Name Solution

Computer Architecture
EE 4720
Midterm Examination

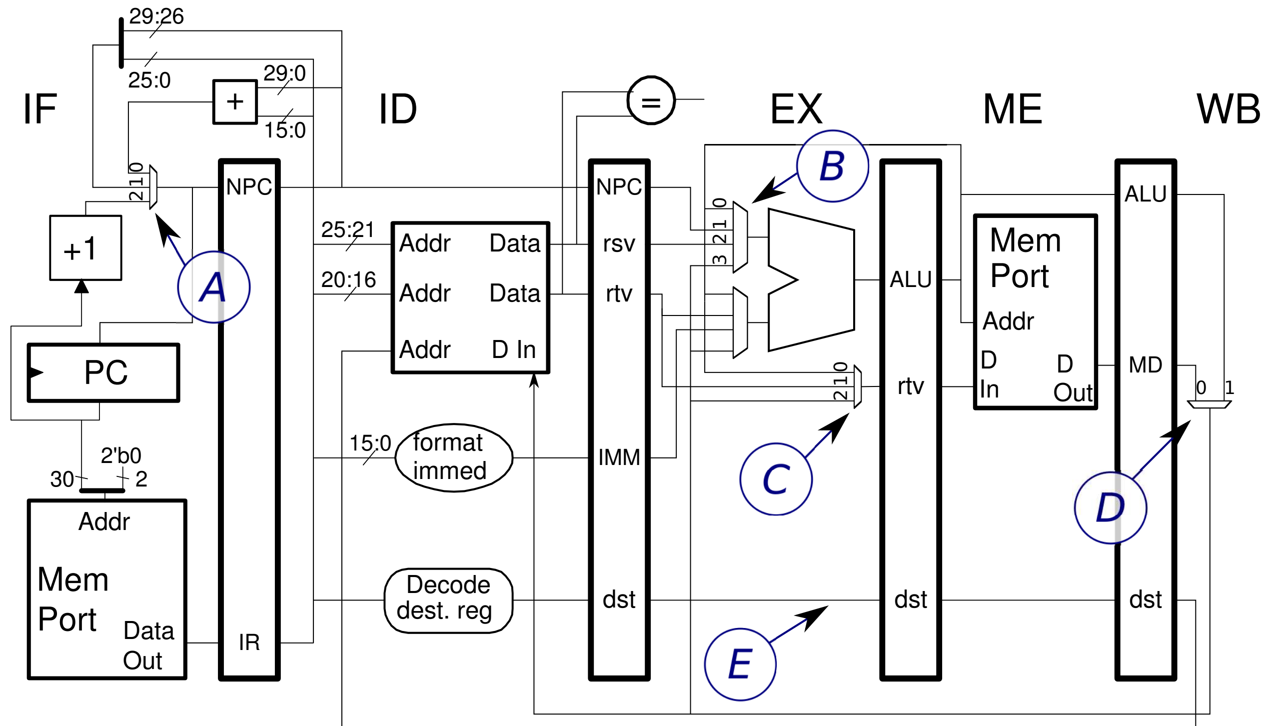
Monday, 31 March 2014, 9:30–10:20 CDT

Problem 1		(20 pts)
Problem 2		(20 pts)
Problem 3		(18 pts)
Problem 4		(12 pts)
Problem 5		(6 pts)
Problem 6		(12 pts)
Problem 7		(12 pts)
Exam Total		(100 pts)

Alias -O11

Good Luck!

Problem 1: [20 pts] The MIPS code fragment below executes on our familiar five-stage scalar MIPS implementation.



(a) Show the execution of the code for enough iterations to determine CPI, and determine the CPI assuming a large number of iterations.

