Name Solution

Computer Architecture
EE 4720
Midterm Examination

Friday, 22 March 2002, 13:40–14:30 CST

Problem 1		(35 pts)
Problem 2		(25 pts)
Problem 3		(40 pts)
Exam Total		(100 pts)

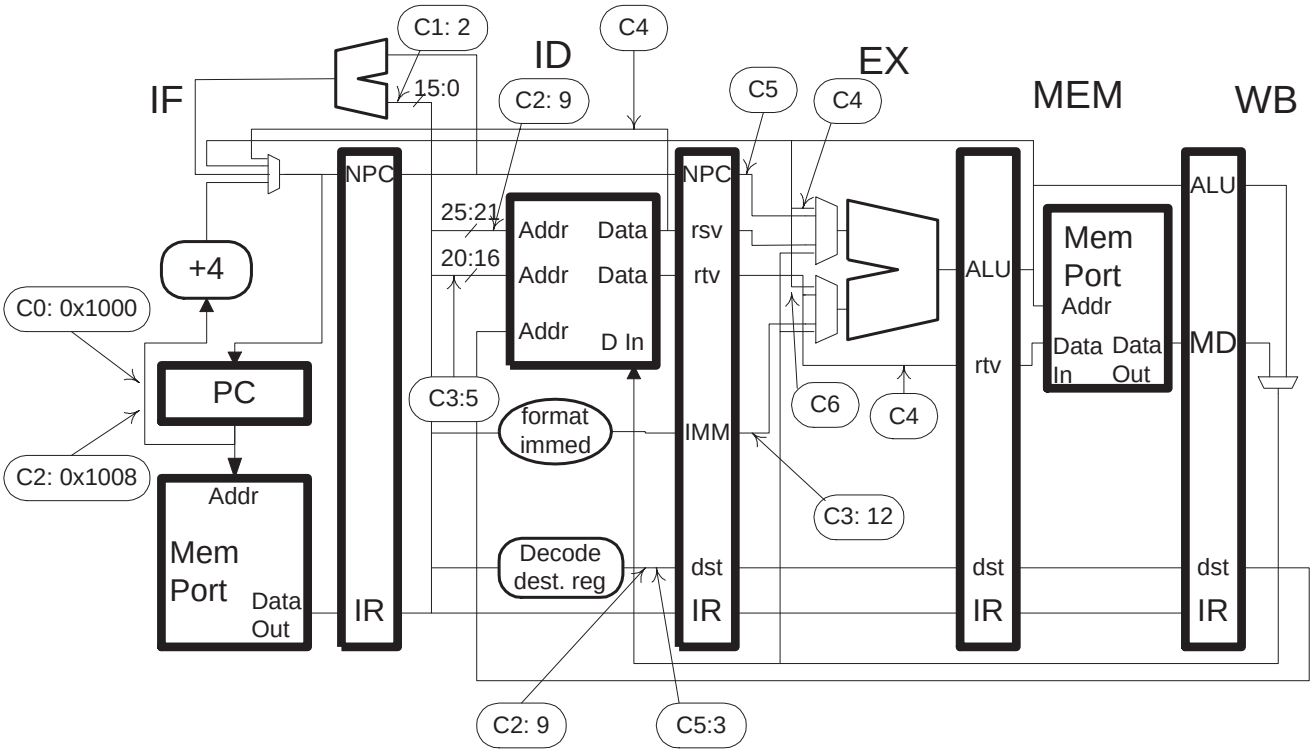
Alias ~~Vaxinated~~

Good Luck!

Problem 1: In the diagram below certain wires are labeled with cycle numbers and values that will then be present, for example, C2:9 indicates that at cycle 2 the wire will hold a 9. Other wires are labeled just with cycle numbers, indicating that the wire is used at that cycle. Write a program consistent with these labels. There are no stalls during the execution of the code. The code should use five instructions. *Note: The diagram in the original exam had an error that resulted in contradictory information about the third instruction (fetched in cycle 2). The error was a C3:5 pointing to the input to the* `ID/EX.dst` *latch; that now points to the rt address input to the register file.* [35 pts]

☑ Write a program consistent with these labels.

☑ Use labels for branch targets (if any) and label the target line.



Solution shown below. Upper-case letters in instruction mnemonics indicate parts that are fixed. For example, `Sw` indicates that the instruction must be some kind of a store instruction, but instead of `sw` it could be `sh` or `sb`. Register numbers below $10 are based on the labels above, registers $10 and above are arbitrary. For example, $3 in the `and` must be $3, but $14 could be $15.

```
 # Cycle            0    1    2    3    4    5    6    7    8    9
 Bne $10,$10 TARG   IF   ID   EX   ME   WB
 addI $9, $9, 12         IF   ID   EX   ME   WB
 Sw $11, 0($9)                IF   ID   EX   ME   WB
TARG:
 JALR $31, $12                     IF   ID   EX   ME   WB
 and $3, $14, $31                       IF   ID   EX   ME   WB
 # Cycle            0    1    2    3    4    5    6    7    8    9
```

Problem 2: Consider the CISCy instruction below: [25 pts]

(*a*)

```
add 0x123456(r1), 8(r2), r3   # 0x123456(r1) is the destination.
```

☑ What makes it CISCy? (CISCy means characteristic of CISC ISAs.)

It's an ALU instruction *and* it accesses memory, that's CISCy. Also, because of the large immediates that can be present, a reasonable coding would have variable-size instructions, another CISCyness.

