Name _____(answer key)_____

Computer Architecture and Engineering
CS152 Quiz #1
February 14th, 2011
Professor Krste Asanović

Name:___<ANSWER KEY>___

This is a closed book, closed notes exam.
80 Minutes
15 Pages

Notes:

- Not all questions are of equal difficulty, so look over the entire exam and budget your time carefully.
- Please carefully state any assumptions you make.
- Please write your name on every page in the quiz.
- You must not discuss a quiz's contents with other students who have not yet taken the quiz. If you have inadvertently been exposed to a quiz prior to taking it, you must tell the instructor or TA.
- You will get **no** credit for selecting multiple-choice answers without giving explanations if the instruction ask you to explain your choice.

Writing name on each sheet	_____	2 Points
Question 1	_____	18 Points
Question 2	_____	30 Points
Question 3	_____	30 Points
TOTAL	_____	**80 Points**

Question 1: Microprogramming (18 points)

In this question we ask you to implement a useful string instruction, *string copy* (**`strcpy`**):

`strcpy Rd, Rs`

6	5	5	16
`strcpy`	**Rd**	**Rs**	**unused**

The `strcpy` instruction provides the programmer the ability to copy a string directly from one location in memory (**M[Rs]**) to another location in memory (**M[Rd]**).

For this problem, think of a string as an array of 4-byte words, with the last element being zero (the string is "null terminated").

Starting from the memory location addressed by **Rs** (**M[Rs]**), keep copying one 4-byte word at a time to an other memory location, starting at the address **M[Rd]**, until you hit the null terminating character (zero). Do not forget to copy the null character too!

Finally, once the `strcpy` has finished, **Rd** and **Rs** will hold the address of the null character at the end of their respective strings.

The instruction definition for `strcpy` requires that the strings pointed to by **Rs** and **Rd** do not overlap in memory.

For reference, we have included the actual bus-based datapath in Appendix A (Page 16) and a MIPS instruction table in Appendix B (Page 17)**.** You do not need this information if you remember the bus-based architecture from the online material. **Please detach the last two pages from the exam and use them as a reference while you answer this question.**

Q1.A (13 points)

Fill out Worksheet 1 for the `strcpy` instruction. You should try to optimize your implementation to reduce the number of cycles necessary and to have as many signals be "don't cares" as possible. You do not have to worry about the busy signal. You may not need all the lines in the table for your solution.

Name _____(answer key)_____

State	PseudoCode	ldIR	Reg Sel	Reg Wr	en Reg	ldA	ldB	ALUOp	en ALU	ld MA	Mem Wr	en Mem	Ex Sel	en Imm	μBr	Next State
FETCH0:	MA <- PC; A <- PC	0	PC	0	1	1	*	*	0	1	*	0	*	0	N	*
	IR <- Mem	1	*	*	0	0	*	*	0	0	0	1	*	0	N	*
	PC <- A+4	0	PC	1	1	0	*	INC_A_4	1	*	*	0	*	0	D	*
. . .																
NOP0:	microbranch back to FETCH0	0	*	*	0	*	*	*	0	*	*	0	*	0	J	FETCH0
strcpy:	MA <- Rs; A <- Rs	0	Rs	0	1	1	*	*	0	1	*	0	*	0	N	
	B <- Mem	0	*	*	0	0	1	*	0	0	0	1	*	0	N	
	MA <-Rd	0	Rd	0	1	0	0	*	0	1	*	0	*	0	N	
	Mem <- B if (B==0) uBr to Fetch0	0	*	*	0	0	0	COPY_B	1	0	1	1	*	0	Z	FETCH0
	Rs <- A+4	0	Rs	1	1	*	*	INC_A_4	1	*	*	0	*	0	N	
	A <- Rd	0	Rd	0	1	1	*	*	0	*	*	0	*	0	N	
	Rd <- A+4 J to strcpy	0	Rd	1	1	*	*	INC_A_4	1	*	*	0	*	0	J	strcpy

Worksheet 1

A few notes:

-LdIR is zero for all uops because we keep needing to read the actual values of Rs, Rd which are stored in the IR register

-ldMA is kept at 0 when performing a memory operation because memory operations are multi-cycle and thus you need to hold the memory address constant (this logic also applies to ldA,ldB when used as sources for memory).

