**Project 2 report**

Alexis Yang, Ning Du, Jaeyoon Kim

**Introduction:**

In this project we are asked to make a 32 KB 2-way associate cache. Each way contains 1024 sets and each set contains the tag of the memory location and the block which contains four words associated with this tag and index of the set. We are asked to follow the write back policy and the write back policy we use is LRU, lest recent used.

**Illustration of instruction:**

The only two instructions that are involved in the project are lw and sw. Both of the instruction new does not interact with the memory directly but access cache instead. In memory stage of the pipeline the controls and input for lw and sw are still the same.

lw now takes 1 cycle if it gets a direct hit in the cache. It may also take 21 cycles to bring the corresponding block of the memory and then read from the cache. In the worst case, it can take up to 41 cycle to load a word if both ways are filled with data and we need to store one of the block to memory which takes 20 cycles and then take 21 cycles to bring the new block back to cache and read takes another 1 cycle.

sw also take also take 1 cycle if hits, 21 cycles if one of the way is vacant or 41 cycles if both of the ways are filled and none of the them hit.

**Design methodology:**

We designed a state machine to control the functionality of the cache. There are 5 different states. You stay at state 0 when you keep getting hits, so it only takes 1 cycle to read or write the word to the cache. State 1 is when we need to read a block from memory to way1 of the cache. State 2 is when we need to write back a block from cache to the memory. State 4 is when a block needs to be brought memory to way0 of the cache. State 8 is when a block needs to be written back to memory from way0.

The general process goes like this:

1. check if hit, if hit read or write the word in 1 cycle.

2. if not hit, check the valid bit of the corresponding set of both ways. If way0 is empty bring the corresponding block from memory and then read or write to it in cache. If way1 is empty, bring the block in to way1 and read or write to it in cache.

3. if both ways have valid bit set to 1, check the value in lru register. First store the block in the way indicated by lru register back to memory if the dirty bit is set to 1, and then bring the new block from the memory. Read or write in 1 cycle after the block is in the cache.

**Step1:** In previous lab, you designed a 32-bit MIPS. Open your project in Modelsim Environment.

**Step2:** Create a new source and name it cache\_wb.

**Code:**

module cache\_wb(input clk, reset, mem\_ready, //mem\_ready become 1 after 20 cycles of memory delay

input read0, write0, //read, write is connected to the controls memreadM and memwriteM

input [31:0] addr\_cpu0, data\_cpu0, //addr\_cpu is connected to OutM, data\_cpuis connected to writedataM

input [127:0] data\_from\_mem, //the block of data read from memeory

output hit, mem\_read, mem\_write, cache\_stall, //cache\_stall stall the entire pipeline when memory access needed

output reg [31:0] mem\_addr, //address of memory location cache wants access

output [31:0] data\_out, //connected to readdataM

output reg [127:0] data\_to\_mem); //block of data to write to memory from cache

reg read1, write1, delay\_cache\_stall;

reg [31:0] addr\_cpu1, data\_cpu1;

always @(posedge clk) begin

delay\_cache\_stall <= cache\_stall;

if(~delay\_cache\_stall) begin

read1 <= read0;

write1 <= write0;

addr\_cpu1 <= addr\_cpu0;

data\_cpu1 <= data\_cpu0;

end

end

wire read; assign read = (delay\_cache\_stall) ? read1 : read0;

wire write; assign write = (delay\_cache\_stall) ? write1 : write0;

wire [31:0] addr\_cpu; assign addr\_cpu = (delay\_cache\_stall) ? addr\_cpu1 : addr\_cpu0;

wire [31:0] data\_cpu; assign data\_cpu = (delay\_cache\_stall) ? data\_cpu1 : data\_cpu0;

wire cpu\_write0;

wire cpu\_write1;

reg cw\_mem0, mem\_wc0, cw\_mem1, mem\_wc1;

wire hit0, hit1, valid0, valid1, dirty0, dirty1;

wire [31:0] addr\_out0, addr\_out1, word\_out0, word\_out1;

wire [127:0] block\_out0, block\_out1;

assign cpu\_write0 = (write & hit0) ? 1 : 0; //if way 0 is hitted in the first cycle, write to cache directly

assign cpu\_write1 = (write & hit1) ? 1 : 0; //if way 1 is hitted in the first cycle, write to cache directly

reg [3:0] state; //state can take number 8, 4, 2, 1, 0

//the two ways of the cache

way way0(clk, reset, cpu\_write0, cw\_mem0, mem\_wc0, mem\_ready, 1'b0,

state,

addr\_cpu, data\_cpu,

data\_from\_mem,

hit0, valid0, dirty0,

addr\_out0, word\_out0,

block\_out0);

way way1(clk, reset, cpu\_write1, cw\_mem1, mem\_wc1, mem\_ready, 1'b1,

state,

addr\_cpu, data\_cpu,

data\_from\_mem,

hit1, valid1, dirty1,

addr\_out1, word\_out1,

block\_out1);

assign hit = hit0 | hit1; //hit is 1 when one of the ways get a hit

//the following logic makes sure mem\_read only stay high for one cycle

assign mem\_read = (state == 0 & (mem\_wc0 | mem\_wc1)) | ((mem\_wc0 | mem\_wc1) & (state == 4'b1000 | state == 4'b0010) & mem\_ready);

//the following logic makes sure mem\_write only stay high for one cyle

assign mem\_write = (state == 0 & (cw\_mem0 | cw\_mem1)) | ((cw\_mem0 | cw\_mem1) & (state == 4'b1000 | state == 4'b0010) & mem\_ready);

assign cache\_stall = (read | write) & ~hit ; //stall cache when we need to read or write but we did not hit

assign data\_out = (hit0) ? word\_out0[31:0] :

