

HW#0 Simulation of a HW-SW Platform



Chun-Jen Tsai
National Yang Ming Chiao Tung University
09/16/2022

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 can optimize the standard library to increase speed
- ❑ This homework is just for practice, there is no deadline

Target Technology of the Aquila SoC

- ❑ The RTL model of the Aquila core is written in Verilog
 - 30 files, 10,043 lines of code
 - verified using two FGPAs from AMD/Xilinx: Kintex XC7K325T and Artix XC7A100T

- ❑ To develop a HW-SW system using Aquila SoC, you must install Xilinx Vivado and RISC-V GCC

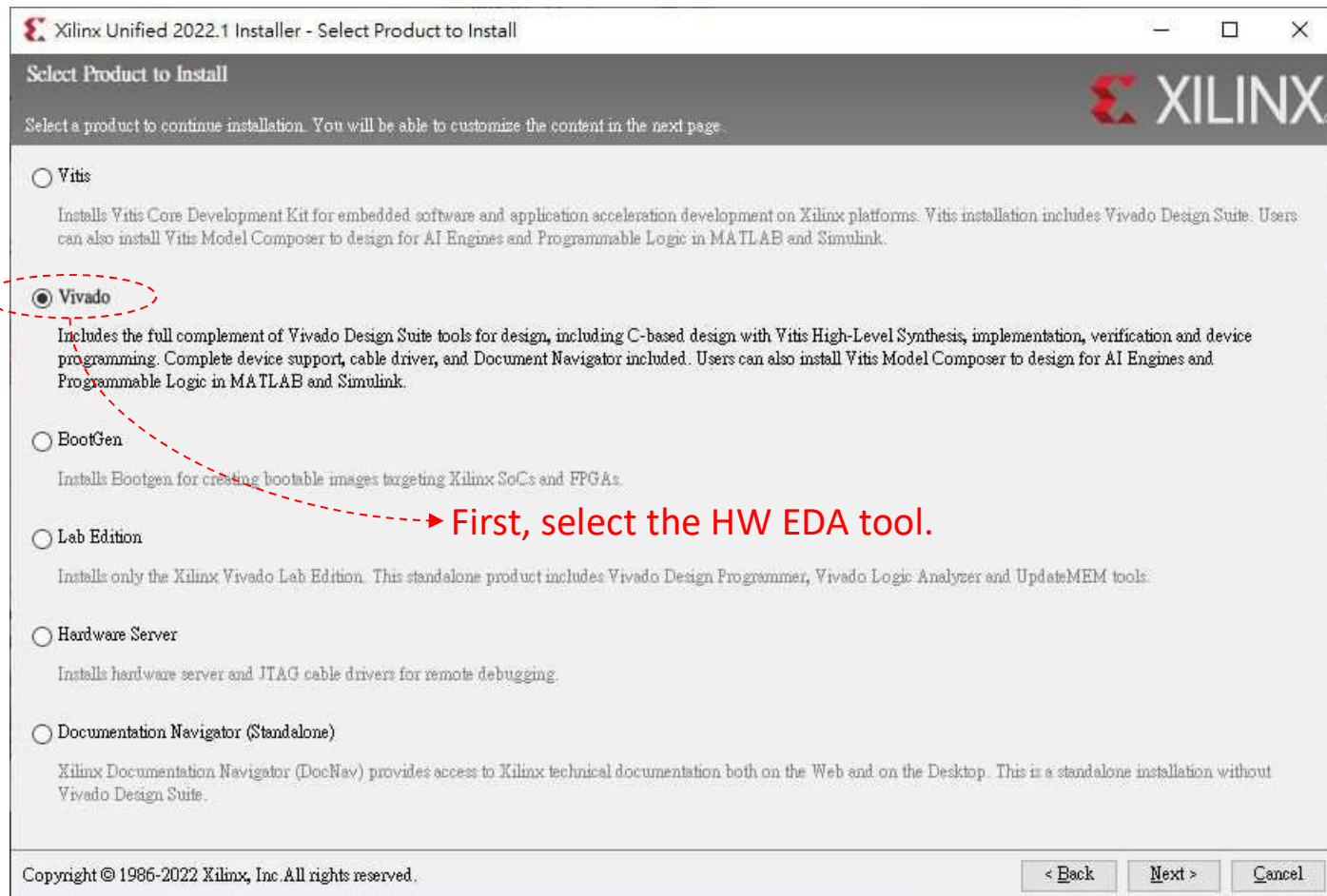
AMD/Xilinx EDA Tools

- ❑ Xilinx has EDA tools for SW-HW system design
 - 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
 - Vivado (for HW design only)
 - HW design using Verilog or VHDL
- ❑ In this course, we use Vivado for HW design, and command-line GCC for SW design

