CprE 381: Computer Organization and Assembly-Level Programming

Project Part 2 Report

Team Members: Anthony Manschula
Henry Shires

Project Teams Group #: TermProj_3_02

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

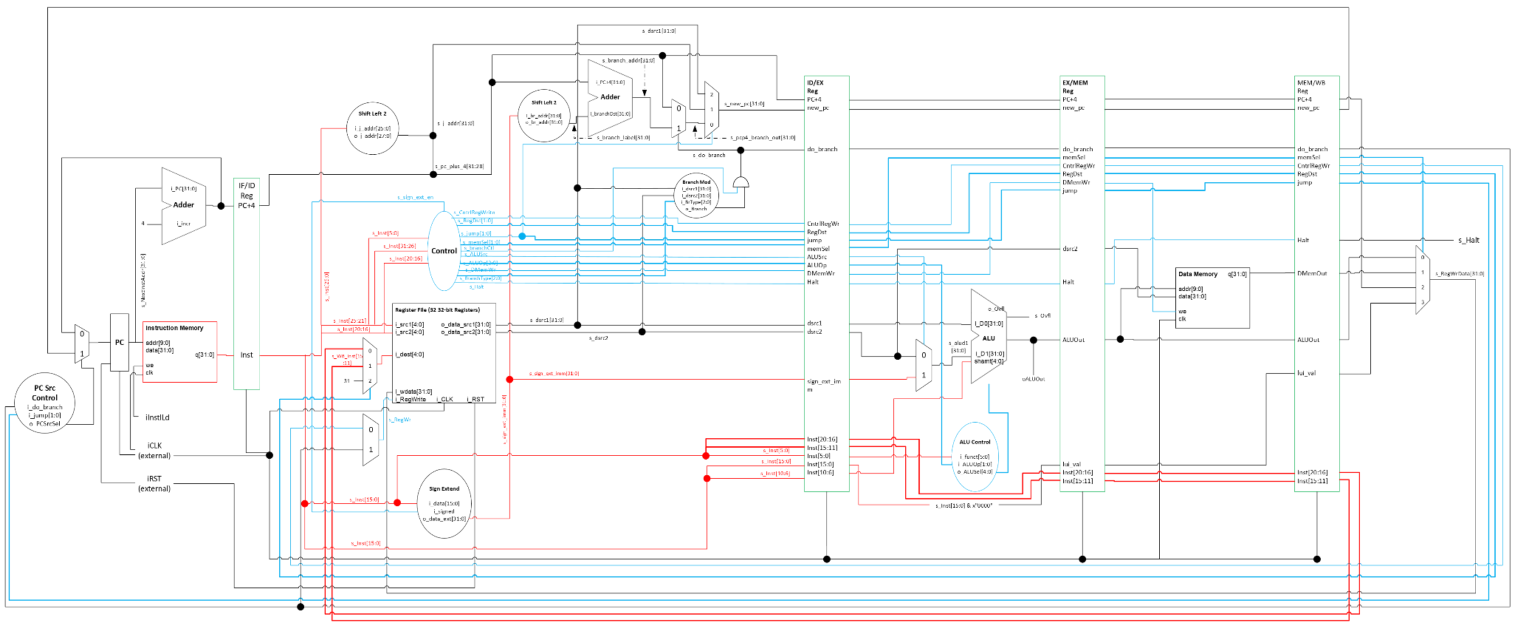
[1.a] Come up with a global list of the datapath values and control signals that are required during each pipeline stage.

IF/ID: PC+4, Instruction
ID/EX:
- Old: PC+4
- New: new_PC, do_branch, CtrlRegWrite, RegDest, jump, memSel, ALUSrc, ALUOp, DMemWrite, halt, dsrc1, dsrc2, sign_ext_imm, Inst[20:16], Inst[15:11], Inst[5:0], Inst[15:0], Inst[10:6]

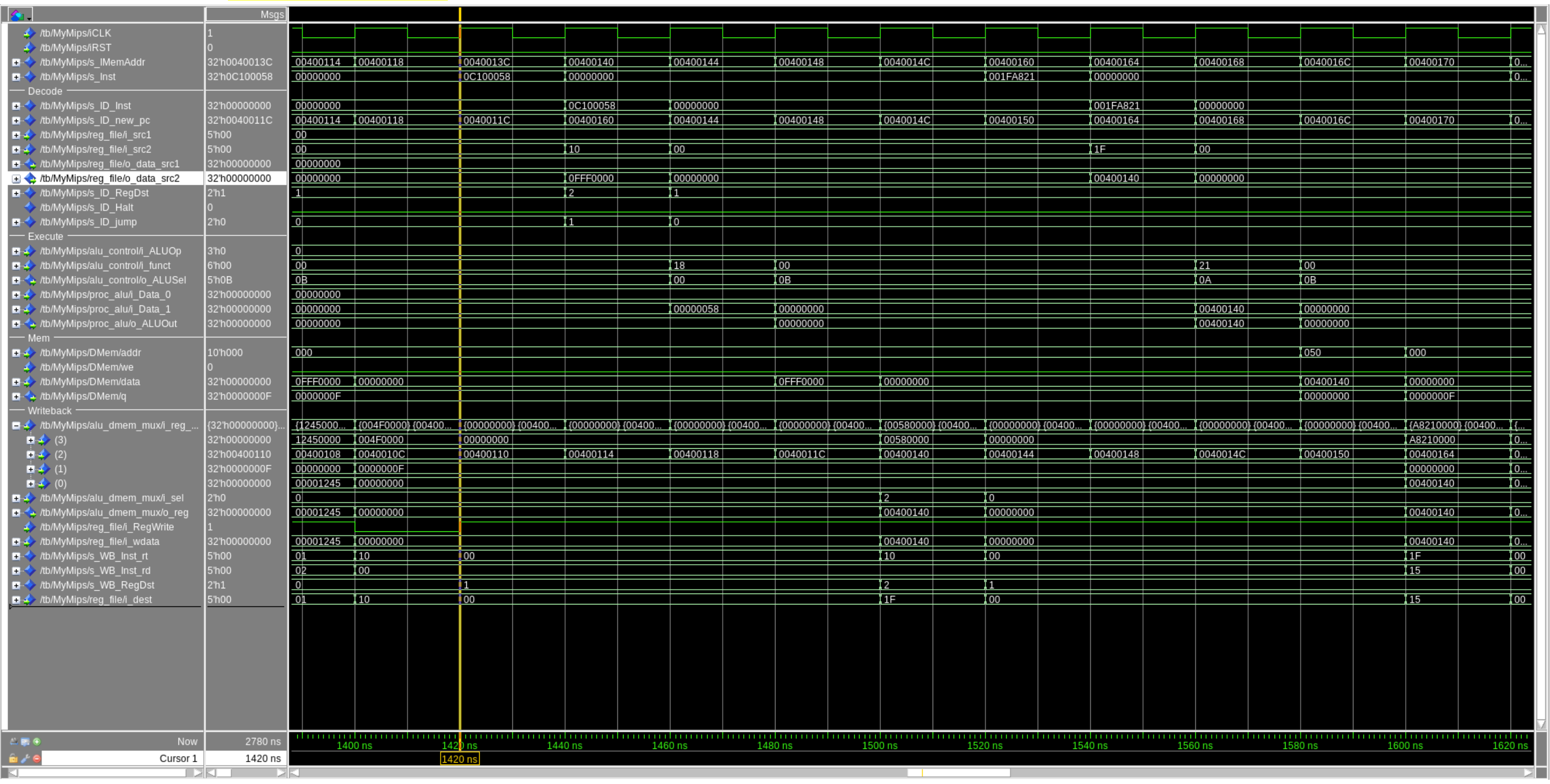
EX/MEM:
- Old: PC+4, new_PC, do_branch, CtrlRegWrite, RegDst, DMemWrite, jump, memSel, halt, dsrc2, Inst[20:16], Inst[15:11]
- New: lui_val, ALUOut

MEM/WB
- Old: PC+4, new_pc, do_branch, memSel, CtrlRegWrite, RegDst, jump, halt, ALUOut, lui_val, Inst[20:16], Inst[15:11]
- New: DMemOut, s_RegWrData

[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.

