Name Solution

Computer Architecture
EE 4720
Midterm Examination

Wednesday, 29 March 2006, 11:40–12:30 CST

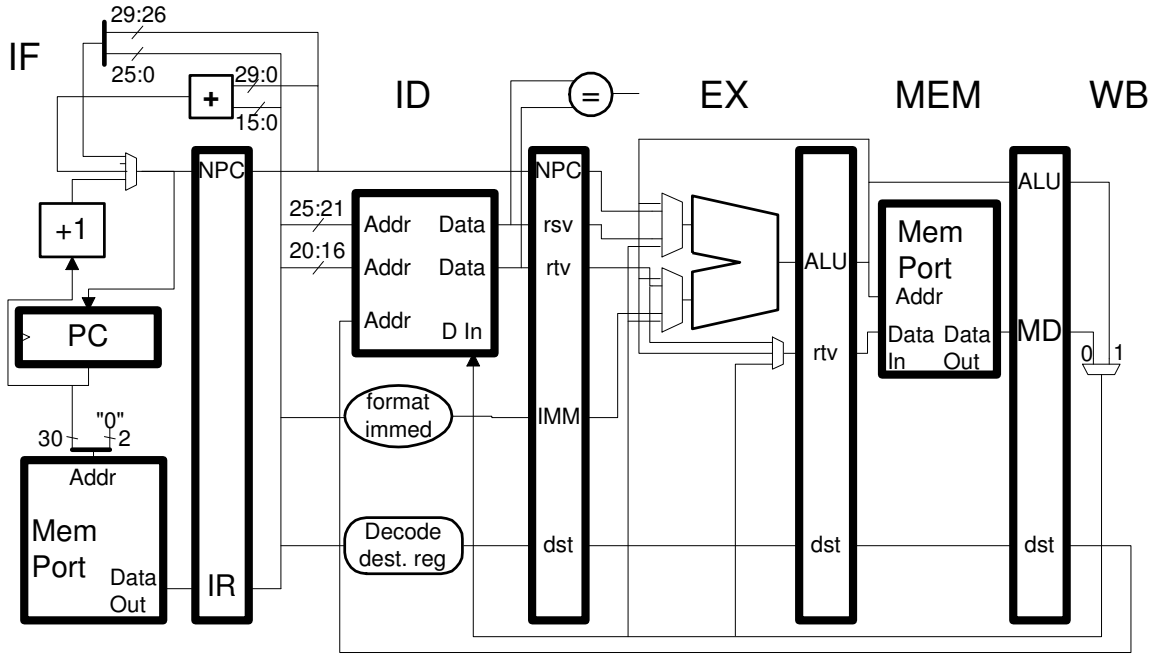
Problem 1		(10 pts)
Problem 2		(40 pts)
Problem 3		(50 pts)
Exam Total		(100 pts)

Alias Four Got In

Good Luck!

Problem 1: The MIPS code below runs on the illustrated implementation. [10 pts]

☑ Show the execution of the code below on the illustrated pipeline.




```
# Solution
# Cycle        0  1  2  3  4  5  6  7
lw r1, 0(r2)   IF ID EX ME WB
add r1, r1, 7     IF ID -> EX ME WB
sw r1, 0(r2)         IF -> ID EX ME WB
```

There is a stall in cycle 3 because the add must wait for the loaded value. (That value is bypassed in cycle 4.)

Note: To keep the test from getting too long the following parts were not included on the original exam. There should be at least one stall.

Think about the questions below for a few moments, then look at the problem on the next page.

