Summary Chapter 3 of Computer Architecture and Design.

Chapter 3: Arithmetic for Computers.

3.1 - Intro

This chapter will explain how computers are able to represent numbers, arithmetic algorithms, and hardware that relate to these algorithms and how instruction sets play a role in this.

3.2 – Addition and Subtraction

Addition and subtraction are done from right to left using bits, as most people were taught in school. If you have two ones adding each other, a zero is placed in that column and a one is brought to the next left column. This happens until there is nothing more to calculate and the final answer has been reached. As for subtraction, it is done directly from right to left, or using the two's compliments representation (adding a positive number).

When discussing overflow, overflow cannot occur when adding operands with different signs (adding positive and negative numbers). As for subtraction the same rule applies. Overflow cannot occur when one is subtracting operands with opposite operands.

However, overflow will occur when adding two positive numbers and the answer appears to become negative. As for subtraction, overflow will occur when subtracting two negative numbers and the answer results to be positive.

By using different types of arithmetic instructions, it is possible to ignore and avoid overflow for unsigned integers.

add, addi, and sub will cause exceptions with overflow.
addu, addiu, and subu will not cause exceptions with overflow.

Different compilers will use the different instructions in order to detect or not detect overflow.

3.3 – Multiplication

For multiplication we have 3 standard operands: multiplicand(n), multiplier(m), and product.

Multiplicand
x multiplier

Product

We can multiply by taking the individual numbers of the multiplier and and multiplying them one by one to the digits of the multiplicand. We will then be left with a product.

The product by itself will be n+m bits long. This is due to the fact that that that (n+m) is able to hold all possible solutions.

If you have both a multiplicand and multiplier that are 32 bits, you will need a 64 bit product register because (32+32) = 64.

If we break down the paper-pencil multiplying method down, we can figure out it can sometimes take 3 cycles to calculate one number of the whole equation.

Therefore it is important to operate these numbers in parallel to speed up the process and cut some of the time.

When multiplying a positive and negative number, we negate the signs and proceed with the process mentioned above. Afterwards, we add the negative sign back to the product; granted we are only working with 32 bit numbers.

Due to Moore's law, and the speed of hardware development, it is now possible to expand into even faster multiplication by using 32bit adders per bit of the multiplier.

It is even possible to multiply faster than five add times due to carry save adders and the possibility to pipeline these operations.

MIPS are able to set two 32 bit registers that will hold the 64 bit product, this is called Hi and Lo. MIPS will use two instructions mult and multu each with their own respective function: properly signed and unsigned products.

3.4 - Division

With division we also have set operands. The dividend is getting divided by the divisor. The answer that follows is the quotient, and what is left is called the remainder.

The actual algorithm for division is to subtract the amount of the divisor off of the dividend and to keep a count of this. The computer will first check that the divisor is smaller or equal to the dividend. If the answer is one or greater, it will register a 1 and subtract: dividend – divisor. If the answer becomes negative, the computer will revert the action, and place a 0. It will then register that number and the remainder as the quotient.

When working with signed division, the easiest way to do this is by memorizing the signs of the numbers. If one is negative and the other positive the computer would negate the quotient. If they are both the same, you let the answer be positive.

It would be possible to further optimize this by doing the shifting and subtraction at the same time. It would also be possible to use the SRT Devision technique, using a lookup table which guesses the answer.

When dividing in MIPS, the only requirements are the Hi Lo registers (64bit) and an ALU that can add and subtract. For signed and unsigned, MIPS uses these two instructions respectively: div and divu.

3.5 – Floating Point

Computers are also able to support different type of numbers, which aren't whole numbers. The mathematical term for these numbers is: reals. We mostly use scientific notation when identifying large numbers. Scientific notation is the manner of writing a single number before the decimal point, followed by an amount of numbers times 10^(x).

If there are no numbers behind the decimal point, but there is still use of scientific notation we call this a normalized number.

To normalize a binary number we use a base of 2 so we can increase or decrease the number of bits the number must be shifted by. Due to the base not being 10, we call the name for the new decimal point a binary point.

Floating point (commonly referred to as float) is the name of these numbers which are currently supported in computer arithmetic.

When designing floating-point representations, there must be a good balance between the fraction and exponent. The fraction (also known as a manthissa) is the numbers between 0 and 1.

