Name Solution

Computer Architecture
EE 4720
Midterm Examination

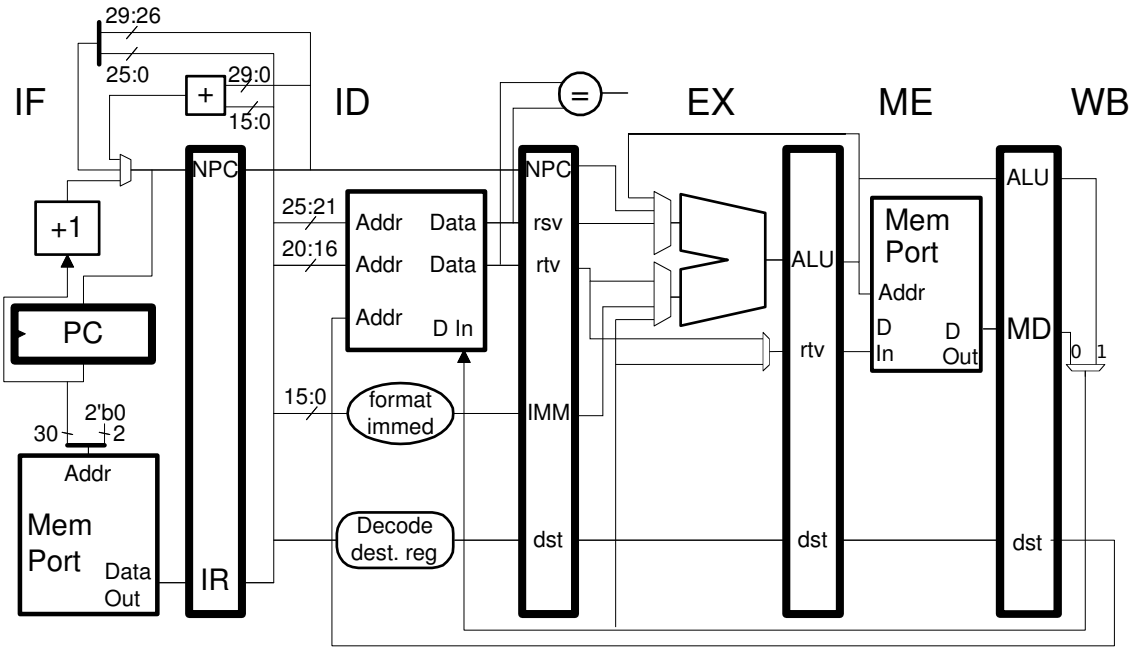
Friday, 23 March 2012, 9:40–10:30 CDT

Problem 1		(20 pts)
Problem 2		(15 pts)
Problem 3		(15 pts)
Problem 4		(20 pts)
Problem 5		(15 pts)
Problem 6		(15 pts)
Exam Total		(100 pts)

Alias A Century of Turing

Good Luck!

Problem 1: [20 pts] The MIPS code below executes on the illustrated **seemingly** familiar implementation. If you look closely you'll notice that certain bypass paths that are present in the five-stage implementation usually used in class are missing from the diagram below.



(*a*) Show the execution of the code below on the illustrated implementation for enough iterations to determine CPI. Determine the CPI.

☑ Execution diagram.

☑ Double-check for bypass paths, stall when necessary.

☑ Compute CPI.

The pipeline execution diagram appears below.

Note the following close dependencies: `lw/add` through `r2`, `add/sw` through `r1`, and `addi/lw` (in the 2nd iteration) through `r4`.

The `add` stalls two cycles (2 and 3) due to the `lw/add` dependence, so that when the `add` can read the value of `r2` when it is in `ID` (bypassed through the register file). In the bypassed implementation used in class the `add` would only stall one cycle, since the value of `r2` could be bypassed from `WB` to `EX`, but in the implementation above there is no bypass from the `WB` stage to the upper input to the ALU, where the `rs` value is needed.

Due to the `add/sw` dependence the `sw` stalls one cycle (5), because the `ME`-to-`EX` bypass to the store value multiplexor (the one connecting to `EX/ME.rtv`) is missing. With that one-cycle stall the `add` is in `WB` while the `sw` is in `EX`, where the `WB`-to-`EX` bypass to the store value multiplexor can be used. In the bypassed implementation used in the class notes there would be no stall here.

The `addi/lw` dependence does not cause a stall because there is a bypass from `ME` to `EX` going to the upper ALU mux.

