Introduction to Binary Logic and Boolean Algebra

Antoine Lavault[1,2]

[1]Apeira Technologies

[2]UMR CNRS 9912 STMS, IRCAM, Sorbonne Université

19 janvier 2024

Contents

Contents

What is going to happen ?

What you should know, use, and understand by the end of the class :

- Boolean logic and friends
- Logic gates, synchronous and asynchronous logic.
- Basics of computer architecture
- Programmable Logical Devices : PAL, CPLD, FPGA
- Circuit description in VHDL

Analog vs. Digital

Digital electronics is defined as opposed to analog electronics.

- Analog : the signal is continuous
- Digital : the signal is discrete, either in time (sampling) or in value (discretization)

And that's about it.
First, let's talk about binary, octal, and hexadecimal.

Radices and bases

Let b be a positive integer such that $b \geq 1$. Then, every positive integer a can be expressed uniquely in the form :

$$a = r_m b^m + r_{m-1} b^{m-1} + \cdots + r_1 b + r_0, \tag{1}$$

where m is a non-negative integer and the r's are integers such that

$$0 < r_m < b; \tag{2}$$

and

$$0 \leq r_i < b; \text{for } i = 0, 1, \cdots, m-1 \tag{3}$$

We say the number a is expressed in base b and is noted a_b.

Binary and Hexadecimal

Binary = base 2, Hexadecimal = base 16.
For bases greater than 10, the numbers are replaced by letters : a for 10, b for 11, etc...

Exemple

$3 = 1 * 2^1 + 1 * 2^0 = 3 * 16^0 = 10_2 = 3_{16}$
$24 = 1 * 2^4 + 1 * 2^3 = 1 * 16 + 8 * 16^0 = 11000_2 = 18_{16}$
$1465 = 5 * 16^2 + 11 * 16^1 + 9 * 16^0 = 5b9_{16}$

Note : hexadecimal numbers are also noted as $0x\cdots$ (e.g $0x5b9$) and binary numbers $0b\cdots$

Exercise

$\underbrace{1010}_{a}\underbrace{0111}_{7} = 0xa7$ is a byte i.e., 8 binary digits (or bits)
What is the hexadecimal value of 0b0100 0101 ?

Contents

Binary variable

A binary variable is a variable with only two possible states.

Définition

A binary variable a is one that takes its values in $\mathcal{B} = \{0, 1\}$

A binary variable can be associated with logical values : True and False. Most of the time, True =1 and False = 0.

Définition

On $\mathcal{B}$, we can define two binary operations called conjunction, noted $\wedge$, and disjunction, noted $\vee$, and an unary operation called negation, noted $\neg$.
For x and y in $\mathcal{B}$, we have :

- $x \wedge y = 1$ if $x = y = 1, x \wedge y = 0$ otherwise
- $x \vee y = 0$ if $x = y = 0, x \vee y = 1$ otherwise
- $\neg x = 1$ if $x = 0, x = 0$ otherwise

Analogy

These operations are similar to the intersection, the union, and the complement in set theory.

Common names

Conjonction = AND, Disjonction = OR, Negation = NOT

For x, y in $\mathcal{B}$,

- $x \wedge y = x \cdot y$
- $x \vee y = x + y$
- $\neg x = \bar{x}$
- Exclusive OR : x XOR $y = x \oplus y$
- Exclusive OR : x XOR $y = x \oplus y$

WARNING : $+$ and $.$ are not operations modulo 2 here

And the related truth table,

x	y	$x \cdot y$	$x + y$	$x \oplus y$
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	1	0

Table – Truth table for basic binary operations on binary variables

Boolean algebra satisfies many of the same laws as ordinary algebra when one matches up with addition and multiplication. In particular, the following laws are common to both kinds of algebra :

