Name Solution

Computer Architecture
EE 4720
Midterm Examination

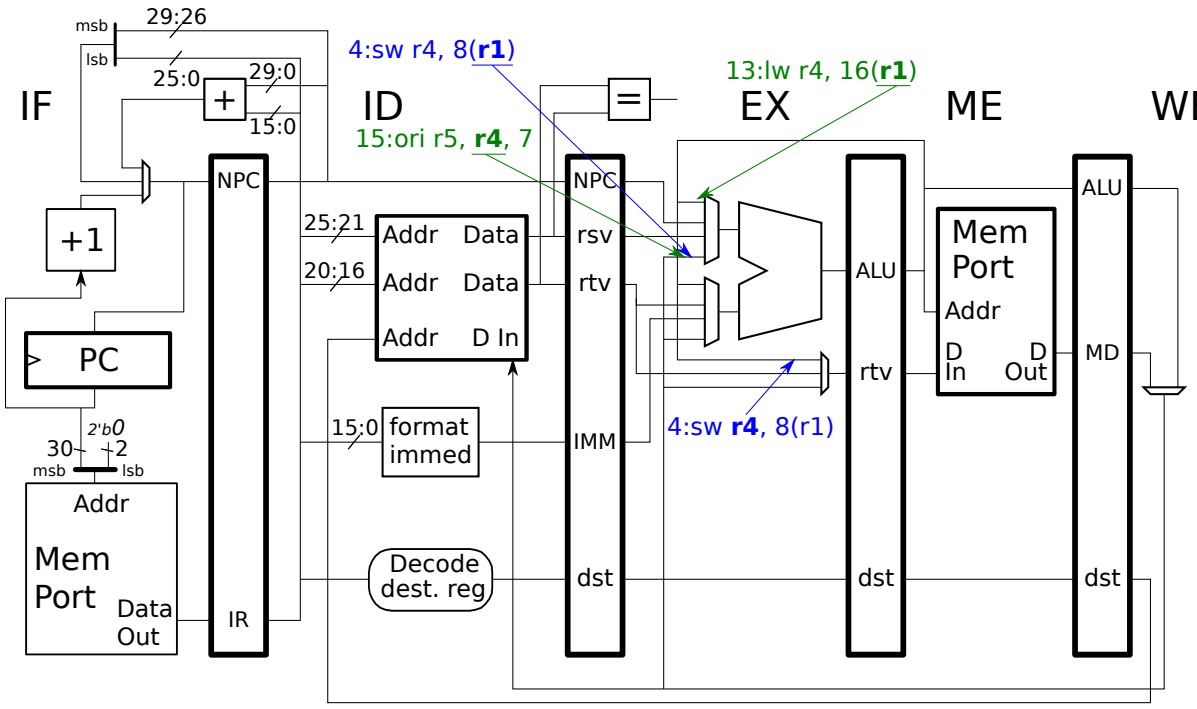
Wednesday, 22 March 2017, 9:30–10:20 CDT

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

Alias MIPS-a-brazo

Good Luck!

Problem 1: [20 pts] Appearing below is our familiar MIPS implementation.



(*a*) Show pipeline execution diagrams for the code fragments below executing on the illustrated implementation and label as indicated.

☑ Show pipeline diagram. ☑ Doublecheck dependencies.

☑ Label bypass paths used **at mux inputs** with $C : I$, where C is the cycle number (such as 2) and I is the instruction **consuming** the bypassed value. Be sure to check for dependencies.

