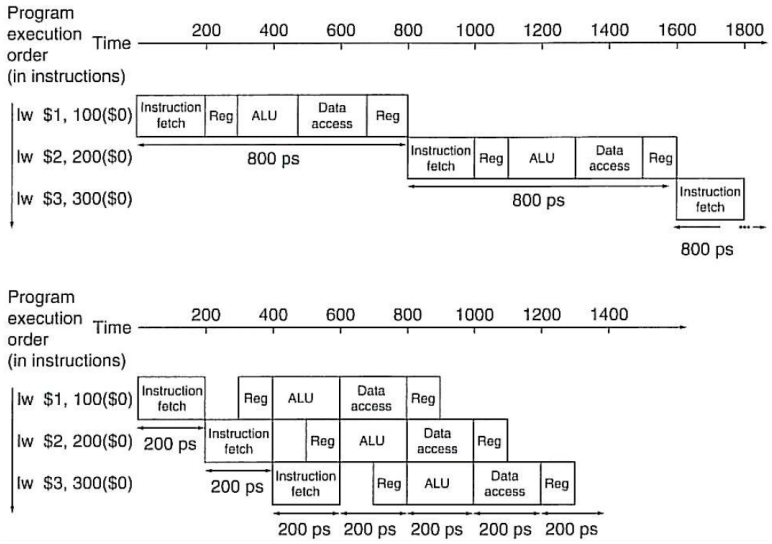
Assignment 5 Report

Long Nguyen
EEL4713
4/18/2012

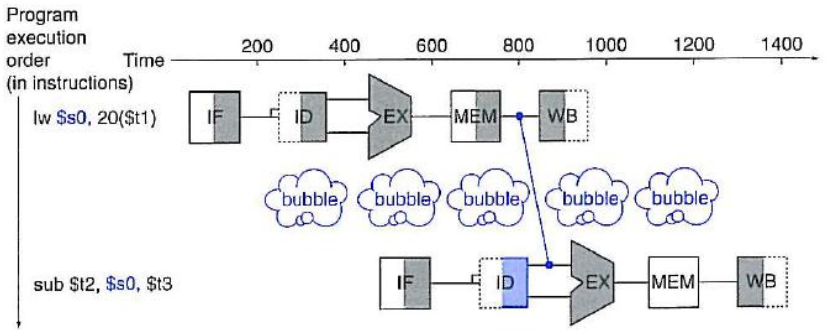
SECTION 1: INTRODUCTION

This lab focuses on taking the previous lab's single cycle processor and implementing a pipelined processor. A pipelined processor takes the single cycle processor and overlaps instructions over the course of several stages. The book provides an example of a program being executed in both a single cycle and pipelined processor.



It is evident that this approach is much more efficient and faster. This does not mean the pipeline increases the rate instructions are completed, it simply increases the throughput of the processor and does not affect execution time. The pipeline will have 5 stages, instruction fetch, instruction decode, execution, memory access, and write back. Instruction fetch will retrieve instructions from memory. The pipeline will then read the register file while decoding the instructions. The next stage will execute the instructions. Following that, data memory will be read and accessed as needed. The final stage writes the result back into the register file.

Pipelining introduces hazards into the processor. These vary from data to structural and control hazards. A common cause of the hazards is the possibility that data read by an instruction isn't written to a register in time for the next instruction to use. Hazards mainly deal with similar issues when reading registers and from memory and the timing of the instruction. In order to solve these hazards, the pipeline also introduces a Hazard Detection Unit and a Forwarding unit. These new hardware pieces add stalls to the pipeline or flush the registers as needed to solve the hazards.



Stalls create bubbles in the pipeline after an instruction so that as the next instruction progress, there is a gap or bubble between the previous. A forwarding unit that data early from registers later in an instruction and passes it on early to the next instruction. This avoids conflicts where an instruction needs data written from the previous instruction, for example the result from an alu is called before the WB stage in a previous instruction.

The hazard table is as follows, provided by the magnificent TA Scott:

