CprE 381: Computer Organization and Assembly-Level Programming

Project Part 2 Report

Team Members: James Minardi

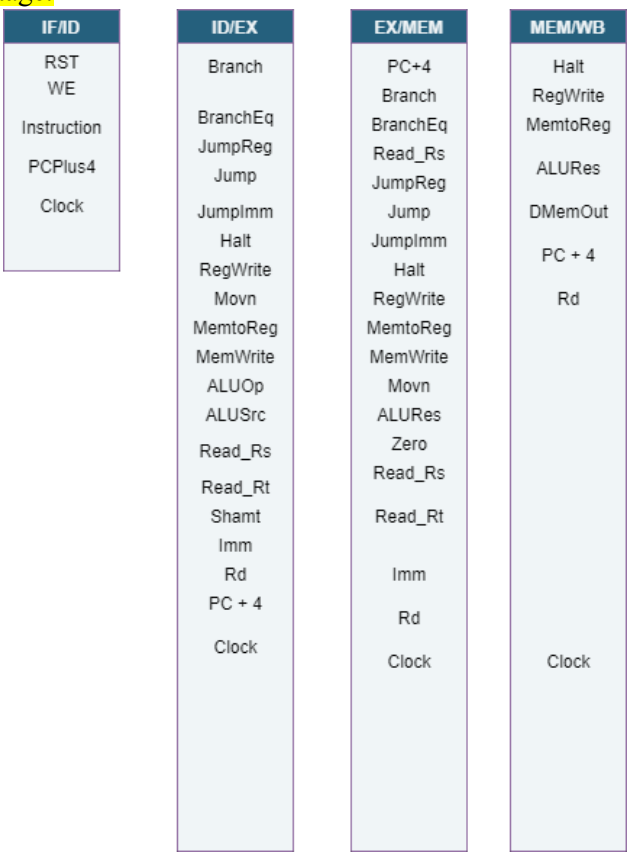
Ella Knott

Project Teams Group #: 306

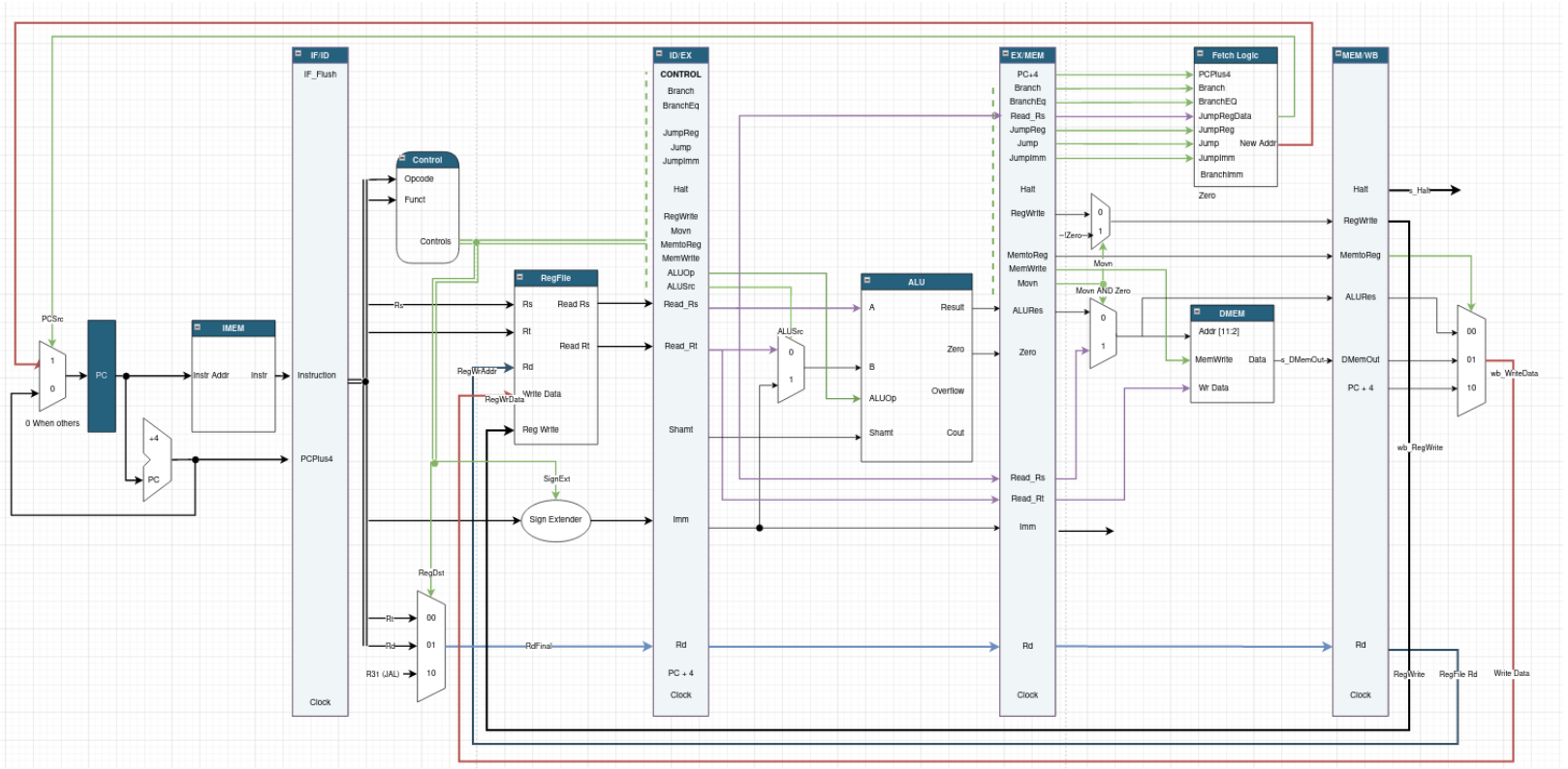
Refer to the highlighted language in the project 1 instruction for the context of the following questions.

Waveforms and timings for both hardware and software can be found in the proj/ directory.

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



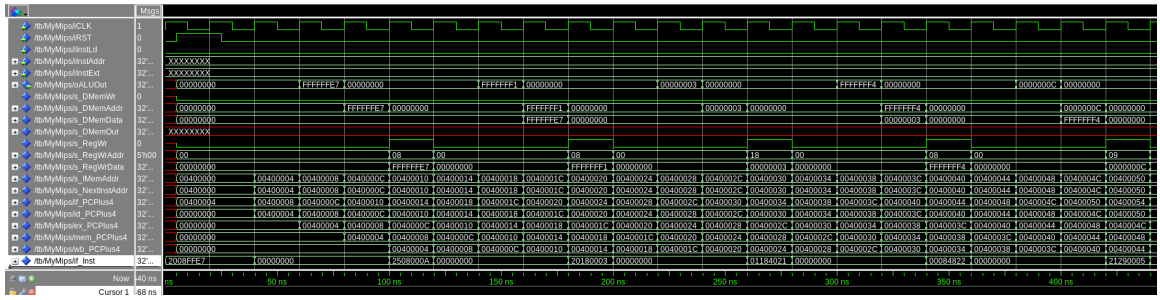
[1.b.ii] high-level schematic drawing of the interconnection between components.

