Computer Architecture

EE 4720

Final Examination

10 May 2014, 10:00–12:00 CDT

Problem 1  ________  (15 pts)
Problem 2  ________  (15 pts)
Problem 3  ________  (15 pts)
Problem 4  ________  (15 pts)
Problem 5  ________  (5 pts)
Problem 6  ________  (10 pts)
Problem 7  ________  (25 pts)

Alias  ________________________________
Exam Total  ________  (100 pts)

Good Luck!
Problem 1: (15 pts) Illustrated below is the **stall-in-ME** version of our MIPS implementation, taken from the solution to the midterm exam.

(a) Show a pipeline execution diagram of the code below on this pipeline.

(b) Wires in the diagram are labeled A, B, C, and D. Under your pipeline execution diagram show the values on those wires when they are in use.

- Show pipeline execution diagram.  
- Show values of A, B, C, and D.

```
add.s f4,f5,f6  
sub.s f7,f8,f9  
lwc1 f1, 0(r2)
```

A:  
B:  
C:  
D:
Problem 2: (15 pts) Illustrated below is a 2-way superscalar MIPS implementation. Design the hardware described below. You can use the following logic blocks (with appropriate inputs) in your solution: The output of logic block isALU is 1 if the instruction’s result is computed by the integer ALU. The output of logic block rtSrc is 1 if the instruction uses the rt register as a source. The output of logic block isStore is 1 if the instruction is a store.

(a) Design logic to generate a signal named STALL, which should be 1 when there is a true (also called data or flow) dependence between the two instructions in ID.

- Control logic to detect true dependence in ID and assign STALL.

(b) The code fragment below should generate a stall in our two-way superscalar implementation when the two instructions are in the same fetch group. However this particular instruction pair is a special case in which the stall is not necessary when the right bypass path(s) and control logic are provided. Note: There was a similar-sounding problem in last year’s final, but the solutions are different.

```
0x1000: add r1, r2, r3
0x1004: sw r1, 0(r5)
```

- Add the bypass path(s) needed so that the code executes without a stall.

- Add control logic to detect this special case and use it to suppress the stall signal from the first part.
Problem 3: (15 pts) Code producing the branch patterns shown below is to run on three systems, each with a different branch predictor. All systems use a $2^{30}$ entry BHT. One system has a bimodal predictor, one system has a local predictor with a 12-outcome local history, and one system has a global predictor with a 12-outcome global history.

(a) Branch behavior is shown below. Notice that $B2$’s outcomes come in groups of three, such as $3$ $q$s. The first outcome of each group is random and is modeled by a Bernoulli random variable with $p = .5$ (taken probability is .5). The second and third outcomes are the same as the first. For example, if the first $q$ is $T$ the second and third $q$ will also be $T$. If the first $s$ is $N$, the second and third $s$ will also be $N$. Answer each question below, the answers should be for predictors that have already warmed up.

$B1$: T T T N N T T T N N ...
$B2$: r r r q q q s s s s u ...
$B3$: T T T T T T T T T T ...

☐ What is the accuracy of the bimodal predictor on branch $B1$?

☐ What is the approximate accuracy of the bimodal predictor on branch $B2$? ☐ Explain.

☐ What is the minimum local history size needed to predict $B1$ with 100% accuracy?

☐ What is the accuracy of the local predictor on branch $B2$, after warmup. ☐ Explain.

☐ What is the best local history size for branch $B2$, taking warmup into consideration. ☐ Explain.

☐ How many different GHR values will there be when predicting $B3$?
Problem 3, continued:

In this part consider the same predictors as on the previous page, except this time the BHT has $2^{14}$ entries. Also, consider the same branch patterns, they are repeated below, along with the address of branches $B1$ and $B2$. The branch predictors are part of a MIPS implementation.

$0x1234$: $B1$: T T T N N T T T N N ...
0x1242: $B2$: r r r q q q s s s s u ...
$B3$: T T T T T T T T T T T T ...