(hit1) ? word\_out1[31:0] : 32'bx;

reg lru [0:1023]; //lru register

reg p\_lru;

wire[9:0] index; assign index = addr\_cpu[13:4];

integer i;

always @(posedge clk) begin

if(reset) begin

state <= 0;

delay\_cache\_stall <= 0;

for(i = 0; i < 1024; i = i + 1) lru[i] <= 0;

end

else if(read | write) begin

if(hit) begin

state <= 0; //if hit, state stays 0

lru[index] <= (hit0) ? 1 : 0;

end

else if (~valid0 | (mem\_ready & state == 4'b1000) | ~p\_lru & ~dirty0)

state = 4'b0100; //state 4, memory writes the requested block to way0

else if (~valid1 | (mem\_ready & state == 4'b0010) | p\_lru & ~dirty1)

state = 4'b0001; //state 1, memory writes the requested block to way1

else if (~p\_lru & dirty0)

state = 4'b1000; //state 8, way0 writes a block to memory

else if (p\_lru & dirty1)

state = 4'b0010; //state 2, way1 writes a block to memory

end

end

always @(\*) begin

if(read | write) begin

if(hit) begin

cw\_mem0 <= 0; mem\_wc0 <= 0; cw\_mem1 <= 0; mem\_wc1 <= 0;

p\_lru <= (hit0) ? 1 : 0;

end

else if (~valid0 | (mem\_ready & state == 4'b1000) | ~p\_lru & ~dirty0) begin

cw\_mem0 <= 0; mem\_wc0 <= 1; cw\_mem1 <= 0; mem\_wc1 <= 0; //control signals to make memory write to way0

mem\_addr <= addr\_cpu;

p\_lru <= lru[index];

end

else if (~valid1 | (mem\_ready & state == 4'b0010) | p\_lru & ~dirty1) begin

cw\_mem0 <= 0; mem\_wc0 <= 0; cw\_mem1 <= 0; mem\_wc1 <= 1; //control signals to make memory write to way1

mem\_addr <= addr\_cpu;

p\_lru <= lru[index];

end

else if (~p\_lru & dirty0) begin

cw\_mem0 <= 1; mem\_wc0 <= 0; cw\_mem1 <= 0; mem\_wc1 <= 0; //control signals to make way0 write to memory

mem\_addr <= addr\_out0;

data\_to\_mem <= block\_out0;

p\_lru <= lru[index];

end

else if (p\_lru & dirty1) begin

cw\_mem0 <= 0; mem\_wc0 <= 0; cw\_mem1 <= 1; mem\_wc1 <= 0; //control signals to make way1 write to memory

mem\_addr <= addr\_out1;

data\_to\_mem <= block\_out1;

p\_lru <= lru[index];

end

end

end

endmodule

module way(input clk, reset, cpu\_write, cw\_mem, mem\_wc, mem\_ready, a, //a = 0 creats way0 module, a = 1 creats way1 module

input [3:0] state,

input [31:0] addr\_in, word\_in, //word\_in is the word from the pipeline

input [127:0] block\_in, //block\_in is the block from memory

output hit, valid, dirty, //hit, valid bit and dirty bit output

output [31:0] addr\_out, word\_out, //addr\_out is address for memory, word\_out is the word output to the pipeline

output [127:0] block\_out);//block\_out is the block written to memory

wire [17:0] tag; assign tag = addr\_in[31:14]; //18 tag bits

wire [9:0] index; assign index = addr\_in[13:4]; //10 index bits

wire [1:0] offset; assign offset = addr\_in[3:2]; //2 offset bits

reg [147:0] sets [0:1023];

assign hit = ((tag == sets[index][145:128]) & sets[index][147]);

assign valid = sets[index][147];

assign dirty = sets[index][146];

assign addr\_out = {sets[index][145:128], index, offset, 2'b00};

mux4#(32) mux (sets[index][31:0], sets[index][63:32], sets[index][95:64], sets[index][127:96], offset, word\_out);

assign block\_out = sets[index][127:0];

integer i;

always @(posedge clk) begin

if(reset) begin

for(i = 0; i < 1024; i = i + 1) sets[i] <= 0;

end

else if(cpu\_write & hit) begin //write the word directly if we get a hit

sets[index][146] <= 1; //dirty bit

sets[index][145:128] <= tag;

case(offset)

2'b00: sets[index][31:0] = word\_in;

2'b01: sets[index][63:32] = word\_in;

2'b10: sets[index][95:64] = word\_in;

2'b11: sets[index][127:96] = word\_in;

endcase

end

else if( ((~a & state == 4'b1000) | (a & state == 4'b0010)) & mem\_ready) begin //write a block back to memory

sets[index][147] <= 0; //valid bit

end

else if( ((~a & state == 4'b0100) | (a & state == 4'b0001)) & mem\_ready) begin //write a block from memory to cache

sets[index][147] <= 1; //valid bit

sets[index][146] <= 0; //dirty bit

sets[index][145:128] <= tag;

sets[index][127:0] <= block\_in;

end

end

endmodule

module mux4#(parameter width = 1) //four input mux for select the output block

(input [width - 1:0] d0, d1, d2, d3,

input [1:0] sel,

output [width - 1:0] y);

assign y = (sel == 2'd0) ? d0 :

(sel == 2'd1) ? d1 :

(sel == 2'd2) ? d2 :

(sel == 2'd3) ? d3 : {width{1'bx}};

endmodule

**Testbench:**

`timescale 1ns / 1ps

module cache\_wb\_tb;

reg clk = 0, reset = 1;

reg read = 0, write = 0;

reg [31:0] addr\_cpu, data\_cpu;

wire mem\_ready;

wire [127:0] data\_from\_mem;

wire hit, mem\_read, mem\_write, cache\_stall;

wire [31:0] mem\_addr, data\_out;

wire [127:0] data\_to\_mem;

integer i = 0;

cache\_wb cache(clk, reset, mem\_ready,

read, write,

addr\_cpu, data\_cpu,

data\_from\_mem,

hit, mem\_read, mem\_write, cache\_stall,

mem\_addr,

data\_out,

data\_to\_mem);

data\_memory mem(clk, mem\_read, mem\_write,

mem\_addr,

data\_to\_mem,

mem\_ready,

data\_from\_mem);

initial begin // initialize test

#10;

reset = 0;

end

always begin // generate clock to sequence tests

clk = 0; #5;

clk = 1; #5;

end

always @(\*) begin

case(i)

1: begin read = 0; write = 1; addr\_cpu = 32'b000000000000000000\_0000000000\_00\_00; data\_cpu = 32'ha000\_000b; end //way0, set0, block0, write 0

2: begin read = 0; write = 1; addr\_cpu = 32'b000000000000000000\_0000000000\_11\_00; data\_cpu = 32'ha000\_001b; end //way0, set0, block3, write 1

3: begin read = 0; write = 1; addr\_cpu = 32'b000000000000000000\_0000000001\_00\_00; data\_cpu = 32'ha000\_002b; end //way0, set1, block0, write 2

4: begin read = 0; write = 1; addr\_cpu = 32'b000000000000000000\_0000000001\_11\_00; data\_cpu = 32'ha000\_003b; end //way0, set1, block3, write 3

5: begin read = 1; write = 0; addr\_cpu = 32'b000000000000000000\_0000000000\_00\_00; data\_cpu = 32'hx; end //read 0

6: begin read = 1; write = 0; addr\_cpu = 32'b000000000000000000\_0000000000\_11\_00; data\_cpu = 32'hx; end //read 1

