16.482 / 16.561: Computer Architecture and Design

Spring 2014

Midterm Exam Solution

1. *(12 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).

- *$t0 = 0x00000009, $t1 = 0x00000004, $t2 = 0x00200000*
- *Contents of memory (all values are in hexadecimal)*

Address				
0x00200000	*02*	*25*	*20*	*10*
0x00200004	*9A*	*A9*	*BC*	*CB*

*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 0x00200000, show the values at addresses 0x00200000-0x00200003.*

a. **Solution:**

```
addi    $t4, $t0, -3
  $t4 = $t0 – 3 = 9 – 3 = 0x00000006

sub     $t4, $t4, $t1
  $t4 = $t4 – $t1 = 6 - 4 = 0x00000002

xor     $t5, $t4, $t0
  $t5 = $t4 XOR $t0 = 0x00000002 XOR 0x00000009 =
0x0000000B

slt     $t6, $t1, $t5
  $t6 = 1 if $t1 < $t5; 0 otherwise
  Since $t1 = 4 and $t5 = 11 (0xB = 11₁₀), $t1 < $t5 →
$t6 = 1
```

b. **Solution:**

```
lbu     $s0, 4($t2)
  $s0 = zero-extended byte at mem[0x00200004] =
0x0000009A

ori     $s1, $s0, 0x8888
  $s1 = $s0 OR 0x8888 = 0x0000009A OR 0x00008888 =
0x0000889A

sll     $s2, $s1, 4
  $s2 = $s1 << 4 = 0x000889A0

sb      $s2, 2($t2)
  mem[0x00200002] = lowest byte of $s2 = 0xA0

  mem[0x00200000] = 0x0225A010 (changed byte underlined)
```

2. *(14 points)* ***Binary multiplication***

*You are given A = -2 and B = -6. Assume each operand uses four bits. Show how the binary multiplication of A * B would proceed using Booth's Algorithm.*

```
     1  1  1  1  0                     Multiplicand (-2)
     0  0  0  1  0                     -Multiplicand (2)

     0  0  0  0  0  1  0  1  0  0      Initial product/multiplier (multiplier = -6)
                                       Step 1: Last 2 bits = 00 → shift right

→    0  0  0  0  0  0  1  0  1  0      Step 2: Last 2 bits = 10 → add –Mcand,
+    0  0  0  1  0                             then shift right
   ----------------
     0  0  0  1  0  0  1  0  1  0

→    0  0  0  0  1  0  0  1  0  1      Step 3: Last 2 bits = 01 → add Mcand,
+    1  1  1  1  0                             then shift right
   ----------------
     1  1  1  1  1  0  0  1  0  1

→    1  1  1  1  1  1  0  0  1  0      Step 4: Last 2 bits = 10 → add –Mcand,
+    0  0  0  1  0                             then shift right
   ----------------
     0  0  0  0  1  1  0  0  1  0

→    0 [0  0  0  0  1  1  0  0] 1      Final product (12) in bold
```

3. (18 points) ***<u>IEEE floating-point format</u>***
Multiply the two IEEE single-precision floating-point values 0xC0000000 and 0xC0400000. For full credit, you must show all work, including:

- *Convert the two values into binary*
- *Perform the multiplication in binary (as done in Homework 2)*
- *Re-encode the result in IEEE single-precision format*

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

0xC0000000 = $1100\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2$
0xC0400000 = $1100\ 0000\ 0100\ 0000\ 0000\ 0000\ 0000\ 0000_2$

We can quickly see that the sign and exponent are the same for both numbers:

Sign = 1 *(negative value)*
Biased exponent = 10000000_2 = 128
→ Actual exponent = [Biased exponent] – bias = 128 – 127 = 1

The fractions are simple as well: 0 for the first value, and .100... for the second. We therefore have the following two values:

$-1.0_2 \times 2^1 = -2_{10}$
$-1.1_2 \times 2^1 = -3_{10}$

To multiply these numbers:

- Add the exponents to get the final exponent:
  - 1 + 1 = 2
- Multiply significands:
  - $1.0_2 * 1.1_2 = 1.1_2$
- Renormalize if necessary (not necessary in this case)
- Determine sign
  - Product of two negative values is positive → sign bit = 0
- To convert back to single-precision format:
  - Sign = 0
  - Biased exponent = actual exponent + bias = 2 + 127 = 129 = 10000001_2
  - Fraction = $100 \ldots 000_2$

