Name: Zhanglu Wang
Ke Ding
Changsheng Su

Introduction:

In this lab we want to scale the size of the memory and implement a 32 kBytes along with the memory.

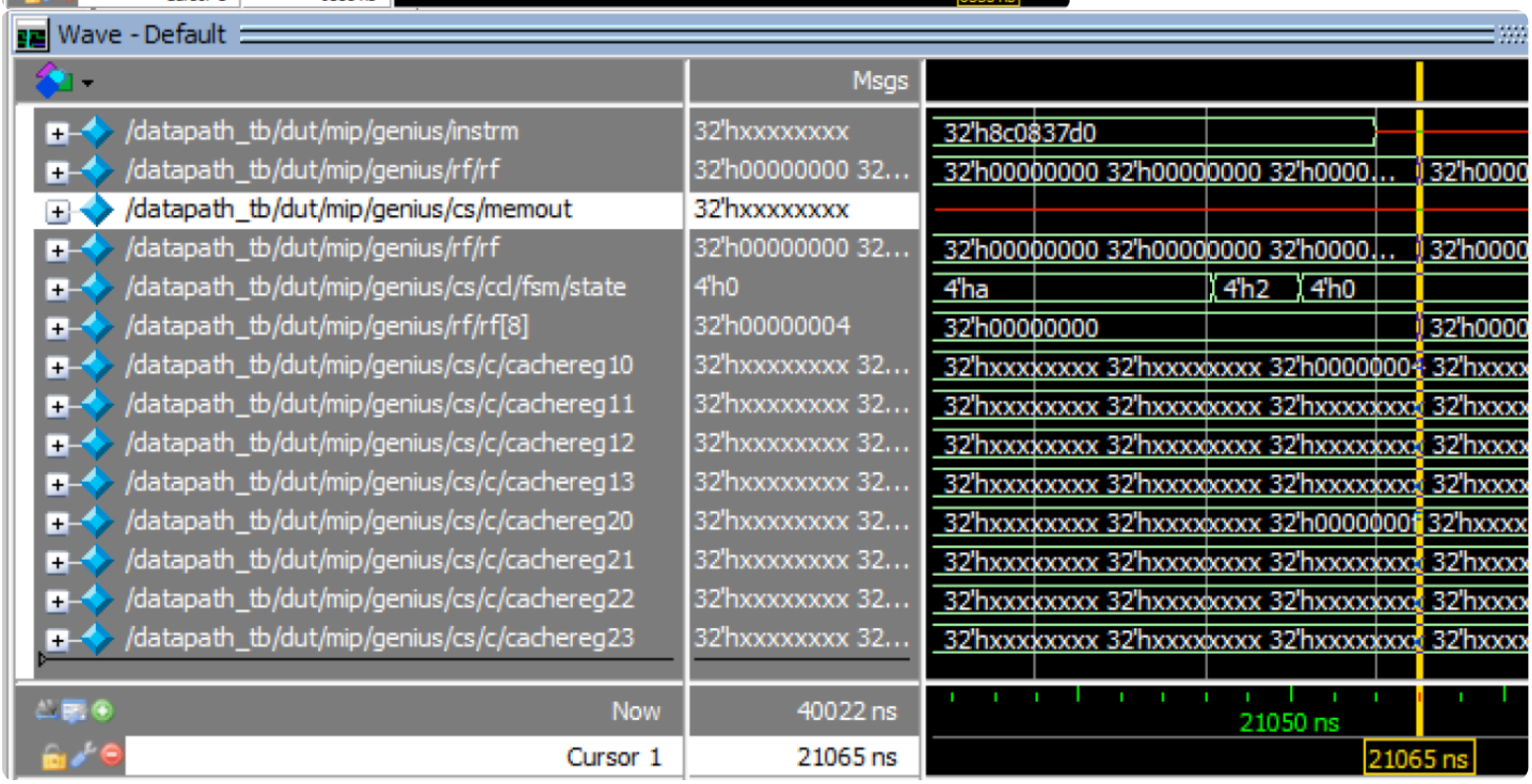
Illustration of instruction:

Fancy Program:

addi t2 zero 0x3	200A0003	t2 = 3
addi t1 zero 0x4	20090004	t1 = 4
sw t1 0x4 zero	AC090004	mem[0x4] = 4
sw t2 0x0 zero	AC0A0000	mem[0x0] = 3
sw t2 0x8 zero	AC0A0008	mem[0x8] = 3
add t3 t1 t2	012A5820	t3 = 7
sw t3 0x4 zero	AC0B0004	mem[0x4] = 7
lw t4 0x8 zero	8C0C0008	t4 = 3
add t3 t4 t1	01895820	t3 = 7
sw t3 0x800 zero	AC0B0800	mem[0x800] = 7
SW t3 0x1000 $zero	AC0B1000	mem[0x1000] = 7
lw $t3 0x800($0)	8C0B0800	t3 = 7

SW t3 0x1800 $zero.	AC0B1800	mem[0x1800] = 7
add t4 t3 t1	01696020	t4 = 11
sw t4 0x200 zero	AC0C0200	mem[0x200] = 11
sw t3 0x100 zero	AC0B0100	mem[0x100] = 7
sw t2 0x400 zero	AC0A0400	mem[0x400] = 3
sw t1 0x510 zero	AC090510	mem[0x510] = 4
lw t5 0x510 zero	8D4D0510	t5 = 4
Sw t4 0x408 zero.	AC0C0408	mem[0x408] = 11
Sw t3 0x410 zero.	AC0B0410	mem[0x410] = 7
Sw t2 0x414 zero.	AC0A0414	mem[0x414] = 3
Sw t1 0x424 zero.	AC090424	mem[0x424] = 4
Lw t6 0x424 zero.	8D4E0424	t6 = 4
Addi t6 zero 0x4.	200E0004	t6 = 4
Sw t4 0x340 zero.	AC0C0340	mem[0x340] = 11
Sw t5 0x320 zero.	AC0D0320	mem[0x320] = 4
Lw t6 0x340 zero.	8C0E0340	t6 = 11
Sw t4 0x380 zero.	AC0C0380	mem[0x380] = 11
Sw t6 0xb80 zero.	AC0E0B80	mem[0xB80] = 11
Lw t6 0xb80 zero.	8c0e0b80	t6 = 11
Add t5 t5 t6.	01AE6820	t5 = 15
Sw t3 0x7D0 zero.	AC0B07D0	mem[0x7D0] = 7
Sw t4 0x784 zero	AC0C0784	mem[0x784] = 11
Sw t1 0x37D0 zero.	AC0937D0	mem[0x37D0] = 4
lw t4 0x7D0 zero.	8c0c07d0	t4 = 7
Sw t5 0x87D0 zero	AC0D87D0	mem[0x87D0] = 15
Lw t6 0x87d0 zero.	8C0E87D0	t6 = 15
Lw t0 0x37D0 zero.	8C0837D0	t0 = 4

Here is the comparison of total reining time for memory with and without cache:

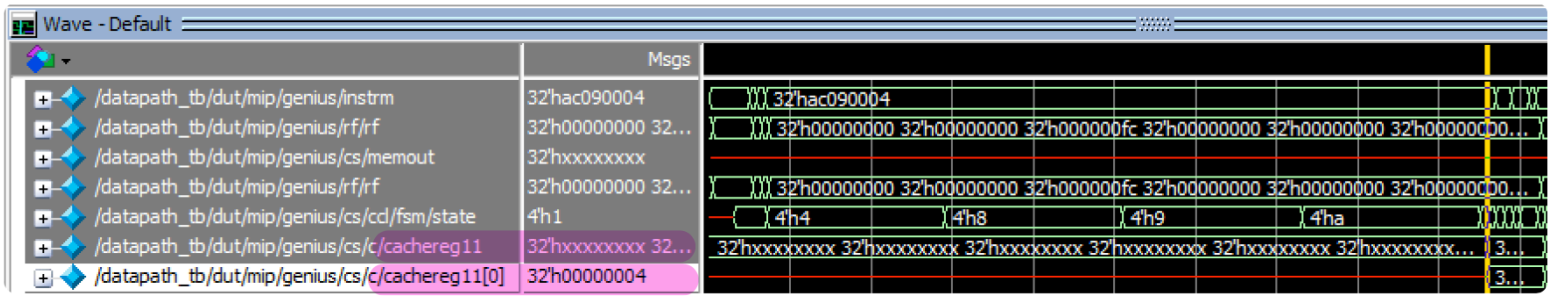


Compare the performance of memory with and without the cache.
For memory without cache, every time it accesses the memory it will take 20 cycles to load or read from memory.
For our fancy program, there are 32 accesses to the main memory, so It takes 634 cycles to complete without cache.(see from the first waveform) And for the memory with cache it takes 2107 cycles (second waveform) when we use 20 cycles for each word. We can optimize this by taking 20 cycles when we first access the memory and nearby words should only need 1 cycle to complete. Which is much more efficiency.

