Name Solution

Computer Architecture
LSU EE 4720
Midterm Solve-Home Examination

Tuesday, 14 April 2020 to Friday, 17 April 2020 23:59 CDT

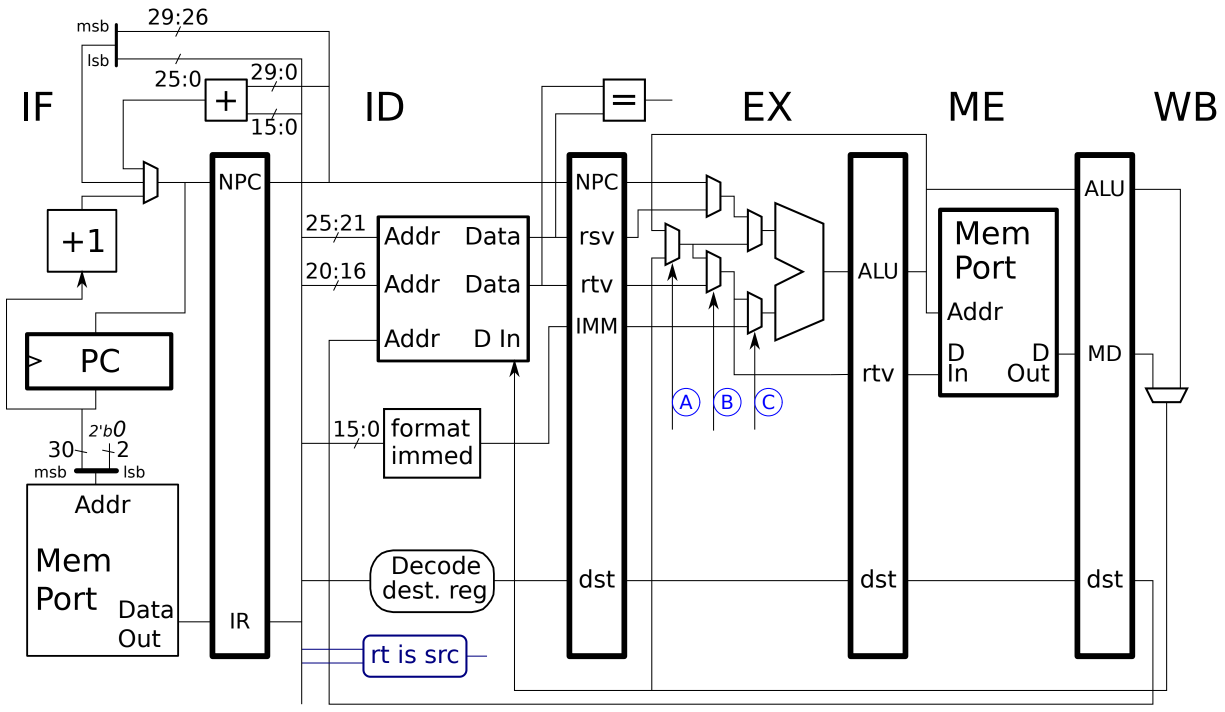
Work on this exam alone. Regular class resources, such as notes, papers, documentation, and code, can be used to find solutions. Do not discuss this exam with classmates or anyone else, except questions or concerns about problems should be directed to Dr. Koppelman.

Problem 1		(15 pts)
Problem 2		(25 pts)
Problem 3		(15 pts)
Problem 4		(15 pts)
Problem 5		(30 pts)
Exam Total		(100 pts)

$r \geq 2\,\mathrm{m} \quad \Rightarrow \quad R_e < 1$

Good Luck! Don't be Foolish!

Problem 1: [15 pts] The pipeline below is a slightly lower cost version of the bypassed MIPS implementation that we've been using. The cost saving is achieved by not allowing an instruction to use a bypassed value from both the `ME` and `WB` stage, the value must come from one stage or the other. Select inputs are shown for three of the re-done `EX` stage multiplexors, they are labeled A, B, and C. For this problem assume that they are connected to properly designed control logic.



