EDA322
Digital Design

Lecture 12:
Arithmetic Units

Ioannis Sourdis

Outline of Lecture 12

- Arithmetic units
  - Multipliers
  - Dividers
- Number representation
  - accuracy and resolution.
  - Fixed and floating point

Arithmetic Units

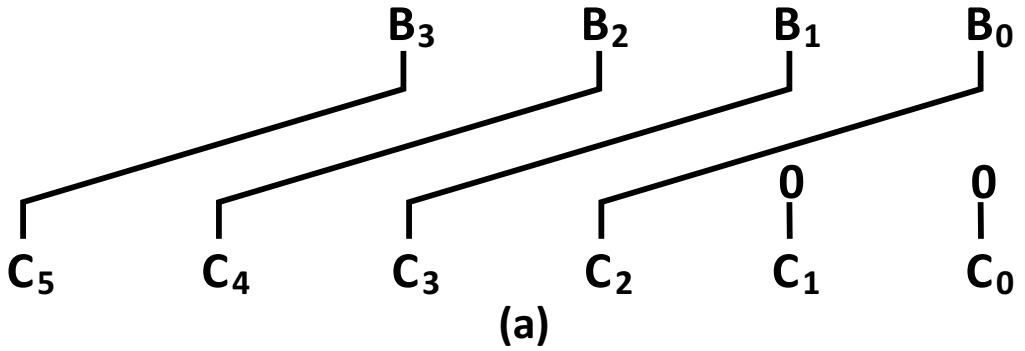
Multipliers, Dividers

Incrementing & Decrementing

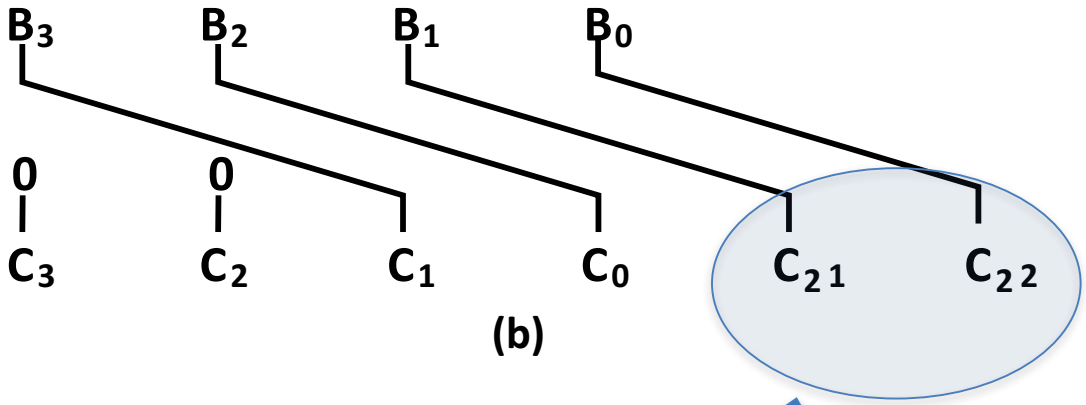
- *Incrementing*
  - Adding a fixed value to an arithmetic variable
  - Fixed value is often 1, called *counting (up*)
  - Examples: A + 1, B + 4
  - Functional block is called *incrementer*
- *Decrementing*
  - Subtracting a fixed value from an arithmetic variable
  - Fixed value is often 1, called *counting (down*)
  - Examples: A - 1, B - 4
  - Functional block is called *decrementer*

Multiplication/Division by 2^n

- (a) Multiplication by 100
  - Shift left by 2

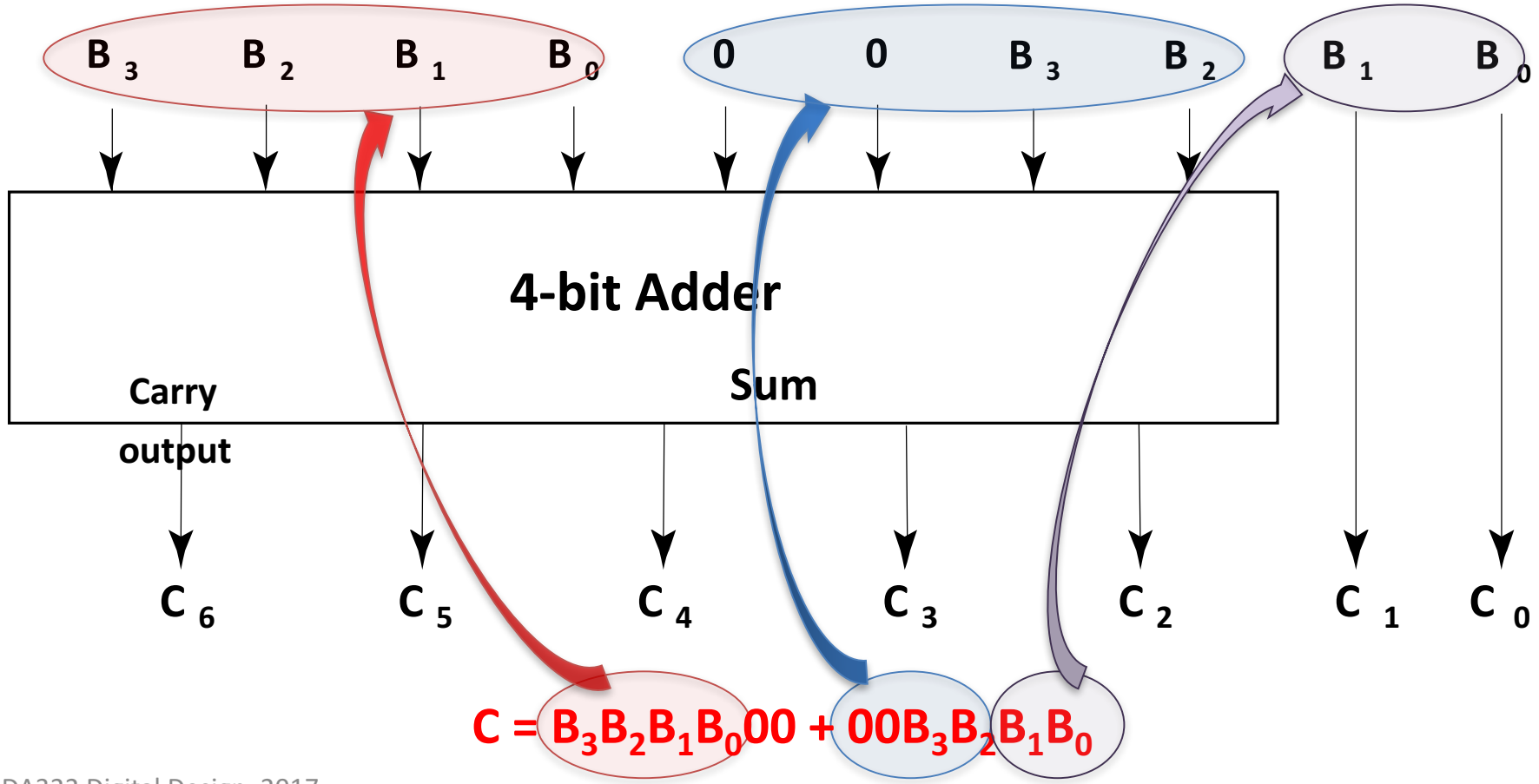


- (b) Division by 100
  - Shift right by 2
  - Remainder preserved



Multiplication by a Constant

- Multiplication of B(3:0) by 101
- C = 4*B + 1*B = B<< by two bits + B



Binary Multiplication (Unsigned)

