Midterm I
SOLUTIONS
March 18, 2009
CS252 Graduate Computer Architecture

<table>
<thead>
<tr>
<th>Problem</th>
<th>Possible</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>
[ This page left for π ]

3.141592653589793238462643383279502884197169399375105820974944
Question #1: Short Answer [30 pts]
For the following questions, please give a short answer (a few lines). Try not to write a book.

**Problem 1a [2 pts]:** Describe why the SPARC processor saves 2 addresses (PC and nPC) after an exceptional event in order to describe a precise exception point. Why can the MIPS processor (which has a similar pipeline to the SPARC) get away with saving only 1 address?

*If the exception point is in a branch-delay slot, then one needs two PCs to restart execution: one that points at the exceptional instruction (delay slot) and another that points at the next instruction. The second PC is needed because the instruction following the delay slot could be either the next sequential instruction (if the branch was not taken) or the branch target (if the branch was taken). Although MIPS has the same issue, it relies on software to back up the exception point to the previous instruction (when it is a branch) before restarting execution; this works because branches are idempotent.*

**Problem 1b [2 pts]:** Why is it advantageous to have a prime number of banks of DRAM in a vector processor?

*Because the vector processor would be able to support many different memory strides efficiently (without causing bank conflicts). The reason is that any stride that is relatively prime to the number of banks would work its way through all banks before repeating; only strides that are divisible by the particular prime number would have bank conflicts.*

**Problem 1c [3 pts]:** Explain why exceptions can occur out of order in an in-order, 5-stage pipeline and describe how such a pipeline reorders these exceptions to provide a precise exception model.

*Because exceptions can occur in different stages. Example: Illegal instruction faults occur in the D stage, overflow errors occur in the E stage, and memory faults occur in the M stage. As a result, instructions that are later in the flow of the program can have exceptions that occur earlier:*

\[
\begin{align*}
F_1 & \quad D_1 \quad E_1 \quad M_1 \quad W_1 \\
F_2 & \quad D_2 \quad E_2 \quad M_2 \quad W_2
\end{align*}
\]

*In the above example, D₂ occurs in time earlier than M₁, even though the first instruction is the precise exception point.*

*The five-stage pipeline reorders exceptions simply by waiting to handle an exception until a well defined point in the pipeline, such as the end of the memory stage. Thus, whenever an exception occurs, the corresponding stage simply sets an exception field in the pipeline, rather than stopping the pipeline. The memory stage will look at this field to decide which instruction should be the precise exception point.*
Problem 1d[2pts]: In both the Transmeta and DAISY papers, the authors address the need to handle self-modifying code (SMC). Explain why SMC presents a problem for Transmeta and DAISY, and present one mechanism for dealing with SMC (could work with either of them):

Self-modifying code presents a problem for these systems because they dynamically compile source instructions into the destination VLIW architectures. As a result, if a program modifies some of its code by writing instructions to its code space, then the dynamically compiled instructions will be inconsistent with the source instructions. One simple fix is to write-protect code pages (with virtual memory) so that any writes to code will cause a page-fault, which can notify the system that recompilation is necessary.

Problem 1e[4pts]: What is trace scheduling and why is it desirable for a VLIW processor? What is the role of the “fixup” code produced by trace scheduling?

Trace scheduling profiles the execution of code to find common “traces” through the code. A trace is distinguished by a set of branch directions. One the trace scheduler has these traces, then it optimizes code as if the conditional branches were not in the code. This type of scheduling is desirable for VLIW because all of the instructions in each trace are optimized together (without branches) yielding large basic blocks to find parallelism (and hence densely utilized VLIW instructions). Fixup code must be generated to correct for those (hopefully uncommon) situations in which branches go a different direction than predicted by the optimized traces; this code is arranged to “undo” operations on the main trace that shouldn’t have occurred.

Problem 1f[6pts]: For the following classes of pipeline hazards, give (1) a definition, (2) an explanation of how they are handled in an in-order 5-stage pipeline, and (3) an explanation of how they are handled in a typical out-of-order, superscalar processor.

