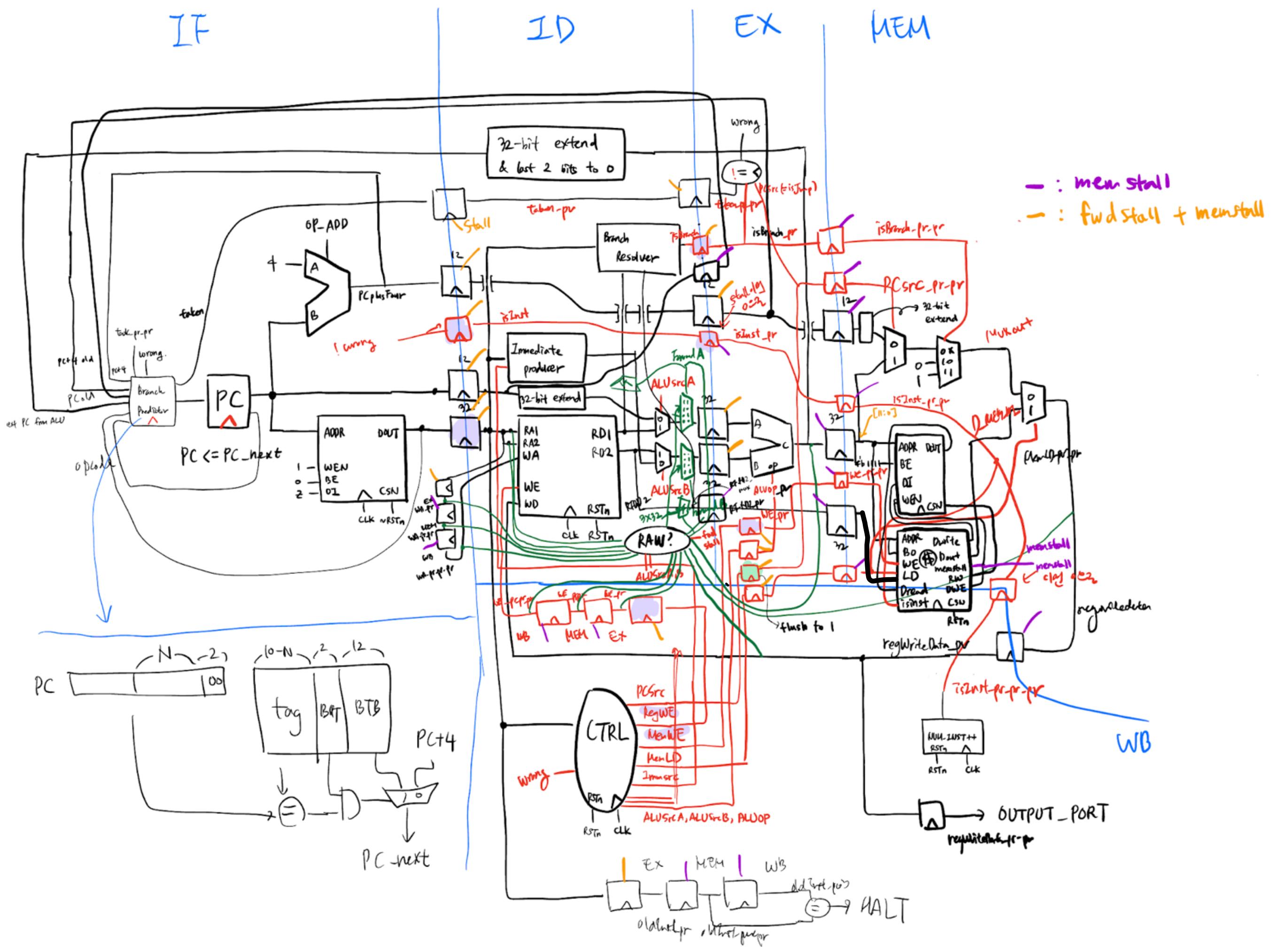
EE312 Lab #6 Report

20180610 정의준

1. Introduction

In Lab 6, we add a cache to our pipelined CPU. In real CPUs, D-memory access takes a few cycles to hundreds of cycles, which makes a huge overhead through the pipeline. Therefore, every CPU has two to three level of cache that temporarily saves the data in the memory which is much faster than DRAM. In this lab, we use a new D-MEM module that takes a few cycles to load and store data. Then, we add a cache module to see whether the added cache can increase the throughput by caching the frequently-used data in the cache. We compare the number of spent cycles to see the impact of the cache.

