Name Solution

Computer Architecture
LSU EE 4720
Midterm Examination

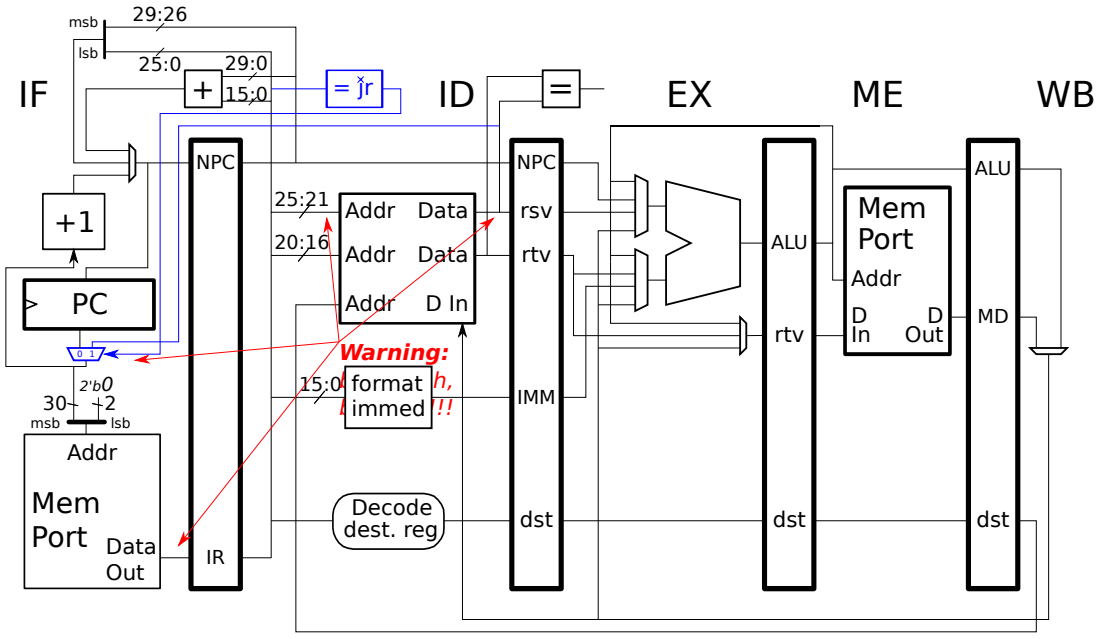
Wednesday, 27 March 2019, 9:30–10:20 CDT

Problem 1	_____	(7 pts)
Problem 2	_____	(17 pts)
Problem 3	_____	(27 pts)
Problem 4	_____	(12 pts)
Problem 5	_____	(12 pts)
Problem 6	_____	(25 pts)
Exam Total	_____	(100 pts)

Alias It's okay when *we* rely on just one.

Good Luck!

Problem 1: [7 pts] The MIPS pipeline below implements a hypothetical MIPS `jr` instruction, the hardware for `jr` is shown in blue. Don't confuse `jr` with the existing MIPS `jr` instruction.



