Unit 7 Arithmetic Logic Unit

Structure:

7.1 Introduction

In this unit we focus on the most complex aspect of ALU and control unit which are the main components of the processing unit. We have seen in the preceding units the task that CPU must do for execution of an instruction. That is it fetches instruction, interprets it, fetches data, processes data, and finally writes result into appropriate location. Internal CPU bus is needed to transfer data between the various registers and the ALU, because the ALU in fact operates only on data in the internal memory of CPU that is registers.

Objectives:

By the end of Unit 7, you should be able to:

1. Explain the ALU
2. Discuss the different number representations.

7.2 Arithmetic Logic Unit

The ALU is the part of the CPU that actually performs arithmetic and logical operations on data. All of the other elements of the computer system - control unit, registers, memory, I/O - are there mainly to bring data into ALU for it to process and then take the results back out.

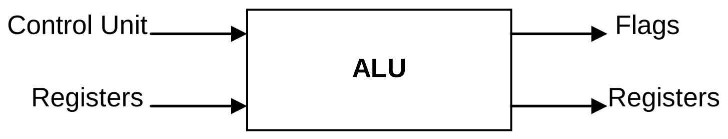

Figure 7.1: ALU Inputs and Outputs

The inputs and outputs of ALU are shown in figure 7.1. The inputs to the ALU are the control signals generated by the control unit of CPU, and the registers of the CPU where the operands for the manipulation of data are stored. The output is a register called status word or flag register which reflects the result and the registers of the CPU where the result can be stored. Thus data are presented to the ALU in registers, and the results of an operation are also stored in registers. These registers are connected by signal paths to the ALU. ALU does not directly interact with memory or other parts of the system (e.g. I/O modules), it only interacts directly with registers. An ALU like all other electronic components of a computer is based on the use of simple digital devices that store binary digits and perform Boolean logic operations.

The control unit is responsible for moving data to memory or I/O modules. Also, it is the control unit that signals all the operations that happen in the CPU. The operations, functions and implementation of Control Unit will be discussed in the tenth unit.

In this unit we will concentrate on the ALU. An important part of the use of logic circuits is for computing various mathematical operations such as addition, multiplication, trigonometric operations, etc. Hence we will be discussing the arithmetic involved in using ALU.

First, before discussing the computer arithmetic we must have a way of representing numbers as binary data.

7.3 Number Representations

Computers are built using logic circuits that operate on information represented by two valued electrical signals. We label the two values as 0 and 1; and we define the amount of information represented by such a signal as a *bit* of information, where bit stands for ***binary digit***. The most natural way to represent a number in a computer system is by a string of bits, called a binary number. A text character can also be represented by a string of bits called a character code. We will first describe binary number representations and arithmetic operations on these numbers, and then describe character representations.

Non-negative Integers

The easiest numbers to represent are the non-negative integers. To see how this can be done, recall how we represent a number in the decimal system. A number such as 2034 is interpreted as:

$$2*10^3 + 0*10^2 + 3*10^1 + 4*10^0$$

But there is nothing special with the base 10, so we can just as well use base 2. In base 2, each digit value is either 0 or 1, which we can represent, for instance, by false and true, respectively.

In fact, we have already hinted at this possibility, since we usually write 0 and 1 instead of false and true.

All the normal algorithms for decimal arithmetic have versions for binary arithmetic, except that they are usually simpler.

Negative Integers

Things are easy as long as we stick to non-negative integers. They become more complicated when we want to represent negative integers as well.

In binary arithmetic, we simply reserve one bit to determine the sign. In the circuitry for addition, we would have one circuit for adding two numbers, and another for subtracting two numbers. The combination of signs of the two inputs would determine which circuit to use on the absolute values, as well as the sign of the output.

While this method works, it turns out that there is one that is much easier to deal with by electronic circuits. This method is called the **'two's complement'** method. It turns out that with this method, we do not need a special circuit for subtracting two numbers.

