Machine Architecture
Binary Arithmetic and Floating Point

Dr. Magnus Bordewich

Durham University

Positional number systems

We start with a particular ordered set of symbols. E.g. a,b,c or 0,1,2

The **base** (or **radix**) of the number system is the number of symbols. E.g. 3

We use positional number systems to represent values

$cab.bc_3$ or 201.12_3

Note: subscript after a number gives the base

The contribution of a symbol *x*, which is the i^{th} symbol in the order, is $(i\text{-}1)$*$\text{base}^{\text{position}}$, where position is number of places to the **left** of the units.

Adding in decimal

```
  11
 1234
+5678
 6912
```

Adding in binary

```
 111
  11100
+ 01110
 101010
```

Based on 8 simple rules:

0 + 0 = 0	Carry + 0 + 0 = 1
0 + 1 = 1	Carry + 0 + 1 = 0 with Carry
1 + 0 = 1	Carry + 1 + 0 = 0 with Carry
1 + 1 = 0 with Carry	Carry + 1 + 1 = 1 with Carry

Overflow

Suppose the accumulator in your CPU is an 8-bit register.
It has 11010010 in it.
You execute the instruction ADD 01010000.

What happens?

```
 11010010
+01010000
100100010
```

The answer **doesn't fit** in the register.

"An error that occurs when the computer attempts to handle a number that is too large for it. Every computer has a well-defined range of values that it can represent. If during execution of a program it arrives at a number outside this range, it will experience an overflow error." [webopedia]

This should trigger a flag in the **status register**, but can cause errors.

Ariane 5

Cause: trying to fit too large a number in a 16-bit register

Overflow error

Multiplication

The same as decimal long multiplication – but easier!

```
    11100
   *01110
    00000
   111000
  1110000
 11100000
000000000
110001000
```

Can be efficiently accomplished with **left-shift** and **add** operations

Decimal example: 5324*123 = 15972 + 106480 + 532400 = 654852

Negative numbers

18, -17, 5749, -0.684,…
How can we represent **negative numbers using only bits?**

Common solutions:

Signed Magnitude Representation:

- add a single-bit flag: **0** for positive or **1** for negative
- **0**000 0110 = 6
- **1**000 0110 = -6 **NOT 134**
- Similar in concept to a minus sign.
- Have two values for 0: 1000 0000 and 0000 0000
- Makes binary arithmetic messy

Negative numbers

Ones-complement:

- The negative of a number is represented by flipping each bit
- For example 0100 1001_2 = 65_{10} becomes 1011 0110_2 = -65_{10}
- The higher order bit still indicates the sign of the number.
- Still has two representations for zero: 00000000 and 11111111
- Makes binary addition a bit simpler

Twos-complement:

- A negative number is obtained by flipping each bit and adding 1.
- For example 0100 1001_2 = 65_{10} becomes 1011 0111_2 = -65_{10}
- The higher order bit still indicates the sign of the number.
- One representation for zero: 00000000. (11111111 is -1.)
- Makes binary arithmetic much simpler.

Negative numbers

Add a bias:

- For k-bit numbers add a bias of 2^{k-1}-1, then store in normal binary. (So for 8-bit add 2^7-1 = 127.)
- Can store numbers between –(2^{k-1}-1) and 2^{k-1}, (-127 and 128)
- For example -65_{10} stored as $-65+127_{10} = 62_{10}$ becomes $0011\ 1110_2$
- The higher order bit **does not indicate** the sign of the number in the normal way.
- Used in storing floating point numbers for some reason!

Negative numbers

We will stick with **Twos-complement**.
We need to be careful about how many bits we are using to represent a number:

4-bits: $3_{10} = 0011_2$, $-3_{10} = 1101_2$
8-bits: $3_{10} = 0000\ 0011_2$, $-3_{10} = 1111\ 1101_2$

Subtracting is now the same as adding: 10 – 3 = 10 + (-3)
$10_{10} = 0000\ 1010_2$, $3_{10} = 0000\ 0011_2$
00001010 – 0000 0011 = 0000 1010 + 1111 1101 = ~~1~~ 0000 0111
Overflow is ignored

Note: 1000 0000 is its own negative! It is always taken to be -128

0000 0000 = 0
0000 0001 = 1
…
0111 1111 = 127
1000 0000 = -128
1000 0001 = -127
…
1111 1111 = -1