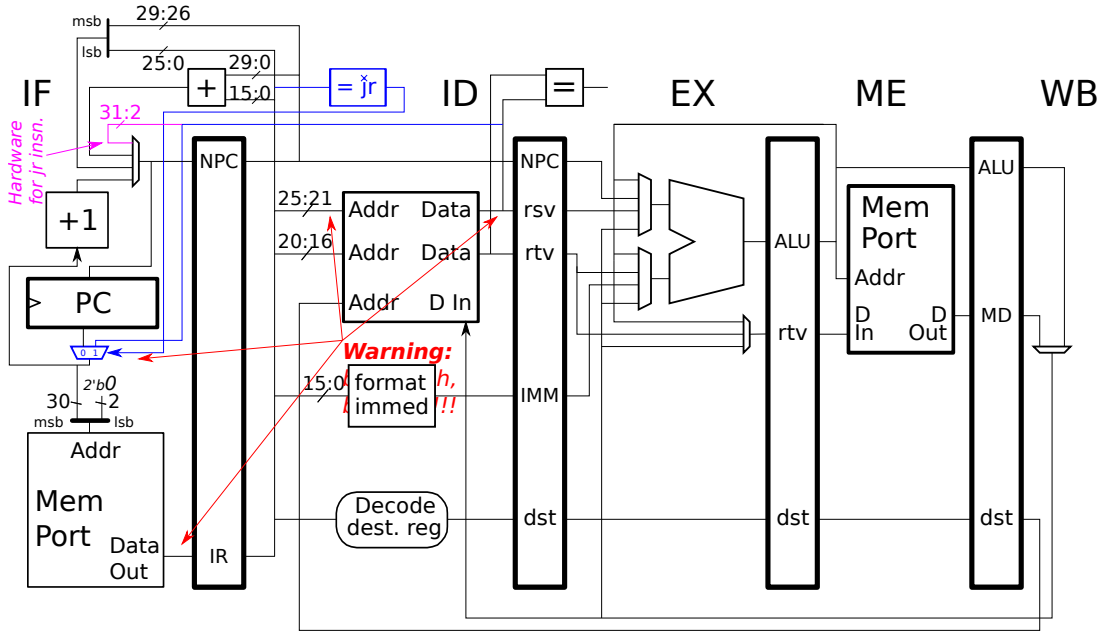
(*a*) The diagram shows a warning in red with lots of arrows and an explanation. Alas, the explanation is covered by the `format immed` box. **Assume that the hardware for `jr` is correct.** Then what can the warning be about?

☑ Reason for warning:

The warning is probably about the critical path which passes through the `IF`-stage memory port. The address input to a memory port should be available at the beginning of a clock cycle but in the `jr` implementation the `IF`-stage memory port address is taken from the register file, and that adds too much to the critical path.

For the record, the text under the `format immed` *box says, "Blah, blah, blah!!!".*

Problem 1, continued: (*b*) *Note: This part did not appear on the exam because the exam was already long enough.* There are two differences between ~~`jr`~~ and `jr`. Fragment A, below, uses `jr`. Complete Fragment B so that it uses ~~`jr`~~, making changes to account for these two differences. Fragment B must jump to the same location and perform the same computation as Fragment A. Register `r9` can be used for intermediate values. *Hint: The differences are when and where.*



☑ Complete code, or for partial credit explain two differences.

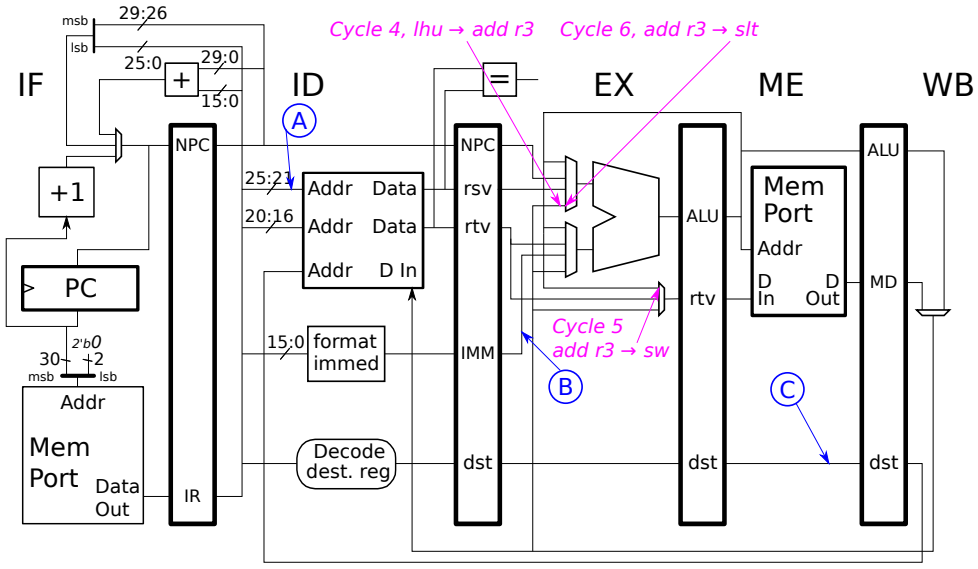
Solution appears below, and the implementation of the ordinary `jr` instruction is shown above in purple. The `jr r1` instruction jumps to the address in `r1` after the delay-slot instruction, in this case `addi r2,r2,4`, is executed. The hardware in purple above shows the path used by the `jr` instruction. It carries `rsv` (the rs value, in the example the value of `r1`) from the register file (in ID) to the PC mux (in IF). As is done with the branch and jump targets, the two least-significant bits of the `rsv` are not used since these will be zero for any valid instruction address. The ~~`jr r1`~~ instruction is different in two ways. First, it jumps to the address `4*r1`. That's because unlike the path used by `jr`, the two-least significant bits of the register value are retained and two zeros are prepended to the less-significant side at the input to the IF-stage memory port. The second difference is that the target address for ~~`jr`~~ is at the input to the memory port when ~~`jr`~~ is in ID, meaning that the target is in IF while the ~~`jr`~~ is in ID, and so the target is fetched one cycle earlier than `jr` (and one cycle earlier than the branch instructions). This also means that ~~`jr`~~ lacks a delay-slot.

