Number Systems

Digital Design and Computer Architecture
Mohammad Sadrosadati
Frank K. Gürkaynak

http://safari.ethz.ch/ddca

What will we learn?

- **How to represent fractions?**
- **Fixed point**
- **Floating point**
- **Briefly:**
  - Adding floating point numbers
  - Life is a bit more complicated

Number Systems

- **For what kind of numbers do you know binary representations?**
  - ***Positive integers***
    Unsigned binary
  - ***Negative integers***
    Sign/magnitude numbers
    Two’s complement

- **What about fractions?**

Fractions: Two Representations

- ***Fixed-point***: **binary point is fixed**

  `1101101.0001001`

- ***Floating-point***: **binary point floats to the right of the most significant 1 and an exponent is used**

  $1.1011010001001 \times 2^6$

Fixed-Point Numbers

- **Fixed-point representation using 4 integer bits and 3 fraction bits:**

```
                 0110110
interpreted as   0110.110
               = ?
```

Fixed-Point Numbers

- **Fixed-point representation using 4 integer bits and 3 fraction bits:**

```
        0110110
interpreted as 0110.110
      = 2^2 + 2^1 + 2^-1 + 2^-2 = 6.75
```

interpreted as $0110.110 = 2^{2} + 2^{1} + 2^{-1} + 2^{-2} = 6.75$

- **The binary point *is not a part* of the representation but is implied**

- **The number of integer and fraction bits must be *agreed upon* by those generating and those reading the number**

Signed Fixed-Point Numbers

- **Negative fractional numbers can be represented two ways:**
  - Sign/magnitude notation
  - Two's complement notation

- **Represent -7.5_{10} using an 8-bit binary representation with 4 integer bits and 4 fraction bits in Two's complement:**
  - +7.5: `01111000`
  - Invert bits: `10000111`
  - Add 1 ***to lsb***: `10001000`

Floating-Point Numbers

- **The binary point floats to the right of the most significant digit**
- **Similar to decimal scientific notation:**
  - For example, 273_{10} in scientific notation is

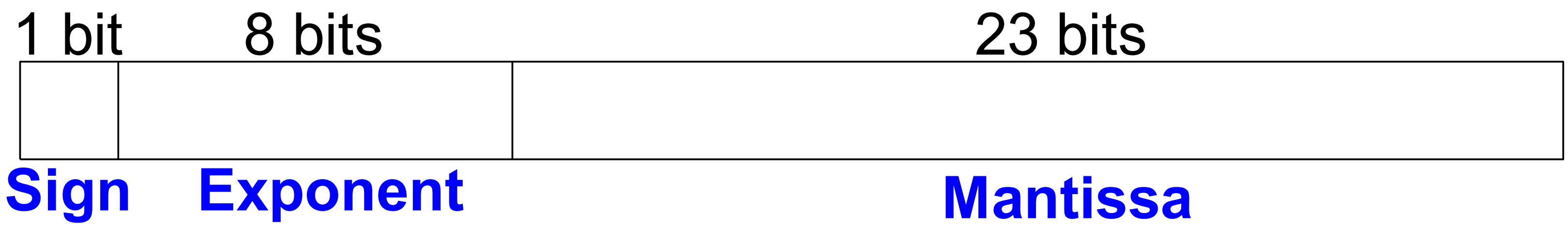
$$273 = 2.73 \times 10^2$$

- **In general, a number is written in scientific notation as:**

$$\pm M \times B^E$$

  Where:
  - **M** = mantissa
  - **B** = base
  - **E** = exponent
- **In the example, M = 2.73, B = 10, and E = 2**

Floating-Point Numbers



- Example: represent the value 228_{10} using a 32-bit floating point representation
- We show three versions; the final version is used in the IEEE 754 floating-point standard

Floating-Point Representation 1

- **Convert the decimal number to binary:**

$$228_{10} = 11100100_2 = 1.11001 \times 2^7$$

- **Fill in each field of the 32-bit number:**
  - The sign bit is positive (0)
  - The 8 exponent bits represent the value 7
  - The remaining 23 bits are the mantissa

1 bit	8 bits	23 bits
0	00000111	11 1001 0000 0000 0000 0000
Sign	**Exponent**	**Mantissa**

Floating-Point Representation 2

- **First bit of the mantissa is always 1:**

$$228_{10} = 11100100_2 = 1.11001 \times 2^7$$

  - Thus, storing the most significant 1, also called the implicit leading 1, is redundant information

- **Instead, store just the fraction bits in the 23-bit field**

  The leading 1 is implied

1 bit	8 bits	23 bits
0	00000111	110 0100 0000 0000 0000 0000
Sign	**Exponent**	**Fraction**

Floating-Point Representation 3 (IEEE)

