1. 实验目的和要求

Purpose

- Understand the principle of Cache Management Unit (CMU) and State Machine of CMU
- Master the design methods of CMU and Integrate it to the CPU.
- Master verification methods of CMU and compare the performance of CPU when it has cache or not

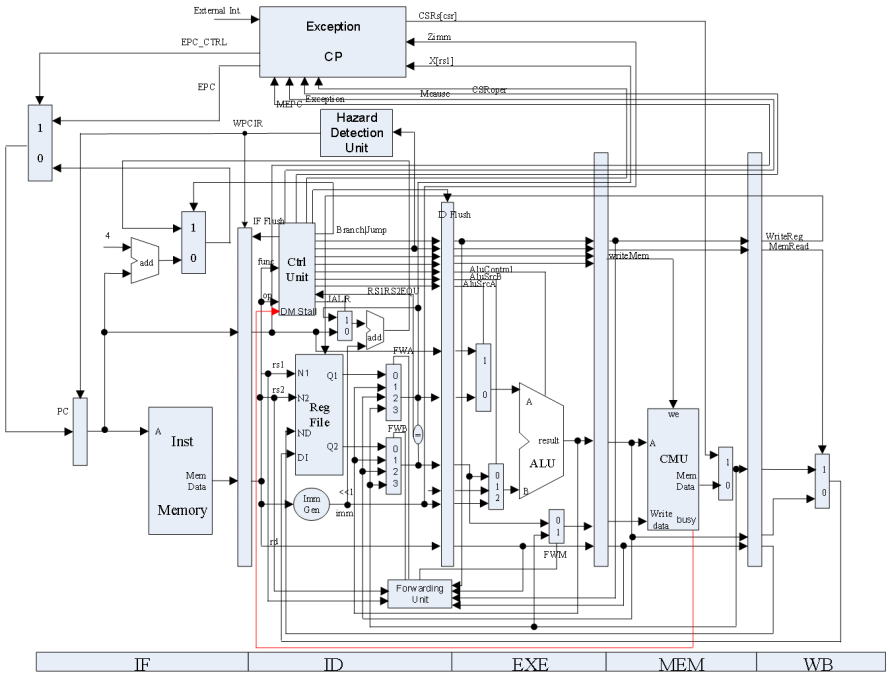
Task

- Design of Cache Management Unit and integrate it to CPU
- Observe and Analyze the Waveform of Simulation
- Compare the performance of CPU when it has cache or not

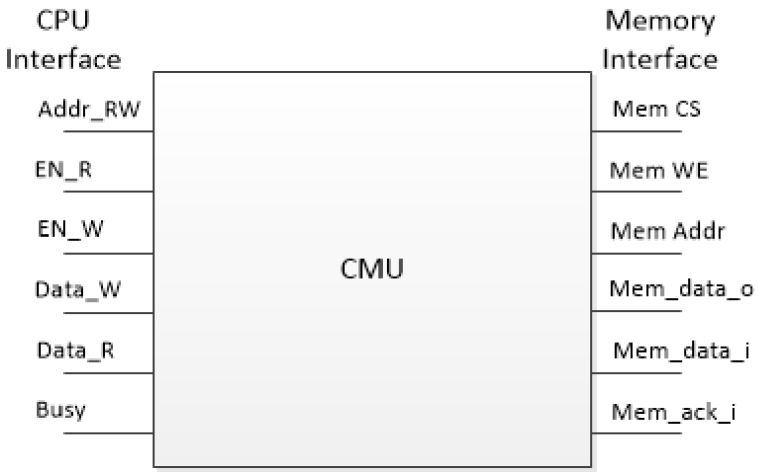
2. 实验内容和原理

CMU

CMU, cache management unit, is the unit controlling the behavior of cache. The memory in the previous pipelined CPU is replaced by a CMU in this experiment. Exactly, CPU processes data in the cache.



