Name Solution

Computer Architecture
EE 4720
Midterm Examination

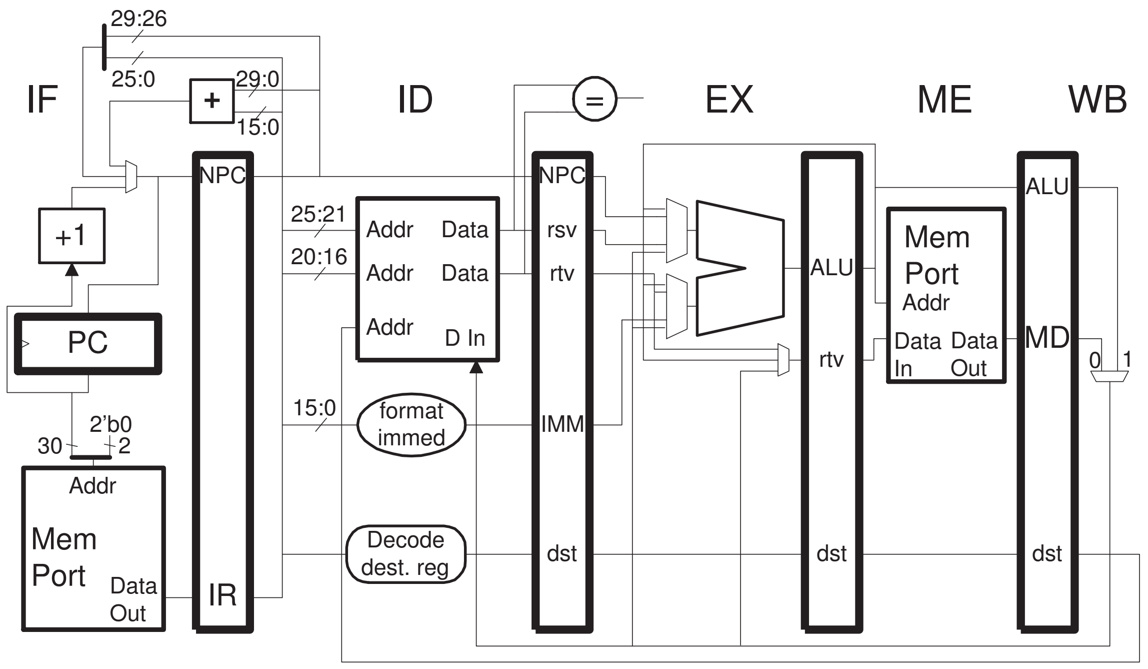
Wednesday, 3 November 2010, 10:40–11:30 CDT

Problem 1		(30 pts)
Problem 2		(15 pts)
Problem 3		(15 pts)
Problem 4		(23 pts)
Problem 5		(10 pts)
Problem 6		(7 pts)
Exam Total		(100 pts)

Alias ¡Mi! ¡Mi! ¡Mi! ¡Ips! ¡Ips! ¡Ips! ¡Cinco etapes hacen MIPS!

Good Luck!

Problem 1: Show the execution of the following code fragments on the illustrated MIPS implementations.



(*a*) [20 pts] The code below executes for many iterations. Show a pipeline execution diagram for the execution of the code on the implementation above for enough iterations to determine the CPI, and determine the CPI.

