3050 PROJECT4 REPORT

1. BASIC IDEA

This project asks us to design a 5-stage pipeline processor that can deal with data/control hazards. In order to achieve the goal, every instruction is divided into 5 stages:

a. IF: Instruction fetch

b. ID: Instruction decode and register file read

c. EX: Execution or address calculation

d. MEM: Data memory access

e. WB: Write Back

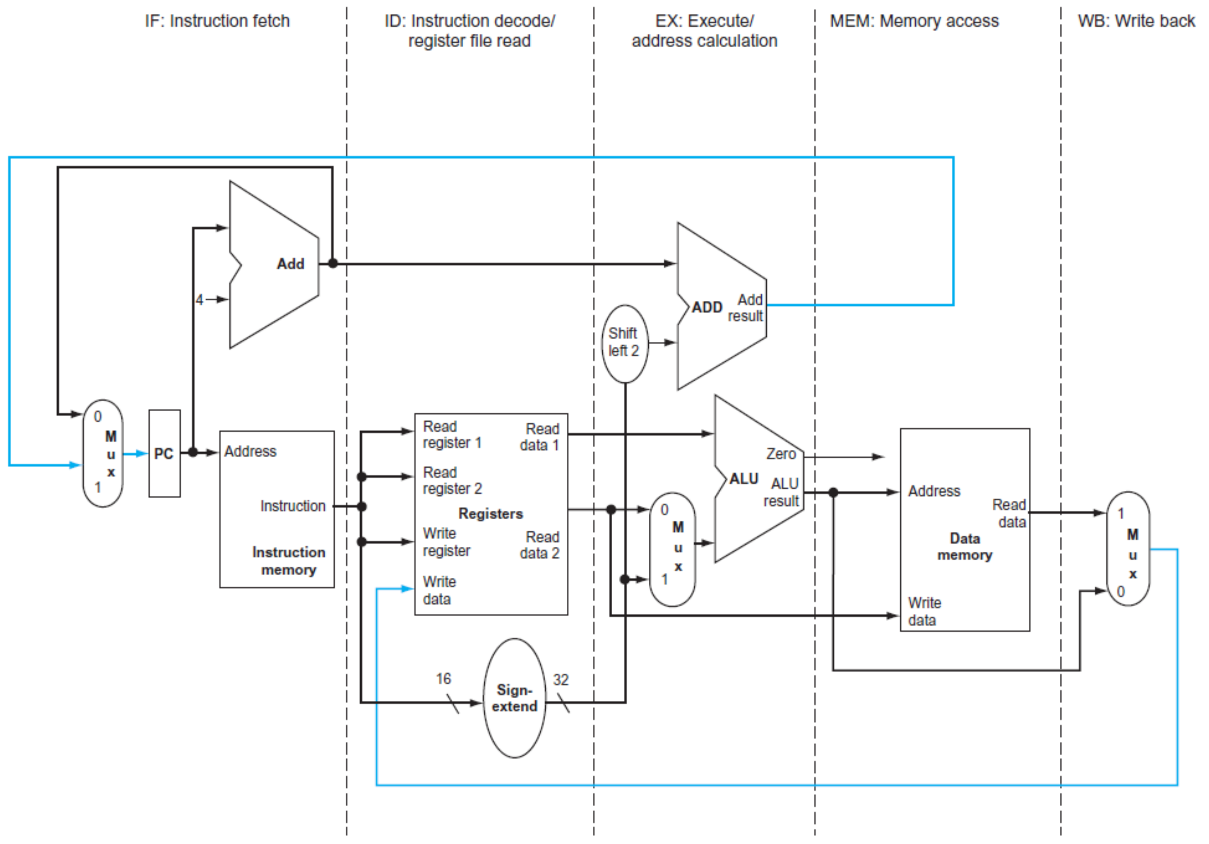


Fig1.1 Single-Cycle Datapath with Five Stages

These five components correspond roughly to the way the data path is drawn; instructions and data move generally from left to right through the five stages as they complete execution. Data flowing from right to left should not affect the current instruction. Namely, what happens in pipelined execution is to pretend that each instruction has its own datapath. To achieve this goal, this program have 5 pipeline registers to save some key information of each instruction so that the instruction can be executed correctly, like the value of Program Counter, Control Signals, outputs of ALU.

However, pipeline technology introduces two major hazards: Data Hazard and Control Hazard. This program will try to solve those two hazards. This program can solve most of the cases. But when the data hazard and control hazard happen together, this program will meet problems, and cannot continue to print out the results, which can be improved.

This project consists of 7 source code files, 1 instruction file, and a test file.

(1) CPU.v: This file is the main module of the project. It instantiates modules from other files and conducts the pipeline design.

(2) ALU.v: It is almost the same as the file in Project 3. I change the input of the module, and some details for some specific instructions.

(3) aluControl.v: It is the same as the file in Project 3, which decides the function of ALU.

(4) control.v: It designs a module that generates the control signals according to the instruction in the ID stage.

(5) Instruction.v: It designs a module that saves the instructions from *ins.txt* file in the simulated memory. The instructions should be written in the ins.txt file. In the appendix of the report, I give some example inputs and outputs. You can change the content in the ins.txt to test my program. I just set 200 blocks for use. Namely, do not attend more than 200 instructions to the txt file. Or you can change the data in this file to have more space to

save the instructions.**If you want to change the location/name of the file, please check the 10th line and change the file name in this file.**

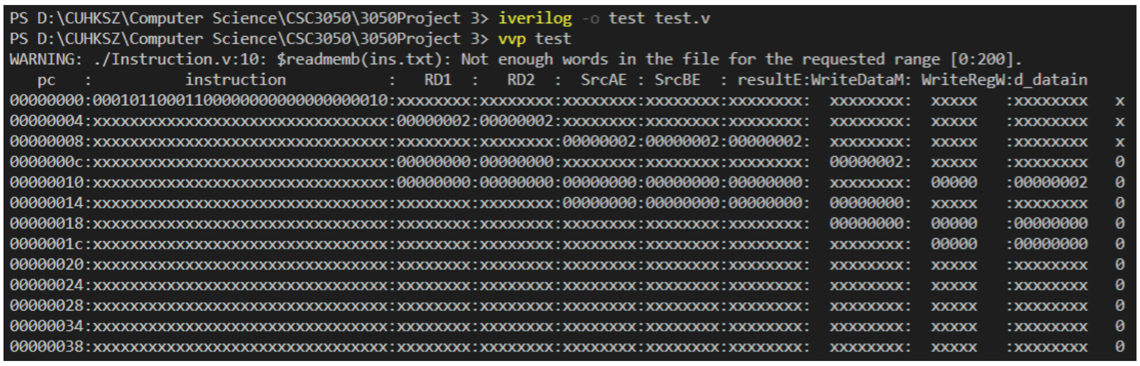
(6) Memory.v: It designs a module that can save the data. It can also output data. The initial data for each address is set to 0. **The maximum word address of it is set to 200.**

(7) Reg.v: It designs a module that simulates the register file. You can get the data according to the serial number for each register. Also, you can change the data for every register except $zero. The initial data for every register (except $zero, $s6, $s7, $ra) is set to 2. $s6 = 32'hFFFF_FFFF, $s7 = 32'h0000_0000, $ra = 32'h0000_0000.

(8) test.v: This file is the test file of this project. The results of it will be introduced in the last part of the report.

(9) ins.txt: This file saves the instructions. And they will be saved into the Instruction Memory when the program is simulated. As a result, I do not write instructions in my test file. I do not give all the instructions that related to the requirement. You can check the appendix to get more inputs and outputs written by myself.

