16.482 / 16.561: Computer Architecture and Design

Spring 2015

Midterm Exam Solution

1. *(8 points)* ***<u>Evaluating instructions</u>***

Assume the following initial state prior to executing the instructions below. Note that the result of each instruction may depend on prior instructions.

- *`$t1 = 0x0000180C, $t2 = 0x00000409, $s0 = 0x00121000`*
- *Contents of memory (all values are in hexadecimal)*

Address	*Lo*			*Hi*
0x00121200	*11*	*CB*	*42*	*98*
0x00121204	*31*	*42*	*93*	*AA*

For each instruction below, list the changed register or memory location(s) and its new value. Note that constant values are in decimal unless specified as hexadecimal by a leading 0x.

```
sub   $t3, $t1, $t2
```

$t3 = $t1 - $t2 = 0x0000180C – 0x00000409 = <u>0x00001403</u>

```
andi  $t3, $t3, 0xF00F
```

$t3 = $t3 AND 0xF00F = 0x00001403 AND 0x0000F00F = <u>0x00001003</u>

```
sb    $t3, 0x203($s0)
```

Address = $s0 + 0x203 = 0x00121000 + 0x00000203 = 0x00121203
mem[0x00121203] = lowest byte of $t3 = 0x03
→ mem[0x00121200] = 0x11CB42<u>03</u> (changed byte underlined)

```
lhu   $t4, 0x206($s0)
```

Address = $s0 + 0x206 = 0x00121206
$t4 = zero-extended halfword from mem[0x00121206] = <u>0x000093AA</u>

```
slti  $s3, $t1, 0x2000
```

$s3 = 1 if $t1 < 0x2000; $s3 = 0 otherwise
$t1 = 0x0000180C → $t1 < 0x2000, so <u>$s3 = 1</u>

```
beq   $s3, $zero, L
```

Branch is taken if $s3 == $zero ($zero is always 0)
Since $s3 is 1, <u>branch is not taken</u>

```
sll   $t3, $t3, 8
```

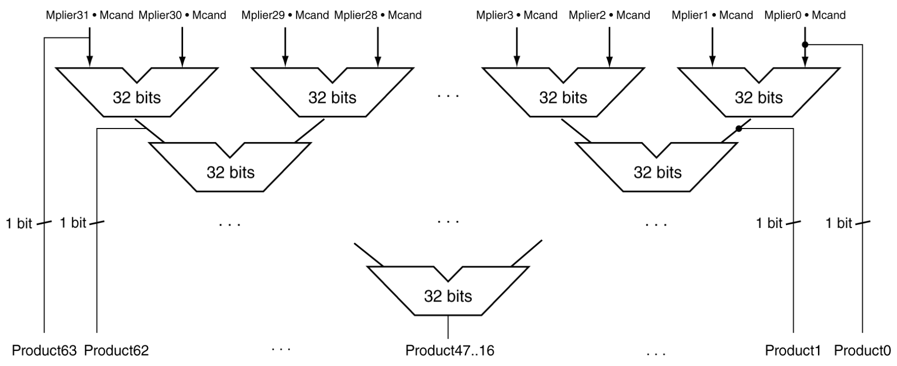
$t3 = $t3 << 8 = 0x00001003 << 8 = <u>0x00100300</u>

```
L: addi  $t1, $t1, 10
```

$t1 = $t1 + 10 = 0x0000180C + 0x0000000A = <u>0x00001816</u>

2. (16 points) ***Binary multiplication***

a. (6 points) The tree multiplier discussed in class is shown below:



Determine the time required for a tree multiplier to multiply two 64-bit numbers if each step of the operation takes 4 ns. (The figure above describes a 32-bit multiplier.) Note that some of the hardware may be able to operate in parallel.

Solution: The time required for this multiplier to complete is based on the number of stages in the multiplier—all adders in a given stage operate in parallel. Note also that the first stage consists of a series of AND gates to compute all partial products before adding them.