- Associativity of $\vee$:$x \vee (y \vee z) = (x \vee y) \vee z$
- Associativity of $\wedge$: $x \wedge (y \wedge z) = (x \wedge y) \wedge z$
- Commutativity of $\vee$: $x \vee y = y \vee x$
- Commutativity of $\wedge$: $x \wedge y = y \wedge x$
- Distributivity of $\wedge$ over $\vee$: $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$
- Identity for $\vee$: $x \vee 0 = x$
- Identity for $\wedge$: $x \wedge 1 = x$
- Annihilator for $\wedge$: $x \wedge 0 = 0$
- Double Negation : $\neg(\neg x) = x$
- Complementation : $x \vee \neg x = 1$ and $x \wedge \neg x = 0$

The following does not hold in a Boolean algebra :

$$\begin{aligned}(-x)(-y) &= xy \\ (-x)+(-y) &= -(x+y)\end{aligned} \tag{4}$$

Instead, we have the De Morgan laws :

$$\begin{aligned}&\text{De Morgan 1} \quad \neg x \wedge \neg y = \neg(x \vee y) \\ &\text{De Morgan 2} \quad \neg x \vee \neg y = \neg(x \wedge y)\end{aligned} \tag{5}$$

We also have the Consensus Theorem : for $x, y, z \in \mathcal{B}$

$$xy \vee \bar{x}z \vee yz = xy \vee \bar{x}z \tag{6}$$

As an exercise, let's prove the following :

- De Morgan laws :

$$\text{De Morgan 1} \quad \neg x \wedge \neg y = \neg(x \vee y)$$
$$\text{De Morgan 2} \quad \neg x \vee \neg y = \neg(x \wedge y)$$

- Consensus Theorem : for $x, y, z \in \mathcal{B}$

$$xy \vee \bar{x}z \vee yz = xy \vee \bar{x}z$$

Contents

Outline

A problem

Problem

A tank is filled by 2 valves : V_1 and V_2. We consider 3 levels :

- Warning (W)
- Bottom (B)
- top (T)

When the level is below W, V_1 et V_2 are opened.
When the level is between W and B, only V_1 is opened.
When the level is between B and T, only V_2 is opened.
When the level is greater than T, we close the pipe.

How to deal with it ?

- Make a drawing (visualization)
- Make the truth table

Reframing the problem

Problem

A tank is filled by 2 valves : V_1 and V_2. We consider 3 levels :

- Warning (W)
- Bottom (B)
- top (T)

When the level is below W, V_1 et V_2 are opened.
When the level is between W and B, only V_1 is opened.
When the level is between B and T, only V_2 is opened.
When the level is greater than T, we close the pipe.

Analysis

We can model the state of the valves and the state of the sensors by binary variables. Here, the sensors control the valves : the sensors are the input of a system whose outputs are the valve states.
Note : not all states are physically attainable.

Towards a truth table

w	b	t	v_1	v_2
0	0	0	1	1
0	0	1	X	X
0	1	0	X	X
0	1	1	X	X
1	0	0	1	0
1	0	1	X	X
1	1	0	0	1
1	1	1	0	0

Table – Truth table for the introductory valve problem

The X symbol ("do not care") is used when the output is not physically meaningful.
Naively, we have :

$$\begin{aligned} v_1 &= \overline{wbt} + w\overline{bt} \\ v_2 &= \overline{wbt} + wb\bar{t} \end{aligned} \tag{7}$$

Outline

Minterms

Définition

For a Boolean algebra B, a minterm is a function from B^n to$\mathrm{B}$ such that it exists an unique $(a_1, \cdots, a_n) \in \mathrm{B}^n$ with :

$$minterm(a_1, \cdots, a_n) = 1 \tag{8}$$

Hands-on interpretation

A minterm is a Boolean expression resulting in 1 for the output of a single input and 0s otherwise.

Maxterm

Définition

For a Boolean algebra B, a maxterm is a function from B^n to B such that it exists an unique $(a_1, \cdots, a_n) \in \mathrm{B}^n$ with :

$$maxterm(a_1, \cdots, a_n) = 0 \tag{9}$$

Hands-on interpretation

A maxterm is a Boolean expression resulting in 0 for the output of a single input and 1s otherwise.

