Preferred Networks Intern Screening 2019 Coding Task for Chip Field

Changelog

- 2019-04-19 : Initial version
- 2019-04-22 : Fix the example assembly code in problem-2
- 2019-04-23 : Problem 2 Fixed the explanation of read.reg
- 2019-04-25 :
    - Problem 2 Clarified instructions what simplifications are permissable
    - Problem 2 Unified the explanations
    - Problem 2 2D array is used in the example
    - **Instead of Problem 2, if you wish, you can solve and submit Problem 1 of the Performance Optimization Field**
        - Therefore, please choose one of the following combinations to submit
            - Chip Problem 1, Chip Problem 2 and Chip Problem 3 or
            - Chip Problem 1, Performance Optimization 1 and Chip Problem 3

Notice

- Please use one of the following hardware description languages in Problem 1:
    - SystemVerilog, VerilogHDL, or VHDL
- Please tackle the task by yourself. Do not share or discuss this coding task with anyone including other applicants. Especially, do not upload your solution and/or problem description to public repository of GitHub during screening period. If we find any evidence of leakage, the applicant will be disqualified. If one applicant allows another applicant to copy answers, both applicants will be disqualified.
- We expect you to spend up to two days for this task. You can submit your work without solving all of the problems. Please do your best without neglecting your coursework.

Things to submit

Source code (Problem 1) and report (Problem 2-3)

Evaluation

It is desirable that your submission satisfies the following. (This is not mandatory. Your submission does not need to satisfy all of them if you are too busy.)

- The source code satisfies the requirements in Problem 1.
- The source code is reader friendly (identifier name, indent...)
- The submitted report is concise and easy to follow.

During the interview after the first screening process, we may ask some questions on your source code.

How to submit

- Create a zip archive with all of the submission materials and submit it on this form. Due date is Tuesday, May 7th, 2019, 12:00 JST.

Inquiry

- If you have any question regarding this problem description, please contact us at intern2019-admin@preferred.jp. An updated version will be shared with all applicants if any changes are made. Note that we cannot comment on the approach or give hints to the solution.

Task description

Problem 1

Write RTL (Register Transfer Level) code and create a testbench that shows the following waveform. In the design, signal inc_count increases and signal dec_count decreases alternatively.

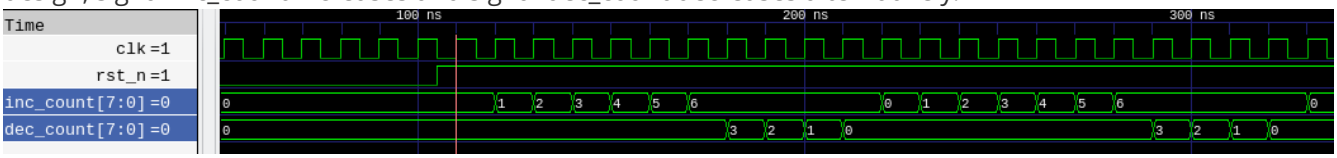

Fig. 1:Waveform

Language and filename

Either SystemVerilog, Verilog HDL, or VHDL can be used. Any language version is allowed. All modules must be written in a single file. The filename is as below.

HDL	Filename
SystemVerilog	q1.sv
Verilog HDL	q1.v
VHDL	q1.vhd

Hierarchy

The testbench and the design must have the following hierarchy.
e.g. counter_top is the module name and i_counter_top is the instance name.