Q1.B Changing the micro-architecture to speed up strcpy (5 points)

You probably found the current micro-architecture presented in Appendix#A to be somewhat awkward for implementing `strcpy` in **Q1.A**. In particular, a lot of information has to be repeatedly re-read from the register file and moved back to the ALU to check for null-terminating characters and updating addresses.

Can you think of a change to the datapath, ALU, and/or control logic that can improve the performance (cycles / instruction) of `strcpy`? (hint: adding ALU operations that touch the ALU operand B register may be helpful).

Also, please *explain* why your modification(s) will be an improvement.

There are quite a few ideas that would improve performance (here are a few):

Modification:
    Add a INC_B_4 alu op.

This allows you to keep one of the addresses permanently in register B, and increment it by 4 when necessary.

-OR-

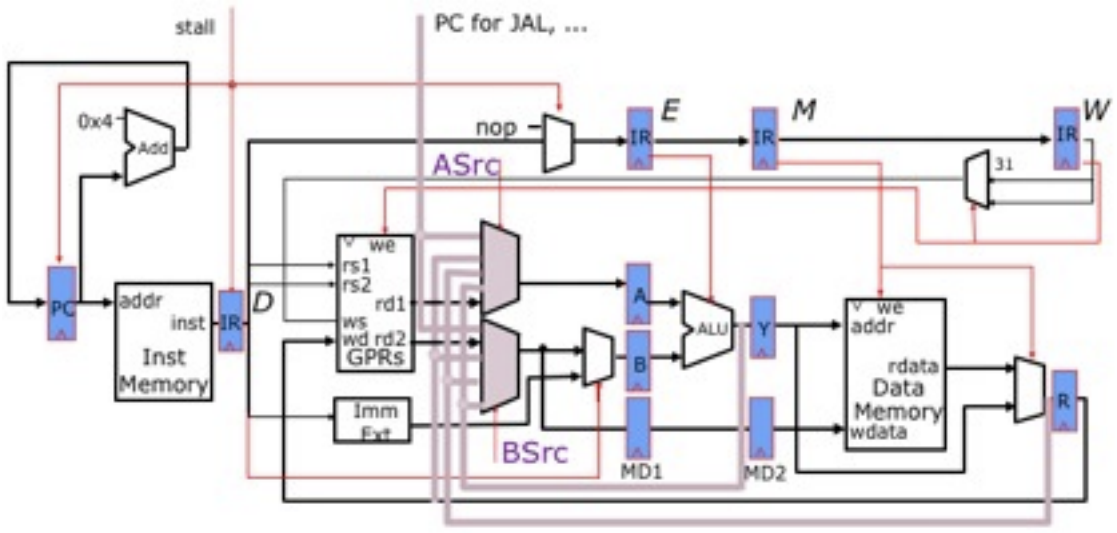
Modification:
    Add a third ALU operand register C
    Add INC_B_4
    Add COPY_C

Keep Rs,Rd in registers A,B, and move the temporary value to C and check for the null-terminator with COPY_C.

Question 2: Load Value Speculation (30 points)

In Lecture 4, we introduced a fully bypassed 5-stage MIPS pipeline. We have reproduced the pipeline diagram below.

For this problem ***ignore branches and jumps***.



Q2.A.i (2 points)

Even a fully bypassed 5-stage pipeline has to stall sometimes. Can you provide an instruction sequence that would cause a stall in a fully bypassed 5-stage pipeline?

```
LW  R1, 0(R2)
ADD R2, R1, R0
```

Q2.A.ii (2 points)

How many bubbles get inserted into the pipeline due to the stall condition you produced in **Q2.A.i**?

1 nop gets inserted

(stall dependent instruction in the *Decode* stage for one cycle, while waiting for load value to become available and bypassed)

Q2.A.iii (2 points)

If 20% of all instructions are loads, and 50% of these loads are followed by dependent instructions, what is the CPI of the typical fully bypassed 5-stage pipeline? ***Ignore control hazards***.

10% of all instructions cause a single-cycle stall, so CPI = 1.1

(if program was N instructions, that would take (N+0.1N) cycles to execute,
or CPI = (N+0.1N)/N = 1.1).

