VLSI CİRCUİT DESİGN II
HOMEWORK 8 REPORT

Name: Selman Yusufoğlu

Number:040160037

1-) 5 Stage Pipeline Without Any Hazard Handler

a) Implementation

First, I drew the design scheme to include all the blocks. By specifying the inputs and outputs of each block, I determined where to connect them. Then I divided the design into 4 parts by placing the 4 pipeline blocks between the other process blocks and determined the input and output signals and data for each stage. Finally, I put all these together and determined the final design.

Since some blocks I used in single cycle operation do not fit the pipeline structure, I updated them. For example, in the previous design, I generated the necessary signals for the muxes to be used in the program counter in the branch controller, gave them to the program counter, and made the addition operations here. Since it would be difficult to design the pipeline in this way, I gave the new address from the branch controller output by doing all the addition and mux operations related to the branch and jump. I also generated a PC_Src signal that indicates whether the program counter will take a new value or continue with the normal 4 increments. I sent this signal and the new address information generated by the branch controller at the EX stage to the IF step through the EX_MEM register block. Here, I also connected the new pc value from the branch controllers with the increased pc value of 4 to the pc input with a mux and PC_src signal. I just did the pc update process in the program counter.

After implementing the new design with verilog, I checked whether the signals were transferred correctly. The first error I encountered was that, since the select signal of the mux at the pc input was "X" in the initial state, new pc value could not be assigned and the result could be produced by going from pc 0 to X. To solve this, I connected the rst signal to the 4 blocks containing the registers in between so that all signals are 0 when the program is started and rst is set to 1. Then the program worked properly and the pc values were produced and the signals were transferred until the last stage. I fixed the bad connections by tracking the signals and the verilog implementation is finished.

Verilog Codes for New Branch Controller, New Program Counter, IF_ID, ID_EX, EX_MEM, WB_MEM, and Top Module Pipelined CPU:

```
module BranchController(
    input [31:0] imm, imm_rs1, prev_pc,
    input [2:0] BMC,
    input PL, JL, JLR, Z,
    output [31:0] jump_pc,
    output PC_Src
    );

    reg BE;    // branch enable
    always @(*)
    begin
        if(Z)
        begin
            if(BMC == 3'b000 || BMC == 3'b101 || BMC == 3'b111 )
                BE = 1;
            else
                BE = 0;
        end

        else
        begin
            if(BMC == 3'b001 || BMC == 3'b100 || BMC == 3'b110 )
                BE = 1;
            else
                BE = 0;
        end
    end

    assign PC_Src = ~PL&(BE | JL | JLR);

    wire MIJ;
    assign MIJ = ~BE&~JL&JLR;

    wire [31:0] sum_imm;

    cla_32bit_adder adder_imm(
        .a(imm), .b(prev_pc),
        .cin(0), .cout(), .sum(sum_imm)
    );

     mux_2to1 mux_ij(
        .in0(sum_imm), .in1(imm_rs1),
        .s(MIJ),
        .out(jump_pc)
    );

endmodule

module ProgramCounter(
    input clk, rst,
    input [31:0] prev_pc,
    output reg [31:0] next_pc
    );

    always @(posedge clk)
    begin
        if (rst)
            next_pc<=32'b0;
        else
            next_pc <= prev_pc;
    end

endmodule

module IF_ID(
    input clk, rst, IF_ID_Write,
    input [31:0] IF_Inst, IF_Next_Pc, IF_Pc4,
    output reg [31:0] ID_Inst, ID_Next_Pc, ID_Pc4
    );


    always @(posedge clk)
    begin
```

```
            EX_FS<=0;
            EX_BMC<=0;
        end

        else
        begin
            EX_Next_Pc <= ID_Next_Pc;
            EX_Imm <= ID_Imm;
            EX_Pc4 <= ID_Pc4;

            EX_DA <= ID_DA;
            EX_AA <= ID_AA;
            EX_BA <= ID_BA;

            EX_RA<=ID_RA;
            EX_RB<=ID_RB;

            EX_MA<=ID_MA;
            EX_MB<=ID_MB;
            EX_MD<=ID_MD;

            EX_RW<=ID_RW;
            EX_MW<=ID_MW;
            EX_MR<=ID_MR;

            EX_PL<=ID_PL;
            EX_JL<=ID_JL;
            EX_JLR<=ID_JLR;

            EX_FS<=ID_FS;
            EX_BMC<=ID_BMC;
        end


    end

endmodule

module EX_MEM(
    input clk, rst,
    input [31:0] EX_Jump_Pc, //branch
    input [31:0] EX_Pc4,
    input [31:0] EX_Fout, EX_Data_Out,
    input [4:0] EX_DA, EX_BA,
    input EX_MD, EX_PC_Src,
    input EX_RW, EX_MW, EX_MR,
    input EX_JL, EX_JLR,
    input [2:0] EX_BMC,

    output reg [31:0] MEM_Jump_Pc, //branch
    output reg [31:0] MEM_Pc4,
    output reg [31:0] MEM_Fout, MEM_Data_Out,
    output reg [4:0] MEM_DA, MEM_BA,
    output reg  MEM_MD, MEM_PC_Src,
    output reg  MEM_RW, MEM_MW, MEM_MR,
    output reg  MEM_JL, MEM_JLR,
    output reg  [2:0] MEM_BMC
    );


    always @(posedge clk)
    begin
        if(rst)
        begin
            MEM_Jump_Pc <= 0;
            MEM_Pc4 <= 0;
            MEM_Fout <= 0;
            MEM_PC_Src <= 0;
            MEM_Data_Out <= 0;

            MEM_DA <= 0;
            MEM_BA <= 0;
            MEM_MD <= 0;
```

```
        if(rst)
        begin
                ID_Inst <= 32'b00000000000000000000000000010011;
                ID_Next_Pc <= 0;
                ID_Pc4 <= 0;
        end

        else
        begin
            if(IF_ID_Write == 1)
            begin
                ID_Inst <= IF_Inst;
                ID_Next_Pc <= IF_Next_Pc;
                ID_Pc4 <= IF_Pc4;
            end
        end
    end


endmodule


   module ID_EX(
    input clk, rst,

    input [31:0] ID_Pc4, ID_Next_Pc, ID_Imm,
    input [4:0] ID_DA, ID_AA, ID_BA,
    input ID_MA, ID_MB, ID_MD,
    input ID_RW, ID_MW, ID_MR,
    input ID_PL, ID_JL, ID_JLR,
    input [3:0] ID_FS,
    input [2:0] ID_BMC,
    input [31:0] ID_RA, ID_RB,

    output reg [31:0] EX_Pc4, EX_Next_Pc, EX_Imm,
    output reg [4:0] EX_DA, EX_AA, EX_BA,
    output reg EX_MA, EX_MB, EX_MD,
    output reg EX_RW, EX_MW, EX_MR,
    output reg EX_PL, EX_JL, EX_JLR,
    output reg [3:0] EX_FS,
    output reg [2:0] EX_BMC,
    output reg [31:0] EX_RA, EX_RB
    );

    always @(posedge clk)
    begin

        if(rst)
        begin

            EX_Next_Pc <= 0;
            EX_Imm <= 0;
            EX_Pc4 <= 0;

            EX_DA <= 0;
            EX_AA <= 0;
            EX_BA <= 0;

            EX_RA<=0;
            EX_RB<=0;

            EX_MA<=0;
            EX_MB<=0;
            EX_MD<=0;

            EX_RW<=0;
            EX_MW<=0;
            EX_MR<=0;

            EX_PL<=1;
            EX_JL<=0;
            EX_JLR<=0;