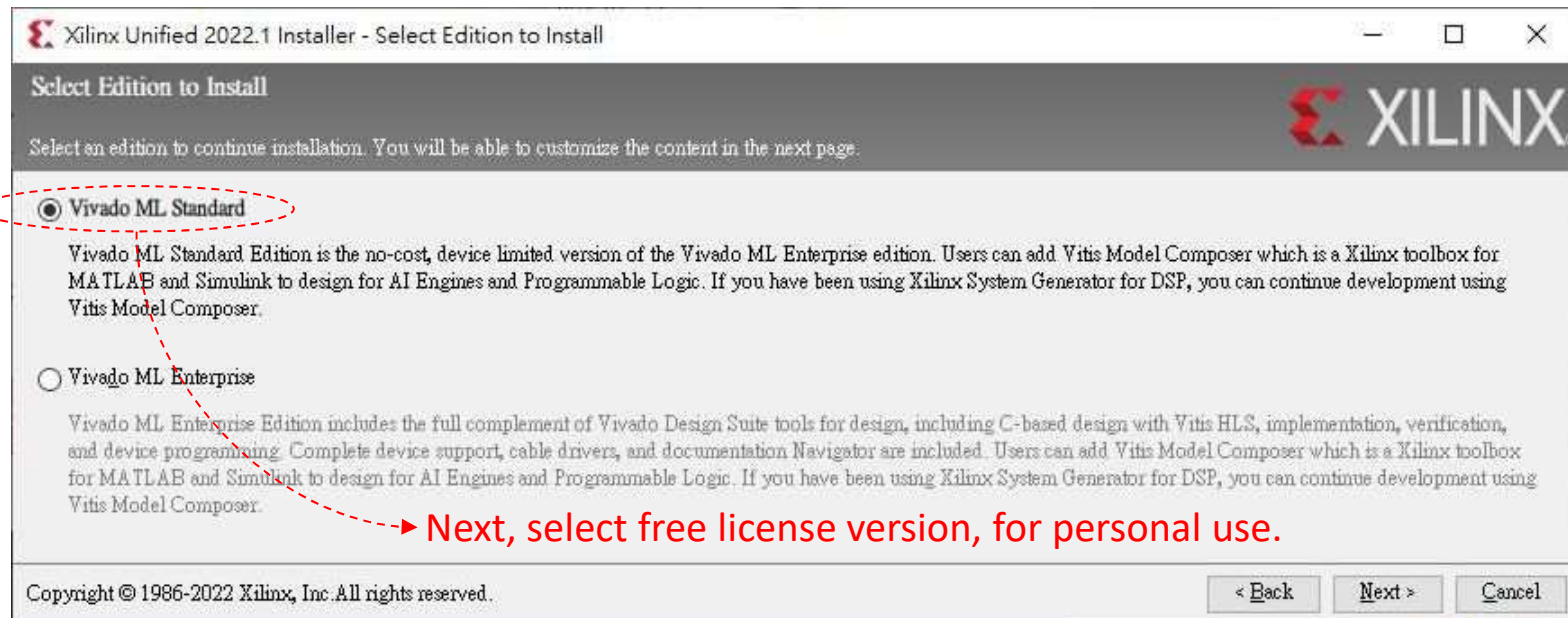
Install Your Vivado Design Suite

- ❑ You can download the installer for Windows or Linux:
 - <https://www.xilinx.com/support/download.html>
 - The website calls it “Vivado **ML** 2022.1”
 - MacOS is not supported by AMD/Xilinx!
- ❑ The installation requires at least 120+ 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

