Pipeline CPU

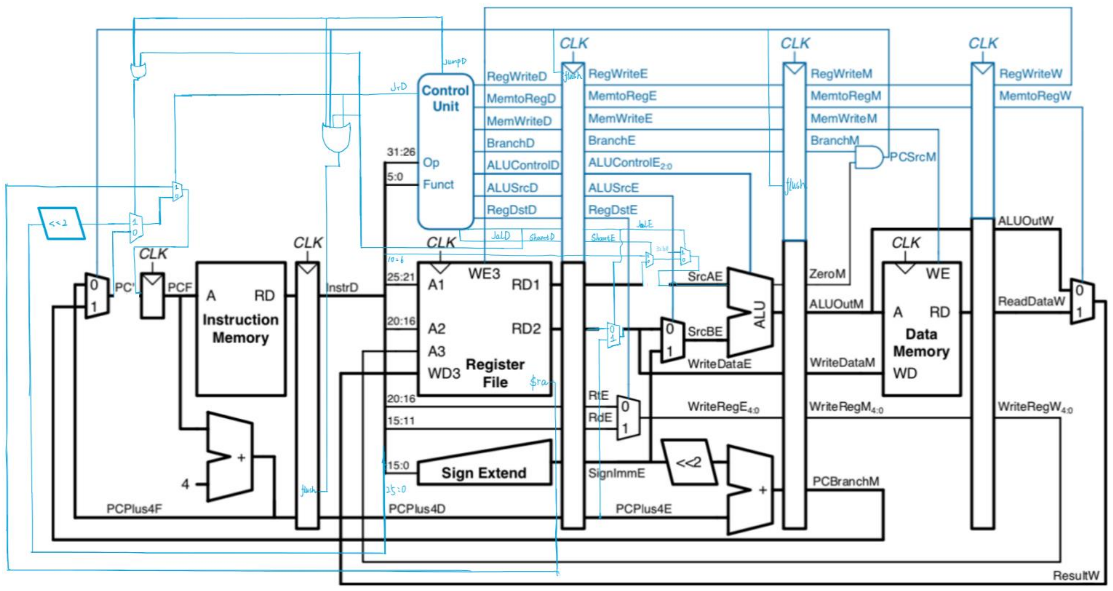
Some information about the project

The file CPU.v is the main program which implements the CPU. The files test_CPU_1.v, test_CPU_2.v, test_CPU_data_hazard.v and test_CPU_lw_stall.v are the test files used to test the CPU.v.

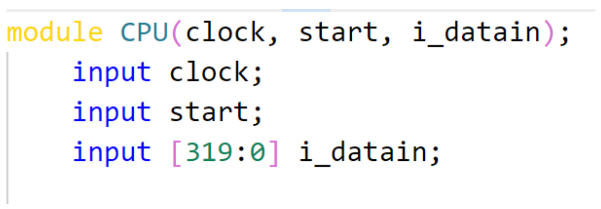
The main idea of the project

The project implements a pipeline CPU and solves the hazards of the pipeline. The details of the implementation are shown in the following.

Block diagram



The diagram above is the main structure of this project. Since the original diagrams given in the project is not completed, some extra circuits are added in this diagram to implement the required instructions. The whole CPU is built with one module, which will take the clock signal, start signal and the instruction data as input. The interface of the module is shown in the following:

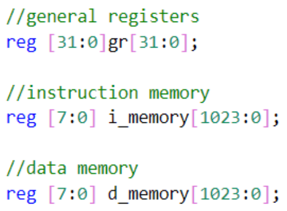


```
module CPU(clock, start, i_datain);
    input clock;
    input start;
    input [319:0] i_datain;
```

Components of the module

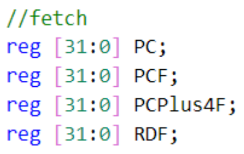
This module implements all nearly all the register shown in the diagrams and connect them in the way shown in the diagrams.

(1) general registers, instruction memory, data memory:



```
//general registers
reg [31:0]gr[31:0];

//instruction memory
reg [7:0] i_memory[1023:0];

//data memory
reg [7:0] d_memory[1023:0];
```

(2) registers in the fetch part:



```
//fetch
reg [31:0] PC;
reg [31:0] PCF;
reg [31:0] PCPlus4F;
reg [31:0] RDF;
```

(3) registers in the decoding part:

```
//decoding

//instruction holder
reg [31:0] InstrD;
reg [31:0] PCPlus4D;

//registers from control unit
reg RegWriteD;
reg MemtoRegD;
reg MemWriteD;
reg BranchD;
reg [4:0] ALUControlD;
reg ALuSrcD;
reg RegDstD;
reg ShamtD;
reg JumpD;
reg JalD;
reg JrD;
reg MemreadD;

reg [25:0] TargetD;
reg [31:0] ShamtImmD;
reg [4:0] A1D;
reg [4:0] A2D;
reg [4:0] A3D;
reg [31:0] WD3D;
reg [31:0] RD1D;
reg [31:0] RD2D;
reg [31:0] SignExtend;
reg [4:0] shamt;
reg [4:0] rt;
reg [4:0] rd;
```

(4) registers in the execution part:

```
//execution
reg RegWriteE;
reg MemtoRegE;
reg MemWriteE;
reg BranchE;
reg [4:0] ALUControlE;
reg ALuSrcE;
reg RegDstE;
reg ShamtE;
reg JalE;
reg RsHazardE;
reg RtHazardE;
reg MemreadE;
reg stall;

reg [31:0] ShamtImmE;
reg [4:0] RtE;
reg [4:0] RdE;
reg [4:0] WriteRegE;
reg [31:0] SignImmE;
reg [31:0] PCPlus4E;
reg [31:0] SrcAE;
reg [31:0] SrcBE;
reg [31:0] WriteDataE;
reg [31:0] ALUOutE;
reg [31:0] PCBranchE;
reg ZeroE;
reg negativeE;
reg overflowE;
```

(5) Registers in the memory part:

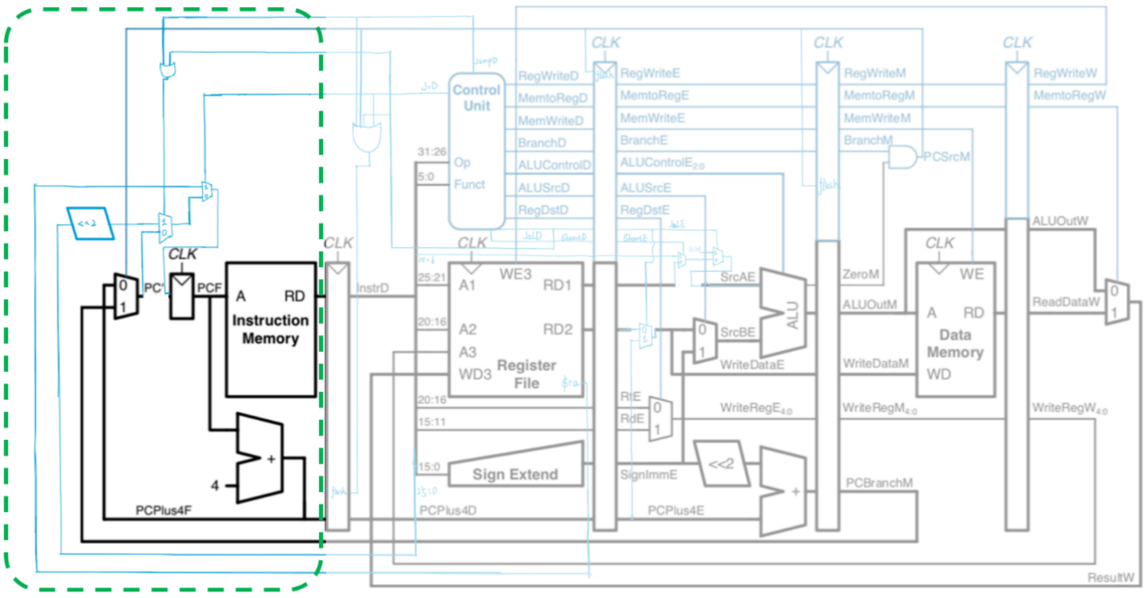
```
//memory
reg RegWriteM;
reg MemtoRegM;
reg MemWriteM;
reg BranchM;
reg RsHazardM;
reg RtHazardM;

reg ZeroM;
reg PCSrcM;
reg [31:0] AM;
reg [31:0] WDM;
reg [31:0] ALUOutM;
reg [4:0] WriteRegM;
reg [31:0] PCBranchM;
reg [31:0] RDM;
```

