CprE 381: Computer Organization and Assembly-Level Programming

Project Part 2 Report

Team Members: _________________Alek Norris____

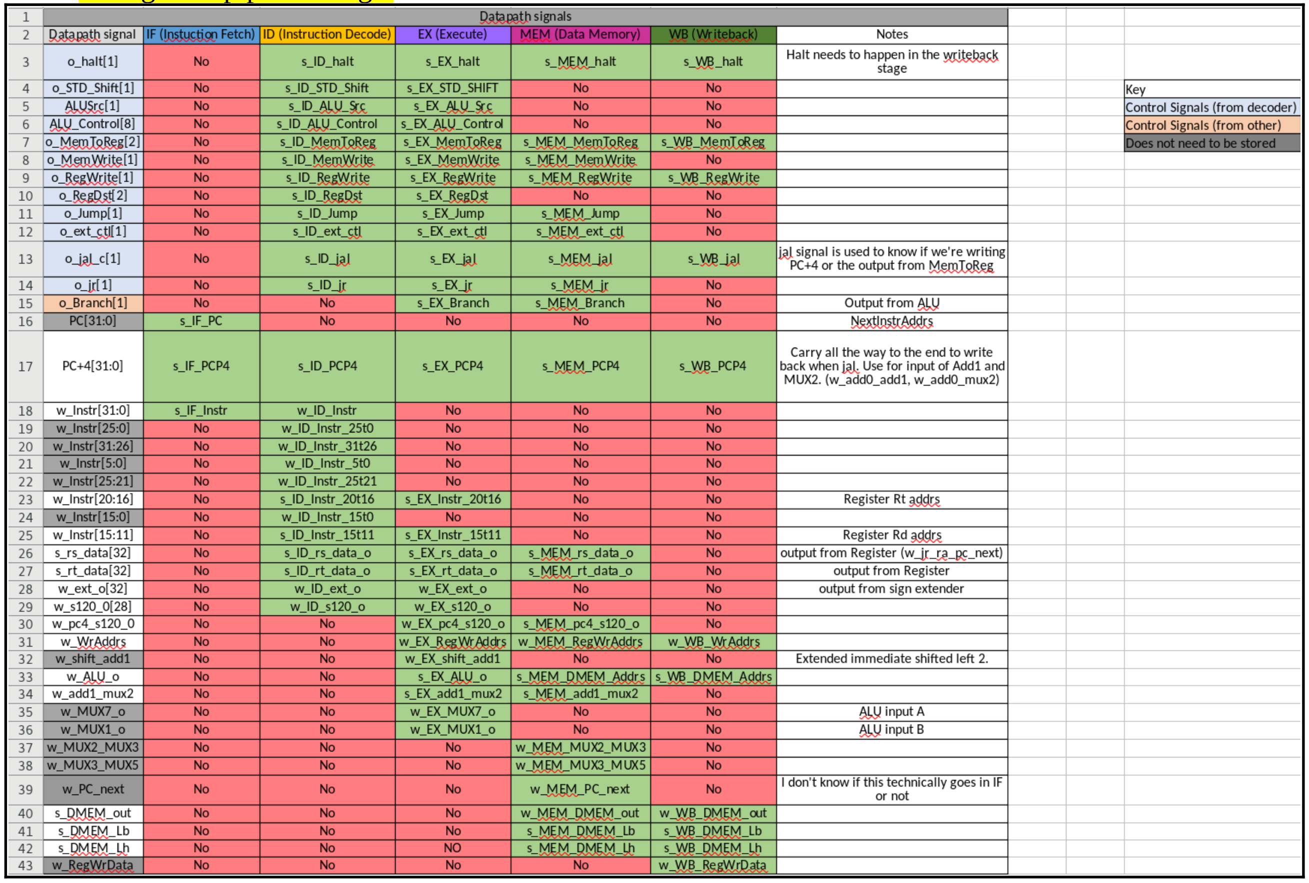
______________________Drew Kearns_________________

____

Project Teams Group #:_________Term Proj1_2s_07_______

Refer to the highlighted language in the project 1 instruction for the context of the following questions.

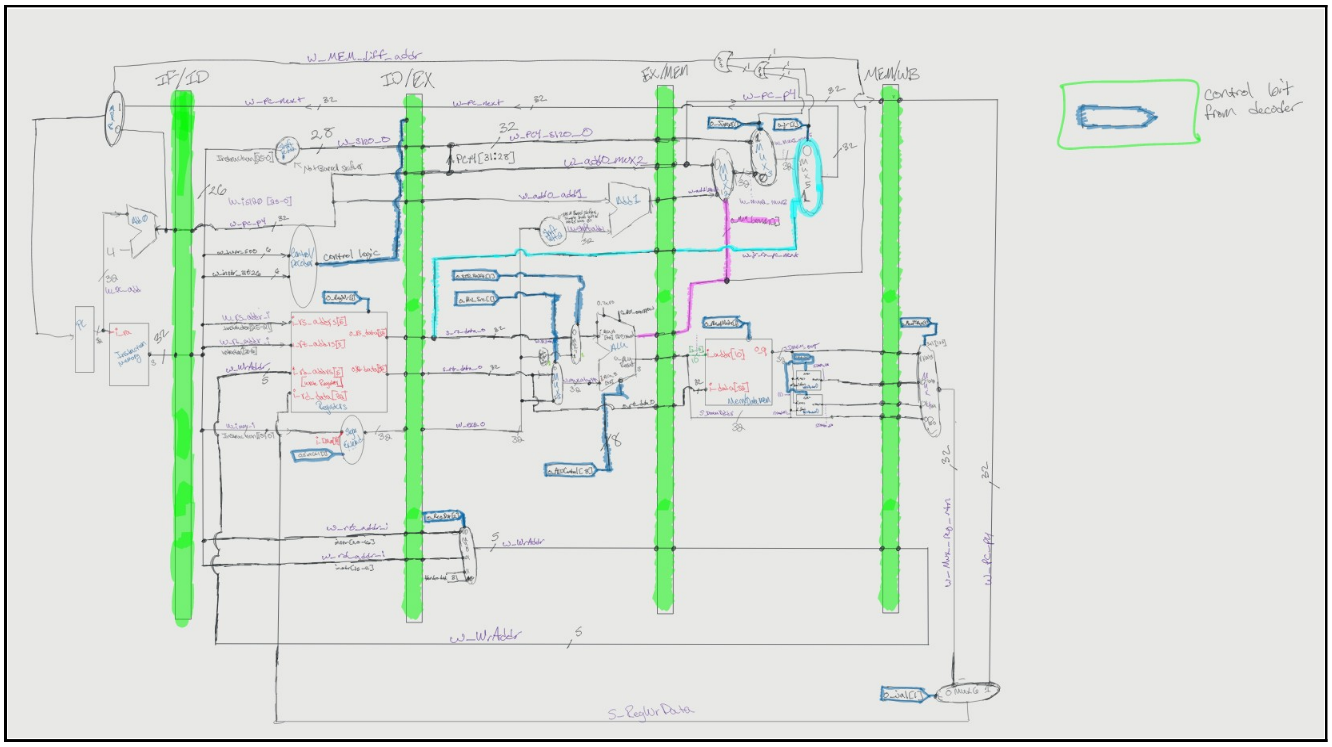
[1.a] Come up with a global list of the datapath values and control signals that are required during each pipeline stage.



1	Datapath signals						Notes
2	Datapath signal	IF (Instuction Fetch)	ID (Instruction Decode)	EX (Execute)	MEM (Data Memory)	WB (Writeback)	Notes
3	o_halt[1]	No	s_ID_halt	s_EX_halt	s_MEM_halt	s_WB_halt	Halt needs to happen in the writeback stage
4	o_STD_Shift[1]	No	s_ID_STD_Shift	s_EX_STD_SHIFT	No	No	
5	ALUSrc[1]	No	s_ID_ALU_Src	s_EX_ALU_Src	No	No	
6	ALU_Control[8]	No	s_ID_ALU_Control	s_EX_ALU_Control	No	No	
7	o_MemToReg[2]	No	s_ID_MemToReg	s_EX_MemToReg	s_MEM_MemToReg	s_WB_MemToReg	
8	o_MemWrite[1]	No	s_ID_MemWrite	s_EX_MemWrite	s_MEM_MemWrite	No	
9	o_RegWrite[1]	No	s_ID_RegWrite	s_EX_RegWrite	s_MEM_RegWrite	s_WB_RegWrite	
10	o_RegDst[2]	No	s_ID_RegDst	s_EX_RegDst	No	No	
11	o_Jump[1]	No	s_ID_Jump	s_EX_Jump	s_MEM_Jump	No	
12	o_ext_ctl[1]	No	s_ID_ext_ctl	s_EX_ext_ctl	s_MEM_ext_ctl	No	
13	o_jal_c[1]	No	s_ID_jal	s_EX_jal	s_MEM_jal	s_WB_jal	jal signal is used to know if we're writing PC+4 or the output from MemToReg
14	o_jr[1]	No	s_ID_jr	s_EX_jr	s_MEM_jr	No	
15	o_Branch[1]	No	No	s_EX_Branch	s_MEM_Branch	No	Output from ALU
16	PC[31:0]	s_IF_PC	No	No	No	No	NextInstrAddrs
17	PC+4[31:0]	s_IF_PCP4	s_ID_PCP4	s_EX_PCP4	s_MEM_PCP4	s_WB_PCP4	Carry all the way to the end to write back when jal. Use for input of Add1 and MUX2. (w_add0_add1, w_add0_mux2)
18	w_Instr[31:0]	s_IF_Instr	w_ID_Instr	No	No	No	
19	w_Instr[25:0]	No	w_ID_Instr_25t0	No	No	No	
20	w_Instr[31:26]	No	w_ID_Instr_31t26	No	No	No	
21	w_Instr[5:0]	No	w_ID_Instr_5t0	No	No	No	
22	w_Instr[25:21]	No	w_ID_Instr_25t21	No	No	No	
23	w_Instr[20:16]	No	s_ID_Instr_20t16	s_EX_Instr_20t16	No	No	Register Rt addrs
24	w_Instr[15:0]	No	w_ID_Instr_15t0	No	No	No	
25	w_Instr[15:11]	No	s_ID_Instr_15t11	s_EX_Instr_15t11	No	No	Register Rd addrs
26	s_rs_data[32]	No	s_ID_rs_data_o	s_EX_rs_data_o	s_MEM_rs_data_o	No	output from Register (w_jr_ra_pc_next)
27	s_rt_data[32]	No	s_ID_rt_data_o	s_EX_rt_data_o	s_MEM_rt_data_o	No	output from Register
28	w_ext_o[32]	No	w_ID_ext_o	w_EX_ext_o	No	No	output from sign extender
29	w_s120_0[28]	No	w_ID_s120_o	w_EX_s120_o	No	No	
30	w_pc4_s120_0	No	No	w_EX_pc4_s120_o	s_MEM_pc4_s120_o	No	
31	w_WrAddrs	No	No	w_EX_RegWrAddrs	w_MEM_RegWrAddrs	w_WB_WrAddrs	
32	w_shift_add1	No	No	w_EX_shift_add1	No	No	Extended immediate shifted left 2.
33	w_ALU_o	No	No	s_EX_ALU_o	s_MEM_DMEM_Addrs	s_WB_DMEM_Addrs	
34	w_add1_mux2	No	No	s_EX_add1_mux2	s_MEM_add1_mux2	No	
35	w_MUX7_o	No	No	w_EX_MUX7_o	No	No	ALU input A
36	w_MUX1_o	No	No	w_EX_MUX1_o	No	No	ALU input B
37	w_MUX2_MUX3	No	No	No	w_MEM_MUX2_MUX3	No	
38	w_MUX3_MUX5	No	No	No	w_MEM_MUX3_MUX5	No	
39	w_PC_next	No	No	No	w_MEM_PC_next	No	I don't know if this technically goes in IF or not
40	s_DMEM_out	No	No	No	w_MEM_DMEM_out	w_WB_DMEM_out	
41	s_DMEM_Lb	No	No	No	s_MEM_DMEM_Lb	s_WB_DMEM_Lb	
42	s_DMEM_Lh	No	No	NO	s_MEM_DMEM_Lh	s_WB_DMEM_Lh	
43	w_RegWrData	No	No	No	No	w_WB_RegWrData	

