Computer Organization and Architecture

Module 5

Prof. Indranil Sengupta

Dr. Sarani Bhattacharya

Department of Computer Science and Engineering

IIT Kharagpur

Division

Introduction

- Division is more complex than multiplication.
- *<u>Example</u>*: Typical values in Pentium-3 processor →
    - Not easy to construct high-speed dividers.
- The ratios have not changed much in later processors.

Instruction	Latency	Cycles / Issue
Load / Store	3	1
Integer Multiply	4	1
Integer Divide	36	36
Floating-point Add	3	1
Floating-point Multiply	5	2
Floating-point Divide	38	38

- **Latency**:
  - Minimum delay after which the first result is obtained, starting from the time when the first set of inputs is applied.
- **Cycles/Issue**:
  - Whenever a new set of inputs is applied to a functional unit (e.g. adder), it is called an *issue*.
  - Pipelined implementation of arithmetic unit reduces the number of clock cycles between successive issues.
  - For non-pipelined arithmetic units (e.g. divider), the number of clock cycles between successive issues is much higher.
    - Next input can be applied only after the previous operation is complete.

The Process of Integer Division

- In integer division, a *divisor* M and a *dividend* D are given.
- The objective is to find a third number Q, called the *quotient*, such that

$$D = Q \times M + R$$

where R is the *remainder* such that $0 \le R < M$.

- The relationship $D = Q \times M$ suggests that there is a close correspondence between division and multiplication.
  - Dividend, quotient and divisor correspond to product, multiplicand and multiplier, respectively.
  - Similar algorithms and circuits can be used for multiplication and division.

- One of the simplest division methods is the sequential digit-by-digit algorithm similar to that used in pencil-and-paper methods.

	0 1 1 0	Quotient $Q = Q_0Q_1Q_2Q_3$	
Divisor M 1 1 0	1 0 0 1 0 1	Dividend $D = R_0$	
	1 1 0	$Q_0.M$	*(Does not go; $Q_0 = 0$)*
	-------------		
	1 0 0 1 0 1	R_1	
	- 1 1 0	$Q_1.2^{-1}.M$	*(Does go; $Q_1 = 1$)*
	-------------		
	0 1 1 0 1	R_2	
	- 1 1 0	$Q_2.2^{-2}.M$	*(Does go; $Q_2 = 1$)*
	-------------		
	0 0 0 1	R_3	
	1 1 0	$Q_3.2^{-3}.M$	*(Does not go; $Q_3 = 0$)*
	-------------		
	0 0 1	R_4 = Remainder R	

$D = 37 = (1\,0\,0\,1\,0\,1)_2$

$M = 6 = (1\,1\,0)_2$

Quotient $Q = 6$

Remainder $R = 1$

- In the example, the quotient $Q = Q_0Q_1Q_2$… is computed one bit at a time.
  - At each step i, the divisor shifted i bits to the right (i.e. $2^{-i}.M$) is compared with the current partial remainder R_i.
  - The quotient bit Q_i is set to 0 (1) if $2^{-i}.M$ is greater than (less than) R_i .
  - The new partial remainder R_{i+1} is computed as:

$$R_{i+1} = R_i - Q_i.2^{-i}.M$$

- **Machine implementation:**
  - For hardware implementation, it is more convenient to shift the partial remainder to the left relative to a fixed divisor; thus

$$R_{i+1} = 2R_i - Q_i.M \quad (\text{instead of } R_{i+1} = R_i - Q_i.2^{-i}.M)$$

  - The final partial remainder is the required remainder shifted to the left, so that $R = 2^{-3}.R_4$ (see next slide).

Divisor M			Quotient Q
1 1 0	1 0 0 1 0 1	Dividend = $2R_0$	
	1 1 0	$Q_0.M$	0
	---------------		
	1 0 0 1 0 1	R_1	
	1 0 0 1 0 1 0	$2R_1$	
	1 1 0	$Q_1.M$	0 1
	---------------		
	0 1 1 0 1 0	R_2	
	0 1 1 0 1 0 0	$2R_2$	
	1 1 0	$Q_2.M$	0 1 1
	---------------		
	0 0 0 1 0 0	R_3	
	0 0 0 1 0 0 0	$2R_3$	
	1 1 0	$Q_3.M$	0 1 1 0
	---------------		
	0 0 1 0 0 0	$R_4 = 2^3.R$	