Common Mistakes: Often beginners forget that the store value, `r1` in this case, is a source value. In this case it doesn't matter, but another common mistake is forgetting that the branch operands are sources, and that these sources cannot be bypassed to in most of the implementations used in class. People in a hurry sometimes mistake the first branch operand as an output.

To compute the CPI, divide the iteration time by the number of instructions. The first iteration starts at cycle 0 (the iteration starts when the first instruction of the loop body is in `IF`), the second at cycle 8, and so the first iteration takes 8 cycles. Since the first

and second iterations will have the same stalls we can conclude that all subsequent iterations will take 8 cycles. An iteration has five instructions, so the CPI is $\frac{8}{5} = 1.6\,\mathrm{CPI}$.

```
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
 lw  r2, 0(r4)     IF ID EX ME WB
 add r1, r2, r3       IF ID ----> EX ME WB
 sw r1, 4(r4)            IF ----> ID -> EX ME WB
 bne r4, r5  LOOP                IF -> ID EX ME WB
 addi r4, r4, 8                        IF ID EX ME WB
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
 lw  r2, 0(r4)                            IF ID EX ME WB
```

Problem 1, continued:

(*b*) The code above should have encountered at least one of the missing bypass paths. Choose one of those missing bypass paths and design control logic for it. The `ID`-stage control logic should generate a 1-bit signal `STALL` that will be logic `1` when the instruction in `ID` will have to stall because of the missing bypass path.

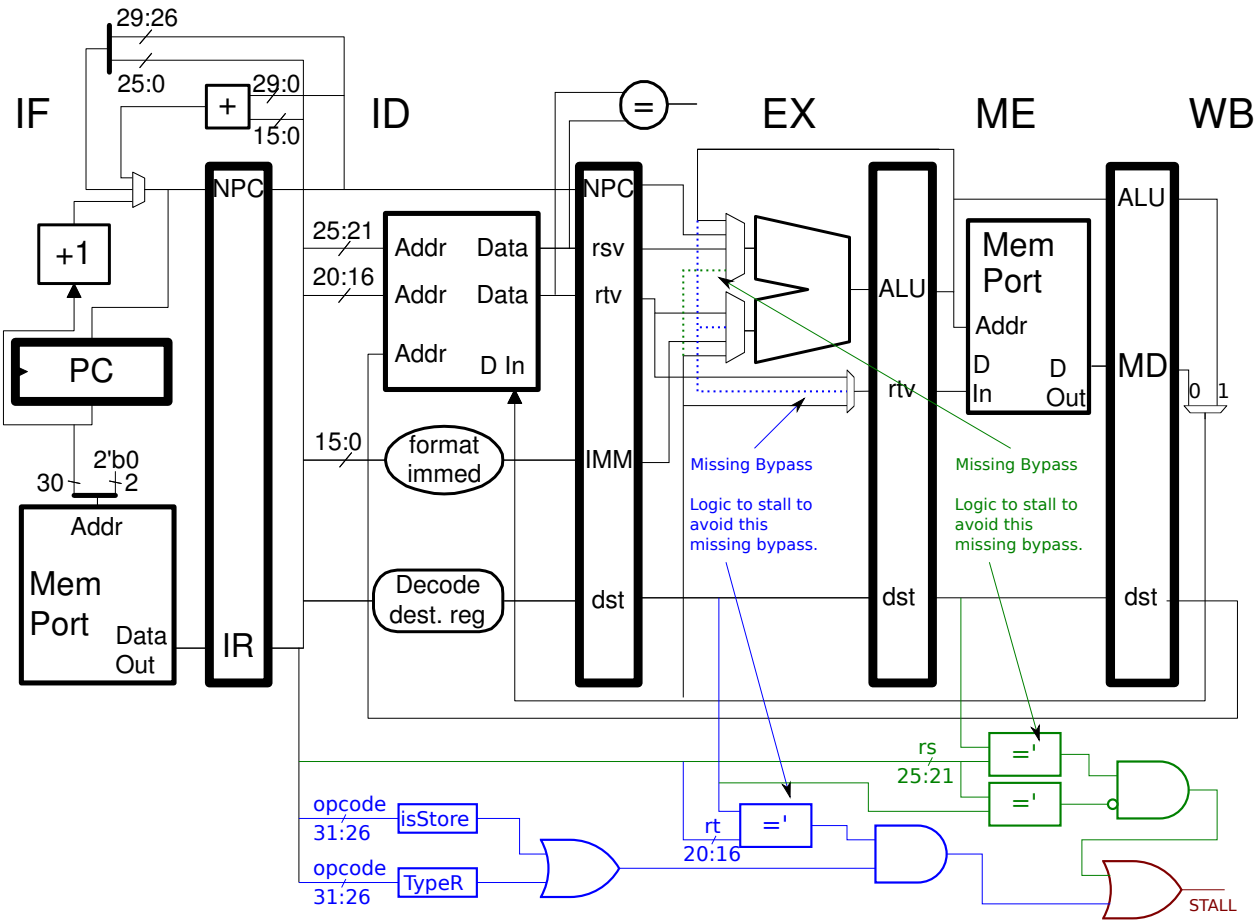
☑ Show which missing bypass path the control logic is for.

