Homework 5: System-level DNN Accelerator

Submission Due Dates:
2024/5/26 23:59

Objective

Integrating the LeNet accelerator engine into the Embedded Scalable Platform (ESP).

Prerequisites

1 Introduction of 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).

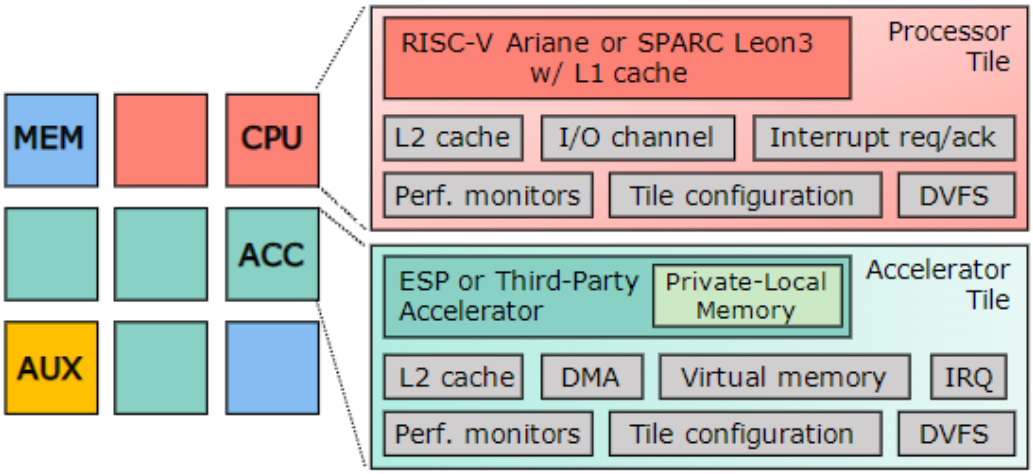


Figure referenced from: https://www.esp.cs.columbia.edu

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 Homework 5 requires you to embed the LeNet accelerator in Homework 4 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).

2 Reference of DMA controller interface of ESP

Read the document before moving on.
https://www.esp.cs.columbia.edu/docs/specs/esp_accelerator_specification.pdf

Introduction

1 Accelerator tile of ESP

Since ESP provides a complete system architecture, there is no need to design the entire system by ourselves, the thing we need to do is to design a DMA interface compatible accelerator kernel.

The ESP provides a holistic system platform. So, designers can focus on developing their hardware accelerator and integrating it into the system once wrapped by a compatible interface. The interface to an accelerator kernel includes the read and write port for data transfers through DMA (direct memory access) requests, configuration port, and IRQ (interrupt query). Please refer to the documentation in **Prerequisites** for further details.

2 Memory mapping

In the Homework 5 project, the software memory in the bare-metal application has 32 bits per access. But the DMA's data width is 64 bits since the Ariane CPU we used is the 64-bit version. Figure 1 shows the word addresses for the software memory vs. double-word addresses for DMA. If the accelerator accesses Data 2, Data 3, and Data 4, the (word) addresses are 2, 3, and 4 in the software memory. However, the DMA sees the double-word addresses. So, the corresponding addresses are 1 and 2 in DRAM. For another example, if you want to read twenty-one 32-bit data in the software memory from the offset of 883, you need to access 11 ($\left\lceil \frac{21}{2} \right\rceil$) 64-bit data from the address 441 ($\left\lfloor \frac{883}{2} \right\rfloor$) by the DMA controller.

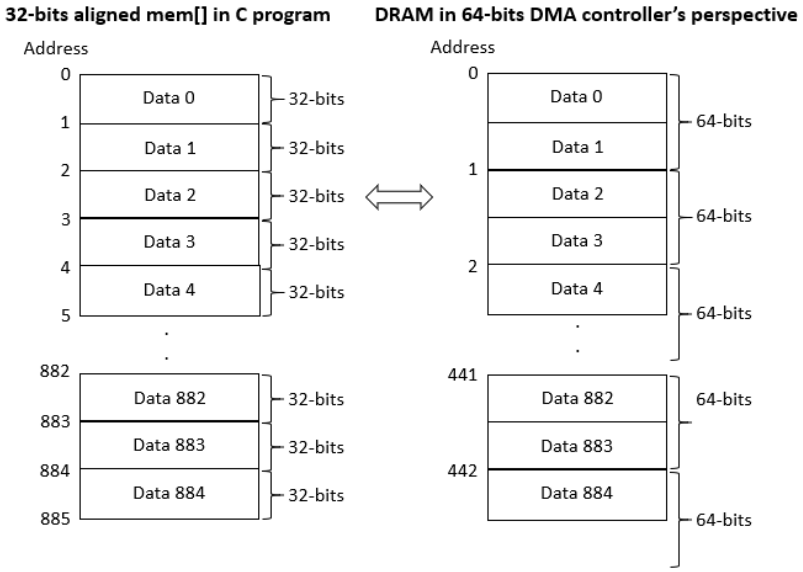


Figure 1: Software memory and DRAM mapping.

Overall System Architecture

In the Homework 5, you need to implement an accelerator kernel that includes the LeNet accelerator engine in Homework 4, SRAM block, and DMA controller interface (see Figure 2).

