2017-10-24

- Integer addition and subtraction:
  - When adding operands with different signs, overflow cannot occur.
  - When we subtract operands of the same sign we end up by adding operands of different signs.
  - Unsigned integers are commonly used for memory addresses where overflows are ignored.
  - Add (*add*), add immediate (*addi*), and subtract (*sub*) cause exceptions on overflow.
  - Add unsigned (*addu*), add immediate unsigned (*addiu*), and subtract unsigned (*subu*) do not cause exceptions on overflow.
  - Because C ignores overflows, the MIPS C compilers will always generate the unsigned versions of the arithmetic instructions *addu*, *addiu*, and *subu*, no matter what the type of the variables.
  - MIPS detects overflow with an **exception**, also called an **interrupt** on many computers.
  - The address of the instruction that over owed is saved in a register, and the computer jumps to a predefined address to invoke the appropriate routine for that exception.
  - **Exception program counter (EPC)**: the address of the instruction that caused the exception.
  - **Saturation**: when a calculation over ows, the result is set to the largest positive number or most negative number, rather than a modulo calculation as in two's complement arithmetic.
- Integer multiplication:
  - Data flows from top to bottom to resemble more closely the paper-and-pencil method.
  - The least significant bit of the multiplier (Multiplier 0) determines whether the multiplicand is added to the Product register.
  - The left shift in step 2 has the effect of moving the intermediate operands to the left, just as when multiplying with paper and pencil.
  - The shift right in step 3 gives us the next bit of the multiplier to examine in the following iteration.
  - These three steps are repeated 32 times to obtain the product.
  - The speed-up comes from performing the operations in parallel: the multiplier and multiplicand are shifted while the multiplicand is added to the product if the multiplier bit is a 1.
  - Almost every compiler will perform the strength reduction optimization of substituting a left shift for a multiply by a power of 2.
  - For signed multiplication, first convert the multiplier and multiplicand to positive numbers

and then remember the original signs.
    - Faster multiplications are possible by essentially providing one 32-bit adder for each bit of the multiplier.
    - MIPS instructions: **multiply (mult)**, **multiply unsigned (multu)**, **multiply (mul)**.
    - To fetch the integer 32-bit product, the programmer uses **move from low (mflo)** and **move from high (mfhi)**.
- Integer division:
    - Each iteration of the algorithm needs to move the divisor to the right one digit, so we start with the divisor placed in the left half of the 64-bit Divisor register and shift it right 1 bit each step to align it with the dividend.
    - The **SRT division** technique tries to predict several quotient bits per step, using a table lookup based on the upper bits of the dividend and remainder.
    - MIPS instructions: **divide (div)**, **divide unsigned (divu)**.
- Floating-point representation: $x = (-1)^{\text{sign}} \times (1 + \text{fraction}) \times 2^{\text{exponent}-\text{bias}}$
    - **Single precision**: sign (1 bit) + exponent (8 bits) + fraction (23 bits). Bias = 127.
    - **Double precision**: sign (1 bit) + exponent (11 bits) + fraction (52 bits). Bias = 1023.
    - **Overflow**: the exponent is too large to be represented in the exponent field.
    - **Underflow**: the nonzero fraction become so small that it cannot be represented.
    - Exponents 000…0 and 111…1 reserved.
    - Placing the exponent before the significand also simplifies the sorting of floating-point numbers using integer comparison instructions.
    - **Infinity**: exponent = 111…1; fraction = 000…0.
    - **Not-a-Number (NaN)**: exponent = 111…1; fraction ≠ 000…0.
- Floating-point addition:
    - Align binary points.
    - Add significands.
    - Normalize result & check for over/underflow.
    - Round and renormalize if necessary.
- Floating-point multiplication:
    - Add exponents.
    - Multiply significands.
    - Normalize result & check for over/underflow.
    - Round and renormalize if necessary.
    - Determine sign.
- The MIPS designers decided to add separate floating-point registers called $f0, $f1, $f2, etc, used either for single precision or double precision.

- A double precision register is really an even-odd pair of single precision registers, using the even register number as its name.
- Floating-point load and store instructions: **lwc1, ldc1, swc1, sdc1**.
- The benefits of separate floating-point registers are having twice as many registers without using up more bits in the instruction format, having twice the register bandwidth by having separate integer and floating-point register sets, and being able to customize registers to floating point.
- IEEE 754 always keeps two extra bits on the right during intermediate additions, called **guard** and **round**, respectively.
- **Units in the last place (ULP)**: accuracy in floating point is normally measured in terms of the number of bits in error in the least significant bits of the significand.
- **Subword parallelism**: By partitioning the carry chains within a 128-bit adder, a processor could use *parallelism* to perform simultaneous operations on short vectors of sixteen 8-bit operands, eight 16-bit operands, four 32-bit operands, or two 64-bit operands. An example of **data-level parallelism**, **vector parallelism**, or **single instruction, multiple data (SIMD)**.
- Left shift instruction can replace a negative integer multiply by a power of 2, but a right shift is *not* the same as a negative integer division by a power of 2.
- Floating-point addition is not associative.
- Parallel execution strategies that work for integer data types does *not* work for floating-point data types.
- The MIPS instruction add immediate unsigned (*addiu*) sign-extends its 16-bit immediate field.