Do not subtract

$D = 37 = (1\ 0\ 0\ 1\ 0\ 1)_2$

$M = 6 = (1\ 1\ 0)_2$

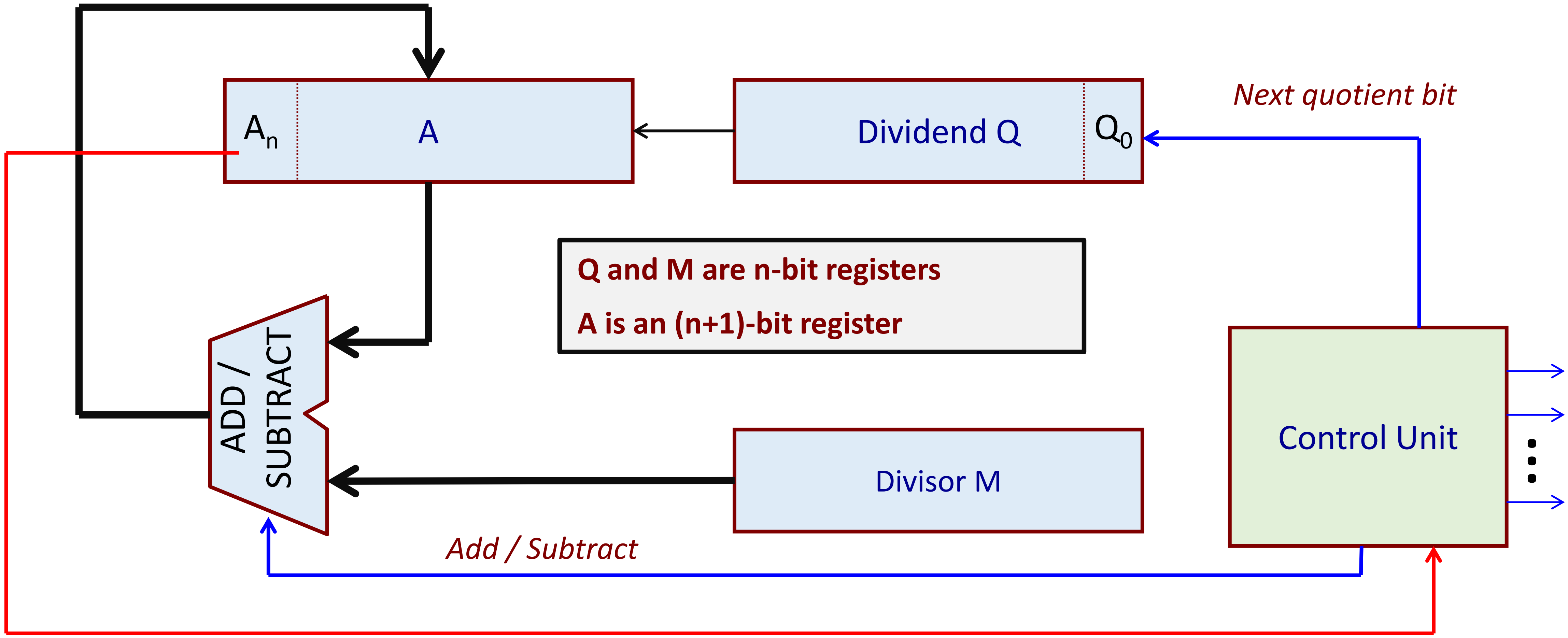
Quotient $Q = 6$

Remainder $R = 1$

Two alternatives to division

- We shall discuss two approaches:
  a) Restoring division
  b) Non-restoring division

(a) Restoring Division: The Data Path



Basic Steps (Restoring Division)

Repeat the following steps *n* times:

a) Shift the dividend one bit at a time starting into register *A*.

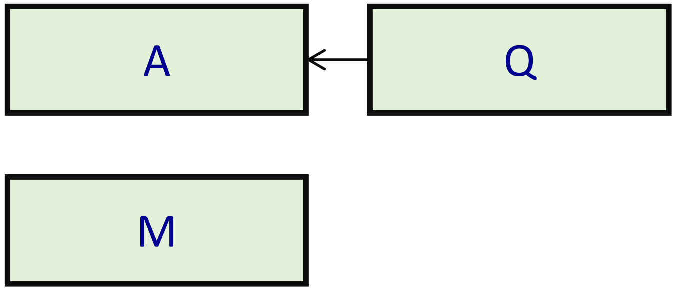
b) Subtract the divisor *M* from this register *A* (*trial subtraction*).

c) If the result is negative (*i.e. not going*):

- Add the divisor *M* back into the register *A* (*i.e. restoring back*).
- Record 0 as the next quotient bit.

