Chapter 3

Arithmetic

- Digits are added bit-by-bit from right to left, with carries passed to the next digit to the left
- Subtraction is simply addition with the appropriate operand negated before the addition
- Overflow occurs when the result from an operation cannot be represented with the available hardware
    - Overflow cannot occur in addition with numbers of differing signs because the result is no larger than one of the operands
        - Therefore, in subtraction, overflow cannot occur when the signs are the <u>same</u>
    - Adding/subtracting two 32-bit numbers can yield a result that needs 33 bits
        - We don't have 3 bits, so that means when overflow occurs, the sign bit is set with the value of the result instead of the proper sign

Overflow

- Therefore, overflow occurs when adding two positive numbers and the sum is negative, or vice-versa
    - In subtraction, it occurs when we subtract a negative number from a positive number and get a negative result or subtract a positive number from a negative number and get a positive result
- The above is for signed (Two's Complement) overflow. For unsigned overflow, we usually choose to ignore it, therefore MIPS includes two types of arithmetic instructions:
    - *add, addi,* and *sub* cause exceptions on overflow (signed)
    - *addu, addiu,* and *subu* do *not* cause exceptions on overflow
- C always chooses to ignore overflow so the compiler will always generate the unsigned versions of the instructions
- If we were to detect overflow, MIPS would do so with an exception/interrupt, which is an unscheduled event that disrupts program execution
    - The address of the instruction that overflowed is saved in a register and the computer jumps to a predefined address to invoke the appropriate routine for that exception
        - MIPS includes a register called exception program counter (EPC) that contains that address of the instruction that caused the exception so that the software can return to it if it wants to
            - This also leads to the existence of \$k0 and \$k1 registers reserved for the operating system so that it can be used when jumping back to the offending instruction and resetting all general-purpose registers without using a general purpose register to jump back with

Floating Points

- We also need support for fractions (real numbers) and not just signed and unsigned numbers
- A number like 3,155,760,000 is too large to be stored in 32 bits, but can be represented in a more compact form using scientific notation in the form of 3.15576×10^9
    - A normalized number is a number in floating-point notation that has no leading 0s
- We can also show binary numbers in scientific notation
- Floating point refers to computer arithmetic that represents numbers in which the binary point is not fixed
    - C use s the name *float* for such numbers

Fields

- Must find a compromise between the size of the fraction and the exponent
    - Increasing the size of the fraction enhances the precision of the faction
    - Increasing the size of the exponent increases the range of numbers that can be represented
- Floating point numbers are of the form: $(-1)^S \times F \times 2^E$
- For a 32 bit floating point number, we have:
    - *Sign (1 bit)*: sign of the fraction (1 if it is negative, 0 if it is positive)
    - *Exponent (8 bits)*: Value of the exponent; includes sign
    - *Fraction* (*23 bits)*: This is a 23-bit number

Range

-
    - These sizes give an extraordinary range:
        - Fractions can be almost as small as 2.0×10^{-38} and numbers almost as large as 2.0×10^{38}
- Numbers can still be too large
    - Overflow occurs when exponent is too large to be represented in the exponent field
- We also have issues of numbers being too small
    - **Underflow** occurs when the negative exponent has become too large to fit in the exponent field

Double Precision

- We can reduce the number of overflow/underflow cases by introducing a double precision floating point arithmetic
    - It is represented in two 32-bit words (therefore it is 64 bits)
    - In C, this number is called a *double*
- In double-precision, the breakdown is the following:
    - *Sign*: 1 bit
    - *Exponent*: 11 bits
    - *Fraction*: 20 bits
    - This thus allows numbers as small as 2.0×10^{-308} and as large as 2.0×10^{308}
- These formats are all part of the *IEEE 754 floating-point standard* that is virtually universal to all computers

Normalized

- IEEE 754 makes the leading 1-bit of the normalized binary numbers implicit

- There is leading 1-bit in the normalized version because in the normalized version of scientific notation, there is only 1 digit to the left of the decimal point. Because this is binary, the only number to the left of the binary point is a 1. Therefore, to save 1 bit we can make this implicit.
- The significand is therefore 24 bits for single-precision and 53 bits for double-precision
  - Significand refers to these lengths whereas fraction refers to the 23-bit and 52-bit numbers

Special Cases

- 0 has no implicit 1, so to represent this, the fraction must be all 0 and the exponent must be all 0
- For the rest of the numbers, the modified representation includes the hidden 1: $(-1)^S \times (1 + \text{Fraction}) \times 2^E$
  - Numbering the bits of the fraction from left to right s_1, s_2, s_3, ..., then the value is:
    - $(-1)^S \times (1 + (s_1 \times 2^{-1}) + (s_2 \times 2^{-2}) + (s_3 \times 2^{-3}) + \ldots) \times 2^E$
- An exponent of 255 (the largest exponent) and a fraction of 0 yields either positive or negative infinity
  - Programmers can use this in instances such as dividing by 0 instead of interrupting or throwing an error
- An exponent of 255 (the largest exponent) and a nonzero fraction produces a NaN (Not a Number)
  - This is used as a result of invalid operations, such as 0/0 or subtracting infinity from infinity
- An exponent of 0 and a nonzero fraction is a de-normalized number
  - In this case, there is no implicit 1, which goes hand in hand with the definition of a de-normalized number
    - Therefore, we are able to reach even smaller numbers, because previously the smallest number was $(1 + 0.00...00) \times 2^{-126}$ because of the implicit 1
    - Once taken out, with denorms, the smallest number is then $0.00....01 \times 2^{-126} = 2^{-23} \times 2^{-126}$

Bias Encoding

- The designers wanted a floating-point representation that could easily be processed by integer comparisons, especially sorting
  - The most significant bit is the sign bit which allows for a quick test of less than, greater than, or equal to 0
  - The exponent bit comes before the significand because numbers with bigger exponents look larger than numbers with smaller exponents as long as both exponents have the same sign
    - We can't use two's complement here because negative exponents have a 1 in the most significant bit of the exponent field, which makes it look like a big number when it's actually the opposite
    - Instead, we encode the exponent with bias encoding
      - Represent the most negative exponent as all 0s
      - The most positive exponent as all 1s
        - For single precision, use a bias of 127

- The real form is then $(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent - Bias})}$
- Because the exponent of 255 is reserved for NaN's and infinity, the biggest exponent we can have is 254 which yields an upper bound of 127 on the exponent and a lower bound of -126