ECE 154B Lab Report 3

Ke Ding
Changsheng Su
Zhanglu Wang

Introduction:
In this lab we are implementing three types of branch predictor to improve the branch efficiency that is static predictor(always not taken), dynamic predictor and dynamic branch predictor (with one more level that is global branch history buffer)

Illustration of instruction:

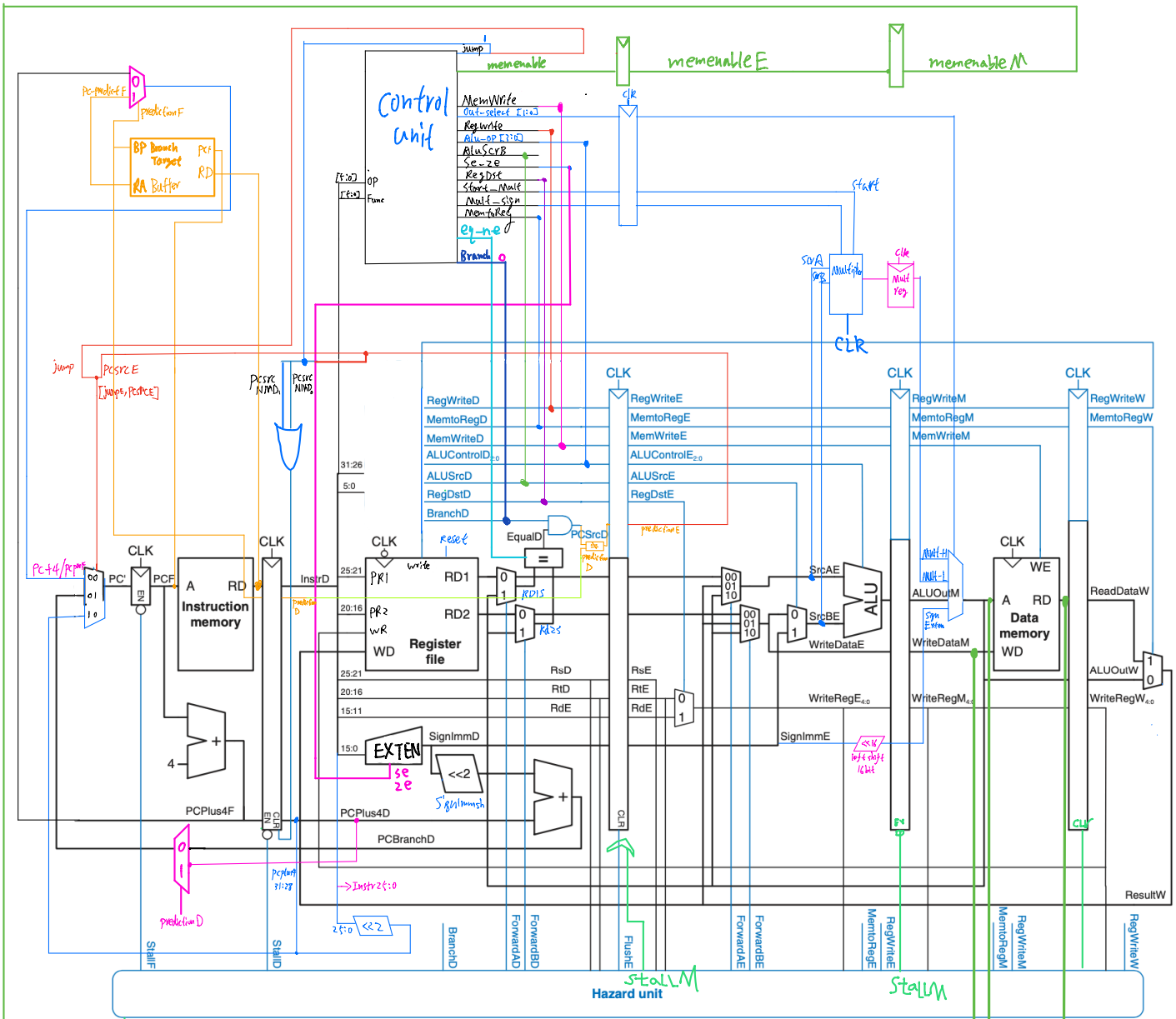
For step 1 static branch predictor.
For step 2 we need to design a dynamic predictor.
For step 3 we need to add one more level to our dynamic predictor with global history buffer.

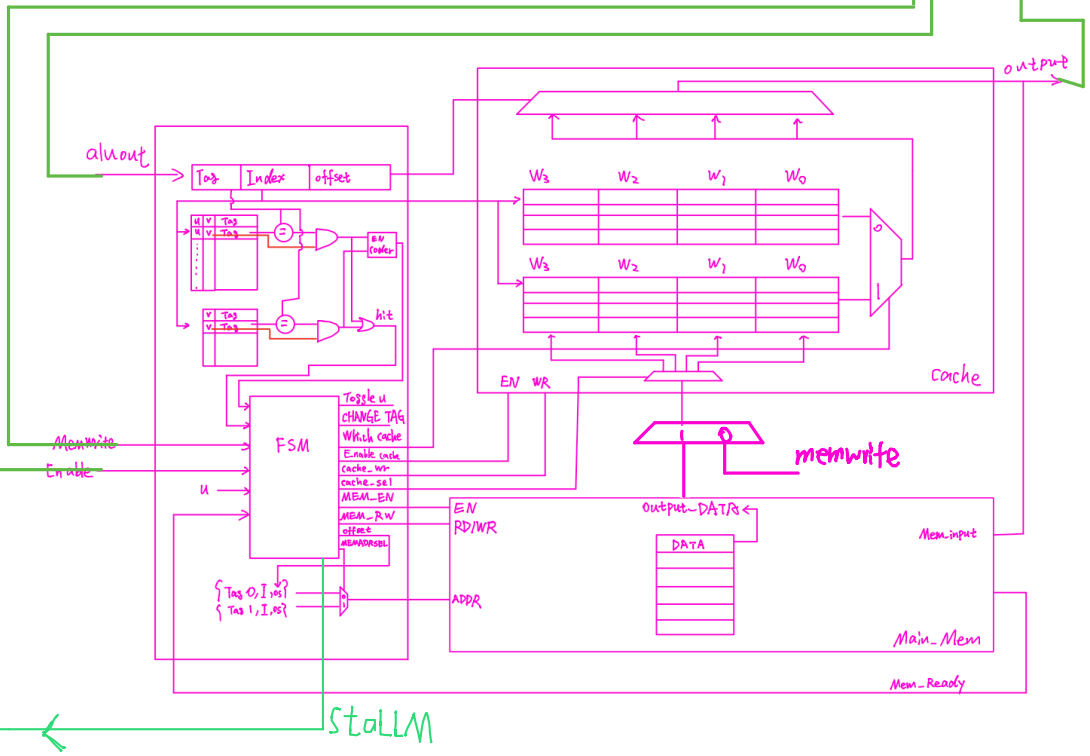
Design methodology:

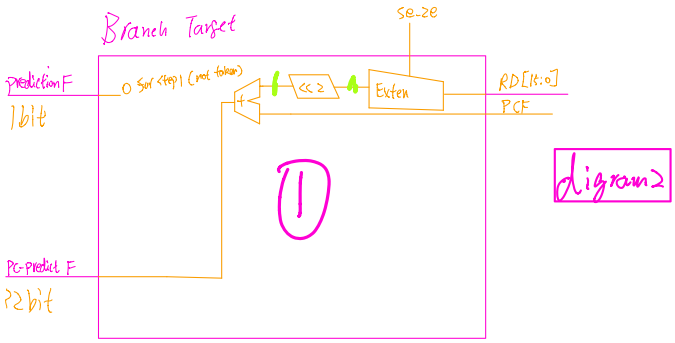
For **static branch predictor**, we treat branch as always not taken, use PC+4 as our next PC, and check our prediction of correctness in decode stage. If we see it is different than our prediction then we know we predict it wrong and we flush the fetch to decode register and use our calculated branch address from our decode stage as our nextPC. If we see it it not different than our prediction then we know we predict it correctly, so we can directly use our pc+4 instruction for the following stages.

For **dynamic branch predictor**. There is a branch history table, which will predict the outcome of the branch, and a target address buffer, which will store the target address in case of predicted taken branch. In our project, these modules are used in IF stage, in parallel to the fetching of the instruction. This will allow to supply the predicted branch address for the next instruction without stalling pipeline if we predict it correctly and such PC is stored in our table. There are two bits per branch in history table, and there are 2^7 entries in the history table and branch target predictor. For each branch, the two bits are used to encode the four states of finite state machine. At the beginning, the BHT is empty (zero) and it corresponds to NN state of the state machine. Note that two bit saturation counter has to be always updated when branch outcome is determined.

For our third predictor, **two-level correlating global predictor**, we add one more level to it that is global branch history. It is basically the same as step 2, except we add 4 entries in each branch. We will assume that there are two bits in the global history register, and therefore there will be 4 entries per each branch . Note that now both global history register and corresponding two bits of global history table should be updated by the end of execution stage for all three cases that is 1) not in the table but branch taken, 2) in the table but branch not taken, 3) in the table and branch taken.


memenable
memenableE
memenableM
Control unit
MemWrite
RegWrite
ALUSrcB
Se_ze
RegDst
Start_Mult
Mult_sign
MemtoReg
eq_ne
Branch
jump
PCSrcE
PCSrc1,PCSrc0
RegWriteD
MemtoRegD
MemWriteD
ALUControlD2:0
ALUSrcD
RegDstD
BranchD
RegWriteE
MemtoRegE
MemWriteE
ALUControlE2:0
ALUSrcE
RegDstE
RegWriteM
MemtoRegM
MemWriteM
RegWriteW
MemtoRegW
EqualD
PCSrcD
reset
Instruction memory
Register file
Data memory
ALU
EXTEN
SignImmD
SignImmE
SrcAE
SrcBE
WriteDataE
WriteDataM
ALUOutM
ALUOutW
ReadDataW
ResultW
WriteRegE4:0
WriteRegM4:0
WriteRegW4:0
RsD
RtD
RdD
RsE
RtE
RdE
PCPlus4F
PCPlus4D
PCBranchD
InstrD
StallF
StallD
FlushE
ForwardAD
ForwardBD
ForwardAE
ForwardBE
Hazard unit
stallM
stallM


output
aluout
Tag
Index
offset
W3
W2
W1
W0
hit
Cache
EN
WR
memwrite
FSM
MemWrite
Enable
Toggle u
CHANGE TAG
Which cache
Enable cache
cache_wr
cache_sel
MEM_EN
MEM_RW
offset
MEMORYSEL
u
EN
RD/WR
ADDR
output_DATA
DATA
Mem_input
Main_Mem
Mem_Ready
stallM



Step 1: Static Branch Predictor.