- **Bias for 8 bits** **= 127_{10} = 01111111_2**
- **Biased exponent** **= bias + exponent**
  - Exponent of 7 is stored as:

    127 + 7 = 134 = 10000110_2

- **The IEEE 754 32-bit floating-point representation of 228_{10}**

1 bit	8 bits	23 bits
0	10000110	110 0100 0000 0000 0000 0000
Sign	**Biased Exponent**	**Fraction**

Floating-Point Example

Write the value -58.25_{10} using IEEE 754 32-bit floating-point standard

- **First, convert the decimal number to binary:**

  58.25_{10} =

- **Next, fill in each field in the 32-bit number:**
  - Sign bit:
  - 8 exponent bits:
  - 23 fraction bits:

Floating-Point Example

Write the value -58.25_{10} using IEEE 754 32-bit floating-point standard

- **First, convert the decimal number to binary:**

  $58.25_{10} = 111010.01_2 = 1.\mathbf{1101001} \times 2^{5}$

- **Next, fill in each field in the 32-bit number:**
  - Sign bit: **1** (negative)
  - 8 exponent bits: $(127 + 5) = 132_{10} = \mathbf{10000100}_2$
  - 23 fraction bits: $\mathbf{110\ 1001\ 0000\ 0000\ 0000\ 0000}_2$

1 bit	8 bits	23 bits
1	100 0010 0	110 1001 0000 0000 0000 0000
Sign	**Exponent**	**Fraction**

In hexadecimal: `0xC2690000`

Floating-Point Numbers: Special Cases

- **The IEEE 754 standard includes special cases for numbers that are difficult to represent, such as 0 because it lacks an implicit leading 1**

Number	Sign	Exponent	Fraction
0	X	00000000	00000000000000000000000
∞	0	11111111	00000000000000000000000
- ∞	1	11111111	00000000000000000000000
NaN	X	11111111	non-zero

- **NaN (= Not a Number) is used for numbers that don't exist, such as sqrt(-1) or log(-5)**

Floating-Point Number Precision

- ***Single-Precision*:**
  - 32-bit notation
  - 1 sign bit, 8 exponent bits, 23 fraction bits
  - bias = 127
- ***Double-Precision*:**
  - 64-bit notation
  - 1 sign bit, 11 exponent bits, 52 fraction bits
  - bias = 1023

Floating-Point Numbers: Rounding

- **Problems:**
  - ***Overflow:*** number is too large to be represented
  - ***Underflow:*** number is too small to be represented

- **Rounding modes:**
  - Down
  - Up
  - Toward zero
  - To nearest

Floating-Point Numbers: Rounding Example

- **Round `1.100101` (`1.578125`) so that it uses only 3 fractional bits**
  - Down:
  - Up:
  - Toward zero:
  - To nearest:

Floating-Point Numbers: Rounding Example

- **Round `1.100101` (`1.578125`) so that it uses only 3 fractional bits**
  - Down: `1.100`
  - Up: `1.101`
  - Toward zero: `1.100`
  - To nearest: `1.101` (1.625 is closer to 1.578125 than 1.5 is)

Floating-Point Addition

- **Steps for floating point addition:**
  1. Extract exponent and fraction bits
  2. Prepend leading 1 to form mantissa
  3. Compare exponents
  4. Shift smaller mantissa if necessary
  5. Add mantissas
  6. Normalize mantissa and adjust exponent if necessary
  7. Round result
  8. Assemble exponent and fraction back into floating-point format

- **Not so easy as binary addition!**

Floating-Point Addition: Example

- **Add the following floating-point numbers:**

```
0x3FC00000
0x40500000
```

Floating-Point Addition: Example

1. Extract exponent and fraction bits

1 bit	8 bits	23 bits
0	01111111	100 0000 0000 0000 0000 0000
Sign	**Exponent**	**Fraction**

1 bit	8 bits	23 bits
0	10000000	101 0000 0000 0000 0000 0000
Sign	**Exponent**	**Fraction**

- For first number (N1): S = `0`, E = `127`, F = **`.1`**
- For second number (N2): S = `0`, E = `128`, F = **`.101`**

2. Prepend leading 1 to form mantissa

- N1: **`1.1`**
- N2: **`1.101`**

Floating-Point Addition: Example

3. Compare exponents

`127 – 128 = -1` so shift N1 right by 1 bit

4. Shift smaller mantissa if necessary

shift N1's mantissa:

$1.1 \gg 1 = 0.11 \quad (\times 2^1)$

5. Add mantissas

$$\begin{array}{r} 0.11 \times 2^1 \\ +\ 1.101 \times 2^1 \\ \hline 10.011 \times 2^1 \end{array}$$