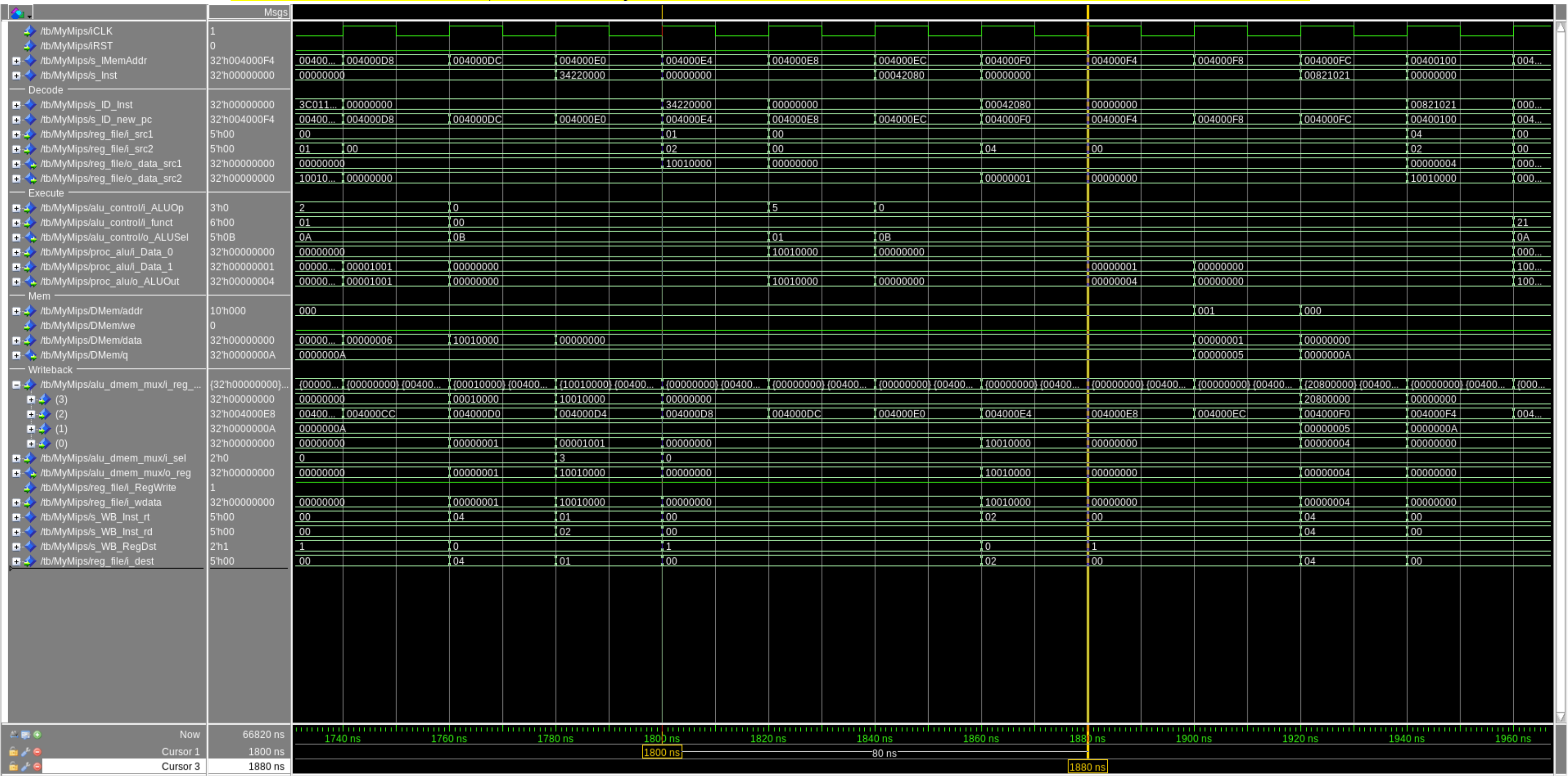


	Msgs
/tb/MyMips/iCLK	1
/tb/MyMips/iRST	0
/tb/MyMips/s_IMemAddr	32'h0040013C
/tb/MyMips/s_Inst	32'h0C100058
Decode	
/tb/MyMips/s_ID_Inst	32'h00000000
/tb/MyMips/s_ID_new_pc	32'h0040011C
/tb/MyMips/reg_file/i_src1	5'h00
/tb/MyMips/reg_file/i_src2	5'h00
/tb/MyMips/reg_file/o_data_src1	32'h00000000
/tb/MyMips/reg_file/o_data_src2	32'h00000000
/tb/MyMips/s_ID_RegDst	2'h1
/tb/MyMips/s_ID_Halt	0
/tb/MyMips/s_ID_jump	2'h0
Execute	
/tb/MyMips/alu_control/i_ALUOp	3'h0
/tb/MyMips/alu_control/i_funct	6'h00
/tb/MyMips/alu_control/o_ALUSel	5'h0B
/tb/MyMips/proc_alu/i_Data_0	32'h00000000
/tb/MyMips/proc_alu/i_Data_1	32'h00000000
/tb/MyMips/proc_alu/o_ALUOut	32'h00000000
Mem	
/tb/MyMips/DMem/addr	10'h000
/tb/MyMips/DMem/we	0
/tb/MyMips/DMem/data	32'h00000000
/tb/MyMips/DMem/q	32'h0000000F
Writeback	
/tb/MyMips/alu_dmem_mux/i_reg_...	{32'h00000000}...
(3)	32'h00000000
(2)	32'h00400110
(1)	32'h0000000F
(0)	32'h00000000
/tb/MyMips/alu_dmem_mux/i_sel	2'h0
/tb/MyMips/alu_dmem_mux/o_reg	32'h00000000
/tb/MyMips/reg_file/i_RegWrite	1
/tb/MyMips/reg_file/i_wdata	32'h00000000
/tb/MyMips/s_WB_Inst_rt	5'h00
/tb/MyMips/s_WB_Inst_rd	5'h00
/tb/MyMips/s_WB_RegDst	2'h1
/tb/MyMips/reg_file/i_dest	5'h00

Now 2780 ns
Cursor 1 1420 ns

- The cursor is placed at the point at which a jal instruction is decoded. Over the next four clock cycles, we can see the instruction make its way through the pipeline. We move to the decode stage in the second cycle (1440ns), where some important control signals are generated, such as RegDst and jump. We can also see that the branch address is calculated in this stage, as s_ID_new_pc is not equal to s_IMemAddr + 4. The execute stage does not do anything, since a jal instruction does not require the use of the ALU, and nor does the memory stage. When it enters the writeback stage, we see that the address of the instruction following the jal is selected by the mux, and that the proper register destination is selected (in this case, $ra for a jal) and written to.

[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.



	Msgs
/tb/MyMips/iCLK	1
/tb/MyMips/iRST	0
/tb/MyMips/s_IMemAddr	32'h004000F4
/tb/MyMips/s_Inst	32'h00000000
Decode	
/tb/MyMips/s_ID_Inst	32'h00000000
/tb/MyMips/s_ID_new_pc	32'h004000F4
/tb/MyMips/reg_file/i_src1	5'h00
/tb/MyMips/reg_file/i_src2	5'h00
/tb/MyMips/reg_file/o_data_src1	32'h00000000
/tb/MyMips/reg_file/o_data_src2	32'h00000000
Execute	
/tb/MyMips/alu_control/i_ALUOp	3'h0
/tb/MyMips/alu_control/i_funct	6'h00
/tb/MyMips/alu_control/o_ALUSel	5'h0B
/tb/MyMips/proc_alu/i_Data_0	32'h00000000
/tb/MyMips/proc_alu/i_Data_1	32'h00000001
/tb/MyMips/proc_alu/o_ALUOut	32'h00000004
Mem	
/tb/MyMips/DMem/addr	10'h000
/tb/MyMips/DMem/we	0
/tb/MyMips/DMem/data	32'h00000000
/tb/MyMips/DMem/q	32'h0000000A
Writeback	
/tb/MyMips/alu_dmem_mux/i_reg_...	{32'h00000000}...
(3)	32'h00000000
(2)	32'h004000E8
(1)	32'h0000000A
(0)	32'h00000000
/tb/MyMips/alu_dmem_mux/i_sel	2'h0
/tb/MyMips/alu_dmem_mux/o_reg	32'h00000000
/tb/MyMips/reg_file/i_RegWrite	1
/tb/MyMips/reg_file/i_wdata	32'h00000000
/tb/MyMips/s_WB_Inst_rt	5'h00
/tb/MyMips/s_WB_Inst_rd	5'h00
/tb/MyMips/s_WB_RegDst	2'h1
/tb/MyMips/reg_file/i_dest	5'h00
Now	66820 ns
Cursor 1	1800 ns
Cursor 3	1880 ns