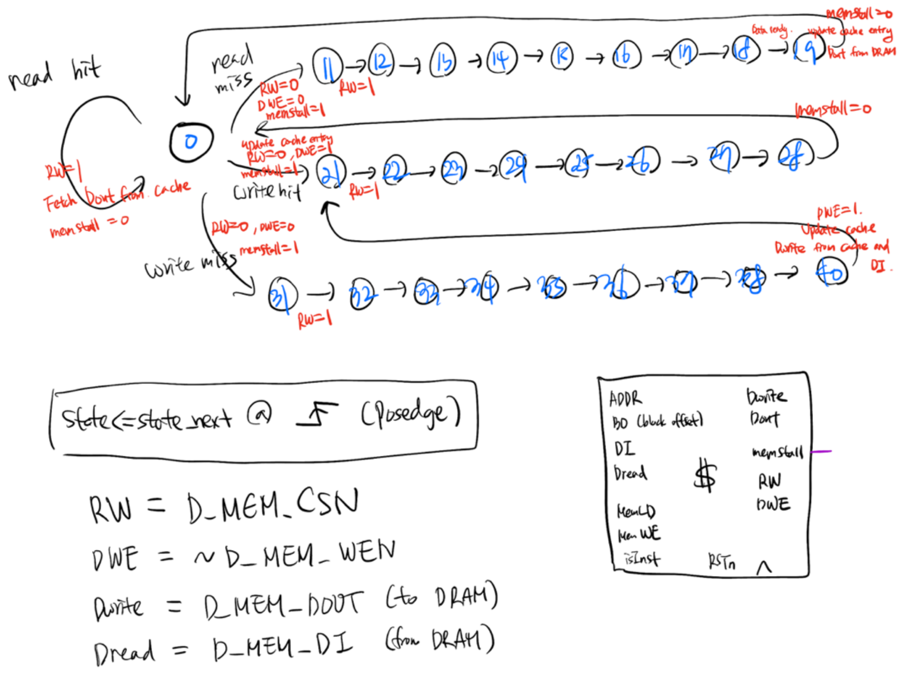
2. Design



Compared to Lab 5, there are two major changes that have occurred to our pipelined CPU. First, to use the D-RAM with latency, every stage of the pipeline should stall. I created a stall signal denoted in dark purple, named 'memstall'. This signal is generated by the cache module, and controls most of the pipeline registers. Also, the orange stall signal is changed from 'fwdstall' to 'memstall OR fwdstall.' The fwdstall signal is only activated when there is RAW hazard after load instruction.

Second, a new cache module is added to **MEM stage** along the D-MEM module. Instead of fetching the data directly from the D-mem module, the data is fetched from the cache module. Also, the cache module's input and output ports (Dread and Dwrite, 128 bit) are directly connected to the D-mem module. Also, D-mem's CSN signal is controlled by the cache module as well. Turning off CSN signal to 0 is like starting a timer in the D-MEM to make it start to fetch or store data. CSN signal is turned off for 1 cycle when there occurs a read-miss, write-hit or write-miss. The detailed explanation for the cache module is in the next section.

3. Implementation



To make a cache that count the number of stalls, I made a FSM(finite state machine) as the above. The blue numbers in the bubbles denote the state. This FSM can count the number of stalled cycles for each cache status. It can count and do some proper works for each read-hit, read-miss, write-hit, and write-miss scenarios. The state is updated on the rising edge.

Let's go over the ports of the cache module. The 12-bit address is fed into ADDR and BO. The last two bits are used as the block offset (BO). DI is the 32bit data that is to be stored by store instruction. Dread and Dwrite are 128bit and directly connected to the D-MEM module. Each MemLD, MemWE flags are turned on when the instruction is load or store respectively. isInst flag is to distinguish whether the instruction is flushed or stalled. This number is 1 only when the instruction is valid. Using the statement (MemLD &&

isInst) makes the pipeline distinguish load-RAW hazards and stall the pipeline properly. Also, this statement makes the CPU to properly count the number of instructions and cache-hits.

Memstall is the signal that is generated when the whole pipeline should wait for D-MEM module's delay. Since the whole pipeline stalls during the cache access, our cache is blocking cache. RW is connected to D-mem's CSN, and DWE is connected to D-mem's WE (write enable). Dout is connected to the WB stage of the pipeline.

As we can see from the FSM diagram, the D_MEM_CSN signal is only turned off for one cycle, when the D-mem module needs to fetch or store some data to the DRAM. The FSM helped me to rigorously deal with this signal.

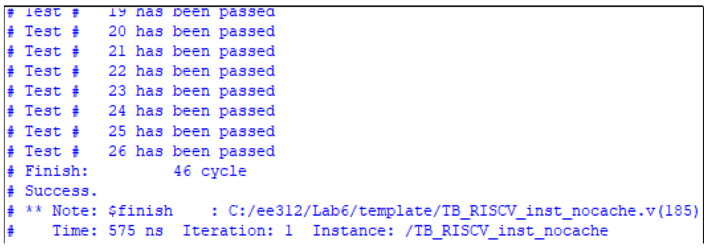
The size of the cache was explicitly denoted in the lecture note. Since there should be 32 words in the cache, there are 8 rows of cache-line in the databank. So, the last 3 bits of the address (except the last two bit for granularity 2^2=4) are used as the index, and the rest 7 bits are used as the tags.