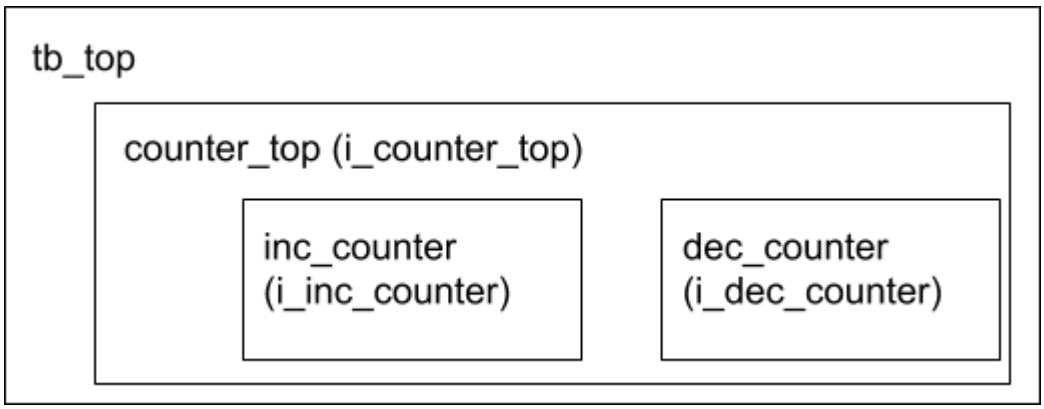

Fig. 2:Hierarchy

Module structure

Each module must have the following functionality.

Module	Role
tb_top	drive the clock and reset signal. This module is a testbench, so it is not required to be logic synthesizable.
counter_top	Instantiate and connect the module inc_counter and the module dec_counter.
inc_counter	Start checking the output of i_dec_counter once the reset is released. When the output of i_dec_counter becomes 0, then the output of i_inc_counter increases from zero to six by one per clock cycle.
dec_counter	Start checking the output of i_inc_counter once the reset is released. When the output of i_inc_counter becomes 6, then the output of i_dec_counter decreases from three to zero by one per clock cycle.

counter_top, inc_counter, and dec_counter must be ready for logic synthesis.

Signals

The module tb_top must contain at least the following four signals.

Signal	Description
clk	Clock signal
rst_n	Reset signal low active asynchronous reset
inc_count	The 8bit output from i_inc_counter via i_counter_top.
dec_count	The 8bit output from i_dec_counter via i_counter_top.

You may have any internal signals other than the four signals above.

Runtime environment

If you do not have a simulation environment, you can install open source software such as Icarus Verilog and GTK wave. Another way is to visit an online service such as EDA playground. Note that the software and web sites are not related to PFN and we can not answer any questions regarding them.

Problem 2 (Please choose either this problem or the Performance Optimization Problem 1)

The program below performs a matrix multiplication of matrices a and b. To perform this operation in hardware, we use an 8-lane SIMD accelerator (shown in Fig. 3). Fig. 3(a) shows an overview of the parallel architecture consisting of eight parallel local memories and functional units. The local memories are connected to a 64-bit wide shared bus which can transfer two 32-bit words. Since the memory is 32-bit wide the local memory will select between one of two 32-bit words using a multiplexer (see Fig. 3(b)). Each local memory has such a select signal to select appropriate word on the bus.

1. Your task is to use the operations specified in the `Instruction set` section and to describe in assembly code how they can be used so that the computation time of the matrix computation shown in `Program` is minimized. You may shorten your assembly code as in the `Scheduled assembly` example below.
2. You are allowed to do one modification in the accelerator to further minimize the computation time of the matrix computations. Support your modification with an appropriate example.