$1\ 0\ 0\ 0_{two} = 8_{ten}$ multiplicand

$1\ 0\ 0\ 1_{two} = 9_{ten}$ multiplier

---

1 0 0 0 partial products

0 0 0 0

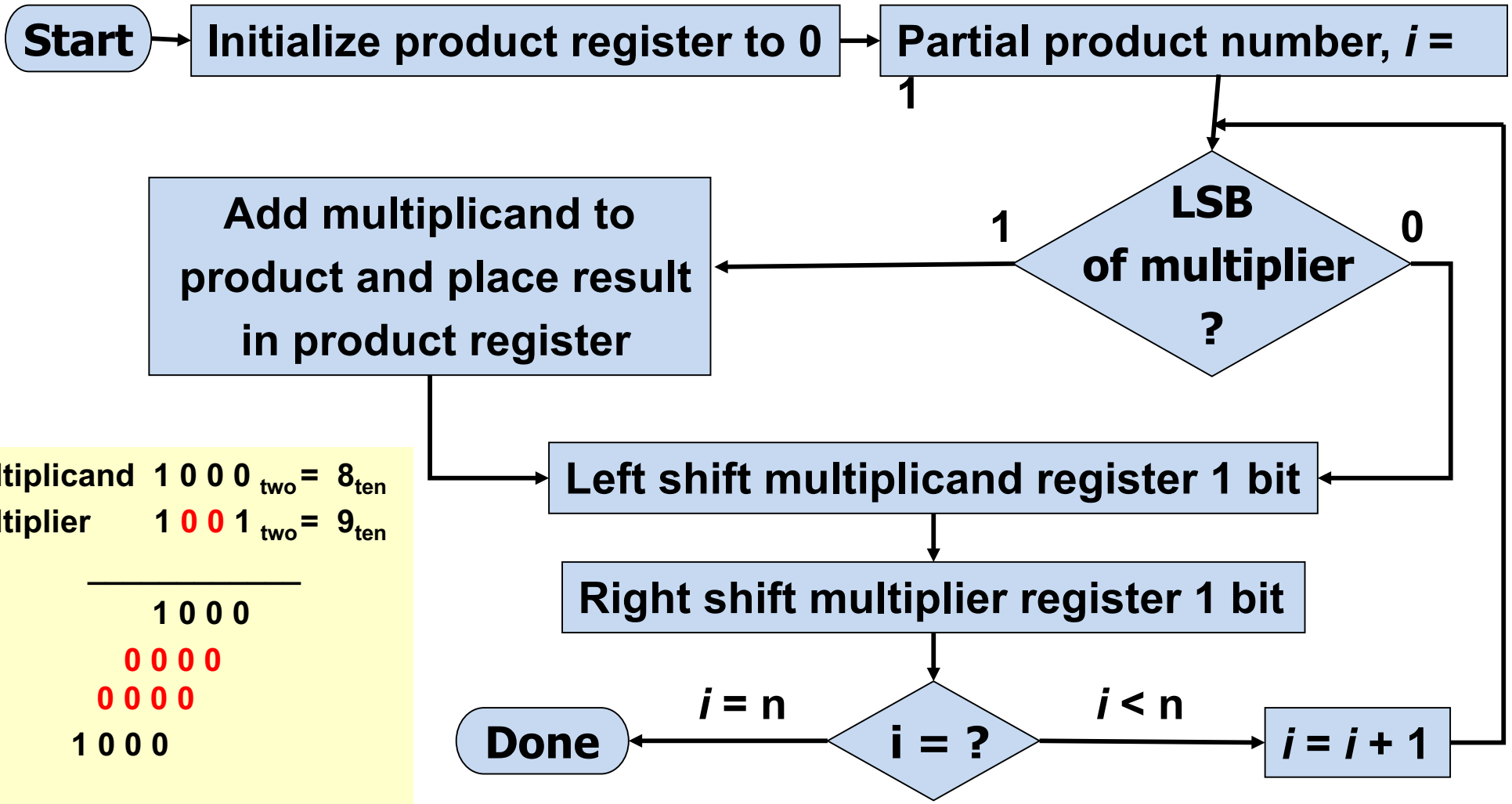
0 0 0 0

1 0 0 0

$1\ 0\ 0\ 1\ 0\ 0\ 0_{two} = 72_{ten}$

Basic algorithm: For i = 1 to n, Where n is the total number of bits of the operands, only If *i*-th bit of multiplier is 1, then add multiplicand $\times\ 2^{i-1}$ (multiplicand shifted left by i-1 bits) to product

Multiplication Flowchart



multiplicand $1\,0\,0\,0_{two} = 8_{ten}$
multiplier $1\,0\,0\,1_{two} = 9_{ten}$

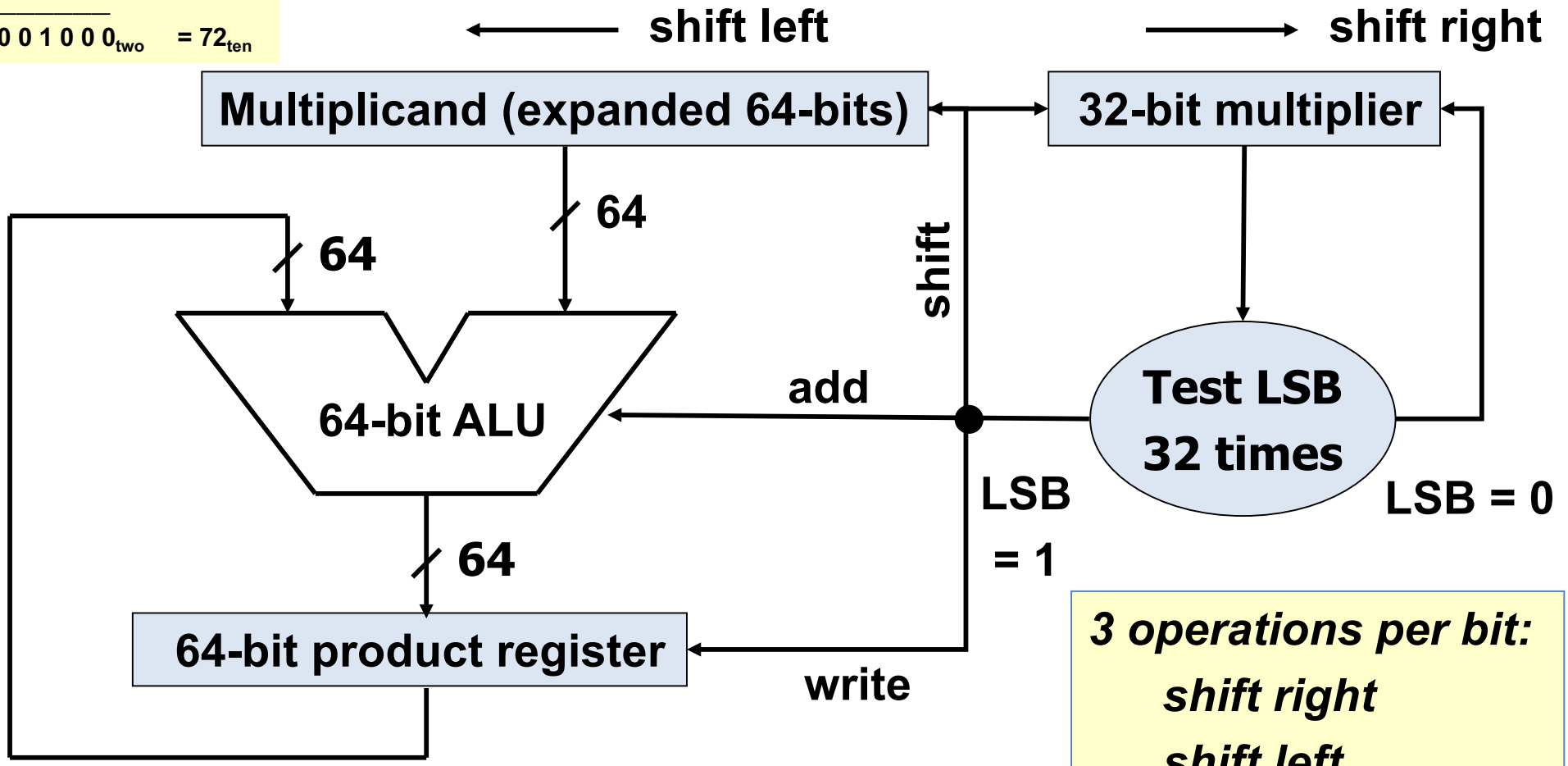
```
    1 0 0 0
  0 0 0 0
0 0 0 0
1 0 0 0
-----------------
1 0 0 1 0 0 0
```

$1\,0\,0\,1\,0\,0\,0_{two} = 72_{ten}$

$1000_{two} = 8_{ten}$
$1001_{two} = 9_{ten}$

```
  1000
 0000
0000
1000
---------
1001000two = 72ten
```

Serial Multiplication



3 operations per bit:

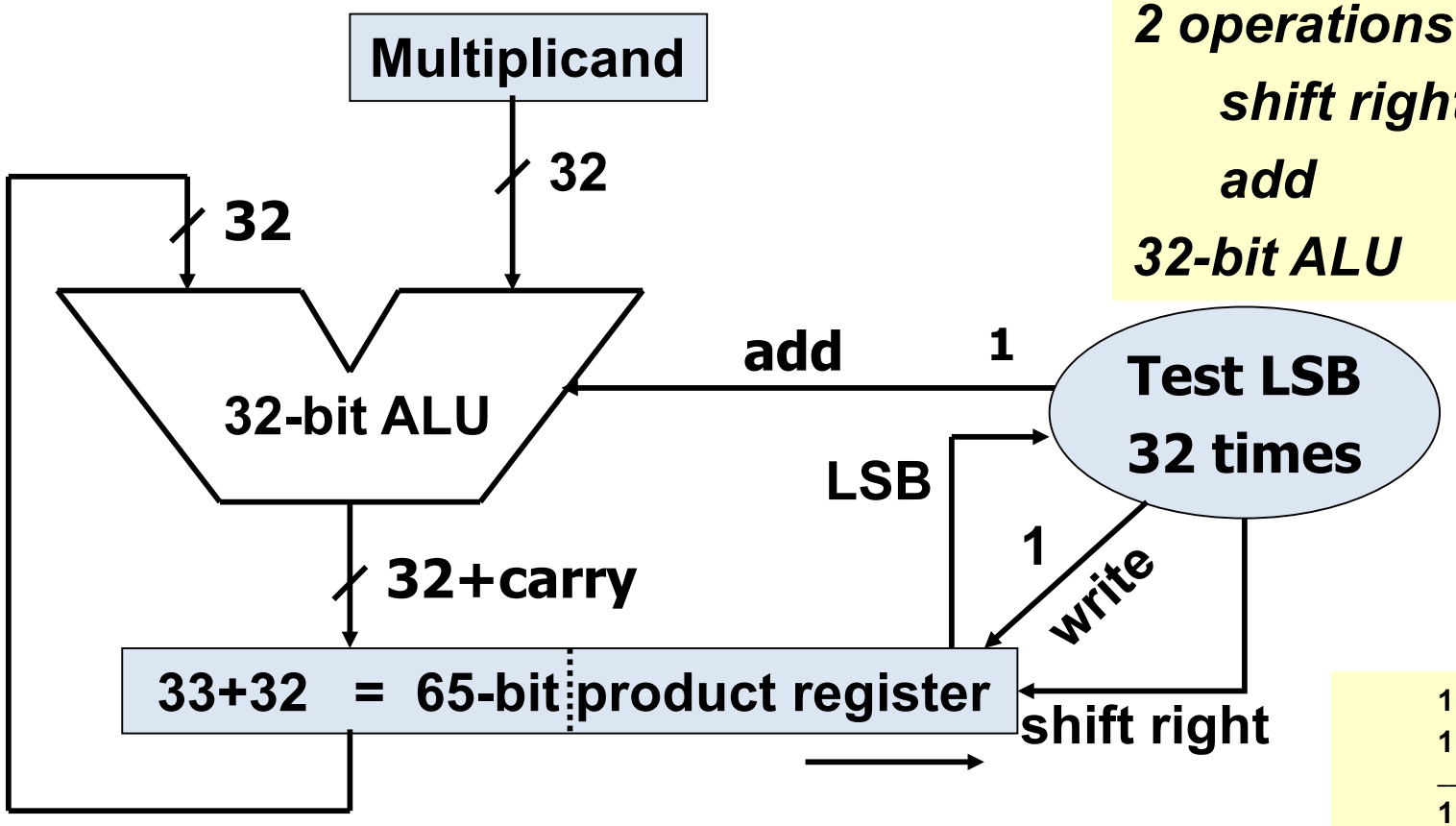
shift right

shift left

add

Need 64-bit ALU

Serial Multiplication (Improved)



2 operations per bit:

shift right

add

32-bit ALU

$1000_{two} = 8_{ten}$

$1001_{two} = 9_{ten}$

```
  1000
 0000
0000
1000
-------
1001000two = 72ten
```

Example: $0010_{two} \times 0011_{two}$

$0010_{two} \times 0011_{two} = 0110_{two}$, i.e., $2_{ten} \times 3_{ten} = 6_{ten}$

Iteration	Step	Multiplicand	Product
0	Initial values	0010	00000 0011
1	LSB=1 => Prod=Prod+Mcand	0010	00010 0011
	Right shift product	0010	00001 0001
2	LSB=1 => Prod=Prod+Mcand	0010	00011 0001
	Right shift product	0010	00001 1000
3	LSB=0 => no operation	0010	00001 1000
	Right shift product	0010	00000 1100
4	LSB=0 => no operation	0010	00000 1100
	Right shift product	0010	00000 0110

Multiplying with Signs

- Convert numbers to magnitudes.
- Multiply the two magnitudes through 32 iterations.
- Negate the result if the signs of the multiplicand and multiplier differed.

Adding Partial Products

multiplicand $1\,0\,0\,0_{two} = 8_{ten}$
multiplier $1\,0\,0\,1_{two} = 9_{ten}$

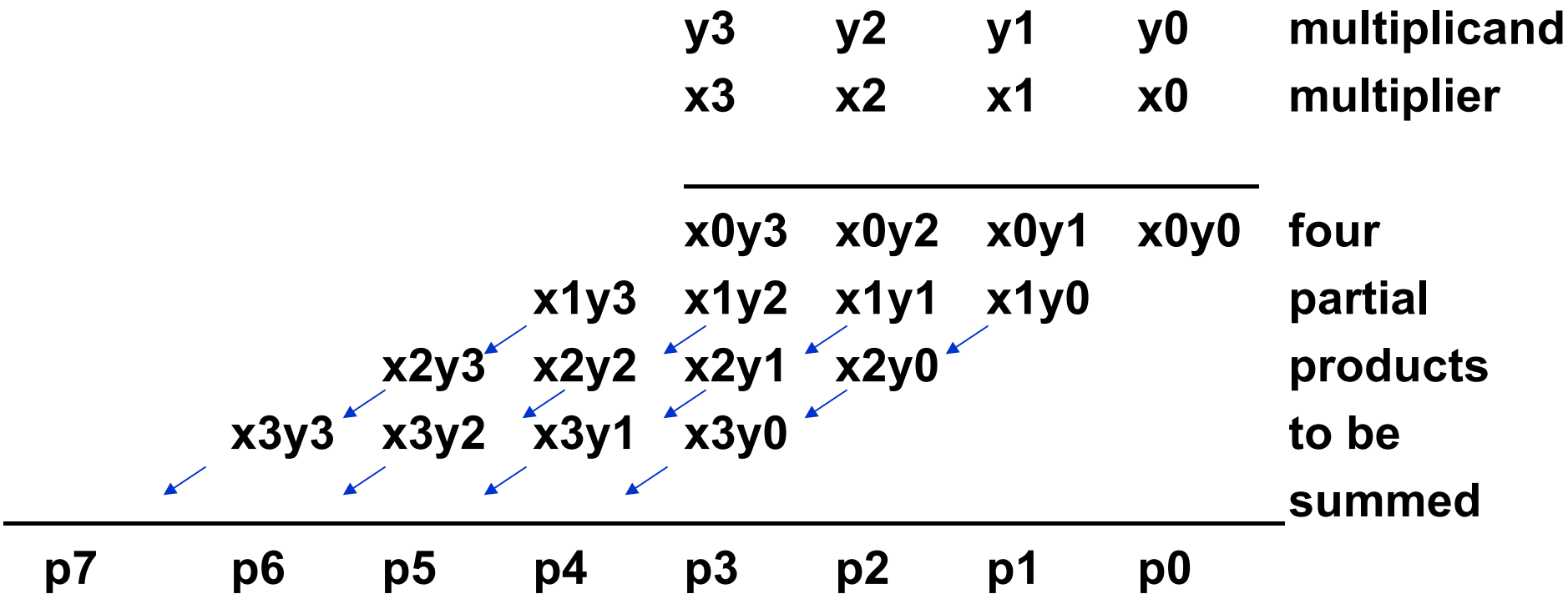
```
      1 0 0 0
    0 0 0 0
  0 0 0 0
1 0 0 0
-----------------
1 0 0 1 0 0 0 two = 72ten
```

```
                                  y3    y2    y1    y0        multiplicand
                                  x3    x2    x1    x0        multiplier
                                ------------------------
                                x0y3  x0y2  x0y1  x0y0        four
                          x1y3  x1y2  x1y1  x1y0              partial
                    x2y3  x2y2  x2y1  x2y0                    products
              x3y3  x3y2  x3y1  x3y0                          to be
                                                              summed
------------------------------------------------------
p7    p6    p5    p4    p3    p2    p1    p0
```

carry← x1y3 (carry)
carry← x2y3 x2y2 (carry)
carry← x3y3 x3y2 x3y1 (carry)

Requires three 4-bit additions. Slow.

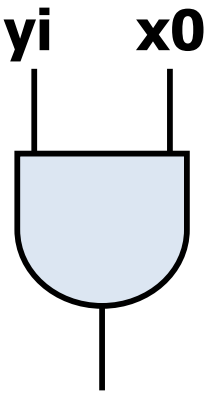
Array Multiplier: Carry Forward



Note: Carry is added to the next partial product (carry-save addition). Adding the carry from the final stage needs an extra (ripple-carry stage. These additions are faster but we need four stages.