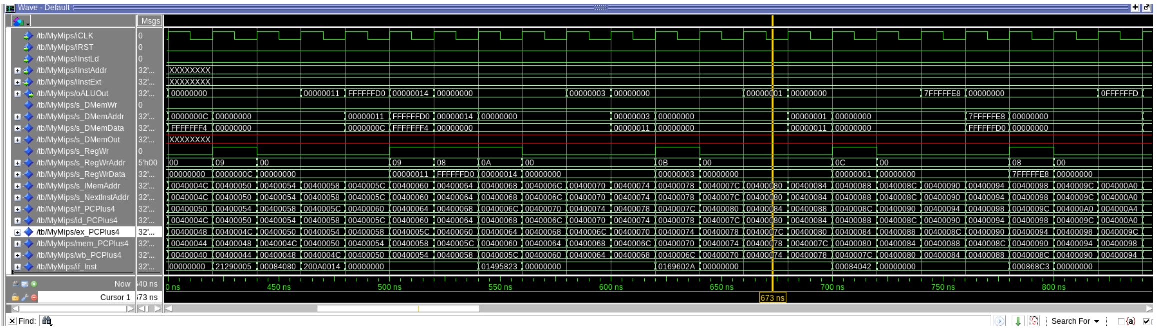


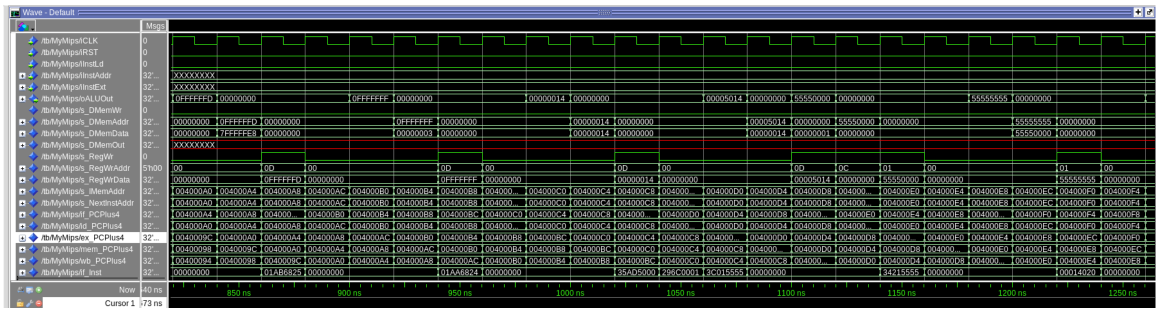
[1.c.i] include an annotated waveform in your writeup and provide a short discussion of result correctness.(The software test that uses all instructions is called software_base.s in mips directory)

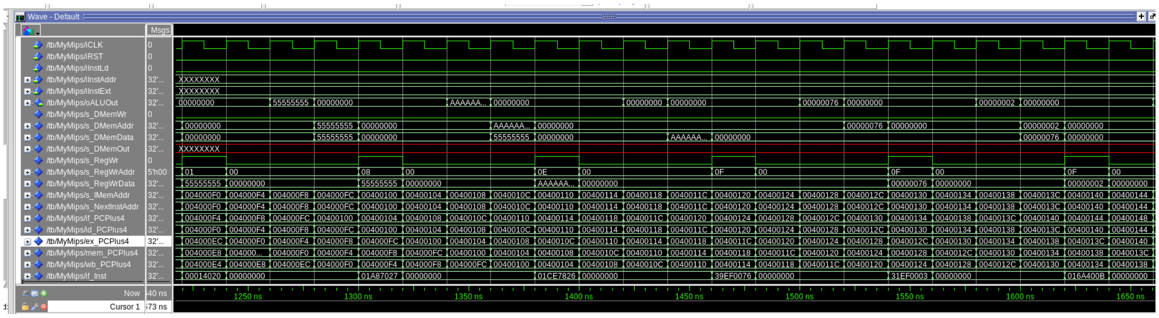
This program uses every instruction supported by the software scheduled pipeline processor. It does not conflict and cause hazards as demonstrated by the comparison with the MARS simulation pictured below.

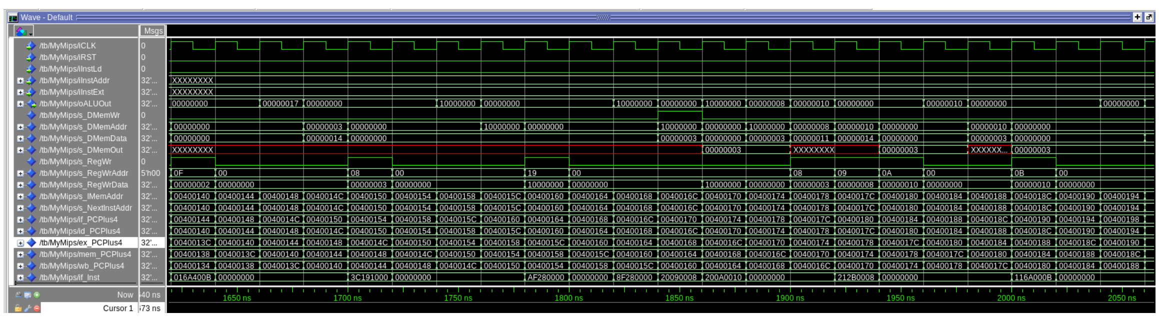
```
bash-4.2$ ./381_tf.sh test --asm-file ./proj/mips/software_base.s
Using VDI Python Environment
Testing
WARNING: Software tree location not set or invalid, using $MGC_HOME=/usr/local/mentor/calibre
Running: ./proj/mips/software_base.s
Starting Mars Simulation for : ./proj/mips/software_base.s ...Success
starting compilation...
Successfully compiled proj/src/MIPS_types.vhd
Successfully compiled internal/testpy/tb.vhd
Successfully compiled proj/src/MIPS_types.vhd
Successfully compiled proj/src/Multiplexers/mux32t1_N.vhd
Successfully compiled proj/src/Multiplexers/mux2t1.vhd
Successfully compiled proj/src/Multiplexers/mux2t1_N.vhd
Successfully compiled proj/src/Multiplexers/mux32t1.vhd
Successfully compiled proj/src/SubComponents/full_adder.vhd
Successfully compiled proj/src/SubComponents/decoder_N.vhd
Successfully compiled proj/src/SubComponents/invg.vhd
Successfully compiled proj/src/SubComponents/full_adder_N.vhd
Successfully compiled proj/src/SubComponents/dffg_N.vhd
Successfully compiled proj/src/SubComponents/ones_compg.vhd
Successfully compiled proj/src/SubComponents/xorg2.vhd
Successfully compiled proj/src/SubComponents/PC_dffg.vhd
Successfully compiled proj/src/SubComponents/barrel_shifter.vhd
Successfully compiled proj/src/SubComponents/andg2.vhd
Successfully compiled proj/src/SubComponents/extender.vhd
Successfully compiled proj/src/SubComponents/dffg.vhd
Successfully compiled proj/src/SubComponents/org2.vhd
Successfully compiled proj/src/SubComponents/add_sub.vhd
Successfully compiled proj/src/TopLevel/regfile.vhd
Successfully compiled proj/src/TopLevel/IDEX_reg.vhd
Successfully compiled proj/src/TopLevel/MIPS_Processor.vhd
Successfully compiled proj/src/TopLevel/control.vhd
Successfully compiled proj/src/TopLevel/mem.vhd
Successfully compiled proj/src/TopLevel/fetch.vhd
Successfully compiled proj/src/TopLevel/ALU.vhd
Successfully compiled proj/src/TopLevel/EXMEM_reg.vhd
Successfully compiled proj/src/TopLevel/PC_reg.vhd
Successfully compiled proj/src/TopLevel/MEMWB_reg.vhd
Successfully compiled proj/src/TopLevel/IFID_reg.vhd
Starting VHDL Simulation...Success
Victory!! Your processes matches MARS expected output with no mismatches!!
Instructions: 111       Cycles: 120     CPI: 1.081
bash-4.2$ modelsim
Reading pref.tcl
```