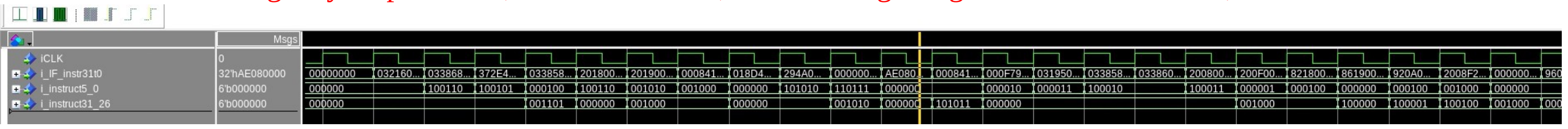
Key
Control Signals (from decoder)
Control Signals (from other)
Does not need to be stored

[1.b.ii] high-level schematic drawing of the interconnection between components.



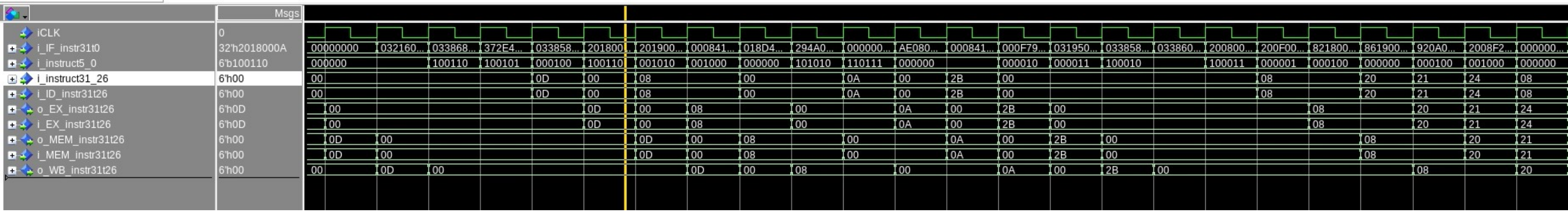
[1.c.i] include an annotated waveform in your writeup and provide a short discussion of result correctness.

For this part I simply modified the code from out original tests we did when making the single cycle processor, then modified it, restructuring things to not have hazards, but still



```
                Msgs
iCLK            0
i_IF_instr31t0  32'hAE080000   00000000 032160... 033868... 372E4... 033858... 201800... 201900... 000841... 018D4... 294A0... 000000... AE080... 000841... 000F79... 031950... 033858... 033860... 200800... 200F00... 821800... 861900... 920A0... 2008F2... 000000... 960
i_instruct5_0   6'b000000      000000 100110 100101 000100 100110 001010 001000 000000 101010 110111 000000 000010 000011 100010 100011 000001 000100 000000 000100 001000 000000
i_instruct31_26 6'b000000      000000 001101 000000 001000 000000 001010 000000 101011 000000 001000 100000 100001 100100 001000 000
```

allowing the code to run. I verified it worked by running it through the test program which compares the mars output to the Questasim output and can be verified If you check the sw pipeline folder. But what you see above is the Decoder input for instructions 5-0 and 26-31, which is the opcode and function fields of the instruction. Any opcode that

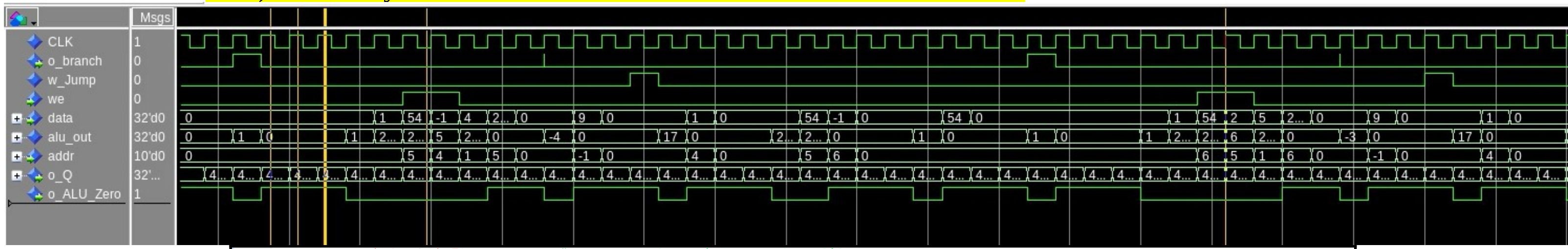


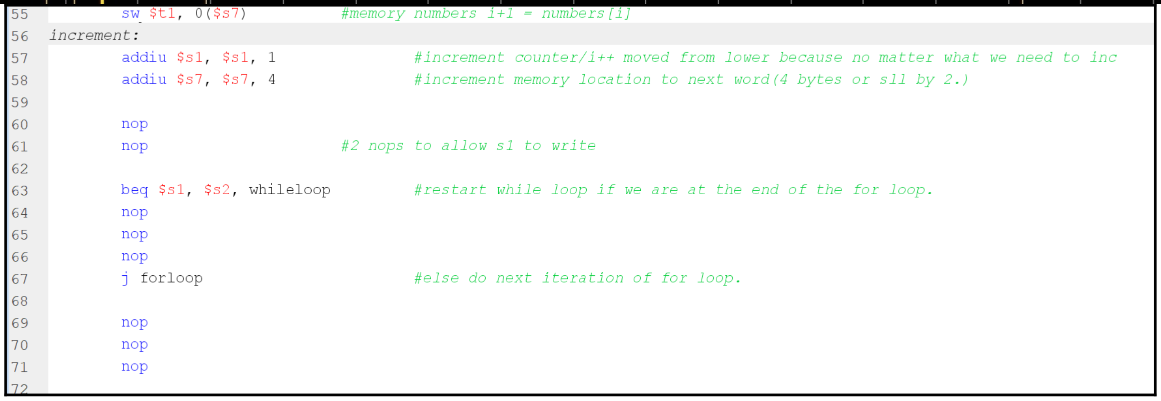
```
                 Msgs
iCLK             0
i_IF_instr31t0   32'h2018000A  00000000 032160... 033868... 372E4... 033858... 201800 201900... 000841... 018D4... 294A0... 000000... AE080... 000841... 000F79... 031950... 033858... 033860... 200800... 200F00... 821800... 861900... 920A0... 2008F2... 000000...
i_instruct5_0    6'b100110     000000 100110 100101 000100 100110 001010 001000 000000 101010 110111 000000 000010 000011 100010 100011 000001 000100 000000 000100 001000 000000
i_instruct31_26  6'h00         00 0D 00 08 00 0A 00 2B 00 08 20 21 24 08
i_ID_instr31t26  6'h00         00 0D 00 08 00 0A 00 2B 00 08 20 21 24 08
o_EX_instr31t26  6'h0D         00 0D 00 08 00 0A 00 2B 00 08 20 21 24
i_EX_instr31t26  6'h0D         00 0D 00 08 00 0A 00 2B 00 08 20 21 24
o_MEM_instr31t26 6'h00         0D 00 0D 00 08 00 0A 00 2B 00 08 20 21
i_MEM_instr31t26 6'h00         0D 00 0D 00 08 00 0A 00 2B 00 08 20 21
o_WB_instr31t26  6'h00         00 0D 00 0D 00 08 00 0A 00 2B 00 08 20
```

has 0x00 uses the funct, if not uses the opcode portion. Starting at 01101 of 31-26 you can see LW coming through the pipe, 001000 addi, slti etc.

Then in this photo you can follow the instructions specifically 31-26 as they go in and out of every register changing on the clock cycles.

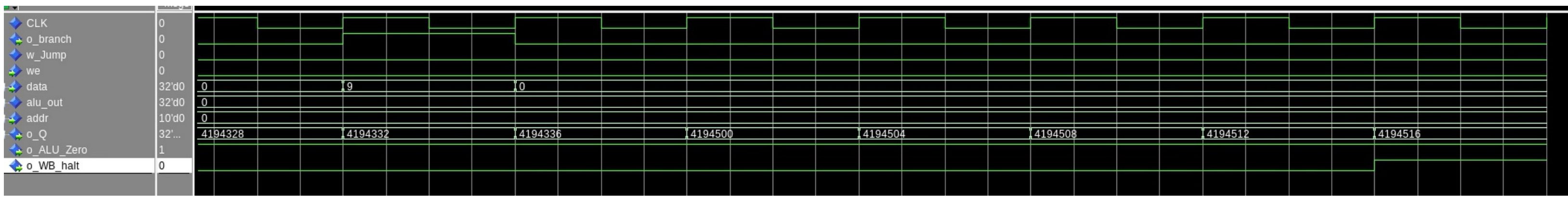
[1.c.ii] Include an annotated waveform in your writeup of two iterations or recursions of these programs executing correctly and provide a short discussion of result correctness. In your waveform and annotation, provide 3 different examples (at least one data-flow and one control-flow) of where you did not have to use the maximum number of NOPs.