A. Control Hazards:
   Defn: Control hazards are hazards involving branches, in which branch resolution takes long enough that it is not clear which instructions should be fetched. The 5-stage, in-order pipeline handles control hazards by specifying that there is a single branch delay-slot. Out-of-order, superscalar processors handle control hazards by performing branch prediction (and including a mechanism for undoing the result of incorrectly predicted branches).

B. Data Hazards:
   Defn: Data Hazards are hazards involving the use of the same register name between multiple instructions in the pipeline. Or – they are hazards involving the flow of information from one instruction to another. RAW hazards result from a real flow of information from one instruction to a following one. WAR and WAW hazards result from the reuse of names from one instruction to the other without the flow of information.

   The 5-stage pipeline handles RAW hazards through bypassing; WAR and WAW hazards do not exist in the five-stage pipeline since all writes are committed in order.

   An out-of-order processor handles RAW hazards through either a scoreboard or Tomasulo. It handles WAR and WAW hazards through register renaming (either implicit or explicitly).
Problem 1g[3pts]: Define Simultaneous Multithreading (SMT). Name two different resources that would need to be duplicated in order to convert a single threaded, out-of-order superscalar processor into a two-threaded SMT processor.

Simultaneous Multithreading is a multithreading technique that mixes instructions from multiple threads into a single superscalar pipeline, thereby getting higher utilization of the functional units. Among other things, you would need to duplicate the logical register state or rename table (depending on how registers were handled), the reorder buffer, and possibly the branch-prediction state.

Problem 1h[2pts]: Name two dynamic quantities other than branch outcomes that can be predicted in an out-of-order processor.

Some options: Load Values, Functional Unit Results, Load-Store dependencies

Problem 1i[3pts]: Suppose that we start with a basic Tomasulo architecture that includes branch prediction. What changes are required to execute 4 instructions per cycle?

You would need a rename table mechanism that could rename registers into reservation tags four instructions at a time. You would need four result buses. You would probably need to increase the number of functional units and/or reservation stations to make up for increased number of instructions in flight. If you have a reorder buffer, it would need to be able to handle the addition of four instructions at a time and the commitment of four instructions at a time.

Problem 1j[3pts]: What is chaining in a vector machine? Why is it helpful?

Chaining is like forwarding for vector operations: it is the mechanism which allows feeding a result from one vector instruction directly (i.e. one element at a time) to a subsequent pipeline. It is useful because it can greatly increase performance.
Question #2: Branch Prediction Pipeline [30 pts]
Consider a fetch pipeline that is loosely based on the UltraSparc-III processor. Our processor is an in-order superscalar processor that fetches four instructions each cycle. These instructions must be aligned on a four-word boundary. In this question, we evaluate the impact of branch prediction on the processor’s performance. The fetch portion of the pipeline looks as follows:

<table>
<thead>
<tr>
<th>Stage</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>PC Generation/Mux</td>
</tr>
<tr>
<td>B</td>
<td>Branch Address Calc/Begin Decode</td>
</tr>
<tr>
<td>C</td>
<td>Complete Decode</td>
</tr>
<tr>
<td>D</td>
<td>Steer Instructions to Functional units</td>
</tr>
<tr>
<td>E</td>
<td>Register File Read (up to 4 insts)</td>
</tr>
<tr>
<td>F</td>
<td>Instruction Fetch Stage 1</td>
</tr>
<tr>
<td>G</td>
<td>Instruction Fetch Stage 2</td>
</tr>
<tr>
<td>H</td>
<td>Instruction Fetch Stage 3</td>
</tr>
<tr>
<td>I</td>
<td>Instruction Fetch Stage 4</td>
</tr>
<tr>
<td>J</td>
<td>Instruction Fetch Stage 5</td>
</tr>
<tr>
<td>K</td>
<td>Instruction Fetch Stage 6</td>
</tr>
<tr>
<td>L</td>
<td>Instruction Fetch Stage 7</td>
</tr>
<tr>
<td>M</td>
<td>Instruction Fetch Stage 8</td>
</tr>
<tr>
<td>N</td>
<td>Instruction Fetch Stage 9</td>
</tr>
<tr>
<td>O</td>
<td>Instruction Fetch Stage 10</td>
</tr>
<tr>
<td>P</td>
<td>Instruction Fetch Stage 11</td>
</tr>
<tr>
<td>Q</td>
<td>Instruction Fetch Stage 12</td>
</tr>
<tr>
<td>R</td>
<td>Instruction Fetch Stage 13</td>
</tr>
<tr>
<td>S</td>
<td>Instruction Fetch Stage 14</td>
</tr>
<tr>
<td>T</td>
<td>Instruction Fetch Stage 15</td>
</tr>
<tr>
<td>U</td>
<td>Instruction Fetch Stage 16</td>
</tr>
<tr>
<td>V</td>
<td>Instruction Fetch Stage 17</td>
</tr>
<tr>
<td>W</td>
<td>Instruction Fetch Stage 18</td>
</tr>
<tr>
<td>X</td>
<td>Instruction Fetch Stage 19</td>
</tr>
<tr>
<td>Y</td>
<td>Instruction Fetch Stage 20</td>
</tr>
<tr>
<td>Z</td>
<td>Instruction Fetch Stage 21</td>
</tr>
</tbody>
</table>

