University of California, Berkeley – College of Engineering

Department of Electrical Engineering and Computer Sciences

Fall 2003 Instructor: Dave Patterson 2003-10-8

CS 152 Exam #1

Personal Information

First and Last Name	Answer Key
Your Login	cs152-____ ____
Lab/Discussion Section Time & Location ***you attend***	
"All the work is my own. I have no prior knowledge of the exam contents nor will I share the contents with others in CS152 who have not taken it yet."	***(Please sign)***

Instructions

- Partial credit may be given for incomplete answers, so please write down as much of the solution as you can.
- Please write legibly! If we can't read it from 3 feet away, we won't grade it!
- Put your name and login on each page.
- This exam will count for 16% of your grade.

Grading Results

Question	***Max. Points***	***Points Earned***
1	**30**	
2	**35**	
3	**35**	
Total	**100**	

Name: ________________________ Login: _______________________

Question 1: Pipelined Processors (Jack & John's Questions)

Suppose we design **a 7 stage pipelined processor with 4 execution/memory stages (EX1 through EX4) and hardware interlocks:**

Assume an integer ALU latency of 0 and branches are still delayed and calculated in the decode stage (like in the 5-stage pipeline). Additionally, assume that the register file is designed so that when a value is written then it will be ready later in that same cycle.

Stage usage for R-type integer instructions:

IF	→	ID	→	EX1	→	EX2	→	EX3	→	EX4	→	WB
fetch		decode		ALU		nop		nop		nop		regwr

Suppose that data memory accesses take 2 EX cycles: one cycle to calculate the effective address (***addrc***), and one cycle to access the result (***mem***) (like the 5-stage pipeline) for both floating-point stores and integer stores.

Stage usage for memory access instructions:

IF	→	ID	→	EX1	→	EX2	→	EX3	→	EX4	→	WB
fetch		decode		addrc		mem		nop		nop		regwr

We have additional stages (EX3 and EX4) because our processor supports floating-point operations.

Name: ________________________ **Login:**_______________________

Question 1: Pipelined Processors [continued]

1a: Suppose that this processor requires a 2-cycle latency between the following instructions:

```
add.d     F4, F0, F2
s.d       F4, 0(R1)
```

(*add.d* is a floating-point addition instruction, and *s.d* is a floating-point store. F0, F2, F4 refer to floating point registers, and R1 refers to a regular integer register.)

Similar to the pictures for integer and memory instructions, fill in the values for the stages used for a FLOATING POINT ALU operation. If a stage is unused, put 'nop'. If a label is not obvious (like 'fetch') please explain it.

Stages used for a floating-point ALU instruction:

IF → **ID** → **EX1** → **EX2** → **EX3** → **EX4** → **WB**

fetch -> decode -> calc1-> calc2->calck3->calc4->writeback

The important thing to recognize is that floating point operations take all 4 execute cycles. Therefore, you need something like calc1, calc2, calc3, and calc4.
3 points if correct
2 points if you only said it finished in EX4
1 point for finishing in EX3 (due to thinking add.d forwards back to EX1 for the s.d.)

1b: How did you figure out the stages in **1a** without us telling you? Please be brief but precise.

2 cycle latency between add.d and s.d. This means that s.d. doesn't use the value F4 until EX2, so add.d must finish in EX4 to have this 2 cycle latency.

3 points for a correct explanation using 2-cycle latency
1 point for at least knowing that FP ops take longer than integer ops, 0 otherwise

Note: if you assumed 2 cycle latency meant 1 cycle latency, then you got 1/3 for 1a, and 2/3 for 1b if good explanation.

Name: ________________________ Login:_______________________

Question 1: Pipelined Processors [continued]

1c: Imagine that the following loop had just finished executing its 100th iteration on the 7-stage pipeline. How many more clock cycles will it take for the pipeline to finish the 101st iteration?

```
Loop:    l.d      F0, 0(R1)
         stall
         add.d    F4, F0, F2
         stall
         stall
         s.d      F4, 0(R1)
         stall
         addiu    R1, R1, -8
         bne      R1, R2, Loop
         stall (nop)
```

The stalls are shown above. This question was worth 7 points. Minus 1 for each stall not detected or extra stall, minus 2 for including drain/fill or counting wrong, plus 1 if the answer was wrong but consistent with 2 cycle latency meaning 1 cycle latency

Answer:___10____ clock cycles

1d: Reorder the instructions from question 1c so that the number of stalls is minimized. How many cycles are there between the finishing of the 100th and 101st iterations now?