Therefore, the final result is **$1.1_2 \times 2^2$ = 0x40C00000 in single-precision format = 6_{10}.**

4. (14 points) ***<u>Pipelining</u>***

Consider the following code sequence for both parts of this question.

```
loop:     add  $t0, $t1, $t2
          lw   $t3, 10($t0)
          lw   $t4, 14($t0)
          sub  $t5, $t4, $t3
          sw   $t5, 18($t0)
          addi $t2, $t2, 4
          slti $t6, $t2, 200
          bne  $t6, $zero, loop
```

<u>For both parts of this problem, show all work for full credit.</u>

a. *(8 points) If we assume we have a pipelined datapath* **without forwarding***, how many cycles will one loop iteration take?*

<u>Solution:</u> 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  $t0, $t1, $t2
          nop
          nop
          lw   $t3, 10($t0)
          lw   $t4, 14($t0)
          nop
          nop
          sub  $t5, $t4, $t3
          nop
          nop
          sw   $t5, 18($t0)
          addi $t2, $t2, 4
          nop
          nop
          slti $t6, $t2, 200
          nop
          nop
          bne  $t6, $zero, loop
```

The revised loop body now has 18 instructions—the original 8 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 = 18, giving a total of 5 + (18-1) = **22 cycles**.

4 (continued)
Again, consider the following code sequence:

```
loop:     add  $t0, $t1, $t2
          lw   $t3, 10($t0)
          lw   $t4, 14($t0)
          sub  $t5, $t4, $t3
          sw   $t5, 18($t0)
          addi $t2, $t2, 4
          slti $t6, $t2, 200
          bne  $t6, $zero, loop
```

b. *(6 points) If we now assume a pipelined datapath* ***with forwarding****, how many cycles will one loop iteration 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  $t0, $t1, $t2
          lw   $t3, 10($t0)
          lw   $t4, 14($t0)
          nop
          sub  $t5, $t4, $t3
          sw   $t5, 18($t0)
          addi $t2, $t2, 4
          slti $t6, $t2, 200
          bne  $t6, $zero, loop
```

Using the same logic as above, this 9-instruction sequence takes 5 + (9-1) = **13 cycles**

5. *(20 points)* ***Dynamic branch prediction***

a. *(14 points) Say you are executing a program that contains two branches, as shown below. You are given the addresses of each branch in both decimal and hexadecimal.*

```
    Address
Decimal  Hex
      8  0x08  loop  …
                     …
     20  0x14        BNE R4, R0, else
                     …
     44  0x2C        BEQ R7, R8, loop
```

Your processor contains a four-entry, 2-bit branch history table. Initially, entry 0 (the first line of the table) has the state 01, entry 1 is 11, entry 2 is 10, and entry 3 is 00.

Complete the table below to show which BHT entry is used to predict each branch, what predictions are made based on that entry, and how the state of each BHT entry changes throughout the program. You are given the actual outcome for each branch.

Solution: Note that, to determine the BHT entry number in the 4-entry table, you must use the 2 lowest-order address bits that actually change—the lowest two bits of every instruction address are always 0. Therefore, the BNE at address 20 = $0001\ \underline{01}00_2$ accesses entry 1, and the BEQ at address 44 = $0010\ \underline{11}00_2$ accesses entry 3.

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

Loop Iteration	Branch	BHT Entry #	BHT Entry State	Pred.	Actual Outcome	New BHT Entry State
1	BNE	**1**	**11**	**T**	NT	**10**
1	BEQ	**3**	**00**	**NT**	T	**01**
2	BNE	**1**	**10**	**T**	T	**11**
2	BEQ	**3**	**01**	**NT**	T	**11**
3	BNE	**1**	**11**	**T**	NT	**10**
3	BEQ	**3**	**11**	**T**	T	**11**
4	BNE	**1**	**10**	**T**	T	**11**
4	BEQ	**3**	**11**	**T**	NT	**10**

5 (continued)

b. (6 points) Assume you have a (2,2) correlating predictor in the state shown below:

00	*01*	*10*	*11*
10	*01*	*01*	*10*
11	*10*	***01***	*00*
01	*01*	*11*	*10*

1	*0*

If we have a branch at address 40 (0x28 in hex), what entry of the predictor will we access, and what will the prediction be? As part of your answer, circle the appropriate entry above, and briefly explain how we determine which entry to access.