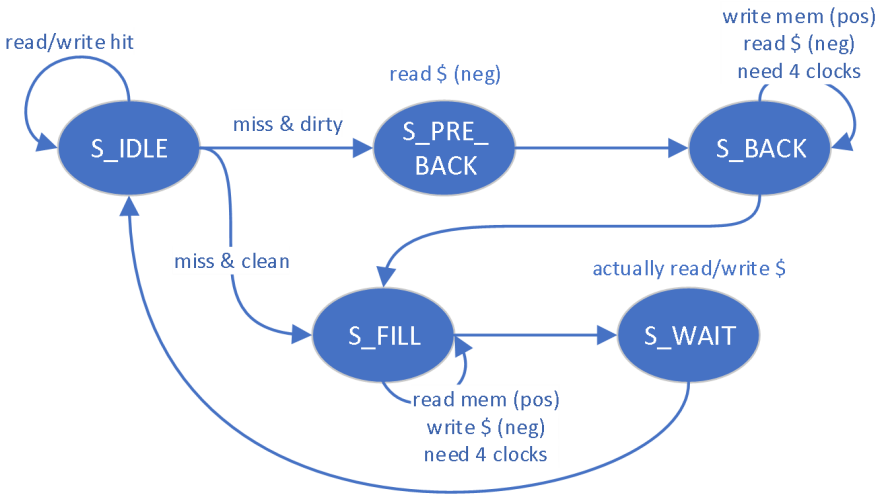
A CMU provides many ports for communicating with CPU and memory. If there is a data miss and CMU is loading data from memory, CPU should stall to wait for the data to be ready.



Cache Management State Machine

All the cache operations occur at the negative edge and all the memory operations occur at the negative edge. When the cache interacts with memory, CMU should change its state only after receiving acknowledgement signal from memory.

- `S_IDLE`: no memory operations in this state
- `S_PRE_BACK`: read cache once to get the data to be written back
- `S_BACK`: write data back to memory in several cycles
- `S_FILL`: load data to cache from memory in several cycles
- `S_WAIT`: read out the newly arrived data



3. 实验过程和数据记录

There is only one module to be filled out.

cmu.v

In the sequential block controllng state, there are only a few blanks to be filled out.

Before finishing other code, we should notice that there is a block updating state and the counter at the positive edge.

```
    always @ (posedge clk) begin
        if (rst) begin
            state <= S_IDLE;
            word_count <= 2'b00;
        end
        else begin
            state <= next_state;
            word_count <= next_word_count;
        end
    end
```

According to the state diagram, if there is a hit, the machine remains `S_IDLE`. If there is a miss and the data to be repalced is dirty, the dirty line should be written back to memory at first, thus the machine entering `S_PRE_BACK` reading the dirty data. If not, the data is immediately loading in, and the machine enters `S_FILL`.

```
S_IDLE: begin
    if (en_r || en_w) begin
        if (cache_hit)
            next_state = S_IDLE;
        else if (cache_valid && cache_dirty)
            next_state = S_PRE_BACK;
        else
            next_state = S_FILL;
    end
    next_word_count = 2'b00;
end
```

According to the diagram, there is only one destination after this state. The machine enters `S_BACK` writing dirty data back.

```
S_PRE_BACK: begin
    next_state = S_BACK;
    next_word_count = 2'b00;
end
```

According to the diagram, the machine is writing back dirty data. The counter increments only after the acknowledgement is received. If the last acknowledgement is received, the machine turns to `S_FILL` loading new data. If not, the machine remains `S_BACK`.

```
S_BACK: begin
    if (mem_ack_i && word_count == {ELEMENT_WORDS_WIDTH{1'b1}})    // 2'b11 in
default case
        next_state = S_FILL;
    else
        next_state = S_BACK;

    if (mem_ack_i)
        next_word_count = word_count + 2'b01;
    else
        next_word_count = word_count;
end
```

According to the diagram, the counter increments only after the acknowledgement is received. If the last acknowledgement is received, the machine turns to `S_WAIT`. If not, the machine remains `S_FILL`.

```
S_FILL: begin
    if (mem_ack_i && word_count == {ELEMENT_WORDS_WIDTH{1'b1}})
        next_state = S_WAIT;
    else
        next_state = S_FILL;

    if (mem_ack_i)
        next_word_count = word_count + 2'b01;
    else
        next_word_count = word_count;
end
```

According to the diagram, reset the counter and turn the machine to `S_IDLE`. (This is simplified here according to my comprehension)

```
S_WAIT: begin
    next_state = S_IDLE;
    next_word_count = 2'b00;
end
```