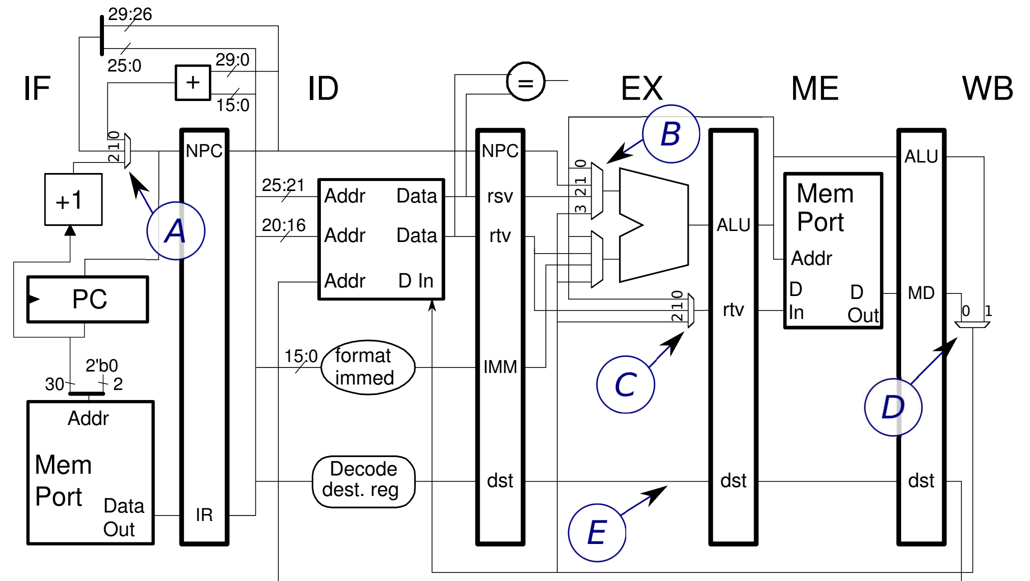
(b) There are five labels in the diagram, labels A through D point at multiplexor control inputs and label E points at the output of the `EX.dst` pipeline latch. Show the values of these signals up until the second execution of `lw r1`. For A show only values $\neq 2$, for E show only values $\neq 0$, for the others only show values at cycles in which the multiplexor is in use.

USE NEXT PAGE FOR SOLUTION

```
LOOP: Cycle        0
 A                 2
 B
 C
 D
 E                 0
 lw r2, 0(r5)      IF
LOOP:
 lw r1, 0(r2)
 sh r1, 16(r2)
 bne r1, r0, LOOP
 add r2, r2, r3
 xor r4, r5, r2
```

USE NEXT PAGE FOR SOLUTION

Problem 1, continued:



- [x] Check code for dependencies.
- [x] Show pipeline execution diagram for enough iterations to determine CPI.
- [x] Determine the CPI.
- [x] Show values for A through E.

Solution appears below. The pipeline execution diagram is straightforward. Don't forget that for the `sh r1, 16(r2)` instruction both `r1` and `r2` are source registers, and that there is no destination register. Also notice that `lw r1` only stalls in the first iteration. In the row for signal A, non-default values are present when a branch instruction is in ID, this is in cycles 6 and 11, the value is 0, picking up the output of the adder. Signal B is the upper ALU mux. A value of 2 indicates that the value from the register file is used (not a bypassed one). The `lw r2, 0(r5)` and `add r2,r2,r3` instructions use this value. A value of 0 indicates a bypass from ME, this occurs in cycle 9 when the `lw r1` picks up `r2` from the `add`. In cycle 4 the `lw` bypasses from WB. Signal C is for the write value of store instructions (register `r1` in this case), D picks which to write back, an ALU output or a memory output. Signal E is the destination register number of the instruction in EX.

