UNIT 4: FLOATING POINT ARITHMETIC

Estructura de Computadores (Computer Organization)

Course 2018/2019

ETS Ingeniería Informática

Universitat Politècnica de València

Unit goals

- To visit some of the issues related to the representation of real numbers in digital computers
- To learn the measurement units employed for operators on real numbers
- To learn the programmer’s view of real arithmetic and the associated resources on a MIPS processor
- To grasp the basics for designing floating point operators

Bibliography

- D. Patterson, J. Hennessy. *Computer organization and design. The hardware/software interface*. 4th edition. 2009. Elsevier
- W. Stallings. *Computer Organization and Architecture. Designing for Performance*. 7th edition. 2006. Prentice Hall
- D. Goldberg: *Computer Arithmetic*
  - Appendix H of J. Hennessy, D. Patterson: Computer Architecture: A Quantitative Approach. 4th Edition. Morgan-Kaufmann, 2002
    - (PDF) http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.65.3375&rep=rep1&type=pdf
    - (PDF) http://www.cs.clemson.edu/~mark/464/appH.pdf
- D. Goldberg: *What every computer scientist should know about floating-point arithmetic*
  - PDF available in a number of web sites (just search by title)

Unit contents

Introduction

- Floating-point (FP) is an approximation for the representation of real numbers in digital computers
- FP arithmetic may or may not be supported by hardware operators
  - FP is not essential for a CPU to work
  - FP operations can be *emulated* by software using integer arithmetic
- FP hardware evolution
  - Up to mid 80's, FP operators resided in a separate chip close to the CPU (the *Floating-Point Coprocessor*)
  - From then on, most general-purpose CPUs include FP operators
  - Since the 90's, graphical adapters include Graphic Processing Units (GPUs) with an increasing number of FP operators

Measuring FP performance

- Most used metric: Floating Point Operations per Second (FLOPS), with prefixes M=10^6, G=10^9, T=10^{12}, P=10^{15}
- FLOPS are commonly used in two contexts:
  - FP operator design: inversely proportional to the operator's delay
  - Comparatives between computers or GPUs: number of FP operations the device is able to perform per second
    - Depends on the number and characteristics of FP operators and how they can be used (degree of parallelism)
- The *peak FP performance* of a CPU or GPU is the sum of the productivities of its FP operators
  - It's never attainable under normal circumstances
    - no realistic program makes use of FP instructions only
    - it's almost impossible for any program to make the most effective use of all operators all the time

Peak performance

Intel Core2 Duo @2GHz
16 GFLOPS

Intel Core i7 965 XE
70 GFLOPS

GPU ATI Radeon HD7990
8.2 TFLOPS

Sunway TaihuLight (China)
93 PFLOPS
(1st place in Top500.org, Nov 2017)

2. IEEE 754 and MIPS

- Notes on representation of real numbers
- Formats, special values, rounding
- Programmer's view: register file and data transfer
- Floating point instruction set

Notes on representation of real numbers

- $\mathcal{R}$ is a *dense set*: infinite values among two any values
- Computer representation is limited and often inexact
  - With *n* bits only 2^n numbers can be represented
- Some numbers have an exact representation; others don't
  - ½, ¼, … are representable; ⅓, or π are not
- Real numbers are coded using arbitrary formats, e.g. IEEE 754
  - Uses three fields for sign (S), exponent (E) and mantissa (M, AKA *significand*)
  - Base 10 example:
    - The number −4,595.3 is represented as -4.5953×10^3, where sign = "−"; exponent = 3; mantissa = 45953
- The IEEE 754 format also reserves some combinations for special values, such as zero or ∞

Notes on representation of real numbers

- Numbers with integer and fractional part are also represented in a positional system

Bit:	3rd	2nd	1st	0th	.	-1st	-2nd	-3rd	-4th
Weight:	2^3	2^2	2^1	2^0	.	2^{-1}	2^{-2}	2^{-3}	2^{-4}

- What is 0.1011_2 in decimal?
  - $0.1011 = 2^{-1} + 2^{-3} + 2^{-4} = 0.5 + 0.125 + 0.0625 = 0.6875$
- How to obtain the fractional part in binary?
  - $0.6875 \times 2 = 1.375$
  - $0.375 \times 2 = 0.75$
  - $0.75 \times 2 = 1.5$
  - $0.5 \times 2 = 1.0$
  - $0.0 \times 2 = 0.0$
  - … (rest all zeroes)