Floating-Point Addition: Example

6. Normalize mantissa and adjust exponent if necessary

$$10.011 \times 2^1 = 1.0011 \times 2^2$$

7. Round result

No need (fits in 23 bits)

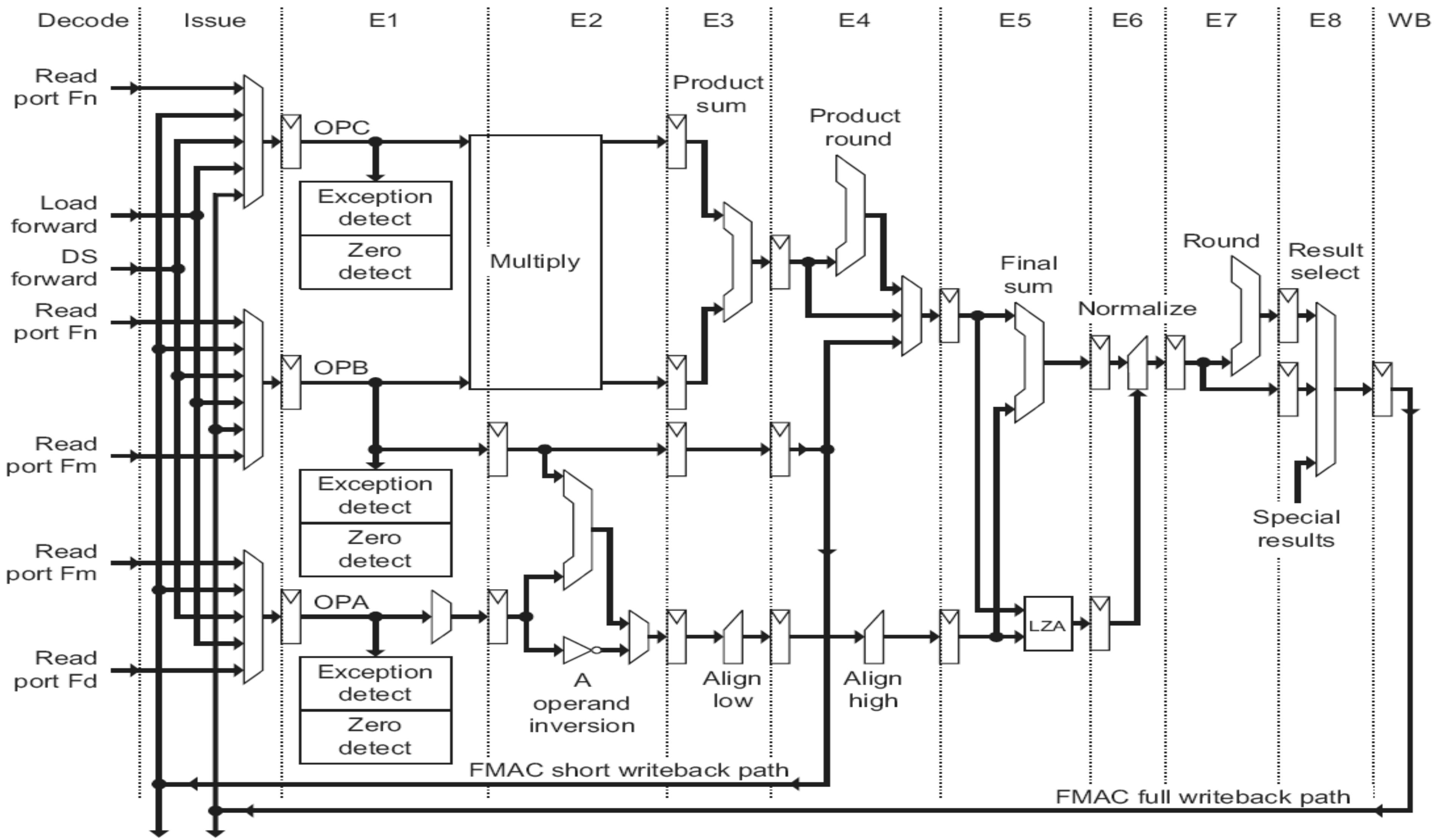
8. Assemble exponent and fraction back into floating-point format

S = `0`, E = 2 + 127 = 129 = $\texttt{10000001}_2$, F = `001100..`

1 bit	8 bits	23 bits
0	10000001	001 1000 0000 0000 0000 0000
Sign	**Exponent**	**Fraction**

Written in hexadecimal: `0x40980000`

Floating-Point Unit of ARM



What did we learn

- **How to express real numbers in binary**
  - Fixed point
  - Floating point
- **IEEE Standard to express floating point numbers**
  - Sign
  - Exponent (biased)
  - Mantissa
- **Briefly**
  - Adding floating point numbers