16.482 / 16.561: Computer Architecture and Design

Fall 2013

Midterm Exam Solution

1. (14 points) ***Computer performance***

We use two compilers to generate machine code for the same program, on the same processor, which has a 10 ns cycle time. The instruction breakdown is shown below:

Compiler	*ALU instructions*	*Loads*	*Stores*	*Branches*
1	*30*	*45*	*15*	*10*
2	*80*	*45*	*20*	*55*

Assume that ALU instructions take 1 cycle, loads and stores each take 4 cycles, and branches take 2 cycles. ***For all parts of this problem, show all work for full credit.***

a. (8 points) Calculate the average CPI for each code sequence.

Solution: Note that sequence 1 has 100 instructions, while sequence 2 has 200 instructions. The instruction frequency for each type can therefore be calculated as follows:

Seq	ALU instructions	Loads	Stores	Branches
1	30 / 100 = **0.3**	45 / 100 = **0.45**	15 / 100 = **0.15**	10 / 100 = **0.1**
2	80 / 200 = **0.4**	45 / 200 = **0.225**	20 / 200 = **0.1**	55 / 200 = **0.275**

Therefore:

CPI_1 = (0.3 * 1) + (0.45 * 4) + (0.15 * 4) + (0.1 * 2)
= 0.3 + 1.8 + 0.6 + 0.2 = **2.9**

CPI_2 = (0.4 * 1) + (0.225 * 4) + (0.1 * 4) + (0.275 * 2)
= 0.4 + 0.9 + 0.4 + 0.55 = **2.25**

Note: You could have just calculated the number of cycles directly (290 for sequence 1; 450 for sequence 2) and divided by the total instruction count to get average CPI.

b. (6 points) Which sequence runs faster? By how much?

Solution: If you calculated the total cycle count in part (a), this part is easy; otherwise, you need to look at the ratio of execution times:

ET_1 = (100 instructions) * (2.9 cycles/instruction) * (cycle time) = 290 * CT
ET_2 = (200 instructions) * (2.25 cycles/instruction) * (cycle time) = 450 * CT

Because both sequences run on the same machine, the cycle time is the same. We can therefore see that **sequence 1 is faster**, by a factor of 450 / 290 ≈ **1.55**

2. *(16 points)* ***Evaluating instructions***

For each part of the following question, assume the following initial state. Note that your answers to each part should use the values below—your answer to part (a), for example, should not affect your answer to part (b).

- *$s0 = 0x00100000, $t0 = 0x00000006, $t1 = 0x00000007*
- *Contents of memory (all values are in hexadecimal)*

Address				
0x00100000	0C	15	27	30
0x00100004	FF	27	DD	CC

*For each instruction sequence below, list **all** changed registers and/or memory locations and their new values. When listing memory values, list the entire word—for example, if a byte is written to 0x00100000, show the values at addresses 0x00100000-0x00100003.*

a. *(8 points)*

```
lh      $t2, 6($s0)
add     $t3, $t0, $t1
addi    $t4, $t1, -6
sub     $t5, $t3, $t4
```

Solution:

$t2 = sign-extended halfword at address 0x00100006 = **0xFFFFDDCC**
$t3 = $t0 + $t1 = 0x00000006 + 0x00000007 = **0x0000000D**
$t4 = $t1 + (-6) = 0x00000007 + (-6) = **0x00000001**
$t5 = $t3 - $t4 = 0x0000000D – 0x00000001 = **0x0000000C**

b. *(8 points)*

```
ori     $s1, $t0, 0xFFF0
sll     $s2, $s1, 16
sra     $s3, $s2, 16
sb      $s3, 2($s0)          (typo on exam—had $t8, not $s3)
```

$s1 = $t0 OR 0xFFF0 = 0x00000006 OR 0x0000FFF0
*= **0x0000FFF6***
*$s2 = $s1 << 16 = 0x0000FFF6 << 16 = **0xFFF60000***
*$s3 = $s2 >> 16 = 0xFFF60000 >> 16 = **0xFFFFFFF6***
*mem[2 + $s0] = mem[0x00100002] = lowest byte of $s3 = **0xF6***

