Computer Architecture 2025 Fall Lab 3

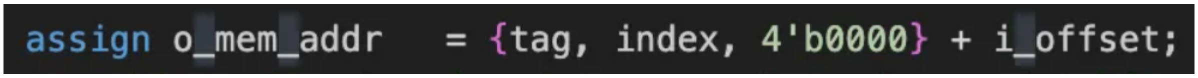
1. Problem Description

- In this lab, you're going to **implement a cache for** a single-cycle CPU.
- CPU features:
  - 32 registers
  - Input: 32-bit binary instructions
- Area Limitation (**Do not be concerned with this constraint.**)
  - CPU: 12000 mm^2(In HW1, the maximum difference in area between the TA's result and any student submission is 1000. TA implemented Lab 3's CPU with 9800 mm^2)
  - Cache: 20000 mm^2

1.1 Data Path

- Figure 1 describes the CPU data path.
  - **Provided by TAs: Registers and Memory (written in TB)**
  - The Data Memory has read/write latency; you are asked to implement a cache to accelerate it.
- The single-cycle CPU architecture builds upon the HW1 implementation. However, you are required to support only the functionality explicitly covered in the provided test cases.
  - **R-type:** ADD, SUB, AND, XOR, MUL
  - **I-type:** ADDI, SLLI, SLTI, SRAI, LW, JALR
  - **S-type:** SW

  - **B-type:** BEQ, BNE, BLT, BGE
  - **U-type:** AUIPC
  - **J-type:** JAL
  - **System:** ECALL (Triggers `o_finish, 00000073)`
- Data memory
  - Data memory has a read/write latency of 10 cycles.
  - Ensure the **Read/Write (R/W)** control signals for the data memory are held stable until the data is successfully returned.
  - A block in data memory is 128 bits, containing four 32-bit words (or "distinct data") per request.
  - The last 4 bits of the data memory address are structured as: **2 bits for the block offset and 2 bits for the byte offset.** (**Note: You may set all 4 bits to 0 in `cache.v` when addressing the main memory.**)



```
assign o_mem_addr   = {tag, index, 4'b0000} + i_offset;
```

- Cache
  - Used to accelerate **Data Memory**
  - Check the TA's implementation in the reference.
  - Be careful when handling the processor address. Before performing any address operations on `i_proc_addr`, **subtract** `i_offset`. Before outputting to the data memory (`o_DMEM_addr`), **add** the offset back.