Here is a table to clarify when the direction and the target of a branch/jump is known.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Taken known? (At the end of)</th>
<th>Target known? (At the end of)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEQZ/BNEZ</td>
<td>R</td>
<td>B</td>
</tr>
<tr>
<td>J</td>
<td>B (always taken)</td>
<td>B</td>
</tr>
<tr>
<td>JR</td>
<td>B (always taken)</td>
<td>R</td>
</tr>
</tbody>
</table>
Problem 2a[3pts]: Assume that no branch target buffer or other branch prediction mechanism is used in this pipeline and that the A unit simply assumes that branches are not taken until proven otherwise (in fact, the A unit does not even have access to instructions). **Assume, for a moment, that dependencies do not prevent instructions from dispatching in the J stage (i.e. instructions pass through the J stage as soon as they arrive).** What is the maximum number of instructions that follow a branch in program order that have already been fetched (or are already committed to be fetched) by the time the branch is finally resolved? Explain. **Hint:** consider the position of the branch in its fetch group.

Assume that branch is first instruction in group of 4 ⇒ 3 instructions after branch. Further, branch decision made at end of R stage. By this time, there are groups of 4 instructions in stages P, F, B, I, and J. Further, the address computed in A has already been committed, so will lead to group of 4 instructions. Thus: Fetched instructions after branch = 6 × 4 + 3 = 27

Problem 2b[3pts]: Now, assume a worst-case scenario of dependencies. How would your answer to (2a) change? Explain carefully.

In this case, stage J will have 4 groups of 4 instructions instead of just 1 group of 4. So, modifying the solution from (2a), we get 9 × 4 + 3 = 39

Problem 2c[2pts]: Assume that no branch target buffer or other branch prediction mechanism is used in this pipeline, that the A unit simply assumes that branches are not taken until proven otherwise. How many branch delay slots would this pipeline have to expose to the programmer in order to avoid stalling or wasting fetch slots after a branch? Explain your answer.

As shown in (2b), in a worst-case scenario, there can be 39 instructions fetched after a branch before the branch is resolved. Thus, there would need to be 39 delay slots after a branch in order to avoid stalling or wasting fetch slots.

Problem 2d[2pts]: Obviously, the number given in (2c) is too big for a compiler to handle. The SPARC architecture provides a single branch delay slot from its early, 5-stage-pipeline days. Assuming that we must honor the single delay slot (for binary compatibility with original code) and that there is no branch prediction, what is the maximum number of lost potentially fetched instructions in this pipeline for each *taken* branch? What about each *untaken* branch? Explain your answers.

If there is only 1 delay slot, then the remaining 38 instructions (from 2b/c) will have to be nullified when there is a taken branch. On the other hand, an untaken branch will not have to nullify these instructions (assuming that there are no taken branches in that instruction set).

