Project 2: Problem 2-2

Cache block size 수정을 위한 Verilog source code 수정

> 코드 변경 사항은 소스 코드 내에서 '->'를 검색(Ctrl+F)하시면 간편하게 직접 확인하실 수 있습니다.

- 본 프로젝트에서는 cache의 cache block size를 32 byte에서 64 byte로 2배 증가시킨 뒤, 각 경우에 대한 hit ratio를 계산함으로써 cache block size가 hit ratio에 미치는 영향을 확인하는 것을 목표로 한다.
- 본 프로젝트에서 사용하는 cache는 다음과 같은 특성을 가진다.
  - 2-way set associative
  - 16 KB size
  - cache block size: 32 byte → 64 byte
- cache block size의 변화에 따라 `dcache_core.v`의 address mapping을 수정하고, `dcache_tag_ram.v`의 tag RAM size 및 tag RAM의 address size를 수정하였다. 수정된 코드는 각각 다음과 같다.
- `dcache_core.v`
  - Cache block size가 32 byte → 64 byte로 증가함에 따라, 다음과 같이 address mapping을 수정하였다.
    - Offset
      - 5-bit → 6-bit
      - 기존 5-bit은 다음과 같이 계산된다. $32\ byte = 2^5$
      - 기존 cache block size가 32 byte 였으므로, 5-bit이었지만, cache block size가 64 byte로 증가하여, 6-bit으로 증가시켜준다. $64\ byte = 2^6$
    - Index
      - 8-bit → 7-bit
      - 기존 8-bit은 다음과 같이 계산된다. $\frac{2^{14}(16KB)}{2^5(32B)} = 2^9$. 또한, 2-way set associative 이므로, 하나의 set은 두 개의 cache line을 포함하고 있다. 따라서, $2^9 \rightarrow 2^8$.
      - Total cache size는 16 KB로 고정되어 있는데, cache block size가 32 byte → 64 byte로 증가하면, 당연히 cache 내의 set 수는 절반으로 감소한다.
      - 즉, cache 내 set 수가 256 → 128로 감소하기 때문에, index는 8-bit → 7-bit으로 감소시켜준다.
    - Tag
      - 19-bit → 19-bit (변화 x)
      - Physical address는 총 32-bit이고, Tag는 전체 address에서 index와 offset을 제외한 상위 비트이기 때문에 Tag의 총 bit은 다음과 같이 계산된다.
      - 32 - 8 - 5 = 19 → 32 - 7 - 6 = 19
      - 따라서, Tag는 cache block size 수정 전후 둘 다 19-bit이다.
    - 한 블럭 안의 word 개수
      - 8 → 16
      - 하나의 Word는 4 byte이다. cache block size가 32 byte → 64 byte로 늘어났으므로, cache block size 수정 전의 경우, 한 블럭당 8개의 word가 있고, 수정 이후의 경우, 16개의 word가 있다.
  - 이러한 address mapping의 수정을 코드에도 반영하였고, 이는 다음과 같다.

