Number Systems and Logic

Aslam Safla
<aslam@cs.uct.ac.za>

Signpost

We need to use binary representations for kind of data.

- Computers operate on binary values (as a consequence of being built from transistors).
- Bit = “Binary digit”
  - Building a computer using base 10 would be much much much more difficult than using binary.

In a computer chip, transistors switch between two binary states -- 0 and 1.

One computer chip can have millions of transistors continually switching, to complete complex calculations.

Signpost

Bits are bundled together into 8-bit collections, called **bytes**.

With these bytes, there are 3 types of data we want to represent:

- integers (both positive and negative)
- floating point values
- characters

Number Representations

Numeric information is fundamental to computers – to encode data and instructions.

Notation: N_r for number N using radix r

- radix refers to the base – the number of possible symbols for each digit.

General radix *r* number representation:

$$d_p d_{p-1} d_{p-2} \ldots d_2 d_1 d_0 . d_{-1} d_{-2} \ldots d_{-q}$$

Decimal Codes

Common representation for humans.

Radix = 10.

- Possible values for digits: 0-9

Example:

$1041.2_{10} = 1*10^3 + 0*10^2 + 4*10^1 + 1*10^0 + 2*10^{-1}$

An n-digit decimal number can represent values from 0_{10} to $(10^n-1)_{10}$

Binary Codes

Computers use presence/absence of voltage.

Radix = 2.

- Possible values for digits: 0 and 1

Example:

- $1001.1_2 = 1*2^3 + 0*2^2 + 0*2^1 + 1*2^0 + 1*2^{-1}$
  $= 9.5_{10}$
- $11.01 = 1*2^1 + 1*2^0 + 0*2^{-1} + 1*2^{-2}$
  $= 3.25_{10}$

n-bit **binary number** can represent numbers from

0_{10} to $(2^n-1)_{10}$

- Largest **8-bit** (unsigned) number: $11111111_2 = 255_{10}$

Decimal to Binary Conversion

Algorithm:

```
quot = number;
i = 0;
repeat until quot == 0
    quot = quot/2;
    digit i = remainder;
    i++;
```

Example:

Convert 37_{10} to binary.

Calculation:

- 37/2 = 18 rem 1 least sig. digit
- 18/2 = 9 rem 0
- 9/2 = 4 rem 1
- 4/2 = 2 rem 0
- 2/2 = 1 rem 0
- 1/2 = 0 rem 1 most sig. digit

Result:

- $37_{10} = 100101_2$

Quick Decimal to Binary

Algorithm ("make change" algorithm):

- Let i = largest power of 2 less than or equal to the number.
- $N = N - 2^i$
- Set digit i of result to 1
- repeat until N == 0

Example:

- Convert 37_{10} to binary.
- Calculation:
  - N >= 32, so digit 5 is 1 and N=5
  - N >= 4, so digit 2 is 1 and N=1
  - N >= 1, so digit 0 is 1 and N=0
- Result:
  - $37_{10} = 100101_2$

Note: Use this only to check your answers!

Converting Fractional Numbers

- Algorithm

  i = 0

  repeat until N == 1.0 or i == n

  N = FracPart(N);

  N *= 2;

  digit i = IntPart(N);

  i++

- Example
  - Convert 0.125_{10} to binary
  - Calculation:
    - 0.125*2 = 0.25; IntPart = 0 most significant digit
    - 0.250*2 = 0.50; IntPart = 0
    - 0.500*2 = 1.00; IntPart = 1 least significant digit
  - Result: $0.125_{10} = 0.001_2$

  Convert integer and fractional parts **separately**.

- Many numbers **cannot be represented accurately**:
  - $0.3_{10} = 0.0[1001]..._2$ (bracket repeats, limited by bit size)

Binary Addition

Adding binary numbers:

- 1+0 = 0+1 = 1
- 0 + 0 = 0
- 1 + 1 = 0 carry 1

Binary math works just like decimal math, except that the value of each bit can be only **0 or 1**.

