Name ___________________________

# Computer System Architecture
# 6.823 Quiz #3
# April 25th, 2014
# Professors Daniel Sanchez and Joel Emer

*This is a closed book, closed notes exam.*

80 Minutes
16 Pages

Notes:
- Not all questions are of equal difficulty, so look over the entire exam and budget your time carefully.
- Please carefully state any assumptions you make.
- Show your work to receive full credit.
- Please write your name on every page in the quiz.
- You must not discuss a quiz's contents with other students who have not yet taken the quiz.

| | | |
|---|---|---|
| Part A | ________ | 45 Points |
| Part B | ________ | 35 Points |
| Part C | ________ | 20 Points |
| **TOTAL** | ________ | **100 Points** |

## Part A: Let's Talk About Loads (45 pts)

Consider the following code sequence:

```
…
I1:  DIV R3, R1, 8
I2:  BNEZ R9, Somewhere
I3:  ST R2, 0(R3)
I4:  LD R1, 8(R4)
I5:  ADD R5, R1, 8
I6:  SUB R10, R6, R7
I7:  MUL R8, R9, R10
I8:  BEQZ R8, Somewhere else
…
```

We will explore how this program behaves on different architectural styles. In all cases, assume the following execution latencies:

- `ADD, SUB`: 2 cycles
- `BNEZ, BEQZ`: 2 cycles
- `LD`: 2 cycles if cache hit, 8 cycles if miss
- `MUL`: 5 cycles
- `DIV`: 10 cycles

Additionally, the `LD (I4)` in this sequence *misses* in the data cache and therefore has a long latency of 8 cycles.

Assume that the branch at `I2` is not taken and fetch and decode never stall (e.g., by missing on the instruction cache or the BTB). Also assume that there are no structural hazards.

***Question 1 (5 points):***

Loads are often a bottleneck in processor performance, and as such compilers will try to move the loads as early as possible in the program to "hide" their latency. However, in the preceding code sequence, an optimizing compiler *cannot* move the load earlier in the program. Explain why in one or two sentences.

We need to explain why the LD can't be moved before the ST. (Otherwise, it *could* be moved earlier, even if not to the very beginning.) The reason is that there could be a RAW hazard through memory—maybe 0(R3)==8(R4).

Answers that there is a control hazard at I2 or a WAW hazard with I1 do not explain the difficulty of moving the LD earlier.

***Question 2 (6 points):***

Show how this program would work on a single-issue in-order pipeline that tracks dependencies with a simple scoreboard. Instructions are issued (i.e., dispatched for execution) in order, but can complete out of order. Assume infinite functional units and full bypassing. Fill in the remainder of the table below.

| Instruction | Issue Cycle | Completion Cycle |
|---|---|---|
| `I1: DIV R3, R1, 8` | 1 | 11 |
| `I2: BNEZ R9` | 2 | 4 |
| `I3: ST R2, 0(R3)` | 11 | n/a |
| `I4: LD R1, 8(R4)` | 12 | 20 |
| `I5: ADD R5, R1, 8` | 20 | 22 |
| `I6: SUB R10, R6, R7` | 21 | 23 |
| `I7: MUL R8, R9, R10` | 23 | 28 |
| `I8: BEQZ R8` | 28 | 30 |

There is no hazard preventing issue of I6, so it can issue at 21. It can't issue earlier because the processor is in-order. Following I6 is a string of RAW dependencies, so the latency of I6, I7, and I8 determine the code sequence's completion time.

***Question 3 (6 points):***

Assuming a single-issue out-of-order processor, show at which cycles instructions are issued (i.e., dispatched for execution) and complete. Assume that instructions are dispatched in program order if multiple are ready in the same cycle, and *do not speculate on data dependencies.* Again assume infinite functional units and full bypassing.

| Instruction | Issue Cycle | Completion Cycle |
|---|---|---|
| `I1: DIV R3, R1, 8` | 1 | 11 |
| `I2: BNEZ R9` | 2 | 4 |
| `I3: ST R2, 0(R3)` | 11 | n/a |
| `I4: LD R1, 8(R4)` | 12 | 20 |
| `I5: ADD R5, R1, 8` | 20 | 22 |
| `I6: SUB R10, R6, R7` | 3 | 5 |
| `I7: MUL R8, R9, R10` | 5 | 10 |
| `I8: BEQZ R8` | 10 | 12 |

Because we are not speculating on data dependencies, we cannot issue the LD before we know the ST address. So the earliest that the LD can issue is when I1 completes. Since the ST appears earlier in program order, it is issued first, and the LD is delayed until cycle 12. We can, however, begin issuing I6 at cycle 3 while waiting for I1 to complete.