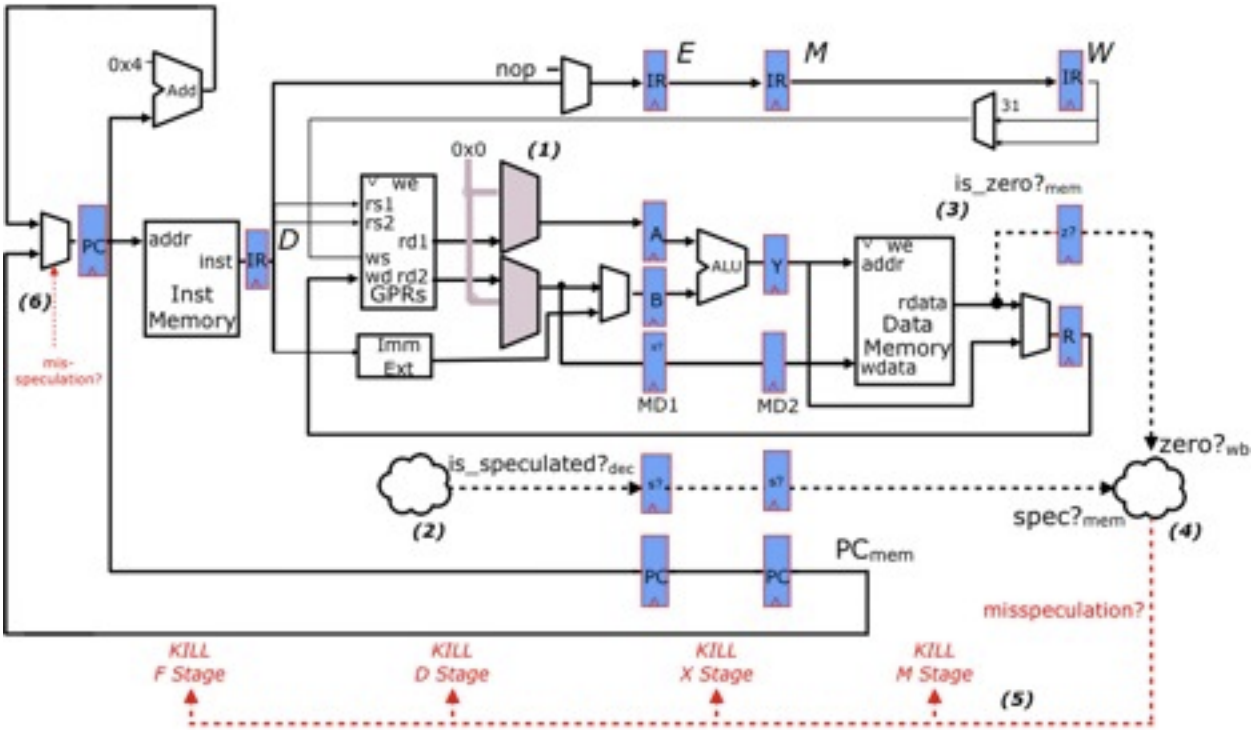
Q2.Part B

One way to get around this stall situation is to speculate that the loaded value returning from memory will be zero (which is true relatively often).

Therefore, we can (1) mux in zero to the bypass mux in the *Decode* stage when the load value is unavailable, and (2) set a bit to denote the load-use value is "speculated". Once the load finishes, (3) check if the load value was in fact "zero". At the *Writeback* stage we can then (4) check for misspeculations, (5) flush the pipeline if the load value was misspeculated, and (6) restart from the PC of the instruction which used the misspeculated load value.

Again, for the purposes of this problem, ***ignore control hazards***.

This new, proposed datapath is shown below:



Note: for the sake of clarity, some of the bypass lines and control signals have been removed *from the drawing*, but this is still a fully bypassed pipeline.

Q2.B.i (4 points)

While a new datapath has been designed (shown above), we still need to get the control logic right.

What is the logic that generates the "is_speculated?" signal in the *Decode* stage?

Is_Speculated? $_{dec}$ =

$$(OPC_{EXE} == LW) \;\&\&\; (RD_{EXE} != 0) \;\&\&\; [((RD_{EXE} == RS1_{DEC}) \;\&\&\; RE1_{DEC}) \;||\; ((RD_{EXE} == RS2_{DEC}) \;\&\&\; RE2_{DEC})]$$

```
OPC = opcode
RD    = register-destination for instruction
RS1 = register-source #1
RE1 = register-enable for reg source #1
RS2 = register-source #2
RE2 = register-enable for reg source #2
```

You have to check that RD_exe is not writing to $0, because register $0 can not be modified (and is always zero). You also need to check that the source register is actually used (read enable signal) otherwise you could be bypassing and speculating when the register isn't even used! While neither of these two things actually hurt *correctness* (because you would be overly conservative), it would *kill* performance.

