Lab 0: Setting up ESP Environment

Submission Due Dates:
2022/4/19 23:59

Objective

Build the system-on-chip environment and get familiar with its basic operations. The environment will be used later in our midterm and final projects.

Introduction

1 Embedded System Platform (ESP)

Reference: https://www.esp.cs.columbia.edu/

ESP is an open-source research platform for the heterogeneous system-on-chip design that combines a scalable tile-based architecture and a flexible system-level design methodology. ESP provides RTL design flow to integrate the accelerator into a complete system that includes the 64-bit RISC-V Ariane CPU, main memory (DRAM), and auxiliary components (Ethernet port, UART port, etc.). These components are connected by an AXI-based (Advanced eXtensible Interface) network-on-chip (NoC).

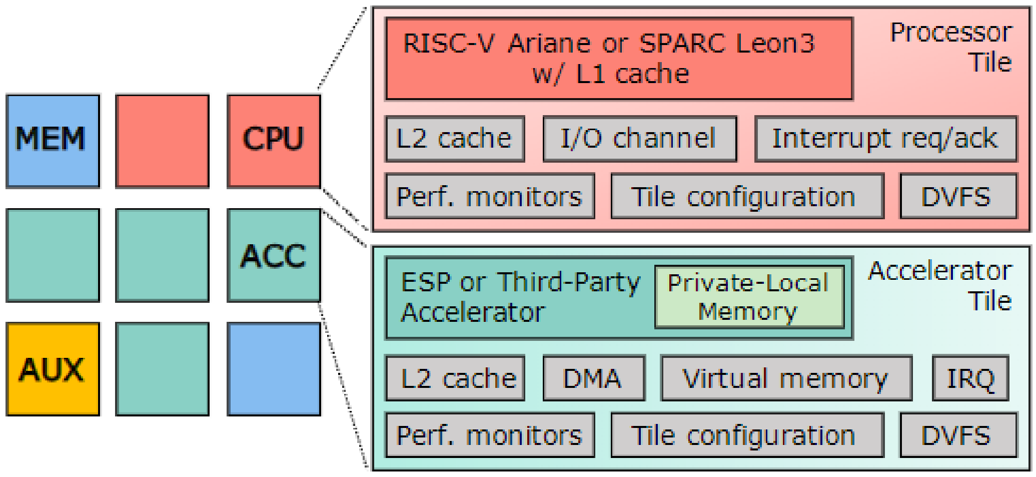


Figure referenced from: https://www.esp.cs.columbia.edu

This system allows us to control our accelerator by the bare-metal application (a standalone program without OS) running in RISC-V CPU. For example, we can write a C program to initialize the main memory, send the start signal to launch our accelerator computation, and verify the result that the accelerator stored in the main memory.

The midterm project requires you to embed the LeNet accelerator in Homework 3 as an Accelerator Tile of ESP and simulate the entire system through ModelSim, an RTL simulator. Specifically, you will design a DMA (Direct Memory Access) interface to communicate with the AXI-based DMA controller, enabling your accelerator in ESP to access the data in Memory Tile (DRAM).

Action Items

1 CAD tool setup script

Save below commands as **setup.sh**, **remember to execute it every time you log into a CAD workstation**. The first seven commands are for setting up the CAD tool environment, and the last eight commands are for setting up the environment variables of the ESP toolchain.

Note: We also provide setup.sh on EECLASS.

```
source setup.sh


 ---- Add following scripts to setup.sh---
source /tools/linux/xilinx/Vivado/2019.2/settings64.csh
source /usr/cad/mentor/CIC/modelsim.cshrc
source /usr/cad/synopsys/CIC/verdi.cshrc
source /usr/cad/synopsys/CIC/synthesis.cshrc
source /usr/cad/synopsys/CIC/spyglass.cshrc
setenv LC_ALL "en_US.UTF-8"
setenv PATH ${PATH}:"/usr/cad/mentor/modelsim/2020.4/modeltech/bin"
setenv AMS_MODEL_TECH "/usr/cad/mentor/modelsim/2020.4/modeltech"
setenv RISCV "/users/student/mr110/CS5120/ESP_toolchain/riscv"
setenv PATH ${PATH}:"/users/student/mr110/CS5120/ESP_toolchain/riscv/bin"
setenv PATH ${PATH}:"/users/student/mr110/CS5120/ESP_toolchain/riscv32imc/bin"
setenv PATH ${PATH}:"/users/student/mr110/CS5120/ESP_toolchain/leon/bin"
setenv PATH ${PATH}:"/users/student/mr110/CS5120/ESP_toolchain/leon/mklinuximg"
setenv  PATH  ${PATH}:"/users/student/mr110/CS5120/ESP_toolchain/leon/sparc-elf/bin"
```