Example of minterm and maxterm

e_1	e_2	e_3	Minterm	Maxterm
0	0	0	$\bar{e}_1 \bar{e}_2 \bar{e}_3$	$e_1 + e_2 + e_3$
0	0	1	$\bar{e}_1 \bar{e}_2 e_3$	$e_1 + e_2 + \bar{e}_3$
0	1	0	$e_1 \bar{e}_2 \bar{e}_3$	$e_1 + \bar{e}_2 + e_3$
0	1	1	$\bar{e}_1 e_2 e_3$	$e_1 + \bar{e}_2 + \bar{e}_3$
1	0	0	$e_1 \bar{e}_2 \bar{e}_3$	$\bar{e}_1 + e_2 + e_3$
1	0	1	$e_1 \bar{e}_2 e_3$	$\bar{e}_1 + e_2 + \bar{e}_3$
1	1	0	$e_1 e_2 \bar{e}_3$	$\bar{e}_1 + \bar{e}_2 + e_3$
1	1	1	$e_1 e_2 e_3$	$\bar{e}_1 + \bar{e}_2 + \bar{e}_3$

Table – Minterms and maxterms in a 3 value case

Back to the introductory example

We derived previously this pair of equations :

$$\begin{aligned} v_1 &= \overline{wbt} + w\overline{bt} \\ v_2 &= \overline{wbt} + wb\bar{t} \end{aligned} \tag{10}$$

This set of equations for the valve control example is the minterm expression of the system, where the "do not care" condition is set to 0.
However, we don't know if this is optimal.

Outline

LSB and MSB

Définition

In computing, the LSB (least significant bit) is the bit corresponding to the lowest power of 2 in the binary decomposition of the integer.
Similarly, the most significant bit (MSB) corresponds to the highest power of 2 in the binary decomposition of the integer (bound to a 1).

Exemple

For 1001_b, the MSB is 1, and the LSB is 1. For 1000_b, the MSB is 1, and the LSB is 0. For 0110_b, both the MSB and the LSB are 0.

Gray Code

Définition

The reflected binary code (RBC), also known as reflected binary (RB) or Gray code (after Frank Gray), is an ordering of the binary numeral system such that two successive values differ in only one bit (binary digit).

Property

The Hamming distance between two consecutive words is 1.

Gray Code

Decimal	Binary	Gray	Decimal of Gray
0	0000	0000	0
1	0001	0001	1
2	0010	0011	3
3	0011	0010	2
4	0100	0110	6
5	0101	0111	7
6	0110	0101	5
7	0111	0100	4
8	1000	1100	12
9	1001	1101	13
10	1010	1111	15
11	1011	1110	14
12	1100	1010	10
13	1101	1011	11
14	1110	1001	9
15	1111	1000	8

Table – Gray Code, 4-bit case

Karnaugh Map

Définition

A Karnaugh map is a diagram consisting of a rectangular array of squares, each representing a different combination of the variables of a Boolean function.

Property

A Karnaugh map uses the Gray code to label its rows and columns : two adjacent squares differ by the complementation of one and only one variable.

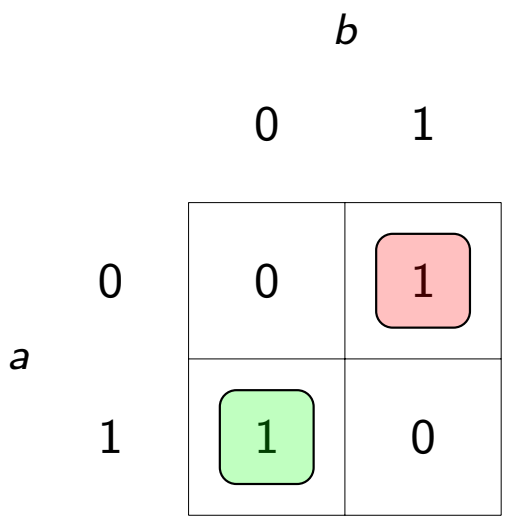
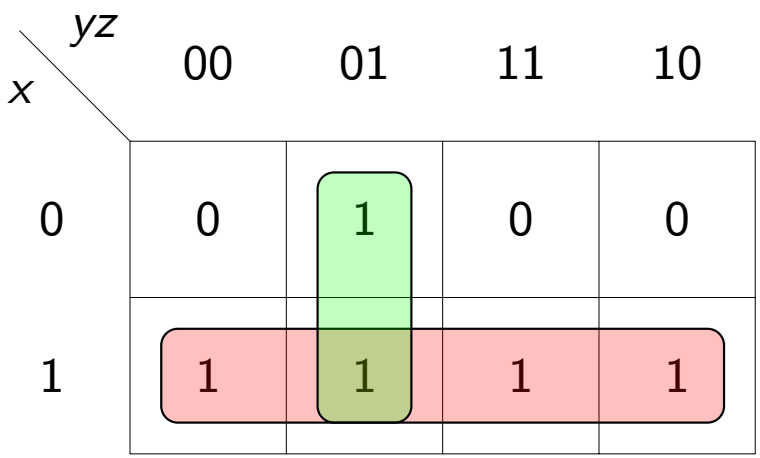