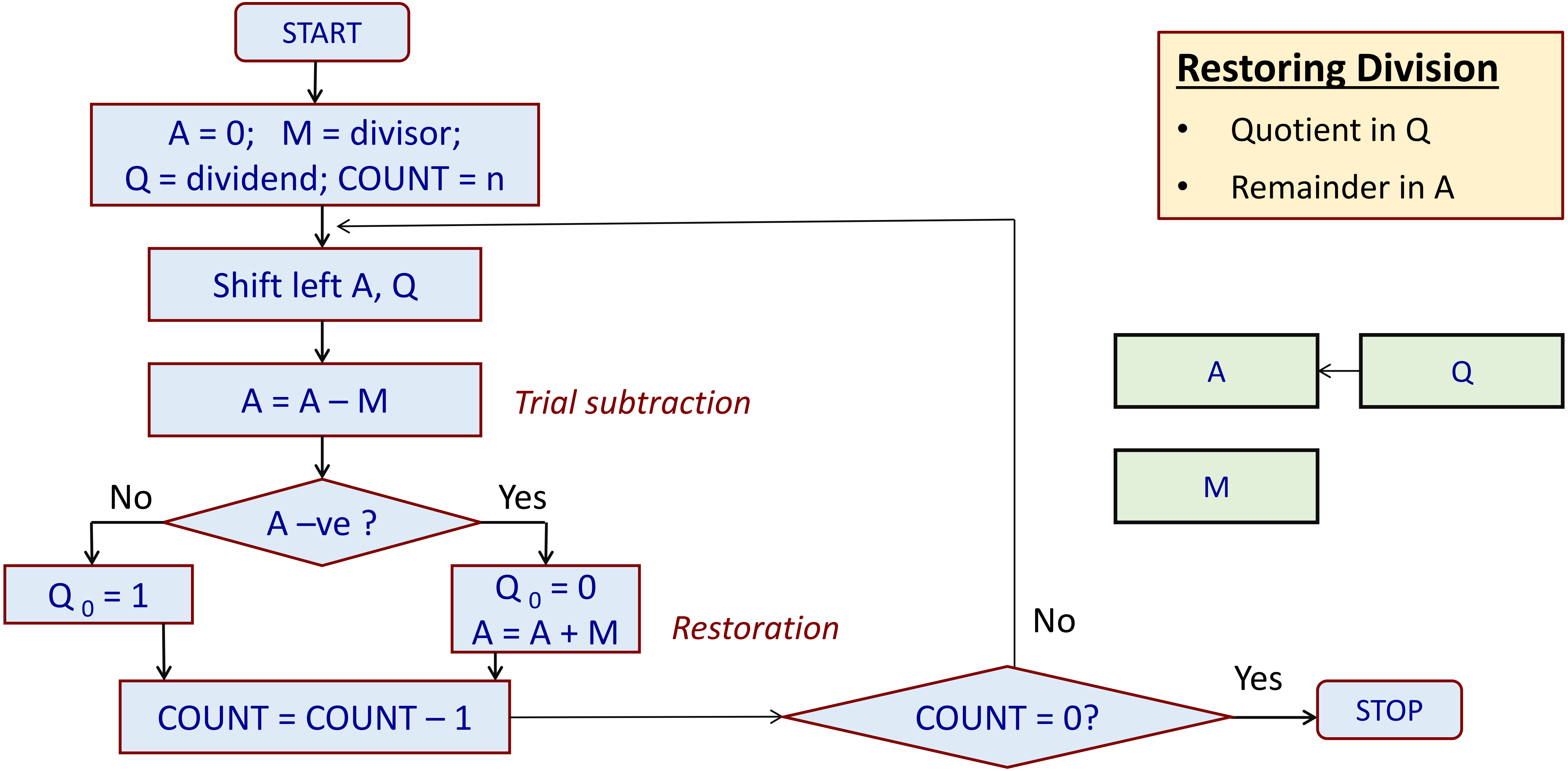
d) If the result is positive:

- Do not restore the intermediate result.
- Record 1 as the next quotient bit.



Restoring Division

- Quotient in Q
- Remainder in A



- **Analysis**:
  - For *n*-bit divisor and *n*-bit dividend, we iterate *n* times.
  - Number of trial subtractions: *n*
  - Number of restoring additions: *n/2* on the average
    - Best case: *0*
    - Worst case: *n*

A Simple Example: 8/3 for 4-bit representation (n=4)

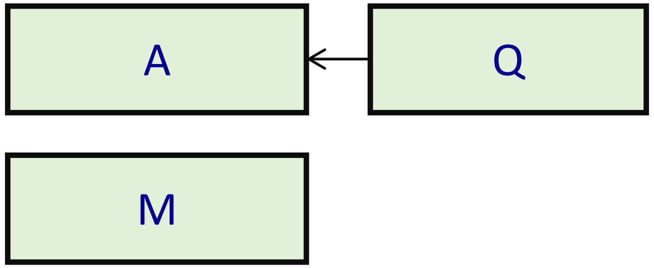
```
Initially:   0 0 0 0 0    1 0 0 0

Shift:       0 0 0 0 1    0 0 0 -
Subtract:      0 0 1 1
             ---------
Set Q0:      1 1 1 1 0
Restore:       0 0 1 1
             ---------
             0 0 0 0 1    0 0 0 0
Shift:       0 0 0 1 0    0 0 0 -
Subtract:      0 0 1 1
             ---------
Set Q0:      1 1 1 1 1
Restore:       0 0 1 1
             ---------
             0 0 0 1 0    0 0 0 0
```

```
Shift:       0 0 1 0 0    0 0 0 -
Subtract:      0 0 1 1
             ---------
Set Q0:      0 0 0 0 1
             0 0 0 0 0    0 0 0 1
Shift:       0 0 0 1 0    0 0 1 -
Subtract:      0 0 1 1
             ---------
Set Q0:      1 1 1 1 1
Restore:       0 0 1 1
             ---------
             0 0 0 1 0    0 0 1 0
```