2 Clone the ESP repository

It may take about 20 minutes to download. Stay calm and have a tea break.

```
git  clone  --recursive  https://github.com/sld-columbia/esp.git
```

3 Change branch

Change to the **rtl-flow** branch for the RTL design flow in ESP

```
cd esp
git checkout rtl-flow
git submodule update
```

4 Install python module

Since CAD workstation doesn’t install Python module **Pmw**, you should install it by yourself.

```
pip3 install Pmw --user
```

5 Create a new RTL accelerator skeleton

ESP provides an interactive script that generates a skeleton of the accelerator, its software test applications, and the accelerator device driver. The accelerator skeleton is simply an empty Verilog top module of the accelerator with an interface to the ESP system. You can modify the RTL code in the skeleton as long as the interface remains the same. Currently, ESP supports Verilog, System Verilog, and VHDL.

For more details, please refer to https://www.esp.cs.columbia.edu/docs/rtl_acc/rtl_acc-guide/ and https://www.esp.cs.columbia.edu/docs/singlecore/singlecore-guide/

```
 cd <esp>
 ./tools/accgen/accgen.sh
=== Initializing ESP accelerator template ===
   * Enter accelerator name [dummy]: example
   * Select design flow (Stratus HLS, Vivado HLS, hls4ml, RTL) [S]: R
   * Enter ESP path [/home/davide/Repos/esp/esp-rtlflow]: press ENTER
   * Enter unique accelerator id as three hex digits [04A]: 075
   * Enter accelerator registers
     - register 0 name [size]: reg1
     - register 0 default value [1]: 8
     - register 0 max value [8]: 8
     - register 1 name []: reg2
     - register 1 default value [1]: 8
     - register 1 max value [8]: 8
     - register 2 name []: reg3
     - register 2 default value [1]: 8
     - register 2 max value [8]: 8
     - register 3 name []: press ENTER
   * Configure PLM size and create skeleton for load and store:
     - Enter data bit-width (8, 16, 32, 64) [32]: press ENTER
     - Enter input data size in terms of configuration registers (e.g. 2 * reg2}) [reg2]: press ENTER
       data_in_size_max = 8
     - Enter output data size in terms of configuration registers (e.g. 2 * reg2) [reg2]: press ENTER
       data_out_size_max = 8
     - Enter an integer chunking factor (use 1 if you want PLM size equal to data size) [1]: press ENTER
       Input PLM has 8 32-bits words
       Output PLM has 8 32-bits words
     - Enter the number of input data to be processed in batch [1]: press ENTER
       batching_factor_max = 1
     - Is output stored in place? [N]: press ENTER
=== Generated accelerator skeleton for example ===
```

6 Check the design skeleton

Step 5 generates an empty accelerator skeleton, located at path <esp>/accelerators/rtl/example_rtl/hw. In addition, the accelerator's device driver, bare-metal application, and user-space Linux application are generated at the path <esp>/accelerators/rtl/example_rtl/sw.

```
<esp>/accelerators/rtl/example_rtl/
├── hw
|   ├── example.xml                      # Accelerator description and register list
|   ├── hls
|   |   └── Makefile -> ../../../common/hls/Makefile
|   └── src
|       ├── example_rtl_basic_dma32
|       |   └── example_rtl_basic_dma32.v    # Empty top level of the ACC (32bit SoC)
|       └── example_rtl_basic_dma64
|           └── example_rtl_basic_dma64.v    # Empty top level of the ACC (64bit SoC); we use this template.
└── sw
    ├── baremetal                        # Bare-metal application, used to test accelerator functionality.
    |   ├── example.c
    |   └── Makefile
    └── linux                            # Not used in lab0 or midterm project
        ├── app                          # Linux test application
        |   ├── cfg.h
        |   ├── example.c
        |   └── Makefile
        ├── driver
        |   ├── example_rtl.c
        |   ├── Kbuild
        |   └── Makefile
        └── include
            └── example_rtl.h
```