The pipeline diagram appears below and the labeled diagram appears above with the solution to this part in blue. The labels shown above include the complete instruction, not just the instruction name which would have been sufficient for full credit.

```
# Cycle          0  1  2  3  4  5  6
add r1, r2, r3   IF ID EX ME WB
sub r4, r5, r6      IF ID EX ME WB
sw r4, 8(r1)           IF ID EX ME WB
```

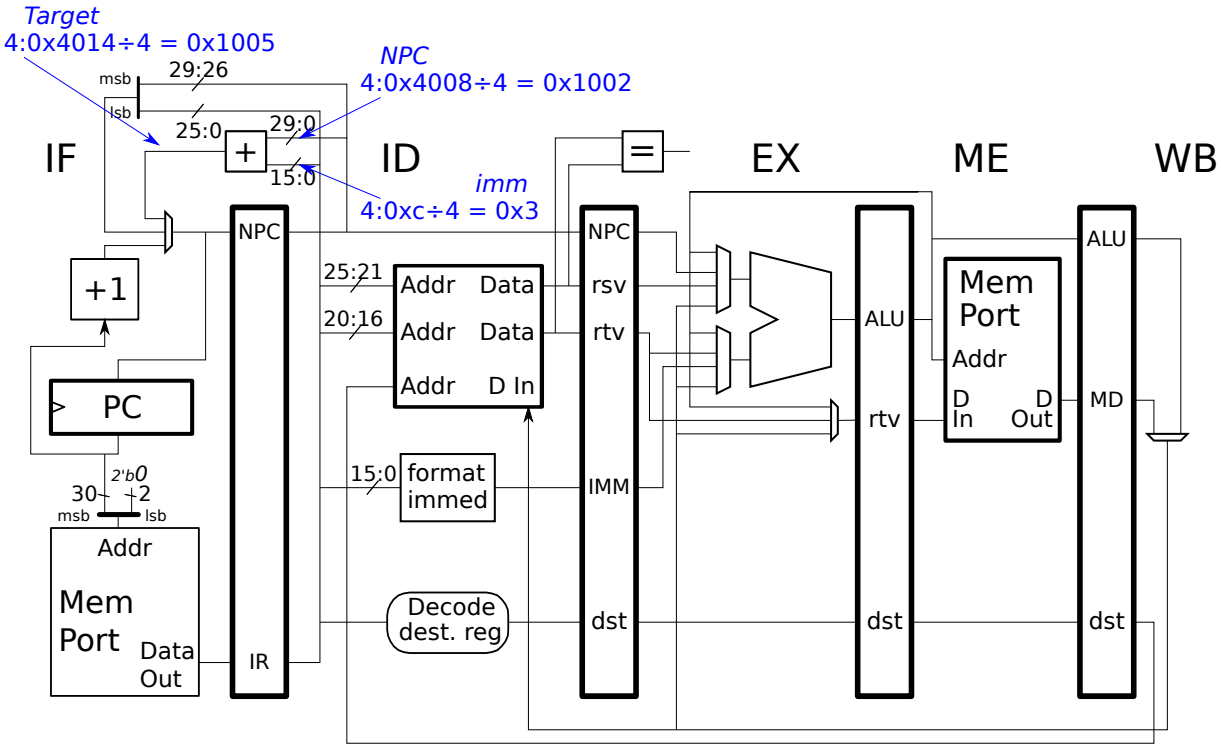
☑ Show pipeline diagram. ☑ Doublecheck dependencies.

☑ Label bypass paths used **at mux inputs** with $C : I$, where C is the cycle number (such as 2) and I is the instruction **consuming** the bypassed value. Be sure to check for dependencies.

The pipeline diagram appears below and the labeled diagram appears above with the solution to this part in green. The cycle numbers start at 10 to avoid confusion with part a.

```
# Cycle          10 11 12 13 14 15 16 17
and r1, r2, r3   IF ID EX ME WB
lw r4, 16(r1)       IF ID EX ME WB
ori r5, r4, 7          IF ID -> EX ME WB
```

Problem 1, continued:



(*b*) Show a pipeline execution diagram for an execution of the code fragment below when the branch is taken. Label the `ID`-stage unit as indicated.

☑ Show pipeline diagram. ☑ Pay close attention to the branch.