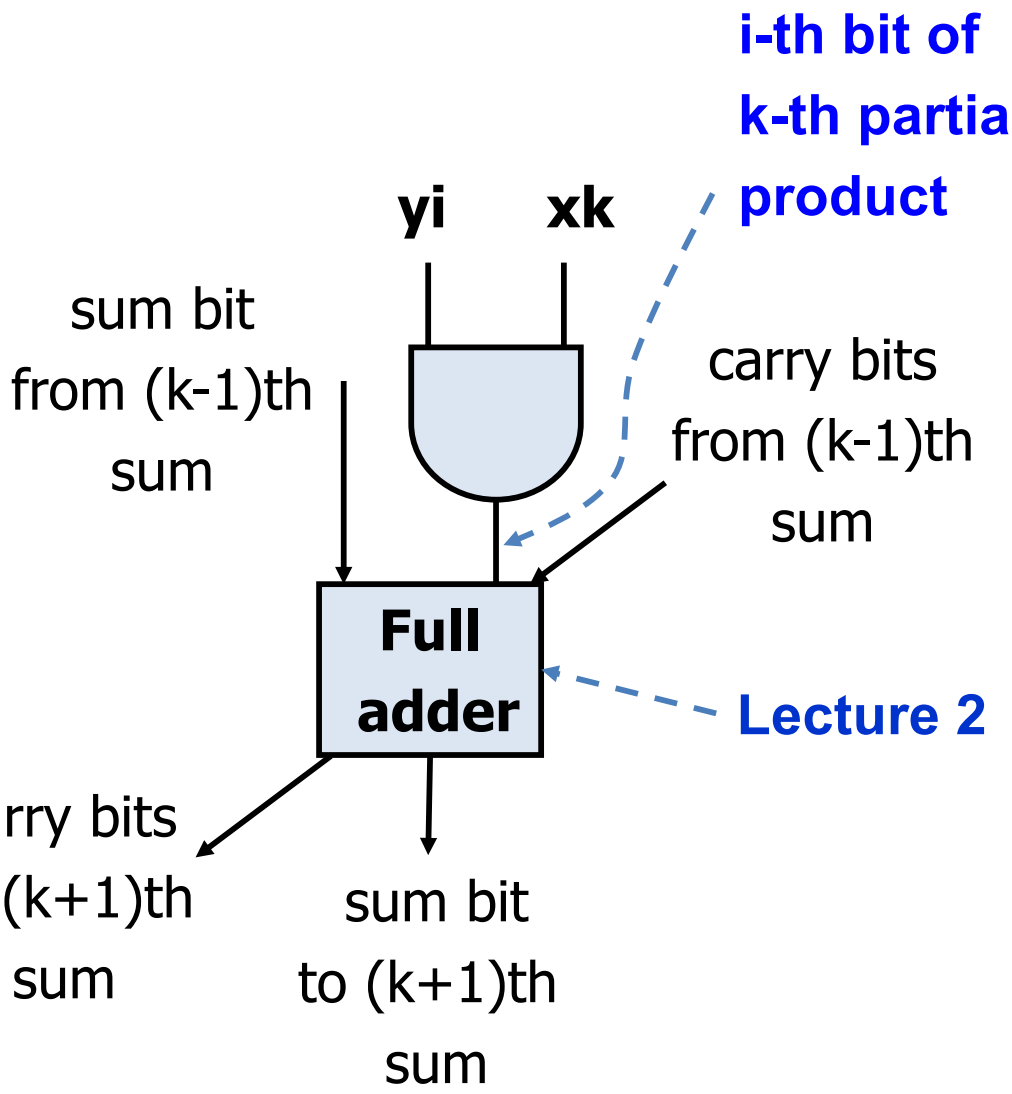
Basic Building Blocks

- Two-input AND
  - **for the partial product**
- Full-adder

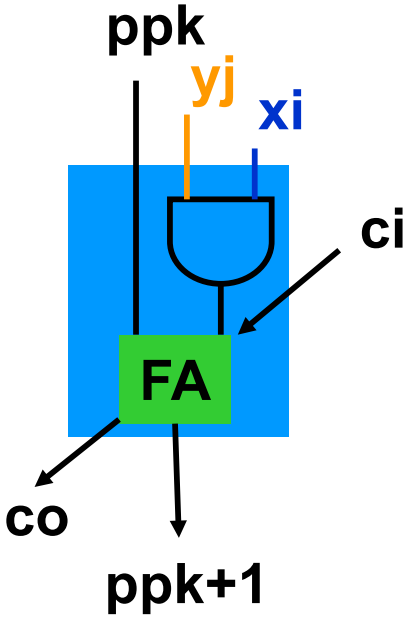


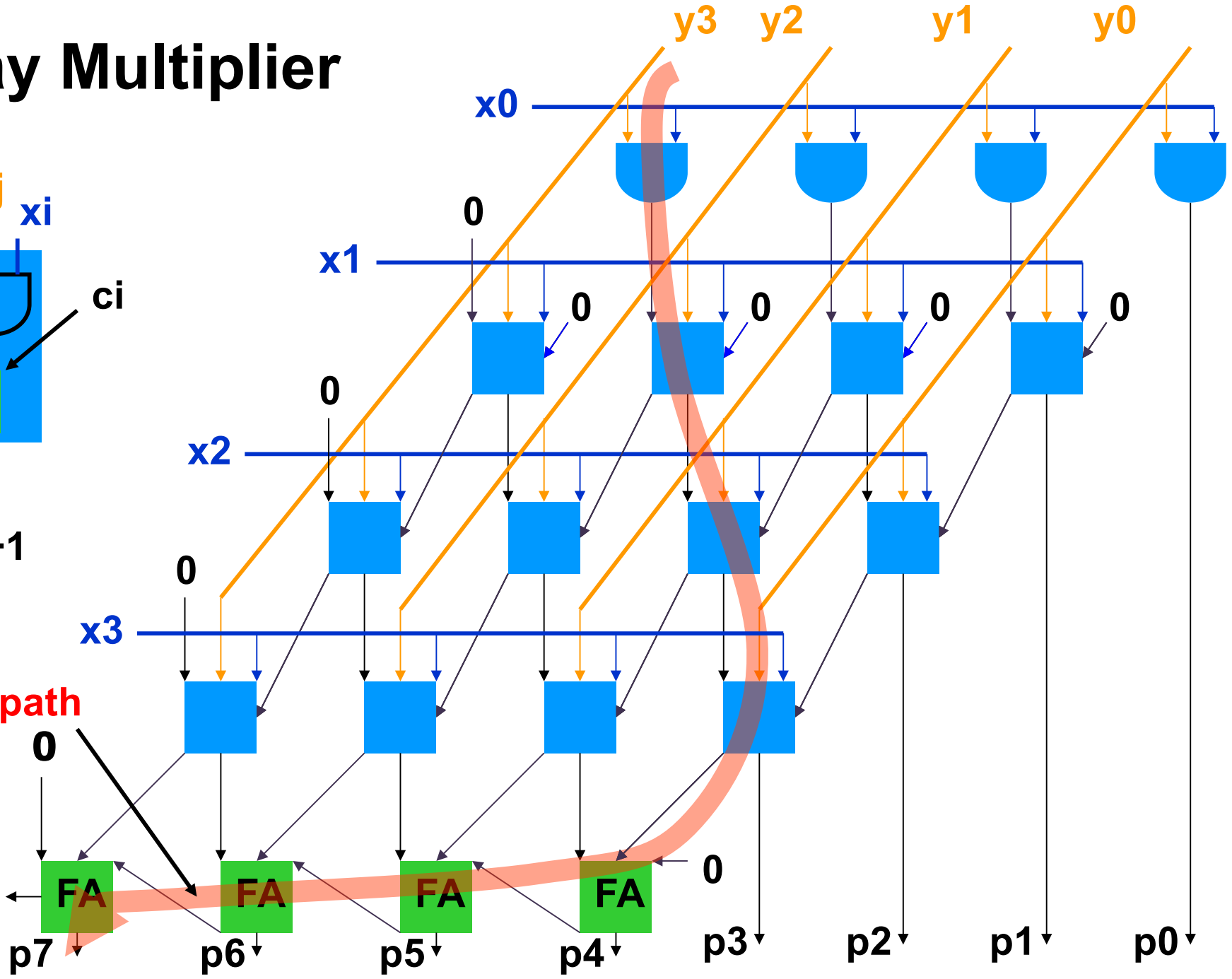
p0i = x0*yi

0th partial product

