Floating-point Numbers

Representing Fractional Numbers

- A binary number with fractional part

  $B = b_{n-1}\, b_{n-2} \,.....b_1\, b_0 \,.\, b_{-1}\, b_{-2} \,.....\, b_{-m}$

  corresponds to the decimal number

  $$D = \sum_{i=-m}^{n-1} b_i\, 2^i$$

- Also called *fixed-point numbers*.
  - The position of the radix point is fixed.

If the radix point is allowed to move, we call it a floating-point representation.

Some Examples

1011.1 ➔ $1x2^3 + 0x2^2 + 1x2^1 + 1x2^0 + 1x2^{-1}$ = 11.5

101.11 ➔ $1x2^2 + 0x2^1 + 1x2^0 + 1x2^{-1} + 1x2^{-2}$ = 5.75

10.111 ➔ $1x2^1 + 0x2^0 + 1x2^{-1} + 1x2^{-2} + 1x2^{-3}$ = 2.875

Some Observations:

- Shift right by 1 bit means divide by 2
- Shift left by 1 bit means multiply by 2
- Numbers of the form $0.111111..._2$ has a value less than 1.0 (one).

Limitations of Representation

- In the fractional part, we can only represent numbers of the form $x/2^k$ exactly.
  - Other numbers have repeating bit representations (i.e. never converge).
- Examples:

3/4 = 0.11

7/8 = 0.111

5/8 = 0.101

1/3 = 0.10101010101 [01] ….

1/5 = 0.001100110011 [0011] ….

1/10 = 0.0001100110011 [0011] ….

- More the number of bits, more accurate is the representation.
- We sometimes see: (1/3)*3 ≠ 1.

Floating-point Number Representation (IEEE-754)

- For representing numbers with fractional parts, we can assume that the fractional point is somewhere in between the number (say, *n* bits in integer part, *m* bits in fraction part). → *Fixed-point representation*
  - Lacks flexibility.
  - Cannot be used to represent very small or very large numbers (for example: 2.53×10^{-26}, $1.7562 \times 10^{+35}$, etc.).
- **Solution** :: use *floating-point number representation*.
  - A number *F* is represented as a triplet *<s, M, E>* such that

    $F = (-1)^s M \times 2^E$

$F = (-1)^s\, M \times 2^E$

- s is the *sign bit* indicating whether the number is negative (=1) or positive (=0).
- M is called the *mantissa*, and is normally a fraction in the range [1.0,2.0].
- E is called the *exponent*, which weights the number by power of 2.

Encoding:

- Single-precision numbers: total 32 bits, E 8 bits, M 23 bits
- Double-precision numbers: total 64 bits, E 11 bits, M 52 bits

Points to Note

- The number of *significant digits* depends on the number of bits in *M*.
  - 7 significant digits for 24-bit mantissa (23 bits + 1 implied bit).
- The *range* of the number depends on the number of bits in *E*.
  - 10^{38} to 10^{-38} for 8-bit exponent.

How many significant digits?

$2^{24} = 10^x$

$24 \log_{10}2 = x \log_{10}10$

$x = 7.2 \rightarrow$ 7 significant decimal places

Range of exponent?

$2^{127} = 10^y$

$127 \log_{10}2 = y \log_{10}10$

$y = 38.1 \rightarrow$ maximum exponent value 38 (in decimal)

"Normalized" Representation

- We shall now see how *E* and *M* are actually encoded.
- Assume that the actual exponent of the number is *EXP* (i.e. number is $M \times 2^{EXP}$).
- Permissible range of *E*: $1 \le E \le 254$ (the all-0 and all-1 patterns are not allowed).
- **Encoding of the exponent E**:

  The exponent is encoded as a biased value: $E = EXP + BIAS$

  where BIAS = 127 ($2^{8-1} - 1$) for single-precision, and

  BIAS = 1023 ($2^{11-1} - 1$) for double-precision.

- **Encoding of the mantissa M**:
  - The mantissa is coded with an implied leading 1 (i.e. in 24 bits).

    M = 1 . xxxx...x
  - Here, xxxx…x denotes the bits that are actually stored for the mantissa. We get the extra leading bit for *free*.
  - When xxxx…x = 0000…0, *M* is minimum (= 1.0).
  - When xxxx…x = 1111…1, *M* is maximum (= 2.0 – ε).