- Possibility of **overflow**

Add 109_{10} to 136_{10}:

$01101101_2 + 10001000_2 = 11110101_2 = 245_{10}$

Add 254_{10} to 2_{10}:

$11111110_2 + 00000010_2 = [1]00000000_2 = 256_{10}$

- We only have 8 bits to store answer...so it's zero!
- Program can generate an “exception”' to let us know.
- Usually number of bits is quite large: 32 bits or 64 bits.

Signed Numbers (representing positive and negative numbers without a '-' sign)

Can use left-most bit to code sign (*sign and magnitude*)

- 0=positive and 1=negative
- Gives symmetric numbers from $-(2^7-1)\ldots 2^7-1$ AND two zeros!
  - This is wasteful: can use extra bit pattern.
- **Addition not straight-forward (bad for hardware** implementors).

Try *one's complement* representation*:*

- negative numbers obtained by flipping signs.
- positive numbers unchanged.
- e.g. -5_{10} = complement(00000101_2) = 11111010_2

- Left-most bit still indicates sign.
- Gives symmetric numbers from $-(2^7-1)\ldots 2^7-1$ AND two zeros.
- Addition easier.

2's Complement Addition

1's Complement still has two zeros.

An alternative to address this is *two's complement*

- Complement then add 1
- Our number range is now asymmetric: $-2^7 \ldots 2^7-1$
- Used extra zero bit pattern
  - 7 = 0111 ... 2 = 0010, 1 = 0001
    0 = 0000
    -1 = 1111, -2 = 1110 ... -8 = 1000

Now when we add, discard carry

10 - 7

= 00001010 + 2complement(00000111)

= 00001010 + 11111001

= 00000011 carry 1 (discard)

= 3

Current computing devices use two's complement –

Principal advantages over 1's complement is single zero and that the addition is faster.

Negative number representations

To get the negative representation for a positive number:

two's complement: invert all bits, then add one.

one's complement: invert all bits.

sign/magnitude: invert just the sign bit.

Alternative for unsigned numbers: Binary Coded Decimal

Can use Binary Coded Decimal (BCD) to represent integers.

- Map 4 bits per digit (from 0000 to 1001)

Wasteful: only 10 bit patterns required – 6 are wasted. Binary code is more compact code.

- Example:
  - $256_{10} = 100000000_2 = 0010\ 0101\ 0110_{BCD}$
  - … 9 vs 12 bits in this case

Complicates hardware implementation.

- How to add/subtract, deal with carries etc.

But more readable, *most pocket calculators do all the calculations in BCD.*

Octal and Hexadecimal

Base 8 (octal) and base 16 (Hexadecimal) are sometimes used to **concisely** represent **binary** (both are powers of 2).

Octal uses digits 0-7.

Hex uses “digits” 0-9, A-F.

Examples: $17_{10} = 10001_2 = 21_8 = 11_{16}$

Conversion as for decimal to binary:

- divide/multiply by 8 or 16 instead

Binary to octal or hexadecimal is **simple**:

- group bits into 3 (octal) or 4 (hex) from LS bit.
- pad with leading zeros if required.

Hexadecimal colours in HTML (and elsewhere)

Typical HTML code to create background colors:

<body bgcolor="#FFFFFF"> for an all white background

<body bgcolor="#CCCCCC"> for a light-gray background

The three PRIMARY colors (RED, GREEN and BLUE) each assigned TWO HEXADECIMAL DIGITS (=1 byte).

White	#FFFFFF
Silver	#C0C0C0
Gray	#808080
Black	#000000
Red	#FF0000
Maroon	#800000
Yellow	#FFFF00
Olive	#808000
Lime	#00FF00
Green	#008000
Aqua	#00FFFF
Teal	#008080
Blue	#0000FF
Navy	#000080
Fuchsia	#FF00FF
Purple	#800080

Floating Point Numbers

Fixed point numbers have very limited range (determined by bit length).

A 32-bit value can hold integers from -2^{31} to $2^{31}-1$ or smaller range of fixed point fractional values.

