Midterm I
October 18, 2000
CS252 Graduate Computer Architecture

<table>
<thead>
<tr>
<th>Problem</th>
<th>Possible</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>
[ This page left for $\pi$ ]

3.141592653589793238462643383279502884197169399375105820974944
Question #1: Short Answer

1a) What are precise exceptions and why are they desirable? Give 3 reasons.

1b) What hardware structure can be used to support branch prediction, data prediction, and precise exceptions in an out-of-order processor? Explain what this structure is (including what information it holds) and how it is used with implicit register renaming to recover from a bad prediction or exception.

1c) Do you need the structure of (1b) in order to achieve precise exceptions in an in-order pipeline? Why or why not? Explain.
1d) Given that enough transistors were available, would it make sense to design a 64-way superscalar processor? Give 3 reasons why or why not.

1e) Name two components of a modern superscalar architecture whose delay scales quadratically with the square of the issue-width. Could you pinpoint a single root cause for these quadratic trends?

1f) You read a paper on Simultaneous Multithreading. Can you give a short description on the basic idea and why it is desirable? What metric is a Simultaneous Multithreading processor attempting to optimize over a basic out-of-order processor?
1g) What is memory disambiguation? Describe the minimal hardware support required to perform conservative memory disambiguation in an out-of-order processor (which means that you never send a load to the memory system unless you know that you should) and list pseudo-code that is followed when dispatching loads and/or stores.

1h) Why might we want to do a bit of guessing about load dependencies instead of adhering to the strict algorithm that you gave in (1g)?

1i) Why does prediction work?

1j) What is the problem with aliasing in branch predictors? Draw and name one of the simplest branch predictors that helps to avoid aliasing and describe why it is less sensitive to aliasing than other predictors.
Problem #2: Two-way superscalar processors

Consider a dual-issue, in-order pipeline with one fetch stage, one decode stage, multiple execution stages (which include memory access) and a single write-back stage. Assume that the execution stages are organized into two parallel execution pipelines (call them even and odd) that support all possible simultaneous combinations of two instructions. Instructions wait in the decode stage until all of their dependencies have been satisfied. Further, since this is an in-order pipeline, new instructions will be forced to wait behind stalled instructions.

On each cycle, the decode stage takes zero, one, or two ready instructions from the fetch stage, gathers operands from the register file or the forwarding network, then dispatch them to execution stages. If less than 2 instructions are dispatched on a particular cycle, then “NOPs” are sent to the execution stages. When two instructions are dispatched, the even pipeline receives the earlier instruction. When only one instruction is dispatched, it is placed in the even pipeline.

Assume that each of the execution pipelines consist of a single linear sequence of stages in which later stages serve as no-ops for shorter operations (or: every instruction takes the same number of stages to “execute”, but results of shorter operations are available for forwarding sooner). All operations are fully pipelined and results are forwarded as soon as they are complete. Assume that the execution pipelines have the following execution latencies: addf (2 cycles), multf (3 cycles), divf (4 cycles), integer ops (1 cycle). Assume that memory instructions take 3 cycles of execution: one for address calculation – done by the integer execution stage, and two unbreakable cycles for the actual memory access. Finally, assume that branch-conditions are computed by integer execution units.

2a) Explain why we would be unable to pick a single optimum number of branch delay slots for the above processor.

2b) Can we tell the programmer that the number of branch delay slots varies by circumstances? If so, explain the programmer specification for branches. If not, explain why not and (1) indicate how we would “fix” the hardware to have only a specific number of branch delay slots and (2) indicate what that number would be.
2c) Does this processor have WAW hazards? Explain. If “yes”, give an efficient way to fix the problem.

2d) Does this processor have WAR hazards? Explain. If “yes”, give an efficient way to fix the problem.

2e) Assume that the fetch unit presents instructions to the decode stage for execution. The decode stage is free to dispatch zero, one, or two instructions every cycle. *Once instructions have passed decode, they execute to completion (no further blocking).* Assume that enough bypassing hardware has been included to handle every arrow given in (2g).

Suppose that we have the following instruction sequence:

```
  ld   r1, 0(r2)
  add r4, r1, r2
```

How many cycles must be inserted between these two instructions by the decode stage to ensure correct execution? How does this translate to user-visible load-delay slots? Explain.