An Encoding Example

- Consider the number F = 15335

  $15335_{10} = 11101111100111_2 = 1.1101111100111 \times 2^{13}$

- Mantissa will be stored as: $M = 1101111100111\ 0000000000_2$

- Here, EXP = 13, BIAS = 127. ➔ $E = 13 + 127 = 140 = 10001100_2$

0	10001100	11011111001110000000000

466F9C00 in hex

Another Encoding Example

- Consider the number F = -3.75

  $-3.75_{10} = -11.11_2 = -1.111 \times 2^1$

- Mantissa will be stored as: $M = 11100000000000000000000_2$

- Here, EXP = 1, BIAS = 127. ➔ $E = 1 + 127 = 128 = 10000000_2$

1	10000000	11100000000000000000000

40700000 in hex

Special Values

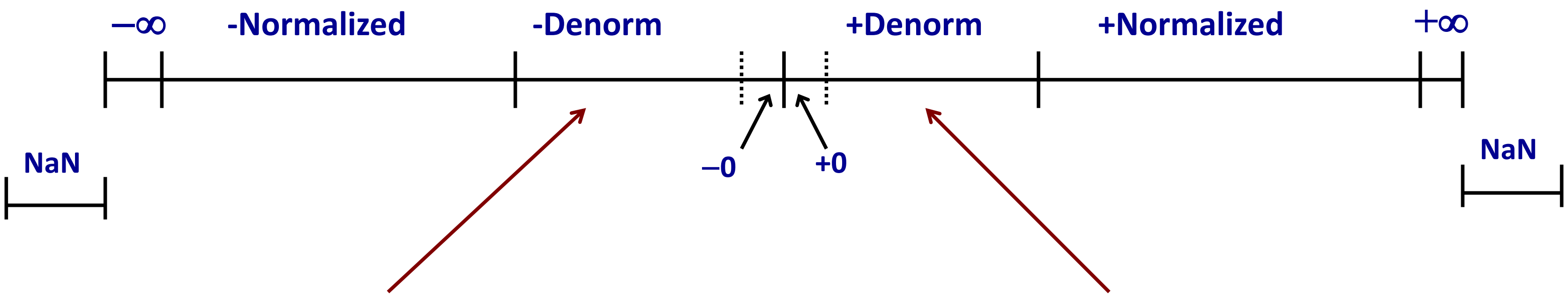
- **When E = 000…0**
  - M = 000…0 represents the value 0.
  - M ≠ 000…0 represents numbers very close to 0.
- **When E = 111…1**
  - M = 000…0 represents the value ∞ (infinity).
  - M ≠ 000…0 represents *Not-a-Number* (NaN).

Zero is represented by the *all-zero string*.

Also referred to as *de-normalized* numbers.

NaN represents cases when no numeric value can be determined, like uninitialized values, ∞*0, ∞-∞, square root of a negative number, etc.

Summary of Number Encodings



Denormal numbers have very small magnitudes (close to 0) such that trying to normalize them will lead to an exponent that is below the minimum possible value.

- Mantissa with leading 0’s and exponent field equal to zero.
- Number of significant digits gets reduced in the process.

Rounding

- Suppose we are adding two numbers (say, in single-precision).
  - We add the mantissa values after shifting one of them right for exponent alignment.
  - We take the first 23 bits of the sum, and discard the residue *R* (remaining bits).
- IEEE-754 format supports four rounding modes:
  a) Truncation
  b) Round to $+\infty$ (similar to ceiling function)
  c) Round to $-\infty$ (similar to floor function)
  d) Round to nearest

- To implement rounding, two temporary bits are maintained:
  - *Round Bit (r)*: This is equal to the MSB of the residue *R*.
  - *Sticky Bit (s)*: This the logical OR of the rest of the bits of the residue *R*.
- Decisions regarding rounding can be taken based on these bits:
  a) $R > 0$: if $r + s = 1$
  b) $R = 0.5$: if $r.s' = 1$
  c) $R > 0.5$: if $r.s = 1$ // '+' is logical OR, '.' is logical AND
- Renormalization after Rounding:
  - If the process of rounding generates a result that is not in normalized form, then we need to re-normalize the result.