3. (18 points) ***Binary multiplication***

*You are given A = -7 and B = 3. Assume each operand uses four bits. Show how the binary multiplication of A * B would proceed using Booth's Algorithm.*

	1	1	0	0	1						Multiplicand (-7)
	0	0	1	1	1						-Multiplicand (7)
	0	0	0	0	0	0	0	1	**1**	**0**	Initial product/multiplier
+	0	0	1	1	1						Step 1: Last 2 bits = 10 → add –Mcand,
	0	0	1	1	1	0	0	1	1	0	then shift right
→	0	0	0	1	1	1	0	0	**1**	**1**	Step 2: Last 2 bits = 11 → shift right
→	0	0	0	0	1	1	1	0	**0**	**1**	Step 3: Last 2 bits = 01 → add Mcand,
+	1	1	0	0	1						then shift right
	1	1	0	1	0	1	1	0	0	1	
→	1	1	1	0	1	0	1	1	**0**	**0**	Step 4: Last 2 bits = 00 → shift right
→	1	**1**	**1**	**1**	**0**	**1**	**0**	**1**	**1**	0	Final product (-21) in bold

4. (18 points) ***IEEE floating-point format***
For each part of this problem, show all work for full credit.

a. (9 points) Convert the decimal value 7.75 into single-precision floating-point format.

Solution: We first need to convert this value to binary and then normalize it:

$$7.75 = 111.11_2 = 1.1111 \times 2^2$$

We can directly determine each of the fields in our single-precision floating-point value:

Sign = 0 *(positive value)*
Exponent = [actual exponent] + bias = 2 + 127 = 129 = 10000001_2
Fraction = 1111_2 = $111\ 1000\ 0000\ 0000\ 0000\ 0000_2$ *(fraction is 23 bits)*

Therefore, as a single-precision floating-point value:

$$7.75 = 0100\ 0000\ 1111\ 1000\ 0000\ 0000\ 0000\ 0000_2 = \mathbf{0x40F80000}$$

b. (9 points) Convert the single-precision floating-point value 0xC1280000 into decimal.

Solution: We break the value given into the three fields of a single-precision floating-point value: sign (1 bit), biased exponent (8 bits), and fraction (23 bits):

$$0xC1280000 = 1100\ 0001\ 0010\ 1000\ 0000\ 0000\ 0000\ 0000_2$$

Sign = 1 *(negative value)*
Biased exponent = 10000010_2 = 130
→ Actual exponent = [Biased exponent] – bias = 130 – 127 = 3
Fraction = $010\ 1000\ 0000\ 0000\ 0000\ 0000_2 = 0101_2$

We can then write the magnitude as a normalized binary number, shift it into a binary form that is not normalized, and convert to decimal:

$$1.0101_2 \times 2^3 = 1010.1_2 = 10.5$$

Therefore, the single-precision floating-point value 0xC1280000 represents the decimal value **-10.5**.

5. (18 points) ***Pipelining***

Consider the following code sequence. Assume both branches are not taken—all instructions in the loop are executed:

```
loop:     add $s3, $s0, $s2
          lbu $t0, 0($s3)
          beq $t0, $zero, end
          addi $t1, $zero, 90
          slt $t2, $t1, $t0
          bne $t2, $zero, end
          sb $t1, 0($s3)
          addi $s2, $s2, 1
          sw $s2, 0($s1)
          j loop
end:      ...
```

For all parts of this problem, show all work for full credit.

a. (10 points) If we assume we have a pipelined datapath ***without forwarding****, how long will one loop iteration take?*

Solution: Without forwarding, we have to figure out where the dependences are and how many no-ops are necessary. Remember that dependent instructions must have at least two cycles between them; given this rule of thumb, we can see that the loop body should be rewritten with no-ops as follows:

```
loop:     add $s3, $s0, $s2
          nop
          nop
          lbu $t0, 0($s3)
          nop
          nop
          beq $t0, $zero, end
          addi $t1, $zero, 90
          nop
          nop
          slt $t2, $t1, $t0
          nop
          nop
          bne $t2, $zero, end
          sb $t1, 0($s3)
          addi $s2, $s2, 1
          nop
          nop
          sw $s2, 0($s1)
          j loop
```

5a (continued) The revised loop body now has 20 instructions—the original 10 plus 10 no-ops. To determine the number of cycles, you could draw a pipeline diagram, or remember that a program with N instructions running on an M-stage pipeline takes M + (N-1) cycles. In this case, M = 5 and N = 20, giving a total of 5 + (20-1) = **24 cycles**.

