Lab #3: PE implementation & BRAM modeling

03/22/2018

4190.309A: Hardware System Design
(Spring 2018)

FAQ : “*generate for*” vs “*for*”

- **Both statements do loop-unrolling**
  - **“For loop” in verilog is different from that of C!**
- **However, the times of execution must be determined for synthesizable design.**
  - **Don’t forget we are designing the hardware. Size of the hardware must be determined before synthesis.**

FAQ : “*generate for*” vs “*for*”

- **“For” statement can change elements in an array, or change the same value in multiple times**

- **ex) tb_mac in Lab1**

```
`timescale 1ns / 1ps
module tb_mac();
    parameter BITWIDTH = 32;

    //for my IP
    reg [BITWIDTH-1:0] ain;
    reg [BITWIDTH-1:0] bin;
    reg clk;
    reg en;
    wire [2*BITWIDTH-1:0] dout;
    wire    overflow;

    //for test
    integer i;
        //random test vector generation
    initial begin
        clk<=0;
        en<=0;
        #30;
        en<=1;
      for(i=0; i<32; i=i+1) begin
        ain = $urandom%(2**31);
        bin = $urandom%(2**31);
        #10;
      end
    end

    //my IP
    my_mac #(BITWIDTH) MY_MAC(
        .ain(ain),
        .bin(bin),
        .en(en),
        .clk(clk),
        .dout(dout)
    );

    always #5 clk = ~clk;

endmodule
```

FAQ : “*generate for*” vs “*for*”

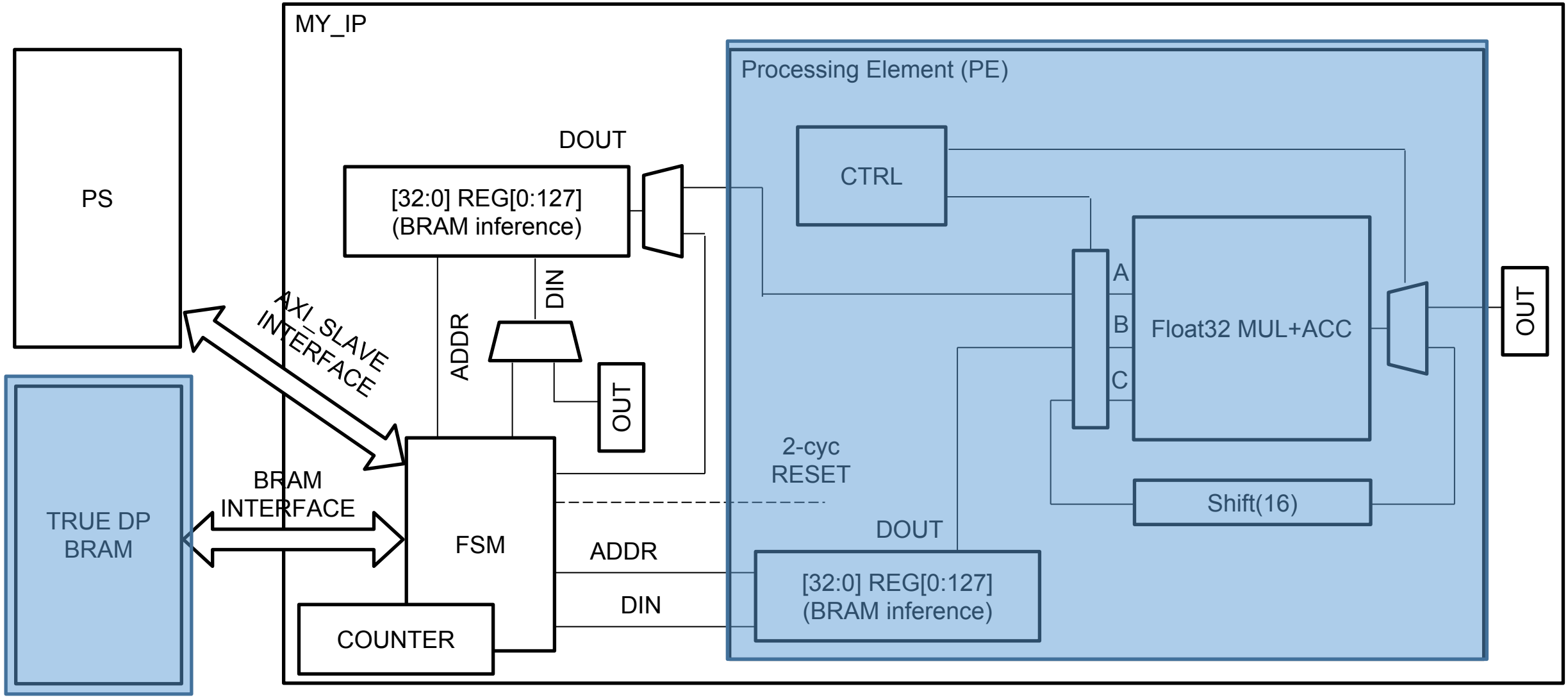
- **“generate” statement allows Verilog code to be generated for creation of parameterized models at compile time.**