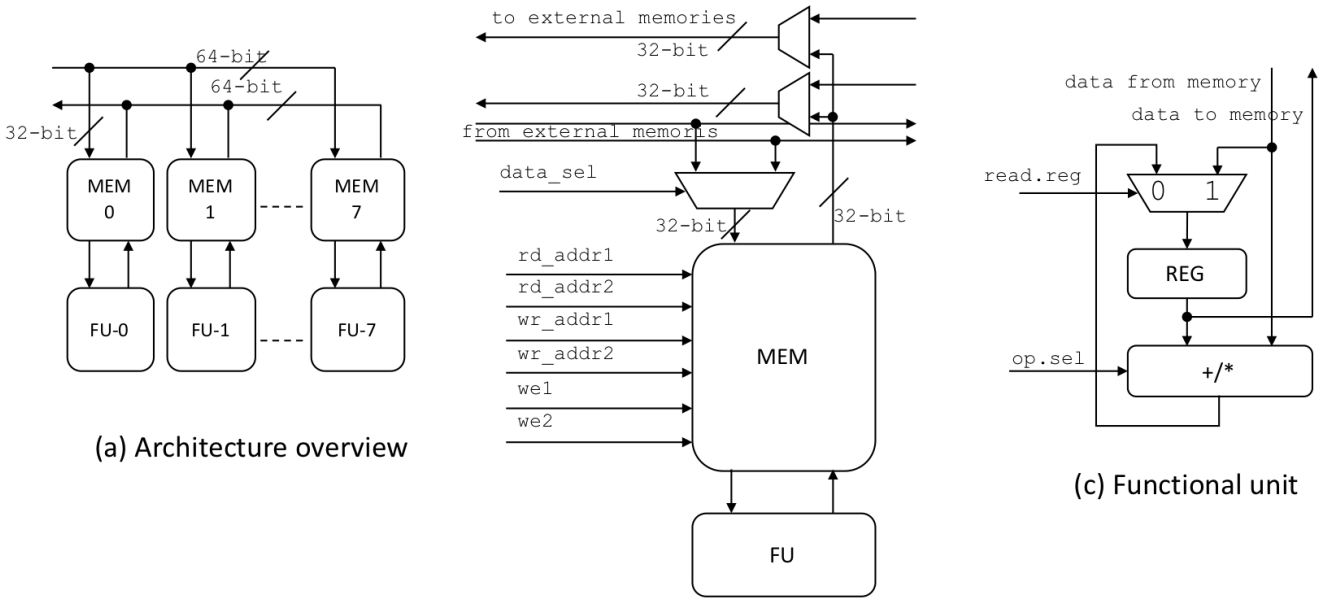

Fig. 3:Architecture

Program

```
#define N 32
int a[N][N], b[N][N], c[N][N];

void matrix_mult (int a[N][N], int b[N][N], int c[N][N]) {
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      c[i][j] = 0;
      for (int k = 0; k < N; k++) {
        c[i][j] += a[i][k] * b[k][j];
      }
    }
  }
}
```

Instruction set

Data transfer from external memory to local memory

`mov.a {a[row_idx0][col_idx],a[row_idx1][col_idx]}->{mem[lane0][addr],mem[lane1][addr]};`

This instruction transfers two words, a[row_idx0][col_idx] and a[row_idx1][col_idx] of a[N][N], from the external memory to the two local memories. The first pair of indices, lane0 and lane1, of mem represent the lane number and the second pair of indices, addr, is the target address of each local memory.

`mov.a {a[row_idx0][start_idx,len],a[row_idx1][start_idx,len]}->{mem[lane0][addr0,len],mem[lane1][addr1,len]};`

This instruction transfers two times len number of words data from the external memory to the local memories as follows: len words, a[row_idx0][col_idx], a[row_idx0][col_idx+1], ..., a[row_idx0][col_idx+len-1], moved to the local memories mem[lane0][addr0], mem[lane0][addr0+1], mem[lane0][addr0+2], ..., mem[lane0][addr0+len-1]. As before, the first indices represent the lane number, and the addr0 and addr1 in second indices are the start write address of the each local memory.

`mov.b a[row_idx][col_idx]->mem[][addr];`

This instruction broadcasts the content of a[row_idx][col_idx] to the local memories of the accelerator. Since the data will broadcast, the lane address in the first index is left empty. The second index is the target address of the local memories.

`mov.b a[row_idx][col_idx,len]->mem[][addr,len];`

This instruction broadcasts len words from the a[row_idx][col_idx] to a[row_idx][col_idx+len-1] to the local memories. The start address and the length are mentioned in the second indices as addr and len.

Data transfer from local memory to off-chip memory

`mov.h {mem[lane0][addr],mem[lane1][addr]}->{a[row_idx0][col_idx],a[row_idx1][col_idx]};`

This instruction transfers content of local memories to the external memory. The first indices, lane0 and lane1, is the lane number and the second indices, addr, is the read address whereas row_idx0 and col_idx as well as row_idx1 and col_idx are the target addresses in the external memory.

Operations for the functional unit

The following instructions are performed in parallel in all the functional units.

`read.reg mem[addr]; //reg = mem[addr]`

This instruction loads the local memory at addr and stores the content to the register.