☑ Control logic for missing bypass.

Solution appears below, and includes logic for both bypass paths. The logic for the `lw/add` dependence appears in green, and logic for the `add/sw` dependence appears in blue. The stall signal appears in maroon.

The logic for the `lw/add` dependence will stall for any instruction that needs a value bypassed from the `WB` stage to the upper ALU input. For the `lw/add` case, this logic will generate the stall signal in cycle 3, but not in cycle 2. We assume that some other logic generates the cycle-2 stall. This logic assumes that every instruction uses an `rs` source (and most do).

The logic for the `add/sw` dependence will stall for any instruction that needs a value bypassed from the `ME` stage for the lower ALU input. The logic detects an instruction that uses the `rs` value as a source, assuming any Type R instruction and store instructions. The stall is generated if such an instruction is present in `ID` and if the instruction in `EX` writes the same register as the `ID` instruction's `rt` source.



Problem 2: [15 pts] Consider a MIPS implementation in which the memory port is split into two stages, `Ma` and `Mb`. With this change the clock frequency can increase from 1 GHz to 1.2 GHz; with this added stage our implementation is now six stages. A sample execution appears below.

```
add r2, r3, r4 IF ID EX Ma Mb WB
lw r1, 4(r2)      IF ID EX Ma Mb WB
```

(*a*) Ignoring the cost of `Ma` and `Mb`, how is the cost of the pipeline changed in this design? Consider bypass paths.

☑ Cost change, ignore `Ma` and `Mb`, consider bypass, etc.

There is another pipeline latch, which includes an **ALU** register (32 bits) and a **dst** register, 5 bits, plus control bits. The **ALU** value needs to be bypassed to the **EX** stage, the cost of that is 32 bits times three mux inputs (the two at the ALU inputs, and the one for the store value).

(*b*) Provide an example of a code fragment which will execute more slowly with the six-stage pipeline. Assume that all reasonable bypass paths are provided.

☑ Code fragment that will run more slowly.

The code fragment below runs more slowly, that's because of the dependence of the **add** on the **lw**, called a *load/use* pair.

```
# SOLUTION
# Cycle           0  1  2  3  4  5  6  7  8
lw r1, 0(r2)      IF ID EX Ma Mb WB
add r3, r1, r3       IF ID ----> EX Ma Mb WB
```

(*c*) Provide an argument that this change is good.

☑ This change is good because:

The change will add one stall cycle to a load/use pair, as in the example above. Suppose 1 out of every 3 instructions was a load followed by an instruction that uses the loaded value, also suppose that there are no other stalls. The ideal execution time for 300 million instructions is 300 million cycles. If 1 out of 3 is a load/use, then there are 100 million load/uses which will add 100 million stall cycles on the five-stage implementation and 200 million stall cycles on the six-stage implementation. Then the execution time for 300 million instructions would increase from 400 million cycles $400 \times 10^6 \frac{1}{1\,\mathrm{GHz}} = 400\,\mathrm{ms}$ to 500 million cycles which is $500 \times 10^6 \frac{1}{1.2\,\mathrm{GHz}} = 416\,\mathrm{ms}$, which is slower.

But, it's likely that far fewer than 1 out of 3 instructions will be the annoying load/use case, suppose it is 1 out of 10. Then with the ordinary **ME** stage execution is $330 \times 10^6 \frac{1}{1\,\mathrm{GHz}} = 330\,\mathrm{ms}$ and with the split stage $360 \times 10^6 \frac{1}{1.2\,\mathrm{GHz}} = 300\,\mathrm{ms}$ which is faster.

