Name Solution

Computer Architecture

EE 4720

Midterm Examination

Wednesday, 30 March 2016, 9:30–10:20 CDT

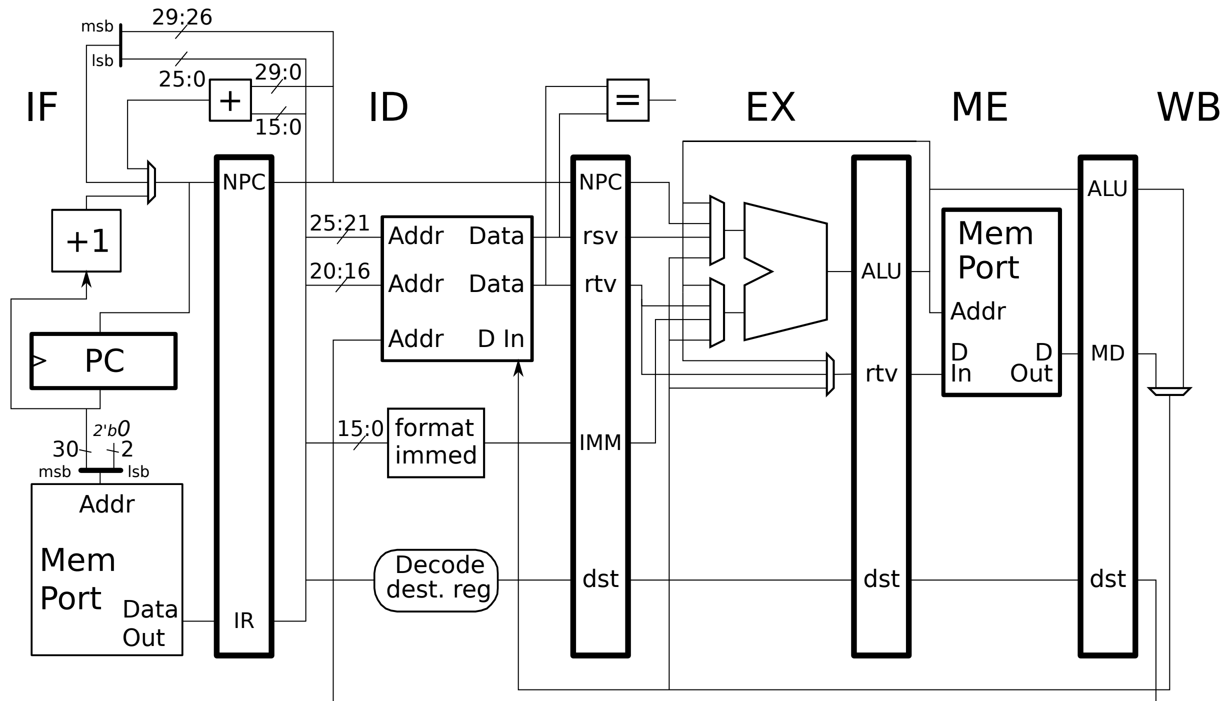
Problem 1	______	(25 pts)
Problem 2	______	(25 pts)
Problem 3	______	(12 pts)
Problem 4	______	(28 pts)
Problem 5	______	(10 pts)
Exam Total	______	(100 pts)

Alias Apple/Maté?

Good Luck!

Problem 1: [25 pts] Appearing below are what are supposed to be pipeline execution diagrams (PEDs) of code fragments executing on the illustrated implementation. The PEDs are incorrect.

(*a*) Correct the PEDs.



☑ Correct the PED below.

```
add r1, r2, r3  IF ID EX ME WB
lw r3, 0(r1)       IF ID -> EX ME WB

# SOLUTION
add r1, r2, r3  IF ID EX ME WB
lw r3, 0(r1)       IF ID EX ME WB
```

Solution appears above. Note that the dependency can be bypassed and so there is no need to stall.

☑ Correct the PED below.

```
lw r3, 0(r1)    IF ID EX ME WB
add r4, r3, r5     IF ID -> EX ME WB
sub r6, r7, r8        IF ID EX ME WB

# SOLUTION
# Cycle         0  1  2  3  4  5  6  7
lw r3, 0(r1)    IF ID EX ME WB
add r4, r3, r5     IF ID -> EX ME WB
sub r6, r7, r8        IF -> ID EX ME WB
```

The `sub` has to stall in cycle 3 since `ID` is occupied by the `add`.

☑ Correct the PED below.