- In the waveform, Two cursors are placed within an iteration of BubbleSort where less than the maximum number of NOPs for our pipeline design (3 nop instructions) are utilized to avoid data hazards while executing the minimum number of instructions necessary. According to the opcode values from I-Mem in the fetch stage at 1800 ns, the instruction sequence until 1880 ns is `ori -> nop -> nop -> sll`. This `ori` corresponds to the decoded `lasw` pseudo instruction (see Example 1 below) writing the address of the data array to register `$2`. Thus, the waveform correctly displays the correct instructions, PC register updates, and required values by each instruction. Additionally, the array address 0x10010000 can be seen on the decode stage of the `lasw` instruction and later seen on the writeback stage two cycles later, showing the data hazard was avoided and the address was properly written to register `$2`.

- Including the one in the waveform above, our program contains three examples of using less than the maximum number of no-ops between data dependencies to prevent data hazards. In our specific design of the software-scheduled pipeline, we are not able to utilize less than the maximum number of no-ops for *control* hazards, as the order of branch and jump instructions in our program cannot be adjusted in a way to enable a solution with less no-ops. Additionally, the maximum no-ops required to prevent control hazards is *four* instead of three for data hazards, as the control flow instructions take an extra cycle to propagate signals to the PC register in order to properly branch or jump to the correct next instruction.

Example 1: Data flow lines 66-69: Here the `lasw` instruction can be used as one of three instructions executed before `sll` is fetched, since `sll` has a read dependency of `$4` which is written by the `addiu` instruction. This is more efficient than using three no-ops, requiring only two.

```
addiu   $4,$2,1
lasw    $2, array
nop
nop
sll     $4,$4,2
```

Example 2: Data flow lines 113-117: Similarly to example 1, the `lasw` instruction can be used as one of three instructions executed before `sll` is fetched, since `sll` has a read dependency of `$3` which is written by the `addiu` instruction. This requires only two additional no-ops.

```
addiu   $3,$2,1
lasw    $2, array
nop
nop
sll     $3,$3,2
```

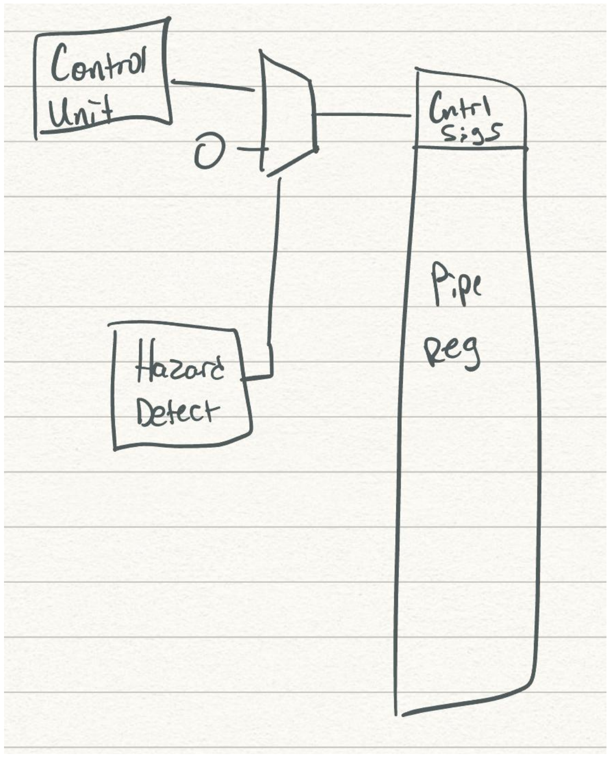
Example 3: Data flow lines 124-128: Here the `lw` and `lasw` instructions have been reordered to execute before `sll` is fetched, since `sll` has a read dependency of `$4` which is written by the first `lw` instruction. The `lw` and `lasw` don't have any dependencies, so they can execute individually and both before the `sll` dependency. Overall, this requires only one additional no-op.

```
lw      $4,12($fp)
lw      $3,0($2)
lasw  $2, array
nop
sll     $4,$4,2
```

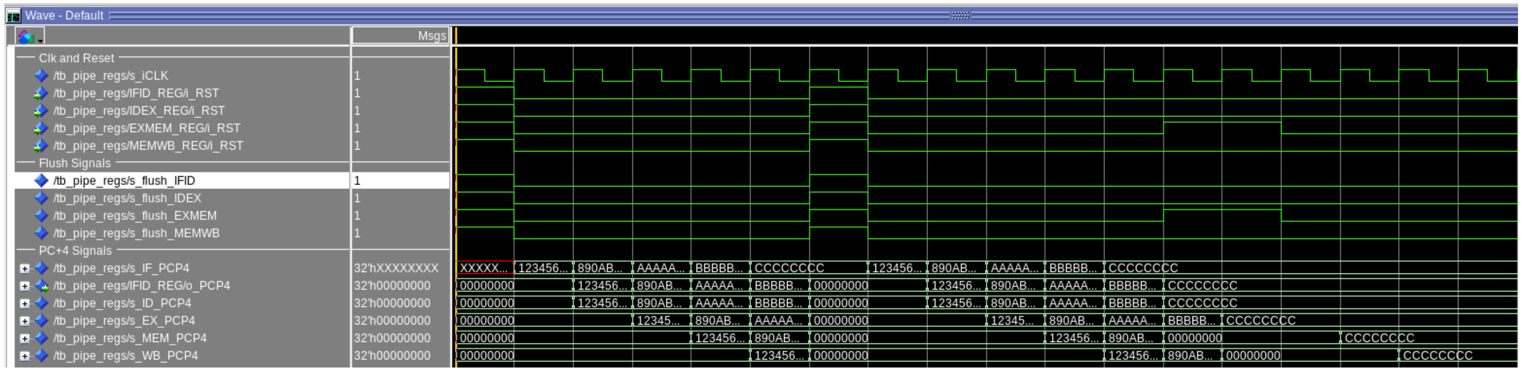
[1.d] report the maximum frequency your software-scheduled pipelined processor can run at and determine what your critical path is (specify each module/entity/component that this path goes through).

- Our reported frequency is **50.86 MHz**
- The critical path is **ID/EX Register > ALU Control Unit > ALU > EX/MEM Register**
- Given the software scheduled pipeline is a requirement (instead of implementing hardware scheduled hazard detection), the most beneficial component to optimize here would be the ALU, as it takes the most time in the critical path (15.56 ns), which is located in the Execute Stage
- Since this processor design is pipelined, or each stage has its own path each cycle, the critical path is located in the slowest STAGE. So in this case, our slowest stage is Execute with the ALU taking the longest to propagate signals. This is reasonable, as our hazard detection hardware is not in this design, although it's interesting Decode is faster despite the PC add and jump logic executing with the register file there

[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.



```
Clk and Reset
/tb_pipe_regs/s_iCLK                1
/tb_pipe_regs/IFID_REG/i_RST        1
/tb_pipe_regs/IDEX_REG/i_RST        1
/tb_pipe_regs/EXMEM_REG/i_RST       1
/tb_pipe_regs/MEMWB_REG/i_RST       1
Flush Signals
/tb_pipe_regs/s_flush_IFID          1
/tb_pipe_regs/s_flush_IDEX          1
/tb_pipe_regs/s_flush_EXMEM         1
/tb_pipe_regs/s_flush_MEMWB         1
PC+4 Signals
/tb_pipe_regs/s_IF_PCP4             32'hXXXXXXXX
/tb_pipe_regs/IFID_REG/o_PCP4       32'h00000000
/tb_pipe_regs/s_ID_PCP4             32'h00000000
/tb_pipe_regs/s_EX_PCP4             32'h00000000
/tb_pipe_regs/s_MEM_PCP4            32'h00000000
/tb_pipe_regs/s_WB_PCP4             32'h00000000
```

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

- Instructions produce values on **s_EX_ALUOut** signal**:** add, addi, addiu, addu, and, andi, nor, xor, xori, or, ori, slt, slti, sll, srl, sra, sw, sub, subu
- Instructions produce values on **s_DMemOut** signal: lw
- Instructions produce values on **s_EX_lui_val**: lui

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

- Instructions consume values on **s_Inst_Rd, s_Inst_Rs, s_Inst_Rt, s_dsrc1, s_dsrc2, s_Inst[10:6] (R-type):** add, addu, and, nor, xor, or, slt, sll, srl, sra, sub, subu
- Instructions consume values on **s_Inst_Rd, s_Inst_Rs, s_Inst_Rt, s_dsrc1, s_dsrc2, s_sign_ext_imm (I-type):** addi, addiu, andi, lui, lw, xori, ori, slti, sw

[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.