- Solution: use *floating point* (scientific notation)

  Thus $0.000000000000976 = 9.76*10^{-14}$

Consists of two parts: **mantissa** & **exponent**

- **Mantissa:** the number multiplying the base
- **Exponent:** the power

$\mathbf{9.76}*10^{-14}$

- The **significand** is the part of the mantissa after the decimal point

$\mathbf{9.76}*10^{-14}$

Floating Point Range

Range of numbers is very large, but accuracy is limited by significand.

- So, for 8 digits of precision in decimal:

  $97637529732 1 = 9.7637529*10^{11}$,

  and we lose accuracy (**truncation error**)

We can normalise any floating point number:

$34.34*10^{12} = 3.434*10^{13}$

- Shift point until only one non-zero digit is to left.
  - add 1 to exponent for each left shift
  - subtract 1 for each right shift

Binary Floating Point

- We can use FP notation for binary: use base of 2

  $0.111001*2^{-3} = 1.11001*2^{-4} = 1.11001 * 2^{11111100}$ (2's complement exponent)

- For binary FP numbers, normalise to:

  $1.xxx\ldots xxx*2^{yy\ldots yy}$

- Problems with FP:
  - Many different floating point formats; problems exchanging data.
  - FP arithmetic not associative: x + (y + z) != (x + y) + z

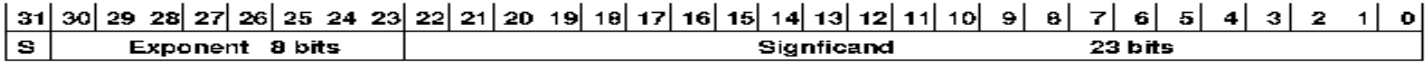
```
x=(0.1+0.2)+0.3
y=0.1+(0.2+0.3)
>>> x==y
False
```

Floating Point Formats

IEEE 754 format introduced: **single** (32-bit) and **double** (64-bit) formats; much needed standard!

- Also extended precision - 80 bits (long double).

- Single precision number represented internally as
  - sign bit
  - followed by exponent (8-bits)
  - then the fractional part of normalised number (23 bits)
- The leading 1 is implied; **not stored.**
- Double precision
  - has 11-bit exponent and
  - 52-bit significand
- Single precision range: $2*10^{-38}$ to $2*10^{38}$
- Double range: $2*10^{-308}$ to $2*10^{308}$



IEEE 754 - Single (32-bit) floating point format

Floating Point Templates

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	11-bit Exponent											Signficand 20 bits																			

Significand (cont'd) 32-bits

IEEE 754 - Double (64-bit) floating point format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	Exponent 8 bits								Signficand 23 bits																						

IEEE 754 - Single (32-bit) floating point format

Floating Point Exponents

The exponent is “biased‘”: no explicit negative number.

- Single precision: 127, Double precision 1023

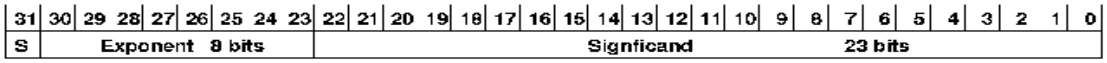
- So, for single precision:

stored value of 255 indicated exponent of 255-127 = 128,
and 0 is 0 - 127 = -127 (can't be symmetric, because of zero)

Most positive exponent: 111...11, most negative: 00....000

- Makes some hardware/logic easier for exponents (easy sorting/compare).
- Numeric value of stored IEEE FP is actually:

$$(-1)^S * (1 + \text{significand}) * 2^{\text{exponent - bias}}$$



IEEE 754 - Single (32-bit) floating point format

Real to IEEE754 Single

Example

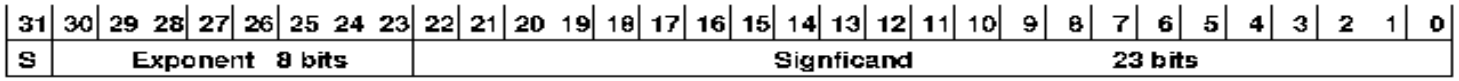
- Convert -0.75 to IEEE 754 Single Precision