Reordered code:

```
l.d      F0, 0(R1)
addiu    R1, R1, -8
add.d    F4, F0, F2
stall
bne      R1, R2, Loop
s.d      F4, 8(R1) #we must change the offset!
```

Any other solution with only 1 stall was accepted. Minus 2 for each extra stall, minus 2 if new solution doesn't execute properly, minus 1 for serious miscounting. If 1c was wrong, 4/7 for good reduction relative to 1c.

Answer:__6_____ clock cycles

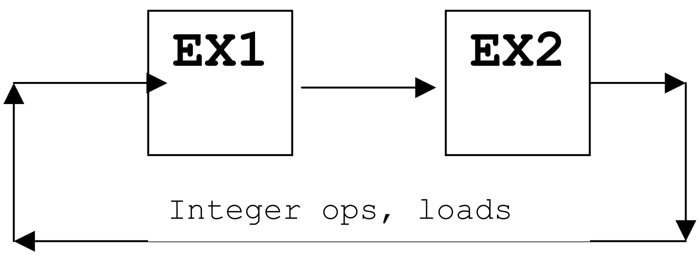
1e: A forwarding path from stage **X** to stage **Y** is written as:

X to Y

This means that the register after stage **X** can forward some value to the beginning of stage **Y** (i.e. after the register between stage Y-1 and Y).

In the table below, we have listed all of the possible forwarding paths to the ID and EX2 stages. We'd like you to tell us which ones are useful (in the sense that a forwarding circuit between the two stages will do useful work). If a forwarding path is useful, then also tell us what kinds of data will be forwarded along the path. For this problem, you may assume that **there are three general types of *data* that can be forwarded: integer ops, loads, and floating point ops.**

The path from EX2 to EX1 and the data that it carries is provided as an example.



Forwarding Path	**Useful?**	**If yes, which values can be forwarded? For which types of instructions?**
EX2 to EX1	YES NO	We can forward load values from memory back for integer ops.
ID to ID	YES NO	Note: We also accepted yes if jal was given as an explanation.
EX1 to ID	YES NO	We can forward integer ops to branch comparisons.
EX2 to ID	YES NO	We can forward load values or integer ops from memory to branch comparisons.
EX3 to ID	YES NO	We can forward integer ops or load values to branch comparisons.
EX4 to ID	YES NO	We can forward integer ops or load values to branch comparisons.

Name: ______________________ **Login:**______________________

Forwarding Path	**Useful?**	**If yes, which values can be forwarded? For which types of instructions?**
ID to EX2	YES NO	
EX1 to EX2	YES NO	
EX2 to EX2	YES NO	We can forward the results of a load word to a store word.
EX3 to EX2	YES NO	We can forward the results of a load word to a store word.
EX4 to EX2	YES NO	We can forward the results of a floating point op or a load word to a store word.

Attempts were made to keep the grading of this section consistent with wrong answers to 1a.

Name: ________________________ **Login:**_______________________

Question 2: Single Cycle Processor (Jack's Question)

Your single-cycle processor seems to be executing random instructions. You have been chosen by your group to investigate and find out why. On the next page you will find a picture of your datapath (note that this is a **slightly different datapath** than shown in lecture), and the control table is below. You suspect that the controller may be broken. You may assume that the modules within the datapath (i.e. extender, alu) all work.

	PCSrc	RegDst	RegWr	ExtOp	ALUSrc	ALUctr	MemWr	MemToReg
addu	0	0	1	1	X	0	0	0
subu	0	1	1	X	0	0	0	0
ori	0	1	1	0	X	2	0	0
Lw	0	1	1	1	1	0	1	1
Sw	0	x	0	0	1	0	1	x
beq	Equal	x	0	X	0	3	0	x
Jr	2	x	0	X	x	X	0	x

"Equal" means that PCSrc takes on the value of the equal signal coming out of the =0? module. This will either be 0 or 1.

Looking at your partners' online notebooks, you find the following (you may assume these to be correct):

- The register file (regWr) and data memory (MemWr) both write when their respective write signals are 1
- The extender will zero extend if the ExtOp bit is 0, and the extender will sign extend when the ExtOp control bit is 1.
- The data memory reads asynchronously but has synchronous writes (just like your single cycle lab).
- The =0? module will output 1 if the input to the module is 0, else it will output 0.