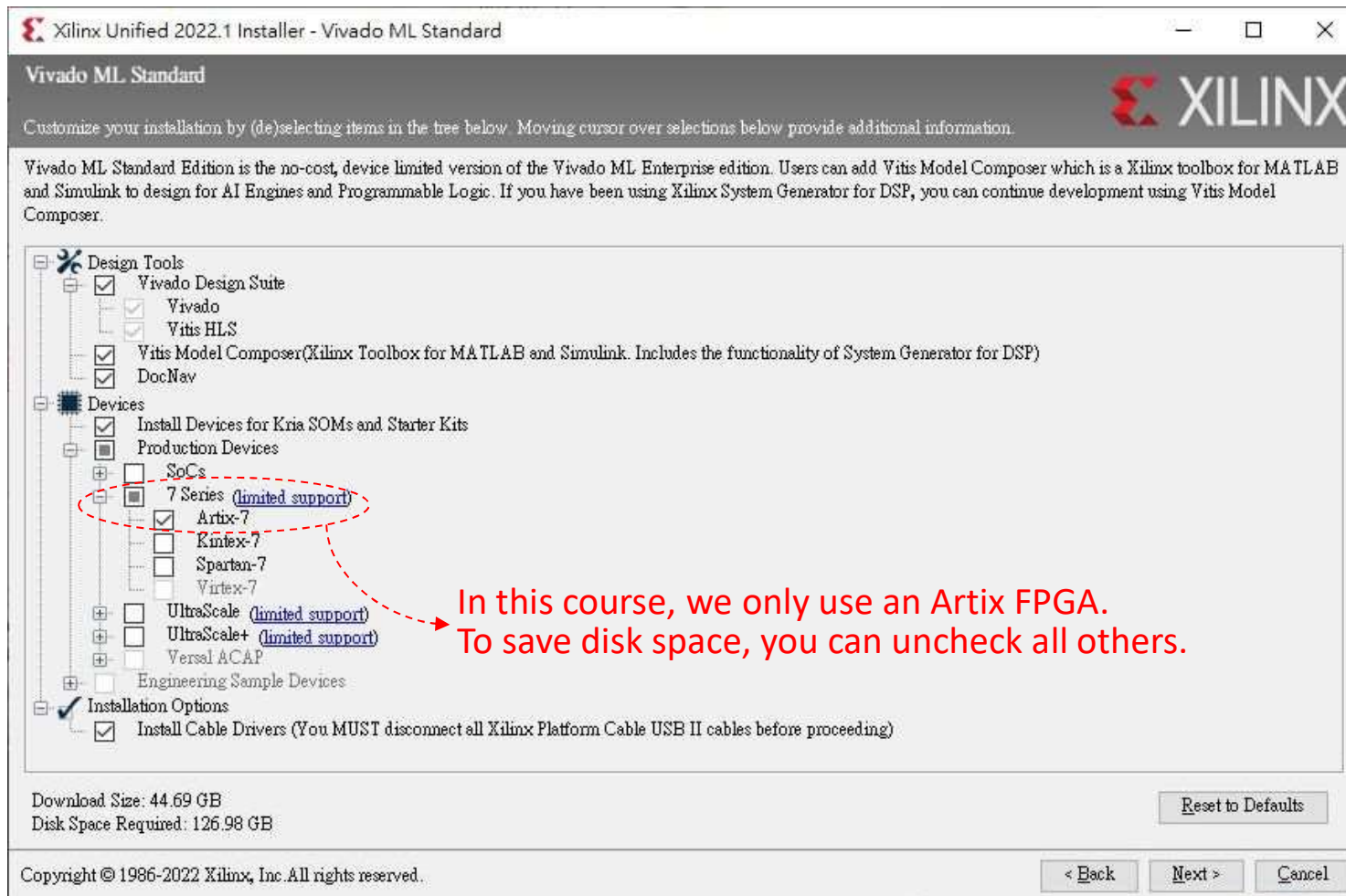
Vivado Installation Guide (1/3)



Vivado Installation Guide (2/3)



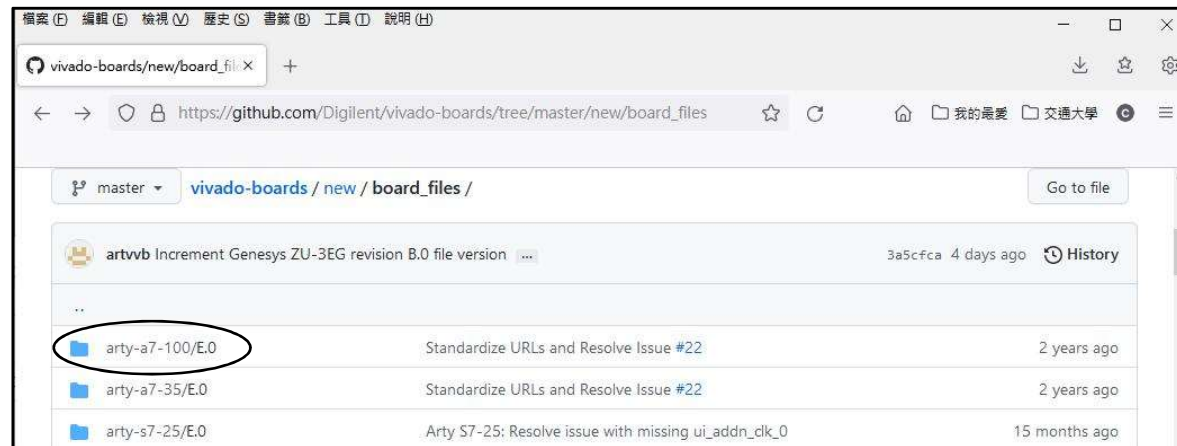
Vivado Installation Guide (3/3)