```

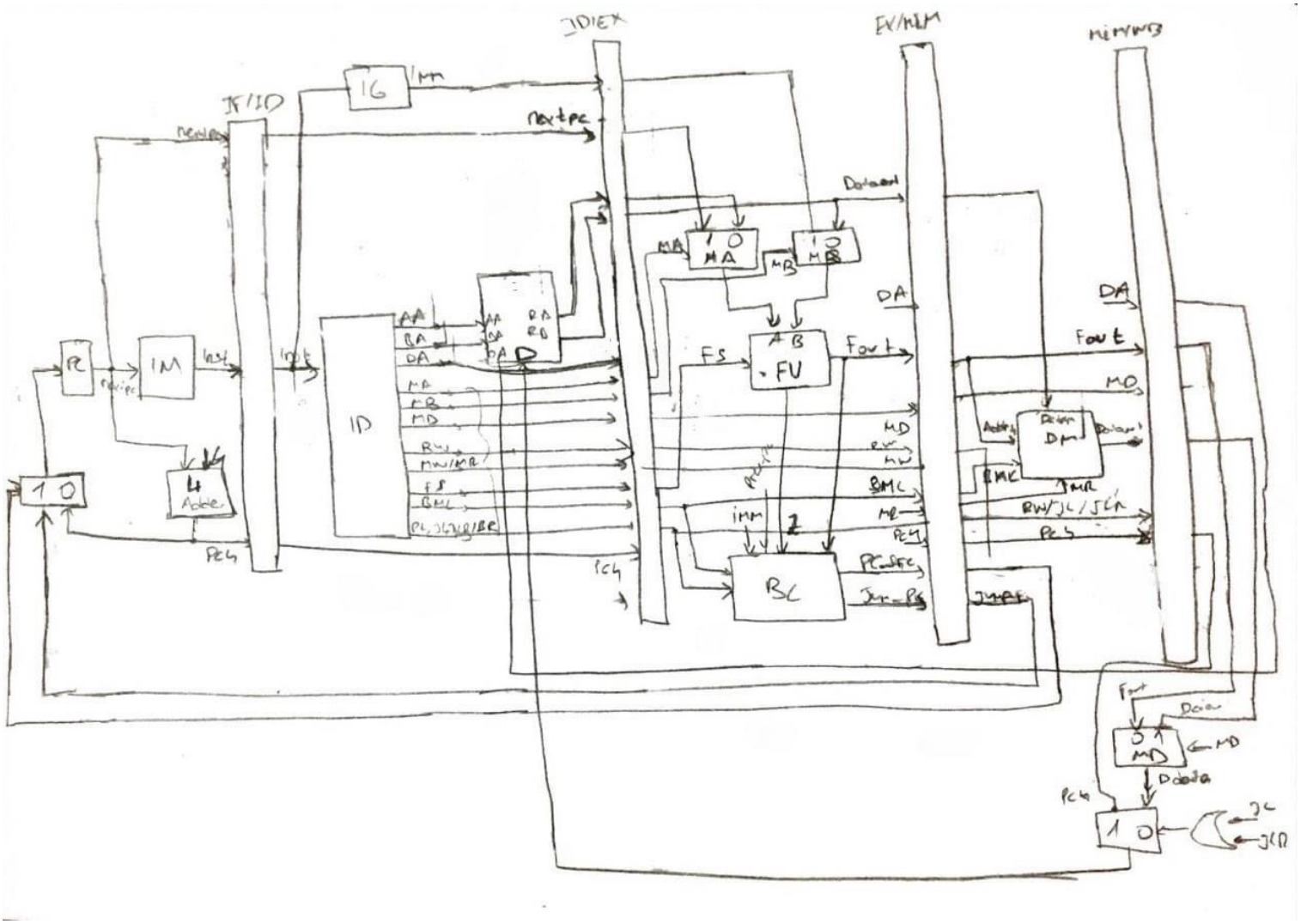
```
220
221             MEM_RW <= 0;
222             MEM_MW <= 0;
223             MEM_MR <= 0;
224
225             MEM_JL <= 0;
226             MEM_JLR <= 0;
227             MEM_BMC <= 0;
228         end
229
230         else
231         begin
232             MEM_Jump_Pc <= EX_Jump_Pc;
233             MEM_Pc4 <= EX_Pc4;
234             MEM_Fout <= EX_Fout;
235             MEM_PC_Src <= EX_PC_Src;
236             MEM_Data_Out <= EX_Data_Out;
237
238             MEM_DA <= EX_DA;
239             MEM_BA <= EX_BA;
240             MEM_MD <= EX_MD;
241
242             MEM_RW <= EX_RW;
243             MEM_MW <= EX_MW;
244             MEM_MR <= EX_MR;
245
246             MEM_JL <= EX_JL;
247             MEM_JLR <= EX_JLR;
248             MEM_BMC <= EX_BMC;
249         end
250
251     end
252 endmodule
253
254 module MEM_WB(
255     input clk,
256     input [31:0] MEM_Fout,
257     input [31:0] MEM_Data_In,
258     input [31:0] MEM_Pc4,
259     input [4:0] MEM_DA,
260     input MEM_RW, MEM_MW, MEM_MD,
261     input MEM_JL, MEM_JLR,
262
263     output reg [31:0] WB_Fout,
264     output reg [31:0] WB_Data_In,
265     output reg [31:0] WB_Pc4,
266     output reg [4:0] WB_DA,
267     output reg WB_RW, WB_MW, WB_MD,
268     output reg WB_JL, WB_JLR
269     );
270
271     always @(posedge clk)
272     begin
273         WB_Fout <= MEM_Fout;
274         WB_Data_In <= MEM_Data_In;
275         WB_Pc4 <= MEM_Pc4;
276         WB_DA <= MEM_DA;
277
278         WB_RW <= MEM_RW;
279         WB_MW <= MEM_MW;
280         WB_MD <= MEM_MD;
281
282         WB_JL <= MEM_JL;
283         WB_JLR <= MEM_JLR;
284     end
285
286 endmodule
287
288 module MCP_RISC_V(
289     input clk, rst, IM_we
290     );
291
292     wire [31:0] jump_pc, prev_pc, next_pc, pc4;
```

```
293       wire MEM_PC_Src;
294       assign pc4 = next_pc + 4;
295
296       mux_pc muxpc(
297          .in0(pc4), .in1(jump_pc),
298          .s(MEM_PC_Src),
299          .out(prev_pc)
300       );
301
302       ProgramCounter pc0(
303           .clk(clk), .rst(rst),
304           .prev_pc(prev_pc),
305           .next_pc(next_pc)
306       );
307
308       wire [31:0] IF_Inst;
309       InsMem IM(
310           .clk(clk), .rst(rst), .we(IM_we),
311           .addr(next_pc),
312           .data(IM_data),
313           .out(IF_Inst)
314       );
315
316       wire [31:0] ID_Inst;
317       wire [31:0] ID_Next_Pc;
318       wire [31:0] ID_Pc4;
319
320       IF_ID if_id0(
321           .clk(clk),
322           .IF_Inst(IF_Inst), .IF_Next_Pc(next_pc), .IF_Pc4(pc4),
323           .ID_Inst(ID_Inst), .ID_Next_Pc(ID_Next_Pc), .ID_Pc4(ID_Pc4)
324       );
325
326       wire [31:0] ID_Imm;
327       ImmGen IG(
328           .IR(ID_Inst),
329           .Imm(ID_Imm)
330       );
331
332       wire MA, MB, MD, RW, MW, PL, JL, JLR;
333       wire [3:0] FS;
334       wire [2:0] BMC;
335       wire [4:0] AA, BA, DA;
336
337       InsDec ID(
338           .IR(ID_Inst),
339           .MA(MA), .MB(MB), .MD(MD),
340           .RW(RW), .MW(MW),
341           .PL(PL), .JL(JL), .JLR(JLR),
342           .FS(FS), .BMC(BMC),
343           .AA(AA), .BA(BA), .DA(DA)
344       );
345
346
347       wire [31:0] Adata, Bdata, Ddata;
348       wire [31:0] BusA, BusB, BusD;
349       wire WB_RB;
350       wire [4:0] MEM_DA;
351       wire [4:0] WB_DA;
352
353       RegisterFile rf0(
354           .clk(clk), .RW(WB_RW),
355           .AA(AA), .BA(BA), .DA(WB_DA),
356           .D(BusD), .A(Adata), .B(Bdata)
357       );
358
359       wire [31:0] EX_Pc4, EX_Next_Pc, EX_Imm;
360       wire [4:0] EX_DA;
361       wire EX_MA, EX_MB, EX_MD;
362       wire EX_RW, EX_MW;
363       wire EX_PL, EX_JL, EX_JLR;
364       wire [3:0] EX_FS;
365       wire [2:0] EX_BMC;
```

```
366       wire [31:0] EX_RA, EX_RB;
367
368       ID_EX id_ex0(
369           .clk(clk),
370
371           .ID_Next_Pc(ID_Next_Pc),
372           .ID_Imm(ID_Imm),
373           .ID_DA(DA), .ID_Pc4(ID_Pc4),
374           .ID_MA(MA), .ID_MB(MB), .ID_MD(MD),
375           .ID_RW(RW), .ID_MW(MW),
376           .ID_PL(PL), .ID_JL(JL), .ID_JLR(JLR),
377           .ID_FS(FS), .ID_BMC(BMC),
378           .ID_RA(Adata), .ID_RB(Bdata),
379
380           .EX_Next_Pc(EX_Next_Pc),
381           .EX_Imm(EX_Imm),
382           .EX_DA(EX_DA), .EX_Pc4(EX_Pc4),
383           .EX_MA(EX_MA), .EX_MB(EX_MB), .EX_MD(EX_MD),
384           .EX_RW(EX_RW), .EX_MW(EX_MW),
385           .EX_PL(EX_PL), .EX_JL(EX_JL), .EX_JLR(EX_JLR),
386           .EX_FS(EX_FS), .EX_BMC(EX_BMC),
387           .EX_RA(EX_RA), .EX_RB(EX_RB)
388       );
389
390
391       mux_2to1 mux_ma(
392           .in0(EX_RA), .in1(EX_Next_Pc),
393           .s(EX_MA),
394           .out(BusA)
395       );
396
397       mux_2to1 mux_mb(
398           .in0(EX_RB), .in1(EX_Imm),
399           .s(EX_MB),
400           .out(BusB)
401       );
402
403       wire [31:0] Fout;
404       wire Z, C, V;
405
406       FunctionUnit fu0(
407           .A(BusA), .B(BusB),
408           .FS(EX_FS),
409           .Result(Fout),
410           .Z(Z), .C(C), .V(V)
411       );
412
413       wire [31:0] EX_Jump_Pc;
414       wire PC_Src;
415       BranchController branch0(
416           .BMC(EX_BMC), .Z(Z),
417           .PL(EX_PL), .JL(EX_JL), .JLR(EX_JLR),
418           .imm(EX_Imm), .imm_rs1(Fout),
419           .prev_pc(EX_Next_Pc),
420           .jump_pc(EX_Jump_Pc),
421           .PC_Src(PC_Src)
422       );
423
424       wire [31:0] MEM_Pc4;
425       wire [31:0] MEM_Fout, MEM_Data_Out;
426       wire [2:0] MEM_BMC;
427       wire MEM_MD;
428       wire MEM_RW, MEM_MW;
429       wire MEM_JL, MEM_JLR;
430
431
432       EX_MEM ex_mem0(
433           .clk(clk),
434
435           .EX_Jump_Pc(EX_Jump_Pc),  .EX_Pc4(EX_Pc4),
436           .EX_Fout(Fout), .EX_Data_Out(EX_RA),
437           .EX_DA(EX_DA), .EX_PC_Src(PC_Src),
438           .EX_MD(EX_MD), .EX_BMC(EX_BMC),
```

```
439             .EX_RW(EX_RW), .EX_MW(EX_MW),
440             .EX_JL(EX_JL), .EX_JLR(EX_JLR),
441 
442             .MEM_Jump_Pc(jump_pc),  .MEM_Pc4(MEM_Pc4),
443             .MEM_Fout(MEM_Fout), .MEM_Data_Out(MEM_Data_Out),
444             .MEM_DA(MEM_DA), .MEM_PC_Src(MEM_PC_Src),
445             .MEM_MD(MEM_MD), .MEM_BMC(MEM_BMC),
446             .MEM_RW(MEM_RW), .MEM_MW(MEM_MW),
447             .MEM_JL(MEM_JL), .MEM_JLR(MEM_JLR)
448 
449         );
450 
451         wire MEM_Data_In;
452         DataMemory DM0(
453             .clk(clk),
454             .MW(MEM_MW), .BMC(MEM_BMC),
455             .Address(MEM_Fout),
456             .Data_in(MEM_Data_Out),
457             .Data_out(MEM_Data_In)
458         );
459 
460         wire [31:0] WB_Fout;
461         wire [31:0] WB_Data_In;
462         wire [31:0] WB_Pc4;
463         wire  WB_MD;
464         wire  WB_JL, WB_JLR;
465 
466         MEM_WB mem_wb0(
467             .clk(clk),
468             .MEM_Fout(MEM_Fout),
469             .MEM_Data_In(MEM_Data_In),
470             .MEM_DA(MEM_DA),
471             .MEM_Pc4(MEM_Pc4),
472             .MEM_RW(MEM_RW), .MEM_MD(MEM_MD),
473             .MEM_JL(MEM_JL), .MEM_JLR(MEM_JLR),
474 
475             .WB_Fout(WB_Fout),
476             .WB_Data_In(WB_Data_In),
477             .WB_DA(WB_DA),
478             .WB_Pc4(WB_Pc4),
479             .WB_MD(WB_MD),.WB_RW(WB_RW),
480             .WB_JL(WB_JL), .WB_JLR(WB_JLR)
481         );
482 
483 
484         mux_2to1 mux_md(
485             .in0(WB_Fout), .in1(WB_Data_In),
486             .s(WB_MD),
487             .out(Ddata)
488         );
489 
490         wire MP;
491         assign MP = WB_JL | WB_JLR;
492         mux_2to1 mux_mp(
493             .in0(Ddata), .in1(WB_Pc4),
494             .s(MP),
495             .out(BusD)
496         );
497 
498     endmodule
499 
500 
501 
502 
503 
504 
```

b) Design Drawing



c) Assembly and Machine Code for The Algorithm

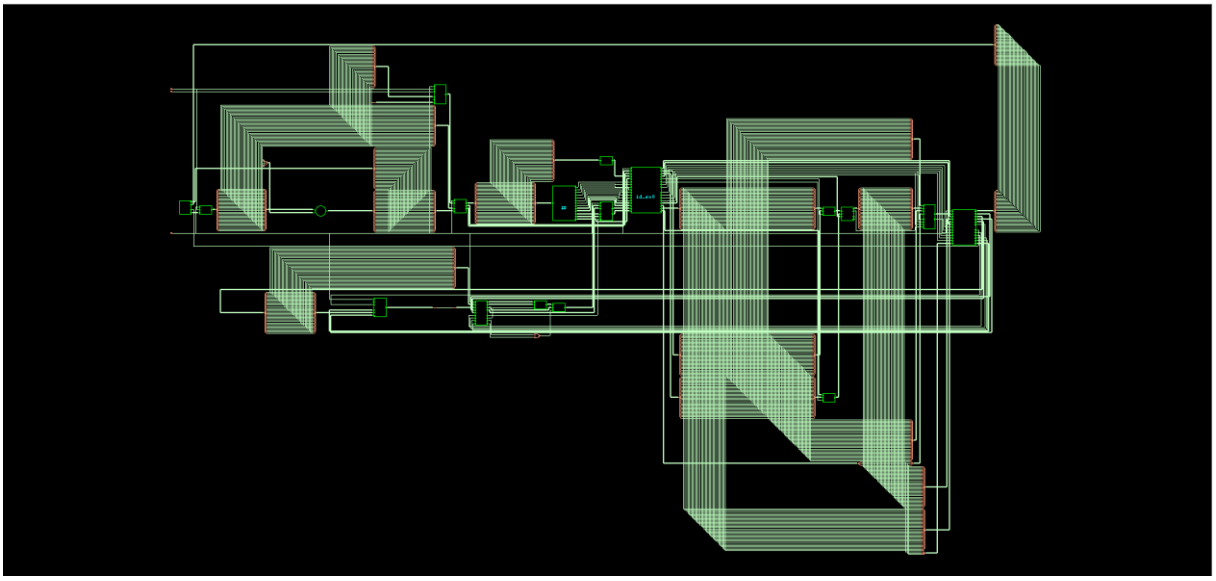
Assembly Code:

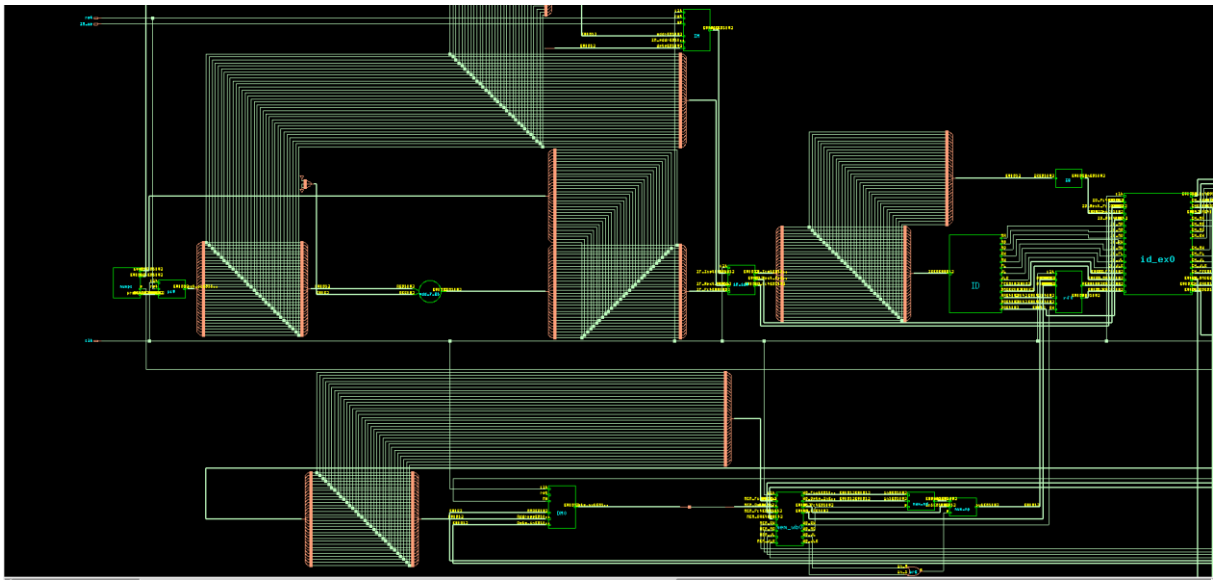
```
0       lui     r1, 0
4       addi    r2, r0, 5
8       addi    r3, r0, 4
12      addi    r4, r0, 3
16      addi    r5, r0, 9
20      addi    r6, r0, 1
24      beq     r5, r6, 0
28      sub     r5, r5, r6
32      sll     r1, r1, r6
36      addi    r7, r3, 0
40      srli    r7, r7, 31
44      sll     r3, r3, r6
48      blt     r1, r2, 8
52      sub     r1, r1, r2
56      bne     r7, r6, -28
60      add     r1, r1, r4
64      blt     r1, r2, -36
68      sub     r1, r1, r2
72      jal     r7, -48
```