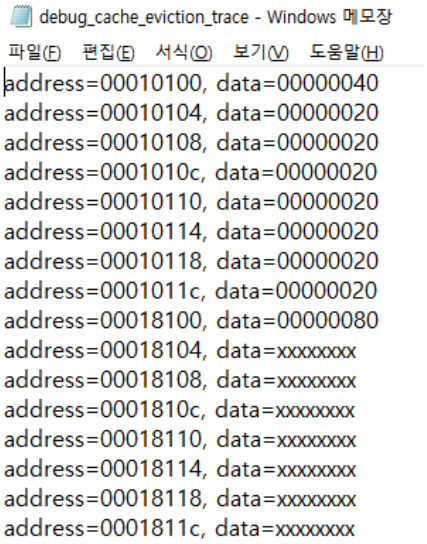
```
//------------------------------------------------------------------
// This cache instance is 2 way set associative. (Line 80)
// The total size is 16KB.
// The replacement policy is a limited pseudo random scheme
```

```
// (between lines, toggling on line thrashing).
// The cache is a write back cache, with allocate on read and write.
//-----------------------------------------------------------------

// Number of ways
localparam DCACHE_NUM_WAYS           = 2; // 2-way set associative

// Number of cache lines
localparam DCACHE_NUM_LINES          = 128; // # of sets 256 -> 128
localparam DCACHE_LINE_ADDR_W        = 7;  // Index 8-bit 8 -> 7

// Line size (e.g. 32->64-bytes)
localparam DCACHE_LINE_SIZE_W        = 6;  // Offset 5-bit -> 6-bit
localparam DCACHE_LINE_SIZE          = 64;  // 32 byte -> 64 byte
localparam DCACHE_LINE_WORDS         = 16;  // # of words 8 -> 16

// Request -> tag address mapping
localparam DCACHE_TAG_REQ_LINE_L     = 6;  // DCACHE_LINE_SIZE_W 5 -> 6
localparam DCACHE_TAG_REQ_LINE_H     = 12; // DCACHE_LINE_ADDR_W+DCACHE_LINE_SIZE_W-1
localparam DCACHE_TAG_REQ_LINE_W     = 7;  // DCACHE_LINE_ADDR_W, index 8-bit -> 7-bit
`define DCACHE_TAG_REQ_RNG          DCACHE_TAG_REQ_LINE_H:DCACHE_TAG_REQ_LINE_L

// Tag fields
`define CACHE_TAG_ADDR_RNG          18:0
localparam CACHE_TAG_ADDR_BITS       = 19;
localparam CACHE_TAG_DIRTY_BIT       = CACHE_TAG_ADDR_BITS + 0;
localparam CACHE_TAG_VALID_BIT       = CACHE_TAG_ADDR_BITS + 1;
localparam CACHE_TAG_DATA_W          = CACHE_TAG_ADDR_BITS + 2;

// Tag compare bits
localparam DCACHE_TAG_CMP_ADDR_L     = DCACHE_TAG_REQ_LINE_H + 1;
localparam DCACHE_TAG_CMP_ADDR_H     = 32-1;
localparam DCACHE_TAG_CMP_ADDR_W     = DCACHE_TAG_CMP_ADDR_H - DCACHE_TAG_CMP_ADDR_L + 1;
`define   DCACHE_TAG_CMP_ADDR_RNG   31:13
```

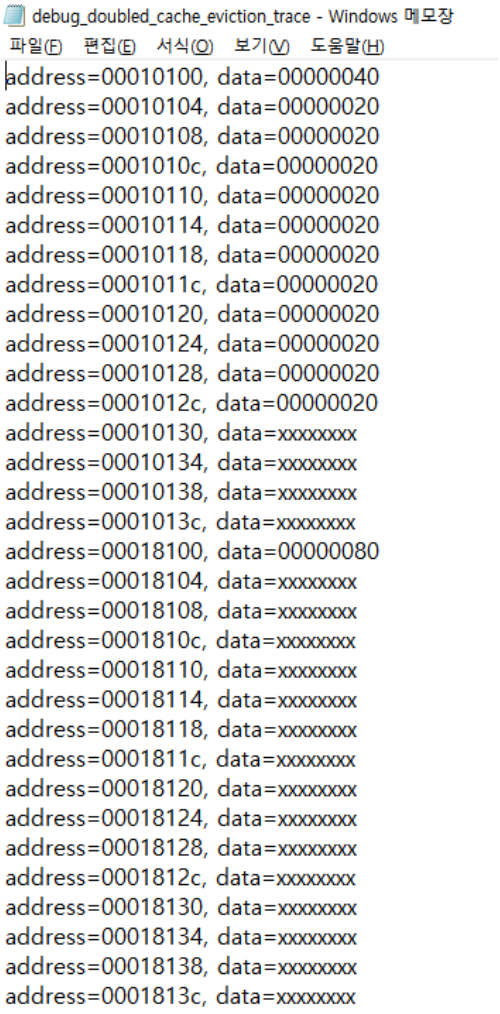
- `DCACHE_NUM_LINES`
  - Cache 내 set의 개수를 의미한다.
  - 따라서, 256 → 128로 수정하였다.
- `DCACHE_LINE_ADDR_W`
  - Cache line address width, 즉 index의 width를 의미한다.
  - 따라서, 8 → 7로 수정하였다.
- `DCACHE_LINE_SIZE_W`
  - Cache line size width, 즉 offset의 width를 의미한다.
  - 따라서, 5 → 6으로 수정하였다.
- `DCACHE_LINE_SIZE`
  - Cache block size를 의미한다.
  - 따라서, 32 → 64로 수정하였다.
- `DCACHE_LINE_WORDS`
  - 한 블럭 안의 word 개수를 의미한다.
  - 따라서, 8 → 16으로 수정하였다.