	R	AI	L	S	JR	LUI	B	JAL
R	1a: exmem.rd=idex.rs(1) =idex.rt(2) 2a: memwb.rd=idex.rs(3) =idex.rt(4) 3a: memwb.rd=ifid.rs(5) =ifid.rt(6)	1a: exmem.rd=idex.rs(1) 2a: memwb.rd=idex.rs(3) 3a: memwb.rd=ifid.rs(5)	1a: exmem.rd=idex.rs(1) 2a: memwb.rd=idex.rs(3) 3a: memwb.rd=ifid.rs(5)	1a: exmem.rd=idex.rs(1) =idex.rt(2) 2a: memwb.rd=idex.rs(3) =idex.rt(4) 3a: memwb.rd=ifid.rs(5) =ifid.rt(6)	1a: idex.rd=ifid.rs(stall) 2a: exmem.rd=ifid.rs(8) 3a: memwb.rd=ifid.rs(5)		1a: idex.rd=ifid.rs(stall) =ifid.rt(stall) 2a: exmem.rd=ifid.rs(8) =ifid.rt(10) 3a: memwb.rd=ifid.rs(5) =ifid.rt(6)	
AI	1a: exmem.rt=idex.rs(1) =idex.rt(2) 2a: memwb.rt=idex.rs(3) =idex.rt(4) 3a: memwb.rt=ifid.rs(5) =ifid.rt(6)	1a: exmem.rt=idex.rs(1) 2a: memwb.rt=idex.rs(3) 3a: memwb.rt=ifid.rs(5)	1a: exmem.rt=idex.rs(1) 2a: memwb.rt=idex.rs(3) 3a: memwb.rt=ifid.rs(5)	1a: exmem.rt=idex.rs(1) =idex.rt(2) 2a: memwb.rt=idex.rs(3) =idex.rt(4) 3a: memwb.rt=ifid.rs(5) =ifid.rt(6)	1a: idex.rt=ifid.rs(stall) 2a: exmem.rt=ifid.rs(8) 3a: memwb.rt=ifid.rs(5)		1a: idex.rt=ifid.rs(stall) =ifid.rt(stall) 2a: exmem.rt=ifid.rs(8) =ifid.rt(10) 3a: memwb.rt=ifid.rs(5) =ifid.rt(6)	
L	1a: ifid.rt=pre.rs(stall) =pre.rt(stall) 2a: memwb.rt=idex.rs(3) =idex.rt(4) 3a: memwb.rt=ifid.rs(5) =ifid.rt(6)	1a: ifid.rt=pre.rs(stall) 2a: memwb.rt=idex.rs(3) 3a: memwb.rt=ifid.rs(5)	1a: ifid.rt=pre.rs(stall) 2a: memwb.rt=idex.rs(3) 3a: memwb.rt=ifid.rs(5)	1a: ifid.rt=pre.rs(stall) =pre.rt(stall) 2a: memwb.rt=idex.rs(3) =idex.rt(4) 3a: memwb.rt=ifid.rs(5) =ifid.rt(6)	1a: ifid.rt=pre.rs(stall) 2a: idex.rt=pre.rs(stall) 3a: memwb.rt=ifid.rs(5)		1a: ifid.rt=pre.rs(stall) =pre.rt(stall) 2a: idex.rt=pre.rs(stall) =pre.rt(stall) 3a: memwb.rt=ifid.rs(5) =ifid.rt(6)	
S								
JR								
LUI	1a: exmem.rt=idex.rs(11) =idex.rt(12) 2a: memwb.rt=idex.rs(13)	1a: exmem.rt=idex.rs(11) 2a: memwb.rt=idex.rs(13)	1a: exmem.rt=idex.rs(11) 2a: memwb.rt=idex.rs(13)	1a: exmem.rt=idex.rs(11) =idex.rt(12) 2a: memwb.rt=idex.rs(13)	1a: idex.rt=ifid.rs(17) 2a: exmem.rt=ifid.rs(18)		1a: idex.rt=ifid.rs(17) =ifid.rt(19) 2a: exmem.rt=ifid.rs(18)	

	=idex.rt(14) 3a: memwb.rt=ifid.rs(15) =ifid.rt(16)	3a: memwb.rt=ifid.rs(15)	3a: memwb.rt=ifid.rs(15)	=idex.rt(14) 3a: memwb.rt=ifid.rs(15) =ifid.rt(16)	3a: memwb.rt=ifid.rs(15)		=ifid.rt(20) 3a: memwb.rt=ifid.rs(15) =ifid.rt(16)	
B								
JAL	1a: exmem.31=idex.rs(21) =idex.rt(22) 2a: memwb.31=idex.rs(23) =idex.rt(24) 3a: memwb.31=ifid.rs(25) =ifid.rt(26)	1a: exmem.31=idex.rs(21) 2a: memwb.31=idex.rs(23) 3a: memwb.31=ifid.rs(25)	1a: exmem.31=idex.rs(21) 2a: memwb.31=idex.rs(23) 3a: memwb.31=ifid.rs(25)	1a: exmem.31=idex.rs(21) =idex.rt(22) 2a: memwb.31=idex.rs(23) =idex.rt(24) 3a: memwb.31=ifid.rs(25) =ifid.rt(26)	1a: idex.31=ifid.rs(27) 2a: exmem.31=ifid.rs(28) 3a: memwb.31=ifid.rs(25)		1a: idex.31=ifid.rs(27) =ifid.rt(29) 2a: exmem.31=ifid.rs(28) =ifid.rt(30) 3a: memwb.31=ifid.rs(25) =ifid.rt(26)	

SECTION 2:

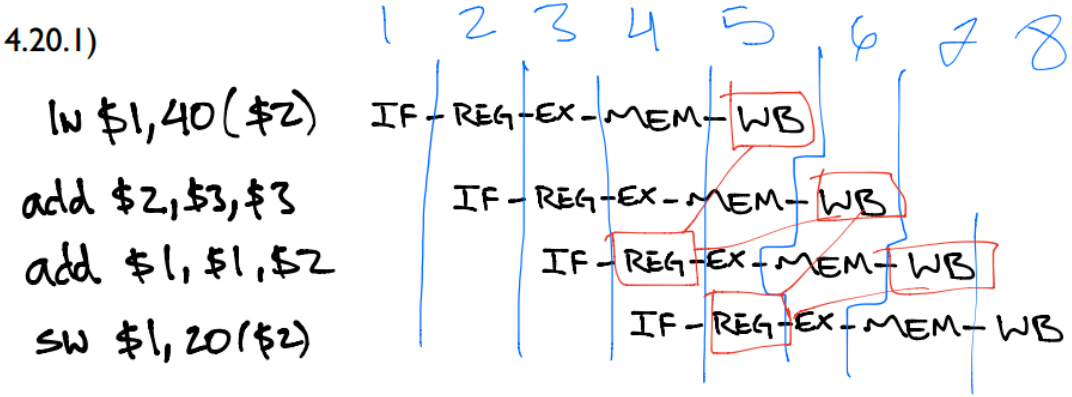
Textbook Questions (Blue book 4th edition)

4.12.1) A pipelined processor has a clock cycle that accommodates the longest hardware unit which is memory storage and load. A single cycle non pipelined processor has a clock cycle of the longest instruction ie the total time it takes for one instruction to move through each stage of the processor.
a) pipeline: 500ps unpipelined: 300+400+350+500+100 = 1650 ps
b) pipeline: 200ps unpipelined: 800ps

4.12.2)
a) latency for lw is 5*500=2500ps in pipelined
unpipelined is total clock time, 1650ps
b) latency is 5*200=1000ps
unpipelined is 800ps

4.12.3)
a) The MEM stage should be split in order to decrease the overall clock cycle time.
b) Since IF is the highest latency, that stage should be split to increase clock cycle time.

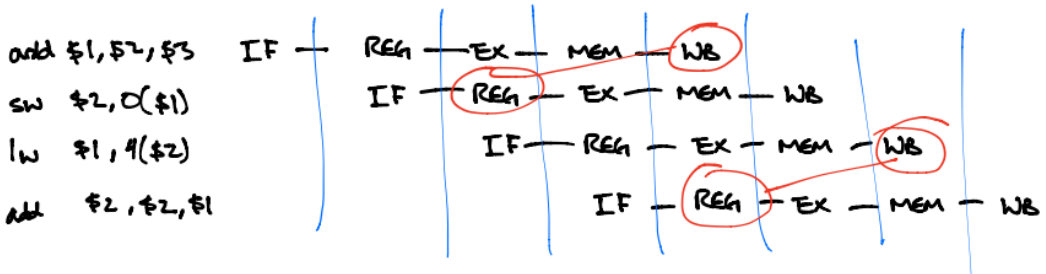
4.20.1)



Read after write : lw to add 2
Add1 to add2 and sw
add2 to sw

Write after read : lw to add1

Write after write: lw to add2



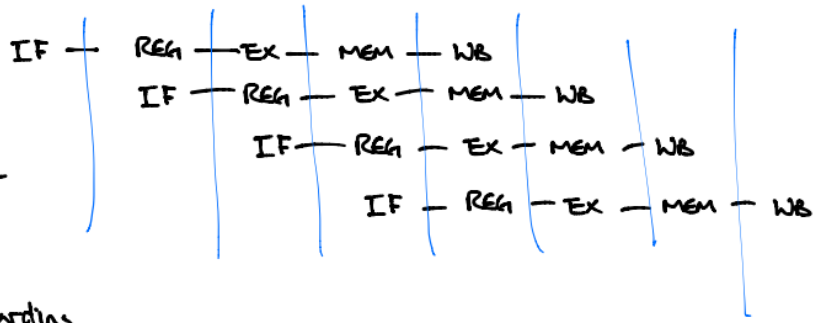
Read after write : Add to sw
lw to add

Write after read : $2 add to sw
$1 add to lw

Write after write: add to lw

4.20.2)

```
lw $1,40($2)
add $2,$3,$3
add $1,$1,$2
sw $1,20($2)
```



without forwarding

a) ($1) lw to add2
($2) add1 to add2
($1) add2 to sw

b) ($1) add to sw
($1) lw to add

4.24.1)

T, T, NT, T 3/4 Always taken
1/4 Not taken

T, T, T, NT, NT 3/5 Always taken
2/5 Not taken

4.24.2)

a) All outcomes predicted are wrong

b) only one prediction is correct: Third taken is correct

4.24.3)

a) T, T, NT, T, T, T, NT, T, T, T, NT, T ...
NT NT T NT T T T T T T T T ...
If pattern repeats forever, accuracy is 3/4