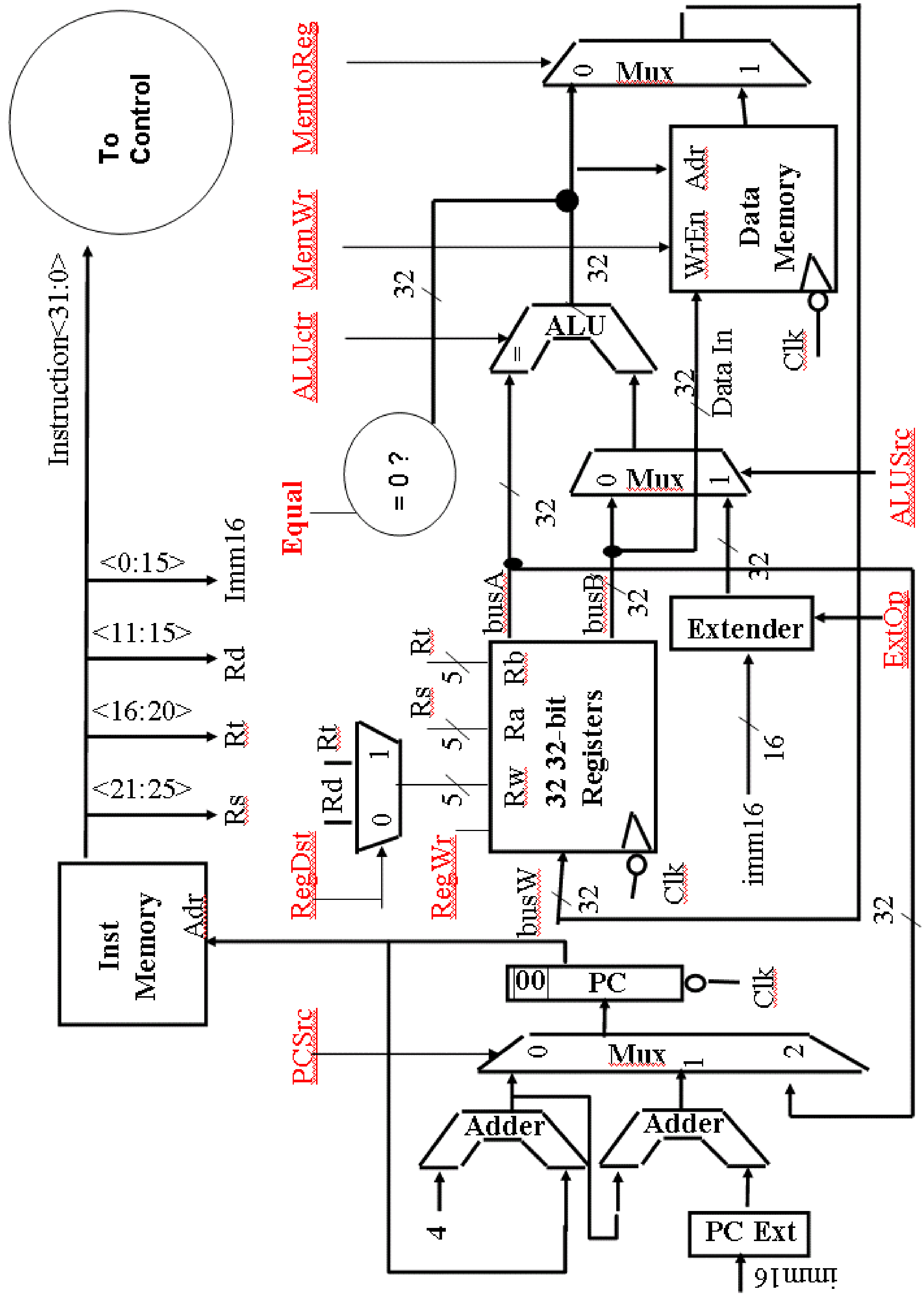
The ALUctr encoding is as follows:

Control bits	Operation
0	add
1	sub
2	or
3	Xor

For the following stream of instructions, what does your broken processor actually do? The first instruction has already been done for you as an example. If there is more than one possibility, please list all of them (note that this may be a different instruction, correct behavior, or an undefined instruction). If the incorrect result does not match a valid MIPS instruction, please give a sequence of instructions that correspond to the behavior. Also give a **very brief** explanation of your possibilities. For simplicity, we have used the actual register numbers rather than the names.