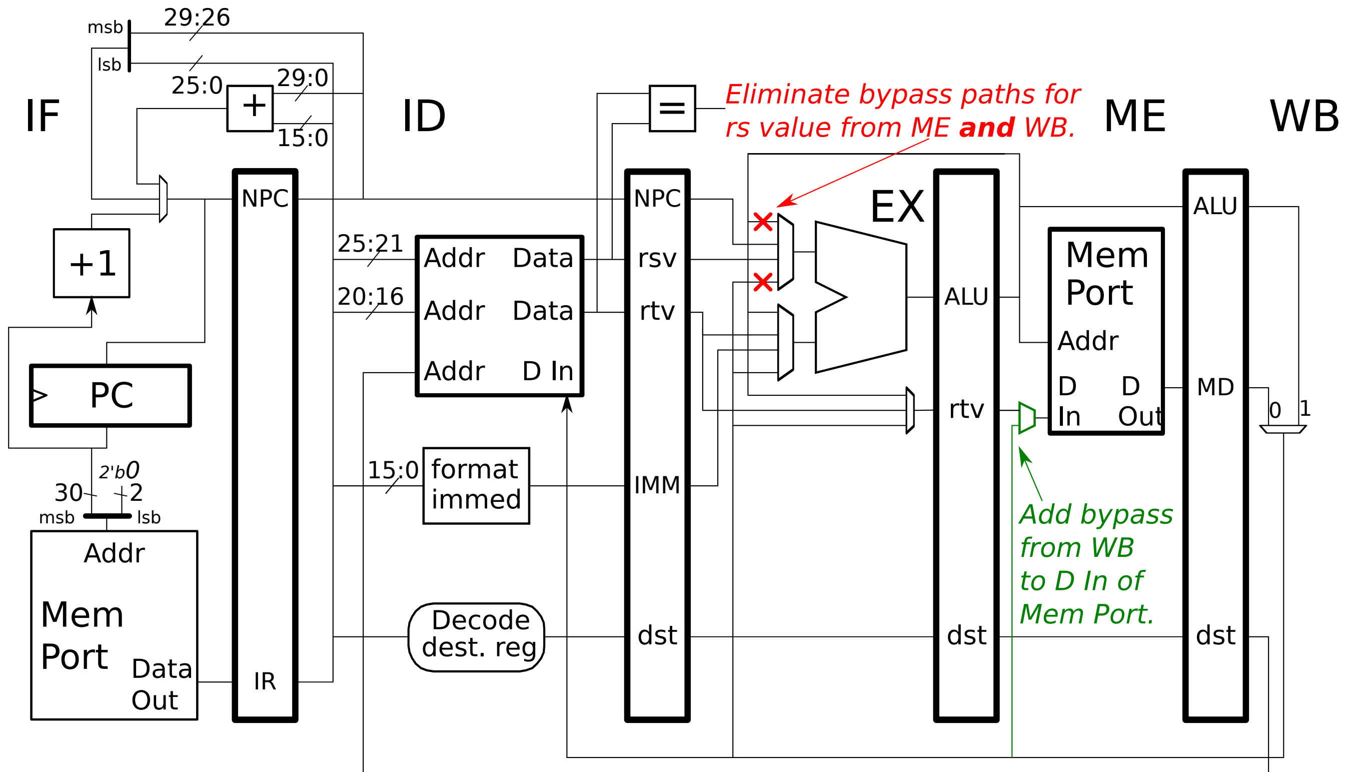
```
# Cycle           0  1  2  3  4  5  6  7
 beq r1, r1  TARG IF ID EX ME WB   # Branch is taken.
 xor r5, r6, r7      IF IDx
 add r8, r9, r10        IFx
TARG:
 sub r2, r3, r4            IF ID EX ME WB
# Cycle           0  1  2  3  4  5  6  7

 # SOLUTION
 # Cycle          0  1  2  3  4  5  6  7
 beq r1, r1  TARG IF ID EX ME WB   # Branch is taken.
 xor r5, r6, r7      IF ID EX ME WB
 add r8, r9, r10
TARG:
 sub r2, r3, r4         IF ID EX ME WB
# Cycle           0  1  2  3  4  5  6  7
```

The delay-slot instruction should be executed, and the branch is resolved in ID so the target must be fetched when the branch is in EX.

(*b*) Appearing below are more PEDs which are not correct for the illustrated implementation. This time **modify the implementation** so that the executions are correct. Only make necessary changes.

- Delete a bypass path by showing an × at the **mux input** where it ends.
- Do not delete or add more hardware than is necessary.



☑ Modify the implementation so that the execution below is correct.

