University of California, Berkeley
College of Engineering
Computer Science Division — EECS

Spring 2003 John Kubiatowicz

Midterm II
SOLUTIONS
May 7th, 2003
CS152 Computer Architecture and Engineering

Your Name:	
SID Number:	
Discussion Section:	

Problem	**Possible**	**Score**
1	20	
2	25	
3	30	
4	25	
Total		

[This page left for π]

3.14159265358979323846264338327950288419716939937510582097494 4

Problem 1: Short Answers

Problem 1a[2pts]: Give a simple definition of precise interrupts/exceptions. Why is this important?

A precise interrupt or exception leaves the machine in a state for which there is a single instruction (PC) such that all instructions before that PC have committed their state and the instruction at the specified PC and all following instructions have not committed their state.

This is important because it makes restarting the user program very simple.

Problem 1b[3pts]: Explain why exceptions can occur out of order (in time) in an in-order, 5-stage pipeline. Give an example and explain how to achieve a precise exception point anyway.

Exceptions can occur out of order because they can occur in different pipeline stages. For example, you could have a page-fault in the MEM stage of an early instruction after an illegal instruction fault in the DECODE stage of another:

PC:	*F*	*D*	*E*	***M***	*W*		*← Fault in mem stage*
PC+4:		*F*	***D***	*E*	*M*	*W*	*← Fault in decode stage (earlier)*

Precise exceptions can be recovered by waiting until the M/W boundary to declare a fault.

Problem 1c[3pts]: Name and *define* 3 types of pipeline data hazards. For each hazard, explain how it is prevented in the 5-stage pipeline:

RAW: Read after Write hazard. This occurs if a later instruction reads a register from an earlier instruction while it is still in the pipeline. Fixed by forwarding.

WAR: Write after read hazard. This occurs if a later instruction writes a register read by an earlier instruction before that instruction gets the old value. Fixed by the fact that the 5-stage pipeline executes in order and reads values early and writes them late.

WAW: Write after Write hazard. This occurs if a later instruction writes a register written by an earlier instruction – and the result of the earlier instruction persists. Fixed by the fact that the 5-stage pipeline commits instructions in order.

Problem 1d[2pts]: How do you refresh a DRAM, and *why does this work (i.e. what is happening internally)?*

You refresh a DRAM by reading each ROW once every so often. This works because the DRAM reading process is restoring (the SENSE amps reinforce the values stored in the cells).

Problem 1e[2pts]: What are load-delay slots? Does the programmer need to know about them (explain carefully):

There are two options here. In non-stalling pipelines, the load-delay slots are the set of instructions following a load which cannot use the value of the load (because they will get the wrong value). In stalling pipelines, the load delay slots are the set of instructions following a load which will stall if they use the value of the load. In both cases the programmer needs to know about them because they will either affect correctness or performance.

Problem 1f[2pts]: Why is it important for Tomasulo to issue instructions in-order?

The Tomasulo algorithm must issue instructions in-order so that the dataflow is properly evaluated.

Problem 1g[2pts]: What is a victim cache? What is it good for?

A victim cache is a small caching data structure below a regular cache that holds values after they are kicked out of the cache (i.e. it holds on to victims). This cache is typically fully-associative and thus helps to reduce the effects of conflict misses.

Problem 1h[2pts]: Name and describe the structure that permits an out-of-order processor to achieve precise interrupts. How does this work?

This is the Reorder Buffer (ROB). It is a FIFO buffer used to commit values to registers and memory in instruction order. During issue, an out-of-order processor enters instructions into the ROB in program order. Completed instructions place their results in the ROB rather than the register file. Values are written back to the register file and memory when they exit the bottom of the buffer (i.e. values are written in program order). As a result, the machine can achieve a precise interrupt or exception point by throwing out all values in the ROB and using the PC of the oldest, non-committed instruction in the ROB as the exception point.

Problem 1i[2pts]: Suppose you have a processor with a 4K page size, 16K first-level data cache with 128-bit cache lines. Assume that you want to overlap TLB lookup with cache lookup. What is the required associativity of the first-level cache?

Since the index (and cache-line offset) must fit entirely in the page, we can only do this if we have an associativity of 16K/4K = 4.