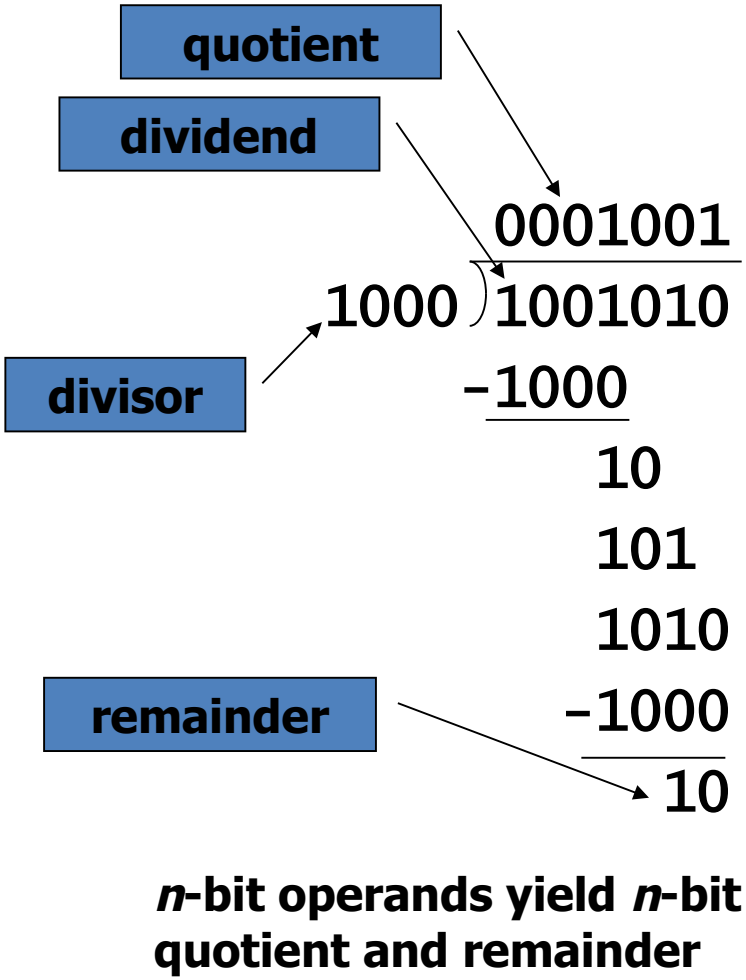


Array Multiplier





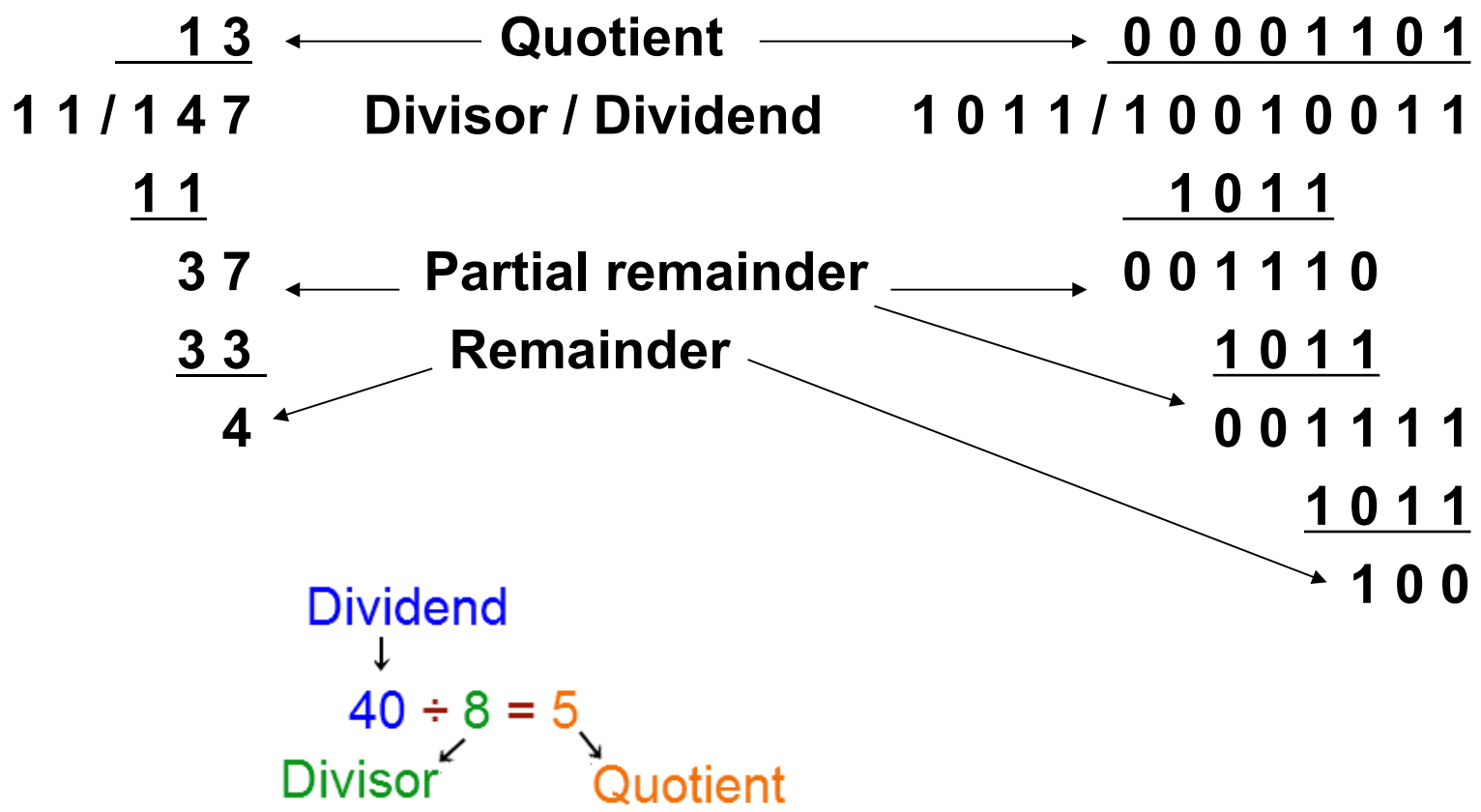
Binary Division



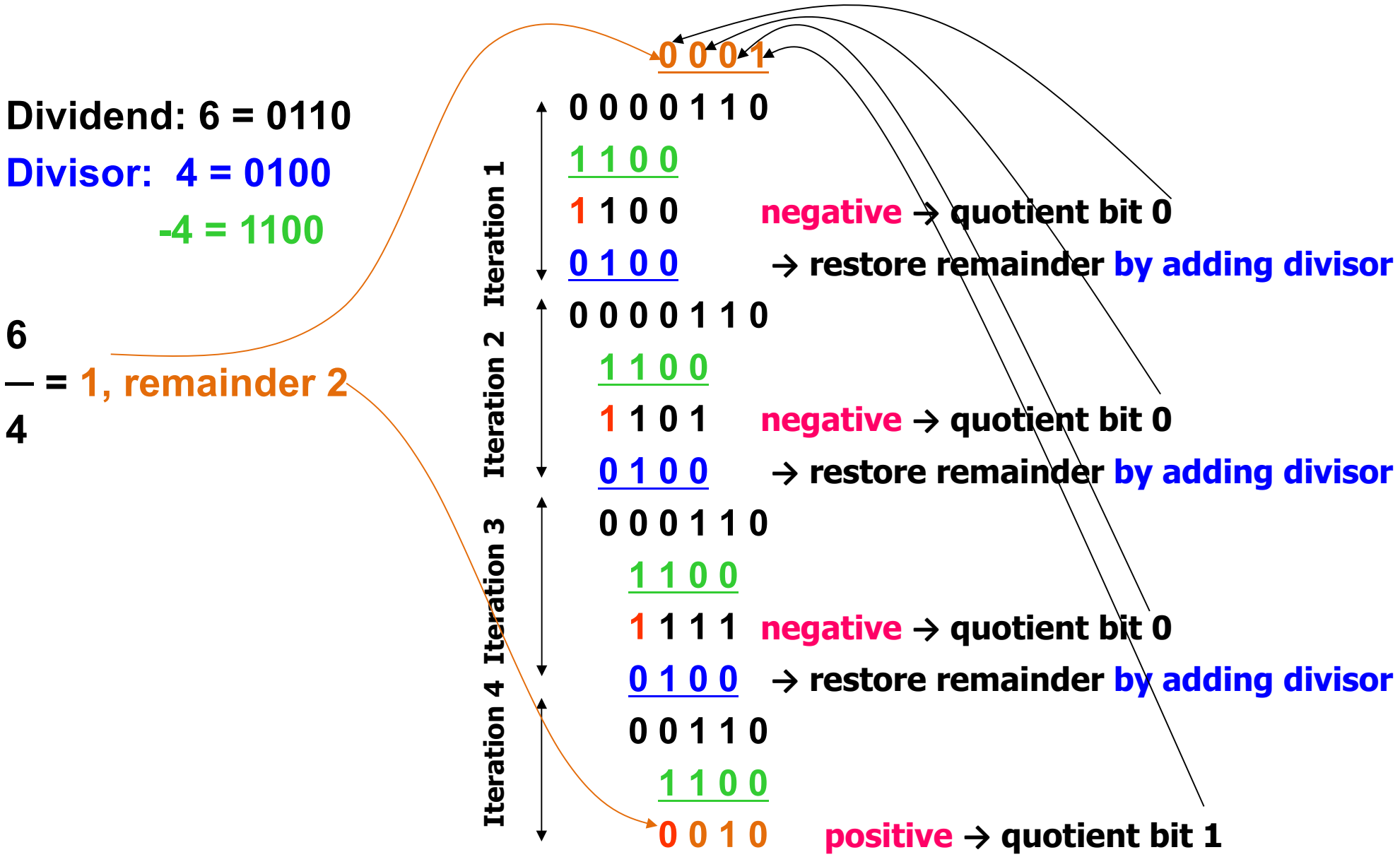
***n*-bit operands yield *n*-bit quotient and remainder**