2f) Suppose that we have the following instruction sequence:

```
  mult f1, f2, f3
  st   0(r1), f1
```

How many cycles will be inserted between these two instructions by the decode stage? How many lost instructions does this represent?
2g) Draw a simple diagram for the pipelines of this processor. Draw pipeline stages as boxes with letters inside: Use “F” for the fetch stage, “D” for the decode stage, EX₁ through EX₄ for the execution stages of each of the pipelines (including memory accesses), and “W” for writeback. Draw simple forward arrows to indicate the flow of information from stage to stage. Clearly label which pipeline is the even pipeline. Finally, describe what is computed in each stage and show all of the bypass paths (as arrows). Your goal is to design a pipeline that never stalls unless a value is not ready. Label each of the bypass arrows with the types of instructions that will forward their results along that path (i.e. use “M” for multf, “D” for divf, “A” for addf, “I” for integer operations, and “Ld” for load results). [Hint: be careful about inputs to store instructions!]
2h) Briefly describe the logic that would be required in the decode stage of this pipeline. In five (5) sentences or less (and possibly a small figure), describe a mechanism that would permit the decode stage to decide which of two instructions presented to it could be dispatched.

2i) Suppose that we use a Tomasulo architecture to schedule this pipeline instead of an in-order dispatch unit. What are the minimal changes that need to be made to a Tomasulo architecture in order to support a sustained throughput of two instructions per cycle? Describe and/or draw these changes.
Problem #3: Software Scheduling

For this problem, assume that we have fully pipelined, single-issue, in-order processor with the following number of execution cycles for:

1. Floating-point multiply: 4 cycles
2. Floating-point adder: 2 cycles
3. Integer operations: 1 cycle

Assume that there is one branch delay slot, that there is no delay between integer operations and dependent branch instructions, and that the load-use latency is 2 cycles. Assume that all functional units are fully pipelined and bypassed.

The following code computes a portion of a filter operation. Assume that r1 contains a pointer to the beginning of a window of floating-point numbers. Further, r2 contains an array of floating-point constants for the filter. Let r3 be the size of the window, f2 be a scaling constant. Finally, assume that f5 is initialized to a bias constant outside the loop.

```
filter:  ldf  F3,0(r1)
multf F10,F3,F2
ldf  F4,0(r2)
multf F11,F10,F4
addf F5, F5, F11
addi r1,r1,#8
addi r2,r2,#8
subi r3,r3,#1
bneq r3,filter
      nop
```

3a) How many cycles does this loop take per iteration? Indicate stalls in the above code by labeling each of them with a number of cycles of stall:

3b) Reschedule this code to run with as few cycles per iteration as possible. Do not unroll it or software pipeline it. How many cycles do you get per iteration of the loop now?
3c) Unroll the loop once and schedule it to run with as few cycles as possible. *Ignore startup code.* What is the average number of cycles per iteration of the original loop?

3d) Software pipeline this loop to avoid stalls. Use as few instructions as possible. Your code should have no more than one copy of the original instructions. What is the average number of cycles per iteration? *Ignore startup and exit code (just show the interior of the loop):*
[ This page intentionally left blank!]
3e) For the software-pipelined version of the loop, what is the maximum execution latency for \texttt{multf} and \texttt{addf}, and maximum use-latency for \texttt{ld} such that the loop runs without stalls? Explain.

3f) Now, add startup and exit code to your answer for (3d) such that your code performs exactly the same computation as the original loop (i.e. in 3a), but without stalls. Write out the complete filter function. Assume that there is enough space on either side of the arrays off of \texttt{r1} and \texttt{r2} such that accesses beyond the bounds of the arrays do not cause memory faults. Also assume that \texttt{multf} instructions don’t cause exceptions. You should be able to do this with a small number of instructions. [Serious hint: Rather than trying to “startup” the software pipeline, try setting up registers to nullify the effects of partial iterations on entrance to your code of 3d, and ignore partial results on exit]:

3g) Suppose that you can design a VLIW architecture to perform this filter function. What is the minimal mix of functional units that you would need to get one cycle per iteration? Assume that you will use a combination of software pipelining and loop unrolling. You may choose from load/store units, floating-point multipliers, floating-point adders, and integer units (which also do branches). EXPLAIN. Hint: if you are spending a long time on this, you are probably working too hard!