[This page intentionally left blank]

Problem 2: Memory Hierarchy

Problem 2a[2pts]: Assume that we have a byte-addressed 64-bit processor with 64-bit words. Suppose that this processor has a 48-word, three-way, set-associative cache (LRU replacement) with 2-word cache lines. Split the 64-bit address into "tag", "index", and "cache-line offset" pieces. Which address bits comprise each piece (one is given)?

tag: *bits 63 – 7*
index: *bits 6 – 4*
cache-line offset: **bits 3 – 0 (Given)**

Problem 2b[2pts]: How many sets does this cache have? Explain.

There are 8 sets in this cache (size of 2^{index}).

Problem 2c[7pts]: Assume that the processor makes the following byte accesses. Label each reference address as a Hit (H) or a Miss (M). Also, identify each cache miss as a **compulsory, conflict,** or **capacity** miss.

Scratch (Tag|index|offset)	Byte Address	Hit/Miss?	Miss Type
0000|010|0110	38(0x026)	Miss	Compulsory
0001|010|1100	172(0x0AC)	*Miss*	*Compulsory*
0001|001|0000	144(0x090)	*Miss*	*Compulsory*
0000|101|0101	85(0x055)	*Miss*	*Compulsory*
0011|010|1000	424(0x1A8)	*Miss*	*Compulsory*
0000|110|1111	111(0x06F)	*Miss*	*Compulsory*
0001|010|1110	174(0x0AE)	*Hit*	—
0100|010|0111	551(0x227)	*Miss*	*Compulsory*
0000|101|1010	90(0x05A)	*Hit*	—
0000|010|0000	32(0x020)	*Miss*	*Conflict*
0011|010|1100	428(0x1AC)	*Miss*	*Conflict*
0100|010|0000	544(0x220)	*Hit*	—
0000|110|0000	96(0x060)	*Hit*	—
0011|010|0110	422(0x1A6)	*Hit*	—
0001|010|1010	170(0x0AA)	*Miss*	*Conflict*

Problem 2d[2pts]: Calculate the cache hit rate (you can leave as a fraction).

Cache Hit rate = 5/15

[This page intentionally left blank]

Problem 2e[7pts]: You have a 500 MHz processor with 2-levels of cache, 1 level of DRAM, and a DISK for virtual memory. Assume that it has a unified first-level cache. Assume that the memory system has the following parameters:

Component	Hit Time	Miss Rate	Block Size
First-Level Cache	1 cycle	5% Data 1% Instructions	64 bytes
Second-Level Cache	20 cycles + 1 cycle/64bits	2%	128 bytes
DRAM	50ns+ 25ns/8 bytes	0.1% (Page Fault)	16K bytes (Page Size)

Disk Parameters: 50 Mbytes/sec transfer, 10ms average seek, 6000 RPM, 5ms controller time. Finally, assume that there is a TLB that misses 0.1% of the time on data (doesn't miss on instructions) and which has a fill penalty of 100 cycles. What is the average memory access time (AMAT) for Instructions? For Data (assume all reads)? *[1M = 1K × 1K = 1024×1024]*

$AMAT_{disk}$ = 5ms+10ms+[(½×60SPM/6000RPM)+16Kbytes/(50Mbytes/S×1024K/M)]×1000ms/S
= 15ms+5.3125ms=20.3125ms⇒20.3125ms×(5×10^5cycles/ms)=1.016×10^7 cycles
$AMAT_{dram}$ = (50ns+25ns×16)×0.5cycles/ns + 0.001×$AMAT_{disk}$ = 1.0385×10^4cycles
$AMAT_{2lev}$ = (20+8)cycles + 0.02×$AMAT_{dram}$ = 235.7cycles

$AMAT_{1lev}^{Inst}$ = 1 + 0.01×$AMAT_{2lev}$=1+2.357 = 3.357 cycles

$AMAT_{1lev}^{Data}$ = 1 + 1 + 0.05×$AMAT_{2lev}$+0.001×100 = 13.885cycles

Problem 2f[5pts]: Suppose that we measure the following instruction mix for benchmark "X":
Loads: 20%, Stores: 10%, Integer: 30%, Floating-Point: 20% Branches: 20%
Assume we have a single-issue processor with minimum CPI of 1.0. Assume we have a branch predictor that is correct 90% of the time, and that an incorrect prediction costs 3 cycles. Finally, assume that data hazards cause an average penalty of 2 cycles for floating point operations. Integer operations run at maximum throughput. What is the average CPI of Benchmark X, including memory misses (from part a)? *[hint: don't forget structural memory hazards]*

CPI = 1 + 0.1×0.2×3 + 0.2×2+ 0.3×(13.885-1) + 1×(3.357-1) ≈ 7.68cycles

Note that we have to subtract off an extra "1" from the $AMAT_{1lev}^{Inst}$ *and* $AMAT_{1lev}^{data}$ *to convert to stall penalties.*

Problem #3: 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.