Problem continues on next page.

(*a*) Show the values on the labeled select signals for an execution of the code below for those cycles in which an instruction below is in the EX stage. If the value on a select signal does not matter, show an X.

☑ Show values of A, B, and C for when EX occupied by code below. ☑ Use X if value does not matter, blank when no insn in EX.

Solution appears below.

The following is an explanation to help those studying. There is no need to provide such a long-winded answer on an exam. The A select signal is set to 1 when a bypass from WB is needed and is set to 0 when a bypass from ME is needed. The **add** instruction does not use a bypassed value, so A is shown as X (meaning it could be either 0 or 1) in cycle 2 (when **add** is in EX). The **sub** bypasses from ME so A is 0 in cycle 3 and **sw** bypasses from WB in cycle 4 so A is 1.

The B select signal is 0 if the **rt** source is bypassed. Only the **sub** bypasses an **rt** source, so B is 0 in cycle 3, the other instructions use the value from the register file so B is 1 in cycles 2 and 4.

The C select signal is 1 if the immediate is needed at the lower ALU input. That is only true for the **sw**, where the store memory address is computed by adding the immediate, 8, to the **r1** value. The **sw** instruction uses the **rt** value too, but that's the store data which is delivered to D In.

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

#    Cycle         0     1     2     3     4     5     6     7 SOLUTION
 A                             X     0     1
 B                             1     0     1
 C                             0     0     1
#    Cycle         0     1     2     3     4     5     6
```

(*b*) Show a code fragment that would stall on the implementation above but would not stall on our usual bypassed MIPS (which appears in Problem 3).

☑ Code fragment that stalls on this implementation, but not our usual 5-stage MIPS.

Solution appears below. The **xor** instruction uses the result of both the **addi** and **or**. The execution is for "our usual bypassed MIPS," where both values can be bypassed and so no stall is needed.