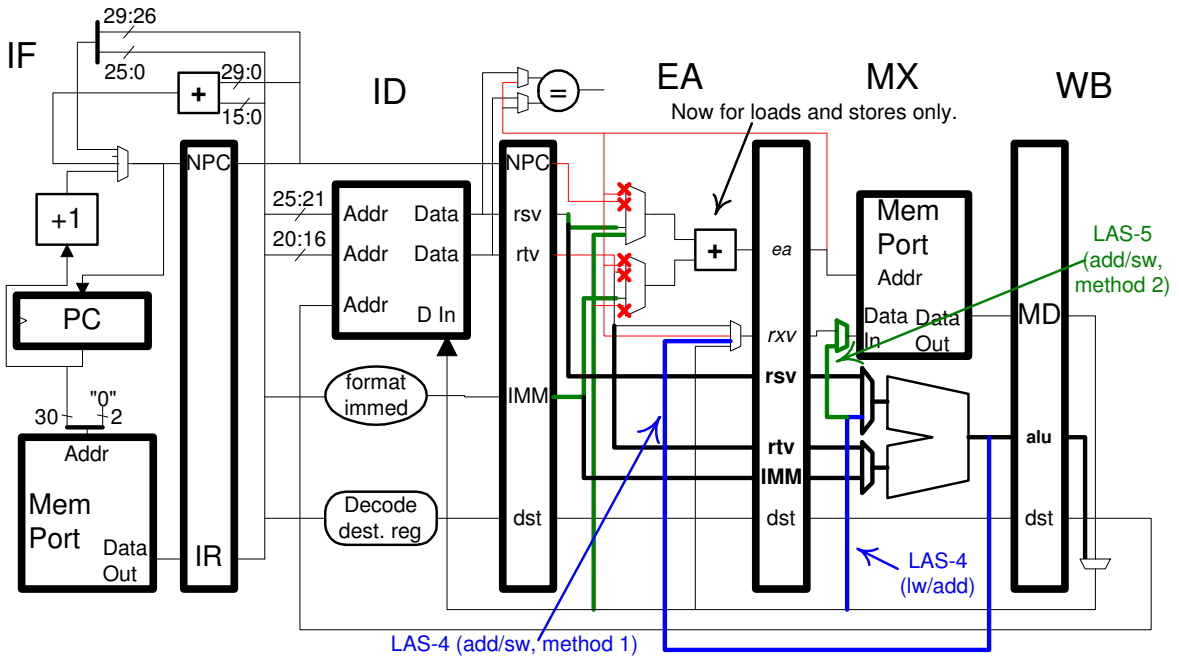
☑ Is it possible to add bypass connections to eliminate the stall and gain performance?

No, because without the stall the `add` would need the value of `r1` before the `lw` retrieved it from memory.

☑ If bypasses won't work does that mean it's impossible to eliminate the stall?

It's not impossible if the pipeline can be re-arranged. See the next problem.

Problem 2: The five-stage MIPS implementation below has an ALU in the MX (new name for MEM) stage (connections for that ALU are shown bold) and what was the ALU in the EA (new name for EX) stage is now just an adder and is to be used only for loads and stores.



(*a*) Add bypass connections needed so that the code below (same as last problem) executes as shown. Label those bypass connections: LAS-*c*, where *c* is the cycle number in which the bypass connection is used. Do not add unneeded bypass connections!

☑ [10 pts] Add bypass connections, label with LAS-*c*.

See diagram. One bypass is shown for the lw/addi dependency, two possible bypasses are shown for the addi/sw dependency; either would be correct.

```
# Cycle          0  1  2  3  4  5  6
lw r1,0(r2)      IF ID EA MX WB
addi r1, r1, 7      IF ID EA MX WB
sw r1, 0(r2)           IF ID EA MX WB
# Cycle          0  1  2  3  4  5  6
```

(*b*) Several connections to the EA-stage adder are now unneeded (but were needed when it was an ALU). (Don't add new connections here.)

☑ [10 pts] Label unneeded EA-adder connections with an X at mux inputs.

The X's are shown in red in the diagram. The EA adder is now only used to compute effective addresses. For MIPS that's computed by adding the rs register value to an immediate. Inputs for the rt value, bypassed or from the register file, are not needed. The value bypassed from the EA stage is never needed since it's not a register value. NPC is also never used to compute an address. The table below summarizes the reasons:

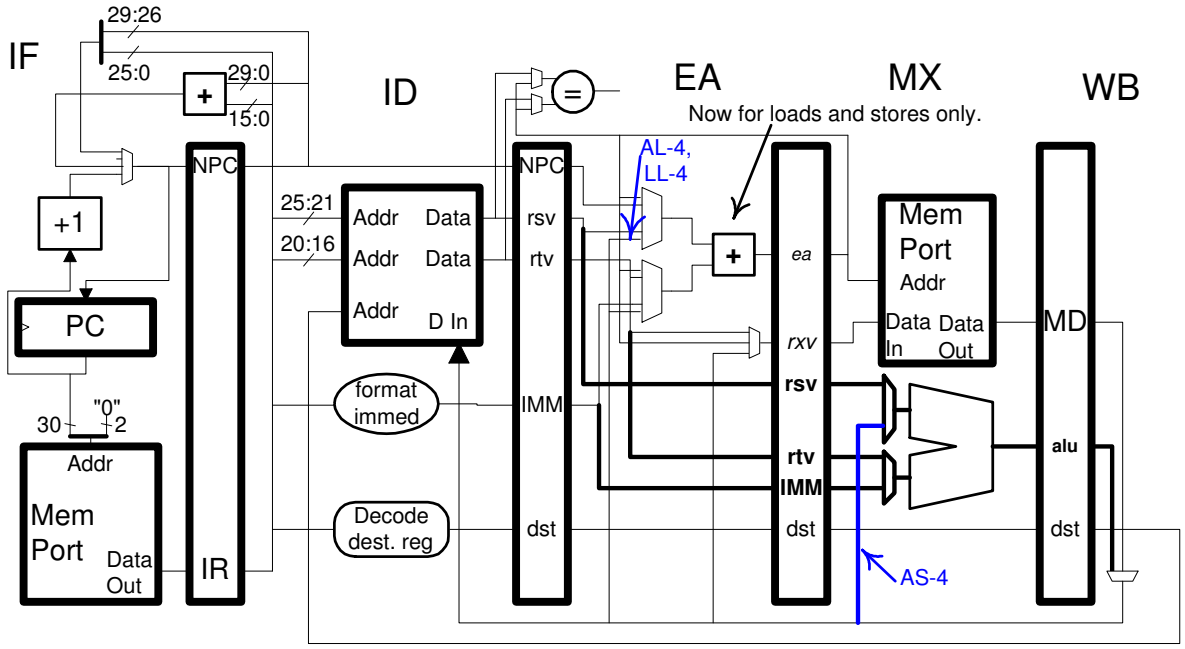
Upper Multiplexer:

```
0:X Bypass from EA is prior instruction's ea value. Never needed.
1:X NPC. Not used to compute loads or store addresses.
2:  rsv. Needed.
3:  Bypass from WB. Needed to bypass newer rs value.

Lower Multiplexer:

0:X Bypass. Addresses are never computed using an rt register value.
1:X rtv. Addresses are never computed using an rt register value.
2:  IMM. Needed.
3:X Bypass. Addresses are never computed using an rt register value.
```

Problem 2, continued: The implementation below is identical to the one on the previous page. But solution on diagram is different on the two pages.



(*c*) For each code fragment below: Add reasonable bypass connections to eliminate stalls (if any). Show the execution and label the bypass connections that are used (new or existing) with a cycle number. If there is a stall circle Yes if an EX-stage ALU (in addition to the ME stage ALU) would remove the stall, if an EX-stage ALU wouldn't help circle No; if there is no stall don't circle anything. [20 pts]

☑ Add bypasses (if needed). Label bypasses used AS-*c*. Show execution. EX ALU would help: Yes No.

```
# Cycle          0  1  2  3  4  5  6
add r1, r2, r3   IF ID EA MX WB
sub r4, r1, r5      IF ID EA MX WB
```

☑ Add bypasses (if needed). Label bypasses used AL-*c*. Show execution. EX ALU would help: [Yes] No.

```
# Cycle          0  1  2  3  4  5  6
add r1, r2, r3   IF ID EA MX WB
lw r4, 4(r1)        IF ID -> EA ME WB
```

☑ Add bypasses (if needed). Label bypasses used LL-*c*. Show execution. EX ALU would help: Yes [No].

```
# Cycle         0  1  2  3  4  5  6
lw r1, 12(r2)   IF ID EA MX WB
lw r3, 16(r1)      IF ID -> EA MX WB
```

☑ Add bypasses (if needed). Label bypasses used AB-*c*. Show execution. EX ALU would help: [Yes] No.

```
# Cycle          0  1  2  3  4  5  6  7
addi r1, r1, 1   IF ID EA MX WB
beq r1,r2 TARG      IF ID ----> EA MX WB
nop
```

Problem 3: Answer each question below.