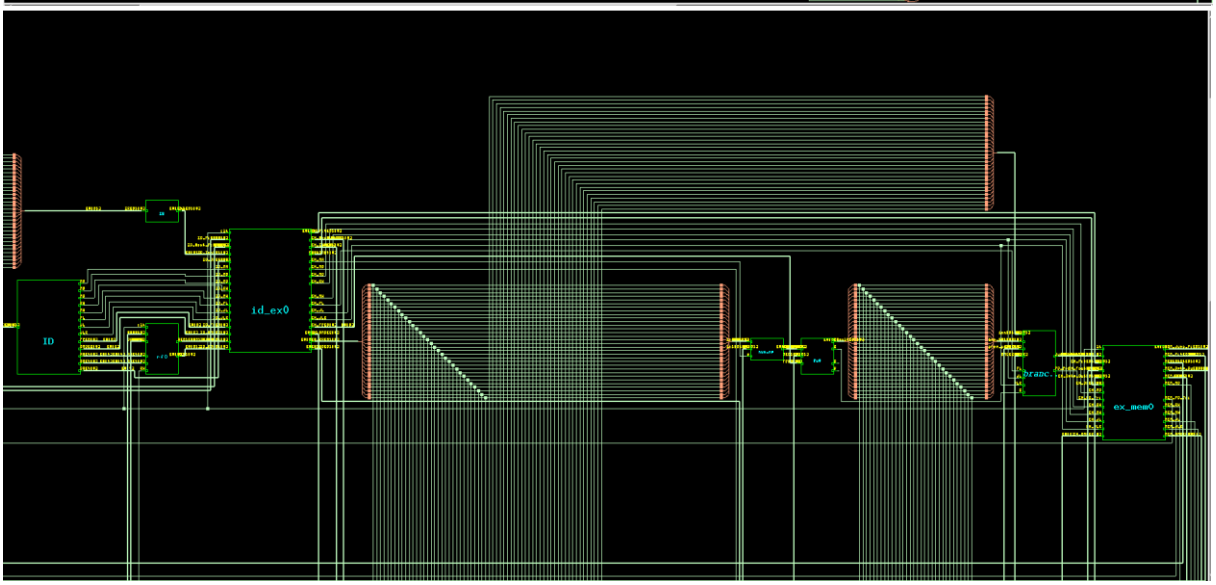
Machine Code:

```
00000000000000000000000010110111
00000000010100000000000100010011
00000000010000000000000110010011
00000000001100000000001000010011
00000010000100000000001010010011
00000000000100000000001100010011
00000000011000101000000001100011
01000000011000101000001010110011
00000000011000001001000010110011
00000000000000001100000111010011
00000001111100111101001110010011
00000000011000011001000110110011
00000000001000001100010001100011
01000000001000001000000010110011
11111110011000111001000011100011
00000000010000001000000010110011
11111100001000001100110011100011
01000000001000001000000010110011
11111101000111111111001111101111
```

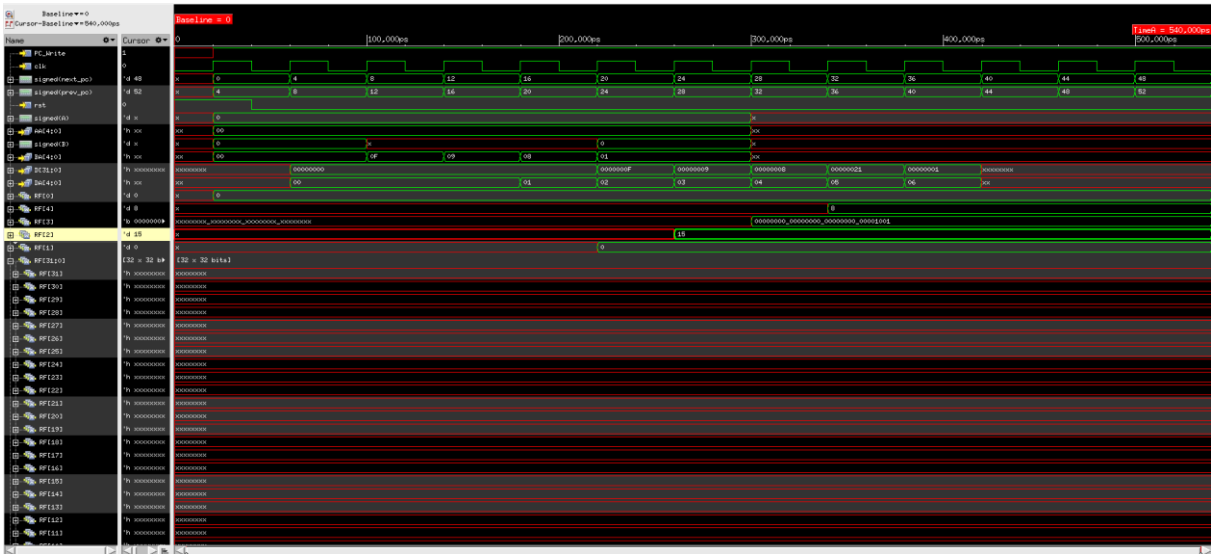
e) Post Elaboration Schematics







f) Behavioral Simulation



As it can be seen from simulation waveform, new A and B values are not loaded after first branch instruction and simulation fails.

g) To solve this problem we need to add nop operations to the places that may result in an hazard. There are three types of hazard; structural , data, and control hazard. First of all, we need to add three nop operations after every branch and jump instruction, as it is obligatory and may clear of some other possible hazard. After these nops, there are three hazards left which are data hazards. We place three nop operations in theses places too. After adding nops, offset values for branches and jumps, so we need update these finally.

Assembly Code After Nops:

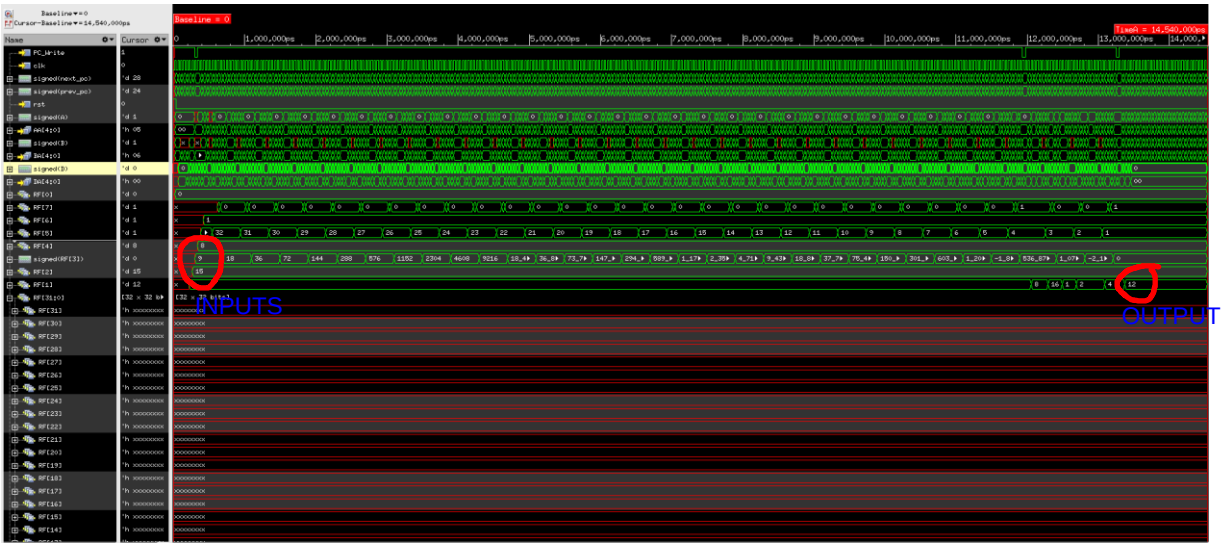
```
lui     r1, 0
addi    r2, r0, 5
addi    r3, r0, 4
addi    r4, r0, 3
addi    r5, r0, 9
addi    r6, r0, 1
nop nop nop
beq     r5, r6, 0
nop nop nop
sub     r5, r5, r6
sll     r1, r1, r6
addi    r7, r3, 0
nop nop nop
srli    r7, r7, 31
sll     r3, r3, r6
blt     r1, r2, 8
nop nop nop
sub     r1, r1, r2
bne     r7, r6, -28
nop nop nop
add     r1, r1, r4
nop nop nop
blt     r1, r2, -36
nop nop nop
sub     r1, r1, r2
jal     r7, -48
```

We fill instruction memory with new machine codes and repeat simulation with same A=9, B=8 , N=15 values.

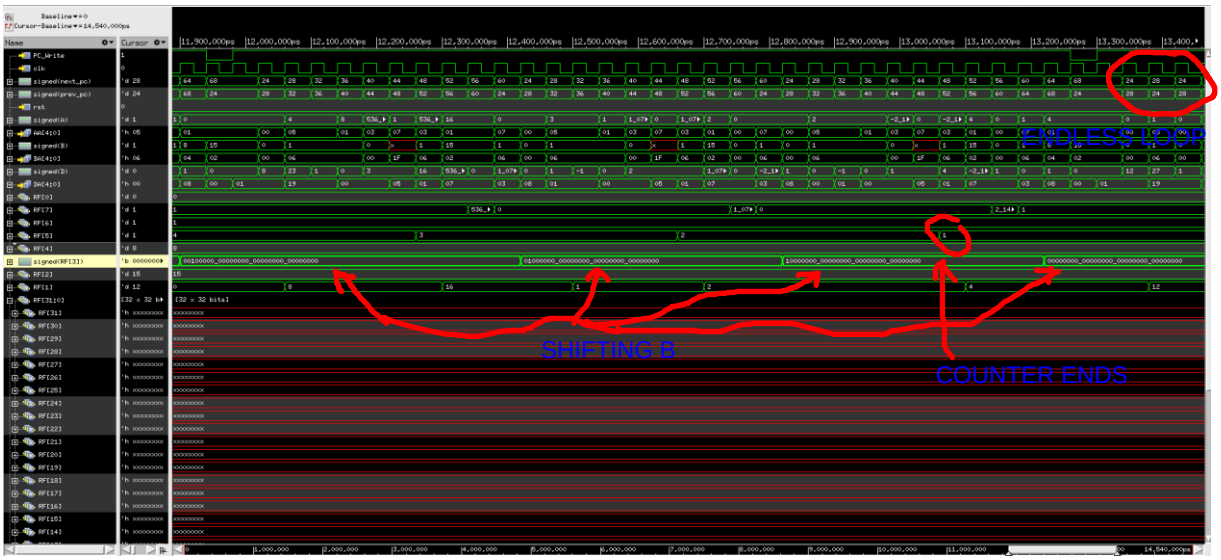
Testbench Code:

```
`timescale 1ns / 1ps

module MCP_RISC_V_tb();

    reg clk, rst, IM_we;

    MCP_RISC_V UUT(
        .clk(clk),
        .rst(rst),
        .IM_we(IM_we)
    );

    always #20 clk = ~clk;

    initial begin
        clk = 0; rst = 0;

        IM_we = 0; rst = 1;
        #40;
        rst = 0;

    end

endmodule
```

Simulation Result:



As it can be seen simulation waveform, when counter(r5) reached 4, first addition is made and other shift and subtraction operations follows until counter is reset. Result is written in r2 register, and it is 12.



If we zoom the area that opearations start and end, we can see that r3 register which holds B value is shifted to the left and when msb of B is 1 addition is made. It can also be seen that, when the counter is reset program enters endless loop as it is written so.

2) Structural Hazard

a) In the following sequences of code, there are three structural hazards which ocur in the last three lines of code. First one is that, while r3 register is being to subtract, the value is written into register. Therefore, no value is going to be read from r3 register, because initial value of r3 is X and it is now written when it is read. Same hazard occurs in following two instructions too.

```
addi    r2, r0, 5
addi    r3, r0, 4
addi    r4, r0, 3

add     r7, r2, r3
sub     r8, r2, r3
sll     r9, r2, r3
sw      r4, 0(r3)
```

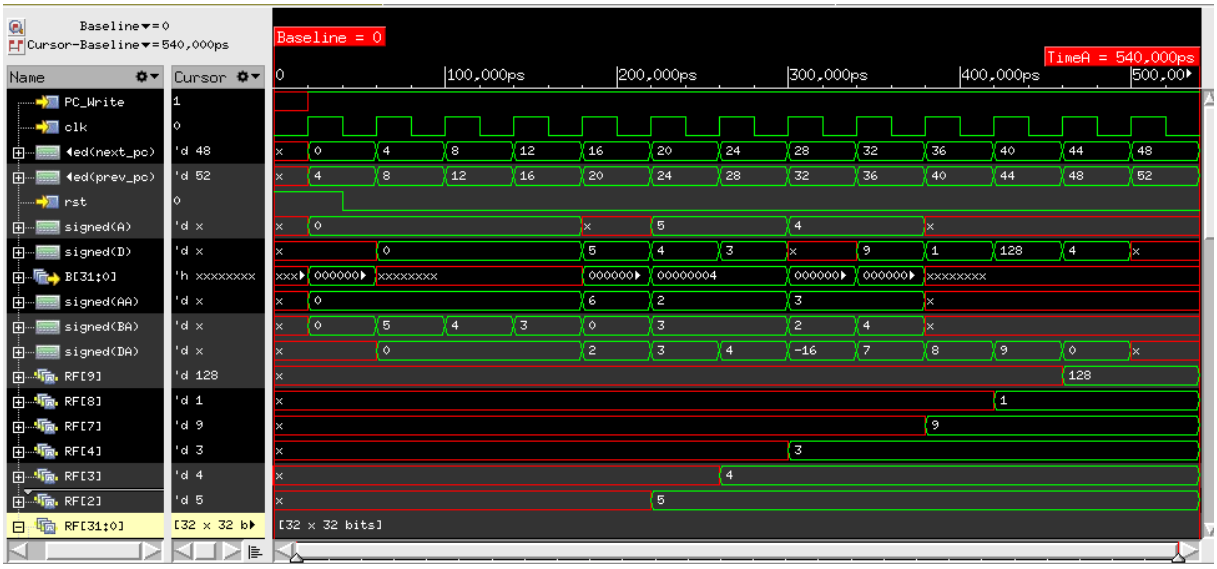
b) We can avoid this hazard by checking the status if new value is written to the registers with RW signal and checking if address and writing address equalites. If these situations occurs, we assign reading value as the input destinatin value came in.

c) **Verilog Code For Changes**

```
always @(*)
begin
    if(RW == 1 && AA == DA)
        A = D;
    else
        A = rA;

    if(RW == 1 && BA == DA)
        B = D;
    else
        B = rB;

end
```

rA and rB are the values that read from registers via muxes.

d) Simulation Result



As it can be seen from simulation waveform, r8 = r2 - r3 (5 - 4 = 1) and r9 = sll r2, r3(5*2^4 = 125) vaues are written correctly to the corresponding registers. If the hazard was not handled, r8 and r9 register values would be “X”.