7 Move to the working directory

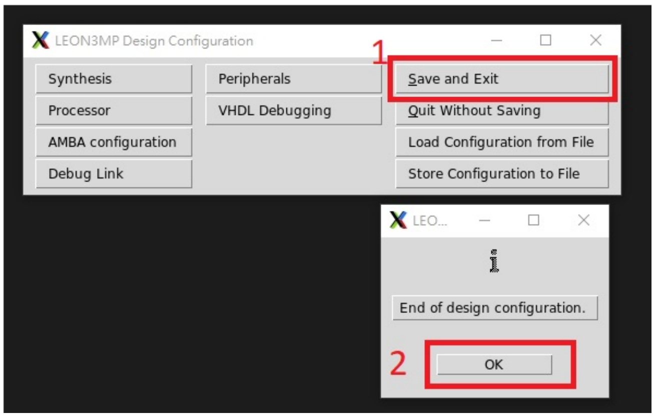
ESP supports different FPGA boards for the system prototype, but we don't need an FPGA board in this course. We will continue with the RTL simulation.

```
cd <esp>/socs/xilinx-vcu128-xcvu37p/
```

8 Set ESP Ethernet IP configurations

Since we are not going to prototype the system by FPGA, there is no need to modify any setting of this part.

```
make grlib-xconfig
```



9 Design your accelerator

In Lab 0, there is no need to modify any RTL code. Note that in the Midterm Project, you need to design your own accelerator tile with LeNet design by modifying template<design_name>_rtl_basic_dma64.v in <esp>/accelerators/rtl/<design_name>_rtl/hw/src/.

10 Update the design to ESP registered design

You should type this command to **update your design every time before simulation**.

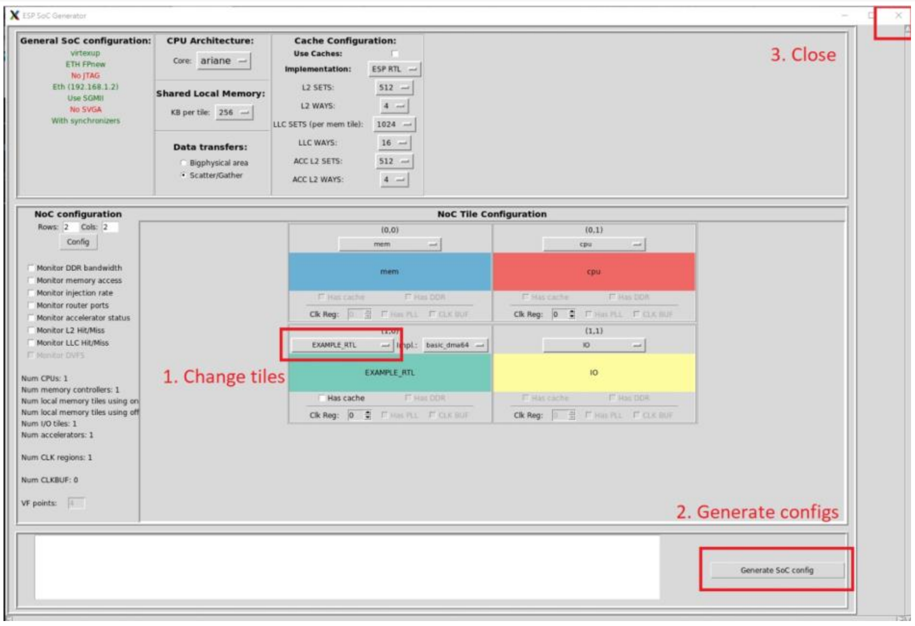
```
make example_rtl-hls
```

11 Set ESP accelerator tiles configuration

Change Accelerator Tiles to EXAMPLE_RTL and generate the configs.