- Calculation
  - Sign is negative: so S = 1
  - Binary fraction:

    0.75*2 = 1.5 (IntPart = 1)

    0.50*2 = 1.0 (IntPart = 1), so $0.75_{10} = 0.11_2$
  - Normalise: $0.11*2^0 = 1.1*2^{-1}$
  - Exponent: -1, add bias(127) = 126 = 01111110;
- Answer: **1** 01111110 **10000000000000000000000**

s 8 bits **23 bits**



IEEE 754 - Single (32-bit) floating point format

IEEE754 Single to Real

Example

- What is the value of the FP number:
  1 10000001 10010000000000000000000

- Calculation
  - Negative number (s=1)
  - Biased exponent: 10000001 = 128+1 = 129
  - Actual exponent = 129-127 = 2
  - Significand: 0.1001 = 0.5+0.0625 = 0.5625
- Result = $-1 * (1 + 0.5625)*2^2 = -6.25_{10}$

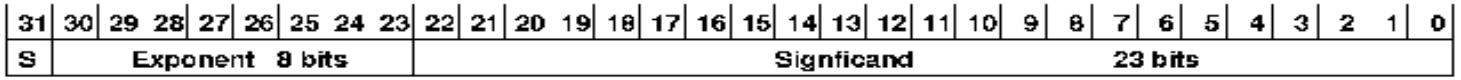
For checking your work…

https://www.h-schmidt.net/FloatConverter/IEEE754.html

IEEE 754 Special Codes

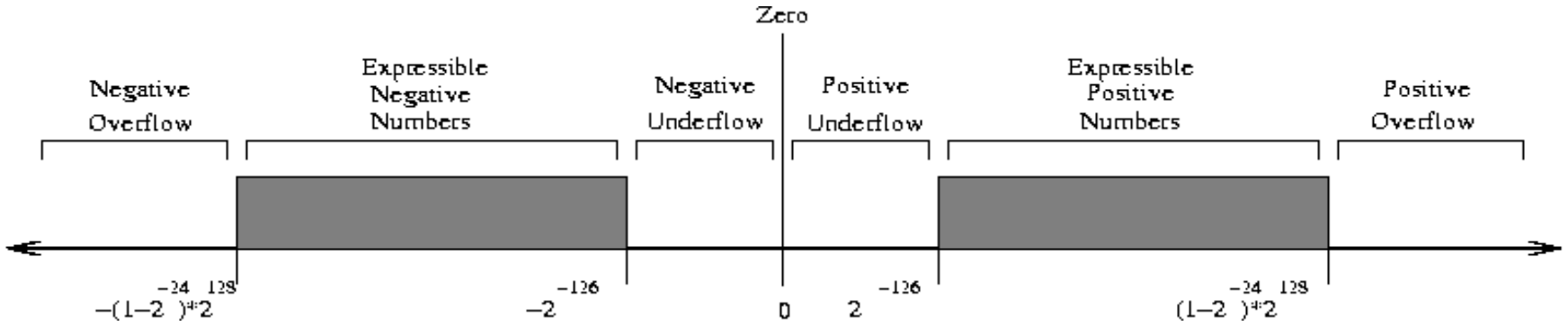
IEEE 754 has special codes for zero, error conditions (0/0 etc).

- **Zero**: exponent and significand are zero.
- **Infinity**: exp = 1111...1111, significand = 0
- **NaN** (not a number): 0/0; exponent = 1111...1111, significand != 0
- Underflow/overflow conditions…



IEEE 754 - Single (32-bit) floating point format

Range of Single Precision FPs



Overflow means that values have grown too large for the representation, much in the same way that you can overflow integers.

Underflow is a less serious problem because is just denotes a loss of precision, which is guaranteed to be closely approximated by zero.

FP Operations and Errors

Addition/Subtraction: normalise, match to larger exponent then add, normalise.