Figure – Karnaugh map for $f = a \oplus b$

The following map is for the function $f = \bar{x}\bar{y}z + x\bar{y}\bar{z} + x\bar{y}z + xy\bar{z} + xyz$



By finding groups of 2 adjacent 1s, we can simplify. For instance, the green square represents an expression independent of x, $\bar{y}z$. Similarly, the red rectangle is bound to an expression independent of y and z, so x.
All in all, we have $f = x + \bar{y}z$

Contents

Outline

Bestiary of Logic Gates

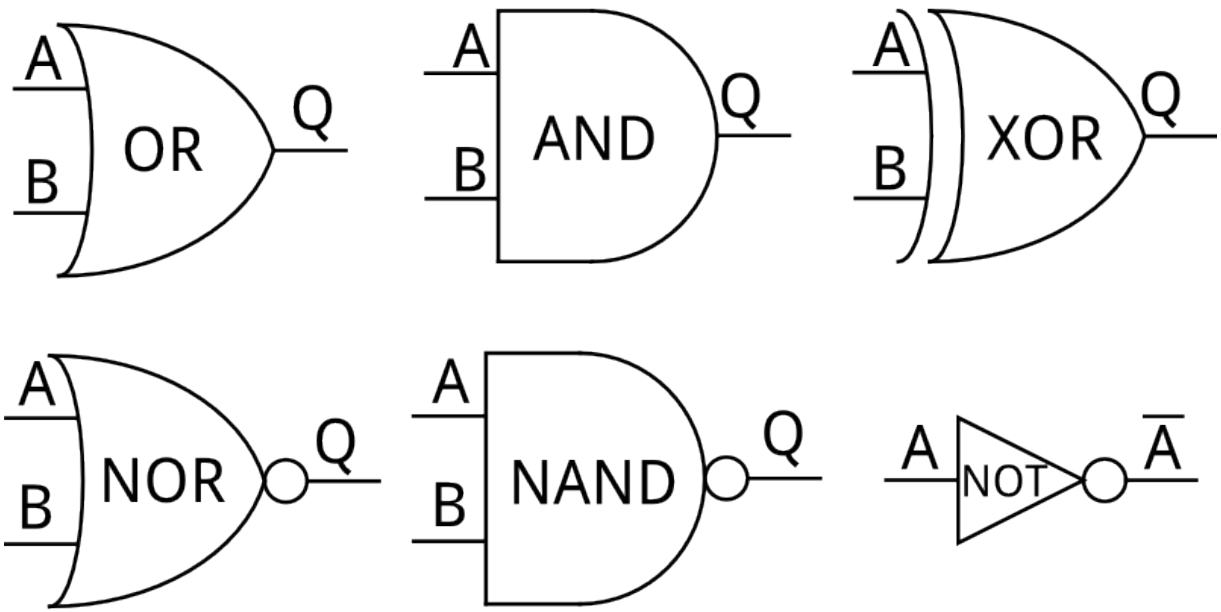


Figure – Logic Gates, ANSI style

NAND and NOR logic

Property

Every logic circuit can be written with only NAND (Not And) gates. Every logic circuit can be written with only NOR (Not Or) gates.