- Check for 0 divisor
- Long division approach
  - If divisor ≤ dividend bits
    - 1 bit in quotient, subtract
  - Otherwise
    - 0 bit in quotient, bring down next dividend bit
- Restoring division
  - Do the subtract, and if remainder goes < 0, add divisor back
- Signed division
  - Divide using absolute values
  - Adjust sign of quotient and remainder as required

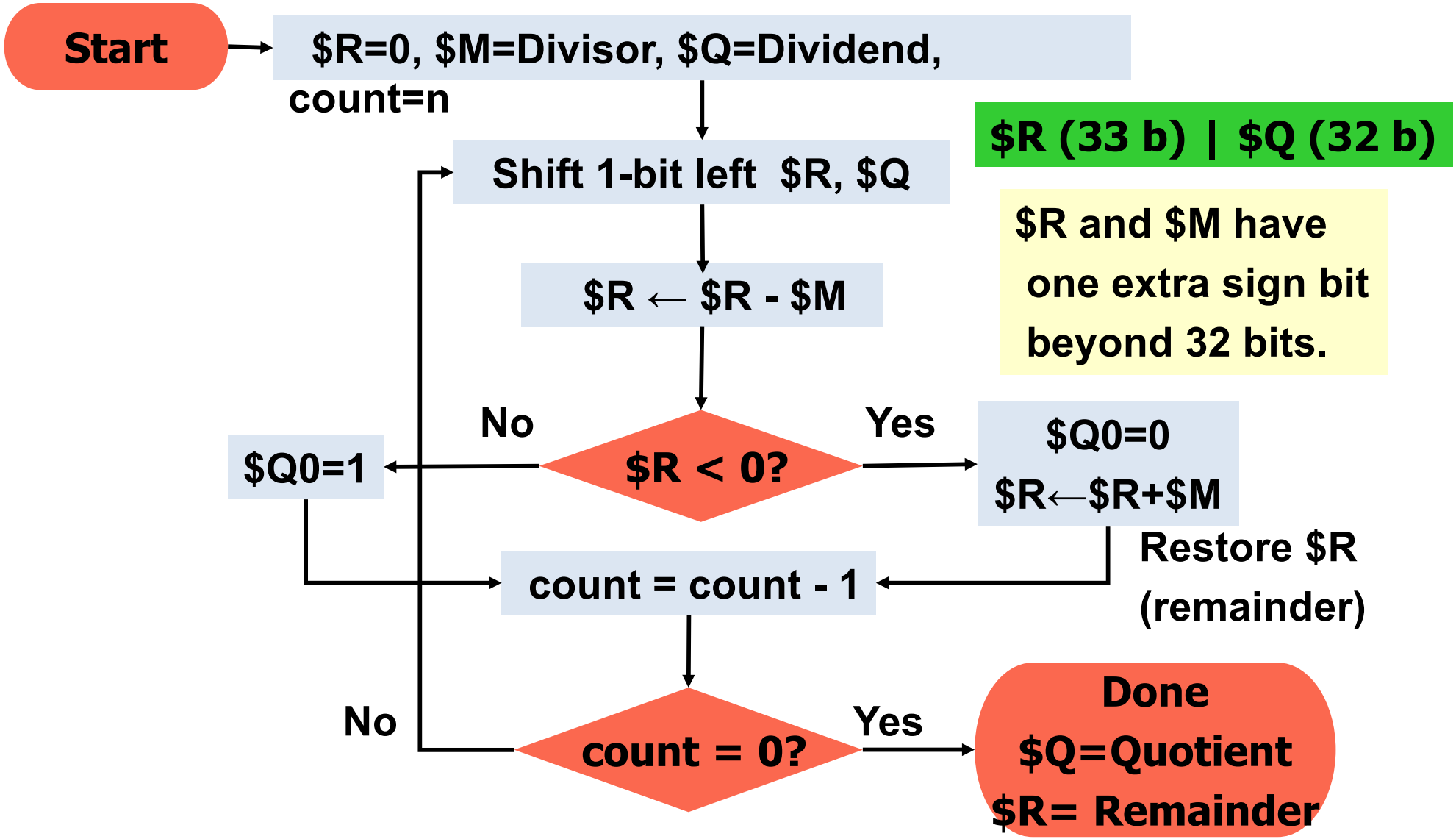
Binary Division (Unsigned)



4-bit Binary Division (Unsigned)



