Homework Assignment #4

Submission Due: 2021/06/06 23:59

Objective

1. In this assignment, you will implement DNN in Verilog and C. You should partition the application into hardware and software parts. For the hardware part, wrap your Verilog modules using the **memory-mapped I/O (MMAP)** method. For the software part, compile your C code with the RISC-V toolchain. The program will run on PicoRV32, a CPU core that implements the RISC-V ISA. Profiling tools are provided for you to see how these architectures affect the performance.

References

1. PicoRV32: https://github.com/cliffordwolf/picorv32 (MUST READ!!!)
2. RISC-V: https://riscv.org/specifications/ (For Your Reference)

Design Descriptions

Setup

1. Login only to **ic56-ic58**. We have built the RISC-V cross-compiler toolchain on these machines under `/users/course/tools/riscv32i`
2. Download the source code from eeclass
3. Under directory `picorv32`, compile the program and run a Verilog simulation using the following command:

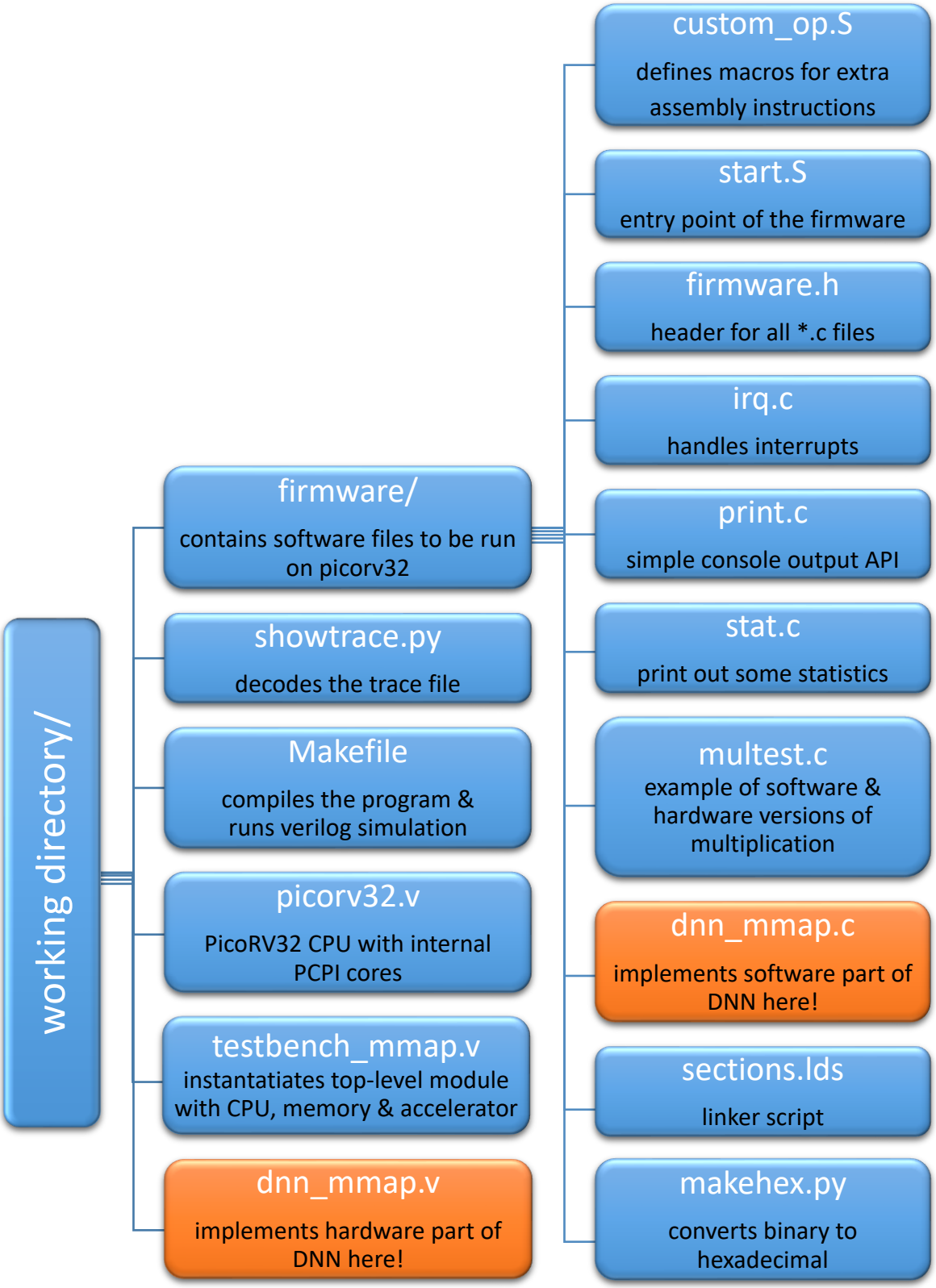
```
make pcpi
```

4. You should see the following output:

```
----------------------------------------------------------
EBREAK instruction at 0x000004A8
pc  000004AB    x8  00000000    x16 01E7223E    x24 00000003
x1  00000478    x9  00000000    x17 4C000000    x25 00000001
x2  00C00000    x10 20000000    x18 00000002    x26 00009002
x3  DEADBEEF    x11 20002470    x19 00000160    x27 00009002
x4  DEADBEEF    x12 0000004F    x20 00000000    x28 8B578493
x5  00002078    x13 0000004E    x21 00000015    x29 022D5E12
x6  00000008    x14 00000045    x22 000004A8    x30 00000000
x7  00000000    x15 000001E0    x23 000004A8    x31 000000C4
----------------------------------------------------------
Number of fast external IRQs counted: 0
Number of slow external IRQs counted: 0
Number of timer IRQs counted: 0
TRAP after 311338 clock cycles
ALL TESTS HAVE FINISHED
```

The “ALL TESTS HAVE FINISHED” indicates that the PicoRV32 has finished all the instructions. Note that we do not check the execution results. You should print out the results and compare them with golden data by yourself.

Directory Structure



1. The main Verilog and C codes you need to work with are those in the orange boxes.

Description on the Multest Example

`start.S` is the entry point of our program. It contains the startup routine, which is responsible for initializing and calling the rest of the program. Undefine the corresponding macros at the top if you don't want to run all of the functions.

Without underlying operating system support, we can't utilize the standard IO library. Fortunately, `print.c` defines some basic functions to handle I/Os by writing values to address 0x1000000 and leaves the job to `testbench.v`.

