

HW 05: RISC-V with GCD Accelerator

Goals

In this assignment, you will implement GCD in Verilog & C. You should partition the application into hardware and software parts. For the hardware part, you can either wrap your modules with **Pico Co-Processor Interface (PCPI)** or use **memory-mapped I/O (MMAP)** method. **In this homework, you are going to practice both.** The software part is responsible for controlling the whole system. It will be run on PicoRV32, a CPU core that implements RISC-V ISA. We provide tools for you to profile the program and see how this architecture benefits the performance.

References

Picorv32: <https://github.com/cliffordwolf/picorv32> **MUST READ!!!**

RISC-V: <https://riscv.org/specifications/>

Setup

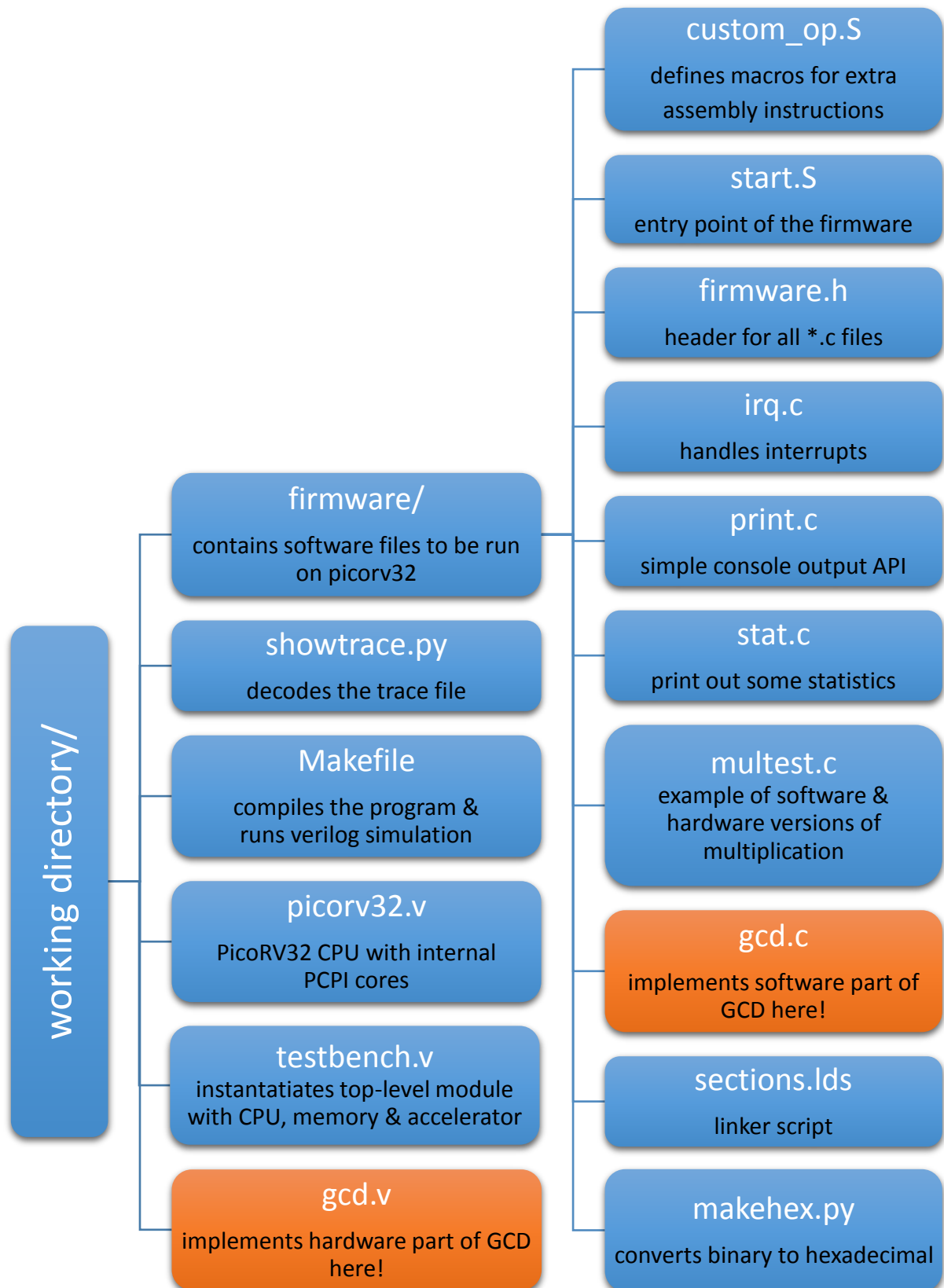
1. We have built RISC-V cross-compiler toolchain for you on **ic54~58** under:
/users/course/2017F/cs412500/tools/riscv32.
Login only to these machines.
2. Download source code from iLMS
3. Under directory picorv32, compile the program and then run Verilog simulation by:

```
make pcpi
```