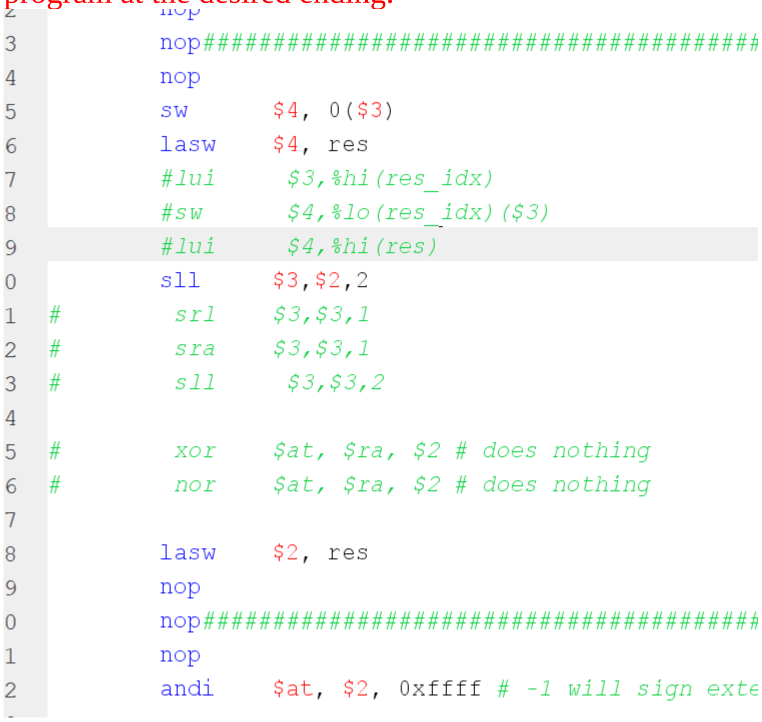


```
        sw $t1, 0($s7)          #memory numbers i+1 = numbers[i]
increment:
        addiu $s1, $s1, 1               #increment counter/i++ moved from lower because no matter what we need to inc
        addiu $s7, $s7, 4               #increment memory location to next word(4 bytes or sll by 2.)

        nop
        nop                     #2 nops to allow s1 to write

        beq $s1, $s2, whileloop         #restart while loop if we are at the end of the for loop.
        nop
        nop
        nop
        j forloop                       #else do next iteration of for loop.

        nop
        nop
        nop
```

Seen above is the bubble sort example of a data hazard and a control hazard avoidance. You can see its bubble sort being performed by seeing the memory locations of the first (Addr) 5,4 being compared, then 6,5 being compared, you can also see if write enable is up, then we were doing a swap of the numbers in memory. You can confirm the swap by seeing that 54,-1 swapped, then the next comparison is 54,0. This is because it always moves the smaller number to the left more position. To see the data hazards being delt with you can see the code, I was able to swap the addiu around to allow only 2 nops before the BEQ comparison. That's why the larger address is on the left and not the right, and originally it made more sense to have the lower address on the left and the higher on the right as visually it made more sense. You can then see that after every Beq or jump there is 3 nops. Confirmed with the yellow lines, you can see 3 alu zero signals being asserted, meaning that only 3 nops had to be used. We were able to reduce our control hazard potentials from 4 to 3 by moving our jump logic into the Memory stage. Later we'd go on to reduce it even further by updating the main registers to update on a negative clock edge, but that was explicitly stated not to do for the software pipeline.

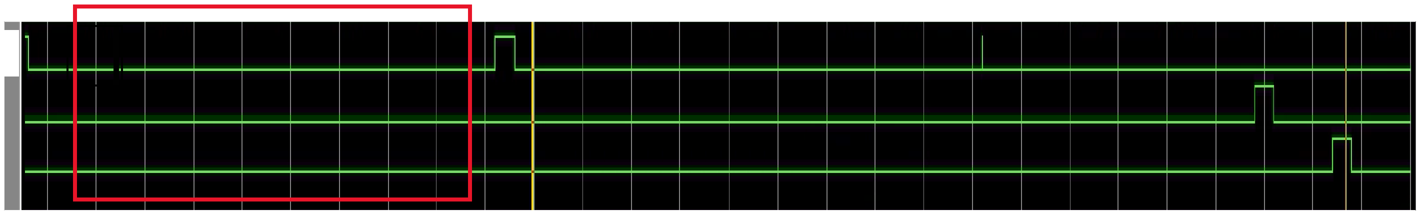


Besides being able to pass the bubblesort with the given testing software, we can tell the program executed correctly, because of the final branch, , followed by 3 nops, then 3 cycles later in the WB stage the Halt signal is high, meaning we successfully exited the program at the desired ending.

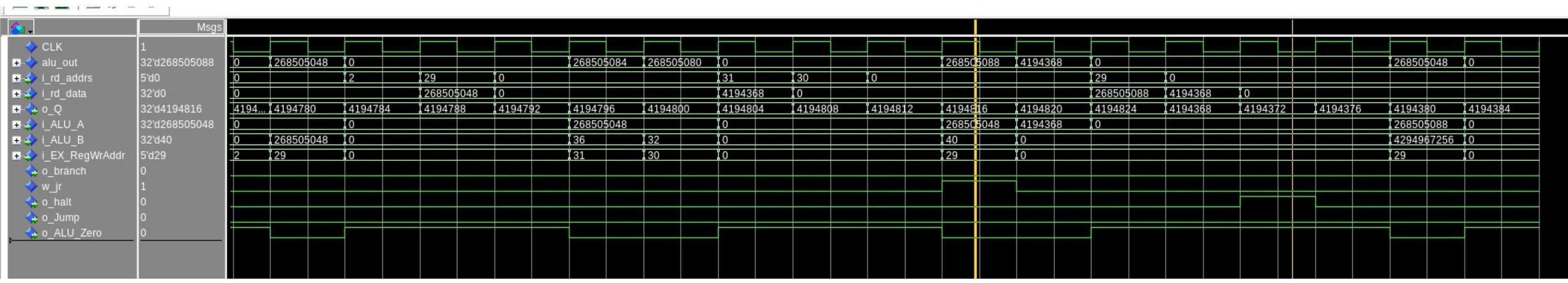


```
        nop
        nop#######################################
        nop
        sw      $4, 0($3)
        lasw    $4, res
        #lui     $3,%hi(res_idx)
        #sw      $4,%lo(res_idx)($3)
        #lui     $4,%hi(res)
        sll     $3,$2,2
#        srl     $3,$3,1
#        sra     $3,$3,1
#        sll      $3,$3,2

#        xor     $at, $ra, $2 # does nothing
#        nor     $at, $ra, $2 # does nothing

        lasw    $2, res
        nop
        nop#######################################
        nop
        andi    $at, $2, 0xffff # -1 will sign exte
```

Here we were able to avoid data hazards by just removing redundent instruction. I saw while looking at them them that we were shifting and the value of $3 was not changing, reguardless of the three instrucitons running as no matter what they'd end with the same value. This was an example of data hazard avoidance.(because of it being very hard to find instruction ~insturction 1000 on the waveform I did not include a waveform corresponding for this.)



Because of the length of this phot it does look kind of weird, and I choose to crop out the signals. However, I felt it showed the jumping and recursion the best, see below for more details. But this is at the exiting of grendle, where it attempts to do a branch inside welcome, does not branch, then executedsthe last loaded $31, using jr, which takes it to the pump label. It then halts and this is seen as the halt signal becoming high.



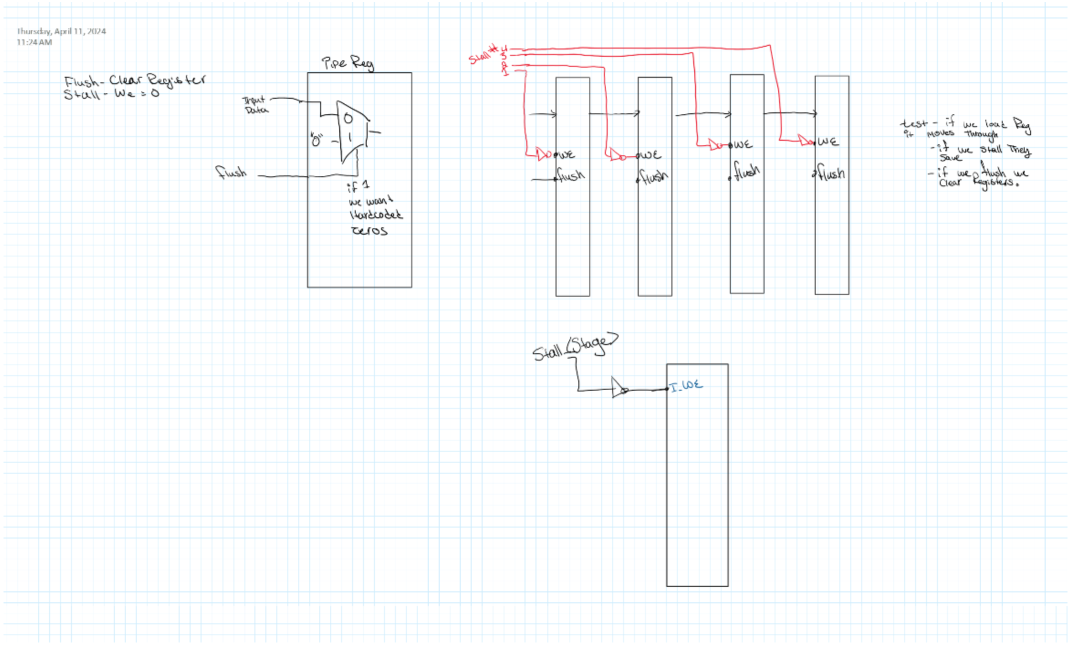
	Msgs
CLK	1
alu_out	32'd268505088
i_rd_addrs	5'd0
i_rd_data	32'd0
o_Q	32'd4194816
i_ALU_A	32'd268505048
i_ALU_B	32'd40
i_EX_RegWrAddr	5'd29
o_branch	0
w_jr	1
o_halt	0
o_Jump	0
o_ALU_Zero	0

Here at the two yellow lines you can see that first the Jr Signal is asserter, then its followed by a halt. On the left red circle you can see when register 31 is written to via the load word, then in the middle you can see the alu out putting out the new jump location, and 2 cycles later the pc changing to the same location. You can also look at some additional data hazard avoidance by seeing that if there are 3 cycles where alu a and alu b are zero, a nop was performed(less than the max of 4) to avoid data dependencies, or if we were looking at a jump occuring just before the nops, a case of avoiding a control hazard.