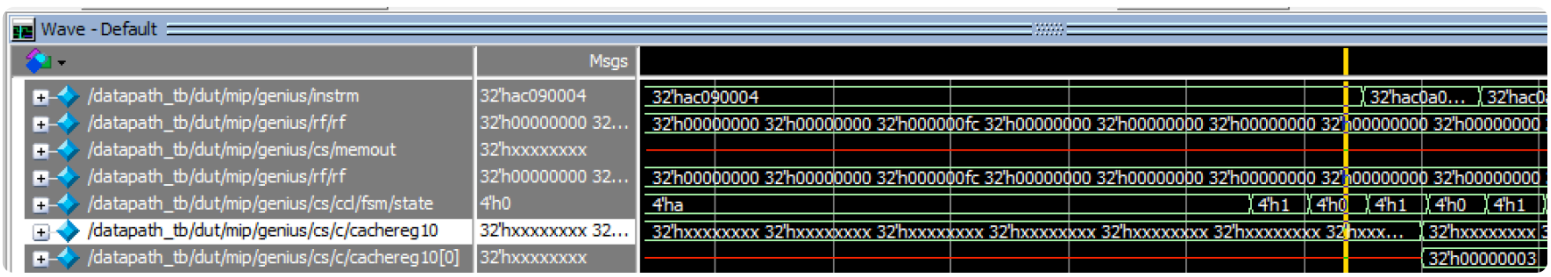
Miss Rate:
Total number of Miss: 19
Total access of cache system: 32
Miss rate = 19/32 = 59.375% (specifically for our fancy program)

Different situations:

1)
First: Miss and write.
mips code: sw t1 0x4 zero AC090004 mem[0x4] = 4
We can see the state is going from state 0 -> 4 -> 8 -> 9 -> A -> 1. (State were explained above) We can see we successfully stored the 4 into cachereg11[0] which is shown in the last row of the wave form at state 1.



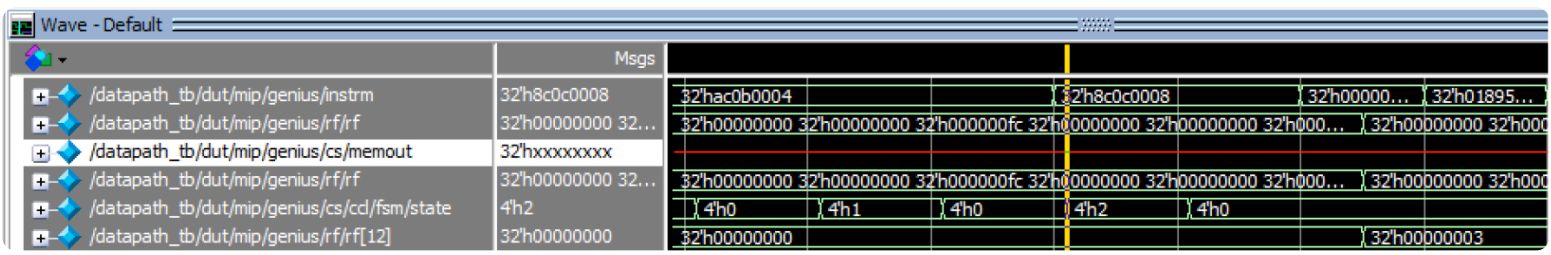
2)
Second: Hit and write.
Mips code: sw t2 0x0 zero AC0A0000 mem[0x0] = 3
We can see for this instruction, it is hit , so the state only went through the 0->1->0. And the 3 is successfully stored into the cachereg10[0] is shown in the last row of the wave form at state 1.



3)
Third: hit and read.

sw t1 0x4 zero	AC090004	mem[0x4] = 4
sw t2 0x0 zero	AC0A0000	mem[0x0] = 3
sw t2 0x8 zero	AC0A0008	mem[0x8] = 3
add t3 t1 t2	012A5820	t3 = 7
sw t3 0x4 zero	AC0B0004	mem[0x4] = 7
lw t4 0x8 zero	8C0C0008	t4 = 3

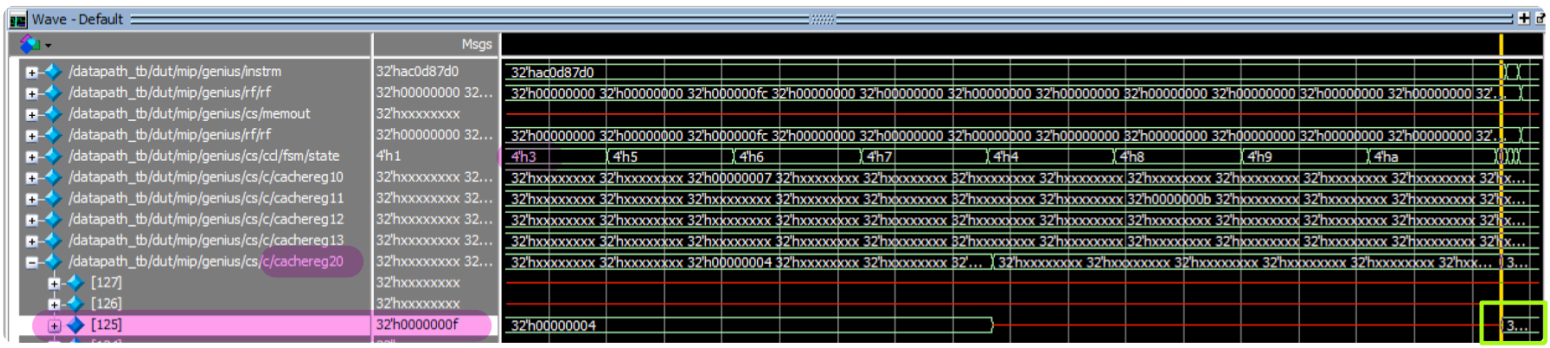
This is part of our fancy program, for the instruction" lw t4 0x8 zero 8C0C0008
t4 = 3" hit and read is happened, which we already have access to 0x8 in previous step, so in the wave form below we can see it go through state 0->2->0. And rf[12] is changed to 3 on the next cycle.



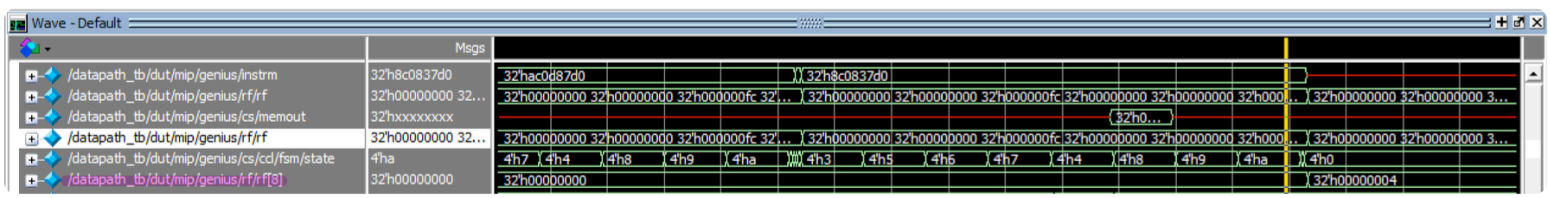
4)
Fourth: Miss write_back, write

Sw t3 0x7D0 zero.	AC0B07D0	mem[0x7D0] = 7
Sw t4 0x784 zero	AC0C0784	mem[0x784] = 11
Sw t1 0x37D0 zero.	AC0937D0	mem[0x37D0] = 4
lw t4 0x7D0 zero.	8c0c07d0	t4 = 7
Sw t5 0x87D0 zero	AC0D87D0	mem[0x87D0] = 15
Lw t6 0x87d0 zero.	8C0E87D0	t6 = 15
Lw t0 0x37D0 zero.	8C0837D0	t0 = 4