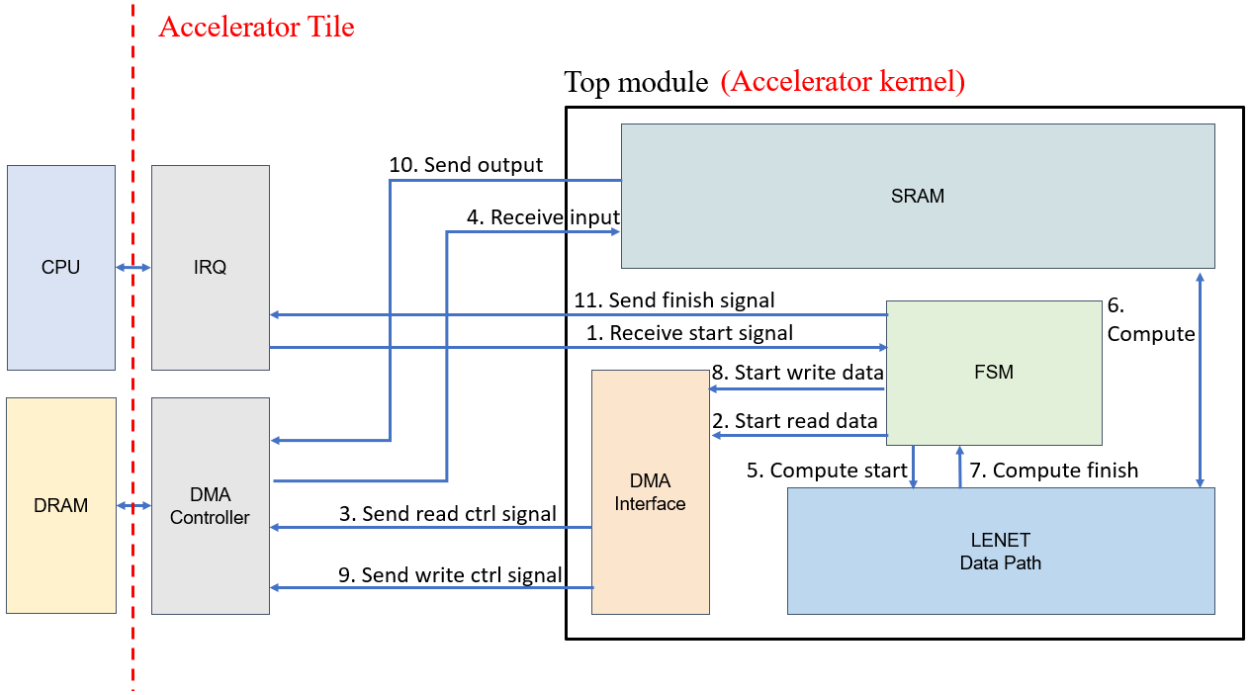


Figure 2: Accelerator block diagram.

The computation flow of the accelerator tile is summarized as follows:

(1) Receiving the start signal

As Figure 3 shows, the bare-metal application (C program) in the CPU generates a start interrupt to the IRQ, which will send a **single-cycle active-high** signal (***conf_done***) to notify the accelerator to start computing.

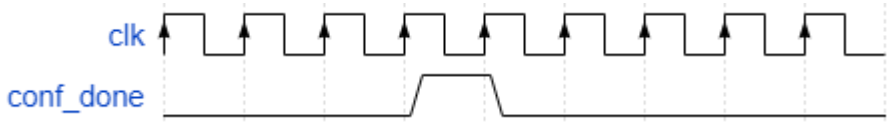


Figure 3: Start signal waveform.

(2) Reading the data

The accelerator utilizes the DMA interface to load input feature maps, weights, and biases from the external DRAM to the internal SRAM via an external DMA controller. You may integrate the DMA interface into the top-level module instead of putting it in a separate module.

(3) Sending the read-control signal

Every time you want to read the data from DRAM, you should send the control signals to DMA controller first, then wait for the acknowledgment.

Note that the software memory is arranged with 32-bit words. That is, the

dma_read_ctrl_size needs to be **3'b010**. Please refer to **Prerequisites** and Figure 4 for more details.

(4) Receiving the input

You need to store the received data into the corresponding SRAM with the layout described in **Action Item** to ensure that the datapath can access the correct data.

Please refer to the tutorial in **Prerequisites** to study the protocol of the DMA controller's handshaking. We provide an example to load eight 32-bits data from the software memory with the offset of 0x20. Please refer to **introduction: memory mapping** for the reason of using the index of 0x10 and length of 4.

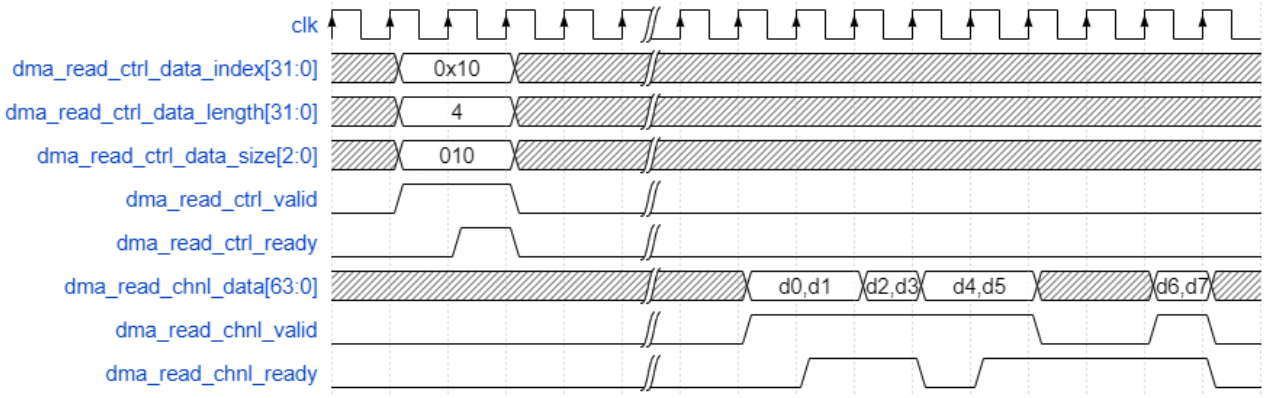


Figure 4: DMA reading waveform.