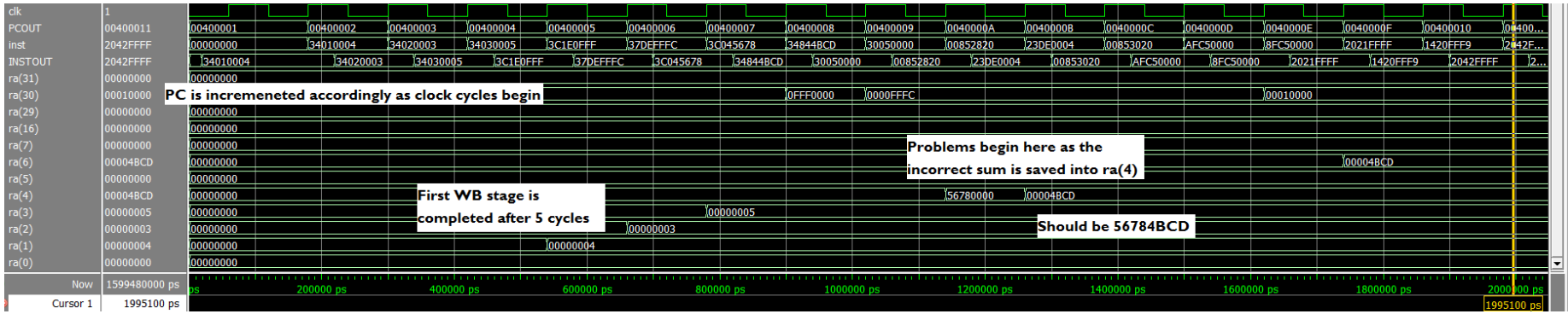
b) T, T, T, NT, NT, T, T, T, NT, NT, T, T, T, NT, NT
NT NT T T T NT T T T T NT T T T T

Accuracy is 2/5

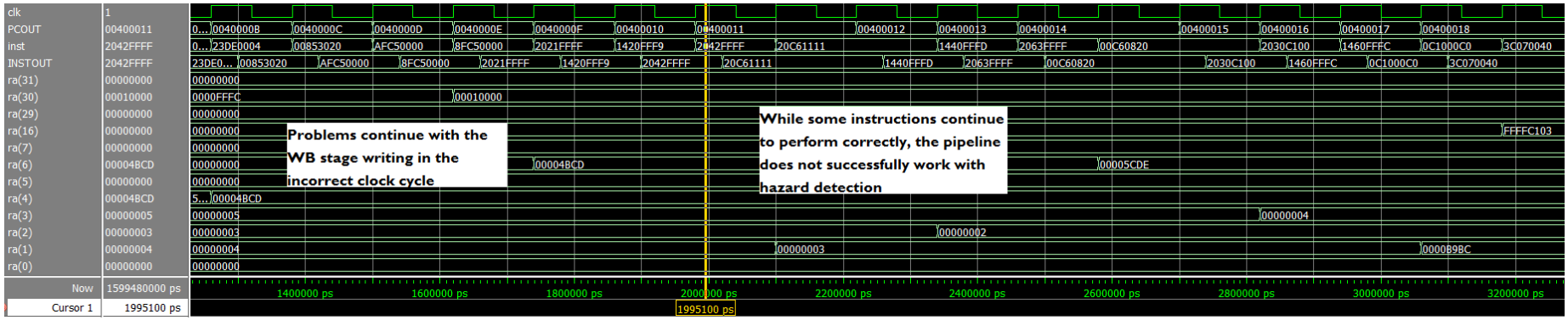
SECTION 3: INSTRUCTION DEMONSTRATION

This section will show the simulation of the working pipeline using the lab4demo.mif. However due to reasons to be expounded upon later, this pipeline is faulty. As such anything that works will be explained and any faults will be shown.