```
 # SOLUTION
 # Cycle            0  1  2  3  4  5  6   Note: Execution is for usual bypassed MIPS.
 addi R1, r2, 3     IF ID EX ME WB
 or   R4, r5, r6       IF ID EX ME WB
 xor  r7, R1, R4          IF ID EX ME WB
```

Problem 2: [25 pts] Appearing below is the lower-cost MIPS implementation from the previous problem. Design the control logic specified below. The output of $\boxed{\text{rt is src}}$ is 1 if the `rt` field of the instruction specifies a source value, as it does in most type R but only a few type I, such as `sw`. The Inkscape SVG source for the image below can be found at `https://www.ece.lsu.edu/ee4720/2020/mt-p1.svg`.

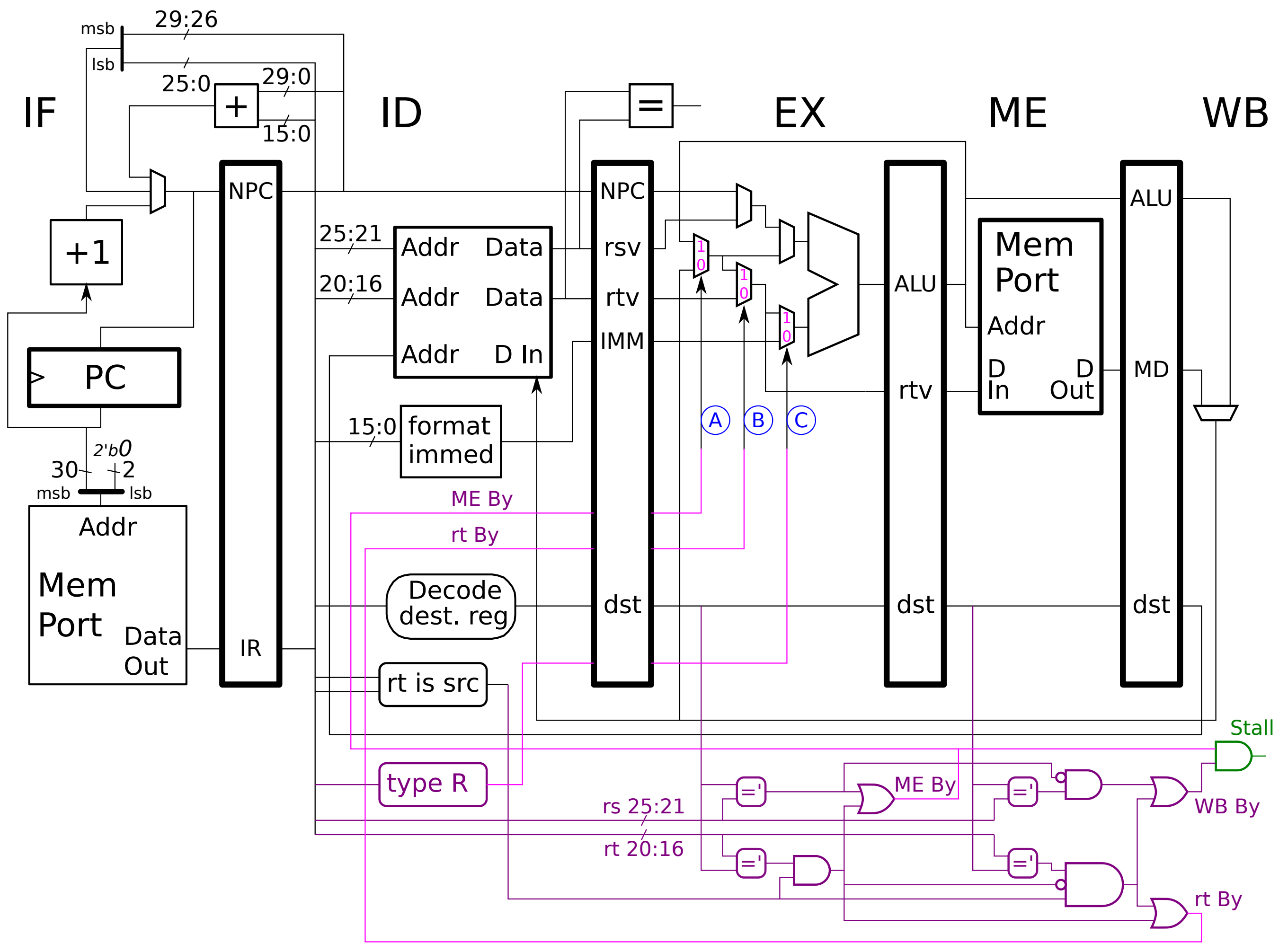
☑ Design control logic for the labeled multiplexor select signals, A, B, and C.

☑ Design control logic to generate a stall signal when a bypass would have been from both `ME` and `WB`.

☑ Pay attention to the usual stuff: ☑ Cost and critical path. ☑ The stage that instructions are in when the select signals are computed and the stage in which they are used.

Solution appears below. Logic for the select signals is shown in two shades of purple (two shades to help emphasize longer wires). Signal `ME By` is 1 if there is a dependence between the instruction in `ID` and the instruction in `EX`. It is used **in the next cycle** for select signal A. Signal `rt By` is 1 if the `rt` register of the instruction in `ID` is a source and it depends on either the instruction in `EX` or `ME`. It is used for B in the next cycle. Signal C is easiest, it is 1 if the instruction is a type R, otherwise it is zero. Type I instructions that use the ALU need the immediate at the lower input. The few type I instructions that use the `rt` value as a source, such as `sw` and `beq`, use the `rt` value in some place other than the ALU, such as the `ID`-stage comparison unit for a `beq`. Logic for the stall signal, in green, simply checks for a bypass from both stages.