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/2022.1/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\2022.1\bin\vivado.bat -mode batch -source build.tcl
```

Or, if you use Linux, under bash prompt:

```
<INST_DIR>/Vivado/2022.1/bin/vivado -mode batch -source build.tcl
```

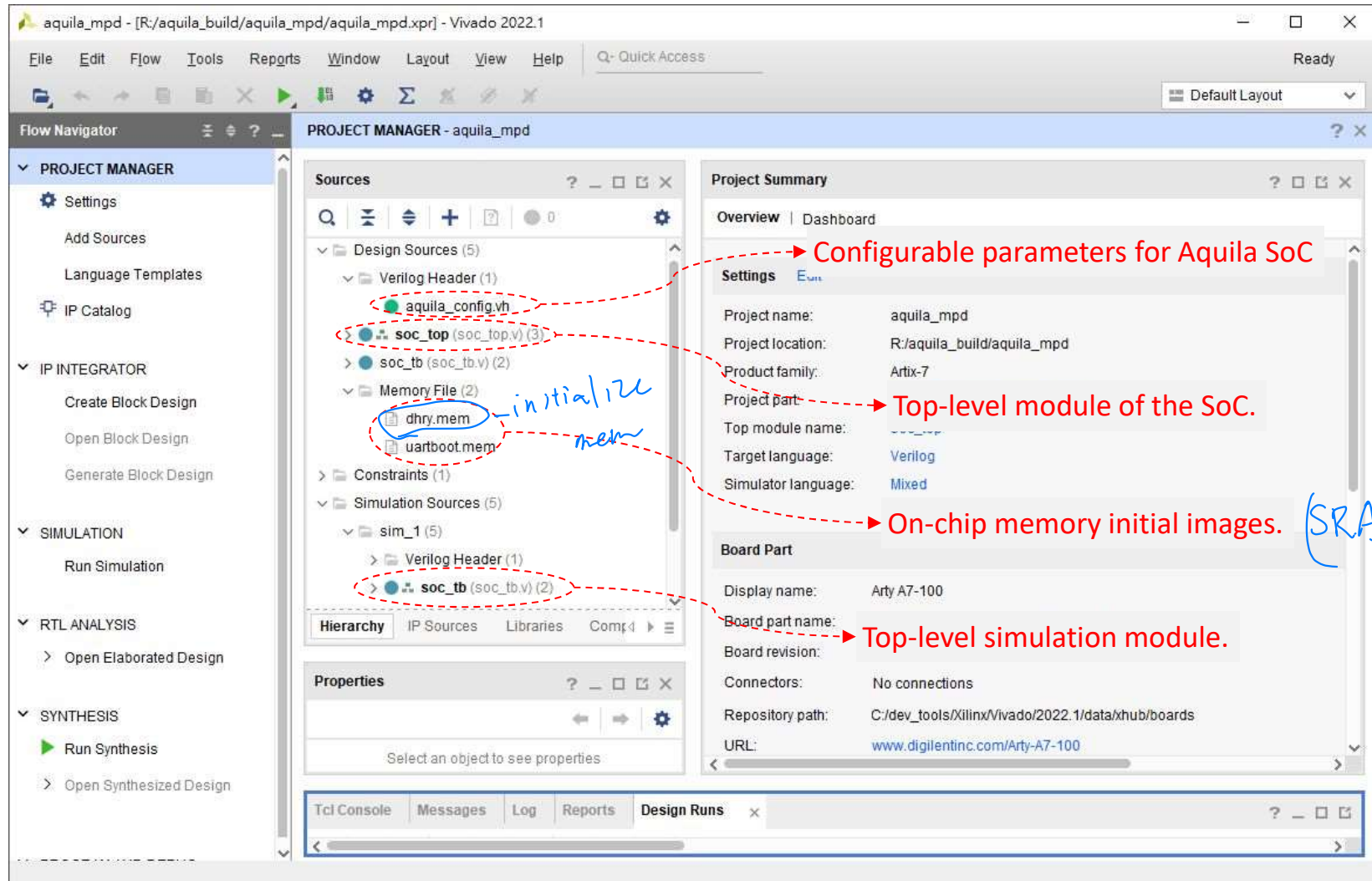
The generated workspace will be in `./aquila_mpd/`.

Open the Workspace

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