3-) Data Hazards- Read-After-Write (RAW) Hazards:

a) Hazard Cases

1. Case

```
lui     r1, 0
addi    r2, r0, 5
addi    r3, r0, 4
addi    r4, r0, 3
addi    r5, r0, 8
addi    r6, r0, 1

sub     r1, r2, r3
add     r4, r1, r5
sub     r6, r3, r1,
add     r7, r1, r1
```

2. Case

```
addi    r2, r0, 5
addi    r3, r0, 4

sw      r2, 0x0(r3)
lw      r7, 0x0(r3)
```

3. Case

```
addi    r2, r0, 5
addi    r3, r0, 4

addi    r4, r0, 3
add     r4, r4, r2
add     r4, r4, r3
```

b) Forwarding Unit

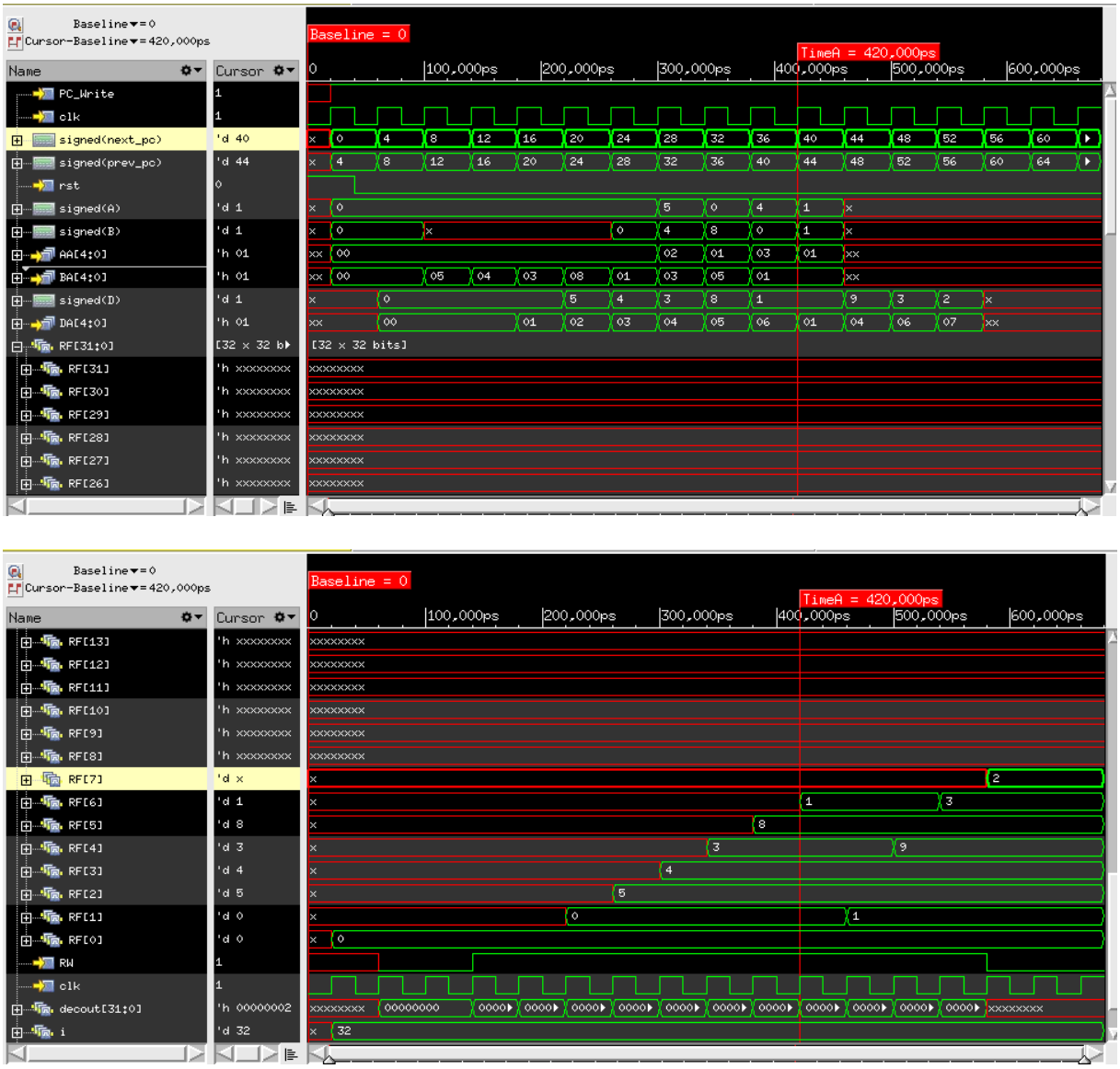
```
module ForwardUnit(
    input EX_MEM_RW, MEM_WB_RW,
    input [4:0] EX_MEM_DA, MEM_WB_DA, ID_EX_AA, ID_EX_BA,
    input [4:0] EX_MEM_BA,
    input [4:0] AA, BA,
    input EX_MEM_MW, BR,
    output reg ForwardC, ForwardD, ForwardM,
    output reg [1:0] ForwardA, ForwardB
    );


    always @(*) begin
        // RAW Forward
        if((EX_MEM_RW == 1) && (EX_MEM_DA != 0) && (EX_MEM_DA == ID_EX_AA))
            ForwardA = 2'b01;
        else if((MEM_WB_RW == 1) && (MEM_WB_DA != 0) && (MEM_WB_DA == ID_EX_AA) && (EX_MEM_DA != ID_EX_AA))
            ForwardA = 2'b10;
        else
            ForwardA = 2'b00;

        // RAW Forward
        if((EX_MEM_RW == 1) && (EX_MEM_DA != 0) && (EX_MEM_DA == ID_EX_BA))
            ForwardB = 2'b01;
        else if((MEM_WB_RW == 1) && (MEM_WB_DA != 0) && (MEM_WB_DA == ID_EX_BA) && (EX_MEM_DA != ID_EX_BA) )
            ForwardB = 2'b10;
        else
            ForwardB = 2'b00;

        // RAW-Mem Forward
        if( EX_MEM_MW == 1 && MEM_WB_RW == 1 && EX_MEM_BA == MEM_WB_DA)
            ForwardM = 1;
        else
            ForwardM = 0;

    end

endmodule
```

c) Changes in Top Module

```
wire MEM_RW, MEM_MW;
wire [4:0] MEM_BA;
wire [1:0] ForwardA, ForwardB;
wire ForwardM;

ForwardUnit forward(
    .EX_MEM_RW(MEM_RW), .MEM_WB_RW(WB_RW),
    .EX_MEM_DA(MEM_DA), .MEM_WB_DA(WB_DA),
    .ID_EX_AA(EX_AA), .ID_EX_BA(EX_BA),
    .ForwardA(ForwardA), .ForwardB(ForwardB),

    .EX_MEM_MW(MEM_MW),
    .EX_MEM_BA(MEM_BA),
    .ForwardM(ForwardM)

);

wire [31:0] FA, FB;
wire [31:0] WB_Fout;
//wire [31:0] MEM_Fout;

mux_4to1 mux_forwa(
    .in0(EX_RA), .in1(MEM_Fout),
    .in2(BusD), .in3(EX_RA),
    .sel(ForwardA), .out(FA)
);

mux_4to1 mux_forwb(
    .in0(EX_RB), .in1(MEM_Fout),
    .in2(BusD), .in3(EX_RB),
    .sel(ForwardB), .out(FB)
);

mux_2to1 mux_ma(
    .in0(FA), .in1(EX_Next_Pc),
    .s(EX_MA),
    .out(BusA)
);

mux_2to1 mux_mb(
    .in0(FB), .in1(EX_Imm),
    .s(EX_MB),
    .out(BusB)
);
```

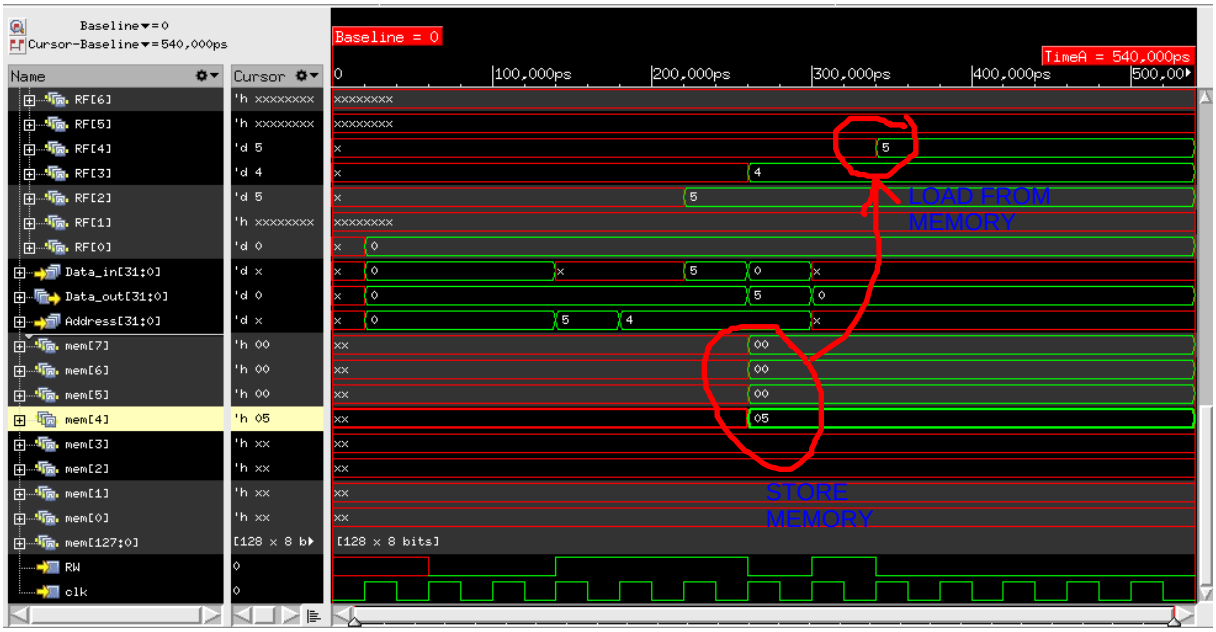
d) Behavioral Simulation

Case 1:



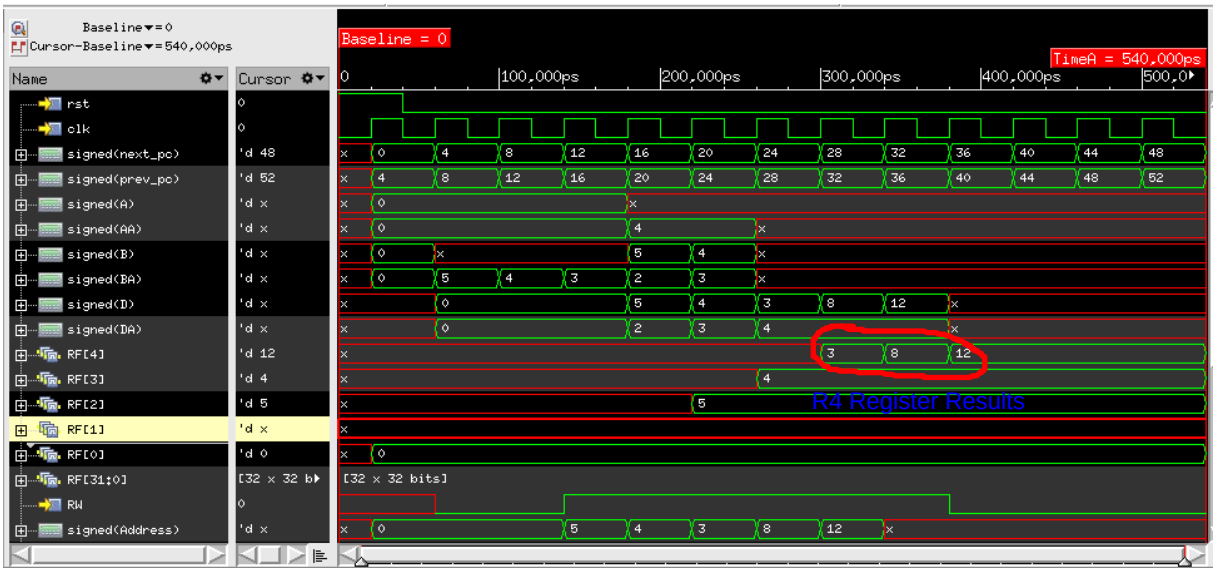
For the first case, hazard is handled fort he last three instructions. As it can be observed from waveform, for r4 = r1 + r5 (1+8), r6 = r3 – r1 (4-1) , and r7 = r1 + r1(1+1) register, correct values are calculated and written to corresponding registers without any hazard happening.

Case 2:



As it can be seen from waveform that, right after r2 and r3 register are loaded with 5 and 4 values, the value in r2 register is attempted to be stored in the address pointed by sum of r3 register with zero immediate value which is 4. It is seen at the bottom memory elements that, the value 5 is stored and right after that, same value is loaded from that location in memory is loaded to r4 register as expected. This shows that hazard has been handled.

Case 3:



As it can be seen from case 2 assembly code that, after r2, r3, and r4 register are loaded with 5, 4 and 3 values, two sequential instruction that attempts to load same r4 register with previous some operations on the same register r4. These operations are r4 = r4 + r2 (3+5) and

r4 = r4 + r3 (8 + 4). Therefore it is expected r4 register to take 3, 8 , and 12 values sequentianlly. It can be confirmed from r4 register in the waveform that it takes right values in the right order without any hazard.