Our cache is **direct-mapped**, the write-hit policy is **write-through**, and the write-miss policy is **write-allocate**. By accessing the single cache databank only with the memory address, the cache is direct-mapped. Also, the last two policies implemented by the above FSM. The cache waits for 8 seconds to write to the DRAM when the cache entry is updated. Also, on write-miss, the cache fetches the data from DRAM and updates the cache entry.

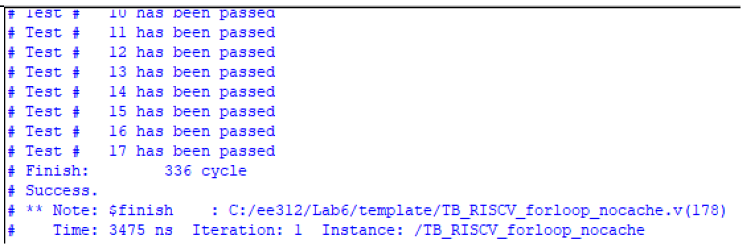
I made a similar FSM to count the D-MEM latencies when implementing the CPU with no cache. The latency counter module is at RISCV_LATENCY_CNT.v, and it is only used for the no-cache CPU. Since the FSM for the no-cache version is much simpler, I will omit the diagram.

4. Evaluation

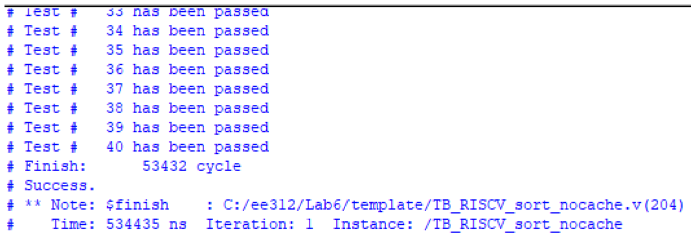
- Testbench results (console, Cache X)



```
# Test #   19 has been passed
# Test #   20 has been passed
# Test #   21 has been passed
# Test #   22 has been passed
# Test #   23 has been passed
# Test #   24 has been passed
# Test #   25 has been passed
# Test #   26 has been passed
# Finish:          46 cycle
# Success.
# ** Note: $finish    : C:/ee312/Lab6/template/TB_RISCV_inst_nocache.v(185)
#    Time: 575 ns  Iteration: 1  Instance: /TB_RISCV_inst_nocache
```



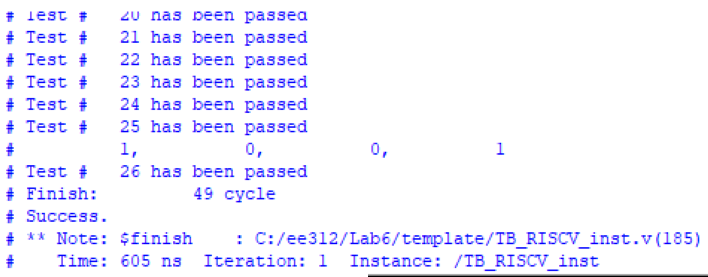
```
# Test #   10 has been passed
# Test #   11 has been passed
# Test #   12 has been passed
# Test #   13 has been passed
# Test #   14 has been passed
# Test #   15 has been passed
# Test #   16 has been passed
# Test #   17 has been passed
# Finish:         336 cycle
# Success.
# ** Note: $finish    : C:/ee312/Lab6/template/TB_RISCV_forloop_nocache.v(178)
#    Time: 3475 ns  Iteration: 1  Instance: /TB_RISCV_forloop_nocache
```



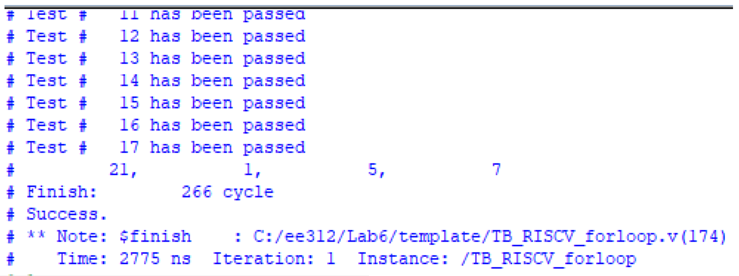
```
# Test #   33 has been passed
# Test #   34 has been passed
# Test #   35 has been passed
# Test #   36 has been passed
# Test #   37 has been passed
# Test #   38 has been passed
# Test #   39 has been passed
# Test #   40 has been passed
# Finish:       53432 cycle
# Success.
# ** Note: $finish    : C:/ee312/Lab6/template/TB_RISCV_sort_nocache.v(204)
#    Time: 534435 ns  Iteration: 1  Instance: /TB_RISCV_sort_nocache
```