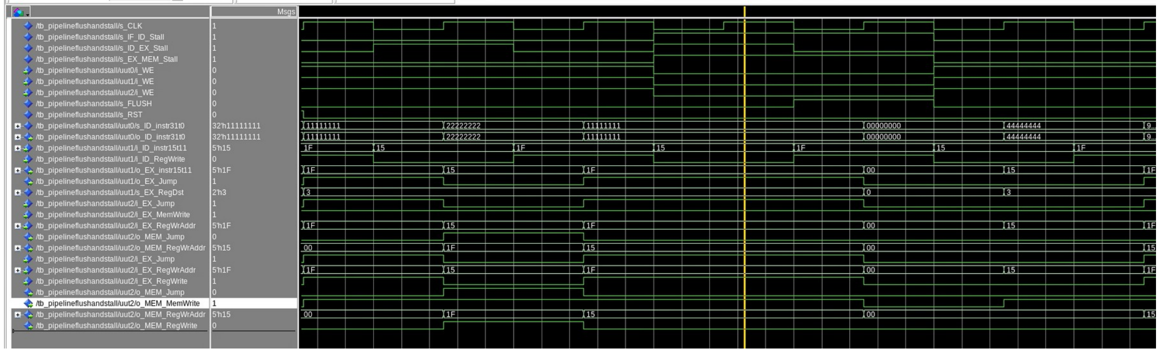
[1.d] report the maximum frequency your software-scheduled pipelined processor can run at and determine what your critical path is (specify each module/entity/component that this path goes through).

53.02mhz, the critical path for this is DMEM/WordDecoder/MEMWBPipe. The memory stage takes the longest as the DMEM is a beefy boi. So the critical path for this processor is as above.

[2.a.ii] Draw a simple schematic showing how you could implement stalling and flushing operations given an ideal N-bit register.

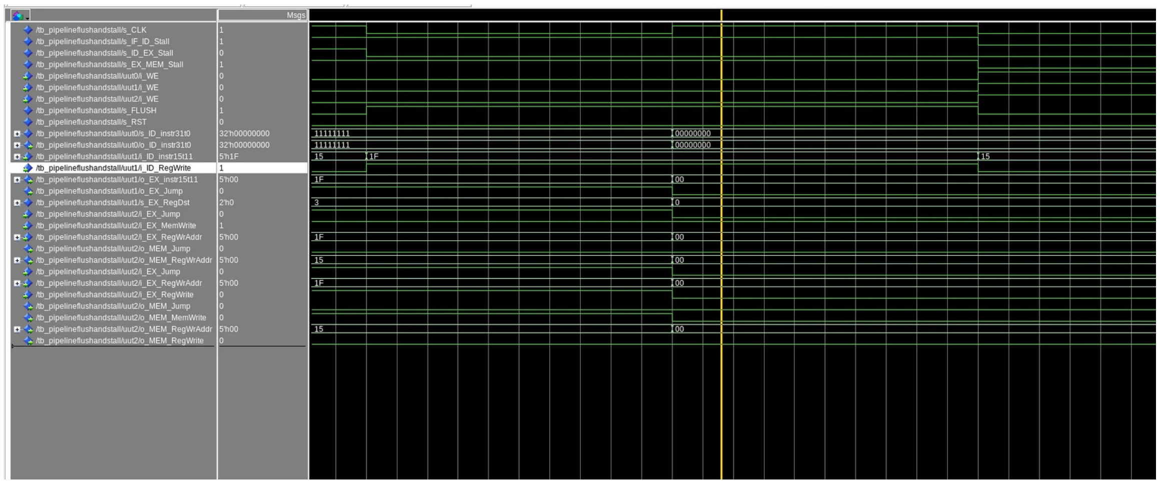


[2.a.iii] Create a testbench that instantiates all four of the registers in a single design. Show that values that are stored in the initial IF/ID register are available as expected four cycles later, and that new values can be inserted into the pipeline every single cycle. Most importantly, this testbench should also test that each pipeline register can be individually stalled or flushed.



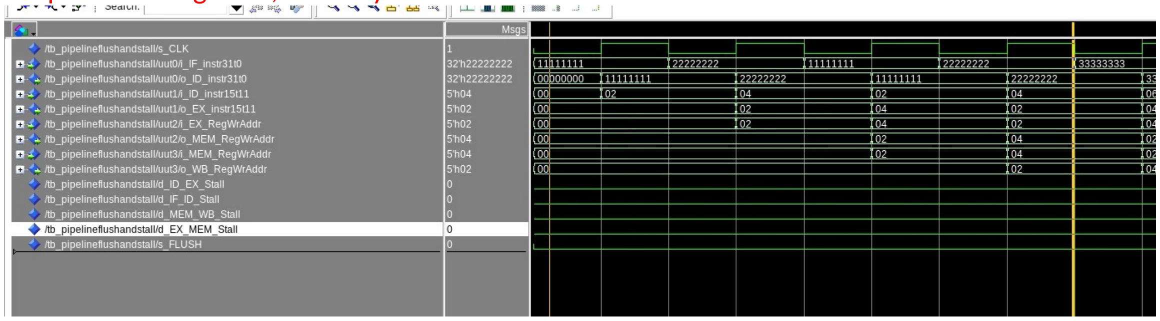
```
                                                   Msgs
/tb_pipelineflushandstall/s_CLK                    1
/tb_pipelineflushandstall/s_IF_ID_Stall            1
/tb_pipelineflushandstall/s_ID_EX_Stall            1
/tb_pipelineflushandstall/s_EX_MEM_Stall           1
/tb_pipelineflushandstall/uut0/i_WE                0
/tb_pipelineflushandstall/uut1/i_WE                0
/tb_pipelineflushandstall/uut2/i_WE                0
/tb_pipelineflushandstall/s_FLUSH                  0
/tb_pipelineflushandstall/s_RST                    0
/tb_pipelineflushandstall/uut0/s_ID_instr31t0      32'h11111111
/tb_pipelineflushandstall/uut0/o_ID_instr31t0      32'h11111111
/tb_pipelineflushandstall/uut1/i_ID_instr15t11     5'h15
/tb_pipelineflushandstall/uut1/i_ID_RegWrite       0
/tb_pipelineflushandstall/uut1/o_EX_instr15t11     5'h1F
/tb_pipelineflushandstall/uut1/o_EX_Jump           1
/tb_pipelineflushandstall/uut1/s_EX_RegDst         2'h3
/tb_pipelineflushandstall/uut2/i_EX_Jump           1
/tb_pipelineflushandstall/uut2/i_EX_MemWrite       1
/tb_pipelineflushandstall/uut2/i_EX_RegWrAddr      5'h1F
/tb_pipelineflushandstall/uut2/o_MEM_Jump          0
/tb_pipelineflushandstall/uut2/o_MEM_RegWrAddr     5'h15
/tb_pipelineflushandstall/uut2/i_EX_Jump           1
/tb_pipelineflushandstall/uut2/i_EX_RegWrAddr      5'h1F
/tb_pipelineflushandstall/uut2/i_EX_RegWrite       1
/tb_pipelineflushandstall/uut2/o_MEM_Jump          0
/tb_pipelineflushandstall/uut2/o_MEM_MemWrite      1
/tb_pipelineflushandstall/uut2/o_MEM_RegWrAddr     5'h15
/tb_pipelineflushandstall/uut2/o_MEM_RegWrite      0
```

You can see here that when stall is active, none of the outputs change, even though the physical input i_idInstr15t11 is still driving them.



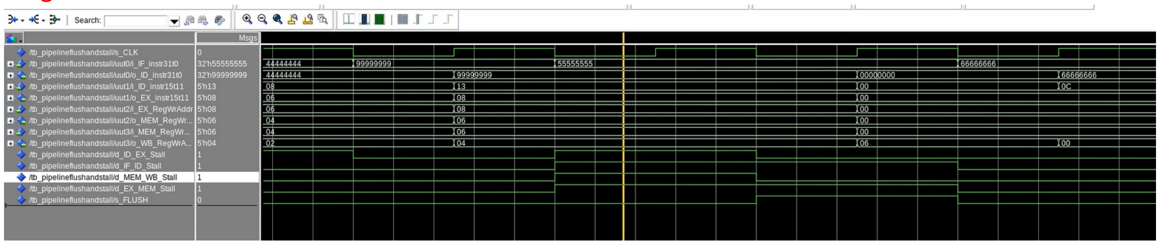
```
/tb_pipelineflushandstall/s_CLK                      1
/tb_pipelineflushandstall/s_IF_ID_Stall              1
/tb_pipelineflushandstall/s_ID_EX_Stall              0
/tb_pipelineflushandstall/s_EX_MEM_Stall             1
/tb_pipelineflushandstall/uut0/i_WE                  0
/tb_pipelineflushandstall/uut1/i_WE                  0
/tb_pipelineflushandstall/uut2/i_WE                  0
/tb_pipelineflushandstall/s_FLUSH                    1
/tb_pipelineflushandstall/s_RST                      0
/tb_pipelineflushandstall/uut0/s_ID_instr31t0        32'h00000000
/tb_pipelineflushandstall/uut0/o_ID_instr31t0        32'h00000000
/tb_pipelineflushandstall/uut1/i_ID_instr15t11       5'h1F
/tb_pipelineflushandstall/uut1/i_ID_RegWrite         1
/tb_pipelineflushandstall/uut1/o_EX_instr15t11       5'h00
/tb_pipelineflushandstall/uut1/o_EX_Jump             0
/tb_pipelineflushandstall/uut1/s_EX_RegDst           2'h0
/tb_pipelineflushandstall/uut2/i_EX_Jump             0
/tb_pipelineflushandstall/uut2/i_EX_MemWrite         1
/tb_pipelineflushandstall/uut2/i_EX_RegWrAddr        5'h00
/tb_pipelineflushandstall/uut2/o_MEM_Jump            0
/tb_pipelineflushandstall/uut2/o_MEM_RegWrAddr       5'h00
/tb_pipelineflushandstall/uut2/i_EX_Jump             0
/tb_pipelineflushandstall/uut2/i_EX_RegWrAddr        5'h00
/tb_pipelineflushandstall/uut2/i_EX_RegWrite         0
/tb_pipelineflushandstall/uut2/o_MEM_Jump            0
/tb_pipelineflushandstall/uut2/o_MEM_MemWrite        0
/tb_pipelineflushandstall/uut2/o_MEM_RegWrAddr       5'h00
/tb_pipelineflushandstall/uut2/o_MEM_RegWrite        0
```