In order to explain this method, we first show how it would work in decimal arithmetic with infinite precision. Then we show how it works with binary arithmetic, and finally how it works with finite precision.

Infinite-Precision Ten's Complement

Imagine the odometer of an automobile. It has a certain number of wheels, each with the ten digits on it. When one wheel goes from 9 to 0, the wheel immediately to the left of it advances by one position. If *that* wheel already showed 9, it too goes to 0 and advances the wheel to *its* left, etc.

Now suppose we have an odometer with an *infinite number of wheels*. We are going to use this infinite odometer to represent all the integers.

When all the wheels are 0, we interpret the value as the integer 0.

A positive integer *n* is represented by an odometer position obtained by advancing the rightmost wheel *n* positions from 0. Notice that for each such positive number, there will be an infinite number of wheels with the value 0 to the left.

A negative integer *n* is represented by an odometer position obtained by decreasing the rightmost wheel *n* positions from 0. Notice that for each such negative number, there will be an infinite number of wheels with the value 9 to the left.

In fact, we don't need an *infinite* number of wheels. For each number only a finite number of wheels are needed. We simply assume that the leftmost wheel (which will be either 0 or 9) is duplicated an infinite number of times to the left.

While for each number we only need a finite number of wheels, the number of wheels is *unbounded*, i.e., we cannot use a particular finite number of wheels to represent *all* the numbers. The difference is subtle but important (but perhaps not that important for this particular course). If we need an infinite number of wheels, then there is no hope of ever using this representation in a program, since that would require an infinite-size memory. If we only need an unbounded number of wheels, we may run out of memory, but we can represent a lot of numbers (each of finite size) in a useful way. Since any program that runs in finite time only uses a finite number of numbers, with a large enough memory, we might be able to run our program.

Now suppose we have an addition circuit that can handle non-zero integers with an infinite number of digits. In other words, when given a number starting with an infinite number of 9s, it will interpret this as an infinitely large positive number, whereas our interpretation of it will be a negative number. Let us say, we give this circuit the two numbers ...9998 (which we interpret as -2) and ...0005 (which we interpret as +5). It will add the two numbers. First it adds 8 and 5 which gives 3 and a carry of 1. Next, it adds 9 and the carry 1, giving 0 and a carry of 1. For all remaining (infinitely many) positions, the value will be 0 with a carry of 1, so the final result is ...0003.

This result is the correct one, even with our interpretation of negative numbers. You may argue that the carry must end up somewhere, and it does, but in infinity. In some ways, we are doing arithmetic modulo infinity.

Some implementations of some programming languages with arbitrary precision integer arithmetic (*Lisp* for instance) use exactly this representation of negative integers.

Finite-Precision Ten's Complement

What we have said in the previous section works almost as well with a fixed bounded number of odometer wheels. The only problem is that we have to deal with *overflow* and *underflow*.

Suppose we have only a fixed number of wheels say 3. In this case, we shall use the convention that if the leftmost wheel shows a digit between 0 and 4 inclusive, then we have a positive number, equal to its representation. When instead the leftmost wheel shows a digit between 5 and 9 inclusive, we have a negative number, whose absolute value can be computed with the method that we have in the previous section.

We now assume that we have a circuit that can add *positive* three-digit numbers, and we shall see how we can use it to add negative numbers in our representation.

Suppose, again we want to add -2 and +5. The representations for these numbers with three wheels are 998 and 005 respectively. Our addition circuit will attempt to add the two positive numbers 998 and 005, which gives 1003. But since the addition circuit only has three digits, it will truncate the result to 003, which is the right answer for our interpretation.

A valid question at this point is in which situation our finite addition circuit will not work. The answer is somewhat complicated. It is clear that it always gives the correct result when a positive and a negative number are added. It is incorrect in two situations. The first situation is when two positive numbers are added, and the result comes out looking like a negative number, i.e. with a first digit somewhere between 5 and 9. You should convince yourself that no addition of two positive numbers can yield an overflow and still look like a positive number. The second situation is when two negative numbers are added and the result comes out looking like a non-negative number, i.e. with a first digit somewhere between 0 and 4. Again, you should convince yourself that no addition of two negative numbers can yield an underflow and still look like a negative number.