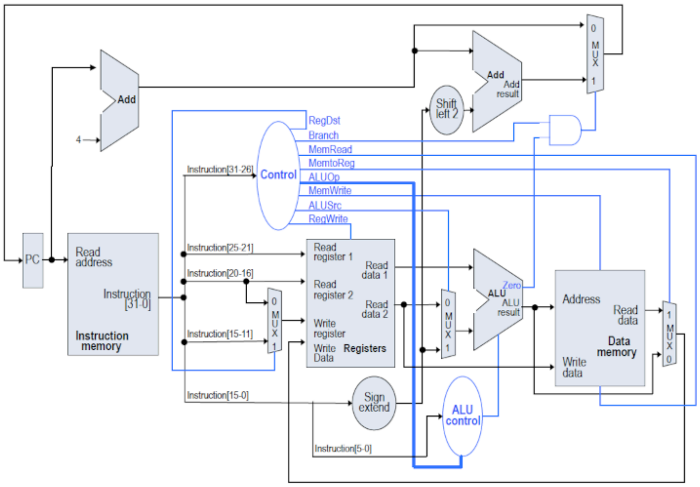


Figure 1: Referenced Single Cycle CPU

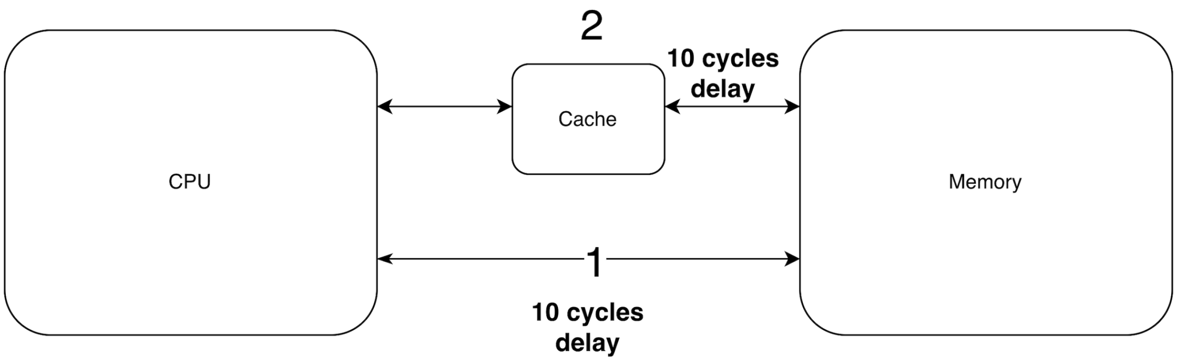
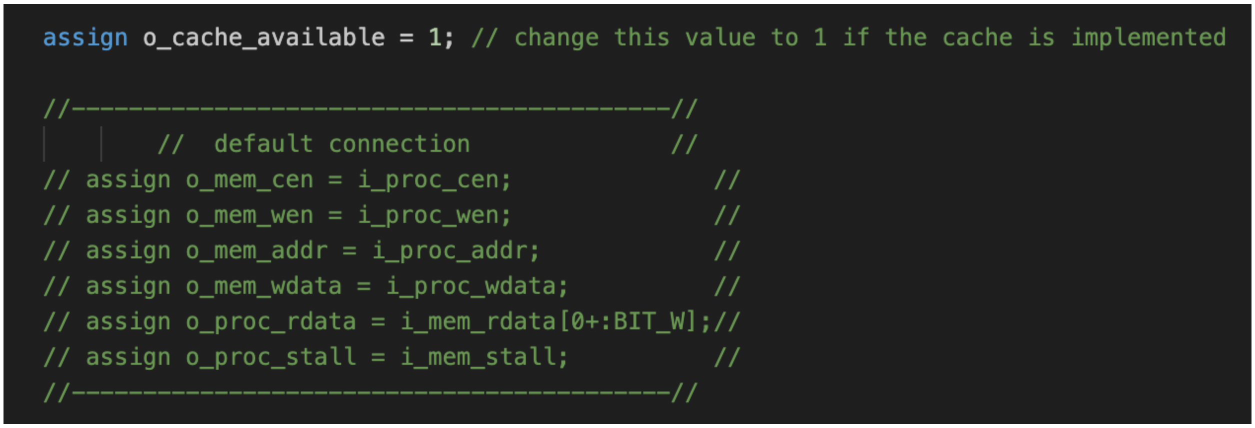


Figure 2: Memory Hierarchy and Delay

1.2 Cache

- You can choose any cache implementation
  - Write through, Write back
  - Write around

Modify the value `o_cache_available` **to 1** in your implementation once cache is implemented.



```
assign o_cache_available = 1; // change this value to 1 if the cache is implemented

//------------------------------------------//
        //  default connection              //
// assign o_mem_cen = i_proc_cen;              //
// assign o_mem_wen = i_proc_wen;              //
// assign o_mem_addr = i_proc_addr;            //
// assign o_mem_wdata = i_proc_wdata;          //
// assign o_proc_rdata = i_mem_rdata[0+:BIT_W];//
// assign o_proc_stall = i_mem_stall;          //
//------------------------------------------//
```