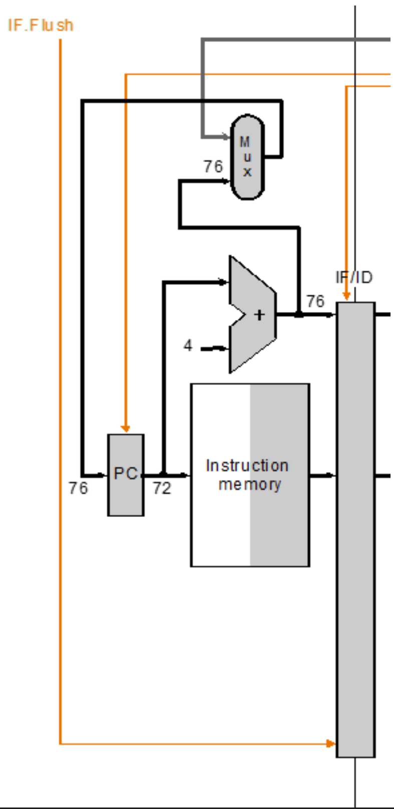
Similar to Proj3, you can still check my project by typing "iverilog -o test test.v" and then type "vvp test" in the terminal.



```
PS D:\CUHKSZ\Computer Science\CSC3050\3050Project 3> iverilog -o test test.v
PS D:\CUHKSZ\Computer Science\CSC3050\3050Project 3> vvp test
WARNING: ./Instruction.v:10: $readmemb(ins.txt): Not enough words in the file for the requested range [0:200].
   pc    :           instruction            :   RD1   :   RD2   :  SrcAE : SrcBE   : resultE:WriteDataM: WriteRegW:d_datain
00000000:00010110001100000000000000000010:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:  xxxxxxxx:  xxxxx   :xxxxxxxx   x
00000004:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:00000002:00000002:xxxxxxxx:xxxxxxxx:xxxxxxxx:  xxxxxxxx:  xxxxx   :xxxxxxxx   x
00000008:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxx:xxxxxxxx:00000002:00000002:00000002:  xxxxxxxx:  xxxxx   :xxxxxxxx   x
0000000c:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:00000000:00000000:xxxxxxxx:xxxxxxxx:xxxxxxxx:  00000002:  xxxxx   :xxxxxxxx   0
00000010:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:00000000:00000000:00000000:00000000:00000000:  xxxxxxxx:  00000   :00000002   0
00000014:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxx:xxxxxxxx:00000000:00000000:00000000:  00000000:  xxxxx   :xxxxxxxx   0
00000018:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:  00000000:  00000   :00000000   0
0000001c:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:  xxxxxxxx:  00000   :00000000   0
00000020:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:  xxxxxxxx:  xxxxx   :xxxxxxxx   0
00000024:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:  xxxxxxxx:  xxxxx   :xxxxxxxx   0
00000028:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:  xxxxxxxx:  xxxxx   :xxxxxxxx   0
00000034:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:  xxxxxxxx:  xxxxx   :xxxxxxxx   0
00000038:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:  xxxxxxxx:  xxxxx   :xxxxxxxx   0
```

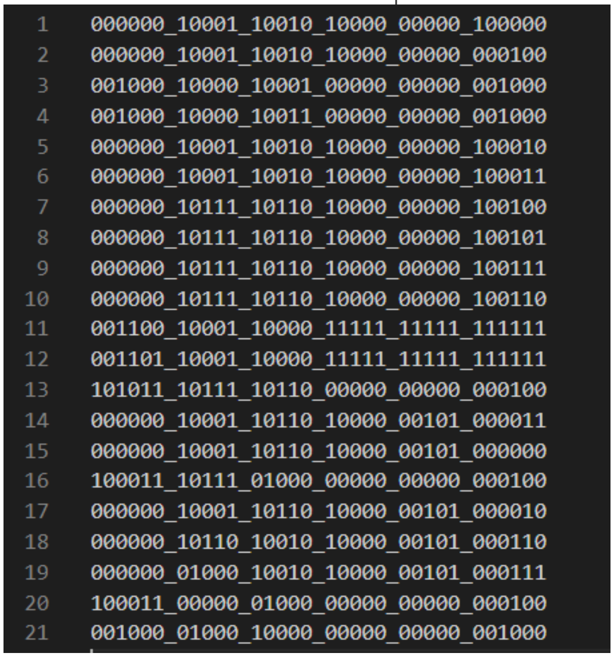
2. DETAILS FOR EVERY STAGE

A. First Stage: Instruction Fetch



In the first stage, this program changes the value of *Program Counter* when the clock reaches its positive edge. Then it will fetch the instruction from the *Instruction Memory* block according to the value of *PC*. After that, the program has a 64-bit register *IFID* to save the result of (pc+4) and the fetched instruction.

Design of Instruction Memory: In the Instruction Memory module, the program will reads the file *ins.txt* to save the instructions in that file. Since each instruction is 32-bit long like the instructions in Fig2.1, this module sets an 32×200 array to simulate the memory that can save at most 200 instructions. The instruction memory module will give the instruction according to *PC* as the byte address.



```
000000_10001_10010_10000_00000_100000
000000_10001_10010_10000_00000_000100
001000_10000_10001_00000_00000_001000
001000_10000_10011_00000_00000_001000
000000_10001_10010_10000_00000_100010
000000_10001_10010_10000_00000_100011
000000_10111_10110_10000_00000_100100
000000_10111_10110_10000_00000_100101
000000_10111_10110_10000_00000_100111
000000_10111_10110_10000_00000_100110
001100_10001_10000_11111_11111_111111
001101_10001_10000_11111_11111_111111
101011_10111_10110_00000_00000_000100
000000_10001_10110_10000_00101_000011
000000_10001_10110_10000_00101_000000
100011_10111_01000_00000_00000_000100
000000_10001_10110_10000_00101_000010
000000_10110_10010_10000_00101_000110
000000_01000_10010_10000_00101_000111
100011_00000_01000_00000_00000_000100
001000_01000_10000_00000_00000_001000
```

Figure 2.1 Content of ins.txt

Design of IFID Pipeline Register: This register has 64bits to save the address of the next instruction in its first 32bits and the last 32bits for the instructions fetched from the *Instruction Memory*.

B. Second Stage: Instruction Decode and Register File Read

In this stage, we set a 185-bit register ***IDEX*** to save the data and control signals.

Like Figure 2.1, we first fetch the instruction *instrD* from ***IFID*** register, and save the address of the next instruction as pc*D* into the IDEX register (IDEX [31:0]). Then we sign extend the immediate part of the instruction to 32 bits as *immD*, and save it into ***IDEX***[63:32] (it will be used for *beq, bne*, I-type instructions).

The next 15 bits [78:74], [73:69], [68:64] save the choice of the three registers ($rs, $rt, $rd). Those three outputs will be used for *lw*, and *jal* instrcutions. Then the next 64 bits [142:111], [110:79] save the data that we get from the *Register File,* which will be used in ALU module.

IDEX[174:143] will save the instruction that will be used in the third stage.
The remaining space will save the control signals that are introduced below.

Design of Control Unit: This module will take a part of instruction as the inputs (opcode, and func code). Then it will generate the control signals that will be passed to the next three stages and saved into pipeline registers. Those signals is so important that they change the way how the CPU deal with those data. Here is a table of control signals and their corresponding functions:

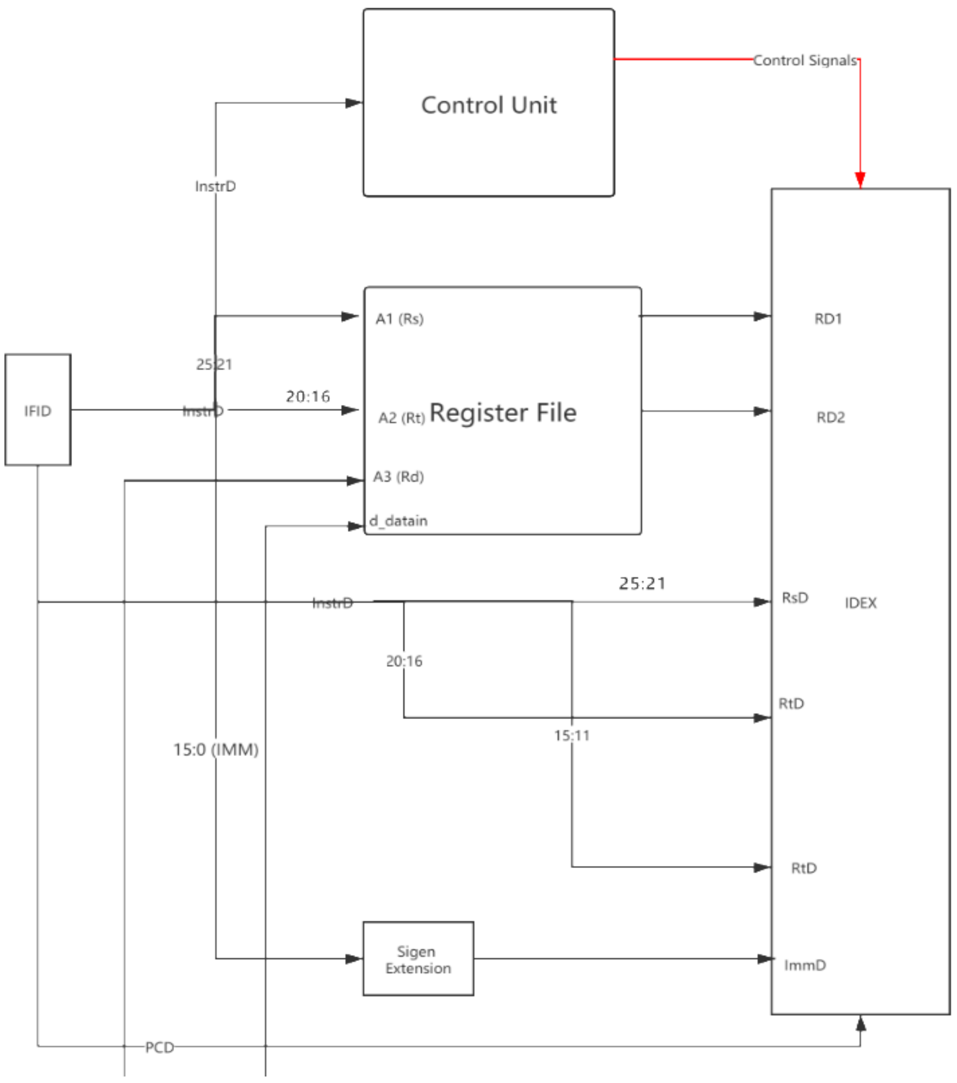


Figure 2.2 Schematic Program of Stage 2

Control Signal	Instructions
RegDstD ([176:175])	This signal sets which register will be written in the WB stage. If it is 2'b10: Choose $rs (jal) 2'b01: Choose $rd (R-type) 2'b00: Choose $rt(I-type)
aluSrcD([177])	This signal chooses the second output of ALU. 0 -> Rt (R-type) 1 -> imm (I-type)
aluOpD([179:178])	This signal chooses the function of ALU that has been introduced in Project 3 report.
BranchD([180])	This signal and the result of ALU (zeroM) will decide whether the branch is taken. 0 -> Not Taken 1 -> Depends on the Result of ALU
MemWriteD([181])	This signal sets whether the data will be saved into the **Data Memory** 0 -> Not Write 1 -> Write
MemtoRegD([182])	This signal decides which result will be written back to the **Register File** 0 -> Result from ALU 1 -> Data From Memory

RegWriteD([183])	This signal decides whether the **Register File** will be changed. 0 -> Not Write 1 -> Write
JumpD([184])	This signal shows this instruction is a jump instruction. It will flush the next instruction and jump to the target address.

Design of Register File: This module takes the instruction as an input and give its data. This module also supports the write of data. The initial data in this module is as follows.

$zero = 0;

$s6 = -1;

$s7 = 0; (Used to check bitwise algorithm instructions)

$ra = 0; (Used to check jr, jal instructions)

C. Third Stage: Execution or address calculation

Figure 2.3 shows the basic idea of this stage. In this part, we deal with the data from ***IDEX*** register. We choose the two inputs of ALU, and choose the address of the register that will be written back in WB stage. We also calculate the branch address for branch instruction. Finally, those results and some control signals will be saved into ***EXMEM*** register that will be used in the next stage.

Note that those control signals all have the suffix "E". Those signals have the same value as the signals generated by **Control Unit** in the ID Stage.

Since we have to deal with *jal* instruction, and data hazard, I add one more bit to RegDstE to make it can choose Rs as the address of register to write back. Also, MemSrcE signal is used to save the value of PC+4 that will be the data written back to $ra.

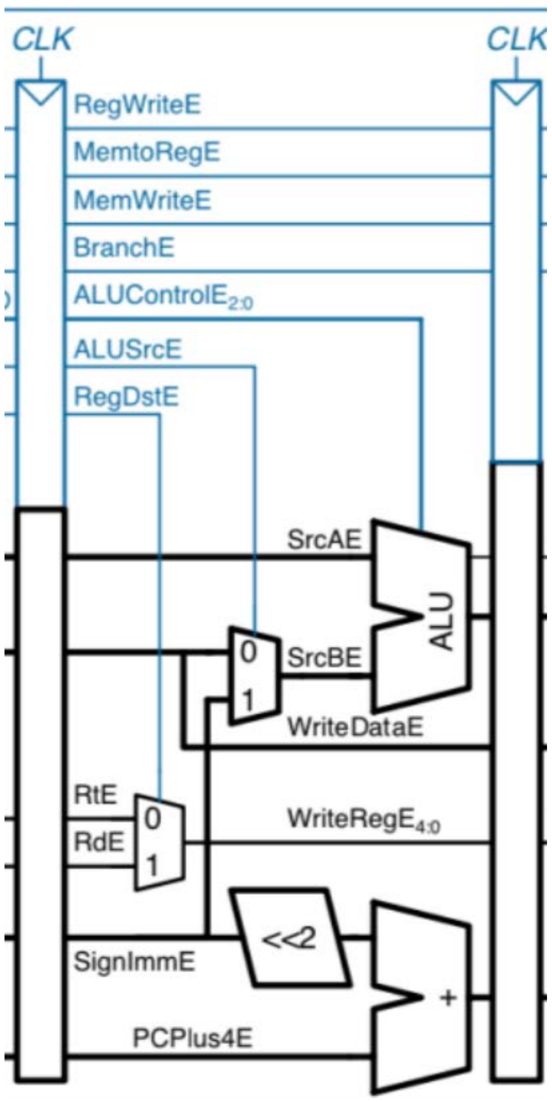


Figure 2.3 Schematic Program of EX Stage

Design of EXMEM Register: This pipeline register still need to save some control signals that will be used in the next two stages. Besides the control signals, this register saves the main result of ALU ([100:69]), overflow flag ([102]), zero flag([101]), neg flag([103]), WriteDataE (Data that will be written to Memory[68:37]), WriteRegE (Address that will be used to write/read Memory[36:32]), and PCBranchE([31:0]) that decides the address of PC if the branch is taken.

Design of ALU:

The design of ALU is almost the same as Proj3, except shift instructions. I changed the detail on how to get the value, and let ALU to read all the instruction, because shamt is not a part of ***IDEX***. This step can be optimized by saving shamt in ***IDEX*****.**

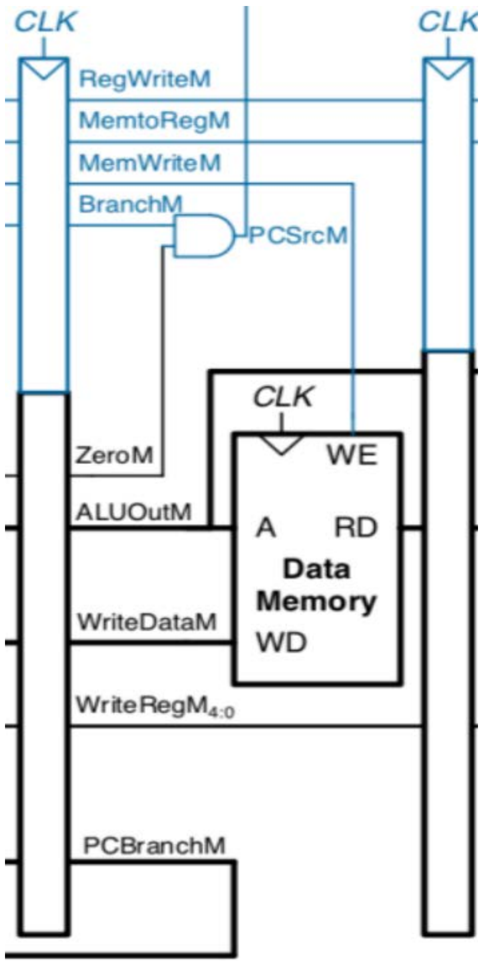
D. Fourth Stage: Data memory access

In this stage, we still read data and control signals from the ***IDEX***. Then we read from/write to the **Data Memory Block**. Also, we need to check whether the branch is taken in this stage according to the zeroM from ALU and BranchM control signal. Finally, we save the result from ALU, the data read from Memory, the address of target register into the 70-bit MEMWB register.