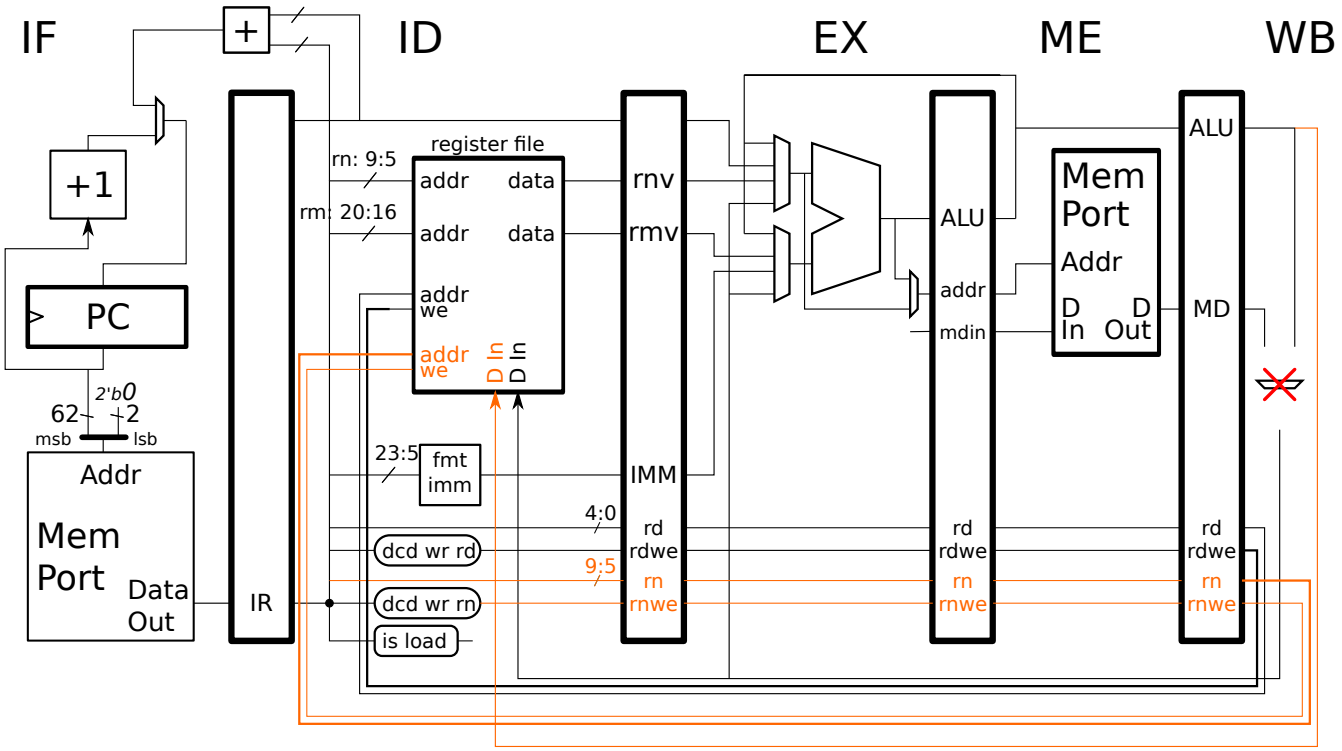
Solution appears below. Note that the branch stalls due to a dependency carried by `r1`. Because there are no bypass paths to the `ID`-stage comparison unit the `bne` must stall until `addi` reaches `WB`.

☑ Label the **inputs and outputs** of the `ID`-stage unit that computes the branch target. Label with $c : v$, where c is the cycle number and v is the value on the input or output.

