16.482 / 16.561: Computer Architecture and Design

Fall 2014

Midterm Exam Solution

1. *(8 points)* ***Evaluating instructions***

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

- *$t4 = 0x0000000A, $t5 = 0x00000004, $t6 = 0x00101000*
- *Contents of memory (all values are in hexadecimal)*

Address	Lo			Hi
0x00101600	AA	14	92	44
0x00101604	08	22	11	13

For each instruction below, list the changed register or memory location(s) and its new value.

add $s0, $t4, $t5

$s0 = $t4 + $t5 = 0x0000000A + 0x00000004 = 0x0000000E

ori $s0, $s0, 0xCC11

$s0 = $s0 OR 0xCC11 = 0x0000000E OR 0x0000CC11 = 0x0000CC1F

sh $s0, 0x606($t6)

Address = 0x606 + $t6 = 0x00000606 + 0x00101000 = 0x00101606
mem[0x00101606] = lowest halfword in $s0 = 0xCC1F
→ mem[0x00101604] = 0x0822CC1F (changed bytes underlined)

lb $s1, 0x600($t6)

Address = 0x600 + $t6 = 0x00000600 + 0x00101000 = 0x00101600
$s1 = sign-extended byte from mem[0x00101600]
mem[0x00101600] = 0xAA → $s1 = 0xFFFFFFAA

slt $s2, $t4, $t5

$s2 = 1 if ($t4 < $t5), $s2 = 0 otherwise
$t4 = 0x0000000A and $t5 = 0x00000004 → $t4 > $t5 → $s2 = 0

bne $s2, $zero, L

Branch to L if ($s2 ≠ $zero)
Since $s2 = $zero = 0, branch is not taken

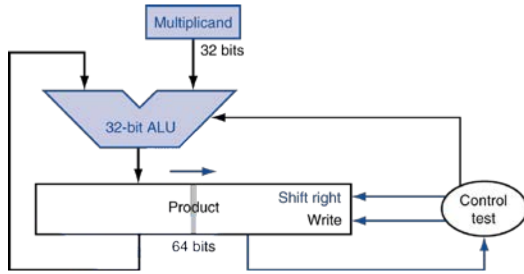
sra $s1, $s1, 8

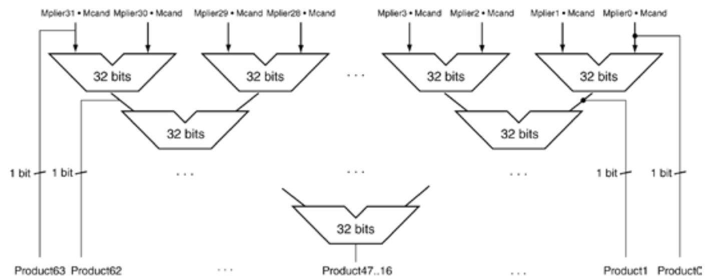
$s1 = $s1 >> 8 (keep sign) = 0xFFFFFFAA >> 8 = 0xFFFFFFFF

L: addi $s1, $s1, -1
$s1 = $s1 – 1 = 0xFFFFFFFF – 1 = 0xFFFFFFFE

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

a. *(6 points) The two multipliers discussed in class are shown below:*





Describe one major advantage of the optimized multiplier (on the left), and one major advantage of the tree multiplier (on the right).

Solution: The descriptions below assume the multiplication of two n-bit values:

The optimized multiplier uses less hardware than the tree multiplier—it requires only one adder, an n-bit register for the multiplicand, a 2n-bit shift register for the product/multiplier, and control logic. The tree multiplier also uses a 2n-bit register (to store the final result), the initial stage consists of n 2-bit AND gates, and the multiplier tree contains n-1 adders.

The tree multiplier is faster than the optimized multiplier—it takes $O(\log_2(n))$ cycles, as opposed to O(n) cycles for the optimized multiplier.

b. *(10 points) 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 the optimized multiplier.*

Solution: First of all, please note that this question was poorly formulated—I essentially asked you to do signed multiplication in a multiplier that can't properly handle signed multiplication. If you simply plug in $-6 = 1010_2$ and $3 = 0011_2$, you end up with 30 as your result if you follow the appropriate steps. Sign-extending your product/multiplier at each step doesn't help, either—your result will be 14 in that case.

The only way to therefore get a correct answer is to treat both values as positive and correct the signs after the fact. I've shown that solution below. However, because of the issues inherent in the problem, the grading was handled very, very leniently.