`read.add mem[addr]; //reg = reg + mem[addr]`

This instruction performs an addition of the data received from the local memories at addr and the register content. The result is written back to the register.

`read.mult mem[addr]; //reg = reg * mem[addr]`

This instruction performs a multiplication of the data received from the local memories at addr and the register content. The result is written back to the register.

`write mem[addr];`

This instruction writes the content of the register back to the local memories at address addr.

Constraints of the architecture

- The number of the parallel functional units and local memory pairs is 8.
- Each local memory has a size of 128x32bit, and has two read and write ports.
- One write port is connected with the shared bus to store the data on the memory from external memory and the other is with the functional unit (see Fig. 3(b)). Likewise, each of the read ports are connected with the shared bus and functional unit.
- Only one move command can be issued per cycle. I.e. a mov.[a,b] and mov.h command cannot be issued in parallel.
- However, mov.[a,b,h] and read/write commands may be issued in parallel
- A multiword transfer, e.g. `mov.b a[row_idx][col_idx,len]->mem[][addr,len];` takes always len cycles to complete. Within this period, no other move commands can be issued.
- REG(Fig. 3(c)) of each FU can contain a single word.
- Keep in mind that it is a SIMD architecture, pairs of move and compute commands will be executed on all units (SIMD == Single Instruction Multiple Data)

Example

2D array addition

```
#define H    32
#define W    16
int v1[H][W], v2[H][W], r[H][W];
void array2d_add (int v1[H][W], int v2[H][W], int r[H][W]) {
  for (unsigned y = 0; y < H; ++y) {
    for (unsigned x = 0; x < W; ++x) {
      r[y][x] = v1[y][x] + v2[y][x];
    }
  }
}
```

Memory data partitioning

There are 8-lanes and each lane is equipped with a 128-word memory to store the input and output. The total memory needed for this code is 3x32*16=1536, and the total local memory size is 8x128=1024, which is smaller than the total memory requirement.

Thus, first we partition each array into 2 blocks, and each block's size per lane is 3x32*16/2/8=96-word. This data is then distributed among the 8-parallel local memories. The modified code is as follows:

```
#define H    32
#define W    16
int v1[H][W], v2[H][W], r[H][W];
void array2d_add (int v1[H][W], int v2[H][W], int r[H][W]) {
  for (unsigned i = 0; i < 2; ++i) {
    for (unsigned y = 0; y < H/2; ++y) {
      for (unsigned x = 0; x < W; ++x) {
        r[(i * H/2) + y][x] = v1[(i * H/2) + y][x] + v2[(i * H/2) + y][x];
      }
    }
  }
}
```