4-) Data Hazards – Load-Use Hazards

a) Load-Use hazard occurs when a value is laoaded from memory to a registers and that register is used in the next three instructions. One case where all these three hazards occur is defined in the blow code.

```
lui     r1, 0
addi    r2, r0, 5
addi    r3, r0, 4
addi    r4, r0, 3
addi    r5, r0, 8
addi    r6, r0, 1

sw      r2, 0x0(r3)
lw      r7, 0x0(r3)

add     r4, r7, r5
sub     r8, r7, r6
add     r9, r3, r7

lw      r3, 0x0(r3)
slt     r10, r3, r3
slt     r11, r3, r5
```

b) We need stall if the loaded register value is needed one instruction later and we also need to ad forwarding to get the correct value.

For stall we need to an hazard detetion unit n ID stage, stall the cicuit if a possible hazard detected by chechking memory write and read-write address.

c) Stalling and Forwarding

Verilog Code for Hazard Detection and Stalling:

```
module HazardDetector(
    input ID_EX_MR, ID_EX_BR,
    input [4:0] IF_ID_AA, IF_ID_BA,
    input [4:0] ID_EX_DA,
    output reg PC_Write, IF_ID_Write, M_Stall
    );

    always @(*)
    begin
        if((ID_EX_MR == 1) && ((ID_EX_DA == IF_ID_AA) || (ID_EX_DA == IF_ID_BA)))
        begin
            M_Stall = 1;
            PC_Write = 0;
            IF_ID_Write = 0;
        end

        else
        begin
            M_Stall = 0;
            PC_Write = 1;
            IF_ID_Write = 1;
        end
    end
endmodule
```

If the hazard is detected M_Stall will replace current ID stage instruction with nop. This is done with the following changes in the ID stage:

```
wire [31:0] Control;
wire [31:0] COUTS;

assign Control = {16'b0, MA, MB, MD, RW, MW, MR, PL, JL, JLR, FS, BMC};

mux_2to1 mux_idex(
    .in0(Control), .in1(0),
    .s(M_Stall),
    .out(COUTS)
);

ID_EX id_ex0(
    .clk(clk), .rst(rst),

    .ID_Next_Pc(ID_Next_Pc),
    .ID_Imm(ID_Imm),
    .ID_DA(DA), .ID_AA(AA), .ID_BA(BA),
    .ID_Pc4(ID_Pc4),
    .ID_MA(COUTS[15]), .ID_MB(COUTS[14]), .ID_MD(COUTS[13]),
    .ID_RW(COUTS[12]), .ID_MW(COUTS[11]), .ID_MR(COUTS[10]),
    .ID_PL(COUTS[9]), .ID_JL(COUTS[8]), .ID_JLR(COUTS[7]),
    .ID_FS(COUTS[6:3]), .ID_BMC(COUTS[2:0]),
    .ID_RA(Adata), .ID_RB(Bdata),
```

Beside replacing current id stage instruction with nope, we also need to prevent pc increment and flush IF_ID block with PC_Write and IF_ID_Write signals. This is done with the following changes in IF_ID and Program Counter.

Program Counter Change:

```
always @(posedge clk)
begin
    if (rst)
        next_pc <= 32'b0;
    else
        if(PC_Write)
            next_pc <= prev_pc;

end
```

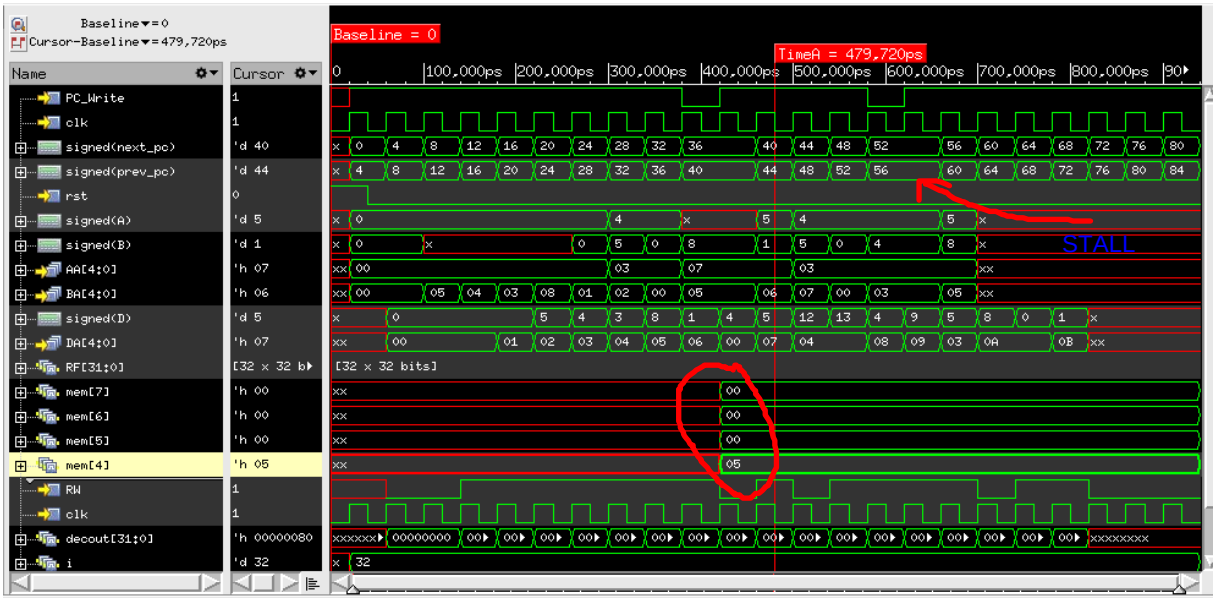
IF_ID Change:

```
always @(posedge clk)
begin
    if(rst)
    begin
            ID_Inst <= 32'b00000000000000000000000000010011;
            ID_Next_Pc <= 0;
            ID_Pc4 <= 0;
    end

    else
    begin
        if(IF_ID_Write == 1)
        begin
            ID_Inst <= IF_Inst;
            ID_Next_Pc <= IF_Next_Pc;
            ID_Pc4 <= IF_Pc4;
        end
    end
end
```

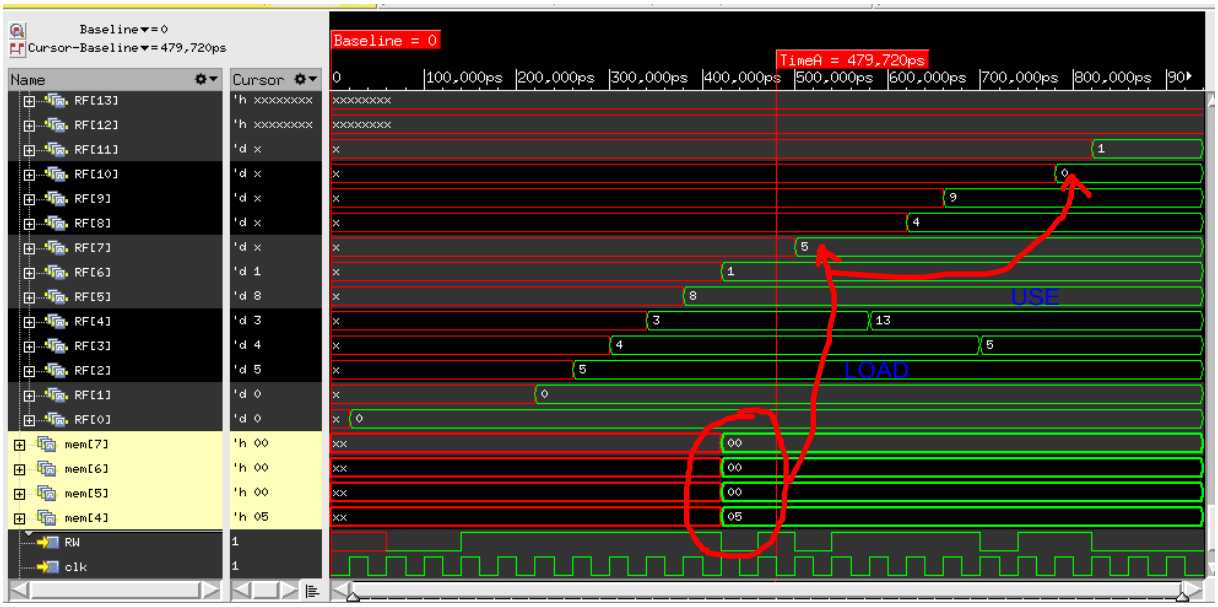
d) Changes in Top Module:

```
wire [31:0] new_Data_Out;
wire [31:0] WB_Data_In;

mux_2to1  mux_fm(
    .in0(MEM_Data_Out), .in1(WB_Data_In),
    .s(ForwardM), .out(new_Data_Out)
);

wire [31:0] MEM_Data_In;
DataMemory DM0(
    .clk(clk),
    .MW(MEM_MW), .MR(MEM_MR), .BMC(MEM_BMC),
    .Address(MEM_Fout),
    .Data_in(new_Data_Out),
    .Data_out(MEM_Data_In)
);
```

e) Simulation



As it can be seen from waveform that, when load-use hazard is detected, pc increment is stopped for one cycle for two different situations (next_pc:36,52)



As it can be seen from waveform, the value 5 which is loaded from memory and will be written to r7 register, is correctly added to 8(r5 register) and result 13 is written into r4 register thanks to stalling and forwarding. Forwarding works fine for the next instructions come by checking r8 = 5-1 and r9 = 5+4. I added one more situation with slt instructions coming after lw instruction that uses same register. It can be observed from r10 and r11 that is produces true results.

5,6-) Control Hazards, Jumps, and Branches:

a) I decide jumps in BranchController unit together with branches, so i will handle hazard handler for both jumps and branches with one solution. I will simulate them individuallly. For CPI = 2, we need to handle branches in ID stage, and this requires two adders, one for immediate value plus current pc and the other for read value from a register plus immediate valueWe also need to decide this inside BranchController, as we cannot use ALU for branch decision. We also need flush. I will use a mux at the input of IF_ID. Two inputs of mux is going to be instruction fetched from memory and nop instructin. Select signal of this mux is going to be PC_Src signal which is also used to determine PC.

b) New Branch Controller:

```
module BranchController(
    input [31:0] imm, rs1, rs2, prev_pc,
    input [2:0] BMC,
    input PL, JL, JLR, BR,
    output [31:0] jump_pc,
    output PC_Src
    );

    wire equal, signed_less, unsigned_less;

    assign equal = (rs1 == rs2);
    assign signed_less = ($signed(rs1) < $signed(rs2));
    assign unsigned_less = (rs1 < rs2);

    reg BE;    // branch enable
    always @(*)
    begin
        if(BR)
        begin
            case (BMC)
            3'b000:
            begin
                if( equal )
                    BE = 1;
                else
                    BE = 0;
            end

            3'b001:
            begin
                if( ~equal )
                    BE = 1;
                else
                    BE = 0;
            end

            3'b100:
            begin
                if( signed_less )
                    BE = 1;
                else
                    BE = 0;
            end


            3'b101:
            begin
                if( ~signed_less )
                    BE = 1;
                else
                    BE = 0;
            end


            3'b110:
            begin
                if( unsigned_less )
                    BE = 1;
                else
                    BE = 0;
            end


            3'b111:
            begin
                if( ~unsigned_less )
                    BE = 1;
                else
                    BE = 0;
            end

            default:
                    BE = 0;
```

```
74              endcase
75
76
77
78          end
79
80          else
81              BE = 0;
82      end
83
84      assign PC_Src = ~PL&(BE | JL | JLR);
85
86      wire MIJ;
87      assign MIJ = ~BE&~JL&JLR;
88
89      wire [31:0] sum_imm, imm_rs1;
90
91
92      cla_32bit_adder adder_imm(
93          .a(imm), .b(prev_pc),
94          .cin(0), .cout(), .sum(sum_imm)
95      );
96
97      cla_32bit_adder adder_jalr(
98          .a(imm), .b(rs1),
99          .cin(0), .cout(), .sum(imm_rs1)
100     );
101
102      mux_2to1 mux_ij(
103         .in0(sum_imm), .in1(imm_rs1),
104         .s(MIJ),
105         .out(jump_pc)
106     );
107
108
109 endmodule
110
```

c) Added Part to Forward Unit:

```
//Branch Forward
if( (BR == 1) && (EX_MEM_DA != 0) && (EX_MEM_DA == AA))
    ForwardC = 1;
else
    ForwardC = 0;

//Branch Forward
if( (BR == 1) && (EX_MEM_DA != 0) && (EX_MEM_DA == BA))
    ForwardD = 1;
else
    ForwardD = 0;
```

New conditions are added for Branch. These two signals are going to be used to choose between readed value from register file and same register address value coming from exe stage.

Added Part to Hazard Detection:

```
always @(*)
begin
    if((ID_EX_MR == 1 || ID_EX_BR == 1) && ((ID_EX_DA == IF_ID_AA) || (ID_EX_DA == IF_ID_BA)))
    begin
        M_Stall = 1;
        PC_Write = 0;
        IF_ID_Write = 0;
    end

    else
    begin
        M_Stall = 0;
        PC_Write = 1;
        IF_ID_Write = 1;
    end
end
```

BR signal is used same as we did in Load-Use hazard. If BR is 1 and EX stage destination address matches with ID stage read address, then stall the circuit.