For the diagram above, is the data path for the static branch predictor.

Step description:
In this step we always predict the branch is not taken and pass PC+4 as our next PC.
In decode stage we will calculate the prediction outcome as you can see in our decode stage we check if our prediction signal and pass as PredictionD1, and pass this value back to fetch stage as the two bit selector and if our prediction is correct it proceed to the normal instruction. If it is incorrect, that means branch taken, our nextPC will be the branch target address and the instruction at this time from fetch to decode stage will be flushed. If our prediction is correct, there is no penalty any branch can be done in 1 cycle. If it is incorrect it will have a penalty with 1 cycle.

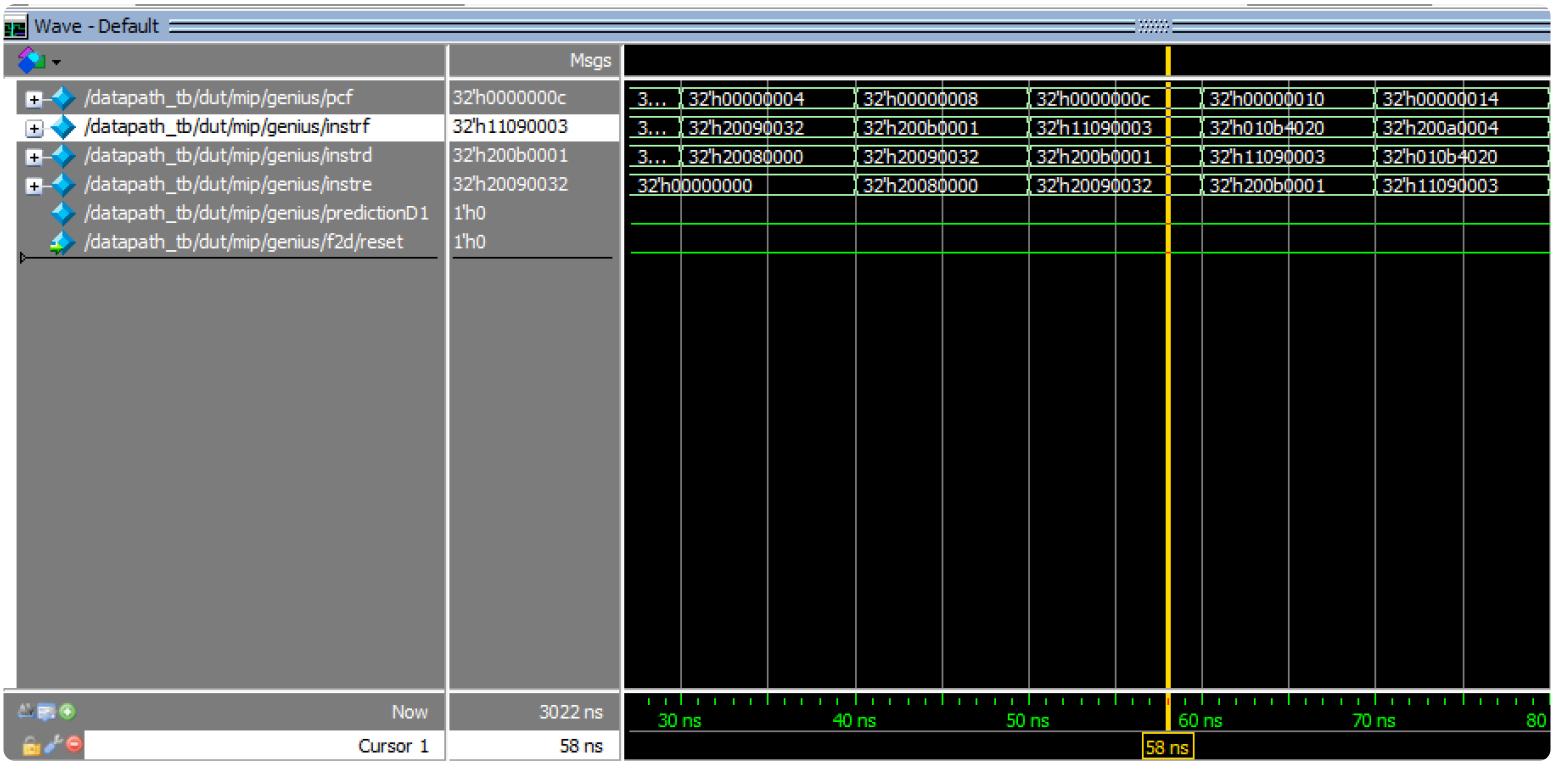
How does this improve the performance?
It improves performance by reducing the stall for correct prediction branch.

How do wee deal with EX, MEM and WB registers when prediction is wrong?
When our prediction is wrong, the instruction from fetch to decode is flushed. Thus the instruction in following stage EX, MEM and WB registers will also be cleared.

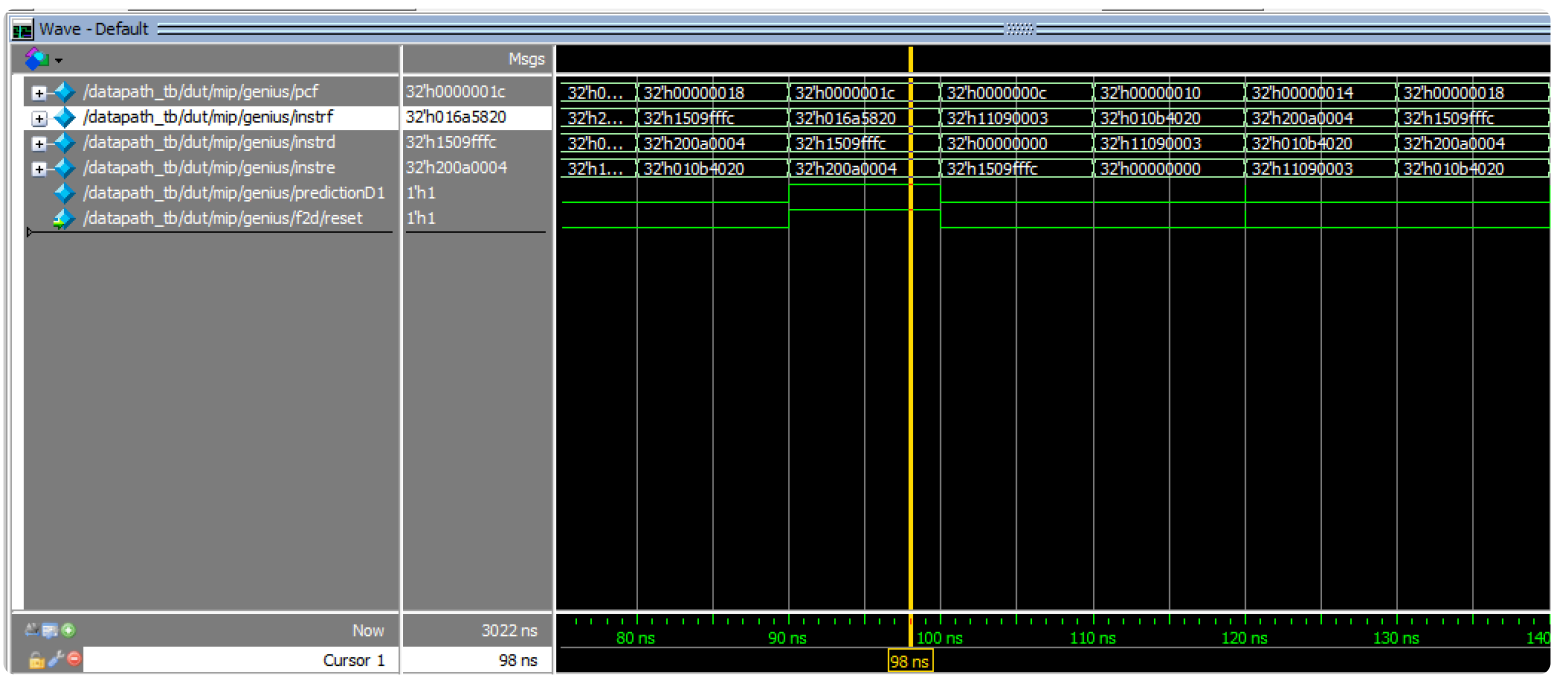
To test our program we designed following test program to test it:

	instruction	address	hexcode
Main:			
	addi t0 zero 0x0	00	20080000
	addi t1 zero 0x32 // t1 =50	04	20090032
	addi t3 zero 0x1	08	200B0001
Loop:			
	beq t0 t1 0x3 // if taken will move to end	0C	11090003
	add t0 t0 t3	10	010B4020
	addi t2 zero 0x4	14	200A0004
	bne t0 t1 0xFFFC	18	1509FFFC
End:			
	add t3 t3 t2	1C	016A5820
	j 0x3	20	08000003