- Testbench results (console, Cache O)

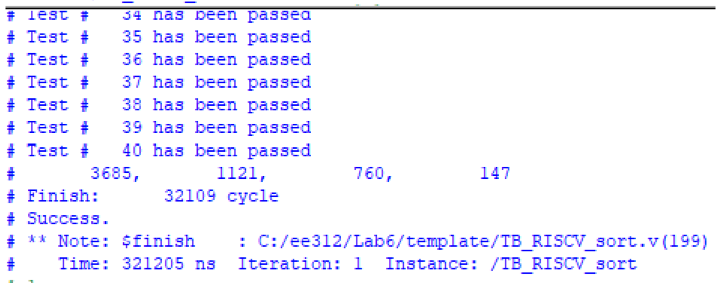
Four numbers denote the # of cache hits/misses.



```
# Test #   20 has been passed
# Test #   21 has been passed
# Test #   22 has been passed
# Test #   23 has been passed
# Test #   24 has been passed
# Test #   25 has been passed
#          1,           0,            0,            1
# Test #   26 has been passed
# Finish:         49 cycle
# Success.
# ** Note: $finish    : C:/ee312/Lab6/template/TB_RISCV_inst.v(185)
#    Time: 605 ns  Iteration: 1  Instance: /TB_RISCV_inst
```

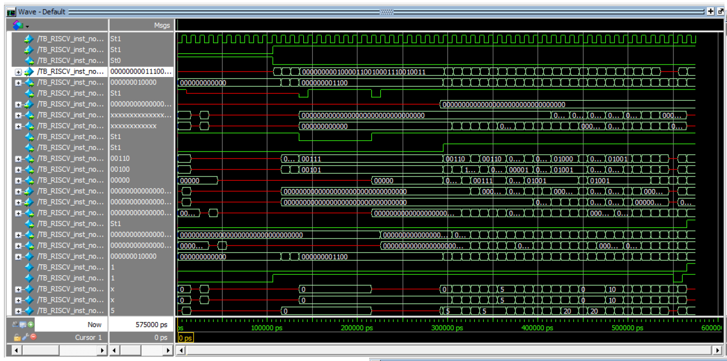
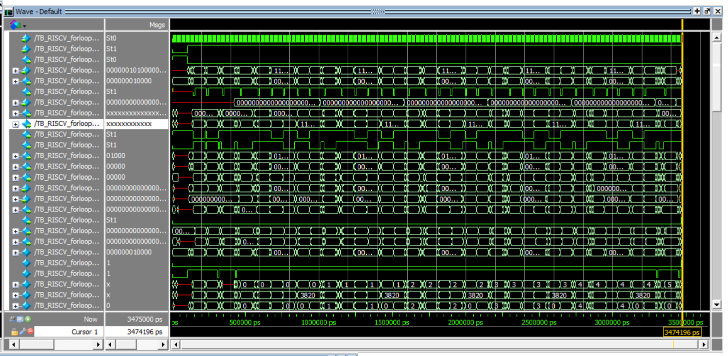
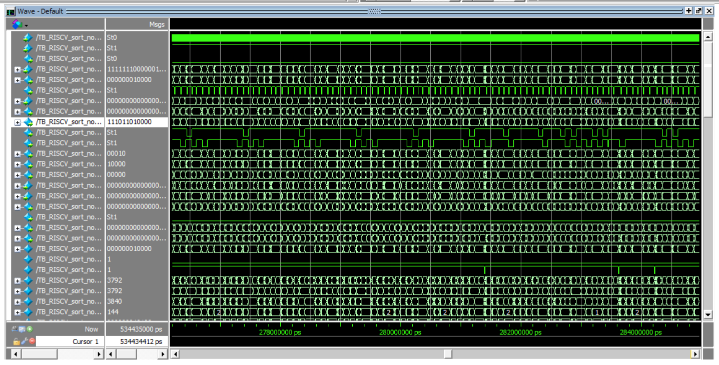


```
# Test #   11 has been passed
# Test #   12 has been passed
# Test #   13 has been passed
# Test #   14 has been passed
# Test #   15 has been passed
# Test #   16 has been passed
# Test #   17 has been passed
#          21,           1,            5,            7
# Finish:         266 cycle
# Success.
# ** Note: $finish    : C:/ee312/Lab6/template/TB_RISCV_forloop.v(174)
#    Time: 2775 ns  Iteration: 1  Instance: /TB_RISCV_forloop
```