- `DCACHE_TAG_REQ_LINE_L`
  - 위의 comment를 통해 알 수 있듯이, 이는 line address의 lowest bit을 마킹해주기 위한 변수이다.
  - `DCACHE_LINE_SIZE_W`와 같은 값을 가진다.
  - Offset width가 1 증가했으므로, 5 → 6으로 수정하였다.
- `DCACHE_TAG_REQ_LINE_H`
  - 마찬가지로 위의 comment에서, line address의 highest bit을 마킹해주기 위한 변수임을 알 수 있다.
  - `DCACHE_LINE_ADDR_W` + `DCACHE_LINE_SIZE_W` - 1로 계산된다.
  - `DCACHE_LINE_ADDR_W`, 즉 index width가 1 감소하고, `DCACHE_LINE_SIZE_W`, 즉 offset width가 1 증가했으므로, 결과적으로 값이 변하지 않는다. (12 → 12)
- `DCACHE_TAG_REQ_LINE_W`
  - Cache line address width, 즉 index의 width와 동일한 값을 가진다.
  - 즉, `DCACHE_LINE_ADDR_W`와 동일한 값을 가진다.
  - 따라서, 8 → 7로 수정하였다.
- 이후, Tag의 width는 변하지 않기 때문에, Tag fields 및 Tag compare bits 부분의 파라미터는 수정하지 않는다.

- 수정된 cache block size에 맞게 eviction size도 변경되어야한다. 이는 다음과 같다.
  - Word size는 4B이며, 수정 전 cache block size는 32B이다.
  - 따라서, 수정 전에는 하나의 cache block에 대해서 eviction이 발생할 때, 순차적으로 8개의 word가 evict된다.
  - 이는 cache block size 수정 전, `cache_test.v`를 시뮬레이션하여 얻은 `debug_cache_eviction_trace.txt`를 통해 확인할 수 있다.
    - 해당 파일은 Problem 2 폴더의 reference 폴더에서 확인하실 수 있습니다.



- 수정 후, cache block size는 64B이다.
- 따라서, 수정 후에는 하나의 cache block에 대해서 eviction이 발생할 때, 순차적으로 16개의 word가 evict되어야 정상적으로 eviction이 동작한다고 볼 수 있다.
- 이는 마찬가지로 cache block size 수정 후, `cache_test.v`를 시뮬레이션하여 얻은 `debug_doubled_cache_eviction_trace.txt`를 통해 확인할 수 있다.
  - 해당 파일은 Problem 2 폴더의 reference 폴더에서 확인하실 수 있습니다.



- 따라서 위와 같이 수정 된 cache block size에 따라 eviction이 정상적으로 동작하도록 다음과 같이 AXI Request 부분의 코드 일부 및 그와 관련된 코드 일부(Generated File, Registers / Wires, Outport)를 적절하게 수정하였다.