The example in the tutorial includes two kinds of data size (3'b010 and 3'b011). You only need to focus on the case with **dma_read_ctrl_size** of 3'b010.

(5) Starting the compute

Please refer to the testbench in Homework 4 to understand how to send a **single-cycle active-high start signal** to start computing.

(6) Computing

LeNet datapath performs the computation based on the image and weights in the two SRAMs, then writes the results back to the activation SRAM.

(7) Finishing the compute

Just like Homework 4, a **single-cycle active-high** finish signal notifies the FSM that the computing is finished.

(8) Writing the data

The accelerator utilizes the DMA interface to store the activation from the internal SRAM to the external DRAM via an external DMA controller. Again, you may integrate the DMA interface into the top-level module instead of putting it in a separate module.

(9) Sending the write control signal

Every time you want to write the data to DRAM, you should send the control signals to the DMA controller first, then wait for the acknowledgment.

Note that the software memory is arranged with 32-bit words. That is, the

dma_read_ctrl_size needs to be 3'b010. Please refer to **Prerequisites** and Figure 5 for more details.

(10) Sending the output

You need to read the results from the internal SRAM, then write them to the external DRAM with the proper memory layout described in the section **Action Item** to ensure that the CPU can access them correctly. Be careful of the activation data size: FC2 result is **32 bits**, but others are **8 bits**. You need to **puzzle them into 64 bits** as the basic transaction unit for the DMA controller. Please refer to the tutorial in **Prerequisites** to study the protocol of DMA controller's handshaking. We also provide an example to store eight 32-bits data to the software memory with the offset of 0x20. Please refer to the **introduction: memory mapping** for the reason of using the index of 0x10 and length of 4.

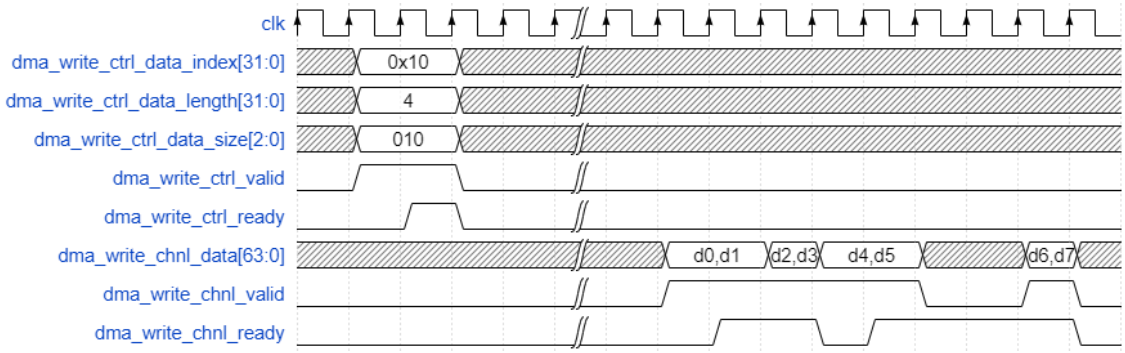


Figure 5: DMA writing waveform.

(11) Sending the finish signal

To inform the CPU that the results are ready, you need to send a **single-cycle pulse active high** signal (**acc_done**) to the IRQ. After that, the IRQ will generate an interrupt to notify the CPU.

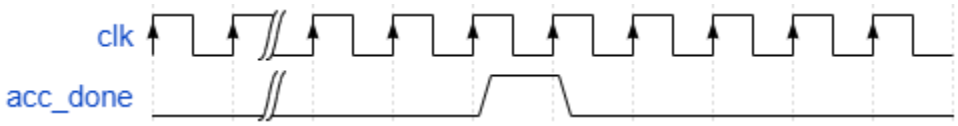


Figure 6: Finish signal waveform.

3 I/O port of the accelerator tile

Signals	I/O	Description
clk	Input	Accelerator clock
rst	Input	Accelerator reset (**active low**)
conf_done	Input	Configuration registers are valid and computation can start. This signal is **active high** and asserted for one clock cycle to trigger the accelerator