The solution discussion continues on the next page.



Problem 2 Common Mistakes: Many solutions would generate a stall signal when a bypass was needed and the instructions in EX and ME wrote the same register. For example in the following code fragment . . .

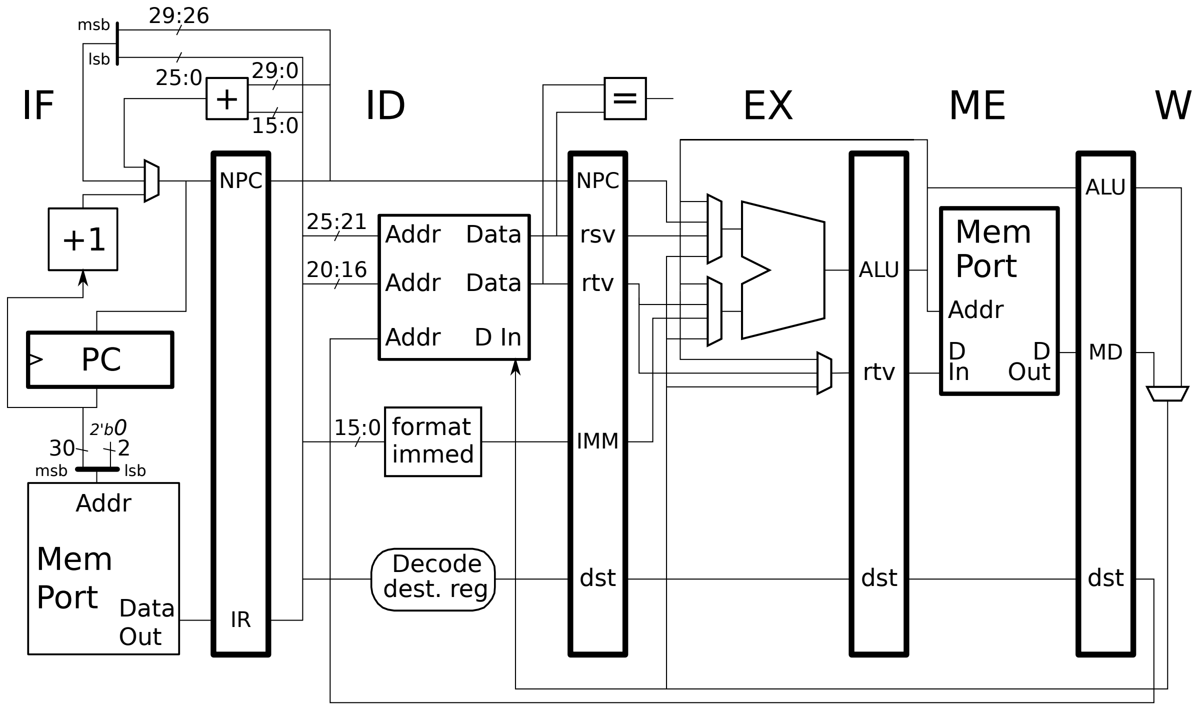
```
# Cycle         0  1  2  3  4  5  6
sub r1, r2, r3  IF ID EX ME WB
add r1, r1, r4     IF ID EX ME WB
and r5, r1, r6        IF ID EX ME WB
```

. . . the **and** instruction uses the value of **r1** computed by the **add**. That value will by bypassed to the **add** in cycle 4 from the ME stage. The value of **r1** carried by the **sub** instruction is irrelevant to the **and**. The correct solution here handles this case using the AND gate with a bubbled input.

In too many solutions the $\boxed{\text{rt is src}}$ was optimistically assumed to compute the exact signal needed by the B select input. I'm not sure if this counts as a mistake, or as a face-saving way to avoid putting in the necessary effort or time management discipline to solve the problem.

On the original exam there was no description of what the $\boxed{\textit{rt is src}}$ *logic block did, but it had been used in earlier assignments in the Spring 2020 semester, and students were free to ask what that block did.*

Problem 3: [15 pts] Show the execution of the code fragment below on the illustrated implementation.



✓ Show execution. ✓ Note that the branch is taken. ✓ Pay attention to the timing of the branch. ✓ Check for dependencies, ✓ including for the branch.

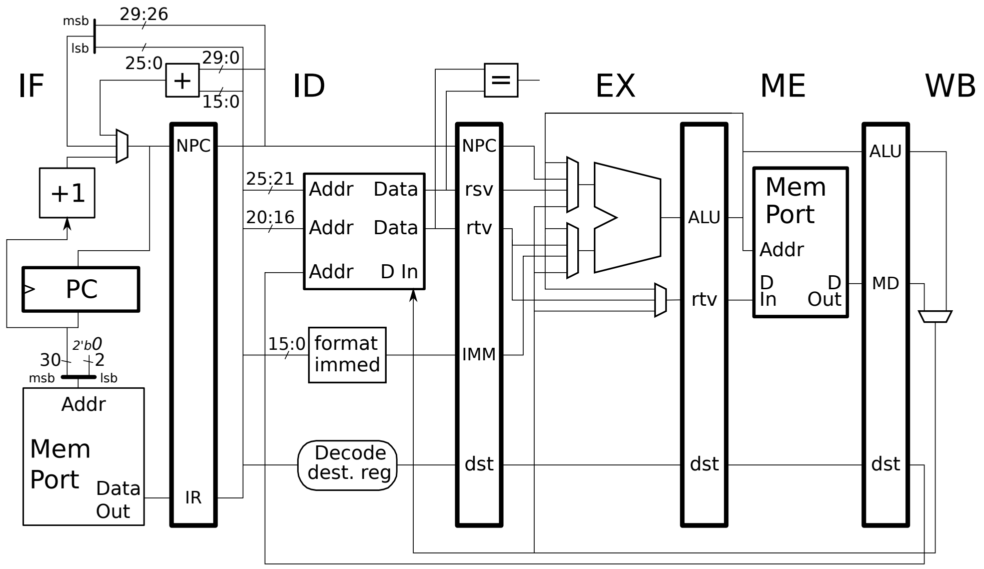
The solution appears below.

One common mistake was overlooking that there is no bypass path to supply the `beq` with `r3` (which is needed in the `ID` stage), and so the branch must stall until `slt` reaches `WB`.

Another common mistake is stalling when a bypass path is available.