7: begin read = 1; write = 0; addr\_cpu = 32'b000000000000000000\_0000000001\_00\_00; data\_cpu = 32'hx; end //read 2

8: begin read = 1; write = 0; addr\_cpu = 32'b000000000000000000\_0000000001\_11\_00; data\_cpu = 32'hx; end //read 3

9: begin read = 0; write = 1; addr\_cpu = 32'b000000000000000001\_0000000000\_00\_00; data\_cpu = 32'ha000\_004b; end //way1, set0, block0, write 4

10: begin read = 0; write = 1; addr\_cpu = 32'b000000000000000001\_0000000000\_11\_00; data\_cpu = 32'ha000\_005b; end //way1, set0, block3, write 5

11: begin read = 0; write = 1; addr\_cpu = 32'b000000000000000010\_0000000000\_00\_00; data\_cpu = 32'ha000\_006b; end //way0, set0, block0, write 6

12: begin read = 0; write = 1; addr\_cpu = 32'b000000000000000010\_0000000000\_11\_00; data\_cpu = 32'ha000\_007b; end //way0, set0, block3, write 7

13: begin read = 1; write = 0; addr\_cpu = 32'b000000000000000000\_0000000000\_00\_00; data\_cpu = 32'hx; end //read 0

14: begin read = 1; write = 0; addr\_cpu = 32'b000000000000000000\_0000000000\_11\_00; data\_cpu = 32'hx; end //read 1

15: begin read = 1; write = 0; addr\_cpu = 32'b000000000000000001\_0000000000\_00\_00; data\_cpu = 32'hx; end //read 4

16: begin read = 1; write = 0; addr\_cpu = 32'b000000000000000001\_0000000000\_11\_00; data\_cpu = 32'hx; end //read 5

17: begin read = 1; write = 0; addr\_cpu = 32'b000000000000000010\_0000000000\_00\_00; data\_cpu = 32'hx; end //read 6

18: begin read = 1; write = 0; addr\_cpu = 32'b000000000000000010\_0000000000\_11\_00; data\_cpu = 32'hx; end //read 7

endcase

end

always @(posedge clk) begin

if(~cache\_stall) begin

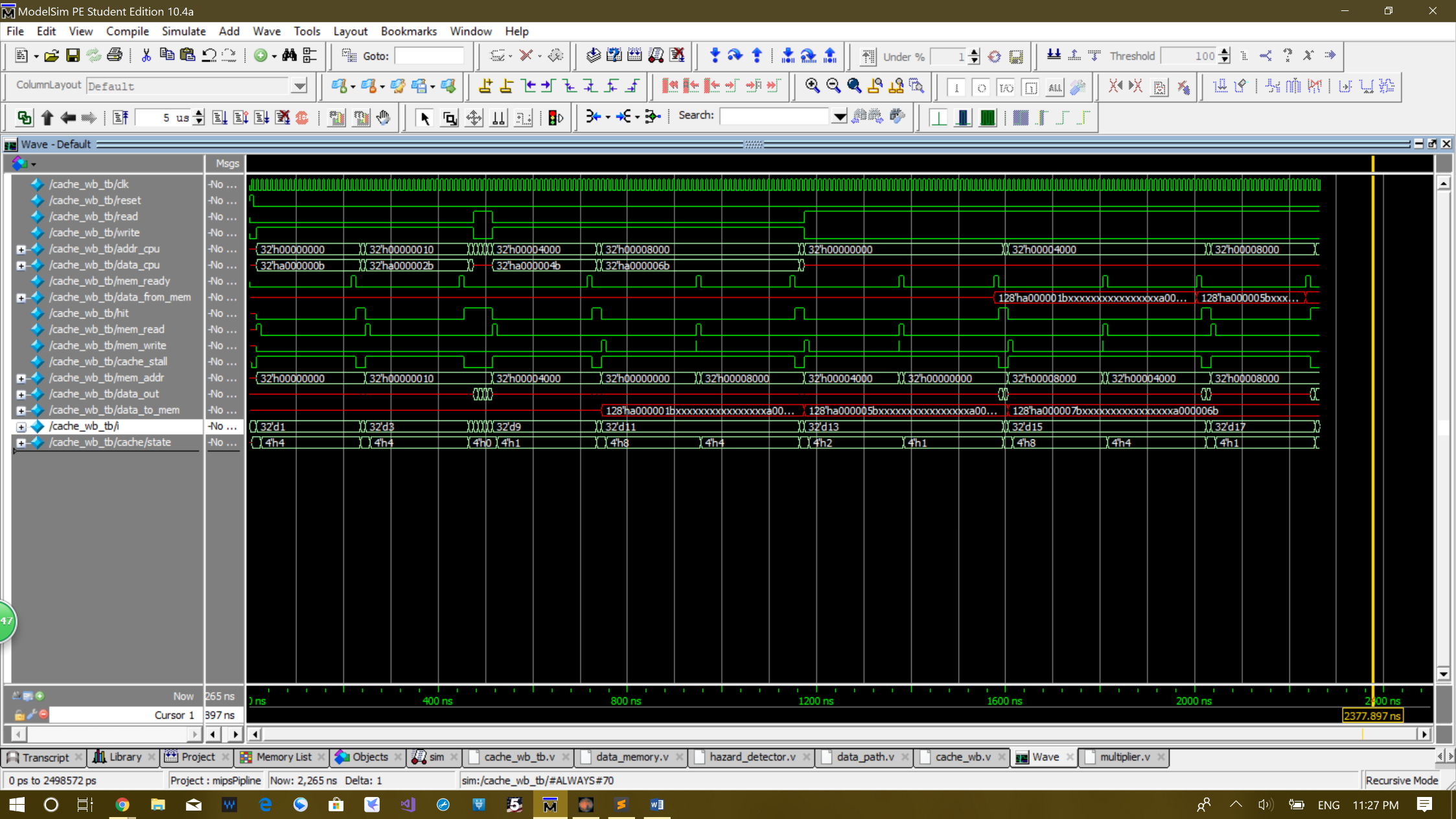
i = i + 1;

if(i == 19) $stop;

end

end

endmodule

**Waveform:**

The index i highlighted in the signals is the test case number.