`multest.c` tests four kinds of RISC-V standard integer multiplication instruction, each with software or hardware implementation. MUL performs a 32-bit multiplication and places the lower 32 bits in the destination register. MULH, MULHU, and MULHSU perform the same multiplication but return the upper 32 bits of the full 64-bit product for signed-signed, unsigned-unsigned, and signed-unsigned multiplication, respectively.
When the compiler parses a multiplication operator "*****" in the program, it looks up the corresponding function routine in gcc software library to generate assembly code. On the other hand, hard_mul() is a wrapper for assembly instruction MUL. After picorv32 decodes it, the picorv32_pcpi_mul PCPI submodule is activated, and then the multiplication is performed by this hardware coprocessor.

Rerun the simulation, but this time we also generate a trace log and a waveform file

```
make pcpi_fsdb
```

To measure the latency of each instruction, we've inserted **tick()**s to get the current cycle count of CPU. The figure below is the final output. The first line displays the two integers we'd like to multiply. The following lines show the output value and the cycle count for each kind of multiplication. We can observe a substantial speedup with the hardware implementations.

```
input [FFFFFFFF] B11DDC17 [00000000] 59781258
            mul      mulh    mulhsu     mulhu
hard   258545E8  E46E61DC  E46E61DC  3DE67434
time        115       151       147       151
soft   258545E8  E46E61DC  E46E61DC  3DE67434
time        811      2232      2219      1470
```

Save the trace log in a readable format:

```
python3 showtrace.py testbench.trace firmware/firmware.elf > trace.txt
```

Now we can examine the hexadecimal assembly in `trace.txt`. The four entries in a row are the destination, current program counter, hexadecimal instruction, and decoded assembly instruction, from left to right. The destination field starts with a symbol indicating the value type of subsequent digits. More precisely, ">" means "branch target"; "@" means "load, store destination address"; "=" means "ALU or register output".

```
dest.     | PC       | hex inst.| inst.
>00000a68 | 0000046c | 5fc000ef | jal ra,a68 <multest>
=002f3f50 | 00000a68 |     7171 | addi sp,sp,-176
@002f3ffc | 00000a6a |     d706 | sw ra,172(sp)
=00000470 | 00000a6a |     d706 | sw ra,172(sp)
```

Memory Interface

In this assignment, you need to design a specific accelerator to do DNN calculation. In the test bench, both CPU and accelerator are connected to the same memory through PicoRV32's native memory interface, a valid-ready interface that can run one memory transfer at a time. Refer to the GitHub pages of PicoRV32 for further information. In practice, the accelerator can be controlled with either PCPI or MMAP. Both of them will be introduced in the following sections.