Note: To change Accelerator Tiles, you need to press-then-select (長按拖曳選擇), not click-and-select.

```
make esp-xconfig
```



General SoC configuration:
virtexup
ETH FPnew
No JTAG
Eth (192.168.1.2)
Use SGMII
No SVGA
With synchronizers

CPU Architecture:
Core: ariane

Shared Local Memory:
KB per tile: 256

Data transfers:
Bigphysical area
[✓ Scatter/Gather]

Cache Configuration:
Use Caches:
Implementation: ESP RTL
L2 SETS: 512
L2 WAYS: 4
LLC SETS (per mem tile): 1024
LLC WAYS: 16
ACC L2 SETS: 512
ACC L2 WAYS: 4

NoC configuration
Rows: 2 Cols: 2
Monitor DDR bandwidth
Monitor memory access
Monitor injection rate
Monitor router ports
Monitor accelerator status
Monitor L2 Hit/Miss
Monitor LLC Hit/Miss
Monitor DVFS

Num CPUs: 1
Num memory controllers: 1
Num local memory tiles using on
Num local memory tiles using off
Num I/O tiles: 1
Num accelerators: 1

Num CLK regions: 1

Num CLKBUF: 0

VF points: 4

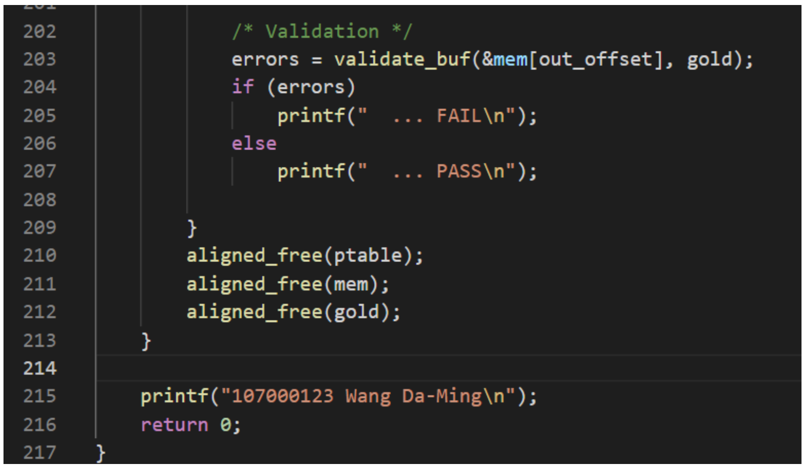
NoC Tile Configuration

(0,0)	(0,1)
mem	cpu
Has cache, Has DDR	Has cache, Has DDR
Clk Reg: 0, Has PLL, CLK BUF	Clk Reg: 0, Has PLL, CLK BUF

(1,0)	(1,1)
EXAMPLE_RTL, Impl.: basic_dma64	IO
Has cache, Has DDR	Has cache, Has DDR
Clk Reg: 0, Has PLL, CLK BUF	Clk Reg: 0, Has PLL, CLK BUF

12 Modify the bare-metal application

Modify the bare-metal application, <design_name>.c located in <esp>/accelerators/rtl/<design_name>_rtl/sw/baremetal/. Please add the following code to print your student ID and English name at the end of the **main** function.



```
            /* Validation */
            errors = validate_buf(&mem[out_offset], gold);
            if (errors)
                printf("  ... FAIL\n");
            else
                printf("  ... PASS\n");

        }
        aligned_free(ptable);
        aligned_free(mem);
        aligned_free(gold);
    }

    printf("107000123 Wang Da-Ming\n");
    return 0;
}
```

13 Compile the bare-metal application

Execute these commands to compile your C program before running the system-level simulation.

```
cd <esp>/socs/xilinx-vcu128-xcvu37p/
make soft
make  example_rtl-baremetal
```

14 Run the system-level RTL simulation

This step will run the simulation of the entire system, including NoC, CPU, memory, auxiliary components, and accelerator. At first, the binary machine code of the bare-metal application compiled in step 13 will be loaded into the main memory, in which the CPU will fetch instructions to execute at the beginning. Just like a regular C program, if you use **printf**, the text will be displayed on the screen.