Some Exercises

Decode the following single-precision floating-point numbers.

a) 0011 1111 1000 0000 0000 0000 0000 0000
b) 0100 0000 0110 0000 0000 0000 0000 0000
c) 0100 1111 1101 0000 0000 0000 0000 0000
d) 1000 0000 0000 0000 0000 0000 0000 0000
e) 0111 1111 1000 0000 0000 0000 0000 0000
f) 0111 1111 1101 0101 0101 0101 0101 0101

100

Floating-point Arithmetic

Floating Point Addition/Subtraction

- Two numbers: $M1 \times 2^{E1}$ and $M2 \times 2^{E2}$, where $E1 > E2$ (say).
- Basic steps:
  - Select the number with the smaller exponent (i.e. $E2$) and shift its mantissa right by ($E1$-$E2$) positions.
  - Set the exponent of the result equal to the larger exponent (i.e. $E1$).
  - Carry out $M1 \pm M2$, and determine the sign of the result.
  - Normalize the resulting value, if necessary.

102

Addition Example

- Suppose we want to add F1 = 270.75 and F2 = 2.375

  $F1 = (270.75)_{10} = (100001110.11)_2 = 1.0000111011 \times 2^8$

  $F2 = (2.375)_{10} = (10.011)_2 = 1.0011 \times 2^1$

- Shift the mantissa of F2 right by 8 – 1 = 7 positions, and add:

```
1000 0111 0110 0000 0000 0000
           1 0011 0000 0000 0000 0000 000
---------------------------------------------
1000 1000 1001 0000 0000 0000 0000 000
```

- **Result**: 1.00010001001×2^8

Residue

Subtraction Example

- Suppose we want to subtract F2 = 224 from F1 = 270.75

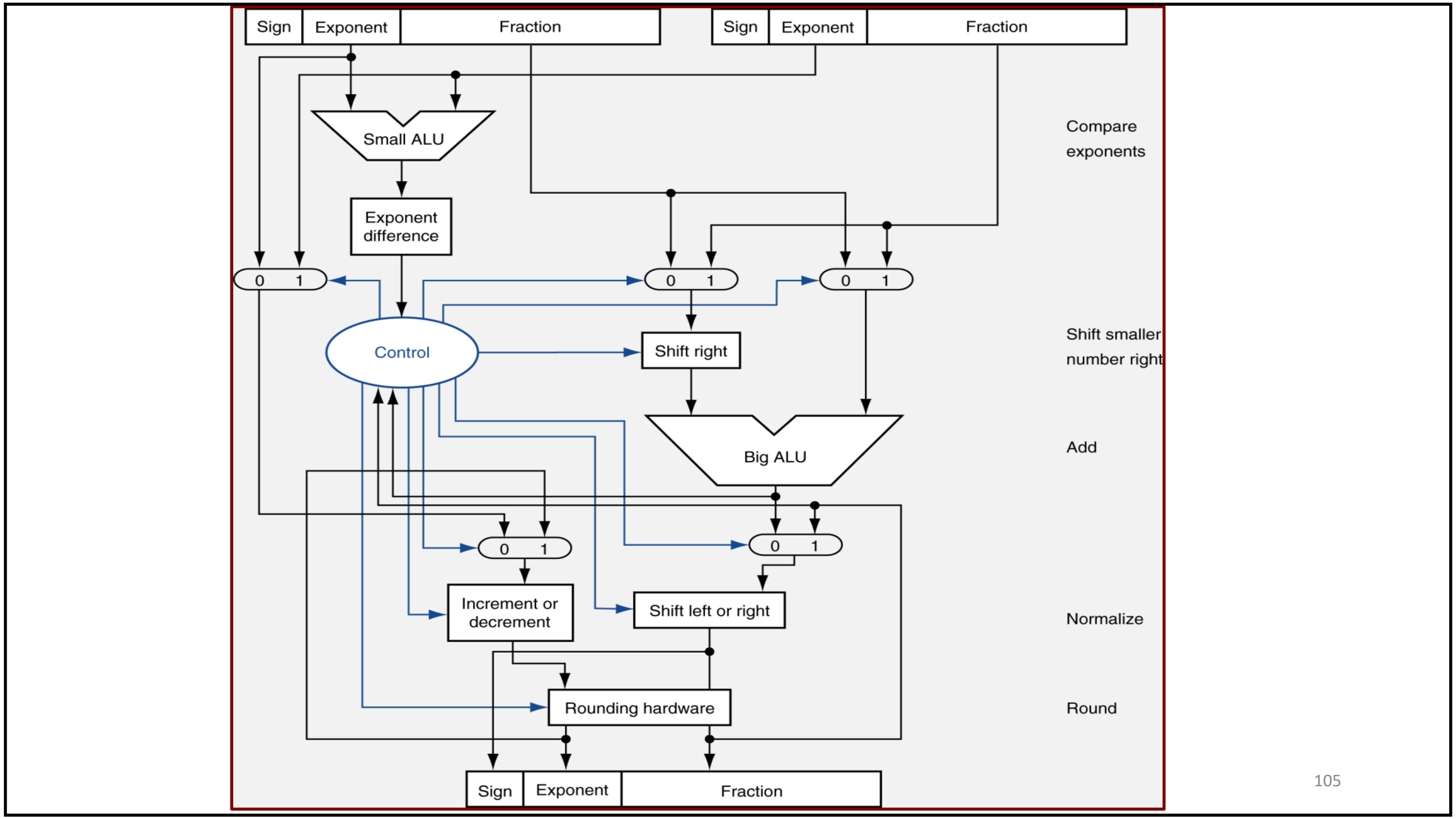
  $F1 = (270.75)_{10} = (100001110.11)_2 = 1.0000111011 \times 2^8$