```
 # Branch taken.    SOLUTION
 # Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12
 lw r1, 0(r2)       IF ID EX ME WB
 slt r3, r1, r4        IF ID -> EX ME WB
 beq r3, r0  SKIP         IF -> ID ----> EX ME WB
 addi r2, r2, 4                 IF ----> ID EX ME WB
 xor r5, r5, r9
 or  r6, r6, r9
SKIP:
 addi r7, r7, 4                          IF ID EX ME WB
 sw r1, 0(r7)                               IF ID EX ME WB
 # Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12
```

Problem 4: [15 pts] The code fragment below runs inefficiently. Modify the code so that it runs faster on the implementation below. Instructions can be re-arranged, changed, or removed, and registers can be changed. Don't forget that the modified needs to do the same thing as the original code.



☑ Re-write code so that it is faster but, of course, does the same thing as the original.

```
LOOP:
 lw r1, 0(r2)
 andi r1, r1, 0xff
 addi r2, r2, 4
 lw r3, 0(r2)
 srl r3, r3, 24
 add r9, r9, r1
 add r9, r9, r3
 addi r2, r2, 4
 sub r8, r2, r11
 bne r8, r0  LOOP
 nop
```

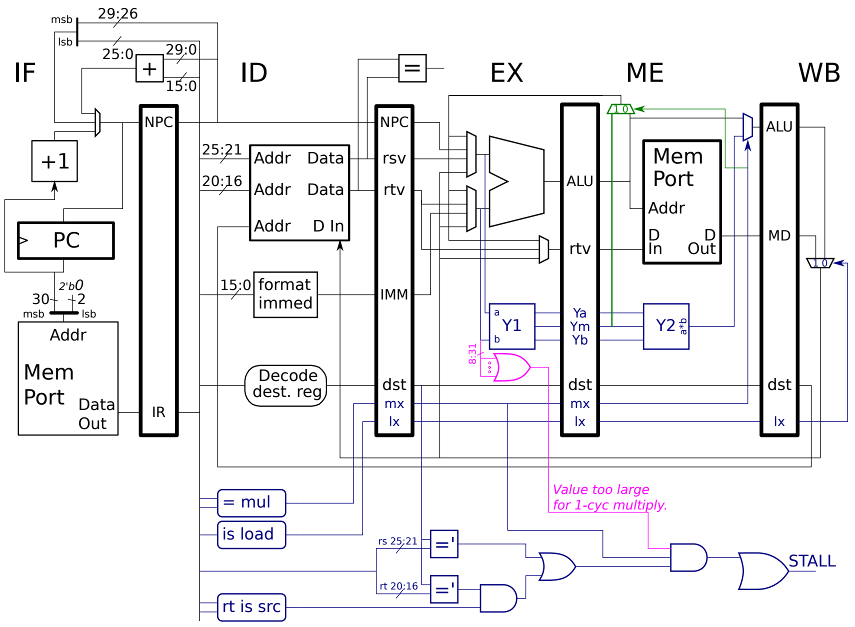
First, the easier optimizations: The two **addi** instructions were replaced by one by using offsets on the load instructions. (More about those offsets later.) Instructions were re-arranged to avoid load-use stalls (for example the stall suffered by the **andi** in the original code waiting for **r1** to reach WB), and the branch delay slot was filled.

The **sub** was eliminated by just checking for equality in the branch, **bne r2, r11**. The **andi** sets **r1** to the 8 least-significant bits of the loaded value (which was in **r1**). The **lbu r1, 3(r2)** loads just those 8 bits, and so the **andi** is no longer necessary. (The offset is 3 because MIPS is big-endian, meaning the memory address, **r2+0**, is where the 8 most significant bits are.) The **srl** instruction extracts the 8 most significant bits of **r3**. The **srl** can be avoided by again using an **lbu** to load just the needed byte. Note that the offset is **-4** because **r2** is incremented by 8 between the two **lbu** instructions.