In one or two sentences, what is the advantage of an out-of-order architecture vs. the in-order pipeline for this code sequence?

We are able to execute I6, I7, and I8 while the processor is waiting on memory, shortening the completion time.

***Question 4 (5 points):***

Suppose the out-of-order processor chose to execute the load first, *before all other instructions in the code sequence*. What events could cause the load to be aborted, and what mechanisms are required to detect mis-speculation and roll back? Ignore exceptions in your answer.

Two events are relevant: the ST writes the address read by the LD, or the branch at I2 is mispredicted.

The former requires a speculative load buffer to detect RAW memory hazards. The latter requires detection of mis-speculation and redirecting fetch to the right address. Both require flushing the ROB for mis-speculated instructions.

Name ______________________________

***Question 5 (6 points):***

Write VLIW code for this instruction sequence, assuming that the VLIW format is:

| Memory operation | ALU operation | ALU operation / Branch |
|---|---|---|

Try to make your VLIW code as efficient as possible, including re-ordering any instructions that do not have dependencies. For this VLIW code just use standard MIPS instructions to fill slots without predication or new, VLIW-specific instructions. (That is, simply schedule the instructions already provided.) Assume that the VLIW architecture has a scoreboard that stalls when a result is used before it is ready (e.g., on a cache miss).

| | | |
|---|---|---|
| | `DIV R3, R1, 8` | `BNEZ R9` |
| `ST R2, 0(R3)` | `SUB R10, R6, R7` | |
| `LD R1, 8(R4)` | | |
| | `MUL R8, R9, R10` | |
| | `ADD R5, R1, 8` | `BEQZ R8` |
| | | |
| | | |
| | | |

This code schedule is effectively what the OOO processor does, with some independent operations scheduled in parallel. I6 is moved earlier in the program, and I7 & I8 execute while the LD is waiting. The one subtlety of this code is that the MUL is delayed one instruction so that the LD is not delayed. This is important because the critical path of this computation is DIV→ST→LD→ADD (issued).

In one or two sentences, what is the advantage/disadvantage of a VLIW architecture for this code sequence vs. the out-of-order pipeline?

For this code sequence, the VLIW code can achieve similar performance to an OOO processor with much simpler hardware logic. This is possible because it pushes the scheduling complexity into the compiler.

The disadvantage is similar—for VLIW to work well, the compiler must be able to schedule instructions effectively. Often this is not possible in practice.

Josh Fisher points out that if it has a scoreboard, it's not a *true* VLIW. How would the code sequence change if we didn't have a scoreboard?

We would need to schedule NOPs explicitly to handle the latency of each operation. This becomes complicated with variable latency operations, like LDs with a cache.

***Question 6 (5 points):***

VLIW architectures rely heavily on the compiler to expose instruction-level parallelism in the program, so hiding load latency is a major challenge. VLIW compilers developed a technique called *trace scheduling* that merges multiple basic blocks into a single code sequence with software checks to ensure correctness. We profile our program and find that the first branch (`I2`) is almost never taken, so merging both basic blocks is a good idea.

If we use trace scheduling to move the load (`I4`) to be the *first* instruction, what conditions must software check to ensure correctness of the load for this code sequence? Ignore exceptions in your answer.

The answer is: "Same as OOO, except in software." We must check that there was no RAW hazard between ST→LD. We also must check R9 to make sure that the I2 branch was not taken.

Name ______________________________

***Question 7 (6 points):***

To mitigate load latency, you decide to implement a prefetch instruction. `PREFETCH Imm(rs)` takes a single argument, an address, and *hints* to the processor that the given address may be used soon. Crucially, `PREFETCH` is side-effect free—the processor can choose to ignore `PREFETCH`'s without affecting program behavior.