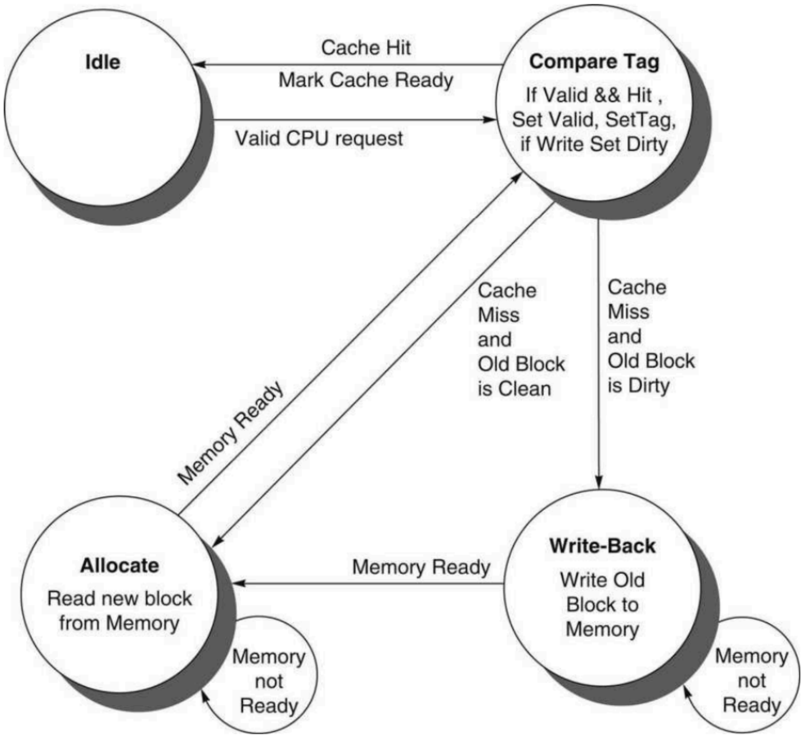


FIGURE 5.39 **Four states of the simple controller.**

Easy FSM for the cache implementation

1.3 Input / Output Data

- Provided: `testbench.v`, mem_I.dat, mem_D.dat, golden.dat, mem_I_listing (In I3, I4)
- All test cases should finish in 10000 cycles.

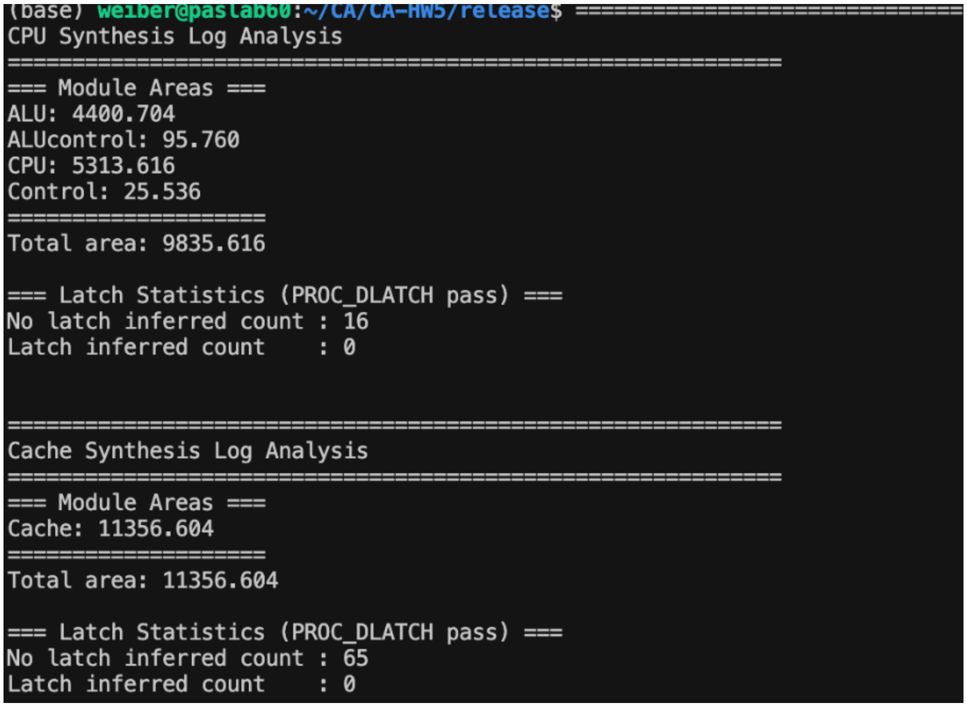
- You may change the test pattern by executing Python in the "./code/Gen_pattern"
    - I1: factorial
    - I2: arithmetic operations
    - I3: insertion sort
    - I4: bubble sort
- All assembly code finishes with **ECALL** (00000073)

1.4 Required Modules

- **CPU (single cycle)**
- Cache
- **Others**
    - You may add extra helper modules if needed.

1.5 Synthesizability and Area

- All RTL code must be synthesizable (no unintended latches).
- The TA will check your design using **yosys** (`synth.log`). You do not need to modify any yosys scripts. The results can be found in Section 10.6 of `./log/synth.log`, including the estimated area of your module.
- Run parse.py to get area and latch results for CPU and cache, paste the results, and fill in the numbers in the report



```
(base) weiber@paslab60:~/CA/CA-HW5/release$ ==============================
CPU Synthesis Log Analysis
============================================================
=== Module Areas ===
ALU: 4400.704
ALUcontrol: 95.760
CPU: 5313.616
Control: 25.536
====================
Total area: 9835.616

=== Latch Statistics (PROC_DLATCH pass) ===
No latch inferred count : 16
Latch inferred count    : 0


============================================================
Cache Synthesis Log Analysis
============================================================
=== Module Areas ===
Cache: 11356.604
====================
Total area: 11356.604

=== Latch Statistics (PROC_DLATCH pass) ===
No latch inferred count : 65
Latch inferred count    : 0
```