- Forwarding Dependencies:
  - **EX/MEM s_Inst_Rd = ID/EX s_Inst_Rs, EX/MEM s_Inst_Rd = EX/MEM s_Inst_Rt:** Forward *s_ALUOut* back to input of Execute stage of next instruction
- Stalling Dependencies (Load-use dependencies):
  - **MEM/WB s_Inst_Rd = ID/EX s_Inst_Rs, MEM/WB s_Inst_Rd = ID/EX s_Inst_Rt:** *s_DMemOut* needs stalls and a bubble in the pipeline until it can propagate to the register file

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

IF/ID: PC+4, Instruction
ID/EX:
- Old: PC+4
- New: new_PC, do_branch, branchCtrl, CtrlRegWrite, RegDest, jump, memSel, ALUSrc, ALUOp, DMemWrite, halt, dsrc1, dsrc2, sign_ext_imm, Instr[25:21] (rs), Instr[20:16] (rt), Instr[15:11] (rd), Instr[15:0] (lui), Instr[10:6] (shamt), Instr[5:0] (function)

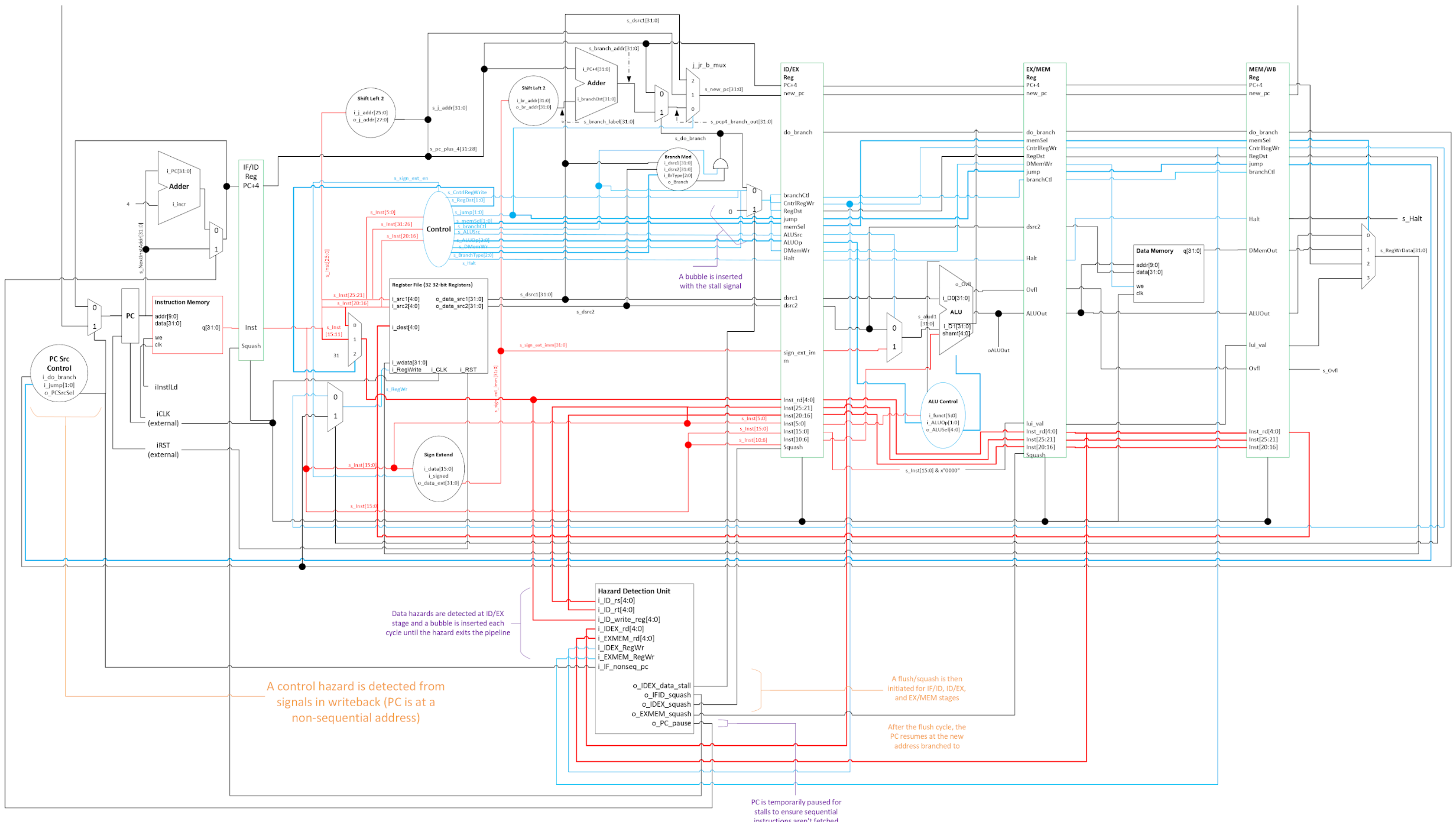
EX/MEM:
- Old: PC+4, new_PC, do_branch, branchCtrl, CtrlRegWrite, RegDst, jump, memSel, halt, dsrc2, Instr[20:16] (rt), Instr[15:11] (rd), Inst[25:21] (rs)
- New: lui_val, overflow, ALUOut

MEM/WB
- Old: PC+4, new_pc, do_branch, branchCtrl, memSel, CtrlRegWrite, RegDst, jump, ALUOut, halt, Inst[25:21] (rs), Instr[20:16] (rt), Instr[15:11] (rd), lui_val, overflow
- New: DMemOut, s_RegWrData

[2.c.i] list all instructions that may result in a non-sequential PC update and in which pipeline stage that update occurs.
Non-sequential PC updates are calculated in Decode, but this new PC value is not available to use until it is in the Writeback stage for the following instructions: **beq, bne, j, jal, jr, bgez, bgezal, bgtz, blez, bltzal, bltz**. The PC fetches this new PC value in Fetch from a MUX output

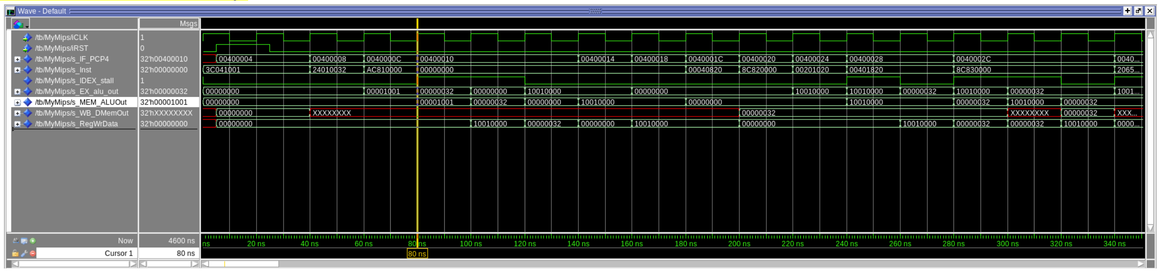
[2.c.ii] For these instructions, list which stages need to be stalled and which stages need to be squashed/flushed relative to the stage each of these instructions is in.
- Stages stalled: **ID/EX**
- Stages squashed/flushed: **IF/ID, ID/EX, EX/MEM**

[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.