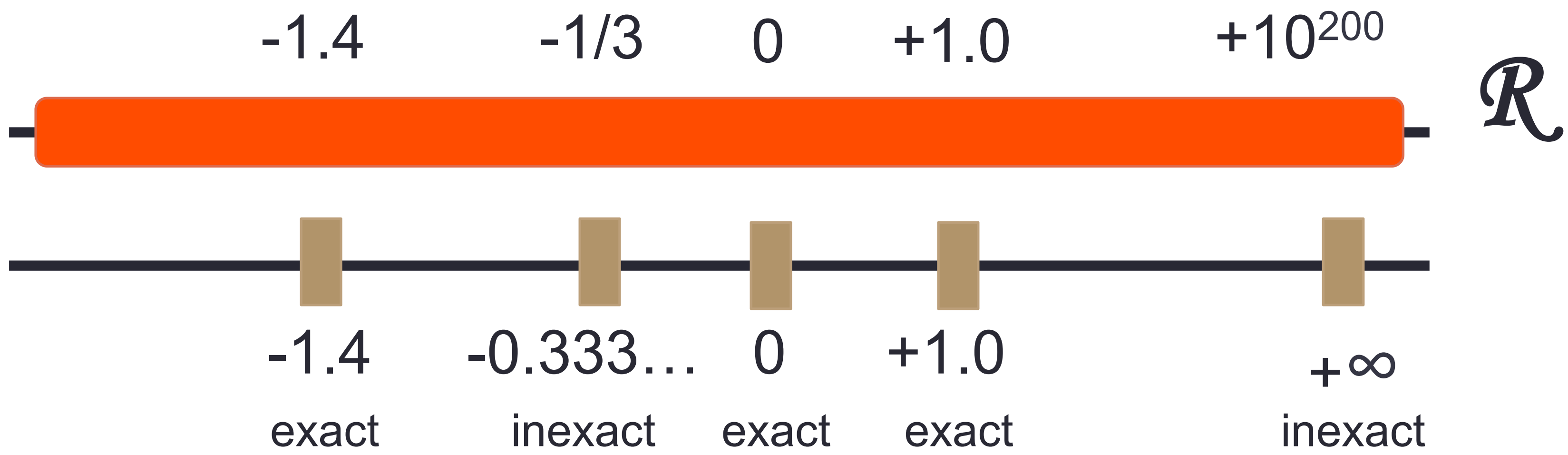
Hence $0.6875_{10} = 0.10110_2$

Notes on representation of real numbers

- What is the binary representation of 0.1_{10}?
  - $0.1 \times 2 = 0.2$
  - $0.2 \times 2 = 0.4$
  - $0.4 \times 2 = 0.8$
  - $0.8 \times 2 = 1.6$
  - $0.6 \times 2 = 1.2$
  - $0.2 \times 2 = 0.4$
  - $0.4 \times 2 = 0.8$
  - … (forever)
- Hence $0.1_{10} = 0.0001100110011\ldots$
- There is no exact representation for 0.1_{10}

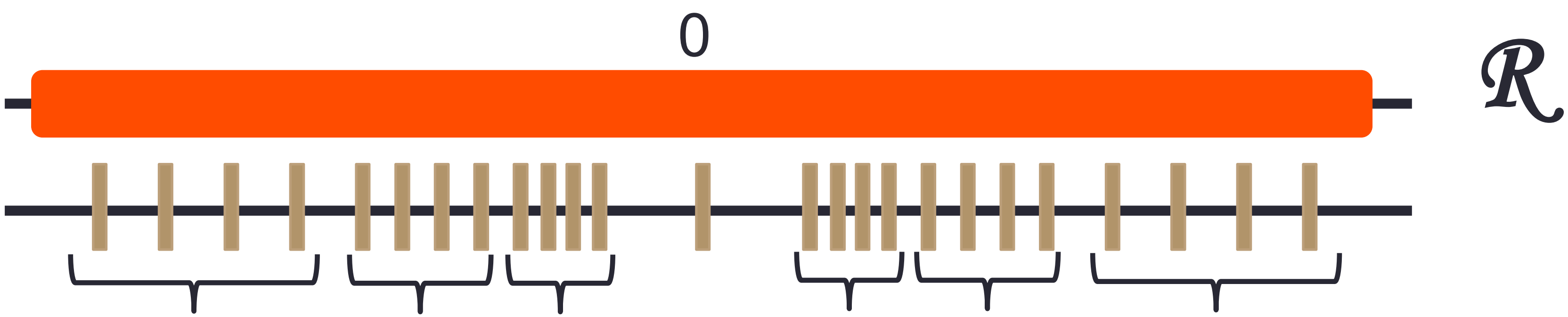
Notes on representation of real numbers

- It is interesting to widen both
  - The representation range and
  - The amount of represented values (density)
- Both aspects depend on sizes of mantissa and exponent



Notes on representation of real numbers

- For a given exponent, consecutive mantissas are separated by the same amount
- The larger the exponent, the larger the distance between two consecutive representable values
  - Relative error may be short, but the absolute error grows with the represented value
    - Don't use floating point for currency or time!



Each group of values have the same exponent but different mantissa

The IEEE 754 standard

- Scope:
  - Encoding: how to represent real numbers in several formats (single, double, and extended precision) and how to deal with particular cases (±∞, 0, NaN – *Not a Number*)
  - Operating modes: eg. the default rounding method
  - A set of mathematical operations that must be provided either by hardware operators or by software emulation
  - Exceptions: conditions that must be solved by the implementation
    - Namely, over/underflow, division by zero, inexact, invalid (eg. 0.0/0.0)

The IEEE 754 standard

- Encoding: representation of real numbers
  - Several formats based on number of bits. For 32 and 64 bits:
    - Single precision

1	8	23
S	E	M

$(-1)^S \cdot 1.M \cdot 2^{E-127}$

    - Double precision

1	11	52
S	E	M

$(-1)^S \cdot 1.M \cdot 2^{E-1023}$

- Denormalized values (values close to zero)

S	000..00	M ≠ 0

$(-1)^S \cdot 0.M \cdot 2^{-126}$

$(-1)^S \cdot 0.M \cdot 2^{-1022}$

- Special values

±0

S	000...00	000...00

±∞

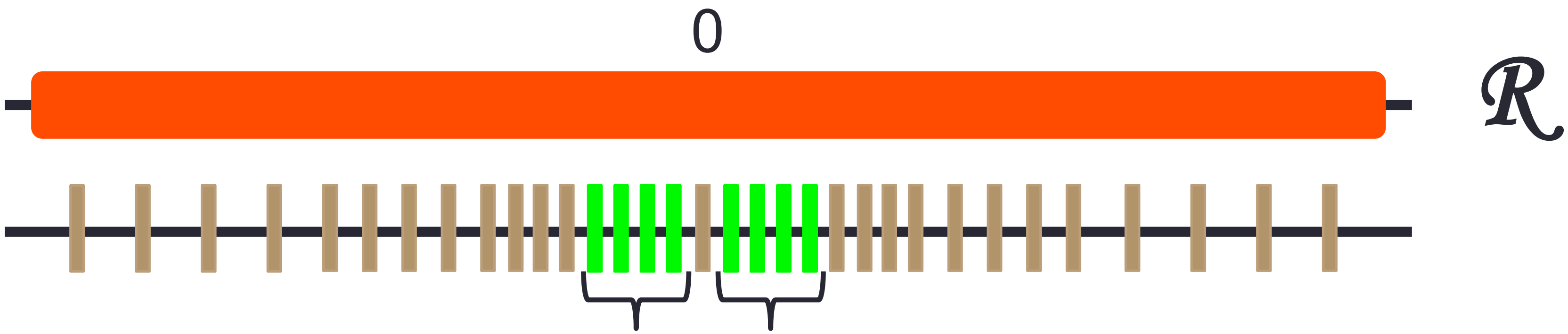
S	111...11	000...00

NaN

x	111...11	M ≠ 0

The IEEE 754 standard

- Near-zero values
  - Beyond the mantissa's resolution (~number of bits), IEEE 754 reserves a set of values for extremely small numbers (near zero)
  - These near-zero values follow a particular, different representation convention
  - They are known as denormalized values
    - Not all FP units support denormalized values



Special, near-zero, denormalized values
(all use the same exponent)

The IEEE 754 standard

- Special values
  - They are taken as operands just like other values
  - Zero and infinity
    - Taken as mathematical limits, hence

      $+\infty + +\infty = +\infty$; $-\infty + -\infty = -\infty$; etc.