Problem 3a[2pts]: Explain why we would be unable to pick a single optimum number of branch delay slots for the above processor.

Branch delay slots affect correctness (they represent functional behavior – things always *executed when a branch is executed), we have to pick a single number. The result wouldn't be optimal under all circumstances, since we issue 0, 1, or 2 instructions per cycle after the branch.*

Problem 3b[3pts]: 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.

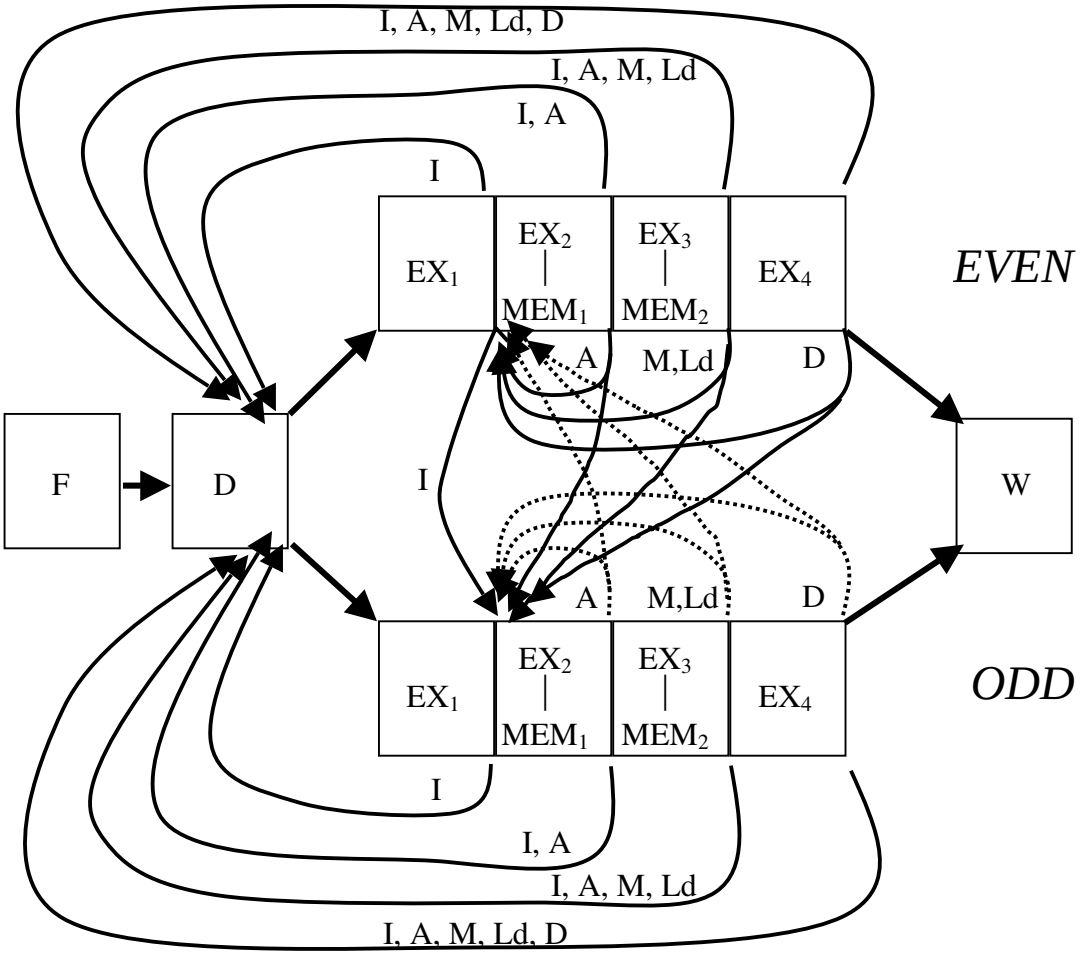
No. This would be very complicated and highly timing dependent (especially when cache misses are taken into account). Perhaps we could derive a couple of rules that a compiler could follow. Better to (1) pick a number, say one *delay slot and (2) fix the slack with branch prediction.*