(6) registers in the write back part:

```
//write
reg RegWriteW;
reg MemtoRegW;

reg [31:0] ALUOutW;
reg [31:0] ResultW;
reg [4:0] WriteRegW;
```

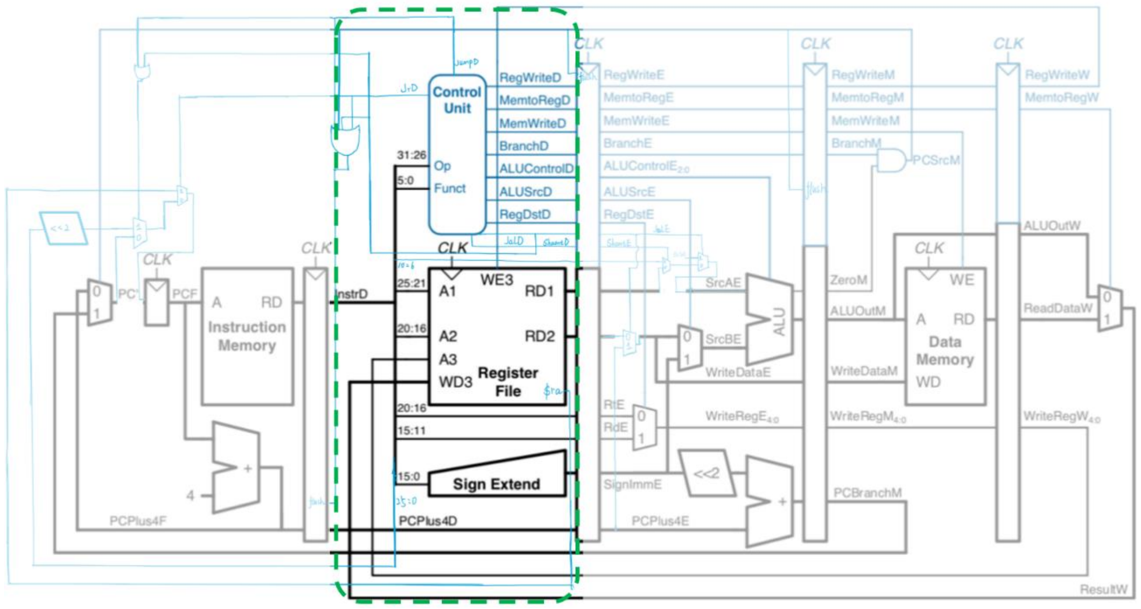
Five stages of the module:

(1) Instruction fetch:



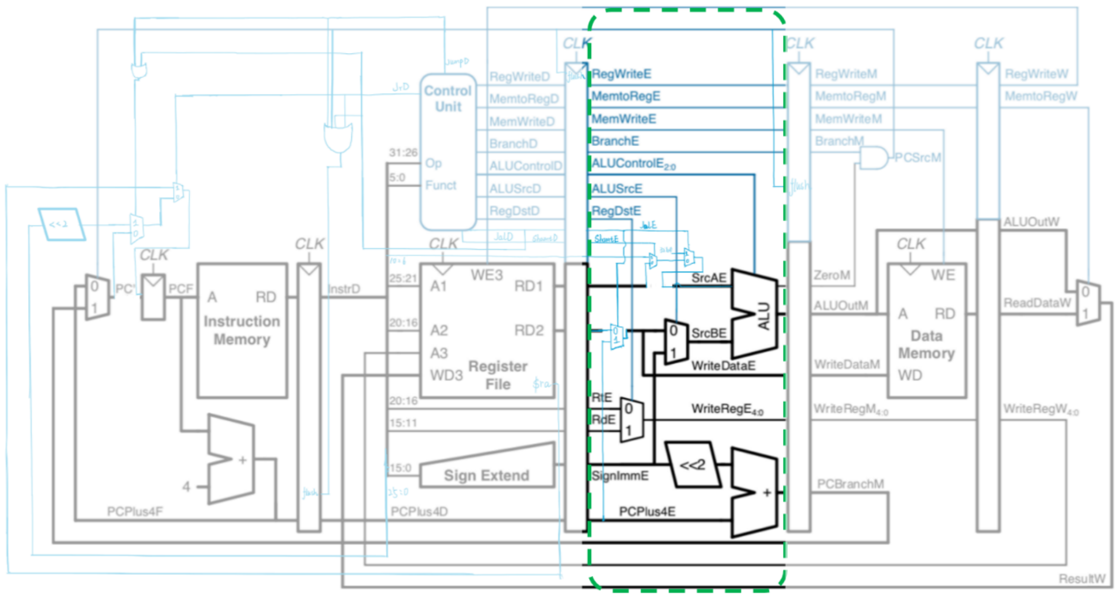
This part will decide the address of the instruction to be taken. According to the branch, j, jal, jr instructions, the multiplexers will select the corresponding data. After taking the address, this part will output the instructions stored in the address.

(2) Instruction decoding:



In this part, the circuits would do the decoding of the input instructions, do select the related general registers and output some corresponding data to the next stage. The control unit will output the control signals like RegWriteD and MemtoRegD according to the opcode and funct part of the instructions.

(3) Execution:

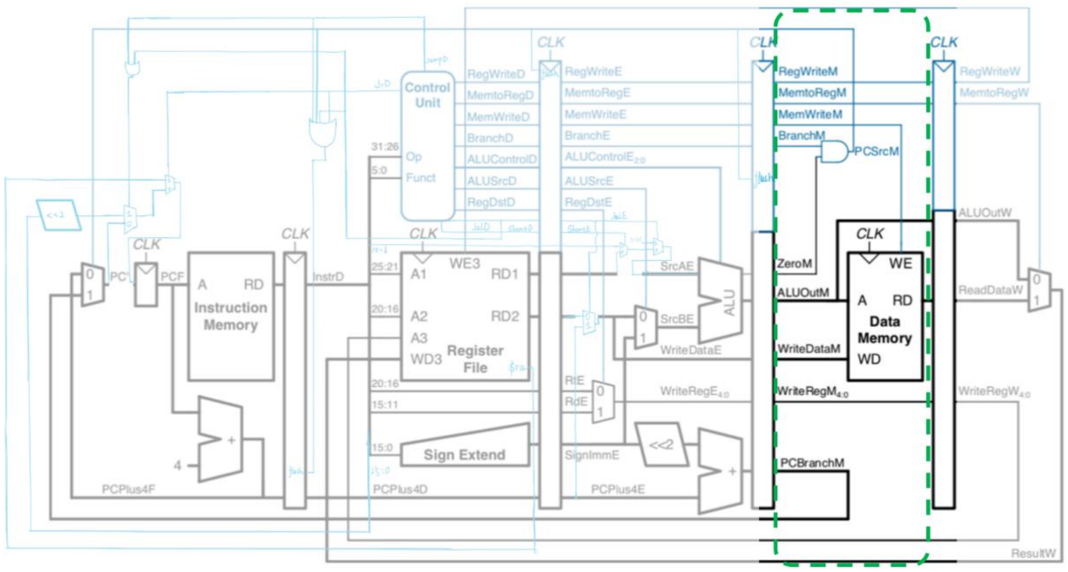


The ALU would do the corresponding operation in this part. The data of SrcAE and SrcBE will be determined by the corresponding signals. After the SrcAE and SrcBE are sent to the ALU, the ALU will do the operation according to the ALUControlE signal.