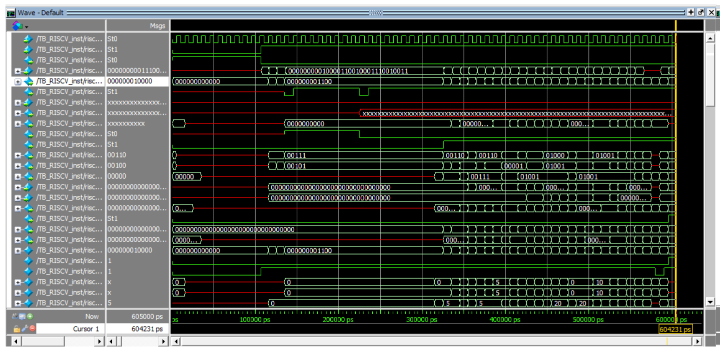
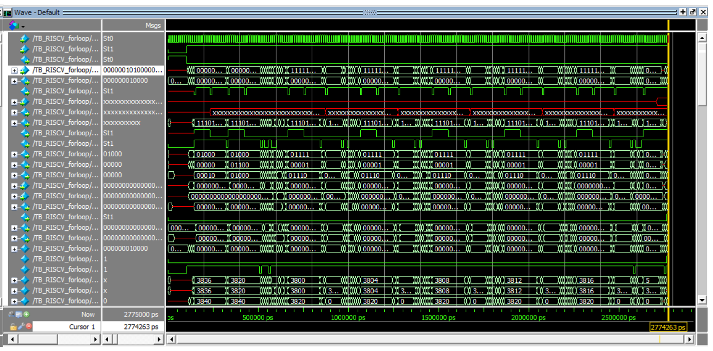


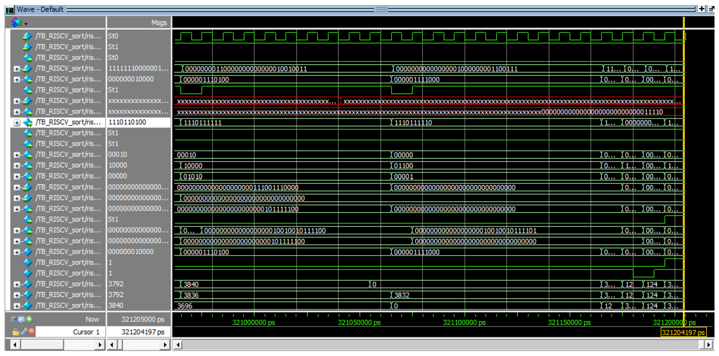
```
# Test #   34 has been passed
# Test #   35 has been passed
# Test #   36 has been passed
# Test #   37 has been passed
# Test #   38 has been passed
# Test #   39 has been passed
# Test #   40 has been passed
#        3685,        1121,         760,         147
# Finish:        32109 cycle
# Success.
# ** Note: $finish    : C:/ee312/Lab6/template/TB_RISCV_sort.v(199)
#    Time: 321205 ns  Iteration: 1  Instance: /TB_RISCV_sort
```

- Testbench results (waveforms, Cache X)

- Testbench results (waveforms, Cache O)



- # of cycles & Impact

Testbench	Cycles – Cache X	Cycles – Cache O	∆Cycle	**Impact (Cycles)**
Inst	46	49	+3	+6.52%
Forloop	336	266	-70	-20.83%
Sort	53432	32109	-21323	-39.91%

The impact of the cache is calculated as the percentage of the reduced number of cycles. We can see that the number of cycles has been reduced up to ~40% in the last testbench.

- # of cache hits/misses & Hit ratio

Testbench	Read-hit	Read-miss	Write-hit	Write-miss	Hit ratio
Inst	1	0	0	1	50%
Forloop	21	1	5	7	76.5%
Sort	3685	1121	760	147	77.8%

Hit ratio is calculated as the portion of hits among all hits and misses.

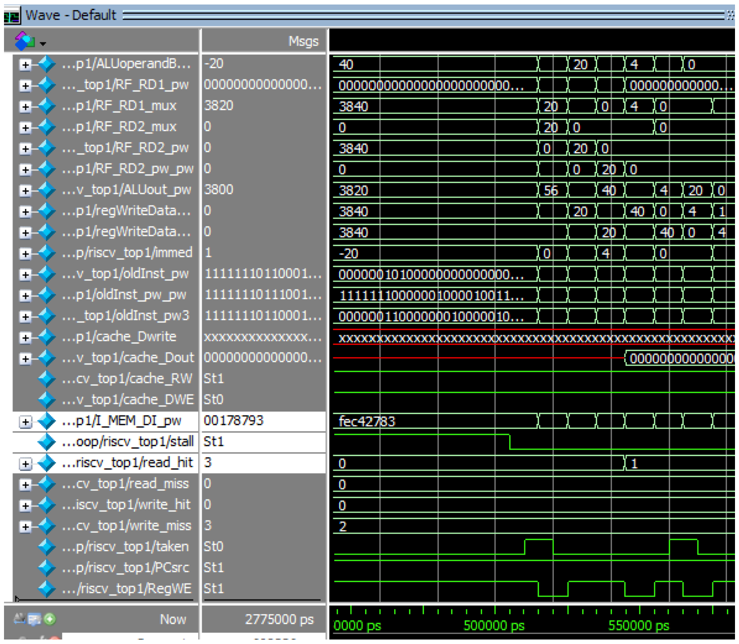
- Comparing # of hits/misses and the ∆Cycle

