

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

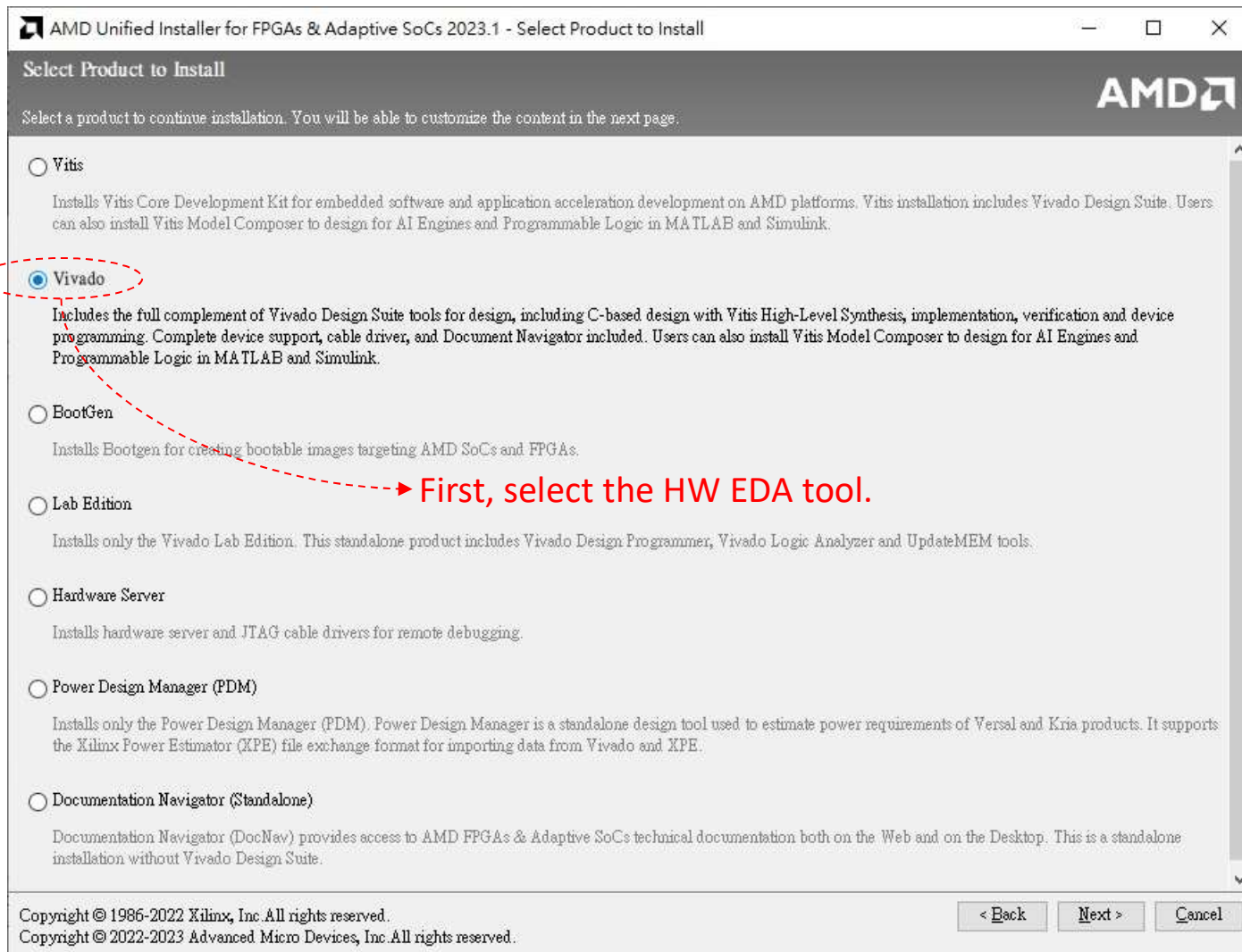
- ❑ Vivado (for HW design only)
 - Support traditional HW design flow using Verilog or VHDL
 - Includes circuit synthesizer, simulator, and embedded logic analyzer