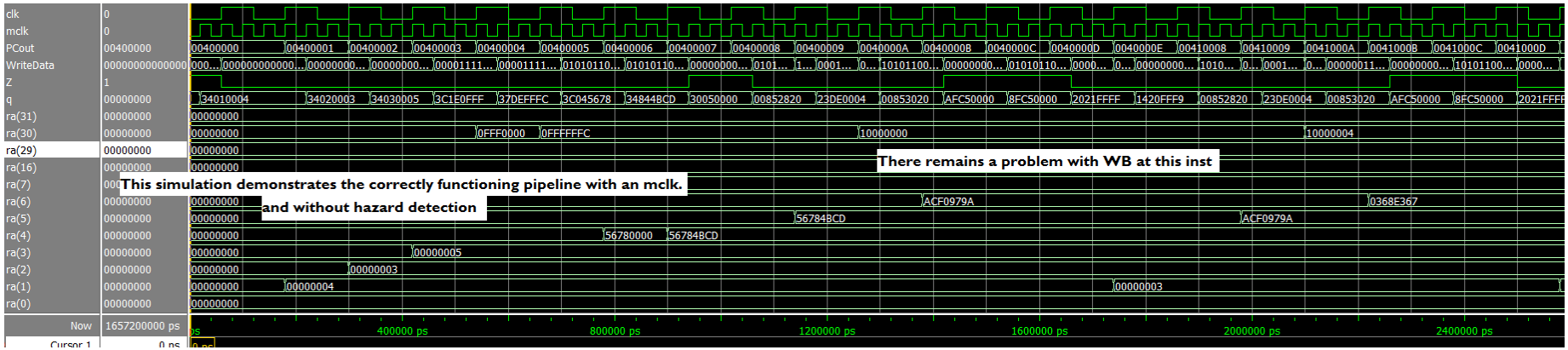
First off, the pipeline can be seen correctly incrementing PC and Inst Fetch as the clock cycles go on. 5 clock cycles should produce the first WB stage which is then written to ra(1) as seen in the figure.



After nights of trouble shooting, however, the pipeline could not correctly write ra(30) in the right cycle. However, I was able to stall the pipeline and add a clock cycle before the addition takes place in ra(4). While I was able to stall, it still was not performing as it should and in the end did not help.



Because I was unable to get the simulation working, the rest of values began to skew as the incorrect operations were performed. However, I can compare the simulation with the pipeline without hazard detection used in the first checkpoint of the lab and the values do match up correctly.



The simulation correctly runs through the instructions up until the final instructions of the lab4demo mif file.

There are a number of possible reasons why the pipeline failed to work correctly. To troubleshoot my pipeline I began by checking the signal path in the bdf (Yes I built it in BDF because I could visually see everything easier). Screenshots of the BDF are attached in the Appendix of the lab report. Following example pipelines in the book and in the lecture slides, everything should be correctly wired. Moving on, I checked my controller signals. I am mostly confident that the controller signals do not change because of the additional pipeline registers. With that in mind, since the pipeline correctly functions without hazard detection it stands to reason that something went wrong when adding the Hazard Detection Unit. Before adding a stall to the pipeline, I tried performing a flush of the registers as this would be more effective. To do this, I tied the Branch Detection logic gates to the clr functions of the pipeline registers. If a branch was detected, the registers would flush, clearing the instructions that were loaded beforehand. Flushing caused anomalies in the pipeline register which did not help solve what was wrong with the pipeline. I rebuilt the pipeline twice from scratch and encountered the same problems. This made me think that the hardware vhdl files I wrote were wrong but simulating the hardware did not produce any results. Needless to say, this leads to the present with no progress being made.

SECTION 4: HAZARD DETECTION

Without a working pipeline I did not move in further into this simulation. However, I will decode the mif as best as I can. The numbered mif file is as follows:

```
1-  lui $5, 0x0302
2-  ori $5, $5, 0x0100
3-  ori $6, $0, 2312
4-  lui $29, 0x1000
5-  sw $5, 0($29)
6-  sh $6, 4($29)
7-  ori $6, $0, 2826
8-  sh $a2, 6($29)
9-  lui $30, 4096
10- ori $30, $30, 8
11- add $15, $30, $0
12- notmain:
13- lbu $18, 0($29)
14- lbu $17, 4($29)
15- add $16, $17, $18
16- slti $10, $16, 11
17- beq $10, $0, case2
18- add $17, $16, $18
19- j case3
20- case2:
21- sub $17, $16, $18
22- case3:
23- slti $10, $17, 11
24- beq $10, $0, case4
25- add $18, $17, $16
26- j end
27- case4:
28- sub $18, $16, $17
29- end:
30- sb $18, 0($fp)
31- addi $30, $30, 1
32- addi $29, $29, 1
33- bne $29, $15, notmain
34- ori $19, $0, 8
35- subu $29, $29, $19
36- lw $1, 0($29)
37- lw $2, 4($29)
38- lw $3, 8($29)
39- lw $4, 12($29)
40- here:
41- j here
```

From line 1 to 2 ,$5 is loaded in the first instruction but also ORI in the second instruction. This would require a stall in order for ORI to correctly perform.

From line9 to line 10, line 11, the consecutive LUI and ORI and ADD instructions share the same registers creating a data hazard that can be solved with stalling.

From line 13 to line 15, $18 needs to be forward in order avoid data dependence.