```
<INST_DIR>/Vivado/2022.1/bin/vivado aquila_mpd.xpr
```

Overview of the Aquila Workspace



Installation of the SW Toolchain

- ❑ RISC-V GCC 32-bit are used for software development
 - Generic ELF/Newlib version should be used
 - You can download RISC-V gcc v10.2 for Linux[†] from the link:
https://www.cs.nctu.edu.tw/~cjtsai/download/riscv32_gcc.tgz
Untar it under `/opt` of your Linux and add `/opt/riscv/bin` to your command path.

- ❑ You can also build the latest GCC version by yourself:
 - Follow the instruction at:
<https://github.com/riscv/riscv-gnu-toolchain>
 - Note: the configuration parameters you should use are:
`--with-arch=rv32ima --with-abi=ilp32`

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

Sample Software

- ❑ The sample software source tree for HW#0:

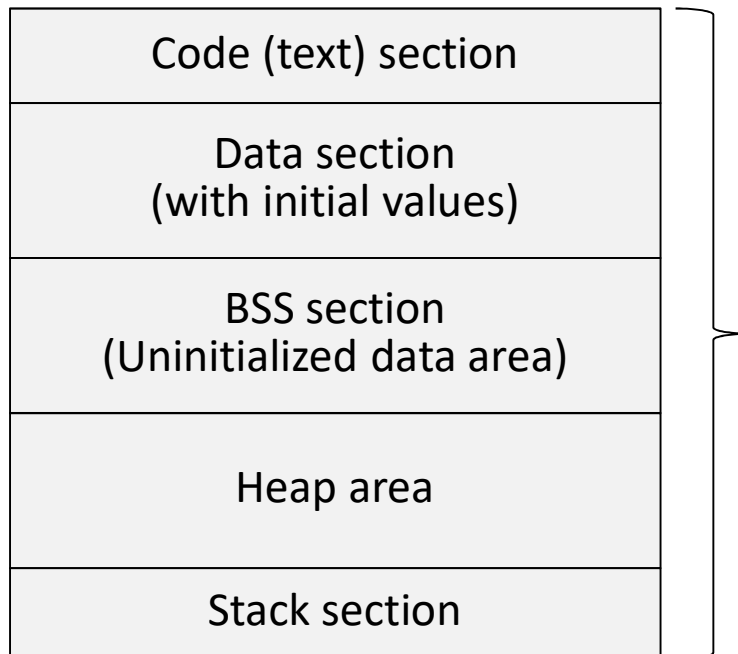
```
aquila_sw +- uartboot/  → A boot code that loads ELF files
           |
           +- elibc/      → A small "unoptimized" C library
           |
           +- Dhrystone/  → The ELF version of Dhrystone
           |
           +- dhryboot/   → The boot code of the Dhrystone
```

- ❑ 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.

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

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

Linker Script Example

- For 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
}
```

c run time (setting up environment)

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

Program Binary File Formats

- ❑ In this course, we used three different program binary file format:
 - *.mem – used for the initialization of 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` uses the part of the TCM to load the ELF file, so, the file cannot be larger than 64KB.
 - If DRAM is enabled, you can recompile `uartboot.mem` to use the DRAM as the load buffer.

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

- ❑ The `uartboot.mem` RAM image contains a boot loader of the Aquila SoC
 - It uses the UART device to load an executable for execution
 - We do not support “simulated” loading of an executable in the `uart.v` module

- ❑ For simulation, you must initialize the RAM with a program image, such as the `dhry.mem`

Selecting The TCM Initial Image

- ❑ The TCM is initialized in `sram_dp.v`:

```
module sram_dp
#(parameter DATA_WIDTH = 32, N_ENTRIES = 1024,
  ADDRW = $clog2(N_ENTRIES))
(
    input                                clk1_i,
    input                                en1_i,
    input                                we1_i,
    . . .

    output reg [DATA_WIDTH-1 : 0] data2_o,
    output reg                                ready2_o
);

reg [DATA_WIDTH-1 : 0] RAM [N_ENTRIES-1 : 0];

initial
begin
    $readmemh("uartboot.mem", RAM);
end
```

→ Change this file to dhry.mem!