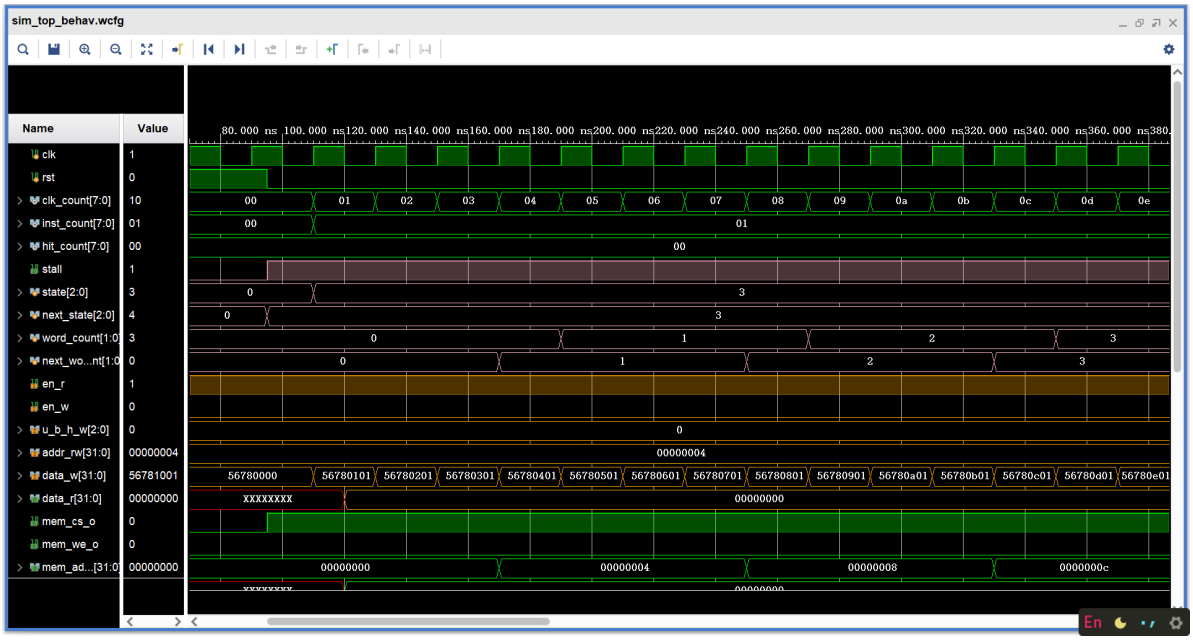
At last, if the machine is not turning into `S_IDLE`, which indicates there is a miss and a latency for required data is necessary, CPU should stall to wait for the missing data.

```
assign stall = (next_state != S_IDLE);
```