Name: ______________________________ Login: ____________________________

Datapath for Question 2 (Feel free to tear out.)



Name: ________________________ Login: _______________________

Question 2: Single Cycle Processor [continued]

Original Instruction	Possibilities
addu $1, $2, $0	addu $1, $2, $0 (if aluSrc is 0—correct behavior) addiu $1, $2, 33(if aluSrc is 1—incorrect behavior)
subu $4, $5, $6	addu $6, $5, $6 (regDst and ALUctr are wrong) 3 pts for changing the subu to addu 3 points for changing the destination register from $4 to $6
ori $7, $8, 0x0025	or $7, $8, $7 (if aluSrc is 0, we get incorrect behavior) ori $7, $8, 0x0025 (if aluSrc is 1, we get correct behavior) worth 9 points minus 3 if registers were slightly wrong ($8, $7) minus 4 for other register mistakes
beq $11, $12, 24	beq $11, $12, 24 (correct behavior). Despite using xor, the beq will still work because xor will output a 0 when the values are equal as well. worth 5 points. minus 1.5 for writing out an xor instruction in conjunction with a beq minus 2.5 for changing the "24" to a "6" (you were told that the modules and datapath were correct, so no reason to assume we didn't shift by 2)

sw $10, -12($31)	ExtOp is wrong, so it doesn't sign extend. sw $10, 0x0000FFF4($31)-even though this isn't a real instruction, it describes the behavior correctly. This sequence below was also accepted (note there were other similar ones that performed the same thing):
	lui $at, 0 (minus .5 for not have lui)
	ori $at, $at, 0xFFF4
	add $at, $at, $31
	sw $10, 0($at)
	2/5 if recognized ExtOp problem
	3/5 if tried to get there
	4/5 if sw $10, FFF4($31) -wrong because this instruction will sign extend. Different behavior

lw $9, -16($29)	memWr is 1, so this instruction will do a lw $9, -16($29) and sw $9, -16($29). However, you have to be careful of the timing, and need to recognize both instructions happen at the same time. In essence, this is a "swap" instruction. worth 10 points. 8 points for having both instructions 7 points for both instructions but bad explanation 6 points for something with lw 5 points for just having sw 4 points for some sort of understanding 3 points for writing down correct behavior

Name: ______________________ **Login:**_____________________

Question 3: Multicycle Processor (Kurt's Question)

We'd like to give you a feel for how microprogramming can help out with tricky CISC instructions. We'd like you to implement a new addressing mode (register indirect; i.e. register value is an address with no offset) for the sub instruction:

sub.mem $rd $rs $rt # **Mem**[$rd] = **Mem**[$rs] **-** **Mem**[$rt]

Your solutions to 3a-d will be graded, in part, on elegance!

3a: Please come up with a suitable machine representation for sub.mem. You may assume that opcode 44_{hex} and funct 44_{hex} are both unused. Make your representation clean and complementary to the MIPS datapath.

Here's an example of what we are looking for (for *nor*):

NOR

nor rd, rs, rt	0	rs	rt	rd	0	0x27
	6	5	5	5	5	6

Now do the same for sub.mem.

5 Points

We gave full credit for the following answers:

0x14 rs rt rd 0 0

0 rs rt rd 0 0x14

We gave 1pt extra credit for this answer:

0x14 rs rt rd 0 34 (The funct code for sub)