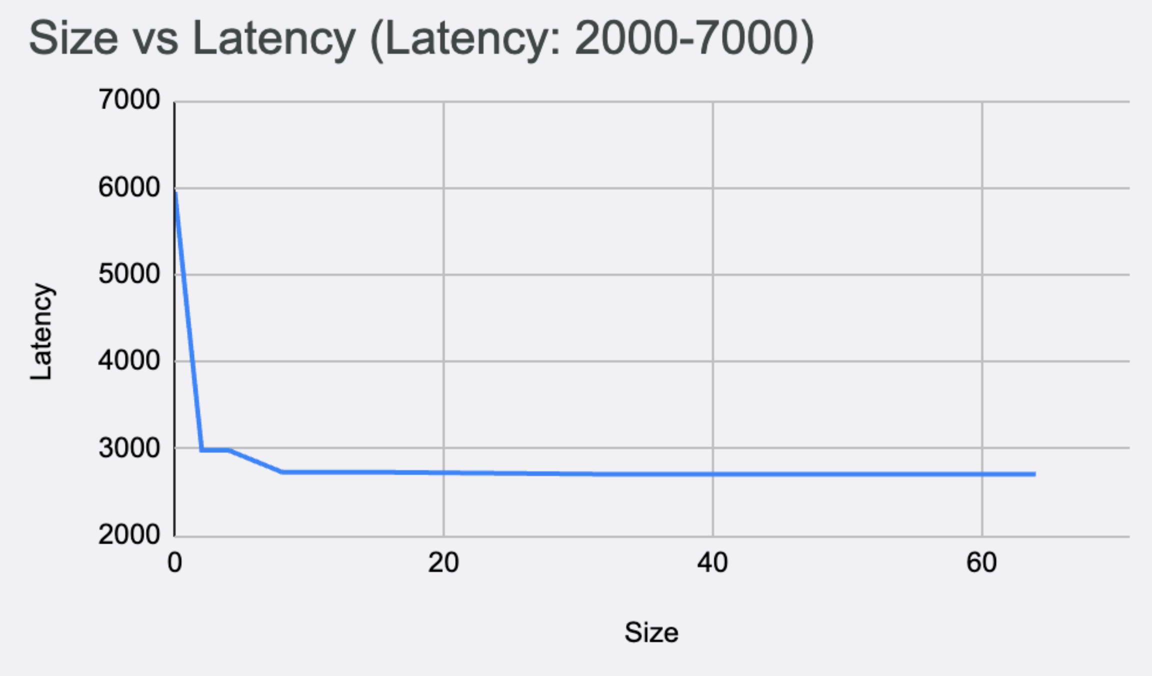
1.6 Testcases

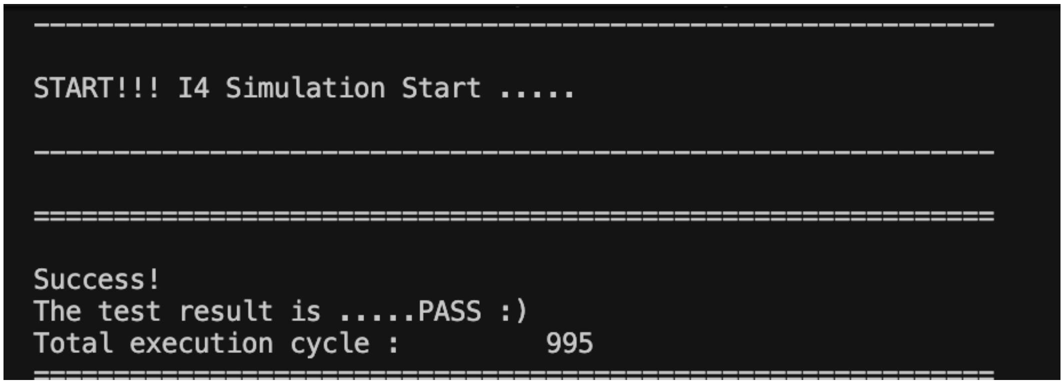
- TAs will test your modules with public and hidden patterns.
- Create your own test pattern through the provided Python!