- **Generated instantiations can be**
  - **Modules**
  - **User defined primitives**
  - **Verilog gate primitives**
  - **Continuous assignments**
  - **initial and always blocks**

- **ex) adder_array.v TA implemented**

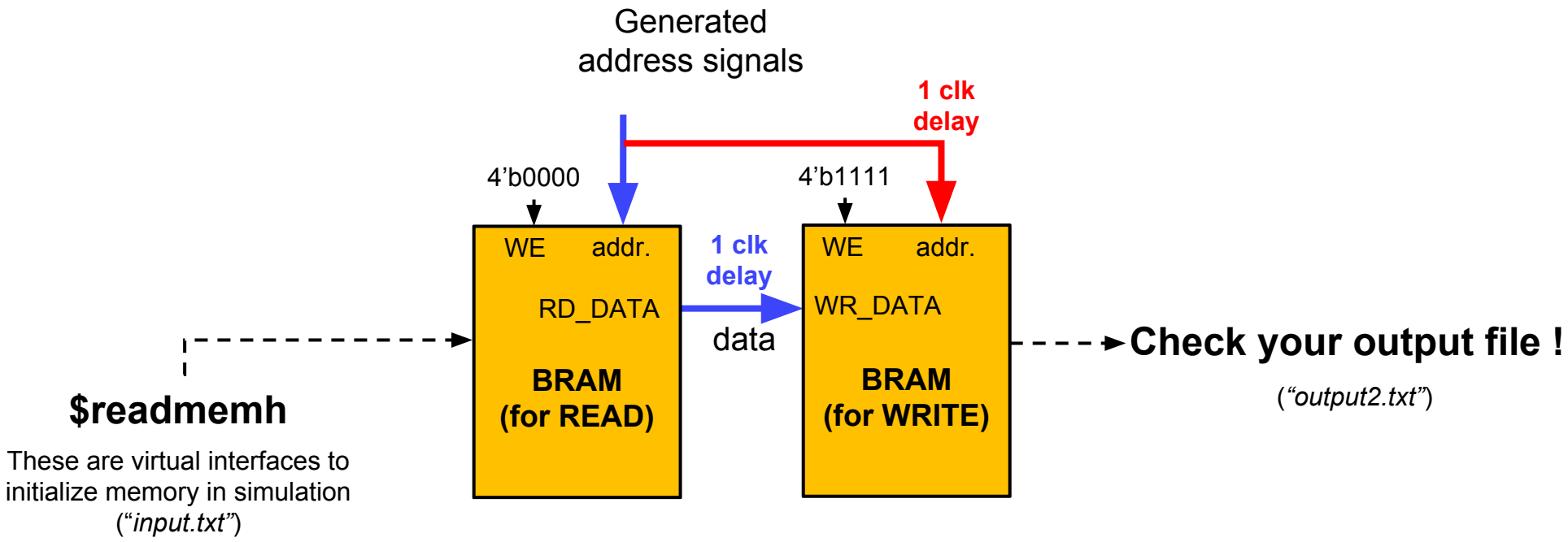
```
genvar i;
generate
  for(i=0; i<NUM_ADDER; i=i+1) begin: INST_ADD
    my_add add_unit (
      .ain(ain[i]),
      .bin(bin[i]),
      .dout(dout[i]),
      .overflow(overflow[i])
    );
  end
endgenerate
```

Custom HW Example: Matrix Multiplication IP



Practice #1: BRAM Implementation

BRAM Implementation Overview



Block diagram of BRAM testbench
(Minor signals were omitted)

I/O functions of Verilog

- `$readmemh("file_name", destination[, start_addr, end_addr]); // [] : optional params`
  - **Read hexadecimal data from file, and write to memory array.**
    - **ex) reg [31:0] mem [0:15];**
      $readmemh("input.txt", mem);
  - **Using an absolute path as file_name is also possible**
    - **ex) $readmemh("/home/k16diablo/input.txt", mem);**
    - **To get an absolute path, use 'pwd' command line**
  - **Useful when initializing the block memory**