So that Fragment B jumps to the same address as Fragment A, the address must be shifted right by two bits. Because ~~`jr`~~ lacks a delay slot, the delay slot instruction for `jr`, `addi`, must be moved before the ~~`jr`~~.

```
# Fragment A -- Uses jr. Don't modify it.
lw r1, 0(r6)
jr r1
addi r2, r2, 4
xor r3, r4, r5

# Fragment B --  ☑  Finish code below, use ~~jr~~   SOLUTION
lw r1, 0(r6)     # Load address of instruction to jump to.
srl r1, r1, 2    # Divide address by 4.
addi r2, r2, 4   # ~~jr~~ lacks a delay slot, so move addi before ~~jr~~
~~jr~~ r1            # Jump to target
xor r3, r4, r5   # Not executed. (Unless reached by some other code.)
```

Problem 2: [17 pts] The code below executes on the illustrated implementation. The implementation has hardware that enables the `bne` to avoid the stall, but that hardware is not shown.



(*a*) The illustration has several circled letters pointing to wires. In the diagram below show the values on those wires on each cycle the value is used.

☑ Show values for ☑ A, ☑ B, and ☑ C, show these ☑ for each cycle used.

Solution appears below. In cycle 7 the B row (the immediate) shows the branch displacement, -6 instructions. For the C row it is important to show zeros for instructions that don't write a register, `sw` and `bne`.

```
LOOP: # Cycle    0    1    2    3    4    5    6    7    8    9    10   11
 lhu r1, 8(r2)   IF   ID   EX   ME   WB
 addi r2, r2, 2       IF   ID   EX   ME   WB
 add r3, r1, r3            IF   ID   EX   ME   WB
 sw r3, 12(r6)             IF   ID   EX   ME   WB
 slt r5, r3, r4                 IF   ID   EX   ME   WB
 bne r5, r0  LOOP               IF   ID   EX   ME   WB  # No stall? Next prob.
 addi r6, r6, 4                      IF   ID   EX   ME   WB

#       Cycle    0    1    2    3    4    5    6    7    8    9    10   11
A:                    2    2    1    6    3    5    6                   # SOLUTION
B:                    8    2         12        -6    4                   # SOLUTION
C:                              1    2    3    0    5    0    6         # SOLUTION
#       Cycle    0    1    2    3    4    5    6    7    8    9    10   11
```

(*b*) For each dependency below highlight, on the illustration, the multiplexor input that provides the bypassed value. Also indicate the cycle in which the bypass occurs.

☑ Dependence from `lhu` to `add r3`. ☑ Cycle when bypass used.

☑ Dependence from `add r3` to `sw`. ☑ Cycle when bypass used.