  $F2 = (224)_{10} = (11100000)_2 = 1.111 \times 2^7$

- Shift the mantissa of F2 right by 8 – 7 = 1 position, and subtract:

```
  1000 0111 0110 0000 0000 0000
   111 0000 0000 0000 0000 0000 000
--------------------------------------
 0001 0111 0110 0000 0000 0000 000
```

- For normalization, shift mantissa left 3 positions, and decrement E by 3.
- **Result**: 1.01110110×2^5


Sign
Exponent
Fraction
Sign
Exponent
Fraction
Small ALU
Exponent difference
0 1
0 1
0 1
Control
Shift right
Big ALU
0 1
0 1
Increment or decrement
Shift left or right
Rounding hardware
Sign
Exponent
Fraction
Compare exponents
Shift smaller number right
Add
Normalize
Round
105

Floating-point Multiplication

- Two numbers: $M1 \times 2^{E1}$ and $M2 \times 2^{E2}$
- Basic steps:
  - Add the exponents *E*1 and *E*2 and subtract the *BIAS*.
  - Multiply *M*1 and *M*2 and determine the *sign* of the result.
  - Normalize the resulting value, if necessary.

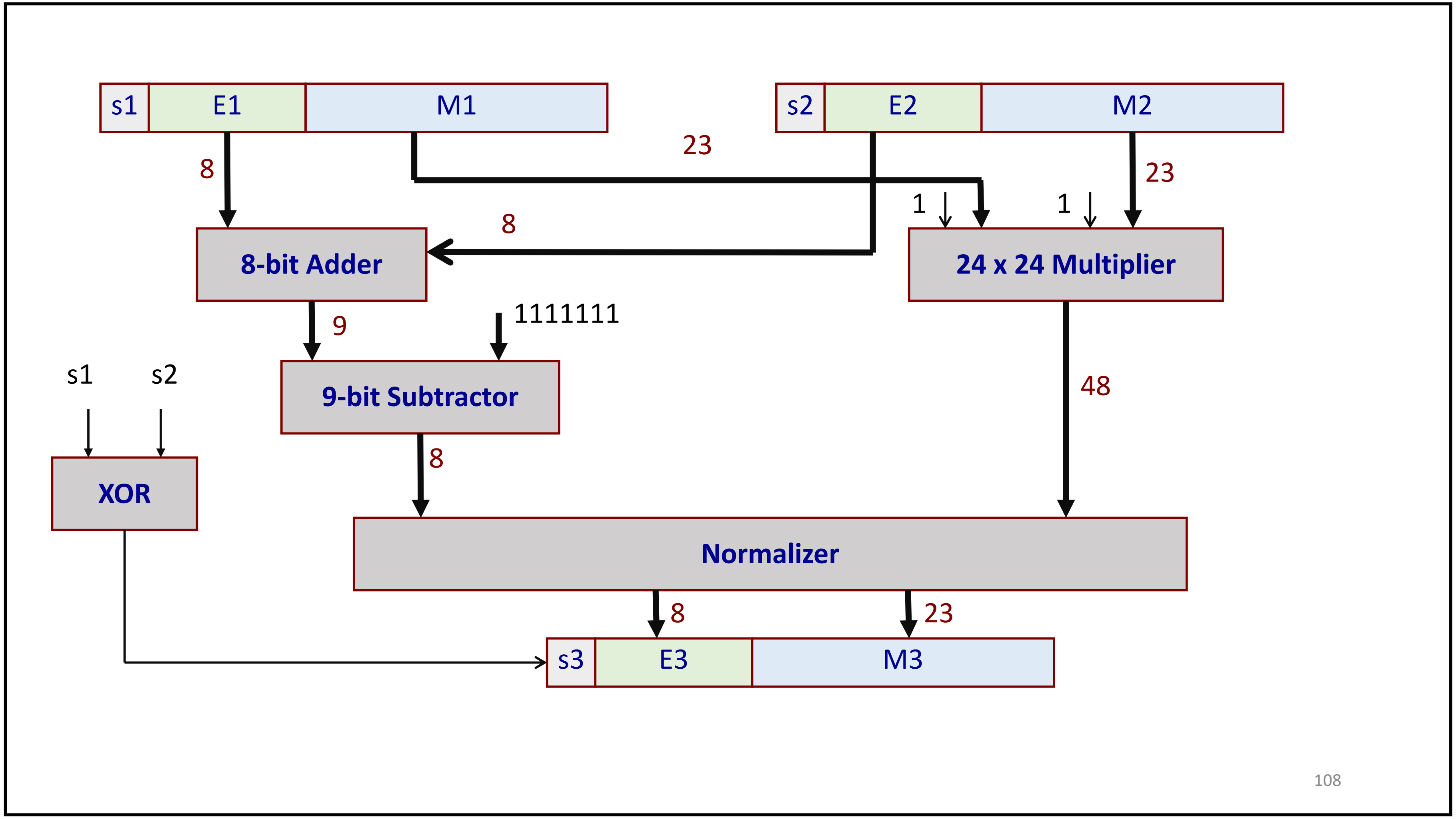
Multiplication Example

- Suppose we want to multiply F1 = 270.75 and F2 = -2.375

  F1 = $(270.75)_{10}$ = $(100001110.11)_2$ = 1.0000111011×2^8

  F2 = $(-2.375)_{10}$ = $(-10.011)_2$ = -1.0011×2^1

- Add the exponents: 8 + 1 = 9
- Multiply the mantissas: 1.01000001100001
- **Result**: $- 1.01000001100001 \times 2^9$


s1
E1
M1
s2
E2
M2
23
8
23
1
1
8
8-bit Adder
24 x 24 Multiplier
9
1111111
s1
s2
9-bit Subtractor
48
XOR
8
Normalizer
8
23
s3
E3
M3
108

108

Floating-point Division

- Two numbers: $M1 \times 2^{E1}$ and $M2 \times 2^{E2}$
- Basic steps:
  - Subtract the exponents *E1* and *E2* and add the *BIAS*.
  - Divide *M1* by *M2* and determine the *sign* of the result.
  - Normalize the resulting value, if necessary.