(b) Choose an address for branch $B3$ that will result in a BHT collision with branch $B1$.

☐ Address for $B3$ that results in a collision.

(c) How does the collision change the prediction accuracy of the bimodal predictor on the two branches?

☐ Change in $B1$ and ☐ Change in $B3$.

(d) (The answer to the following question does not depend on the sample branch patterns above.) Suppose we detect a BHT collision (perhaps by using tags). Why should we predict not taken?

☐ Reason for predicting not-taken for a collision.
Problem 4: (15 pts) The diagram below is for a 256 kiB \(2^{18}\) B set-associative cache. Hints about the cache are provided in the diagram.

(a) Answer the following, formulae are fine as long as they consist of grade-time constants.

- Fill in the blanks in the diagram.
- Complete the address bit categorization. Label the sections appropriately. (Index, Offset, Tag.)

Address: [31, 0]

- Associativity:
- Memory Needed to Implement (Indicate Unit!!):

- Line Size (Indicate Unit!!):

- Show the bit categorization for a **fully associative** cache with the same capacity and line size.

Address: [ ]
Problem 4, continued: The problems on this page are **not** based on the cache from the previous page. The code in the problems below run on a 4 MiB ($2^{22}$ byte) 4-way set-associative cache with a line size of 128 bytes.

Each code fragment starts with the cache empty; consider only accesses to the arrays.

(b) Find the hit ratio executing the code below.

```c
double sum = 0;
double *a = 0x2000000; // sizeof(double) == 8
int i;
int ILIMIT = 1 << 11; // = 2^{11}

for (i=0; i<ILIMIT; i++) sum += a[i];
```

What is the hit ratio running the code above? Show formula and briefly justify.

(c) Find the largest value for $BSIZE$ for which the second for loop will enjoy a 100% hit ratio.

```c
struct Some_Struct {
    double val;       // sizeof(double) = 8
    double norm_val;
    double a[14];
};

const int BSIZE = ; <- FILL IN
Some_Struct *b;
for ( int i = 0; i < BSIZE; i++ ) sum += b[i].val;
for ( int i = 0; i < BSIZE; i++ ) b[i].norm_val = b[i].val / sum;
```
Problem 5: (5 pts) The displacement in MIPS branches is 16 bits. Consider a new MIPS branch instruction, \texttt{bfeq rsn, rtn} (branch far), where \texttt{rsn} and \texttt{rtn} are 2-bit fields that refer to registers 4-7. As with \texttt{beq}, branch \texttt{bfeq} is taken if the contents of registers \texttt{rsn} and \texttt{rtn} are equal. With six extra bits \texttt{bfeq} can branch 64 times as far.

(a) Show an encoding for this instruction which requires as few changes to existing hardware as possible.

✓ Encoding for \texttt{bfeq}. ✓ Explain how minimizes changes.

The encoding for \texttt{bfeq} is shown below. The 22-bit displacement is \texttt{is3,it3,Immed} (in Verilog notation).

<table>
<thead>
<tr>
<th>Opcode</th>
<th>is3</th>
<th>rs2</th>
<th>rt2</th>
<th>Immed</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>21</td>
<td>15</td>
<td>0</td>
</tr>
</tbody>
</table>

In the encoding above the \texttt{rs} and \texttt{rt} register fields each have been shortened from five to two bits. (The original format 1 encoding is shown below.) Since the fields have been shortened but not moved the four remaining bits can be connected directly to the register file. (See the solution to the next part.)

MIPS I:

<table>
<thead>
<tr>
<th>Opcode</th>
<th>rs</th>
<th>rt</th>
<th>Immed</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>21</td>
<td>15</td>
</tr>
</tbody>
</table>

The encoding below, which would receive partial credit, would result in more costly implementations. This inferior encoding is certainly more organized with \texttt{rs2} and \texttt{rt2} next to each other and with the immediate occupying 22 contiguous bits. However the multiplexors at the register file inputs would need to be five bits instead of three bits. Notice that it does not make a difference whether or not the immediate bits are contiguous, as they are below, or split, as they are in the solution above.