There are some special modifications in the ALU:

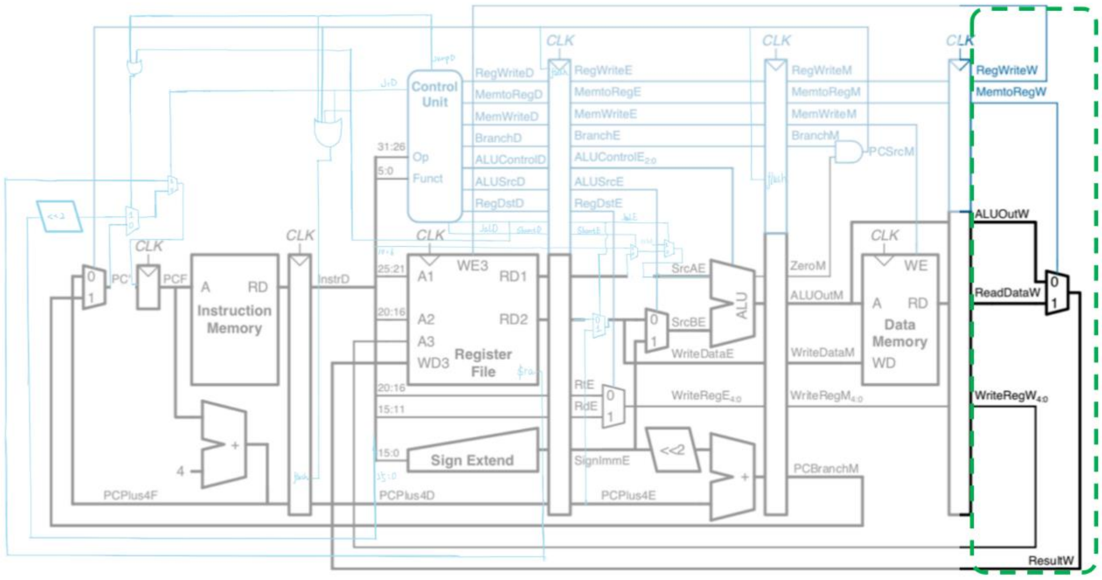
a. For andi and ori, the ALU will do the zero extension for the imm, which means after the sign extend imm passed into the ALU, the ALU will change the input as zero extend inside.
b. For bne, the result of Zero flag will be opposite to its real input, that is, it the rs equals to rt, zero flag will be set to 0 and it will be 1 if rs is not equal to rt. Designing in this way simplify the circuits and also perform the expected function.

(4) Memory operation



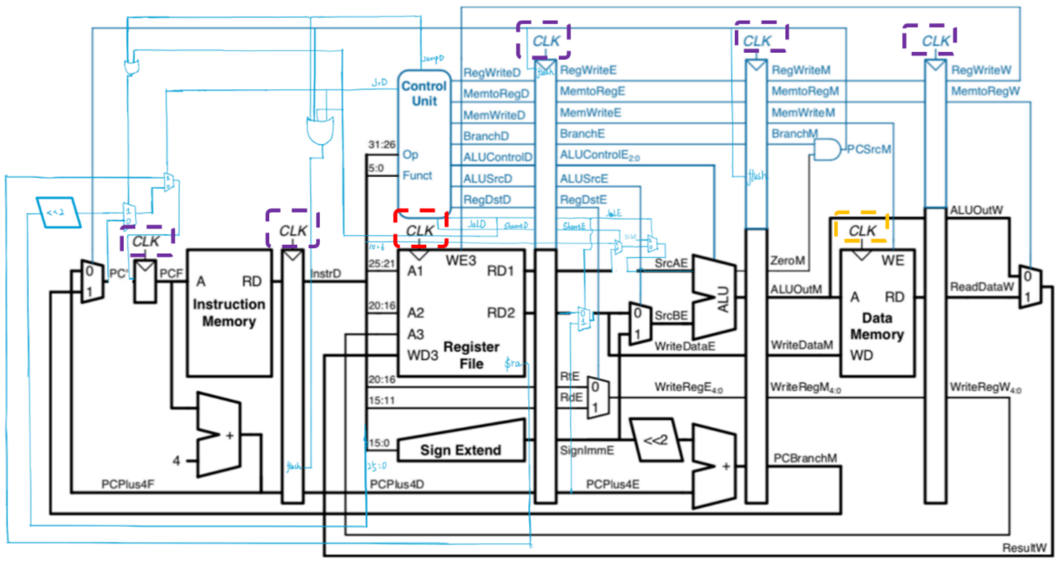
The operation related to the memory will be done in this part (lw, sw) according to the MemWriteM signal. Also, the branch signal (BranchM) will be generated in this stage, which will be sent to the corresponding places for branching instruction.

(5) Write back



In this part, some result might be written back to the registers. The MemtoRegW will select the data to write while the WriteRegW will select the register to be written. RegWriteW will decide whether to write the data or not.

Clock signals:



All the CLK signals in the dotted line circles are connect to the same clock signal.

For the part connect with CLK in the purple circles, it will trigger at the posedge of the clock. When triggered, it will pass the data from one register to another registers, which achieve the effect of pipeline.

For the part connect with CLK in red circles, which is the general registers part, the design is a little bit different. During the negedge of the clock signal, it will perform the function of output data in registers, that is, output the data in registers specified by A1 and A2. During the posedge of the clock signal, it will write the data to the register specified by A3.

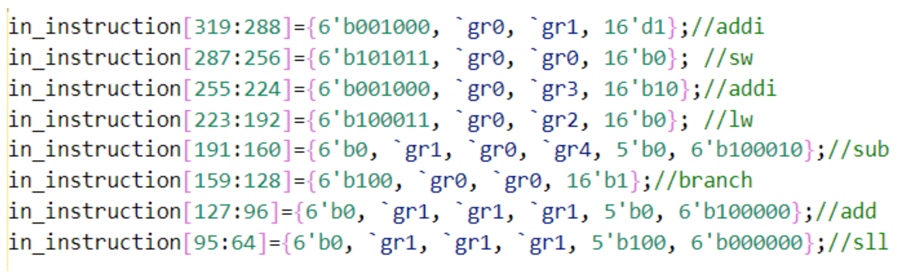
For the data memory part, which connects with CLK in yellow circle, it will output the data (32 bit) stored from the address of A during the posedge of the clock. If the signal MemWriteM is 1, the data in the WD will be written into the memory and output the data written into the WD.

Output sample:

There will be x test files to test the program.