```
LOOP:   # SOLUTION
 lbu r1, 3(r2)      # Replacement for:   lw r1,0(r2)     andi r1, r1, 0xff
 addi r2, r2, 8     # Replacement for:   addi r2, r2, 4  addi r2, r2, 4
 lbu r3, -4(r2)     # Replacement for:   lw r3,0(r2)     srl r3, r3, 24
 add r9, r9, r1
 bne r2, r11  LOOP  # Replacement for:    sub r2,r2,r11  beq r8, r0, LOOP
 add r9, r9, r3     # Fill branch delay slot.
```

Problem 5: [30 pts] Answer each question below.

(*a*) The code fragments below are to run on the implementation with the small-multiply bypass from Homework 3 and shown to the right. For each code fragment, indicate whether our small-value bypass feature always eliminates a stall, sometimes, or never? Explain.



☑ Eliminates stall on code below: [✓ Always] Sometimes Never

```
andi r3, r5, 0x3f    # SOLUTION: In bitwise AND result the high 24 bits all 0.
mul r1, r2, r3
add r6, r6, r1
lw r10, 0(r6)
```

☑ Eliminates stall on code below: Always Sometimes [✓ Never]

```
ori r3, r5, 0x63f    # SOLUTION: This bitwise OR result must be > 255.
mul r1, r2, r3
add r6, r6, r1
lw r10, 0(r6)
```

☑ Eliminates stall on code below: [✓ Always] Sometimes Never

```
lbu r3, 0(r4)        # SOLUTION: Loaded unsigned byte must fit in 8 bits.
mul r1, r2, r3
add r6, r6, r1
lw r10, 0(r6)
```

☑ Eliminates stall on code below: Always [✓ Sometimes] Never