Flushing:

```
wire [31:0] Check_Inst;
wire [31:0] IF_Inst;
InsMem IM(
    .clk(clk), .rst(rst), .we(IM_we),
    .addr(next_pc),
    .data(),
    .out(Check_Inst)
);

mux_2to1 muxj(
    .in0(Check_Inst), .in1(32'd19),
    .s(PC_Src),
    .out(IF_Inst)
);
```

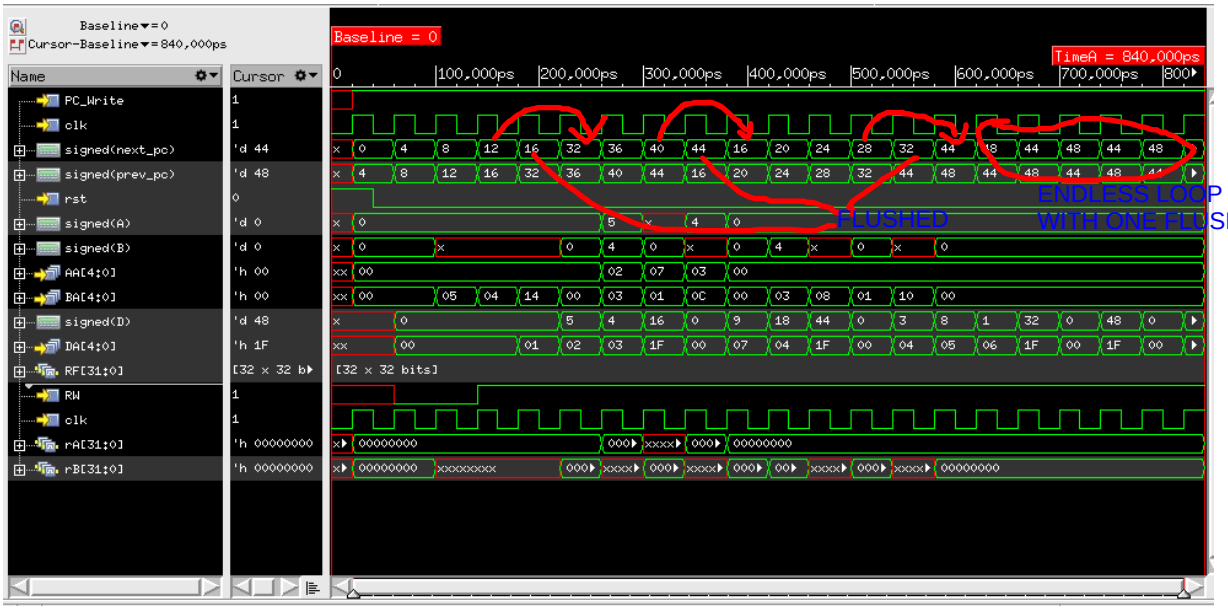
Select signal PC_Src coming from BranchController decides flushing. If it is 1 then we change current instruction with nop(32’d19).

d) Simuation for Jumps and Branches

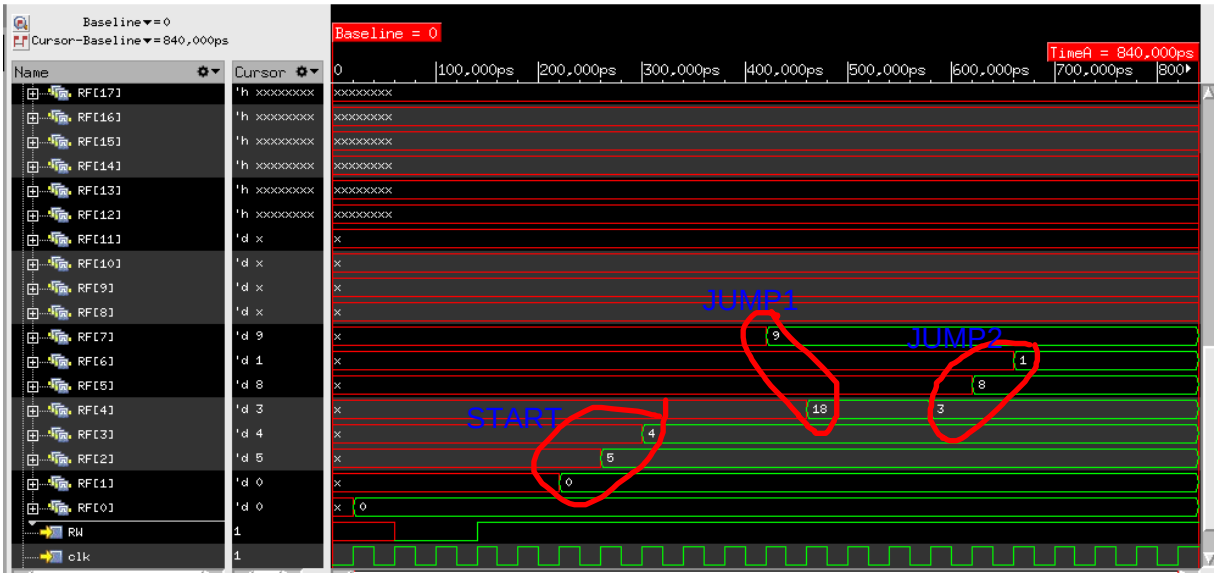
Simulation for Jump Instructions:

```
start:
0       lui     r1, 0
4       addi    r2, r0, 5
8       addi    r3, r0, 4
12      jal             r31, +20(jump1)

jump2:
16      addi    r4, r0, 3
20      addi    r5, r0, 8
24      addi    r6, r0, 1
28      jal     r31, 16 (stop))

jump1:
32      add     r7, r2, r3
36      slli    r4, r7, 1
40      jalr    r31, r5, -32 (jump2)(8+8)

stop:
44      jal     r31, 0  (stop)
```

Fort this set of instructions, program need to follow start(0x0)-jump1(0x32)-jump2(0x16)-stop(0x44) and it needs to jump in the next cycle.



As it can be observed from waveform, PC follows this sequence: 0-4-8-12-**16-32**-36-40-**44-16**-20-24-28-**32-44**-48-44-48 which shows that hazard is handled with CPI = 2. We can also observe from register file if the instruction under labels are correctly executed.



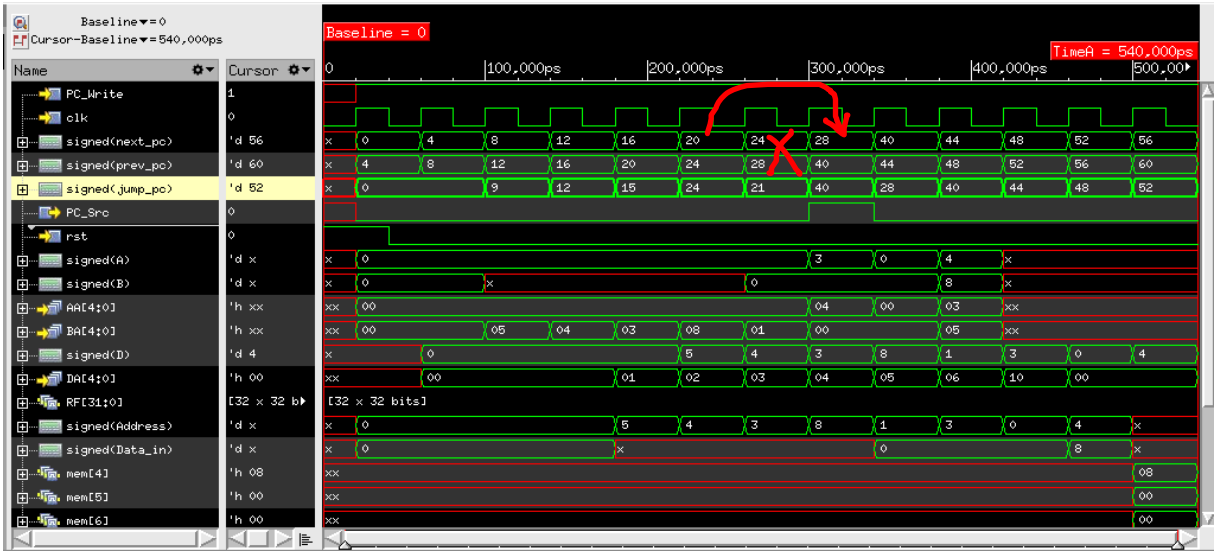
We can see that, firstlt r1 = 0, r2 = 5 , and r3 = 4 values are loaded. Then, it jumps to label jump1 wheher an addition (r7 = 5 + 4) and a shift left operations (r4 = 9*2) are done. After these are executed succesfully, program jumps to jump2 label where r4, r5 and r6 register are loaded with 3, 8 , and 8. At the end, program jumps to stop label where it enters infinite loop.

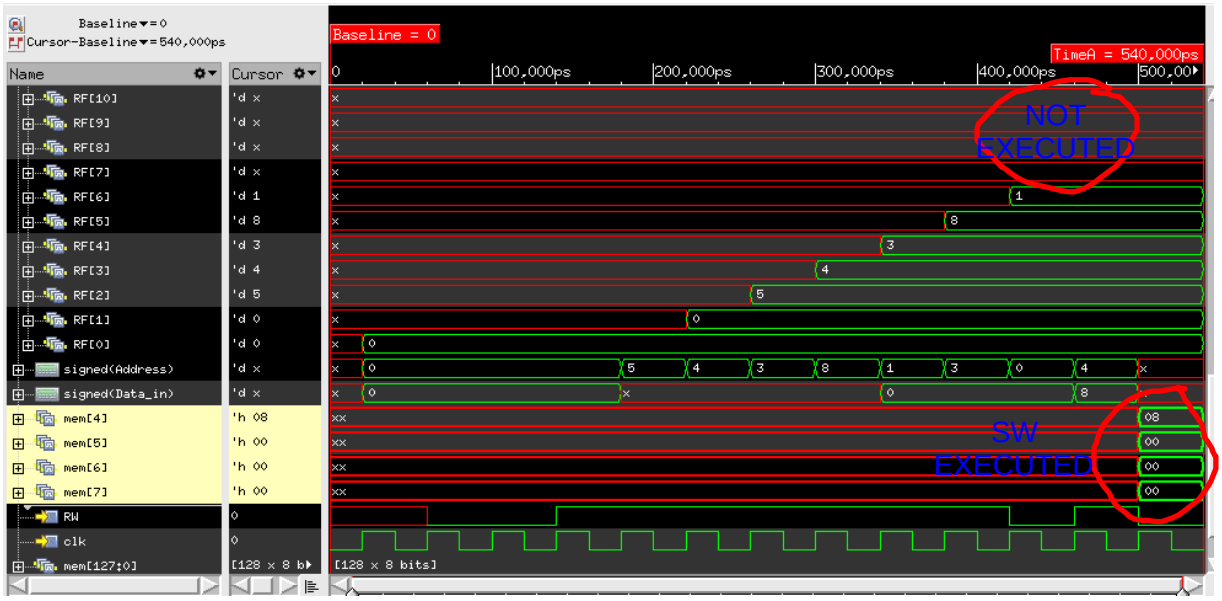
Simulation for Branch Instructions:

```
0       lui     r1, 0
4       addi    r2, r0, 5
8       addi    r3, r0, 4
12      addi    r4, r0, 3
16      addi    r5, r0, 8
20      addi    r6, r0, 1

24      bne     r2, r0, 16
28      add     r7, r2, r3
32      sub     r8, r2, r3
36      sll     r9, r7, r8
40      sw      r5, 0(r3)
```

In this set of instructions, registers between r1-r6 are loaded with some constant values then branch instruction comes. There are three possible hazards that may happen here.

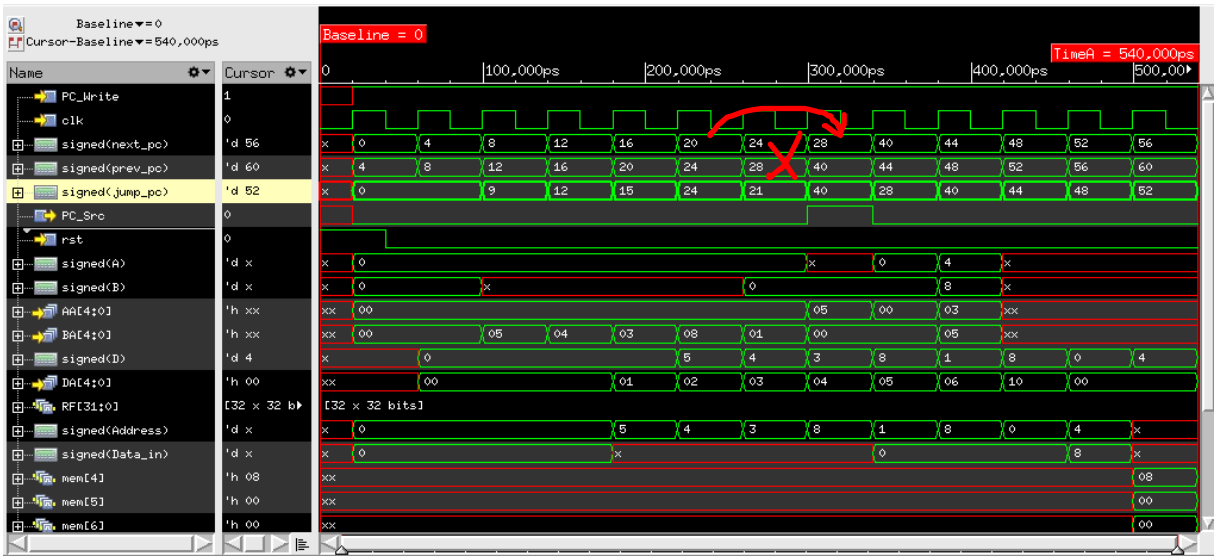
Case 1: bne r4, r0, 16

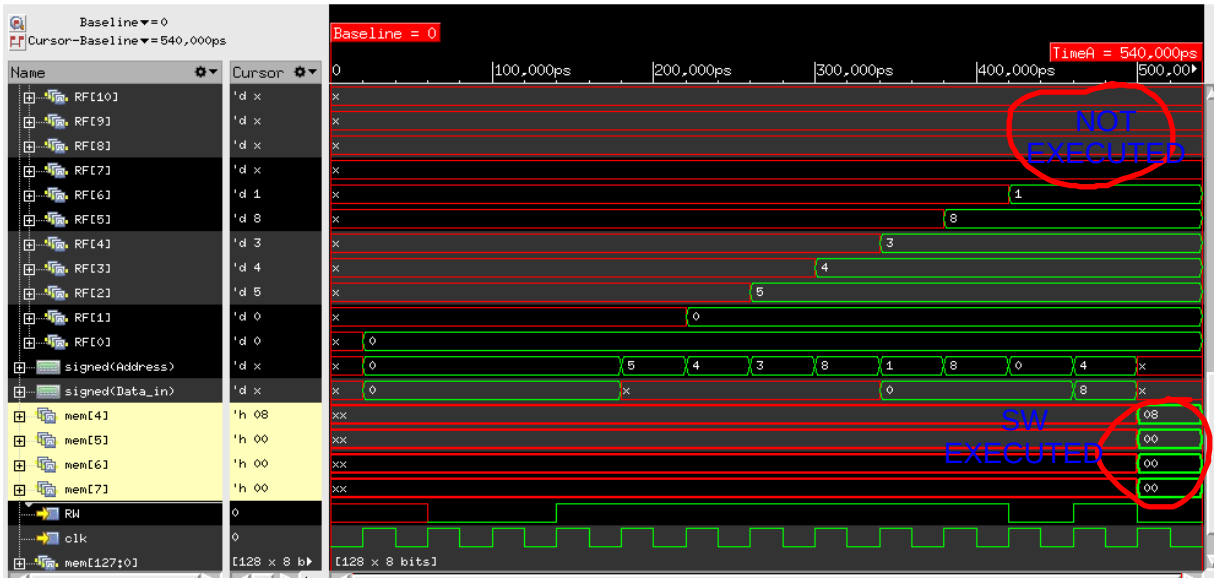




It can be seen from waveform that, when next_pc reached 24 which is branch instruction, next instruction in 28 address is converted to nop then program jumps to 0x40 address without making any disruption on data. It executes only store instruction and do not execute instructions on 0x28,0x32,0x36 address as expected. It can be confirmed by checking r7,r8, and r9 registers. R5 register value 8 is stored in memory as it can be see the bottom.