- ❑ 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

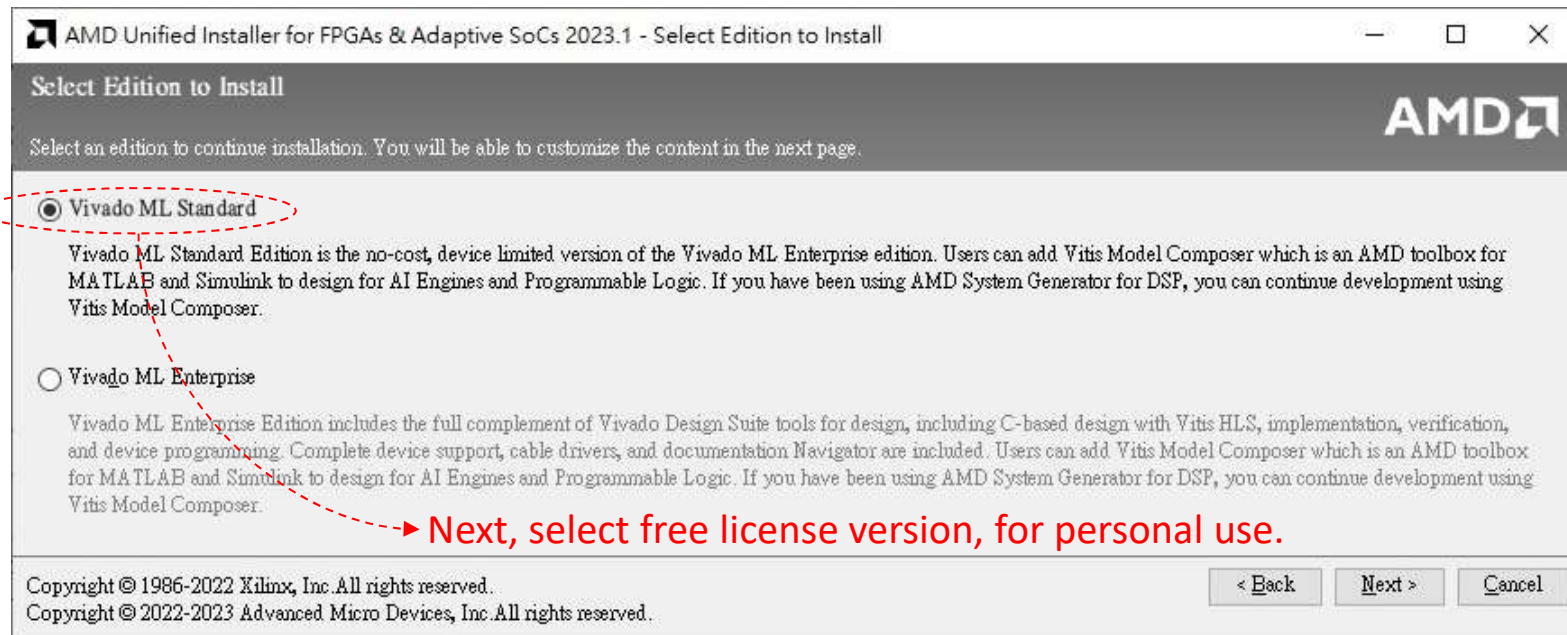
Installation of Vivado Design Suite

- ❑ 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