IF/ID Reg
ID/EX Reg
EX/MEM Reg
MEM/WB Reg
Adder
Shift Left 2
Control
Register File (32 32-bit Registers)
Instruction Memory
PC
PC Src Control
Sign Extend
Branch Mod
ALU
ALU Control
Data Memory
s_Halt
s_Ovfl
iInstLd
iCLK (external)
iRST (external)
A bubble is inserted with the stall signal
Hazard Detection Unit
i_ID_rs[4:0]
i_ID_rt[4:0]
i_ID_write_reg[4:0]
i_IDEX_rd[4:0]
i_EXMEM_rd[4:0]
i_IDEX_RegWr
i_EXMEM_RegWr
i_IF_nonseq_pc
o_IDEX_data_stall
o_IFID_squash
o_IDEX_squash
o_EXMEM_squash
o_PC_pause
Data hazards are detected at ID/EX stage and a bubble is inserted each cycle until the hazard exits the pipeline
A control hazard is detected from signals in writeback (PC is at a non-sequential address)
A flush/squash is then initiated for IF/ID, ID/EX, and EX/MEM stages
After the flush cycle, the PC resumes at the new address branched to
PC is temporarily paused for stalls to ensure sequential instructions aren't fetched

For better viewing of annotations, please view the additional diagram PDF included in this submission.

[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
/tb/MyMips/iCLK 1
/tb/MyMips/iRST 0
/tb/MyMips/s_IF_PCP4 32'h00400010
/tb/MyMips/s_Inst 32'h00000000
/tb/MyMips/s_IDEX_stall 1
/tb/MyMips/s_EX_alu_out 32'h00000032
/tb/MyMips/s_MEM_ALUOut 32'h00001001
/tb/MyMips/s_WB_DMemOut 32'hXXXXXXXX
/tb/MyMips/s_RegWrData 32'h00000000
Now 4600 ns
Cursor 1 80 ns

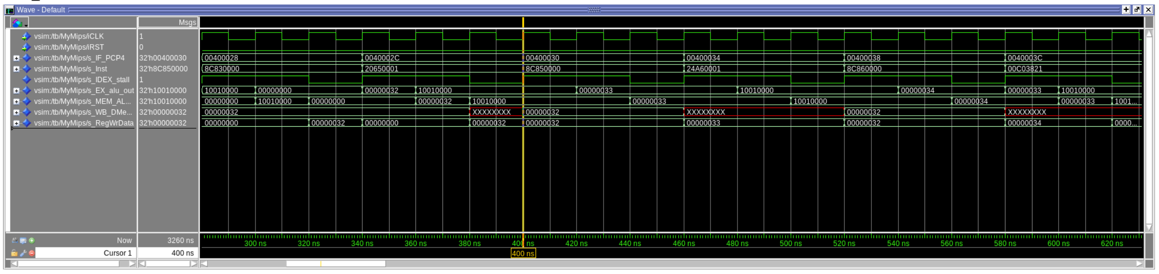
/data_hazard_tests/**data.s** - This combines tests from id-ex.s, ex-mem.s, and mem-wb.s. These tests introduce ID/EX, EX/MEM, and MEM/WB Rd and MemRead data hazards in the pipeline. The test works by using a combination of data dependencies used sequentially (no instructions between the dependency), after one instruction in between the dependency, and with `lw` instructions. Additionally the use of the dependency alternates between the rs field and rt field to ensure the hazard is detected for both fields.

In the above combined test, the output works as expected. The ID/EX stall signal goes high on the cycle detecting a hazard. In this case, the first example this occurs in MIPS is:

addiu $1, $0, 50
sw $1, 0($4)

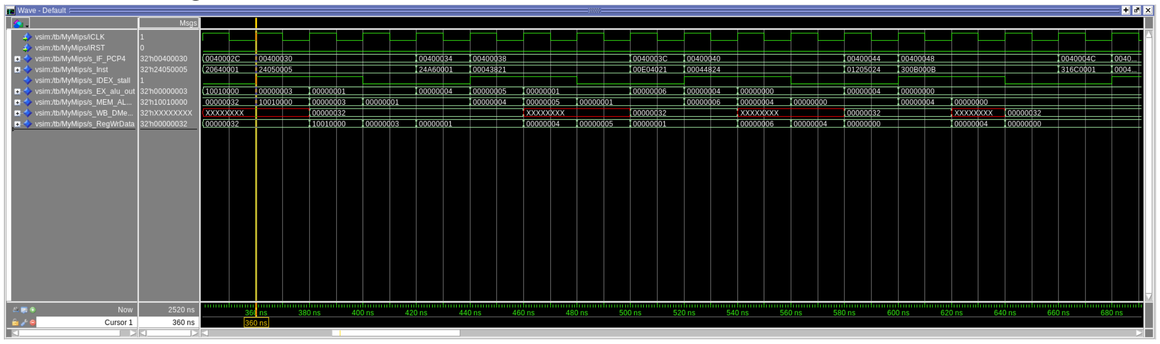
The sw instruction has a dependency on addiu writing a value to $1. Here, the PC+4 signal is paused, no instruction is fetched, and the ALU continues processing values to the right of the stalled pipeline register. The stall signal remains high until the hazard clears the pipeline and the instruction fetching resumes. Later on DMemOut, when a `lw` reads the same memory location, the value 0x32 = 50 is read, which means the pipeline properly wrote the value at the end of the instruction execution from the previous `addiu` and the dependency was properly stalled for.

Separated data hazard tests:

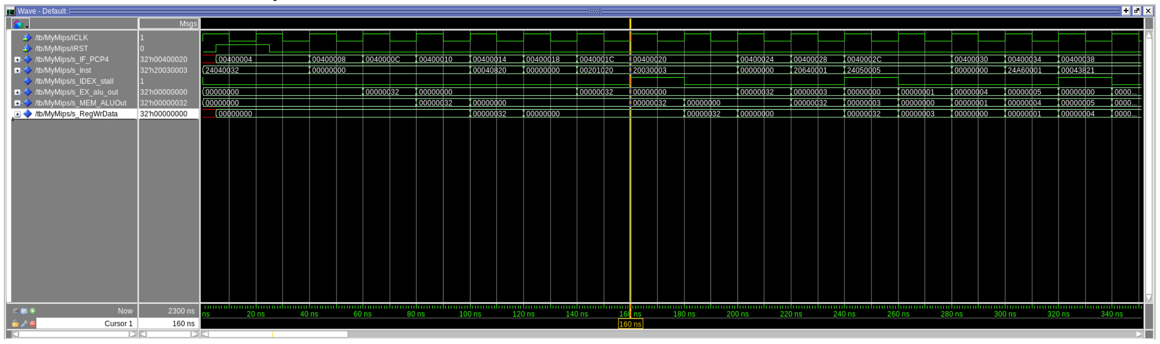

Wave - Default
Msgs
vsim:/tb/MyMips/iCLK 1
vsim:/tb/MyMips/iRST 0
vsim:/tb/MyMips/s_IF_PCP4 32'h00400030
vsim:/tb/MyMips/s_Inst 32'h8C850000
vsim:/tb/MyMips/s_IDEX_stall 1
vsim:/tb/MyMips/s_EX_alu_out 32'h10010000
vsim:/tb/MyMips/s_MEM_AL... 32'h10010000
vsim:/tb/MyMips/s_WB_DMe... 32'h00000032
vsim:/tb/MyMips/s_RegWrData 32'h00000032
Now 3260 ns
Cursor 1 400 ns

Load-use hazards - This works the same as the explanation given above

EX/MEM Rd hazards - The value written to on the previous instruction is correctly read in the following instruction from the Rs or Rt fields.