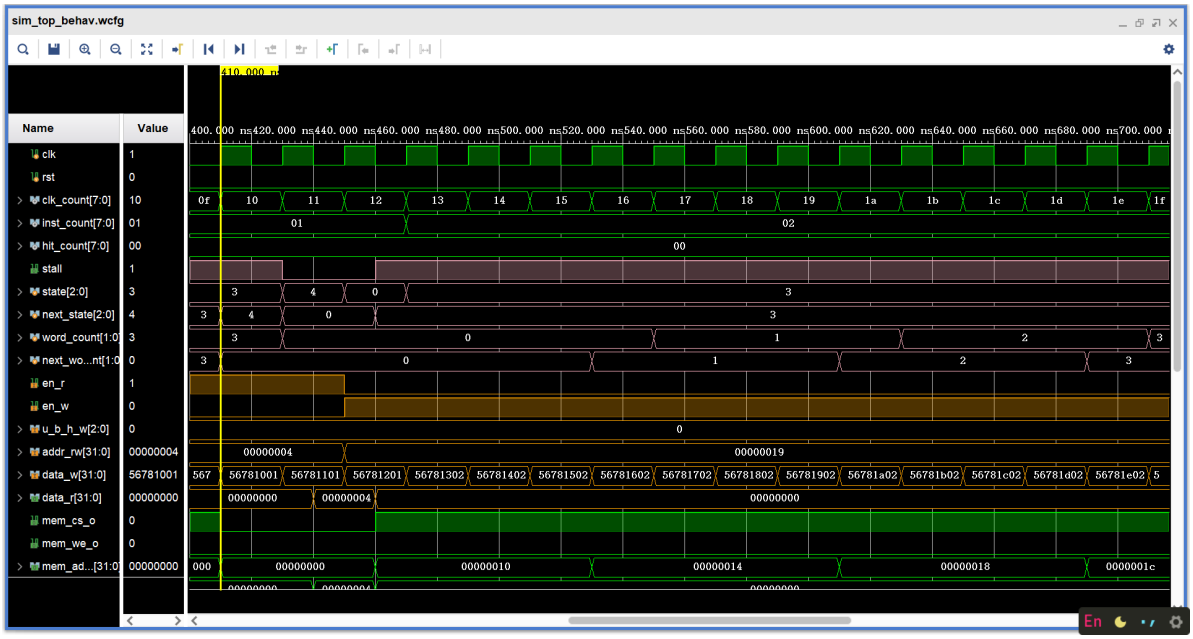
4. 实验结果分析

Simulation

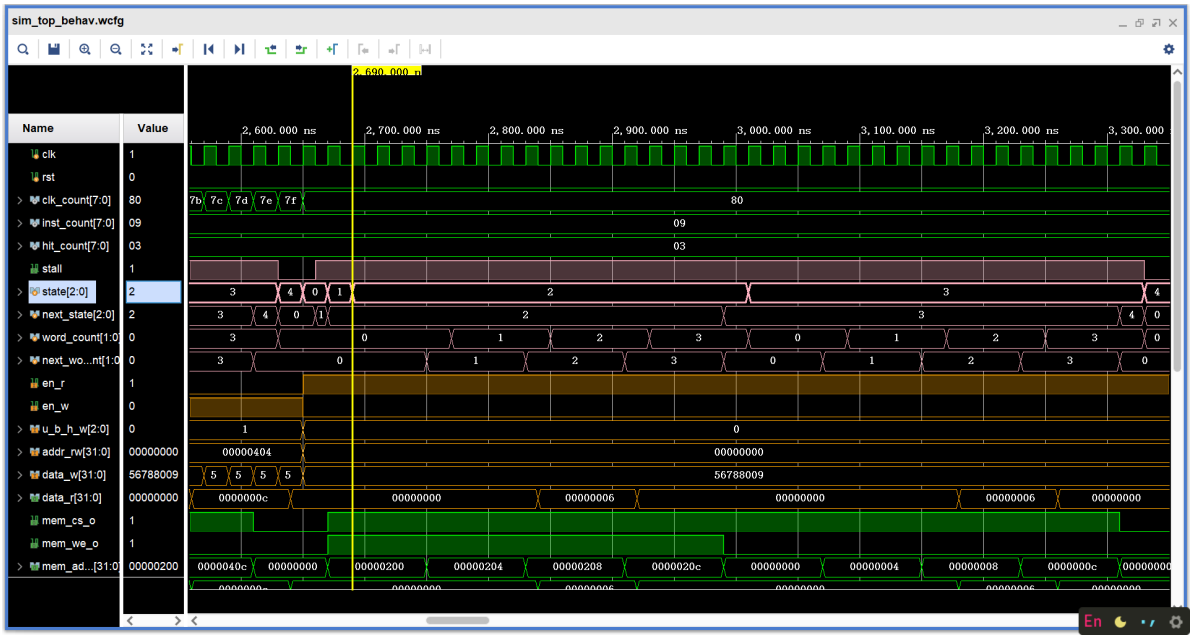
Run the simulation and we can see the result below. The first `lb` instruction causes a miss and we can see `stall` signal is set for the miss. Since this is the first miss and there is no data in the cache, we can see the state changes from `S_IDLE` to `S_FILL` immediately. The counter increments per cycle, which indicates data is being loaded from memory.



We can see state `S_FILL` lasts for 16 cycles. This is because the bandwith between memory and cache is one word per 4 cycles, and one cache line consists of 4 words, which is 16 bytes. Cache needs 16 cycles to complete loading a cache line from memory or writing a cache line to memory.