      $+\infty \times$ *positive* $= +\infty$; $+\infty \times$ *negative* $= -\infty$; etc.

      positive / $+0 = +\infty$; *positive* / $-0 = -\infty$; etc.
    - For comparison, $+0 = -0$
  - Not a Number (NaN)
    - Generation: NaN is the result of $(+\infty) + (-\infty)$, $\pm 0 \times \pm\infty$, $\pm 0 / \pm 0$, $\pm\infty / \pm\infty$ and others
    - Propagation: if a NaN is an operand, the result becomes NaN
    - Comparing a NaN with any other number is always FALSE

The IEEE 754 standard

- The standard and programming languages
  - The set of special values enable mathematically coherent results

Java

```java
float x = +0.0f;
float y = 1/x;
float z = Float.NEGATIVE_INFINITY;
float t = 1/z;
float u = x*z;
System.out.println("x = " + x);
System.out.println("1/x = " + y);
System.out.println("z = " + z);
System.out.println("1/z = " + t);
System.out.println("x * z = " + u);
```

```
x = 0.0
1/x = Infinity
z = -Infinity
1/z = -0.0
x * z = NaN
```

The IEEE 754 standard

- Rounding
  - It is not unusual that an operation generates a mantissa M (*p* bits) larger than the allocated space (*m* bits)
    - The m most significant bits of the mantissa are called retained bits
    - The other *p* – *m* bits are called guard bits
  - Cases
    - When M is exactly representable within the format: the (*p* – *m*) non-retained bits are all zero and can be eliminated: 010000 → 0100
    - When M is between two representable values M_- and M_+ ($M_- < M < M_+$): rounding is needed by selecting either M_- or M_+
  - Rounding modes
    - Directed towards +∞
    - Directed towards –∞
    - Directed towards 0
    - Rounding to the nearest (default mode)

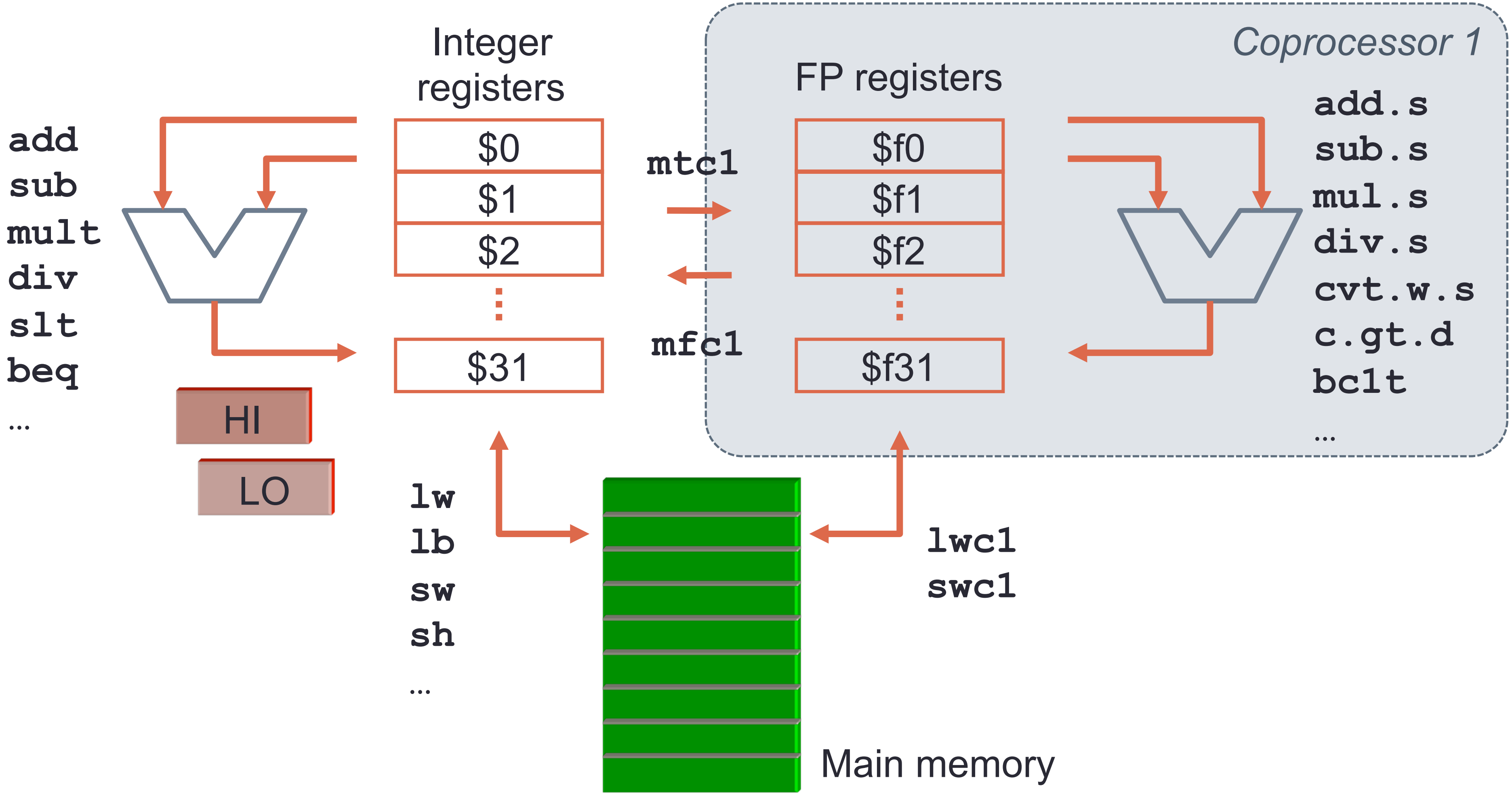
The IEEE 754 standard

- Round to nearest (*ties to even* – default option)
  - When M is exactly halfway between M_- and M_+ the even mantissa is selected
  - Examples:

M	M chosen	result M
010000	(exact)	0100
010001	M_-(nearest)	0100
010010	M_- (even)	0100
010011	M_+(nearest)	0101
010100	(exact)	0101
010101	M_-(nearest)	0101
010110	M_+ (even)	0110
010111	M_+(nearest)	0110
011000	(exact)	0110

FP in MIPS

Programmer's view



FP register file

- 32 registers named $f0, $f1, … $f31 for 32-bit float values
- They can be paired to hold a 64-bit value (double)
  - If $f0 contains a double, then $f0 holds the least significant half and $f1 the most significant part

Register use convention

Name of register	Conventional use
$f0	Function return (real part)
$f2	Function return (imaginary part)
$f4,$f6,$f8,$f10	Temporary registers
$f12,$f14	Function parameters
$f16,$f18	Temporary registers
$f20,$f22,$f24,$f26,$f28,$f30	Registers to preserve among function calls

Data transfer instructions

- Data interchange with memory and integer registers

operation	instruction
read: *\$ft ← Mem[X+\$rs]*	`lwc1 $ft,X($rs)`
write: *Mem[X+\$rs] ← \$ft*	`swc1 $ft,X($rs)`
transfer: *\$ft ← \$rs*	`mtc1 $rs,$ft`
transfer: *\$rt ← \$fs*	`mfc1 $rt,$fs`

fs, ft: FP registers

rs, rt: Int registers

```
        .data
x:      .float 3.1416
y:      .double 0.1
        .text
        la $t0,x
        lwc1 $f0,0($t0) # f0 <- x
        la $t0,y
        lwc1 $f2,0($t0)
        lwc1 $f3,4($t0) # f2 <- y
        mtc1 $0,$f4     # f4 <- 0.0
```

FP instructions do not handle immediate operands.
Constants must be allocated in memory or built into integer registers and then moved

Type conversion

- FP registers may contain
  - `s`: Single-precision FP values
  - `d`: Double-precision FP values
  - `w`: 32-bit integer values
- Type conversion is possible via `cvt._._ fd,fs`
  - Eg., `cvt.d.w $f4,$f7` converts the integer in f7 into a double in f4
- In combination with transfers to-from integer registers, values of different types can be used in arithmetic expressions

Basic arithmetic operations

- Each operation has S and D versions (single and double)
  - Eg., `add.s $f0,$f1,$f2` vs. `add.d $f0,$f2,$f4`

operation	**instruction**
addition	`add._ fd,fs,ft`
subtraction	`sub._ fd,fs,ft`
multiplication	`mul._ fd,fs,ft`
division	`div._ fd,fs,ft`
comparison	`c.`*`cond`*`._ fs,ft`
copy	`mov._ fd,fs`
sign change	`neg._ fd,fs`
absolute value	`abs._ fd,fs`

Immediate load pseudoinstructions

```
li.s $f0, 5.678
li.d $f4, 9.012
```

Comparison instructions

- Comparison instructions store their result in bit **FPc**
  - TRUE = 1; FALSE = 0
- FPc is kept in a control register of coprocessor 1 and is used by conditional branch instructions
- There is a set of comparison instructions for each data type
- Eg., `c._.s fd, fs` or `c._.d fd, fs`

fd>fs	**fd=fs**	**fd<fs**
`gt`	`eq`	`lt`
`le`	`neq`	`ge`
fd≤fs	**fd≠fs**	**fd≥fs**

Flow control

- Two conditional branch instructions:
  - **`bc1t`** *`label`* – if FPc = 1 then branch to *label*
  - **`bc1f`** *`label`* – if FPc = 0 then branch to *label*
- Combined with comparison instructions, they enable complex conditional branches
- Each condition accepts two implementations
  - SP example: ***if*** *($f0 > $f2)* ***then*** *branch to label*

```
; check wether $f0>$f2
      c.gt.s $f0,$f2
; branch if true
      bc1t label
```

```
; check wether $f0<=$f2
      c.le.s $f0,$f2
; branch if false
      bc1f label
```

3. Floating point operators

- Sign change operator
- Type conversion operators
- Multiply operator

Floating point operators

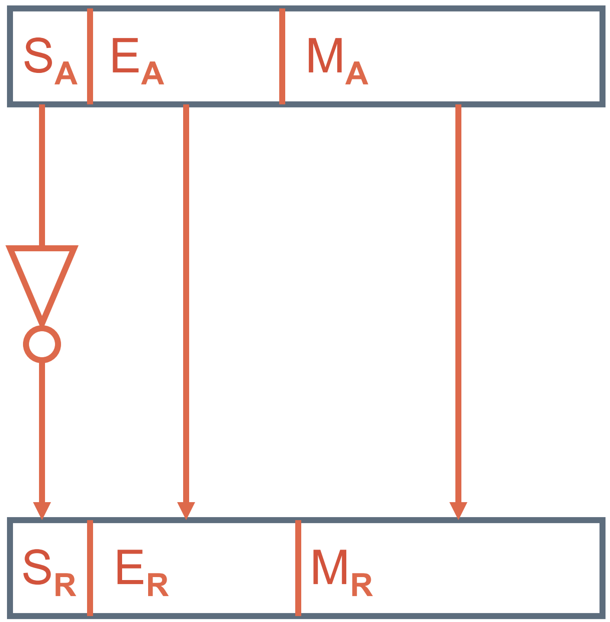
- FP operators take one or two arguments of a given FP format
- Their output is a standard FP value
  - With the exception of comparison operators
- They are relatively complex because they must:
  - Normalize result
  - Handle special values
  - If needed, round the result according to the current rounding mode
  - Raise the exceptions dictated by the standard
- We'll study the basic structure of some operators and how some of these details are solved

Floating point operators: roadmap

- NEG.S and NEG.D (sign change)
  - Structure
- CVT.D.S (conversion SP to DP)
  - Basic structure
  - Detail: dealing with special values
- CVT.S.D (conv. DP to SP)
  - Basic structure
  - Detail: rounding
- CVT.D.W (conv. integer to DP)
  - Basic structure
  - Detail: normalization
- MULT.S and MULT.D (multiplication)
  - Basic structure
  - Detail: renormalization

Sign change operator (neg)