You can then see here, that when flush is enabled, everything is cleared to 0 on the positive edge of the clock. Flush for this tb was hooked up to one signal to test how it works, but in the final design is hooked up with individual signals, you can also see here that stall is enabled on a couple of the registers, but they still default to 0.



```
/tb_pipelineflushandstall/s_CLK                      1
/tb_pipelineflushandstall/uut0/i_IF_instr31t0        32'h22222222
/tb_pipelineflushandstall/uut0/o_ID_instr31t0        32'h22222222
/tb_pipelineflushandstall/uut1/i_ID_instr15t11       5'h04
/tb_pipelineflushandstall/uut1/o_EX_instr15t11       5'h02
/tb_pipelineflushandstall/uut2/i_EX_RegWrAddr        5'h02
/tb_pipelineflushandstall/uut2/o_MEM_RegWrAddr       5'h04
/tb_pipelineflushandstall/uut3/i_MEM_RegWrAddr       5'h04
/tb_pipelineflushandstall/uut3/o_WB_RegWrAddr        5'h02
/tb_pipelineflushandstall/d_ID_EX_Stall              0
/tb_pipelineflushandstall/d_IF_ID_Stall              0
/tb_pipelineflushandstall/d_MEM_WB_Stall             0
/tb_pipelineflushandstall/d_EX_MEM_Stall             0
/tb_pipelineflushandstall/s_FLUSH                    0
```

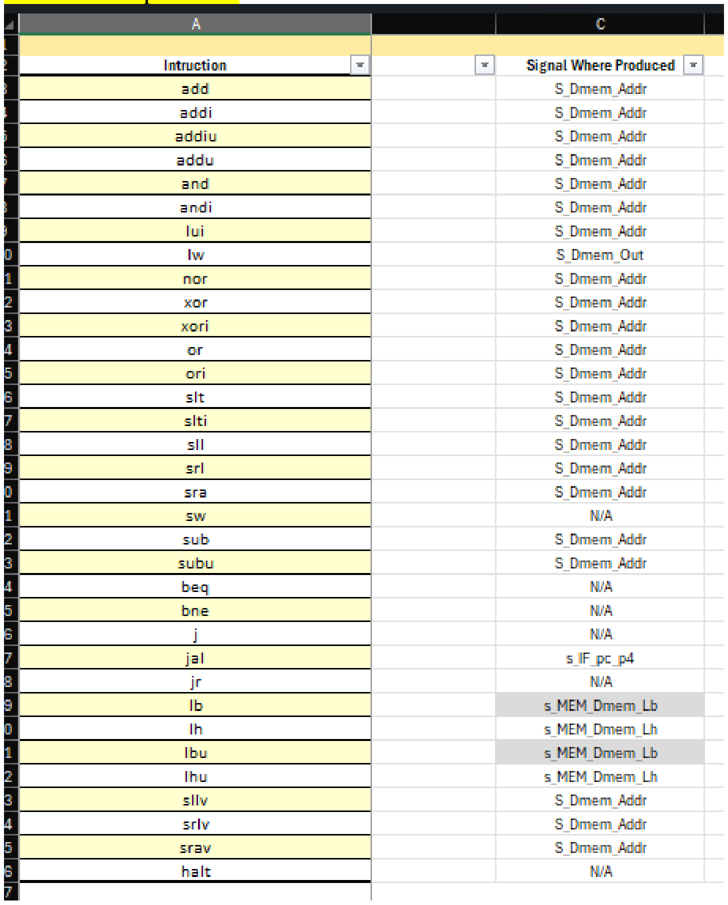
You can see here that the written information in the IF stage is available in the WB output 4 stages later



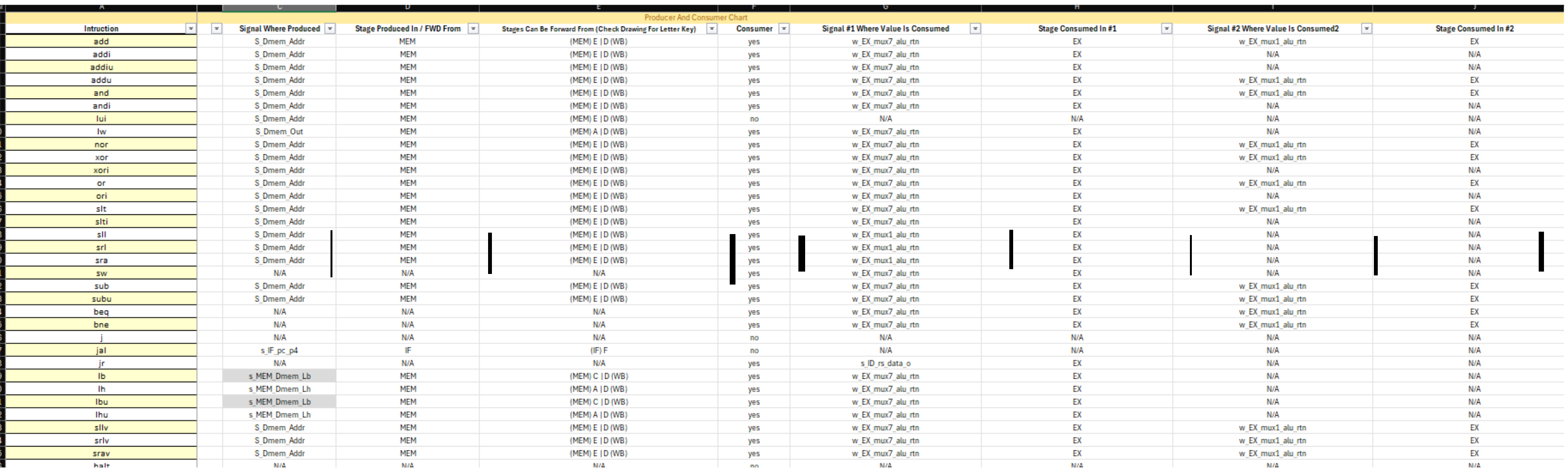
```
/tb_pipelineflushandstall/s_CLK                      0
/tb_pipelineflushandstall/uut0/i_IF_instr31t0        32'h55555555
/tb_pipelineflushandstall/uut0/o_ID_instr31t0        32'h99999999
/tb_pipelineflushandstall/uut1/i_ID_instr15t11       5'h13
/tb_pipelineflushandstall/uut1/o_EX_instr15t11       5'h08
/tb_pipelineflushandstall/uut2/i_EX_RegWrAddr        5'h08
/tb_pipelineflushandstall/uut2/o_MEM_RegWr...        5'h06
/tb_pipelineflushandstall/uut3/i_MEM_RegWr...        5'h06
/tb_pipelineflushandstall/uut3/o_WB_RegWrA...        5'h04
/tb_pipelineflushandstall/d_ID_EX_Stall              1
/tb_pipelineflushandstall/d_IF_ID_Stall              1
/tb_pipelineflushandstall/d_MEM_WB_Stall             1
/tb_pipelineflushandstall/d_EX_MEM_Stall             1
/tb_pipelineflushandstall/s_FLUSH                    0
```

If a stall occurs across the board or just in a local value, you can see the written output values do not change in regards to that register stack.

[2.b.i] list which instructions produce values, and what signals (i.e., bus names) in the pipeline these correspond to.



Intruction		Signal Where Produced
add		S_Dmem_Addr
addi		S_Dmem_Addr
addiu		S_Dmem_Addr
addu		S_Dmem_Addr
and		S_Dmem_Addr
andi		S_Dmem_Addr
lui		S_Dmem_Addr
lw		S_Dmem_Out
nor		S_Dmem_Addr
xor		S_Dmem_Addr
xori		S_Dmem_Addr
or		S_Dmem_Addr
ori		S_Dmem_Addr
slt		S_Dmem_Addr
slti		S_Dmem_Addr
sll		S_Dmem_Addr
srl		S_Dmem_Addr
sra		S_Dmem_Addr
sw		N/A
sub		S_Dmem_Addr
subu		S_Dmem_Addr
beq		N/A
bne		N/A
j		N/A
jal		s_IF_pc_p4
jr		N/A
lb		s_MEM_Dmem_Lb
lh		s_MEM_Dmem_Lh
lbu		s_MEM_Dmem_Lb
lhu		s_MEM_Dmem_Lh
sllv		S_Dmem_Addr
srlv		S_Dmem_Addr
srav		S_Dmem_Addr
halt		N/A