Solution appears above in blue. Note that `IF.PC`, `ID.NPC`, and `EX.NPC` are all 30 bits and hold bits 31:2 of an instruction address. There's no need to store bits 1:0 since they are zero due to the instruction alignment requirement imposed by MIPS. (That is, the address of a MIPS instruction must be a multiple of 4.)

```
#       SOLUTION
#       Cycle             0  1  2  3  4  5  6  7  8  9  10
0x4000: addi r1, r2, 3    IF ID EX ME WB
0x4004: bne r1, r2, TARG     IF ID ----> EX ME WB
0x4008: lw r4, 0(r5)            IF ----> ID EX ME WB
0x400c: addi r5, r5, 4
0x4010: xor r8, r9, r10
TARG:
0x4014: add r5, r4, r4                      IF ID -> EX ME WB
#       Cycle             0  1  2  3  4  5  6  7  8  9  10
```

Problem 2: [20 pts] Appearing below is a partial implementation of ARM A64 taken from the solution to Homework 4. The WB-stage mux is crossed out because it's wasteful to use a 64-bit mux when the same functionality can be realized using less expensive logic in the ID stage. For reference, some A64 instructions are shown below, the comments show which field registers are encoded in.



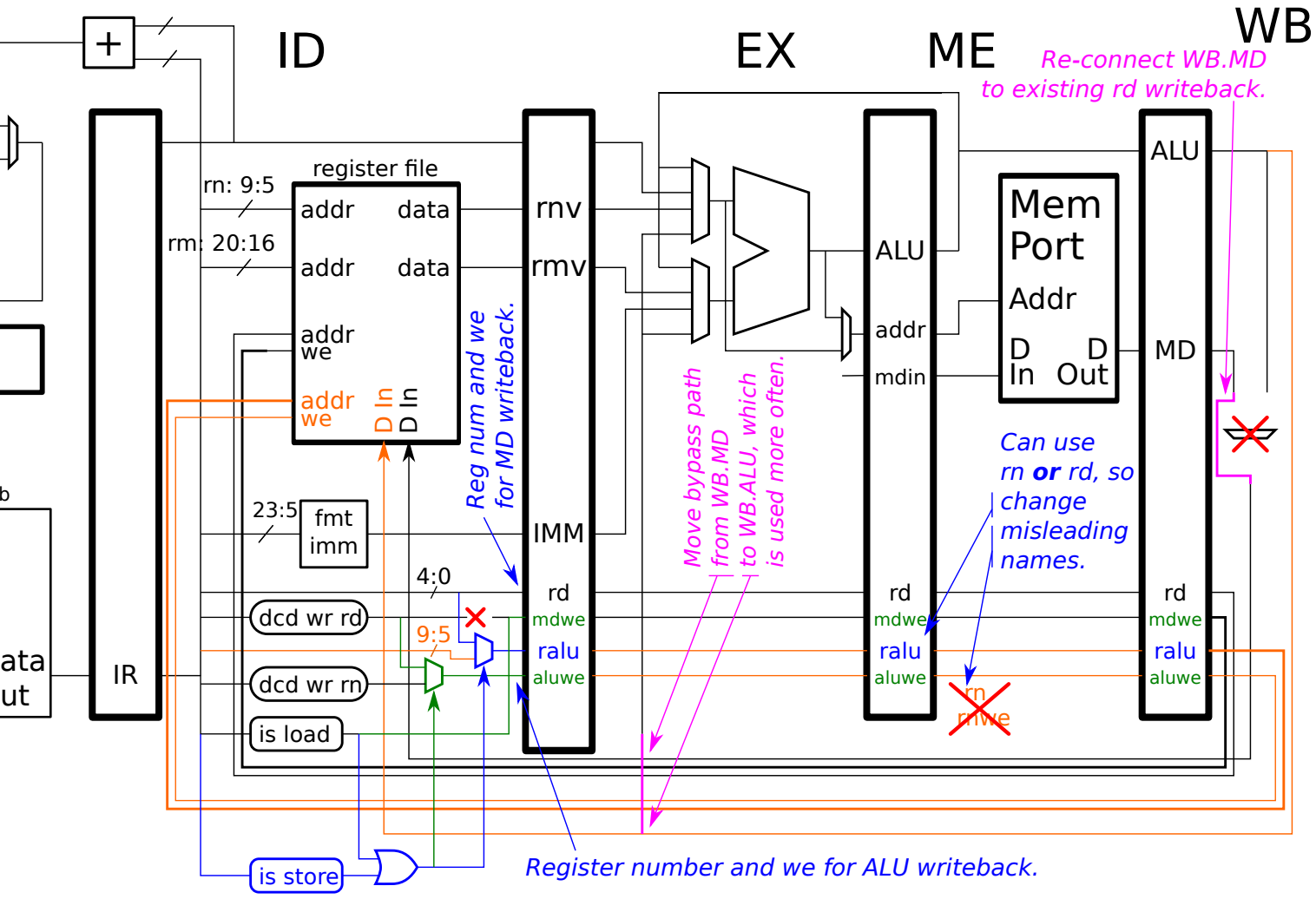

```
@   rd   rn         : Writes rd and rn
ldr x1, [x2], #8    @ x1 = Mem[x2];  x2 = x2 + 8

@   rd, rn, rm      : Writes rd
add x3, x4, x1      @ x3 = x4 + x1

@   rd, rn          : Writes rd
and x1, x2, #34     @ x1 = x2 & 34;
```