Note that our pipeline might have everywhere from 0 to 5 instructions issued before the branch condition is resolved in the execute stage (maximal comes from 2 cycles x 2 insts + 1 for branch in even slot). We could pick some number of delay slots, issue these and stall afterwards until the condition is resolved (although we may waste a lot of cycles). The complexity here arises because of the variation (say we pick 4 delay slots – we may need to issue these in 2 to 4 cycles!). Worse, if we have too many delay slots, the compiler will have trouble finding useful instructions (even 2 delays slots are hard to fill).

Problem 3c[15pts]: Below is a ***start*** at a simple diagram for the pipelines of this processor. **a)[3pts]** Finish the diagram. Stages are boxes with letters inside: Use "F" for a fetch stage, "D" for a decode stage, EX_1 through EX_4 for the execution stages of each of the pipelines (including memory accesses), and "W" for a writeback stage. Clearly label which is the *even* pipeline. Include arrows for forward information flow if this is not obvious.

b)[2pts] Next, describe what is being computed in each EX stage (including partial results).

c)[10pts] Show all bypass paths (as arrows*). Your pipeline should never stall unless a value is not ready.* Label each bypass arrow 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: think carefully about inputs to store instructions!]*



EX Stages: EX_1: Integer ops, Branches, Memory address computation, First stage of A, M, D
EX_2: First stage of load/store, Finish A, Second stage of M, D
EX_3: Final stage of load/store, Finish M. Third stage of D
EX_4: Final stage of D

Notes: The primary forwarding is from the end of EX stages back to the end of decode stage. Store forwarding is shown between the two pipes and only involves special cases in which an operation finishes and needs to be forwarded into the beginning of EX_2. Note in particular the very *special case of integer forwarding from an integer op in even pipeline to store in odd. With this arc, you can actually issue a integer op and a store of the result in the same cycle.*

Problem 3d[2pts]: Does this processor have WAW hazards? Explain. If "yes", give an efficient way to fix the problem.

Yes. Two instructions that write the same destination register can be issued in same cycle. This can be fixed by giving precedence (final say) to the odd pipeline. The same issue could happy with stores to the same address in the same cycle, but we didn't look for this.

Problem 3e[2pts]: Does this processor have WAR hazards? Explain. If "yes", give an efficient way to fix the problem.

No, this processor doesn't have WAR hazards with the register file because values are read early and written late in the pipeline (and do not get out of order).

*Note that there **may** be a WAR hazard through memory if a store in the EX_2 stage could impact a load in the EX_3 stage. This gets fixed by making sure that stores do not become visible to other instructions in the pipeline until EX_3. We weren't really looking for this but gave credit for it.*

Problem 3f[3pts]: 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 (3c).

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.

*There needs to be 2 cycles between the **ld** and **add**. Load-delay slots represent instruction locations in which either the pipeline will stall or give incorrect results to users of the load result. Thus, in the worst case, these 2 cycles will represent either 4 or 5 delay-slot instructions (depending on whether or not the **ld** is in the odd or even pipeline).*

Problem 3g[3pts]: Suppose that we have the following instruction sequence:

```
multf 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?

*There needs to be 1 cycle between **multf** and **st** (this is one of the special arcs in 3c). As above, this represents a potential (in the worst case) loss of either 2 or 3 instructions.*

[This page intentionally left blank!]

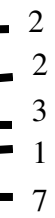
Problem #4: Fixing the loops

Assume that we have a superpipelined architecture with the following use latencies:

Between a **multf** and an **addf:**	**3 insts**	Between a **load** and a **multf:**	**2 insts**
Between an **addf** and a **divf:**	**1 insts**	Between a **divf** and a **store:**	**7 insts**
Between an int op and a **store:**	**0 insts**	Between two integer ops:	**0 insts**
Number of branch delay slots:	**1 insts**		

Consider the following loop which performs a restricted rotation and projection operation. In this code, F0 and F1 contain **sin(θ)** and **cos(θ)** for rotation. The array based at register **r1** contains pairs of single-precision (32-bit) values which represent x,y coordinates. The array based at register **r2** receives a projected coordinate along the observer's horizontal direction:

```
project:   ldf      F3,0(r1)
           multf    F10,F3,F0
           ldf      F4,4(r1)
           multf    F11,F4,F1
           addf     F12,F10,F11
           divf     F13,F12,F2
           stf      0(r2),F13
           addi     r1,r1,#8
           addi     r2,r2,#4
           subi     r3,r3,#1
           bne      r3,zero,project
             nop
```

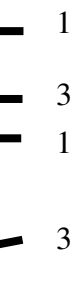


Problem 4a[3pts]: How many cycles does this loop take per iteration? Indicate stalls in the above code by labeling each of them with a number of cycles of stall:

There are 27 cycles per iteration (15 cycles of stall)

Problem 4b[4pts]: 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?

```
project:   ldf      F3,0(r1)
           ldf      F4,4(r1)
           multf    F10,F3,F0
           multf    F11,F4,F1
           addf     F12,F10,F11
           divf     F13,F12,F2
           addi     r1,r1,#8
           addi     r2,r2,#4
           subi     r3,r3,#1
           bne      r3,zero,project
             stf    -4(r2),F13
```



*This comes out at 19 cycles total. Notice that the offset of **stf** was adjusted because it moved past the addi instruction. Note that there are several related versions to this. You could move some of the integer operations up to fill the stall slots mentioned above. However, this will just move stalls from the beginning to the end.*

Problem 4c[7pts]: Unroll the loop once and schedule it to run with as few cycles as possible per iteration of the original loop. How many cycles do you get per iteration now?

```
project:   ldf     F3,0(r1)
           ldf     F4,4(r1)
           ldf     F5,8(r1)
           multf   F10,F3,F0
           ldf     F6,12(r1)
           multf   F11,F4,F1
           multf   F14,F5,F0
           multf   F15,F6,F1      <-- 1
           addf    F12,F10,F11
           addf    F16,F14,F15    <-- 1
           divf    F13,F12,F2
           divf    F17,F16,F2
           addi    r1,r1,#16
           addi    r2,r2,#8
           subi    r3,r3,#2       <-- 3
           stf     -8(r2),F13
           bne     r3,zero,project
              stf  -4(r2),F17
```

Total cycles per iteration = (18 + 5)/2 = 23 / 2 = 11.5 cycles per iteration. Notice that the stalls act like inserted no-op instructions – meaning that all computations advance. Notice the adjustment of the offsets for the store instructions. Make sure to get these correct!

Problem 4d[4pts]: Your loop in (4c) will not run without stalls. Without going to the trouble to unroll further, what is the minimum number of times that you would have to unroll this loop to avoid stalls? How many cycles would you get per iteration then?

If you add just one more iteration, it will be enough to all you to move all loads together, all multiplies together, all adds together, etc. Thus, the main thing to focus on is the ***divf→stf*** *latency. Assuming that we have* ***N*** *total iterations, then from the first* ***divf*** *to the first* ***stf*** *will involve (****N-1****)* ***divf*** *instructions and 3 integer instructions. This must fill the stalls so:*

N-1 + 3 = 7 ⇒ N = 5 *iterations (unrolled 4 times). This would be 7+4/5=7.8 cycles/iteration*

Problem 4e[7pts]: Software pipeline this loop to avoid stalls. Overlap 5 different iterations. What is the average number of cycles per iteration? Your code should have no more than one copy of the original instructions. Ignore startup and exit code.

```
project:   stf     0(r2),F13     } Iteration 1
           divf    F13,F12,F2    } Iteration 2
           addf    F12,F10,F11   } Iteration 3
           multf   F10,F3,F0     }
           multf   F11,F4,F1     } Iteration 4
           ldf     F3,32(r1)     }
           ldf     F4,36(r1)     } Iteration 5
           addi    r1,r1,#8
           subi    r3,r3,#1
           bne     r3,zero,project
             addi  r2,r2,#4
```

Now, you get 11 cycles/iteration (since there are no stalls.)

[This page intentionally left blank!]

[This is an extra page for scratch!]