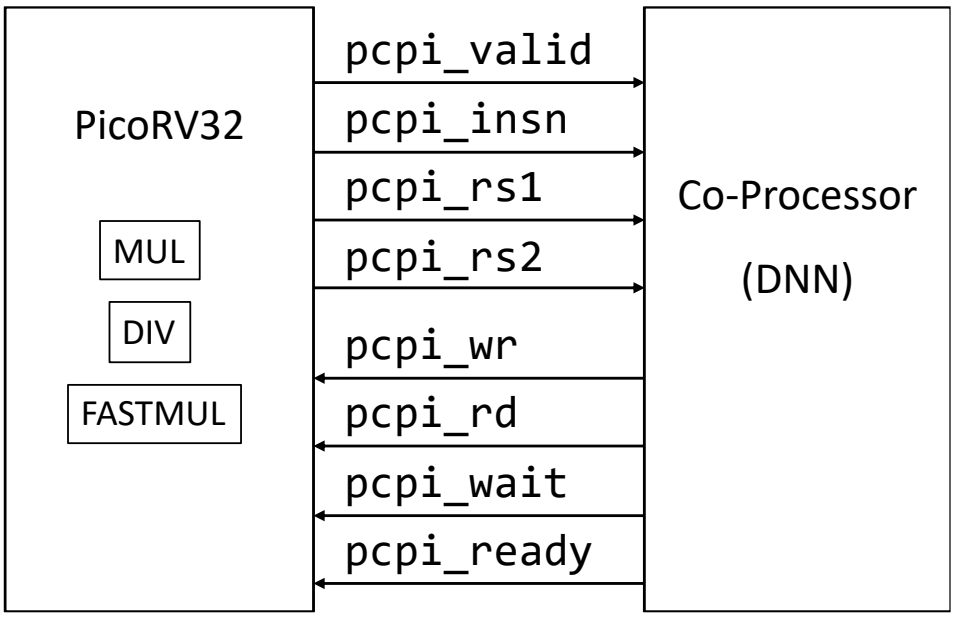
Memory Layout

Some memory regions are reserved for special purposes. Their relationship is summarized in the table below:

Address Range	Memory Region Description
0x00000000~ 0x0000FFFF	Text region for our program
0x00BF0000~ 0x00C00000	Stack region; Stack pointer points to the end of the memory initially
0x10000000	The remaining addresses can be used for memory mapped I/Os. For instance, 0x1000000 is specialized for the console output
0x20000000	`start.S` writes here to test the correctness of programs
The following addresses is needed only for MMAP (Method 2)!	
0x40000000	DNN_MMAP_READ_FINISH: read the finish signal from DNN core
0x40000004	DNN_MMAP_WRITE_CONV: write the DNN convolution flag
0x40000008	DNN_MMAP_WRITE_N: write the DNN input/output batch size
0x4000000c	DNN_MMAP_WRITE_C: write the DNN input channel
0x40000010	DNN_MMAP_WRITE_H: write the DNN input height
0x40000014	DNN_MMAP_WRITE_W: write the DNN input width
0x40000018	DNN_MMAP_WRITE_R: write the DNN weight height
0x4000001c	DNN_MMAP_WRITE_S: write the DNN weight width
0x40000020	DNN_MMAP_WRITE_M: write the DNN output channel
0x40000024	DNN_MMAP_WRITE_P: write the DNN output height
0x40000028	DNN_MMAP_WRITE_Q: write the DNN output width
0x4000002c	DNN_MMAP_INPUT_OFFSET: write the input data location as an offset
0x40000030	DNN_MMAP_WEIGHT_OFFSET: write the weight data location as an offset
0x40000034	DNN_MMAP_OUTPUT_OFFSET: write the output data location as an offset
0x40000038	DNN_MMAP_WRITE_START: write anything here to start calculation