Complete the changes so that instructions such as the ones above can write back their results. Assume that only the load instructions write the rn-field register.

Solution on next page.



☑ In WB consider re-connecting wires broken by the removal of the mux.

The WB.MD latch is re-connected to the register file D In (which was connected to the mux output before it was removed), shown in purple. The other input to the mux, WB.ALU, already has a path to the register file and because of the changes in ID that existing path suffices.

☑ In ID make changes so that instruction results can be written back to the correct registers.

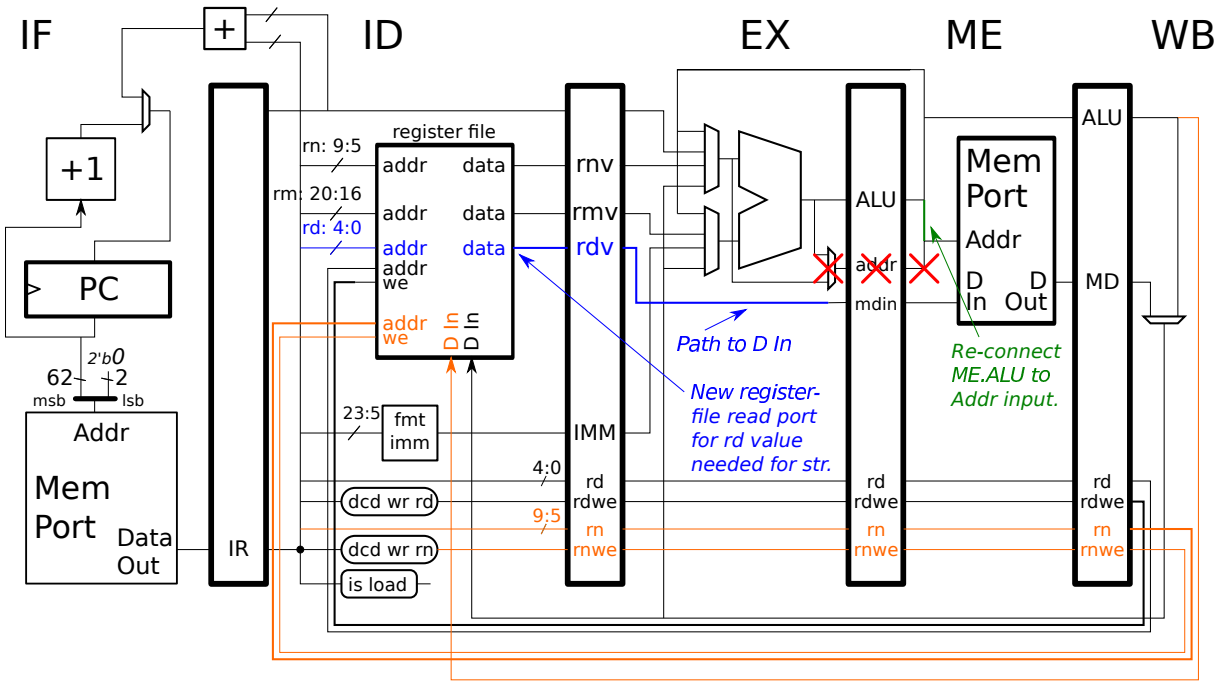
Previously, the black inputs to the register file were used for destinations specified in the instruction rd field (called rt in load instructions), and the orange inputs were used for destinations specified in the rn field, which according to the problem, could only be pre- and post-index memory instructions. Now, the black register file inputs will be used exclusively for WB.MD, which means load instructions, and the orange inputs will be used for WB.ALU, which is used by a variety of instructions. The WB.MD can only be written to rd (also called rt), so the pipeline latches carrying the register number, labeled rd, are unchanged. Because the black register file input is only used by loads, the write enable signal is set to is load rather than dcd wr rd (the logic is near the red × in ID) and the pipeline latch labels have been changed from rdwe to mdwe.