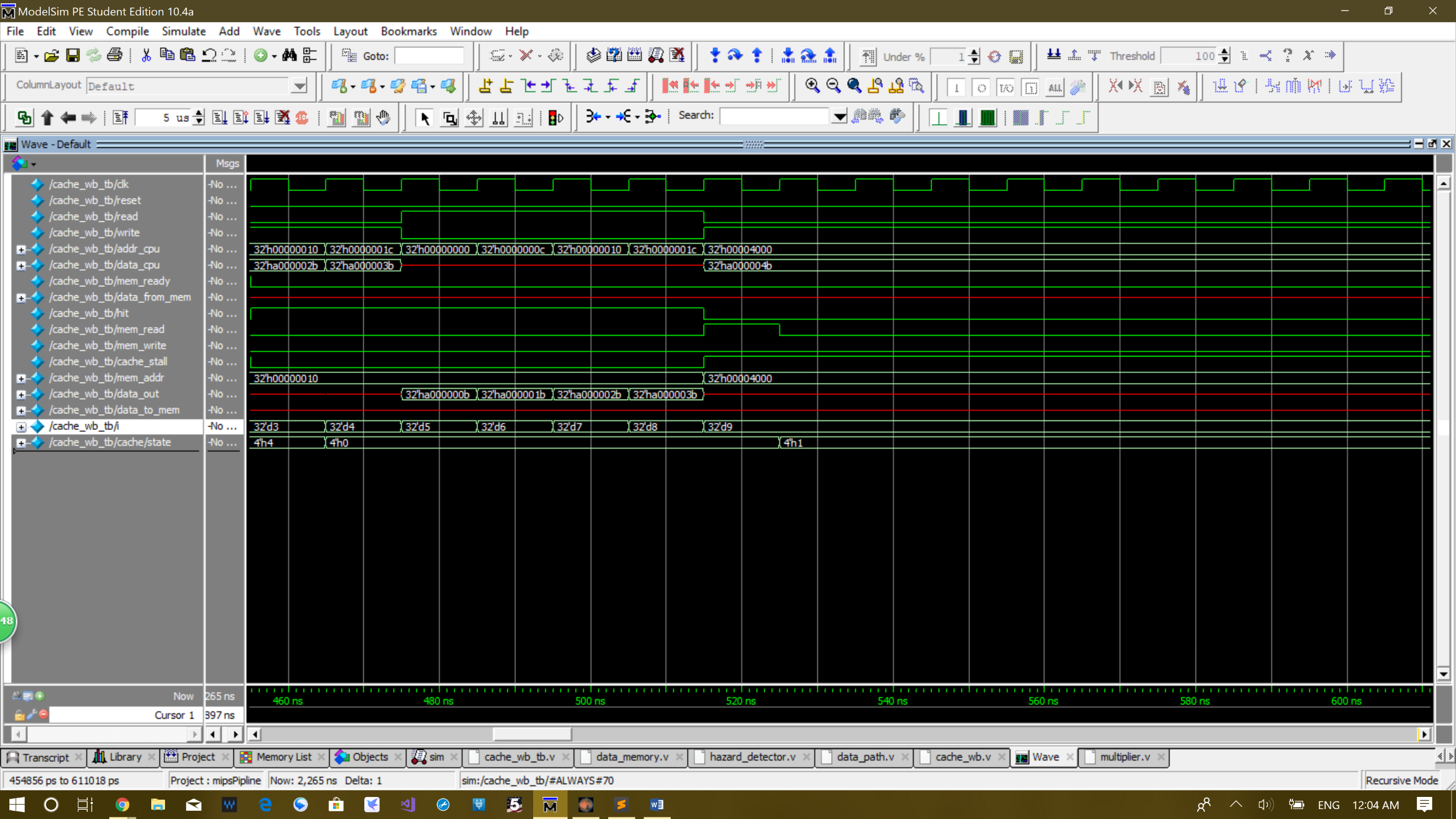
Tests:

1. store 0xa000\_000b to memory address 000000000000000000\_0000000000\_00\_00. It is in cache way0, set0, block0. The state is at first 0 for one cycle, follow by 20 cycles of state 4 which is bring the block from memory. Dirty bit is set to 1.

2. store 0xa000\_001 to memory address 000000000000000000\_0000000000\_11\_00. It is in cache way0, set0, block3. Since the block is already in the cache, it only takes 1 cycle to write it in. The state of this cycle is 0.

3. store 0xa000\_002b to memory address 000000000000000000\_0000000001\_00\_00. It is in cache way0, set1, block0. This writes to another block, so the state is 4 again for 20 cycles to bring the block to the cache. Dirty bit is set to 1.

4. store 0xa000\_003b in memory address 000000000000000000\_0000000001\_11\_00. It is in cache way0, set1, block3. Since the block is in the cache, it only takes 1 cycle to write it in. The state of the cycle is 0.

**The picture below shows the detail of test 5, 6, 7, 8**

5. read from memory address 000000000000000000\_0000000000\_00\_00. Read out 0xa000\_000b from cache way0, set0, block0. It only takes 1 cycle to read because it is already in the cache. The state is 0.

6. read from memory address 000000000000000000\_0000000000\_11\_00. Read out 0xa000\_001b from cache way0, set0, block3. It only takes 1 cycle to read because it is already in the cache. The state is 0.

7. read from memory address 000000000000000000\_0000000001\_00\_00. Read out 0xa000\_002b from cache way0, set1, block0. It only takes 1 cycle to read because it is already in the cache. The state is 0.

8. read from memory address 000000000000000000\_0000000001\_11\_00. Read out 0xa000\_003b from cache way0, set1, block3. It only takes 1 cycle to read because it is already in the cache. The state is 0.

9. store 0xa000\_004b to memory address 000000000000000001\_0000000000\_00\_00. It is in cache way1, set0, block0. It stays at state 1 for 20 cycle because memory needs to write block to way1. Dirty bit is set to 1.

