000a2b7          lui      t0,0xa
    1398:         14c2a103          lw       sp,332(t0) # a14c <sp_store>
    139c:         00c12083          lw       ra,12(sp)
    13a0:         01010113          add      sp,sp,16
    13a4:         00008067          ret

000013a8 <getchar>:
    13a8:         ff010113          add      sp,sp,-16
    13ac:         00112623          sw       ra,12(sp)

                                . . . .
```

Dhrystone Benchmarks Issues

- There is no perfect benchmarks. For Dhrystone, it's much less than perfect†:
  - Too many fixed-length string operations (`strcpy()` and `strcmp()`)
  - Code/data size too small to test cache performance
  - Dirty compilers that optimize for Dhrystone can achieve extra 50% higher DIMPS numbers
  - Did not take into account architecture features (e.g., RISC, VLIW, SIMD, and superscalar)
  - Code patterns do not reflect modern applications (Big Question: is CPU critical for modern applications?)

† https://www.eembc.org/techlit/datasheets/dhrystone_wp.pdf

Your Homework

- Get familiar with the behavior simulation of a complex HW-SW system

- Rewrite `strcpy()` and `strcmp()`, see if you can increase the DMIPS/MHz performance
  - A useful reference is the *Bit Twiddling Hacks*† from Stanford
  - You can also study other optimized standard C libraries, such as Newlib, to see how others do it
  - Don't forget to use the simulator to analyze and compare the execution of your optimized code against the original code

† https://graphics.stanford.edu/~seander/bithacks.html