The orange register file inputs can now be used to the register specified in the rd **or** rn field. The only instructions that can use the rn field as a destination are load and store instructions. If there is a load or store instruction in ID, detected by the OR gate, the rn field and dcd wr rn logic are used for the orange writeback signals, otherwise rd and dcd wr rd.

☑ Make the best use of the existing from-WB bypass path.

We would expect that there are more arithmetic instructions than load instructions, so the bypass path has been moved from the WB.MD latch to the WB.ALU latch.

Problem 3: [20 pts] Appearing below again is the partial implementation of ARM A64 taken from the solution to Homework 4.




```
@   rd   rn         : Writes rd and rn.  Post-index
ldr x1, [x2], #8    @ x1 = Mem[x2];  x2 = x2 + 8

@   rd   rn         : Writes rd and rn.  Pre-index
ldr x1, [x2, #8]!   @ x2 = x2 + 8;  x1 = Mem[x2];

@   rd   rn   rm
str x1, [x2, x3]    @ Mem[x2+x3] = x1
```

(*a*) Make changes needed to implement the store instruction, see the example above. Just show datapath, not control logic.

☑ Changes for the store instruction.

Changes are in blue. The store instruction writes the value from the register specified in the `rd` (also called `rt`) field and so we need to add a new read port to the register file. We can't share the port currently used for `rn` or `rm` because some store instructions need both of those values.

(*b*) Do we really need *both* pre-index and post-index addressing for loads and stores? Eliminating either of them will reduce cost, but one's elimination would reduce cost by more than the other's. Indicate which saves more and show the hardware that can be removed and other needed changes. *Note: The phrase* and other needed changes *was not in the original exam.*

☑ Greater cost reduction by eliminating: ◯ *pre-index* ☑ *post-index* (check exactly one).

☑ Show the hardware that's not needed ☑ and other needed changes.

See the red × above. The removed hardware disconnects the Mem Port Addr input, so that's reconnected to `ME.ALU`.

Problem 4: [20 pts] When MIPS routine `coursei` is called register `a0` will hold an entry number, referring to the table at label `courses`. Complete the routine so that when it returns register `v0` will have the integer representation of entry number `a0` in the table. Note that the table itself holds floats. For example, when called with `a0=0` it should return with `v0=2740`, when called with `a0=2` it should return with `v0=3755`, etc.

☑ Complete so `v0` is integer representation of `a0`th table entry.

☑ Read the table as it is, don't modify it or read a different table.

Solution appears below.

```
        .data
courses:
        .float 2740
        .float 3750
        .float 3755
        .float 4755
        .float 4720
        .float 7722
        .float 7725
        .text
        # CALL VALUE: $a0: Entry in table to look up.
        # RETURN:     $v0: Table entry #$a0 represented as an integer.
coursei:
        la $t0, courses


        ## SOLUTION
        #
        sll $t1, $a0, 2    # Multiply by size of float, four bytes.
        add $t2, $t0, $t1  # Compute address of table entry.
        lwc1 $f0, 0($t2)   # Load entry into a FP regiser.
        cvt.w.s $f1, $f0   # Convert to an integer.
        jr $ra
        mfc1 $v0, $f1      # Move integer version of entry to result register.
```

Problem 5: [20 pts] Answer each question below.