Initially, **product/multiplier = 00000011 (underlined bit determines next step)**

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

```
      00000011
  +   0110
      01100011
      00110001      ← Product/multiplier after shift
```

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

```
      00110001
  +   0110
      10010001
      01001000      ← Product/multiplier after shift
```

Step 3: LSB = 0 → shift right

```
      00100100      ← Product/multiplier after shift
```

Step 3: LSB = 0 → shift right

```
      00010010      ← Final result = 18
```

After negation, $-(00010010_2) = 11101101_2 + 1 = 11101110_2 = -18$

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

Multiply the two IEEE single-precision floating-point values 0x40900000 and 0xc1480000. For full credit, you must show all work, including:

- *Convert the two values into binary*
- *Perform the multiplication 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:

0x40900000 = $0100\ 0000\ 1001\ 0000\ 0000\ 0000\ 0000\ 0000_2$
Sign = 0 *(positive value)*
Biased exponent = 10000001_2 = 129
→ Actual exponent = [Biased exponent] – bias = 129 – 127 = 2
Fraction = $001\ 0000\ 0000\ 0000\ 0000\ 0000_2 = 001_2$
0x40900000 = $1.001_2 \times 2^2 = 4.5_{10}$

0xC1480000 = $1100\ 0001\ 0100\ 1000\ 0000\ 0000\ 0000\ 0000_2$
Sign = 1 *(negative value)*
Biased exponent = 10000010_2 = 130
→ Actual exponent = [Biased exponent] – bias = 130 – 127 = 3
Fraction = $100\ 1000\ 0000\ 0000\ 0000\ 0000_2 = 1001_2$
0xC0980000 = $-1.1001_2 \times 2^3 = -12.5_{10}$

To multiply these numbers:

- Add exponents
  - 2 + 3 = 5
- Multiply significands:
  - $1.001_2 * 1.1001_2 = 1.1100001_2$
- Renormalize if necessary (not necessary in this case)
- Determine sign
  - Positive * negative = negative
- To convert back to single-precision format:
  - Sign = 1 (negative value)
  - Biased exponent = actual exponent + bias = 5 + 127 = 132 = 10000100_2
  - Fraction = $1100001 \ldots 000_2$

Therefore, the final result is **$-1.1100001_2 \times 2^5$ = 0xC2610000 in single-precision format = -56.25_{10}**

4. *(16 points)* ***Datapaths and pipelining***

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

a. *(4 points) You are given two simple processors—one using a single-cycle design, the other a basic 5-stage pipeline—in which the instruction stages take the following amount of time:*

- *Fetch (IF): 20 ns*
- *Decode (ID): 25 ns*
- *Execute (EX): 30 ns*
- *Memory access (MEM): 40 ns*
- *Write back (WB): 25 ns*

How much time does each processor take to execute a single instruction, from start to finish?

Solution: In the single-cycle design, the cycle time—which is the time required to execute a single instruction—is the sum of all component stages: 20 + 25 + 30 + 40 + 25 = **140 ns**

In the pipelined design, the cycle time is determined by the longest stage (40 ns, for MEM), and each instruction takes 5 cycles, so the time for a single instruction is 5 * 40 = **200 ns**.

b. *(4 points) Now, assume each processor is redesigned with a faster ALU that reduces the time for the EX stage to 20 ns. How much time does a single instruction take on the redesigned processors? Explain your answer.*

Solution: Changing a single stage affects the time for the single-cycle design—since the EX stage is now 10 ns faster, the time per instruction is also 10 ns faster, making it **130 ns**.

Changing a single stage only affects the pipelined design if that stage is the longest cycle. Since the longest cycle in this processor is the MEM stage, the change to the EX stage does not affect the overall time for a single instruction—it's still **200 ns.**

4 (continued)

c. (8 points) Consider the following code sequence:

```
lw   $t0, 0($s1)
add  $t2, $t0, $s0
lw   $t1, 8($s1)
add  $t3, $t1, $s0
xor  $t4, $t2, $t3
sw   $t4, 16($s1)
and  $t5, $t2, $t3
sw   $t5, 16($s1)
```

Determine the time required to execute this sequence on the pipelined processor from part (b), assuming that processor uses a 5-stage pipeline with forwarding. Express your answer in ns, not cycles.

Solution: Recall that forwarding in a 5-stage pipeline resolves all potential data hazards except those due to a load instruction followed by a dependent add, which requires a single stall cycle between those instructions. This code sequence has two such pairs of instructions (the lw instructions followed by dependent adds), which essentially adds two no-op instructions.

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 = 10 (8 original instructions + 2 no-ops), giving a total of 5 + (10-1) = 14 cycles.

Since each cycle takes 40 ns on this processor, the total execution time is 14 * 40 = **560 ns**.

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*	
148	*0x94*	`BEQ $t0, $s0, label1`
		…
236	*0xEC*	`BNE $t5, $t6, label2`

Your processor contains a sixteen-entry, 2-bit branch history table. Initially, entries 0-7 (the first eight lines of the table) all have the state 10, and entries 8-15 (the last eight 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 16-entry table, you must use the 4 lowest-order address bits that actually change—the lowest two bits of every instruction address are always 0. Therefore, the BEQ at address 148 = $10\underline{01\ 01}00_2$ accesses entry 5, and the BNE at address 236 = $11\underline{10\ 11}00_2$ accesses entry 11.

