# CS/ECE 552: Introduction to Computer Architecture

**Prof. David A. Wood**

**Midterm Exam**
**March 9, 2010**
**7:15-9:15pm, B371 Chemistry**
**Approximate Weight: 25%**

**CLOSED BOOK**
**ONE SHEET OF NOTES**

**NAME: _____Solution______**

**DO NOT OPEN THE EXAM UNTIL TOLD TO DO SO!**

Read over the entire exam before beginning. Verify that your exam includes all 8 pages. It is a long exam, so use your time carefully. Budget your time according to the weight of the questions, and your ability to answer them. Limit your answers to the space provided, if possible. If not, write on the BACK OF THE SAME SHEET. Use the back of the sheet for scratch work. WRITE YOUR NAME ON EACH SHEET.

| Problem | Possible Points | Points |
|---|---|---|
| Problem 1 | 15 | |
| Problem 2 | 15 | |
| Problem 3 | 20 | |
| Problem 4 | 25 | |
| Problem 5 | 25 | |
| Total | 100 | |

NAME: ____________________________________

## Problem 1: (15 points)

### Part A: (3 points)

What is the *iron law of performance*?

Time/Program = Instructions/Program * Cycles/Instruction * Time/Cycle

### Part B: (3 points)

An out-of-order pipelined processor fetches and dispatches instructions in order, executes them (potentially) out of program order, then re-orders and commits (aka retires) them in order. Why does it reorder them?

Reordering instructions before commit allows the processor to enforce precise exceptions (i.e., sequential semantics). Precise exceptions requires that all instructions before the excepting instruction must complete, all instructions after it must appear to have never begun (i.e., flushed from the pipeline without modifying architectural state), and the address of the excepting instruction must be passed to the exception handler. Commiting instructions in order makes this easy to do.

### Part C: (3 points)

The VAX instruction set has very large instructions (up to 61 bytes). Explain why the VAX is *not* considered a *VLIW (Very Large Instruction Word)* instruction set architecture?

A VLIW instruction combines a group of potentially different operations into a single instruction. For example, the IA-64 architecture combines three independent instructions into a 128-bit instruction bundle. Each instruction can operate on different data (e.g., different registers) and specify different operations (e.g., add or sub). (Most) VAX instructions perform a single (perhaps complex) operation on one set of operands.

### Part D: (3 points)

Explain the difference between a dependence and a hazard.

A dependence is a property of the instructions in a program, for example a true dependence arises when one instruction uses the value produced by an earlier instruction.

A hazard is a potential problem in a pipeline that may arise from a dependence. For example, a true data dependence causes a hazard when the value produced by the earlier instruction is not yet available in the register file when the dependent instruction attempts to fetch it.

### Part E: (3 points)

The MIPS instruction set has fixed size instructions with only three instruction formats with key fields always occuring in the same place. Explain why these two properties make it easier to implement a pipelined processor. Give brief examples.

Fixed size instructions means that it is, in the absence of a control hazard, possible to determine the next instruction before decoding the current instruction. This allows fetch of instruction i+1 to proceed in parallel with the decode of instruction i.

Fixed field placement makes it easier to decode instructions in parallel and perhaps partially overlap execution. For example, in MIPS it is possible to read the register file in parallel with instruction decode because the source register specifiers are always in the same place.

NAME: ________________________________________

## Problem 2: (15 points)

### Part A: (5 points)

Indicate the *true data dependences* in the following MIPS code sequence:

```
add   $4, $3, $1
sw    $4, 0($2)
add   $4, $4, $1
lw    $1, 0($4)
add   $4, $3, $1
```

### Part B: (5 points)

What is meant by a *name dependence*? Are there any examples of name dependences in the code above?

A name dependence can arise between two instructions that use the same register (or memory location) to hold different values. Output dependences and anti-dependences are both name dependences. An output dependence occurs between two instructions that write different values to the same register (or memory location). An anti-dependence occurs between an instruction that reads a register (or memory location) and a subsequent instruction writes a new value to the same location.

Output dependences exist between the three add instructions, which all write to register $4

An anti-dependence exists between the lw instruction, which reads $4, and the last add instruction, which overwrites $4 with a new value.

### Part C: (5 points)

What problems, if any, do name dependences cause in the MIPS 5-stage pipeline that we have analyzed in class? Explain.

Name dependences do not cause any problems in the MIPS 5-stage pipeline.

Output dependences are not a problem because all instructions write to the register file (and memory) in program order.

Anti-dependences are not a problem because all instructions execute in order and always read the register file before they write it (i.e., read in Decode and write in Writeback), thus preventing a later instructions write from occuring before an earlier instructions read.