Now consider the following simplified code sequence:

```
DIV R3, R1, 8
ST R2, 0(R3)
LD R1, 8(R4)
ADD R5, R1, 8
```

The diagram below shows how this code executes on an in-order issue processor with scoreboarding. Show how performance can be improved using `PREFETCH`.

| **Cycle** | **In-order** | **In-order w/ Prefetch** |
|---|---|---|
| **1** | DIV | DIV |
| **2** | ↓ | PREFETCH |
| **3** | ↓ | |
| **4** | ↓ | |
| **5** | ↓ | |
| **6** | ↓ | |
| **7** | ↓ | |
| **8** | ↓ | |
| **9** | ↓ | |
| **10** | ↓ | |
| **11** | ST | ST |
| **12** | LD | LD |
| **13** | ↓ | |
| **14** | ↓ | ADD |
| **15** | ↓ | |
| **16** | ↓ | Complete |
| **17** | ↓ | |
| **18** | ↓ | |
| **19** | ↓ | |
| **20** | ADD | |
| **21** | ↓ | |
| **22** | Complete | |

Scheduling the PREFETCH before the DIV is correct but wastes a cycle unnecessarily.

Name ______________________________

***Question 8 (6 points):***

In lecture we discussed an alternative instruction, “load-speculate”:

`LD.S rt, Imm(rs)`

Load-speculate will fetch the value from memory but if the access faults it instead returns zero and does not cause an exception. Unlike prefetch, it gives not just the address but the source address *and* the destination register, which receives a value from memory. A load-speculate is followed in the program by a “load-check”:

`CHK.S rt, cleanup`

Load-check checks if the register was written by a `LD.S` that should have caused an exception (e.g., due to a page fault). If it was, then `CHK.S` branches to somewhere else to service the exception and handle any necessary cleanup. `CHK.S` executes in 1 cycle.

Show how to use `LD.S/CHK.S` to speed up the code even further than was possible with `PREFETCH`. Assume scoreboarding and infinite functional units. Assume that in this case the compiler knows that the load (`I4`) can be scheduled before the store (`I3`) safely. Do not show cleanup code.

| **Cycle** | **In-order** | **In-order+`LD.S`+`CHK.S`** |
|---|---|---|
| **1** | `DIV` | `DIV` |
| **2** | ↓ | `LD.S` |
| **3** | ↓ | |
| **4** | ↓ | |
| **5** | ↓ | |
| **6** | ↓ | |
| **7** | ↓ | |
| **8** | ↓ | |
| **9** | ↓ | |
| **10** | ↓ | `ADD` |
| **11** | `ST` | `ST` |
| **12** | `LD` | `CHK.S` |
| **13** | ↓ | Complete |
| **14** | ↓ | |
| **15** | ↓ | |
| **16** | ↓ | |
| **17** | ↓ | |
| **18** | ↓ | |
| **19** | ↓ | |
| **20** | `ADD` | |
| **21** | ↓ | |
| **22** | Complete | |

The benefit of LD.S is that it allows for speculative computation on data before the check occurs. This can lead to significant performance gains.

Name ______________________________

## Part B: Exploiting Parallelism (35 points)

Consider the following C code sequence:

```
const int size = 64 * 1024;
int a[SIZE], b[SIZE], c[SIZE];
for (int i = 0; i < SIZE; i++) {
     if (a[i] > b[i]) {
          c[i] = a[i] + b[i];
     }
}
```

This is a repetitive computation with a simple dependency graph. If we look at the MIPS assembly code, we see that a large percentage of the instructions are doing bookkeeping. We'd like to reduce this overhead.

```
          // R1 points to a, R2 points to b, R3 points to c
          // R6 is i
          ADD R6, R0, SIZE
Loop:     LD R4, 0(R1)
          LD R5, 0(R2)
          SUB R8, R4, R5
          BGEZ R8, Skip
          ADD R4, R5, R4
          ST R4, 0(R3)
Skip:     ADD R1, R1, 4
          ADD R2, R2, 4
          ADD R3, R3, 4
          SUB R6, R6, 1
          BNEZ R6, Loop
```

***Question 1 (5 points):***

Circle the MIPS instructions in the assembly above that perform "useful work" rather than bookkeeping.