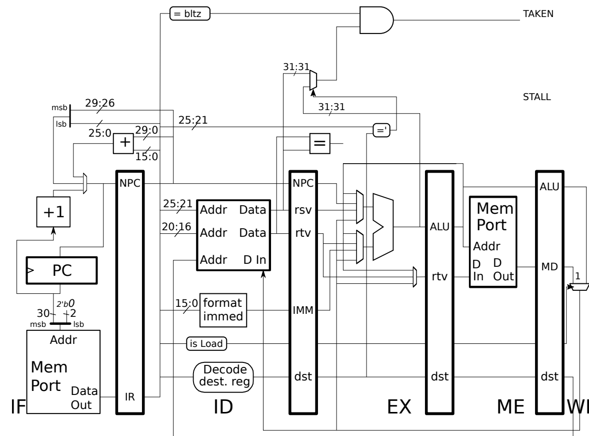
```
add r1, r2, r3   IF ID EX ME WB
sub r3, r1, r5      IF ID ----> EX ME WB
```

☑ Modify the implementation so that the execution below is correct.

```
lw r1, 0(r2)   IF ID EX ME WB
sw r1, 0(r3)      IF ID EX ME WB
```

Problem 2: [25 pts] The implementation below is based on the solution to Homework 2 Problem 2 in which a bypass was added for `bltz` instructions.

Use Next Page for Solution



Use Next Page for Solution

(*a*) The implementation can only bypass values in `EX` to a `bltz`. Modify the implementation on the next page so that values can be bypassed from both `EX` and `ME`. With these changes the two fragments below should run without a stall and of course bypass the correct value.

```
# Example 1
add r1, r2, r3
sub r4, r5, r6
bltz r1, TARG

# Example 2.
add r1, r2, r3
sub r1, r1, r6
bltz r1, TARG
```

(*b*) A bypass from EX isn't possible for the code fragment below, and a bypass from `ME` is problematic too. On the next page add logic to generate a stall signal for these situations (load/`bltz` dependencies) and connect it to the word `STALL` in the upper-right of the diagram. Notice that there is an [is Load] logic block in `ID`.

```
lw r1, 0(r2)
bltz r1, TARG
```

(*c*) Explain why it would not be a good idea to bypass the load value to the `bltz` when the load is in `ME`.

☑ Bypassing load from `ME` not a good idea because:

Because data is available at the `D Out` output of the memory port very late in the clock cycle and so it can't be used for anything without increasing the clock period. That is, `D Out` is on the critical path.

Problem 2, continued:

☑ Modify implementation so `bltz` can bypass from `EX` and `ME`.