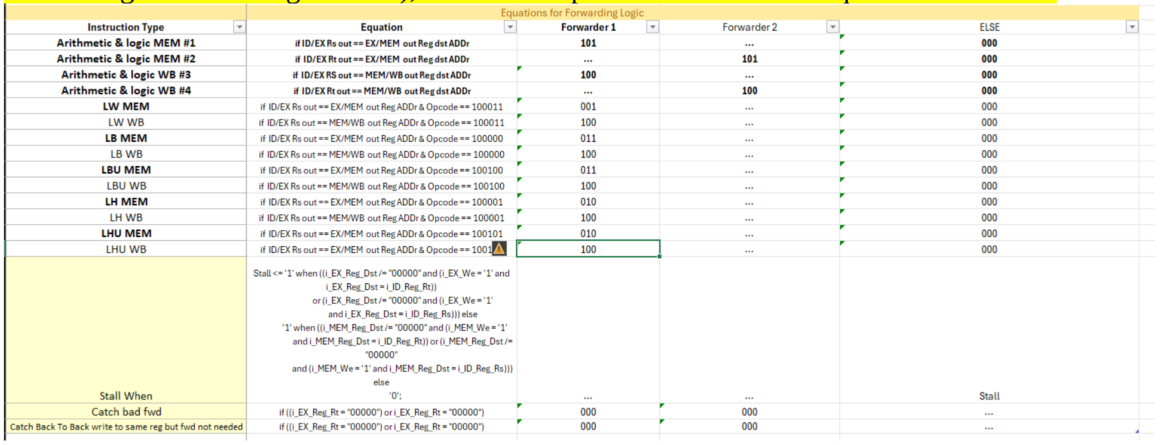


Producer And Consumer Chart

Intruction	Signal Where Produced	Stage Produced In / FWD From	Stages Can Be Forward From (Check Drawing For Letter Key)	Consumer	Signal #1 Where Value Is Consumed	Stage Consumed In #1	Signal #2 Where Value Is Consumed2	Stage Consumed In #2
add	S_Dmem_Addr	MEM	(MEM) E | D (WB)	yes	w_EX_mux7_alu_rtn	EX	w_EX_mux1_alu_rtn	EX
addi	S_Dmem_Addr	MEM	(MEM) E | D (WB)	yes	w_EX_mux7_alu_rtn	EX	N/A	N/A
addiu	S_Dmem_Addr	MEM	(MEM) E | D (WB)	yes	w_EX_mux7_alu_rtn	EX	N/A	N/A
addu	S_Dmem_Addr	MEM	(MEM) E | D (WB)	yes	w_EX_mux7_alu_rtn	EX	w_EX_mux1_alu_rtn	EX
and	S_Dmem_Addr	MEM	(MEM) E | D (WB)	yes	w_EX_mux7_alu_rtn	EX	w_EX_mux1_alu_rtn	EX
andi	S_Dmem_Addr	MEM	(MEM) E | D (WB)	yes	w_EX_mux7_alu_rtn	EX	N/A	N/A
lui	S_Dmem_Addr	MEM	(MEM) E | D (WB)	no	N/A	N/A	N/A	N/A
lw	S_Dmem_Out	MEM	(MEM) A | D (WB)	yes	w_EX_mux7_alu_rtn	EX	N/A	N/A
nor	S_Dmem_Addr	MEM	(MEM) E | D (WB)	yes	w_EX_mux7_alu_rtn	EX	w_EX_mux1_alu_rtn	EX
xor	S_Dmem_Addr	MEM	(MEM) E | D (WB)	yes	w_EX_mux7_alu_rtn	EX	w_EX_mux1_alu_rtn	EX
xori	S_Dmem_Addr	MEM	(MEM) E | D (WB)	yes	w_EX_mux7_alu_rtn	EX	N/A	N/A
or	S_Dmem_Addr	MEM	(MEM) E | D (WB)	yes	w_EX_mux7_alu_rtn	EX	w_EX_mux1_alu_rtn	EX
ori	S_Dmem_Addr	MEM	(MEM) E | D (WB)	yes	w_EX_mux7_alu_rtn	EX	N/A	N/A
slt	S_Dmem_Addr	MEM	(MEM) E | D (WB)	yes	w_EX_mux7_alu_rtn	EX	w_EX_mux1_alu_rtn	EX
slti	S_Dmem_Addr	MEM	(MEM) E | D (WB)	yes	w_EX_mux7_alu_rtn	EX	N/A	N/A
sll	S_Dmem_Addr	MEM	(MEM) E | D (WB)	yes	w_EX_mux1_alu_rtn	EX	N/A	N/A
srl	S_Dmem_Addr	MEM	(MEM) E | D (WB)	yes	w_EX_mux1_alu_rtn	EX	N/A	N/A
sra	S_Dmem_Addr	MEM	(MEM) E | D (WB)	yes	w_EX_mux1_alu_rtn	EX	N/A	N/A
sw	N/A	N/A	N/A	yes	w_EX_mux7_alu_rtn	EX	N/A	N/A
sub	S_Dmem_Addr	MEM	(MEM) E | D (WB)	yes	w_EX_mux7_alu_rtn	EX	w_EX_mux1_alu_rtn	EX
subu	S_Dmem_Addr	MEM	(MEM) E | D (WB)	yes	w_EX_mux7_alu_rtn	EX	w_EX_mux1_alu_rtn	EX
beq	N/A	N/A	N/A	yes	w_EX_mux7_alu_rtn	EX	w_EX_mux1_alu_rtn	EX
bne	N/A	N/A	N/A	yes	w_EX_mux7_alu_rtn	EX	w_EX_mux1_alu_rtn	EX
j	N/A	N/A	N/A	no	N/A	N/A	N/A	N/A
jal	s_IF_pc_p4	IF	(IF) F	no	N/A	N/A	N/A	N/A
jr	N/A	N/A	N/A	yes	s_ID_rs_data_o	EX	N/A	N/A
lb	s_MEM_Dmem_Lb	MEM	(MEM) C | D (WB)	yes	w_EX_mux7_alu_rtn	EX	N/A	N/A
lh	s_MEM_Dmem_Lh	MEM	(MEM) A | D (WB)	yes	w_EX_mux7_alu_rtn	EX	N/A	N/A
lbu	s_MEM_Dmem_Lb	MEM	(MEM) C | D (WB)	yes	w_EX_mux7_alu_rtn	EX	N/A	N/A
lhu	s_MEM_Dmem_Lh	MEM	(MEM) A | D (WB)	yes	w_EX_mux7_alu_rtn	EX	N/A	N/A
sllv	S_Dmem_Addr	MEM	(MEM) E | D (WB)	yes	w_EX_mux7_alu_rtn	EX	w_EX_mux1_alu_rtn	EX
srlv	S_Dmem_Addr	MEM	(MEM) E | D (WB)	yes	w_EX_mux7_alu_rtn	EX	w_EX_mux1_alu_rtn	EX
srav	S_Dmem_Addr	MEM	(MEM) E | D (WB)	yes	w_EX_mux7_alu_rtn	EX	w_EX_mux1_alu_rtn	EX
halt	N/A	N/A	N/A	no	N/A	N/A	N/A	N/A

[2.b.ii] List which of these same instructions consume values, and what signals in the pipeline these correspond to.

[2.b.iii] generalized list of potential data dependencies. From this generalized list, select those dependencies that can be forwarded (write down the corresponding pipeline stages that will be forwarding and receiving the data), and those dependencies that will require hazard stalls.

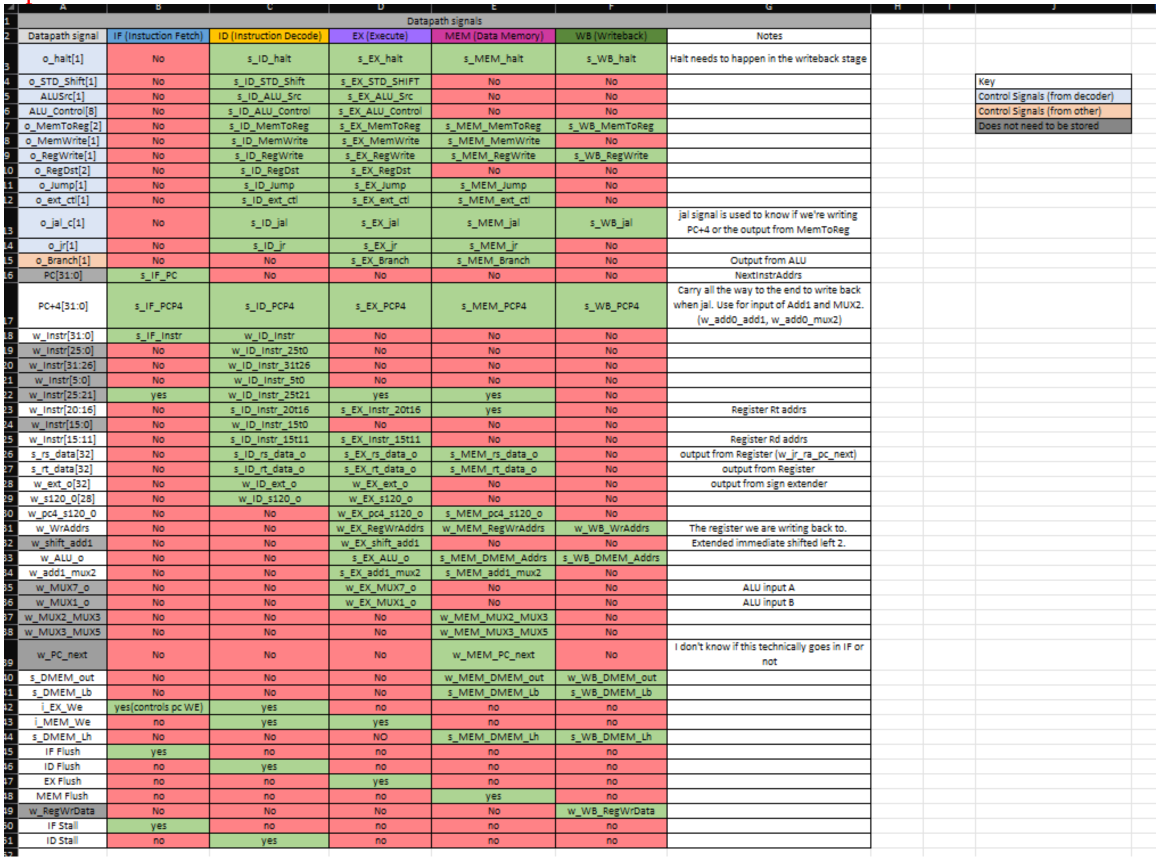


Equations for Forwarding Logic

Instruction Type	Equation	Forwarder 1	Forwarder 2	ELSE
Arithmetic & logic MEM #1	**if ID/EX Rs out == EX/MEM out Reg dst ADDr**	**101**	...	**000**
Arithmetic & logic MEM #2	**if ID/EX Rt out == EX/MEM out Reg dst ADDr**	...	**101**	**000**
Arithmetic & logic WB #3	**if ID/EX RS out == MEM/WB out Reg dst ADDr**	**100**	...	**000**
Arithmetic & logic WB #4	**if ID/EX Rt out == MEM/WB out Reg dst ADDr**	...	**100**	**000**
LW MEM	if ID/EX Rs out == EX/MEM out Reg ADDr & Opcode == 100011	001	...	000
LW WB	if ID/EX Rs out == MEM/WB out Reg ADDr & Opcode == 100011	100	...	000
LB MEM	if ID/EX Rs out == EX/MEM out Reg ADDr & Opcode == 100000	011	...	000
LB WB	if ID/EX Rs out == MEM/WB out Reg ADDr & Opcode == 100000	100	...	000
LBU MEM	if ID/EX Rs out == EX/MEM out Reg ADDr & Opcode == 100100	011	...	000
LBU WB	if ID/EX Rs out == MEM/WB out Reg ADDr & Opcode == 100100	100	...	000
LH MEM	if ID/EX Rs out == EX/MEM out Reg ADDr & Opcode == 100001	010	...	000
LH WB	if ID/EX Rs out == MEM/WB out Reg ADDr & Opcode == 100001	100	...	000
LHU MEM	if ID/EX Rs out == EX/MEM out Reg ADDr & Opcode == 100101	010	...	000
LHU WB	if ID/EX Rs out == EX/MEM out Reg ADDr & Opcode == 1001	100	...	000
Stall When	Stall <= '1' when ((i_EX_Reg_Dst /= "00000" and (i_EX_We = '1' and i_EX_Reg_Dst = i_ID_Reg_Rt)) or (i_EX_Reg_Dst /= "00000" and (i_EX_We = '1' and i_EX_Reg_Dst = i_ID_Reg_Rs))) else '1' when ((i_MEM_Reg_Dst /= "00000" and (i_MEM_We = '1' and i_MEM_Reg_Dst = i_ID_Reg_Rt)) or (i_MEM_Reg_Dst /= "00000" and (i_MEM_We = '1' and i_MEM_Reg_Dst = i_ID_Reg_Rs))) else '0';	...	...	Stall
Catch bad fwd	if ((i_EX_Reg_Rt = "00000") or i_EX_Reg_Rt = "00000")	000	000	...
Catch Back To Back write to same reg but fwd not needed	if ((i_EX_Reg_Rt = "00000") or i_EX_Reg_Rt = "00000")	000	000	...