(*a*) What kind of implementations were RISC ISAs designed to simplify?

☑ Kinds of implementations that RISC designed to simplify:

Pipelined implementations.

(*b*) Describe how the features below simplify RISC ISA implementations.

☑ Fixed-size instructions.

Short Answer: Because the hardware for fetching the next instruction does not need to check the size of the current instruction nor does it need shift and mask logic to extract the next instruction from the memory fetched in the current and previous cycles.

Detailed Answer: The address of the next instruction is always a constant distance (in many cases 4 bytes) from the address of the current instruction, and so we can compute the next instruction address with an adder connected to the program counter. In contrast, if instruction sizes varied the exact address of the next instruction could not be obtained until the current instruction was decoded. An implementation would either have to wait for the exact address before fetching or it would have to conservatively guess (adding only a small value to the address of the current instruction) and perhaps fetch a larger chunk of data than it would need, and then in a subsequent stage, when the current instruction is decoded, it would have to extract next instruction from the chunk of memory that was fetched.

☑ Avoiding arithmetic instructions that access memory.

Short Answer: There is no need for extra memory stages before the **EX** stage.

More Details: In RISC implementations instructions access the data memory port in a particular stage, **ME** in our five-stage MIPS implementation. If the arguments to arithmetic instruction could come from memory then instructions might have to access a data memory port both before and after they reach the **EX** stage. That would either require additional memory ports, which are costly, or logic that would allow a single memory port to be accessed from multiple stages, which would add stalls and complicate control logic.

(*c*) The SPECcpu package contains the source code for the SPEC benchmarks and scripts to compile and run them, but it does not come with compilers. The tester provides his or her own. Consider a *SPECcpu+* package that comes with compilers, and the requirement that those compilers be used. Why would that make SPECcpu+ less useful to computer engineers?

☑ SPECcpu+ less useful because:

Short Answer: SPECcpu+ cannot be used for the development of a new ISA since there is no way an existing compiler could target the new ISA (otherwise it wouldn't be a new ISA). For the same reason, it could not optimize for a new implementation.

More Details: If we are designing a brand new ISA then it could not possibly be one of the targets of SPEC's compiler and so we would have no way to run SPECcpu+ on our brand new system. If we are designing a new implementation of an existing ISA that SPEC's compiler can generate code for, then the optimizations performed by the compiler might not be right for our brand new implementation. These are not problems if we provide our own compiler.

(*d*) In class we described some optimizations as high-level, and some as low-level, performed by the back end. What distinguishes high- and low-level optimizations? Provide an example of a low-level optimization that could only be performed by the compiler back end.

☑ Difference between high- and low-level optimization.

A high-level optimization can be made without knowing the ISA, but low-level optimizations are made based on the ISA and also the implementation.

☑ Example of an optimization that must be low-level (that can only be done in the back end).

Register assignment.

☑ Briefly explain why.

Because registers can't be assigned if you don't know how many there are, or for that matter, if you don't know the exact instructions that will be accessing those registers.

(*e*) MIPS I has instruction `bgtz r1, TARG` in which the branch is taken if `r1>0` but it lacks an instruction like `bgt r1, r2, TARG` that would branch if `r1>r2`. Why?

☑ Why does MIPS lack `bgt r1, r2, TARG`?

Because the magnitude comparison will take just a bit too long.

☑ What would be the impact on performance of including `bgt r1, r2, TARG` in MIPS on "our" five-stage implementation?

It would lower the clock frequency slowing down all instructions.

The Good News: By including a `bgt` instruction we can eliminate `slt` (set less than) instructions, reducing the instruction count for many programs by 2%.

The Bad News: By lengthening the critical path our clock frequency has been reduced by 5%, which affects **all** instructions.