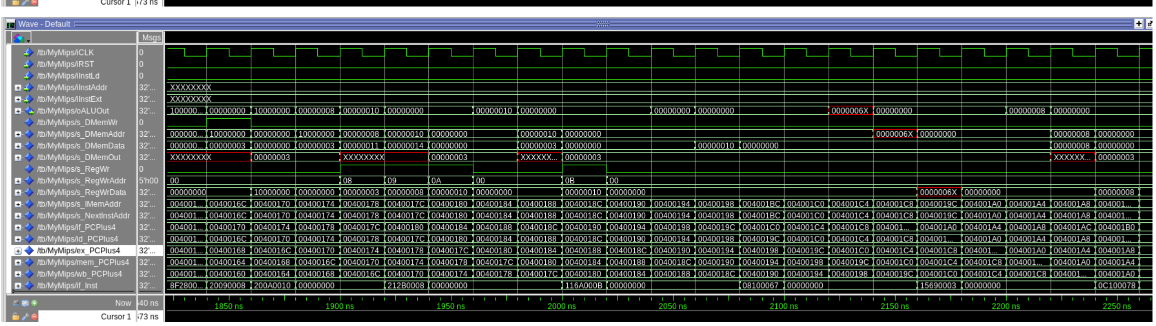


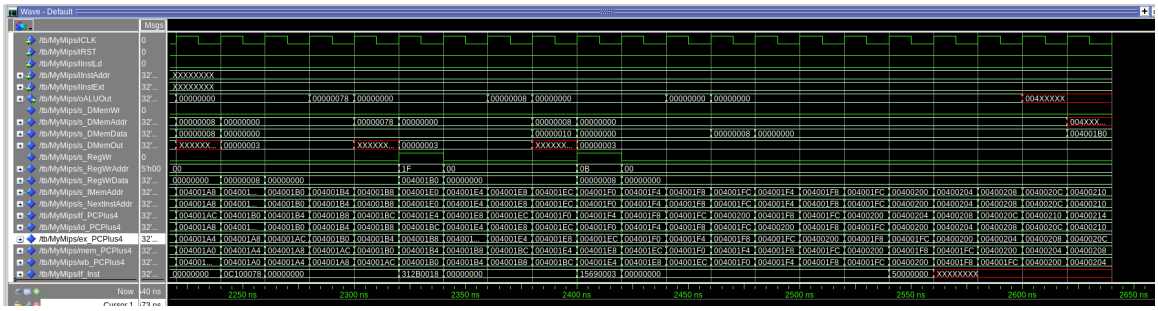

Wave - Default
Msgs
/tb/MyMips/iCLK
/tb/MyMips/iRST
/tb/MyMips/iInstLd
/tb/MyMips/iInstAddr
/tb/MyMips/iInstExt
/tb/MyMips/oALUOut
/tb/MyMips/s_DMemWr
/tb/MyMips/s_DMemAddr
/tb/MyMips/s_DMemData
/tb/MyMips/s_DMemOut
/tb/MyMips/s_RegWr
/tb/MyMips/s_RegWrAddr
/tb/MyMips/s_RegWrData
/tb/MyMips/s_IMemAddr
/tb/MyMips/s_NextInstAddr
/tb/MyMips/if_PCPlus4
/tb/MyMips/id_PCPlus4
/tb/MyMips/ex_PCPlus4
/tb/MyMips/mem_PCPlus4
/tb/MyMips/wb_PCPlus4
/tb/MyMips/if_Inst
Now
Cursor 1











[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.

The waveform below is annotated with comments on what the program is doing based on the instruction in the ID stage. Given an array of 1,3,2,5,4,0,7,6 it behaves as expected and swaps the elements needed while leaving alone the pairs that are in order. For example the first iteration of (1,3) is left alone and proceeds to swap the next pair of elements which is (3,2).

Because of where our fetch logic is contained (in the mem stage), when we use nops, it is always in pairs of 3 because solving data hazards requires 3 and control flow hazards requires 3 as well. However, there are a few instances in this program where 3 nops were not required.

In this example, we have two load word instructions back to back without any nops in between them. This is because while they access the same register to use for the memory address, they don't actually modify that value. Where they put the new data are in separate registers meaning they don't create a hazard. Code shown below:

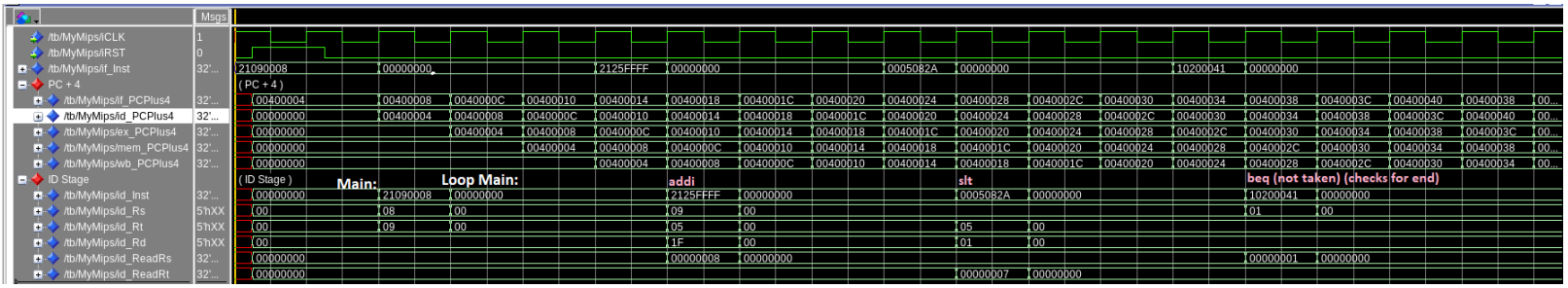
```
lw $s1, 0($a0)      # loads first element in $s1
lw $s2, 4($a0)      # loads second element in $s2
```

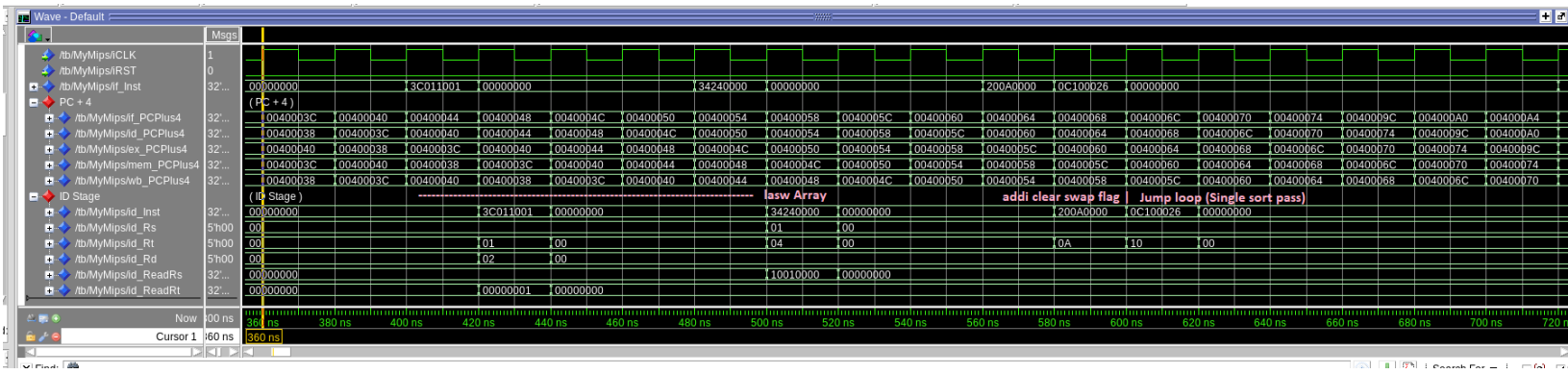
Because 3 instructions are always required after every control flow instruction, there are no locations in this program where the max number of nops were not needed for a control flow hazard.

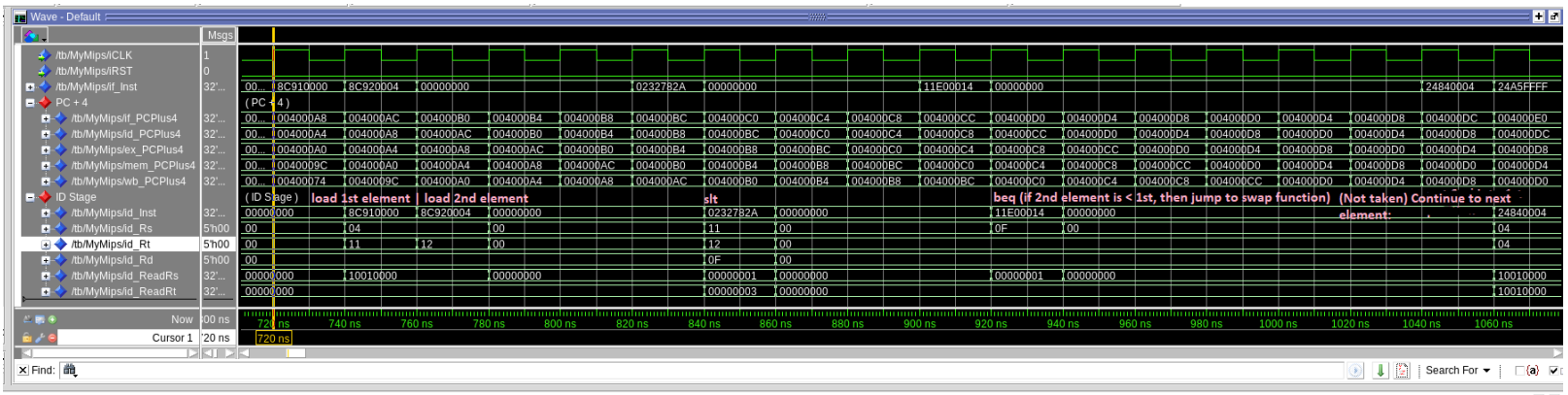
Here are a few more examples of areas where nops were not needed. Firstly, the store words, while accessing $a0 for the memory address, because of the immediate value they are not storing into the same location thus avoiding a data hazard. Similarly, the addi instruction can sit freely without any nops inbetween the jump instruction and store words because it does not access any modifiable values. And below the jump instruction, three nops are required because the new PC address is calculated in the mem stage.