NAME: ______________________________

## Problem 3: (20 points)

Consider two implementations A and B of the MIPS instruction set, both built using the same technology, but using different pipelines. Both machines have a base CPI of 1.0, but have different cycle times and different stalls for control and data hazards. In particular, the pipelines stall differently for taken and not-taken branches, when loads are followed by dependent instructions, and Machine B stalls a cycle on all stores.

| | Machine A | Machine B |
|---|---|---|
| Cycle time | 500ps | 300ps |
| Taken branch stalls | 1 | 5 |
| Not-taken branch stalls | 0 | 1 |
| Load-use stalls | 1 | 3 |
| Store stalls | 0 | 1 |

**Part A: (12 points)**

For the two workloads below, assume that 65% of branches are taken and 40% of loads are followed by a dependent instruction.

| Workload | % Branches | % Loads | % Stores | % Other |
|---|---|---|---|---|
| W1 | 10% | 30% | 15% | 45% |
| W2 | 20% | 20% | 10% | 50% |

Compute the SCPIs and overall CPI for both datapaths.

| | Machine A | | Machine B | |
|---|---|---|---|---|
| | WI | W2 | W1 | W2 |
| $SCPI_{branch-taken}$ | .065 | .13 | .325 | .65 |
| $SCPI_{branch-nottaken}$ | 0 | 0 | .035 | .07 |
| $SCPI_{load-use}$ | .12 | .08 | .36 | .24 |
| $SCPI_{stores}$ | 0 | 0 | .15 | .1 |
| CPI | 1.185 | 1.21 | 1.87 | 2.06 |

**Part B: (4 points)**

Which machine is faster? Compute the Speedup of Machine B over Machine A (i.e., Machine A is the "old" machine). Show your work.

| Workload | Speedup of B | Faster machine? |
|---|---|---|
| W1 | 1.06 | B |
| W2 | 0.98 | A |

$Speedup_B = Time_A / Time_B = (N \times CPI_A \times 500ps) / (N \times CPI_B \times 300ps)$

W1: 1.185 * 500 / 1.87 * 300 = 1.06

W2: 1.21 * 500 / 2.06 * 300 = 0.98

**Part C: (4 points)**

The slower machine would perform better with a faster clock. How fast would the slower machine's clock need to be to have the same performance as the faster machine? Show your work.

| Workload | Slower Machine | Clock cycle time to achieve equal performance |
|---|---|---|
| W1 | A | 473ps |
| W2 | B | 294ps |

$CPI_{slow} * Cycle_{slow} = CPI_{fast} * Cycle_{fast}$

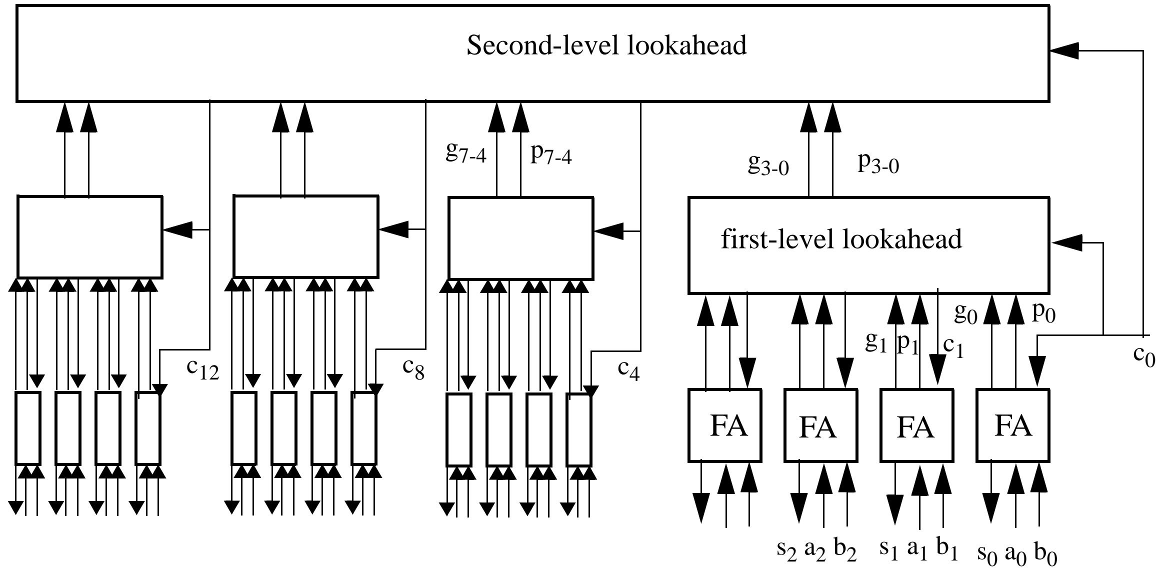
W1: 1.185 * C = 1.87 * 300

W2: 1.21 * 500 = 2.06 * C

NAME: ______________________________

## Problem 4: (25 points)

A 16-bit carry-lookahead adder composes multiple 4-bit carry-lookahead blocks into a two level tree structure.