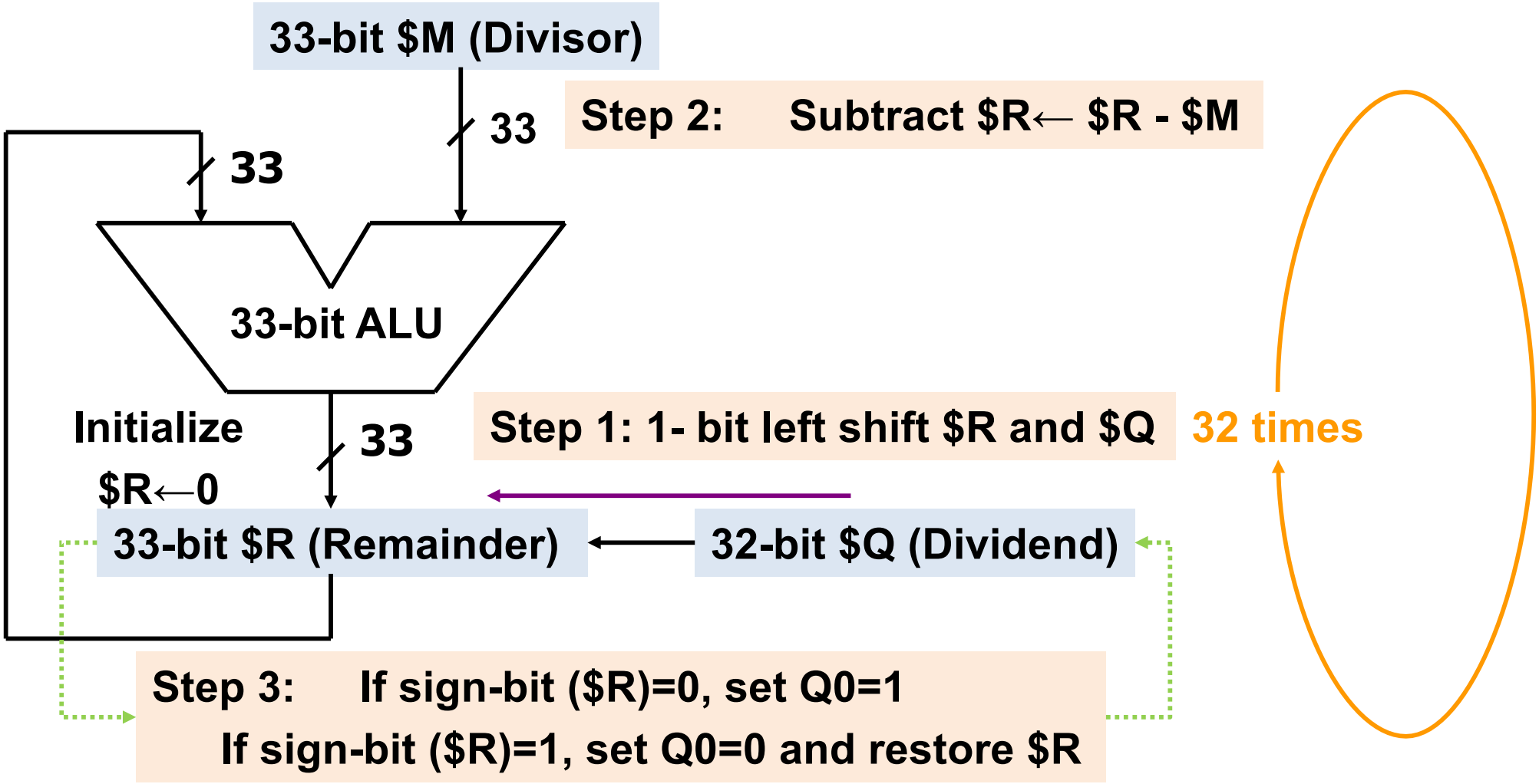
32-bit Binary Division Flowchart



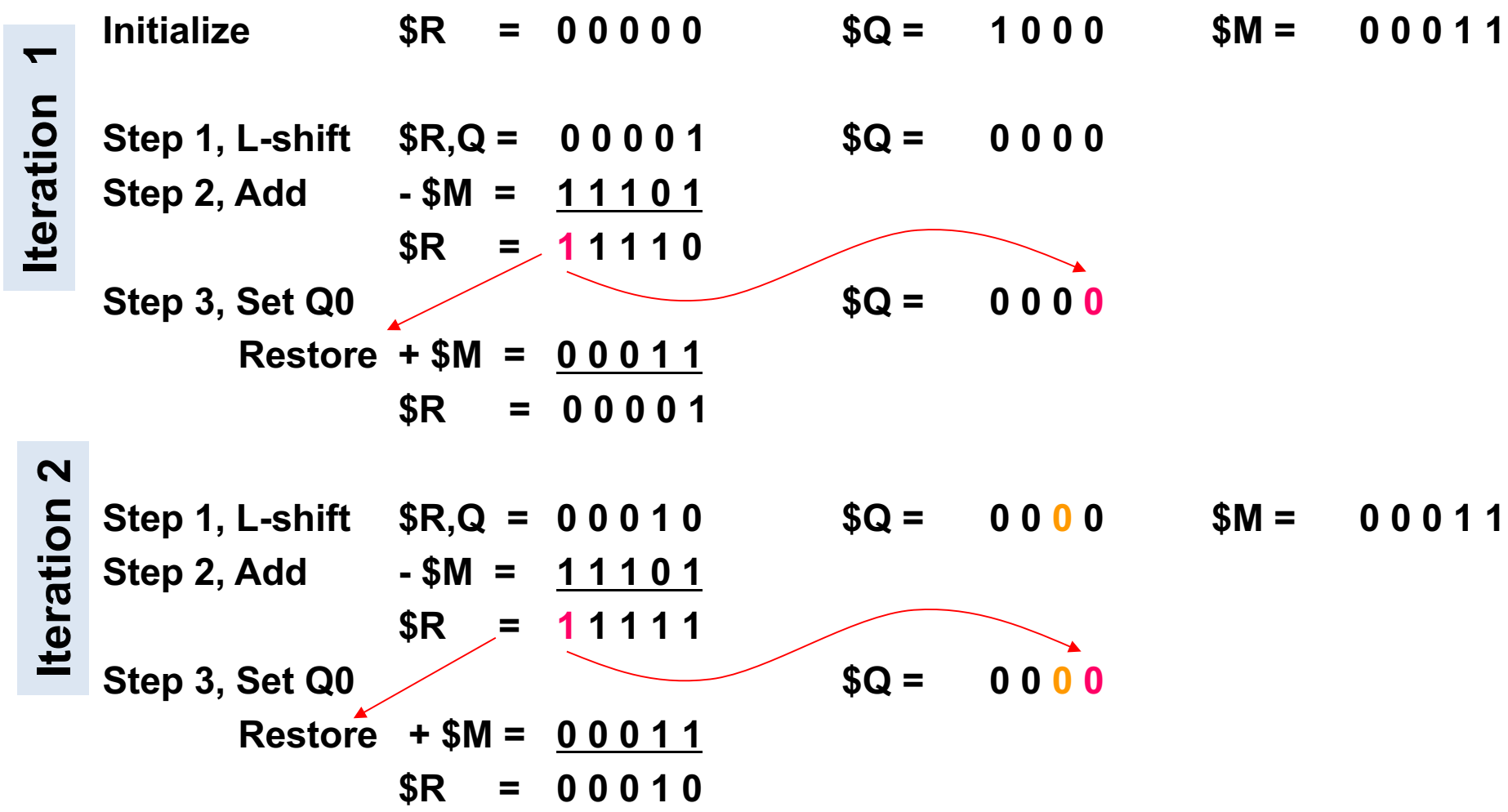
4-bit Example: 6/4 = 1, Remainder 2

count	Actions	n	$R, $Q	$M = Divisor
4	Initialize	4	0 0 0 0 0 | 0 1 1 0	0 0 1 0 0
	Shift left $R, $Q	4	0 0 0 0 0 | 1 1 0 0	0 0 1 0 0
	Add – $M (11100) to $R	4	1 1 1 0 0 | 1 1 0 0	0 0 1 0 0
	Restore, add $M (00100) to $R	3	0 0 0 0 0 | 1 1 0 0	0 0 1 0 0
3	Shift left $R, $Q	3	0 0 0 0 1 | 1 0 0 0	0 0 1 0 0
	Add – $M (11100) to $R	3	1 1 1 0 1 | 1 0 0 0	0 0 1 0 0
	Restore, add $M (00100) to $R	2	0 0 0 0 1 | 1 0 0 0	0 0 1 0 0
2	Shift left $R, $Q	2	0 0 0 1 1 | 0 0 0 0	0 0 1 0 0
	Add – $M (11100) to $R	2	1 1 1 1 1 | 0 0 0 0	0 0 1 0 0
	Restore, add $M (00100) to $R	1	0 0 0 1 1 | 0 0 0 0	0 0 1 0 0
1	Shift left $R, $Q	1	0 0 1 1 0 | 0 0 0 0	0 0 1 0 0
	Add – $M (11100) to $R	1	0 0 0 1 0 | 0 0 0 0	0 0 1 0 0
	Set LSB of $Q = 1	0	0 0 0 1 0 | 0 0 0 1	0 0 1 0 0
0			**Remainder | Quotient**	

Division


33-bit $M (Divisor)
33
Step 2: Subtract $R← $R - $M
33
33-bit ALU
Initialize
$R←0
33
Step 1: 1- bit left shift $R and $Q
32 times
33-bit $R (Remainder)
32-bit $Q (Dividend)
Step 3: If sign-bit ($R)=0, set Q0=1
If sign-bit ($R)=1, set Q0=0 and restore $R

Ex: 8/3 = 2, Remainder = 2



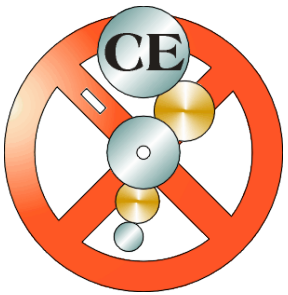
Ex: 8/3 = 2 (Remainder = 2) (Con'd)

	$R = 0 0 0 1 0	$Q = 0 0 0 0	$M = 0 0 0 1 1	
Iteration 3				
Step 1, L-shift	$R ,Q = 0 0 1 0 0	$Q = 0 0 0 0	$M = 0 0 0 1 1	
Step 2, Add	- $M = 1 1 1 0 1			
	$R = 0 0 0 0 1			
Step 3, Set Q0		$Q = 0 0 0 1		
Iteration 4				
Step 1, L-shift	$R,Q = 0 0 0 1 0	$Q = 0 0 1 0	$M = 0 0 0 1 1	
Step 2, Add	- $M = 1 1 1 0 1			
	$R = 1 1 1 1 1			
Step 3, Set Q0		$Q = 0 0 1 0 *Final quotient*		
Restore	+ $M = 0 0 0 1 1			
	$R = 0 0 0 1 0 *Remainder*			

Note "Restore $R" in Steps 1, 2 and 4. This method is known as *the RESTORING DIVISION. An improved method, NON-RESTORING DIVISION, is possible (Hamacher, et al.)*

Signed Division

- Remember the signs and divide magnitudes.
- Negate the quotient if the signs of divisor and dividend disagree.
- There is no other direct division method for signed division.