First, compile the Verilog source code of the system, which may take about 30 minutes to 1 hour the first time you launch the simulation.

```
cd  <esp>/socs/xilinx-vcu128-xcvu37p/
setenv  TEST_PROGRAM  ./soft-build/ariane/baremetal/example_rtl.exe
make  sim-gui
```

Note: If you don't want to monitor the waveform, you can use the following commands to invoke the CLI mode, then type **"run -a"** in ModelSim to start the simulation.

```
cd  <esp>/socs/xilinx-vcu128-xcvu37p/
setenv  TEST_PROGRAM  ./soft-build/ariane/baremetal/example_rtl.exe
make  sim
```

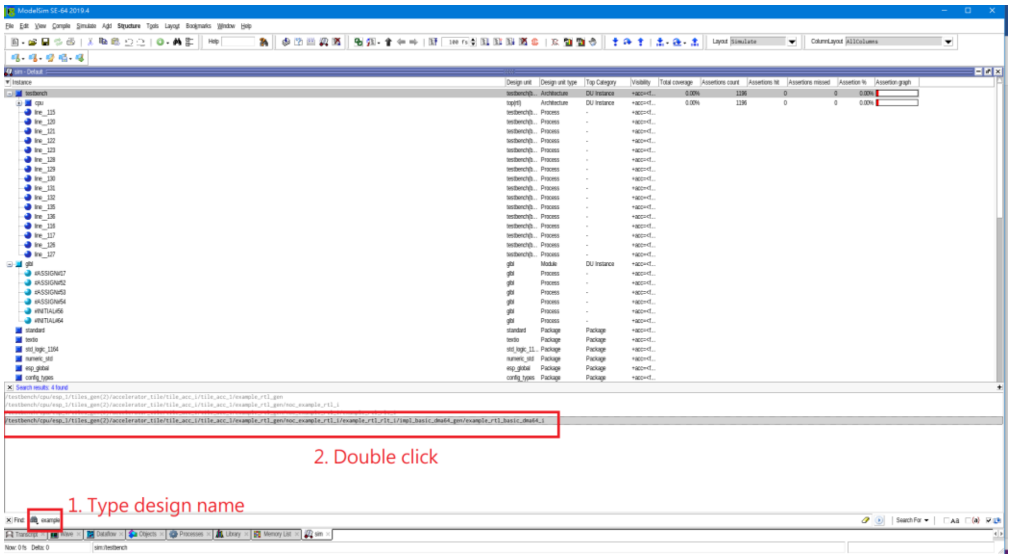
Second, start and monitor the simulation waveform in ModelSim. Please follow the procedures in the figures below. Make sure you understand what you are doing. Feel free to post any questions on EECLASS forum.

(1) Select design

Step 1: Type the design name “**example”** to find design.

Step 2: Double click the instance …/**example_rtl_basic_dma64_i**

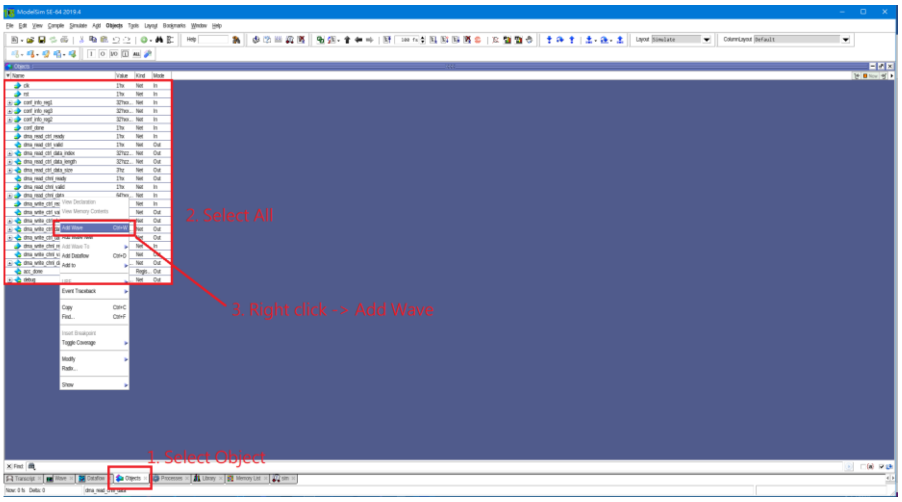
Note: if the **Find** section doesn’t show up in the window, use **ctrl + F** bring it up.



