

CS 573
Midterm Exam
March 2, 2005

The following exam is open book and open notes. You may feel free to use whatever additional reference material you wish, but **no electronic aids** are allowed. Please note the following instructions. There will be a ten point deduction for failure to comply with them:

- start each problem on a new sheet of paper
- write your social security number, but not your name, on each sheet of paper you turn in
- show your work whenever appropriate. There can be no partial credit unless I see how answers were arrived
- be succinct. You may lose points for facts that, while true, are not relevant to the question at hand

You have until 12:20 to finish the exam. The questions are equally weighted. Two of the questions (3 and 5) include space for answers. You may answer these questions on this exam, but make sure you put your SSN on those pages when you turn them in!

1. In the original MIPS instruction set, both loads and branches were delayed: in the case of load instructions, the value loaded from memory was not guaranteed to be in the destination register when the following instruction was executed, so the value could not be used in that instruction (it also couldn't be guaranteed not to be in the register, so a programmer couldn't depend on the old value being there either). In the case of branches, the instruction following the branch instruction would be executed whether the branch was taken or not.

In later implementations of the processor, it was necessary to emulate one of these behaviors, but not the other. Which one had to be emulated, and which did not? Why (for the one) and why not (for the other)?

2. Assume you have an EPIC-style computer system, capable of (at least) the following instructions:

cmp.eq Ra, Rb, Pc, Pd compares registers *Ra* and *Rb*; if they are equal then predicate register *Pc* is set to 1 and *Pd* is set to 0, otherwise *Pc* is set to 0 and *Pd* is set to 1.

load.s Ra, Rb loads the memory location pointed to by *Rb* into register *Ra*. If there is an error, register *Ra*'s Not a Thing (NaT) bit is set.

chk Ra checks whether valid data is in register *Ra* (*ie* *Ra*'s NaT bit is not set); if NaT is set, a segmentation violation occurs.

limm Ra, const performs a load immediate; register *Ra* is set to *const*.

All instructions can be predicated by specifying the predicate register in parentheses before the instruction. A predicated instruction is only executed if the specified predicate register contains a 1.

Finally, any number of instructions can execute in parallel. All parallel instructions read their operands simultaneously, execute simultaneously, and write their results simultaneously. You can separate these groups of parallel instructions with two semicolons “; ;”.

Assuming *ptr* is in register *R1* and *x* is to be placed in register *R2*, translate the following C code into EPIC-style code, so that (1) the pointer is loaded as early as possible, and (2) the code executes in as few cycles as possible, subject to (1).

```
if (ptr != NULL)
    x = *ptr;
else
    x = 17;
```

3. Consider a direct-mapped cache combined with a two line, fully-associative victim cache using LRU replacement. The cache's total size is 16 bytes; the line size is 4 bytes (these numbers are artificially small to make the problem tractable). The cache and victim cache are initially empty. After the following sequence of memory accesses is performed, what are the contents of the cache and the victim cache? You may use the table including the accesses to show the contents of cache and victim cache after each access.

		Memory Accesses									
		0x3b	0xef	0xf0	0xb1	0xe9	0xae	0xdf	0xc0	0xdf	0xe6
Cache Line	Cache Contents										
00											
01											
10											
11											
		Victim Cache Contents									
MRU											
LRU											

4. Ordinarily, the Program Counter (PC) is a very special register - even in a computer that does register renaming it doesn't get renamed; its incrementer isn't one of the standard ALUs. Of course, there's no immediate reason why it couldn't be considered just like any of the other registers; it could be renamed to a physical register, and an extra μop could be created for every instruction which would increment the PC. Discuss the advantages and disadvantages of this idea. Would it take more or less hardware than the current scheme? Would it be likely to improve the speed of the machine, or slow it down? Would it make control easier, or more difficult?

5. Consider a computer with the following programmer-visible instruction set:

The computer has four architected registers named RA, RB, RC, and RD.

It uses a two-operand instruction set, so an instruction like

```
add RA, RB
```

adds the contents of register RA to the contents of register RB, and puts the results in register RA, while an instruction like

```
add RC, Memloc
```

adds the contents of register RC to the contents of memory location Memloc, and puts the results in register RC, and an instruction like