The number of adders depends on the number of bits being added. Since each adder in the first stage adds two partial products, and the number of partial products is equal to the number of bits, an n-bit multiplier will contain n/2 adders at the first level, each of which performs an n-bit addition. The next stage contains half as many adders, as you need one adder for every two results from the first stage. That pattern continues until the final stage, which is just a single level. From that description, you can see that the number of stages is $\log_2(n)$.

Therefore, in a 64-bit tree multiplier, there are $\log_2(64) = 6$ adder stages, and we also have to account for the initial set of AND gates to compute the partial products. The time for this multiplier can therefore be calculated as (7 steps) * (4 ns/step) = **28 ns**.

2 (continued)

b. *(10 points) You are given A = 5 and B = 7. Assume each operand uses four bits. Show how the binary multiplication of A * B would proceed using the iterating multiplier discussed in class.*

Solution: A = 5 = 0101_2 and B = 7 = 0111_2. We'll use A as the multiplicand and B as the multiplier in this solution.

Initially, **product/multiplier = $0000011\underline{1}$ (underlined bit determines next step)**

Step 1: LSB of product/multiplier = 1 → add multiplicand into left half of register & shift right

$$\begin{array}{rl} & 00000111 \\ + & \underline{0101} \\ & 01010111 \\ & 0010101\underline{1} \quad \leftarrow \text{Product/multiplier after shift} \end{array}$$

Step 2: LSB = 1 → add multiplicand into left half of register & shift right

$$\begin{array}{rl} & 00101011 \\ + & \underline{0101} \\ & 01111011 \\ & 0011110\underline{1} \quad \leftarrow \text{Product/multiplier after shift} \end{array}$$

Step 3: LSB = 1 → add multiplicand into left half of register & shift right

$$\begin{array}{rl} & 00111101 \\ + & \underline{0101} \\ & 10001101 \\ & 0100011\underline{0} \quad \leftarrow \text{Product/multiplier after shift} \end{array}$$

Step 4: LSB = 0 → shift right

$$\begin{array}{rl} & 00100011 \quad \leftarrow \text{Product/multiplier after shift} \end{array}$$

3. (20 points) ***<u>IEEE floating-point format</u>***

Add the two IEEE single-precision floating-point values 0x42960000 and 0x41ea0000. 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*

<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), then convert each value:

0x42960000 = $0100\ 0010\ 1001\ 0110\ 0000\ 0000\ 0000\ 0000_2$

Sign = 0 *(positive value)*

Biased exponent = 10000101_2 = 133

→ Actual exponent = [Biased exponent] – bias = 133 – 127 = 6

Fraction = $001\ 0110\ 0000\ 0000\ 0000\ 0000_2$ = 001011_2

0x42960000 = $1.001011_2 \times 2^6$ = 75_{10}

0x41ea0000 = $0100\ 0001\ 1110\ 1010\ 0000\ 0000\ 0000\ 0000_2$

Sign = 0 *(positive value)*

Biased exponent = 10000011_2 = 131

→ Actual exponent = [Biased exponent] – bias = 131 – 127 = 4

Fraction = $110\ 1010\ 0000\ 0000\ 0000\ 0000_2$ = 110101_2

0x41ea0000 = $1.110101_2 \times 2^4$ = 29.25_{10}

To add these numbers:

- Align binary points by shifting number with smaller exponent:
  - $1.110101_2 \times 2^4 = 0.01110101_2 \times 2^6$
- Add significands:
  - $1.001011_2 + 0.01110101_2 = 1.10100001_2$
- Renormalize if necessary *(not necessary in this case)*
- Final result = $\mathbf{1.10100001_2 \times 2^6}$ = **0x42d08000 in single-precision format** = 104.25_{10}

4. *(16 points)* ***<u>Datapaths and pipelining</u>***

a. *(4 points) Explain the differences between a single-cycle datapath and a pipelined datapath.*

<u>Solution:</u> In a single-cycle datapath, only one instruction executes at a time, and that instruction must go through all stages of execution before the next instruction is allowed to start. The clock cycle time in such a datapath is determined by the longest instruction (the instruction that uses the most datapath elements.)