[2.b.iv] global list of the datapath values and control signals that are required during each pipeline stage

Updated from above



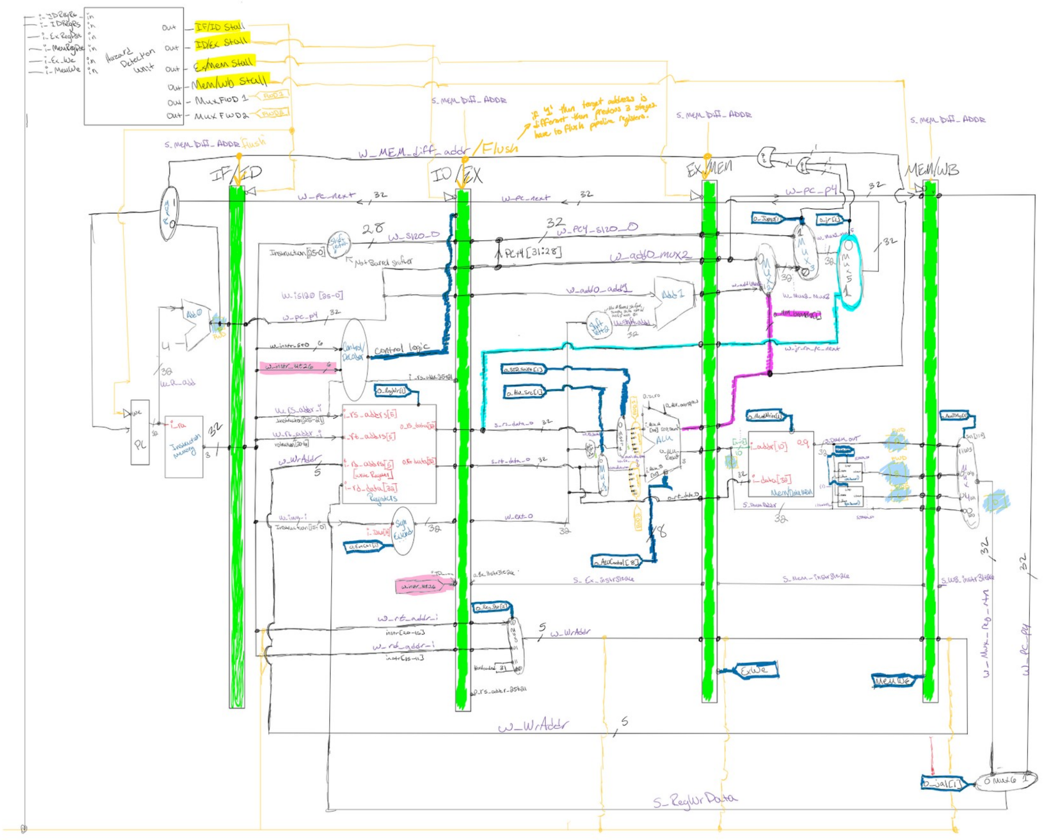
Datapath signals						
Datapath signal	IF (Instuction Fetch)	ID (Instruction Decode)	EX (Execute)	MEM (Data Memory)	WB (Writeback)	Notes
o_halt[1]	No	s_ID_halt	s_EX_halt	s_MEM_halt	s_WB_halt	Halt needs to happen in the writeback stage
o_STD_Shift[1]	No	s_ID_STD_Shift	s_EX_STD_SHIFT	No	No	
ALUSrc[1]	No	s_ID_ALU_Src	s_EX_ALU_Src	No	No	
ALU_Control[8]	No	s_ID_ALU_Control	s_EX_ALU_Control	No	No	
o_MemToReg[2]	No	s_ID_MemToReg	s_EX_MemToReg	s_MEM_MemToReg	s_WB_MemToReg	
o_MemWrite[1]	No	s_ID_MemWrite	s_EX_MemWrite	s_MEM_MemWrite	No	
o_RegWrite[1]	No	s_ID_RegWrite	s_EX_RegWrite	s_MEM_RegWrite	s_WB_RegWrite	
o_RegDst[2]	No	s_ID_RegDst	s_EX_RegDst	No	No	
o_Jump[1]	No	s_ID_Jump	s_EX_Jump	s_MEM_Jump	No	
o_ext_ctl[1]	No	s_ID_ext_ctl	s_EX_ext_ctl	s_MEM_ext_ctl	No	
o_jal_c[1]	No	s_ID_jal	s_EX_jal	s_MEM_jal	s_WB_jal	jal signal is used to know if we're writing PC+4 or the output from MemToReg
o_jr[1]	No	s_ID_jr	s_EX_jr	s_MEM_jr	No	
o_Branch[1]	No	No	s_EX_Branch	s_MEM_Branch	No	Output from ALU
PC[31:0]	s_IF_PC	No	No	No	No	NextInstrAddrs
PC+4[31:0]	s_IF_PCP4	s_ID_PCP4	s_EX_PCP4	s_MEM_PCP4	s_WB_PCP4	Carry all the way to the end to write back when jal. Use for input of Add1 and MUX2. (w_add0_add1, w_add0_mux2)
w_Instr[31:0]	s_IF_Instr	w_ID_Instr	No	No	No	
w_Instr[25:0]	No	w_ID_Instr_25t0	No	No	No	
w_Instr[31:26]	No	w_ID_Instr_31t26	No	No	No	
w_Instr[5:0]	No	w_ID_Instr_5t0	No	No	No	
w_Instr[25:21]	yes	w_ID_Instr_25t21	yes	yes	No	
w_Instr[20:16]	No	s_ID_Instr_20t16	s_EX_Instr_20t16	yes	No	Register Rt addrs
w_Instr[15:0]	No	w_ID_Instr_15t0	No	No	No	
w_Instr[15:11]	No	s_ID_Instr_15t11	s_EX_Instr_15t11	No	No	Register Rd addrs
s_rs_data[32]	No	s_ID_rs_data_o	s_EX_rs_data_o	s_MEM_rs_data_o	No	output from Register (w_jr_ra_pc_next)
s_rt_data[32]	No	s_ID_rt_data_o	s_EX_rt_data_o	s_MEM_rt_data_o	No	output from Register
w_ext_o[32]	No	w_ID_ext_o	w_EX_ext_o	No	No	output from sign extender
w_s120_0[28]	No	w_ID_s120_o	w_EX_s120_o	No	No	
w_pc4_s120_0	No	No	w_EX_pc4_s120_o	s_MEM_pc4_s120_o	No	
w_WrAddrs	No	No	w_EX_RegWrAddrs	w_MEM_RegWrAddrs	w_WB_WrAddrs	The register we are writing back to.
w_shift_add1	No	No	w_EX_shift_add1	No	No	Extended immediate shifted left 2.
w_ALU_o	No	No	s_EX_ALU_o	s_MEM_DMEM_Addrs	s_WB_DMEM_Addrs	
w_add1_mux2	No	No	s_EX_add1_mux2	s_MEM_add1_mux2	No	
w_MUX7_o	No	No	w_EX_MUX7_o	No	No	ALU input A
w_MUX1_o	No	No	w_EX_MUX1_o	No	No	ALU input B
w_MUX2_MUX3	No	No	No	w_MEM_MUX2_MUX3	No	
w_MUX3_MUX5	No	No	No	w_MEM_MUX3_MUX5	No	
w_PC_next	No	No	No	w_MEM_PC_next	No	I don't know if this technically goes in IF or not
s_DMEM_out	No	No	No	w_MEM_DMEM_out	w_WB_DMEM_out	
s_DMEM_Lb	No	No	No	s_MEM_DMEM_Lb	s_WB_DMEM_Lb	
i_EX_We	yes(controls pc WE)	yes	no	no	no	
i_MEM_We	no	yes	yes	no	no	
s_DMEM_Lh	No	No	NO	s_MEM_DMEM_Lh	s_WB_DMEM_Lh	
IF Flush	yes	no	no	no	no	
ID Flush	no	yes	no	no	no	
EX Flush	no	no	yes	no	no	
MEM Flush	no	no	no	yes	no	
w_RegWrData	No	No	No	No	w_WB_RegWrData	
IF Stall	yes	no	no	no	no	
ID Stall	no	yes	no	no	no	

Key
Control Signals (from decoder)
Control Signals (from other)
Does not need to be stored

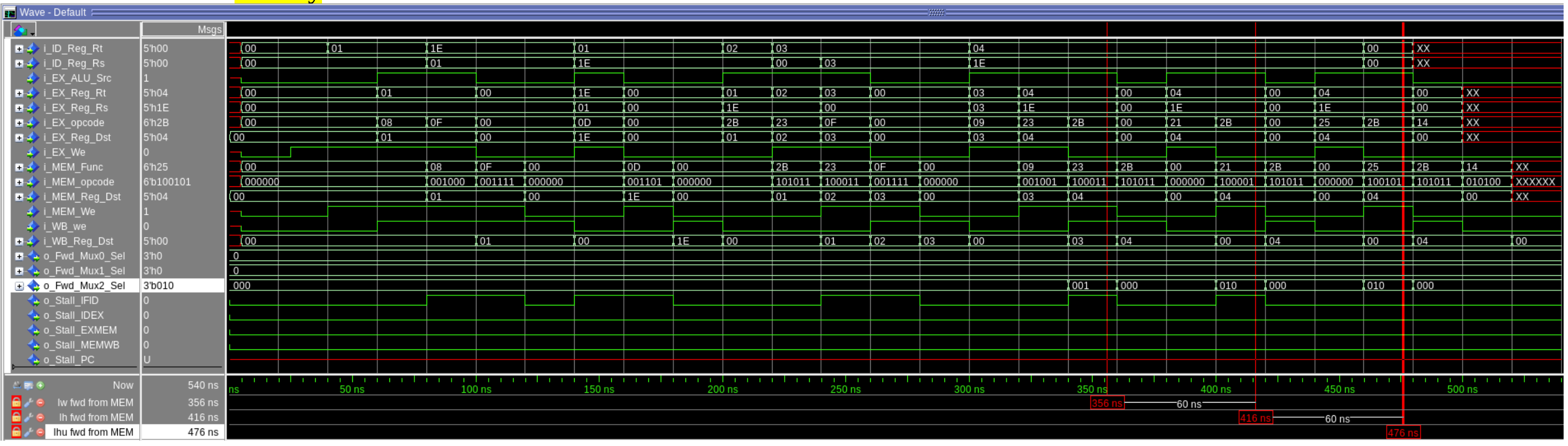
[2.c.i] list all instructions that may result in a non-sequential PC update and in which pipeline stage that update occurs. Branch, jump, and jal (or anything that jumps.) this is executed for us in the Memory stage, because of this we had to add a special instruction to the PC We to not allow a stall to occur in the PC if flush or stall is occurring at that exact time (flush would occur during the execute stage for us in anything that results in a jump)

[2.c.ii] For these instructions, list which stages need to be stalled and which stages need to be squashed/flushed relative to the stage each of these instructions is in.
For Anything that meets the criteria for the stall we gave above we had it stall the IF_ID registers, as well as the pc. Along with this, we also had the ID_EX registers flushed as to make sure a junk instruction doesn't make its way through.