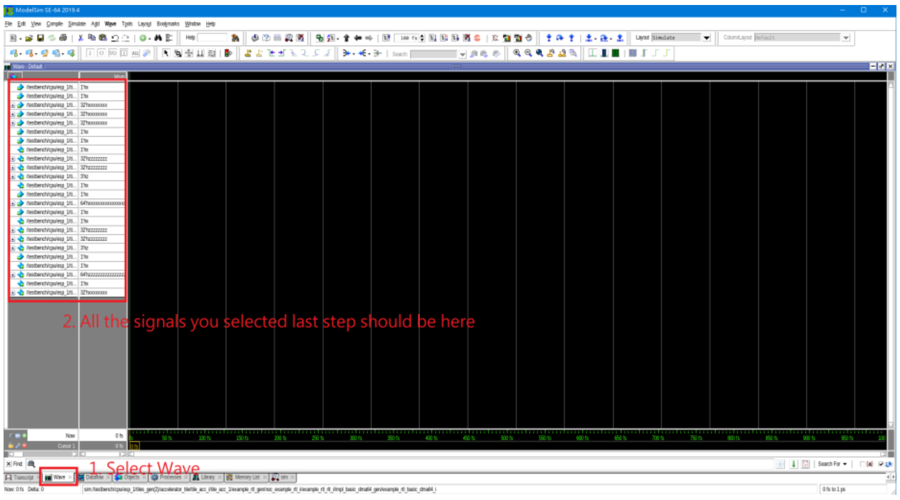
```
/testbench/cpu/esp_1/tiles_gen(2)/accelerator_tile/tile_acc_i/tile_acc_1/example_rtl_gen
/testbench/cpu/esp_1/tiles_gen(2)/accelerator_tile/tile_acc_i/tile_acc_1/example_rtl_gen/noc_example_rtl_i
[illegible]
/testbench/cpu/esp_1/tiles_gen(2)/accelerator_tile/tile_acc_i/tile_acc_1/example_rtl_gen/noc_example_rtl_i/example_rtl_rlt_i/impl_basic_dma64_gen/example_rtl_basic_dma64_i
```

(2) Select signals and add to the waveform

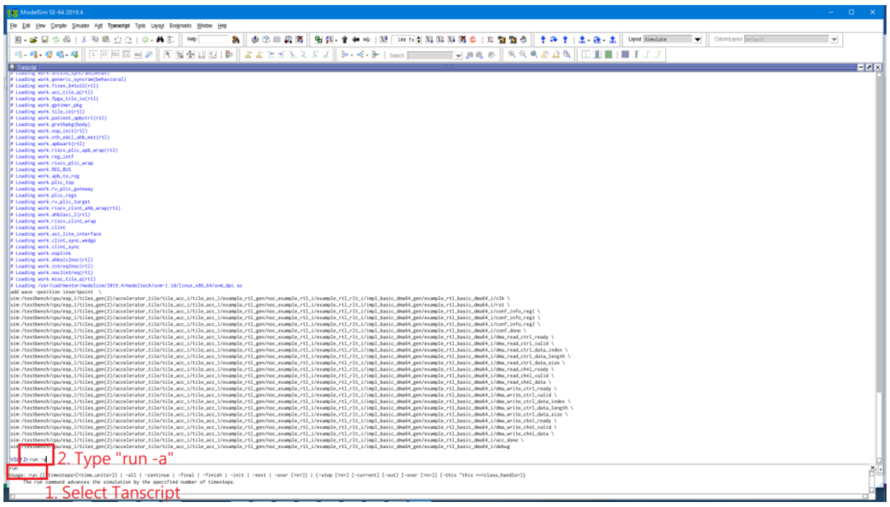
Note: **Object** section may appear on other sides, please find it in ModelSim window.



(3) Check the selected signals in the waveform list.

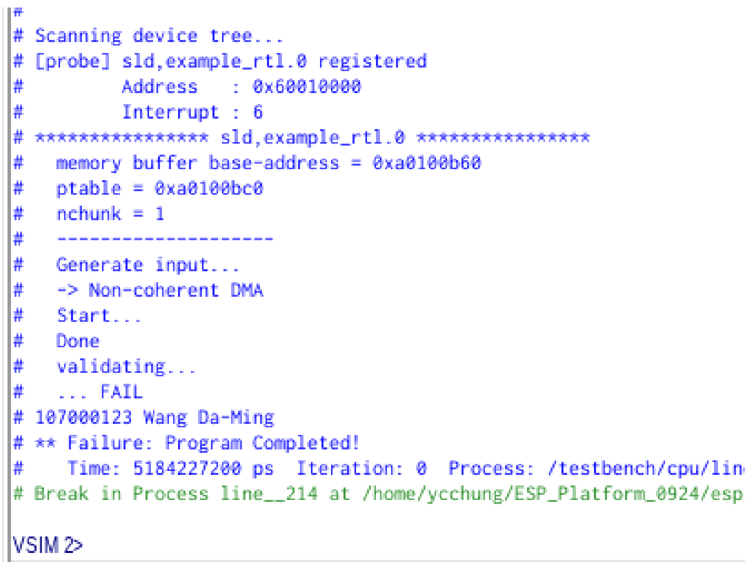


(4) Start the simulation by typing **run -a**.