Q2.B.ii(2 points)

What is the logic driving the "misspeculation" signal, as a misspeculation is detected in the *Writeback* stage?

Misspeculation $_{Wb}$ =

$$is\_speculated?_{MEM} \;\&\&\; !(is\_zero?_{WB})$$

Q2.Part C

Q2.C.i(6 points)

To get a better understanding of how the pipeline behaves, please fill out the following instruction/time diagrams for the following scenarios: 1) without any load-speculation, 2) with load speculation and correctly speculated, and 3) with load speculation but misspeculated, using the instruction sequence below (the load-use dependency has been bolded):

```
I1: LW  R1, 0(R2)
I2: ADD R3, R1, R0
I3: SUB R5, R0, R0
I4: XOR R6, R0, R0
I5: AND R7, R0, R0
I6: OR  R8, R0, R0
```

In each chart, the very first instruction (`LW`) has been done for you, as well as the first cycle of the (`ADD`) instruction. Please fill out the rest of the diagrams for the remaining instructions. The sequence of six instructions may finish before cycle $\mathbf{t_{12}}$.

Name _____(answer key)_____

Chart 1: Without Load Speculation (regular fully bypassed 5-stage pipeline)

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}
LW	F	D	X	M	W									
ADD		F	D	D	X	M	W							
SUB			F	F	D	X	M	W						
XOR					F	D	X	M	W					
AND						F	D	X	M	W				
OR							F	D	X	M	W			

Chart 2: With Load Speculation, and correctly speculated

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}
LW	F	D	X	M	W									
ADD		F	D	X	M	W								
SUB			F	D	X	M	W							
XOR				F	D	X	M	W						
AND					F	D	X	M	W					
OR						F	D	X	M	W				

Chart 3: With Load Speculation, but misspeculated11/100-17

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}
LW	F	D	X	M	W									
ADD		F	D	X	-	F	D	X	M	W				
SUB			F	D	-		F	D	X	M	W			
XOR				F	-			F	D	X	M	W		
AND					-				F	D	X	M	W	
OR										F	D	X	M	W

Q2.C.ii(2 points)

Using the information you discovered in **Q2.C.i** (particularly Chart 3), how many bubbles are inserted into the pipeline when a load value is misspeculated?

4 bubbles, since we are killing the F, D, X, and M stages

-1 point for getting this wrong, but being consistent with the chart from Q2.C.i

Q2.Part D: Measuring CPI

Q2.D.i(2 points)

Assume that 100% of all load values that are speculated are correctly speculated (absolute best case). If, for a given program, 20% of all instructions are loads, and 50% of those loads are immediately followed by a dependent instruction, what is the CPI?

CPI = 1

(since the load-uses are being perfectly predicted, no stalls occur)

Q2.D.ii(2 points)

Let us be more realistic. Assume for a given program that 20% of all instructions are loads, and 25% of all loads in the program return a load value of zero.

Also assume that 50% of all loads are followed immediately by a dependent instruction (also assume that there is *no correlation* between loads that return zero and loads that are followed by dependent instructions).

What is the new CPI?

15% of loads are misspeculated, but only half of these are actually followed by a dependent instruction, so 7.5% of all instructions cause a misspeculated load-use.

For a program of N instructions, CPI = (N+0.075NX) /N, where X is the number of bubbles that get inserted (4, as found in Q2.C.ii).

so CPI = (N+0.3N)/N
= 1.3

Q2.D.iii(2 points)

What fraction of loads must be correctly speculated for the new datapath to be worthwhile?

Let Y be the fraction of instructions that cause 4 bubbles to be added due to a misspeculation (i.e., the fraction of loads that return non-zero AND are followed by a dependent instruction)
Let N be the number of instructions in the program

CPI without load speculation is 1.1 (as found in Q2.A.iii)

CPI = (N+4YN)/N = (1+4Y)

Set equal to 1.1 and solve to find the break-even point:

1.1 = 1+4Y
0.1 = 4Y
Y = 0.025, or 2.5%

Only 2.5% of all instructions are allowed to cause a pipeline kill due to a misspeculation.