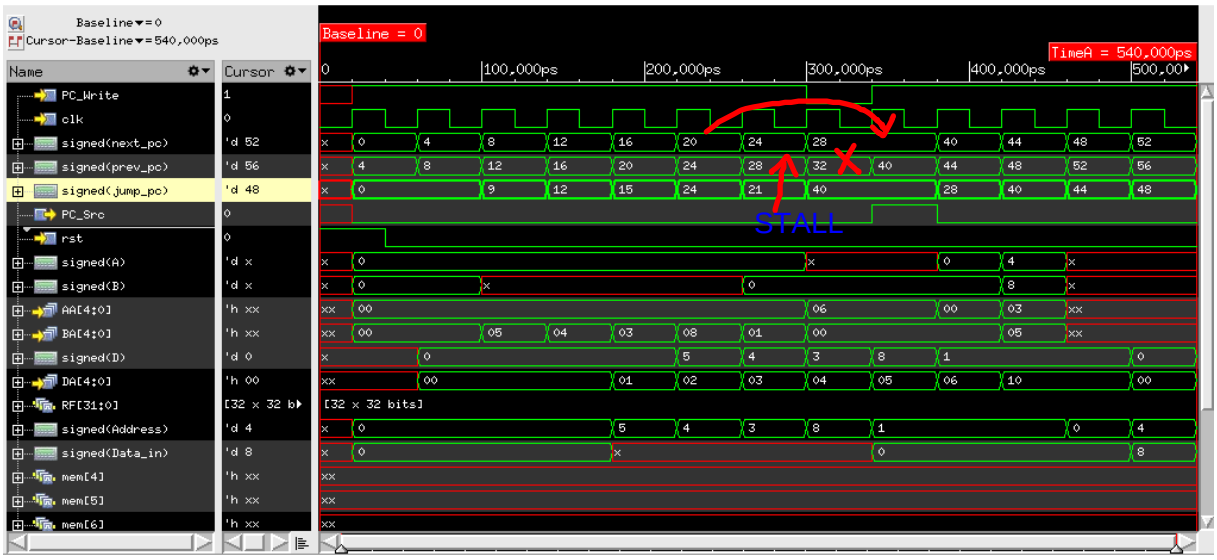
Case 2: bne r5, r0, 16

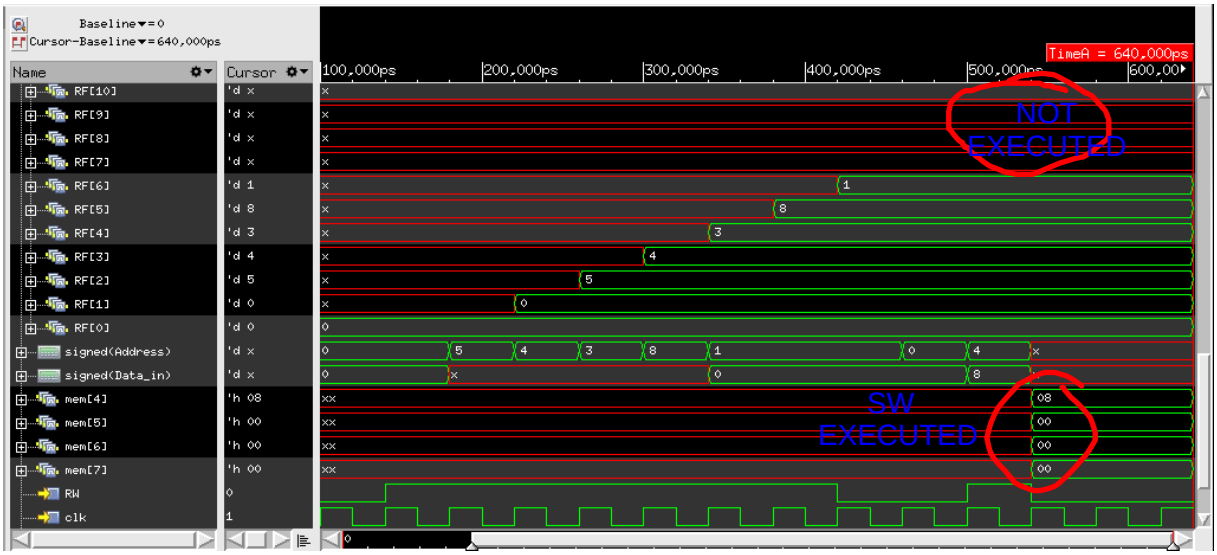




It can be seen from waveform that, when next_pc reached 24 which is branch instruction, next instruction in 28 address is converted to nop then program jumps to 0x40 address without making any disruption on data. It executes only store instruction and do not execute instructions on 0x28,0x32,0x36 address as expected. It can be confirmed by checking r7,r8, and r9 registers. R5 register value 8 is stored in memory as it can be see the bottom.

Case 3: bne r6, r0, 16

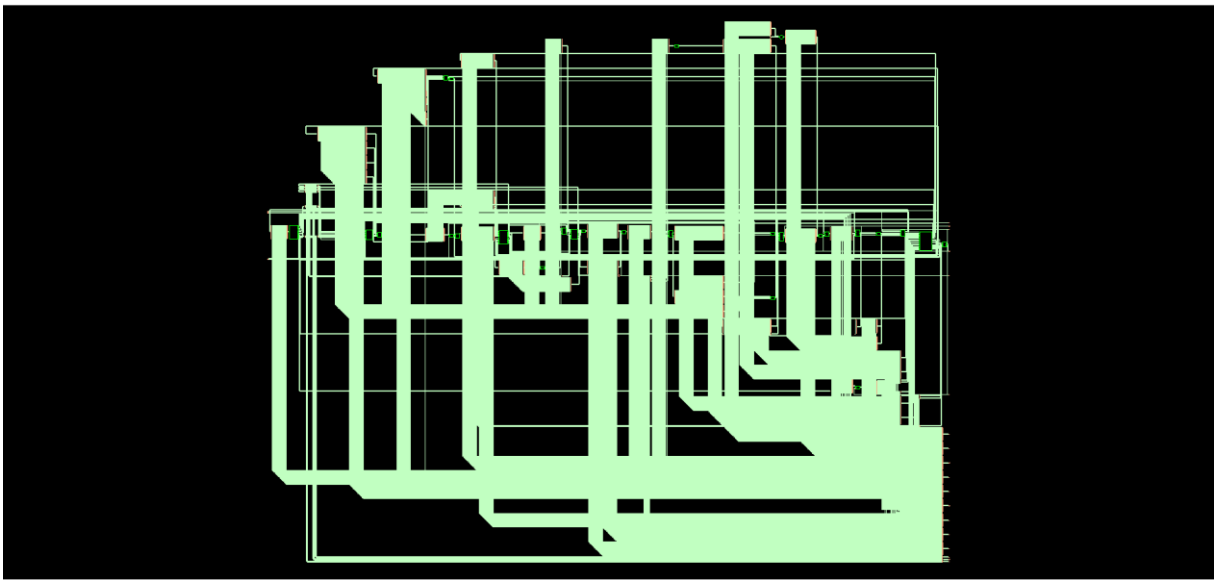


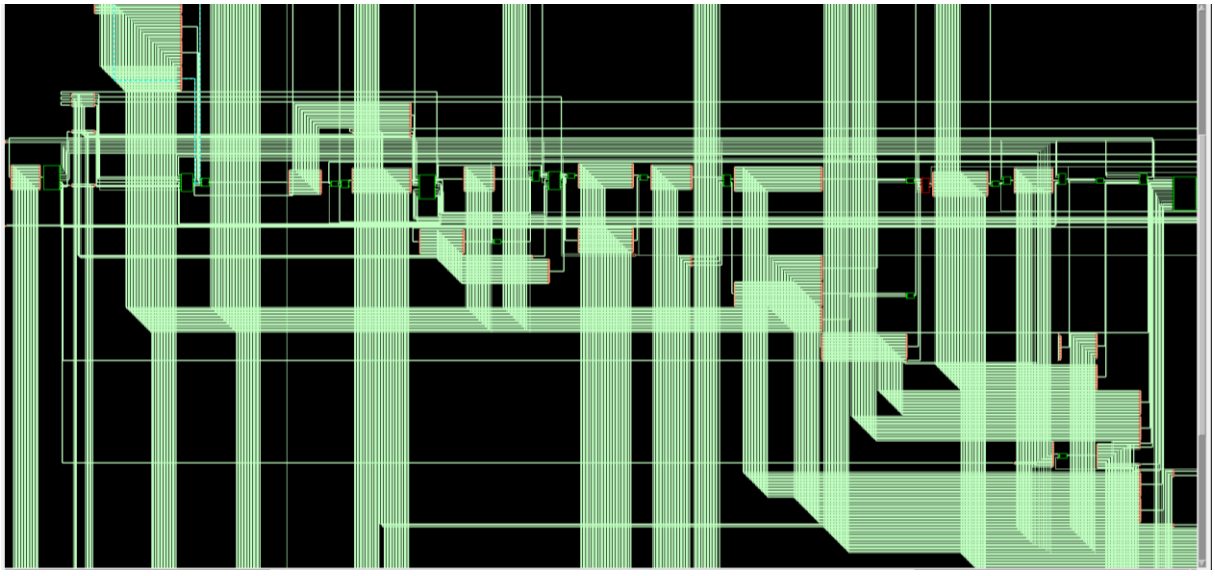


It can be seen from waveform that, when next_pc reached 24 which is branch instruction, load_use hazard is detected and stall is applied for the. Then next instruction in 32 address is converted to a nop then program jumps to 0x40 address without making any disruption on data. It executes only store instruction and do not execute instructions on 0x28,0x32,0x36 address as expected. It can be confirmed by checking r7,r8, and r9 registers. R5 register value 8 is stored in memory as it can be see the bottom.

7) Synthes

a) Post Elaboration Schematic

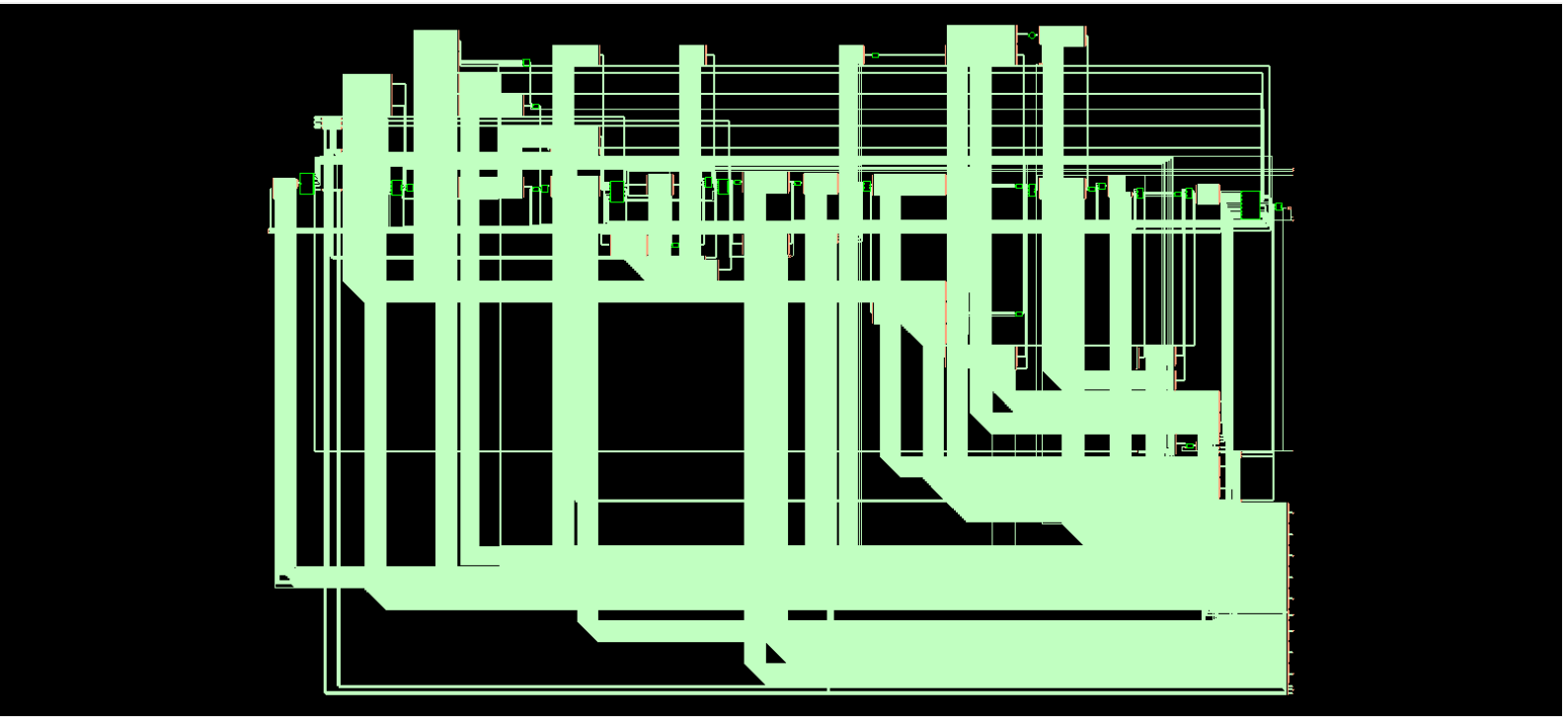




b) Instruction without NOP operations and corresponding machine code are shown below.