Design of Data Memory:

This module take the target address (resultM from ALU), write enable signal (MemWriteM), and datain(WriteDataM) as inputs. Then it outputs the corresponding data based on the address calculated by ALU.

The initial value for every data block is set to 0.



E. Fifth Stage: Write Back:

In this stage we just write the data back to the register. Here we have two possible choices:

(1) Result calculated by ALU (All instructions except *lw, jal*) (Value of PC for $ra is combined here)

(2) Result from read from Memory (lw instrcution).

Figure 2. 4 Schematic Program of Memory Access Stage

3. DETAILS FOR SOME INSTRUCTIONS AND HAZARDS

(1) Jump Instructions: This instruction will cause **Control Hazard** because some instructions should be not conducted. The decision of whether to jump is decided in the second **ID** stage. As a result, we only need to stall the next instruction, then we will jump to the target address by changing PC. I set the ***IFID*** register to be 0 to stall the instruction. Although my program will still print out the result for the stalled instruction, this process will not change any data in Memory or Register.

(2) Branch Instructions: Those instructions will also result in **Control Hazard**. This program takes the same data flow from Figure 3.1. Namely, the branch decision is done in the fourth stage. As a result, I have to stall the instructions for three times by maintaining the PC and keep the value of *IFID* register to be 0 until the branch decision is done. (Please Check the example 5 in Appendix)

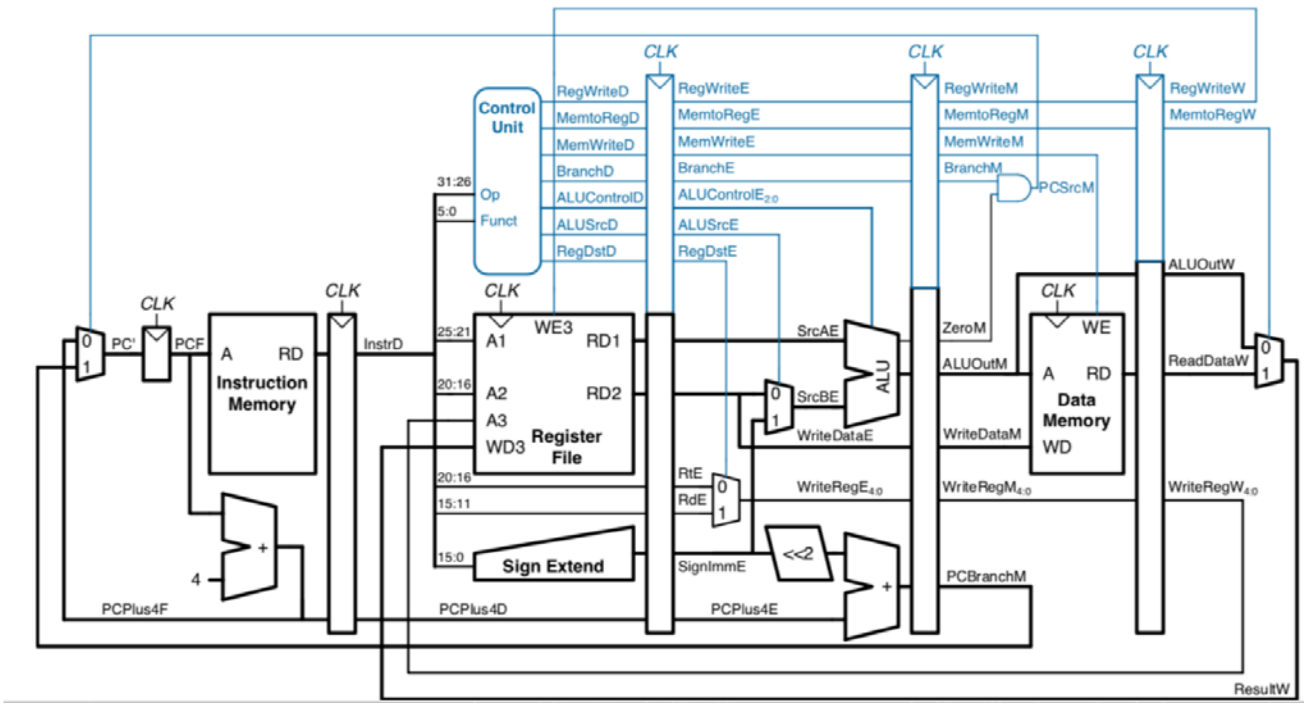
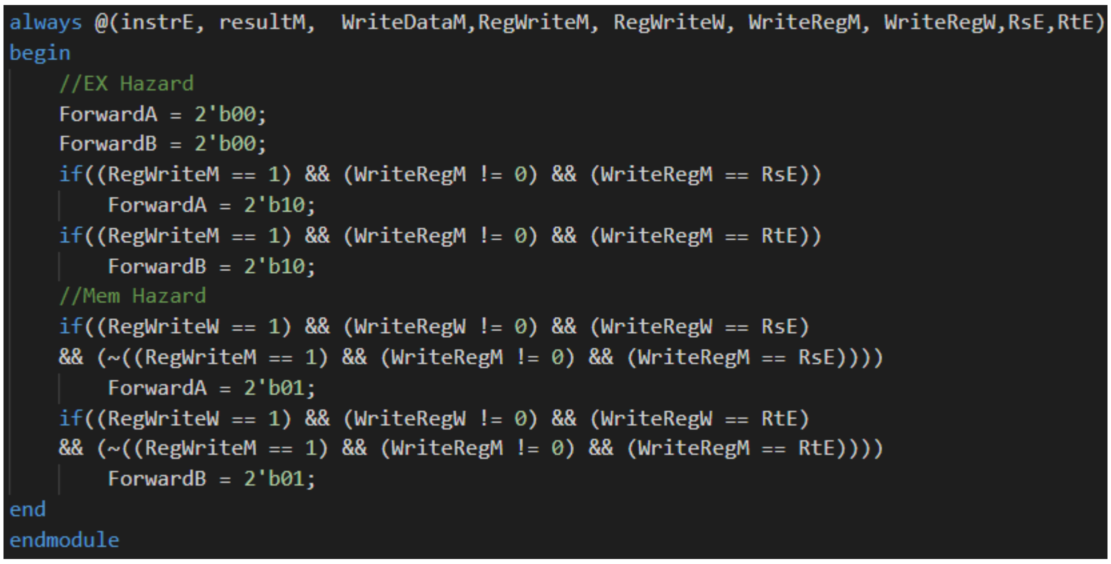


Figure 3.1 Dataflow for Pipeline Register

(3) Data Hazard: This program can handle the EX hazard, Mem Hazard, load-use Hazar. It checks whether the instructions need to be stalled or whether the data should