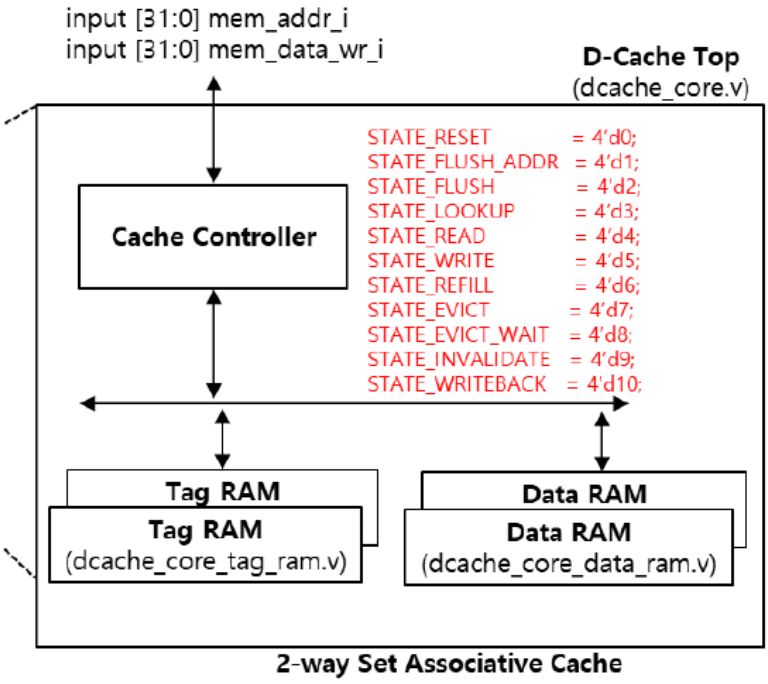
```
//------------------------------------------------------------------
// AXI Request (Line 851)
//------------------------------------------------------------------

//------------------------------------------------------------------
// Original Code
//------------------------------------------------------------------

// reg [7:0] pmem_len_q;
// always @ (posedge clk_i or posedge rst_i)
// if (rst_i)
//     pmem_len_q   <= 8'b0;
// else if (state_q != STATE_EVICT && next_state_r == STATE_EVICT)
//     pmem_len_q   <= 8'd7;
// else if (pmem_rd_w && pmem_accept_w)
//     pmem_len_q   <= pmem_len_w;
// else if (state_q == STATE_REFILL && pmem_ack_w)
//     pmem_len_q   <= pmem_len_q - 8'd1;
// else if (state_q == STATE_EVICT && pmem_accept_w)
//     pmem_len_q   <= pmem_len_q - 8'd1;

// assign pmem_last_w = (pmem_len_q == 8'd0);

//------------------------------------------------------------------
// Changed Code
//------------------------------------------------------------------

reg [15:0] pmem_len_q; // reg [7:0] pmem_len_q; -> reg [15:0] pmem_len_q;
always @ (posedge clk_i or posedge rst_i)
```

```
if (rst_i)
    pmem_len_q   <= 16'b0; // 8'b0 -> 16'b0
else if (state_q != STATE_EVICT && next_state_r == STATE_EVICT)
    pmem_len_q   <= 16'd15; // 8'd7 -> 16'd15
else if (pmem_rd_w && pmem_accept_w)
    pmem_len_q   <= pmem_len_w;
else if (state_q == STATE_REFILL && pmem_ack_w)
    pmem_len_q   <= pmem_len_q - 16'd1; // 8'd1 -> 16'd1
else if (state_q == STATE_EVICT && pmem_accept_w)
    pmem_len_q   <= pmem_len_q - 16'd1; // 8'd1 -> 16'd1

assign pmem_last_w = (pmem_len_q == 16'd0); // 8'd0 -> 16'd0
```

```
//-----------------------------------------------------------------
//                        Generated File  (Line 43)
//-----------------------------------------------------------------
module dcache_core
(
    // Inputs
    // ...

    // Outputs
    // ...
    ,output [  15:0]  outport_len_o // Line 72 ,output [  7:0]  outport_len_o -> ,output [  15:0]  outport_len_o
    // ...
);

// ...

//-----------------------------------------------------------------
// Registers / Wires (Line 233)
//-----------------------------------------------------------------
// ...
wire  [  15:0]  pmem_len_w; // Line 239, wire  [  7:0]  pmem_len_w; -> wire  [  15:0]  pmem_len_w;
// ...

// ...

//-----------------------------------------------------------------
// Outport (Line 930)
//-----------------------------------------------------------------

// ...

// AXI Read channel
// ...
assign pmem_len_w        = (refill_request_w || pmem_rd_q || (state_q == STATE_EVICT && pmem_wr0_q)) ? 16'd15 : 16'd0;
// Line 943, ? 8'd7 : 8'd0; -> ? 16'd15 : 16'd0;
```

- 먼저, eviction이 어떻게 동작을 간략하게 살펴보자.
  - STATE_EVICT로 state transition이 일어나면서 Eviction이 시작되고, 특정 cache block에 대해서 eviction이 발생할 때, 해당 cache block 내의 모든 word를 순차적으로 evict시켜준다.
  - 이후, `pmem_accept_w && pmem_last_w`의 condition이 TRUE로 되면, STATE_EVICT_WAIT로 state transition이 발생한다. 이는 End of evict, wait for write completion의 상태를 의미한다.
  - 이때 위의 코드와 같이 `pmem_last_w`는 `pmem_len_q`가 0이되면 1이 할당된다.
    - `assign pmem_last_w = (pmem_len_q == 16'd0);`
  - 이 `pmem_len_q` 값은 `state_q != STATE_EVICT && next_state_r == STATE_EVICT`의 condition이 TRUE일 때, 즉, eviction이 시작되기 직전에 (cache block내의 word 개수 - 1)만큼 할당되고, 각각의 word가 하나씩 evict될 때 마다 1씩 감소한다.
  - 따라서, `pmem_len_q` 값이 0이면, end of evict의 상태, 즉, `pmem_accept_w && pmem_last_w`의 condition이 TRUE로 되어, STATE_EVICT_WAIT로 state transition이 발생하게되는 상태라고 볼 수 있다.
- 수정 전에는 cache block size는 32B이고, word size는 4B이므로, 하나의 cache block에 8개의 word가 있었다.