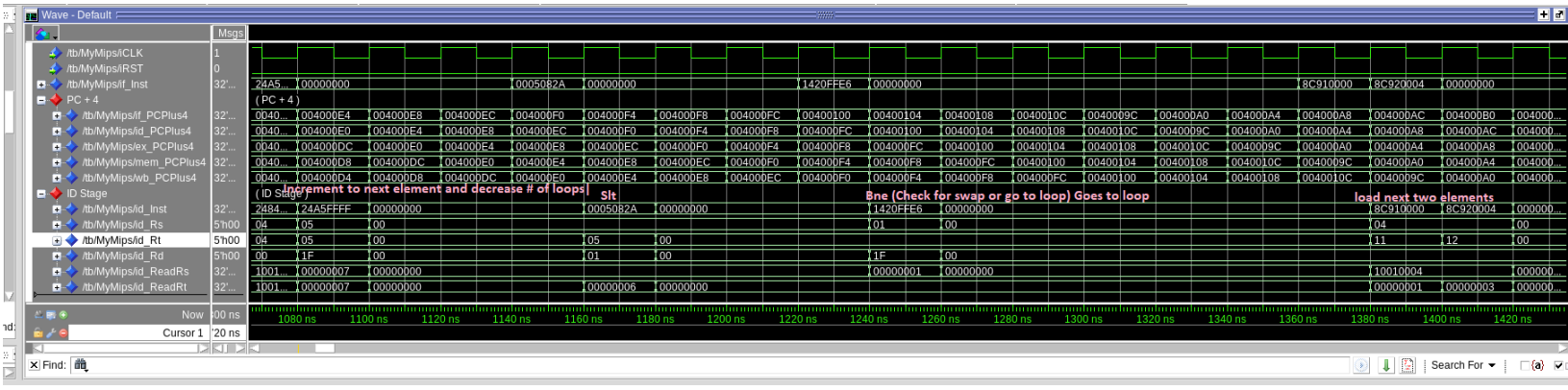
```
swap:
    sw $s1, 4($a0)      # puts value of [i+1] in s1
    sw $s2, 0($a0)      # puts value of [i] in s2
    addi $t2, $0, 1     # notes a swap

    j next          # jump to next
    nop
    nop
    nop

end:
```

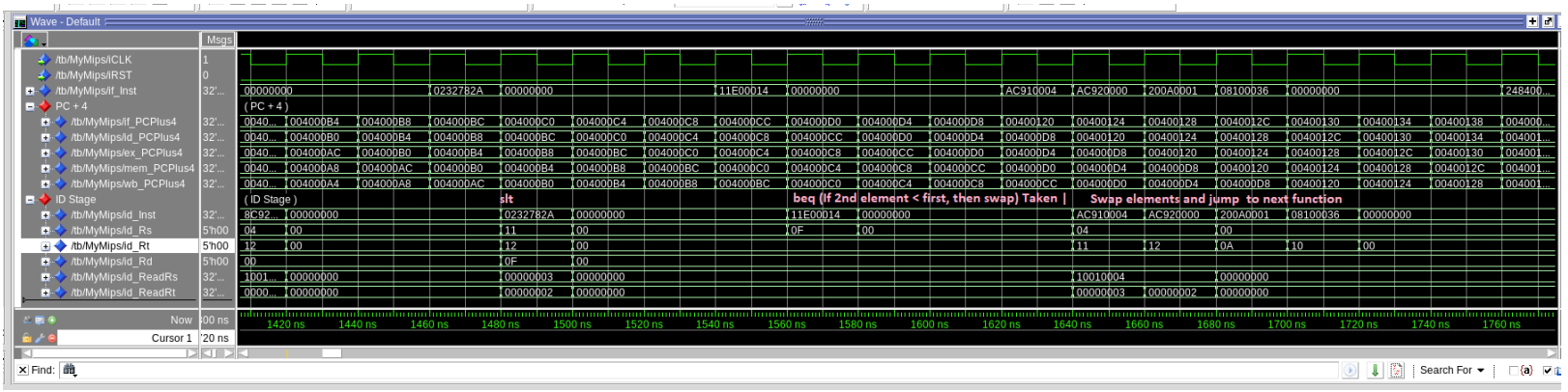
```
bash-4.2$ ./381_tf.sh test --asm-file ./proj/mips/software_bubblesort.s
Using VDI Python Environment
Testing
WARNING: Software tree location not set or invalid, using $MGC_HOME=/usr/local/mentor/calibre
Running: ./proj/mips/software_bubblesort.s
Starting Mars Simulation for : ./proj/mips/software_bubblesort.s ...Success
starting compilation...
Successfully compiled proj/src/MIPS_types.vhd
Successfully compiled internal/testpy/tb.vhd
Successfully compiled proj/src/MIPS_types.vhd
Successfully compiled proj/src/Multiplexers/mux32t1_N.vhd
Successfully compiled proj/src/Multiplexers/mux2t1.vhd
Successfully compiled proj/src/Multiplexers/mux2t1_N.vhd
Successfully compiled proj/src/Multiplexers/mux32t1.vhd
Successfully compiled proj/src/SubComponents/full_adder.vhd
Successfully compiled proj/src/SubComponents/decoder_N.vhd
Successfully compiled proj/src/SubComponents/invg.vhd
Successfully compiled proj/src/SubComponents/full_adder_N.vhd
Successfully compiled proj/src/SubComponents/dffg_N.vhd
Successfully compiled proj/src/SubComponents/ones_compg.vhd
Successfully compiled proj/src/SubComponents/xorg2.vhd
Successfully compiled proj/src/SubComponents/PC_dffg.vhd
Successfully compiled proj/src/SubComponents/barrel_shifter.vhd
Successfully compiled proj/src/SubComponents/andg2.vhd
Successfully compiled proj/src/SubComponents/extender.vhd
Successfully compiled proj/src/SubComponents/dffg.vhd
Successfully compiled proj/src/SubComponents/org2.vhd
Successfully compiled proj/src/SubComponents/add_sub.vhd
Successfully compiled proj/src/TopLevel/regfile.vhd
Successfully compiled proj/src/TopLevel/IDEX_reg.vhd
Successfully compiled proj/src/TopLevel/MIPS_Processor.vhd
Successfully compiled proj/src/TopLevel/control.vhd
Successfully compiled proj/src/TopLevel/mem.vhd
Successfully compiled proj/src/TopLevel/fetch.vhd
Successfully compiled proj/src/TopLevel/ALU.vhd
Successfully compiled proj/src/TopLevel/EXMEM_reg.vhd
Successfully compiled proj/src/TopLevel/PC_reg.vhd
Successfully compiled proj/src/TopLevel/MEMWB_reg.vhd
Successfully compiled proj/src/TopLevel/IFID_reg.vhd
Starting VHDL Simulation...Success
Victory!! Your processes matches MARS expected output with no mismatches!!
Instructions: 936       Cycles: 1213    CPI: 1.296
bash-4.2$
```

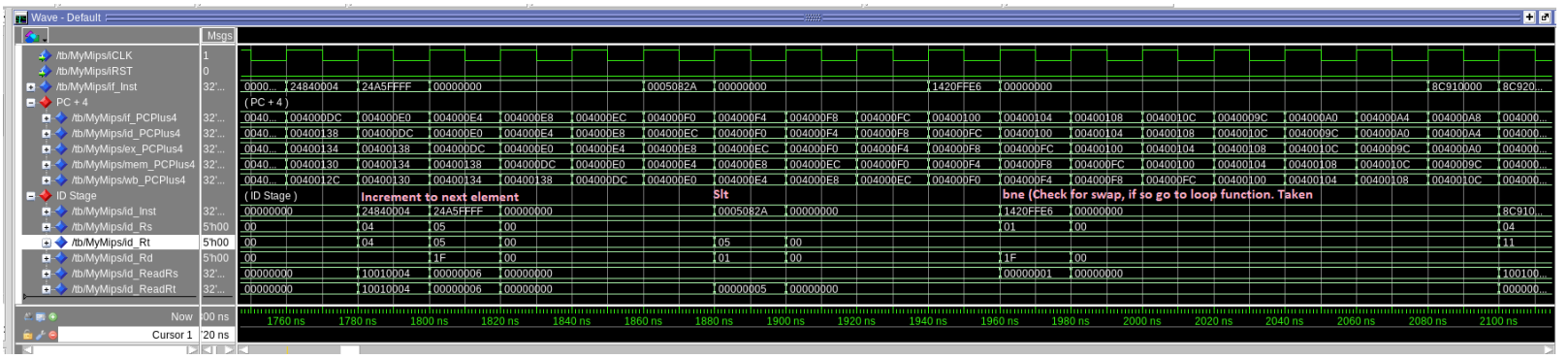

Main:
Loop Main:
addi
slt
beq (not taken) (checks for end)

