Jetic Gū

1. Handwritten submissions and proprietary formats (e.g. Pages or MS Word) **will not be graded**.
2. Mathematical expressions must be written entirely using LaTeX, otherwise **50%-100%** of marks will be deducted.
3. Untested design will receive 0.

Submission File structure:

```
submission.zip
    — mycache.py
    — sram.v
    — ram.v
    — sram.sim.vwf
    — ram.sim.vwf
```

The `vwf` files are 3pt each, `mycache.py` is worth 4pt.

Lab 2

1. Cache simulator (5pt). Here are the instructions:
    1. Download jetic.org/dl/mycache.py and jetic.org/dl/mycache_test.py
    2. `MyCache` class takes L1-L3 cache latency and block sizes as parameters, as well as main memory access latency.
    3. You should modify the `access` method in `mycache.py`, this method takes a `list` of `int` as parameters, for which you'll need to simulate cache operations.
    4. You can implement the levels of cache using whatever data structures you want. If a target address block is a hit in the current level, access is provided and you log the latency. If it's a miss, you'll need to copy the entry from a lower-level cache, and log both the latency at the current level and lower-levels.
    5. If a new block needed to be added to the current cache but the cache is full, the oldest entry is replaced with the new one.
    6. You can test your implementation by using the provided `mycache_test.py`. You should add more test cases, and do not change the interface of the `MyCache` class.
    7. Submit `mycache.py` only for this question.
2. Implement an SRAM cell using Verilog (2.5pt).
    1. The component must have `B`, `nB`, `Select`, `Q`, and `nQ` as IO;
    2. Show your component working in `sram.sim.vwf`. You must ensure you cover all possible input/output/state combinations.
3. Implement a 16bit RAM unit in Verilog (2.5pt).
    1. Your module must be named ram, and has the following IO:

1. Standard input: bidirectional 16bit `db` bus, 13bit `a` bus for address, 1bit inputs `clk`, `we`.
2. QDM (Data Query Mask) control: 1bit inputs `LDQM` and `UDQM`. This allows masking for Lower Byte and Upper Byte during read/write operations.

2. Here's how QDM control works. In byte addressable memory, we are always able to individually manipulate every single byte of a word regardless of CPU's spec. Say you have a 16bit CPU, your memory unit should permit you to write to only the upper byte, or lower byte.

Mode	WE	LDQM	UDQM
Read Mode DQM Controls Output	0	If 1: output the lower byte to db[7:0]; If 0: db[7:0] gets high impedance	If 1: output the upper byte to db[15:8]; If 0: db[15:8] gets high impedance
Write Mode DQM Controls Output	1	If 1: write db[7:0] to the lower byte of mem[a]; If 0: Do not change mem[a][7:0]	If 1: write db[15:8] to the upper byte of mem[a]; If 0: Do not change mem[a][15:8]

3. Show your component working in `ram.sim.vwf` . You must cover at least 5 read operations and 5 write operations minimum, including 1 instance of changing the content of 1 memory address multiple times (2.5pt) Hint: make sure you are running `Timing Simulation`, not `Functional`. You will also notice some delays in the timing diagram, please make note of that. These delays will require you to adjust your grid size and CLK frequency to make the whole thing work/look correctly. In your own notes, you should include (not graded, but these questions will come up in the following quiz):
    1. How much time is required for a write operation to complete? Should it be measured in `ns` or Positive/Negative pulses?
    2. How much time is required for a read operation to complete? Should it be measured in `ns` or Positive/Negative pulses?
    3. In our examples, we used `posedge` for all state transitions. If we change them to `negedge`, does it change the answers you have above?
    4. Is this memory unit byte-addressable?