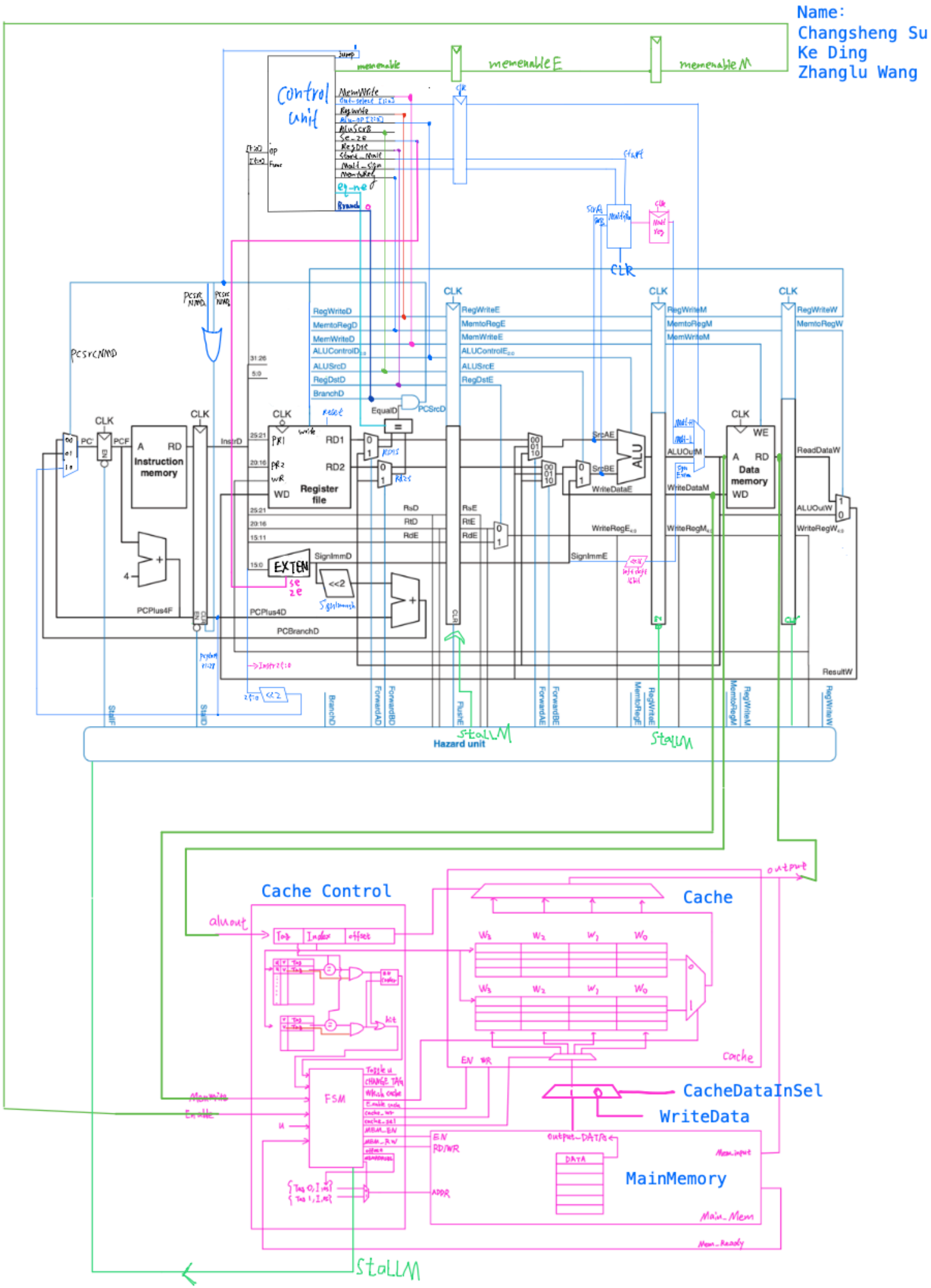
This is part of our fancy program, and we firstly write into 0x7D0, and secondly we write into 0x37D0, and thirdly we write into 0x87D0, because for these three address, they have the same index, So for the instruction "Sw t5 0x87D0 zero", will trigger the write_back state, which is state 3. And we successfully write the 15 (0xF) into the cachereg20[125].



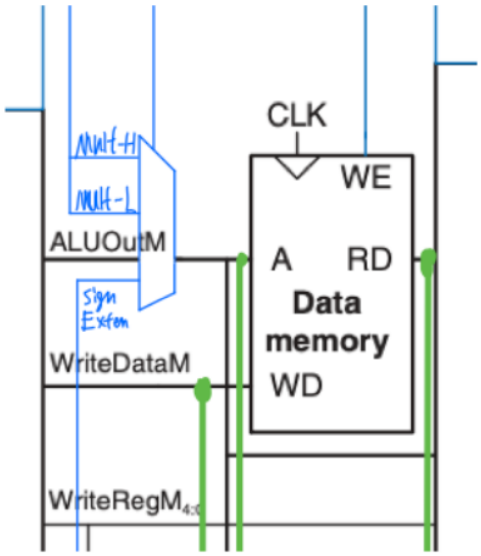
5)
Fifth: write_back, Read
Mips code:Lw t0 0x37D0 zero.
This is last part of our fancy program, is the last instruction of our program, which triggered the write_back state, we can see from the wave form below.
It loaded the 4 into rf[8] which is t0 register.



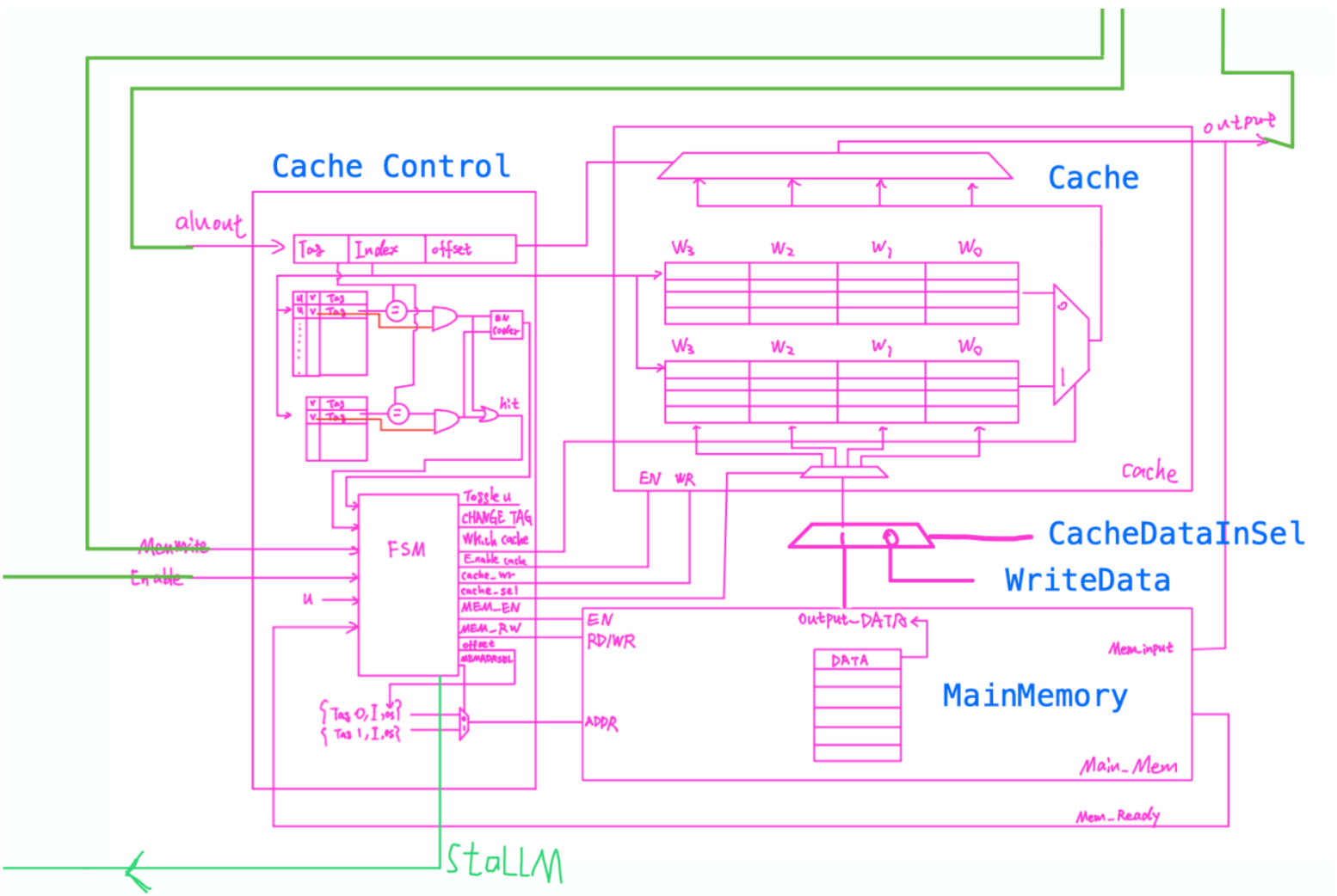
Methodology:



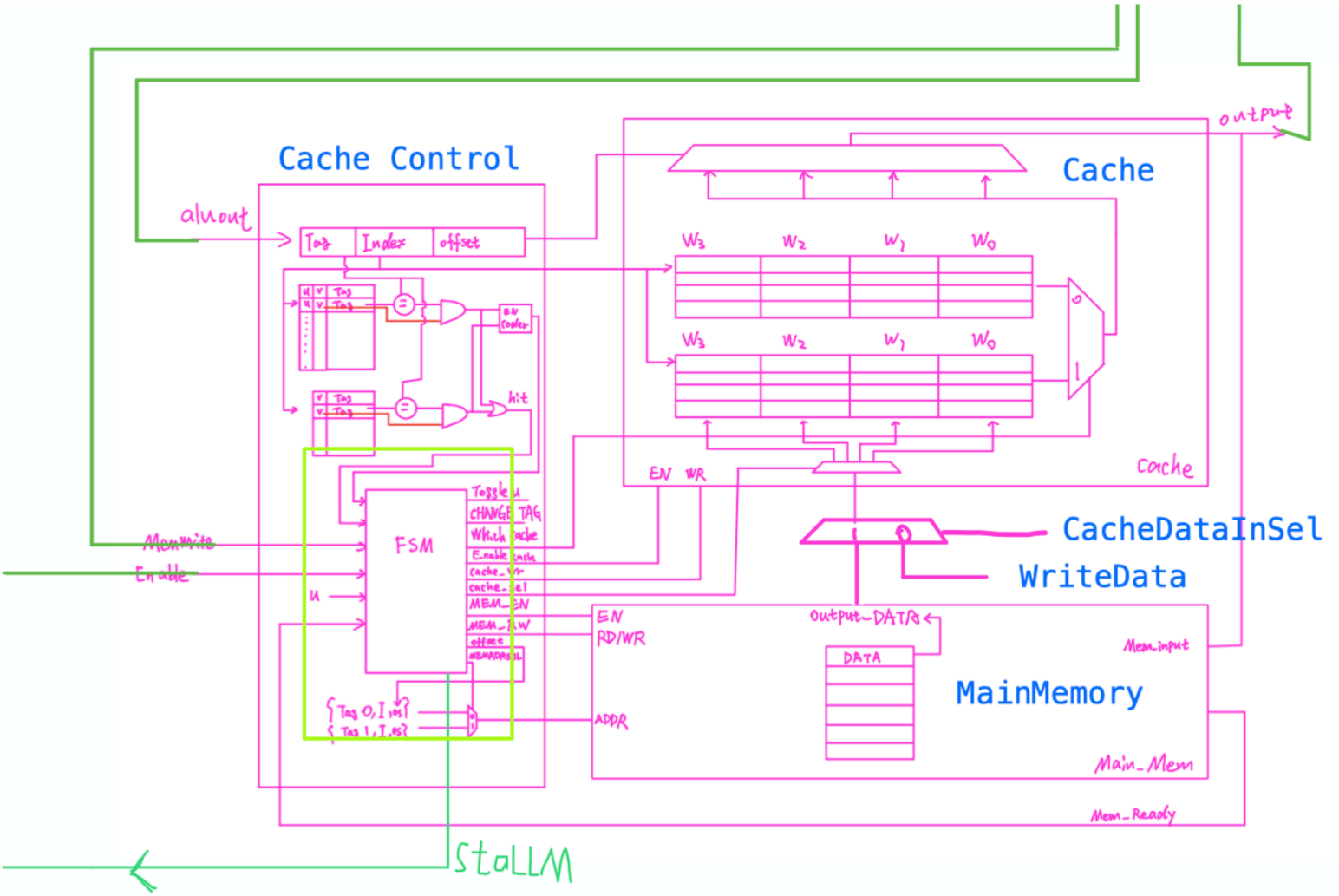
For the memory module, before our memory is without cache, and the size is 32 X 4 Bytes = 128 Bytes. And for LW and SW instructions, they will take 1 cycle when there is no hazard. If it has hazard, the LW instruction will take 2 cycles to execute.



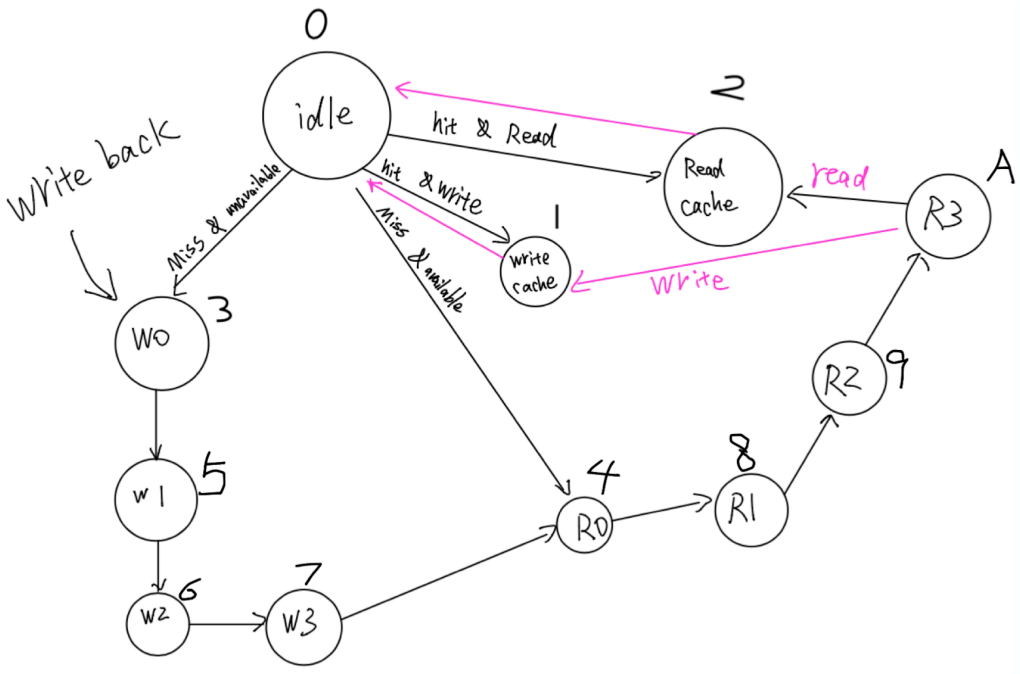
Then we modified our memory module added with a cache system as shown below:



Now we have a cache system. This cache system includes 'cache control', 'cache', 'memory' three parts, inside the cache control we are using a FSM to control the system. And the following diagram is our state machine:



For the FSM below, toggle_u signal will toggle the u bit, change_tag signal will change the tag to the input tag. Which_cache signal tells which set of cache we are currently using. Enable cache signal enable the cache. Cache_wr signal tells if we write the cache or read from the cache. Mem_en signal enables the memory. Mem_Rw signal tells if we read/write the memory. Offset signal tells us which columns of the specific set. Memadrsel controls the which set is put to the MainMemory.



For the STATE MACHINE:

Sate 0 is the idle state.

The FSM will start if enable signal for cache system is triggered.
If it detects hit signal, and write signal (write = 1) in the state 0 (idle state), we go to state 1 and write in the cache. If it detects hit signal, and read signal (read = 0) in the state 0 (idle state), we go to state 2 and read from the cache.