4. You should see the following output:

```
-----  
EBREAK instruction at 0x000006C4  
pc 000006C7  x8 00000000  x16 7C235965  x24 00000003  
x1 00000694  x9 00000000  x17 84A97420  x25 00000001  
x2 00010000  x10 20000000  x18 00000002  x26 00009002  
x3 DEADBEEF  x11 2000AD00  x19 00000160  x27 00009002  
x4 DEADBEEF  x12 0000004F  x20 00000000  x28 00000001  
x5 00000F42  x13 0000004E  x21 00000015  x29 7C235965  
x6 00000008  x14 00000045  x22 000006C4  x30 00000000  
x7 00000000  x15 000001E0  x23 000006C4  x31 000000C4  
-----  
Number of fast external IRQs counted: 0  
Number of slow external IRQs counted: 0  
Number of timer IRQs counted: 20  
TRAP after 317614 clock cycles  
ALL TESTS PASSED.
```

Directory Structure



※The orange blocks are the files you need to modify!

Detailed Explanation of 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. If you don't want to run all of the functions, undefine the corresponding macros at the top of the file.

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

multest.c tests 4 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 sees a "*" operator, it looks up multiplication function routine in gcc software library and generates assembly code. On the other hand, hard_mul() is just a wrapper for assembly instruction MUL. After picorv32 decodes it, the picorv32_pcp_i_mul PCPI submodule is activated, and then the multiplication is performed by this hardware accelerator.

Run the simulation again, but this time we also generate a trace log & a waveform file

```
make pcp_i_fsdb
```

To measure the latency of each instruction, we've inserted **tick()**s to get current cycle count of the 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 cycle count for each kind of multiplication. We could observe a substantial speedup with hardware implementation.

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 readable format

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

Now we can examine the generated hexadecimal assembly code easily. The 4 columns from left to right are destination, current program counter, hexadecimal instruction, and decoded assembly instruction. The destination field starts with a symbol indicating the value type of subsequent digits. More precisely, ">" means "branch target", "@" means "load, store destination address" and "=" 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 GCD calculation. In the test bench, both CPU and accelerator are connected to the same memory through PicoRV32's native memory interface, which is a simple valid-ready interface that can run one memory transfer at a time. You could refer to the [GitHub pages](#) of PicoRV32 for further information. In practice, the accelerator could be controlled with either PCPI (Method 1) or MMAP (Method 2). Both of them will be introduced in the following sections.

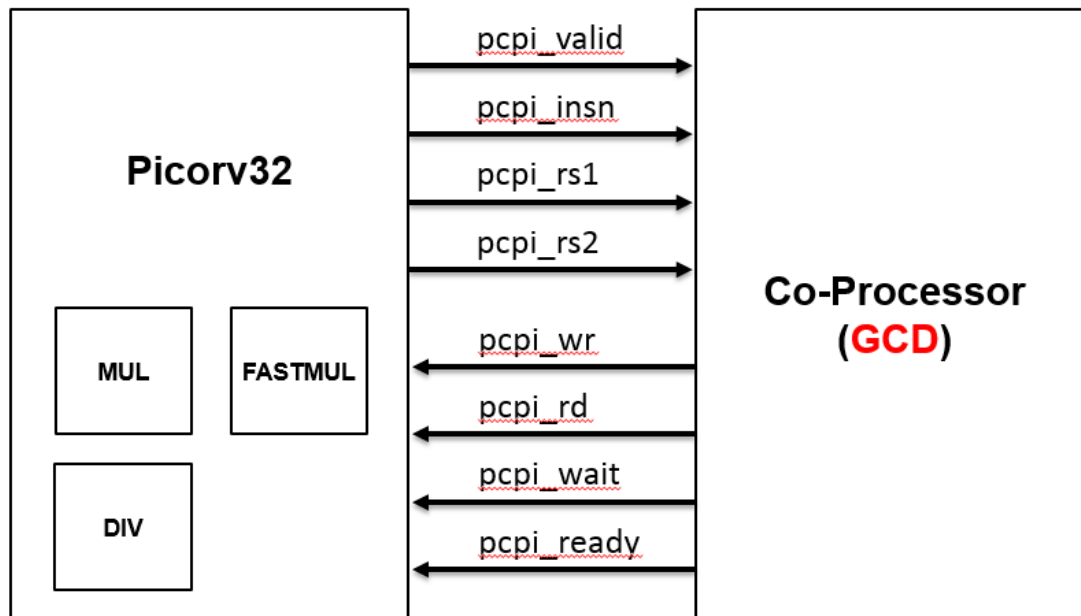
Memory Layout

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 initially points to the end of the memory
0x10000000	We can use remained memory address for memory mapped I/O. For instance, 0x10000000 is specialized for console output
0x20000000	start.S writes to this address in order to test program correctness
The following addresses is needed only for MMAP (Method 2)!	
0x40000000	GCD_MMAP_READ_STATUS: read the status of GCD core
0x40000004	GCD_MMAP_READ_Y: read the GCD output
0x40000008	GCD_MMAP_WRITE_A: write first integer
0x4000000c	GCD_MMAP_WRITE_B: write second integer
0x40000010	GCD_MMAP_START: write anything here to start calculation