Shown in red above.

***Question 2 (5 points):***

If the loads in the preceding code take four cycles, then this code sequence will stall and performance will suffer. Explain how an in-order, fine-grain multithreaded processor with two threads could mitigate this effect?

Fine-grain multithreaded processors use round robin to schedule threads. So with two threads, each thread executes one instruction (or tries to, at least) every two cycles. This effectively halves the load latency, and therefore leads to fewer stalls.

How would the program need to change for multhreading? (You do *not* need to write the code.)

You need to split the iterations evenly between the threads. This can be done in many ways; one simple way is to have the first do even iterations and the second to do odd iterations.

***Question 3 (10 points):***

An alternative approach is to hide the load latency within a single thread by using loop unrolling. Loads take four cycles and adds take one cycle. Write a loop unrolled VLIW version of the preceding code using the same VLIW instruction format as in Part A:

| Memory operation | ALU operation | ALU operation / Branch |
|---|---|---|

Unroll the fewest number of loop iterations necessary to cover the load's latency. Whatever degree of unrolling you choose, assume it divides the array size. Also assume that predication is allowed:

`(p1) instruction` executes the instruction if predicate register `p1` is set.
`cmp.gt p1, r1, r2` sets predicate register `p1` if `r1` is greater than `r2`.

Finally, `R1` points to `a`, `R2` points to `b`, `R3` points to `c`, and `R6` is `i`.

| | | |
|---|---|---|
| LD R4, 0(R1) | | |
| LD R5, 0(R2) | | |
| LD R4, 4(R1) | | |
| LD R5, 4(R2) | | |
| LD R4, 8(R1) | | |
| LD R5, 8(R2) | ADD R6, R4, R5 | CMP.GT P1, R4, R5 |
| | ADD R1, R1, 12 | ADD R2, R2, 12 |
| (P1) ST R6, 0(R3) | ADD R6, R4, R5 | CMP.GT P1, R4, R5 |
| | ADD R3, R3, 12 | SUB R6, R6, 3 |
| (P1) ST R6, 4(R3) | ADD R6, R4, R5 | CMP.GT P1, R4, R5 |
| | | |
| (P1) ST R6, -4(R3) | BNEZ R6, Loop | |

Name ______________________________

***Question 4 (10 points):***

Write a vector version using vector instructions and the vector mask register. Assume that the vector machine can do up to 64 operations per instruction, and note that `SIZE` is a multiple of 64.

`VLR` register stores the vector length.
`LV v1, r1, Imm` loads vector register `v1` with memory starting at address `r1` and stride `Imm`. `SV v1, r1, Imm` behaves similarly for stores.
`ADDV v1, v2, v3` adds `v2` and `v3` and puts the result in `v1`.
`SGTVV v1, v2` sets the vector mask register for each vector element in `v1` greater than the corresponding element in `v2` (mask set means the operation is enabled).
`CVM` resets the vector mask register (turns on all elements).

```
          // R1 points to a, R2 points to b, R3 points to c
          // R6 is i
          ADD R6, R0, SIZE
          LI VLR, 64
Loop:





          CVM
          LV V1, R1, 4
          LV V2, R2, 4
          SGTVV V1, V2
          ADDV V1, V1, V2
          SV V1, R3, 4






Skip:     ADD R1, R1, 64*4
          ADD R2, R2, 64*4
          ADD R3, R3, 64*4
          SUB R6, R6, 64
          BNEZ R6, Loop
```

***Question 5 (5 points):***

Is this program easy to map to GPUs? What inefficiencies may arise? Explain your answer in one or two sentences.

This program is easy to write for GPUs because each iteration is completely independent. It may be inefficient, however, due to branch divergence, depending on the distribution of A[i] > B[i] within the array.

