# 16.482 / 16.561: Computer Architecture and Design

Summer 2014

Midterm Exam Solution

1. *(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).*

- *$s1 = 0x00000005, $s2 = 0x00000003, $s3 = 0x00002000*
- *Contents of memory (all values are in hexadecimal)*

| ***Address*** | | | | |
|---|---|---|---|---|
| *0x00002014* | *BE* | *12* | *EF* | *33* |
| *0x00002018* | *06* | *05* | *20* | *14* |

*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 0x00002014, show the values at addresses 0x00002014-0x00002017.*

a.
```
sub     $t0, $s1, $s2
```
***$t0 = $s1 - $s2 = 5 – 3 = 0x00000002***

```
xori    $t1, $t0, 0xFFFF
```
***$t1 = $t0 XOR 0xFFFF = 0x00000002 XOR 0x0000FFFF***
***= 0x0000FFFD***

```
sh      $t1, 0x16($s3)
```
***mem[0x16 + $s3] = lowest halfword of $t1***
***→ mem[0x2016] = 0xFFFD***
***→ mem[0x2014] = BE12FFFD (last 16 bits of word changed)***

```
sll     $t2, $t1, 8
```
***$t2 = $t1 << 8 = 0x0000FFFD << 8 = 0x00FFFD00***

b.
```
addi    $t3, $zero, 0x2014
```
***$t3 = $zero + 0x2014 = 0 + 0x2014 = 0x00002014***

```
lh      $t4, 6($t3)
```
***$t4 = sign-extended halfword at mem[0x0000201A]***
***= 0x00002014***

```
and     $t5, $t4, $s1
```
***$t5 = $t4 AND $s1 = 0x0002014 AND 0x00000005 = 0x00000004***

```
slt     $t6, $t5, $s2
```
***$t6 = 1 if $t5 < $s2, = 0 otherwise***
***Since $t5 = 0x00000004 and $s2 = 0x00000003, $t6 = 0***

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

*You are given A = 6 and B = -3. Assume each operand uses four bits. Show how the binary multiplication of A * B would proceed using Booth's Algorithm.*

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 1 | 1 | 0 | | | | | | Multiplicand (6) |
| | 1 | 1 | 0 | 1 | 0 | | | | | | -Multiplicand (-6) |
| | | | | | | | | | | | |
| | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | **1** | **0** | Initial product/multiplier (multiplier = -3) |
| + | 1 | 1 | 0 | 1 | 0 | | | | | | Step 1: Last 2 bits = 10 → add –Mcand, |
| | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | then shift right |
| | | | | | | | | | | | |
| → | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | **0** | **1** | Step 2: Last 2 bits = 01 → add Mcand, |
| + | 0 | 0 | 1 | 1 | 0 | | | | | | then shift right |
| | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | |
| | | | | | | | | | | | |
| → | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | **1** | **0** | Step 3: Last 2 bits = 10 → add –Mcand, |
| + | 1 | 1 | 0 | 1 | 0 | | | | | | then shift right |
| | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | |
| | | | | | | | | | | | |
| → | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | **1** | **1** | Step 4: Last 2 bits = 11 → shift right |
| → | 1 | **1** | **1** | **1** | **0** | **1** | **1** | **1** | **0** | 1 | Final product (-18) in bold |

*3. (20 points)* ***IEEE floating-point format***

*Add the two IEEE single-precision floating-point values 0x41380000 and 0xc0980000. For full credit, you must show all work, including:*

- *Convert the two values into binary*
- *Perform the addition in binary, not decimal*
- *Re-encode the result in IEEE single-precision format*

**Solution:** 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), then convert each value:

0x41380000 = $0100\ 0001\ 0011\ 1000\ 0000\ 0000\ 0000\ 0000_2$
- Sign = 0 *(positive value)*
- Biased exponent = $10000010_2$ = 130
  - → Actual exponent = [Biased exponent] – bias = 130 – 127 = 3
- Fraction = $011\ 1000\ 0000\ 0000\ 0000\ 0000_2 = 0111_2$

0x41380000 = $1.0111_2 \times 2^3 = 11.5_{10}$

0xC0980000 = $1100\ 0000\ 1001\ 1000\ 0000\ 0000\ 0000\ 0000_2$
- Sign = 1 *(negative value)*
- Biased exponent = $10000001_2$ = 129
  - → Actual exponent = [Biased exponent] – bias = 129 – 127 = 2
- Fraction = $001\ 1000\ 0000\ 0000\ 0000\ 0000_2 = 0011_2$

0xC0980000 = $-1.0011_2 \times 2^2 = -4.75_{10}$

To add these numbers:

- Align binary points by shifting number with smaller exponent:
  - $-1.0011_2 \times 2^2 = -0.10011 \times 2^3$
- Add significands:
  - $1.0111_2 \times 2^3 + -0.10011 \times 2^3 = 0.11011 \times 2^3$
- Renormalize if necessary
  - $0.11011 \times 2^3 = 1.1011 \times 2^2$
- To convert back to single-precision format:
  - Sign = 0
  - Biased exponent = actual exponent + bias = 2 + 127 = 129 = $10000001_2$
  - Fraction = $1011\ ...\ 000_2$

Therefore, the final result is **$1.1011_2 \times 2^2$ = 0x40D80000 in single-precision format = $6.75_{10}$.**

*4. (14 points)* ***Pipelining***

*Consider the following code sequence for both parts of this question. Assume the use of a five-stage pipeline.*

```
lw   $s0, 0($t1)
lw   $s1, 4($t1)
add  $t0, $t3, $s0
sub  $t1, $s1, $s0
add  $t2, $t1, $t0
lw   $s2, 0($t2)
add  $t3, $t1, $t2
xor  $t4, $t3, $s1
sw   $s2, 4($t2)
```

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

*a. (8 points) If we assume we have a five stage pipelined datapath* ***without forwarding****, how many cycles will the instructions above 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:

```
lw   $s0, 0($t1)
lw   $s1, 4($t1)
nop
add  $t0, $t3, $s0
sub  $t1, $s1, $s0
nop
nop
add  $t2, $t1, $t0
nop
nop
lw   $s2, 0($t2)
add  $t3, $t1, $t2
nop
nop
xor  $t4, $t3, $s1
sw   $s2, 4($t2)
```

The revised loop body now has 16 instructions—the original 9 plus 7 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 = 16, giving a total of 5 + (16-1) = **20 cycles**.