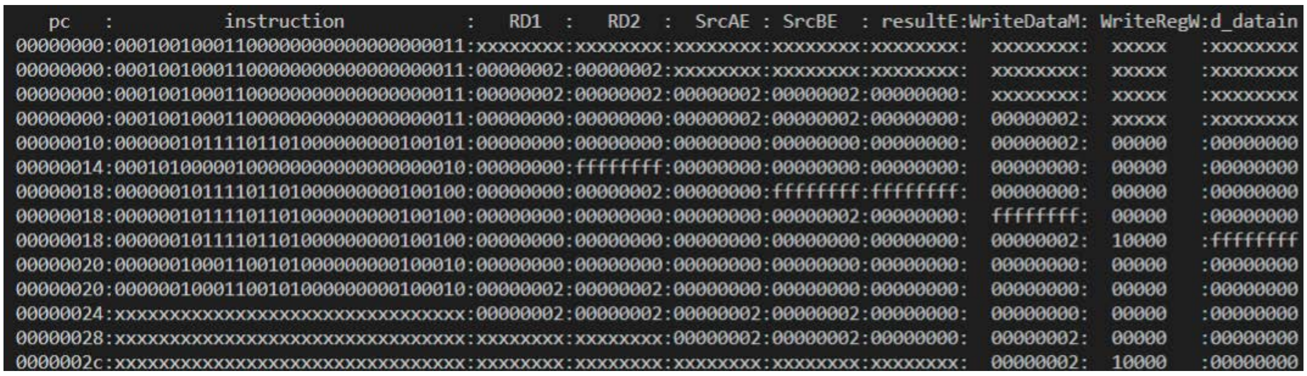
be forwarded by checking the control signals, registers that are used. (Please check the example 1 in appendix).



```verilog
always @(instrE, resultM,  WriteDataM,RegWriteM, RegWriteW, WriteRegM, WriteRegW,RsE,RtE)
begin
    //EX Hazard
    ForwardA = 2'b00;
    ForwardB = 2'b00;
    if((RegWriteM == 1) && (WriteRegM != 0) && (WriteRegM == RsE))
        ForwardA = 2'b10;
    if((RegWriteM == 1) && (WriteRegM != 0) && (WriteRegM == RtE))
        ForwardB = 2'b10;
    //Mem Hazard
    if((RegWriteW == 1) && (WriteRegW != 0) && (WriteRegW == RsE)
    && (~((RegWriteM == 1) && (WriteRegM != 0) && (WriteRegM == RsE))))
        ForwardA = 2'b01;
    if((RegWriteW == 1) && (WriteRegW != 0) && (WriteRegW == RtE)
    && (~((RegWriteM == 1) && (WriteRegM != 0) && (WriteRegM == RtE))))
        ForwardB = 2'b01;
end
endmodule
```

4. RESULTS

The result of this project is presented by the *test.v* test file. It will show the following results. Usually, you will get the result in the shape of triangle.



```
   pc    :            instruction           :   RD1   :   RD2   :  SrcAE  : SrcBE   : resultE:WriteDataM: WriteRegW:d_datain
00000000:00010010001100000000000000000011:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:  xxxxxxxx:  xxxxx    :xxxxxxxx
00000000:00010010001100000000000000000011:00000002:00000002:xxxxxxxx:xxxxxxxx:xxxxxxxx:  xxxxxxxx:  xxxxx    :xxxxxxxx
00000000:00010010001100000000000000000011:00000002:00000002:00000002:00000002:00000000:  xxxxxxxx:  xxxxx    :xxxxxxxx
00000000:00010010001100000000000000000011:00000000:00000000:00000002:00000002:00000000:  00000002:  xxxxx    :xxxxxxxx
00000010:00000010111101101000000000100101:00000000:00000000:00000000:00000000:00000000:  00000002:  00000    :00000000
00000014:00010100000100000000000000000010:00000000:ffffffff:00000000:00000000:00000000:  00000000:  00000    :00000000
00000018:00000010111101101000000000100100:00000000:00000002:00000000:ffffffff:ffffffff:  00000000:  00000    :00000000
00000018:00000010111101101000000000100100:00000000:00000000:00000000:00000002:00000000:  ffffffff:  00000    :00000000
00000018:00000010111101101000000000100100:00000000:00000000:00000000:00000000:00000000:  00000002:  10000    :ffffffff
00000020:00000010001100101000000000100010:00000000:00000000:00000000:00000000:00000000:  00000000:  00000    :00000000
00000020:00000010001100101000000000100010:00000002:00000002:00000000:00000000:00000000:  00000000:  00000    :00000000
00000024:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:00000002:00000002:00000002:00000002:00000000:  00000000:  00000    :00000000
00000028:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxx:xxxxxxxx:00000002:00000002:00000000:  00000002:  00000    :00000000
0000002c:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:  00000002:  10000    :00000000
```

First Stage:

(1) Program Counter

(2) Instruction: This instruction is got from the **Instruction Memory** module based on the value of Program Counter. If PC is changed, instruction will also change.

Second Stage:

(3)RD1: The first output from **Register File.** Usually, it is the value of $rs.

(4)RD2: The second output from **Register File.** Usually, it is the value of $rt.

Third Stage:

(5)SrcAE: This is the first input from ***IDEX*** register, or it may be the forwarded result from the proceeding instructions.

(6)SrcBE: This is the second input from ***IDEX*** register, or it may be the forwarded result from the proceeding instructions. It may also be the value of immediate part for I-type instructions.

(7)ResultE: This is the output of ALU Module.

Fourth Stage:

(8)WriteDataM: This is the data that will be written to the Memory. In the most cases, it has the same value as the register $rt. I also use it to save the value of PC for *jal* instruction, to save some space for the pipeline registers.

Fifth Stage:

(9)WriteRegW: This is the target address of Register File. Usually it is $rd for R-type instructions and $rt for I-type instructions. If RegWriteW = 1, the value of the register with this address will be changed.

(10)d_datain: This is the value that will finally be written to the register. Usually it have two choices: results from ALU or data from Memory.

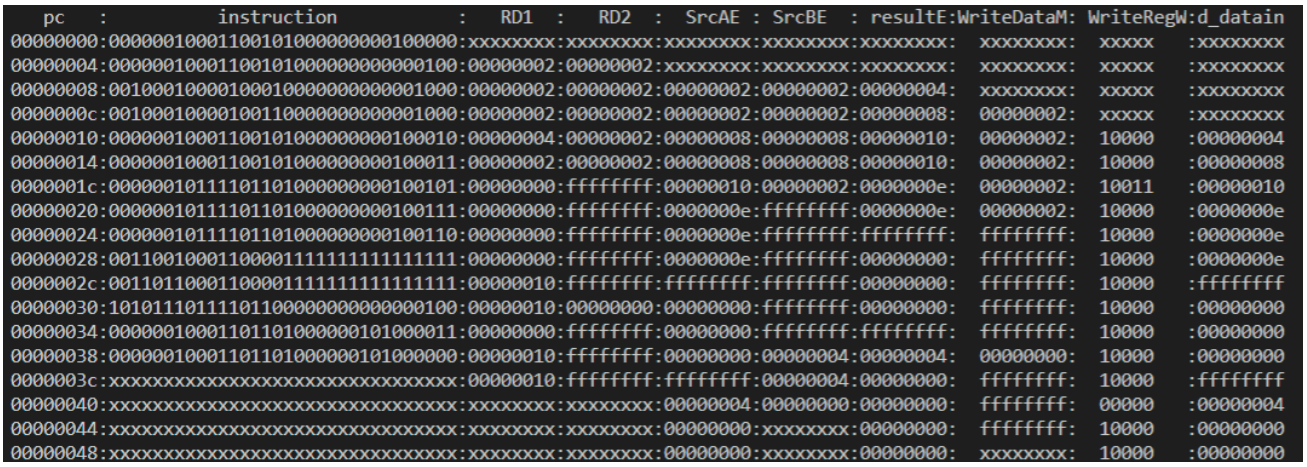
APPENDIX: TEST EXAMPLE

Since *.txt* file does not support notations, this appendix saves the Machine Code in the *ins.txt* file and their corresponding MIPS instructions. Also, it has the anticipated output for each instruction. Also, it has the outputs generated by my program.