2.1 Report

- **Explain your CPU and Cache.**
- The report also includes:
    - Synthesizability / latch check screenshot.
    - Development environment (OS, compiler, IDE).
- **Pareto figure** for cache size versus execution cycles (Draw the figure with the **I4 test case**).
    - Clearly specify the **number of different cache sizes** tested.
        - You must evaluate **at least four distinct cache sizes.**
        - Paste the “pass for simulation results” for each cache size. (see below)
    - You don’t need to use a public testcase if the figure is not clear enough.

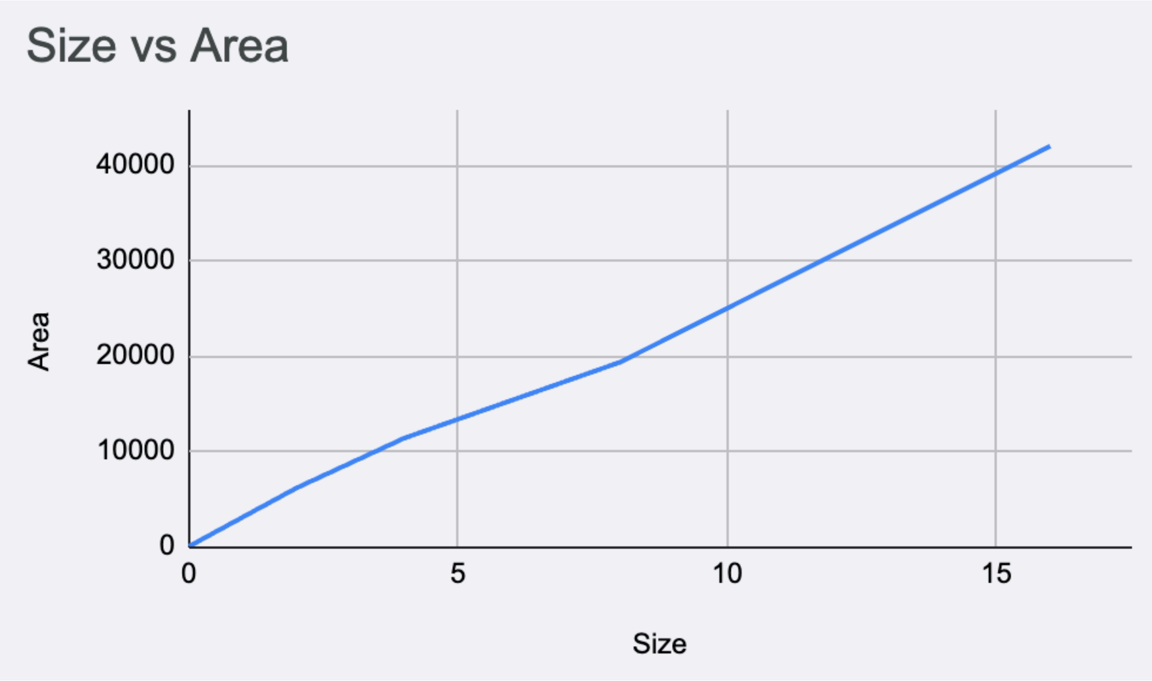
    - TA reserves the right to assess and grade based on the quality of the figures, but you can add a text description for clarity.





```
------------------------------------------------------------

START!!! I4 Simulation Start .....

------------------------------------------------------------

============================================================

Success!
The test result is .....PASS :)
Total execution cycle :         995
============================================================
```

- Figure for cache size versus cache area
    - TA reserves the right to assess and grade based on the quality of the figures.



3.1 Development Environment

Requirement

- **Docker** is required.

Directory Structure (after unzip)

```
Lab3/
├── code
│   ├── Gen_Pattern
│   │   ├── I1_fact_gen.py
│   │   ├── I2_hw1_gen.py
│   │   ├── I3_insertion_sort_gen.py
│   │   └── I4_bubble_sort_gen.py
│   ├── Pattern
│   │   ├── I1
│   │   │   ├── golden.dat
│   │   │   ├── mem_D.dat
│   │   │   └── mem_I.dat
│   │   ├── I2
│   │   │   ├── golden.dat
│   │   │   ├── mem_D.dat
│   │   │   └── mem_I.dat
│   │   ├── I3
│   │   │   ├── golden.dat
│   │   │   ├── mem_D.dat
│   │   │   ├── mem_I.dat
│   │   │   └── mem_I_listing.txt
│   │   └── I4
│   │       ├── golden.dat
│   │       ├── mem_D.dat
│   │       ├── mem_I.dat
│   │       └── mem_I_listing.txt
│   ├── src
│   │   ├── cache.v
│   │   └── CHIP.v
│   ├── supplied
│   │   ├── memory.v
│   │   └── Regfile.v
│   └── tb
│       └── tb.v
├── docker-compose.yml
├── dockerfile
├── judge.yaml
├── log
│   ├── cache_syn.log
│   ├── cpu_syn.log
│   ├── output_1.txt
│   ├── output_2.txt
│   ├── output_3.txt
│   └── output_4.txt
├── Makefile
├── parse.py   # check for latch and area
└── run_all.sh # use for simulation
```