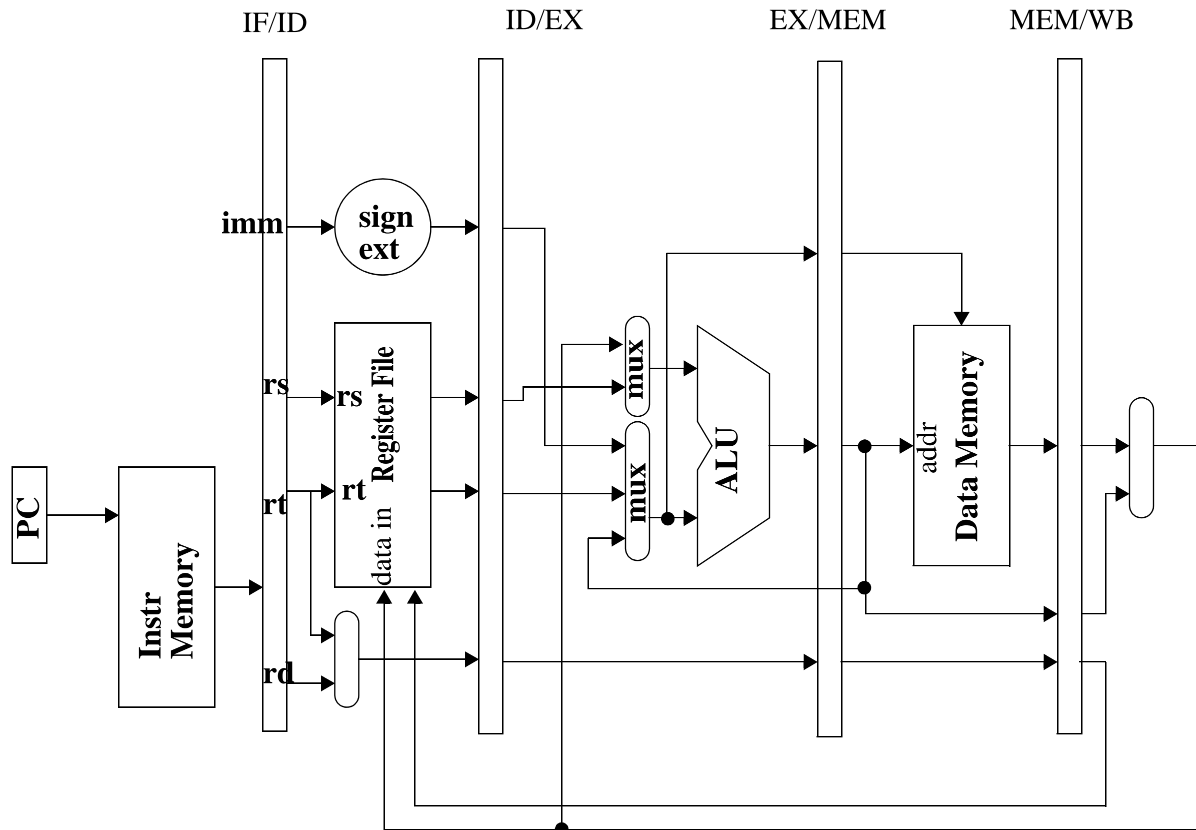



Write the boolean equation for each output signal listed in the table below. The equations should be optimized to minimize the delay from module inputs to outputs, where the modules are the full adder (FA), and the first- and second-level lookahead blocks. Compute the delays using the model below. The *worst case module delay* is the critical path from *any* input of a module to the output. The *critical path delay* is the critical path from the basic inputs $a_i$, $b_i$ and $c_0$, which are assumed to change at time 0. Assume that you have only AND and OR gates available, but that each gate generates both the true output f and its complement $\overline{f}$. You also have the complements of the basic inputs available as well. The delay is computed using the formula *delay = (5 + 4n)*τ, where n is the number of inputs to the gate. Thus a 2-input AND gate has delay 13τ and the logic function $f = ab + cde$ has delay 30τ (2-input OR with delay 13τ plus a 3-input AND with delay 17τ).