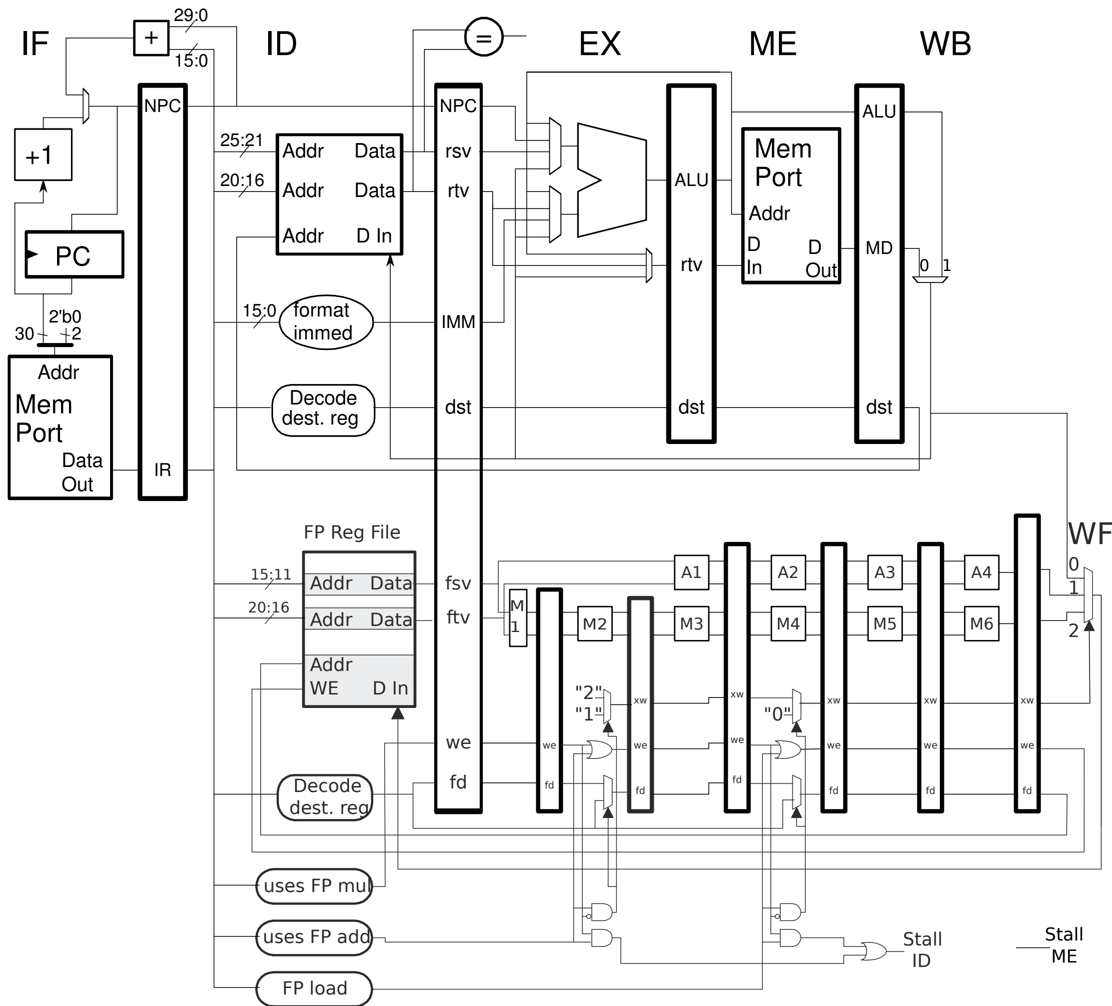
The CPI is computed using execution between cycles 12 and 7, in which the iterations start identically. The value is $\frac{12-7}{4} = 1.25$.

```
#     SOLUTION
LOOP: # Cycle     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
 A                2                 0              0           Blanks = 2
 B                      2     3     2     2  0
 C                                  2              2
 D                            0     0           1  0
 E                0     2     1           2  1           2     Blanks = 0
 lw r2, 0(r5)     IF ID EX ME WB
LOOP: # Cycle     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
 lw r1, 0(r2)        IF ID -> EX ME WB
 sh r1, 16(r2)          IF -> ID -> EX ME WB
 bne r1, r0, LOOP             IF -> ID EX ME WB
 add r2, r2, r3                     IF ID EX ME WB
 xor r4, r5, r2
LOOP: # Cycle     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
 lw r1, 0(r2)                          IF ID EX ME WB
 sh r1, 16(r2)                            IF ID -> EX ME WB
 bne r1, r0, LOOP                            IF -> ID EX ME WB
 add r2, r2, r3                                    IF ID EX ME WB
LOOP: # Cycle     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
 lw r1, 0(r2)                                         IF ID EX ME WB
```

Problem 2: [20 pts] The pipeline execution diagram below shows the *incorrect* execution of the MIPS code for the illustrated pipeline. That is, in the pipeline below the stall to avoid the structural hazard would have occurred when `lwc1` was in the `ID` stage.

```
# Cycle            0  1  2  3  4  5  6  7
add.s f1, f2, f3   IF ID A1 A2 A3 A4 WF
addi r1, r1, 4        IF ID EX ME WB
lwc1  f4, 0(r1)          IF ID EX ME -> WF
```

USE NEXT PAGE FOR SOLUTION



USE NEXT PAGE FOR SOLUTION

As described below, modify the pipeline so that FP load instructions, such as `lwc1`, will stall when they are in `ME` rather than `ID` to avoid a `WF` structural hazard.

