CA Final Project

Due: 23:59 on 12/20

---

1 Introduction

In this final project, you need to implement a single-cycle CPU on your own. The architecture of the single cycle CPU is in Fig. 1. Note that this is not the complete architecture since you also need to implement instructions such as `jal` or `jalr`. You should be able to find the design details in the lecture slide.
Here are some goals that we hope you can learn from this final project:

- Implement a single-cycle CPU.
- Add multiplication/division unit (mulDiv) to CPU (HW2).
- Handle multi-cycle operations.
- Get more familiar with assembly and Verilog.

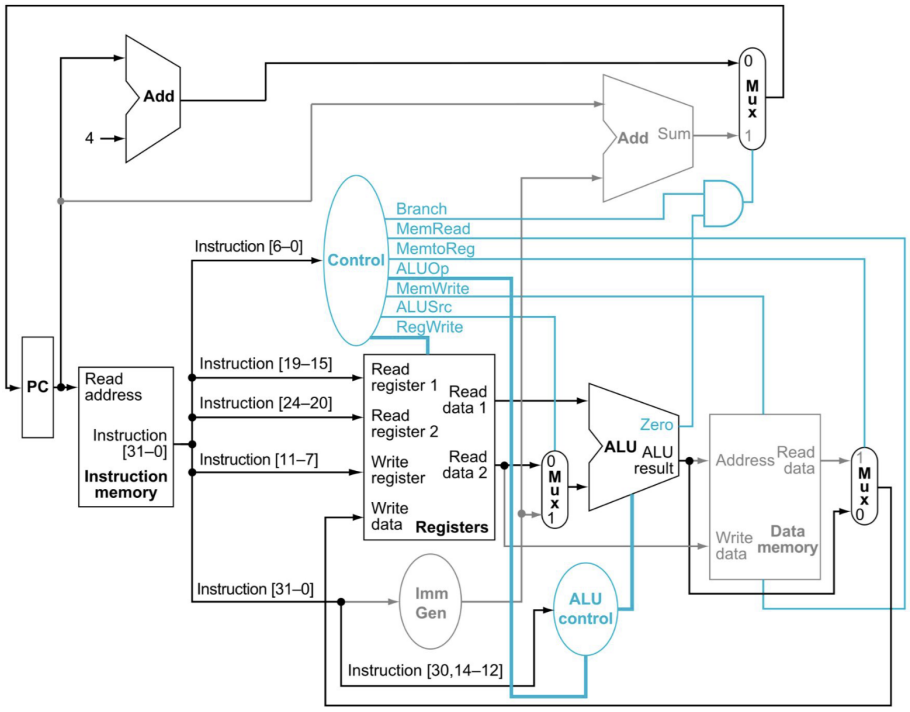


Figure 1: architecture

2 Design Specification

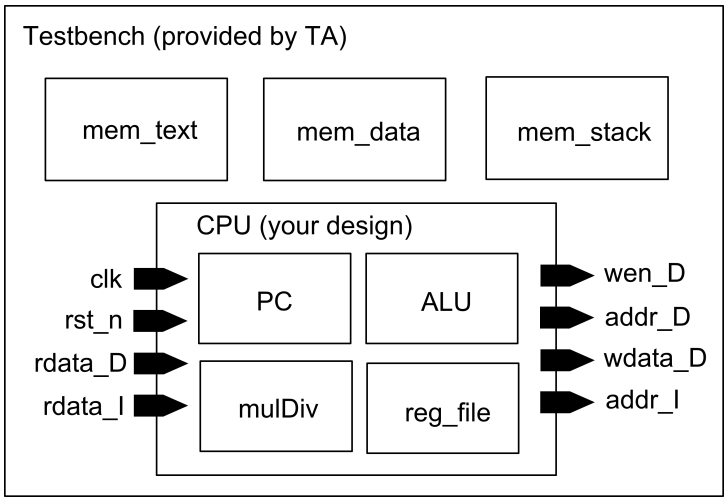


Figure 2: Spec.

2.1 Port Definitions

Name	I/O	Width	Description
clk	I	1	Positive edge-trigger clock
rst_n	I	1	Asynchronous negative edge reset
wen_D	O	1	0: read, 1: write, from data/stack memory
addr_D	O	32	Address of data/stack memory
rdata_D	I	32	Data read from data/stack memory
wdata_D	O	32	Data to be write to data/stack memory
addr_I	O	32	Address of instruction memory
rdata_I	I	32	Data read from instruction memory

2.2 Needed instructions

You need at least the following instructions to pass all the patterns.