- Specifications
  - Single precision:
    - Input: S_A (1 bit), E_A (8 bits), M_A (23 bits)
    - Output: S_R (1 bit), E_R (8 bits), M_R (23 bits)
  - Double precision:
    - Input: S_A (1 bit), E_A (11 bits), M_A (52 bits)
    - Output: S_R (1 bit), E_R (11 bits), M_R (52 bits)
- Operation
  - Change sign: S_R = not S_A
  - Copy exponent: $E_R = E_A$
  - Copy mantissa: $M_R = M_A$



Sign change: software emulation

```
float x = 1.0;

x = -x;
```

```
x:      .float 1.0

        lw $t0, x             # $t0 ← x (float 1.0)
        lui $t1, 0x8000       # $t1 ← 0x80000000
        xor $t0, $t0, $t1     # $t0 ← - 1.0
        sw $t0, x             # x ← $t0 (float -1.0)
```

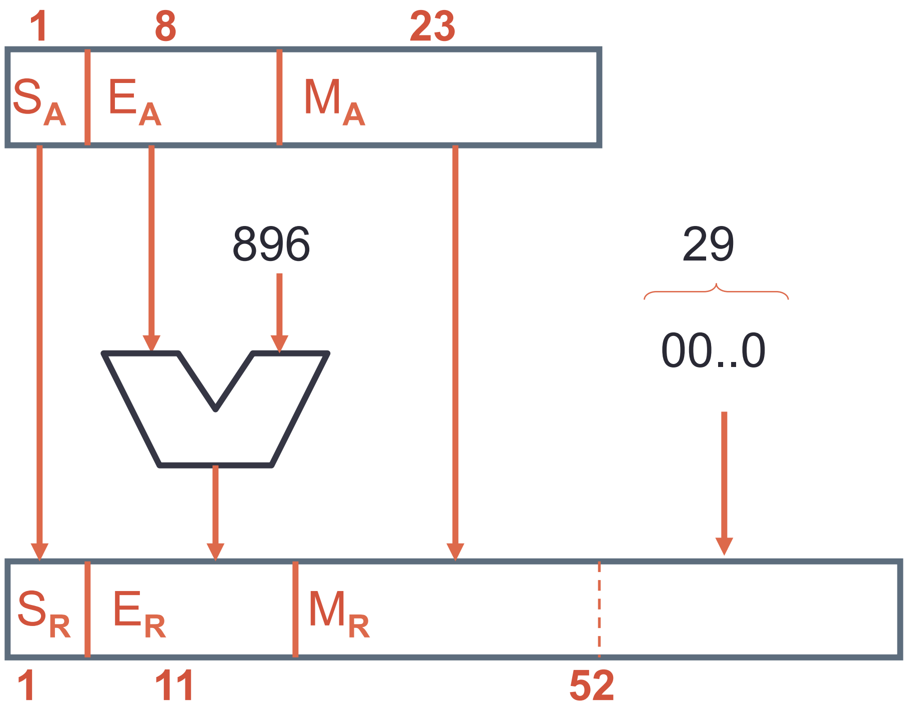
Single to double precision (cvt.d.s)

- Specification
  - Input: S_A (1 bit), E_A (8 bits), M_A (23 bits)
  - Output: S_R (1 bit), E_R (11 bits), M_R (52 bits)
- Operation
  - Sign doesn't change: $S_R = S_A$
  - Exponent: change excess 127 into excess 1023: $E_R = E_A + 896$
  - Mantissa: add 52 – 23 = 29 zeros to the right: $M_R = M_A \,||\, 00....0$
- Handling special values

	E_A	S_R	E_R	M_R
zero and denormalized:	00000000_2	S_A	00000000000_2	$M_A \,\|\|\, 00....0$
±∞ and NaN:	11111111_2	S_A	11111111111_2	$M_A \,\|\|\, 00....0$
Regular values:	others	S_A	$E_A + 896$	$M_A \,\|\|\, 00....0$

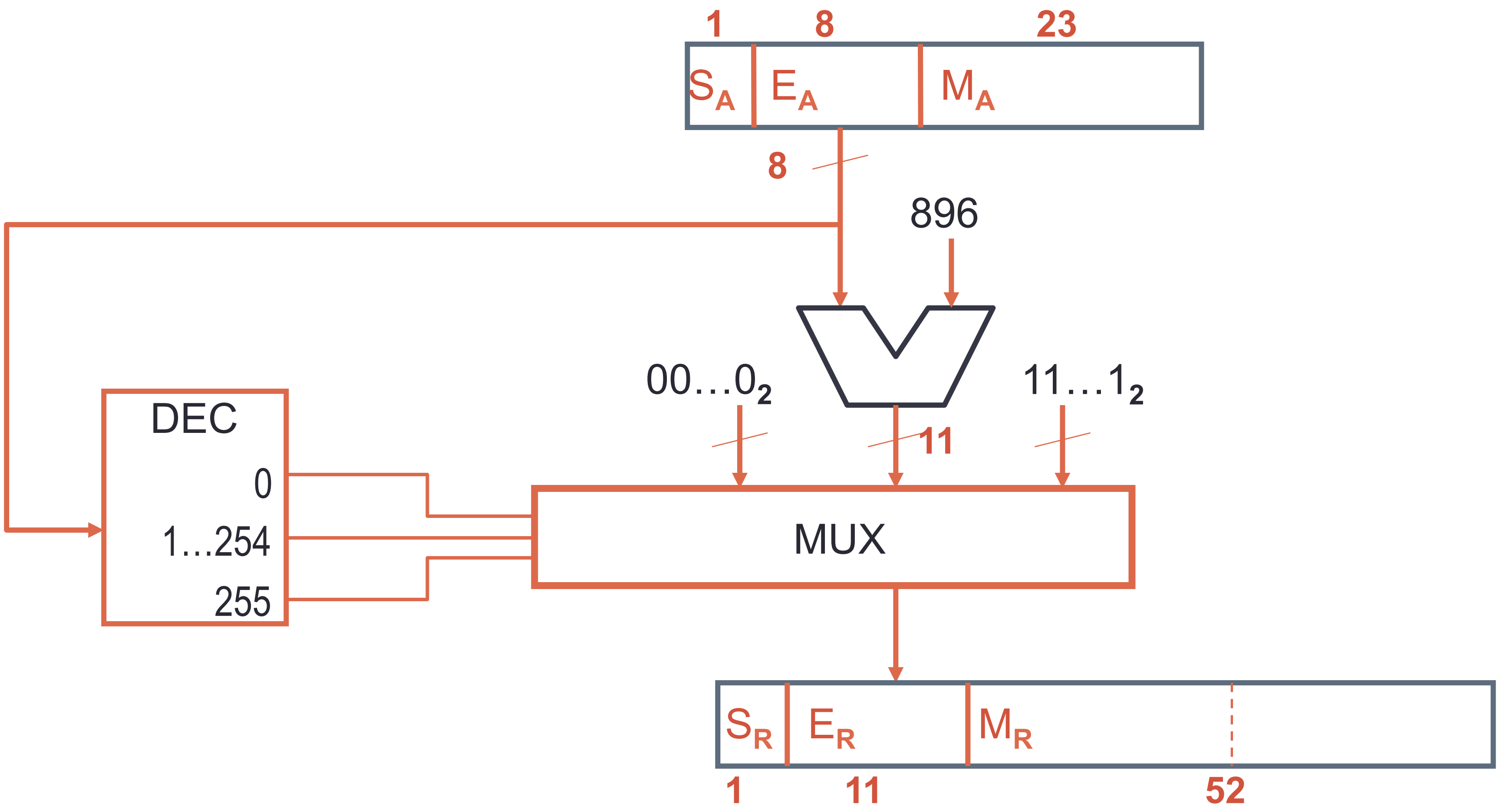
Single to double precision (cvt.d.s)