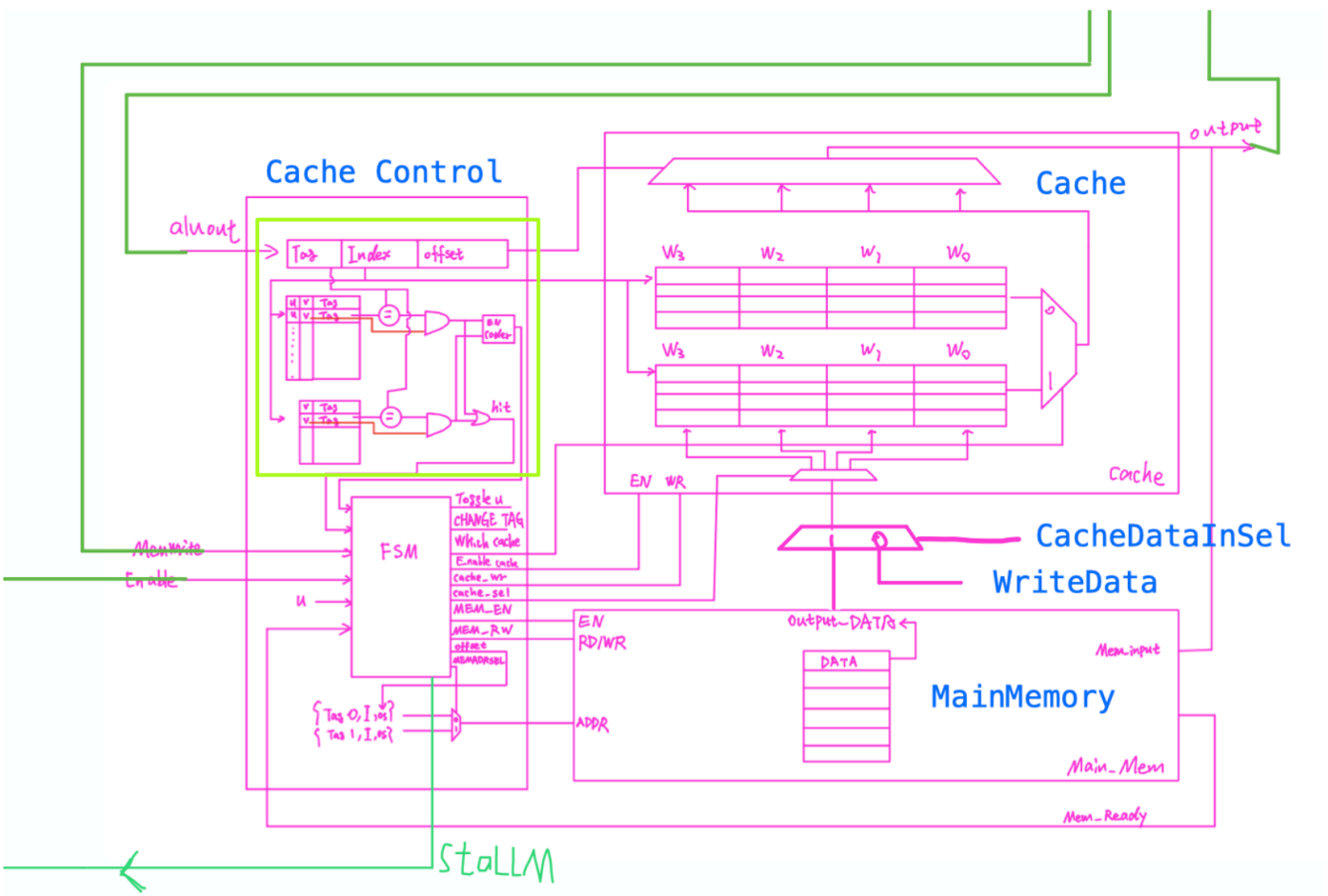
If we see a miss (hit = 0), and unavailable signal (both ways are occupied) in the state 0 (idle state), we go to state 3 (w0),

which is where it starts to write-back to memory. It is a serial write-back. We write one word (total of 4 words for one block) one state at a time, and each state that access to the memory will take 20 cycles. There will be a counter inside memory to do the 20 cycles delay. From state 3, 5, 6, 7 (w0 to w3) it will take total of 80 cycles to write one block back to memory. After state 7 (w3), it goes to state 4 (r0), state 4 will be explained in the next paragraph.

If we see a miss (hit =0), and available signal (there is empty space in two way sets) in the state 0 (idle state), we go to state 4 (r0), in this state, cache load one word from memory. From state 4, 8 ,9, A (R0 to R3), it preloads the data from memory to cache, it loads one word at a time and a total of 4 words (one block). Each states need to acquire from memory, and it takes 20 cycles for each states. Thus it takes total of 80 cycles from state 4, 8, 9, A (R0 to R3).

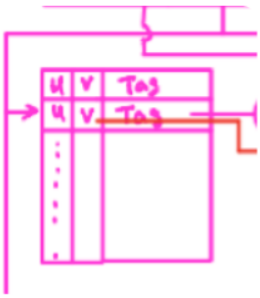
In the state A (R3), if it detects write signal (write = 1), it goes to state 1 and writes to the cache. If it detects read signal (write =0), it goes to state 2 and read from the cache.

In both state 1 and state 2 it will go to the state 0 (idle state) and wait for the next instruction.



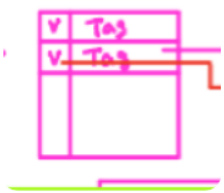
Now we will explain the cache control (in the yellow square part), It first receives address from aluout signal. Tag has 21 bits (aluout[31:11]), index has 7 bits (aluout[10:4]), and offset is the word offset which has 2 bits (aluout[3:2]). This offset will be directed to the cache module.
There are two cache control reg tables in this module.

One table is like



This table includes u (1bit), v(1 bit), tag(21 bits), and total of 128 rows.

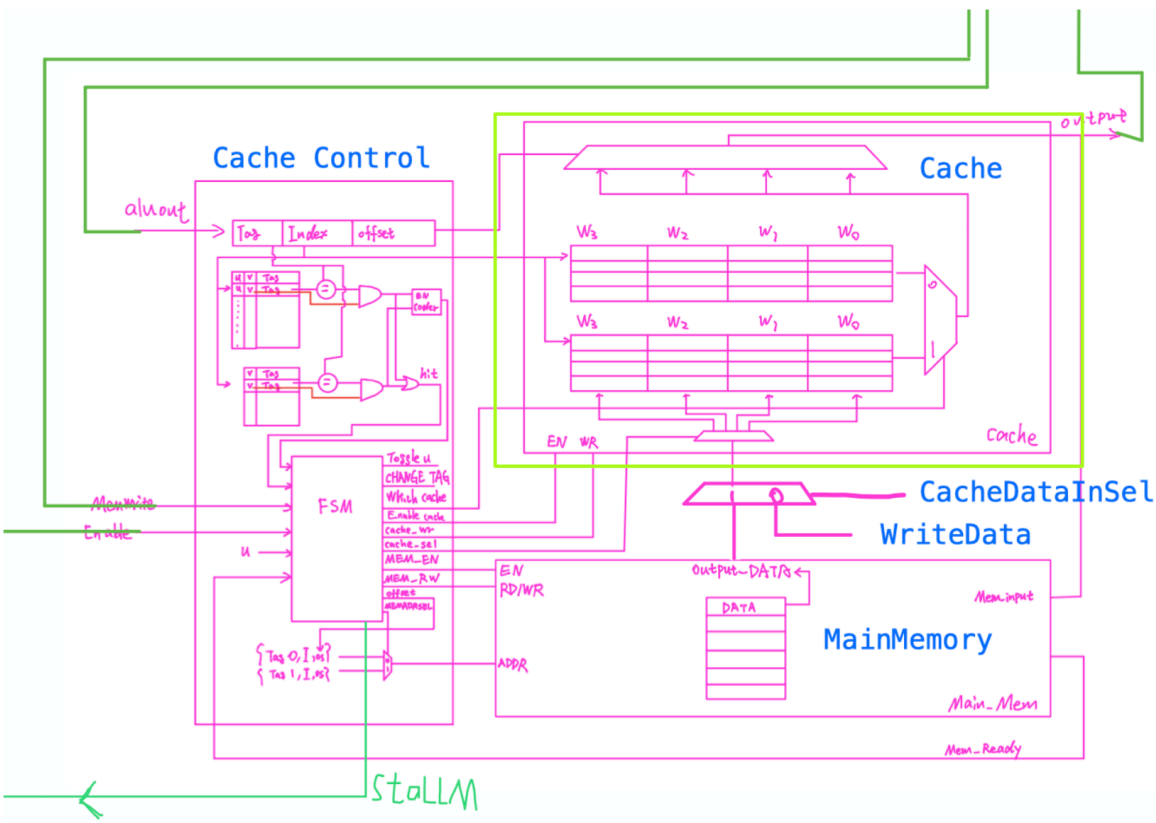
The other one is like



This table includes v(1 bit), tag(21 bits), and total of 128 rows.