☑ Dependence from `add r3` to `slt`. ☑ Cycle when bypass used.

Solution appears in the implementation diagram in purple.

Problem 3: [27 pts] In the previous problem the `bne` did not stall despite a dependence with `slt`. Code with a similar dependence appears below. Add hardware to the implementation below that correctly sets the Taken signal for such `slt rX, rY, rZ` to `bne rX, r0` dependencies. Branches without the dependence should not be effected. Note that the result of `slt` is 0 or 1. Some useful hardware is shown in blue.

```
LOOP: # Cycle      0  1  2  3  4  5  6
 slt r5, r3, r4    IF ID EX ME WB       # Note: r5 is 0 or 1.
 bne r5, r0  LOOP     IF ID EX ME WB
 addi r6, r6, 4          IF ID EX ME WB
```

☑ Add logic to set `Taken` correctly for the dependence described above.

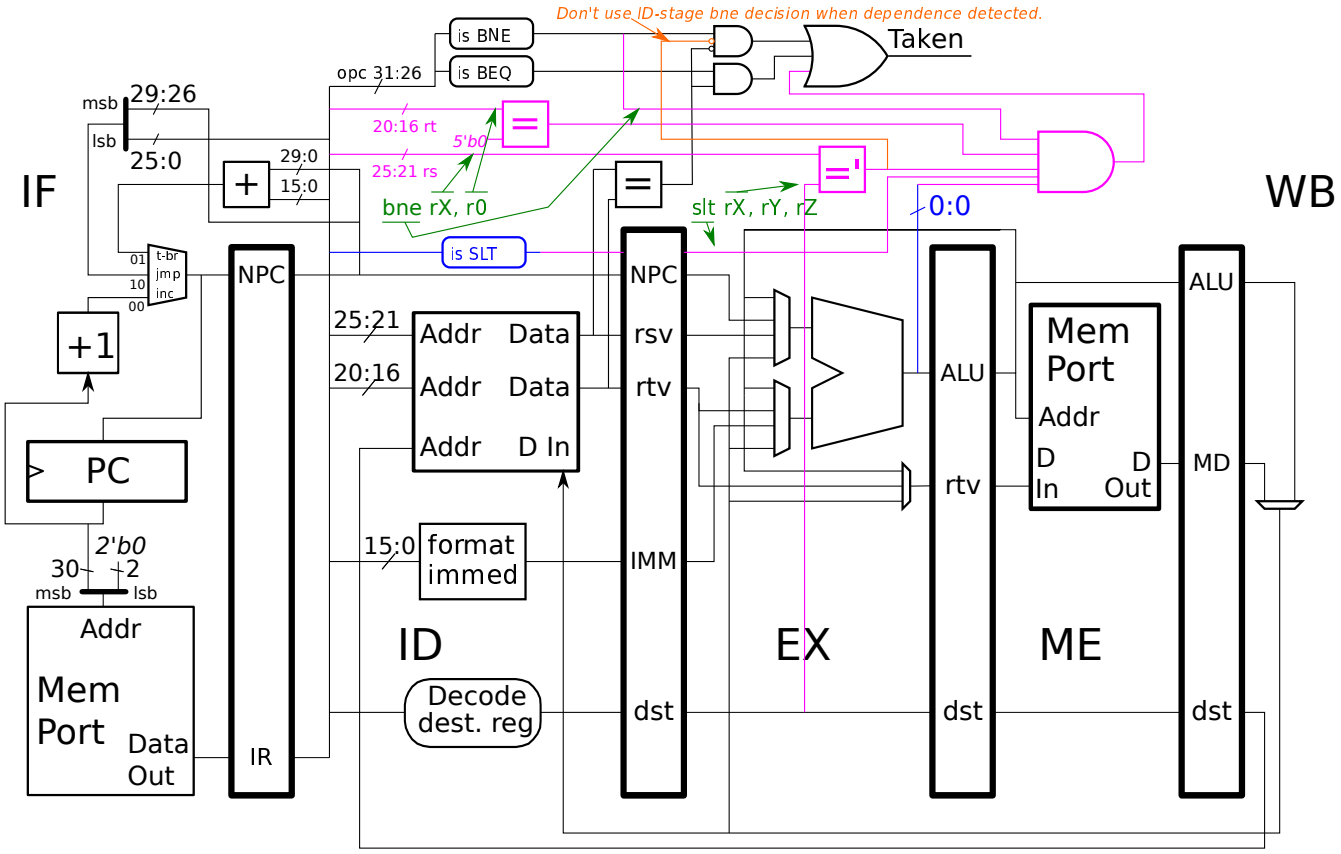
☑ Avoid costly solutions. No part of the solution should operate on 32 bits.