- `$writememh("file_name", destination[, start_addr, end_addr]); // [] : optional params`
  - **Read data from destination, and write to file in hexadecimal format.**
    - **ex) reg [31:0] mem [0:15];**
      $writememh("output.txt", mem);
  - **Using an absolute path as file_name is also possible**
    - **ex) $writememh("/home/k16diablo/output.txt", mem);**

Practice #1: Implement BRAM model

- **Specification**
  - **7 input pins**
    - **BRAM_ADDR : 6 bit address pin.**
    - **BRAM_CLK : Clock signal pin.**
    - **BRAM_WRDATA : 32 bit data pin for write operation.**
    - **BRAM_EN : BRAM enable signal pin. Both read and write operation are available only when BRAM_EN == 1.**
    - **BRAM_RST : Negative edge reset signal. Reset BRAM_RDDATA and all data in BRAM to zero.**
    - **BRAM_WE : 4 bit BRAM Write Enable signal pin. Each bit points where to write data.**
      ex) (BRAM_WE[i] == 1) enables write BRAM_WRDATA[8*(i+1)-1:8*i] to mem[addr][8*(i+1)-1:8*i]
    - **done : Write data in BRAM to output file.**
  - **1 output pin**
    - **BRAM_RDDATA : 32 bit data pin for read operation.**
  - **Both read and write operation takes 1 cycle.**

Practice #1: Implement BRAM testbench

- **All you have to do is verify that your BRAM model read and write data in 1 cycle.**
  - **To do**
    - **Download “input.txt” from eTL. It has hexadecimal numbers from 0 to 15.**
    - **Implement BRAM model.**
    - **Design testbench that can verify your BRAM model.**
      - **You will need 2 BRAM model in testbench. One for read operation and the other for write operation.**
      - **Don’t forget both read and write operation cost 1 cycle!**
  - **Tips for verification**
    - **You may need a register to store the data to be read of previous cycle. (dout)**
    - **TA will check both output file and simulation result.**
    - **Use “$stop” to see whether your BRAM stores data in 1 cycle. “mem” cannot be added to wave window since it is too big!**

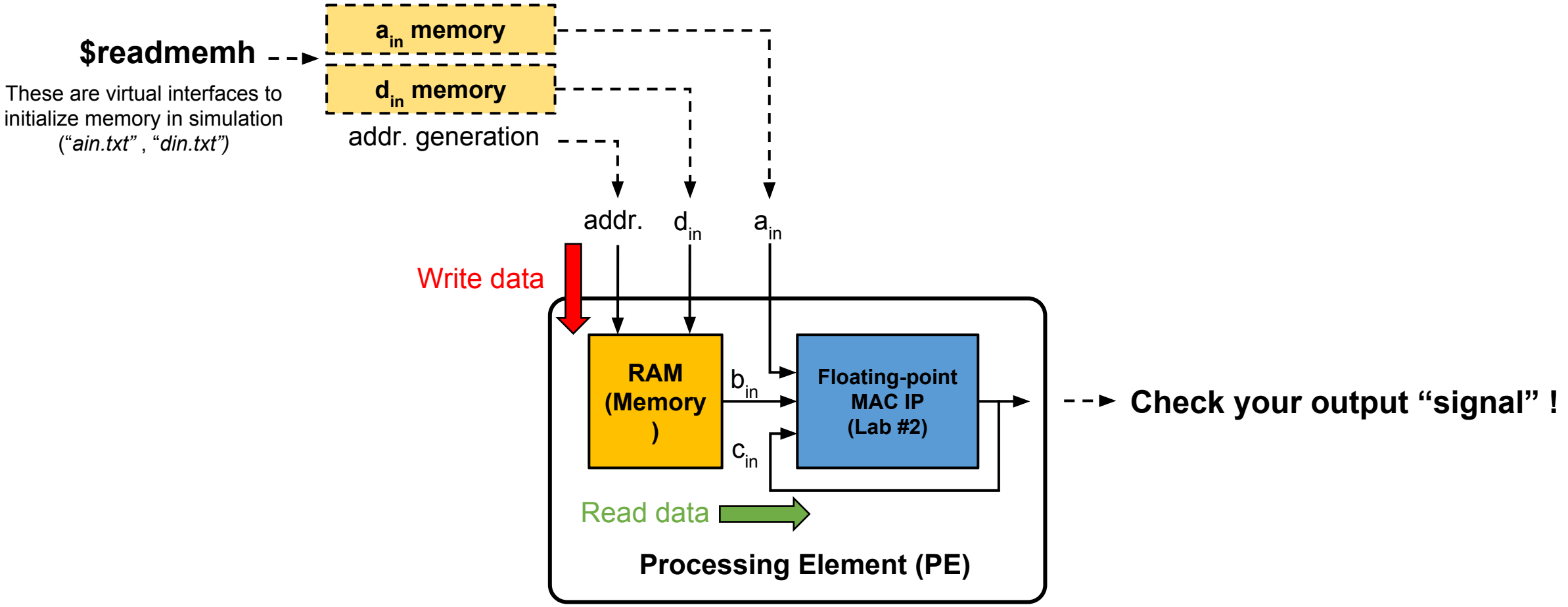
Skeleton of BRAM Model

```
module my_bram # (
   parameter integer BRAM_ADDR_WIDTH = 6, // 2**6 = 4x16
   parameter INIT_FILE = “input.txt“,
   parameter OUT_FILE = “output.txt”
 )(
   input wire [BRAM_ADDR_WIDTH-1:0] BRAM_ADDR,
   input wire BRAM_CLK,
   input wire [31:0] BRAM_WRDATA,
   output reg [31:0] BRAM_RDDATA,
   input wire BRAM_EN,
   input wire BRAM_RST,
   input wire [3:0] BRAM_WE,
   input wire done
 );
   reg [31:0] mem[0:15];
   wire [BRAM_ADDR_WIDTH-3:0] addr = BRAM_ADDR[BRAM_ADDR_WIDTH-1:2];
   reg [31:0] dout;
   initial begin
    if (INIT_FILE != "") begin
      $readmemh(INIT_FILE, mem);
    end
    wait(done)
      $writememh(OUT_FILE, mem);
   end
  // code for BRAM implementation
   …
endmodule
```