In a pipelined datapath, instructions are split into multiple stages, with one instruction per stage executing (as long as there are no hazards). The clock cycle time is determined by the longest of these stages.

b. *(4 points) Explain how forwarding helps remove data hazards.*

<u>Solution:</u> A data hazard occurs when an instruction must wait for one of its operands because that operand is being produced by an instruction that has not yet made that result available. Forwarding helps remove these hazards by (1) making instruction results available sooner—usually at the end of the instruction's execute stage, rather than waiting until the write back stage—and (2) allowing instructions to read operands immediately before starting execution, rather than forcing operands to be read during decoding.

4 (continued)

c. *(8 points) Consider the following code sequence (see original exam; modified sequence shown below in solution):*

Determine the time required to execute this sequence on a processor with a basic 5-stage pipeline <u>without</u> forwarding. Assume the branch is not taken. Express your answer in cycles, not ns.

<u>Solution:</u> In a 5-stage pipeline without forwarding, dependent instructions must be separated by at least 2 cycles. Therefore, the following pairs of dependent instructions, which are fewer than 2 cycles apart, will cause data hazards. I'm listing the dependences in the format we used in HW 5:

- `$t1: 2nd lw → add`
    - Two other potential hazards (`$t0: 1st lw → add` and `$t1: 2nd lw → xor`) will be resolved by taking care of this hazard.
- `$t4: xor → sub`
    - Resolving this hazard also takes care of `$t3: add → sub`
- `$t5: sub → beq`
- `$s1: addi → sw`

In all cases, the dependent instructions are consecutive. To separate them by two cycles, 2 no-ops should be placed between those instructions, giving us the following code sequence:

```
lw   $t0, 0($s1)
lw   $t1, 4($s1)
nop
nop
add  $t3, $t0, $t1
xor  $t4, $t0, $t1
nop
nop
sub  $t5, $t3, $t4
nop
nop
beq  $t5, $zero, L1
addi $s1, $s1, 8
nop
nop
sw   $t3, 0($s1)
```

This sequence contains 16 instructions. As noted in class and in HW 3, a sequence of M instructions executing on a processor with N pipeline stages will take N + (M-1) cycles. This sequence will therefore take 5 + (16-1) = **20 cycles**.

5. *(21 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. Note that these branches are inside a loop, but neither one controls the number of loop iterations.*

```
    Address
Decimal  Hex
     56  0x38    BEQ $t0, $s0, label1
                 ...
     68  0x44    BNE $t5, $t6, label2
```

Your processor contains a thirty-two entry, 2-bit branch history table. Initially, entries 0-15 (the first sixteen lines of the table) all have the state 10, and entries 16-31 (the last sixteen lines of the table) all have the state 01.

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 32-entry table, you must use the 5 lowest-order address bits that actually change—the lowest two bits of every instruction address are always 0. Therefore, the BEQ at address 56 = $0\underline{011\ 10}00_2$ accesses entry 14, and the BNE at address 68 = $0\underline{100\ 01}00$ accesses entry 17.

Students were responsible for completing the table entries shown in **bold, underlined font.**

Loop Iteration	Branch	BHT Entry #	BHT Entry State	Pred.	Actual Outcome	New BHT Entry State
1	BEQ	**14**	**10**	**T**	T	**11**
1	BNE	**17**	**01**	**NT**	NT	**00**
2	BEQ	**14**	**11**	**T**	NT	**10**
2	BNE	**17**	**00**	**NT**	T	**01**
3	BEQ	**14**	**10**	**T**	T	**11**
3	BNE	**17**	**01**	**NT**	T	**11**
4	BEQ	**14**	**11**	**T**	NT	**10**
4	BNE	**17**	**11**	**T**	NT	**10**

5 (continued)

b. *(4 points) Determine the number of address bits required to choose the appropriate row in a (4,2) correlating branch predictor with 4096 entries. (Hint: 1024 = 2^{10}) Show all work to justify your answer.*

<u>**Solution:**</u> A correlating predictor is essentially a two-dimensional array, where the row is chosen by the appropriate bits from the instruction address and the column is chosen by the global history. You are given the total array size and must therefore determine the number of rows, which determines the number of address bits required to choose a row. Note that:

- The predictor contains 4096 = 2^{12} entries.
- A (4,2) predictor uses 4 bits of global history, which means the predictor has 2^4 = 16 columns.