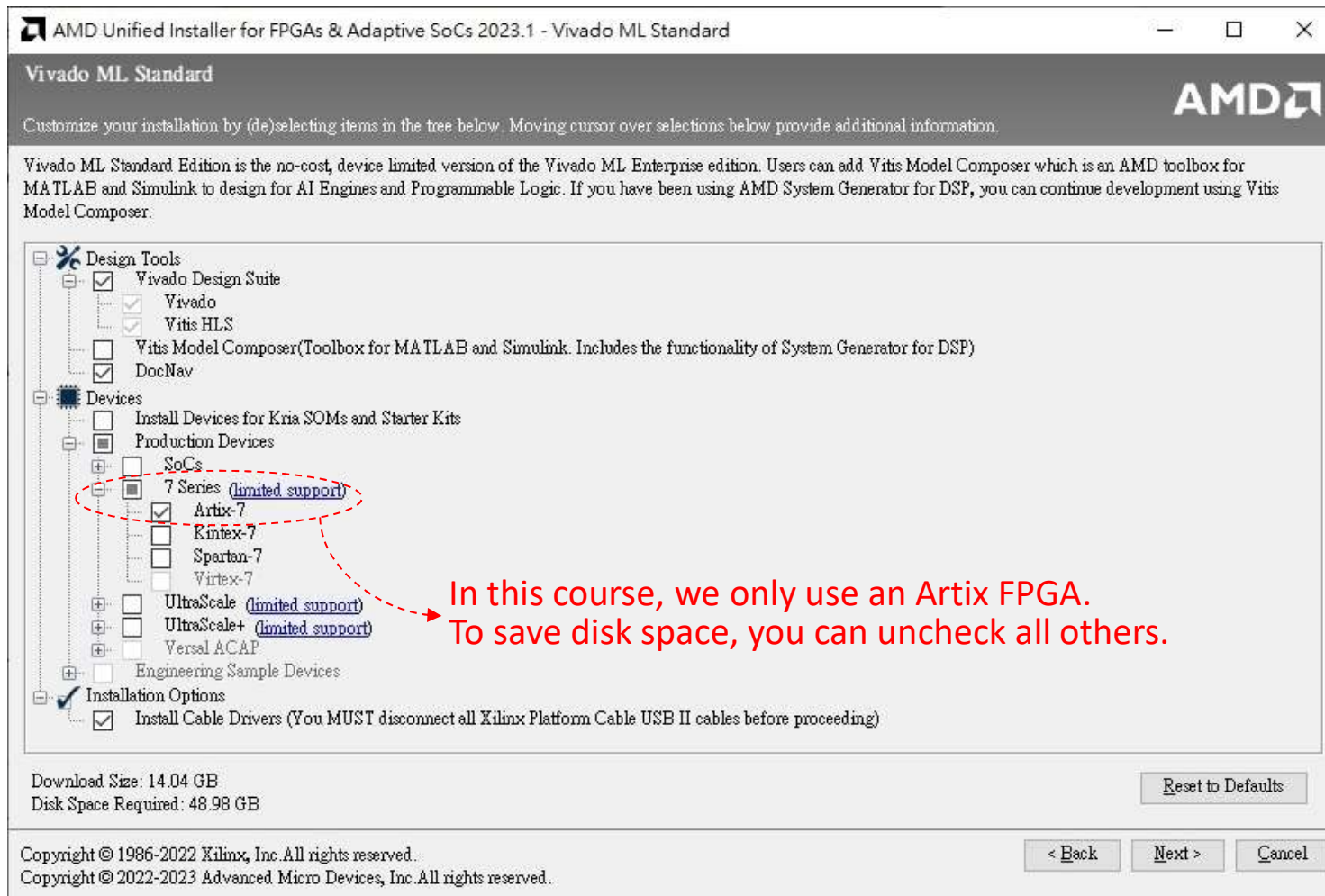
Vivado Installation Guide (1/3)



Vivado Installation Guide (2/3)



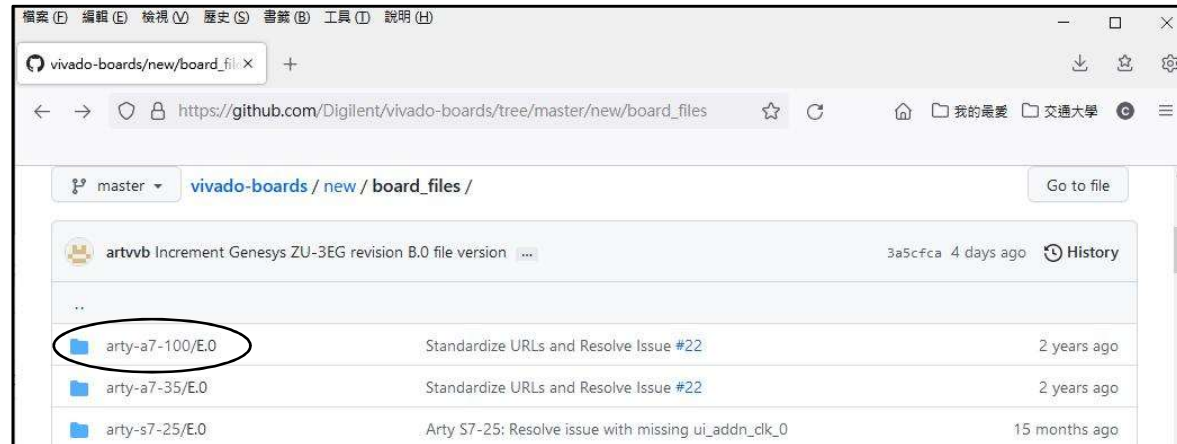
Vivado Installation Guide (3/3)