    - 따라서, 이전의 code에서는 `state_q != STATE_EVICT && next_state_r == STATE_EVICT`의 condition이 TRUE일 때, `pmem_len_q` 에 `8d'7`을 할당하였다.
    - 이에 따라 cache block에 대해서 한번 eviction이 발생할 때, 앞서 확인한 것과 같이 순차적으로 8개의 word가 evict되는 것을 `debug_cache_eviction_trace.txt`에서 확인할 수 있었다.
  - 수정 후에는 cache block size가 64B이고, word size는 4B이므로, 하나의 cache block에 16개의 word가 있다.
    - 따라서, 수정 후의 code에서는 `state_q != STATE_EVICT && next_state_r == STATE_EVICT`의 condition이 TRUE일 때, `pmem_len_q` 에 `16d'15`를 할당하였다.
    - 이에 따라 cache block에 대해서 한번 eviction이 발생할 때, 앞서 확인한 것과 같이 순차적으로 16개의 word가 evict되는 것을 `debug_doubled_cache_eviction_trace.txt`에서 확인할 수 있었다.
  - 이러한 코드 수정에 따라 위와 관련있는 변수들의 크기를 8-bit → 16-bit로 변경하였다. (`outport_len_o`, `pmem_len_w`)
    - 추가적으로 `pmem_len_w`의 경우 그 값이 `pmem_len_q`에 직접 할당되어 지는 코드가 있기 때문에 `pmem_len_w`가 assign되는 코드에서도 그에 assign되는 값을 `(Condition) ? 16'd15 : 16'd0;`로 수정하였다.
- `dcache_core_tag_ram.v`
  - `dcache_core.v`의 코드를 참고하면, 하나의 cache 내에 두 개의 Tag RAM module이 포함되어 있는 구조로 구현되었음을 확인할 수 있다.
    - 이는 본 프로젝트에서 구현된 cache가 2-way associative cache이기 때문이다. `dcache_core.v` 구조는 다음과 같다.



    - 위의 그림에서, 두 개의 Tag RAM module이 포함되어 있는 것을 확인할 수 있다.
    - 수정 전 각각의 Tag RAM size는 256 x 21이었다.
      - 이때, 256은 cache 내의 set 수와 동일하며, 하나의 Tag RAM에 포함된 tag block의 개수를 뜻한다.
      - 21은 각 tag block의 size로 Tag 19-bit + valid 1-bit + dirty 1-bit으로 이루어져 있다.
      - 따라서, cache block size가 2배로 증가하여 cache 내의 set 수가 절반으로 감소한다면, Tag RAM size 또한 128 x 21로 수정해야한다.

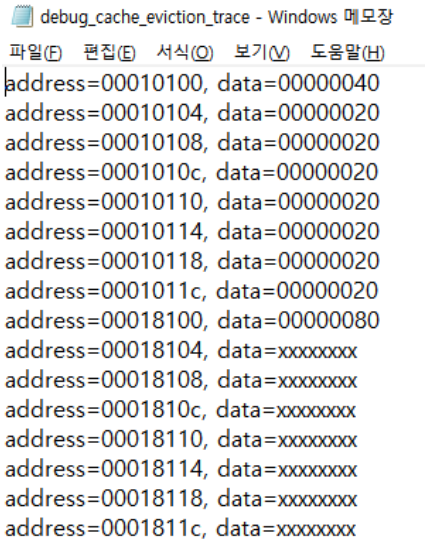
- 다시 한번 정리하면, 2-way associative cache이기 때문에 cache 내에 두 개의 Tag RAM(`u_tag0`, `u_tag1`)이 포함되어 있다.
- 또한, `u_tag0`와 `u_tag1`는 각각 256개의 tag block으로 이루어져 있고, tag RAM module의 interface port 중에서 `addr_i`를 통해 각각의 index에 맞는 tag block에 access한다.
  - 이 `addr_i`의 width는 `dcache_core.v`의 `DCACHE_TAG_REQ_LINE_W` 값과 일치한다. 즉, index width 값과 동일한 값을 가진다. (`[DCACHE_TAG_REQ_LINE_W-1:0]`)

- 따라서, 위에서 설명한 Tag RAM의 구현과 `dcache_core.v`에서 수정한 address mapping을 고려하여, 다음과 같이 `dcache_core_tag_ram.v`를 적절하게 수정하였다.