- `lw, sw`
- `add, sub, addi, slli, srli, srai, slti`
- `beq, bne, bge, blt, jal, jalr, auipc, lui`
- `mul, divu, remu` (Use your mulDiv in HW2)

2.2.1 Supplement: Instruction "auipc"

op	funct3	funct7	Type	Instruction	Description	Operation
0010111 (23)	-	-	U	`auipc rd, upimm`	add upper immediate to PC	`rd = {upimm, 12'b0} + PC`

- Add upper immediate to PC, and store the result to rd.
- Example: *auipc* x5, 1 (PC = 0x0001001c)
- Store 0x0001001c + 0x00001000(U-immediate left-shift by 12 bits) into x5.

2.2.2 Supplement: Instruction "mul", "divu", "remu"

op	funct3	funct7	Type	Instruction	Description	Operation
0110011 (51)	000	0000001	R	`mul rd, rs1, rs2`	multiply	$rd = (rs1 * rs2)_{31:0}$
0110011 (51)	101	0000001	R	`divu rd, rs1, rs2`	divide unsigned	`rd = rs1 / rs2`
0110011 (51)	111	0000001	R	`remu rd, rs1, rs2`	remainder unsigned	`rd = rs1 % rs2`

- Your mulDiv in HW2 can support these instructions.
- However, these instructions are multi-cycle instructions.
  - Once CPU decodes mul, divu or remu operations, issue valid to your mulDiv.
  - Once CPU receives ready from mulDiv after 32 cycles, store the 32-bits result to rd.
  - You might have to design FSM in your CPU.

See the provided **instruction_set.pdf** for more details.

3 Relate Memory to Testbench

In Jupiter simulator, the memory layout is shown in Fig. 3.

- Text: Store the program code.
- Data: Store the variables, arrays, etc.
- Stack: Automatic storage.

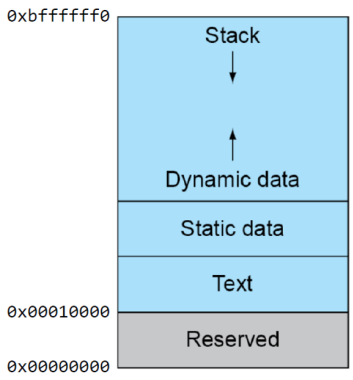


Figure 3: Memory

In this project, the block diagram of each modules is shown in Fig. 4

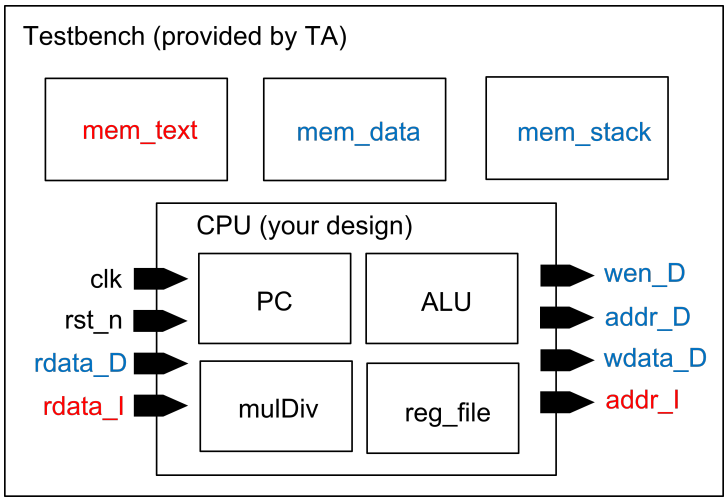


Figure 4: Block diagram

In the testbench provided by TA, we instantiated 3 memory blocks to mimic the behavior of the whole memory (one memory block) in the Jupiter simulator, as shown in Fig. 3. When we start the simulation, testbench will load the instruction code and the input variables into mem_text and mem_data.
The designed single-cycle CPU should

- Read instruction by setting the output addr_I and get the instruction from input rdata_I.
- Write or read data to stack or memory by setting the output wen_D, addr_D and wdata_D, and get the data from input rdata_D.

Here are some notes about the provided testbench:

- We define offset address for each memory block so the address you see in each testcase will be similar to the one you see in Jupiter. (Don't need to modify it yourself.)
- The provided *memory.v* is written in not synthesizable coding style just to show the behavior of the memory when simulating.
- You can check and compare the values in the memory blocks by nWave and by Jupiter simulator to debug.