(5) Wait for a while for the simulation. Your student ID and name should appear in the result. Note: Don’t worry about the FAIL and Failure here. **FAIL** is because we don’t have proper validating program; **Failure** is because of terminating the simulation. Both of them are fine in this lab.

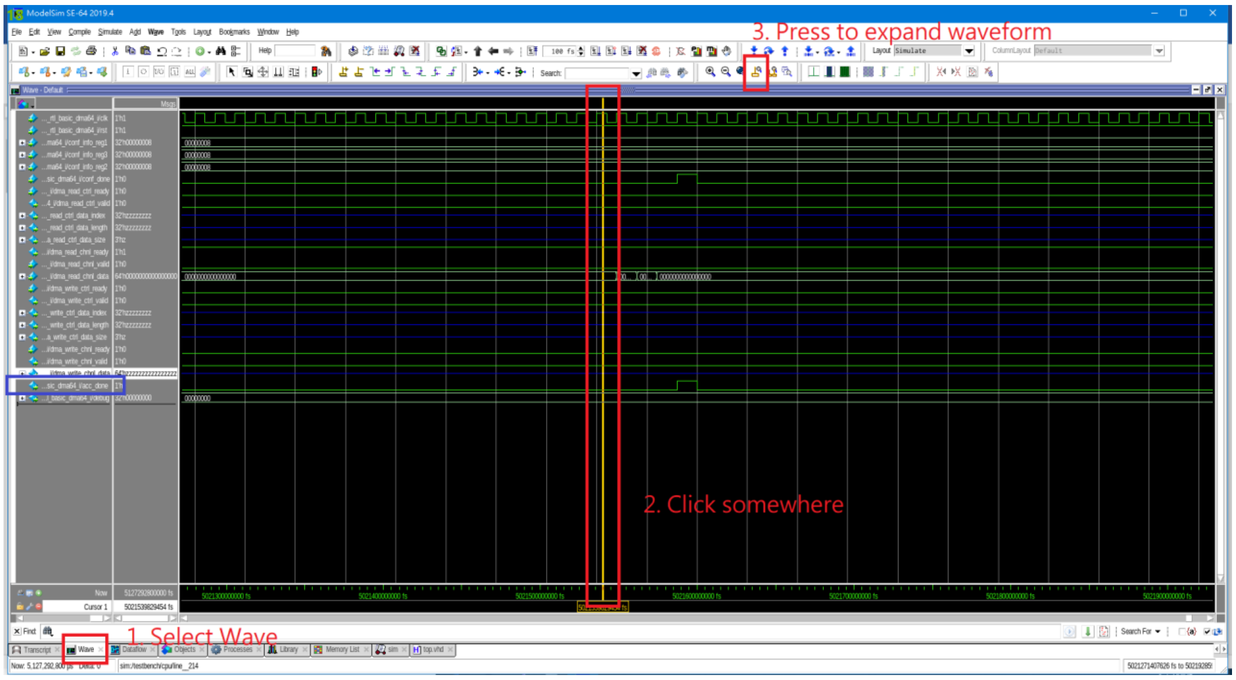
Checkpoint 1: Take a screenshot of your result and paste it to the report.