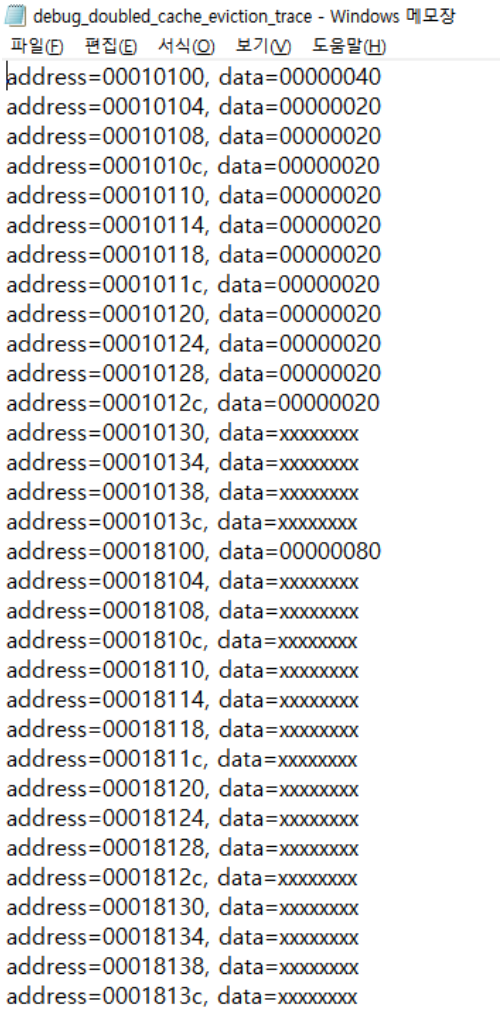
```
module dcache_core_tag_ram
(
    // Inputs
     input           clk0_i
    ,input           rst0_i
    ,input  [  6:0]  addr0_i // ,input  [  7:0]  addr0_i -> ,input  [  6:0]  addr0_i
    ,input           clk1_i
    ,input           rst1_i
    ,input  [  6:0]  addr1_i // ,input  [  7:0]  addr1_i -> ,input  [  6:0]  addr1_i
    ,input  [ 20:0]  data1_i
    ,input           wr1_i

    // Outputs
    ,output [ 20:0]  data0_o
);



//-----------------------------------------------------------------
// Tag RAM 0KB (256 x 21)
// Mode: Write First
//-----------------------------------------------------------------
/* verilator lint_off MULTIDRIVEN */
reg [20:0]   ram [127:0] /*verilator public*/; // reg [20:0]   ram [255:0] -> reg [20:0]   ram [127:0]
/* verilator lint_on MULTIDRIVEN */
```

- `addr_i`
  - 앞서 설명하였듯이, index width과 동일한 값을 가진다.
  - 따라서, `DCACHE_TAG_REQ_LINE_W`의 값이 1만큼 감소함에 따라 `addr_i`의 값도 1만큼 감소시켜준다.
  - 이는 물론 `addr0_i`, `addr1_i` 모두 반영해준다.
    - 각각 `[7:0]` → `[6:0]`으로 수정하였다.
    - 이때 `addr0_i`은 read일 때, `addr1_i`은 write일 때 사용된다.
- Tag RAM size (`ram`)
  - `reg [20:0]   ram [127:0] /*verilator public*/;`
  - 앞서 설명하였듯이, cache block size를 2배로 증가(32B → 64B)시켰고, 이에 따라 cache 내의 set 수가 절반으로 감소(256 → 128)했으므로, Tag RAM size 또한 256 x 21 → 128 x 21로 감소시켜줘야한다.
  - 따라서, Tag RAM의 size를 256 x 21 → 128 x 21로 수정하였다. (`[256:0]` → `[127:0]`)

- 마지막으로, 앞서 살펴보았듯이 Problem 1의 riscv_cache_test.S (input_inst.dat)를 통해 수정된 cache의 eviction size가 64B임을 확인하였다.
  - 그 결과는 다음과 같다.
    1. Cache block size = 32B인 경우



2. Cache block size = 64B인 경우



- 이렇게 eviction 동작이 정상적으로 수행되는지 확인함으로써, cache block size가 32B → 64B로 올바르게 수정하였음을 한번 더 검증할 수 있었다.

- 이는 `debug_cache_eviction_trace.txt` 및 `debug_doubled_cache_eviction_trace.txt`를 통해 확인할 수 있다. (해당 파일들은 Problem 2의 reference 폴더에서 확인하실 수 있습니다.)