

CS 573

Midterm Exam

Solutions

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)?

It wasn't necessary to emulate the delayed load, since its action was unpredictable anyway. A new machine with no delayed load would still work for correct code written for the old machine.

It was necessary to emulate the delayed branch, since it was possible to depend on its presence (and optimized code would have done so). So, if it were eliminated, correct code for old machines would break on the new ones unless the "feature" were emulated.

The most frequent error in this question was to not understand what was and what was not important. Quite a few answers were based on things like when a stall is necessary, and seemed to think that (especially in the case of the delayed load) that if any stall was necessary then exactly one cycle of stall was necessary. There's no reason to assume that, which is why the basic lesson of Alpha is that these issues should not affect your instruction set.

So what does? Backward compatibility. The delayed branch always happened, so one could expect compilers to have taken advantage of it (and they did), so it would have to be retained to keep from breaking old code. On the other hand, the delayed load didn't always happen, so there was no way for a compiler to take advantage of it (by doing things like starting a load and then using the old value in the target register in the next instruction) so it could be done away with.

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;
# First group
load.s R2, R1    # speculatively load *ptr into x
limm    R3, NULL # need NULL for compare
;;

# Second group
cmp.eq R1, R3, P1, P2 # check for null pointer
;;

# Third group
(P2) chk R2          # If pointer wasn't NULL, load is complete
(P1) limm R2, 17     # If it was NULL, 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.

With a four byte line and four lines in the sixteen-byte cache, the tag is four bits and the index is two bits. So we get:

		Memory Accesses								
		0x3b	0xef	0xf0	0xb1	0xe9	0xae	0xdf	0xc0	0xdf
Cache Line	Cache Contents									
00			f0-f3	b0-b3	b0-b3	b0-b3	b0-b3	c0-c3	c0-c3	c0-c3
01										e4-e7
10	38-3b	38-3b	38-3b	38-3b	e8-eb	e8-eb	e8-eb	e8-eb	e8-eb	e8-eb
11		ec-ef	ec-ef	ec-ef	ec-ef	ac-af	dc-df	dc-df	dc-df	dc-df
		Victim Cache Contents								
MRU				f0-f3	38-3b	ec-ef	ac-af	b0-b3	b0-b3	b0-b3
LRU					f0-f3	38-3b	ec-ef	ac-af	ac-af	ac-af

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 μ op 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?

I'd expect the implementation to be simpler – essentially, everything special about the PC could be stripped out of the CPU and replaced with an extra μ op, which could be leveraged using the existing decoder.

But I wouldn't expect it to be a good idea. This looks to me like a case where different access patterns call for different management strategies; the PC's access is extremely predictable, and each new PC value has a dependence on its old value (so except when branches are cancelled, PC updates really can't be handled out of order). So it looks like you'd be polluting the physical registers with many copies of something that only needs a single copy, and polluting the reservation stations with a lot of instructions that have to be handled in-order.

You'd also have to be very careful with the implementation: a naive implementation would be likely to end up with the PC only getting incremented once every several cycles!

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 μ ops: `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).