We now have a circuit for addition of integers (positive or negative) in our representation. We simply use a circuit for addition of only positive numbers, plus some circuits that check:

- If both numbers are positive and the result is negative, then report overflow.
- If both numbers are negative and the result is positive, then report underflow.

Finite-Precision Two's Complement

So far, we have studied the representation of negative numbers using ten's complement. In a computer, we prefer using base two rather than base ten. Luckily, the exact method described in the previous section works just as well for base two. For an *n*-bit adder (*n* is usually 32 or 64), we can represent positive numbers with a leftmost digit of 0, which gives values between 0 and $2^{(n-1)} - 1$, and negative numbers with a leftmost digit of 1, which gives values between $-2^{(n - 1)}$ and -1.

Exactly the same rule works for overflow and underflow detection. If, when adding two positive numbers, we get a result that looks negative (i.e. with its leftmost bit 1), then we have an overflow. Similarly, if, when adding two negative numbers, we get a result that looks positive (i.e. with its leftmost bit 0), then we have an underflow.

Rational Numbers

Integers are useful, but sometimes we need to compute with numbers that are not integer.

An obvious idea is to use *rational* numbers. Many algorithms, such as the simplex algorithm for linear optimization, use only rational arithmetic whenever the input is rational.

There is no particular difficulty in representing rational numbers in a computer. It suffices to have a pair of integers, one for the numerator and one for the denominator.

To implement arithmetic on rational numbers, we can use some additional restrictions on our representation. We may, for instance, decide that:

- Positive rational numbers are always represented as two positive integers (the other possibility is as two negative numbers),
- Negative rational numbers are always represented with a negative numerator and a positive denominator (the other possibility is with a positive numerator and a negative denominator),
- The numerator and the denominator are always relative prime (they have no common factors).

Such a set of rules makes sure that our representation is *canonical*, i.e., that the representation for a value is unique, even though, a priori, many representations would work.

Circuits for implementing rational arithmetic would have to take such rules into account. In particular, the last rule would imply dividing the two integers resulting from every arithmetic operation with their largest common factor to obtain the canonical representation.

Rational numbers and rational arithmetic is not very common in the hardware of a computer. The reason is probably that rational numbers don't behave very well with respect to the size of the representation. For rational numbers to be truly useful, their components, i.e., the numerator and the denominator both need to be arbitrary-precision integers. As we have mentioned before, arbitrary precision anything does not go very well with fixed-size circuits inside the CPU of a computer.

Programming languages, on the other hand, sometimes use arbitrary-precision rational numbers. This is the case, in particular, with the language *Lisp*.

7.4 Summary

The ALU operates only on data using the registers which are internal to the CPU. All computers deal with numbers. We have studied the instructions that perform basic arithmetic operations on data operands in the previous unit. To understand the task carried out by the ALU, we have introduced the representation of numbers in a computer and how they are manipulated in addition and subtraction operations.

Self Assessment Questions

1. In binary arithmetic, we simply reserve ___________ bit to determine the sign.
2. The infinite precision ten's complement of -2 and +5 is ___________ and ___________.
3. The finite precision three digit ten's complement of -2 and +5 is ___________ and ___________.
4. Using finite precision ten's complement, if both numbers for addition are positive and results negative, then the circuit reports ___________.
5. ___________ is not very common in the hardware of a computer.

7.5 Terminal Questions

1. Explain the ALU operations.
2. Discuss various number representations in a computer system.

7.6 Answers

Self Assessment Questions:

1. one
2. 98, 05
3. 998, 005
4. overflow
5. rational numbers and rational arithmetic

Terminal Questions:

1. Refer Section 7.2
2. Refer Section 7.3