(1) Test 1: (in the file: **test_CPU_1.v**)

Input:

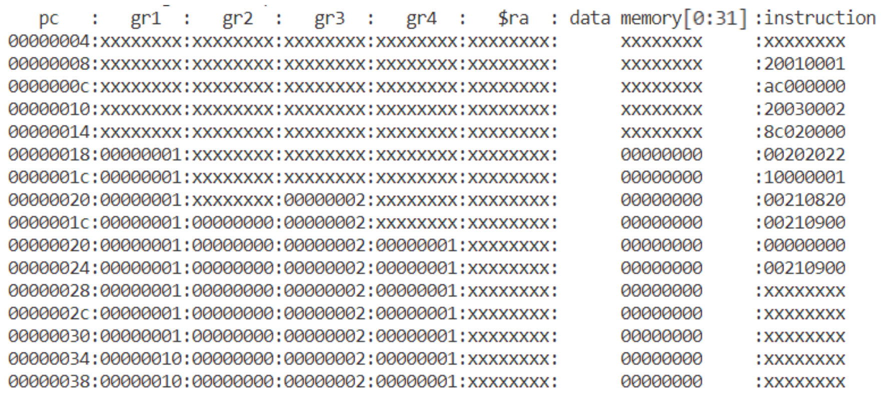


```
in_instruction[319:288]={6'b001000, `gr0, `gr1, 16'd1};//addi
in_instruction[287:256]={6'b101011, `gr0, `gr0, 16'b0}; //sw
in_instruction[255:224]={6'b001000, `gr0, `gr3, 16'b10};//addi
in_instruction[223:192]={6'b100011, `gr0, `gr2, 16'b0}; //lw
in_instruction[191:160]={6'b0, `gr1, `gr0, `gr4, 5'b0, 6'b100010};//sub
in_instruction[159:128]={6'b100, `gr0, `gr0, 16'b1};//branch
in_instruction[127:96]={6'b0, `gr1, `gr1, `gr1, 5'b0, 6'b100000};//add
in_instruction[95:64]={6'b0, `gr1, `gr1, `gr1, 5'b100, 6'b000000};//sll
```

Meaning:

1. gr0 + 1 => gr1
2. gr0 => data_memory[(gr0+0)+31, gr0+0]
3. gr0 + 2 => gr3
4. data_memory[(gr0+0)+31, gr0+0] => gr2
5. gr1 – gr0 => gr4
6. if (gr0 == gr0) branch to the next (1+1) instruction
7. gr1 + gr1 => gr1
8. gr1 << 4 => gr1

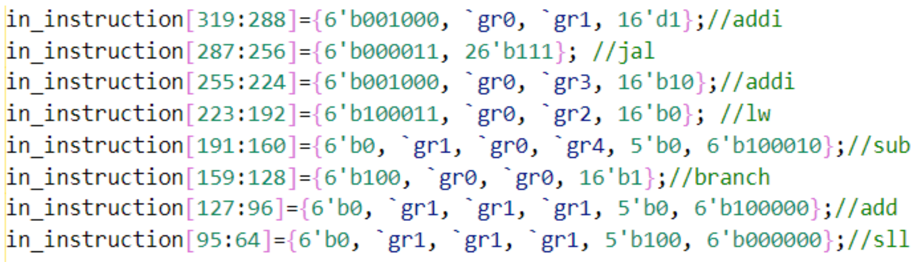
Output:



```
   pc    :   gr1   :   gr2   :   gr3   :   gr4   :   $ra   : data memory[0:31] :instruction
00000004:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:      xxxxxxxx     :xxxxxxxx
00000008:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:      xxxxxxxx     :20010001
0000000c:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:      xxxxxxxx     :ac000000
00000010:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:      xxxxxxxx     :20030002
00000014:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:      xxxxxxxx     :8c020000
00000018:00000001:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:      00000000     :00202022
0000001c:00000001:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:      00000000     :10000001
00000020:00000001:xxxxxxxx:00000002:xxxxxxxx:xxxxxxxx:      00000000     :00210820
0000001c:00000001:00000000:00000002:xxxxxxxx:xxxxxxxx:      00000000     :00210900
00000020:00000001:00000000:00000002:00000001:xxxxxxxx:      00000000     :00000000
00000024:00000001:00000000:00000002:00000001:xxxxxxxx:      00000000     :00210900
00000028:00000001:00000000:00000002:00000001:xxxxxxxx:      00000000     :xxxxxxxx
0000002c:00000001:00000000:00000002:00000001:xxxxxxxx:      00000000     :xxxxxxxx
00000030:00000001:00000000:00000002:00000001:xxxxxxxx:      00000000     :xxxxxxxx
00000034:00000010:00000000:00000002:00000001:xxxxxxxx:      00000000     :xxxxxxxx
00000038:00000010:00000000:00000002:00000001:xxxxxxxx:      00000000     :xxxxxxxx
```

The result is correct with the input. Notice that in instruction2, the word is stored in the memory which is at the same time when the gr1 was written in instruction 1, which shows the order of the pipeline is correct. After branch (instruction 6), the pc is branch to the target place. After the branch, the instructions already in the fetch and decode part is clear out, which can be shown that the instruction 7 is not executed (the details of the branch hazard will be shown in the branch hazard part). After the branch, the target instruction (instruction 8) is executed after 4 instructions since the branch flush some data in the first three stages.

(2) Test 2: (in the file: **test_CPU_2.v**)



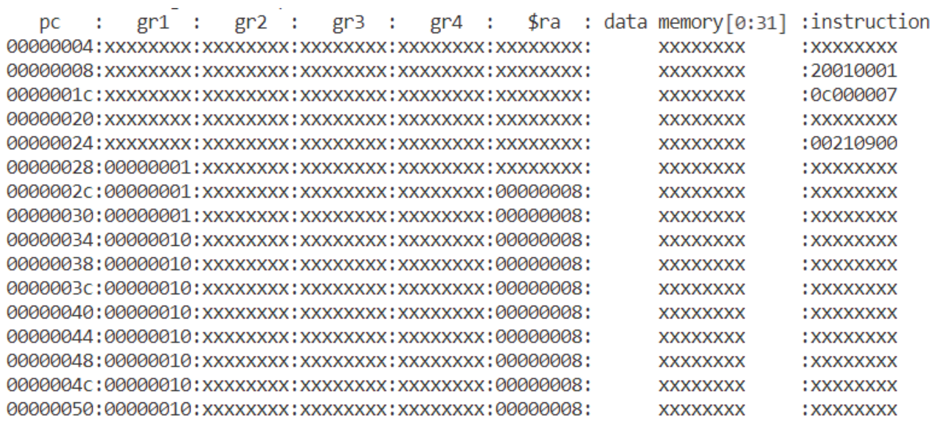
```
in_instruction[319:288]={6'b001000, `gr0, `gr1, 16'd1};//addi
in_instruction[287:256]={6'b000011, 26'b111}; //jal
in_instruction[255:224]={6'b001000, `gr0, `gr3, 16'b10};//addi
in_instruction[223:192]={6'b100011, `gr0, `gr2, 16'b0}; //lw
in_instruction[191:160]={6'b0, `gr1, `gr0, `gr4, 5'b0, 6'b100010};//sub
in_instruction[159:128]={6'b100, `gr0, `gr0, 16'b1};//branch
in_instruction[127:96]={6'b0, `gr1, `gr1, `gr1, 5'b0, 6'b100000};//add
in_instruction[95:64]={6'b0, `gr1, `gr1, `gr1, 5'b100, 6'b000000};//sll
```

Meaning:

1. gr0 + 1 => gr1
2. jump to {PC[31:28], 26’b111, 2’b00}
3. gr0 + 2 => gr3
4. data_memory[(gr0+0)+31, gr0+0] => gr2
5. gr1 – gr0 => gr4
6. if (gr0 == gr0) branch to the next (1+1) instruction

7. gr1 + gr1 => gr1
8. gr1 << 4 => gr1

Output:



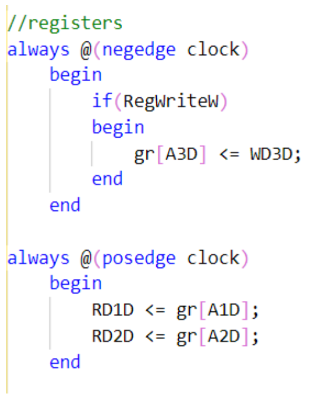
```
   pc    :   gr1   :   gr2   :   gr3   :   gr4   :   $ra   : data memory[0:31] :instruction
00000004:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:      xxxxxxxx     :xxxxxxxx
00000008:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:      xxxxxxxx     :20010001
0000001c:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:      xxxxxxxx     :0c000007
00000020:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:      xxxxxxxx     :xxxxxxxx
00000024:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:      xxxxxxxx     :00210900
00000028:00000001:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:      xxxxxxxx     :xxxxxxxx
0000002c:00000001:xxxxxxxx:xxxxxxxx:xxxxxxxx:00000008:      xxxxxxxx     :xxxxxxxx
00000030:00000001:xxxxxxxx:xxxxxxxx:xxxxxxxx:00000008:      xxxxxxxx     :xxxxxxxx
00000034:00000010:xxxxxxxx:xxxxxxxx:xxxxxxxx:00000008:      xxxxxxxx     :xxxxxxxx
00000038:00000010:xxxxxxxx:xxxxxxxx:xxxxxxxx:00000008:      xxxxxxxx     :xxxxxxxx
0000003c:00000010:xxxxxxxx:xxxxxxxx:xxxxxxxx:00000008:      xxxxxxxx     :xxxxxxxx
00000040:00000010:xxxxxxxx:xxxxxxxx:xxxxxxxx:00000008:      xxxxxxxx     :xxxxxxxx
00000044:00000010:xxxxxxxx:xxxxxxxx:xxxxxxxx:00000008:      xxxxxxxx     :xxxxxxxx
00000048:00000010:xxxxxxxx:xxxxxxxx:xxxxxxxx:00000008:      xxxxxxxx     :xxxxxxxx
0000004c:00000010:xxxxxxxx:xxxxxxxx:xxxxxxxx:00000008:      xxxxxxxx     :xxxxxxxx
00000050:00000010:xxxxxxxx:xxxxxxxx:xxxxxxxx:00000008:      xxxxxxxx     :xxxxxxxx
```

The result is correct with the input. In the instruction 2, the PC directly jump to the {PC[31:28], 26'b111, 2'b00}, which is the 8th instruction. The 8th instruction (sll) is performed which is shown in the output. Also, the address of the next instruction of the jump which is the 32'h8 is stored in the $ra.

Hazard:

(1) data hazard (data forwarding):

As mentioned in the Clock signals part, the general registers are doing the reading and writing at different time, that is, output the data in A1 and A2 registers at the negedge and writing the data in to the A3 register during the posedge, which is shown in the following:

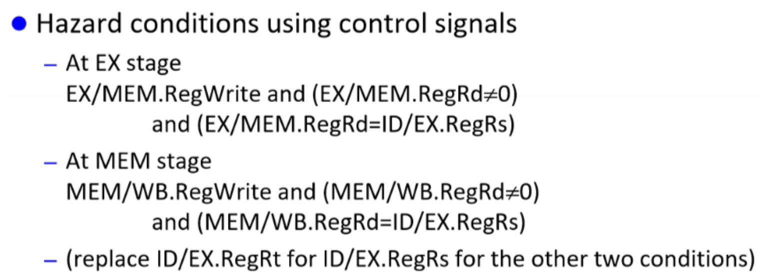


```
//registers
always @(negedge clock)
    begin
        if(RegWriteW)
        begin
            gr[A3D] <= WD3D;
        end
    end

always @(posedge clock)
    begin
        RD1D <= gr[A1D];
        RD2D <= gr[A2D];
    end
```

The program is also equipped with data forwarding to handle the data hazard.

With data forwarding, the program could use the data stored in the next stage or the stage after next stage.

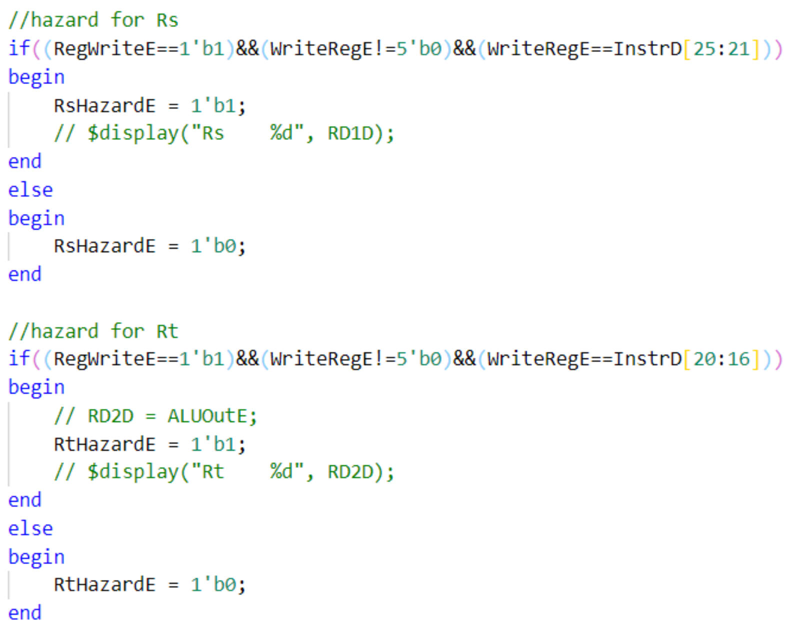
The algorithm is shown in the following:



- Hazard conditions using control signals
  - At EX stage
    EX/MEM.RegWrite and (EX/MEM.RegRd≠0)
    and (EX/MEM.RegRd=ID/EX.RegRs)
  - At MEM stage
    MEM/WB.RegWrite and (MEM/WB.RegRd≠0)
    and (MEM/WB.RegRd=ID/EX.RegRs)
  - (replace ID/EX.RegRt for ID/EX.RegRs for the other two conditions)

It is implemented in the way of:

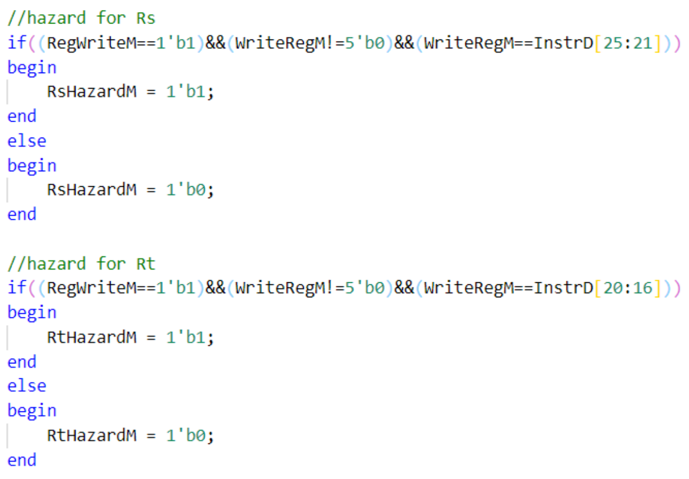
Part 1:



```
//hazard for Rs
if((RegWriteE==1'b1)&&(WriteRegE!=5'b0)&&(WriteRegE==InstrD[25:21]))
begin
    RsHazardE = 1'b1;
    // $display("Rs    %d", RD1D);
end
else
begin
    RsHazardE = 1'b0;
end

//hazard for Rt
if((RegWriteE==1'b1)&&(WriteRegE!=5'b0)&&(WriteRegE==InstrD[20:16]))
begin
    // RD2D = ALUOutE;
    RtHazardE = 1'b1;
    // $display("Rt    %d", RD2D);
end
else
begin
    RtHazardE = 1'b0;
end
```

(Notice that this forwarding is also work for the I-type instruction.)