[2.d] implement the hardware-scheduled pipeline using only structural VHDL. As with the previous processors that you have implemented, start with a high-level schematic drawing of the interconnection between components.

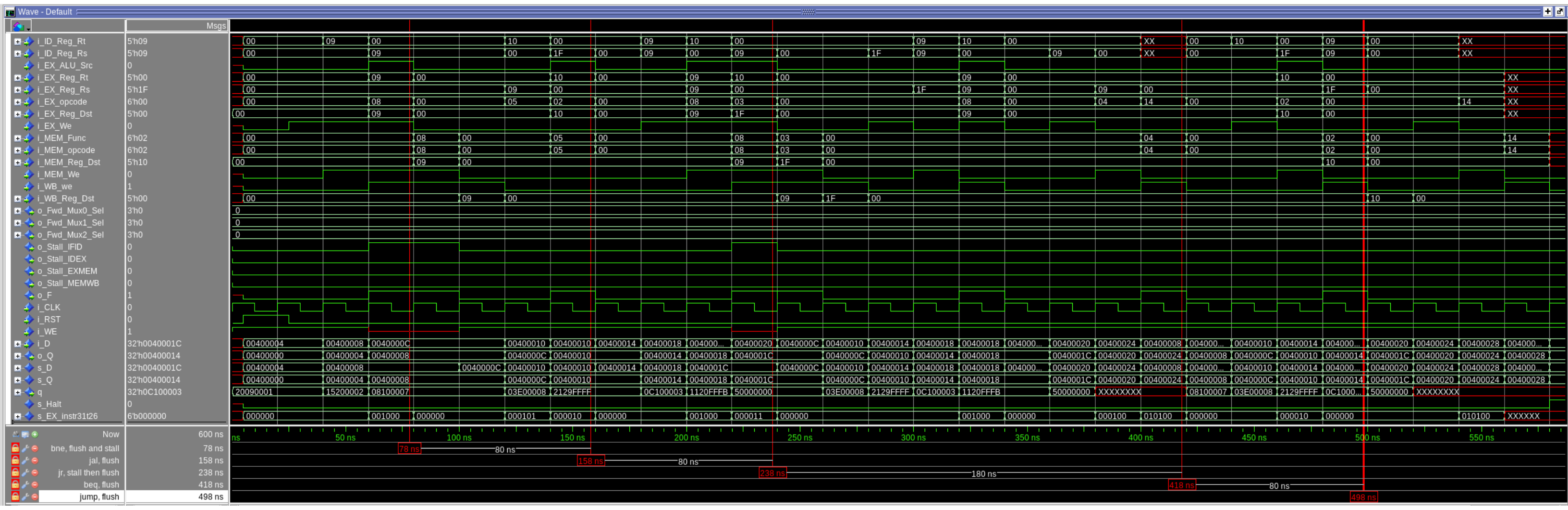


[2.e – i, ii, and iii] In your writeup, show the QuestaSim output for each of the following tests, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.



```
Wave - Default
                      Msgs
i_ID_Reg_Rt           5'h00
i_ID_Reg_Rs           5'h00
i_EX_ALU_Src          1
i_EX_Reg_Rt           5'h04
i_EX_Reg_Rs           5'h1E
i_EX_opcode           6'h2B
i_EX_Reg_Dst          5'h04
i_EX_We               0
i_MEM_Func            6'h25
i_MEM_opcode          6'b100101
i_MEM_Reg_Dst         5'h04
i_MEM_We              1
i_WB_we               0
i_WB_Reg_Dst          5'h00
o_Fwd_Mux0_Sel        3'h0
o_Fwd_Mux1_Sel        3'h0
o_Fwd_Mux2_Sel        3'b010
o_Stall_IFID          0
o_Stall_IDEX          0
o_Stall_EXMEM         0
o_Stall_MEMWB         0
o_Stall_PC            U

Now                   540 ns
lw fwd from MEM       356 ns
lh fwd from MEM       416 ns
lhu fwd from MEM      476 ns
```

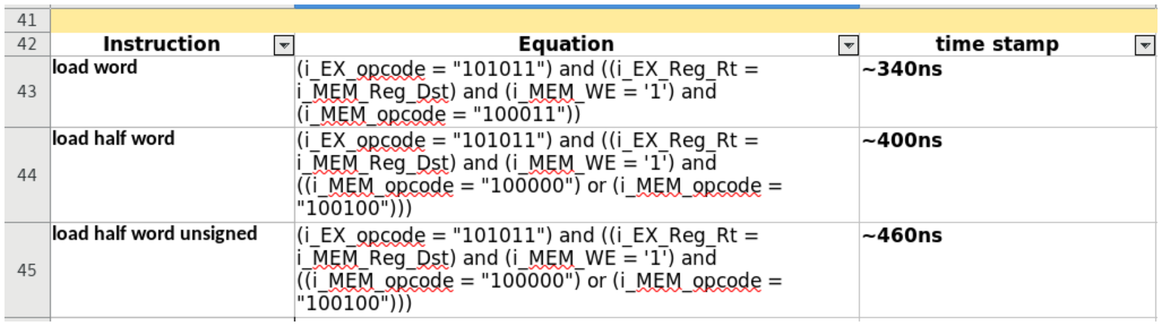
These forwards are correct because after each instruction that is mentioned in the annotations there is a sw function. So naturally, a forward would need to occur after each instruction of the types stated in the annotations.



	Msgs
i_ID_Reg_Rt	5'h09
i_ID_Reg_Rs	5'h09
i_EX_ALU_Src	0
i_EX_Reg_Rt	5'h00
i_EX_Reg_Rs	5'h1F
i_EX_opcode	6'h00
i_EX_Reg_Dst	5'h00
i_EX_We	0
i_MEM_Func	6'h02
i_MEM_opcode	6'h02
i_MEM_Reg_Dst	5'h10
i_MEM_We	0
i_WB_we	1
i_WB_Reg_Dst	5'h00
o_Fwd_Mux0_Sel	3'h0
o_Fwd_Mux1_Sel	3'h0
o_Fwd_Mux2_Sel	3'h0
o_Stall_IFID	0
o_Stall_IDEX	0
o_Stall_EXMEM	0
o_Stall_MEMWB	0
o_F	1
i_CLK	0
i_RST	0
i_WE	1
i_D	32'h0040001C
o_Q	32'h00400014
s_D	32'h0040001C
s_Q	32'h00400014
q	32'h0C100003
s_Halt	0
s_EX_instr31t26	6'b000000
Now	600 ns
bne, flush and stall	78 ns
jal, flush	158 ns
jr, stall then flush	238 ns
beq, flush	418 ns
jump, flush	498 ns

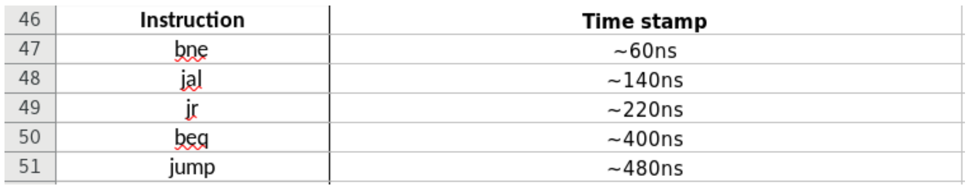
Seen above is the control hazard avoidance. As you can see when any kind of control instruction is executed a flush happens, and it happens after a stall if necessary for the variables being used in the control instruction.

[2.e.i] Create a spreadsheet to track these cases and justify the coverage of your testing approach. Include this spreadsheet in your report as a table.



41			
42	**Instruction**	**Equation**	**time stamp**
43	load word	(i_EX_opcode = "101011") and ((i_EX_Reg_Rt = i_MEM_Reg_Dst) and (i_MEM_WE = '1') and (i_MEM_opcode = "100011"))	**~340ns**
44	load half word	(i_EX_opcode = "101011") and ((i_EX_Reg_Rt = i_MEM_Reg_Dst) and (i_MEM_WE = '1') and ((i_MEM_opcode = "100000") or (i_MEM_opcode = "100100")))	**~400ns**
45	load half word unsigned	(i_EX_opcode = "101011") and ((i_EX_Reg_Rt = i_MEM_Reg_Dst) and (i_MEM_WE = '1') and ((i_MEM_opcode = "100000") or (i_MEM_opcode = "100100")))	**~460ns**

[2.e.ii] Create a spreadsheet to track these cases and justify the coverage of your testing approach. Include this spreadsheet in your report as a table.



46	**Instruction**	**Time stamp**
47	bne	~60ns
48	jal	~140ns
49	jr	~220ns
50	beq	~400ns
51	jump	~480ns

[2.f] report the maximum frequency your hardware-scheduled pipelined processor can run at and determine what your critical path is (specify each module/entity/component that this path goes through).

The maximum frequency of the hardware-scheduled pipelined processor is 47.39mhz. And just like the software_scheduled pipelined processor, the critical path goes through DMEM/WordShifter/MEM_WB_Pipe because the memory stage is the longest stage and that's the longes path through that stage.