We can use the above table to theoretically calculate ∆Cycle. In no-cache CPU, every load and store instructions require 8 cycles. In cache CPU, R-hit, R-miss, W-hit, W-miss respectively requires 1, 10, 9, 18 cycles as denoted in the lab instruction. So, this means that our cache increases the number of cycles by (-7), 2, 1 and 10 for each cache hits and misses. If we calculate the number of reduced cycles with the above table, we can obtain the following results.

- Inst: (-7) * 1 + 2 * 0 + 1 * 0 + 10 * 1 = **+3**
- Forloop: (-7) * 21 + 2 * 1 + 1 * 5 + 10 * 7 = **-70**
- Sort: (-7) * 3685 + 2 * 1121 + 1 * 760 + 10 * 147 = **-21323**

We can see that the calculated results are exactly the same as the ∆Cycle.

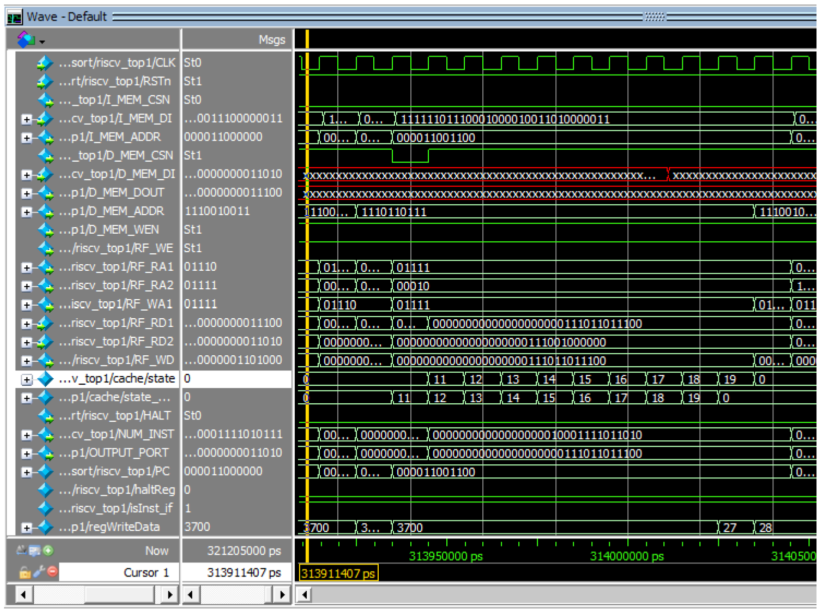
- Waveform of Read-hit



```
Wave - Default
                          Msgs
...p1/ALUoperandB...      -20
..._top1/RF_RD1_pw        00000000000000...
...p1/RF_RD1_mux          3820
...p1/RF_RD2_mux          0
..._top1/RF_RD2_pw        0
...p1/RF_RD2_pw_pw        0
...v_top1/ALUout_pw       3800
...p1/regWriteData...     0
...p1/regWriteData...     0
...p/riscv_top1/immed     1
...v_top1/oldInst_pw      1111111011001...
...p1/oldInst_pw_pw       1111111011001...
..._top1/oldInst_pw3      1111111011001...
...p1/cache_Dwrite        xxxxxxxxxxxxxx...
...v_top1/cache_Dout      00000000000000...
...cv_top1/cache_RW       St1
...v_top1/cache_DWE       St0
...p1/I_MEM_DI_pw         00178793
...oop/riscv_top1/stall   St1
...riscv_top1/read_hit    3
...cv_top1/read_miss      0
...iscv_top1/write_hit    0
...cv_top1/write_miss     3
...p/riscv_top1/taken     St0
...p/riscv_top1/PCsrc     St1
.../riscv_top1/RegWE      St1
Now                       2775000 ps
```

Look at the time where the read_hit (highlighted in white) changes from 0 to 1. We can see that 'stall' signal is zero, and the I_MEM_DI (Instruction in ID stage) is changing every cycle. Also, the variable cache_Dout changes its value from x to some number. This means that the load instruction has successfully loaded the needed data from the cache in a single cycle.