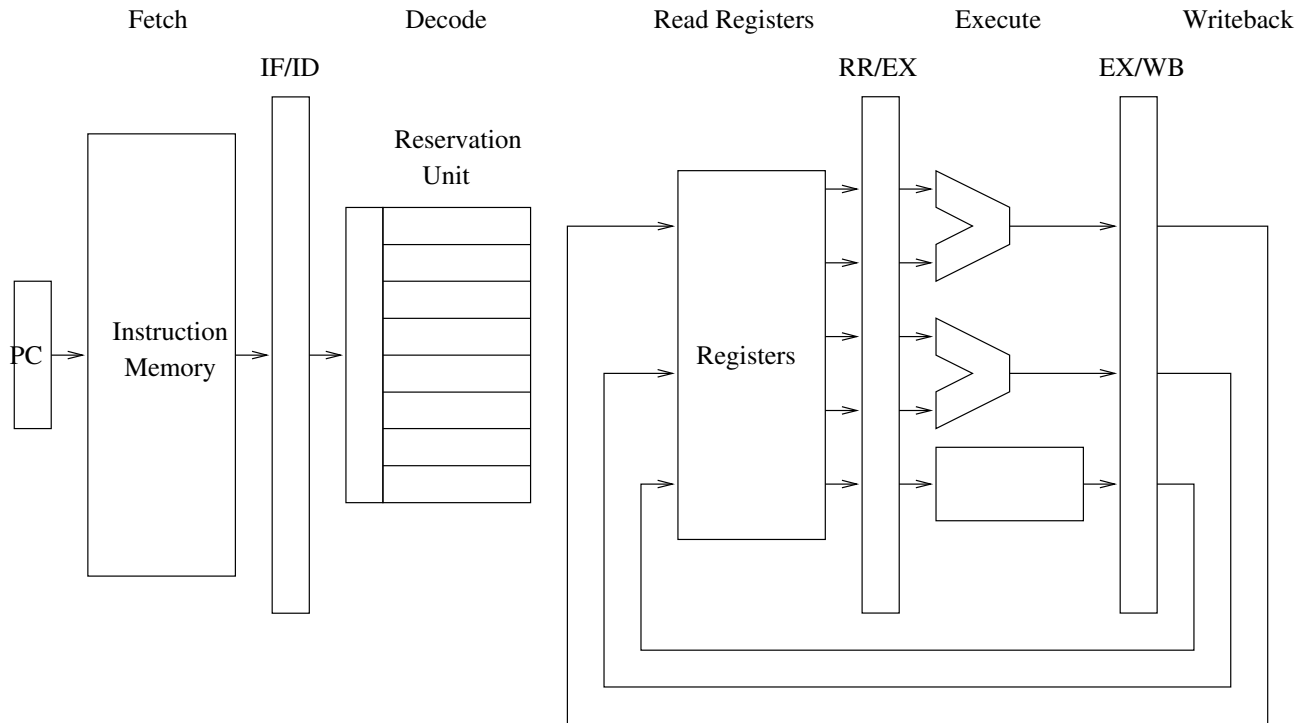
```
add Memloc, RC
```

adds the contents of memory location Memloc to the contents of register RD, and puts the result in memory location Memloc.

The implementation of this computer uses register renaming to rename the architected registers into a set of physical registers named R0-R7.

It has three μops : ld, st, and add, which have the expected meanings: ld loads a value from memory into a physical register, st stores a value from a physical register to memory, and add adds the contents of two physical registers together and puts the result into a third physical register (note that while the architected instruction set is not a load-store instruction set, the microarchitecture is).

Here's a picture of the computer's pipeline.



An instruction is executed as follows (every stage requires one cycle):

- On the first cycle (Fetch stage), the instruction is fetched from the instruction memory and placed in the IF/ID pipeline register. At most one instruction can be fetched per cycle.
- On the second cycle (Decode stage), the instruction is decoded into one, two, or three μ ops, physical registers are allocated for it, and the μ ops are stored in the reservation unit. Each of the μ ops now waits until its operands are ready.
- When a μ op can execute, it fetches its operands from the registers (Register stage). Up to three μ ops at a time can execute on a cycle; two of them have to be adds and one has to be a ld or st. The operands are stored in the RR/EX pipeline register. The μ op remains in the reservation unit until it is retired.
- The μ op is now executed by the appropriate functional unit (Execute stage).
- The μ op writes its results back to its physical register (Writeback stage).
- μ ops are retired in-order, and removed from the retirement unit at that time. An arbitrary number of μ ops can be retired on a cycle (Retirement stage, not shown in the figure); notice that this must come one cycle later than the register writeback.

Complete the following table. Use one row of the table for each μ op (there is room for three μ ops per instruction; since not all instructions decode to three μ ops, you will have some blank rows). In the timing section, mark each cycle in which an instruction or μ op is active. Each row in which an architected instruction is fetched should have exactly six marked cycles; where an architected instruction translates to more than one μ op, the succeeding rows should have exactly five marked cycles.

Instruction	μ ops	Timing													
add RA, Mem1															
add RB, RC															
add Mem2, RC															
add RB, RA															