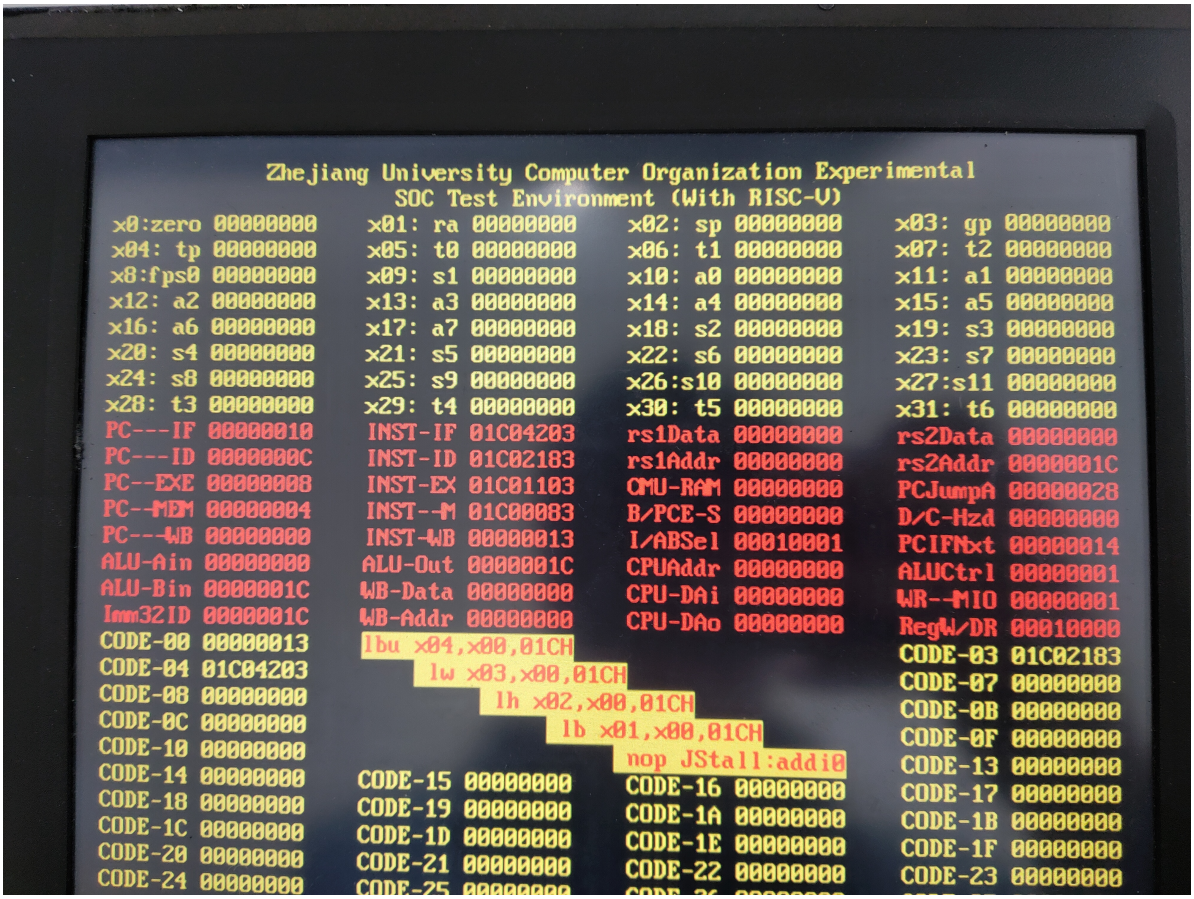


The following figure shows that if the data to be replaced is dirty, the design also takes 16 cycles to write back dirty data. The other part of the result is the same as that given in the guidance, so is omitted here, the analysis of which is similar to this part.



Program Device

The first instruction causing a miss just enters phase MEM. At this time, the state of CMU is still `S_IDLE`.



Then we can see, in the next cycle, CPU stalls. CMU changes to state `S_FILL`, and the counter of RAM begins to increment as needed data is loaded to chche.

Loading continues. As the analysis of the simulation, the counter of RAM goes from 0 to 3 for 4 times, which indicates 4 words are loaded at a time and it takes 16 cycles.

Finally, CMU reaches state `S_WAIT`.

CMU returns to state `S_IDLE` and CPU goes on.

The following figures show that if there is a hit, CPU works continually.

5. 讨论与心得

This experiment is relatively easy since there is not much work to do and all the blanks can be filled out according to the state diagram. It just takes some effort to completely understand how CMU works.