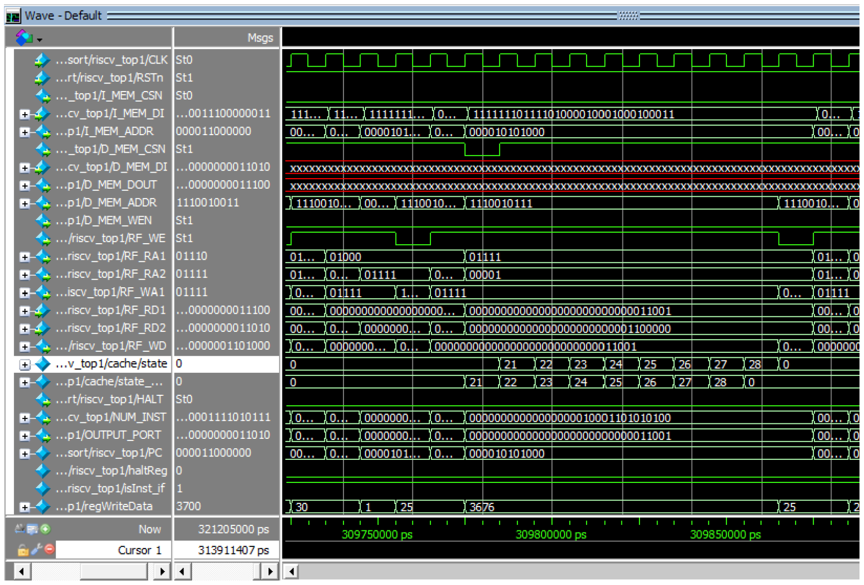
- Waveform of Read-miss



```
Wave - Default
                          Msgs
...sort/riscv_top1/CLK    St0
...rt/riscv_top1/RSTn     St1
..._top1/I_MEM_CSN        St0
...cv_top1/I_MEM_DI       ...001110000011
...p1/I_MEM_ADDR          000011000000
..._top1/D_MEM_CSN        St1
...cv_top1/D_MEM_DI       ...0000000011010
...p1/D_MEM_DOUT          ...0000000011100
...p1/D_MEM_ADDR          1110010011
...p1/D_MEM_WEN           St1
.../riscv_top1/RF_WE      St1
...riscv_top1/RF_RA1      01110
...riscv_top1/RF_RA2      01111
...iscv_top1/RF_WA1       01111
...riscv_top1/RF_RD1      ...0000000011100
...riscv_top1/RF_RD2      ...0000000011010
.../riscv_top1/RF_WD      ...0000001101000
...v_top1/cache/state     0
...p1/cache/state_...     0
...rt/riscv_top1/HALT     St0
...cv_top1/NUM_INST       ...0001111010111
...p1/OUTPUT_PORT         ...0000000011010
...sort/riscv_top1/PC     000011000000
.../riscv_top1/haltReg    0
...riscv_top1/isInst_if   1
...p1/regWriteData        3700
Now                       321205000 ps
Cursor 1                  313911407 ps
```

As shown in the FSM diagram, I named the read-miss state (highlighted in white) as 11 to 19. So, the state 0 and the states from 11 to 19 take 10 cycles. The cache is accessed during the first cycle, data fetched from D-MEM for 8 cycles, and updated during the last cycle. We can see the regWriteData (the wire connected to OUTPUT_PORT) and OUTPUT_PORT is not changing for 10 cycles. This means that the pipeline has been stalled for 10 cycles. After 10 cycles of delay, the cache updates and produces the data and hands it to regWriteData wire.