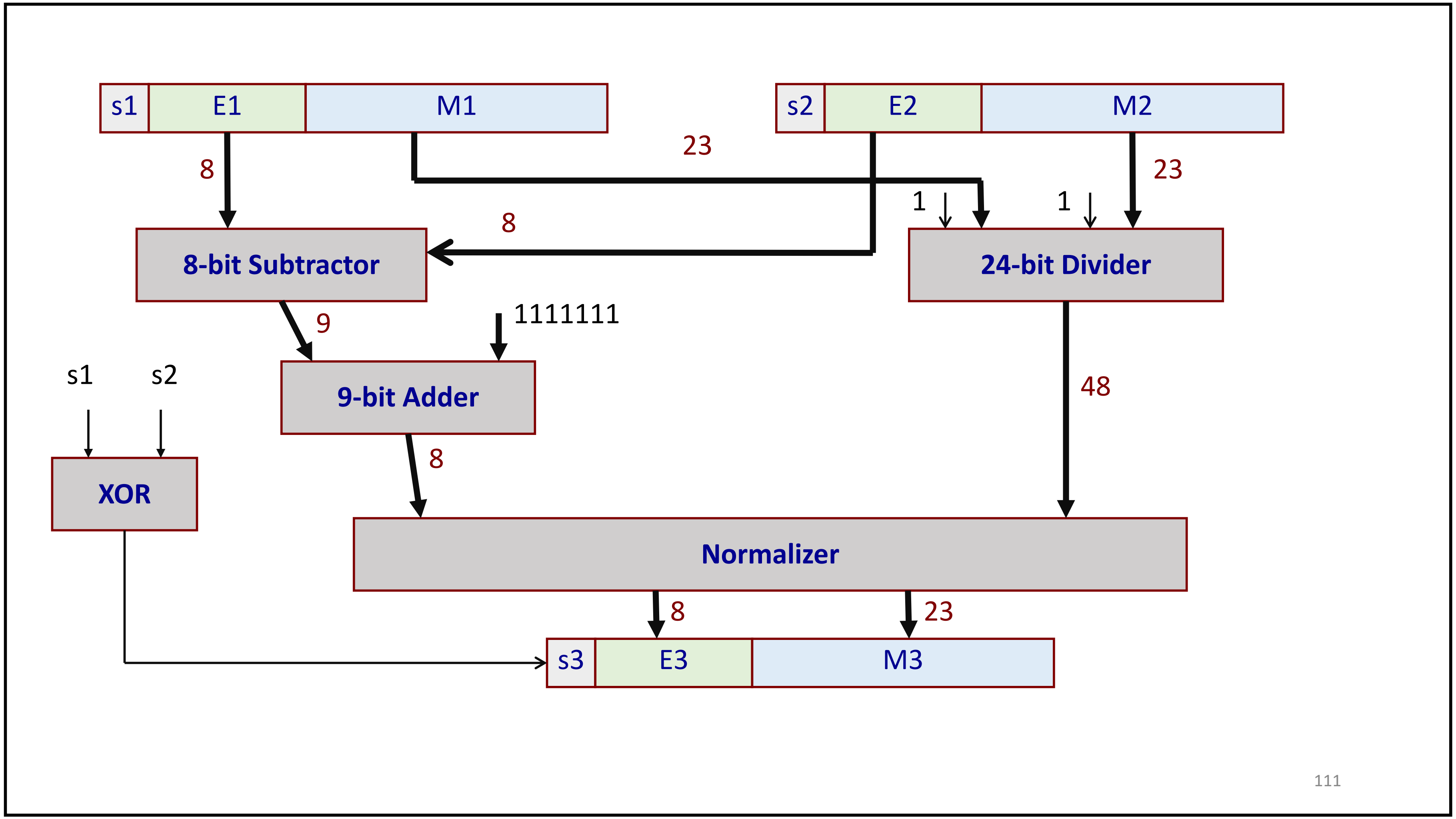
Division Example

- Suppose we want to divide F1 = 270.75 by F2 = -2.375

  $F1 = (270.75)_{10} = (100001110.11)_2 = 1.0000111011 \times 2^8$

  $F2 = (-2.375)_{10} = (-10.011)_2 = -1.0011 \times 2^1$

- Subtract the exponents: 8 – 1 = 7
- Divide the mantissas: 0.1110010
- **Result**: -0.1110010×2^7
- **After normalization**: -1.110010×2^6