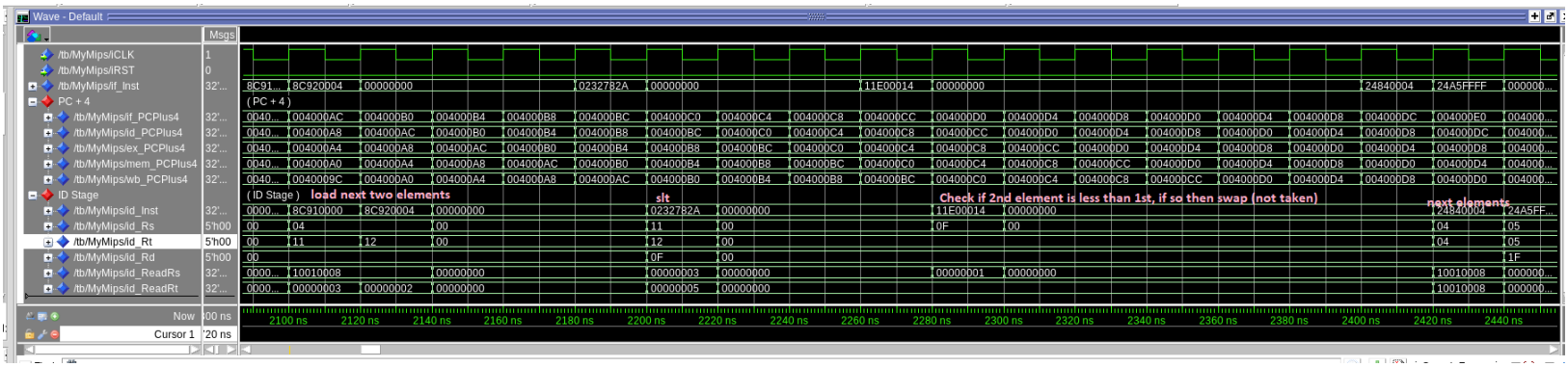

lasw Array
addi clear swap flag
Jump loop (Single sort pass)


load 1st element
load 2nd element
slt
beq (if 2nd element is < 1st, then jump to swap function)
(Not taken) Continue to next element:


Increment to next element and decrease # of loops
slt
Bne (Check for swap or go to loop) Goes to loop
load next two elements


Wave - Default
/tb/MyMips/iCLK
/tb/MyMips/iRST
/tb/MyMips/if_Inst
PC + 4
/tb/MyMips/if_PCPlus4
/tb/MyMips/id_PCPlus4
/tb/MyMips/ex_PCPlus4
/tb/MyMips/mem_PCPlus4
/tb/MyMips/wb_PCPlus4
ID Stage
/tb/MyMips/id_Inst
/tb/MyMips/id_Rs
/tb/MyMips/id_Rt
/tb/MyMips/id_Rd
/tb/MyMips/id_ReadRs
/tb/MyMips/id_ReadRt
slt
beq (If 2nd element < first, then swap) Taken
Swap elements and jump to next function
Now
Cursor 1
Find:
Search For


Wave - Default
/tb/MyMips/iCLK
/tb/MyMips/iRST
/tb/MyMips/if_Inst
PC + 4
/tb/MyMips/if_PCPlus4
/tb/MyMips/id_PCPlus4
/tb/MyMips/ex_PCPlus4
/tb/MyMips/mem_PCPlus4
/tb/MyMips/wb_PCPlus4
ID Stage
/tb/MyMips/id_Inst
/tb/MyMips/id_Rs
/tb/MyMips/id_Rt
/tb/MyMips/id_Rd
/tb/MyMips/id_ReadRs
/tb/MyMips/id_ReadRt
Increment to next element
Slt
bne (Check for swap, if so go to loop function. Taken
Now
Cursor 1


Wave - Default
/tb/MyMips/iCLK
/tb/MyMips/iRST
/tb/MyMips/if_Inst
PC + 4
/tb/MyMips/if_PCPlus4
/tb/MyMips/id_PCPlus4
/tb/MyMips/ex_PCPlus4
/tb/MyMips/mem_PCPlus4
/tb/MyMips/wb_PCPlus4
ID Stage
/tb/MyMips/id_Inst
/tb/MyMips/id_Rs
/tb/MyMips/id_Rt
/tb/MyMips/id_Rd
/tb/MyMips/id_ReadRs
/tb/MyMips/id_ReadRt
load next two elements
slt
Check if 2nd element is less than 1st, if so then swap (not taken)
next elements
Now
Cursor 1

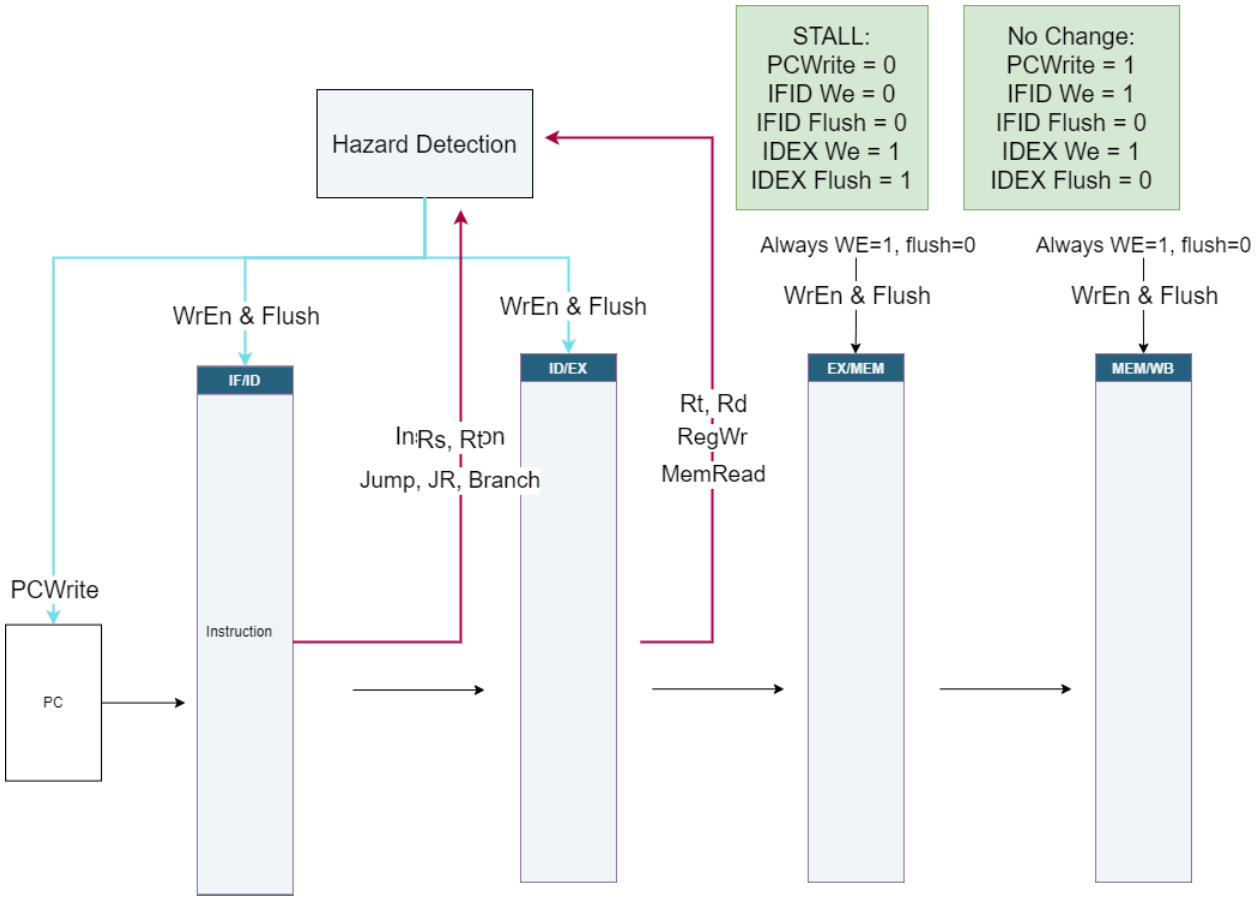
[1.d] report the maximum frequency your software-scheduled pipelined processor can run at and determine what your critical path is (specify each module/entity/component that this path goes through).
FMax: 55.82mhz