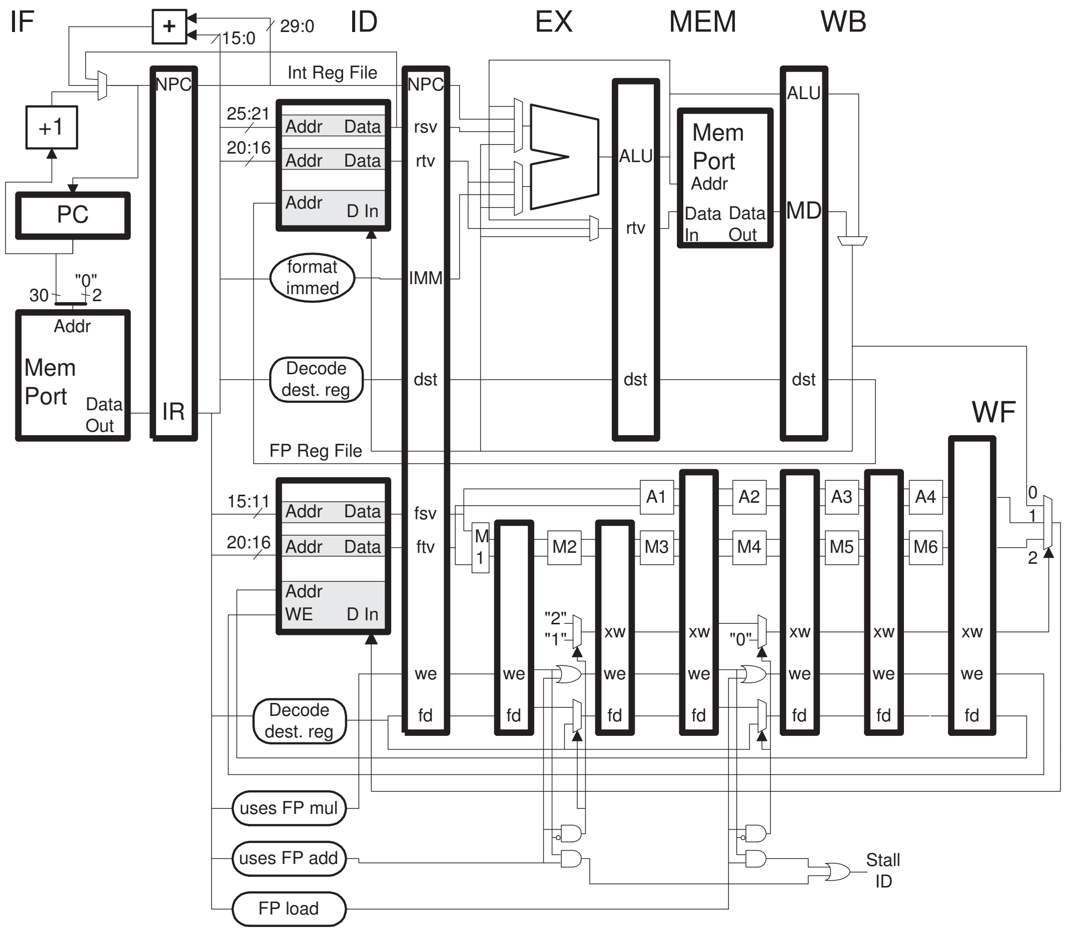
☑ Pipeline diagram of execution. Don't forget to check for dependencies!

```
# SOLUTION
LOOP: # Cycle  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20
 lw r1, 0(r2)  IF ID EX ME WB                                  First Iteration
 lw r3, 0(r1)     IF ID -> EX ME WB
 bne r3, r4          IF -> ID ----> EX ME WB
 lw r2, 8(r1)              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 19 20
 lw r1, 0(r2)                       IF ID -> EX ME WB         Second Iteration
 lw r3, 0(r1)                          IF -> ID -> EX ME WB
 bne r3, r4                                  IF -> ID ----> EX ME WB
 lw r2, 8(r1)                                      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 19 20
 lw r1, 0(r2)        Third Iteration                          IF ID -> EX ME WB
```

☑ CPI for a large number of iterations.

The first iteration starts in cycle 0, the second in cycle 7, the third in cycle 15. The state of the pipeline at the beginning of the second and third iterations is identical: `lw r1` in IF, `lw r2` in ID, and `bne` in EX. Therefore the third iteration will execute identically to the second and the time for the second iteration, $15 - 7 = 8$ cycles, will be the same as the third, etc. The $\boxed{\text{CPI is } \frac{8}{4} = 2}$.

Problem 1, continued:



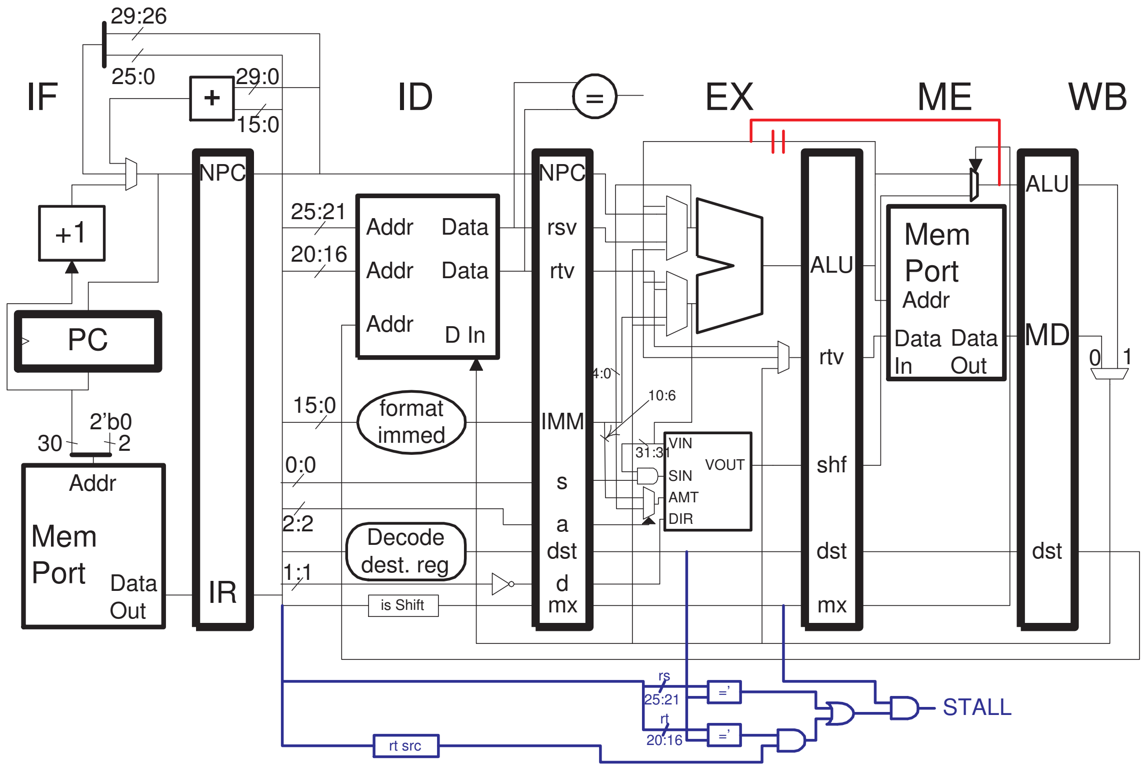
(*b*) [10 pts] Show the execution of the code below on the implementation above.

☑ Pipeline diagram of execution.

```
# SOLUTION
# Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
mul.s f1, f2, f3  IF ID M1 M2 M3 M4 M5 M6 WF
add.s f4, f5, f6     IF ID A1 A2 A3 A4 WF
add.s f7, f8, f9        IF ID -> A1 A2 A3 A4 WF
lwc1 f10, 0(r1)            IF -> ID ----> EX ME WF
```

Problem 2: The MIPS implementation below includes the shift unit (taken from the Homework 3 solution). Notice that it is not possible to bypass a shift result to the EX stage. For that reason the **add** in the code below stalls:

```
# Cycle        0  1  2  3  4  5  6
sll r1, r2, r3 IF ID EX ME WB
add r4, r1, r5    IF ID -> EX ME WB
```



(*a*) [10 pts] Design control logic to generate a signal named `STALL` which will be logic `1` when a stall is needed due to a shift result that cannot be bypassed, as in the example above. (Generate the signal, but don't do anything with it.)

☑ Logic to generate stall signal.

Solution appears above in blue, on the bottom of the diagram. (Ignore for now the changes appearing in red at the top of the diagram.) Consider the state of the pipeline in cycle 2 in the example above, this is the cycle that the pipeline stalls. The two comparison units, $\boxed{='}$, compare the destination of the instruction in EX, `r1` in the example, with the sources of the instruction in ID, `r1` and `r5` in the example. The `rs` register is always compared (which isn't 100% right), but the `rt` register comparison is set to false if the instruction in ID does not use the `rt` register as a source. The stall signal should only be generated if the instruction in EX is a shift, that is the purpose of the last AND gate.

(*b*) [5 pts] The stall above can be avoided by disconnecting the ME-to-EX bypass from the EX/ME.ALU latch and instead connecting it to the output of the ME-stage mux. What was the reason for **not** doing something like that in Homework 3.

☑ Problem with ME-to-EX bypass for shift results.