ANS: Taken branch: 38 instructions
Untaken branch: 0 instructions
Next, let’s add a branch predictor and branch target buffer (BTB) to our design. The following diagram shows how we have enhanced the design. The earliest stage that the BTB can finish is the P stage, since the A stage is still computing the address. The Branch History Table portion of the branch predictor is shown in the B stage, since this is the earliest stage at which the branch target address is available:

Unless redirected by (1) the BTB, (2) the BHT, or (3) branch resolution in stage R, the A unit will keep fetching the next 4 aligned instructions. The only time that less than 4 instructions may be fetched is in the first cycle after the A unit has been directed to a new address.

In the P stage, the BTB is checked for a match with the address of the current fetch group. Each BTB entry holds an [entryPC, targetPC] pair for jumps and branches predicted to be taken. If an entry does not exist, the BTB is assumed to predict not taken. Assume that we can search for all four PCs in a fetch group simultaneously and that the target PC predicted by the BTB is always correct for this question (even if the direction is wrong). If there is a match for the current fetch group, the PC is redirected to the target PC (of the earliest valid branch instruction in the group), and instructions that have already been fetched after the branch delay slot are invalidated. The BTB determines if the PC needs to be redirected at the end of the P stage, and the new PC is inserted at the beginning of the A stage. (The transmission of PCs to A and multiplexing of these PCs takes a whole cycle).

In the B stage, a conditional branch checks the BHT, but an unconditional jump does not. If the branch is predicted to be taken, the PC is redirected to the calculated branch target address and fetched instructions after the branch delay slot are invalidated. As with the BTB, the BHT determines if the PC needs to be redirected at the end of the B stage and the new PC is inserted at the beginning of the A stage. For a particular branch, if the BHT predicts “taken” and the BTB did not make a prediction, the BHT will redirect the PC. If the BHT predicts “not-taken” it does not redirect the PC.

**Problem 2e[3pts]:** Explain why this pipeline could accommodate a multi-level branch predictor. What stages could be used to compute the various levels of a PAg branch predictor?

*Because it only needs to use the result of a branch prediction at the end of the B stage (when the branch address is ready). Since the instruction address is available in the P stage, this pipeline can use three cycles: P, F, and B to perform branch prediction. Thus, since the PAg predictor requires two table lookups, we could use the F stage for the BHR (Branch History Register) and the B stage for the PHT (Pattern History Table).*
**Problem 2f [4pts]:** Fill out the following table of the maximum number of *lost dispatch cycles* (namely cycles in which the J stage will have no instructions to dispatch) for each combination of BTB and BHT results:

<table>
<thead>
<tr>
<th>Conditional Branches</th>
<th>BTB Hit?</th>
<th>(BHT) Predicted Taken?</th>
<th>Actually Taken?</th>
<th>Maximum # of cycles of pipeline bubble</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y Y Y</td>
<td></td>
<td></td>
<td>Y</td>
<td>1</td>
</tr>
<tr>
<td>Y Y N</td>
<td></td>
<td></td>
<td>N</td>
<td>6</td>
</tr>
<tr>
<td>Y N Y</td>
<td></td>
<td></td>
<td>Y</td>
<td>1</td>
</tr>
<tr>
<td>Y N N</td>
<td></td>
<td></td>
<td>N</td>
<td>6</td>
</tr>
<tr>
<td>N Y Y</td>
<td></td>
<td></td>
<td>Y</td>
<td>3</td>
</tr>
<tr>
<td>N Y N</td>
<td></td>
<td></td>
<td>N</td>
<td>6</td>
</tr>
<tr>
<td>N N Y</td>
<td></td>
<td></td>
<td>Y</td>
<td>6</td>
</tr>
<tr>
<td>N N N</td>
<td></td>
<td></td>
<td>N</td>
<td>0</td>
</tr>
</tbody>
</table>

**Problem 2g [3pts]:** Suppose that we have perfect branch prediction (BTB and BHT). What feature of the pipeline causes us to have lost dispatch cycles under some circumstances? *Hint: some feature of this pipeline is different from what we discussed in class.* Why might we hope that we will be able to avoid these lost dispatch cycles for real code? Explain carefully.