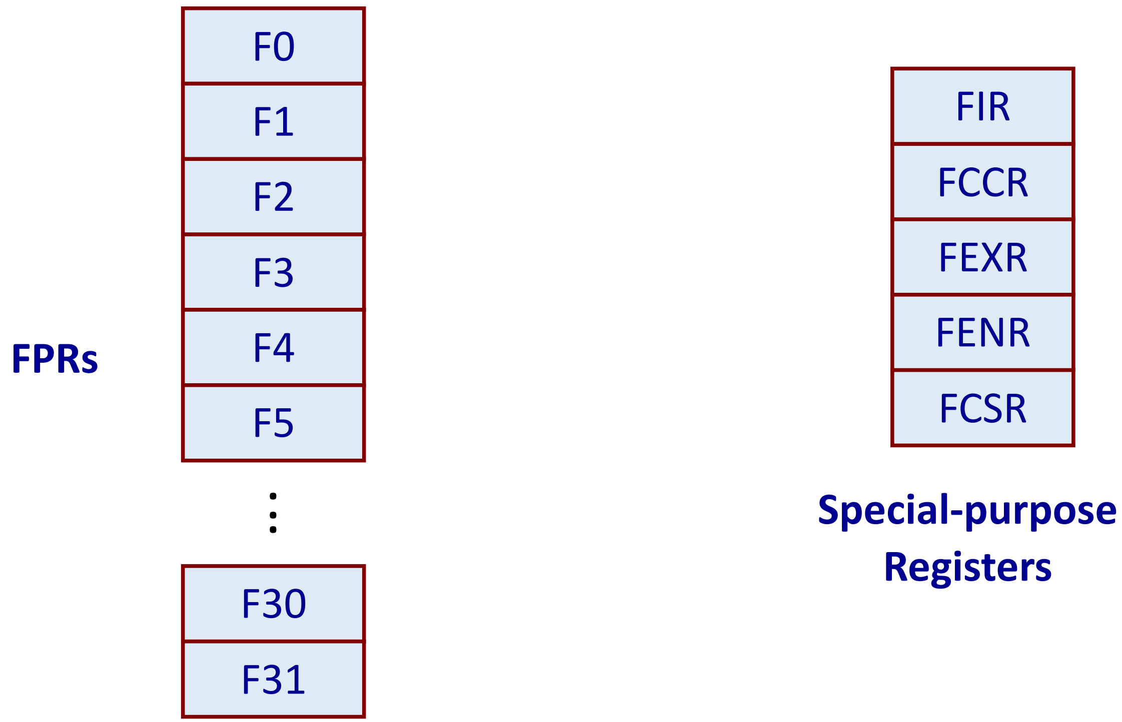

s1
E1
M1
s2
E2
M2
23
8
23
1
1
8
8-bit Subtractor
24-bit Divider
9
1111111
s1
s2
9-bit Adder
48
XOR
8
Normalizer
8
23
s3
E3
M3
111

FLOATING-POINT ARITHMETIC in MIPS32

112

112

- The MIPS32 architecture defines the following floating-point registers (FPRs).
  - 32 32-bit floating-point registers *F0* to *F31*, each of which is capable of storing a single-precision floating-point number.
  - Double-precision floating-point numbers can be stored in even-odd pairs of FPRs (e.g., (*F0,F1*), (*F10,F11*), etc.).
- In addition, there are five *special-purpose FPU control registers*.


FPRs
F0
F1
F2
F3
F4
F5
F30
F31
FIR
FCCR
FEXR
FENR
FCSR
Special-purpose
Registers
114

114

Typical Floating Point Instructions in MIPS32

- Load and Store instructions
  - Load Word from memory
  - Load Double-word from memory
  - Store Word to memory
  - Store Double-word to memory
- Data Movement instructions
  - Move data between integer registers and floating-point registers
  - Move data between integer registers and floating-point control registers

- Arithmetic instructions
  - Floating-point absolute value
  - Floating-point compare
  - Floating-point negate
  - Floating-point add
  - Floating-point subtract
  - Floating-point multiply
  - Floating-point divide
  - Floating-point square root
  - Floating-point multiply add
  - Floating-point multiply subtract

116

- Rounding instructions:
  - Floating-point truncate
  - Floating-point ceiling
  - Floating-point floor
  - Floating-point round
- Format conversions:
  - Single-precision to double-precision
  - Double-precision to single-precision

Example: Add a scalar s to a vector A

```
for (i=1000; i>0; i--)
  A[i]= A[i] + s;
```

```
Loop:  L.D     F0,0(R1)
       ADD.D   F4,F0,F2
       S.D     F4,0(R1)
       ADDI    R1,R1,-8
       BNE     R1,R2,Loop
```

R1: initially qoints to *A[1000]*

(*F2,F3*): contains the scalar *s*

R2: initialized such that 8(R2) is the *address of A[1]*

We assume double precision (64 bits):

- Numbers stored in (*F0,F1*), (*F2,F3*), and (*F4,F5*).