Inferior \texttt{bfeq}:

<table>
<thead>
<tr>
<th>Opcode</th>
<th>rs2</th>
<th>rt2</th>
<th>Imm22</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>24</td>
<td>22</td>
</tr>
</tbody>
</table>
(b) Modify the pipeline below to implement the new instruction. Use as little hardware as possible.

Briefly show changes.

Changes appear below. At the register file address inputs, shown in blue, the high three bits of each register number is determined by a multiplexor. If a bfeq is present the upper inputs are used, making the upper three bits of each register number $001_2$, otherwise the upper bits come from the instruction. The lower two bits are taken from the instruction regardless of whether bfeq is present.

If a bfeq is present the displacement includes the extra six bits, these changes are shown in green.
Problem 6: (10 pts) Illustrated below is a dynamically scheduled four-way superscalar MIPS implementation and the execution of code on that implementation.

(a) On the diagram above indicate when each instruction will commit.

☑ Show commits on diagram above.

Commits shown above (they are indicated with a C). Note that commits must occur in program order and that there cannot be more than four commits per cycle. (Program order means that a commit for an instruction cannot occur before the commit for a preceding instruction.)

(b) What is the execution rate, IPC, for the code above for a large number of iterations assuming perfect branch prediction. Note that the system is dynamically scheduled.

☑ IPC for code above for large number of iterations.

Short answer: the code will execute at just 1 insn/cycle due to the add.s.

Longer answer: Notice that one source of the add.s instruction, f4, is the result of the add.s instruction in the prior iteration. This means that execution can go no faster than four cycles per iteration. The sample execution (the one in the unsolved problem, not just the solution) shows that there are bypass paths so the second iteration add.s can start as soon as the first iteration add.s result is ready. This suggests that four cycles per iteration is possible. Instruction fetch and decode goes much faster, one cycle per
iteration. Eventually though the ROB will fill causing IF to periodically stall. On average each iteration will take four cycles and each iteration consists of four instructions, for an IPC of \( \frac{4}{4} = 1 \).

(c) On the next page there is a table showing the values of selected signals during the execution of the code, the signals are related to register renaming. Show values where indicated on the table. Note that ID: incmb is already shown in cycle 1, show its values for later cycle(s).
Show values where indicated.

SOLUTION shown in blue.

LOOP:  Cycle 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
lwc1 f2, 0(r1)  IF ID Q RR EA ME WB C
add.s f4, f4, f2 IF ID Q RR A1 A2 A3 A4 WB C
bneq r1, r2, LOOP IF ID Q RR B WB C
addi r1, r1, 4 IF ID Q RR EX WB C

LOOP:  Cycle 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
lwc1 f2, 0(r1)  IF ID Q RR EA ME WB C
add.s f4, f4, f2 IF ID Q RR A1 A2 A3 A4 WB C
bneq r1, r2, LOOP IF ID Q RR B WB C
addi r1, r1, 4 IF ID Q RR EX WB C

# Cycle 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
ID:dstPR 0 (lwc1) 65 62 for register f2
ID:dstPR 1 (add.s) 97 69 for register f4
ID:dstPR 3 (addi) 60 79 for register r1

# Show values for signals below, including incmb.
# Cycle 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
ID:incmb 0 (lwc1) 83 65
ID:incmb 1 (add.s) 20 97
ID:incmb 3 (addi) 67 60

ID:rsPR 0 (lwc1) 67 60
ID:rsPR 1 (add.s) 20 97
ID:rsPR 3 (addi) 67 60

ID:rtPR 1 (add.s) 65 62 <- rt, not rs

# Cycle 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
WB:dstPR 0 (lwc1) 65 62
WB:dstPR 1 (add.s) 97 69
WB:dstPR 3 (addi) 60 79

# Cycle 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
Problem 7: (25 pts) Answer each question below.

(a) Describe how cost and performance limit the practical largest value of width (value of $n$) in an $n$-way superscalar implementation.