		execution.
conf_info_scale_CONV1	Input	Quantization scale for CONV1
conf_info_scale_CONV2	Input	Quantization scale for CONV2
conf_info_scale_CONV3	Input	Quantization scale for CONV3
conf_info_scale_FC1	Input	Quantization scale for FC1
conf_info_scale_FC2	Input	Quantization scale for FC2
acc_done	Output	**Single-cycle pulse (active high)**. This flag indicates that the accelerator has completed its task. The pulse should occur only after the last DMA write transaction has been completed and all output data have been transferred from the SRAM to the memory hierarchy.
debug	Output	Not used, just set to **32'b0**
dma_[read|write]_ctrl_data_index	Output	The offset of a DMA read or write transaction.
dma_[read|write]_ctrl_data_length	Output	The length of a DMA read or write transaction.
dma_[read|write]_ctrl_data_size	Output	Bitwidth of the data token for the DMA transaction. This signal is used to correct the NoC flits when the processor architecture follows the big-endian convention to store the data in the memory. In Homework 5, we use a 64-bits CPU, but the software memory is aligned to 32 bits. It should be set to **3'b010.**
dma_[read|write]_ctrl_valid	Output	Flag indicating a new DMA transaction request. When set, all data fields must be valid. This flag must **not depend combinationally** on the corresponding ready signal.
dma_[read|write]_ctrl_ready	Input	Flag indicating that the ESP socket is ready to accept a new DMA request. This flag must **not depend combinationally** on the corresponding valid signal.

Action Items

0 Set up Environment

(1) CAD tool setup script

We provide setup.sh on EECLASS, **remember to execute it every time you log into a CAD workstation**.

```
source setup.sh
```

(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
```

1 Set up the working folder

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/

(1) Login to an available server and create Homework 5 accelerator template

```
cd <esp>
./tools/accgen/accgen.sh
=== Initializing ESP accelerator template ===
  * Enter accelerator name [dummy]: lenet
  * Select design flow (Stratus HLS, Vivado HLS, hls4ml, RTL) [S]: R
  * Enter ESP path [/home/ycchung/ESP_Platform_0924/esp]: press ENTER
```

```
  * Enter unique accelerator id as three hex digits [04A]: 058
  * Enter accelerator registers
    - register 0 name [size]: scale_CONV1
    - register 0 default value [1]: 8
    - register 0 max value [8]: press ENTER
    - register 1 name []: scale_CONV2
    - register 1 default value [1]: 8
    - register 1 max value [8]: press ENTER
    - register 2 name []: scale_CONV3
    - register 2 default value [1]: 8
    - register 2 max value [8]: press ENTER
    - register 3 name []: scale_FC1
    - register 3 default value [1]: 8
    - register 3 max value [8]: press ENTER
    - register 4 name []: scale_FC2
    - register 4 default value [1]: 8
    - register 4 max value [8]: press ENTER
    - register 5 name []:press ENTER
  * Configure PLM size and create skeleton for load and store:
    - Enter data bit-width (8, 16, 32, 64) [32]: 32
    - Enter input data size in terms of configuration registers [scale_FC1]: press ENTER
      data_in_size_max = 8
    - Enter output data size in terms of configuration registers [scale_FC1]: press ENTER
      data_out_size_max = 8
    - Enter an integer chunking factor [1]: press ENTER
      Input PLM has 8 32-bits words
      Output PLM has 8 32-bits words
    - Enter 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 lenet ===
```

(2) Check the accelerator folder and add the template

- Hardware design

  Please add the design in Homework 4 (lenet.v and related HDL files) and template files (lenet_rtl_basic_dma64.v, SRAM_activation_1024x32b.v, and SRAM_weight_16384x32b.v) in template/hardware/ into this folder:

  ```
  <esp>/accelerators/rtl/lenet_rtl/hw/src/lenet_rtl_basic_dma64/
  ```

- Software C program

  Please add lenet.c and pattern/ in template/software/ into this folder:

```
<esp>/accelerators/rtl/lenet_rtl/sw/baremetal/
```

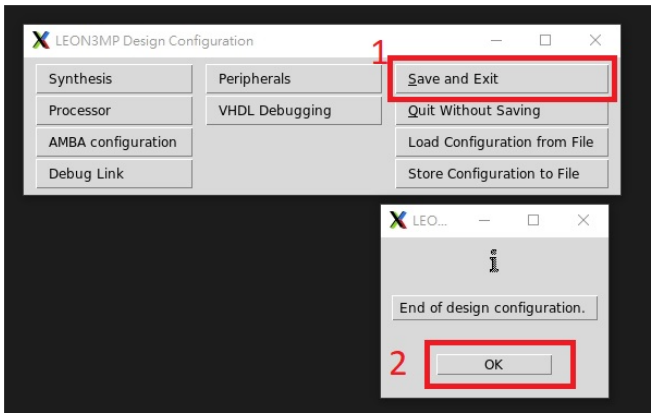
(3) Move to the working directory.

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

(4) 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
```



