**Homework 3**

**3.1.) How can integer divisions and multiplications by a constant that is a power of 2 be made**

**fast?**

Through the process of replacing the complex multiplication arithmetic with less complex shifting of bits. One bit to the left represents a number twice as large in base 2, shifting the bits left has the same effect as multiplying by a power of 2. Compiler very good about optimizing code.

**3.2) What benefit do SSE (i.e., vector) instructions provide over regular (i.e., scalar) instructions?**

SSE instructions provide the architecture with increased register space that allow multiple floating point values (data) to be stored and operated on simultaneously through a single instruction (SIMD).This subword parallelism results in vector instructions (SIMD) performing faster than scalar instructions.

**3.3) Describe in words what happens when an overflow occurs while adding two negative twos-complement values.**

When adding two negative numbers produces a positive result. That is illogical and means a carry occurred in the 32-bit. A value is there instead instead of its sign. Some compilers throw exceptions: saving the pc in epc register, jumping with error message, and returning to epc after corrective action.

**3.4) Why does the presented ALU compute an xor result even for non-xor instructions?**

Because the xor hardware is always “on”, i.e., it always performs this computation, even when not needed. A MUX later chooses this result or some other result to send to the ALU’s output. This is the cheapest and fastest way to implement an ALU.

**3.5) When rounding the real value -7.5 to an integer value, show what the result is for each of the**

**following four rounding modes. round down, round up, truncate, round to nearest even**

**round down:** -8

**round up:** -7

**truncate:** -7

**nearest even:** -8

**3.6) In an array of positive floating-point values that are sorted in decreasing order and that need**

**to be summed up, it is better to start summing from the end of the array. Explain why.**

Better to start from the end. When summing floating-point values you want to minimize numerical error. In order to do so you want to sum as many small values as possible first to give them an opportunity to reach the magnitude of the larger numbers that will cause error.

**3.7) When the hardware normalizes a result of 0.00012 after a floating-point computation, in**

**which direction does the mantissa have to be shifted, by how many bit positions, and how**

**does the exponent have to be corrected (assuming no overflow or underflow of any kind)?**

The mantissa has to be shifted to the left by four bits and the exponent has to be decremented by four.

**3.8) Explain what a NaN is?**

NaN – not a number – an undefined result of invalid operations

Ex:

1. / 0.0

infinity – infinity

sqrt(-5.6)

Is often used by programmers to run code but delay decisions and assessments until it is convenient.

**3.9) Why is a 32-bit floating-point addition generally slower than a 32-bit integer addition?**

Floating point addition requires multiple clock cycles for each FP operation (equalize exponents, add mantissas, normalize result), whereas integer addition only takes one cycle. Attempting to combine all operations under one clock cycle would result in penalizing the clock cycle of all instructions. Therefor FP adder takes more cycles (slower).

**3.10) Write, in hexadecimal format, the content of a MIPS single-precision floating-point register**

**holding the value -4.510.**

 0xC0900000