Therefore, the predictor contains 4096 / 16 = $2^{12}/2^4 = 2^8$ = 256 rows.

The number of bits required to choose a row is $\log_2$(# rows). Since there are 2^8 rows, **8 bits are required** to choose a row in the predictor.

c. *(3 points) Describe a case in which a branch target buffer (BTB) would not hold the target address of the branch being predicted.*

<u>**Solution:**</u> Recall that the BTB contains a list of all previously calculated target addresses for executed branches. Therefore, the most likely reason the BTB would not hold a branch's target address is that the branch is being executed for the first time, and its target address has therefore never been calculated.

It's also possible that the branch uses a register as its target rather than a constant address, which makes the target address potentially different every time that branch executes. In these cases, the BTB is ineffective.

6. *(19 points)* ***<u>Dynamic scheduling</u>***

a. *(3 points) Explain how a dynamically scheduled processor without speculation determines if an instruction is allowed to write its result to the register file once it completes.*

<u>Solution:</u> The register result status table tracks which instruction was the most recent to specify each register as a destination. In a non-speculative processor, this table stores the reservation station name for that most recent instruction.

When an instruction completes, its reservation station name is compared against the status table entry for the destination register. If both match, the instruction writes its result to the register file. If they don't match, that means a later instruction will write the same register, making the earlier result obsolete.

b. *(3 points) Under what conditions would two instructions in a dynamically scheduled processor be allowed to start executing in the exact same cycle without causing <u>any</u> type of hazard?*

<u>Solution:</u> In order for two instructions to start executing without any hazards, they must satisfy the following conditions:

- Be independent instructions, thus avoiding potential data hazards.
- Execute in different functional units, thus avoiding potential structural hazards.
- Not depend on any in-flight branches, thus avoiding potential control hazards.

c. *(3 points) In a dynamically scheduled processor with speculation, why are registers renamed based on their reorder buffer entries, not the reservation stations to which the instructions are issued?*

<u>Solution:</u> In speculative processors, instruction results are stored in the reorder buffer between the write back and commit stages. Reservation stations are freed as soon as the instruction writes back, so the ROB is the only location in which the data exists during that time. Registers must therefore be renamed based on ROB entries, not reservation stations.

d. *(10 points) Complete the pipeline diagram below to show how the given code is executed on a dynamically scheduled processor <u>without speculation</u>. Assume the following latencies, which refer to the number of execution cycles unless otherwise noted, and assume that instructions listed separately use different functional units:*

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

Also, assume that the processor only contains one common data bus. Note that your solution may not use all 20 cycles shown below, but it should not use more than 20 cycles.

<u>Solution:</u> The pipeline diagram is shown below.

Inst.	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
`L.D F0,0(R1)`	IF	IS	EX	M	WB															
`L.D F2,8(R1)`		IF	IS	EX	M	WB														
`ADD.D F4,F0,F2`			IF	IS	S	E1	E2	E3	WB											
`DADDUI R1,R1,24`				IF	IS	EX	WB													
`SLT R2,R1,R4`					IF	IS	EX	WB												
`L.D F6,-8(R1)`						IF	IS	EX	M	WB										
`DIV.D F8,F4,F6`							IF	IS	S	E1	E2	E3	E4	E5	E6	E7	E8	WB		
`S.D F8,0(R1)`								IF	IS	EX	S	S	S	S	S	S	S	M		
`BNE R2,R0,Loop`									IF	IS	EX	WB								