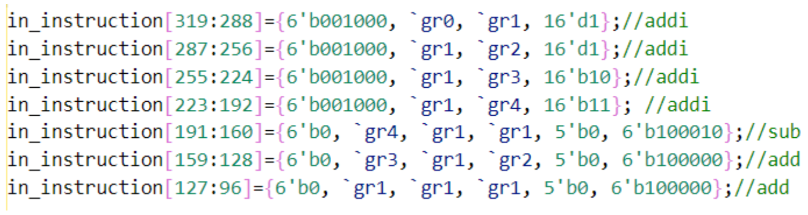
Part 2: (built inside the data memory part for convenience)



```
//hazard for Rs
if((RegWriteM==1'b1)&&(WriteRegM!=5'b0)&&(WriteRegM==InstrD[25:21]))
begin
    RsHazardM = 1'b1;
end
else
begin
    RsHazardM = 1'b0;
end

//hazard for Rt
if((RegWriteM==1'b1)&&(WriteRegM!=5'b0)&&(WriteRegM==InstrD[20:16]))
begin
    RtHazardM = 1'b1;
end
else
begin
    RtHazardM = 1'b0;
end
```

Test sample for data hazard:

Input: (in the file: **test_CPU_data_hazard.v**)

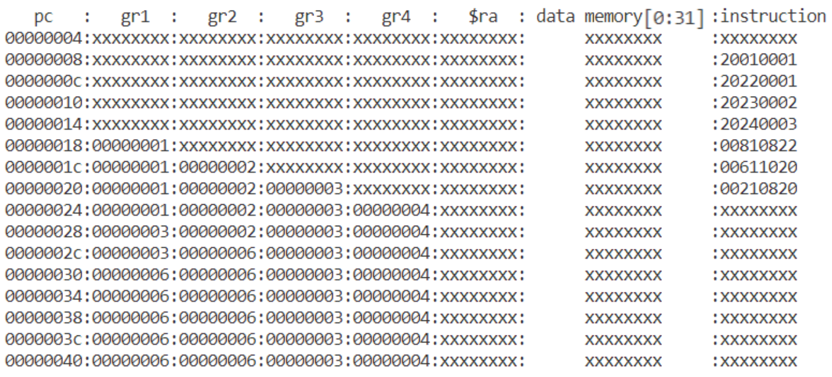


```
in_instruction[319:288]={6'b001000, `gr0, `gr1, 16'd1};//addi
in_instruction[287:256]={6'b001000, `gr1, `gr2, 16'd1};//addi
in_instruction[255:224]={6'b001000, `gr1, `gr3, 16'b10};//addi
in_instruction[223:192]={6'b001000, `gr1, `gr4, 16'b11}; //addi
in_instruction[191:160]={6'b0, `gr4, `gr1, `gr1, 5'b0, 6'b100010};//sub
in_instruction[159:128]={6'b0, `gr3, `gr1, `gr2, 5'b0, 6'b100000};//add
in_instruction[127:96]={6'b0, `gr1, `gr1, `gr1, 5'b0, 6'b100000};//add
```

Meaning:

1. gr0 + 1 => gr1
2. gr1 + 1 => gr2
3. gr1 + 2 => gr3
4. gr1 + 3 => gr4
5. gr4 – gr1 => gr1
6. gr3 + gr1 => gr2
7. gr1 + gr1 => gr1

Output:

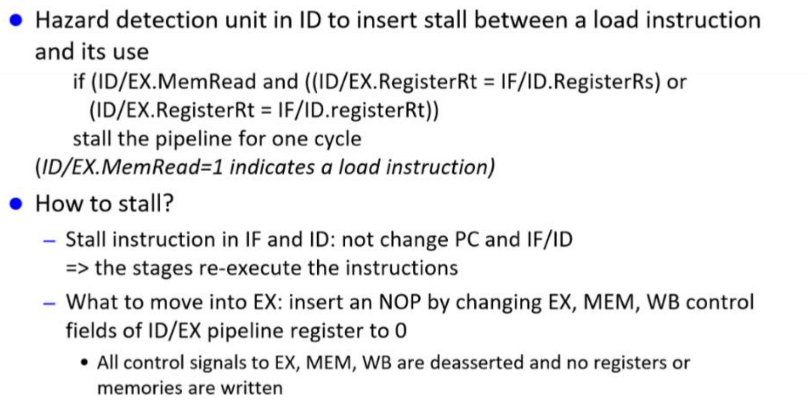


```
   pc    :   gr1   :   gr2   :   gr3   :   gr4   :   $ra   : data memory[0:31] :instruction
00000004:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:      xxxxxxxx     :xxxxxxxx
00000008:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:      xxxxxxxx     :20010001
0000000c:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:      xxxxxxxx     :20220001
00000010:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:      xxxxxxxx     :20230002
00000014:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:      xxxxxxxx     :20240003
00000018:00000001:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:      xxxxxxxx     :00810822
0000001c:00000001:00000002:xxxxxxxx:xxxxxxxx:xxxxxxxx:      xxxxxxxx     :00611020
00000020:00000001:00000002:00000003:xxxxxxxx:xxxxxxxx:      xxxxxxxx     :00210820
00000024:00000001:00000002:00000003:00000004:xxxxxxxx:      xxxxxxxx     :xxxxxxxx
00000028:00000003:00000002:00000003:00000004:xxxxxxxx:      xxxxxxxx     :xxxxxxxx
0000002c:00000003:00000006:00000003:00000004:xxxxxxxx:      xxxxxxxx     :xxxxxxxx
00000030:00000006:00000006:00000003:00000004:xxxxxxxx:      xxxxxxxx     :xxxxxxxx
00000034:00000006:00000006:00000003:00000004:xxxxxxxx:      xxxxxxxx     :xxxxxxxx
00000038:00000006:00000006:00000003:00000004:xxxxxxxx:      xxxxxxxx     :xxxxxxxx
0000003c:00000006:00000006:00000003:00000004:xxxxxxxx:      xxxxxxxx     :xxxxxxxx
00000040:00000006:00000006:00000003:00000004:xxxxxxxx:      xxxxxxxx     :xxxxxxxx
```

The result is correct with the input. Notice that this input shows that it performs the data forwarding correctly since it solves the data hazards correctly in the input.