*b. (8 points) If we now assume a pipelined datapath **with forwarding**, how many cycles will the code take?*

Solution: Recall that forwarding removes most data hazards; the only one that cannot be completely removed occurs when a load instruction produces a result used in the very next instruction. That situation occurs once in this program and requires one no-op:

```
loop:     add $s3, $s0, $s2
          lbu $t0, 0($s3)
          nop
          beq $t0, $zero, end
          addi $t1, $zero, 90
          slt $t2, $t1, $t0
          bne $t2, $zero, end
          sb $t1, 0($s3)
          addi $s2, $s2, 1
          sw $s2, 0($s1)
          j loop
```

Using the same logic as above, this 11-instruction sequence takes 5 + (11-1) = **15 cycles**

6. (16 points) ***Dynamic branch prediction***

a. (10 points) Your processor executes a program containing the high-level code snippet below:

```
for (i = 0; i < 2; i++) {
     for (j = 0; j < 2; j++)
     { <body of loop> }
}
```

When compiled, this code contains two branches, as shown below. The BNE controls the end of the inner loop (with index variable `j`). The BEQ controls the end of the outer loop (with index variable `i`). You are given the addresses of each branch in both decimal and hexadecimal.

Address			
Decimal	Hex		
4	0x04	loop1	…
			…
16	0x10	loop2	…
			…
28	0x1C		BNE R4, R0, loop2
			…
56	0x38		BEQ R7, R8, loop1

Your processor contains a 2-bit branch history table with 8 entries. All entries are initially set to 11. Complete the table shown below, which tracks the predictions made by this predictor for the code above. Remember that "T" stands for "taken" and "NT" for "not taken".

Solution: Note that, to determine the BHT entry number in the 8-entry table, you must use the 3 lowest-order address bits that actually change—the lowest two bits of every instruction address are always 0. Therefore, the BNE at address 28 = 0001 1100$_2$ accesses entry 7, and the BEQ at address 56 = 0011 1000$_2$ accesses entry 6.

Students were responsible for completing the table entries with ***underlined, bold-faced font.***

Outer Loop Iteration	Inner Loop Iteration	Branch	BHT Entry #	BHT Entry Value	Pred.	Actual Outcome	New BHT Entry Value
1	1	BNE	***7***	11	T	T	11
1	2	BNE	***7***	11	T	NT	***10***
1	-	BEQ	***6***	11	T	T	11
2	1	BNE	***7***	***10***	***T***	T	***11***
2	2	BNE	***7***	***11***	***T***	NT	***10***
2	-	BEQ	***6***	11	T	NT	***10***

b. *(6 points) Assume you have a 4 entry (2,2) correlating predictor. For each part of this question, you are given a single line of the predictor, which is used to predict the given branch, as well as the current global history. For the given branch outcome, determine the prediction, new predictor state, and new global history.*

i. *Predictor entries:*

00	***10***	01	11

Current global history:

0	1

Branch outcome: Not taken

Solution: Given a global history of 01, the predictor will use the second entry in the row (the underlined entry above). That entry generates a **prediction of taken**, since its current state is 10.

Because the actual branch outcome is not taken, the **new predictor state is 00.**

The global history is based on the actual branch outcomes, with a 0 shifted in for a not taken branch. Therefore, the **new global history is 10.**

ii. *Predictor entries:*

10	01	11	00

Current global history:

0	0

Branch outcome: Taken

Solution: Given a global history of 00, the predictor will use the first entry in the row (the underlined entry above). That entry generates a **prediction of taken**, since its current state is 10.

Because the actual branch outcome is taken, the **new predictor state is 11.**

The global history is based on the actual branch outcomes, with a 1 shifted in for a taken branch. Therefore, the **new global history is 01.**