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

```
lw   $s0, 0($t1)
lw   $s1, 4($t1)
add  $t0, $t3, $s0
sub  $t1, $s1, $s0
add  $t2, $t1, $t0
lw   $s2, 0($t2)
add  $t3, $t1, $t2
xor  $t4, $t3, $s1
sw   $s2, 4($t2)
```

*b. (6 points) If we now assume a five stage pipelined datapath **with forwarding**, how many cycles will the instructions above 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 does not occur in this program—there's always at least one cycle between each load and the instruction or instructions that depend on it. Therefore, 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* | | |
| *10* | *0x0A* | `loop` | … |
| | | | … |
| *56* | *0x38* | | `BEQ R4, R0, else` |
| | | | … |
| *72* | *0x48* | | `BNE R7, R8, loop` |

*Your processor contains an eight entry, 2-bit branch history table. Initially, entries 0-3 (the first four lines of the table) all have the state 01, and entries 4-7 (the last four lines of the table) all have the state 10.*

*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 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 BEQ at address 56 = 0011 1000$_2$ accesses entry 6, and the BNE at address 72 = 0100 1000$_2$ accesses entry 2.

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 | BEQ | ***6*** | ***10*** | ***T*** | T | ***11*** |
| 1 | BNE | ***2*** | ***01*** | ***NT*** | T | ***11*** |
| 2 | BEQ | ***6*** | ***11*** | ***T*** | NT | ***10*** |
| 2 | BNE | ***2*** | ***11*** | ***T*** | T | ***11*** |
| 3 | BEQ | ***6*** | ***10*** | ***T*** | T | ***11*** |
| 3 | BNE | ***2*** | ***11*** | ***T*** | T | ***11*** |
| 4 | BEQ | ***6*** | ***11*** | ***T*** | NT | ***10*** |
| 4 | BNE | ***2*** | ***11*** | ***T*** | NT | ***10*** |

*5 (continued)*

*b. (6 points) Assume you have a (3,2) correlating predictor in the state shown below:*

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| *00* | *00* | *00* | *00* | *11* | *01* | *10* | *10* |
| *10* | *01* | *01* | *10* | *10* | *01* | *01* | *01* |
| *11* | *11* | *10* | *00* | *01* | *10* | *10* | *10* |
| *01* | *01* | *11* | *11* | *00* | ***01*** | *00* | *11* |

| | | |
|---|---|---|
| *1* | *0* | *1* |

*If we have a branch at address 28 (0x1C 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 101, which means that column 5 (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:

28 = 0x1C = 0001 **11**00

we can see that the bits used to choose a row are 11, so we'll choose row 3, 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. (16 points)* ***Dependences***
*Answer the following questions about the code sequence below:*

```
I0:     L.D     F0, 0(R4)
I1:     ADD.D   F4, F0, F2
I2:     S.D     F4, 8(R4)
I3:     DIV.D   F6, F0, F6
I4:     MUL.D   F2, F4, F0
I5:     ADDI    R4, R4, 32
I6:     SUB.D   F4, F6, F2
I7:     S.D     F4, -16(R4)
I8:     BLT     R4, R3, 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.*

**Solution:** Loop-carried dependences are marked with (LC):

```
F0:  I0 → I1
F0:  I0 → I3
F0:  I0 → I4
F4:  I1 → I2
F4:  I1 → I4
F6:  I3 → I6
F6:  I3 → I3 (LC)
F2:  I4 → I6
F2:  I4 → I1 (LC)
R4:  I5 → I7
R4:  I5 → I8
R4:  I5 → I0 (LC)
R4:  I5 → I2 (LC)
F4:  I6 → I7
```

*6 (continued)*
*Again, consider the following code, and assume the branch at the end of the loop is taken at least once:*

```
I0:       L.D       F0, 0(R4)
I1:       ADD.D     F4, F0, F2
I2:       S.D       F4, 8(R4)
I3:       DIV.D     F6, F0, F6
I4:       MUL.D     F2, F4, F0
I5:       ADDI      R4, R4, 32
I6:       SUB.D     F4, F6, F2
I7:       S.D       F4, -16(R4)
I8:       BLT       R4, R3, I0
```

*b. (4 points) List all anti-dependences in this code.*

**Solution:** Loop-carried anti-dependences are marked with (LC).

```
F0:  I0 & I1 (LC)
F0:  I0 & I3 (LC)
F0:  I0 & I4 (LC)
F4:  I1 & I7 (LC)
F6:  I3 & I3 (LC)
F6:  I3 & I6 (LC)
F2:  I4 & I1
F2:  I4 & I6 (LC)
R4:  I5 & I0
R4:  I5 & I2
R4:  I5 & I7 (LC)
F4:  I6 & I2
F4:  I6 & I4
```

*c. (4 points) List all output dependences in this code.*

**Solution:** The only output dependence is: `F4:  I1 & I6`