The exponent is the where the value of the floating point is placed.

Floating point numbers hold the common form (-1)^s * F * 2^B where F is the fraction value and B is the exponent amount.

In a MIPS arithmetic, floating points can be as small as 2.0* 10^(-38) and as large as 2.0 * 10^(38), however there are limits which means overflow is possible (too large of an exponent value).

The exact opposite of overflow can happen, which is underflow. Underflow is then the negative exponent value is too large, and the number doesn't fit into the exponent field.

To lower the chances of both over and underflow, a double or better known as double precision floating point can be used. That is when a floating point value is represented in two 32 bit words. On the other hand, a single precision floating point is when a floating point value is represented as a single 32 bit word.

Doubles allow for numbers to range from 2 *10^(308) and 2 * 10^(-308) values. The main purpose of this is to try and offer better precision. Due to them being so large, they now fall under IEEE 745 floating-point standards.

Due to the fact that 0 has no leading 1, it was given the value 0 and so 00 represents 0.

Some saved operations are there NaN (Not a Number) for later testing

Negative exponents can be an issue when storing sometimes, because double negatives, would eventually make a simple short number very large. Using the biased notation, we have to make sure the subtracting number needs to be unsigned to determine its real value. This is the way how floating points are subtracted.

The way how floating point adds is very special. It starts by comparing two numbers and checking if the decimals are in the same position. If this is not the case, it shift the decimal to the left until the numbers than the same scientific notation.

After this it proceeds to normally add the significants. When that if complete, it rounds off the number and adds the scientific notation back to it.

If it was an addition of a negative number, the algorithm repeats steps 3 and 4 again, and checks for no overflow or underflow possibility.

Multiplying in floating point:

The algorithm starts with calculating the exponent of the final answer by simply adding the two exponents of the two numbers. When adding biased numbers, you must subtract the bias from the sum of the added exponent numbers.

The second step is to multiply the significands of the numbers by each other. The product then gets the correct decimal placing that we calculated above.

We then have to finish by properly signify the answer (dropping more than 3 digits from the decimal point). We then finish by properly normalizing the answer, and adding the correct scientific notation.

To test for over or underflow, the product can be shifted by adding 1 to the exponent. When thinking about the sign of the product, it depends on the original number's signs. If they were double negative the answer is positive, and if they are opposite, the product is a negative number.

Between two numbers, there can be an infinite amount of answers, while a fixed number only has one answer. For this reason, floating point numbers are approximations rather than the actual number itself. To greater the chance of approximation, IEEE754 allows the programmer to pick the desired approximation.

It is very important to keep two extra bits on the right when doing intermediate actions. This is know as a guard and round.

A guard: the two extra bits kept on the right during intermediate actions to improve the rounding accuracy.

A round: when you make the intermediate floating-point result fit the format of a floating point. The purpose is to find the two closest numbers.

Accuracy in floating point is determined with ulp (units in the last place). This means that in the worst case, the algorithm will round up to the number that is within one-half ulp, and use that as the number; granted there are no over- underflows, and invalid operation exceptions.

3.6 Parallelism

When computer architects recognized that different computer components calculate the same type of things, they decided to implement parallelism, which allows for a wide arrange of operands to be performed simultaneously. This growing technique can be called data level parallelism.

NEON a multimedia instruction, supports almost all data types besides 64 bit floating point numbers.

3.7 Real Stuff: Steaming SIMD Extensions and Advanced Vector Extensions in x86

With Intel's greatest developments, it is now possible to allow multiple floating-point operands to be packaged in a single 128 bit SSE2. This operation can operate on four singles or two doubles.

3.8 Going faster
This subsection shows that using Intel's new technology, and features to speed up their CPU's, it is possible to make computation and processing much faster.

3.9 Fallacies and Pitfalls

Most of the problems with arithmetic arise when there have been miscalculations. For instance trying to calculate the difference between two numbers, but interpreting the bits wrong.

This also holds when talking about two integers. You must keep your associativity, even if an overflow is prompting. This concept however, doesn't count for floating point numbers. Due to floating point numbers having limited precision which then turns into close to real approximations, they don't suffer from this. Specially floating point addition, due to it being not associative.

With this however, programmers who write parallel code, have to make sure that the results with floating point numbers are credible; something called Numerical Analysis.

End of Summary.