As mentioned before, if we assume that 20% of all instructions are loads, and 50% of these loads are followed by dependent operations then:

10% of all instructions *can* cause a misspeculation, and since only 2.5% of *all* instructions are allowed to be mispredicted, then 75% of all loads must be correctly speculated (non-zero).

Q2.Part E: A Better Datapath?

Q2.E.i(4 points)

As you have shown in previous parts, the penalty of flushing the entire pipeline is very high. Propose a new datapath that can do better. *Describe its control in words*.

On a misspeculation, bypass all instructions and state back a cycle and re-execute them (the X op will now have the correct load value). This requires adding more state (inst registers) to allow for re-executing a stage again and bypass paths to recycle all of the state back a cycle.

(+2/4) was given for saying "move the misspeculation logic to the *Memory* stage". This only says 1 out of 4 bubbles, and it pushes a lot of logic onto the critical path that dramatically hurts seconds/cycle (Mem->Zero Compare->Misspec Calc->Broadcast Kill->Mux in NOP

(+3/4) was given for adding an ALU to the *Memory* stage to redo misspeculations without stalling the pipeline. This still breaks down for instructions that memory ops that depend on that result.

Question 3: Iron Law of Processor Performance

(30 points)

Mark whether the following modifications will cause each of the *three* categories to **increase, decrease**, or whether the modification will have **no effect**.

Assume the rest of the machine remains unchanged. Also, we are measuring these metrics from the viewpoint of the user-code. Thus, an Operating System call will simply appear to be a single instruction that takes many, many cycles to execute.

Explain your reasoning to receive credit.

-2 points for each wrong cell.

"Not ISA visible" means that no change to the actual binary occurs (since the change doesn't affect the ISA which is a contract between the hardware microarchitect and the software programmer/compiler).

Name _____(answer key)_____

		Instructions / Program	Cycles / Instruction	Seconds / Cycle
a)	move branch/jump logic from the *Execute* stage to the *Decode* stage	unchanged (not ISA visible)	decreases (less NOPs inserted by having earlier branch resolution)	increases (instead of using ALU in X, we add a comparator to Decode, and lengthen the cycle by having a reg-reg comparision in Decode after the bypass muxes, which then influences the PC_mux in the IF stage)
b)	Modifying the ISA (and thus the micro-architecture) to use hardware interlocking instead of software interlocking for both branch delay slots and load-use delay slots	decreases (do not have to insert NOPs to fill delay slots)	increases (sometimes we have to stall for load-use dependencies now, and execute NOPs for misspeculated branches)	(probably) increases (control logic added that must stall certain stages or insert NOPs when misspeculations occur) -1 for "no change"
c)	Removing a complex instruction from the hardware implementation, and instead execute it by throwing an illegal opcode trap and letting the exception handler execute the instruction in software	unchanged (not ISA visible)	increase (a lot of software instructions will have to be executed to perform a single complex instruction)	decrease (hopefully removing the complex instruction will simplify the pipeline in a manner that decreases Secs/ Cycle) -1 for "no change"

		Instructions / Program	Cycles / Instruction	Seconds / Cycle
d)	Change the ISA from 32-bits to 64-bits (i.e., all registers in the Register File are now 64-bits wide and the ALU performs 64-bit operations).	decrease (64-bit ops won't require being synthesizing from multiple 32-bit versions) -1 for "no change" unless the answer was qualified by stating no 64-bit arithmetic occurs in the program	unchanged (widening the datapath doesn't change the logic in the pipeline) - OR - increases more cache misses occur (because ints and address pointers are now 64bits which decrease the number of variables that will fit inside the caches)	increases (bigger registers, wider datapaths, and larger ALUs will all work to increase Secs/ Cycle)
e)	Merge the *Decode* and *Execute* stages into a single stage (i.e., perform a register read, then an ALU execution in the same cycle).	unchanged (not ISA visible)	decreases will decrease due to branches/ jumps in more detail: if branches resolved in *Decode*, branch CPI goes down because we no longer must interlock for branch dependent on a load if branches resolved in *Execute*, branch CPI goes down because instead of eating two cycles on a mispredict, we only lose 1 (because of the merged Dec+Exe) -1 for not explaining which hazards are resolved	increases (more work to fit into a single stage)