Note that the extra stage does not slow things down when there is no stall, as in the example below.

```
# Cycle           0  1  2  3  4  5  6  7  8
lw r1, 0(r2)      IF ID EX Ma Mb WB
addi r2, r2, 4       IF ID EX Ma Mb WB
xor r5, r6, r7          IF ID EX Ma Mb WB
add r3, r1, r3             IF ID EX Ma Mb WB
```

Problem 3: [15 pts] Answer the following questions about a `blt` instruction.

(*a*) MIPS lacks an instruction such as `blt r1, r2 TARG` (branch if `r1` less than `r2`). For the questions below, which ask about the suitability of such an instruction, consider implementations similar to the five-stage pipeline covered in class.

☑ Explain why adding such an instruction would slow such implementations even though `beq r1,r2 TARG` is okay.

Because of the time needed to do a magnitude comparison, which starts only after register values are retrieved.

☑ Using a PED, explain why there would be no problem adding a `blt` instruction if the ISA had two delay slots.

Because then the branch could be resolved in EX without penalty, that would give plenty of time for the magnitude comparison. For example, consider the execution of the two code fragments below. The first is for a one-delay–slot MIPS, with a `blt` instruction added. If we want to avoid squashing or stalling for a taken branch, the branch target must be at the input to the PC when the branch is in ID, the end of cycle 2 below. That leaves only one cycle to retrieve `r1` and `r2` from the register file and perform a magnitude comparison. The second code fragment is for a hypothetical two-delay-slot MIPS also with the `blt` instruction. Because there are two delay slots the branch target address does not need to be at the input of the PC until the end of the cycle when the branch is in EX, cycle 2 in the example. That gives the implementation two cycles to retrieve the register values and perform the comparison.

Note that in the example below the second delay slot is filled with a useful instruction. That won't happen all the time for real code.

```
# SOLUTION - Execution on a one-delay-slot MIPS that includes blt.
# Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
 addi r8, r8, 1    IF ID EX ME WB
 blt r1, r2  TARG     IF ID EX ME WB
 add r4, r5, r6          IF ID EX ME WB
 lw r10, 4(r12)

TARG:
 xor r7, r8, r9             IF ID EX ME WB
# Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18

# SOLUTION - Execution on a hypothetical two-delay-slot MIPS.
# Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
 blt r1, r2  TARG  IF ID EX ME WB
 add r4, r5, r6       IF ID EX ME WB
 addi r8, r8, 1          IF ID EX ME WB
 lw r10, 4(r12)

TARG:
 xor r7, r8, r9             IF ID EX ME WB
# Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
```

(*b*) The code fragment below includes the hypothetical MIPS instruction, `blt`.

```
# Hypothetical MIPS Instruction
blt r1, r2, targ
xor r4, r5, r6
```

☑ Show an equivalent MIPS code fragment without using `blt`.

Solution appears below, in which a `slt` instruction is used to perform the magnitude the was performed by the `blt`.

```
# SOLUTION
slt r3, r1, r2
bne r4, r0, targ
xor f4, r5, 46
```

☑ Show an equivalent SPARC V8 code fragment.

Solution appears below. SPARC branch instructions use the value of the integer condition code register, `icc`, to determine whether to take the branch. The `icc` is written by `cc` versions arithmetic and logical instructions, their mnemonics end with `cc`, a common example is `subcc`.

```
# SOLUTION
subcc r1, r2, g0   # g0 = r1 - r2;  icc bits (Negative,oVerflow,Zero,Carry) also set
blt targ           # Branch if negative.
xor r5, r6, r4
```

Problem 4: [20 pts] For the ISA design tradeoff questions below consider the following data: A typical program executes 10^{10} dynamic instructions, of these 1.38×10^9 are branches (half of which are taken), 9.12×10^7 are indirect jumps (`jr` and `jalr`), 2.22×10^8 are direct jumps (`j` and `jal`).

Consider a debate by those developing the MIPS ISA: don't include format J, which means no `j` and no `jal` instructions.

(*a*) Which MIPS instructions can be used to replace `j` and `jal` in the code below, paying attention to the hints in the target names?

☑ Replacement for `j` and `jal` in code below, paying attention to target names.

The solution appears below. The `j` can be replaced by a `beq` instruction that uses a condition that will always be true. The use of a branch was possible because the target of the `j` was not every far away, and so the displacement would fit within the branch's 16-bit displacement (immediate) field.