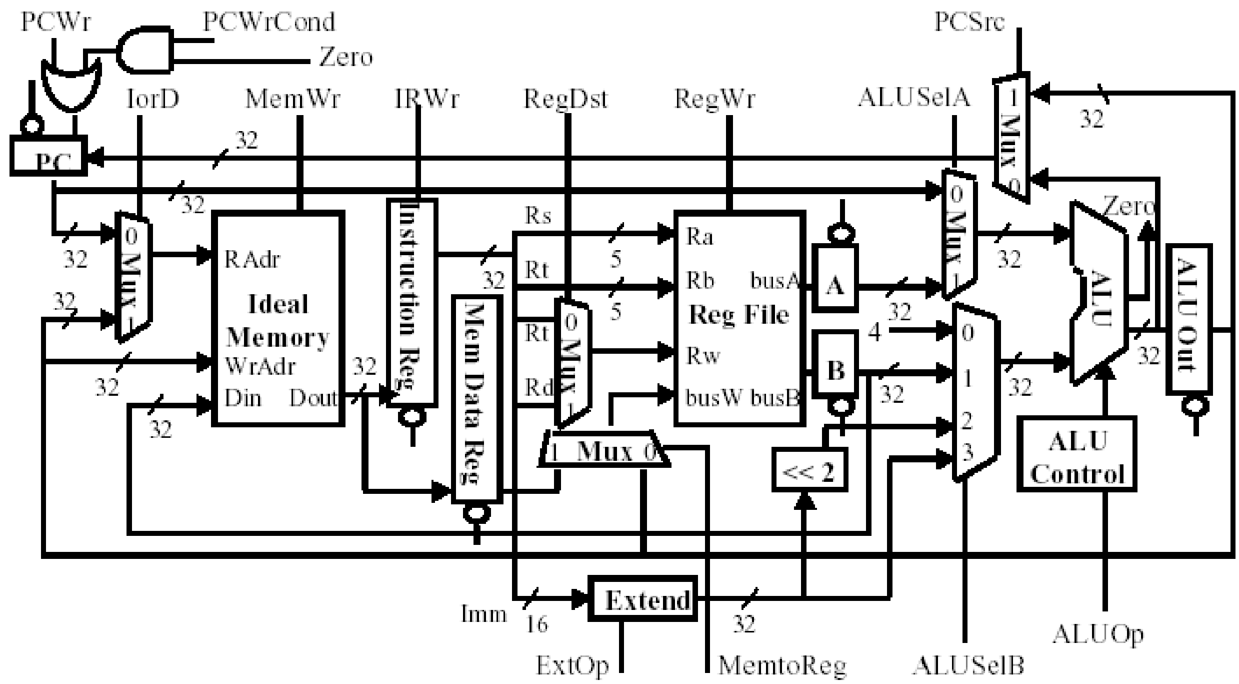
We took off a point for using up extra opcodes/functs (as only one is necessary, or, if you use two, one should be sub) as follows:

0x14 rs rt rd 0 0x14 : 4/5

Anything else got a zero.

Name: ______________________ **Login:** _____________________

Question 3: Multicycle [continued] - Datapath



3b: Above is the multicycle datapath from lecture. Please draw below any changes to the datapath to support your sub.mem. **YOU MAY NOT ADD ANY REGISTERS!!!** (Do everything with muxes and simple, combinational modules.) Don't redraw the entire datapath – just circle the areas you're changing in the above diagram and then re-draw modules and muxes that you have changed (including control line names) below. **DRAW LEGIBLY.**

[16 Points] While we saw all sorts of solutions to this problem, there were 3 main things that had to happen in the datapath:

1. [6 points] A way to put Reg[Rs] (A) and Reg[Rd] (B) on the RAdr line of the Memory. Some people chose to add A-only and B-only signals to the ALU while others chose to simply add to lines to the IorD mux (namely: A and B).
2. [6 points] A way to store Mem[Reg[Rs]] and Mem[Reg[Rt]] simultaneously and feed them to the ALU. Most people chose to add a mux in front of A and B such that Dout (from Memory) could be one of the inputs to A and B. Other people chose to in-line the values on the MDR i.e. Mem[Reg[Rs]] would be on the output wires of the MDR while Mem[Reg[Rt]] would be on the input wires and add new wires on ALUSelA and SLUSelB. Both were acceptable.
3. [4 points] A way to store the result of the ALU subtract into memory. Most people chose to add a mux in front of Din that had Aluout as an input. Rd can't be added as an input to WrAddr. (Why not?) Rather, most people chose to add a mux in front of Ra on the regfile and to add a muc in front of WrAddr with A as an input.

For each of these three ideas, we gave full credit for coming up with something that would work, half credit for something that would almost work (like using Rd rather than Mem[Rr] as WrAddr) and 0 otherwise.