The connection appears on the top of the diagram, in red. Note that the parallel red lines indicate that the wire is broken. In Homework 3 the shifter was to be added to two different systems, one in which the ALU was not on the critical path, and one in which it was. If the ALU is on the critical path then adding logic between the pipeline latch outputs and the input to the `EX/ME.ALU` latch will, by definition of a critical path, slow things down. The change shown in red adds the ME-stage mux to the ALU critical path, which was the reason for not doing it.

Problem 3: Answer the following questions about interrupts.

(*a*) [7 pts] An important part of an ISA's interrupt mechanism is a separate privileged mode (also called system or supervisor mode) and user mode.

☑ Why is it necessary to have these two modes?

Acceptable answer on test: So that an OS (which runs in privileged mode) can control access to resources such as disk I/O that are requested by user code (in user mode).

Longer description: These modes enable one to set up one group of software that manages certain resources and another group of software that can only get access to the resources by requesting them from the first group. A common example of the first group is the operating system kernel and the second group might be a computer science homework assignment, or your word processing program, etc.

Examples of resources at a higher level are disk (or filesystem) access, memory, and CPU time.

☑ Explain the difference between privileged and user mode in how the CPU operates.

Under privileged mode any valid memory address can be accessed. Under user mode only certain memory address can be accessed, an attempt to access other addresses would cause an exception. Under privileged any instruction can be executed, such as instructions to manage the memory system. Under user mode such instructions cannot be executed.

(*b*) [8 pts] In class our 5-stage implementations resolved exceptions only in ME, and by doing so all integer-pipeline exceptions were precise. *Note: In the original exam the phrase "and by doing so..." was not present.* Suppose instead we resolved exceptions in WB. Show a code fragment in which an exception could not be precise on such a system.

☑ Simple code fragment.

```
# SOLUTION
# Cycle       0   1   2   3   4   5
lw r3, 0(r4)  IF  ID  EX  ME  WB
sw r1, 0(r2)      IF  ID  EX  MEx   Too late to stop store.
```

In the code fragment above, the `lw` raises an exception in ME but because it is not resolved until WB, the store initiated by the next instruction cannot be stopped.

☑ What can't the handler do, and why can't it do it?

The handler cannot see the state of the program as it would be before the `lw` because the `sw` has written memory.

Problem 4: Answer each question below.

(*a*) [8 pts] Consider the distinction between an ISA and its implementation.

☑ What was to be achieved by separating computer design into separate ISA design and implementation design?

Software compatibility between the first implementation and implementations done years later. This avoids the effort needed to port code to a newer design, and the risk of having customers switch vendors.

When designing the ISA separately features are chosen not just based on what can be done with the current implementation, but what would be easy or hard on future implementations.

Grading Note: Surprisingly few answers explicitly mentioned software compatibility.

☑ Provide a reason not to have separate ISA and implementation design, consider the point of view of a conservative computer engineer from the 1960s.

When designing an ISA for longevity one might make decisions that hobble the first implementation. An engineer accustomed to designing implementations for a particular purpose (perhaps considering a few benchmark programs) would consider the need to provide for a large address space or have "unnecessary" instructions wasteful of time and vacuum tubes (or those newfangled transistors).

(*b*) [8 pts] One reason to not compile a program with optimization turned on is because you plan to debug the program. Explain why stepping through a program in a debugger can be confusing when the code has been optimized.

☑ Debugging optimized code is confusing because . . .

. . . when single-stepping through a program execution will not be in the same order as the high-level code specifies. In fact, certain lines of code might not be executed at all and the value of certain variables cannot be printed.

☑ Name a particular type of optimization and explain how it can cause confusing results when single stepping through a program.

Instruction Scheduling: Can cause high-level statements to be executed out of order.

Dead-Code Elimination: A line of dead code will not executed, and a variable that the line may write cannot be printed.

The following answers are wrong in the sense that they would not make debugging difficult. Points were not deducted because they are legitimate optimizations.

Profiling / Code Layout: First profile to determine more likely branch directions, then layout code so that the more likely direction is not taken. This won't by itself make debugging more difficult because it does not change the order in which high level statements are executed. (Location in memory is not the same as execution order.)

Operation Substitution: For example, replacing a multiply by a small constant by a sequence of shifts and adds. Another example is replacing multiplication, division, and modulus operations with power-of-two operands by shifts or ANDs, and then reporting the user to the Bad Programmer Registry. This is wrong because it does not change execution order nor does it affect values of variables. The person using the debugger can't tell whether it was a real multiply (or divide or modulus) or something else. (Unless the person prints the assembly code or is using really slow CPU.)