(*a*) A computer's scores on SPECint2000 are baseline, 2011; and result (peak) 2014. These baseline and result scores are close, who might be responsible and should they be proud or ashamed: [10 pts]

☑ How might the people who conducted the test be responsible for the close scores? Should they be proud or ashamed?

Rather than trying every combination of compiler switch they could, they just tried a few (because it was a sunny Friday afternoon) and so did not get as much performance as they could have. They should be ashamed.

Detailed explanation: The people that conduct the test compile and run the code. They are free to set compiler switches (and make other build choices) as long as they comply with SPEC's rules for SPECcpu2000.

If they are responsible for the close scores then it must be because of the way they set the compiler switches. For the base tests reasonable optimization options should be set, but for peak (result) scores much more optimization can be done. For the scores to be close the compiler options for the peak case were not set as well as they could be.

It would not be correct to say that the scores are close because the base scores are already close to the highest performance possible. If that were true then the testers would not be responsible, which would be avoiding the question.

☑ How might the people who wrote the compiler be responsible for the close scores? Should they be proud or ashamed?

They wrote a compiler that is very good at choosing optimizations: Just provide the `-fast` switch and the compiler will make the right choices. A human could not improve on things by specifying that this optimization should not be performed, or that optimization should be. They should be proud because they save programmers time and make the hardware and programmers look good.

Another possible answer is that there are only a few optimization choices and that there is nothing to do beyond base optimization. Specialized optimizations that other compilers performed have been omitted. In this case the compiler writers should be ashamed.

(*b*) On a SPARC system the handler for trap number 4 is located at address `0xa0001000` and is 100 instructions long. [10 pts]

☑ Show the Trap Base Register (TBR) value and Trap Table contents needed so that execution of a trap instruction will reach this handler. (Don't worry about returning.)

The trap table can be placed anywhere in system memory (as long as the address is a multiple of 16); the TBR holds that address. Picking an address above 2^{31} and a multiple of 16: `TBR = 0xe12340000`. The entry for trap 4 will be at address TBR $+ 4 \times 16 =$ `0xe12340040`. The contents of that entry is a call to the handler itself: `call 0xa0001000`.

☑ Why isn't a user program allowed to call the handler directly, for example, using the ordinary call instruction `call 0xa0001000`. (The SPARC call has a 30-bit immediate field so the target address is not too far away.)

The user program might try to call something other than the OS's "official" handlers or it might try to jump into the middle to avoid being denied access to something.

☑ What would happen if the user tried to execute the call instruction above?

The calling instruction would raise an exception. The exception handler would probably kill the program.

Problem 3, continued:

(*c*) Packed integer and BCD data types are very superficially similar. [10 pts]

☑ Describe a situation in which a packed integer data type would be useful.

Two long arrays of smaller numbers need to be added together. If each array element is an integer in the range 0-255 and eight values are packed into a 64-bit register then one packed add instruction would add eight pairs together whereas in conventional code one add instruction would add just one pair together.

☑ Describe a situation in which BCD would be useful.

When there is frequent need to show the decimal representation of a number and conversion is too slow (say, an old fashioned computer).

Rounding error is to be avoided and well behaved floating-point formats are not available or not trusted.

(*d*) Describe the contents of a typical bundle in a VLIW ISA. [10 pts]

☑ Bundle contents.

Several instructions (three is common), information on dependencies between the instructions within the bundle and between bundles, and possibly other information about the instructions.

(*e*) Which is it more important to do a good job on, the ISA or the implementation? Explain. [10 pts]

☑ Good job on ISA or implementation? Explain.

The ISA, because it's long-lasting and so if a mistake is made the only options are to live with it or to discard the ISA and all of its implementations (risking loss of customers).