☑ Ignoring instruction size, why would it be difficult to implement such an instruction on the kind of five-stage pipeline used in class for MIPS?

Lots of reasons. For one, because execution of the instruction includes memory access before the addition (to get the first operand) but in the five-stage pipeline memory is accessed after the execute stage.

Problem 2, continued:

```
add 0x123456(r1), 8(r2), r3   # 0x123456(r1) is the destination.
```

(*b*) How could the single-issue (not superscalar) pipeline used in class be modified so that instructions like the one above could be executed with a potential for a CPI of 1 without impacting clock frequency? (Instructions like the one above have one destination and two sources, the destination and first source can either use a register value or memory value specified with displacement addressing, the second source can be either a register value or an immediate.) Feel free to throw hardware at the problem, but do not assume that the hardware is any faster than what we've been using.

☑ Show a sketch of the pipeline, briefly explaining what should be done in each stage.

Here are the stages that would be needed. A correct solution might include a quick sketch.

IF: Same (pretend, because instructions are variable width).

ID: Same (pretend, because instructions are variable width).

EA: Compute effective addresses for destination and source operands (if needed).

M1: Load first source operand from memory (if needed).

EX: Same

M2: Store result in memory (if necessary).

WB: Store result in register file (if necessary). WB and M2 might be combined.

☑ Show a pipeline execution diagram for the following code on your new pipeline, assuming no dependencies.

Solution appears below.

```
# Solution

add 0x1234(r1), 8(r2), r3  IF ID EA M1 EX M2 WB

or r4, 7(r5), r6              IF ID EA M1 EX M2 WB

and 0x1234(r7), r8, r9           IF ID EA M1 EX M2 WB
```

☑ In the part above, why was it necessary to *assume* no dependencies?

First, about that word "assume." I might assume that you had breakfast this morning, but I never assume that I had breakfast because I know for sure whether I did or didn't. So the question above is asking why the code might still have dependencies alluding to the fact that there are no registers in common. The question does NOT ask "why was an example with no dependencies chosen for the problem."

Back to the question. The assumption was made because dependencies are still possible, through memory. The effective address of the destination of the `add` instruction might be the same as the first source operand of the `or` instruction. That is, the `add` writes the memory location that the `or` reads, a true dependency. (Such dependencies do not affect statically scheduled RISC processors because load and store instructions pass through the same memory stage in order.)

Problem 3: Answer each question below.

(*a*) What'll it be? One FP adder with an initiation interval of 2 and a latency of 3 (four cycles of computation) or two FP adders each with an initiation interval of 4 and a latency of 3? [10 pts]

☑ What is the maximum number of FP adds per second with each alternative on a 1 GHz system?

In either alternative a result can be produced every two cycles, at 1 GHz thats 500 million FP adds per second.

For some reason few answered this question correctly.

☑ Show a code fragment in which one of the alternatives is slightly faster. For the alternative with two adders, use label "A" for one adder and "B" for the other.

```
# Solution.

# On first alternative
add.d f0, f2, f4   IF ID A1 A1 A2 A2 WB
add.d f6, f8, f10     IF ID -> A1 A1 A2 A2 WB

# On second alternative
add.d f0, f2, f4   IF ID A1 A1 A1 A1 WB
add.d f6, f8, f10     IF ID B1 B1 B1 B1 WB
```

☑ Ignoring the cost of the adders themselves, which alternative would be more costly and why?

The second alternative, since the outputs of the two adders have to be multiplexed together.

(*b*) SPEC benchmark numbers are provided in "base" and "peak" forms. [10 pts]

☑ How are they different?

Programs compiled for the base numbers are prepared with normal (actually, a bit on the aggressive side) optimization. Programs compiled for the peak numbers are prepared with your-life-depends-on-it extreme optimizations.

☑ When should an intelligent (passed EE 4720) computer buyer use the base numbers, and when should the buyer use the peak numbers?

Use base numbers when you plan to use purchased software or when you can't put alot of effort in to tuning code. Use the peak numbers when compiling your own software and have the expertise and time to super-optimize, or if purchasing super-optimized software.

(*c*) Stack ISAs had burrowed their way in to our past and percolated to the fore in the past decade. [10 pts]

☑ Why are programs compiled for stack ISAs smaller than the same programs compiled for other ISAs? (Assume all compilers are of high quality.)

Many instructions in stack ISAs refer to the stack so they don't need register operands, and so they can be small, say one byte.

☑ Why would superscalar implementations of stack ISAs be less efficient (further from the ideal CPI) than superscalar implementations of RISC and some other "conventional" ISAs? Ignore instruction fetch problems. Consider the simple stack programs presented in class.

If the instructions in a fetch group in a statically scheduled superscalar processor (the only kind covered so far) are dependent some will stall. The stack organization forces dependent instructions to be near each other, guaranteeing lots of stalls in a superscalar implementation.

(*d*) At last superscalar implementations and VLIW ISAs will wage their epic [tm] battle now at the dawn of the twenty-first century.[10 pts]

☑ What distinguishes a superscalar implementation from a nonsuperscalar implementation? (VLIW isn't part of this question).

Superscalar processors can sustain execution of more than one instruction per cycle by duplicating fetch, decode, register read, and other parts of the system.

☑ Explain two ways in which VLIW machines overcome problems encountered in superscalar implementations of conventional (say RISC) ISAs?

The dependency information in VLIW bundles reduces the need for the dependency-checking hardware that is present in superscalar processors. Limiting branch targets to bundles reduces the inefficiencies due to un-aligned fetch groups.