TEST EXAMPLE 1 (BASIC ALGORITHM INSTRUCTION WITH DATA HAZARD)

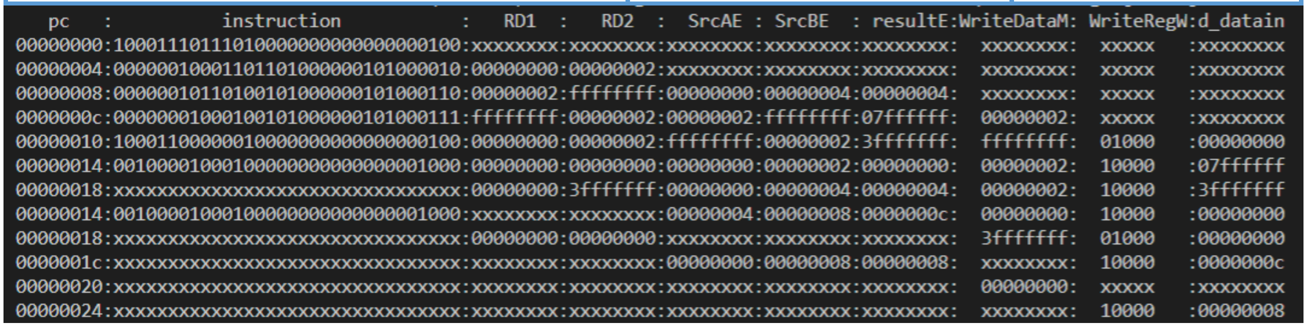
Machine Code	MIPS Instruction	Anticipated Output (in hexdecimal)
000000_10001_10010_10000_00000_100000	add $s0 $s1 $s2	00000004
000000_10001_10010_10000_00000_000100	sllv $s0 $s1 $s2	00000008
001000_10000_10001_00000_00000_001000	addi $s1 $s0 8	00000010
001000_10000_10011_00000_00000_001000	addi $s3, $s0, 8	00000010
000000_10001_10010_10000_00000_100010	sub $s0, $s1, $s2	0000000e
000000_10001_10010_10000_00000_100011	subu $s0, $s1, $s2	0000000e
000000_10111_10110_10000_00000_100100	and $s0, $s7, $s6	00000000
000000_10111_10110_10000_00000_100101	or $s0, $s7, $s6	ffffffff
000000_10111_10110_10000_00000_100111	nor $s0, $s7, $s6	00000000
000000_10111_10110_10000_00000_100110	xor $s0, $s7, $s6	ffffffff

001100_10001_10000_11111_11111_111111	andi $s0, $s1, -1	00000010
001101_10001_10000_11111_11111_111111	ori $s0, $s1, -1	ffffffff
101011_10111_10110_00000_00000_000100	sw $s6, $s7, 4	00000004
000000_10001_10110_10000_00101_000011	sra $s0, $s6, 5	ffffffff
000000_10001_10110_10000_00101_000000	sll $s0, $s6,5	ffffffe0


```
   pc    :            instruction            :   RD1   :   RD2   :  SrcAE  : SrcBE   : resultE:WriteDataM: WriteRegW:d_datain
00000000:00000010001100101000000000100000:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:  xxxxxxxx:  xxxxx   :xxxxxxxx
00000004:00000010001100101000000000000100:00000002:00000002:xxxxxxxx:xxxxxxxx:xxxxxxxx:  xxxxxxxx:  xxxxx   :xxxxxxxx
00000008:00100010000100010000000000001000:00000002:00000002:00000002:00000002:00000004:  xxxxxxxx:  xxxxx   :xxxxxxxx
0000000c:00100010000100110000000000001000:00000002:00000002:00000002:00000002:00000008:  00000002:  xxxxx   :xxxxxxxx
00000010:00000010001100101000000000100010:00000004:00000002:00000008:00000008:00000010:  00000002:  10000   :00000004
00000014:00000010001100101000000000100011:00000002:00000002:00000008:00000008:00000010:  00000002:  10000   :00000008
0000001c:00000010111101101000000000100101:00000000:ffffffff:00000010:00000002:0000000e:  00000002:  10011   :00000010
00000020:00000010111101101000000000100111:00000000:ffffffff:0000000e:ffffffff:0000000e:  00000002:  10000   :0000000e
00000024:00000010111101101000000000100110:00000000:ffffffff:0000000e:ffffffff:ffffffff:  ffffffff:  10000   :0000000e
00000028:00110010001100001111111111111111:00000000:ffffffff:0000000e:ffffffff:00000000:  ffffffff:  10000   :0000000e
0000002c:00110110001100001111111111111111:00000010:ffffffff:ffffffff:ffffffff:00000000:  ffffffff:  10000   :ffffffff
00000030:10101110111101100000000000000100:00000010:00000000:00000000:ffffffff:00000000:  ffffffff:  10000   :00000000
00000034:00000010001101101000000101000011:00000000:ffffffff:00000000:ffffffff:ffffffff:  ffffffff:  10000   :00000000
00000038:00000010001101101000000101000000:00000010:ffffffff:00000000:00000004:00000004:  00000000:  10000   :00000000
0000003c:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:00000010:ffffffff:ffffffff:00000004:00000000:  ffffffff:  10000   :ffffffff
00000040:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxx:xxxxxxxx:00000004:00000000:00000000:  ffffffff:  00000   :00000004
00000044:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxx:xxxxxxxx:00000000:xxxxxxxx:00000000:  ffffffff:  10000   :00000000
00000048:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxx:xxxxxxxx:00000000:xxxxxxxx:00000000:  xxxxxxxx:  10000   :00000000
```

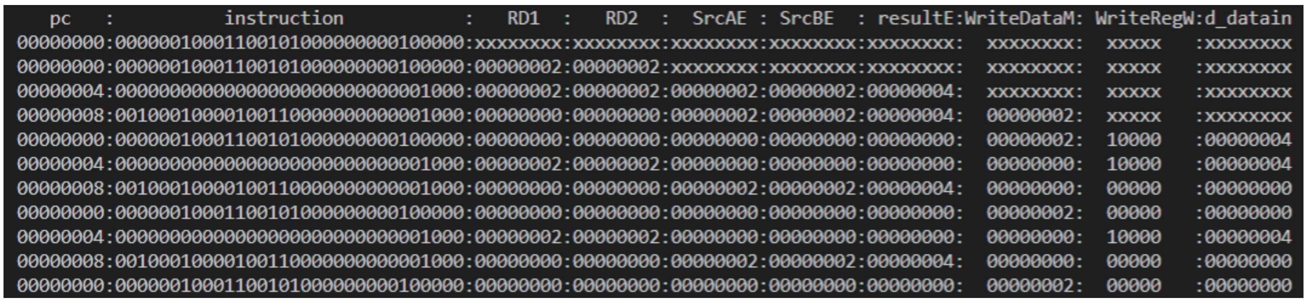
TEST EXAMPLE 2(LW INSTRUCTION, DATA HAZARD, AND STALL)

100011_10111_01000_00000_00000_000100	lw $t0, 4($s7)	00000004
000000_10001_10110_10000_00101_000010	srl $s0, $s6, 5	07ffffff
000000_10110_10010_10000_00101_000110	srlv $s0, $s6, $s2	3fffffff
000000_01000_10010_10000_00101_000111	srav $s0, $t0, $s2	ffffffff
100011_00000_01000_00000_00000_000100	lw $t0, 4($zero)	00000004
001000_01000_10000_00000_00000_001000	addi $s0, $t0, 8	00000008


```
   pc    :           instruction            :   RD1   :   RD2   :  SrcAE  : SrcBE   : resultE:WriteDataM: WriteRegW:d_datain
 00000000:10001110111010000000000000000100:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:  xxxxxxxx:  xxxxx   :xxxxxxxx
 00000004:00000010001101101000000101000010:00000000:00000002:xxxxxxxx:xxxxxxxx:xxxxxxxx:  xxxxxxxx:  xxxxx   :xxxxxxxx
 00000008:00000010110100101000000101000110:00000002:ffffffff:00000000:00000004:00000004:  xxxxxxxx:  xxxxx   :xxxxxxxx
 0000000c:00000001000100101000000101000111:ffffffff:00000002:00000002:ffffffff:07ffffff:  00000002:  xxxxx   :xxxxxxxx
 00000010:10001100000010000000000000000100:00000000:00000002:ffffffff:00000002:3fffffff:  ffffffff:  01000   :00000000
 00000014:00100001000100000000000000001000:00000000:00000000:00000000:00000002:00000000:  00000002:  10000   :07ffffff
 00000018:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:00000000:3fffffff:00000000:00000004:00000004:  00000002:  10000   :3fffffff
 00000014:00100001000100000000000000001000:xxxxxxxx:xxxxxxxx:00000004:00000008:0000000c:  00000000:  10000   :00000000
 00000018:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:00000000:00000000:xxxxxxxx:xxxxxxxx:xxxxxxxx:  3fffffff:  01000   :00000000
 0000001c:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxx:xxxxxxxx:00000000:00000008:00000008:  xxxxxxxx:  10000   :0000000c
 00000020:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:  00000000:  xxxxx   :xxxxxxxx
 00000024:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:  xxxxxxxx:  10000   :00000008
```