| Signal | Equation | Worst case module delay | Critical path delay |
|---|---|---|---|
| $p_2 =$ | $a_2 + b_2$ | 13τ | 13τ |
| $g_2 =$ | $a_2 b_2$ | 13τ | 13τ |
| $c_4 =$ | $g_{3\text{-}0} + p_{3\text{-}0}c_0$ | 26τ | 68τ |
| $c_8 =$ | $g_{7\text{-}4} + p_{7\text{-}4}g_{3\text{-}0} + p_{7\text{-}4}p_{3\text{-}0}c_0$ | 34τ | 85τ |
| $g_{11\text{-}8} =$ | $g_{11} + p_{11}g_{10} + p_{11}p_{10}g_9 + p_{11}p_{10}p_9g_8$ | 42τ | 55τ |
| $p_{11\text{-}8} =$ | $p_{11}p_{10}p_9p_8$ | 21τ | 34τ |
| $c_{11} =$ | $g_{10} + p_{10}g_9 + p_{10}p_9g_8 + p_{10}p_9p_8c_8$ | 42τ | 127τ |
| $s_{11} =$ | $(a_{11}\overline{b}_{11} + \overline{a}_{11}b_{11})\overline{c}_{11} + (a_{11}b_{11} + \overline{a}_{11}\overline{b}_{11})c_{11}$ | 52τ | 153τ |
| $c_{12} =$ | $g_{11\text{-}8} + p_{11\text{-}8}g_{7\text{-}4} + p_{11\text{-}8}p_{7\text{-}4}g_{3\text{-}0} + p_{11\text{-}8}p_{7\text{-}4}p_{3\text{-}0}c_0$ | 42τ | 93τ |
| $c_{15} =$ | $g_{14} + p_{14}g_{13} + p_{14}p_{13}g_{12} + p_{14}p_{13}p_{12}c_{12}$ | 42τ | 135τ |
| $s_{15} =$ | $(a_{15}\overline{b}_{15} + \overline{a}_{15}b_{15})\overline{c}_{15} + (a_{15}b_{15} + \overline{a}_{15}\overline{b}_{15})c_{15}$ | 52τ | 161τ |

NAME: ____________________________________

**Problem 5: (25 points)**




High performance datapaths use bypass paths (also known as data forwarding logic) to reduce pipeline stalls. However, bypass paths are relatively expensive, especially in some wire constrained technologies. To reduce the cost (and potential cycle time impact), some architects have explored omitting some of the possible bypass paths. Consider the datapath illustrated above (note that the PC update logic and all control logic is intentionally omitted). This pipelined datapath is similar to the one in the book, *but has several differences including limited bypass paths*. BE SURE TO STUDY THE DATAPATH CAREFULLY! Assume that the register file internally bypasses the value, so that if register $i is read and written in the same cycle, then the read returns the new value. Assume that the control logic bypasses the data as soon as possible using the given forwarding data paths, and stalls in decode otherwise. You may NOT add additional data paths.

In this problem, you will look at how a program snippet performs on this pipeline. Recall that R-format instructions have the form:

```
opcode rd, rs, rt
```

and I-format instructions have the form

```
opcode rt, imm(rs)
```

or

```
opcode rt, rs, imm
```

Use the table on the next page to show how the given instruction sequence flows through the pipeline and where stalls are necessary to resolve hazards.

NAME: ____________________________________

Consider the code and pipeline schedule below. Show the execution timing of this code on the pipeline above.

| | Cycle | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Instructions | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| add $1, $2, $3 | F | D | X | M | W | | | | | | | | | | | | | | | |
| sub $4, $1, $5 | | F | ~~D~~ | D | X | M | W | | | | | | | | | | | | | |
| or $6, $1, $4 | | | ~~F~~ | F | D | X | M | W | | | | | | | | | | | | |
| and $7, $4, $8 | | | | | F | D | X | M | W | | | | | | | | | | | |
| lw $9, 4($7) | | | | | | F | ~~D~~ | D | X | M | W | | | | | | | | | |
| add $1, $9, $2 | | | | | | | ~~F~~ | F | ~~D~~ | D | X | M | W | | | | | | | |
| sw $1, 4($7) | | | | | | | | | ~~F~~ | F | D | X | M | W | | | | | | |

For each cycle where a stall occurs, explain why below.

Cycle 3: Register $1 in the ‘sub’ instruction is dependent on the preceding ‘add’ instruction. Because $1 is the ‘rs’ register, it cannot forward from the XM latch (labelled EX/MEM in this figure). Instead, it must stall in decode, which also stalls the fetch of the ‘or’ instruction. Because ‘rs’ can be forwarded from the MW latche (MEM/WB), the stall is only a single cycle.

Cycle 7: Register $7 in the ‘lw’ instruction uses the value produced by the ‘and’ instruction. Because $7 is the ‘rs’ register it stalls for one cycle as above.

Cycle 9: Register $9 in the second ‘add’ instruction depends upon the value produced by the ‘lw’ instruction. Loads don’t produce their value until the end of the M (MEM) stage, requiring a load-use stall for the ‘add’ in this cycle. This also stalls the ‘sw’ instruction in fetch.

Cycle 13 & 14: The ‘sw’ instruction cannot be implemented correctly with this datapath. The ‘rt’ mux needs to be set one way to calculate the address and a different way to get the register $1 to memory. To obtain full credit, you needed to identify this problem with the datapath.