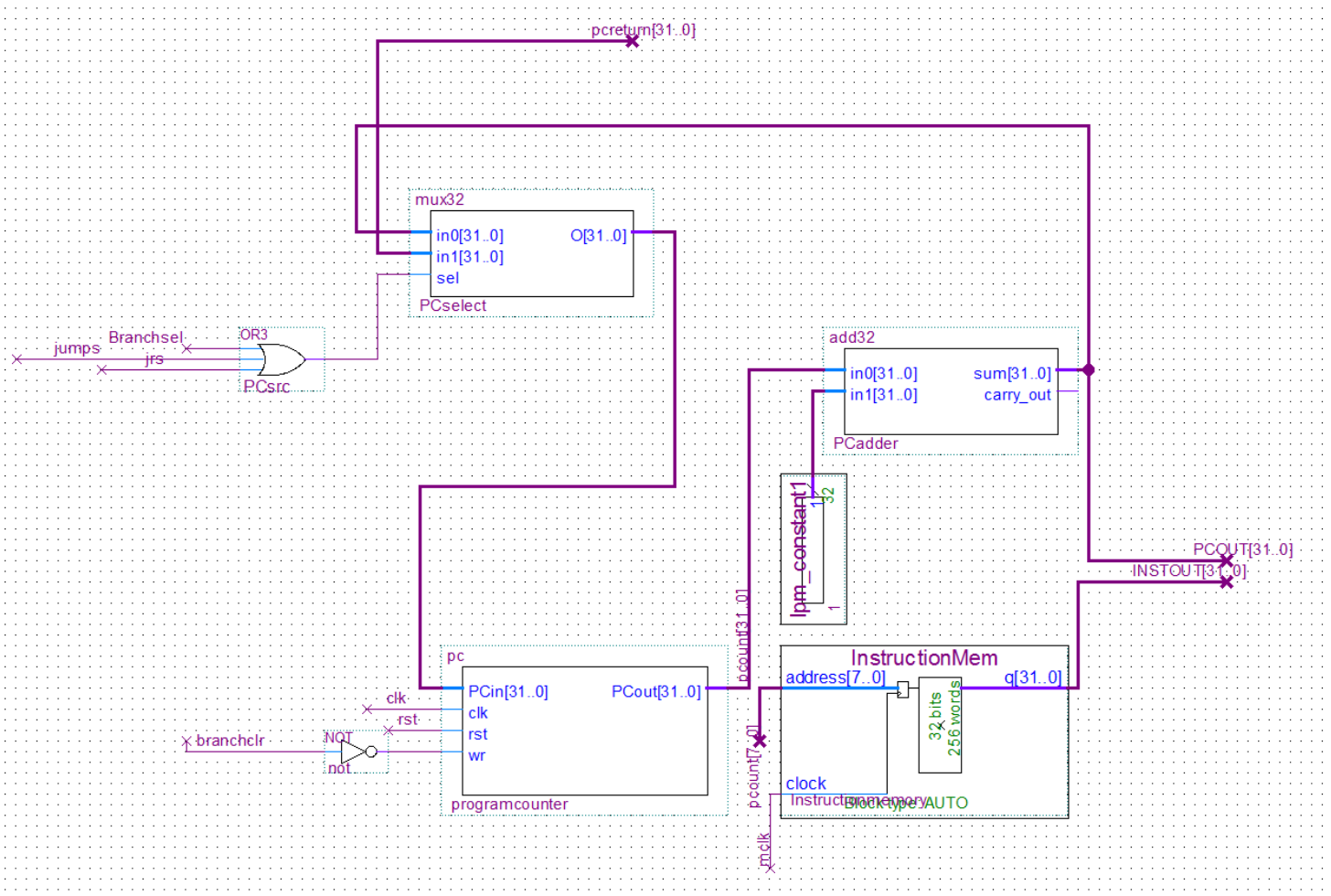
From line 14 to 15, there is a data hazard involving $17 that can be solved with forwarding.

Line 17, 24, and 33 have branch instructions which create need to be stalled or flushed.