- Basic operator (no handling of special values)



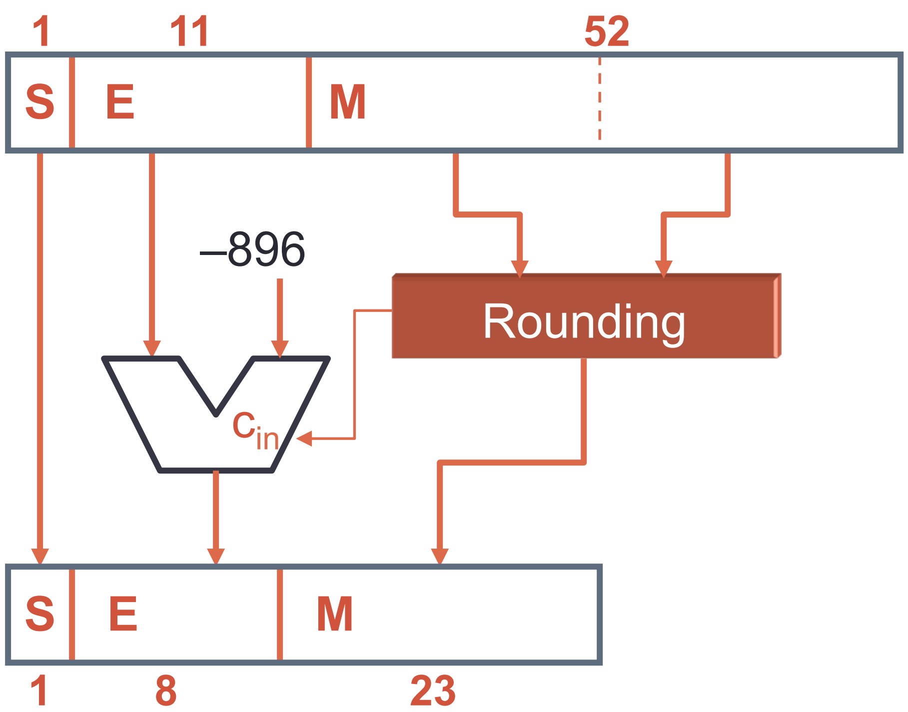
Single to double precision (cvt.d.s)

- Handling special values: obtaining the exponent



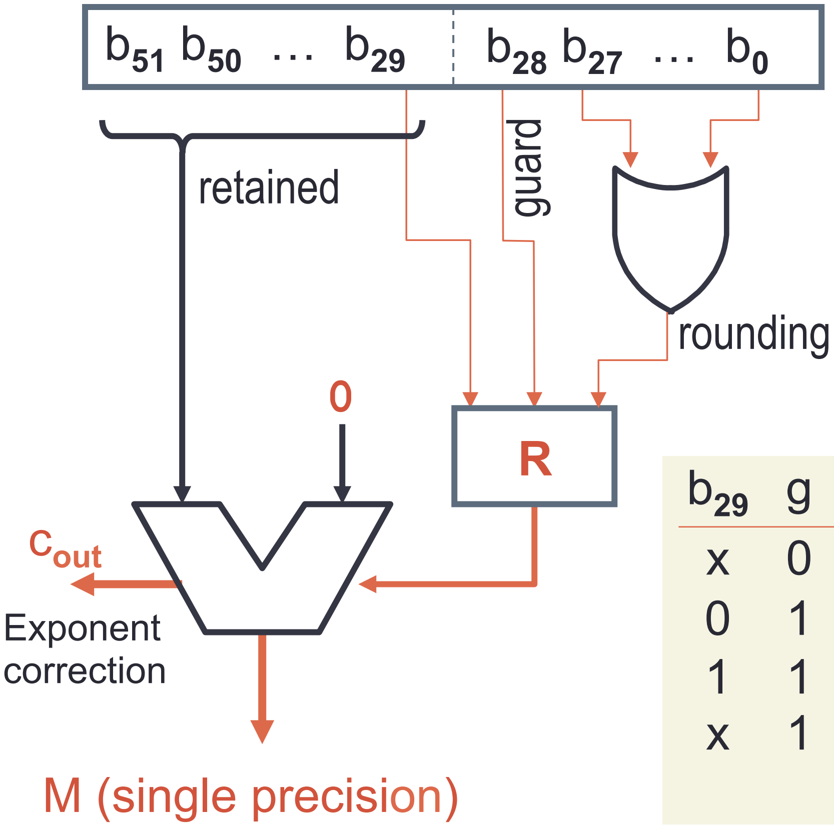
Double to single precision (cvt.s.d)

- Structure
  - Sign: does not change
  - Exponent: convert from 11-bit excess 1023 to 8-bit excess 127
    - Note that this conversion may overflow
  - Mantissa: remove 29 least significant bits, round the resulting M