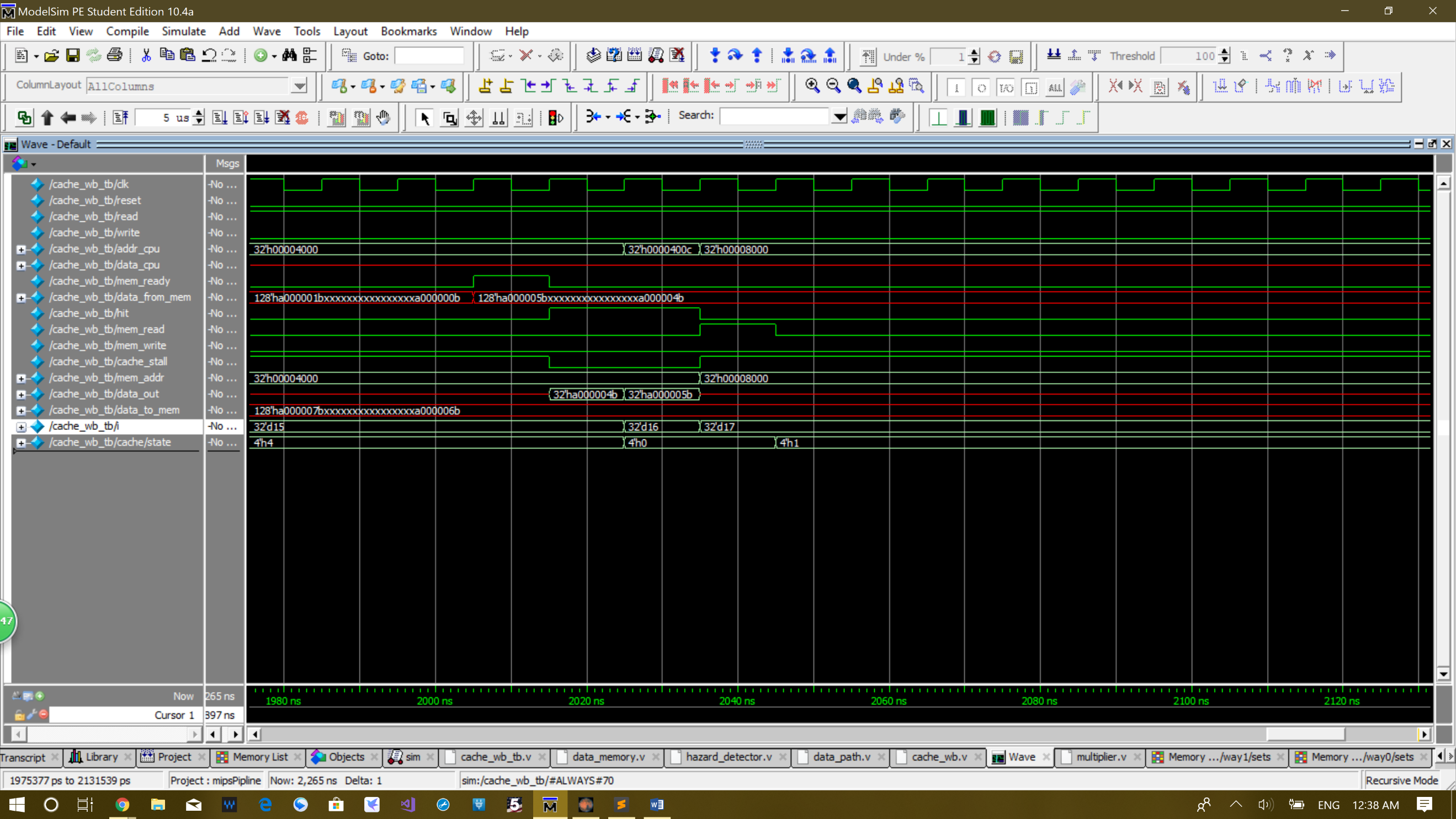
10. store 0xa000\_005b to memory address 000000000000000001\_0000000000\_11\_00. It is in cache way1, set0, block3. State 0 last for 1 cycle be cause in the last test case, the block is already brought in.

11. store 0xa000\_006b to memory address 000000000000000010\_0000000000\_00\_00. It is in cache way0, set0, block0. You can see on the waveform that it stays in state 8 for 20 cycles and then stays in state 4 for 20 cycles. State 8 is needed because dirty bit is 1 and we need to store the way0 set0 block back to memory first. State 4 is for bring the new block from memory. The dirty bit is set to 1.

12. store 0xa000\_007b to memory address 000000000000000010\_0000000000\_11\_00. It is in cache way0, set0, block3. It only takes 1 cycle to write because the block is already in the cache. The state is 0.

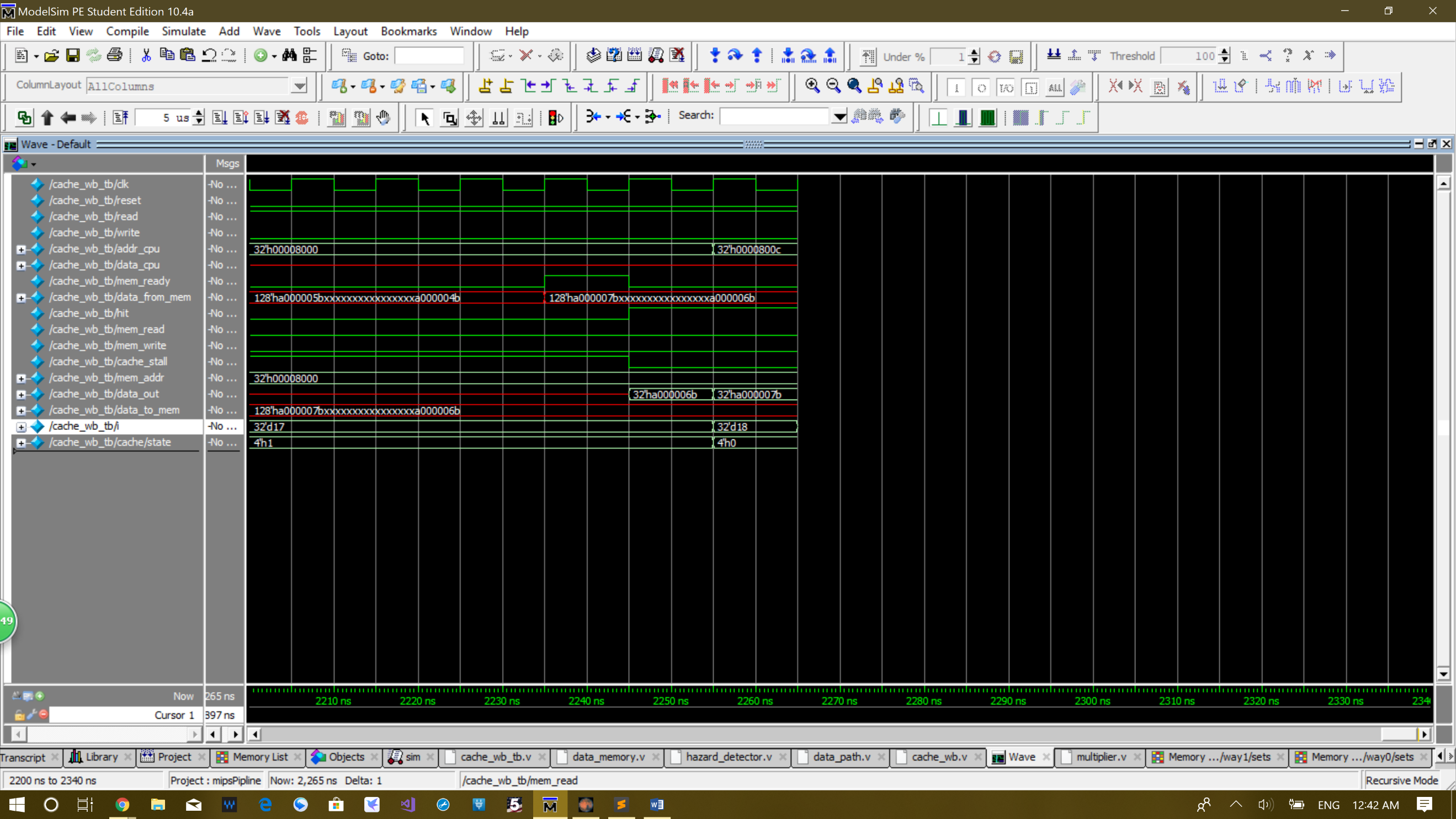
13. read from memory address 000000000000000000\_0000000000\_00\_00. Read out 0xa000\_000b. The state is 2 for 20 cycles and 1 for 20 cycles. State 2 is for storing back the way1 set0 block back to memory, because dirty bit is 1. State 1 is for bring back the block that stores 0xa000\_000b and 0xa000\_001b from the memory.