Waveform Analysis



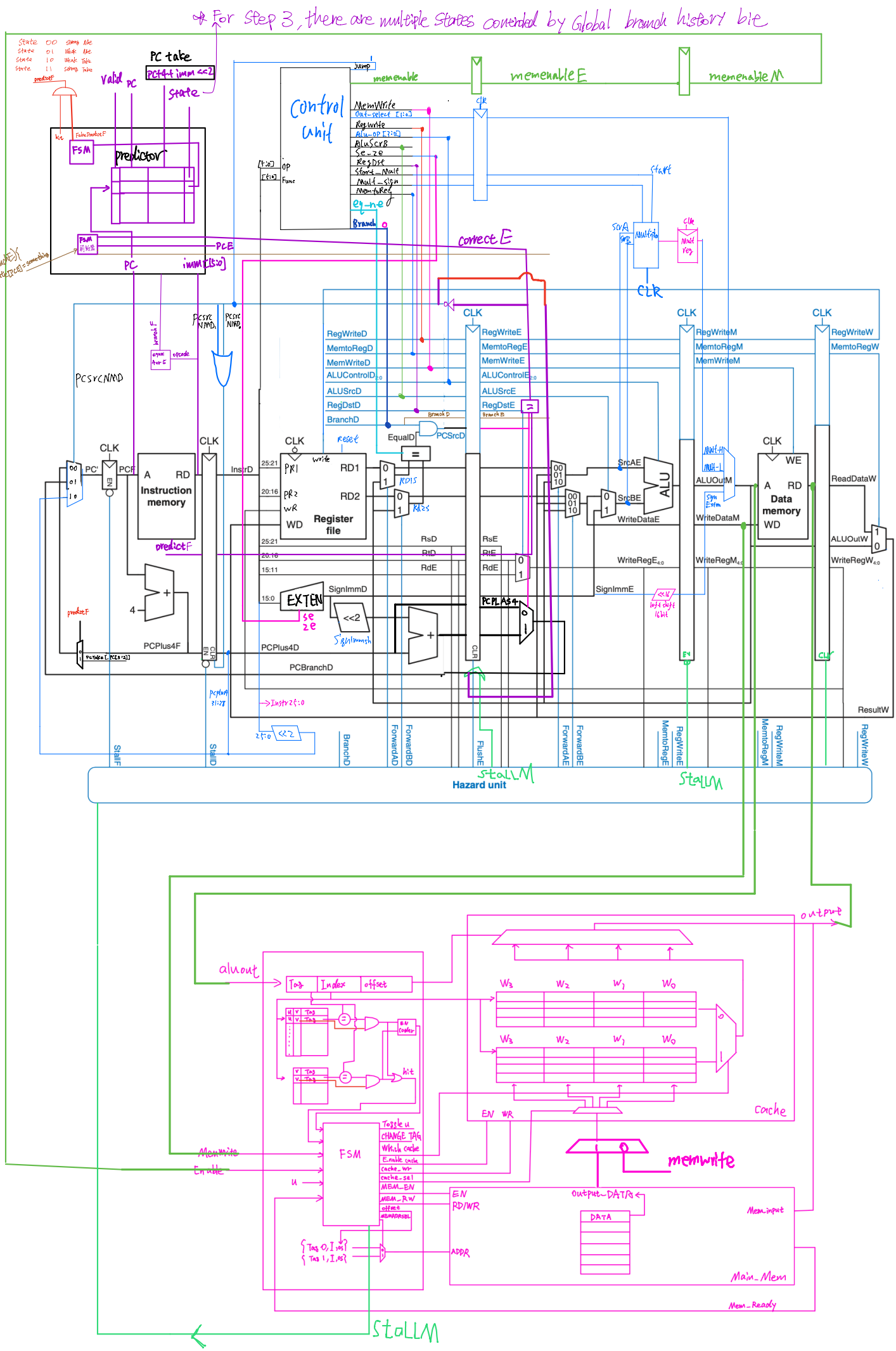
For the first time, when we run the instruction "beq t0 t1 0x3" which is '0x11090003', we predict it as always not take. Our predictionD1, test if the actual result is different than our prediction result. The actual result is not taken and prediction result is also not taken, thus we predict it correctly and there will be no penalty in this case, the program can execute normally, and choose PC + 4 as our next instruction, as you can see our next instruction is 0x0104b4020 and it proceed to decode stage with no stall or flush.



Here is the case that we predict it wrong. That is "bne t0 t1 0xFFFC" (0x1509fffc). Here we predict it as not taken and we see put pc+4 that is 0x016a5820 as our next instruction, at the same time, in decode stage, we detects that our prediction of not taken is different than actual result which is taken, thus predictionD1 will be 1, and the reset signal for fetch to decode register will be flushed. As you can see the undesired instruction 0x016a5820 is flushed in decode stage and also the following stages. And the correct branch address corresponding instruction 0x11090003 will be our next valid instruction.

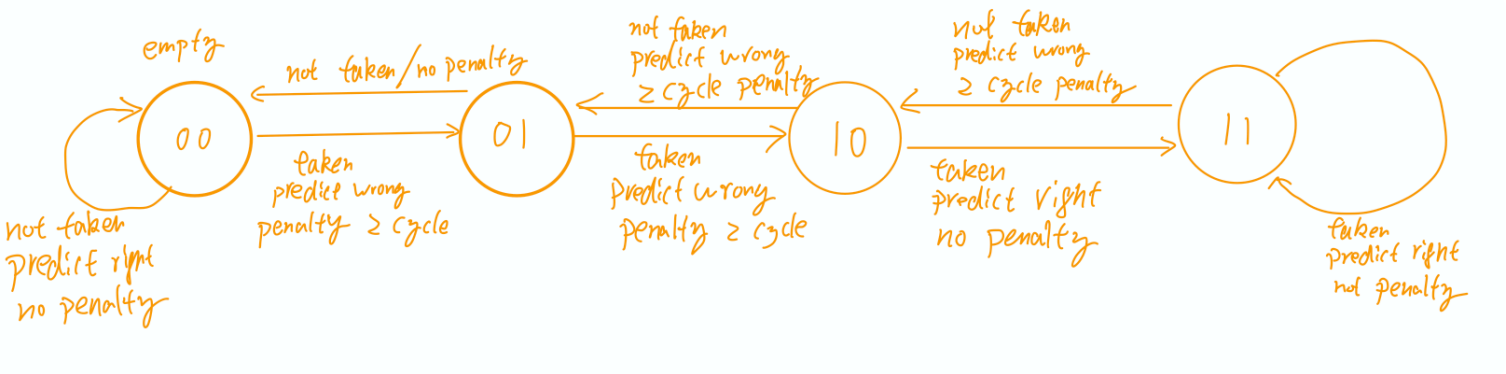
Code for this step will be shown in last part of this document:

Step2: Dynamic branch predictor:


* For Step 3, there are multiple States controlled by Global branch history bit
State 00 Strong Not
State 01 Weak Not
State 10 Weak Take
State 11 Strong Take
predictF
PC take
PCF=t imm <<2
Valid PC
state
FSM
predictor
wire
FakePredictF
PCE
PC
imm[15:0]
Control unit
Jump
memenable
memenableE
memenableM
MemWrite
Out_select [1:0]
RegWrite
ALU_OP [2:0]
ALUSrcB
Se_ze
RegDst
Start_Mult
Mult_sign
MemtoReg
eq_ne
Branch
op
Func
correct E
start
SrcA
Multiplier
Mult Reg
Mult Reg
CLK
RegWriteD
MemtoRegD
MemWriteD
ALUControlID2:0
ALUSrcD
RegDstD
BranchD
RegWriteE
MemtoRegE
MemWriteE
ALUControlE2:0
ALUSrcE
RegDstE
BranchE
RegWriteM
MemtoRegM
MemWriteM
RegWriteW
MemtoRegW
PCSrcD
PCSrcM
PCSrcWMD
opcode
funct
reset
EqualD
PC'
PCF
Instruction memory
InstrD
25:21
20:16
15:11
15:0
PR1
PR2
WR
WD
RD1
RD2
Register file
RD1S
RD2S
RsD
RtD
RdD
RsE
RtE
RdE
SignImmD
EXTEN
se
ze
<<2
SignImmsh
PCPlus4F
PCPlus4D
PCBranchD
PCPLAS4
SrcAE
SrcBE
ALU
WriteDataE
WriteRegE4:0
SignImmE
ALUOutM
WriteDataM
WriteRegM4:0
Mult-H
Mult-L
Sign Extem
left shift 16bit
WE
Data memory
ReadDataW
ALUOutW
WriteRegW4:0
ResultW
StallF
StallD
BranchD
ForwardAD
ForwardBD
FlushE
ForwardAE
ForwardBE
MemtoRegE
RegWriteE
MemtoRegM
RegWriteM
RegWriteW
stallM
Hazard unit
stallM
aluout
Tag
Index
offset
W3
W2
W1
W0
output
Cache
hit
EN
WR
memwrite
MemWrite
Enable
FSM
Toggle u
CHANGE TAG
Which cache
Enable cache
cache_wr
cache_sel
MEM_EN
MEM_RW
offset
MEMADRSEL
u
EN
RD/WR
ADDR
output_DATA
DATA
Mem_input
Main_Mem
Mem_Ready
stallM

STEP2:

For the diagram above, is the data path for the dynamic branch predictor. There is a branch history table in our branch target buffer. The very left column stores valid bit that is to see if this entry is valid. The next column is for PC, and the next column is for predicted PC and the last column is 2 bits value that encodes our state in FSM. Our FSM works as follow.