☑ Check that the second branch register is `r0`. ☑ Branches without the dependence should not be effected.

Solution appears below in purple and orange. Sample instructions are shown in green. The inputs to the big AND gate from top to bottom check for the following: (1) that the instruction in `ID` is a `bne`, (2) that the `rt` register of the branch is `r0`, (3) that the `rs` register of the branch is the same as the destination register of the instruction in `EX`, (4) that the instruction in `EX` is an `slt`, and (5) that the `slt` result is 1 (meaning that the condition is true).

To reduce hardware cost the logic checks whether `rt=0` (a 5-bit comparison) rather than `rtv=0` (a 32-bit comparison).

Note: The following is the solution to 2019 Homework 5 Problem 2. If the `slt/bne` dependence is detected then the branch taken condition computed using `rsv` and `rtv` (using the = logic) needs to be ignored. To do so a new bubbled input, shown in orange, has been added to the AND gate computing the `bne` taken condition. That input only checks for the dependence (`ID`-stage `rs` with `EX`-stage destination). (It doesn't matter what instruction is in the `EX` stage, if there's a dependence that taken signal should be suppressed.)



Problem 4: [12 pts] The loop below writes zeros to a range of memory.

```
 # Call Value: r1 is address of the start of the region to zero.
 # Call Value: r3 is the number of bytes to zero.
 add r2, r1, r3  # Memory location at which to stop.
 addi r2, r2, -1
LOOP:
 sb r0, 0(r1)
 bne r1, r2, LOOP
 addi r1, r1, 1
```

(*a*) Compute the rate that it zeros memory when it runs on our bypassed 5-stage pipeline. Use an appropriate unit.

☑ Rate at which loop above copies data.

Short answer: Rate is $1/(7-3) = .25$ bytes per cycle.

Appearing below is a pipeline diagram showing the execution of the loop. The first iteration completes without a stall, but the dependence from the **addi** to the **bne** causes a one-cycle stall on subsequent iterations. The time for the second iteration is $7 - 3 = 4\,\text{cyc}$ and we can expect subsequent iterations to also take 4 cycles. Since only one byte is zeroed the rate is $\frac{1}{4}\frac{\text{B}}{\text{cyc}}$.

```
 # SOLUTION
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
 sb r0, 0(r1)      IF ID EX ME WB
 bne r1, r2, LOOP     IF ID EX ME WB
 addi r1, r1, 1          IF ID EX ME WB
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
 sb r0, 0(r1)            IF ID EX ME WB
 bne r1, r2, LOOP           IF ID -> EX ME WB
 addi r1, r1, 1                IF -> ID EX ME WB
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
 sb r0, 0(r1)                        IF ID EX ME WB
 bne r1, r2, LOOP                       IF ID -> EX ME WB
 addi r1, r1, 1                            IF -> ID EX ME WB
```

(*b*) Apply loop unrolling and make other changes so that the code writes at the rate of two bytes per clock cycle, assuming favorable values of `r1` and `r3`. State those assumptions.

☑ Show unrolled loop and make other changes for 2 byte per cycle copy.

Short Answer: The unrolled loop appears below.