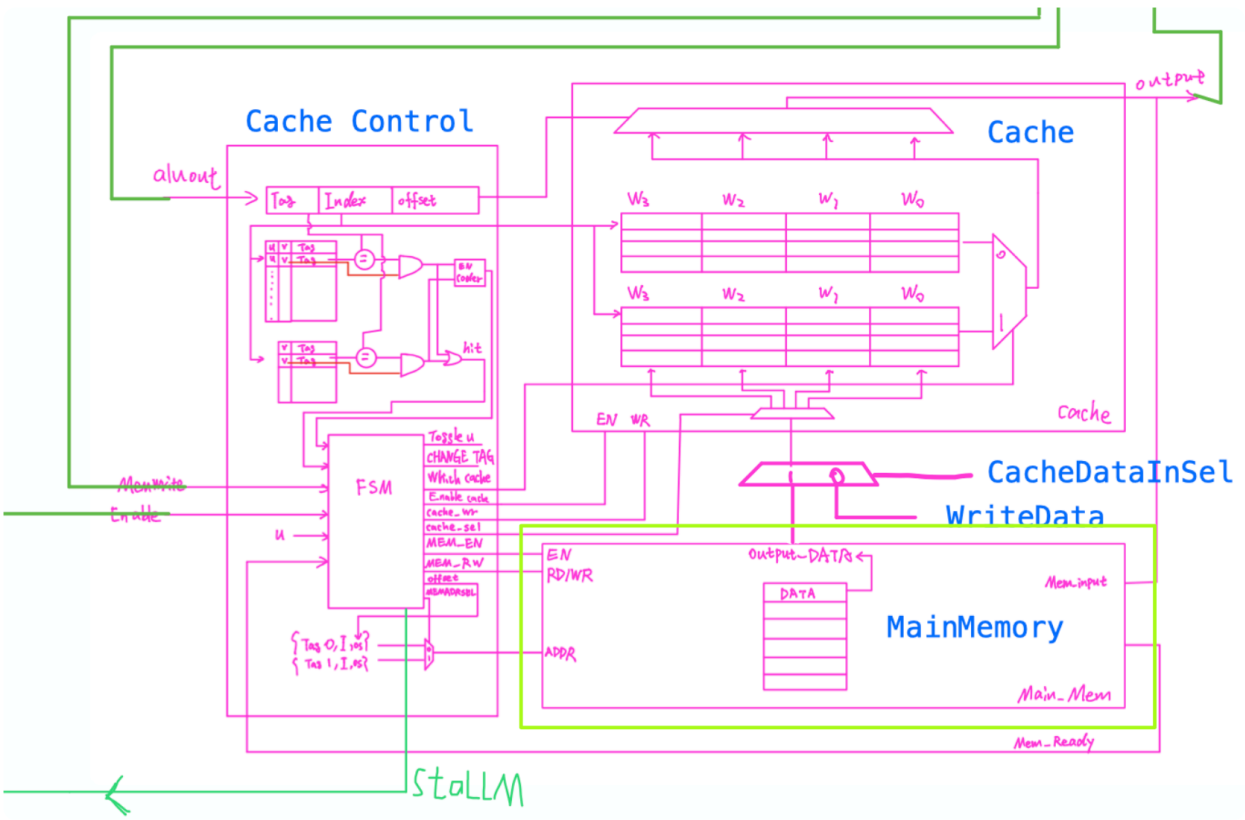
Given specific index, we can access to the tag u and v of both tables.

We first compare if tag in the table equals to the tag received from aluout. If they matched we then see v signal. v signal tells if this table is occupied or not. If v = 1 we can tell it is and that we will receive a hit signal and this hit signal will be wired to the FSM. There is an encoder as we can see diagram to check which cache we care about. If the input of the encoder from first table is 1 then encoder output is 0, which means we use cache 1, if the input of the encoder from second table.



Inside the cache, the data is coming from two input mux. Selection bit is coming cachedatainsel, which is the output from the FSM, which will decide if we should load word from memory or load word directly from the writedata.

There is a signal called cache-sel which is from FSM, determines which column that the data will be stored. This signal is determined in r0 to r3 state. Then there is a signal which-cache, which is from FSM, in which we determine which cache set we want to output. After we select which cache we want to output, we need to specify the word in block, which is coming from the offset in cache control. The cache is 32 Kbytes.



Finally, we will explain the main memory module. In this module, there is a counter that counts for 20 cycles every time it is accessed (or enable). RD/WD signal we decide if we want to write in the memory or read from memory. When it is in the counting state, it means everything should stall, when it finishs counting it will send men-ready signal to FSM to tell that there is no need to stall.

Step 6 Question:

The two way associative caching is usually lower than the direct mapped caching of the same capacity and block size, it is because the 2 way associative caching has lower conflicts. As the cache reads from both ways and a hit will happen if the data is present in any one of the block.

Mainly changed CODE:
We made changes in data_path.v to handle the hazard.

Memory.v

```
module posedgedetector(input data, input clk, output out);
    reg delayeddata;
    reg data_sync;
    //wire data_sync2;
    //reg [2:0] tog;
    //assign {delayeddata, data_sync, data_sync2} = tog;
    always@(posedge clk) begin
```

```
            delayeddata <= data;
            data_sync <= delayeddata;
            //if(data) tog <= 3'b111;
            //else tog <= 0;
        end
        assign out = data & ~data_sync;
endmodule
module changedetector(input [3:0] state, input clk, output out);
        reg [3:0] delayeddata;
        reg [3:0] data_sync;
        always@(posedge clk) begin
            delayeddata <= state;
            data_sync <= delayeddata;
        end
        assign out = (data_sync != state);


endmodule
module memory(input clk, input reset,input enable, input rdwr,
            input [31:0] address,
            input [31:0] mem_input,
            output reg [31:0] outputdata,
            //change add reg
            output mem_ready);

        reg internalready;
        reg [4:0] counter;

        //our memory ram
        reg [31:0] ram [1048576:0];



        //new change
        reg tmpenable;
        //assign mem_ready = (~tmpenable) & internalready;
        wire enableedge;
        posedgedetector detectenable(enable,clk,enableedge);
        assign mem_ready = (~enableedge) & internalready;


        //in this counter it purposelly stall for 20 cycles when accessing the memory.
        always @(posedge clk) begin
            if(reset) begin
                internalready <=1;
                //mem_ready <= (~enable) & internalready;
            end
            else if(counter > 2) begin
                counter <= counter -1;
            end else if(counter ==2) begin
                if(rdwr) ram[address[31:2]] <= mem_input;
                outputdata <= ram[address[31:2]];
                counter <= 1;
                internalready <= 1;
                tmpenable <= 0;
                //mem_ready <= 1;


            end else if(counter==1)begin
                counter <=0;
            end
            else if(enable) begin
                counter <= 21; //changed from 21
                internalready <= 0;
                tmpenable <= enable;
//              mem_ready <= (~enable) & internalready;
            end
        end
endmodule
```

Cache2.v

```
module cachesystem(input clk, reset, input [31:0] aluoutm, writedatam, input memwritem, input enable,
                   output [31:0] mout, output stallm);
    //  This is the top level of the cache system, with the combination of memory, cache and cache contrl.
    wire mem_ready;
    wire [1:0] offset1;
    wire [6:0] index;
    wire which_cache, enable_cache, cache_wr;
    wire [1:0] cache_sel;
    wire mem_en, mem_rw;
    wire [31:0] address;
    wire cachedatainselect;
    wire [31:0] memout;
    wire [31:0] cacheout;

    cache_control ccl (aluoutm, enable, clk, reset, mem_ready, memwritem, offset1, index,
    which_cache, enable_cache, cache_wr,
    cache_sel, mem_en, mem_rw,address, cachedatainselect, stallm,ruozhiout);
    // cache contrl set up

    wire [31:0] muxout;
    assign muxout = cachedatainselect ? memout : writedatam;

    cache c (clk, offset1, index, which_cache, cache_sel, enable_cache, cache_wr, muxout, cacheout); //cache
set up

    wire mem_en2;
    assign mem_en2 = ruozhiout & mem_en;


    memory mem (clk, reset, mem_en2, mem_rw, address, cacheout, memout, mem_ready); //memory set up

    assign mout = cacheout;


endmodule




module cache (input clk, input [1:0] offset, input [6:0] index, input whichcache,
            input [1:0]cachesel, input enable, input wr,
            input [31:0] writedata, output [31:0] outputdata);

    //change start

    //reg [127:0] cachereg1 [127:0];
    //reg [127:0] cachereg2 [127:0];
    //change end

    //each words in one block take its own reg, there aret total of two sets, so we have 8 reg.
    reg [31:0] cachereg10 [127:0];
    reg [31:0] cachereg11 [127:0];
```

```
        reg [31:0] cachereg12 [127:0];
        reg [31:0] cachereg13 [127:0];

        reg [31:0] cachereg20 [127:0];
        reg [31:0] cachereg21 [127:0];
        reg [31:0] cachereg22 [127:0];
        reg [31:0] cachereg23 [127:0];




        // |data0|data1|data2|data3|
        // data0 = cachereg#[index][offset]
        //wire [127:0] offsetstart, offsetend;
        //assign offsetstart = offset << 5+31;
        //assign offsetend = offset << 5;
        //assign outputdata = whichcache ? cachereg1[index][offsetstart:offsetend] : cachereg2[index]
[offsetstart:offsetend];


        //load the words from the cache to the output of the cache, whichcache signal will determine the correct
output.
        wire [31:0] temprawoutput0;
        wire [31:0] temprawoutput1;
        wire [31:0] temprawoutput2;
        wire [31:0] temprawoutput3;

        assign temprawoutput0 = whichcache ? cachereg20[index] : cachereg10[index];
        assign temprawoutput1 = whichcache ? cachereg21[index] : cachereg11[index];
        assign temprawoutput2 = whichcache ? cachereg22[index] : cachereg12[index];
        assign temprawoutput3 = whichcache ? cachereg23[index] : cachereg13[index];

        //assign {temprawoutput0,temprawoutput1,temprawoutput2,temprawoutput3} = whichcache ?
{cachereg10[index], cachereg11[index], cachereg12[index], cachereg13[index]} :
                        //{cachereg20[index], cachereg21[index], cachereg22[index], cachereg23[index]};

        //mux4 #(32) ruozhi(temprawoutput[31:0], temprawoutput[63:32], temprawoutput[95:64],
temprawoutput[127:96],offset,outputdata);
        mux4 #(32) ruozhi(temprawoutput0, temprawoutput1, temprawoutput2, temprawoutput3,offset,outputdata);
        //this decide which particular space should the words be loaded.
        always @(posedge clk) begin
                if(enable) begin
                        if(wr) begin
                                if(whichcache) begin// reg2 is using
                                        case(cachesel)
                                                2'b00: cachereg20[index] = writedata;
                                                2'b01: cachereg21[index] = writedata;
                                                2'b10: cachereg22[index] = writedata;
                                                2'b11: cachereg23[index] = writedata;
                                        endcase
                                end
                                else begin
                                        case(cachesel)

                                                2'b00: cachereg10[index] = writedata;
                                                2'b01: cachereg11[index] = writedata;
                                                2'b10: cachereg12[index] = writedata;
                                                2'b11: cachereg13[index] = writedata;
                                        endcase
```

```
                    end
                end
            end
        end

endmodule
```