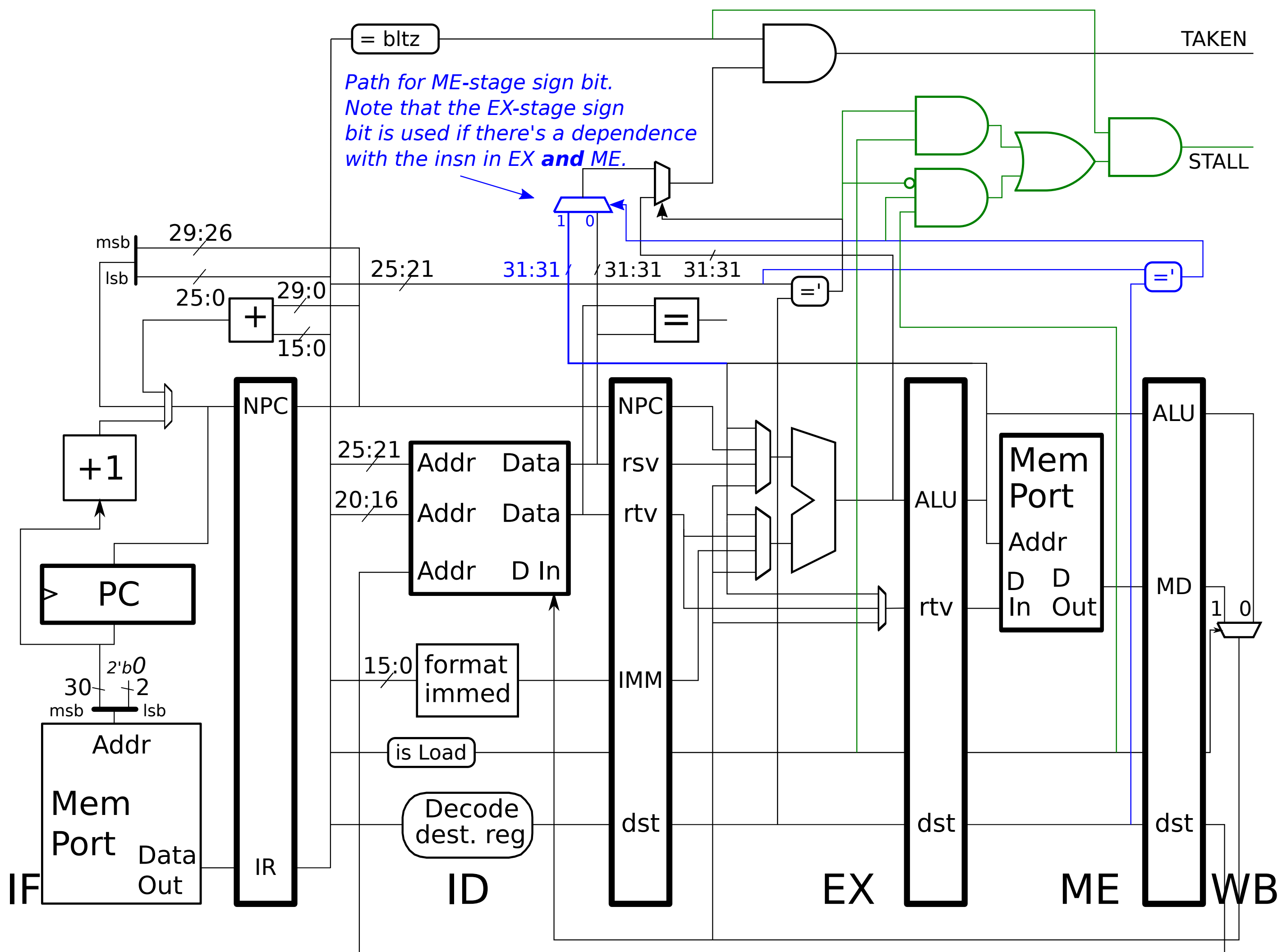
The solution appears below in blue. Note that the multiplexor is positioned such that if there is a dependence both on the instruction in `EX` and `ME`, the sign bit from the `EX` stage will be used. See Example 2 on the previous page.

☑ Logic to generate stall signal for `bltz` dependent on load.

Solution appears in green. Note that the stall is generated if the respective instructions are in the respective stages and if there's a dependence. A common mistake was to also check whether the "sign bit" was `1` (whether the branch is taken). That doesn't make sense because the sign bit isn't really available, if it was we wouldn't need to stall.

☑ Answer part c.

I remembered, but thanks for the reminder.



Problem 3: [12 pts] Answer each question below.

(*a*) Each code fragment below writes register `f30` with the sum `f2 + 4720`.

```
# Plan A
        addi $t0, $0, 4720
        mtc1 $t0, $f17
        cvt.s.w $f16, $f17
        add.s $f30, $f2, $f16


# Plan B
        lui $t0, 0x4593
        ori $t0, $t0, 0x8000
        mtc1 $t0, $f16
        add.s $f30, $f2, $f16
```

☑ What is the difference between `mtc1` and `cvt`?

The `mtc1` instruction moves a value from an integer register to a floating-point (co-processor 1) register. The `cvt` instruction converts a value from one format to another, in the example above from integer to single-precision floating point. So the difference is, `mtc1` moves its operand from an integer to FP register, while `cvt` changes the value of its operand from an integer to FP value.

☑ Why doesn't Plan B need a `cvt`?

Because the register contents is already in a FP format.

(*b*) All MIPS integer instructions have their source register numbers in the `rs` and, if needed, `rt` fields. But the destination register number can be found in either the `rt` or `rd` fields.

☑ How does limiting integer sources to `rs` and `rt` reduce cost and improve performance?

It reduces cost because no mux is needed at the register file `Addr` inputs. It improves performance for the same reason, there is no delay that there would be with a mux, which includes the logic generating the mux's control signal.

☑ Why isn't performance hurt by having the destination in either `rt` or `rd`?

Because the `dst` value is not needed until the end of `ID` (for the pipeline latch) and so the mux and its control logic are not on the critical path.

Problem 4: [28 pts] Answer each question below.

(a) The statement below omits an important reason why customers can be kept by companies that manage an ISA and implementation as two different things.

> *By separating the ISA from the implementation we can keep our customers by offering them a faster implementation when they are ready to buy a new system.*

☑ What is the important reason that has been omitted?

Short Answer: . . ., that runs the software **that they already have**.