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	**5**	**10**	**T**	T	**11**
1	BNE	**11**	**01**	**NT**	NT	**00**
2	BEQ	**5**	**11**	**T**	NT	**10**
2	BNE	**11**	**00**	**NT**	T	**01**
3	BEQ	**5**	**10**	**T**	NT	**00**
3	BNE	**11**	**01**	**NT**	NT	**00**
4	BEQ	**5**	**00**	**NT**	T	**01**
4	BNE	**11**	**00**	**NT**	T	**01**

5 (continued)

b. (4 points) How many entries are in an (8, 2) correlating branch predictor if the predictor uses 6 bits from each branch's address to choose the appropriate row within the predictor? Show all work to justify your answer.

Solution: 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. The number of instruction and global history bits therefore allows you to determine the size of the predictor:

- Since 6 bits are used to determine the row within the predictor, there are $2^6 = 64$ rows.
- The (8, 2) notation tells you that there are 8 global history bits and each entry in the predictor contains 2 bits. Given 8 global history bits, there are $2^8 = 256$ columns.

Therefore, the total number of entries in the predictor is the product of the number of rows and columns:

$64 * 256 = 2^6 * 2^8 = 2^{14} =$ **16384 entries**

c. (3 points) Explain the purpose of a branch target buffer (BTB).

Solution: A BTB stores the target addresses of recently taken branches. When a branch instruction is fetched, the BTB is checked to see if that branch's target has been previously calculated. If the branch is predicted as taken, the target address stored in the BTB is used to determine the address of the next instruction to be fetched.

6. *(19 points)* ***Dynamic scheduling***

a. *(3 points) What potential problems with dynamic scheduling are prevented using register renaming?*

Solution: Register renaming avoids issues that arise due to name dependences. Instructions that have no dataflow between them but use the same register names create potential hazards when those instructions are reordered. Specifically, anti-dependences, in which one instruction reads a register and a later instruction writes it, create potential write after read (WAR) hazards, while output dependences, in which two instructions write the same destination, create potential write after write (WAW) hazards. Both of these hazard types are prevented using register renaming.

b. *(3 points) Explain why instructions in dynamically scheduled processors do not have to execute the exact same stages for each instruction (for example, only loads and stores access memory, branches do not have a write back stage, etc.).*

Solution: In a basic 5-stage pipeline, all instructions use the exact same functional units as they progress through the five stages. However, in a dynamically scheduled processor, each instruction type essentially has its own execution pipeline. All instructions go through the same fetch (IF) and issue (IS) stages, but they are then issued to reservation stations, buffers associated with each type of functional unit that hold instructions waiting for their operands. Since the execution pipelines are separate, each one can be tailored to the specific types of operations that use that pipeline.

c. *(3 points) In a dynamically scheduled processor with speculation, why must instructions commit in order, even though they are allowed to complete out of order?*

Solution: Instructions are allowed to complete out of order to maintain the benefits of dynamic scheduling. However, each instruction is forced to commit (update the permanent state of the processor) in order so that the processor can easily recover from branch mispredictions. That process involves squashing all instructions that were incorrectly fetched after the mispredicted branch, which could not be done correctly if instructions were allowed to commit as soon as they finished computing their results.

a. *(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:*

- *2 cycles (1 EX, 1 MEM) for L.D and S.D*
- *3 cycles for ADD.D and SUB.D*
- *5 cycles for MUL.D*
- *2 cycles for DADDUI*
- *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 full pipeline diagram is below. Note that a potential structural hazard forces the SUB.D to stall its WB stage for one cycle, thus delaying the ADD.D and first S.D as well. Both stores can compute their addresses (in EX) immediately, but must stall their memory stages until the values to be stored are available.

Inst.	1	2	3	4	5	6	7	8	9	10	11	12	13	14
`L.D F2,0(R1)`	IF	IS	EX	M	WB									
`MUL.D F6,F2,F0`		IF	IS	S	E1	E2	E3	E4	E5	WB				
`L.D F4,8(R1)`			IF	IS	EX	M	WB							
`SUB.D F8,F4,F0`				IF	IS	S	E1	E2	E3	S	WB			
`ADD.D F10,F6,F8`					IF	IS	S	S	S	S	E1	E2	E3	WB
`S.D F10,-8(R1)`						IF	IS	EX	S	S	S	S	S	M
`S.D F8,16(R1)`							IF	IS	EX	S	M			
`DADDUI R1,R1,32`								IF	IS	E1	E2	WB		
`SLTI R2,R1,640`									IF	IS	S	EX	WB	
`BNE R2,R0,Loop`										IF	IS	S	EX	