A branch is not a suitable replacement for the `jal` for two reasons: the `jal` needs to save a return address and because the target is too far away (the jump uses a 26-bit field to store the target, which could be too distant for the branch's 16-bit field). For that reason a `jalr` instruction is used, along with two instructions to load the target address. The `upper_half` and `lower_half` assembler macros (made up) return the corresponding parts of the target address. MIPS assembler programs can use the synthetic `la` instruction to replace the `lui` and `ori` instructions.

```
# SOLUTION
 beq r0, r0, NOT_TOO_FAR_AWAY
 xor r1, r2, r3

 lui r4, upper_half(A_FAR_AWAY_PROCEDURE)
 ori r4, r4, %lower_half(A_FAR_AWAY_PROCEDURE)
 jalr r4
 xor r1, r2, r3
```

(*b*) Determine how much longer (as a percentage or absolute time) it would take to run the typical program described above without the format J instructions. Assume execution is always at 1 CPI. (A formula is okay.)

☑ Percent increase in execution time without format J:

Assume that 25% of the direct jumps are `j` instructions that can be replaced by a single branch. The remainder of the direct branches must be replaced by a set of three instructions. No changes are made to indirect jumps or branches. The increase in the number of cycles will be $2 \times 0.25 \times 2.22 \times 10^8 = 1.11 \times 10^8$. Expressed as a percentage, that is $100\frac{1.11 \times 10^8}{10^{10}} = 1.11\%$.

(*c*) What parts of out implementation would be unnecessary without format J? Approximately how many bypass paths could you add with the cost saved by not having the format J instructions?

☑ What parts would be eliminated?

☑ How many bypass paths could be added with cost savings?

The connection from the `ii` field, which goes to the `IF`-stage mux.

Problem 5: [15 pts] Answer the following ISA design questions.

(*a*) Suppose that *one of the first* modern computer designers, working in the 1940s, made the following statement: "Let's define an 'Instruction Set Architecture' [the last three words spoken slowly, for emphasis] and then later design some hardware 'im-ple-men-ta-tions' of that architecture." Would that be a good idea? Explain. *Hint: Pay attention to the slanted words.*

☑ ISA before implementation in 1940s, good or bad? Explain.

No, because computer engineers did not have enough experience to complete the final design of an ISA before working on its implementation. Lacking this experience, they might have included features in the ISA which would not be practical to build.

Grading note: Many answers gave the usual benefits for separating ISA design from implementation. Some tried to use features of 1940's technology in their answers, such as costliness. Cost is always an issue. The key difference is in the amount of computer design experience of the first computer engineers.

(*b*) CISC ISAs have variable-length instructions.

☑ Benefit of variable-length instructions for CISC.

Allows for small programs. Allows for instructions with long immediates.

☑ Benefit of fixed-length instructions for RISC.

The logic needed to fetch and decode instructions is simple because the address of the next instruction is the current instruction plus (usually) four. Branch displacements can be in units of instructions giving them a four-times longer reach, per bit.

Problem 6: [15 pts] Answer each question below.

(*a*) In our π program demo we found that turning optimization on reduced execution time by half (yay!) but reduced the instruction count even further, from 325 million to 100 million dynamic instructions. This means that optimization *increased* (made worse) CPI from 2.77 to 4.93. *Note: The data is from an earlier semester, so the numbers won't exactly match.*

☑ Optimization is supposed to eliminate stalls, why did the CPI go up?

The divide instructions by far had the longest latency and were responsible for the most stalls. The optimized code had the same number of divides as the unoptimized code, but many fewer instructions overall, and so CPI went up.

☑ What category of instruction was completely eliminated from the loop body of the π program by optimization.

Loads and stores. They were unnecessary because there were enough registers for the values that were being loaded and stored.

(*b*) SPECcpu currently has two sets of results, *base* and *peak*. Consider a third set, *shrink-wrap*, for people who buy mass-marketed software (say, buying a word processor at an office supply store).

☑ Who should use the peak numbers and who should use the base numbers?

Base: Those who plan to use code in which an ordinary effort at optimization will be made. Peak: Those using code developed with a heroic effort put into optimization.

Grading Note: Many answers implied that it was the buyers of software who were responsible for setting up the compile switches. The answers should have indicated that it was those developing software for such buyers who would have to choose compiler switches.

☑ Explain how the run and reporting rules might be modified for the new shrink-wrap results.

The key difference is that shrink-wrap software cannot be custom compiled for your particular implementation (unless you are a really important person, if so send LSU lots of money and we'll be happy to name something after you). Therefore the rules would have to forbid setting switches for the particular implementation on which the benchmarks will be run. Writing a rule to that effect would be tricky, perhaps: The compile switches shall not identify the system being tested for purposes of optimization and the compiler shall not generate the same code as would be generated when specifically targeting the implementation.