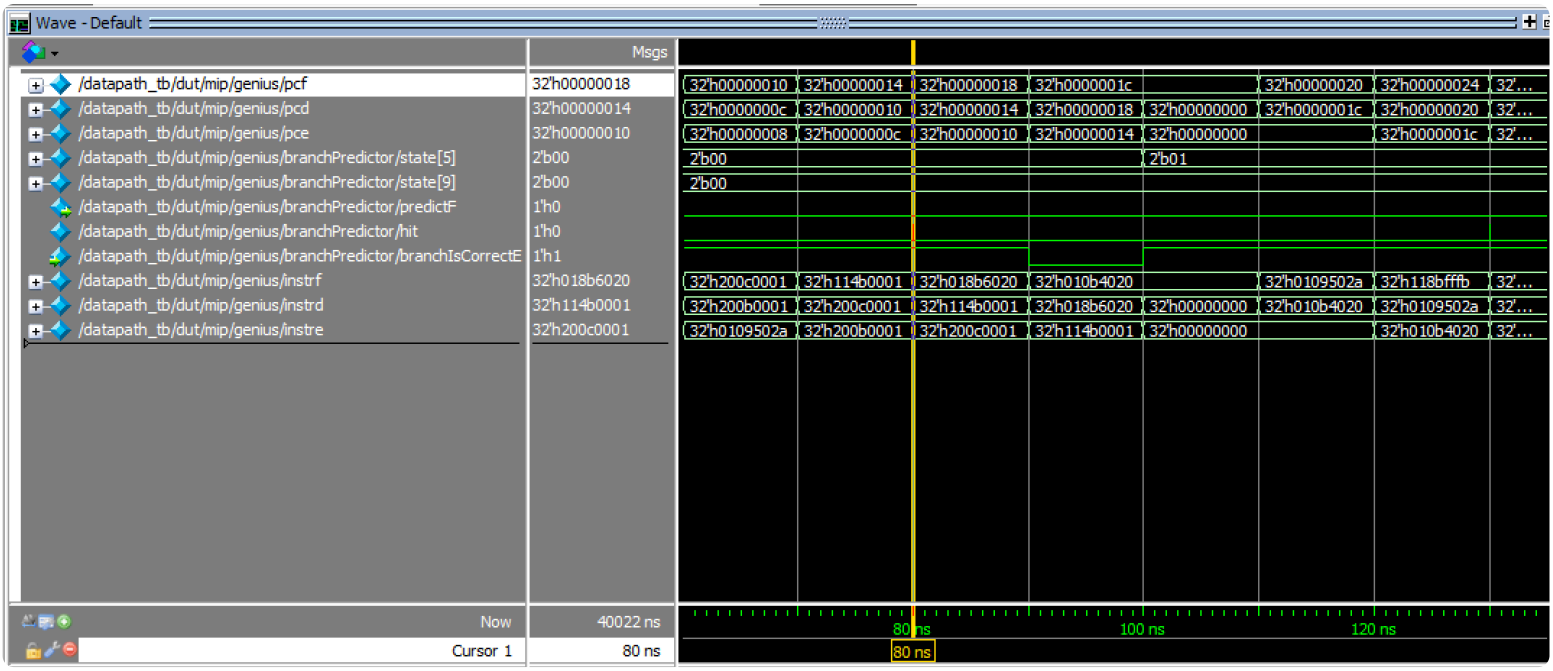
00: Strong not taken, 01: Weak not taken, 10: weak taken, 11: strong taken.
For all entries in the table, we initialized its state as strong not taken that is 00.

Every time PC comes we will check if it's on the table and check the state for prediction to determine if we use branch target address or pc+4. (00/01 is not taken we use pc+4 as predicted address and 10/11 is taken we use branch target address as our prediction outcome).

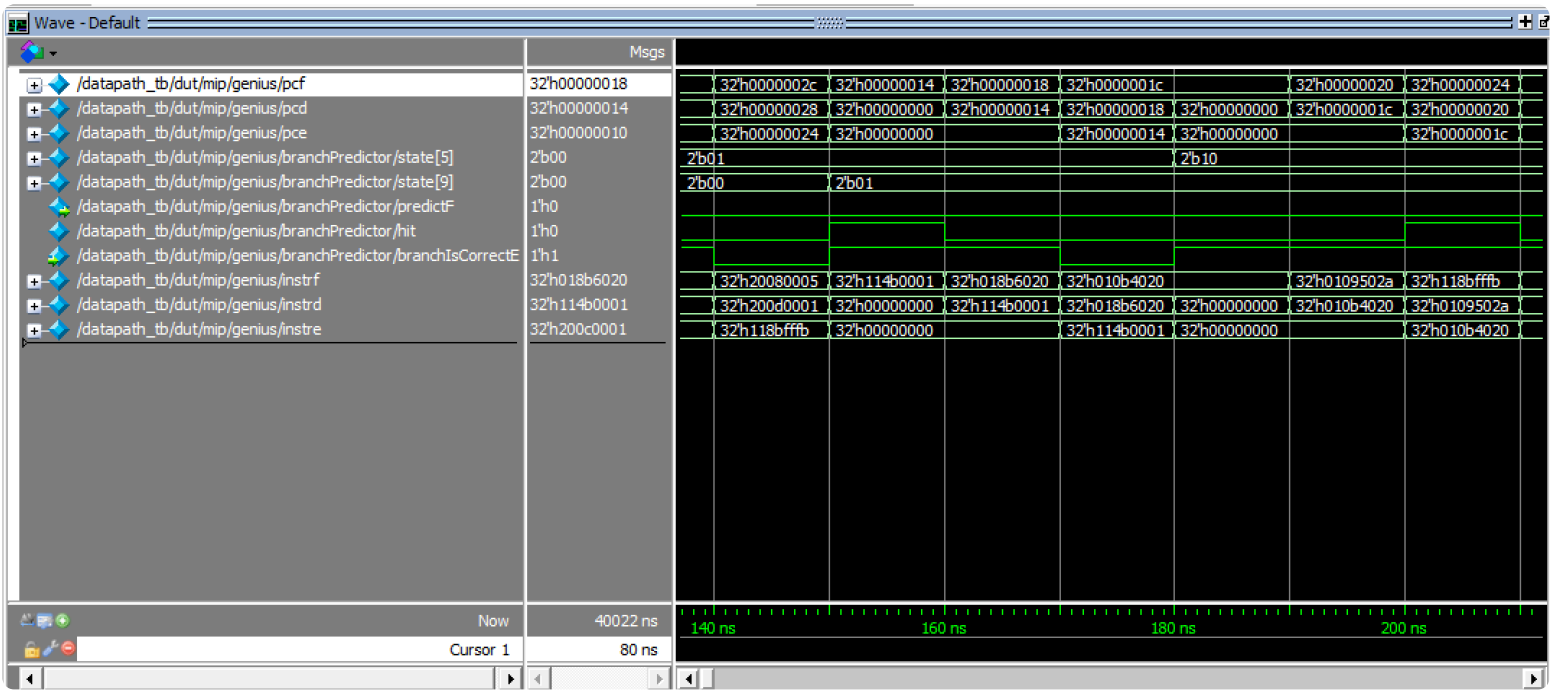
For this step we design a new testing program as follow:

addi t0 zero 0x0	//00 t0 = 0	20080000
addi t1 zero 0x6	//04 t1 = 6	20090006
Slt t2 t0 t1	//08 t2 = t0<t1	0109502A
Addi t3 zero 0x1	//0c t3 = 1	200B0001
Addi t4 zero 0x1	//10 t4 = 1	200C0001
Beq t2 t3 0x1	//14 if t2 == t3 taken branch (1c) else not taken, 18	114B0001
Add t4 t4 t3	//18 t4 = t4 + 1	018B6020
Add t0 t0 t3	//1c t0 = t0 + 1	010B4020
Slt t2 t0 t1	//20 t2 = t0 < t1	0109502A
Beq t4 t3 0xFFFB	//24 if t4 == 1 taken branch (14) else not taken	118BFFFB
Addi t5 zero 0x1	//28 t5 = 1 when all finish	200D0001
Addi t0 zero 0x5	//2c t0 = 0+3	20080005
Slt t2 t0 t1	//30 t2 = t0 < t1	0109502A
Addi t4 zero 0x1	//34 t4 = 1	200C0001
J 0x05	//38 back to beq t2 t3 0x1	08000005