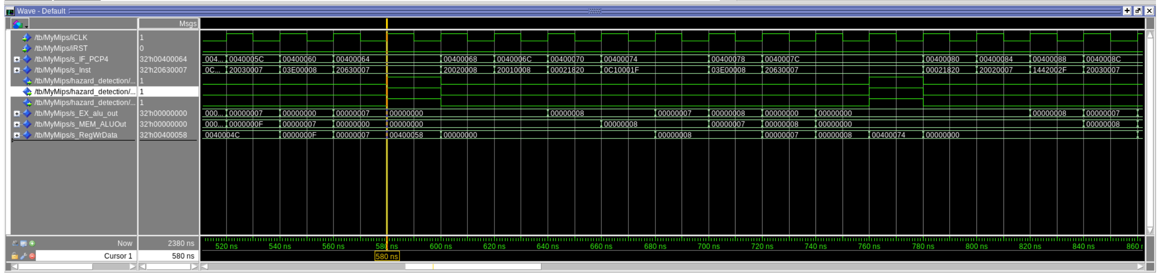

Wave - Default
vsim:/tb/MyMips/iCLK
vsim:/tb/MyMips/iRST
vsim:/tb/MyMips/s_IF_PCP4
vsim:/tb/MyMips/s_Inst
vsim:/tb/MyMips/s_IDEX_stall
vsim:/tb/MyMips/s_EX_alu_out
vsim:/tb/MyMips/s_MEM_AL...
vsim:/tb/MyMips/s_WB_DMe...
vsim:/tb/MyMips/s_RegWrData
Now 2520 ns
Cursor 1 360 ns

MEM/WB Rd hazards - The value written to on two instructions before the current instruction is correctly read on the current in the Rs or Rt fields.



Here, one nop is used to separate the instructions by one clock cycle, to ensure the hazard is still detected. Our output properly detects it and sets the stall signal high for a shorter amount of time than the EX/MEM test case, which is expected, since the dependency is spaced apart by two stages instead of one.

Control Hazards:



/control_hazard_tests/**control.s** - This program combines all of the branch and jump instruction individual tests into one large branching and control flow test program. These tests introduce every type of control hazard in the pipeline. The test works by using a combination of branch dependencies used sequentially (no instructions between the dependency), after one instruction in between the dependency, and two instructions between. Additionally the use of the dependency alternates between the branch condition itself and the instructions after the branch instruction. This exhaustively tests the pipeline's ability to flush itself or remove instructions no longer valid if a branch or jump is taken.

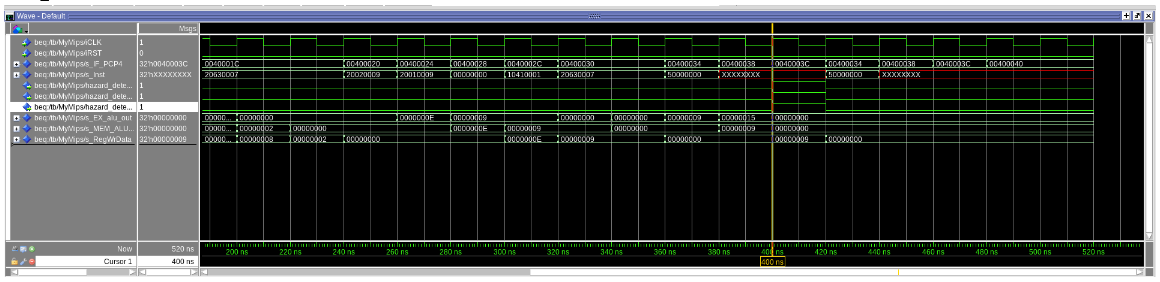
In the above combined test, the output works as expected. The flush signal for each pipeline register (ID/EX, EX/MEM, MEM/WB) goes high on the cycle detecting a hazard. In the above waveform screenshot, the example where this happens is the following in MIPS:

addi $2, $0, 7
jal two

Jal is an instruction that always jumps when executed. When the instruction is brought into fetch, the pipeline cannot act on the address to jump to until the writeback stage, since the $ra register must be written to and the PC must be updated. During that time the pipeline continues fetching instructions, causing a hazard of misinformation in the pipeline (additionally the PC has still been incrementing by 4 as seen in the screenshot). Here, the PC+4 signal is paused, no instruction is fetched, and the ALU continues processing values to the right of the stalled pipeline register. Once it reaches the writeback stage, the pipeline asserts the flush signals, wiping out the unneeded instructions after the jump, up until the jump clears and the PC matches the new line the program jumped to. After, the flush desserts and the pipeline resumes. In writeback, Immediately on RegWriteData, the jal instruction writes to $ra the current PC address to link back to if a jr $ra is executed later. As seen in the waveform, each of these occurrences means the pipeline properly stalled for the dependency and properly flushed any unnecessary instructions following the jump execution. These correct flushes also occur for every other control flow instruction included below:

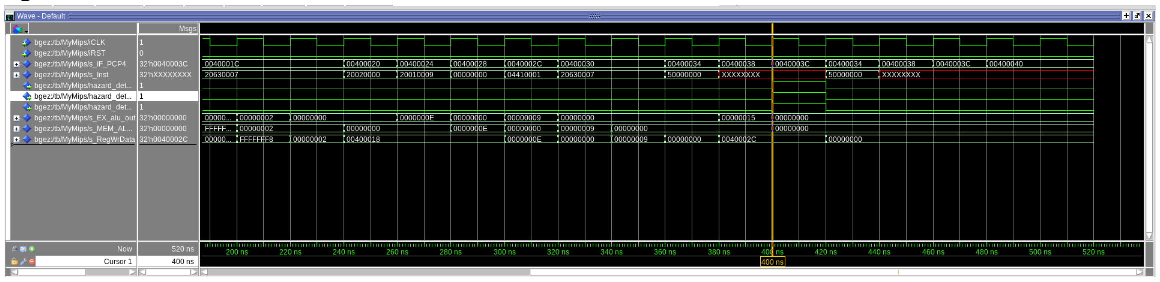
Separated control hazard tests:

beq



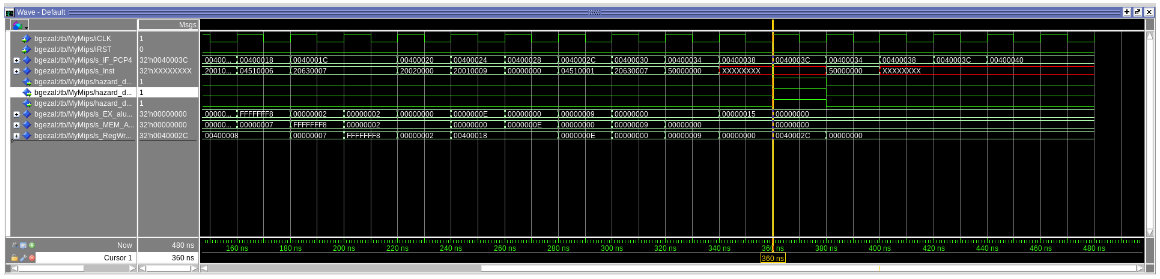
```
Wave - Default                          Msgs
beq:/tb/MyMips/iCLK                     1
beq:/tb/MyMips/iRST                     0
beq:/tb/MyMips/s_IF_PCP4                32'h0040003C
beq:/tb/MyMips/s_Inst                   32'hXXXXXXXX
beq:/tb/MyMips/hazard_dete...           1
beq:/tb/MyMips/hazard_dete...           1
beq:/tb/MyMips/hazard_dete...           1
beq:/tb/MyMips/s_EX_alu_out             32'h00000000
beq:/tb/MyMips/s_MEM_ALU...             32'h00000000
beq:/tb/MyMips/s_RegWrData              32'h00000009
Now                                     520 ns
Cursor 1                                400 ns
```