14. read from memory address 000000000000000000\_0000000000\_11\_00. Read out 0xa000\_001b. It takes 1 cycle to read for the reason that the block is already brought in. State is 0.

**The picture below shows the detail of the end of test 15 and test 16**

15. read from memory address 000000000000000001\_0000000000\_00\_00. Read out 0xa000\_004b. This test stays in state 8 for 20 cycles and state 4 for 20 cycles. State 8 is for storing back way0 set1 block to memory, because dirty bit is 1. State 4 is for bring back the block that stores 0xa000\_004b and 0xa00\_005b from the memory.

16. read from memory address 000000000000000001\_0000000000\_11\_00. Read out 0xa000\_005b. It is state 0 and only takes 1 cycle, because the block is already in the cache.

**The picture below shows the detail of the end of test 17 and test 18**

17. read from memory address 000000000000000010\_0000000000\_00\_00. Read out 0xa000\_006b. It takes 20 cycles of state 1 only, because the dirty bit is 0, so we can directly bring the block from memory without storing anything back to it.

18. read from memory address 000000000000000010\_0000000000\_11\_00. Read out 0xa000\_007b. Only takes 1 cycle of state 0 because the block is already in cache.

**Step3:** You must make some changes to your data memory in order to emulate a realistic memory.

**Code:**

`timescale 1ns / 1ps

module data\_memory(input clk, read, write,

input [31:0] addr,

input [127:0] data\_to\_mem,

output reg ready = 0,

output reg [127:0] data\_from\_mem);

reg [127:0] ram [0:4095];

wire [27:0] i; assign i = addr[31:4];

always @(posedge clk) begin

if(ready) ready <= 0;

if(read) begin

ready <= #190 1; //delay by 190ns, 10ns per cycle, data available at 20th cycle

data\_from\_mem <= #190 ram[i];

end

else if(write) begin

ready <= #190 1; //delay by 190ns, 10ns per cycle, data written at 20th cycle

ram[i] <= #190 data\_to\_mem;

end

end

endmodule

**Step4:** Modify your processor in order to avoid any data hazard.

The only modification we did is to add to all stage of pipeline register and PC a stall signal. The stall signal is called cache\_stall and is generated by the cache. When cache\_stall is pulled high, all stages of pipeline including the PC stalls at the same time until cache finish its operation.

**Step5:** Create a new source and integrate all of your modules into it.

**Code:**

// top level module

module data\_path(input clk, reset);

// fetch stage signals

wire [31:0] pcplus4F, pc\_next, pcF;

wire [31:0] instrF;

wire stallF;

// decode stage signals

wire [31:0] instrD;

wire equalD;

wire [4:0] rsD, rtD, rdD;

wire memreadD, memwriteD, regwriteD, memtoregD, regdstD, alusrcD, se\_zeD, branchD, start\_multD, mult\_signD;

wire [3:0] alu\_opD;

wire [1:0] out\_selD, pcsrcD;

wire [31:0] readdata1D, readdata2D, muxa\_outD, muxb\_outD, sh\_immD, se\_immD, ze\_immD, ext\_immD, sl2\_outD, pcplus4D, pcbranchD, pcjumpD;

wire stallD, forwardaD, forwardbD;

// execute stage signals

wire memreadE, memwriteE, regwriteE, memtoregE, regdstE, alusrcE, start\_multE, mult\_signE;

wire [1:0] out\_selE;

wire [3:0] alu\_opE;

wire [4:0] rsE, rtE, rdE;

wire [4:0] writeregE;

wire [31:0] srcaE, srcbE, aluoutE, readdata1E, readdata2E, writedataE, outE, ext\_immE, sh\_immE, loE, hiE;

wire flushE;

wire [1:0] forwardaE, forwardbE;

// memory stage signals

wire regwriteM, memtoregM, memreadM, memwriteM;

wire [4:0] writeregM;

wire [31:0] writedataM, readdataM, outM;

wire mem\_ready;

wire [127:0] data\_from\_mem;

wire hit, mem\_read, mem\_write, cache\_stall;

wire [31:0] mem\_addr;

wire [127:0] data\_to\_mem;

// write-back stage signals

wire regwriteW, memtoregW;

wire [4:0] writeregW;

wire [31:0] outW, readdataW, resultW;

// fetch stage logic

mux3#(32) pc\_mux(pcplus4F, pcbranchD, pcjumpD, pcsrcD, pc\_next);

register#(32) pc\_reg(clk, reset, stallF | cache\_stall, pc\_next, pcF);

inst\_memory imem(pcF, instrF);

assign pcplus4F = pcF + 4;

// decode stage logic

register#(64) pipeline\_regD(clk, reset | pcsrcD[0] | pcsrcD[1], stallD | cache\_stall , {instrF, pcplus4F}, {instrD, pcplus4D});

controller control(instrD[31:26], instrD[5:0], equalD, memreadD, memwriteD, regwriteD, memtoregD, regdstD, alusrcD, se\_zeD, branchD, start\_multD, mult\_signD, alu\_opD, out\_selD, pcsrcD);

reg\_file rf(clk, reset, regwriteW, instrD[25:21], instrD[20:16], writeregW, resultW, readdata1D, readdata2D);

assign rsD = instrD[25:21];

assign rtD = instrD[20:16];

assign rdD = instrD[15:11];

mux2#(32) forward\_muxaD(readdata1D, outM, forwardaD, muxa\_outD);

mux2#(32) forward\_muxbD(readdata2D, outM, forwardbD, muxb\_outD);

assign equalD = muxa\_outD == muxb\_outD;

sl16 lui\_sh(instrD[15:0], sh\_immD);

signext se(instrD[15:0], se\_immD);

zeroext ze(instrD[15:0], ze\_immD);

mux2#(32) ext\_muxD(ze\_immD, se\_immD, se\_zeD, ext\_immD);

sl2 jump\_sh(se\_immD, sl2\_outD);

assign pcbranchD = sl2\_outD + pcplus4D;

assign pcjumpD = {pcplus4D[31:28], instrD[25:0], 2'b0};