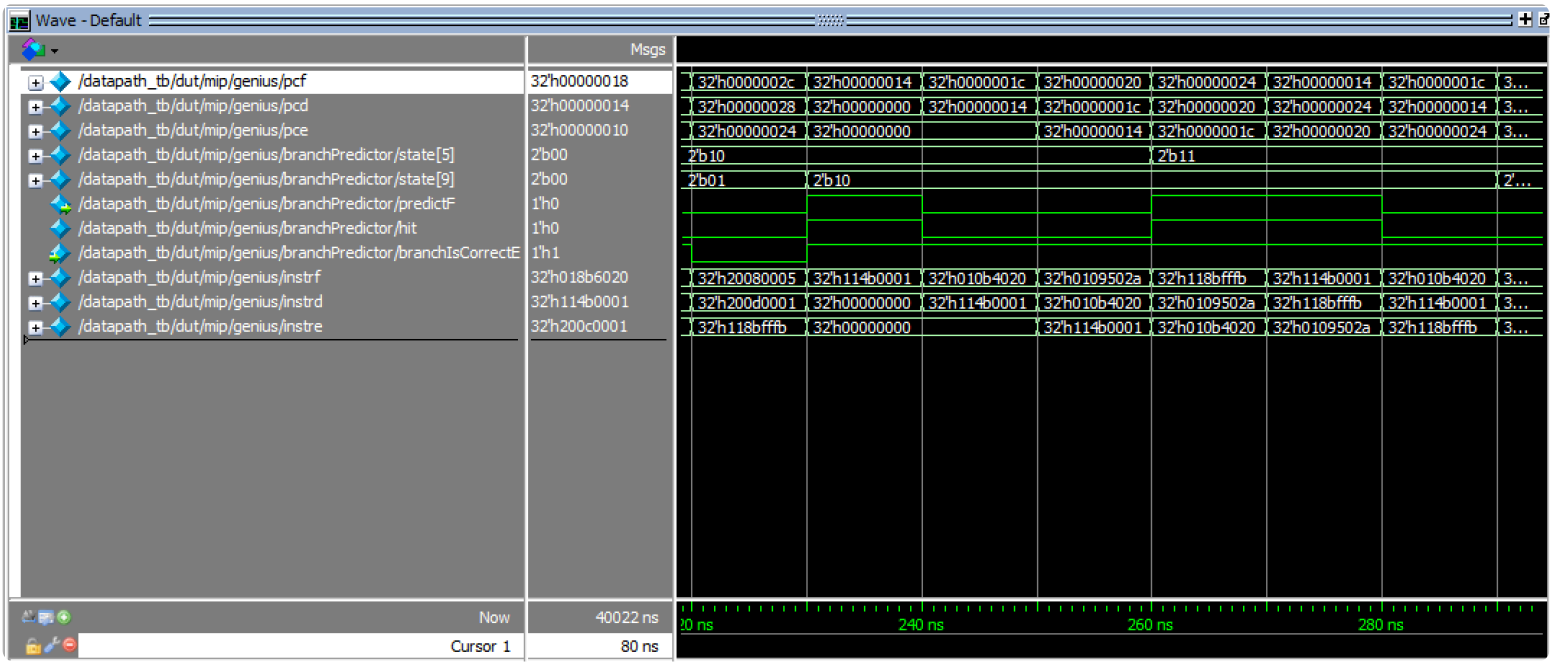
Our test program will take branch 6 times for PC14, (Beq t2 t3 0x1), and also for PC 24, (Beq t4 t3 0xFFFB). After that, both branch will not be taken and proceed to J0x05 eventually and then it will jump back to our first branch and this branch will be taken and go to next branch, this branch will also be taken. Thus to branch will be taken and then not taken next time, and proceed to jump, when it jumps back it repeat the same patter of instruction.



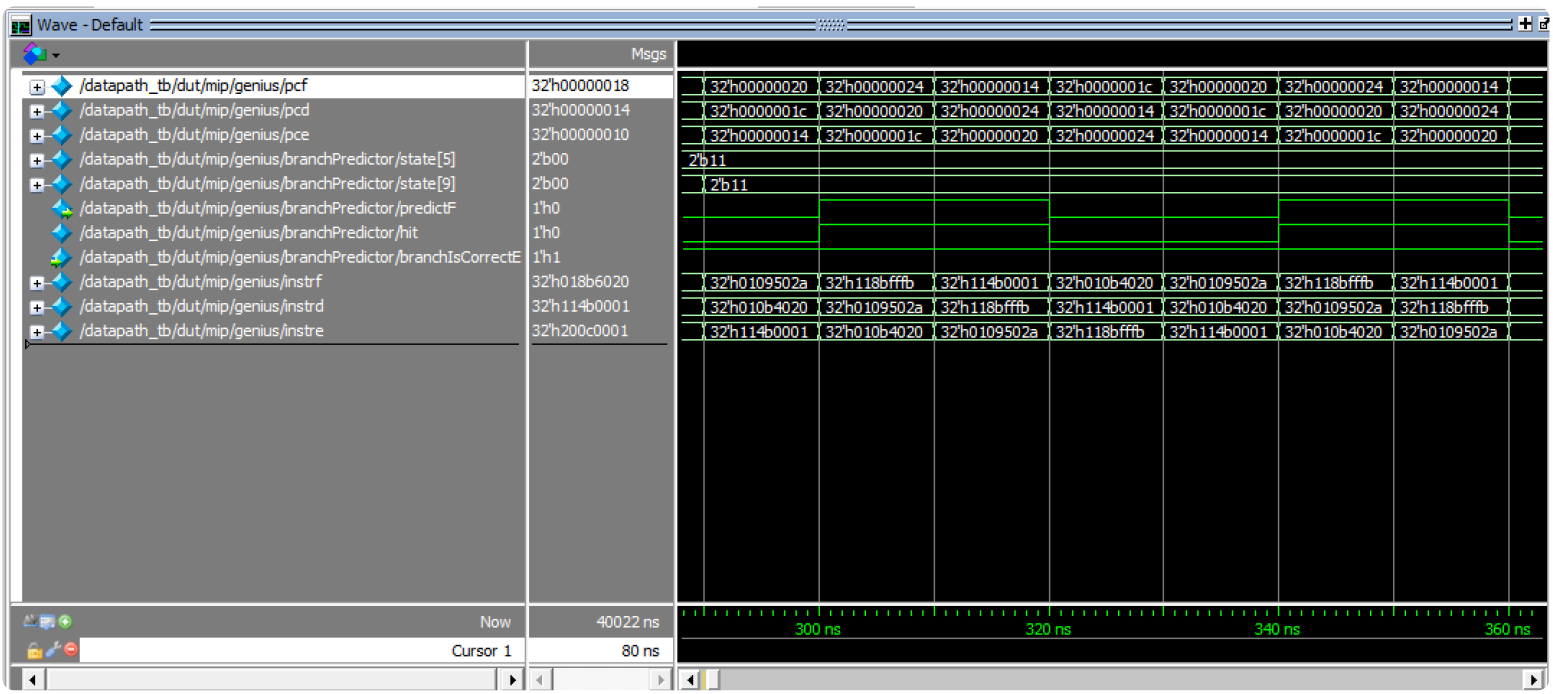
This is the example when the branch is first time taken. Nothing was in the branch history table and the default encode state is 00, that is strong not taken, thus we use pc+4 as our next instruction. At the decode stage we detects that our branch prediction is incorrect and we pass this signal to excuse stage and we have branchiscorrectE notifies us our prediction is incorrect. The table at the same time update its state value for specific index 5 that is for pc=14 from 2'b00 to 2'b01, which is decodes as weak not taken. We flush our previous predict nextPC value(PC+4) and restart it again and use correct branch address as our branch target address. Giving this two cycles of penalty for predicting incorrect action, and flush instruction in decode and execute



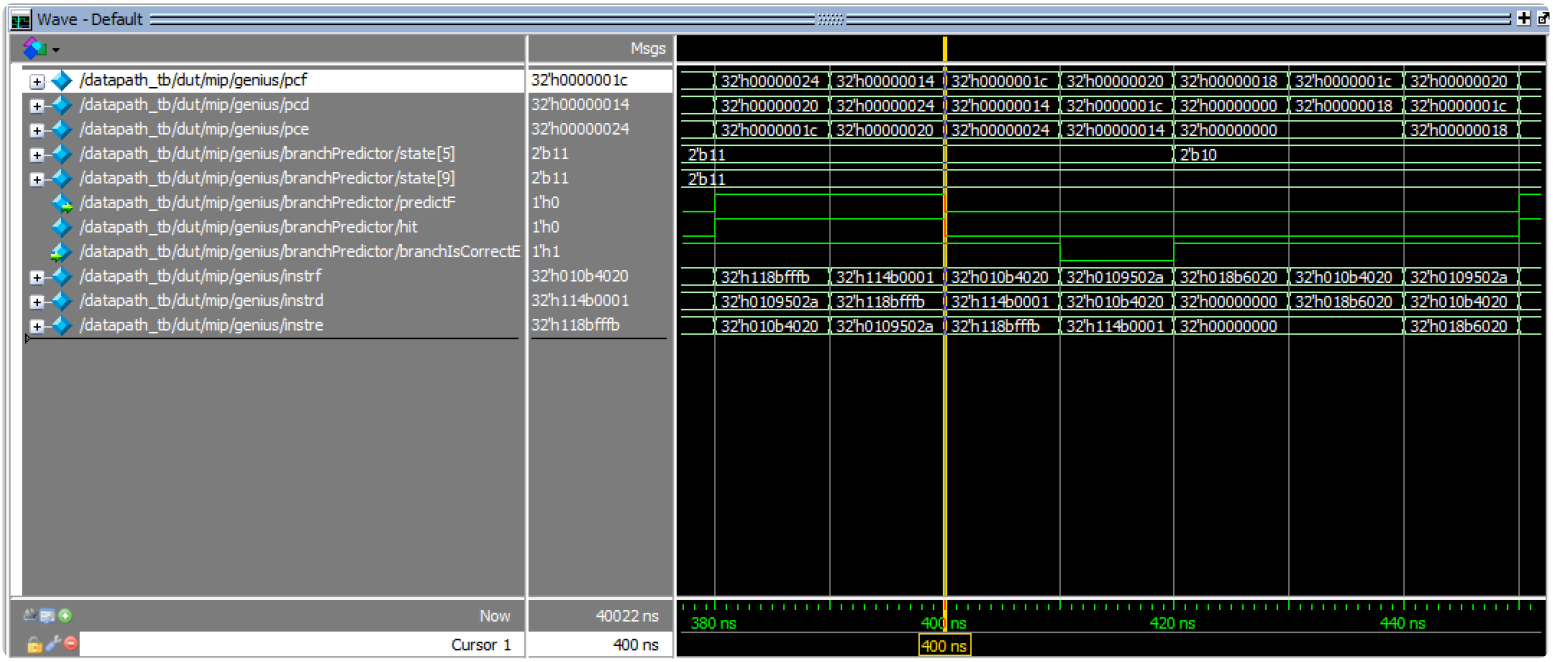
stage.
This case we look at pc = 14, corresponds to index = 5, which is when the branch is taken the second time, as we can see the hit signal is high which means we find the PC in the branch history table, and our state at this time is 2'b01 which is weak not taken so we predicts it as not taken. However our branchiscorrectE signal tells us we predict wrong and it is actually taken, so we update our state value to 2'b01 to 2'b10 (weak taken). We gives it penalty for predict it wrong and let it flush the wrong pc in



address.
In this case, we look at pc = 14 which corresponds to index = 5 which is when branch is taken the third time, we can see the hit signal is triggered that means we find it in the branch history table, this time our prediction state is 2'b10, which is weak taken, so we predict it as taken. In the decode stage we confirms that it predicts it correctly and pass that to execute stage and round back. This case there is no penalty for instruction since we predict it correctly and we update our state for this branch



entry to 2'b11 (strong taken).
This case we look at pc = 14 again that corresponds to index of 5. The hit signal tells us the pc is found in branch history table, and our state is 2'b11 which is strong taken, so we predict it as strong taken and feed it with our predicted pc which is 1c. BranchIscorrectE tells us we predict it correctly and thus there is no penalty for this prediction.