Run Behavioral Simulation

The screenshot displays the Vivado 2022.1 IDE interface for a project named 'aquila_mpd'. The 'Flow Navigator' on the left shows the project hierarchy with 'Run Simulation' circled in red. A red arrow points from this button to a text box that says 'Click this to run simulation.' The 'Sources' window shows the project structure, including 'sim_1' and 'soc_tb'. The 'Project Summary' window shows the source file 'sram_dp.v' with the following code:

```
73 input [DATA_WIDTH-1 : 0] data2_i,  
74 output reg [DATA_WIDTH-1 : 0] data2_o,  
75 output reg ready2_o  
76 );  
77  
78 reg [DATA_WIDTH-1 : 0] RAM [N_ENTRIES-1 : 0];  
79  
80 initial  
81 begin  
82 $readmemh("dhry.mem", RAM);  
83 end  
84  
85 // -----  
86 // Read operation on port #1  
87 // -----  
88 always@(posedge clk1_i)  
89 begin
```

The 'Design Runs' table at the bottom shows the status of the simulation runs:

Name	Constraints	Status	WNS	TNS	WHS	THS	WBSS	TPWS	Total Power	Failed Routes	Methodology	RQA Score	QoR S
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 around the '10 us' value in the simulation controls.
- Zoom waveform properly to fit window:** A red dashed box around the zoom controls in the waveform viewer.
- Pick the module whose signals you want to observe!** A red dashed box around the 'Aquila_SoC' module in the Scope panel.
- Watch TCL console window for simulated output!** A red dashed box around the TCL Console panel.

The Scope panel shows a list of modules and their associated Verilog files. The Objects panel displays a list of signals and their current values. The waveform viewer shows a timing diagram for the selected signals, with a time scale of 1,000.000 ns.

TCL Console 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 = 00:00:03 ; elapsed = 00:00:12 . Memory (MB): peak = 1564.395 ; gain = 0.000
```

On Simulation of `printf()`

- ❑ Note that the testbench and `uart.v` support the simulation of the C library `printf()` function
 - At circuit level, `printf()` sends ASCII's to the `uart` module
 - In simulation mode, the `uart` module will send the ASCII's 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 2022.1 interface during a behavioral simulation. The top menu bar includes File, Edit, Flow, Tools, Reports, Window, Layout, View, Run, and Help. The toolbar shows various simulation controls, including a play button and a time scale set to 10 us.

The left sidebar contains the Flow Navigator, which is currently set to the SIMULATION tab. The main workspace is divided into several panels:

- Scope:** A table listing the design hierarchy. The selected scope is `soc_tb`, which contains `Aquila_SoC` (design unit `aquila_top`). Below it are `RISCV_CORE0` (design unit `core_top`), `TCM` (design unit `sram_dp`), and `CLINT` (design unit `clint`).
- Objects:** A table listing the simulation objects and their values.

Name	Value
ui_clk	1
ui_rst	0
clk	1
rst	0
uart_rx	1
- Source Editor:** Displays the Verilog code for `soc_tb.v`. The code includes a `$display` statement and a `$finish` statement, which are highlighted in yellow.
- Tcl Console:** Shows the simulation results. The output indicates that the simulation took 0.0 seconds and that the program exited with a status code of 0. The simulation finished successfully.

A red dashed circle highlights the `DMIPS/Mhz` value of 0.77 in the Tcl Console output. A red arrow points from this value to a red text annotation: "DMIPS/MHz, when dhry.mem is compiled using gcc 10.2.0".

The bottom status bar shows the simulation time as 3982716 ns and the Verilog source file.

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 `*.objdump` tells you the start address of the function
 - `strcpy()` begins at `0x00001f40` in my build (gcc 10.2.0)
 - Pay attention to the **stall cycles** for program execution

Cross-Referencing the Assembly

- ❑ After you make the RAM 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 boot ROM file :

Aquila execution
begins here!

```
dhry.elf:      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>
```

```
. . . .
```

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 (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 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>