Note : NOT can be seen as a NAND or a NOR gate whose inputs are the same.

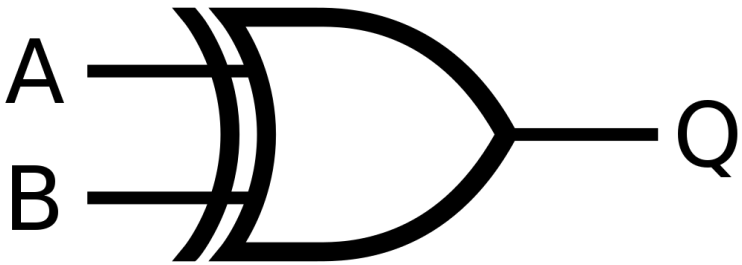


Figure – XOR gate

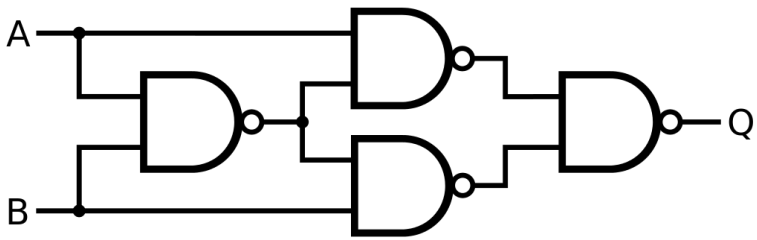


Figure – XOR gate, NAND representation

Iterative circuits

For circuits with many variables, the synthesis using the previously mentioned method does not provide a Boolean expression.
In this case, the complex problem should be broken down into smaller (most likely identical) simple problems.

Let us consider a circuit with two numbers x and y, coded on n bits as inputs and whose outputs are $S_x = X > Y$ and $S_y = Y > X$.
We can break down the problem so that each input bit can be compared.
If we denote, a_i and b_i the cascaded inputs, α_i and β_i the cascaded outputs and x_i,y_i the i-th bit of x and y respectively, we have :

$$\begin{aligned} \alpha_i &= a_i + b_i x_i y_i \\ \beta_i &= b_i + a_i y_i x_i \\ a_{n-1} &= b_{n-1} = 0 \end{aligned} \tag{11}$$

Exercice

Find the NAND representation of such a block.

Outline

Decoders

Définition

A decoder is a combinational circuit with n input lines and a maximum of 2^n output lines. One of these outputs will be active High based on the combination of inputs present when the decoder is enabled. That means the decoder detects a particular code. When enabled, the decoder outputs are nothing but the min terms of n input variables lines.

Exemple (74LS146)

1 :4 (1 to 4) decoder means 2 bits as input and 4 output lines. 00_b is mapped to output 0, 01_b is mapped to output 1 and so on and so forth.

Encoders/Multiplexers

Définition

An Encoder is a combinational circuit that performs the reverse operation of the Decoder. It has a maximum of 2^n input and n output lines. It will produce a binary code equivalent to the input, which is active High. Therefore, the encoder encodes 2^n input lines with n bits.

Définition

A Multiplexer is a combinational circuit with a maximum of 2^n data inputs, n selection lines, and a single output line. One of these data inputs will be connected to the output based on the values of the selection lines.
Since there are $s$election lines, there will be 2^n possible combinations of zeros and ones. So, each combination will select only one data input. Multiplexer is also called as "Mux".

Tri-state logic

Définition

Tri-state logic allows an output or input pin/pad to assume a high impedance state (noted Z), effectively removing the output from the circuit and the 0 and 1 logic levels.

Three-state buffers can also implement efficient multiplexers, especially those with large inputs. To avoid floating pins, pull-up or pull-down resistors might be added.
Three-state buffers are essential to the operation of a shared electronic bus.