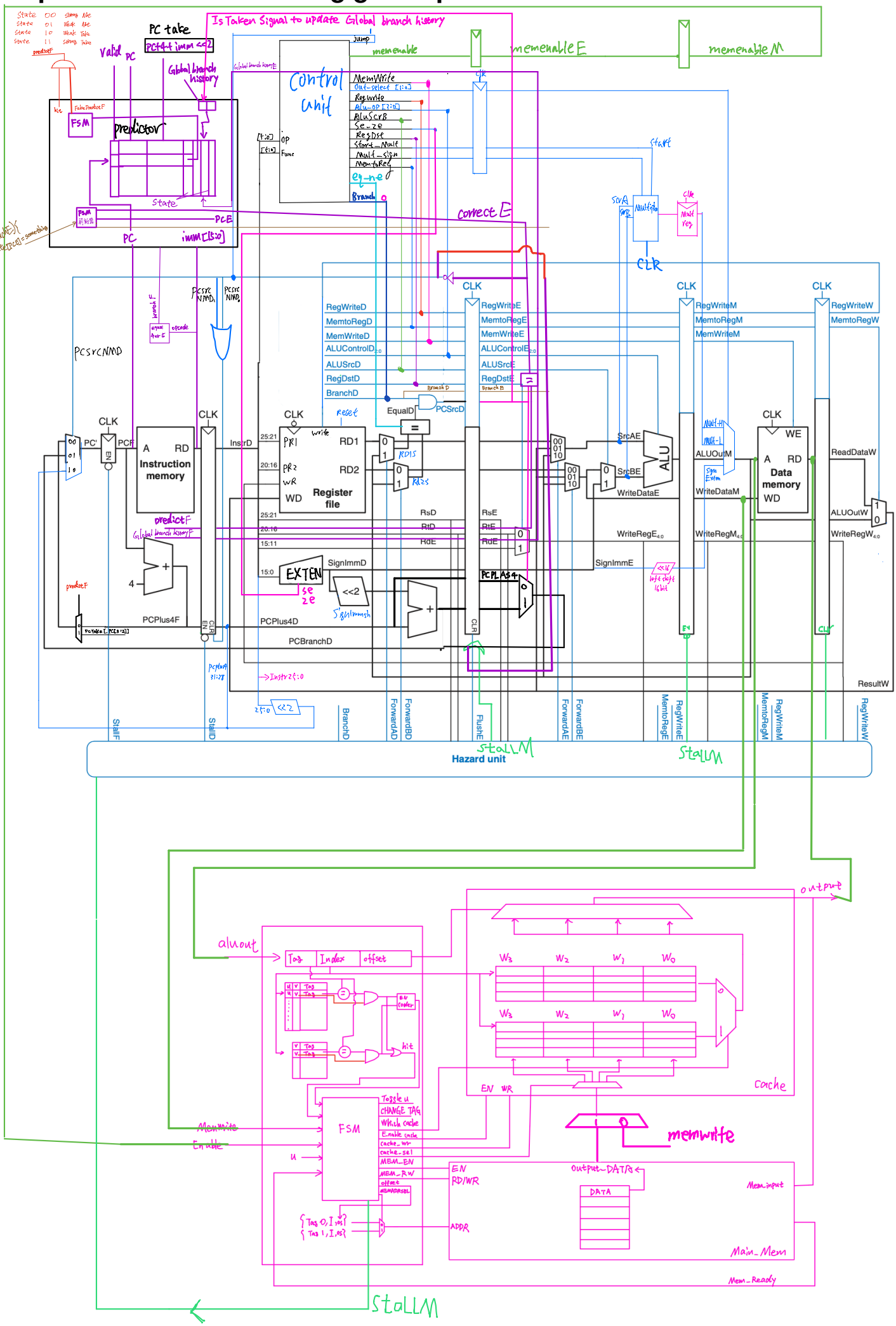


In this case we look at pc = 14 again, corresponds to index = 5. The hit signal tells that the PC is found in the table. The state of this is 2'b11 which is strong taken, so we predict it as taken. BranchIsCorrectE signal is 0 which tells us we predict it wrong, and so we give it penalty to stall additional cycles and flush the signal in decode and execute stages. We also update the state from 2'b11 to 2'b10 as weak taken.

After 5 taken and 1 not taken, the jump will jump back to pc=14 instruction that is branch instruction. This program will run in to the infinite loop that the branch will taken once and not taken once. In this case as we can see from the waveform the bbranchIsCorrectE for pdf will always be false for not taken since it goes from weak taken to strong taken and repeat infinitely.

Code for this step will be shown in last part of this document:

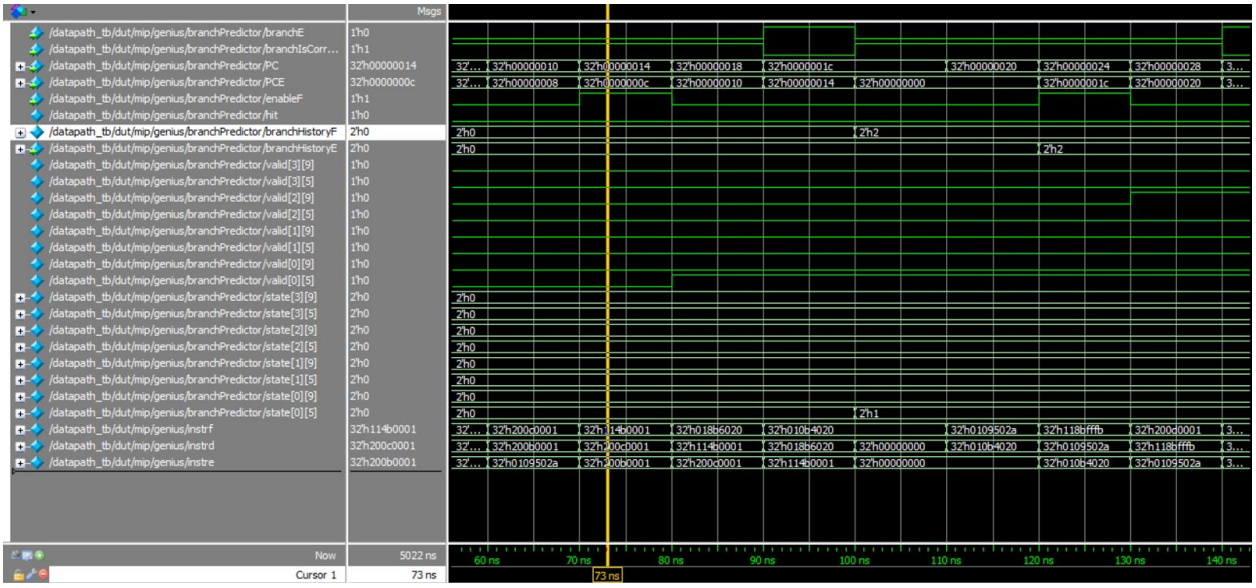
Step3: two-level correlating global predictor



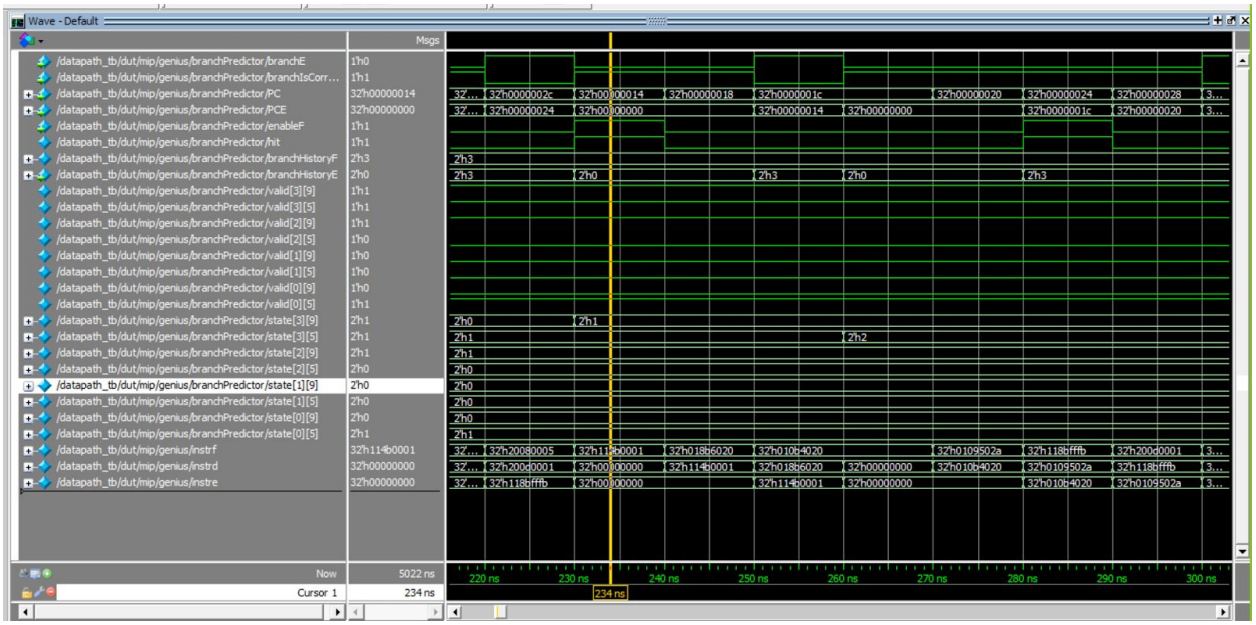
This is our diagram for the step 3. Basically this is similar to what we have in step 2. This new predictor contains 4 entries per slot of instructions and there is a global branch history register(branchHistoryF). Even for instructions have same program counter number the state could be difference (also we have 4 valid for each index). To choose which predictor to use, we use global history register. This two bit register indicates the last two branch instruction were taken or not. The left one indicates last branch instruction was taken or not and the right one indicated the second last instruction was taken or not. This was updated when every branch instruction's execute cycle.