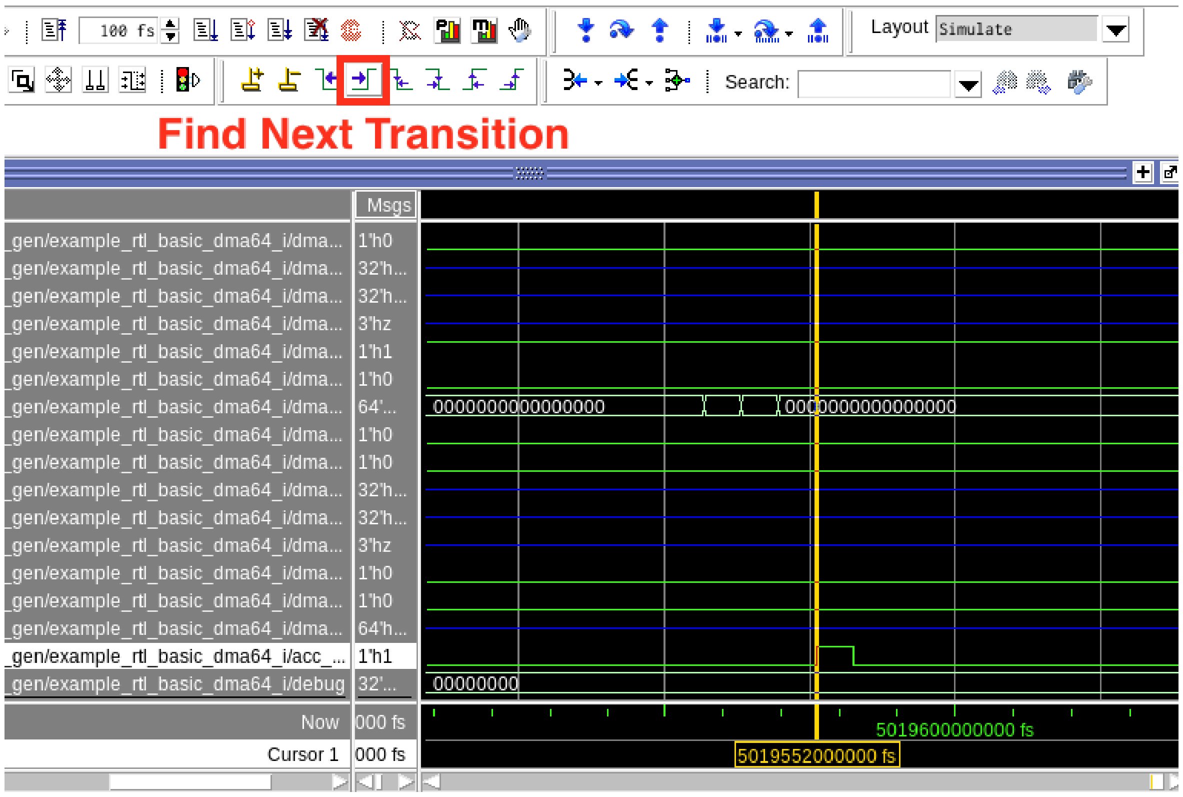
```
#
# Scanning device tree...
# [probe] sld,example_rtl.0 registered
#         Address   : 0x60010000
#         Interrupt : 6
# **************** sld,example_rtl.0 ****************
#   memory buffer base-address = 0xa0100b60
#   ptable = 0xa0100bc0
#   nchunk = 1
#   --------------------
#   Generate input...
#   -> Non-coherent DMA
#   Start...
#   Done
#   validating...
#   ... FAIL
# 107000123 Wang Da-Ming
# ** Failure: Program Completed!
#    Time: 5184227200 ps  Iteration: 0  Process: /testbench/cpu/lin
# Break in Process line__214 at /home/ycchung/ESP_Platform_0924/esp

VSIM 2>
```

(6) Look at the waveform, use the button in the figure to locate the signal **acc_done**.

Checkpoint 2: Take a screenshot that includes the pulse of acc_done and paste it to report.



Note: You may utilize the features on the toolbar to help debugging (e.g., **Find Next Transition** button to find the pulse). Hover your mouse cursor over these buttons to check them out.



Submission

Please answer the following questions in your report, named it as
<student_ID>_lab0_report.pdf
e.g. 107000123_lab0_report.pdf

(1) Which is the interconnect protocol that ESP uses to integrate CPU, memory, and accelerator? **Hint: Introduction**
(2) What does make example_rtl-hls do? And when should it be used?
(3) What does make example_rtl-baremetal do? And when should it be used?
(4) There is a syntax error in example_rtl_basic_dma64.v. Please explain how to fix it.
(5) Where is the C program executed?
(6) How does the C program notify the accelerator to start the computing?
(7) How does the C program know the computing in the accelerator is finished?
(8) Figure of checkpoint 1.
(9) Figure of checkpoint 2.