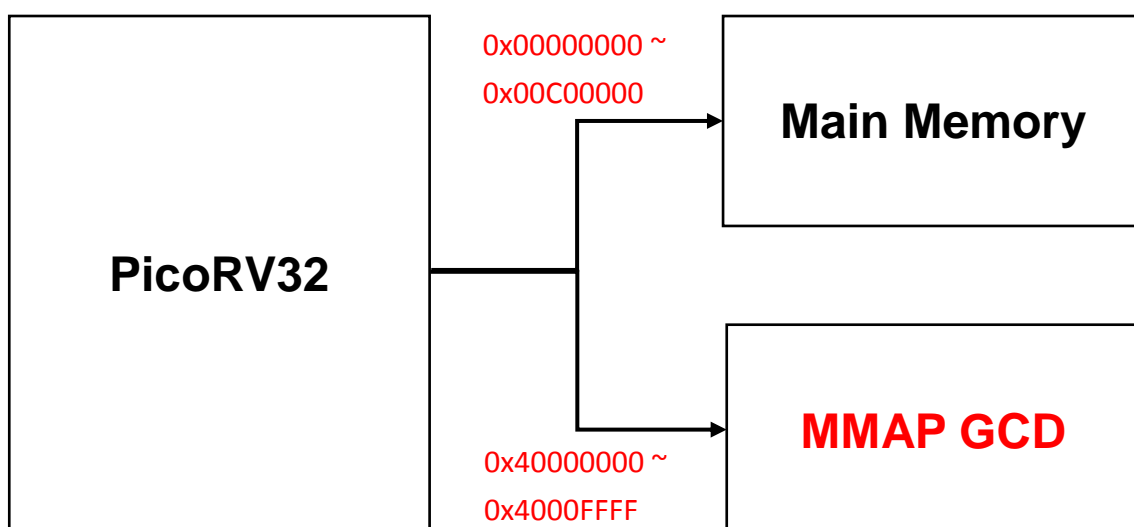
Method 1: PCPI

The Pico Co-Processor Interface (PCPI) can be used to implement non-branching instructions in external cores. For the purpose of 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 and finish your own PCPI core in `pcpi.v`.



Method 2: MMAP

Devices that supports MMAP can be controlled by Read/Write signals. Reading/Writing to specific memory address will be treated as the control signals for configuring the registers or performing I/Os. For instance, writing to `0x40000010` could trigger the calculation of our GCD module in the diagram below.



Run the simulation of MMAP example:

Comment out `ENABLE_GCD_PCPI`, uncomment `ENABLE_GCD_MMAP` in `start.S`. And then run

```
make mmap
```

Problem Description

Euclidean Algorithm Using Subtraction Only: <http://www.naturalnumbers.org/EuclidSubtract.html>

Working Items

1. Go through all the files, study the purposes of each file. Follow the **TODO** marks to complete the program
2. Implement hardware version of GCD in `gcd_pcpi.v` or `gcd_mmap.v`. The required signals have already been declared for you. But you should design your own state machine. The GCD module takes two 32-bit integers as inputs. Use successive subtraction method to obtain the final result.
3. (For those using **Method 1**) Use the customized instruction `hard_gcd()` in `firmware/gcd_pcpi.c` to initiate the GCD accelerator.
4. (For those using **Method 2**) Access the MMAP address region in `firmware/gcd_mmap.c` to control the GCD accelerator.
5. Implement software version of GCD in `gcd_pcpi.c` or `gcd_mmap.c` for comparison.
6. Profile your program.

Questions & Discussion

1. What are the advantages of running applications on a hardware accelerator?
2. Which version scales better if the input values to the GCD are getting larger?
3. Given a specific application, under what circumstances does its software version outperform its hardware version?
4. Roughly describe the hardware architecture of PicoRV32 (`picorv32.v`).
5. Optional:
 - i. PicoRV32 has several Verilog module parameters (e.g., `FAST_MEMORY`, etc.). Try different combinations of them and explain their functionality.
 - ii. Discuss anything interesting you've discovered