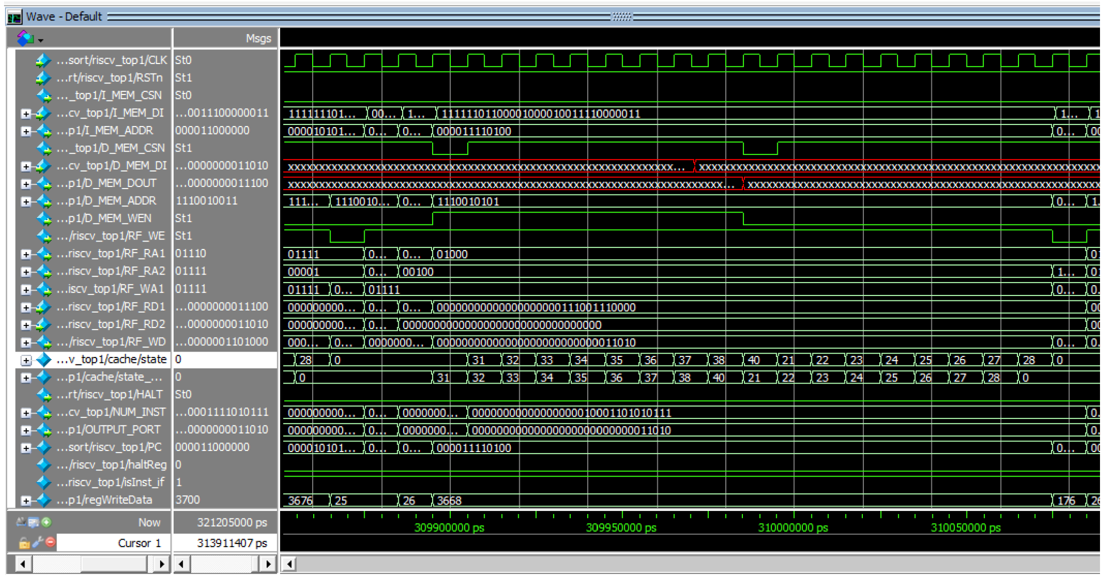
- Waveform of Write-hit



```
...sort/riscv_top1/CLK      St0
...rt/riscv_top1/RSTn       St1
..._top1/I_MEM_CSN          St0
...cv_top1/I_MEM_DI         ...0011100000011
...p1/I_MEM_ADDR            000011000000
..._top1/D_MEM_CSN          St1
...cv_top1/D_MEM_DI         ...0000000011010
...p1/D_MEM_DOUT            ...0000000011100
...p1/D_MEM_ADDR            1110010011
...p1/D_MEM_WEN             St1
.../riscv_top1/RF_WE        St1
...riscv_top1/RF_RA1        01110
...riscv_top1/RF_RA2        01111
...iscv_top1/RF_WA1         01111
...riscv_top1/RF_RD1        ...0000000011100
...riscv_top1/RF_RD2        ...0000000011010
.../riscv_top1/RF_WD        ...0000001101000
...v_top1/cache/state       0
...p1/cache/state_...       0
...rt/riscv_top1/HALT       St0
...cv_top1/NUM_INST         ...0001111010111
...p1/OUTPUT_PORT           ...0000000011010
...sort/riscv_top1/PC       000011000000
.../riscv_top1/haltReg      0
...riscv_top1/isInst_if     1
...p1/regWriteData          3700
Now                         321205000 ps
Cursor 1                    313911407 ps
```

I named the write-hit state (highlighted in white) as 21 to 28. The cache is updated at the first cycle, and the D-MEM is accessed for 8 cycles to follow the write-through policy. We can see that the store instruction then occupies for 9 cycles in the waveform. So, we can see the regWriteData wire is stalled for 9 cycles and gets a new output from the next instruction.

- Waveform of Write-miss



```
...sort/riscv_top1/CLK      St0
...rt/riscv_top1/RSTn       St1
..._top1/I_MEM_CSN          St0
...cv_top1/I_MEM_DI         ...0011100000011
...p1/I_MEM_ADDR            000011000000
..._top1/D_MEM_CSN          St1
...cv_top1/D_MEM_DI         ...0000000011010
...p1/D_MEM_DOUT            ...0000000011100
...p1/D_MEM_ADDR            1110010011
...p1/D_MEM_WEN             St1
.../riscv_top1/RF_WE        St1
...riscv_top1/RF_RA1        01110
...riscv_top1/RF_RA2        01111
...iscv_top1/RF_WA1         01111
...riscv_top1/RF_RD1        ...0000000011100
...riscv_top1/RF_RD2        ...0000000011010
.../riscv_top1/RF_WD        ...0000001101000
...v_top1/cache/state       0
...p1/cache/state_...       0
...rt/riscv_top1/HALT       St0
...cv_top1/NUM_INST         ...0001111010111
...p1/OUTPUT_PORT           ...0000000011010
...sort/riscv_top1/PC       000011000000
.../riscv_top1/haltReg      0
...riscv_top1/isInst_if     1
...p1/regWriteData          3700
Now                         321205000 ps
Cursor 1                    313911407 ps
```

I named the write-miss state (highlighted in white) as 31 to 38, 40, and 21 to 28. The later 8 cycles are the same as the write-hit case, because the cache adds a new entry on 31~38 and 40 state and 21~28 is for implementing the write-through policy. We can see that the whole pipeline including the regWriteData wire is stalled for 18 cycles.

5. Discussion

I think the two-week deadline is too short for the LAB 6. It took about the same period of time (10~15 hours) for me to finish LAB 5 and LAB 6, but the 2-week deadline was too short because the LAB 5 has 4-week deadline. The cache module itself was not that hard to design, because it took only about 2 to 3 hours to code and debug everything. However, before making the cache module, designing a latency counter for no-cache-CPU took about a whole day. This was because I didn't properly understand how the new D-memory module works. I tried to get rid of everything and start from the scratch by using an FSM that updates its states at every rising edge. Thankfully, the FSM method worked, and I was able to make the latency counter.

6. Conclusion

In this lab, I was able to design a cache module for my pipeline CPU. By comparing with the no-cache CPU, I was able to get a maximum of about 40% performance boost. Also, I was able to learn how the pipelines should be stalled due to other module's delays and interrupts.