bgez

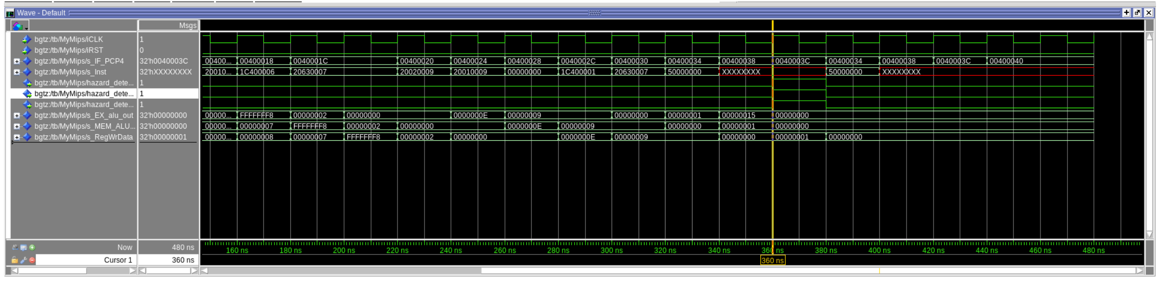


```
Wave - Default                          Msgs
bgez:/tb/MyMips/iCLK                    1
bgez:/tb/MyMips/iRST                    0
bgez:/tb/MyMips/s_IF_PCP4               32'h0040003C
bgez:/tb/MyMips/s_Inst                  32'hXXXXXXXX
bgez:/tb/MyMips/hazard_det...           1
bgez:/tb/MyMips/hazard_det...           1
bgez:/tb/MyMips/hazard_det...           1
bgez:/tb/MyMips/s_EX_alu_out            32'h00000000
bgez:/tb/MyMips/s_MEM_AL...             32'h00000000
bgez:/tb/MyMips/s_RegWrData             32'h0040002C
Now                                     520 ns
Cursor 1                                400 ns
```

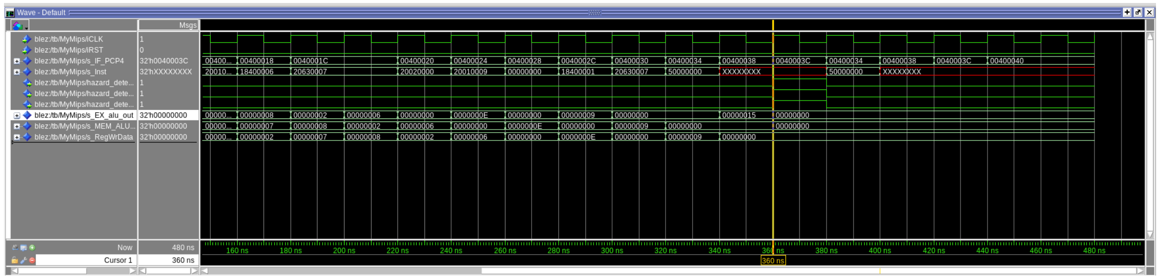
bgezal


Wave - Default
bgezal:/tb/MyMips/iCLK 1
bgezal:/tb/MyMips/iRST 0
bgezal:/tb/MyMips/s_IF_PCP4 32'h0040003C
bgezal:/tb/MyMips/s_Inst 32'hXXXXXXXX
bgezal:/tb/MyMips/hazard_d... 1
bgezal:/tb/MyMips/hazard_d... 1
bgezal:/tb/MyMips/hazard_d... 1
bgezal:/tb/MyMips/s_EX_alu... 32'h00000000
bgezal:/tb/MyMips/s_MEM_A... 32'h00000000
bgezal:/tb/MyMips/s_RegWr... 32'h0040002C
Now 480 ns
Cursor 1 360 ns

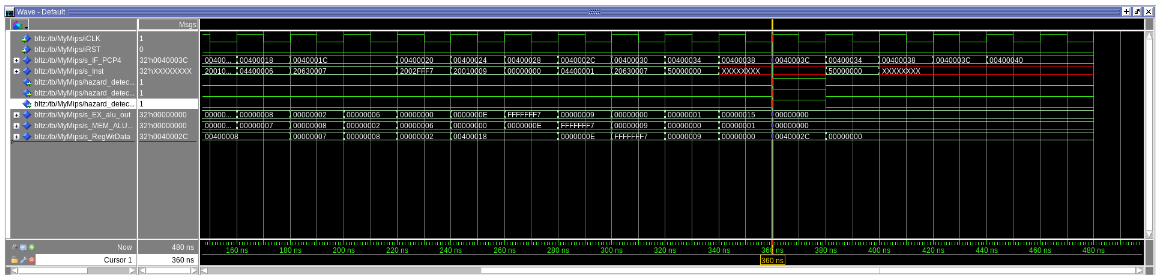
bgtz


Wave - Default
bgtz:/tb/MyMips/iCLK 1
bgtz:/tb/MyMips/iRST 0
bgtz:/tb/MyMips/s_IF_PCP4 32'h0040003C
bgtz:/tb/MyMips/s_Inst 32'hXXXXXXXX
bgtz:/tb/MyMips/hazard_dete... 1
bgtz:/tb/MyMips/hazard_dete... 1
bgtz:/tb/MyMips/hazard_dete... 1
bgtz:/tb/MyMips/s_EX_alu_out 32'h00000000
bgtz:/tb/MyMips/s_MEM_ALU... 32'h00000000
bgtz:/tb/MyMips/s_RegWrData 32'h00000001
Now 480 ns
Cursor 1 360 ns

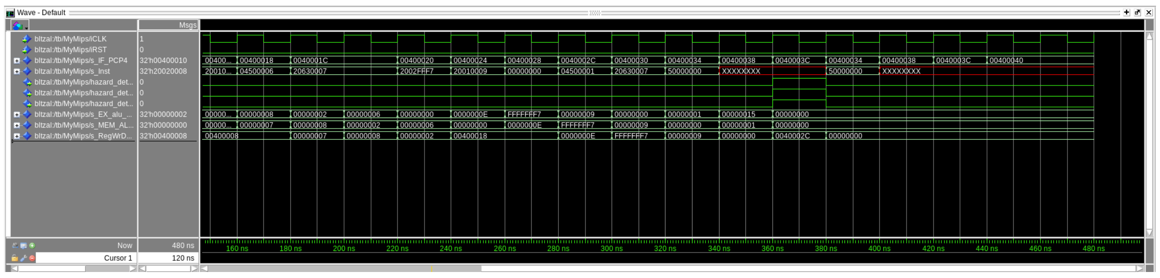
blez


Wave - Default
blez:/tb/MyMips/iCLK 1
blez:/tb/MyMips/iRST 0
blez:/tb/MyMips/s_IF_PCP4 32'h0040003C
blez:/tb/MyMips/s_Inst 32'hXXXXXXXX
blez:/tb/MyMips/hazard_dete... 1
blez:/tb/MyMips/hazard_dete... 1
blez:/tb/MyMips/hazard_dete... 1
blez:/tb/MyMips/s_EX_alu_out 32'h00000000
blez:/tb/MyMips/s_MEM_ALU... 32'h00000000
blez:/tb/MyMips/s_RegWrData 32'h00000000
Now 480 ns
Cursor 1 360 ns

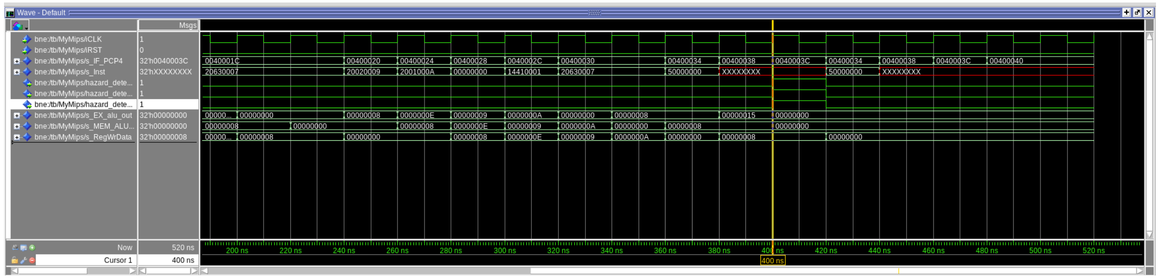
bltz


Wave - Default
bltz:/tb/MyMips/iCLK 1
bltz:/tb/MyMips/iRST 0
bltz:/tb/MyMips/s_IF_PCP4 32'h0040003C
bltz:/tb/MyMips/s_Inst 32'hXXXXXXXX
bltz:/tb/MyMips/hazard_detec... 1
bltz:/tb/MyMips/hazard_detec... 1
bltz:/tb/MyMips/hazard_detec... 1
bltz:/tb/MyMips/s_EX_alu_out 32'h00000000
bltz:/tb/MyMips/s_MEM_ALU... 32'h00000000
bltz:/tb/MyMips/s_RegWrData 32'h0040002C
Now 480 ns
Cursor 1 360 ns