- Error conditions:
  - **Exponent Overflow** Exponent bigger than max permissable size; may be set to “infinity”.
  - **Exponent Underflow** Negative exponent, smaller than minimum size; may be set to zero.
  - **Significand Underflow** Alignment may cause loss of significant digits.
  - **Significand Overflow** Addition may cause carry overflow; realign significands.

Bit/Byte Ordering

All you need to know about memory is that it's one large array containing *bytes (= 8 bits)*.

Endianness: ordering of bits or bytes in computer

- Big Endian: bytes ordered from most significant byte (**MSB**) to least significant byte (**LSB**)
- Little Endian: bytes ordered from LSB to MSB

Example: how is **$90AB12CD_{16}$** (32-bit) represented? Split the 32 bit quantity into 4 bytes

- Big-Endian: 90AB12CD (lowest byte address stores MSB)
- Little-Endian: CD12AB90 (lowest byte address stores LSB)

Problems with multi-byte data: floats, ints, etc

- MIPS, IBM z/Architecture mainframes and internet are Big-Endian, Intel x86 Little-Endian

Address	Value
1000	90
1001	AB
1002	12
1003	CD

Big-Endian

Address	Value
1000	CD
1001	12
1002	AB
1003	90

Little-Endian

Character Representations

- Characters represented using “character set”.
- Examples:
  - ASCII (7/8-bit)
  - Unicode (16-bit, with variable-length encoding)
  - EBCDIC (9-bit)
- ASCII - American Standard Code for Information Interchange
  - Used to be widely used; 7-bits used for std characters etc.; extra for parity or foreign language.

Character Codes

- ASCII codes for Roman alphabet, numbers, keyboard symbols and basic network control.

- Unicode is very popular today: subsumes ASCII, extensible, supported by most languages and OSes.
  - Handles many languages, not just Roman alphabet and basic symbols.

Boolean Algebra & Logic

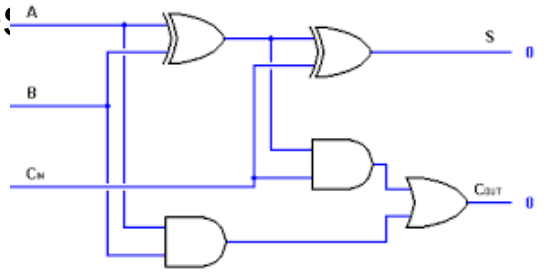
Modern computing devices are digital rather than analog

- Use two discrete states to represent all entities: 0 and 1
- Call these two logical states **TRUE** and **FALSE**

All operations are on these values and can only yield these values.

- George Boole formalised such a logic algebra as “Boolean Algebra”.
- Modern digital circuits are designed and optimised using this theory.
- We implement “functions” (such as add, compare, etc.) in hardware, using corresponding Boolean expres

In a computer chip, the transistors aren't isolated, they are part of an **integrated circuit** (or **microchip**). Computers use transistors in tandem with Boolean algebra to make simple decisions.