✓ Cost limiter.

Bypass paths become a major element when $n$ is large. Consider an $n$-way superscalar design derived from the 5-stage statically scheduled MIPS implementation used in class. The EX stage will have $n$ 2-input ALUs, requiring a set of bypass paths for each of the $2n$ inputs. The ME and WB stage will each hold $n$ results. The number of multiplexer inputs needed to provide bypass paths from those $2n$ results to the $2n$ inputs is $4n^2$. At 32- or 64 bits each, that will quickly dominate cost.

✓ Performance limiter.

There will be several performance limiters. One major limiter is the lack of sufficient instructions to execute in parallel. In a statically scheduled $n$-way superscalar design there cannot be true dependencies between the $n$ instructions in a group to avoid stalls. If there are dependencies then the group will spend two or more cycles in ID, reducing performance. The larger $n$ is the more frequently such dependencies will occur. Dynamic scheduling helps but does not eliminate the problem.

Another major performance limiter with large $n$ are branches. Branches occur frequently in integer code, perhaps once every five or six instructions. In practical designs execution could only reach up to the first taken branch in a group. (In impractical [academic] designs multiple branches can be predicted per cycle and instructions can be fetched from multiple non-adjacent areas per cycle).

(b) What is the most important factor in determining the size of a level 1 cache?

✓ Most important factor in L1 cache size.

Clock frequency and load latency. The amount of time it takes to retrieve data from a memory is a function of its size, the larger the slower. Typically designers would set a target for the clock frequency and for the number of cycles it would take to retrieve data from the L1 cache. The largest L1 size that can meet these requirements would be chosen. For example, suppose we chose two cycles for an L1 hit (the pipeline might have two memory stages, ME1 and ME2), and suppose we chose a clock period of 0.7 ns. That would give us 1.4 ns to retrieve data from the cache. We would choose the largest L1 size that could provide the data in 1.4 ns. The remaining chip area might be used for an L2 cache.
Suppose the 16-bit offset in MIPS `lw` instructions was not large enough. Consider two alternatives. In alternative 1 the offset in the existing `lw` instruction is the immediate value times 4. So, for example, to encode instruction `lw r1, 32(r2)` the immediate would be 8. In alternative 2 the behavior of the existing `lw` is not changed but there is a new load `lws r1, 32(r2)`, in which the immediate is multiplied by 4. Note that alternative 2 requires a new opcode. Which instruction should be added to a future version of MIPS?

☑ Should choose alternative 1 or alternative 2? ☑ Explain.

Alternative 2, `lws`, should be chosen so that existing software continues to run correctly.

The SPECcpu suite comes with the source code for the benchmark programs. How does that help with the goal of measuring new ISAs and implementations?

☑ Source code helps with testing new implementations because:

Since we have the source code we can compile the benchmarks ourselves. In fact, for SPECcpu testers are required to compile the code for themselves, using the compiler of their choosing (within reason). If we are testing a new implementation then we would want to use a compiler that can optimize for that implementation. If we are testing a new ISA then there would be no choice but to use a new compiler, since old compiler would not be able to generate code for it.

What’s the difference between a 4-way superscalar implementation (of say MIPS) and a VLIW system with a 4-slot bundle?

☑ Difference between superscalar and VLIW.

The superscalar implementation must be able to find and handle dependencies between any pair of instructions in a group (the group of 4 instructions traveling together through ID and beyond) and must be able to handle any instruction in any slot within a group. In contrast, the VLIW implementation need only handle a subset of possible instruction in each slot. For example, slot 0 might never have branch instruction and slot 3 might never have a memory instruction. Furthermore, dependencies between instructions in a bundle might be forbidden. The expectation is (was) that these differences would enable less expensive or higher performance VLIW implementations.