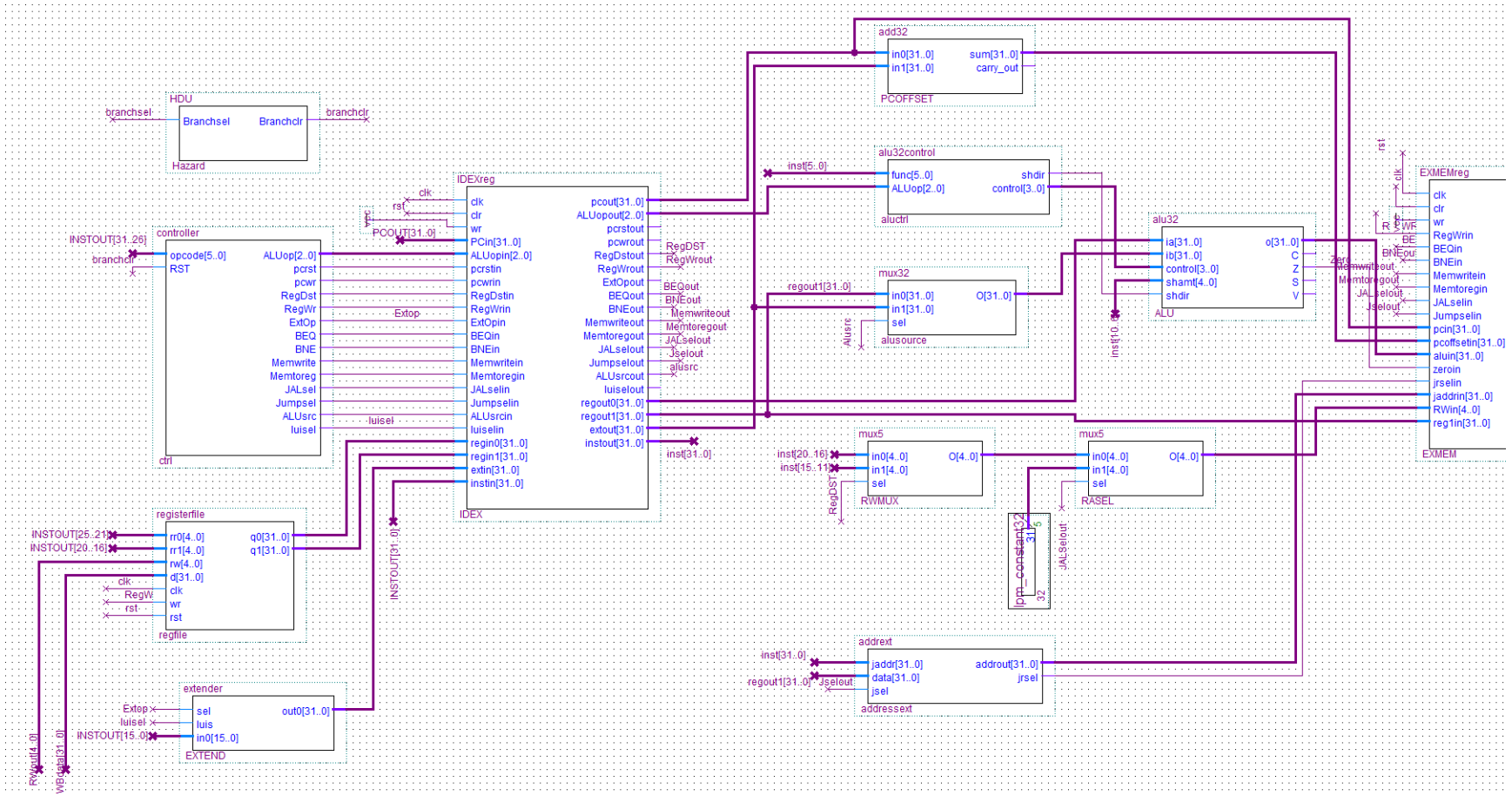
The program ends with 4 LW instructions that load the registers a0-a4 with the following values: 110a0908,03020100,00000b0a,00000000.