Remainder
00010 = 2

Quotient
0010 = 2

(b) Non-Restoring Division



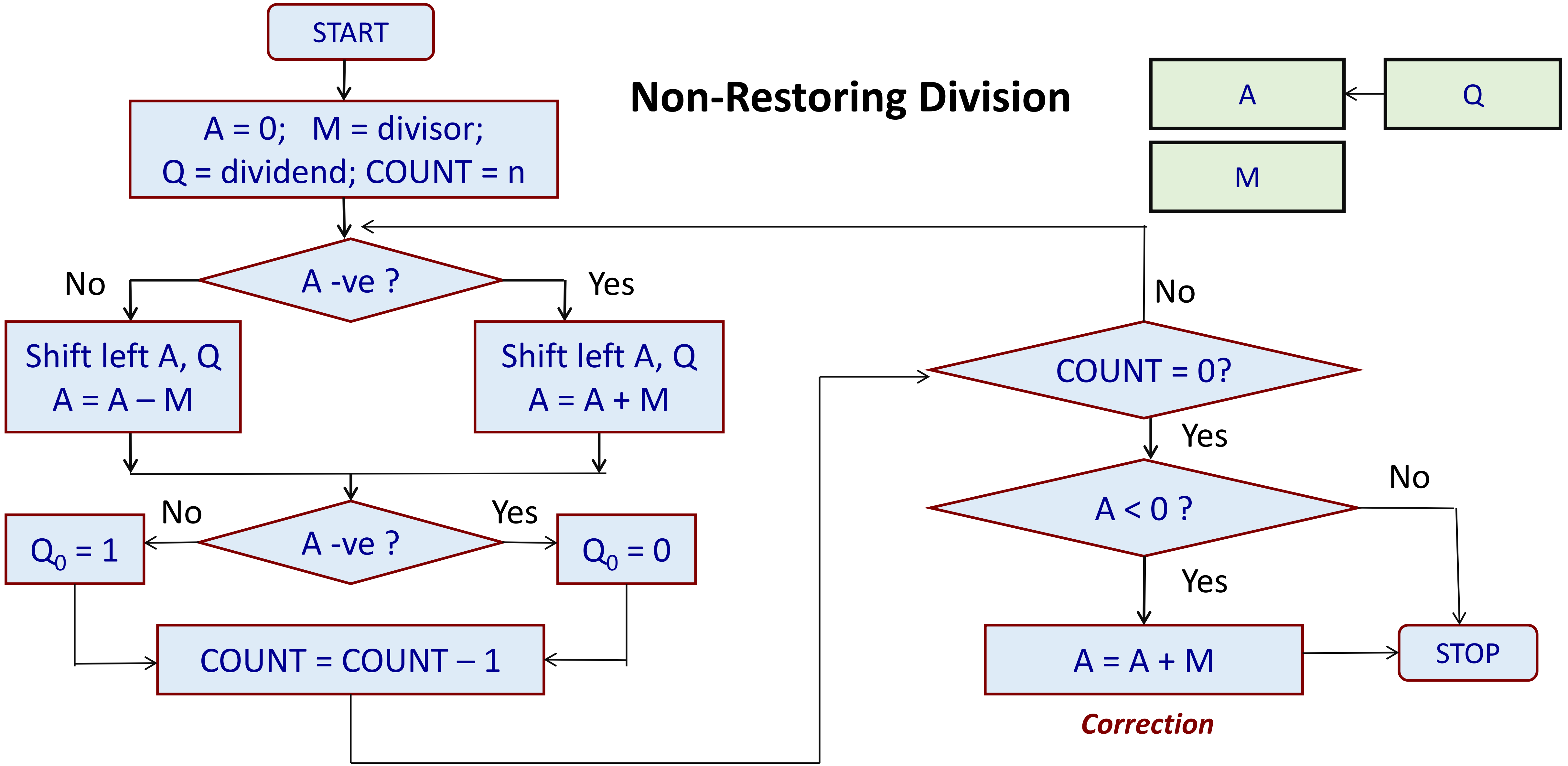
- The performance of restoring division algorithm can be improved by exploiting the following observation.
- In restoring division, what we do actually is:
  - If *A* is positive, we shift it left and subtract *M*.
    - That is, we compute $2A - M$.
  - If *A* is negative, we restore is by doing A+M, shift it left, and then subtract *M*.
    - That is, we compute $2(A + M) - M = 2A + M$.
- We can accordingly modify the basic division algorithm by eliminating the restoring step → *NON-RESTORING DIVISION*.

Shift left means multiplying by 2.