💡 **You should modify files in** `code/src/`

3.2 Run with Docker

After implementation, execute:

```
sudo make run
```

- Results (logs, text, waveforms) will be generated under `log/`.

3.3 Run without Docker (alternative)

If you don’t have Docker, run on **Ubuntu 22.04** with **iverilog 11.0-1.1 & Yosys & nangate45**

```
# Simulation
./run_all.sh

# 合成
yosys -l <log::path>/cpu_syn.log -p "
    read_liberty -lib <lib::path>/NangateOpenCellLibrary_typical.lib;
    read_verilog  <source::path>/*.v;
    hierarchy -top CPU;
    proc; opt_clean;
    fsm; opt_clean;
    techmap; opt_clean;
    flatten CPU;
    dfflibmap -liberty <lib::path>/NangateOpenCellLibrary_typical.lib;
    abc -liberty <lib::path>/NangateOpenCellLibrary_typical.lib;
    stat -liberty <lib::path>/NangateOpenCellLibrary_typical.lib;
  "
yosys -l <log::path>/cache_syn.log -p "
    read_liberty -lib <lib::path>/NangateOpenCellLibrary_typical.lib;
    read_verilog <source::path>/cache.v;
    hierarchy -top Cache;
    proc; opt_clean;
    fsm; opt_clean;
    techmap; opt_clean;
    flatten Cache;
    dfflibmap -liberty <lib::path>/NangateOpenCellLibrary_typical.lib;
    abc -liberty <lib::path>/NangateOpenCellLibrary_typical.lib;
    stat -liberty <lib::path>/NangateOpenCellLibrary_typical.lib;
  "
# area results
python parse.py
```

⚠️ **Note:** You will get **0 points** if your code cannot run correctly on the official environment

4.1 Submission Rules

- Directory structure after unzipping:

  ```
  studentID_lab3/
    ├── src/   (all Verilog codes you wrote)
    ├── studentID_lab2_report.pdf
  ```

- Do **not** include: tb.v, memory.v, Regfile.v
- File name: `studentID_lab3.zip`.
- Submit via NTU COOL.
- Deadline: 2025/12/9 (Tue.) 23:59.

5.1 Evaluation Criteria

- **Pass Testcase (40%)**
  - Public testcases: 20% (4 × 5%)
  - Hidden testcases: 20% (4 × 5%)
- **Performance (30%)**
  - Baseline (20%) : Total execution cycles in public testcases (I1~I4) less than 5000 (implement cache)