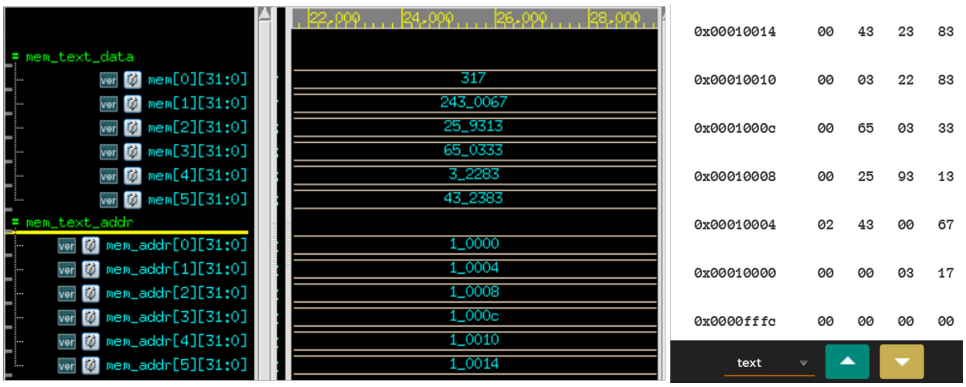


Figure 5: Text memory (to store instrcution)

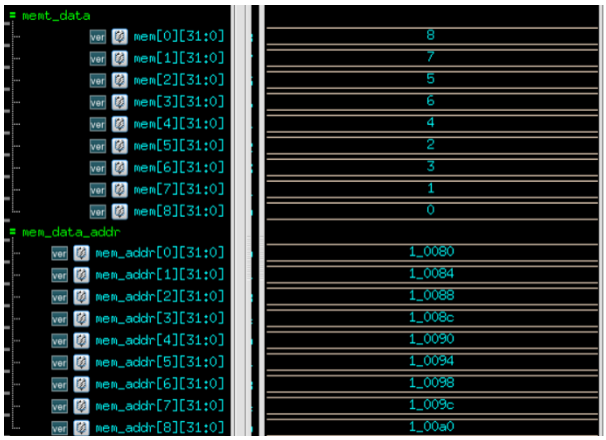


Figure 6: Data memory (to store input variables and output)

4 Public test pattern

There are three public test patterns in the directories **arithmetic**, **sort**, and **gcd**. Each of them contains 4 files:

- *_text.txt: The machine code (instruction) that will be loaded into the text memory in testbench for your CPU to fetch and decode.
- *_data.txt: The content of it will be loaded into the data memory in testbench. Your CPU will load the inputs from it according to the machine code.
- *_data_ans.txt: The correct final content in the data memory. The testbench will compare the content of your data memory and this file to check your design.
- *.s: The original assembly code of the test patterns.

4.1 Arithmetic operation

- This test pattern tests most of the arithmetic operations you will use in all test patterns.
- `add, sub, mul, srli, srai, slti, slli, remu, divu...`
- Handle the muti-cycle operations `mul, remu` and `divu` carefully.

4.2 Sorting

- Modified from the lecture slide.
- The provided machine code will sort the input {8, 7, 5, 6, 4, 2, 3, 1} into {1, 2, 3, 4, 5, 6, 7, 8}.

4.3 GCD

- Modified from HW1.
- The provided machine code will calculate gcd(x,y), a, b, z (inv (x mod y)) as you did in HW1.
- 2 groups of inputs {114, 514} and {1919, 810}will be loaded from data memory by the provided machine code.

5 Simulation

- To run the Verilog simulation you should use the following code

```
$ vcs testbench.v -full64 -R -debug_access+all +define+arithmetic
```

- change `+define+arithmetic` to `+define+sort` or `+define+gcd` to test other public patterns.

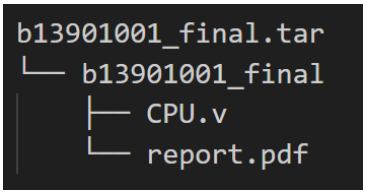
6 Report

- Describe how you designed your CPU architecture
  - Describe how you design the data path of instructions not referred in the lecture slides (jal, jalr, auipc, . . .)
  - Describe how you handle multi cycle instructions (mul, divu, remu)
- Snapshot the Register table using **Design Compiler**
- Describe your observation

7 Submission

- Deadline: 23:59 on Dec. 20, 2024
  - No late submission
- Upload [student_id]_final.tar to NTU COOOL
  - Wrong file or folder format will get -1 point as penalty
  - Compress your files with the following sinstruction

```
$ tar -cvf b13901001_final.tar b13901001_final/
```



8 Grading

- Each pattern (3 public + 3 hidden) worth 3 points.
- Report worth 2 points
- Brute force solution is not allowed
- Using GPT to generate your code is not allowed
- No plagiarism