*The existence of a separate stage for computing addresses (A) prevents the BTB from occurring in the first stage. Thus, by the time the BTB makes a decision, there has already been another group of instructions to be fetched even when the branch is predicted to be taken by the BTB. The hope is that dependencies will cause instructions to stall in the J stage long enough for the 1 cycle bubble to be absorbed.*

**Problem 2h[3pts]:** For the case in which the BTB predicts “taken” and the BHT predicts “not taken”, why is it more probable that the BTB made a more accurate prediction than the BHT, even though the BHT is likely to be bigger? Explain carefully.

*Entries in the BTB are tagged with the instruction address, while the BHT is not tagged. Thus, the BTB has no aliases while the BHT potentially does. In addition, since the BTB is smaller, it is more likely to have recent information than the BHT.*
Problem 2i[4pts]: What is aliasing in a branch predictor? Draw a block diagram for a branch predictor that tries to take advantage of constructive aliasing; explain the theory of operation for your predictor.

Aliasing occurs when two different branches map to the same position in branch prediction table structures; as a result, branches with different behaviors can interfere with each other’s predictions.

The BiMode predictor is an example of a predictor that attempts to separate branches that are highly biased true from those that are highly biased false. By sorting branches, it attempts to take advantage of constructive aliasing between branches that have similar behavior.

Problem 2j[3pts]: What is a tournament branch predictor? Draw a simple block diagram representing a tournament predictor. Name a real processor that utilizes a tournament predictor.

The motivation behind a tournament branch predictor is that a single type of predictor is insufficient to predict all branches. Thus, each branch address is fed to a meta-predictor (usually a simple table of two-bit counters) that predicts which predictor should be used on that branch. Once a predictor is chosen, then the selected predictor is used to predict the branch at that address.

The Alpha 21264 uses a tournament predictor.
Problem #3: Software Scheduling [25pts]

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: **5 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 data cache takes two cycles to access after the address is computed (causing a load-use latency of **2 cycles**). Assume that all functional units are fully pipelined and bypassed.

**Problem 3a [5pts]:** Draw the pipeline stages of an in-order, single-issue processor that would support the above latencies. Make sure to say what happens in each stage. Try to make the pipeline as short as possible. Include and label the bypass paths such that this pipeline will never stall unless it encounters a real dependency. For each bypass arc, label it with the types of instructions that will forward their results along these paths (i.e. use “M” for multif, “A” for addf, “I” for integer operations and “Ld” for loads). [**Hint: make sure to optimize for store instructions**]

Stage:
- F Fetch next instruction
- D Decode stage
- EX1 Integer Ops
  - Address compute (for Ld/St)
  - First stage of: Addf, Multf
- EX2 First stage of: Ld/St.
  - Last Stage of: Addf
  - Second stage of: Multf
- EX3 Last stage of: Ld/St
  - Third stage of: Multf
- EX4 Fourth stage of: Multf
- EX5 Last stage of: Multf
- W Writeback stage
**Problem 3b[3pts]:** 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)
            <2 stall cycles from 2 cycle load-use latency>
multf     F10,F3,F2
ldf       F4,0(r2)
            <3 stall cycles from F10 RAW>
multf     F11,F10,F4
            <4 stall cycles from F11 RAW>
addf      F5, F5, F11
addi      r1,r1,#8
addi      r2,r2,#8
subi      r3,r3,#1
bneq      r3,filter
nop
```

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

10 instruction cycles + 9 stall cycles = 19 cycles

**Problem 3c[3pts]:** 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?

*One possible schedule. No matter how you schedule, there is always going to be 5 stall cycles. Thus total number of cycles is 14 cycles/iteration. Note that we can put useful computation in the delay slot instead of nop.*

```
filter:    ldf      F3,0(r1)
ldf       F4,0(r2)
addi      r1,r1,#8
multf     F10,F3,F2
addi      r2,r2,#8
subi      r3,r3,#1
            <2 stall cycles from F10 RAW>
multf     F11,F10,F4
bneq      r3,filter
            <3 stall cycles from F11 RAW>
addf      F5, F5, F11
```

12
**Problem 3d[5pts]:** 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?

One possible schedule. Note you should only have 2 stall cycles total and be able to remove two addi and a subi instructions and change the constants. This brings the total execution cycle to 16 or 8 cycle/iteration.