(5) Update the design to ESP registered design

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

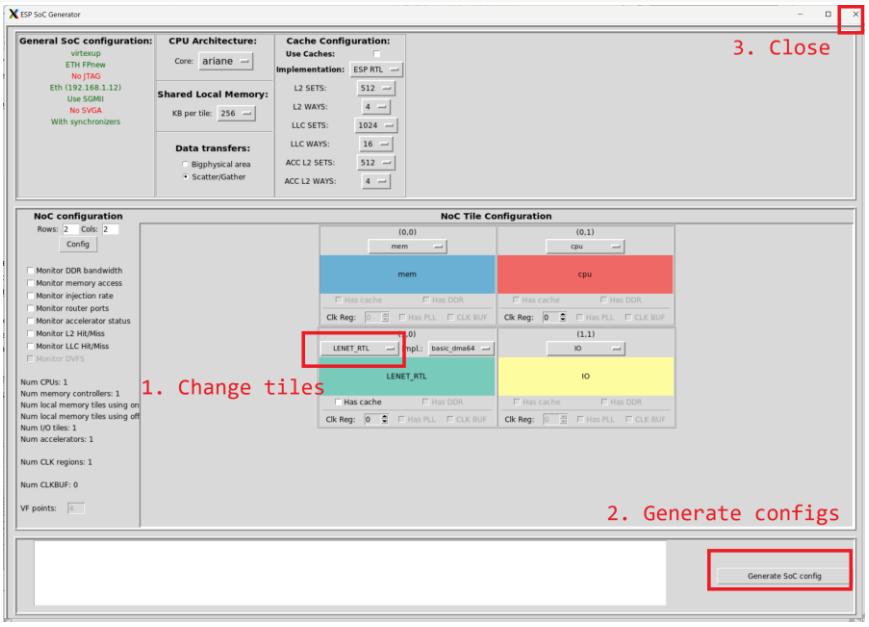
```
make lenet_rtl-hls
```

(6) Set ESP accelerator tiles configuration

Change Accelerator Tiles to LENET_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
```



2 Hardware RTL design

In the block diagram of previous sections, we have four modules to implement.

(a) Self-built SRAM

ESP provides a dual-port SRAM wrapper **BRAM_2048x8**, which is built from the Block RAM (BRMA) component in the Xilinx FPGA cell library (Just treat the Block RAM as SRAM in this homework. But they are essentially different in physical implementation). The following table is the I/O list of the basic SRAM module of **BRAM_2048x8**.

Name	Bitwidth	I/O	Description
CLK	1	Input	Clock.
A0	11	Input	Read or write address of port 0.
D0	8	Input	The data to be written to RAM[A0]
Q0	8	Output	Output of RAM[A0] in the next cycle.
WE0	1	Input	Write enable: set to 1 to write RAM[A0] = D0, otherwise, set to 0.
WEM0	8	Input	Set to 8'b0.
CE0	1	Input	Set to 1'b1 to enable SRAM.
A1	11	Input	Read or write address of port 1
D1	8	Input	The data to be written to RAM[A1]
Q1	8	Output	Output of RAM[A1] in the next cycle.
WE1	1	Input	Write enable: set to 1 to write RAM[A1] = D1, otherwise, set to 0.
WEM1	8	Input	Set to 8'b0.
CE1	1	Input	Set to 1'b1 to enable SRAM.

In Homework 5, you need two kinds of SRAM, which is the same as Homework 4:

Name	Data width	Depth	Type
SRAM_weight_16384x32b	32	16384	Dual-port
SRAM_activation_1024x32b	32	1024	Dual-port

You need to build these two kinds of SRAM by instantiating **BRAM_2048x8**. But you don't need to include this file to your design path. The ESP platform already includes it in the compile command. Please refer to this link to study the details of **BRAM_2048x8.v**.

Name	Data width	Depth	Type
BRAM_2048x8	8	2048	Dual-port

Note: Please use the **templates** SRAM_activation_1024x32b.v and SRAM_weight_16384x32b.v we provided.

(b) Interface for DMA controller

Figure 7 describes the DRAM data layout, indicating how the data are placed in DRAM.

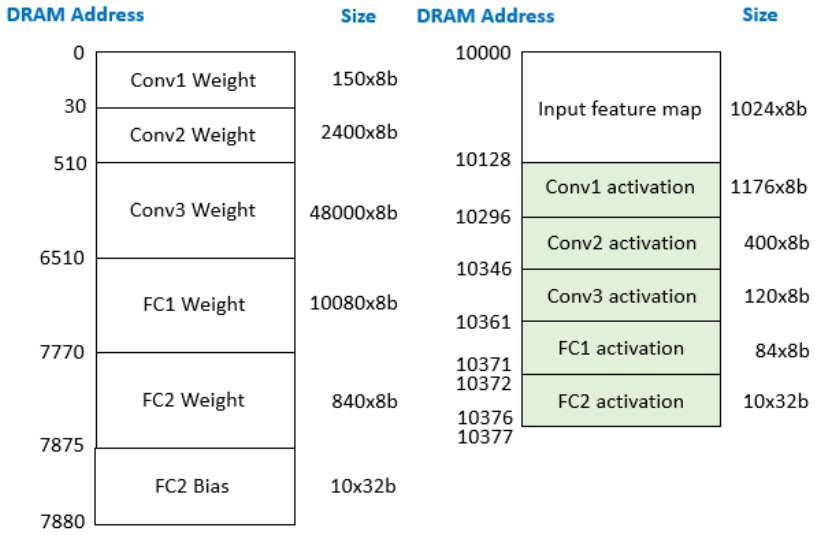


Figure 7: DRAM memory layout.

1. DMA read:

   You need to load 7880 64-bit weight data from DRAM address 0 into the weight SRAM, and 128 64-bit image data from DRAM address 10000 into the activation SRAM as Figure 8 shows.

2. DMA write:

   After the computation, the data layout of the activation SRAM should be the same as the Figure 8. You should write out all images and activation from SRAM to DRAM via the DMA controller.

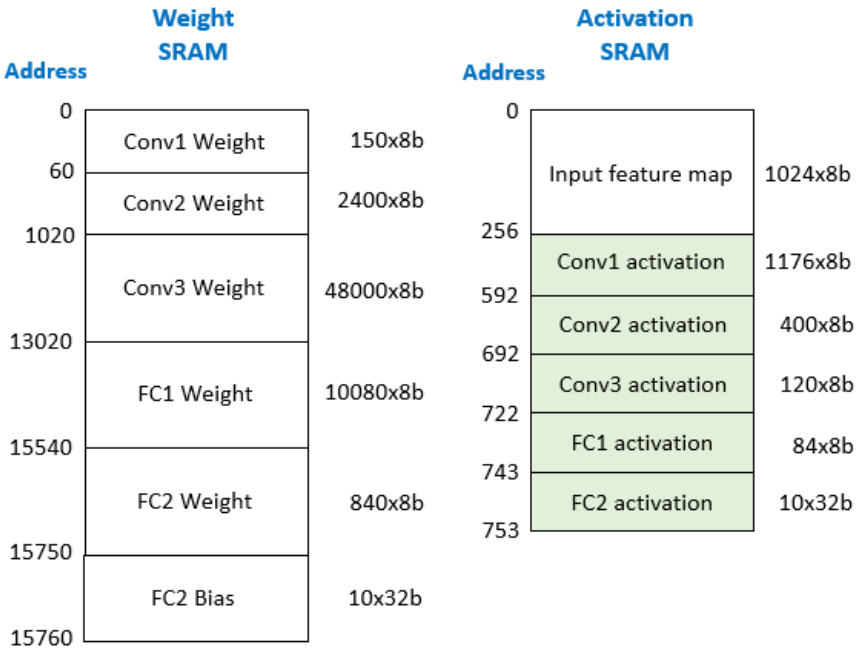


Figure 8: Data layout of the weight & activation SRAMs.