Rounding details

- Rounding to nearest, ties to even



b_{29}	g	rnd.	R
x	0	x	0
0	1	0	0
1	1	0	1
x	1	1	1

If R=0, truncate:

```
  1.1001101
+         0
  1.10011
```

Implicit bit

If R=1, increment:

```
  1.1001111
+         1
  1.10100
```

If c_{out}=1, correct result exp.:

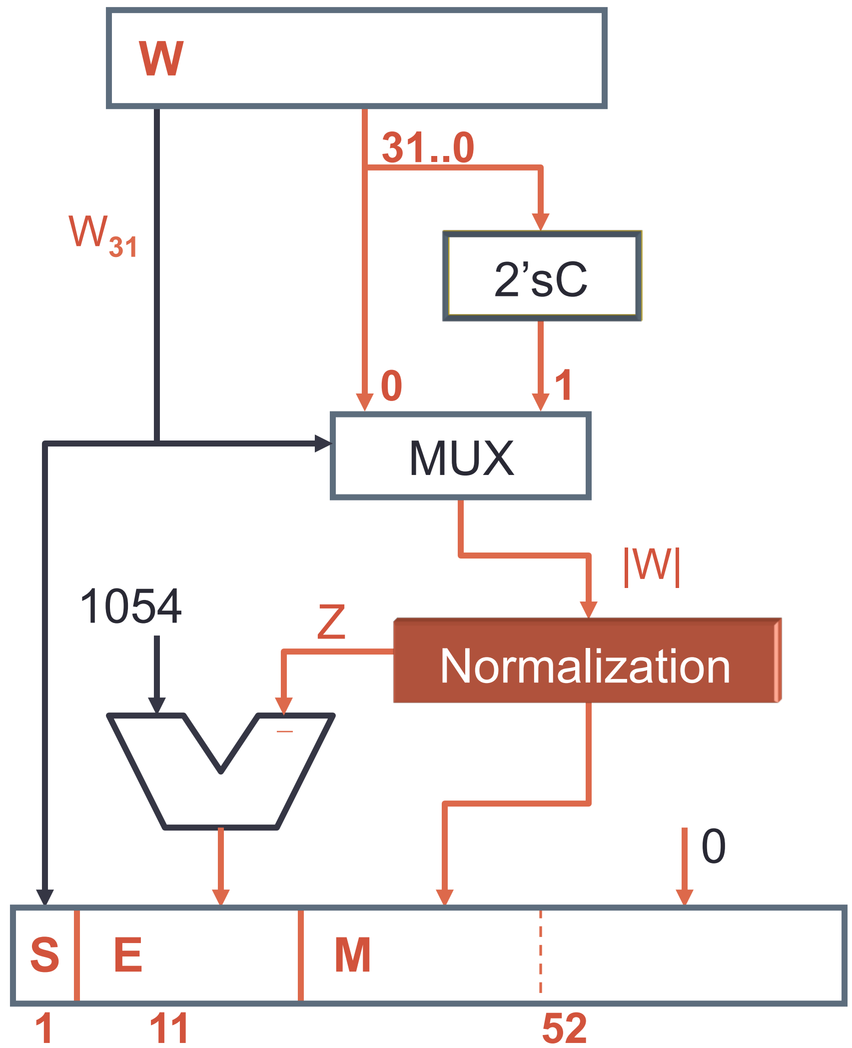
```
   1.1111111
+          1
  10.00000
   1.00000
```

Integer to double precision (cvt.d.w)

- Operating principles
  - A positive integer W can be rewritten as $+0.W \times 2^{32}$
  - A negative integer W can be rewritten as $-0.(-W) \times 2^{32}$
  - The mantissa W has Z leading zeros ($0 \leq Z \leq 32$)
    - Normalization requires a left shift of Z+1 positions and subtract Z+1 to the exponent
- Specification
  - Input: 32-bit integer W
  - Output: S_R (1 bit), E_R (11 bits), M_R (52 bits)
    - $S_R = \text{Sign}(W)$
    - $M_R = |W| << Z+1$
    - $E_R = 1023 + 32 - Z - 1 = 1054 - Z$

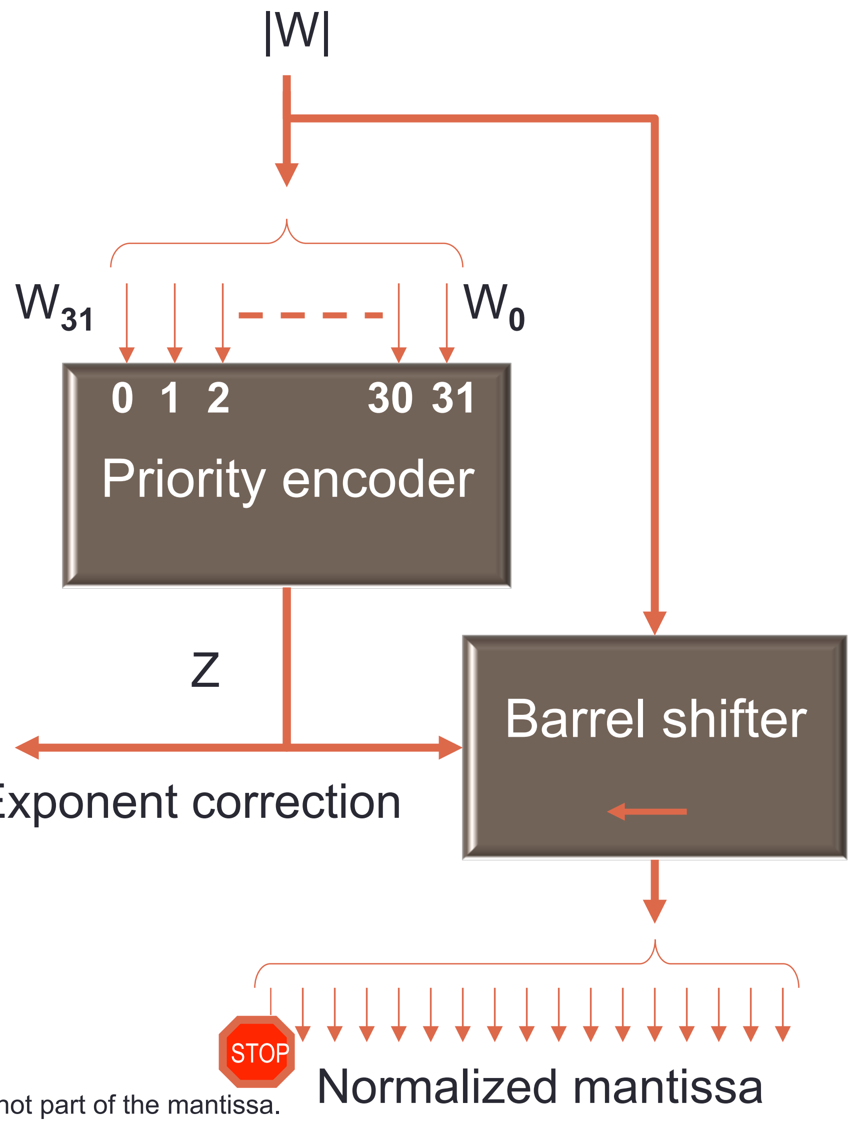
Integer to double precision (cvt.d.w)