(*a*) Show logic to generate a signal `Stall ME` in the correct cycle. When `Stall ME` is logic `1` stages `ME` and earlier will stall.

(*b*) Cross out the logic generating the `Stall ID` for the `lwc1` that is no longer needed.

(*c*) Make other changes, including control logic, so that the correct destination register number, destination value, and write-enable signal are delivered to the FP register file. *Hint: Some solutions are simpler than others.*

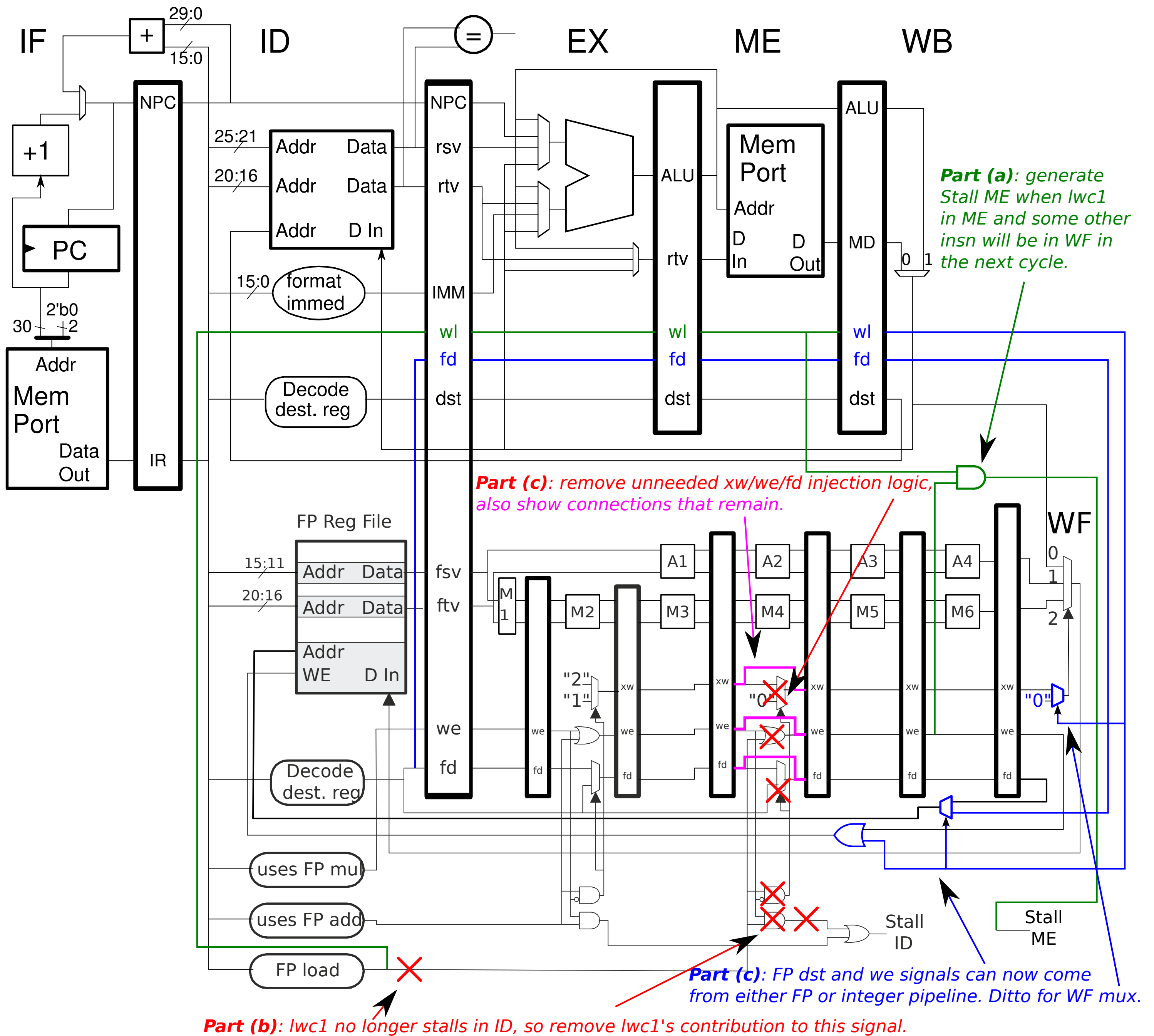
Problem 2, continued:

☑ Generate `Stall ME` to avoid `lwc1 WF` structural hazard. ☑ Make sure signal is `1` in the right cycle.