Boolean Operators

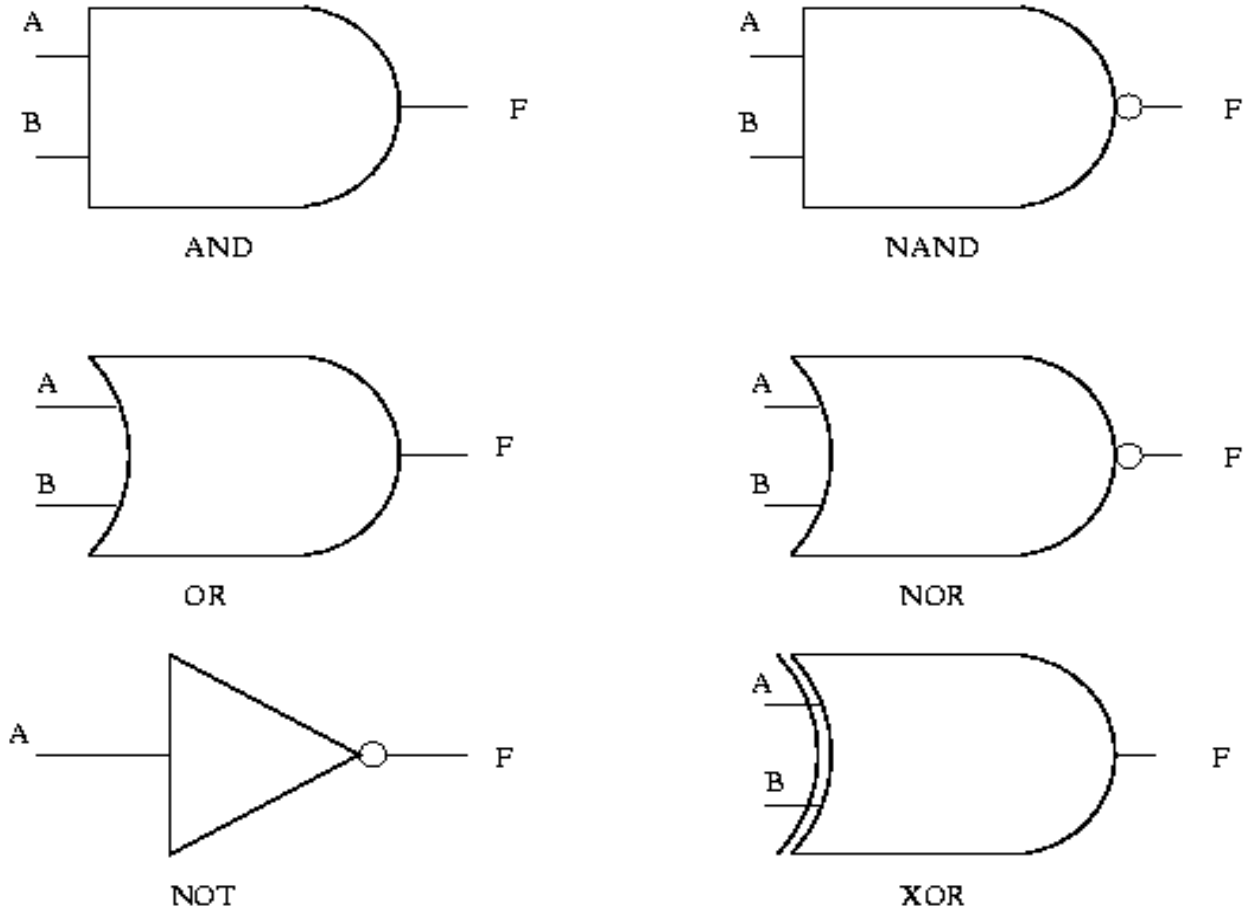
- There are 3 basic logic operators:

Operator	Usage	Notation
AND	A AND B	A.B
OR	A OR B	A+B
NOT	NOT A	A

The arguments (A, B) can only be TRUE or FALSE

- TRUE represented by 1; FALSE by 0

Logic Gate Symbols



Truth Tables

A **truth table** shows the value of each operator (or combinations thereof).

- AND is TRUE only if both arguments are TRUE
- OR is TRUE if either is TRUE
- NOT is a unary operator: inverts truth value

A	B	$F = A.B$	$F = A + B$	$F = \overline{A}$	$F = \overline{B}$
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	0	1
1	1	1	1	0	0

NAND, NOR and XOR

- NAND is FALSE only if both args are TRUE

  [NOT (A AND B)]

- NOR is TRUE only if both args are FALSE

  [NOT (A OR B)]

- XOR is TRUE is either input is TRUE, but not both

A	B	$F=\overline{A.B}$	$F=\overline{A+B}$	$F=A \oplus B$
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	0	0	0

Checkpoint

- What is the Boolean operation *F* represented by this truth table?

A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

Checkpoint

- What is the Boolean operation *F* represented by this truth table?

A	B	F
0	0	0
0	1	1
1	0	1
1	1	1

Checkpoint

- What is the Boolean operation *F* represented by this truth table?