## Part C: TLBs in Outer Space (20 points):

You observe that page table entries exhibit a lot of spatial locality (consecutive pages are often accessed). To increase your processor's TLB hit rate, you increase the block size of your TLB to hold several page table entries:

- Each TLB entry holds 4 page table entries (PTEs).
- Each page table entry is 19 bits containing:
    - A 16-bit physical page number (PPN).
    - Three protection bits:
        - Readable bit
        - Writeable bit
        - Used bit (for page replacement)

NASA has hired you to design a processor for a manned mission to Mars. You need to make sure your TLB works in the hostile environment of outer space.

***Question 1 (10 points):***

One of NASA's programs executes the following memory operations:

| **Cycle** | **Operation** | **Instruction is ACE?** |
|---|---|---|
| **0** | Load from page A (TLB miss) | Yes |
| **15** | Load from page A | Yes |
| **20** | Load from page A | Yes |
| **30** | Load from page A | No |
| **40** | Load from page B (TLB miss, evicting block containing A from the TLB) | Yes |

For which cycles is the PPN for page A itself ACE?

| **Cycle** | **0-15** | **15-20** | **20-30** | **30-40** |
|---|---|---|---|---|
| **Is ACE?** | Yes | Yes | No | No |

An ACE instruction reads a value from page A at cycle 20, so the PPN is ACE from 0-20. This would apply even if the load at cycle 15 weren't ACE, since the TLB entry is only updated at cycle 0.

What is the AVF of the PPNs in the TLB block (consider all PPNs in a block, but not tag or protection bits) over cycles 0-40?

The AVF of the PPN for page A is 20 cycles / 40 cycles = ½. There are four PPNs per TLB block (by question description), so this applies to ¼ of the PPNs overall. Thus the total AVF is $\frac{1}{2} \times \frac{1}{4} = \frac{1}{8}$.

***Question 2 (10 points):***

Discuss the AVF of each protection bit (readable, writeable, used) for PTEs in the TLB.

First, the "used" bit is only used to for page replacement, so no matter what happens to this bit it can only impact performance not correctness. Therefore its AVF is zero.

For the permission bits, the story is quite a bit more complicated. If a permission bit goes from 0 to 1, then it's possible that an application will now do something that it shouldn't. So this is potentially troublesome. But we have to ask ourselves, how likely is this to occur? Generally speaking, most programs do not violate their permissions. So it is quite unlikely that something nefarious will occur due to a bit flip on a simple page in the TLB, which is after all transient.

One example of how this might actually be a problem is when permissions bits are *intended* to be violated. For example, when two processes fork(), they are semantically given two copies of memory. In practice this is implemented by turning off write permissions, and then the operating system will copy pages on demand when they are written. If the write bit goes from 0 to 1 in this case, then it will allow a process to write to a page still being used by the other process. This violates the copy semantics between the two processes. This case shows that violations may not be as improbable as we might otherwise assume.

Finally, permissions transitions from 1 to 0 are either very problematic or no problem at all, depending on assumptions about the operating system. If the operating system is "dumb" and simply kills processes when they violate permissions, then this will probably cause serious problems. But if the operating system is "reliability-aware" then it can check the fault against the permissions in the *page table* (not TLB!) and recognize that a harmless bit flip has occurred. In this case, the O/S can replace the broken TLB entry and the program can continue execution.

Consider the tag of the TLB. Is the AVF of these tag bits relatively high or relatively low? Explain why in one or two sentences.

The TLB tag has relatively low AVF. Consider two kinds of faults that can happen.

False hits: If there is a TLB hit that should not have occurred, then the processor will get the PTE for a wrong page. This is likely to cause problems if the memory access is ACE, but even then it's surprisingly unlikely to cause incorrect output. This is a topic for much deeper discussion, but generally speaking programs are unexpectedly resilient to faults. The main point is that it's unlikely that a single bit flip of the TLB tag will cause a hit on some other address. It's more likely that it will lead to false misses…

False misses: This is a case where the TLB should have hit but didn't. Any single bit flip will cause this to occur. But because our TLB doesn't have a dirty bit (curious design!), the data in the TLB can be simply overwritten with the correct value from the page table.

You want to protect the data in the TLB. Would you recommend using parity or SECDED (single error correction, double error detection)? Explain your answer in one or two sentences.

Parity is sufficient because errors are rare (so double errors are negligible), and the authoritative value of the PTE is always available in memory.