☑ Cross out `Stall ID` logic for FP loads, but leave logic for other instructions.

☑ Make sure that correct `fd`, `we`, and value delivered to FP register file.

The solution appears below, see the next page for more explanation.



The solution appears on the previous page. Looking at the code execution example, notice that in the original pipeline (see the previous page) at cycle 3 the load instruction must be placing its `we` and `fd` values in the `A2` stage, but that would replace (clobber) the corresponding values for the `add.s` instruction which are also in that stage. Since the destination for the `add.s` is lost, there is no way it can execute correctly.

The solution is to place the write enable and FP destination for FP load instructions in the integer pipeline, using new pipeline latch segments `wl` (write-enable FP load) and `fd`. This keeps them separate from those of the add or multiply instructions, and it ensures that the `Stall ME` signal will stall the `lwc1` without affecting instructions in the FP pipeline.

The logic to generate the `Stall ME` signal appears in green. We want to stall `ME` when the `lwc1` instruction is in `ME` and when there is some instruction in `A4/M6`, this is detected by the green AND gate.

For part b, the logic generating `Stall ID` for load instructions is crossed out, shown in red along the bottom.

For part c we must make sure that the write enable and destination that we put in the integer pipeline reach the FP register file, we also want to make sure the WF mux is set correctly. That's shown in blue. The `fd` values from the two pipelines enter a new mux, the control signal is just `wl` (write-enable FP load). For the write enable we just use an OR gate (we could have used a mux, but an OR gate achieves the same result). The zero value for the WF mux is inserted using a new mux.

Also for part c we must remove the old logic inserting `fd`, `we`, and `xw` for the load instruction, that's shown in red with the surviving connections shown in purple.

Problem 3: [18 pts] Answer each question below.

(*a*) Re-write the SPARC code fragment below in MIPS. Pay attention to the load instruction and the instructions related to the branch. Don't forget that in SPARC assembly language the last register is the destination.

```
!
addcc %g1, %g2, %g3
ld [%g5+%g3], %g4
bg TARGET          ! Branch greater than zero.
add %g5, 22, %g6   ! g6 = g5 + 22
```

☑ MIPS equivalent of SPARC code. ☑ Pay attention to load and branch-related insn.

The solution appears below.

Grading Note: Many students incorrectly assumed that the condition for the `bg` *instruction was taken from the* `ld` *instruction. SPARC condition codes are set explicitly: by instructions ending with* `cc`. *In this case the* `addcc` *instruction sets the condition codes.*

```
# SOLUTION
add r3, r1, r2
add r10, r3, r5
lw r4, 0(r10)
bgtz r3  TARGET
addi r6, r5, 22
```

(*b*) Show a MIPS equivalent of the ARM A32 code below. The MIPS code will use more than one instruction. (This is based on Homework 3.) *Note: A32 added in 2017 to avoid confusion with the A64 instruction set.*

```
add r1, r2, r3, LSL #4
```

☑ MIPS equivalent of ARM code above.

```
# SOLUTION
sll r1, r3, 4
add r1, r1, r2
```

(*c*) Explain why the MIPS code fragment below is certain to execute with an error.

```
lw r1, 0(r2)
lw r3, 1(r2)
```

☑ Error is certain because:

In MIPS `lw` memory addresses must be a multiple of 4, this rule is called an *alignment restriction*. The memory addresses used are `r2+0` and `r2+1`. They can't both be a multiple of 4. Note that it is possible that `r2+1` is a multiple of 4, there is no reason why the offset itself (`1` in this case) must be a multiple of 4.

Problem 4: [12 pts] Answer each question below.

(*a*) In the execution below the **ant** instruction raises an illegal instruction exception and the handler is fetched, note that the handler starts execution in cycle 4. Explain why this exception (as executed) cannot be precise and show how execution should proceed for our five-stage pipeline.

```
# Cycle             0  1  2  3  4  5  6
 add r10, r11, r12  IF ID EX ME WB
 sw r3, 4(r5)          IF ID EX x
 ant r1, r2, r5           IF*ID*x
 or  r5, r6, r7              IF x

HANDLER:
 sw                             IF ID ..
```

☑ Reason why this execution not for a precise exception.