Critical path:

Execute stage:

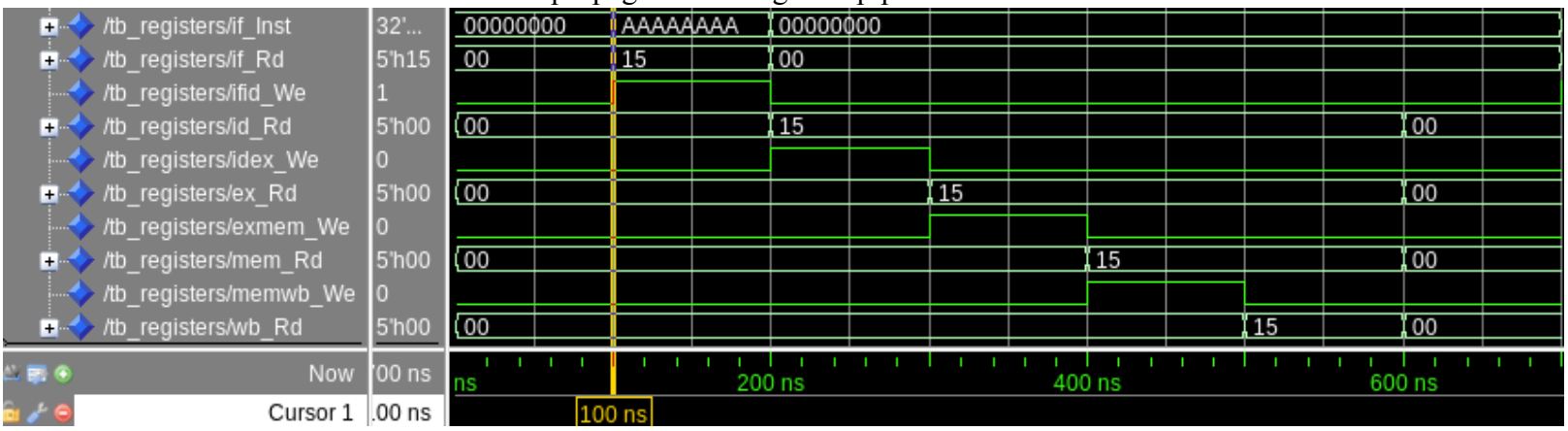
1. IDEX register ALUSrc output
2. ALU Src mux (Selects ReatRt vs Immediate)
3. ALU B input -> add_sub component
4. nbit full adder (32 full adders)
5. ALU Operation mux
6. ALU Zero mux
7. ALU Output Zero
8. EXMEM register Zero input

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

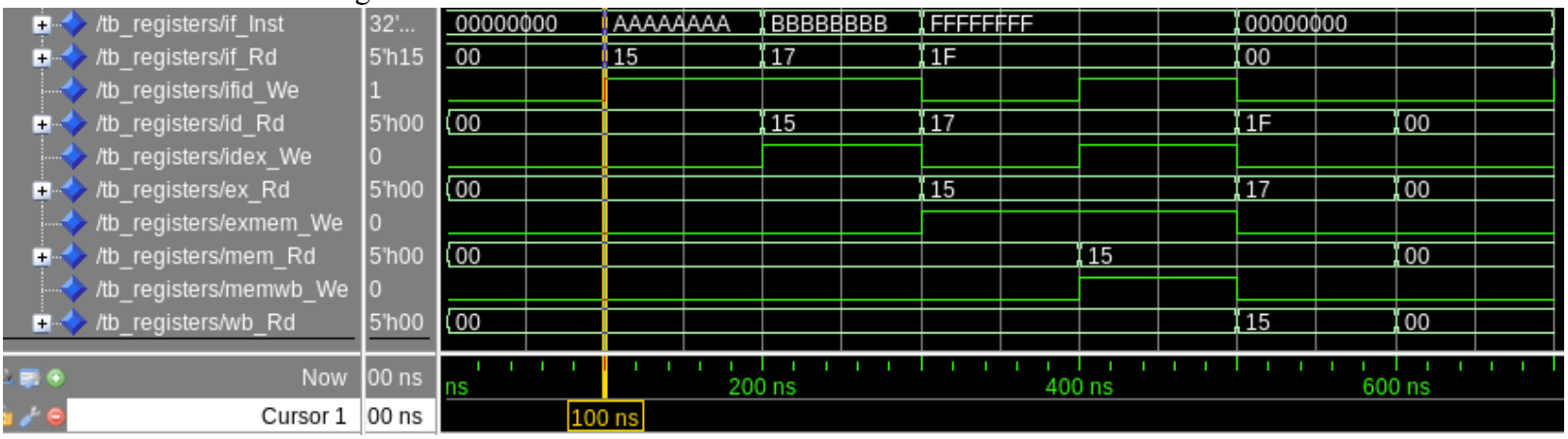


[2.a.iii] Create a testbench that instantiates all four of the registers in a single design. Show that values that are stored in the initial IF/ID register are available as expected four cycles later, and that new values can be inserted into the pipeline every single cycle. Most importantly, this testbench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

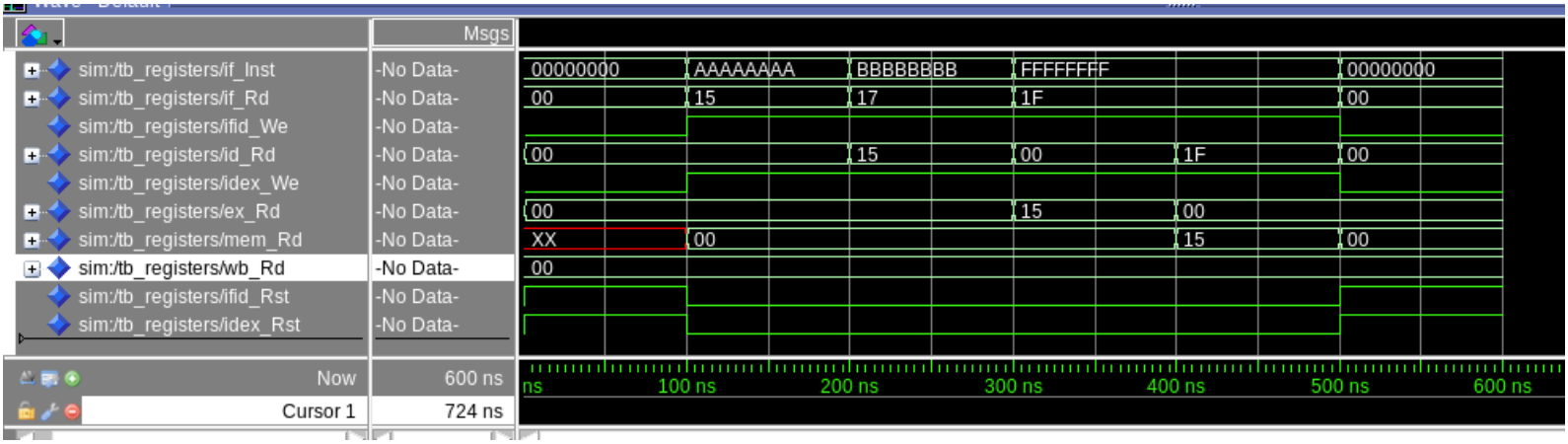
Shows normal instruction propagation through the pipeline:



Demonstrates stalling of pipeline while instruction “BBBBBBBB” (Rd=17) is in the decode stage.



Demonstrates flushing of the IFID stage for instruction “BBBBBBBB”. As you can see, it is replaced by a NOP in the following clock cycle as it moves into the ID stage.