```assembly
filter:  ldf      F3,0(r1)
         ldf      F13,8(r1)
         ldf      F4,0(r2)
         multf   F10,F3,F2
         multf   F20,F13,F2
         ldf      F14,8(r2)
         addi    r1,r1,#16
         <1 stall cycle from F10 RAW>
         multf   F11,F10,F4
         multf   F21,F20,F14
         addi    r2,r2,#16
         subi    r3,r3,#2
         <1 stall cycle from F11 RAW>
         addf    F5, F5, F11
         bneq    r3,filter
         addf    F5, F5, F21
```

**Problem 3e[5pts]:** 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):*

You need to turn the real data dependencies into anti-dependencies. There are no stall cycles and thus this loop execute at 9 cycles/iteration. It's important to alter the offset to the two ldf addresses since they belong to different iterations now.

```assembly
filter:  addf    F5, F5, F11  <- iter 0
         multf   F11,F10,F4  <- iter 1
         multf   F10,F3,F2  <- iter 2 (2 instructions)
         ldf      F4,16(r2)
         ldf      F3,24(r1)  <- iter 3
         addi    r1,r1,#8
         subi    r3,r3,#1
         bneq    r3,filter
         addi    r2,r2,#8
```
Problem 3f[4pts]: Now, add startup and exit code to your answer for (3e) 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 r1 and r2 such that accesses beyond the bounds of the arrays do not cause memory faults. Also assume that 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 (3e), and ignore partial results on exit]:

STARTUP:

; <start up iteration 0>
ldf   F3,0(r1)
ldf   F4,0(r2)
multf F10,F3,F2
multf F11,F10,F4

; <start up iteration 1>
ldf   F3,8(r1)
ldf   F4,8(r2)
multf F10,F3,F2

; <start up iteration 2>
ldf   F3,16(r1)

; <execute the loop 1 less since as soon as we enter the code in 3(e), we have completed one iteration already>
subi  r3,r3,#1

EXIT:

Nothing needs to be done here. Correct value is stored in F5.
Problem #4: Caching away [15pts]

A certain system with a 500 MHz clock uses a Harvard architecture (separate data and instruction caches) at the first level, and a unified second-level cache. The first-level data cache is a direct-mapped, write-through, writes-allocate cache with 8 Kbytes of data total and 8-Byte blocks, and has a perfect write buffer (never causes any stalls). The first-level instruction cache is a direct-mapped cache with 4KBytes of data total and 8-byte blocks. The second-level cache is a two-way set associative, write-back, write-allocate cache with 2MBytes of data total and 32-Byte blocks.

The first-level instruction cache has a miss-rate of 2%. The first-level data cache has a miss-rate of 15%. The unified second-level cache has a local miss rate of 10% (i.e. the miss rate for all accesses going to the second-level cache). Assume that 40% of all instructions are data memory accesses; 60% of those are loads, and 40% are stores. Assume that 50% of the blocks in the second-level cache are dirty at any time. Assume that there is no optimization for fast reads on an L1 or L2 cache miss.

All first-level cache hits cause no stalls. The second-level hit time is 10 cycles (That means that the L1 miss-penalty, assuming a hit in the L2 cache, is 10 cycles). Main memory access time is 100 cycles to the first bus width of data; after that, the memory system can deliver consecutive bus widths of data on each following cycle. Outstanding, non-consecutive memory requests cannot overlap; an access to one memory location must complete before an access to another memory location can begin. There is a 128-bit bus from memory to the L2 cache, and a 64-bit bus from both L1 caches to the L2 cache. Assume that the TLB never causes any stalls.

Problem 4a[2pts]: What fraction of all data memory references cause a main memory access (main memory is accessed before the memory request is satisfied)? **First show the equation, then the numeric result.**