Name: ________________________ **Login:** ________________________

Question 3: Multicycle [continued] - Datapath

Field Name	Values For Field	Function of Field
ALU	Add	ALU Adds
	Sub	ALU subtracts
	Func	ALU does function code (Inst[5:0])
	Or	ALU does logical OR
SRC1	PC	PC $\Rightarrow$ 1st ALU input
	rs	R[rs] $\Rightarrow$ 1st ALU input
SRC2	4	4 $\Rightarrow$ 2nd ALU input
	rt	R[rt] $\Rightarrow$ 2nd ALU input
	Extend	sign ext imm16 (Inst[15:0]) $\Rightarrow$ 2nd ALU input
	Extend0	zero ext imm16 (Inst[15:0]) $\Rightarrow$ 2nd ALU input
	ExtShft	2nd ALU input = sign extended imm16 << 2
ALU Dest	rd-ALU	ALUout $\Rightarrow$ R[rd]
	rt-ALU	ALUout $\Rightarrow$ R[rt]
	rt-Mem	Mem input $\Rightarrow$ R[rt]
Memory	Read-PC	Read Memory using the PC for the address
	Read-ALU	Read Memory using the ALUout register for the address
	Write-ALU	Write Memory using the ALUout register for the address
MemReg	IR	Mem input $\Rightarrow$ IR
PC Write	ALU	ALU value $\Rightarrow$ PCibm
	ALUoutCond	If ALU Zero is true, then ALUout $\Rightarrow$ PC
Sequence	Seq	Go to next sequential microinstruction
	Fetch	Go to the first microinstruction
	Dispatch	Dispatch using ROM

3c: Above is the microassembly language description from lecture. Please describe any additions or modifications to microassembly language necessary to support sub.mem. Be sure to include the field name, the new field values, as well as **EXACTLY** which control lines are set when the field value is asserted. **Be precise and print legibly!**

[7 Points.] Solutions for this section are obviously dependent on changes to the datapath. Generally, we took off one point for each control line of each value of each field that was unspecified or mis-specified. (We gave clock-enables to everyone for free.) Here's a solution for the datapath that stores values into A and B:

Field	Value	Meaning
Memory	A->A	A <= Mem[A]
Memory	B->B	B <= Mem[B]
Memory	ALU->A	Din <= ALUout; WrAddr <= A
MemReg	Rd->A	A <= Reg[Rd]

[There are cleaner solutions that involve creating new fields, but we wanted to keep the example solution closer to what people actually did.]

Question 3: Multicycle [continued] - Datapath

Label	ALU	SRC1	SRC2	ALUDest	Memory	MemReg	PCWrite	Sequence
Fetch	Add	PC	4		ReadPC	IR	ALU	Seq
	Add	PC	ExtShft					Dispatch
RType	Func	rs	rt					Seq
				rd-ALU				Fetch
BEQ	Sub	rs	rt				ALUoutCond	Fetch

3d: Above are the implementations for a few MIPS instructions in our microassembly language. Please give a complete microcode assembly implementation for your sub.mem. You may assume that dispatch will jump to a label named 'sub.mem'.
Print legibly.

Label	ALU	SRC1	SRC2	ALUDest	Memory	MemReg	PCWrite	Sequence
Sub.mem					A->A			seq
					B->B			seq
	sub	rs	rt			Rd->A		seq
					ALU->A			fetch

[7 points] Generally, full credit if it unless...

-5 for using incompatible fields simultaneously.

-7 for stretching the clock cycle (by doing a memory access and a dependent subtract in the same cycle - most people who lost points here tried the in-line MDR approach).

3e. EXTRA CREDIT: QUITE DIFFICULT AND NOT WORTH MANY POINTS: (We suggest that you finish all the other problems on the exam before you attempt this one.) 5 EC points.

Using your new datapath and control from above, please implement the Subtract and Branch if Negative (SBN) instruction in microcode:

sbn $rs $rt immed # **Mem[$rs] = Mem[$rs]-Mem[$rt]**
if (Mem[$rs]<0) goto PC+4+immed

Please give a machine representation (like 3a), draw any changes to the datapath (like 3b), explain any new microassembly fields and values (like 3c), and give the complete microassembly sequence (like 3d).
Again: No new registers.

Hint: You already have the sub.mem part almost done – the hard part is figuring out how to jump.

Machine Representation:

tbd

Datapath Changes
(You may assume your new datapath from 3b.)

tbd

Question 3: Multicycle [continued] - Datapath

3e continued:

Additions to microassembly language:

tbd

SBN microassembly implementation (complete):

tbd