More details: Software compatibility. The newer, faster computer must be an implementation of the **same** ISA or a superset of it.

(*b*) To use profiling to improve performance a program is compiled twice.

☑ What is done between the first and second compilation?

The program is run using typical input data, the run is called a *training run*. (Sorry about using the word *run* three times in one sentence.)

☑ Why does the program need to be compiled a second time?

So that the compiler can read the results of the training run and use that to make better optimization decisions.

☑ Suppose that taken branches have a penalty. Show how profiling helps.

Suppose that in the first code fragment below the branch is mostly taken, meaning that the ELSEPART is frequently executed. When the compiler learns this by reading the output of the training run it will rearrange code so that the branch is mostly not taken. It will also move the less-frequently executed IF_PART out of the way. In the optimized code zero control transfers are needed for the frequent case.

```
 # SOLUTION

 # Before optimization. Either branch or jump always taken.
 # Assume that branch is mostly taken.
 add r1, r2, r3
 beq r4, r5, ELSEPART
 xor r6, r7, r8
IF_PART:
 lw r8, 0(r9)
 ...
 j ENDIF
 add r10, r11, r12
ELSEPART:
 lw r8, 0(r20)
 ...
ENDIF:
 sw r21, 0(r22)

 # After optimization. Now branch is mostly not taken.
 add r1, r2, r3
 bne r4, r5, IFPART
 xor r6, r7, r8
ELSEPART:
 lw r8, 0(r20)
 ...
ENDIF:
 sw r21, 0(r22)
 ...

IF_PART:
 lw r8, 0(r9)
 ...
 j ENDIF
 add r10, r11, r12
```

Problem 4, continued:

(*c*) Consider an instruction such as `add (r1), r2, 4(r3)`. What about it makes it unsuitable for a RISC ISA? Explain why it would be difficult to implement in our pipelined design.

☑ `add (r1), r2, 4(r3)` unsuitable for RISC because:

☑ It would be difficult to implement because:

The instruction is unsuitable for RISC because it both performs arithmetic and accesses memory, a RISC no-no. This makes it hard to implement in a pipelined microarchitecture because the instruction would need to access memory twice, once for the source, `4(r3)` and once to write the result, `(r1)`, which would require extra memory ports, and that's costly, or it would require some way of using the memory port from different stages, which would greatly complicate the design.

(*d*) When we compared the un-optimized and optimized versions of the π program we found that the optimized version had many fewer load and store instructions. Why?

☑ The optimized π program had fewer loads and stores because:

Without optimization, the compiler will emit code to load the value of a variable into a register each time it is used and to write the value to memory each time it is changed. But if a variable is updated and used many times (say, in a loop body) then the value can be loaded just once, into a register, before the loop and stored just once, after the loop. That was the case with the π program where all loads and stores were eliminated from the loop body.

(*e*) A tester preparing a run of the SPECcpu suite is responsible for compiling the benchmarks. Why does that make SPECcpu results interesting to computer engineers?

☑ Tester compilation makes SPECcpu interesting to computer engineers because:

Computer engineers can evaluate the performance of new microarchitectural features, such as bypass paths, by running the SPECcpu benchmarks with a compiler back-end designed to take advantage of the new features. They also can run the benchmarks on new ISAs. Note that the tester is the computer engineer.

Problem 5: [10 pts] Answer the following questions about bypass paths.

(*a*) Consider the two statements below about bypasses in implementations like our five-stage MIPS **running typical programs**.

A: Compiler scheduling makes bypass paths unnecessary.

☑ Explain why the statement above is wrong.

In typical programs the compiler cannot always schedule instructions to avoid stalls because it can't always find enough instructions to put between a dependent pair. The problem is exacerbated without bypass paths because dependent pairs must be farther apart.

B: Bypass paths make compiler scheduling unnecessary.

☑ Explain why the statement above is wrong.

Bypass paths don't exist for every possible dependence, such as a `lw` followed by an `add`.

(*b*) Consider the two above statements (about bypass paths) again as it applies to our MIPS implementation, but this time **running a special set of programs**. We plan to design an implementation for this set of programs. For these programs the two statements are true! *Note: The original exam did not mention the new implementation, and it had an "or both" option below.*

☑ For such programs should we eliminate bypass paths or should we eliminate compiler scheduling?

☑ Explain.

Bypass paths, because that would save money. Note that just because both statements are true does not mean that bypass paths and compiler scheduling can be eliminated at the same time. Statement A implies that compiler scheduling is eliminating stalls without the help of bypass paths. But if scheduling is also eliminated there can be stalls.