cachecontrol.v

```
module cache_control (input [31:0]aluout,
        //input  [31:0]writedatam,
        input  enable, clk, reset, mem_ready, memwrite,
        output [1:0]offset_top, [6:0]index,
        output which_cache, enable_cache, cache_wr,
        output [1:0]cache_sel,
        output mem_en, mem_rw,
        output [31:0] address,
        output cachedatainselect, stallm, ruozhiout);
        wire [1:0] offset1;

        wire [20:0]tag;
        //wire [6:0] index;
        reg u [127:0];
        reg [21:0] cachectr1 [127:0];
        reg [21:0] cachectr2 [127:0];

        assign tag = aluout[31:11];
        assign index = aluout[10:4];
        assign offset1 = aluout[3:2];

        wire available;
        assign available = (~cachectr1[index][21]) || (~cachectr2[index][21]);

        wire encoin1;
        assign encoin1 = (cachectr1[index][20:0] == tag) & cachectr1[index][21];
        wire encoin2;
        assign encoin2 = (cachectr2[index][20:0] == tag) & cachectr2[index][21];

        //wire [21:0] tempfordebug1,tempfordebug1;

        //determine which set of cache should be modified
        wire encoderout;
        assign encoderout = encoin1 ? 0 : (encoin2 ? 1 : 0);
        wire hit;
        assign hit = encoin1 | encoin2;

        wire toggle_u, change_tag, memadrsel;
        wire [1:0] offset2;
        wire setv;

        integer i;


        //set up the u,v,tag table
        always@(posedge clk) begin

                if(toggle_u)begin
                        u[index] = ~u[index];
```

```
            end
            if(setv) begin
                if(which_cache) cachectr2[index][21] <= 1;
                else cachectr1[index][21] <= 1;
            end
            if(change_tag) begin
                if(which_cache) cachectr2[index][20:0] <= tag;
                else cachectr1[index][20:0]<=tag;
            end
            if(reset) begin
                for(i = 0;i<10'b1111111111;i = i + 1) begin
                    u[i] = 0;
                    cachectr1[i][21] = 0;
                    cachectr2[i][21] = 0;
                end
            end
        end

        wire which_offset;

        //load u, v, hit signal to the FSM and output the signal
        FSM fsm (clk, reset, enable, encoderout, hit, u[index], mem_ready, memwrite, available, aluout[3:2] ,toggle_u,
change_tag, which_cache, enable_cache,
                cache_wr, cache_sel, mem_en, mem_rw, offset2, memadrsel, cachedatainselect,
stallm,setv,ruozhiout, which_offset);

        wire [31:0] addrencoin1, addrencoin2;
        assign addrencoin1 = {cachectr1[index][20:0], index, offset2, 2'b00};
        assign addrencoin2 = {cachectr2[index][20:0], index, offset2, 2'b00};
        //assign address = memadrsel ? encoin1 : encoin2;
        assign address = which_cache ? addrencoin2 : addrencoin1;

        assign offset_top = which_offset ? offset2 : offset1;


endmodule

//write = 0 is read otherwise write, hit = 0 is mis otherwise read

module FSM (input clk, reset, enable, encoderout, hit, u, mem_ready, write, v,
        input [1:0] aluoffset,
        output toggle_u, change_tag, which_cache, enable_cache, cache_wr,
        output [1:0] cache_sel,
        output      mem_en, mem_rw,
        output [1:0] offset,
        output memadrsel, cachedatainsel,
        output stallm,
        output setv,
        output ruozhiout, which_offset);
        //toggle_u toggles the u signal, change_tag tells the tag to be changed. Which_cache selects the cache we
want.
        // enable_cache enables the cache. Cache_wr tells if we read or write the cache.
        //cache _Sel tells which specific column should be chosen.
        //mem_en enables the memory.
        //mem_rw tells if we write or read from memory.
        //offset tells the wrods offset of the data
        //memadresel, selects the memory address.
        //cachedatainsel, selects the input source to cache
        //stallm means the system should stall
```

```
	//setv sets the v value in the table.
	//ruozhiout is a helper output
	//which_offset helps to selecet data to memory.
	reg [19:0] FSMcontrols;
	reg [3:0] state;
	//reg [3:0] nextstate;

	reg  tmp;
	wire [3:0] nextstate;
	assign {nextstate, setv, stallm, toggle_u, change_tag, which_cache, enable_cache, cache_wr, cache_sel,
		mem_en, mem_rw, offset, memadrsel, cachedatainsel, which_offset} = FSMcontrols;
	changedetector ruozhi2(state, clk, ruozhiout);




	/*always @(posedge reset) begin
			state <= 0;
			stallm <= 0;
		if(!reset)  state = nextstate;

	end*/
	/*always @(posedge clk) begin
		if(enable) stallm <=1;
	end*/

	always @(posedge clk, posedge reset)  begin
		if(reset) begin
	/*		state <= 0;
			stallm <= 0;
			nextstate <=0;
			setv <= 0;*/
		FSMcontrols = 19'b0;
		end
		else if(mem_ready) #1 state = nextstate;
	end


	//always @(posedge (clk&mem_ready)) begin
	always @(*) begin
		//#2;


		case(state)
			//idle state
			4'h0: FSMcontrols <= (enable) ? (hit? (write? {4'b0001,1'b0,1'b0,2'b00, encoderout, 8'b00000000,
encoderout, 2'b0}:
					{4'b0010,1'b0,1'b0,2'b00, encoderout, 8'b00000000, encoderout, 2'b0}) : ( v?
					{4'b0100,1'b0,1'b1,2'b00, encoderout, 8'b00000000, encoderout, 2'b0}:
					{4'b0011,1'b0,1'b1,2'b00, encoderout, 8'b00000000, encoderout, 2'b0})) : 0;
			//hit and write cache state
			4'h1: FSMcontrols <= {4'b0000,1'b0,1'b1,(u==which_cache),1'b0, encoderout, 2'b11, aluoffset,
4'b0000,
				encoderout, 2'b0};

			//hit and read cache state
			4'h2: FSMcontrols <= {4'b0000,1'b0,1'b1,(u==which_cache),1'b0, encoderout, 2'b10, aluoffset,
4'b0000,
```

```
                        encoderout, 2'b0};
                //writeback state start w0
                4'h3: FSMcontrols <= {4'b0101,1'b0,1'b1,1'b0,1'b0, u, 2'b10, 3'b001, 3'b100,
                        encoderout, 1'b0, 1'b1};
                //miss and write from memory to cache r0
                4'h4: FSMcontrols <= {4'b1000,1'b0,1'b1,1'b0,1'b1, u, 2'b11, 2'b00, 2'b10,2'b00,
                        encoderout, 2'b10};
                //w1
                4'h5: FSMcontrols <= {4'b0110,1'b0,1'b1,1'b0,1'b0, u, 2'b10, 2'b01, 2'b11,2'b01,
                        encoderout, 1'b0, 1'b1};
                //w2
                4'h6: FSMcontrols <= {4'b0111,1'b0,1'b1,1'b0,1'b0, u, 2'b10, 2'b10, 2'b11,2'b10,
                        encoderout, 1'b0, 1'b1};
                //w3
                4'h7: FSMcontrols <= {4'b0100,1'b0,1'b1,1'b0,1'b0, u, 2'b10, 2'b11, 2'b11,2'b11,
                        encoderout, 1'b0, 1'b1};
                //r1
                4'h8: FSMcontrols <= {4'b1001,1'b0,1'b1,1'b0,1'b0, u, 2'b11, 2'b01, 2'b10,2'b01,
                        encoderout, 1'b1,1'b0};
                //r2
                4'h9: FSMcontrols <= {4'b1010,1'b0,1'b1,1'b0,1'b0, u, 2'b11, 2'b10, 2'b10,2'b10,
                        encoderout, 1'b1,1'b0};
                //r3
                4'ha: FSMcontrols <= write ? {4'b0001,1'b1,1'b1,1'b0,1'b0, u, 2'b11, 2'b11, 2'b10,2'b11,
                        encoderout, 1'b1,1'b0} :
                        {4'b0010,1'b0,1'b1,1'b0,1'b0, encoderout, 2'b11, 2'b11, 2'b10,2'b11,
                        encoderout, 1'b1,1'b0};


            endcase
            if (state == 4'h3) $display("time is %0t",$time);
        end
endmodule
```