Explanation: To increase the rate at which memory is zeroed two `sw` instructions are used instead of one `sb`. The second `sw` uses an offset of 4, avoiding the need for a second `addi` instruction. Since the loop body has one more instruction than the original loop the branch does not need to stall, so the loop still takes 4 cycles per iteration. The loop zeroes memory at the rate of $8/4 = 2$ bytes per cycle.

☑ Assumption about `r1`: is a multiple of 4. (See below)

☑ Assumption about `r3`: is a multiple of 8. (See below)

Because the base address of `lw` must be 4-byte aligned, `r1` must be a multiple of 4. Because the address (`r1`) is incremented by 8 each iteration, the number of bytes to zero must be a multiple of 8.

```
 # Call Value: r1 is address of the start of the region to zero.
 # Call Value: r3 is the number of bytes to zero.
 add r2, r1, r3

# SOLUTION - Code
 addi r2, r2, -8
LOOP:
 sw r0, 0(r1)
 sw r0, 4(r1)
 bne r1, r2, LOOP
 addi r1, r1, 8

# Execution of solution code.
#
 add r2, r1, r3     IF ID EX ME WB
 addi r2, r2, -8       IF ID EX ME WB
LOOP: # Cycle       0  1  2  3  4  5  6  7  8  9  10 11  Time: 6-2 = 4 cycles
 sw r0, 0(r1)             IF ID EX ME WB
 sw r0, 4(r1)                IF ID EX ME WB
 bne r1, r2, LOOP               IF ID EX ME WB
 addi r1, r1, 8                    IF ID EX ME WB
LOOP: # Cycle       0  1  2  3  4  5  6  7  8  9  10 11 12 13
 sw r0, 0(r1)                         IF ID EX ME WB
 sw r0, 4(r1)                            IF ID EX ME WB
 bne r1, r2, LOOP                           IF ID EX ME WB
 addi r1, r1, 8                                IF ID EX ME WB
LOOP: # Cycle       0  1  2  3  4  5  6  7  8  9  10 11 12 13
```

Problem 5: [12 pts] The MIPS code below adds an integer value loaded from memory to the constant π.

(*a*) Suppose no other instructions needed the value of `a0` that was loaded. Modify the code to use fewer instructions.

☑ Use fewer instructions.

Solution appears below. Since the value loaded into `a0` is only used by the FP code, it would be more efficient to load it into a FP register (using `lwc1`) and so avoid the need of an `mtc1` instruction to move the value from `a0` to a FP register.

```
        .data
famous_constant: # 0x10010300
        .float 3.141592654
        .text
pie:

        # lw $a0, 0($t2)  # SOLUTION - Replace lw ..
        lwc1 $f1, 0($t2)  # .. with a lwc1

        lui $t0, 0x1001
        lwc1 $f0, 0x300($t0)

        # mtc1 $f1, $a0   # SOLUTION - Remove mtc1, since val already in f1.
        cvt.s.w $f2, $f1
        add.s $f3, $f0, $f2
```

(*b*) Modify the code below so that it would run correctly if `a0` were a floating-point value. Also use fewer instructions.

☑ Fix code so it works correctly if `a0` is FP.

Solution appears below. Since the loaded value is already FP there is no need for the conversion instruction, `cvt.s`. As in the previous problem, the value is loaded directly into a FP register, avoiding a need for the `mtc1`.

```
        .data
famous_constant: # 0x10010300
        .float 3.141592654
        .text
pie:

        # lw $a0, 0($t2)  # a0 is FP!  ☑  Fix code below for FP a0.
        lwc1 $f2, 0($t2)     # SOLUTION: Replace lw with lwc1.

        lui $t0, 0x1001
        lwc1 $f0, 0x300($t0)

        #  mtc1 $f1, $a0     # SOLUTION - Don't need move insn, val in a FP reg.
        #  cvt.s.w $f2, $f1  # SOLUTION - Don't need convert, it's already in FP repr.
        add.s $f3, $f0, $f2
```

Problem 6: [25 pts] Answer each question below.