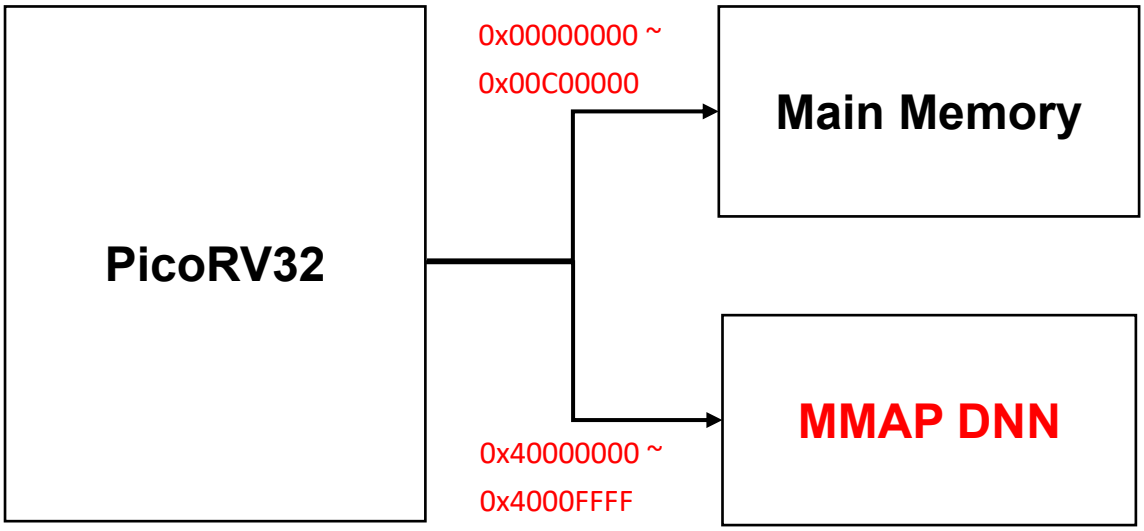
Method 1: PCPI

The Pico Co-Processor Interface (PCPI) can be used to implement non-branching instructions in the external cores. For each signal, refer to the GitHub pages of PicoRV32. PCPI cores can be connected inside or outside the CPU module. Three PCPI submodules have already been implemented in `picorv32.v`. Trace their code carefully.



Method 2: MMAP

Devices that support MMAP can be controlled by Read/Write signals. Reading/Writing to specific memory addresses would be treated as control signals for register configuration or I/Os. For instance, writing to 0x40000038 can trigger the calculation of our DNN module in the diagram below.



Run the simulation for MMAP example:

Comment out ENABLE_DNN_PCPI and ENABLE_MULTST; uncomment ENABLE_DNN_MMAP in `start.S`; then execute the following command:

```
make mmap
```

Gate-level Simulation

We provide the synthesis script, `syn_script.dc`, which may make the synthesis easier. Besides, the file "header.v" defines the memory delay for the gate-level simulation. The default memory delay is 0. You can modify these two files if necessary.

Use Design Compiler to synthesize your DNN accelerator. Then perform the gate-level simulation with the following command.
Note: it will take longer time than the RTL simulation.

```
make mmap_syn
```

If you want to have the waveform for the gate-level simulation, use the following command instead:

```
make mmap_syn_fsdb
```

Requirements

1. Go through all the files, study the purposes of each file. Follow the **TODO** marks to complete the assignment.
2. You should complete `firmware/dnn_mmap.c` first for the software part; then complete `dnn_mmap.v` for the hardware part.
3. Implement the software version of the unfold, convolution, and fully connected layers in `dnn_mmap.c`.
4. Access the MMAP address region in `firmware/dnn_mmap.c` to control the DNN accelerator. You can add additional addresses if necessary.
5. Integrate the systolic hardware in Homework #3 as the DNN accelerator in `dnn_mmap.v`. The necessary signals have been declared for you. You should add your own design and connect it to the DNN module. You need to handle the memory interface for the DNN module and verify the computational results with the golden data in Homework #3 by yourself.
6. Profile your program with the tick function. Refer to the template for the usage of the tick function between the computational functions.
7. Synthesize your accelerator and check the results of the entire system (i.e., software, PicoRV32's platform in RTL and functional model, and the gate-level DNN accelerator).

Questions & Discussion

1. Compare the software and hardware versions based on the profiling.
2. Discuss the possible improvement over the system, e.g., the memory design, interface, or even your own DNN engine.
3. Given a specific application, under what circumstances does its software version outperforms its hardware version?
4. Roughly describe the hardware architecture of PicoRV32 (`picorv32.v`).
5. Optional: Discuss anything interesting you've discovered

Submission

1. Submit the PDF report according to the questions. Include the plots in your write-up. Use the following filename:
   `hw4_YourStudentID.pdf`
2. Use the report template, which consists of Design Concept, Simulation and Discussion, and Summary.
3. Also, hand in your source codes with the following filenames:
   `dnn_mmap.v dnn_mmap.c syn_script.dc dnn_syn.v dnn_syn.sdf`
   Note:
   - ✓ You may hand in any additional files if necessary. State the reason clearly and make a file list.
   - ✓ You should detail how to execute the Verilog simulation in the report explicitly. Also, put proper header/comments in the source codes.