Scheduled assembly

The following shows an example assembly code for the above program.

```
//256-word data transfer time from v1[0-15][0-15] is 128 cycles
mov.a  {v1[0][0, 16], v1[2][0, 16]} -> {mem[0][0, 16],mem[1][0, 16]};
mov.a  {v1[1][0, 16], v1[3][0, 16]} -> {mem[0][16, 16],mem[1][16, 16]};
mov.a  {v1[4][0, 16], v1[6][0, 16]} -> {mem[2][0, 16],mem[3][0, 16]};
mov.a  {v1[5][0, 16], v1[7][0, 16]} -> {mem[2][16, 16],mem[3][16, 16]};
mov.a  {v1[8][0, 16], v1[10][0, 16]} -> {mem[4][0, 16],mem[5][0, 16]};
mov.a  {v1[9][0, 16], v1[11][0, 16]} -> {mem[4][16, 16],mem[5][16, 16]};
mov.a  {v1[12][0, 16], v1[14][0, 16]} -> {mem[6][0, 16],mem[7][0, 16]};
mov.a  {v1[13][0, 16], v1[15][0, 16]} -> {mem[6][16, 16],mem[7][16, 16]};


//256-word data transfer time from v2[0-15][0-15] is 128 cycles
mov.a  {v2[0][0, 16], v2[2][0, 16]} -> {mem[0][32, 16],mem[1][32, 16]};
mov.a  {v2[1][0, 16], v2[3][0, 16]} -> {mem[0][48, 16],mem[1][48, 16]};
----
mov.a  {v2[12][0, 16], v2[14][0, 16]} -> {mem[6][32, 16],mem[7][32, 16]};
mov.a  {v2[13][0, 16], v2[15][0, 16]} -> {mem[6][48, 16],mem[7][48, 16]};


//first 32 words computation time 96 cycles
//iteration 0    (3 cycles)
read.reg    mem[0];
read.add    mem[32];
write        mem[64];

//iteration 1 (3 cycles)
read.reg    mem[1];
```

```
read.add    mem[33];
write        mem[65];
----
//iteration    31
read.reg    mem[31];
read.add    mem[63];
write        mem[95];

//32 results are transferred from local memories to external memory
mov.h  {mem[0][64], mem[1][64]}->{r[0][0], r[2][0]};
----
mov.h  {mem[0][79], mem[1][79]}->{r[0][15], r[2][15]};
mov.h  {mem[0][80], mem[1][80]}->{r[1][0], r[3][0]};
----
mov.h  {mem[0][95], mem[1][95]}->{r[1][15], r[3][15]};

----

mov.h  {mem[6][64], mem[7][64]}->{r[12][0], r[14][0]};
----
mov.h  {mem[6][79], mem[7][79]}->{r[12][15], r[14][15]};
mov.h  {mem[6][80], mem[7][80]}->{r[13][0], r[15][0]};
----
mov.h  {mem[6][95], mem[7][95]}->{r[13][15], r[15][15]};



//similarly the remaining computation are performed as follows:
mov.a  {v1[16][0, 16], v1[18][0, 16]} -> {mem[0][0, 16],mem[1][0, 16]};
mov.a  {v1[17][0, 16], v1[19][0, 16]} -> {mem[0][16, 16],mem[1][16, 16]};
----
mov.a  {v1[28][0, 16], v1[30][0, 16]} -> {mem[6][0, 16],mem[7][0, 16]};
mov.a  {v1[29][0, 16], v1[31][0, 16]} -> {mem[6][16, 16],mem[7][16, 16]};


//256-word data transfer time from v2[0-255] is 128 cycles
mov.a  {v2[16][0, 16], v2[18][0, 16]} -> {mem[0][32, 16],mem[1][32, 16]};
mov.a  {v2[17][0, 16], v2[19][0, 16]} -> {mem[0][48, 16],mem[1][48, 16]};
----
mov.a  {v2[28][0, 16], v2[30][0, 16]} -> {mem[6][32, 16],mem[7][32, 16]};
mov.a  {v2[29][0, 16], v2[31][0, 16]} -> {mem[6][48, 16],mem[7][48, 16]};


//first 32 words computation time 96 cycles
//iteration 0    (3 cycles)
read.reg    mem[0];
read.add    mem[32];
write        mem[64];

----
//iteration    31
read.reg    mem[31];
read.add    mem[63];
```

```
write        mem[95];


//32 results are transferred from local memories to external memory
mov.h  {mem[0][64], mem[1][64]}->{r[16][0], r[18][0]};
----
mov.h  {mem[0][79], mem[1][79]}->{r[16][15], r[18][15]};
mov.h  {mem[0][80], mem[1][80]}->{r[17][0], r[19][0]};
----
mov.h  {mem[0][95], mem[1][95]}->{r[17][15], r[19][15]};

----

mov.h  {mem[6][64], mem[7][64]}->{r[28][0], r[30][0]};
----
mov.h  {mem[6][79], mem[7][79]}->{r[28][15], r[30][15]};
mov.h  {mem[6][80], mem[7][80]}->{r[29][0], r[31][0]};
----
mov.h  {mem[6][95], mem[7][95]}->{r[29][15], r[31][15]};
```

Problem 3

1. Tell us about your favorite computer architecture and explain in detail why you like it. (if you don't have a favorite architecture, just pick one and explain your choice) (max. 500 words)
2. What are the strengths and weaknesses of the architecture you described above? (max. 500 words)
3. What is the relationship between computer architecture and power consumption? (max. 500 words)