Solution:

To determine which entry is accessed, we use the global history (shown at the bottom of the predictor) to choose a column (a specific BHT within the correlating predictor), and the address of the branch to choose a line within that BHT.

We can see that the global history is 10, which means that column 2 (highlighted in red) is the column used for this prediction. (We assume the leftmost column is column 0.)

As for the row, we need to choose the appropriate bits of the branch address. Note that each BHT has 4 = 2^2 rows, so 2 bits from the address are needed. Remember that we do not use the two least significant bits, which will always be 0 in a system with 32-bit instructions. We therefore choose the next two bits. If we look at the binary value of the branch address:

40 = 0x28 = 0010 **10**00

we can see that the bits used to choose a row are also 10, so we'll choose row 2, highlighted in blue above. The appropriate entry, which is equal to 01, is highlighted in purple. The branch will be predicted as **not taken**.

6. (22 points) ***<u>Dependences and dynamic scheduling</u>***
Answer the following questions about the code sequence below:

```
I0:       ADD.D     F4, F2, F6
I1:       L.D       F8, 0(R1)
I2:       L.D       F2, 8(R1)
I3:       MUL.D     F6, F8, F4
I4:       ADD.D     F4, F6, F2
I5:       DADDUI    R1, R1, #16
I6:       S.D       F4, 0(R1)
I7:       SLTI      R2, R1, #160
I8:       BNEZ      R2, I0
```

a. (8 points) List all true data dependences in this code. Assume the branch at the end of the loop is taken at least once. List your dependences in the form:

```
<register number>:<producing inst.> → <consuming inst.>
```

For example, a dependence involving R1 between "I2" and "I3" would be listed as:

```
R1: I2→I3
```

Your list should only contain true dependences—do not list any name dependences.

<u>Solution:</u> Note there are five loop-carried dependences, indicated below with "(LC)".

```
F4: I0→I3
F8: I1→I3
F2: I2→I4
F2: I2→I0 (LC)
F6: I3→I4
F6: I3→I0 (LC)
F4: I4→I6
R1: I5→I6
R1: I5→I7
R1: I5→I1 (LC)
R1: I5→I2 (LC)
R1: I5→I5 (LC)
R2: I7→I8
```

6 (continued)

b. *(9 points) Complete the pipeline diagram below to show how one iteration of the loop shown in part (a) is executed on a dynamically scheduled processor without speculation. Assume the following latencies, which refer to the number of execution cycles unless otherwise noted:*

- *3 cycles (1 EX, 2 MEM) for L.D and S.D*
- *2 cycles for ADD.D and SUB.D*
- *6 cycles for MUL.D*
- *2 cycles for DADDUI*
- *1 cycle for all other instructions*

Solution: The full pipeline diagram is below. Note that the S.D instruction presents an interesting case, as you must stall both the EX and memory stages for different reasons. The stall before EX is due to the wait for R1, which is used in the address computed in that stage. The stalls before M1 occur because the store needs the result of the second ADD.D, which finishes in cycle 15.

Inst.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
`ADD.D F4,F2,F6`	IF	IS	E1	E2	WB															
`L.D F8,0(R1)`		IF	IS	EX	M1	M2	WB													
`L.D F2,8(R1)`			IF	IS	EX	M1	M2	WB												
`MUL.D F6,F8,F4`				IF	IS	S	E1	E2	E3	E4	E5	E6	WB							
`ADD.D F4,F6,F2`					IF	IS	S	S	S	S	S	S	E1	E2	WB					
`DADDUI R1,R1,#16`						IF	IS	E1	E2	WB										
`S.D F4,0(R1)`							IF	IS	S	EX	S	S	S	S	M1	M2				
`SLTI R2,R1,#160`								IF	IS	EX	WB									
`BNEZ R2,Loop`									IF	IS	EX									

6 (continued)

c. *(5 points) Explain the difference between instruction completion and instruction commit. Why must instructions go through these two different cycles in a processor using speculative execution?*

Solution: Instruction completion occurs when an instruction finishes execution and writes its result on the common data bus. Instruction commit occurs when an instruction updates the permanent state of the processor, i.e., registers or memory. We separate these two stages in a speculatively executing processor so that instructions may still proceed before being conclusively marked as non-speculative (control dependent on a correctly predicted branch).