- **Basic steps in non-restoring division:**
  a) Start by initializing register A to 0, and repeat steps (b)-(d) n times.
  b) If the value in register A is positive,
     - Shift A and Q left by one bit position.
     - Subtract M from A.
  c) If the value in register A is negative,
     - Shift A and Q left by one bit position.
     - Add M to A.
  d) If A is positive, set $Q_0 = 1$; else, set $Q_0 = 0$.
  e) If A is negative, add M to A as a final corrective step.

17

Non-Restoring Division



A Simple Example: 8/3 for n=4

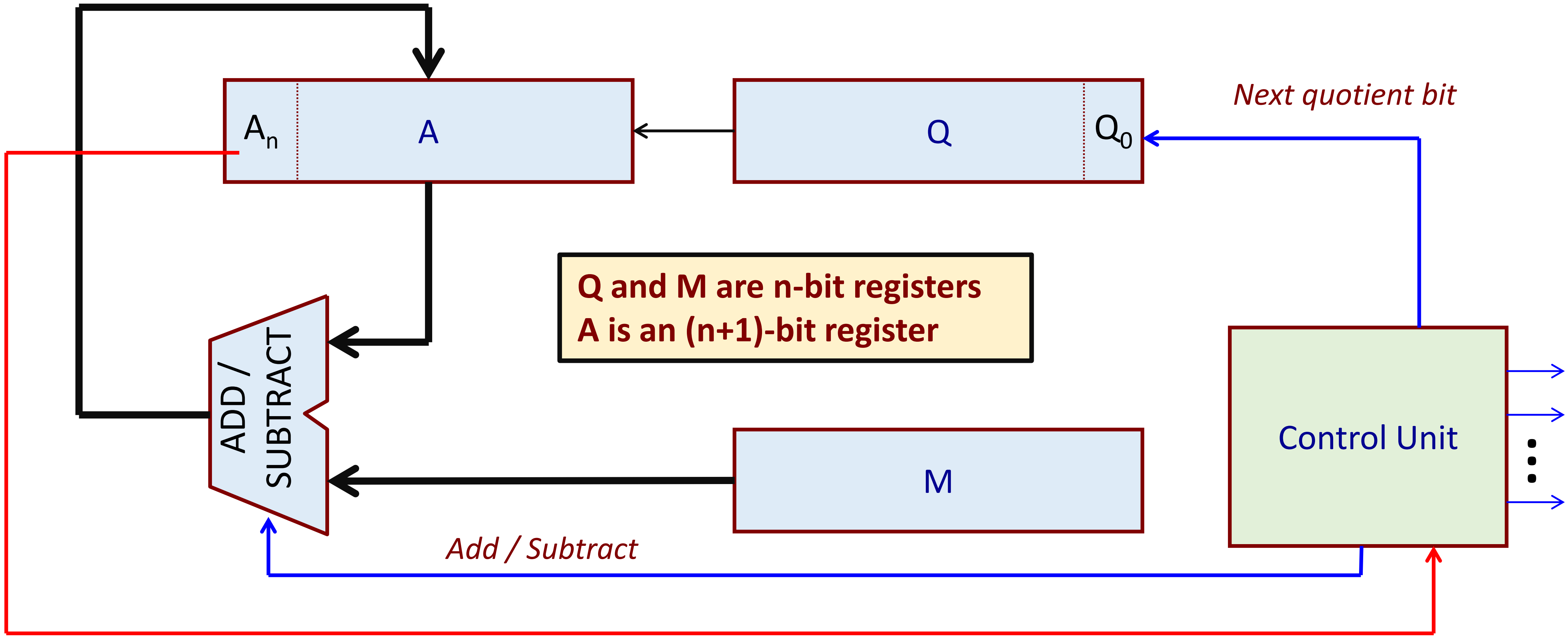
```
Initially:     0 0 0 0 0     1 0 0 0

Shift:         0 0 0 0 1     0 0 0 –
Subtract:   -    0 0 1 1
Set Q0:        1 1 1 1 0     0 0 0 0
Shift:         1 1 1 0 0     0 0 0 –
Add:             0 0 1 1
Set Q0:        1 1 1 1 1     0 0 0 0
Shift:         1 1 1 1 0     0 0 0 –
Add:             0 0 1 1
Set Q0:        0 0 0 0 1     0 0 0 1
```

```
Shift:         0 0 0 1 0     0 0 1 –
Subtract:   -    0 0 1 1
Set Q0:        1 1 1 1 1     0 0 1 0
Correction Add:
               1 1 1 1 1
               0 0 0 1 1
               0 0 0 1 0
```

Quotient
0010 = 2

Remainder
00010 = 2

Data Path for Non-Restoring Division



High Speed Dividers

- Some of the methods used to increase the speed of multiplication can also be modified to speed up division.
  - High-speed addition and subtraction.
  - High-speed shifting.
  - Combinational array divider (implementing restoring division).
- The main difficulty is that it is very difficult to implement division in a pipeline to improve the performance.
  - Unlike multiplication, where carry-save Wallace tree multipliers can be used for pipeline implementation.