bltzal

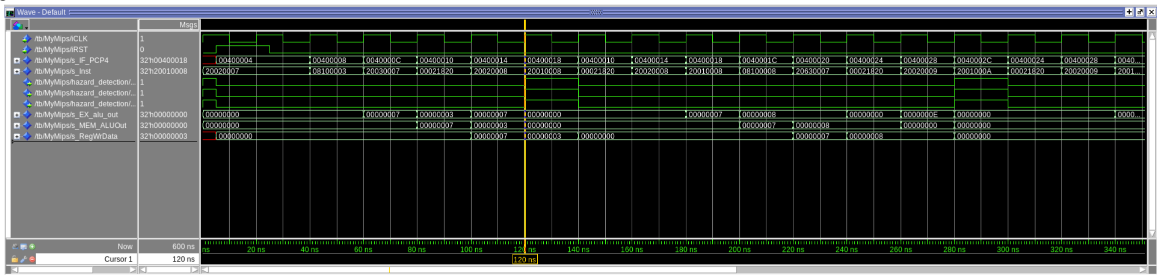

Wave - Default
bltzal:/tb/MyMips/iCLK 1
bltzal:/tb/MyMips/iRST 0
bltzal:/tb/MyMips/s_IF_PCP4 32'h00400010
bltzal:/tb/MyMips/s_Inst 32'h20020008
bltzal:/tb/MyMips/hazard_det... 0
bltzal:/tb/MyMips/hazard_det... 0
bltzal:/tb/MyMips/hazard_det... 0
bltzal:/tb/MyMips/s_EX_alu_... 32'h00000002
bltzal:/tb/MyMips/s_MEM_AL... 32'h00000000
bltzal:/tb/MyMips/s_RegWrD... 32'h00400008
Now 480 ns
Cursor 1 120 ns

ne



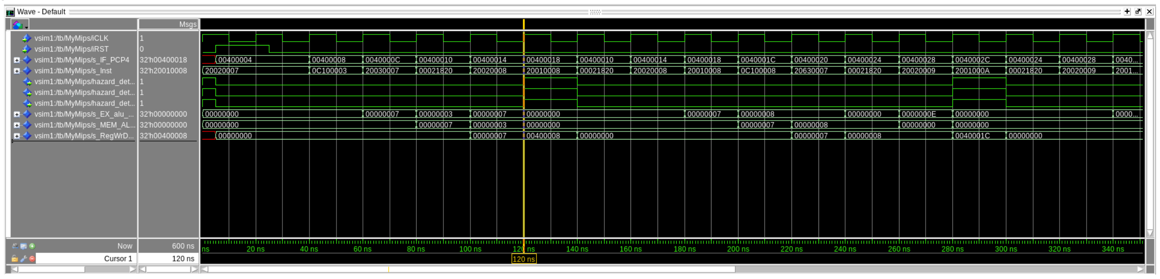
```
Wave - Default
                                    Msgs
bne:/tb/MyMips/iCLK                 1
bne:/tb/MyMips/iRST                 0
bne:/tb/MyMips/s_IF_PCP4            32'h0040003C
bne:/tb/MyMips/s_Inst               32'hXXXXXXXX
bne:/tb/MyMips/hazard_dete...       1
bne:/tb/MyMips/hazard_dete...       1
bne:/tb/MyMips/hazard_dete...       1
bne:/tb/MyMips/s_EX_alu_out         32'h00000000
bne:/tb/MyMips/s_MEM_ALU...         32'h00000000
bne:/tb/MyMips/s_RegWrData          32'h00000008
Now                                 520 ns
Cursor 1                            400 ns
```

j



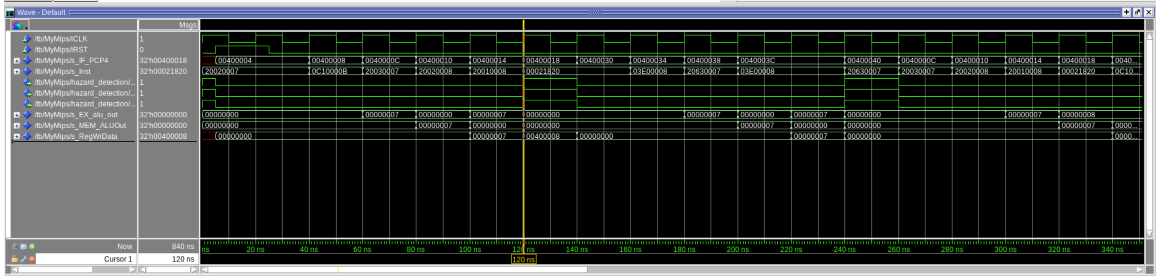
```
Wave - Default
                                    Msgs
/tb/MyMips/iCLK                     1
/tb/MyMips/iRST                     0
/tb/MyMips/s_IF_PCP4                32'h00400018
/tb/MyMips/s_Inst                   32'h20010008
/tb/MyMips/hazard_detection/...     1
/tb/MyMips/hazard_detection/...     1
/tb/MyMips/hazard_detection/...     1
/tb/MyMips/s_EX_alu_out             32'h00000000
/tb/MyMips/s_MEM_ALUOut             32'h00000000
/tb/MyMips/s_RegWrData              32'h00000003
Now                                 600 ns
Cursor 1                            120 ns
```

jal



```
Wave - Default
                                    Msgs
vsim1:/tb/MyMips/iCLK               1
vsim1:/tb/MyMips/iRST               0
vsim1:/tb/MyMips/s_IF_PCP4          32'h00400018
vsim1:/tb/MyMips/s_Inst             32'h20010008
vsim1:/tb/MyMips/hazard_det...      1
vsim1:/tb/MyMips/hazard_det...      1
vsim1:/tb/MyMips/hazard_det...      1
vsim1:/tb/MyMips/s_EX_alu_...       32'h00000000
vsim1:/tb/MyMips/s_MEM_AL...        32'h00000000
vsim1:/tb/MyMips/s_RegWrD...        32'h00400008
Now                                 600 ns
Cursor 1                            120 ns
```

jr



```
Wave - Default
                                    Msgs
/tb/MyMips/iCLK                     1
/tb/MyMips/iRST                     0
/tb/MyMips/s_IF_PCP4                32'h00400018
/tb/MyMips/s_Inst                   32'h00021820
/tb/MyMips/hazard_detection/...     1
/tb/MyMips/hazard_detection/...     1
/tb/MyMips/hazard_detection/...     1
/tb/MyMips/s_EX_alu_out             32'h00000000
/tb/MyMips/s_MEM_ALUOut             32'h00000000
/tb/MyMips/s_RegWrData              32'h00400008
Now                                 840 ns
Cursor 1                            120 ns
```

All instructions with non-sequential PC updates are seen flushing the pipeline when a branch is taken. All of these individual test cases have similarly formatted test programs to clearly identify when the branch is taken. The branch is taken on the third occasion of a branch or jump instruction in each test program.

[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.

Processor Control and Hazard Signals - TermProj_3_02 /Data Hazard Test Program

[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.

Processor Control and Hazard Signals - TermProj_3_02 /Control Hazard Test Program

[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).

- Our reported frequency is **46.24 MHz**
- The critical path is **Register File > Branch Control Unit > Jump/Jump Register/Branch MUX > ID/EX Register**
- For our hardware scheduled pipeline the most beneficial component to optimize here would be the Register File, as it takes the most time in the critical path (6.07 ns), which is located in the Decode stage
- Since this processor design is pipelined, or each stage has its own path each cycle, the critical path is located in the slowest STAGE. So in this case, our slowest stage is Decode with the Register File taking the longest to propagate signals. This is reasonable, as our hazard detection hardware is located in the Decode stage (branch control and hazard detection units), although it's interesting the register file is still the slowest component and no longer faster than the ALU, despite the ALU not changing. This is likely due to us changing the register file to accept writes on the negative edge of the clock instead of the positive edge