Assembly Code			Machine Code
0	lui	r1, 0	00000000000000000000000000110111
4	addi	r2, r0, 5	00000000001010000000000000100011
8	addi	r3, r0, 4	00000000001000000000000000110011
12	addi	r4, r0, 3	00000000000110000000000001000011
16	addi	r5, r0, 9	00000010000100000000000001010011
20	addi	r6, r0, 1	00000000000010000000000001100011
24	beq	r5, r6, 0	00000000011000101000000000100011
28	sub	r5, r5, r6	01000000011000101000000101010011
32	sll	r1, r1, r6	00000000011000000100100000101100
36	addi	r7, r3, 0	00000000000000000110000011100100
40	srli	r7, r7, 31	00000001111110011111010011100100
44	sll	r3, r3, r6	00000000011000001100100011010011
48	blt	r1, r2, 8	00000000000100000110001000110011
52	sub	r1, r1, r2	01000000000100000100000000101100
56	bne	r7, r6, -28	11111111001100011100100001110011
60	add	r1, r1, r4	00000000001000000100000000101100
64	blt	r1, r2, -36	11111100001000000110011001110011
68	sub	r1, r1, r2	01000000000100000100000000101100
72	jal	r7, -48	11111101000111111111001111101111

c) Post Synthesize Schematic



In my first try of synthesize, synthesize was deleting all my instances and giving a warning saying that the instances do not drive an output is deleted. Therefore, i ran following commands before "syn_gen":

"set_db delete_unloaded_seqs false"

"set_db auto_ungroup none"

After these two commands, synthesiz was succesfull and all my modeles with all connections was in place.

d) Testbench Code and Post Syntehsize Simulation :

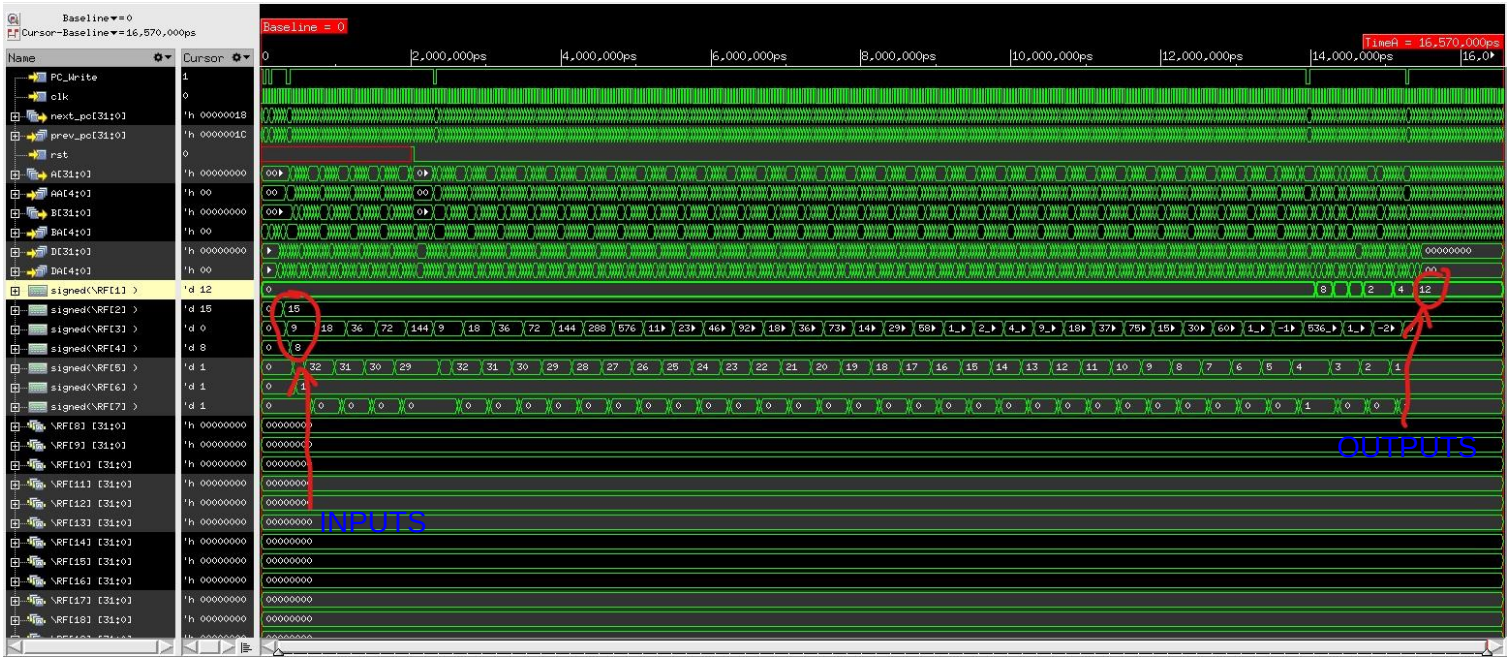
```
`timescale 1ns / 1ps

module MCP_RISC_V_tb();

    reg clk, rst, IM_we;
    reg [31:0] IM_addr, IM_data;

    MCP_RISC_V UUT(
        .clk(clk),
        .rst(rst),
        .IM_we(IM_we),
        .IM_addr(IM_addr),
        .IM_data(IM_data)
    );

    always #20 clk = ~clk;

    reg [31:0] mem_read[0:50];
    integer i;
    initial begin
        clk = 0; IM_we = 1; IM_addr = 0;
        $readmemb("D:/CODE FILES/VLSI CIRCUIT II/HW8/Branch/Branch.srcs/memory/raw2.txt", mem_read);
        for (i = 0; i<5; i = i+1)
        begin
            IM_data = mem_read[i];    //lui
            #40;
            IM_addr = IM_addr + 4;
        end

        rst = 1;
        #40;
        rst = 0; IM_we = 0;


    end

endmodule
```

Post Synthesize Simulation:

Simulation has been done for three different values of A, B, and N.

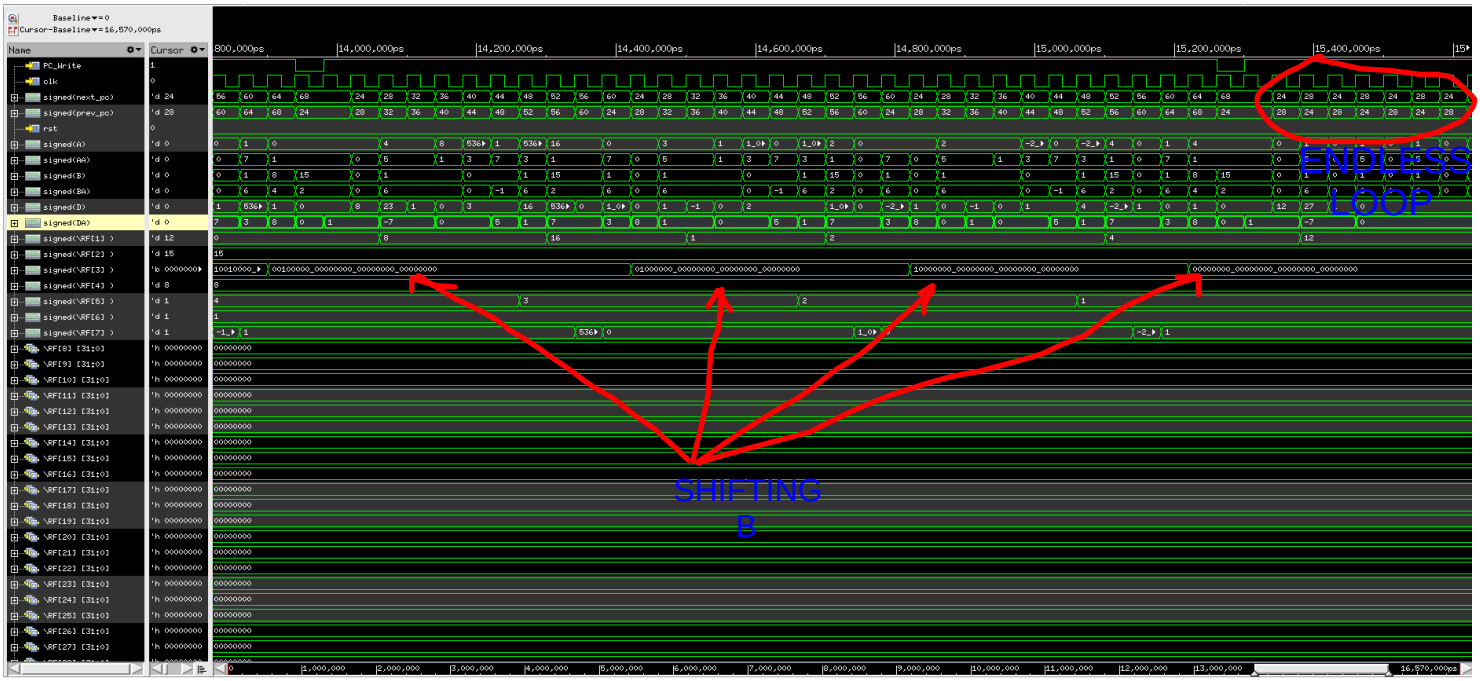
Example 1: A(r3) = 9, B(r4) = 8, N = 15(r2)

Expected result C (r1) = 8 x 9 = 72 mod (15) = 12



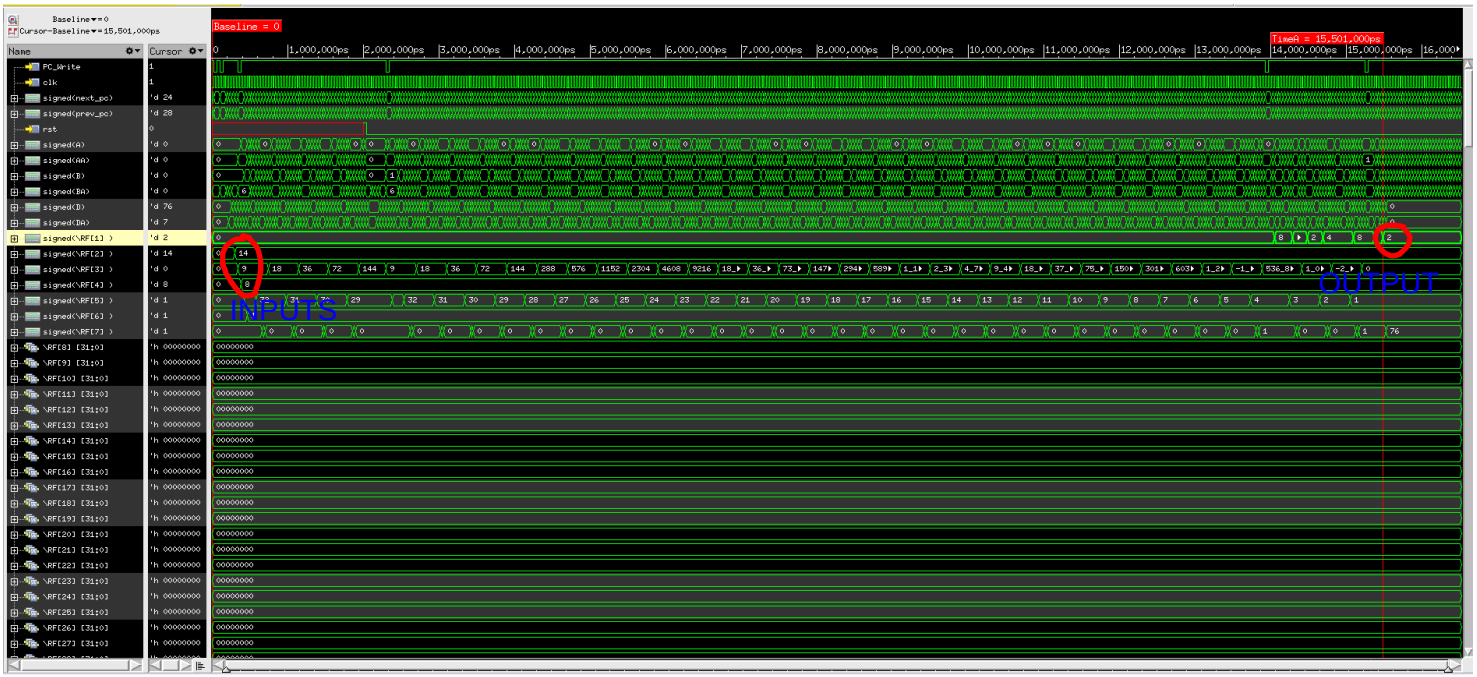
It can be seen from waveform that, expected result 12 has been succesfully calculated and written to r1 register which holds the value of C.

We can also observe program counter entering endless loop at the end and shifting operations made on B value one by one.



Example 2: A(r3) = 9, B(r4) = 8, N = 14(r2)

Expected result C (r1) = 8 x 9 = 72 mod (14) = 2



Example 3: A(r3) = 11, B(r4) = 8, N = 14(r2)

Expected result C (r1) = 11 x 8 = 88 mod (14) = 18