TEST EXAMPLE 3 (FOR JR INSTRUCTION)

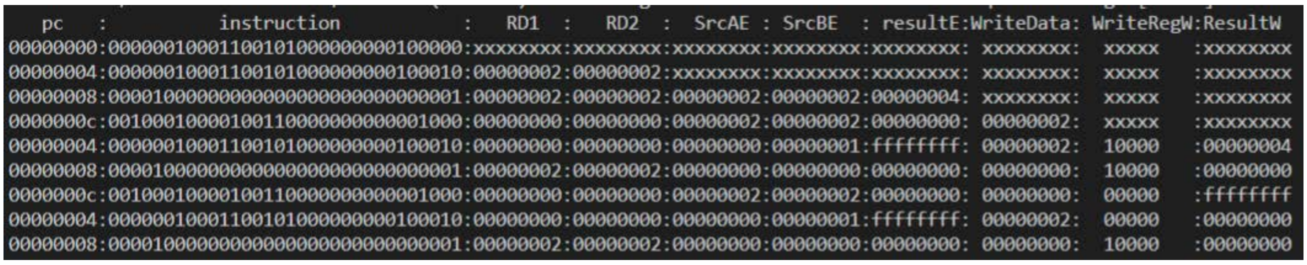
Machine Code	MIPS Instruction	Anticipated Output (in hexdecimal)
000000_10001_10010_10000_00000_100000	add $s0 $s1 $s2	00000004
000000_00000_00000_00000_00000_001000	jr, $zero	NULL
001000_10000_10011_00000_00000_001000	addi $s1 $s0 8	FLUSH



```
    pc    :            instruction             :   RD1   :   RD2   :  SrcAE : SrcBE  : resultE:WriteDataM: WriteRegW:d_datain
 00000000:00000010001100101000000000100000:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:  xxxxxxxx:  xxxxx   :xxxxxxxx
 00000000:00000010001100101000000000100000:00000002:00000002:xxxxxxxx:xxxxxxxx:xxxxxxxx:  xxxxxxxx:  xxxxx   :xxxxxxxx
 00000004:00000000000000000000000000001000:00000002:00000002:00000002:00000002:00000004:  xxxxxxxx:  xxxxx   :xxxxxxxx
 00000008:00100010000100110000000000001000:00000000:00000000:00000002:00000002:00000004:  00000002:  xxxxx   :xxxxxxxx
 00000000:00000010001100101000000000100000:00000000:00000000:00000000:00000000:00000000:  00000002:  10000   :00000004
 00000004:00000000000000000000000000001000:00000002:00000002:00000000:00000000:00000000:  00000000:  10000   :00000004
 00000008:00100010000100110000000000001000:00000000:00000000:00000002:00000002:00000004:  00000000:  00000   :00000000
 00000000:00000010001100101000000000100000:00000000:00000000:00000000:00000000:00000000:  00000002:  00000   :00000000
 00000004:00000000000000000000000000001000:00000002:00000002:00000000:00000000:00000000:  00000000:  10000   :00000004
 00000008:00100010000100110000000000001000:00000000:00000000:00000002:00000002:00000004:  00000000:  00000   :00000000
 00000000:00000010001100101000000000100000:00000000:00000000:00000000:00000000:00000000:  00000002:  00000   :00000000
```

TEST EXAMPLE 4 (FOR J INSTRUCTION WITH CONTROL HAZARD)

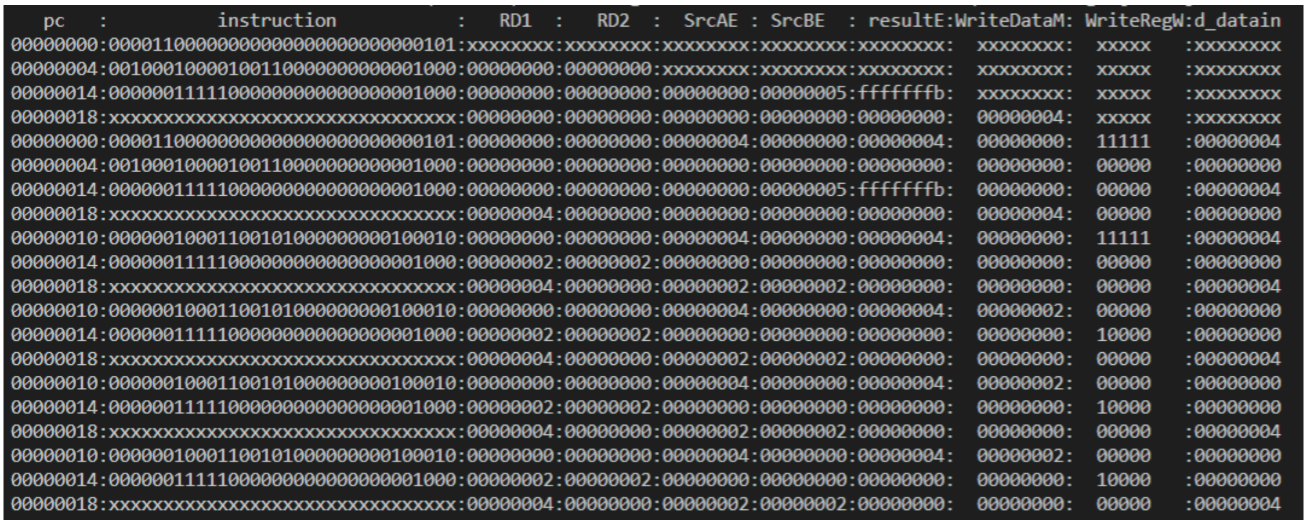
Machine Code	MIPS Instruction	Anticipated Output (in hexdecimal)
000000_10001_10010_10000_00000_100000	add $s0 $s1 $s2	00000004
000000_10001_10010_10000_00000_100010	sub $s0, $s1, $s2	00000000
000010_00000_00000_00000_00000_000001	j,1	NULL
001000_10000_10011_00000_00000_001000	addi $s1 $s0 8	FLUSH



```
   pc    :          instruction            :   RD1   :   RD2   :  SrcAE : SrcBE   : resultE:WriteData: WriteRegW:ResultW
00000000:00000010001100101000000000100000:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx: xxxxxxxx:  xxxxx   :xxxxxxxx
00000004:00000010001100101000000000100010:00000002:00000002:xxxxxxxx:xxxxxxxx:xxxxxxxx: xxxxxxxx:  xxxxx   :xxxxxxxx
00000008:00001000000000000000000000000001:00000002:00000002:00000002:00000002:00000004: xxxxxxxx:  xxxxx   :xxxxxxxx
0000000c:00100010000100110000000000001000:00000000:00000000:00000002:00000002:00000000: 00000002:  xxxxx   :xxxxxxxx
00000004:00000010001100101000000000100010:00000000:00000000:00000000:00000001:ffffffff: 00000002:  10000   :00000004
00000008:00001000000000000000000000000001:00000002:00000002:00000000:00000000:00000000: 00000000:  10000   :00000000
0000000c:00100010000100110000000000001000:00000000:00000000:00000002:00000002:00000000: 00000000:  00000   :ffffffff
00000004:00000010001100101000000000100010:00000000:00000000:00000000:00000001:ffffffff: 00000002:  00000   :00000000
00000008:00001000000000000000000000000001:00000002:00000002:00000000:00000000:00000000: 00000000:  10000   :00000000
```