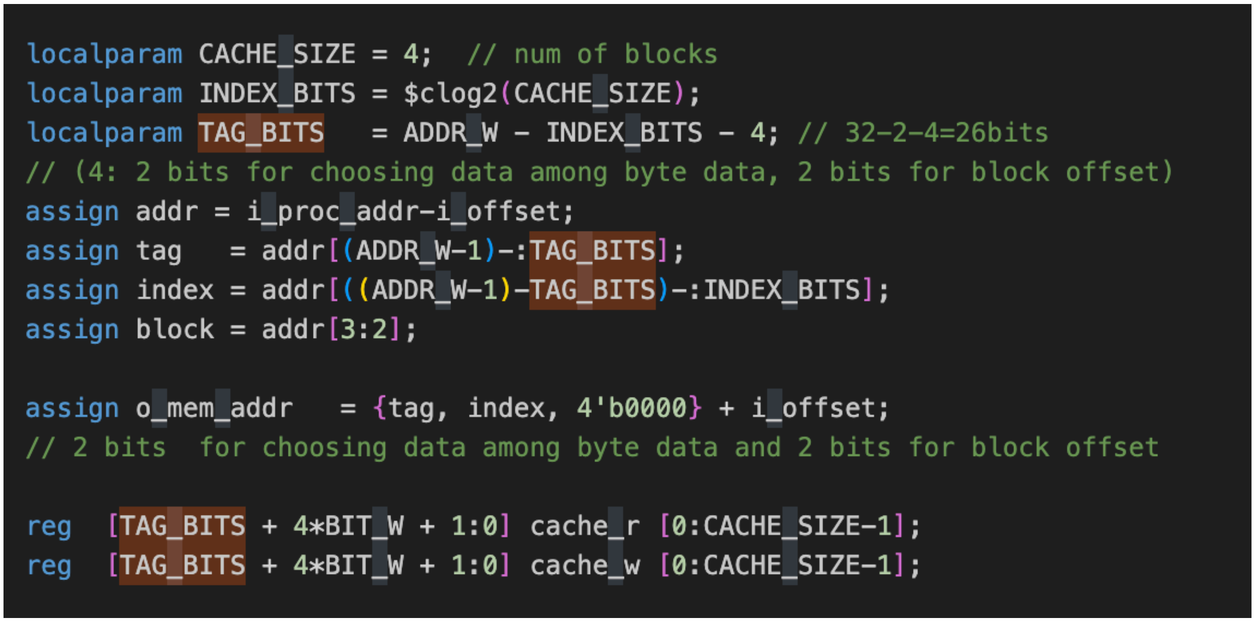
    - Ranking (10%): Shortest total execution cycles
        - You get 0 in ranking once:
            - late submission
            - violate area limitations
            - have latch violations
            - the design is unsynthesizable
            - compile error
        - Score with linear ranking
- **Report (30%)**
    - Description (10%)
        - Describe your CPU and Cache design
        - If you use LLM, clearlys state which parts you use.
        - We take Plagiarism seriously.
    - **Pareto** and area diagram: 10% (5% for each)
    - Demo 10 %
- **Bonus** (20%)
    - TA strongly recommends a novel or customized design of the cache. If you do exceptionally well on this, no worry about the ranking score.
    - If you implement the design from paper or some complicated cache, feel free to write in the report!
    - Judged by the TA, however, implementing the cache mentioned in the class won’t get any bonuses.

- **Other penalties**:
    - Compilation error (naming, or minor errors in 3 lines): −5%
    - Submitting unnecessary files (tb or supplied): -5%

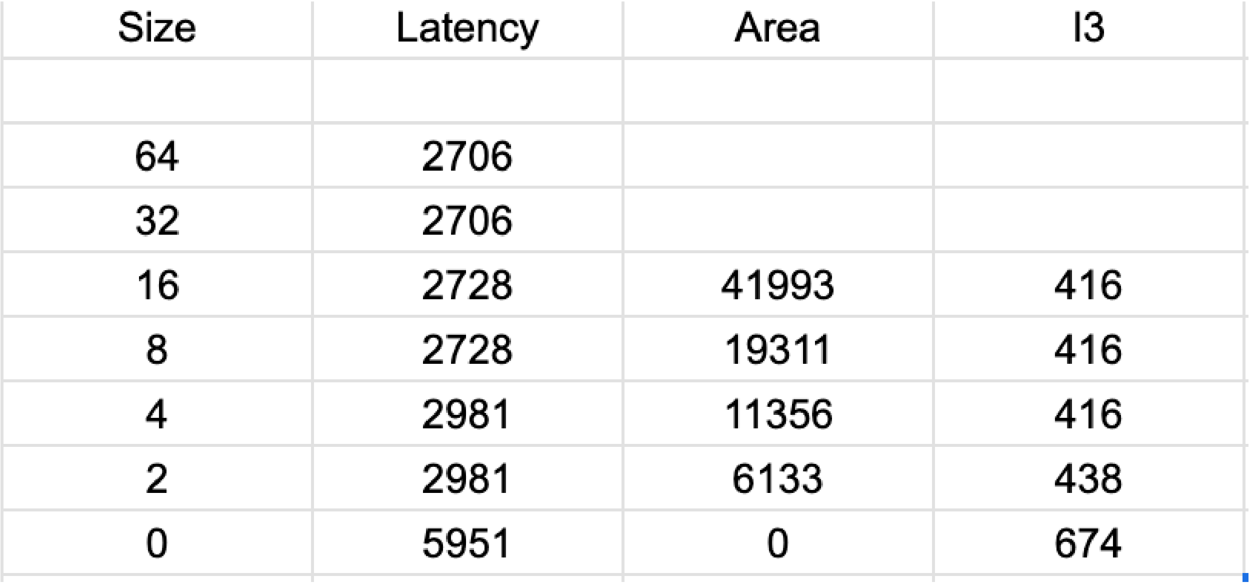
- Wrong directory format: −5%.
- Plagiarism: get 0 in this homework, and TA will inform teacher
- Late submission: -10% per day, at most a week
- Major mistakes causing compilation error → programming part 0. (Judged by TA)

Reference

This is the TA's implementation: Write through, Write around, and Direct-map cache.