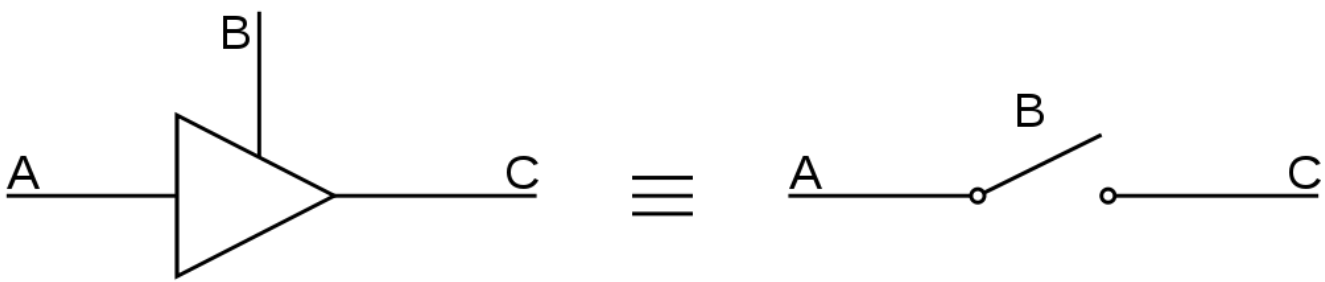


Figure – Tri-state buffer, effectively equivalent to a controlled switch

Outline

Representation of numbers

With n Boolean variables, we have 2^n possibilities. The most useful representations are :

- Unsigned integers
- Signed integers
- Two's complement

Unsigned integers

The 2^n combinations represent integers from 0 to $2^n - 1$.
An operation is not guaranteed to be within the range of values. This case is called overflow.
Let us consider the following example to see how the overflow is handled. On 4 bits, the maximum encoded unsigned integer is 15.
If we compute $5 + 13$, the results on 4 bits should be $0010_b = 2$. The overflow will be signaled by a supplementary carry bit.

Signed Integers - Sign–magnitude

The simplest representation uses the most significant bit as a sign bit, 0 for positive and 1 for negative numbers. The $n-1$ other bits represent the absolute value of the number as an unsigned integer.
Note that there are two zeros in this case!

Two's Complement

A problem with "naive" signed integers is we have to treat the sign bit *before* interpreting the rest of the number.
With two's complement, a number $0 \leq m < 2^{n-1}$ coded on n bits will be represented just like before. For numbers $2^{n-1} \leq m < 0$, m will be represented as $2^n - |m|$.

Exemple

On four bits signed, $5 = 0101$ and $-5 \rightarrow 1011$ $(2^4 - 5) = 11 > 2^3 = 8$

To add two number a, b coded in two's complement and noted α, β, we have four cases :

- $a, b > 0 : \alpha = a, \beta = b, \alpha + \beta = a + b$
- $a, b < 0$ and $|b| \leq a : \alpha = a, \beta = 2^n - |b|, \alpha + \beta = 2^n + a - |b| \equiv a - |b|$
- $a, b < 0$ and $|b| > a : \alpha = a, \beta = 2^n - |b|, \alpha + \beta = 2^n - (|b| - a) \equiv a - |b|$
- $a, b < 0$:
  $\alpha = 2^n - |a|, \beta = 2^n - |b|, \alpha + \beta = 2^n + 2^n - (|a| + |b|) \equiv 2^n - (|a| + |b|)$
  which is the representation of $-(|a| + |b|)$

Remark : for a number a, its negative in two's complement is the inversion of all the bits $+1$.
Proof : If $a \geq 0$, $\bar{a} = 2^n - |a| - 1, \bar{a} + 1 = 2^n - |a| \equiv -a$, if $a < 0$, similar.

Exemple

$-5 = 1011, \overline{-5} = 0100, \overline{-5} + 1 = 0101 \equiv 5$
$5 = 0101, \overline{5} = 1010, \overline{5} + 1 = 1011 \equiv -5$

Full Adder

- Ports : Inputs A,B ; Output S (Sum), $C_{in/out}$ (Carry)
- $S = A \oplus B \oplus C_i$, $C_{out} = (AB) + (C_{in}(A \oplus B))$

Exercise

Proof of the boolean equations above.