(2) data hazard: (lw stall)

A special type of data hazard is that the lw instruction load words to the

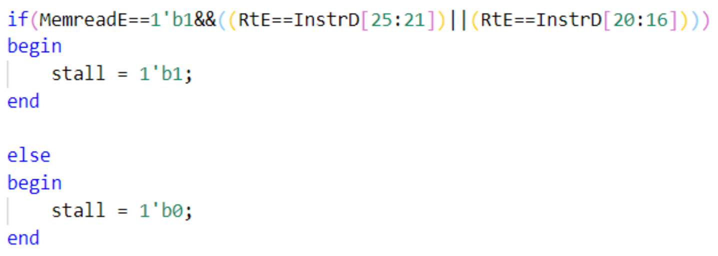


- Hazard detection unit in ID to insert stall between a load instruction and its use
    if (ID/EX.MemRead and ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
      (ID/EX.RegisterRt = IF/ID.registerRt))
    stall the pipeline for one cycle
  (ID/EX.MemRead=1 indicates a load instruction)
- How to stall?
  - Stall instruction in IF and ID: not change PC and IF/ID
    => the stages re-execute the instructions
  - What to move into EX: insert an NOP by changing EX, MEM, WB control fields of ID/EX pipeline register to 0
    - All control signals to EX, MEM, WB are deasserted and no registers or memories are written

register will be used in the next following instruction. To solve this problem, the program is equipped with lw stall, implemented with the algorithm shown in the following:

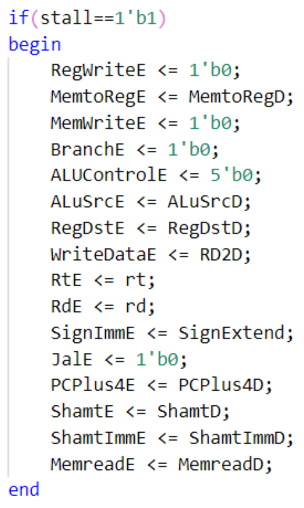
The code is:

Judge condition:



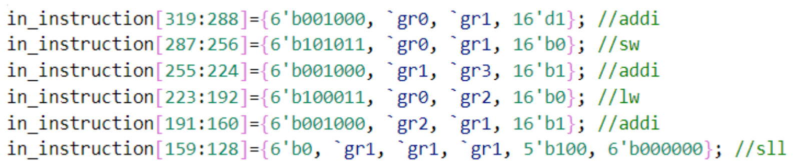
```
if(MemreadE==1'b1&&((RtE==InstrD[25:21])||(RtE==InstrD[20:16])))
begin
    stall = 1'b1;
end

else
begin
    stall = 1'b0;
end
```

Stall operation:



```
if(stall==1'b1)
begin
    RegWriteE <= 1'b0;
    MemtoRegE <= MemtoRegD;
    MemWriteE <= 1'b0;
    BranchE <= 1'b0;
    ALUControlE <= 5'b0;
    ALuSrcE <= ALuSrcD;
    RegDstE <= RegDstD;
    WriteDataE <= RD2D;
    RtE <= rt;
    RdE <= rd;
    SignImmE <= SignExtend;
    JalE <= 1'b0;
    PCPlus4E <= PCPlus4D;
    ShamtE <= ShamtD;
    ShamtImmE <= ShamtImmD;
    MemreadE <= MemreadD;
end
```

Test sample: (in the file: **test_CPU_lw_stall.v**)

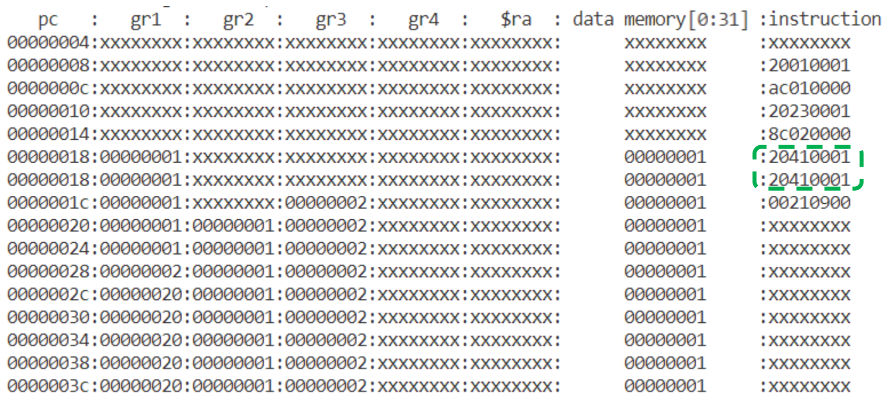


```
in_instruction[319:288]={6'b001000, `gr0, `gr1, 16'd1}; //addi
in_instruction[287:256]={6'b101011, `gr0, `gr1, 16'b0}; //sw
in_instruction[255:224]={6'b001000, `gr1, `gr3, 16'b1}; //addi
in_instruction[223:192]={6'b100011, `gr0, `gr2, 16'b0}; //lw
in_instruction[191:160]={6'b001000, `gr2, `gr1, 16'b1}; //addi
in_instruction[159:128]={6'b0, `gr1, `gr1, `gr1, 5'b100, 6'b000000}; //sll
```

Meaning:

1. gr0 + 1 => gr1
2. gr1 => data_memory[(gr0+0)+31, gr0]
3. gr1 + 1 => gr3
4. data_memory[(gr0+31)+0, gr0] => gr2
5. gr2 + 1 => gr1
6. gr1 << 4 => gr1

Output:



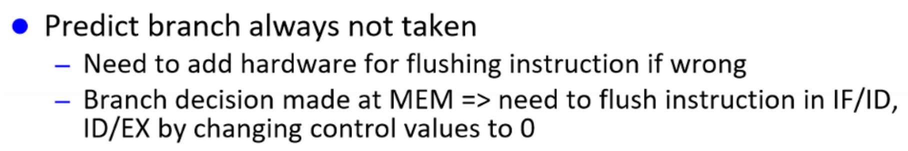
```
   pc    :   gr1   :   gr2   :   gr3   :   gr4   :   $ra   : data memory[0:31] :instruction
 00000004:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:      xxxxxxxx     :xxxxxxxx
 00000008:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:      xxxxxxxx     :20010001
 0000000c:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:      xxxxxxxx     :ac010000
 00000010:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:      xxxxxxxx     :20230001
 00000014:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:      xxxxxxxx     :8c020000
 00000018:00000001:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:      00000001     :20410001
 00000018:00000001:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:      00000001     :20410001
 0000001c:00000001:xxxxxxxx:00000002:xxxxxxxx:xxxxxxxx:      00000001     :00210900
 00000020:00000001:00000001:00000002:xxxxxxxx:xxxxxxxx:      00000001     :xxxxxxxx
 00000024:00000001:00000001:00000002:xxxxxxxx:xxxxxxxx:      00000001     :xxxxxxxx
 00000028:00000002:00000001:00000002:xxxxxxxx:xxxxxxxx:      00000001     :xxxxxxxx
 0000002c:00000020:00000001:00000002:xxxxxxxx:xxxxxxxx:      00000001     :xxxxxxxx
 00000030:00000020:00000001:00000002:xxxxxxxx:xxxxxxxx:      00000001     :xxxxxxxx
 00000034:00000020:00000001:00000002:xxxxxxxx:xxxxxxxx:      00000001     :xxxxxxxx
 00000038:00000020:00000001:00000002:xxxxxxxx:xxxxxxxx:      00000001     :xxxxxxxx
 0000003c:00000020:00000001:00000002:xxxxxxxx:xxxxxxxx:      00000001     :xxxxxxxx
```

The result is correct with the input. Notice that the instruction 5 use the register in the instruction 4 (lw), which caused a stall in the instruction.

The stall is shown in the green circle in the output. After stall, the result of instruction 5 is correct, which is shown by gr1 is changed into 2, which also reflects the value of gr2 is correct.