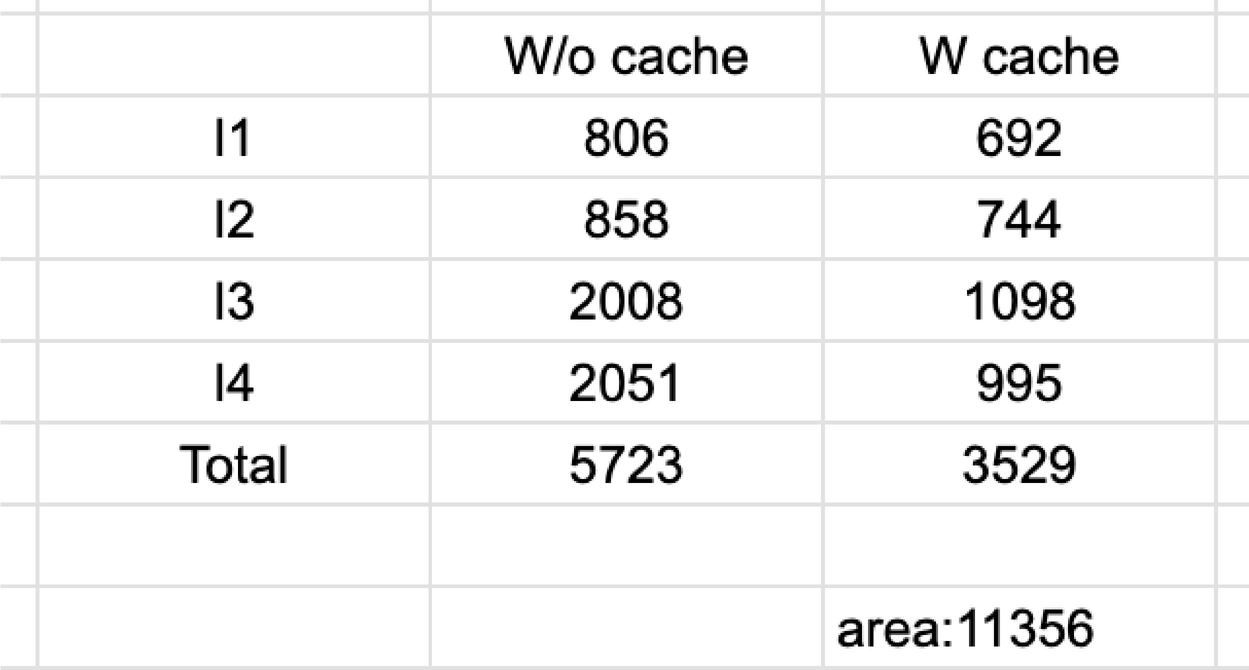


```verilog
localparam CACHE_SIZE = 4;  // num of blocks
localparam INDEX_BITS = $clog2(CACHE_SIZE);
localparam TAG_BITS    = ADDR_W - INDEX_BITS - 4; // 32-2-4=26bits
// (4: 2 bits for choosing data among byte data, 2 bits for block offset)
assign addr = i_proc_addr-i_offset;
assign tag   = addr[(ADDR_W-1)-:TAG_BITS];
assign index = addr[((ADDR_W-1)-TAG_BITS)-:INDEX_BITS];
assign block = addr[3:2];

assign o_mem_addr   = {tag, index, 4'b0000} + i_offset;
// 2 bits  for choosing data among byte data and 2 bits for block offset

reg  [TAG_BITS + 4*BIT_W + 1:0] cache_r [0:CACHE_SIZE-1];
reg  [TAG_BITS + 4*BIT_W + 1:0] cache_w [0:CACHE_SIZE-1];
```



Size	Latency	Area	I3
64	2706		
32	2706		
16	2728	41993	416
8	2728	19311	416
4	2981	11356	416
2	2981	6133	438
0	5951	0	674

It is not the golden testcase.



	W/o cache	W cache
I1	806	692
I2	858	744
I3	2008	1098
I4	2051	995
Total	5723	3529
		area:11356

It is the golden testcase (4 entry cache).

Contact

Questions: email **eclab.ca.ta@gmail.com**