APPENDIX

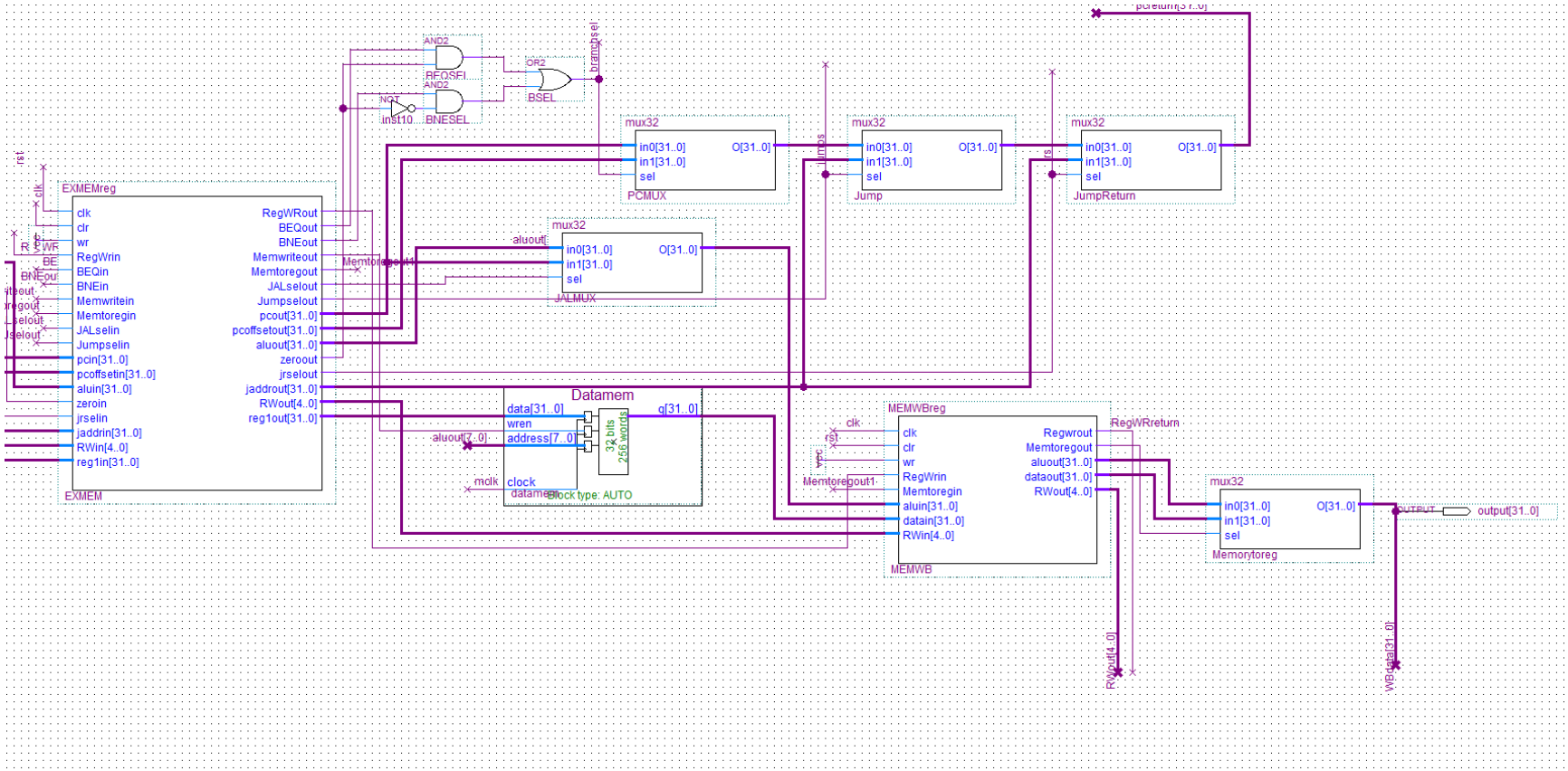
BDF Screen captures



IF/ID


HDU
branchsel
Branchsel
Branchclr
branchclr
Hazard
add32
in0[31..0]
in1[31..0]
sum[31..0]
carry_out
PCOFFSET
alu32control
inst[5..0]
func[5..0]
ALUop[2..0]
shdir
controll[3..0]
aluctrl
IDEXreg
clk
clr
wr
PCin[31..0]
ALUopin[2..0]
pcrstin
pcwrin
RegDstin
RegWrin
ExtOpin
BEQin
BNEin
Memwritein
Memtoregin
JALselin
Jumpselin
ALUsrcin
luiselin
regin0[31..0]
regin1[31..0]
extin[31..0]
instin[31..0]
pcout[31..0]
ALUopout[2..0]
pcrstout
pcwrout
RegDstout
RegWrout
ExtOpout
BEQout
BNEout
Memwriteout
Memtoregout
JALselout
Jumpselout
ALUsrcout
luiselout
regout0[31..0]
regout1[31..0]
extout[31..0]
instout[31..0]
IDEX
PCOUT[31..0]
RegDST
RegWrout
Extop
luisel
alusrc
controller
INSTOUT[31..26]
branchclr
opcode[5..0]
RST
ALUop[2..0]
pcrst
pcwr
RegDst
RegWr
ExtOp
BEQ
BNE
Memwrite
Memtoreg
JALsel
Jumpsel
ALUsrc
luisel
ctrl
mux32
regout1[31..0]
in0[31..0]
in1[31..0]
sel
O[31..0]
alusource
alu32
ia[31..0]
ib[31..0]
controll[3..0]
shamt[4..0]
shdir
o[31..0]
C
Z
S
V
ALU
EXMEMreg
clk
clr
wr
RegWrin
BEQin
BNEin
Memwritein
Memtoregin
JALselin
Jumpselin
pcin[31..0]
pcoffsetin[31..0]
aluin[31..0]
zeroin
jrselin
jaddrin[31..0]
RWin[4..0]
reg1in[31..0]
EXMEM
Zero
Memwriteout
Memtoregout
JALselout
Jselout
mux5
inst[20..16]
inst[15..11]
in0[4..0]
in1[4..0]
sel
O[4..0]
RegDST
RWMUX
RASEL
JALSelout
lpm_constant32
31..5
32
registerfile
INSTOUT[25..21]
INSTOUT[20..16]
rr0[4..0]
rr1[4..0]
rw[4..0]
d[31..0]
clk
wr
rst
q0[31..0]
q1[31..0]
regfile
INSTOUT[31..0]
addrext
inst[31..0]
regout1[31..0]
jaddr[31..0]
data[31..0]
jsel
addrout[31..0]
jrsel
Jselout
addressext
extender
Extop
luisel
INSTOUT[15..0]
sel
luis
in0[15..0]
out0[31..0]
EXTEND
RWout[4..0]
WBLuisel[31..0]

ID/EX



EX/MEM and MEM/WB