A	B	F
0	0	1
0	1	0
1	0	0
1	1	0

Logic Gates

The Boolean operators have symbolic representations: “logic gates”.

- These are the building blocks for all computer circuits.

To work out which logic gates are required for an operation, specify the function, F, using a truth table; then derive the Boolean expression. Then simplify.

What Boolean Expression is represented by this truth table?

F = F(A,B,C); F called “output variable”

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Finding a Boolean Representation

F = F(A,B,C); F called “output variable”

- Find F values which are TRUE:

So, if A=0, B=1, C=0, then F = 1.

So, $F_1 = \bar{A}.B.\bar{C}$

That is, we know our output is TRUE for this expression (from the table).

Also have $F_2 = \bar{A}.B.C$ and $F_3 = A.B.\bar{C}$

F TRUE if F_1 TRUE or F_2 TRUE or F_3 TRUE

$\Rightarrow F = F_1 + F_2 + F_3$

Cases for F FALSE follows from F TRUE

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Algebraic Identities

Commutative: $A.B = B.A$ and $A+B = B+A$

Distributive:

$A.(B+C) = (A.B) + (A.C)$

$A+(B.C) = (A+B).(A+C)$

Identity Elements: $1.A = A$ and $0 + A = A$

Inverse: $A.\overline{A} = 0$ and $A + \overline{A} = 1$

Associative:

$A.(B.C) = (A.B).C$ and $A+(B+C) = (A+B)+C$

De Morgan's Laws:

$\overline{A.B} = \overline{A} + \overline{B}$ and

$\overline{A+B} = \overline{A}.\overline{B}$

De Morgans laws

- NOT (A AND B) = (NOT A) OR (NOT B)

A	B	A.B	A+B	NOT(A.B)	NOT A + NOT B
0	0	0	0	1	1
0	1	0	1	1	1
1	0	0	1	1	1
1	1	1	1	0	0

De Morgans laws

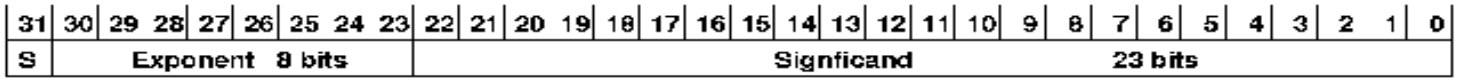
- NOT (A OR B) = (NOT A) AND (NOT B)

A	B	A.B	A+B	NOT(A+B)	NOT A AND NOT B
0	0	0	0	1	1
0	1	0	1	0	0
1	0	0	1	0	0
1	1	1	1	0	0

Useful for simplifying expressions

Checkpoint: Real to IEEE754 Single precision

Convert 20.2 to IEEE 754 Single Precision



IEEE 754 - Single (32-bit) floating point format

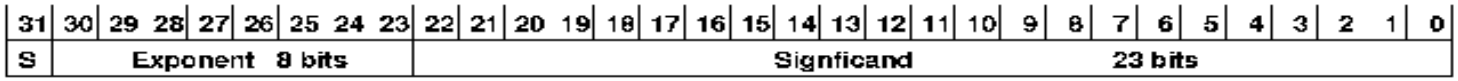
Checkpoint: Real to IEEE754 Single precision

Convert 20.2 to IEEE 754 Single Precision

- Answer: **0** 10000011 **01000011001100110011001**

  s 8 bits **23 bits**

Check with $(-1)^S * (1 + \text{significand}) * 2^{\text{exponent - bias}}$



IEEE 754 - Single (32-bit) floating point format

IEEE754 Single to Real

Example

- What is the value of the FP number:
  0 01111111 11100000000000000000000

- Calculation
  - positive number (s=0)
  - Biased exponent: 01111111 = 127
  - Actual exponent = 127-127 = 0
  - Significand: 0.111 = 0.5+0.25+0.125 = 0.875
- Result = $1 * (1 + 0.875)*2^0 = 1.875_{10}$