Homework 3

Ben Lowman | CS 3330

February 17th, 2014

1. Show the truth table for a binary full adder.

A	B	Carry	Sum	$\text{Carry}_{\text{out}}$
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

2. Write the MIPS instructions necessary to move the results of an integer multiply into the $t0 and $t1 registers.

   If the values to be multiplied are also stored in $t0 and $t1

```
mult $t0 $t1
mfhi $t0
mflo $t1
```

3. Since multiply instructions take a long time to execute, a compiler often substitutes simpler instructions when possible. Give two examples of such a substitution.

   - Since shifting a number left is equivalent to multiplying by 2, some compilers will apply the appropriate numbers of shifts and adds instead of actually multiplying.
   - Compilers can construct look-up tables at compile time for complex operations. Looking up results (or partial results) in such a table is much faster than computing the value.

4. For floating point numbers describe the four different types of overflow that might occur.

   The IEEE 754 specification identifies five arithmetic exceptions for floating point numbers. I have listed four that I think might classify as "different types of overflow."

   - The exponent exceeds the maximum value.
   - The exponent is smaller than the minimum value.
   - A quantity cannot be expressed in the bits reserved for the mantissa (e.g. $\frac{1}{3}$)
   - Dividing by zero causes an infinite computation loop.

5. Describe how the floating point adder (Fig. 3.15 in the text) works.

   1. The exponents of the two numbers are compared; the smaller number is shifted to the right until its exponent matches the larger exponent.
   2. The significands are added.
   3. The sum is normalized by either shifting right and incrementing the exponent or shifting left and decrementing the exponent.
   4. If this causes overflow or underflow, then an exception is thrown.
   5. Else, the significand is rounded to the appropriate number of bits.
   6. If the significand is not normalized, return to step three.
   7. Else, the operation is completed.

6. Why do most computers support both signed and unsigned integers and arithmetic?

   Signed arithmetic allows for computer representation of negative numbers. While this representation is useful for some applications, it reduces the maximum magnitude of any value (approximately halves integer magnitude). Unsigned arithmetic is used when negative numbers are not needed and/or when larger magnitudes are needed (e.g. memory addresses are unsigned)

7. What is a co-processor? Give two examples.

   A coprocessor is a microprocessor designed to supplement the capabilities of the primary processor. MIPS uses two coprocessors, 0 and 1:

0 Handles virtual memory (This is mentioned in the text, although further details are noted to be in a chapter beyond currently covered material).

1 Contains the floating point unit and registers. This coprocessor arose from transistor limitations – both integer and floating point units could not be placed on the same chip. It was offered as an "add-on." to the main integer unit.

8. Text 3.22 to 3.24

   3.22 0x0C000000 = 0 00011000 0...0 = $1 \times 2^{24-127} = 2^{-103}$

   3.23 $63.25/2^5 = 1.9765625 \Rightarrow 63.25 = 1.9765625 \times 2^5$
   Normalized Exponent = -122 = 10000100 (two's compliment)
   Mantissa = 0.9765625 = 01000010011111010...0
   Sign bit = 0
   63.25 = 0 10000110 01000010011111010...0

   3.24 In this problem, only the exponent offset changes.
   Normalized exponent = -1018 = 10000000110 (two's compliment)
   63.25 = 0 10000000110 01000010011111010...0

9. Extra Credit (not required): the arithmetic hardware units described in the text support floating point add, multiply, and divide. Suppose you wanted to compute common functions like $\ln(x)$, $\sin(x)$, etc. with this hardware. How can you do it? Can you then make this a pseudo-instruction?

   One could use a Maclaurin series expansion to compute a function to the desired accuracy.

```
value = 0
count = 0

loop:
       branch computeTerm
continue:
       value = temp + value;
       if (temp < desiredAccuracy)
              branch end
       else
              count = count + 1
              branch loop
end:
```

```
14
15 .data
16 computeTerm:
17         temp = getTerm(count)
18         branch continue
```