TEST EXAMPLE 4 (FOR $JR AND $JAL INSTRUCTIONS WITH CONTROL HAZARD)

Machine Code	MIPS Instruction	Anticipated Output (in hexdecimal)
000011_00000_00000_00000_00000_000101	jal, 5	Doesn't Matter
001000_10000_10011_00000_00000_001000	addi $s1 $s0 8	FLUSH
001000_10000_10001_00000_00000_001111	addi $s1 $s0 15	00000011
000000_10111_10110_10000_00000_100100	and $s0, $s7, $s6	00000000
000000_10001_10010_10000_00000_100010	sub $s0, $s1, $s2	00000000
000000_11111_00000_00000_00000_001000	jr $ra	Doesn't matter

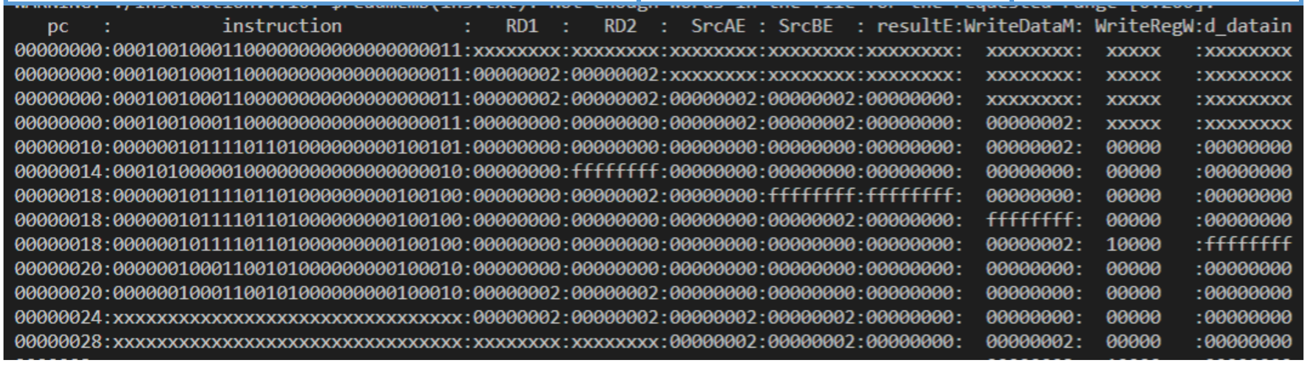


```
   pc    :           instruction           :   RD1   :   RD2   :  SrcAE  : SrcBE   : resultE:WriteDataM: WriteRegW:d_datain
00000000:00001100000000000000000000000101:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:  xxxxxxxx:  xxxxx   :xxxxxxxx
00000004:00100010000100110000000000001000:00000000:00000000:xxxxxxxx:xxxxxxxx:xxxxxxxx:  xxxxxxxx:  xxxxx   :xxxxxxxx
00000014:00000011111000000000000000001000:00000000:00000000:00000000:00000005:fffffffb:  xxxxxxxx:  xxxxx   :xxxxxxxx
00000018:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:00000000:00000000:00000000:00000000:00000000:  00000004:  xxxxx   :xxxxxxxx
00000000:00001100000000000000000000000101:00000000:00000000:00000004:00000000:00000004:  00000000:  11111   :00000004
00000004:00100010000100110000000000001000:00000000:00000000:00000000:00000000:00000000:  00000000:  00000   :00000000
00000014:00000011111000000000000000001000:00000000:00000000:00000000:00000005:fffffffb:  00000000:  00000   :00000004
00000018:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:00000004:00000000:00000000:00000000:00000000:  00000004:  00000   :00000000
00000010:00000010001100101000000000100010:00000000:00000000:00000004:00000000:00000004:  00000000:  11111   :00000004
00000014:00000011111000000000000000001000:00000002:00000002:00000000:00000000:00000000:  00000000:  00000   :00000000
00000018:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:00000004:00000000:00000002:00000002:00000000:  00000000:  00000   :00000004
00000010:00000010001100101000000000100010:00000000:00000000:00000004:00000000:00000004:  00000002:  00000   :00000000
00000014:00000011111000000000000000001000:00000002:00000002:00000000:00000000:00000000:  00000000:  10000   :00000000
00000018:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:00000004:00000000:00000002:00000002:00000000:  00000000:  00000   :00000004
00000010:00000010001100101000000000100010:00000000:00000000:00000004:00000000:00000004:  00000002:  00000   :00000000
00000014:00000011111000000000000000001000:00000002:00000002:00000000:00000000:00000000:  00000000:  10000   :00000000
00000018:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:00000004:00000000:00000002:00000002:00000000:  00000000:  00000   :00000004
00000010:00000010001100101000000000100010:00000000:00000000:00000004:00000000:00000004:  00000002:  00000   :00000000
00000014:00000011111000000000000000001000:00000002:00000002:00000000:00000000:00000000:  00000000:  10000   :00000000
00000018:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:00000004:00000000:00000002:00000002:00000000:  00000000:  00000   :00000004
```

TEST EXAMPLE 5 (FOR BEQ, BNE AND CONTROL HAZARD)

Machine Code	MIPS Instruction	Anticipated Output (in hexdecimal)
000100_10001_10000_00000_00000_000011	beq $s0, $s1, 3	PC = 00000014
001000_10000_10011_00000_00000_001000	addi $s1 $s0 8	FLUSH
001000_10000_10001_00000_00000_001111	addi $s1 $s0 15	FLUSH
000000_10111_10110_10000_00000_100100	and $s0, $s7, $s6	FLUSH
000000_10111_10110_10000_00000_100101	or $s0, $s7, $s6	11111111
000101_00000_10000_00000_00000_000010	bne $s0, $zero, 2	PC = 00000018
000000_10111_10110_10000_00000_100100	and $s0, $s7, $s6	FLUSH
000000_10001_10010_10000_00000_100000	add $s0 $s1 $s2	00000004

000000_10001_10010_10000_00000_100010	sub $s0, $s1, $s2	00000000



```
   pc    :            instruction            :   RD1   :   RD2   :  SrcAE  : SrcBE   : resultE:WriteDataM: WriteRegW:d_datain
00000000:00010010001100000000000000000011:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:  xxxxxxxx:  xxxxx   :xxxxxxxx
00000000:00010010001100000000000000000011:00000002:00000002:xxxxxxxx:xxxxxxxx:xxxxxxxx:  xxxxxxxx:  xxxxx   :xxxxxxxx
00000000:00010010001100000000000000000011:00000002:00000002:00000002:00000002:00000000:  xxxxxxxx:  xxxxx   :xxxxxxxx
00000000:00010010001100000000000000000011:00000000:00000000:00000002:00000002:00000000:  00000002:  xxxxx   :xxxxxxxx
00000010:00000010111110110100000000100101:00000000:00000000:00000000:00000000:00000000:  00000002:  00000   :00000000
00000014:00010100000100000000000000000010:00000000:ffffffff:00000000:00000000:00000000:  00000000:  00000   :00000000
00000018:00000010111110110100000000100100:00000000:00000002:00000000:ffffffff:ffffffff:  00000000:  00000   :00000000
00000018:00000010111110110100000000100100:00000000:00000000:00000000:00000002:00000000:  ffffffff:  00000   :00000000
00000018:00000010111110110100000000100100:00000000:00000000:00000000:00000000:00000000:  00000002:  10000   :ffffffff
00000020:00000010001100101000000000100010:00000000:00000000:00000000:00000000:00000000:  00000000:  00000   :00000000
00000020:00000010001100101000000000100010:00000002:00000002:00000000:00000000:00000000:  00000000:  00000   :00000000
00000024:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:00000002:00000002:00000002:00000002:00000000:  00000000:  00000   :00000000
00000028:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxx:xxxxxxxx:00000002:00000002:00000000:  00000002:  00000   :00000000
```