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

The screenshot displays the Aquila Workspace within the Vivado 2023.1 IDE. The interface is divided into several panels:

- Flow Navigator:** Located on the left, it shows the project hierarchy with sections like PROJECT MANAGER, IP INTEGRATOR, SIMULATION, RTL ANALYSIS, SYNTHESIS, and IMPLEMENTATION.
- PROJECT MANAGER - aquila_mpd:** This panel shows the project's sources and hierarchy. Red dashed circles highlight specific files: `aquila_config.vh`, `soc_top (soc_top.v) (3)`, `uartboot.mem`, and `soc_tb (soc_tb.v) (2)`. Red arrows point from these files to descriptive text on the right.
- Project Summary:** This panel provides an overview of the project settings. Red arrows point from the `soc_top` and `soc_tb` files to their respective entries in the summary.
- Design Runs:** At the bottom, this panel shows the status of various design runs, including synthesis and implementation.

Red annotations highlight key components:

- Configurable parameters for Aquila SoC:** Points to the `aquila_config.vh` file.
- Top-level module of the SoC:** Points to the `soc_top (soc_top.v) (3)` file.
- On-chip memory initial images:** Points to the `uartboot.mem` file.
- Top-level simulation module:** Points to the `soc_tb (soc_tb.v) (2)` file.

The Project Summary panel includes the following information:

- Overview | Dashboard**
- Settings** (Configurable parameters for Aquila SoC)
- Project name: aquila_mpd
- Project location: D:/aquila_mpd
- Product family: Artix-7
- Project part: Arty A7-100 (xc7a100tcsq324-1)
- Top module name: soc_top (Top-level module of the SoC)
- Target language: Verilog
- Simulator language: Mixed
- Board Part**
- Display name: Arty A7-100
- Board part name: soc_top (Top-level simulation module)
- Board revision: E.0
- Connectors: No connections
- Repository path: C:/dev_tools/Xilinx/Vivado/2023.1/data/xhub/boards
- URL: www.digilentinc.com/Arty-A7-100

Name	Constraints	Status	WNS	TNS	WHS	THS	WBSS	TPWS	Total Power	Failed Routes	Methodology	RQA Score	QoR Suggest
synth_1	constrs_1	Not started											
impl_1	constrs_1	Not started											

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:

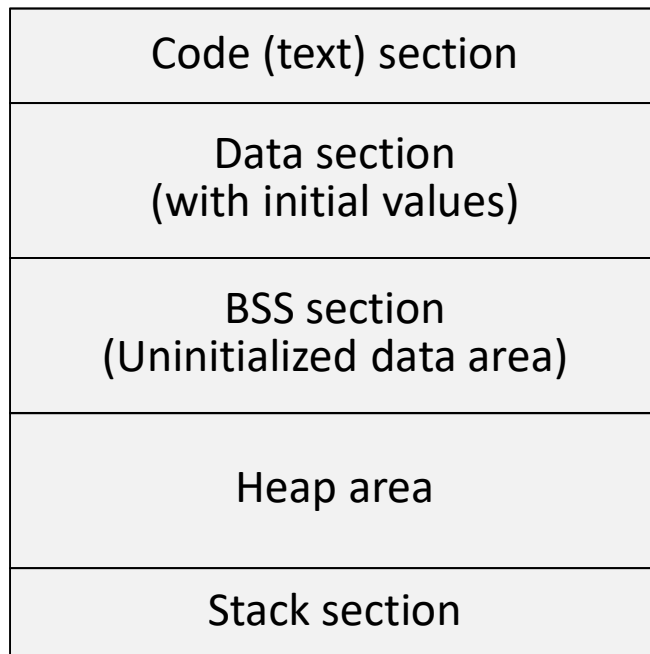
```
aquila_sw +- uartboot/ → A boot code that loads ELF files
           |
           +- elibc/   → A small "unoptimized" C library
           |
           +- CoreMark/ → The CoreMark benchmark program
           |
           +- Dhrystone/ → The Dhrystone benchmark program
```