FLOATING-POINT NUMBERS

Representing Fractional Numbers

- A binary number with fractional part

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

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 $0.111111\ldots_2$ 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: $(1/3)*3 \neq 1$.

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

s	E	M

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?

$2^{24} = 10^x$

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

$x = 7.2$ -- 7 significant decimal places

Range of exponent?

$2^{127} = 10^y$

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

$y = 38.1$ -- 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*: $1 \leq E \leq 254$ (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

4C700000 in hex

Special Values

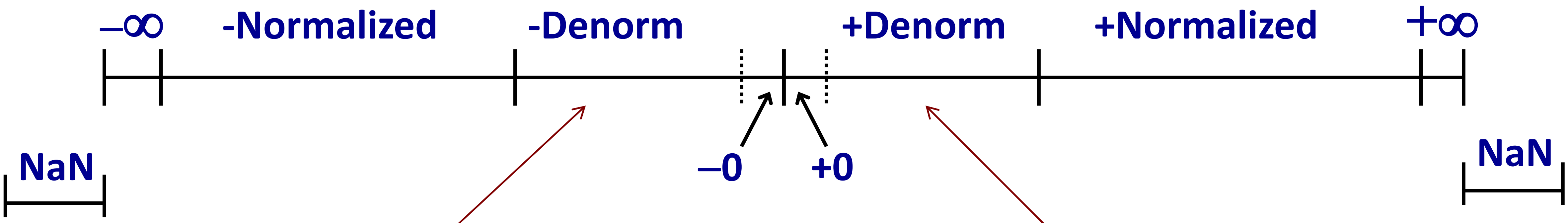
- When E = 000…0
  - M = 000…0 represents the value 0.
  - M ≠ 000…0 represents numbers very close to 0.
- When E = 111…1
  - M = 000…0 represents the value ∞ (infinity).
  - M ≠ 000…0 represents *Not-a-Number* (NaN).

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 (beyond 32 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

1. 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

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.

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

  Residue

- Result: 1.00010001001×2^8

Subtraction Example

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

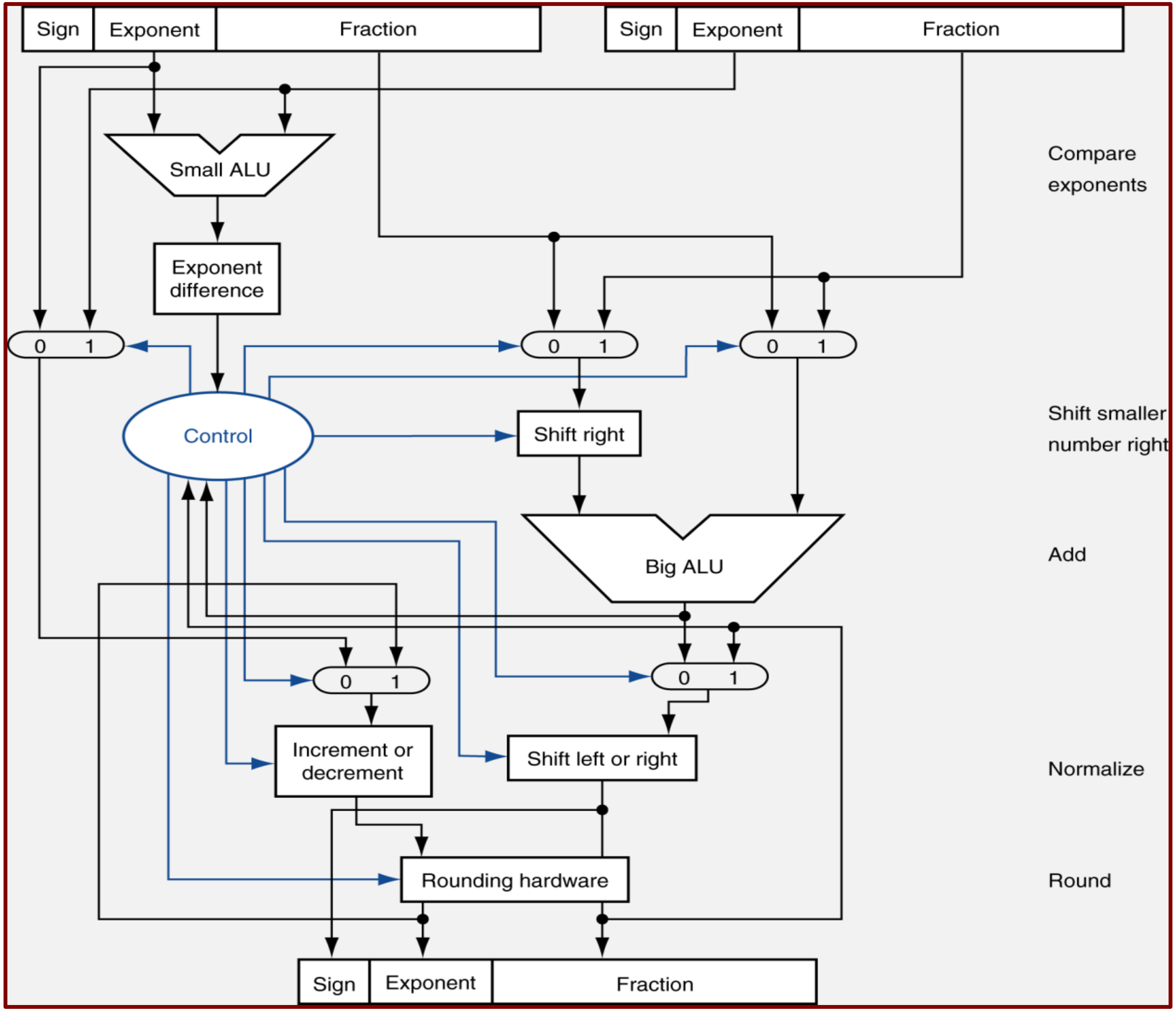
  $F1 = (270.75)_{10} = (100001110.11)_2 = 1.0000111011 \times 2^8$

