# Homework #3

Due 11:59pm Tuesday, 27 September, 2016
Multiple submissions accepted.
25% penalty per day per full or partial.

Problem #1 (not from the text):

In two's complement math, for integers, in which of the following situations is overflow possible vs not possible? Put "P" or "NP" in each cell of the table accordingly. The rows represent a pair of binary numbers starting with "1", or "0" or one of each.

For example, the first cell is where you should indicate if the addition of two numbers whose most significant bit is "1" could result in overflow or not.

| | Add | Sub | Mul | Div |
|---|---|---|---|---|
| Both start with "1" | | | | |
| Both start with "0" | | | | |
| One of each | | | | |

Overflowing means a bit spills out the left side of the range you're expecting. For addition and subtraction, the operands are equal size and match the size of the field set aside to hold the answer.

**How** overflow occur for addition and subtraction differs – for addition, it's possible when the signs match, impossible when they differ.

For subtraction it's the opposite, because the subtraction reverses the sign of the second operand, leaving you with the "one of each" situation we had with addition. And if you had one of each, you won't after the subtraction reverses the second operand's sign.

For multiplication, you set aside space in your ALU that's as big as the multiplicand and multiplier combined, and even all 1's for both result in a number that fits within that space. For unsigned math, where all 32 bits are used for the magnitude and none for the sign bit, that's still true. Further, when you are dealing with signed numbers, you only multiply or divide positive numbers anyway, figuring out the sign separately, so you don't have to worry about space for sign bits anyway.

Remaining problems (from the text):

**2.20** [5] <§2.6> Find the shortest sequence of MIPS instructions that extracts bits 16 down to 11 from register $t0 and uses the value of this field to replace bits 31 down to 26 in register $t1 without changing the other 26 bits of register $t1.

**Hint**: The book has a solution that uses more than 6 commands to perform the needed tasks. I believe I found a solution that uses 3. Either will be accepted. And as a style hint, please represent binary strings (like, say, bit patterns for masking) as hexadecimal numbers. Thanks.
Also remember bits are always numbered with 31 on the left and 0 on the right.

**2.46** Assume for a given processor the CPI of arithmetic instructions is 1, the CPI of load/store instructions is 10, and the CPI of branch instructions is 3. Assume a program has the following instruction breakdowns: 500 million arithmetic instructions, 300 million load/store instructions, 100 million branch instructions.

Hint: Start off this multi-part problem by building a formula to calculate the total number of cycles required to get through all the instructions. You can use this basic formula from §1.6 of the text, p. 36 for each instruction type:

CPU time = Instruction count * CPI * Clock cycle time

Where we assume the clock cycle time unless otherwise stated begins at 1.

**2.46.1** [5] <§2.19> Suppose that new, more powerful arithmetic instructions are added to the instruction set. On average, through the use of these more powerful arithmetic instructions, we can reduce the number of arithmetic instructions needed to execute a program by 25%, and the cost of increasing the clock cycle time by only 10%. Is this a good design choice? Why?

**Hint**: Use the same formula as above, remembering to adjust clock cycle time.

**2.46.2** [5] <§2.19> Suppose that we find a way to double the performance of arithmetic instructions. What is the overall speedup of our machine? What if we find a way to improve the performance of arithmetic instructions by 10 times?

**Hint**: The question is about the situation in 2.46, not 2.46.1.
What does doubling the performance mean? Use Amdahl's Law to calculate your answer. That will ensure you get a number expressed as a multiplier of the original speed. For example, if your math with Amdahl's law results in 1.54, then you know you're dealing with a system that's 54% faster. Conversely, if you know you're 54% faster, then Amdahl's law should give you 1.54.  Express your speedup in this fashion.

Furthermore, this example from Wikipedia was helpful, I thought:

We are given a sequential task which is split into four consecutive parts: P1, P2, P3 and P4 with the percentages of runtime being 11%, 18%, 23% and 48% respectively. [So P1 = .11, P2 = .18, and so on.]

Then we are told that P1 is not sped up, so S1 = 1, while P2 is sped up 5×, P3 is sped up 20×, and P4 is sped up 1.6×. By using the formula P1/S1 + P2/S2 + P3/S3 + P4/S4, we find the new sequential running time is:

$$\frac{0.11}{1} + \frac{0.18}{5} + \frac{0.23}{20} + \frac{0.48}{1.6} = 0.4575.$$

or a little less than $^1/_2$ the original running time. Using the formula $(P1/S1 + P2/S2 + P3/S3 + P4/S4)^{-1}$, the overall speed boost is 1 / 0.4575 = 2.186, or a little more than double the original speed. Notice how the 20× and 5× speedup don't have much effect on the overall speed when P1 (11%) is not sped up, and P4 (48%) is sped up only 1.6 times.

In that wikipedia example, the answer we'd look for when asked about speedup is 2.186.

**2.47** Assume that for a given program

70% of the executed instructions are arithmetic,
10% are load/store, and
20% are branch

**Hint**: use the same equation from 2.46:

CPU time = Instruction count * CPI * Clock cycle time

Assume clock cycle time is 1, which you should always do unless told otherwise.

**2.47.1** [5] <§2.19> Given this instruction mix and the assumption that an arithmetic instruction requires 2 cycles, a load/store instruction takes 6 cycles, and a branch instruction takes 3 cycles, find the average CPI.

**Hint**: you know that for a program with some number N instructions, .7N are math instructions, and each of them takes 2 cycles, and so on. I expressed my # of instructions simply as the percentage given.

We add up the cycles to get 2.6

**2.47.2** [5] <§2.19> For a 25% improvement in performance, how many cycles, on average, may an arithmetic instruction take if load/store and branch instructions are not improved at all?

**Hint**: you're looking for a new value for the arithmetic instruction's CPI that reduces the total cycles to 75% of what it was before. Solve for that variable.

**2.47.3** [5] <§2.19> For a 50% improvement in performance, how many cycles, on average, may an arithmetic instruction take if load/store and branch instructions

are not improved at all?
**Hint**: Now you're looking for a value for the CPI of arithmetic instructions that will drop the average CPI to 50% of what it was originally.

**3.1** [5] <§3.2> What is 5ED4 – 07A4 (subtraction) when these values represent unsigned 16-bit hexadecimal numbers? The result should be written in hex. Show your work!

**3.2** [5] <§3.2> What is 5ED4 – 07A4 when these values represent signed 16-bit hex numbers stored in sign-number format. What is the result now? Answer in hex, show your work.

**3.6** [5] <§3.2> Assume 185 and 122179 are unsigned 8-bit decimal integers. Calculate 185-122. Is there overflow, underflow or neither?

**3.7** [5] <§3.2> Assume 185 and 122 are signed 8-bit decimal integers stored in **sign-magnitude** format. Calculate 185 + 122. Overflow, underflow, or neither?

**Hint**: Sign-magnitude numbers are not 2's complement numbers. Just toggle the sign bit, for negative numbers, and the remaining represent the magnitude. If the answer fits in 7 bits, then you don't have overflow. Use the 8th bit for sign.

**3.8** [5] <§3.2> Assume 185 and 122 are signed 8-bit decimal integers stored in sign-magnitude format. Calculate 185-122. Is there overflow, underflow or neither?