- ❑ 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 linker 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 :
    {
        libelf.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

The screenshot shows the Vivado 2023.1 IDE interface for a project named 'aquila_mpd'. The 'PROJECT MANAGER' pane on the left shows the project hierarchy with 'aquila_config.vh' selected. The 'Sources' pane in the center shows the project structure. The 'Properties' pane at the bottom shows the 'Run Behavioral Simulation' option selected. The 'Design Runs' pane at the bottom shows the simulation results.

Click this to run simulation.

ELF program to be executed!

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

Name	Constraints	Status	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

The screenshot displays the Vivado Simulator Window for a project named 'aquila_mpd'. The interface includes a menu bar, a toolbar, and several panels. Red dashed boxes and arrows highlight key features:

- Simulation time:** A red dashed box highlights the '35 ms' value in the simulation time display.
- Zoom waveform properly to fit window:** A red dashed box highlights the zoom controls in the waveform window.
- Pick the module whose signals you want to observe!** A red dashed box highlights the 'soc_tb' module in the 'Scope' panel.
- Watch TCL console window for simulated output!** A red dashed box highlights the 'Tcl Console' window.

The 'Scope' panel shows the following table:

Name	Design
soc_tb	soc_tb
> Aquila_SoC	aquila_1
UART	uart
glbl	glbl

The 'Objects' panel shows the following table:

Name	Value
sys_reset	0
sys_clock	0
usr_reset	0
ui_clk	0
ui_rst	0
clk	0
rst	0
uart_rx	1
uart_tx	1
IMEM_strobe	Z

The 'Tcl Console' window shows the following output:

```
INFO: [USF-XSim-96] XSim Completed. Design snapshot 'soc_tb_behav' loaded.  
INFO: [USF-XSim-97] XSim simulation ran for 1000ns  
launch_simulation: Time (s): cpu = 00:00:04 ; elapsed = 00:00:06 . Memory (MB): peak = 2543.105 ; gain = 55.535
```

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 ASCIIIs to the `uart` module
 - In simulation mode, the `uart` module will sent the ASCIIIs 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

The screenshot displays the Vivado 2023.1 interface for a behavioral simulation of a soc_tb. The Project Manager on the left shows the project structure, including the soc_tb design unit. The Simulation tab is active, showing the Scope and Objects windows. The Scope window lists the design units: soc_tb, Aquila_SoC, UART, and gbl. The Objects window lists the variables: sys_reset, sys_clock, usr_reset, ui_clk, ui_rst, and clk. The Tcl Console window shows the output messages for a Dhrystone program, including execution time, Dhrystone score, VAX MIPS, and DMIPS/Mhz. Red dashed boxes and arrows highlight specific output messages and values.

Scope Window:

Name	Design Unit	Block Type
soc_tb	soc_tb	Verilog Module
Aquila_SoC	aquila_top	Verilog Module
UART	uart	Verilog Module
gbl	gbl	Verilog Module

Objects Window:

Name	Value	Data Type
sys_reset	0	Logic
sys_clock	1	Logic
usr_reset	0	Logic
ui_clk	1	Logic
ui_rst	0	Logic
clk	1	Logic

Tcl Console Window:

```
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 0.0000000001 seconds.
Microseconds for one run through Dhrystone: 17.992001
Dhrystones per Second: 55580.3
VAX MIPS: 31.6
DMIPS/Mhz: 0.76
```

Output messages used to verify that your modification to the C library is correct!

DMIPS/MHz, when dhry.mem is compiled using gcc 10.2.0

Tracing the Execution of a Function

- ❑ One of the reasons that the DMIPS of Aquila is a bit low because the C library (`ellibc`) 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 `dhrv.elf` program, there is a text file, `dhrv.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 :

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

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

Aquila execution
begins here!



```
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>