State machine are basically same to the step 2. Except we have a "branch history delayed (branch history F(branchHistoryF) pass into 2 of pipeline registers(f2d, d2e) and become branchHistoryE)" to choose which state of that instruction(pce) to update.

This is step3, everything is same to step 2 except there are more registers in BranchPredictor. If branch predicts wrong, it would correct itself as step2.

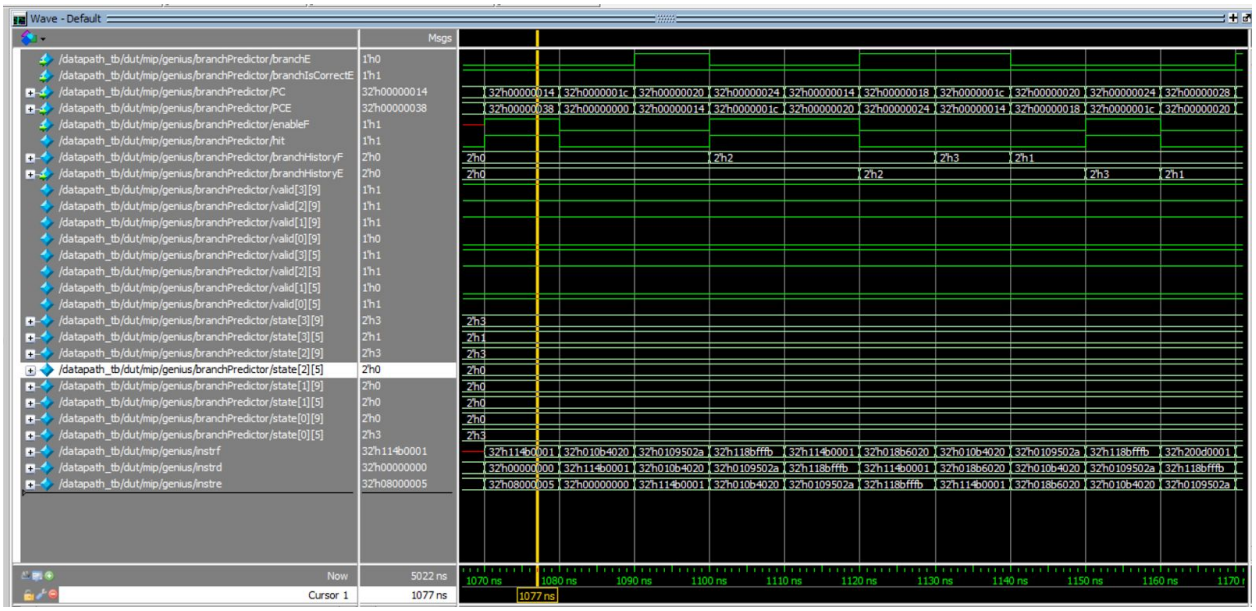


We can see in this waveform, where the first branch instruction PC[0x00000014] was taken. The Global history register contains only zero, so it choose state[0] as its state. Same as step 2, the states are initially 0, so it choose not taken.



Here is instruction with same PC[0x00000014] (in a loop). Here since branch history is 11, we choose from state[3]. Note that it updates a specific state(state[3]) for this PC, which is not same to the one above. The global history register controls which predictor to use and each predictor have their own state. These predictors work independently. We can see here even there is a

valid bit for this branch instruction (valid[2][9]) we still need to update valid[3][9] for this specific BHT is 11 case.



Here is our infinite loop. The test code is same as step2: branch instruction is toggling from taken and not taken. Unlike step2, our predictor perform pretty well in step 3. Note that at 1070ns and 1110ns we both have branch instruction PC[0x00000014], and the state doesn't change at all. But with choosing different predictor, we are able to deal with this case.

Code for this step will be shown in last part of this document:

Conclusion:
As our lab, for the static branch predictor is good for the program that many branch is not taken, and for dynamic branch predictor it good for the program that we do not know the instructions and it will fit the branch taken automatically. And for the last one, it is good for the program that have many various cases for the brach taken, especially for the repeating brach taken program, just like our program it finally turns into a infinity loop, and we can see in the wave form, the performance become much quicker when it turns into the infinity loop.
For this Lab, we spend 40 hrs for this lab, and we learned how to implement the branch predictor and how it works. No mistakes found so far and the references is the lecture Power Point.

Code for step 1:

Branchpredict.v:

```
module StaticBranchBuffer (input [15:0]rd, [31:0] pcf, output predictionF, output
[31:0]pc_predictF);
    wire [31:0]rd1;
    wire [31:0]rd2;
    assign rd1 = {{16{rd[15]}}, rd};
    sl2 s (rd1, rd2);
    assign pc_predictF = rd2 + pcf;
    assign predictionF = 0;
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;
    wire [1:0] firstsel;
    //dec
    wire [31:0] instrd, pcplus4d, rd1, rd2, immext;
    wire [31:0] equaldin1, equaldin2; // for equald input
    wire pcsrcpredictD;
    wire predictionF;
    wire [31:0] pc_predictF;
    wire [31:0] pc4orpref;
    wire [31:0] pcbranchd2;
    wire predictionD;
    wire predictionD1;

    //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;
    wire pcsrcpredictE;
    wire predictionE;

    //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(pc4orpref, pcbranchd2,jumpadr, firstsel, pcnext); // selector need to be
changed ????????????

    //enablereg #(32) pcreg(clk, ~stallf, pcnext, pcf);
    flopr #(32) pcreg(clk, rfreset, ~(stallf | pig2), pcnext, pcf);
    StaticBranchBuffer sbb (instrf[15:0],pcf,predictionF,pc_predictF);
    mux2 #(32) staticmux(pcplus4f, pc_predictF,predictionF,pc4orpref);
    inst_memory i_mem(pcf, instrf); //instruction memory
    assign pcplus4f = pcf + 32'b100; // PC+4
    mux2 #(32) predfromD(pcbranchd,pcplus4d,~predictionD1,pcbranchd2);
    fetchtodec f2d(clk, (pcsrcd[1]|rfreset | predictionD1), ~(stalld | pig2),instrf,
            pcplus4f,predictionF,instrd,pcplus4d,predictionD); // 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
    assign predictionD1 = (pcsrcd != predictionD);

    // Decode to excuted stage register
    wire [31:0] instre;// for debug
    resetclearenablereg #(125+33) 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,predictionD1},
                        {instre,mul_signe,startmule,outsele,regwritee, memtorege,
memwritee,alucontrole,alusrce,regdste,rd1e,rd2e,rse,rte,rde,immexte,memenableE,predictionE});
    assign firstsel = {pcsrcd[1],predictionD1};
    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
    /*wire mem_ready, memenable2; //mei yong de
    changedetector2(memenablem, memenable2);
    memory mem (clk, rfreset, memenablem2, memwritem,
        A,
        writedatam,
        rdm,
        mem_ready);
    assign memstall = mem_ready;*/
    //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,writeregw,regwritew,stallf,stalld,forwardad,forwardbd,flushe,forwardae,forwardbe);



endmodule
```

Code for step 2:

DynamicPredictBuffer.v:

```
module PCBranchBufForStep2(input clk, reset, enableF, branchE, branchIsCorrectE,
              input [31:0] PC, PCE,
```

```
            input [15:0] immF,
            output predictF,
            output [31:0] PCIfTaken);

    reg valid[127:0];
    reg [31:0] PCRam[127:0];
    reg [31:0] PCTake[127:0];
    reg [1:0] state[127:0];
    integer i;

    wire hit;
    assign hit = valid[PC[8:2]] & (PC == PCRam[PC[8:2]]);

    assign predictF = state[PC[8:2]][1] & hit;
    //assign PCIfTaken = valid[PC[8:2]] ? 0 : PCTake[PC[8:2]];
    assign PCIfTaken = PCTake[PC[8:2]];
    always @(posedge clk, reset) begin
        if(reset) begin
            for(i = 0;i <128 ;i = i + 1) begin
                valid[i] <= 0;
                state[i] <= 0;
                PCRam[i] <= 0;
                PCTake[i] <= 0;
            end
        end
        if(enableF) begin
            if(~hit) begin // if miss
                PCRam[PC[8:2]] = PC;
                PCTake[PC[8:2]] = PC + 4 + {{14{immF[15]}},immF,2'b00};
                valid[PC[8:2]] = 1;
                state[PC[8:2]] = 0;
            end

        end
        if(branchE) begin // update state
            case({state[PCE[8:2]], branchIsCorrectE})
                3'b000: state[PCE[8:2]] = 2'b01; // Strong NotTake & incorrect
                3'b001: state[PCE[8:2]] = 2'b00; // Strong NotTake & correct
                3'b010: state[PCE[8:2]] = 2'b10; // Weak NotTake & incorrect
                3'b011: state[PCE[8:2]] = 2'b00; // Weak NotTake & correct
                3'b100: state[PCE[8:2]] = 2'b01; // Weak Take & incorrect
                3'b101: state[PCE[8:2]] = 2'b11; // Weak Take & correct
                3'b110: state[PCE[8:2]] = 2'b10; // Strong Take & incrrect
                3'b111: state[PCE[8:2]] = 2'b11; // Strong take & correct
                default: state[PCE[8:2]] = 2'b00;
            endcase

        end
    end


endmodule



/*
module DynamicPredicBuffer();

    reg [127:0] bht [65:0];
    wire [6:0] index;




endmodule

module takenFSM (input [1:0] state, input taken, clk, reset, output [1:0] nextstate, output
```

```
predictTaken);

reg [3:0] out;
assign {nextstate,predictTaken} = out;

//state 00 is strong not taken, state 01 is weak not taken, state 10 is weak taken, state 11
is strong taken.

    always @* begin
        case (state)
            2'b00:
                out <= taken ? 3'b010 : 3'b000;
            2'b01:
                out <= taken ? 3'b101 : 3'b000;
            2'b10:
                out <= taken ? 3'b111 : 3'b010;
            2'b11:
                out <= taken ? 3'b111 : 3'b101;
        endcase
     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, pcTakeD, pcTakeE, pcbranche, jumpadr, pcnext, pcf, instrf;
    //dec
    wire [31:0] instrd, pcplus4d, pcplus4e, 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;

    wire [31:0] PCIfTaken,pcd,pce;
    wire branchF, branchE, branchIsCorrectE, predictF, predictD;
    wire [15:0] immF;
    assign immF = instrf[15:0];
    assign branchF = (instrf[31:26] == 4) | (instrf[31:26] == 5);
    wire pcsrce;

    wire [31:0] pcSourcePredicted;
    assign pcSourcePredicted = predictF ? PCIfTaken : pcplus4f;


    PCBranchBufForStep2 branchPredictor(clk, rfreset, branchF, branchE, branchIsCorrectE,pcf,
pce,immF, predictF,PCIfTaken);
    //fetch
    mux3 #(32) pcmux(pcSourcePredicted, pcbranche,jumpadr, {pcsrcd[1], ~branchIsCorrectE},
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, ((~branchIsCorrectE) | pcsrcd[1]|rfreset), ~(stalld |
pig2),instrf,pcplus4f, predictF, pcf,instrd,pcplus4d, predictD,pcd); // 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 pcTakeD = {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+67+32) d2e(clk,rfreset, (flushe| ~branchIsCorrectE),~pig2,
{instrd,pcd,pcsrcd[0],pcplus4d, pcTakeD, predictD, branchd,
mul_signd,startmuld,outseld,regwrited, memtoregd,
memwrited,alucontrold,alusrcd,regdstd,rd1,rd2,instrd[25:21],instrd[20:16],instrd[15:11],immext
,memenabled},
                        {instre,pce, pcsrce,pcplus4e,pcTakeE,
predictE,branchE,mul_signe,startmule,outsele,regwritee, memtorege,
memwritee,alucontrole,alusrce,regdste,rd1e,rd2e,rse,rte,rde,immexte,memenableE});
    assign pcbranche = (pcsrce) ? pcTakeE : pcplus4e;
    assign branchIsCorrectE = (pcsrce == predictE);

    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
    wire mem_ready; //mei yong de
    /*memory mem (clk, reset, memenablem, memwritem,
        A,
        writedatam,
        rdm,
        mem_ready);*/
    //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,writeregw,regwritew,stallf,stalld,forwardad,forwardbd,flushe,forwardae,forwardbe);



endmodule
```

Code for step 3:

DynamicPredictBuffer.v:

```
module PCBranchBufForStep3(input clk, reset, enableF, branchE, branchIsCorrectE,
               input [31:0] PC, PCE,
               input [15:0] immF,
               input [1:0] branchHistoryE,
               input pcsrce0,
               output predictF,
               output [31:0] PCIfTaken,
               output [1:0] branchHistoryOut);

    reg valid[3:0][127:0];
    reg [31:0] PCRam[127:0];
    reg [31:0] PCTake[127:0];



    reg [1:0] state[3:0][127:0];
    reg [1:0] branchHistoryF;
    assign branchHistoryOut = branchHistoryF;

    integer i;

    wire hit;
    assign hit = valid[branchHistoryF][PC[8:2]] & (PC == PCRam[PC[8:2]]);

    assign predictF = state[branchHistoryF][PC[8:2]][1] & hit;
    //assign PCIfTaken = valid[PC[8:2]] ? 0 : PCTake[PC[8:2]];
    assign PCIfTaken = PCTake[PC[8:2]];
    always @(posedge clk, reset) begin
        if(reset) begin
            for(i = 0;i <128 ;i = i + 1) begin
                valid[0][i] <= 0;
                valid[1][i] <= 0;
                valid[2][i] <= 0;
                valid[3][i] <= 0;

                state[0][i] <= 0;
                state[1][i] <= 0;
                state[2][i] <= 0;
                state[3][i] <= 0;
                PCRam[i] <= 0;
                PCTake[i] <= 0;
                branchHistoryF <= 0;
            end
        end
        if(enableF) begin
            if(~hit) begin // if miss
                PCRam[PC[8:2]] <= PC;
                PCTake[PC[8:2]] <= PC + 4 + {{14{immF[15]}},immF,2'b00};
                valid[branchHistoryF][PC[8:2]] <= 1;
                state[branchHistoryF][PC[8:2]] <= 0;
            end

        end
        if(branchE) begin // update state
            branchHistoryF <= {pcsrce0, branchHistoryF[1]};
            case({state[branchHistoryE][PCE[8:2]], branchIsCorrectE})
                3'b000: state[branchHistoryE][PCE[8:2]] <= 2'b01; // Strong NotTake &
incorrect
                3'b001: state[branchHistoryE][PCE[8:2]] <= 2'b00; // Strong NotTake & correct
                3'b010: state[branchHistoryE][PCE[8:2]] <= 2'b10; // Weak NotTake & incorrect
                3'b011: state[branchHistoryE][PCE[8:2]] <= 2'b00; // Weak NotTake & correct
```

```
                3'b100: state[branchHistoryE][PCE[8:2]] <= 2'b01; // Weak Take & incorrect
                3'b101: state[branchHistoryE][PCE[8:2]] <= 2'b11; // Weak Take & correct
                3'b110: state[branchHistoryE][PCE[8:2]] <= 2'b10; // Strong Take & incrrect
                3'b111: state[branchHistoryE][PCE[8:2]] <= 2'b11; // Strong take & correct
                default: state[branchHistoryE][PCE[8:2]] <= 2'b00;
            endcase

        end
    end


endmodule



/*
module DynamicPredicBuffer();

    reg [127:0] bht [65:0];
    wire [6:0] index;




endmodule

module takenFSM (input [1:0] state, input taken, clk, reset, output [1:0] nextstate, output
predictTaken);

reg [3:0] out;
assign {nextstate,predictTaken} = out;

//state 00 is strong not taken, state 01 is weak not taken, state 10 is weak taken, state 11
is strong taken.

    always @* begin
        case (state)
            2'b00:
                out <= taken ? 3'b010 : 3'b000;
            2'b01:
                out <= taken ? 3'b101 : 3'b000;
            2'b10:
                out <= taken ? 3'b111 : 3'b010;
            2'b11:
                out <= taken ? 3'b111 : 3'b101;
        endcase
     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, pcTakeD, pcTakeE, pcbranche, jumpadr, pcnext, pcf, instrf;
    //dec
    wire [31:0] instrd, pcplus4d, pcplus4e, 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;

wire [31:0] PCIfTaken,pcd,pce;
wire branchF, branchE, branchIsCorrectE, predictF, predictD;
wire [15:0] immF;
assign immF = instrf[15:0];
assign branchF = (instrf[31:26] == 4) | (instrf[31:26] == 5);
wire pcsrce;

wire [31:0] pcSourcePredicted;
```

```
    assign pcSourcePredicted = predictF ? PCIfTaken : pcplus4f;

    wire [1:0] branchHistoryE;
    wire [1:0] branchHistoryF;
    wire [1:0] branchHistoryD;
    PCBranchBufForStep3 branchPredictor(clk, rfreset, branchF, branchE, branchIsCorrectE,pcf,
pce,immF, branchHistoryE, pcsrce, predictF,PCIfTaken,branchHistoryF);
    //fetch
    mux3 #(32) pcmux(pcSourcePredicted, pcbranche,jumpadr, {pcsrcd[1], ~branchIsCorrectE},
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, ((~branchIsCorrectE) | pcsrcd[1]|rfreset), ~(stalld |
pig2),instrf,pcplus4f, predictF, pcf, branchHistoryF, instrd,pcplus4d,
predictD,pcd,branchHistoryD); // 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 pcTakeD = {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+67+32+2) d2e(clk,rfreset, (flushe| ~branchIsCorrectE),~pig2,
{instrd,branchHistoryD,pcd,pcsrcd[0],pcplus4d, pcTakeD, predictD, branchd,
mul_signd,startmuld,outseld,regwrited, memtoregd,
memwrited,alucontrold,alusrcd,regdstd,rd1,rd2,instrd[25:21],instrd[20:16],instrd[15:11],immext
,memenabled},
                        {instre,branchHistoryE,pce, pcsrce,pcplus4e,pcTakeE,
predictE,branchE,mul_signe,startmule,outsele,regwritee, memtorege,
memwritee,alucontrole,alusrce,regdste,rd1e,rd2e,rse,rte,rde,immexte,memenableE});
    assign pcbranche = (pcsrce) ? pcTakeE : pcplus4e;
    assign branchIsCorrectE = (pcsrce == predictE);

    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
    wire mem_ready; //mei yong de
    /*memory mem (clk, reset, memenablem, memwritem,
        A,
        writedatam,
        rdm,
        mem_ready);*/
    //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,writeregw,regwritew,stallf,stalld,forwardad,forwardbd,flushe,forwardae,forwardbe);



endmodule
```