(c) The LeNet datapath

You need to integrate the accelerator engine of Homework 4 into the top module, making it capable of computing the LeNet.

(d) The FSM

The FSM may not be a separate module, and you may integrate it into the top-level module. Before building the entire accelerator, you may design the data movement first by loading the input feature maps to the **activation SRAM**, then writing them out to DRAM (address 10000 to 10128), which can be verified in the next section with the bare-metal application C program. With the correct DRAM-SRAM access, integrating the LeNet datapath and the rest SRAMs may be easier (hopefully).

3 Software C program

After designing the hardware, we should verify the functional correctness of our accelerator design. This software runs in RISC-V CPU, accessing data from DRAM and controlling the accelerator to do the computation. In Homework 5, we provide a complete program <esp>/accelerators/rtl/lenet_rtl/sw/baremetal/lenet.c to verify your design.

(a) Pattern generation

Since we are running the bare-metal application, there is no file system available (no operating system here). We should prepare the patterns (image, activation, and weight) in the C header file, then compile them with lenet.c.

You need to prepare three C header files: **weight.h**, **image.h**, and **golden.h**, then place them into <esp>/accelerators/rtl/lenet_rtl/ sw/baremetal/pattern/. Please use these three CSV files you generated in Homework 4 for the Homework 5 project. Refer to Table 1 and Figure 9 for more details.

Table 1: Information for Header files

Header file name	CSV file name	Data type	Size
weight.h	weights.csv	int32_t	15760
image.h	image.csv	int32_t	256
golden.h	golden.csv	int32_t	753

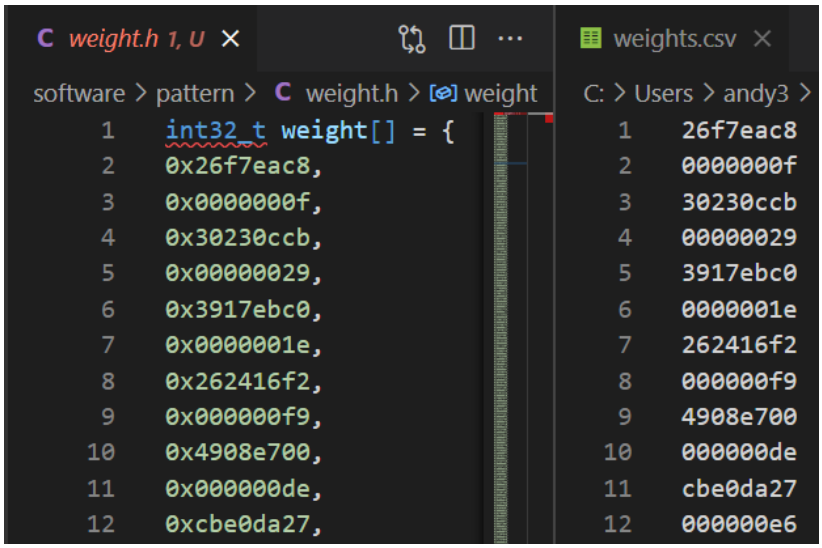


Figure 9: Comparison of header and CSV files.

(b) TODO in lenet.c

Please complete the TODO in C program.

4 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 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.

Note: Spyglass report may show your design has the black box (**BRAM_2048x8**). It is acceptable.

```
make lenet_rtl-hls              # Type this command after modify HDL files
make lenet_rtl-baremetal        # Type this command after modify C program
setenv TEST_PROGRAM ./soft-build/ariane/baremetal/lenet_rtl.exe
make sim-gui
```

Note 1: 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.

```
setenv TEST_PROGRAM ./soft-build/ariane/baremetal/lenet_rtl.exe
make sim
```

Note 2: If the above commands do not work the next few times, it may be because the workstation disconnected halfway through the previous execution, causing some files to crash. Just delete the simulation folder and rerun.

