2 Boolean Arithmetic [p29]

2.1 Background [p30]

2.1.1 Binary numbers and addition [p30]

When we press the keyboard keys labelled **1**, **9** and **Enter**, the equivalent 32-bit binary code (if we are working on a 32-bit machine) **00000000000000000000000000010011** ends up in the register of the computer's memory.

LSB (Least Significant Bits) the right-most digits of a binary number.

MSB (Most Significant Bits) the left-most digits of a binary number.

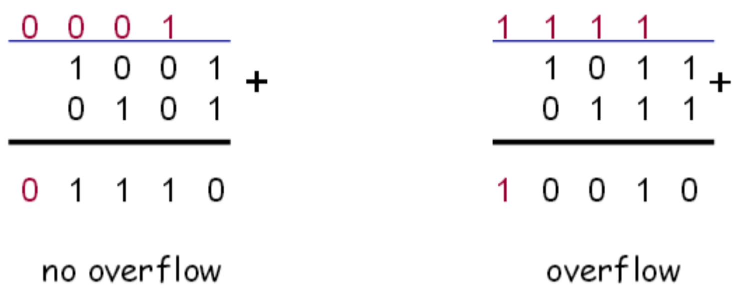


Figure 2.1: *Add digit by digit from right to left.* Observe, computer hardware for binary addition of 2 n-bit numbers can be built from logic gates designed to calculate the **sum of 3 bits** (pair of bits plus carry bit) - hence the **full adder**.

2.1.2 Signed Binary Numbers [p31]

A binary system with n digits can generate a set of 2^n different bit patterns; hence if we need to represent positive and negative numbers, we spilt these arrangements into 2 equal subsets (one for the positive numbers, the other for the negative numbers).

Definition 2.1. The *2's (or radix) complement method* of a number x is

$$\overline{x} = \begin{cases} 2^n - x & \text{if } x \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

This is equivalent to *inverting each digit and adding 1.*

0	0000		
1	0001	1111	-1
2	0010	1110	-2
3	0011	1101	-3
4	0100	1100	-4
5	0101	1011	-5
6	0110	1010	-6
7	0111	1001	-7
		1000	-8

$$2 - 5 = 2 + (-5) = \begin{array}{r} 0\,0\,1\,0 \\ +\,1\,0\,1\,1 \\ \hline 1\,1\,0\,1 \end{array} = -3$$

Figure 2.2: 2's complement representation of signed number in a 4-bit binary system. Observe the total number of numbers represent is 2^n, with 2^{n-1} is each subset. Also, the addition of a number an its inverse is 0000 (e.g. $1 + (-1) = 0001 + 1111 = (1)\,0000$, where the leading 1 is omitted because we're working in a 4-bit binary system. This representation of negative numbers makes subtraction very easy - as shown right. We conclude that all basic arithmetic and logical operators can be perform by a single chip (**ALU**).

Example (E2.1)**.**

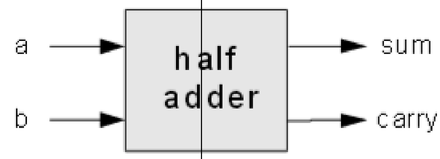
ALU (Arithmetic Logical Unit) the centrepiece chip or the CPU (which is the centrepiece of a computer) that executes all arithmetic and logical operations.

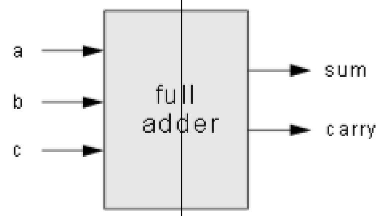
Remark. All positive number begin with **0**, and all negative numbers begin with **1**.

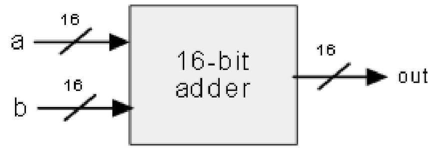
2.2 Specification [p32]

One such ALU is the *adder* chip.