```
lw r3, 0(r4)         # SOLUTION: Loaded value may or may not be < 256.
mul r1, r2, r3
add r6, r6, r1
lw r10, 0(r6)
```

(*b*) In typical practice a company decides upon an ISA, and then makes multiple implementations of that ISA. Let H_I and L_I be two implementations of ISA I, H_I is a high-end system and L_I is low-cost. Let ISA E (for expensive) be an ISA designed for high-end systems, and ISA C (for cheap) be an ISA designed for low-cost systems, and let H_E and L_C be their implementations. All three ISAs and all four implementations were designed by skilled engineers with lots of resources.

☑ Why might H_E be better than H_I and why might L_C be better than L_I? The same reason should apply to both. The answer is related to the ISAs used for the implementations.

H_E would be better then H_I because ISA E would be designed just for high-end implementations and so can include features that are good for these implementations without regard for whether such features would make it difficult to design low-cost implementations. In contrast, I might have omitted expensive-to-implement features and so H_I would not be as good. Similarly, L_C would be better (perhaps cost less) than L_I because C would omit features that are only needed in high-end implementations.

☑ Even if L_C is better than L_I, why might a user still choose L_I?

Software compatibility. The cash-strapped user buys L_I but hopes that later he can afford H_I and run his software on it unmodified. Had the user bought L_C and then later bought H_E, software would have to be re-compiled before H_E could be used.

(*c*) Consider the preparation of a set of SPECcpu results. For each item below indicate who is responsible, SPEC (the organization) or the tester. Also indicate what would be the problem if it were the other way around. For example, if you answered that SPEC chooses the benchmarks, then explain the disadvantage of having the tester choose the benchmarks.

☑ Choose the benchmarks: [✓ *SPEC*] or *The Tester*

A benchmark suite is only useful if all testers use the same benchmarks.

☑ Problem if it were the other way around:

If each tester chose the benchmarks there would be no way to compare results from two different testers. They could differ even if exactly the same system were tested.

☑ Choose the benchmark input data: [✓ *SPEC*] or *The Tester*

Changing the input data can drastically change the run time, so the reasons are the same as for benchmark choice above.

☑ Problem if it were the other way around:

Certain testers would choose inputs for which run time would be low, thus making the tested system appear fast.

☑ Choose the benchmark training data: [✓ *SPEC*] or *The Tester*

Training data is used for profiling runs. Results from the profiling run are used by the compiler to optimize code, for example, to re-arrange code so that most branches are not taken.

A case could be made that the choice of training data should be based on how the compiler will use the profiling information. Since the compiler choice is up to the tester, training data ought to be up to the tester to. But the SPEC R&R rules say no, testers SHALL use the SPEC-provided training inputs.

☑ Problem if it were the other way around:

If the tester were allowed to choose the training data then certain unscrupulous testers might use the reference data (the data used to compute SPEC scores) for training. That would yield the best results, but does not reflect a real-world situation because for all kinds of reasons developers don't know in advance the exact inputs their programs will be run with. For one thing, each run the program usually get different inputs. How many people write the exact same letter each time they use a word processor?

☑ Choose the compiler: ○ *SPEC* or [✓ *The Tester*]

SPECcpu is designed to test new ISAs and implementations, and to show their full potential. If an ISA is truly new then there is no way SPEC could choose the compiler, the compiler would be developed in-house by the company that designed the new ISAand implementation. Furthermore, the compiler can be thought of as part of the system being tested, for example, it might optimizing assume the presence of certain features.

☑ Problem if it were the other way around:

If SPEC did choose the compiler it would be impossible or difficult to test new designs.

☑ Choose the compiler optimization flags: ○ *SPEC* or [✓ *The Tester*]

If the tester chose the compiler, but SPEC choose the flags that would mean that SPEC is requiring compilers to support a set of SPEC flags. A case could be made for this at the base tuning level, in which the programmer is expected to use a set of flags providing good results, for example, `-O3` (optimization level 3) or `-fast`. But for the peak tuning level SPEC-provided flags would preclude experts choosing flags specially chosen for a particular benchmark on the tested system.

☑ Problem if it were the other way around:

If SPEC chose the flags then the peak results would reflect the best possible performance. That's because SPEC could not be expected to choose them for each system and benchmark. (They certainly could not do that years in advance when the suite is developed, because they couldn't know how to set flags for implementations and using compilers that don't yet exist. It would cost way too much money to have a SPEC team updating the flags for each new benchmark, and anyway would be a source of endless squabbling about the amount of effort the team puts in.)

(*d*) The IA-32 ISA has been described as Intel's golden handcuffs. Who slapped on those handcuffs? What does the gold refer to? What do the handcuffs refer to? *This was discussed in class, but it is okay to use Web searches to answer this question.*

☑ The reason for these handcuffs is:

IBM chose the Intel 8086, sort of an IA-32 implementation, for their personal computer, the IBM PC. That product became very successful, not because it was the first personal computer, but because the IBM name signaled that personal computers were now usable by anyone, not just hobbyists.

☑ They are golden because:

IBM sold lots of PCs so Intel made lots of money.

☑ They are handcuffs (a restriction) because:

Short Answer, but sufficient for full credit: A huge base of software ensured customers for future implementations of the ISA despite its many limitations.

Long Answer: Intel may have felt that the 8086 ISA (called IA-32 in class) was saddled with too many restrictions to be useful for a personal computer ISA with a decade or more of implementations ahead, and so would have wanted to develop a new ISA and scrap IA-32. But a computer with a new ISA would not be able to run all the software developed for the very successful PC, and so buyers would have to wait for new software to become available and hope that their favorite programs would be ported. A buyer then could just as easily by a computer using a non-Intel CPU. Intel and IBM were well aware of this, and so they dared not change the ISA, despite its flaws. Among the more irritating flaws was the need to use a pair of registers to specify a 32-bit memory address. That made address arithmetic much more complicated.

(*e*) Appearing below are some hypothetical instructions. Indicate whether each instruction is a better candidate for a RISC ISA or a CISC ISA. Explain why.

☑ Is the instruction below more ○ *RISC* or [✓ ⊗ *CISC*] like? ☑ Explain.