It is not precise because the sw did not finish executing. In a precise exception code should execute correctly up to the instruction just before the faulting instruction, the faulting instruction and those after it cannot modify registers or memory.

☑ Show execution that could be precise when ID stage discovers the bad ant opcode.

Solution appears below. Note that the exception is not acted upon until the faulting instruction reaches the ME stage. We need to wait that long to make sure that the sw instruction does not raise an exception, if it did we'd want to handle that exception first and leave the ant for later.

```
# SOLUTION
# Cycle             0  1  2  3  4  5  6  7
 add r10, r11, r12  IF ID EX ME WB
 sw r3, 4(r5)          IF ID EX ME WB
 ant r1, r2, r5           IF*ID*EX MEx
 or  r5, r6, r7              IF ID EXx

HANDLER:
 sw                                   IF ID ..
# Cycle             0  1  2  3  4  5  6  7
```

(*b*) An interrupt mechanism will switch the processor from user mode to privileged mode when the handler starts. What is the difference between user and privileged mode and why are they necessary to implement an operating system?

☑ Difference between user and privileged mode? ☑ Importance for OS?

In privileged mode the system can execute any instruction and access any memory location. In user mode an attempt to access a *system memory address* or execute a system instruction will result in an exception.

By preventing user programs from touching or even reading certain memory locations, the OS can implement controls over what programs do. For example, preventing a program from making arbitrary disk accesses by mapping the disk hardware control registers to memory locations in system space.

Problem 5: [6 pts] In class we described three broad families of ISAs: CISC, RISC, and VLIW.

(*a*) In which ISA family is it easy to have instructions with long (say, 32 bit) immediates? Explain how.

☑ ISA family with long immediates is:

☑ What about the ISA family enables this?

CISC ISAs have instructions with long immediates. They can accommodate them because of CISC's variable instruction size. If you need a big immediate, just use a big instruction.

(*b*) In which ISA family are dependencies between instructions explicitly given? How is the dependence information supposed to help?

☑ ISA family in which dependence information is part of program is:

☑ Reason for providing dependence information.

Dependence info is part of the program in VLIW ISAs. This information is intended to simplify the design of the hardware by providing it information a little sooner than it would otherwise obtain it. (A little sooner because it does not need to look at instructions' register operands.)

(*c*) Which ISA family tends to have the most compact programs? (That is, the size in bytes of the program is smallest.) What makes them compact?

☑ ISA family with most compact programs is:

☑ Program sizes are small because.

CISC ISAs have the most compact programs, that is because of their variable instruction sizes. All RISC instructions must be 32 bits, in some cases wasting space. Also, CISC instructions can do the work of several RISC instructions, saving space. Since VLIW ISAs bundle RISC-like instructions, they suffer the same program size issues as RISC programs.

Problem 6: [12 pts] Answer each question below.

(*a*) Consider a 2-way superscalar implementation and a 10-stage implementation, both derived from our 5-stage MIPS used in class. All implementations include reasonable bypass paths.

Explain how the instruction sequence below would always result in a stall on the 10-stage pipeline but might not result in a stall on the 2-way system. (The compiler cannot separate the two instructions.) Illustrate your answer with a pipeline execution diagram.

```
add r1, r2, r3
sub r4, r1, r5
```

☑ Reason for maybe stall on superscalar and certain stall on 10-stage.

☑ Illustrate using a pipeline diagram.

The different executions are shown below. The 10-stage implementation must stall because there is no way the sum from the **add** can be ready in the next cycle. But for the superscalar system there is no need to stall if the **add** and **sub** are in different fetch groups. In the first superscalar example below they are in the same fetch group, and there is a stall, but in the second they are in different groups and so there is no stall.