A	B	Cin	Cout	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Table – Truth table of full adder

Arithmetic logic unit

Example : 74181 IC = `https://www.ti.com/lit/ds/symlink/sn54s181.pdf`

- Ports : Inputs A, B ; Output Y
- Opcode : which operations to do = sum, difference, logic etc...
- Status : carry bit, parity, overflow...

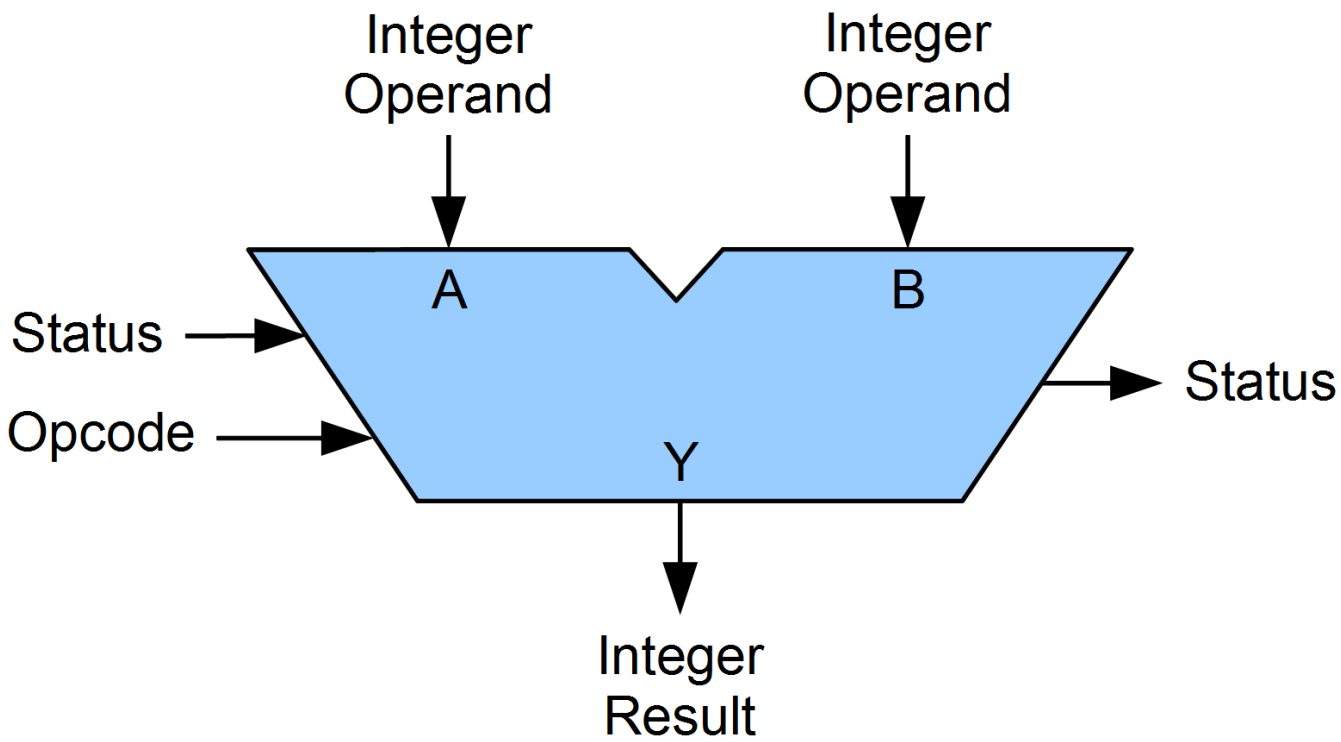


Figure – Arithmetic logic unit

Outline

My 2 cents.

For the moment, we worked with unsigned and signed integers.
What about things with a point ? Like 3.1415. Or 0.02.

Fixed-point arithmetic

Fixed-point arithmetic is the answer to model fractional (non-integers) numbers by allocating a fixed part of their representation to the fractional part.

