On the Science Behind Computing (Part I)

Samir Khuller *

1 Circuits

We shall now examine the question of how a modern electronic computer actually computes functions. Let us consider a simple problem that involves adding two numbers. In ruby, this is easily accomplished by simple saying "x=a+b" and this will store in a memory location named "x" the sum of the values in memory locations "a" and "b".

However, a computer needs to perform this operation "somehow". Ruby is a "high level language" and lets us write programs in a way that makes it easy to talk about memory locations by name and also specify specific operations on these values by simply using the operator "+". This is not how computer hardware works however. Ruby is eventually translated to some lower level machine code that the computer can easily interpret and execute.

However, this still begs the question - how does a computer actually add numbers? This is also complex - but the objective is to examine some basic principles and to show you a glimpse of what is going on inside the machine.

We talked about binary representation earlier as well – a number such as 7283 is really $7 \times 10^3 + 2 \times 10^2 + 8 \times 10^1 + 3 \times 10^0$. However, we need to use 10 types of digits when we work with base 10 numbers – 0,1,2,3,4,5,6,7,8,9. When working with binary arithmetic we have only 0 and 1 (these are our binary digits, or bits) and all information and values are represented using bits. The primary reason is that electrical voltages are used to represent 1 and 0, and so each wire has two possible states. A 2V line may represent 1 and a very low voltage may be used to represent 0.

Suppose we have the numbers 5 and 4, these are represented as 0101 and 0100 (why is this the correct binary representation for these two numbers?). Adding them gives 9, or 1001.

```
1
0101
0100
----
1001
```

How shall a computer actually perform this operation? This operation involves lining up the two bit sequences and then performing the addition operation, generating a "carry bit" every time we need to. The carry bit is then fed to the pair of bits to the left as we add from right to left.

To understand how to implement addition of two bit sequences we will first have to take a short detour into some very basic *Boolean Algebra*. This is a way to define operators that work

*Department of Computer Science, University of Maryland, College Park, MD 20742. E-mail : samir@cs.umd.edu.

with boolean variables - these are variables that are used to generally denote *truth* values of statements and can be either `TRUE` or `FALSE`. We represent `TRUE` as the bit 1 and `FALSE` as the bit 0.

Transistors were used to create logical "gates" – these are basic circuit elements that are used to define boolean functions and operators. We will use these logical gates as basic circuit elements to create a circuit that will perform the operation of adding.

The first operator we discuss is the `AND` gate. Its takes two inputs, X and Y, and outputs 1 if and only if X=1 and Y=1. We can write this is $X \wedge Y = X.Y$.

The second operator we discuss is the `OR` gate. This also takes two inputs X and Y, and outputs 0 if and only if X=0 and Y=0. We write this as $X \vee Y = X + Y$.

The last operator is the logical `NOT` gate that is used to flip a bit. Thus $\overline{X}$ is 1 if and only if X is 0.

The table that shows the precise behaviour of both the AND and OR functions are shown below.

X	Y	X.Y	X+Y
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

We first consider a 2 bit adder (called a HALF ADDER). Suppose we wish to add two 1 bit numbers X and Y. The output is two bits – an S bit (sum) and a C bit (carry).

The logical functionality of S and C are obvious.

X	Y	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

The carry bit is very easy to implement since $C = X.Y$ since we generate a carry only when both X and Y are 1. We now need to figure out a boolean expression for the S (sum) bit – note that the S bit is 1 under precisely two conditions – when X is 0 and when Y is 1, and when X is 1 and Y is 0. Thus we can set $S = \overline{X}.Y + X.\overline{Y}$. (You should verify this.) We could try to set $S = X + Y$ but this is not correct, since S is supposed to be 0 when X and Y are both 1 (but the way the + operator is defined, 1+1 yields 1).

The logic to build a HALF adder is thus very easy. We can do this with some wires, two NOT gates, three AND gates and an OR gate (see Fig. 1).

However, to actually build something that will add say two 4 bit numbers, we need something slightly more sophisticated due to the carry bit. Suppose we wish to add $X_1X_2X_3X_4$ and $Y_1Y_2Y_3Y_4$, then for example while adding X_3 and Y_3 we also need to add a carry bit in case both X_4 and Y_4 are 1 since in this case they will generate a carry. This is accomplished by building something called a FULL ADDER. This takes as input three lines – X, Y and C_i (carry-in) and generates both an S bit and a C_o bit (carry-out), that in turn can be fed into the carry in line for the pair of bits to the left etc.

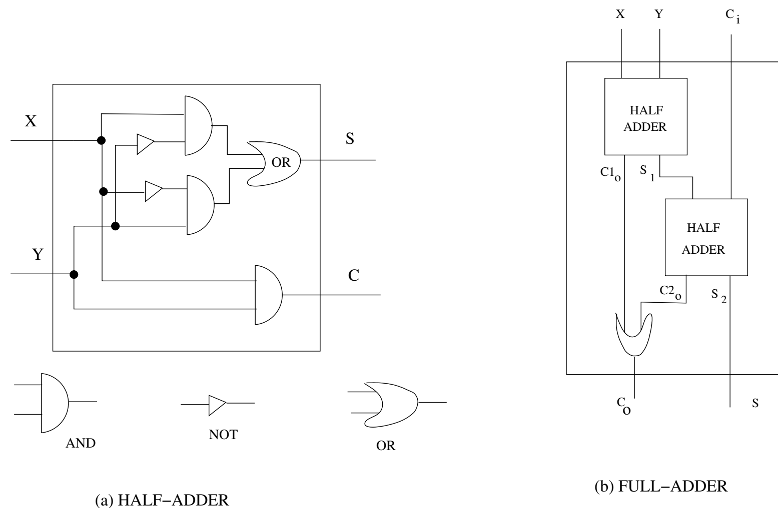

Figure 1: Figure showing ADDER circuits.

X	Y	C_i	S	C_o
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

We can construct this in logic by defining S and C_o as follows.

$$C_o = X.Y + X.C_i + Y.C_i$$

The logic behind this is that the carry bit is 1 if and only if at least two of the input bits are 1. We can also derive this and simplify as follows. Note: we are taking the conditions when the C_o bit is 1, and setting an OR of all these conditions.

$$C_o = X.Y.C_i + X.Y.\overline{C_i} + X.\overline{Y}.C_i + \overline{X}.Y.C_i$$

$$C_o = X.Y.(C_i + \overline{C_i}) + X.\overline{Y}.C_i + \overline{X}.Y.C_i$$

$$C_o = X.Y + X.C_i + Y.C_i$$

$$S = \overline{X}.\overline{Y}.C_i + \overline{X}.Y.\overline{C_i} + X.\overline{Y}.\overline{C_i} + X.Y.C_i.$$

However, we can also use two HALF-ADDERS to build a full adder as follows. The idea is very simple – we first add the bits X and Y, we feed the carry-out line of this into a second HALF-ADDER that adds the original carry bit. With a small amount of effort you can verify that this circuit has the desired behavior by trying each 3 bit sequence for X,Y,C.

Finally, a four FULL-ADDERs can be lined up together to create a circuit that will add two four bit numbers (you basically have to feed the carry out line to the next FULL-ADDERs carry in line.