(*a*) SPARC divides the 32 integer registers an instruction can access into four groups, `%l0 to %l7`, `%g0 to %g7`, `%o0 to %o7`, and `%i0 to %i7`. The names reflect how they might be used in programs, and how SPARC's register windowing feature affects them when `save` and `restore` instructions are executed. Explain what the first letter of each group stands for. Explain what happens to the values in those registers when `save` or `restore` instruction is executed.

☑ Word that `l`, `g`, `o`, and `i` each stand for.

They are: local, global, output, input.

☐ What happens to values on a `save` or `restore`.

Values in the global (`g`) registers aren't changed by `save` and `restore`.

FINISH.

(*b*) Instructions like `addi r1, r1, 1` occur frequently in programs. For this add-one-to-a-register operation CISC ISAs have a specialized instruction, for example `inc r1`. MIPS lacks such an increment instruction.

☑ Why do MIPS and other RISC ISAs lack such an instruction?

Because there would be no benefit to having an `inc` instruction. The `addi` instruction would do exactly the same thing. Furthermore, a new `inc` instruction would require a new opcode, and there are only a limited number of opcodes, so such an opcode would be wasted.

☑ Why do CISC ISAs have such an instruction?

☑ What is the benefit that RISC ISAs don't realize?

Short Answer: A CISC `inc` instruction would be shorter than a CISC `add` instruction, and so programs using `inc` would be shorter.

Explanation: CISC ISAs have variable-length instructions. So an `inc r1` instruction would take less space than an `addi r1, r1, 1` since there would be no need to specify the increment amount (it's always 1) and there would be no need to list a separate source and destination registers (since they'd be the same).

(*c*) A CPU design team has an area budget that they can use for bypass paths. There's not enough area for all the bypass paths they'd like. They are simulating these design alternatives to determine which is best.

Explain the role that the compiler people have in this process.

☑ Role for compiler writers in deciding on bypass paths.

They need to provide back-ends for each alternative design that optimizes taking into account the bypass paths they provide. By doing so compiled code would have fewer stalls than a compiler that optimized for an old design. This would enable the team to properly assess how well each set of bypass paths performs.

(*d*) Provide examples for the following common optimizations. Show code to which this optimization can apply and how it is optimized.

☑ Dead-code elimination. ☑ Example code and code after optimization.

Solution appears below. Because `planb` is false the `if` part is never executed, so after DCE only the else part remains.

```
// SOLUTION: Before DCE
bool planb = false;
if ( planb ) x = value >> ( log2p1(y) >> 1 );  else x = sqrt(y);

// After DCE.
// Note: Code below would be in compiler's intermediate representation.
x = sqrt(y);
```

☑ Constant propagation and folding. ☑ Example code and code after optimization.

Solution appears below.

```
// SOLUTION: Before constant propagation and folding.
int wp_lg = 5;
int wp_sz = 1 << wp_lg;
int wp_num = tid / wp_sz;

// After optimization.
// Note: Code below would be in compiler's intermediate representation.
int wp_num = tid / 32;
// Note: A strength reduction optimization would replace divide by a right shift.
```

(*e*) Described below are three tuning levels for C++ programs, two of which are SPECcpu tuning levels the other was just made up. Identify the one which is not a SPEC tuning level, and explain why it isn't.

Level A: *Each C++ program is compiled without optimization.*

☑ Is it a SPEC level? No ☑ If so, name it: _____

☑ Indicate the users that will benefit from this level, or why no one would benefit.

Not appropriate for anyone, because almost everyone runs programs with some optimization.

Level B: *Each C++ program can have its own set of optimization flags.*

☑ Is it a SPEC level? Yes ☑ If so, name it: Peak

☑ Indicate the users that will benefit from this level, or why no one would benefit.

Peak. For those who plan to have highly motivated experts tune their code.

Level C: *All C++ programs must be compiled with the same set of optimization flags.*

☑ Is it a SPEC level? Yes ☑ If so, name it: Base

☑ Indicate the users that will benefit from this level, or why no one would benefit.

Base. For those who plan to run code prepared with a typical level of effort.