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

- 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

Floating-Point Multiplication

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

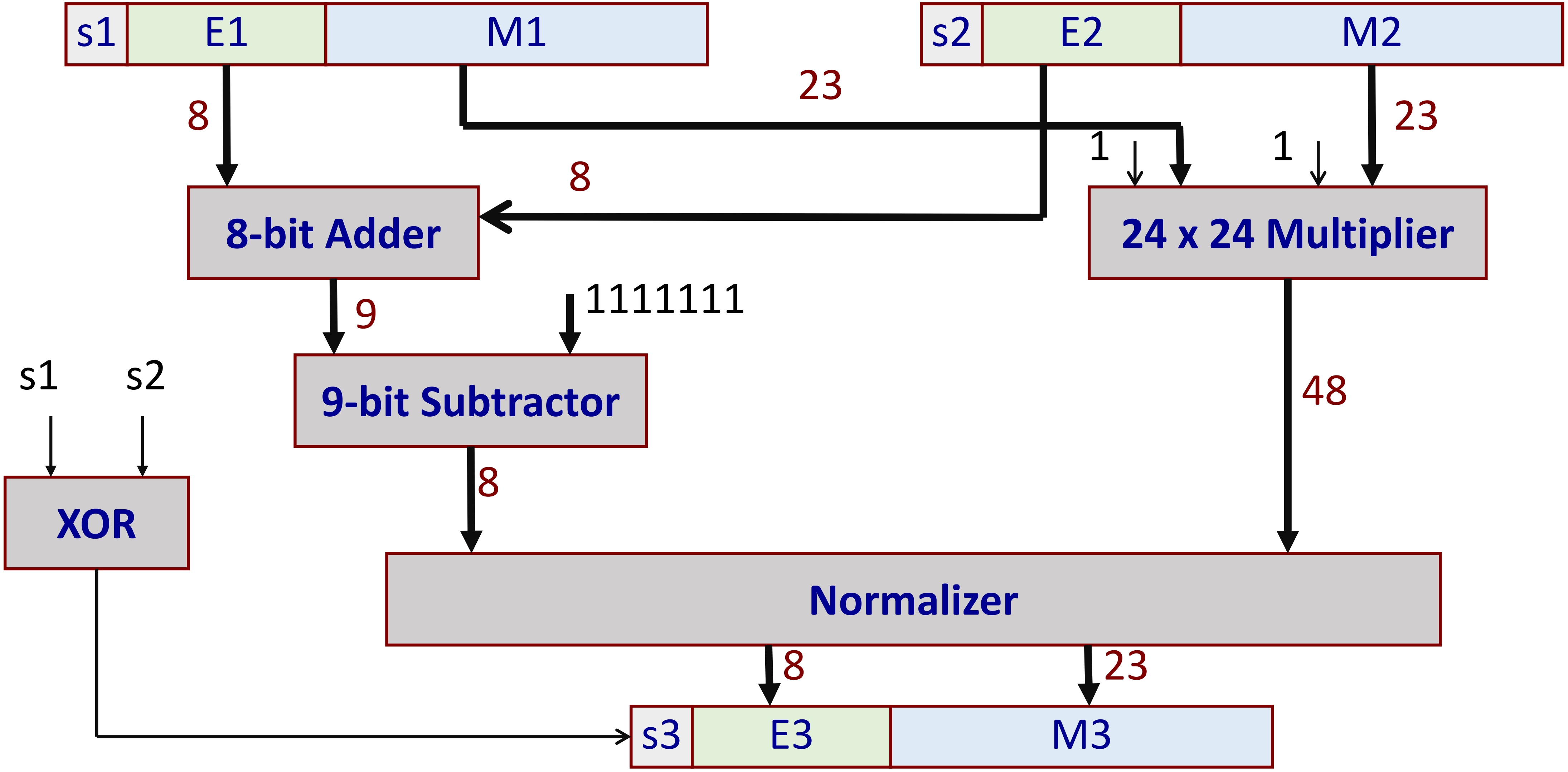
Multiplication Example

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

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

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

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


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

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.