Exemple

$0b0.1 = 0 * 2^0 + 1 * 2^{-1}$
$30.75 = 3 * 10 + 0 * 1 + 7 * 0.1 + 5 * 0.01$

Fixed-point arithmetic

Fixed-point arithmetic is very interesting because the fractional numbers are equivalent to the operation on the underlying integers and a division. To do so :

- Align the decimal points
- Number of bits to represent the numbers : maximum number of bits of the integer and fractional parts

Exemple

$3.25 + 13.125 = (3250 + 13125)/1000 = 16.375$ (to integer and back)
$3.25 + 13.125 = 03.250 + 13.125 = 16.375$ (direct operation)
$0b11,01 + 0b1101,001 = 00011,010 + 01101,001 = 10000,011$ (direct operation)

Signed fixed-point arithemetic

Does not change much. Just be wary of where the sign bit is.

Floating-point numbers - IEEE 754

The IEEE 754 describes a format as a "set of representations of numerical values and symbols." This standard also defines precisely what result will be produced by each fundamental floating-point operation over all possible input values. It describes what a compliant implementation should do with respect to rounding of results that cannot be expressed precisely. A simple example of such a calculation would be 1/3, requiring an infinite number of digits to express precisely in decimal or binary notation.
Floating-point formats use the available space to store three pieces of information about a floating-point number :

- A sign bit (S) that shows whether the number is positive (0) or negative (1).
- An exponent giving its order of magnitude.
- A mantissa giving the fractional binary digits of the number.

For a single precision float (i.e., 32 bits long), bit 31 of the word is the sign bit S, bits 30 to 23 give the exponent e, and bits 22 down to 0 give the mantissa. The value of the number is then $\pm\overline{1.m} \times 2^e$.

Representation of IEEE 754 floats

For a single precision float, as described before, the value encoded is :

$$(-1)^{b_{31}} \times 2^{(b_{30}b_{29}\ldots b_{23})_2 - 127} \times (1.b_{22}b_{21}\ldots b_0)_2$$

Note that the bias is 127 and not 128, like two's complement, for the exponent.

Special values of floats

IEEE 754 standard ensures some special values for exceptional cases.

- Zero is signed : mantissa and exponent equal to 0_b, any sign bit.
- Infinities : mantissa equal to 0_b and exponent $\bar{1}_b$.
- NaN (Not a Number) : used to describe the results of illegal or ambiguous operations ($0/0$, $\infty \times 0$, $\sqrt{-1}$)
- Denormalized numbers : exponent is 0_b, mantissa $\neq 0$.

Operations on floats

See the tutorial. Probably.

Why not using floats ?

- An example of FPU in VHDL :
  `https://opencores.org/websvn/filedetails?repname=openfpu64&path=%2Fopenfpu64%2Ftrunk%2Ffpu_mul.vhd`
- Integer vs. Floating-Point Processing on Modern FPGA Technology, Hettiarachchi et al.

  Historically, FPGA designers have used integer processing whenever possible because floating-point processing was prohibitively costly due to higher logic requirements and speed reduction. (...) Recently, Intel introduced the Arria 10 FPGA, the industry's first FPGA that includes single-precision hardened Floating-Point Units (FPUs)(...) All programs are tested with Intel Stratix V FPGA, which does not have hardened FPUs, and Intel Arria 10 FPGA for comparison. The performance metric indicates that, on average, there is a 20.18% performance increase when Stratix V processes fixed-point operations and a 27.17% performance increase when Arria 10 processes fixed-point operations. The results indicate that the FPGAs perform better when processing converted fixed-point arithmetic operations than floating-point arithmetic, regardless of whether they include hardened FPUs.

Why not use floats

- FPUs are not always available on certain CPUs, i.e., loss of performance using floats :
  `https://developer.arm.com/documentation/102787/latest`
- Non-standard behavior, taken from ARM documentation :

  For performance reasons, such denormal values are often ignored and are flushed to zero. This is strictly a violation of IEEE-754, but denormal values are used rarely enough in real programs that the performance benefit is worth more than the correct handling of these extremely small numbers. Cortex processors with VFP enable code to select between flush-to-zero mode and full denormal support.

Contents

Conclusion

What we have seen :

- Boolean algebra
- Operations on Boolean algebras
- Karnaugh map for logic equation simplification
- Logic gates
- Common logic circuits
- How numbers can be represented in digital circuits.