- Structure
  - Normalization counts the number of leading zeros Z
  - |W| is shifted Z+1 positions to the left
    - The exponent is then adjusted to 31 – Z
    - Adding the excess, E = 1023 + 31 – Z
  - The mantissa is filled with zeros



Integer to double precision (cvt.d.w)

- A priority encoder encodes the position of the input's least-significant "1"
  - Hence we reverse |W|
- A barrel shifter shifts Z positions to the left
- The implicit bit is discarded



Priority encoder operation (reminder)

W_{31}	W_{30}	W_{29}	W_{28}	…	W_1	W_0	Z
1	X	X	X	…	X	X	00000
0	1	X	X	…	X	X	00001
0	0	1	X	…	X	X	00010
…	…	…	…	…	…	…	…
0	0	0	0	…	0	1	11111

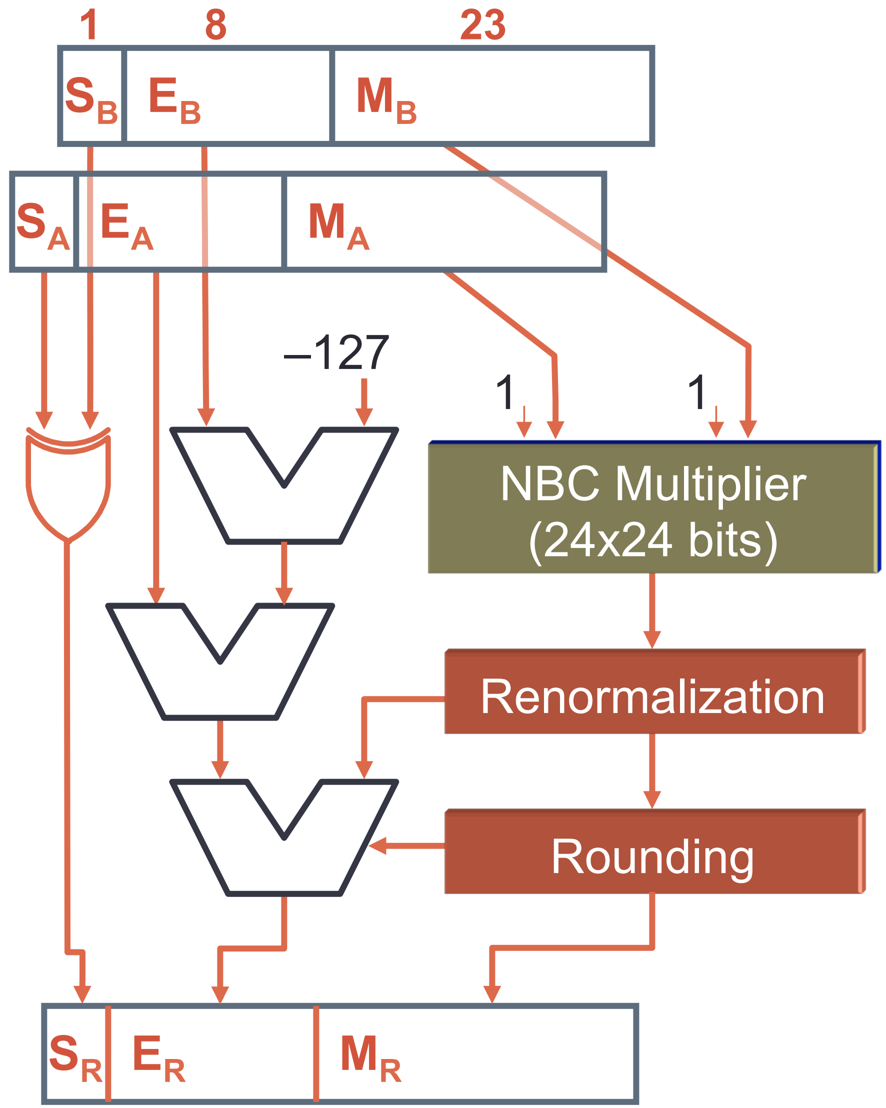
STOP: After left shift by Z positions, the MSB is the implicit bit, which is not part of the mantissa.

Multiplication (mul.s and mul.d)

- Specification
  - Inputs:
    - S_A (1 bit), E_A (8/11 bits), M_A (23/52 bits)
    - S_B (1 bit), E_B (8/11 bits), M_B (23/52 bits)
  - Output: S_R (1 bit), E_R (8/11 bits), M_R (23/52 bits)
  - Sign: $S_R = S_A$ **xor** S_B
  - Exponent: add and compensate the excess
    - SP: $E_R = E_A + E_B - 127$
    - DP: $E_R = E_A + E_B - 1023$
  - Mantissa:
    - Multiply $1.M_A \times 1.M_B$ (consider implicit bit)
      - Size of the multiplier depends on precision: 24 or 53 bit multiplier
    - The result needs to be renormalized (remove implicit bit) and rounded

Multiplication (mul.s)

- Operator (SP)
  - Two adders for the exponent, with excess correction
  - NBC multiplier including implicit bit
  - Removal of implicit bit (renormalization – see later)
    - Perhaps, decrement exponent
  - Rounding to 23 bits



Renormalization after multiplication

- If the product mantissa has a leading “0”:
  - Shift left one position
- If the product mantissa has a leading “1”:
  - Increment exponent by 1
- In all cases
  - Remove the leading implicit bit