2.2.1 Adders

<table>
<tr><th>Adder</th><th>Implementation</th></tr>
<tr><td>Half-adder: designed to add 2 bits. Based on the XOR and AND gates</td><td>
<table>
<tr><th>a</th><th>b</th><th>sum</th><th>carry</th></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td></tr>
</table>


</td></tr>
<tr><td>Full-adder: designed to add 3 bits. Can be based on half-adder gates.</td><td>
<table>
<tr><th>a</th><th>b</th><th>c</th><th>sum</th><th>carry</th></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr>
</table>


</td></tr>
<tr><td>Multi-bit Adder: designed to add two n-bit numbers ($n \in \{16, 32, 64, ...\}$. Array of full-adder gates.</td><td>


<table>
<tr><td>...</td><td>1</td><td>0</td><td>1</td><td>1</td><td>a</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td>+</td></tr>
<tr><td>...</td><td>0</td><td>0</td><td>1</td><td>0</td><td>b</td></tr>
<tr><td>...</td><td>1</td><td>1</td><td>0</td><td>1</td><td>out</td></tr>
</table>
16-bit adder
</td></tr>
</table>

Table 2.1: Hierarchy of three adders.

Incrementer: it is convenient to have a chip dedicated to adding the constant 1 to a given number (e.g. for calculating negative numbers).

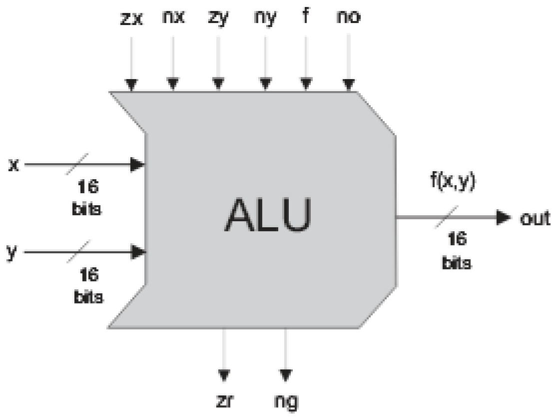
2.2.2 The Arithmetic Logic Unit (ALU) [p35]

This subsubsection describes an ALU that will become the centerpiece of our computer platform *Hack*. The Hack ALU computes a fixed set of function $out = f_i(x, y)$ where x, y are two 16-bit inputs, out a 16-bit ouptut and f_i is an arithmetic or logical function selected from a fixed set of eighteen possible functions:

These bits instruct how to preset the x input		These bits instruct how to preset the y input		This bit selects between + / And	This bit inst. how to postset out	Resulting ALU output
zx	nx	zy	ny	f	no	out=
if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out=x&y	if no then out=!out	f(x,y)=
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	!x
1	1	0	0	0	1	!y
0	0	1	1	1	1	-x
1	1	0	0	1	1	-y
0	1	1	1	1	1	x+1
1	1	0	1	1	1	y+1
0	0	1	1	1	0	x-1
1	1	0	0	1	0	y-1
0	0	0	0	1	0	x+y
0	1	0	0	1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x|y

Figure 2.3: The ALU truth table working with **16-bit** inputs/output (so if $zy = 1$, y would zeroed $y = (000...00)_2$ and in general $0 = (000...00)_2$, $1 = (111...11)_2$; note !=Not, &=And and |=Or (performed **bit-wise**). We designed the ALU by defining which functions were desired and worked backwards to figure out how x, y and *out* can be manipulated by binary operations to achieve these results. We have included the 6 control bits, each using a straightforward binary operation.

Remark. We instruct the ALU which function to compute by setting six input bits, called *control bits*; hence we have $2^6 = 64$ different functions (the function can either be included or not). 18 are of interest to us.




```
Chip name: ALU
Inputs:    x[16], y[16],     // Two 16-bit data inputs
           zx,               // Zero the x input
           nx,               // Negate the x input
           zy,               // Zero the y input
           ny,               // Negate the y input
           f,                // Function code: 1 for Add, 0 for And
           no                // Negate the out output
Outputs:   out[16],          // 16-bit output
           zr,               // True iff out=0
           ng                // True iff out<0
Function:  if zx then x = 0        // 16-bit zero constant
           if nx then x = !x       // Bit-wise negation
           if zy then y = 0        // 16-bit zero constant
           if ny then y = !y       // Bit-wise negation
           if f then out = x + y   // Integer 2's complement addition
                else out = x & y   // Bit-wise And
           if no then out = !out   // Bit-wise negation
           if out=0 then zr = 1 else zr = 0   // 16-bit eq. comparison
           if out<0 then ng = 1 else ng = 0   // 16-bit neg. comparison
Comment:   Overflow is neither detected nor handled.
```

Figure 2.4: ALU specification (not particular efficient; we have chose to specify an ALU hardware with limited functionality and implement as many operations as possible in software - operating system).

Note. The overall functionally of the hardware/software platform is delivered jointly by the **ALU** and the **operating system**.