// execute stage logic

wire [156:0] dE, qE;

assign dE = {memreadD, memwriteD, regwriteD, memtoregD, regdstD, alusrcD, alu\_opD, out\_selD, start\_multD, mult\_signD, readdata1D, readdata2D, rsD, rtD, rdD, sh\_immD, ext\_immD};

assign {memreadE, memwriteE, regwriteE, memtoregE, regdstE, alusrcE, alu\_opE, out\_selE, start\_multE, mult\_signE, readdata1E, readdata2E, rsE, rtE, rdE, sh\_immE, ext\_immE} = qE;

register#(157) pipeline\_regE(clk, reset | flushE, cache\_stall, dE, qE);

mux2#(5) regdst\_mux(rtE, rdE, regdstE, writeregE);

mux3#(32) forward\_muxaE(readdata1E, resultW, outM, forwardaE, srcaE);

mux3#(32) forward\_muxbE(readdata2E, resultW, outM, forwardbE, writedataE);

mux2#(32) alusrc\_mux(writedataE, ext\_immE, alusrcE, srcbE);

ALU alu(srcaE, srcbE, alu\_opE, aluoutE);

wire busy\_multE;

multiplier mult(clk, start\_multE, mult\_signE, srcaE, srcbE, busy\_multE, {hiE, loE});

mux4#(32) out\_muxE(aluoutE, sh\_immE, loE, hiE, out\_selE, outE);

// memory stage logic

wire [72:0] dM, qM;

assign dM = {memreadE, memwriteE, regwriteE, memtoregE, outE, writedataE, writeregE};

assign {memreadM, memwriteM, regwriteM, memtoregM, outM, writedataM, writeregM} = qM;

register#(73) pipeline\_regM(clk, reset, cache\_stall, dM, qM);

//data\_memory dmem(clk, memwriteM, outM, writedataM, readdataM);

cache\_wb cache(clk, reset, mem\_ready,

memreadM, memwriteM,

outM, writedataM,

data\_from\_mem,

hit, mem\_read, mem\_write, cache\_stall,

mem\_addr,

readdataM,

data\_to\_mem);

data\_memory mem(clk, mem\_read, mem\_write,

mem\_addr,

data\_to\_mem,

mem\_ready,

data\_from\_mem);

// write-back stage logic

wire [70:0] dW, qW;

assign dW = {regwriteM, memtoregM, readdataM, outM, writeregM};

assign {regwriteW, memtoregW, readdataW, outW, writeregW} = qW;

register#(71) pipeline\_regW(clk, reset, cache\_stall, dW, qW);

mux2#(32) memtoreg\_muxW(outW, readdataW, memtoregW, resultW);

// hazard detector

hazard\_detector hd (branchD,

memtoregE, regwriteE,

memtoregM,

regwriteW,

start\_multE, busy\_multE, //let hazard unit know when mult is done

rsD, rtD, rsE, rtE,

writeregE, writeregM,

stallF, stallD,

flushE);

forwarding\_unit fu (regwriteM,

regwriteW,

rsD, rtD, rsE, rtE,

writeregM, writeregW,

forwardaD, forwardbD,

forwardaE, forwardbE);

endmodule

// basic building blocks

module register#(parameter width = 1)

(input clk, reset, en,

input [width - 1:0] d,

output reg [width - 1:0] q);

always @(posedge clk) begin

if(reset)

q <= 0;

else if(~en)

q <= d;

end

endmodule

module mux2#(parameter width = 1)

(input [width - 1:0] d0, d1,

input sel,

output [width - 1:0] y);

assign y = sel ? d1 : d0;

endmodule

module mux3#(parameter width = 1)

(input [width - 1:0] d0, d1, d2,

input [1:0] sel,

output [width - 1:0] y);

assign y = (sel == 2'd0) ? d0 :

(sel == 2'd1) ? d1 :

(sel == 2'd2) ? d2 : {width{1'bx}};

endmodule

module mux4#(parameter width = 1)

(input [width - 1:0] d0, d1, d2, d3,

input [1:0] sel,

output [width - 1:0] y);

assign y = (sel == 2'd0) ? d0 :

(sel == 2'd1) ? d1 :

(sel == 2'd2) ? d2 :

(sel == 2'd3) ? d3 : {width{1'bx}};

endmodule

module signext(input [15:0] a, output [31:0] y);

assign y = {{16{a[15]}}, a};

endmodule

module zeroext(input [15:0] a, output [31:0] y);

assign y = {16'b0, a};

endmodule

module sl2(input [31:0] a, output [31:0] y); // shift left by 2

assign y = {a[29:0], 2'b0};

endmodule

module sl16(input [15:0] a, output [31:0] y); // shift left by 16

assign y = {a, 16'b0};

endmodule

**Step6:** Use the provided test pattern and measure the performance of your design.

Compare the results of step 6 and 4. What is the miss rate in each case? Explain all your conclusions.

The miss rate of without cache is 100%. Because without cache, the memory simply needs 20 cycles to be access every time. The miss rate of our program with cache is 25%. Because we write our data consecutively into the memory and we only use addresses that are have small values, we do not encounter events that force the cache to write back some of its content back to memory, so the miss rate is just 1 for each block which is four words, so 25%.

Is the miss rate of a two-way set associative cache always, usually, occasionally, or never better than that of a direct mapped cache of the same capacity and block size?

The miss rate of a two-way set associative cache is usually better than a direct mapped cache of the same capacity and block size, because it can better deal with the Ping-Pong effect, whereas the direct mapped cache cannot deal with Ping-Pong effect at all.