(3) Branch hazard:

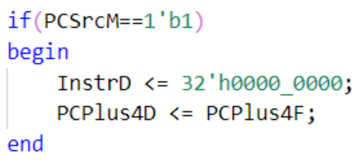
After the branch instruction, the CPU needs to flush the data stored in the previous stages. The algorithm is shown in the following:



- Predict branch always not taken
  - Need to add hardware for flushing instruction if wrong
  - Branch decision made at MEM => need to flush instruction in IF/ID, ID/EX by changing control values to 0

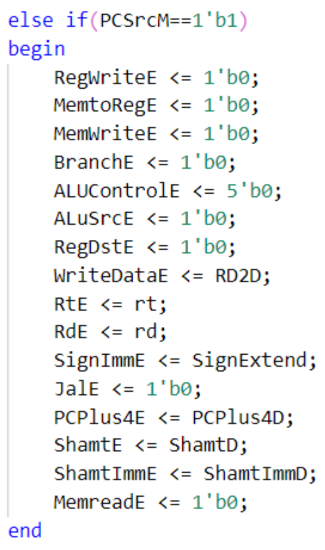
Flush implementation:

Flush the fetch/decode



```
if(PCSrcM==1'b1)
begin
    InstrD <= 32'h0000_0000;
    PCPlus4D <= PCPlus4F;
end
```

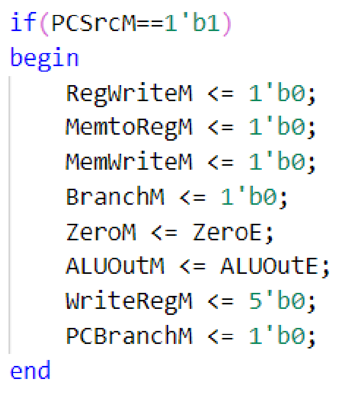
Flush the decode/execution



```
else if(PCSrcM==1'b1)
begin
    RegWriteE <= 1'b0;
    MemtoRegE <= 1'b0;
    MemWriteE <= 1'b0;
    BranchE <= 1'b0;
    ALUControlE <= 5'b0;
    ALuSrcE <= 1'b0;
    RegDstE <= 1'b0;
    WriteDataE <= RD2D;
    RtE <= rt;
    RdE <= rd;
    SignImmE <= SignExtend;
    JalE <= 1'b0;
    PCPlus4E <= PCPlus4D;
    ShamtE <= ShamtD;
    ShamtImmE <= ShamtImmD;
    MemreadE <= 1'b0;
end
```

Due to the special design of the program:

It also needs to flush the execution/memory

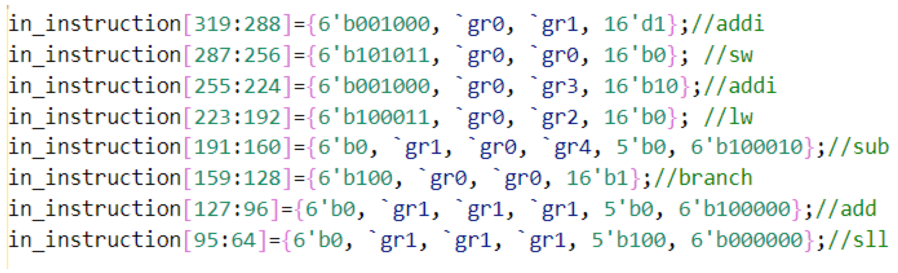


```
if(PCSrcM==1'b1)
begin
    RegWriteM <= 1'b0;
    MemtoRegM <= 1'b0;
    MemWriteM <= 1'b0;
    BranchM <= 1'b0;
    ZeroM <= ZeroE;
    ALUOutM <= ALUOutE;
    WriteRegM <= 5'b0;
    PCBranchM <= 1'b0;
end
```

Test sample: (in the file: **test_CPU_1.v**)

(3) Test: (in the file: **test_CPU_1.v**)

Input:

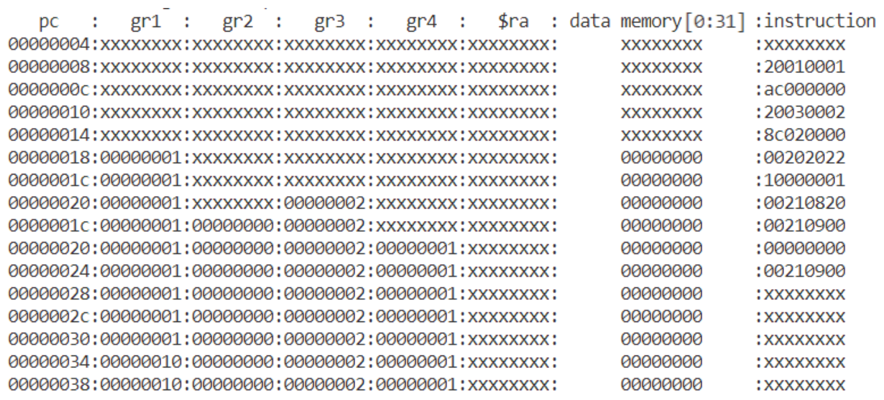


```
in_instruction[319:288]={6'b001000, `gr0, `gr1, 16'd1};//addi
in_instruction[287:256]={6'b101011, `gr0, `gr0, 16'b0}; //sw
in_instruction[255:224]={6'b001000, `gr0, `gr3, 16'b10};//addi
in_instruction[223:192]={6'b100011, `gr0, `gr2, 16'b0}; //lw
in_instruction[191:160]={6'b0, `gr1, `gr0, `gr4, 5'b0, 6'b100010};//sub
in_instruction[159:128]={6'b100, `gr0, `gr0, 16'b1};//branch
in_instruction[127:96]={6'b0, `gr1, `gr1, `gr1, 5'b0, 6'b100000};//add
in_instruction[95:64]={6'b0, `gr1, `gr1, `gr1, 5'b100, 6'b000000};//sll
```

Meaning:

1. gr0 + 1 => gr1
2. gr0 => data_memory[(gr0+0)+31, gr0+0]
3. gr0 + 2 => gr3
4. data_memory[(gr0+0)+31, gr0+0] => gr2
5. gr1 – gr0 => gr4
6. if (gr0 == gr0) branch to the next (1+1) instruction
7. gr1 + gr1 => gr1
8. gr1 << 4 => gr1

Output:



```
   pc    :   gr1   :   gr2   :   gr3   :   gr4   :   $ra   : data memory[0:31] :instruction
00000004:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:      xxxxxxxx     :xxxxxxxx
00000008:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:      xxxxxxxx     :20010001
0000000c:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:      xxxxxxxx     :ac000000
00000010:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:      xxxxxxxx     :20030002
00000014:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:      xxxxxxxx     :8c020000
00000018:00000001:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:      00000000     :00202022
0000001c:00000001:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:      00000000     :10000001
00000020:00000001:xxxxxxxx:00000002:xxxxxxxx:xxxxxxxx:      00000000     :00210820
0000001c:00000001:00000000:00000002:xxxxxxxx:xxxxxxxx:      00000000     :00210900
00000020:00000001:00000000:00000002:00000001:xxxxxxxx:      00000000     :00000000
00000024:00000001:00000000:00000002:00000001:xxxxxxxx:      00000000     :00210900
00000028:00000001:00000000:00000002:00000001:xxxxxxxx:      00000000     :xxxxxxxx
0000002c:00000001:00000000:00000002:00000001:xxxxxxxx:      00000000     :xxxxxxxx
00000030:00000001:00000000:00000002:00000001:xxxxxxxx:      00000000     :xxxxxxxx
00000034:00000010:00000000:00000002:00000001:xxxxxxxx:      00000000     :xxxxxxxx
00000038:00000010:00000000:00000002:00000001:xxxxxxxx:      00000000     :xxxxxxxx
```

The result is correct with the input. After branch (instruction 6), the pc is branch to the target place (instruction 8), which can be shown by that the program performed the instruction 8. The rest of the instruction in the fetch, decode, execute stages during the branch is flushed out, since the program did not perform the instruction 7 (because pc branch from instruction 6 to instruction 8). And this result shows that the program solves the branch hazard correctly.