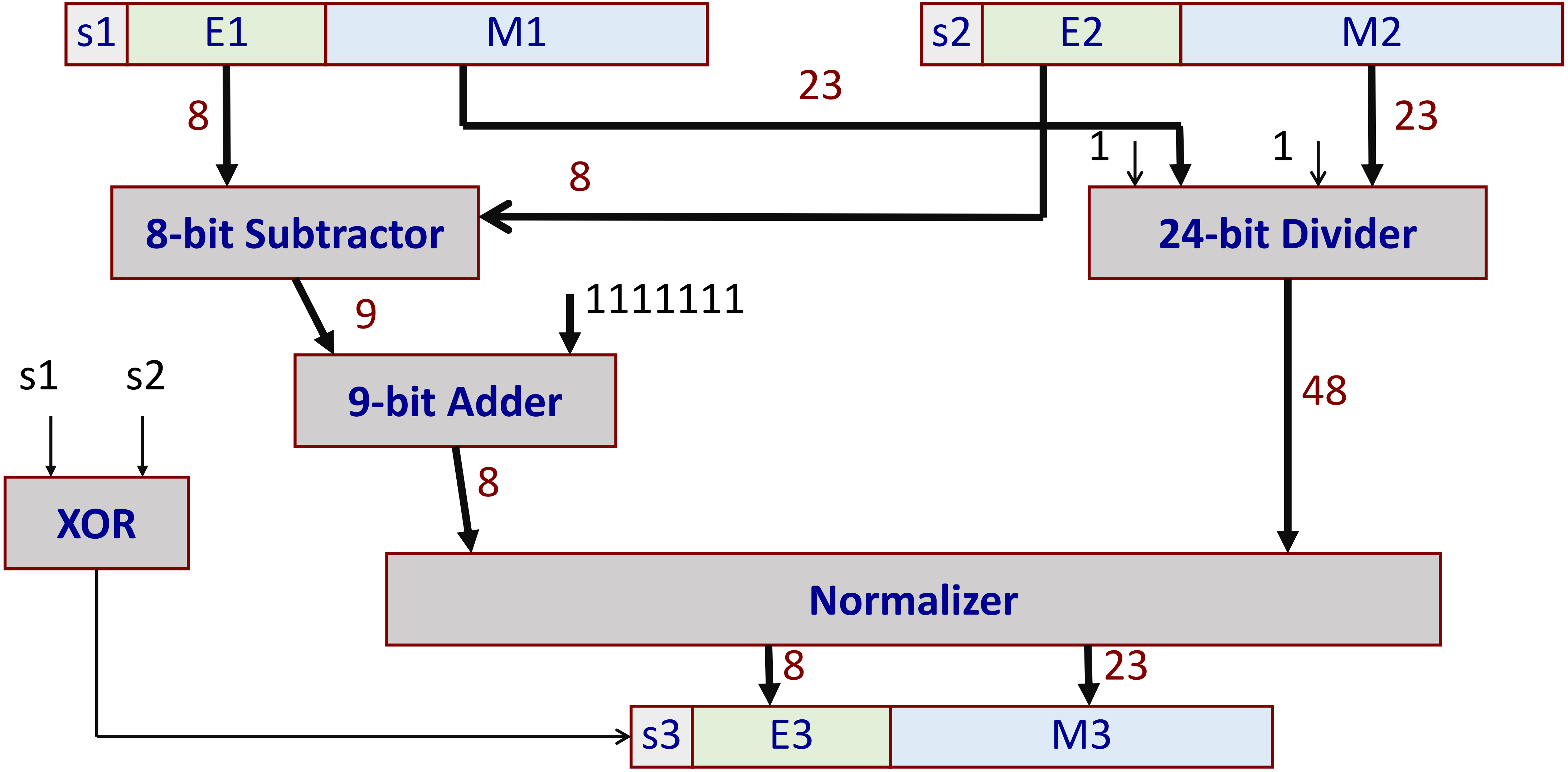
Division Example

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

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

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

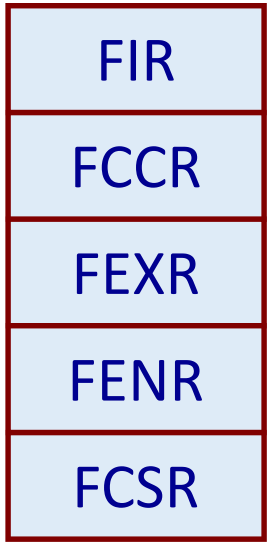
- Subtract the exponents: 8 – 1 = 7
- Divide the mantissas: 0.1110010
- Result: 0.1110010×2^7
- After normalization: 1.110010×2^6


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

FLOATING-POINT ARITHMETIC in MIPS

- 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



Special-purpose Registers

Typical Floating Point Instructions in MIPS

- 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

- 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 points 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).