(*c*) [7 pts] Do you agree or disagree with the statement below regarding the rules for building the SPECcpu benchmarks? Answer with respect to the goals for the SPECcpu benchmarks.

"Testers should not be allowed to use their own compilers because the SPECcpu benchmarks are supposed to test CPUs, not compilers. All testers of a particular ISA should use the same compiler." *Grading note: the phrase "of a particular ISA" was not in the original exam.*

☑ Agree or disagree? Explain.

Note: the key phrase is "are supposed to test CPUs, not compilers."

Disagree. When used properly the compiler will optimize code for a particular implementation, for example, scheduling code to avoid stalls. Certain implementation features may have been designed hand-in-hand with the compiler optimizations needed to fully exploit them. In a sense, the compiler back end is part of the implementation. So to not test the compiler is to not fully test the implementation.

Problem 5: [10 pts] CISC programs generally are smaller than RISC programs.

(*a*) Show how the instruction below is encoded in MIPS and in VAX. For MIPS the name and bit position of each field should be known, and the value of all but one field should be known. For VAX one should know the fields and their sizes, but not every field value needs to be known. *Hint: VAX has 16 general-purpose registers. The size of the two instructions should be the same.*

```
add r1, r2, r3
```

☑ Encoding in MIPS.

```
SOLUTION:
-----------------------------------------------------------------
! opcode    ! rs       ! rt       ! rd       ! sa       ! function  !
! 0         ! 2        ! 3        ! 1        ! 0        ! 0x20      !
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
 3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
```

☑ Encoding in VAX.

```
SOLUTION:
Note: Encoding is for VAX instruction addl3 r1, r2, r3

  !    Opcode     !  !   Source 1    !   !   Source 2    !   ! Destination   !
  !               !  ! Mode  ! Rn    !   ! Mode  ! Rn    !   ! Mode  ! Rn    !
  !    0xc1       !  !   5   ! 2     !   !   5   ! 3     !   !   5   ! 1     !
   7 6 5 4 3 2 1 0    7 6 5 4 3 2 1 0     7 6 5 4 3 2 1 0     7 6 5 4 3 2 1 0
```

(*b*) Identify unused field(s) in the MIPS instruction.

☑ Unused MIPS field.

The **sa** (shift amount) field.

(*c*) Since VAX instruction sizes can vary they should be able to use space more efficiently. Yet even though the MIPS instruction has unused field(s) the two instructions are the same size. Explain why the VAX instruction is larger than one might expect and explain what advantage that provides.

☑ Why is VAX larger than one might expect?

Because **each** operand has an 4-bit field specifying the addressing mode, whereas in MIPS the addressing mode is: part of the opcode, the same for all three operands, and for integer add instructions there are only two choices (the second source is either a register or an immediate).

In more detail: MIPS Opcode: $6 + 6 = 12$ bits. MIPS register numbers $3 \times 5 = 15$ bits. Unused 5 bits (5 more than VAX). VAX Opcode, including operand specifiers: $8 + 3 \times 4 = 20$ bits (8 more than MIPS). Register numbers: $3 \times 4 = 12$ (3 less than MIPS).

☑ What advantage does this larger size provide?

A large variety of addressing modes. That can reduce the total instruction count (compared to MIPS) by eliminating loads, stores, initializing large constants, etc.

Problem 6: Answer each question below.

(*a*) [7 pts] Unlike MIPS, SPARC integer branches use condition codes.

☑ Why might this allow SPARC branch targets to be further away than MIPS branch targets?

Because in MIPS one needs two register fields to indicate the registers to compare, that takes ten bits. In SPARC V8 integer instructions there is just one set of condition codes and so zero bits are needed for them (and in V9 there are just two, taking 1 bit). The space saved for the register numbers in SPARC can be used for a larger displacement.

☑ Why might this enable certain SPARC implementations to have a higher clock frequency than comparable MIPS implementations?

Because in MIPS I one needs to compare up to two 32-bit registers. For resolve-in-ID implementations (such as the one used in class) the register values are not available until partway through the cycle, likely forcing the branch direction logic to be on the critical path. In SPARC one just needs to look at a single 4-bit condition code register, and so the logic would be faster.