If you did not treat all stores as L1 misses:

\[
= (L1 \text{ miss rate}) \times (L2 \text{ miss rate})
\]

\[
= (.15) \times (.1)
\]

\[
= 1.5\%
\]

If you treated all stores as L1 misses:

\[
= (% \text{ of data ref that are stores}) \times (L2 \text{ miss rate}) +
\]

\[
= (% \text{ of data ref that are loads}) \times (L1 \text{ miss rate}) \times (L2 \text{ miss rate})
\]

\[
= (.4) \times (.1) + (.6) \times (.15) \times (.1)
\]

\[
= 4.9\%
\]

Problem 4b[3pts]: How many bits are used to index each of the caches? Assume that the caches are presented physical addresses.

\[
\text{Data} = \frac{8K}{8} = 1024 \text{ blocks} = 10 \text{ bits}
\]

\[
\text{Inst} = \frac{4K}{8} = 512 \text{ blocks} = 9 \text{ bits}
\]

\[
\text{L2} = \frac{2M}{32} = 64k \text{ blocks} = 32k \text{ sets} = 15 \text{ bits}
\]
**Problem 4c[5pts]:** What is the average memory access time in cycles (including instructions and data memory references)? **First show the equation, then the numeric result.** *Hint: don’t forget to consider dirty lines in the L2 cache.*

If you did not treat all stores as L1 misses:

\[
AMAT_{\text{Total}} = \frac{1}{1.4} \cdot AMAT_{\text{inst}} + \frac{0.4}{1.4} \cdot AMAT_{\text{data}}
\]

\[
AMAT = (L1 \text{ hit time}) + (L1 \text{ miss rate}) \times [(L2 \text{ hit time}) + (L2 \text{ miss rate}) \times \text{mem transfer time})]
\]

\[
AMAT_{\text{inst}} = 1 + 0.02(10 + 0.10 \times 1.5 \times 101) = 1.503
\]

\[
AMAT_{\text{data}} = 1 + 0.15(10 + 0.10 \times 1.5 \times 101) = 4.7725
\]

\[
AMAT = 2.44
\]

Note that memory transfer time is multiplied by 1.5 to account for write backs of dirty lines in L2 cache. Also a L2 miss will take 100+1=101 cycles to fill a L2 cache line (the first 16B returns in 100 cycles; the next 16 in the next cycle).

If you treat all stores as L1 misses:

\[
AMAT_{\text{Total}} = \frac{1}{1.4} \cdot AMAT_{\text{inst}} + \frac{0.24}{1.4} \cdot AMAT_{\text{loads}} + \frac{0.16}{1.4} \cdot AMAT_{\text{stores}}
\]

\[
AMAT = (L1 \text{ hit time}) + (L1 \text{ miss rate}) \times [(L2 \text{ hit time}) + (L2 \text{ miss rate}) \times \text{mem transfer time})]
\]

\[
AMAT_{\text{inst}} = 1 + 0.02(10 + 0.10 \times 1.5 \times 101) = 1.503
\]

\[
AMAT_{\text{loads}} = 1 + 0.15(10 + 0.10 \times 1.5 \times 101) = 4.7725
\]

\[
AMAT_{\text{stores}} = 1 + 1(10 + 0.10 \times 1.5 \times 101) = 26.15
\]

\[
AMAT = 4.88
\]

**Problem 4d[2pts]:** Assume that the TLB will be accessed in parallel with the first-level cache (for speed). What is the smallest page size for this system and why?

Smallest page size can either be 4kB or 8kB (same size as L1 cache). To perform parallel lookup with TLB, the address bits that are used to index the line in the cache will need to stay the same before and after the translation. For L1 inst cache, that’s 12 bits (9 line index bits and 3 byte select bits). For L1 data cache, that’s 13 bits.

**Problem 4e[3pts]:** What is a victim cache? Why might we say that a system with a victim cache has a “fractional” associativity between 1 and 2? How might we calculate this associativity number (come up with something reasonable here)?

Victim cache is a small, full-associative buffer that stores cache lines evicted from main cache (direct-mapped in this case). It is intended to reduce conflict misses. Upon data loads, both the main cache (direct mapped) and victim cache (fully-associative) are searched for the data. To the processor, the associativity of the cache subsystem would appear to be more than 1 but not 2 (because the size of victim is << main cache). To calculate this associativity, one can run benchmarks and measure cache miss rate between direct-mapped, 2-way set associative and this system and fit the curve.
[ This page left for scratch ]