**This is what our fancy program does:**

num = 78

sum = 0

create an array

for(i = 1; i < 6; i++) { loop 5 times

sum = sum + num

array[i - 1] = sum

}

product = num \* 5

array[5] = product

j = 5

for(i = 4; i > 0; i--) { loop 4 times

j++

array[j] = array[i] - num

}

multiply 0xFFFF\_FFFF and 0xFFFF\_FFFF

array[10] = reghi

array[11] = reglo

array[12] = 0xFFFF\_FFFF & 0x0000\_FFFF

diff = 0xFFFF\_FFFF - 0x0000\_FF00

array[13] = diff

e

x = (-100 < 100)

array[14] = x

x = (2'complement of -100 < 100)

array[15] = x

x = 0xFF00\_FF00

y = 0x0000\_FFFF

z = x & y

array[16] = z

z = x | y

array[17] = z

z = x xor y

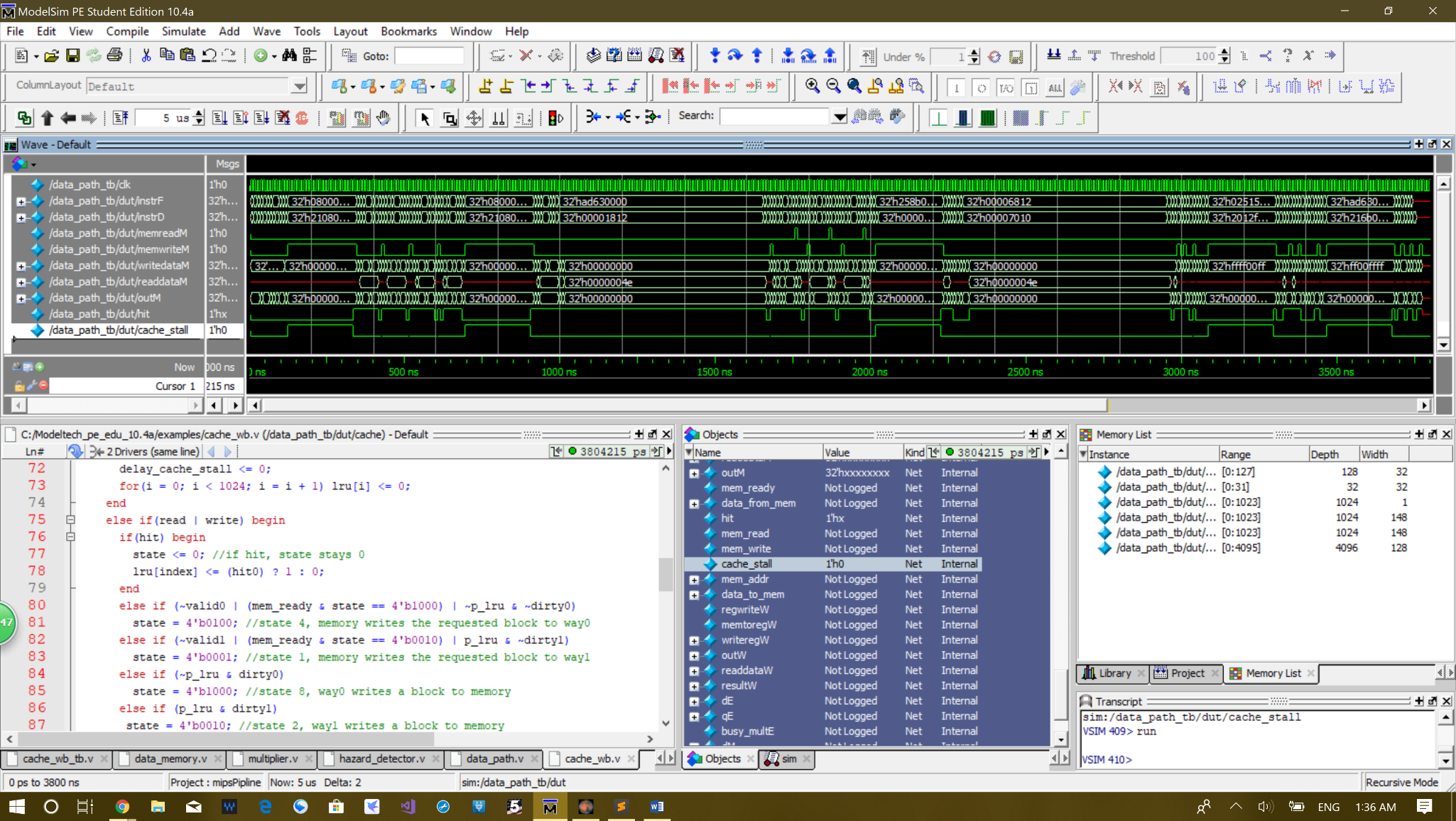
array[18] = z

z = x xor 0x0000\_FFFF

array[19] = z

z = x xnor y

array[20] = z

**Waveform:**

As you can see the cahe\_stall signal is pulled high for 20 cycles 5 times when running the program, since we have in total 20 memory accesses that stores the 20 calculation results of the program. It is only high for 5 times because there are 4 words in a block and our program only stores to memory in consecutive memory positions. In total 5 blocks of memory of 4 words are brought in. You can see on the waveform that there are in total 20 places the signal memwriteM is pulled high and every time it is a 20-cycle delay followed by three 1 cycle memory accesses. From left to right, start at around 50ns the first memory store happens and this 20-cycle high and the next three high are for storing the consecutive multiples of 78 in to the consecutive array positions in the memory. Then the first block is full and the next multiple of 78 has to be stored into another block so the cache\_stall is again 1 for 20 cycles. Then there is a 64-cycle stall created by the multiplication of 78\*5. The it is stored in the block in 1 cycle, because the block is not full yet. The we can see three 1-cycle memreadM signal becomes high. Each of them is followed immediately be a 1-cycle memwriteM signal high. These are generated by consecutively subtracted 78 from multiples of 78 stored in the cache before. Because they are already in the cache, they only take 1 cycle to read and write. Then it is followed by another 64-cycle stall generated by multiply 0xFFFF\_FFFF and 0xFFFF\_FFFF. The next two memwriteM signal is consecutive because mflo and mfhi are executed consecutively. Next, we run out of blocks again so the result of the next 4 calculations can be stored. They are 0xFFFF\_FFFF - 0x0000\_FF00, -100 < 100 signed, -100 < 100 unsigned, 0xFF00\_FF00 & 0x0000\_FFFF. Then we run out of blocks again and we have to stall 20 cycle to store the last four results from 0xFF00\_FF00 | 0x0000\_FFFF, 0xFF00\_FF00 ^ 0x0000\_FFFF, the same thing xored but int immediate form and 0xFF00\_FF00 xnor 0x0000\_FFFF.

**Conclusion:**

In this project we designed a 32KB 2-way associative cache. We tested many combinations and conditions of lw and sw on this cache, and we are rather confident that it works perfectly as a LRU write back policy cache.

Hours doing the project: about 36 hours.