```
rm -rf soft-build/
```

```
rm -rf modelsim/
make lenet_rtl-hls
make lenet_rtl-baremetal
make sim-gui
```

Note 3: If you get a permission denied error, it may be because you have a process still running that locks out the file write permission. Please kill the process and delete the modelsim folder.

```
pkill -u <user_ID>
rm -rf modelsim/
```

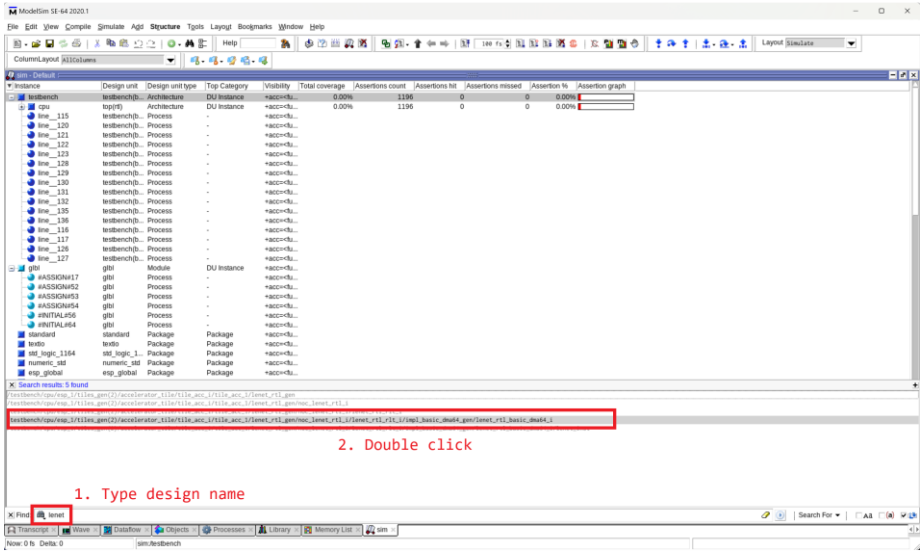
Second, start and monitor the simulation waveform in ModelSim. Please follow the procedures in the figures below. Make sure you understand what you are doing. Feel free to post any questions on EECLASS forum.

(1) Select design

Step 1: Type the design name "**lenet"** to find design.

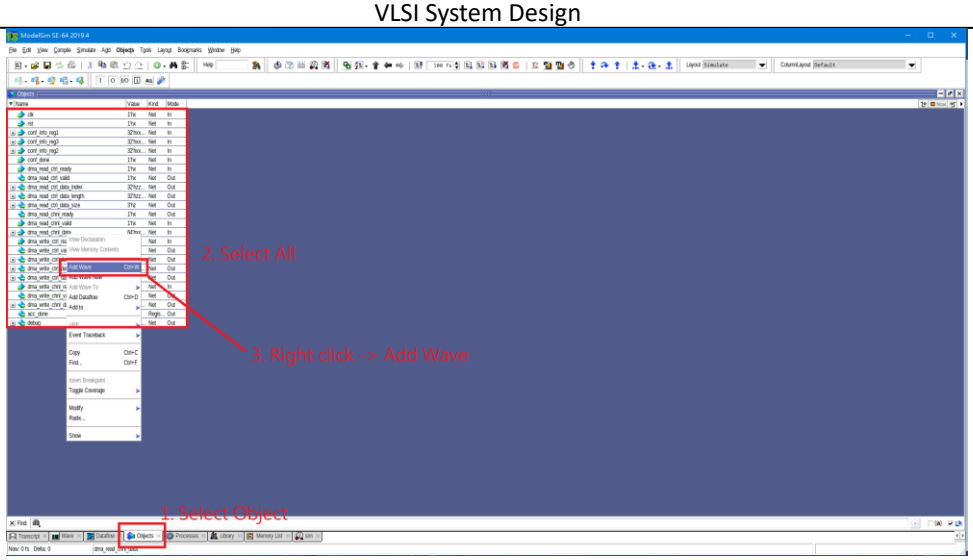
Step 2: Double click the instance …/**lenet_rtl_basic_dma64_i**

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

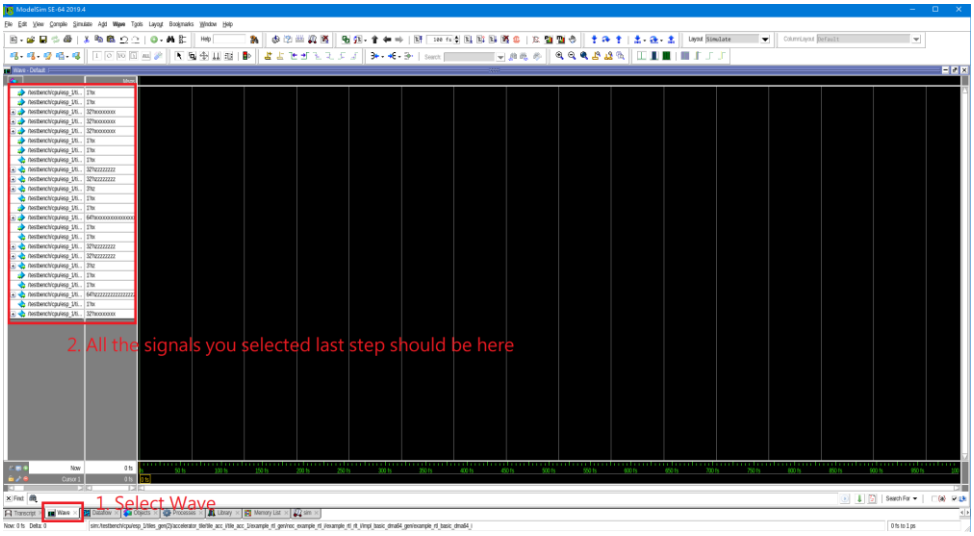


(2) Select signals and add to the waveform

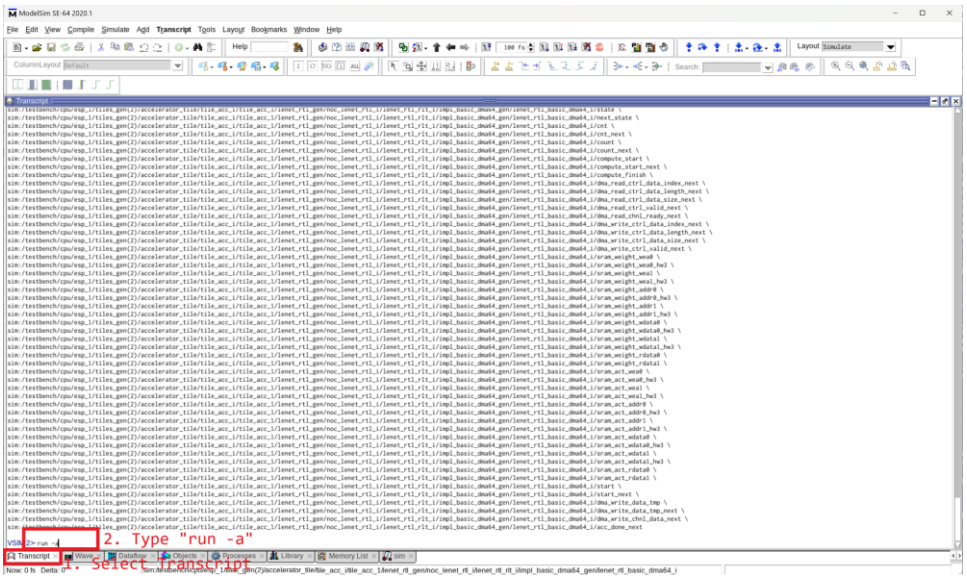
Note: Object section may appear on other sides, please find it in ModelSim window.



(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. Figure 10 shows the message when the simulation is passed.

Note: Don’t worry about the Failure here. Failure is because of terminating the simulation.