```
addi r1, r2, 0x12345678
```

RISC ISAs have fixed size instructions, usually 32 bits. That leaves no room for large immediates, including the one in the example above.

Common Mistake: Some incorrectly assumed that 64-bit ISAs had 64-bit instructions and so that immediate could be accommodated. In current use "32-bit" and "64-bit" ISAs refer to the size of a virtual memory address (the only kind used in class so far). Typically the integer register size matches the virtual address size, so 64-bit systems have 64-bit registers. However there is no need to increase the size of the instruction itself. What would a 64-bit type R instruction use all the extra space for—other than immediates? And it would not make sense to nearly double the size of program for those few cases where a larger immediate was needed.

☑ Is the instruction below more [✓ ⊗ *RISC*] or ○ *CISC* like? ☑ Explain.

```
lw r1, (r2+r3)  # Load r1 = Mem[ r2 + r3 ]
```

The instruction above could easily execute in a RISC pipeline, including our 5-stage MIPS (though MIPS lacks such an instruction) in which the ALU could add `r2` and `r3` just as easily as it could add a register value to an immediate.

☑ Is the instruction below more [✓ ⊗ *RISC*] or ○ *CISC* like? ☑ Explain.

```
bgt r1, r2, TARG  # Branch if r1 < r2
```

This can also easily be pipelined. It is omitted from MIPS I only, one assumes, to make it possible to resolve branches in ID. (Testing equality is easier than comparing magnitude.)

☑ Is the instruction below more ○ *RISC* or [✓ ⊗ *CISC*] like? ☑ Explain.

```
add (r1), r2, (r3)   # Mem[r1] = r2 + Mem[r3]
```

In RISC ISAs memory is accessed by memory instructions (loads and stores) other instructions must get their operands from registers, and so the instruction isn't suitable for RISC.