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

Produce: Change registers or memory
Consume: Access registers or memory

INSTR	PRODUCE	SIGNALs	CONSUME	SIGNALs
add	TRUE	RsData, RtData, ALUResult, RegWrData	TRUE	Rs, Rt
addi	TRUE	RsData, RtData, ALUResult, RegWrData	TRUE	Rs, Imm
addiu	TRUE	RsData, RtData, ALUResult, RegWrData	TRUE	Rs, Imm
addu	TRUE	RsData, RtData, ALUResult, RegWrData	TRUE	Rs, Rt
and	TRUE	RsData, RtData, ALUResult, RegWrData	TRUE	Rs, Rt
andi	TRUE	RsData, RtData, ALUResult, RegWrData	TRUE	Rs, Imm
lui	TRUE	RsData, ALUResult, RegWrData	TRUE	Rs, Imm
lw	TRUE	ALUResult, RegWrData	TRUE	Rs, Imm
nor	TRUE	RsData, RtData, ALUResult, RegWrData	TRUE	Rs, Rt
xor	TRUE	RsData, RtData, ALUResult, RegWrData	TRUE	Rs, Rt
xori	TRUE	RsData, RtData, ALUResult, RegWrData	TRUE	Rs, Imm
or	TRUE	RsData, RtData, ALUResult, RegWrData	TRUE	Rs, Rt
ori	TRUE	RsData, RtData, ALUResult, RegWrData	TRUE	Rs, Imm
slt	TRUE	RsData, RtData, ALUResult, RegWrData	TRUE	Rs, Rt
slti	TRUE	RsData, RtData, ALUResult, RegWrData	TRUE	Rs, Imm
sll	TRUE	RsData, RtData, ALUResult, RegWrData	TRUE	Rs, Rt, Shamt
srl	TRUE	RsData, RtData, ALUResult, RegWrData	TRUE	Rs, Rt, Shamt
sra	TRUE	RsData, RtData, ALUResult, RegWrData	TRUE	Rs, Rt, Shamt
sw	TRUE	RsData, RtData, ALUResult	TRUE	Rt
sub	TRUE	RsData, RtData, ALUResult, RegWrData	TRUE	Rs, Rt
subu	TRUE	RsData, RtData, ALUResult, RegWrData	TRUE	Rs, Rt
beq	FALSE		TRUE	Rs, Rt
bne	FALSE		TRUE	Rs, Rt
j	FALSE		FALSE	
jal	TRUE	RsData, RtData, ALUResult, RegWrData	FALSE	
jr	FALSE		FALSE	
movn	TRUE	RsData, RtData, ALUResult, RegWrData	TRUE	Rs, Rt
repl.qb	TRUE	RsData, RtData, ALUResult, RegWrData	TRUE	Imm
halt	FALSE		FALSE	
Total: 28				

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

See 2.b.i

[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.

These four hazards are what make up our hazard detection and forwarding unit, minus control flow hazards and extra checks. For example, we always check for the WE of the

independent instruction and check if the Rd value of the independent instruction = 0. This handles everything from SW, LW, and normal Rtype instruction hazards.

Dependencies
i-type to i-type
i-type to r-type
r-type to r-type
r-type to r-type to r-type
r-type to r-type to r-type
lw to sw

Hazard	Can FWD?	FWD Path:	Require Stall?	
Load-use	FALSE		TRUE	Stall ID once
Regfile bypass	TRUE	WB Rd -> ID Rs/Rt	FALSE	
mem_Rd = ex_Rs/Rt	TRUE	MEM Rd -> EX Rs/Rt	FALSE	
ex_Rd = id_Rs/Rt & JR/Branch	TRUE	Did not do this FWD path	TRUE	Stall ID
mem_Rd = id_Rs/Rt & JR/Branch	TRUE	Did not do this FWD path	TRUE	Stall ID

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

See 2.d for full diagram that includes the registers, hazard unit, and forwarding unit with names for the signals required.

[2.c.i] list all instructions that may result in a non-sequential PC update and in which pipeline stage that update occurs.

Because we handle control flow data hazards in fwding, no stalling is required for control hazards. All we have to do is flush the fetched instruction and leave the PC write enabled, thus placing a NOP in between the control flow instruction and the following instruction. This flush is required to allow for the fetch logic to fetch the next instruction in the ID stage.

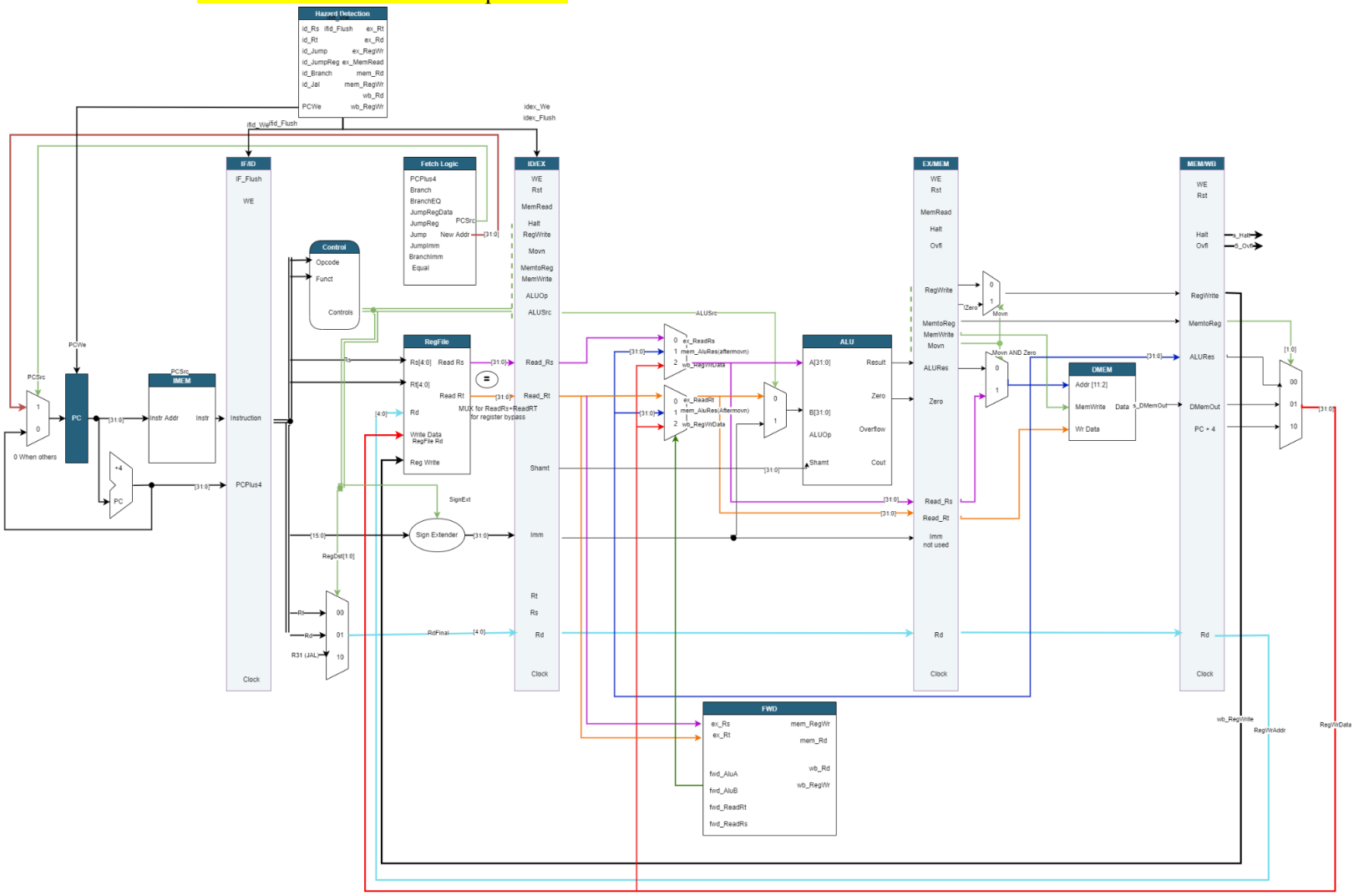
Control Hazard Avoidance

INSTR	Stage of PC Update	Stall/Flush what stages
beq	ID	Flush IFID, continue ID as normal
bne	ID	Flush IFID, continue ID as normal
j	ID	Flush IFID, continue ID as normal
jal	ID	Flush IFID, continue ID as normal
jr	ID	Flush IFID, continue ID as normal
		No Stalling required, handles in FWDing

[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.

See 2.c.i

[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.