```
# ***************** sld,lenet_rtl.0 *****************
#   memory buffer base-address = 0xa0100b30
#   ptable = 0xa011e000
#   nchunk = 1
#   ---------------------
#   Generate input...
#   -> Non-coherent DMA
#   Start...
#   Done
#   validating...
# ===> Image pass!
# ===> Conv1 pass!
# ===> Conv2 pass!
# ===> Conv3 pass!
# ===> FC1 pass!
# ===> FC2 pass!
# [PASS] Congratulation! All results are correct
# ** Failure: Program Completed!
#    Time: 13089494400 ps  Iteration: 0  Process: /te
```

Figure 10: Pass message.

Report

Please answer the following questions in the report.

(1) Which is the interconnect protocol that ESP uses to integrate CPU, memory, and accelerator? Hint: Introduction for ESP

(2) How does the C program notify the accelerator to start the computing?

(3) How does the C program know the computing in the accelerator is finished?

(4) How do you design your accelerator? Please draw the FSM and block diagram to explain the overall architecture.

(5) How do you design your DMA controller interface to transfer data? Please draw the block diagram and FSM.

(6) How do you build two SRAMs in this project? Please draw the block diagram.

(7) Please briefly explain why we write images from mem[20000] to mem[20255], but read images from address 10000 to address 10127 in the accelerator?

(8) What is the function of the following code?

```
iowrite32(dev, LENET_SCALE_CONV2_REG, scale_CONV2);
iowrite32(dev, LENET_SCALE_CONV3_REG, scale_CONV3);
iowrite32(dev, LENET_SCALE_CONV1_REG, scale_CONV1);
iowrite32(dev, LENET_SCALE_FC2_REG, scale_FC2);
iowrite32(dev, LENET_SCALE_FC1_REG, scale_FC1);
```

(9) What is the function of the following code? Please explain line by line

```
done = 0;
while (!done) {
    done = ioread32(dev, STATUS_REG);
    done &= STATUS_MASK_DONE;
}
iowrite32(dev, CMD_REG, 0x0);
```

Grading

1. Simulation pass (70%)
2. Self-built SRAM (10%)
3. Spyglass report to show your design is synthesizable (10%)
4. Report (10%)

Submission

1. Please upload the following files to EECLASS, otherwise you will get a 20-point penalty.
   - SRAM_activation_1024x32b.v
   - SRAM_weight_16384x32b.v
   - lenet_rtl_basic_dma64.v # Include all the other modules (**e.g. hw4 lenet engine. Don't hand in as separate files, you will get zero points.**)
   - lenet.c # Note that **it's not the lenet.v you hand in hw4**
   - weight.h
   - image.h
   - golden.h
   - spyglass.rpt # Please follow the tutorial to generate this file.
   - studentID_hw5_report.pdf # Use the template we provide.
2. **DO NOT** compress them!