Data_Path.v

```
module data_path(input [1:0] pcsrcd, input clk, input rfreset, input se_ze, input eq_ne, input branchd,
                 input mul_signd, startmuld,
                 input [1:0] outseld,
                 input regwrited, memtoregd, memwrited,
                 input [3:0] alucontrold,
                 input memenabled,
                 input alusrcd, regdstd,
                 output [5:0] opd, output [5:0] functd, output equald);
        // fetch
        wire [31:0] pcplus4f, pcbranchd, jumpadr, pcnext, pcf, instrf;
        //dec
        wire [31:0] instrd, pcplus4d, rd1, rd2, immext;
        wire [31:0] equaldin1, equaldin2; // for equald input

        //exe
        wire mul_signe,startmule;
        wire [1:0] outsele;
        wire regwriree, memtorege,memwritee;
        wire [3:0] alucontrole;
        wire alusrce, regdste;
```

```
wire [31:0] rd1e, rd2e, immexte;
wire [4:0] rse, rte, rde;

wire [31:0] srcae, srcbe, writedatae;
wire [4:0] writerege;
wire [31:0] immsl16;
wire [31:0] aluoute;
wire [63:0] muloute;
wire [63:0] mulregout;
wire memenableE;

//mem
wire [31:0] aluoutm;
wire [1:0] outselm;
wire regwritem;
wire memtoregm;
wire memwritem;
wire [31:0] writedatam;
wire [4:0] writeregm;
wire [31:0] immextm;
wire [31:0] A;// also called slected aluoutm
wire [31:0] rdm;//readdatam
wire memenablem;
//wb
wire regwritew;
wire [4:0] writeregw;
wire [31:0] resultw;
wire memtoregw;
wire [31:0] readdataw, aluoutw;
//hazard use
wire stallf, stalld, forwardad,flushe;
wire forwardbd;
wire[1:0] forwardae, forwardbe;
wire isready;
wire ismflomfhi;
wire memstall; //mem enable when lw and sw
assign ismflomfhi = ({instrd[31:26], instrd[5:1]} == 11'b00000001001);

//pulse
wire memenablem_pulse;
posedgedetector ruozhi_edgedetector(memenablem, clk, memenablem_pulse);
reg pig;
always@* begin
	case(memenablem_pulse)
		1'b1: pig <= 1;
		default: pig<=0;
	endcase
end
wire pig2;
assign pig2 = pig|memstall;


//fetch
mux3 #(32) pcmux(pcplus4f, pcbranchd,jumpadr, pcsrcd, pcnext);

//enablereg #(32) pcreg(clk, ~stallf, pcnext, pcf);
flopr #(32) pcreg(clk, rfreset, ~(stallf | pig2), pcnext, pcf);
inst_memory i_mem(pcf, instrf); //instruction memory
assign pcplus4f = pcf + 32'b100; // PC+4
```

```
	fetchtodec f2d(clk, (pcsrcd[0] | pcsrcd[1]|rfreset), ~(stalld | pig2),instrf,pcplus4f,instrd,pcplus4d); // fetch to
decode stage register
	//dec
	assign jumpadr = {pcplus4d[31:28],instrd[25:0],2'b00}; // s25l2
	assign opd = instrd[31:26]; // op-code
	assign functd = instrd[5:0]; //func for control unit
	regfile rf(~clk, regwritew, rfreset, instrd[25:21], instrd[20:16], writeregw, resultw, rd1, rd2); //register file unit
	extend ext(instrd[15:0], se_ze, immext); // extent mudoule output as SignImmD labled in the diagram
	assign pcbranchd = {immext[29:0], 2'b00} + pcplus4d; // PCBranchD

	assign equaldin1 = forwardad ? A : rd1; //2to1 mux for equalD
	assign equaldin2 = forwardbd ? A : rd2; //2to1 mux for equalD
	assign equald = eq_ne ? (equaldin1 == equaldin2) : (equaldin1 != equaldin2); // the equal compare module
	// Decode to excuted stage register
	wire [31:0] instre;// for debug
	resetclearenablereg #(125+32) d2e(clk,rfreset, (flushe),~pig2, {instrd,mul_signd,startmuld,outseld,regwrited,
memtoregd,
memwrited,alucontrold,alusrcd,regdstd,rd1,rd2,instrd[25:21],instrd[20:16],instrd[15:11],immext,memenabled},
					{instre,mul_signe,startmule,outsele,regwritee, memtorege,
memwritee,alucontrole,alusrce,regdste,rd1e,rd2e,rse,rte,rde,immexte,memenableE});
	assign writerege = regdste ? rde : rte;// 2to1 mux for writeRegE
	assign srcae = forwardae[1] ? A : (forwardae[0] ? resultw : rd1e);// 3to1 mux for srcAE
	assign writedatae = forwardbe[1] ? A : (forwardbe[0] ? resultw : rd2e);// 3to1 mux for WriteDataE
	assign srcbe = alusrce ? immexte : writedatae;// 2to1 mux for SrcBE
	ALU alumodule(srcae, srcbe,alucontrole,aluoute);// ALU moudle
	//multiplier multmodule(srcae, srcbe, clk, startmule,mul_signe,muloute);// Multiplier module
	laji_mult multmodule(srcae, srcbe, clk, rfreset, startmule, mul_signe, muloute, isready);
	assign mulregout = muloute; // Multiplier register
	//multreg multregmodule(clk, muloute, mulregout);
	// excution to memory stage register
	wire [31:0] instrm;
	clearenablereg #(107+32) e2m(clk, rfreset,~pig2, {instre,outsele,
regwritee,memtorege,memwritee,aluoute,writedatae,writerege,immexte[15:0],16'b0,memenableE},
			{instrm,outselm,
regwritem,memtoregm,memwritem,aluoutm,writedatam,writeregm,immextm,memenablem});
	//4-1 Mux for Data Memory input A
	mux4 #(32) aluoutmmux(mulregout[63:32],mulregout[31:0],aluoutm,immextm,outselm,A);
	//wire memstall; //mem enable when lw and sw
	cachesystem cs(clk, rfreset, A, writedatam, memwritem, memenablem, rdm, memstall);// coment out to test
the performance

	//data_memory dmem(clk, memwritem, A, writedatam, rdm);// Data Memory module
	//Memory to WriteBack stage register
	clearenablereg #(71) m2w(clk, (rfreset), ~pig2,{regwritem,memtoregm,rdm,A,writeregm},
{regwritew,memtoregw,readdataw,aluoutw,writeregw});
	assign resultw = memtoregw ? readdataw : aluoutw;
	// Hazard Unit
	hazard_unit hazardmodule((isready & ~startmule), ismflomfhi,
instrd[25:21],instrd[20:16],rse,rte,branchd,writerege,memtorege,regwritee,writeregm,memtoregm,regwritem,writere
gw,regwritew,stallf,stalld,forwardad,forwardbd,flushe,forwardae,forwardbe);



endmodule
```