You should fill out the "code for BRAM implementation" part!

Practice #2: PE Implementation

Processing Element (PE) Implementation Overview



Block diagram of PE testbench (Minor signals were omitted)

Practice #2 : Implement Processing Element

- **Specification**
  - **7 input pins**
    - **aclk : Clock signal pin.**
    - **areset : Negative edge reset signal. Reset bin, cin, valid bits and all data in peram to zero.**
    - **ain : 32 bit data pin for 1st input of MAC.**
    - **din : 32 bit data pin to be written in peram.**
    - **addr : L_RAM_SIZE pin specifying which peram entry will be accessed (both for write and read).**
      i.e. if we == 1, din is written to peram[addr]
      if we == 0, peram[addr] is written to bin
    - **we : peram Write Enable pin.**
    - **valid : Valid signal of input data. Validate the inputs of MAC only when this signal is on.**
  - **2 output pins**
    - **dvalid : Output valid signal from MAC.**
    - **dout : Output data from MAC.**
  - **MAC gets 3 input data :**
    - **ain, a 32 bit wire from outside of PE**
    - **bin, a 32 bit register which was read from peram**
    - **cin, a 32 bit register which was the output of previous MAC operation**

Practice #2 : Implement Processing Element

- **All you have to do is verify that your BRAM model read and write data in 1 cycle.**
  - **To do**
    - **Download “ain.txt” and “din.txt” from eTL. They have hexadecimal numbers from 0.1 to 1.6.**
    - **Implement PE model.**
    - **Design testbench that can verify your PE model.**
  - **Tips for verification**
    - **You have to initialize peram by controlling din and addr.**
    - **Note that your inputs are valid only during dvalid is on!**
      - **You may use ‘wait’ statement.**
    - **TA will check both output file and simulation result.**

Skeleton of PE Model

```
module my_pe #(
  parameter L_RAM_SIZE=6
)(
  input aclk,
  input areset,
  input [31:0] ain,
  input [31:0] din,
  input [L_RAM_SIZE - 1:0] addr,
  input we,
  input valid,
  output dvalid,
  output [31:0] dout
);
  (* ram_sytle = "block" *) reg [31:0] peram[0:2**L_RAM_SIZE - 1];
  reg [31:0] bin;
  reg [31:0] cin;
  reg a_tvalid, b_tvalid, c_tvalid; // will be propagated from valid

  // code for PE implementation
  …
endmodule
```

You should fill out the "code for PE implementation" part!