Though the superscalar system might seem to have the advantage, when it does stall it costs two instructions, whereas for the 10-stage system only one instruction execution opportunity is lost.

```
# SOLUTION  - Unavoidable stall in pipeline.
add r1, r2, r3  IF1 IF2 ID1 ID2 EX1 EX2 ME1 ME2 WB1 WB2
sub r4, r1, r5      IF1 IF2 ID1 ID2 --> EX1 EX2 ME1 ME2 WB1 WB2

# SOLUTION  Stall in 2-way Superscalar
0x1000 add r1, r2, r3  IF1 ID1 EX1 ME1 WB1
0x1004 sub r4, r1, r5  IF2 ID2 --> EX2 ME2 WB2
0x1008 xor                 IF1 --> ID1 EX1 ME1 WB1
0x100c lw                  IF2 --> ID2 EX2 ME2 WB2

# SOLUTION  No stall in 2-way Superscalar
0x1004 add r1, r2, r3  IF2 ID2 EX2 ME2 WB2
0x1008 sub r4, r1, r5      IF1 ID1 EX1 ME1 WB1
0x100c xor                 IF2 ID2 EX2 ME2 WB2
0x1010 lw                      IF1 ID1 EX1 ME1 WB1
```

(*b*) Consider two MIPS implementation, one is an n-way superscalar of MIPS-I, with n sets of FP units, the other is a scalar implementation of a hypothetical MIPS-I-vec, which includes an n-lane vector unit for the vector instructions in MIPS-I-vec. Both the superscalar and vector MIPS implementations can sustain execution at n FP operations per cycle. *Note: The phrase "In terms of n" did not appear in the original exam.*

☑ In terms of n how does the cost of bypass paths compare in the two systems?

For the scalar 5-stage implementation the value computed by the ALU is bypassed back to the EX stage when it is in the ME and WB stages. Since there are two ALU inputs the number of connections is $2 \times 2 = 4$. In the superscalar design there are n ALUs and so n values, these must be bypassed back to the two inputs in each of n ALUs, for a total of $2n \times 2n = 4n^2$ connections.

For the n-lane vector unit, we would expect that the value produced by an ALU in a lane would only be bypassed back to the ALU in that lane. So the number of connections is n times that in a scalar processor, or $4n$ connections.

☑ In terms of n how does the cost of registers compare in the two systems?

Short Answer: For the superscalar system the cost is constant (does not change with n). For the n-lane vector unit the cost is proportional to n.

Long Answer: A n-way statically scheduled superscalar processor has the same number registers regardless of n. (Things will be different when we consider dynamic scheduling later in the semester.) An ISA with vector instructions also has vector registers. If those instructions are for an n-lane unit the registers will have space for n operands.

Problem 7: [12 pts] Answer each question below.

(*a*) The SPECcpu benchmarks are designed to evaluate the CPU and memory system in new implementations and ISAs. To realize this testers are responsible for providing a compiler and for compiling the benchmarks.

Suppose SPEC required testers to use a compiler that they provide. How much would this impact the goal of testing new ISAs? New implementations of existing ISas?

☑ Degree of impact of SPEC-provided compilers for testing new ISAs. ☑ Explain.

If SPEC is supplying the compiler and the ISA is new, then SPEC would need a new compiler back-end for the new ISA. That would take a lot of time and effort for SPEC to develop and so there would be a long delay before SPECcpu scores could be obtained on the new ISA, and so the degree of impact would be large. Note that existing SPEC compilers could not be used for the new ISA, since they don't know how to generate instructions for the ISA (since its new).

☑ Degree of impact of SPEC-provided compilers for testing new implementations of old ISAs. ☑ Explain.

When a new implementation is developed compiler optimizations are adjusted or developed for the new implementation. In many cases the improvements are small. For this reason, using an old compiler on a new implementation would have a small impact on the results.

Though the impact would be small, it's still not a good thing to do because in many cases the compiler back end is developed by the same people developing the implementation, and there is no reason why the two (compiler and chip) should not be tested as a unit.

(*b*) A company releases a SPEC disclosure in which they have substituted the bzip2 benchmark with another version of bzip2 which does the same thing but runs faster. Since it does the same thing it provides the correct output to the SPEC verification script. As a result they get very good scores. How will they get caught? How is it against the goal of SPECcpu?

☑ How will the benchmark substitution get caught?

Anyone disclosing a SPECcpu score must provide a "config" file with all of the settings, and any software (such as compilers) needed to build the benchmarks must be publicly available (though not necessarily free). Therefore, it is easy for a 3rd party to re-run the test and they will discover the discrepancy.

☑ How does the substitution undermine what SPECcpu is supposed to measure?

The obvious answer is that if people don't know your substituting benchmarks then they'll attribute the better scores to the computer (until the substitution is discovered).