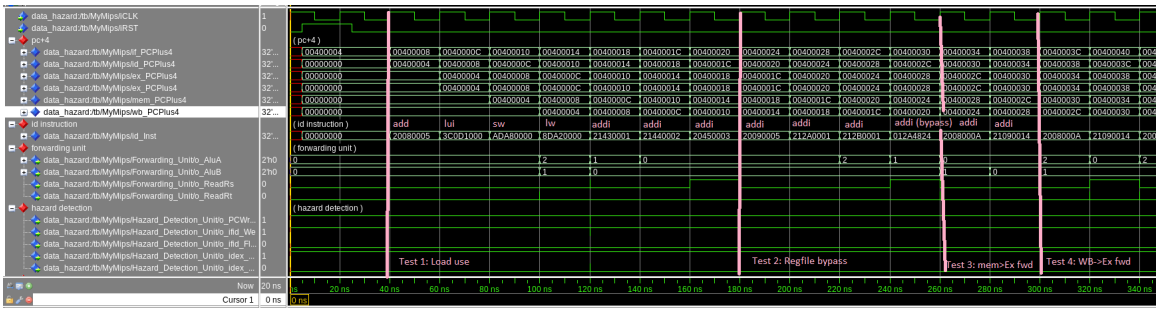


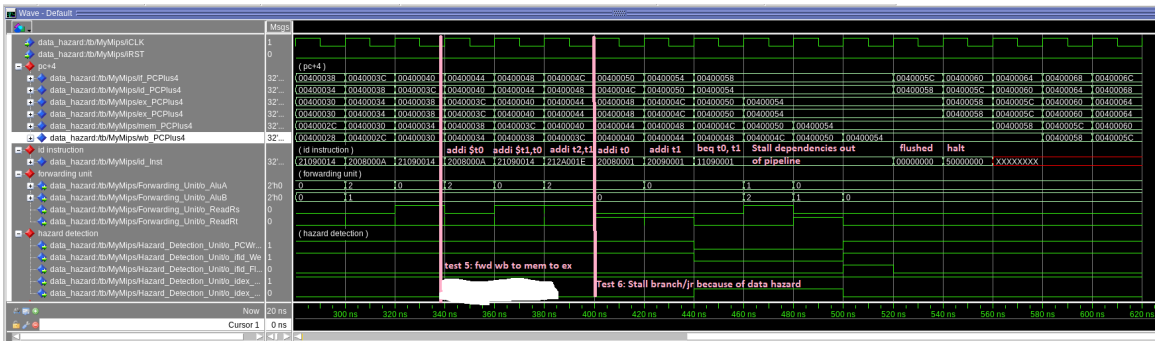
[2.e – i, ii, and iii] In your writeup, show the QuestaSim output for each of the following tests, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.

Below are our waveforms for our data and control flow hazard tests. They are each annotated with the test number and the instructions within that test that correspond to our spreadsheet. Our tests are confirmed to work using the toolflow, matching the MIPS output with no errors. On top of that, we can confirm our functionality by looking at the forwarding unit and hazard unit output signals which match our expectations.

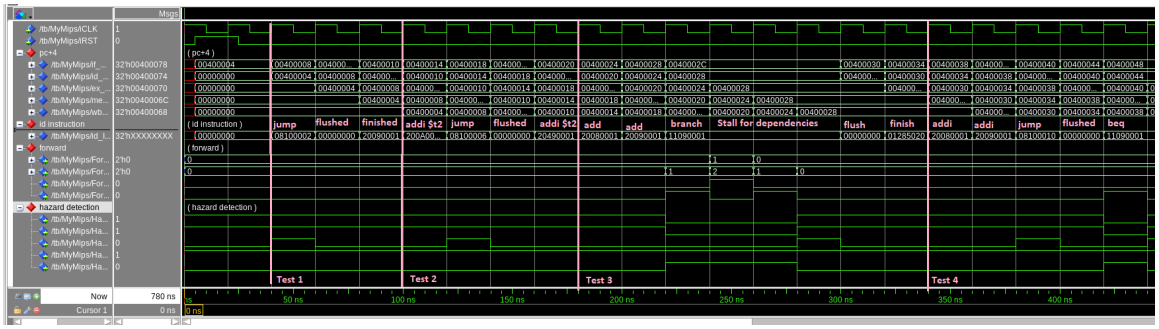
Although not pictured below, our processor also contains no mismatches with the MIPs dump for all other test programs including our original bubble sort, base tests, branch test, fibonacci, and grendel.

Data Hazard Tests





Control Flow Hazard Tests



[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.

Our data hazard assembly program properly tests each of our forwarding and hazard unit edge cases in unit and combination. Really, there aren't that many cases. We must forward from WB to EX, MEM to EX, WB to ID, and stall control flow data hazards, all of which are tested in this program. On the right hand size, there is a coverage list containing an explanation of what our test programs actually test.

Data hazard tests

Test	Instructions
load use	lw **$1**, 0($2)
combination	andi $4, **$1**, 5 (Stall)
	andi $6, **$1**, 7 (Fwd)
	ori $8, **$1**, 9 (ok)
Regfile Bypass	andi $1, $0, 5
combination	andi $2, $1, 1 (6) (fwd)
	andi $3, $1, 1 (6) (fwd)
	and $1, $1, $2 (rf bypass)
fwd mem -> ex	addi $t0, $0, 10
unit test	addi $t1, $t0, 20 (fwd)
fwd wb -> ex	addi $t0, $0, 10
unit test	nop
	addi $t1, $t0, 20 (fwd)
fwd wb-> ex & mem	addi $t0, $0, 10
unit test	addi $t1, $t0, 20 (fwd)
fwd wb -> mem -> ex	addi $t0, $0, 10
combination	addi $t1, $t0, 20 (fwd)
	addi $t2, $t1, 30 (fwd)
stall branch/jr b/c of data hazard	addi $t0, $0, 1
	addi $t1, $0, 1
	beq $t0, $t1, branch (stall)
	branch: halt

Control hazard tests

Test	Instructions
jump	j jump
unit	addi $t0, $0, 5
	jump: addi $t1, $0, 1
data flow & control hazard	addi $t5, $0, 5
combination	j jump
	addi $t0, $0, 3 (flush)
	jump: addi $t1, $5, 1
beq	addi $t0, $0, 1
combination	addi $t1, $0, 1
	beq $t0, $t1, branch (stall for both)
	addi $t0, $0, 3 (flush)
	branch: add $t2, $t1, $t0 (no fwd)
	no fwd needed b/c branch stalls ^
branch + Jump	addi $t0, $0, 1
combination	addi $t1, $0, 1
	j jump
	addi $t0, $0, 3 (flush)
	jump:
	beq $t0, $t1, branch(stall for $t1)
	addi $t0, $0, 3 (flush)
	branch: add $t2, $t1, $t0 (ok)

Coverage

Test	Included:
FWD wb_WriteData to id_ReadRs/Rt (Regfile bypass)	TRUE
FWD mem_Rd to ex_Rs/Rt (EX Hazard)	TRUE
FWD wb_Rd to ex_Rs/Rt (Mem Hazard)	TRUE
STALL Load-Use	TRUE
STALL Branch/Jump for dependency in EX Stage	TRUE
STALL Branch/Jump for dependency in MEM Stage	TRUE
STALL Branch/Jump for dependency in WB Stage	TRUE
FLUSH IF if control flow is in ID (Control flow hazard)	TRUE

[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.

See 2.e.i for spreadsheet. Similar to 2.e.i, our control hazard tests focus on our forwarding and hazard unit edge cases in unit and combination. We test a simple flushing of the next instruction after a control flow instruction, and build upon that to complete multiple control flow instructions each flushing their own instructions and forwarding data hazards on top of that.

[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).

FMax: 48.96mhz
Critical Path:

EXECUTE Stage:

1. IDEX ALUSrc output
2. ALU Src mux (Selects ReadRt vs Immediate)
3. ALU B input
4. Add_sub component
    a. nbit full adder
        i. 32 full adders
5. ALU Operation mux (Select add_sub)
6. ALU Zero mux (Selects zero based on result)
7. ALU Output Zero
8. EXMEM Register Zero input

It is essentially the same as our software processor, but at the beginning it says this about the Dmem:
3.726 0.263 uTco mem:DMem | altsyncram:ram_rtl_0 | altsyncram_eg81:auto_generated | ram_block1a0~porta_we_reg
6.575 2.849 FR CELL DMem | ram_rtl_0 | auto_generated | ram_block1a0 | portadataout[0]

We are not sure what this means.

Waveforms and timings for both hardware and software can be found in the proj/ directory.