Floating point representation

Sometimes we need to deal with numbers outside the usual range:
124000000000000000000000000000000000000000000000000000000000
How can we represent this without using masses of digits?
Scientific notation: 1.24 * 10^{57}

Floating point is very like 'scientific notation'
The typical floating-point representation has three fields:

- **The sign bit** **S**
- **The exponent** **e**
- **The mantissa** **M** (also called the significand)

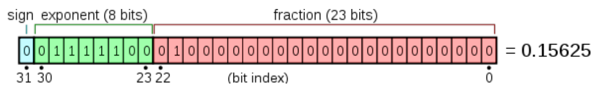
Floating point representation

- **The sign bit S**
- **The exponent e**
- **The mantissa M**

Representing the number + or - M * 2^e

Single precision (32-bit) floating point numbers have:

- 1-bit sign
- 8-bit exponent
- 23-bit mantissa



But you can find other standards and default choices.

Floating point representation

The sign bit S

0 indicates a positive number

1 indicates a negative number!

Floating point representation

The exponent e

Value in the range -126 to 127

Stored with **a bias**: 127 is added giving a number between 1 and 254

The 8-bit exponent field can store values in the range 0 to 255, but 0 and 255 have **special meanings:**

- exponent field 0 with mantissa 0 gives the number zero.
- exponent field 0 with non-zero mantissa: “subnormal numbers”.
- exponent field 255 with mantissa 0 gives + or - infinity.
- exponent field 255 with non-zero mantissa: not a number.

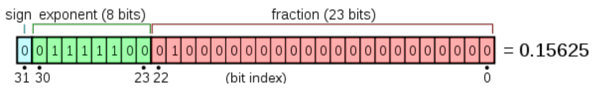
Floating point representation

The mantissa M

Some binary number like 1.10101010110
Always scaled so that the radix point is after the leading 1.
Hence we need not store the leading 1 (we can assume it is there).
We only store 23 binary digits of the fractional part: 10101010110…

Floating point representation

Example:



- sign 0 – a positive number.
- exponent field is 124, so e is 124-127 = -3.
- mantissa field is 010… so the actual mantissa is 1.010000…=1.25
- $1.25*2^{-3} = 1.25/8 = 0.15625$

Floating point representation

Example:

-12.375
$12.375_{10} = 1100.011_2$
$1100.011 = 1.100011 *2^3$

- Sign is 1 to represent a negative
- Mantissa is 1.100011, we will store 100011000…
- Exponent is 3, we will store 130_{10} = $1000\ 0010_2$ after adding the bias.

11000001010001100000000000000000

Floating point representation

What is the binary FP representation of 0.1_{10}?
$0.1_{10} = 0.000110011001100110011001 1\ldots_2$

So the FP has e = −4; M = 1.10011001100110011001101 (limited to 23 digits)

which is actually 0.100000001490116119384765625.
A **rounding error**.

Minimum positive number is 2^{-126}, the **underflow level**.
Maximum positive number is $(2-2^{-23}) \times 2^{127}$, the **overflow level**.

Floating Point Operations should return the closest FP number to the answer. E.g. $1.1*2^{123} - 1.10101*2^{-23} = 1.1*2^{123}$

Ariane 5

Failure converting 64-bit floating point to 16-bit signed integer.

```
P_M_DERIVE(T_ALG.E_BH) :=
UC_16S_EN_16NS (TDB.T_ENTIER_16S ((1.0/C_M_LSB_BH) * G_M_INFO_DERIVE(T_ALG.E_BH)));
```

Convert to 16-bit integer
Multiply by a constant
Horizontal velocity measured as 64-bit floating point

Overflow error lead to total loss of the rocket and cargo.

The failure resulted in a loss of more than US$370 million.

Ariane 5

Solution:

```
L_M_BH_32 := TBD.T_ENTIER_32S ((1.0/C_M_LSB_BH) * G_M_INFO_DERIVE(T_ALG.E_BH));

if L_M_BH_32 > 32767 then
        P_M_DERIVE(T_ALG.E_BH) := 16#7FFF#;
elsif L_M_BH_32 < -32768 then
        P_M_DERIVE(T_ALG.E_BH) := 16#8000#;
else
        P_M_DERIVE(T_ALG.E_BH) := UC_16S_EN_16NS(TDB.T_ENTIER_16S(L_M_BH_32));
end if;
```

Check if the number is outside the range before conversion.
If too large – set at a max value.