The Antikythera mechanism (out of Lego):
https://youtu.be/RLPVCJjTNgk

Quiz 11-1

http://m.socrative.com/student/#joinRoom

room number: 713113

- Q1: In binary a multiplication of an unsigned number with 100 (4) can be performed by:
  - Shifting the number to the left by 2 bits
  - Shifting the number to the right by 2 bits
  - Shifting the number to the left by 3 bits
  - Shifting the number to the right by 3 bits

Q2: On a serial multiplier the product register initially contains ____(a) at its upper 32 bits and ___ (b) at its lower 32-bits. At the end of the multiplication it contains ____(c). Pick 3 of the options for (a), (b), (c):

the Product

Zeros

The Multiplicant

The Multiplier

- Q3: On a serial divider the result register initially contains ____(a) at its upper 33 bits and ___ (b) at its lower 32-bits. At the end of the division it contains ____(c) at its upper 33-bits. Pick 3 of the options for (a), (b), (c):
  - the Remainder
  - Zeros
  - The Divisor
  - The Divident

Fixed-point & Floating-point Numbers Accuracy-Resolution

Integer numbers

$$v = \sum_{i=0}^{n-1} a_i 2^i$$

Integer number: $a_{n-1}, a_{n-2}, \ldots a_1, a_0$

Fixed point numbers

format	Number	r	Integer	Value	
1.3	1.011	0.125	11	1.375	(11/8)
s1.3	01.011	0.125	11	1.375	(11/8)
s1.3	11.011	0.125	-5	-0.625	(-5/8)
2.4	10.0111	0.0625	39	2.4375	(39/16)

$$v = r \sum_{i=0}^{n-1} a_i 2^i$$

Fixed point number: $a_{n-1}, a_{n-2}, \ldots a_1, a_0$

Floating point numbers

- Mantissa m: binary fraction
- Exponent e: binary integer

$$v = m \times 2^{e-x} = r \sum_{i=0}^{n-1} m_i 2^{i-n} \times 2^{\sum_{i=0}^{k-1} e_i 2^k - x}$$

E.g.: m= 10010, e=011 (10010E011): v=18/32 x 8

Could also be written 01001E100: v 9/32 x16,

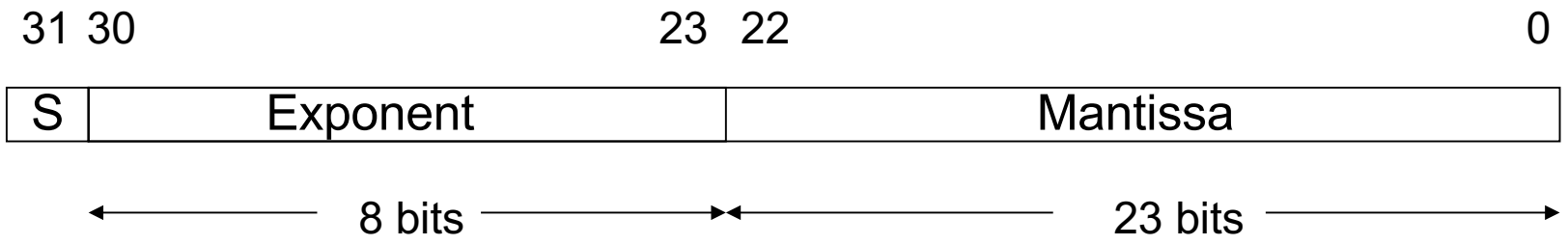
but this is not normalized form of a floating point number

(mantissa should start with 1 unless exponent is 0)

Floating point number: $e_{k-1}, e_{k-2}, \ldots e_1, e_0 \mid m_{n-1}, m_{n-2}, \ldots m_1, m_0$

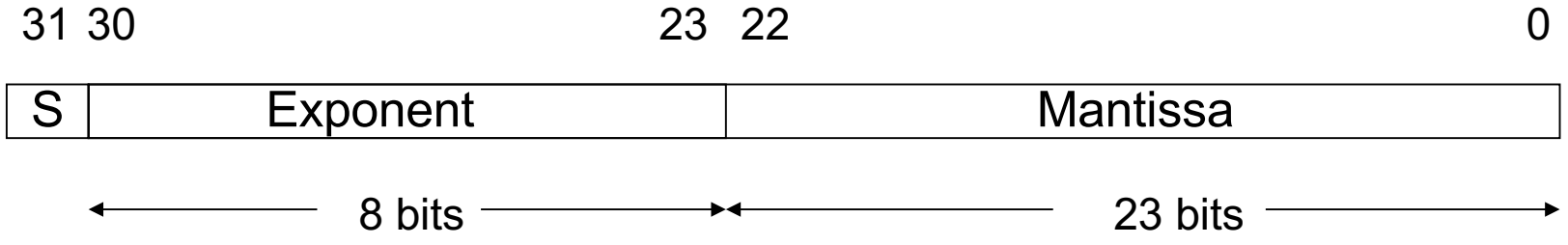
Binary Floating Point

Floating-Point Representation



Floating-Point Format: $(-1)^S \times M \times 2^E$

IEEE 754 Floating-Point Representation



Floating-Point Format: $(-1)^S \times (1 + M) \times 2^{(E-Bias)}$

Value and Representation functions

- R(x) is binary number representing real number x
- V(b) is value of binary number b

- Absolute error
  - $e_a = |V(R(x)) - x|$

- Relative error
  - $e_r = |(V(R(x)) - x)/x|$

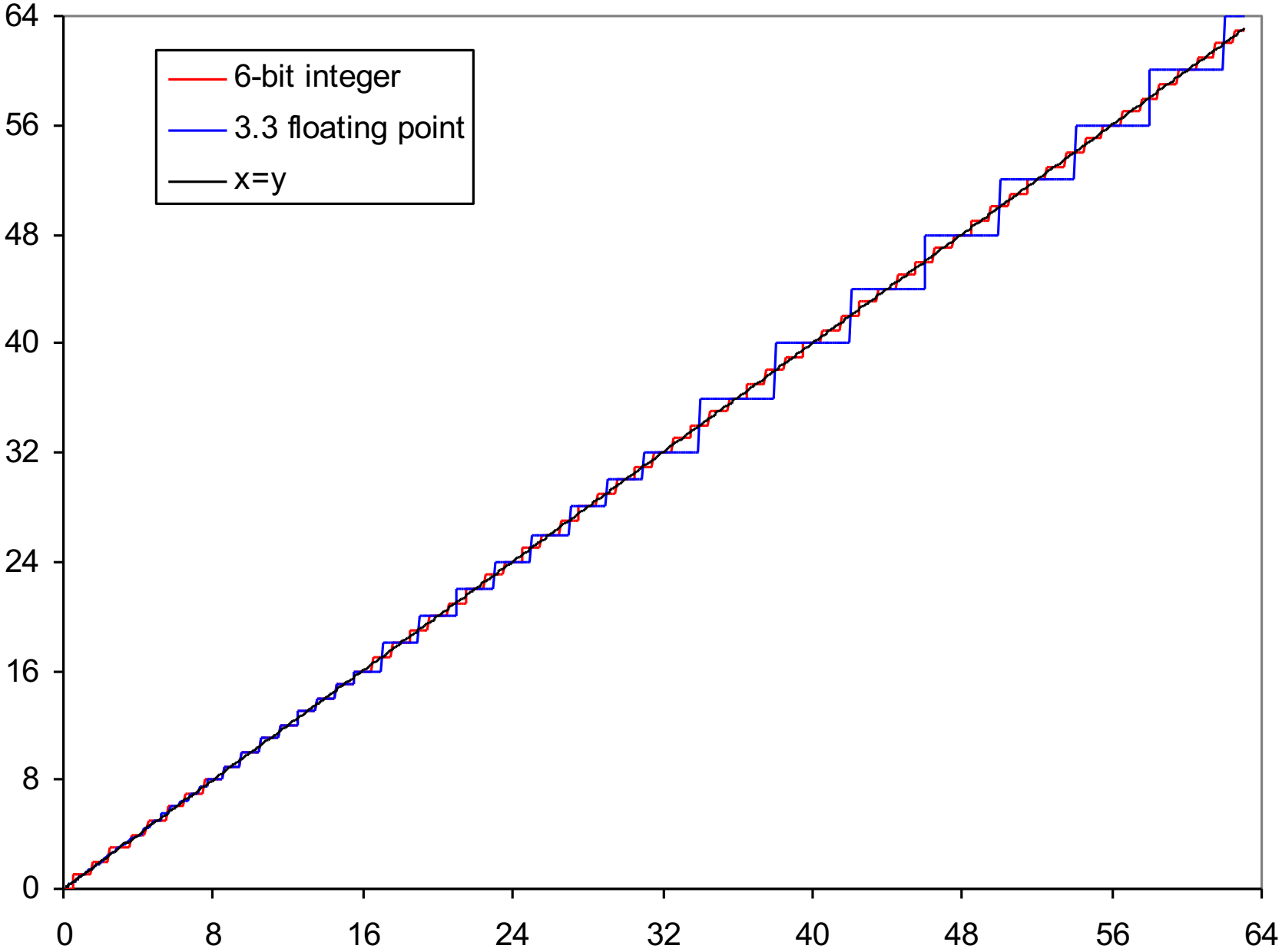
Example

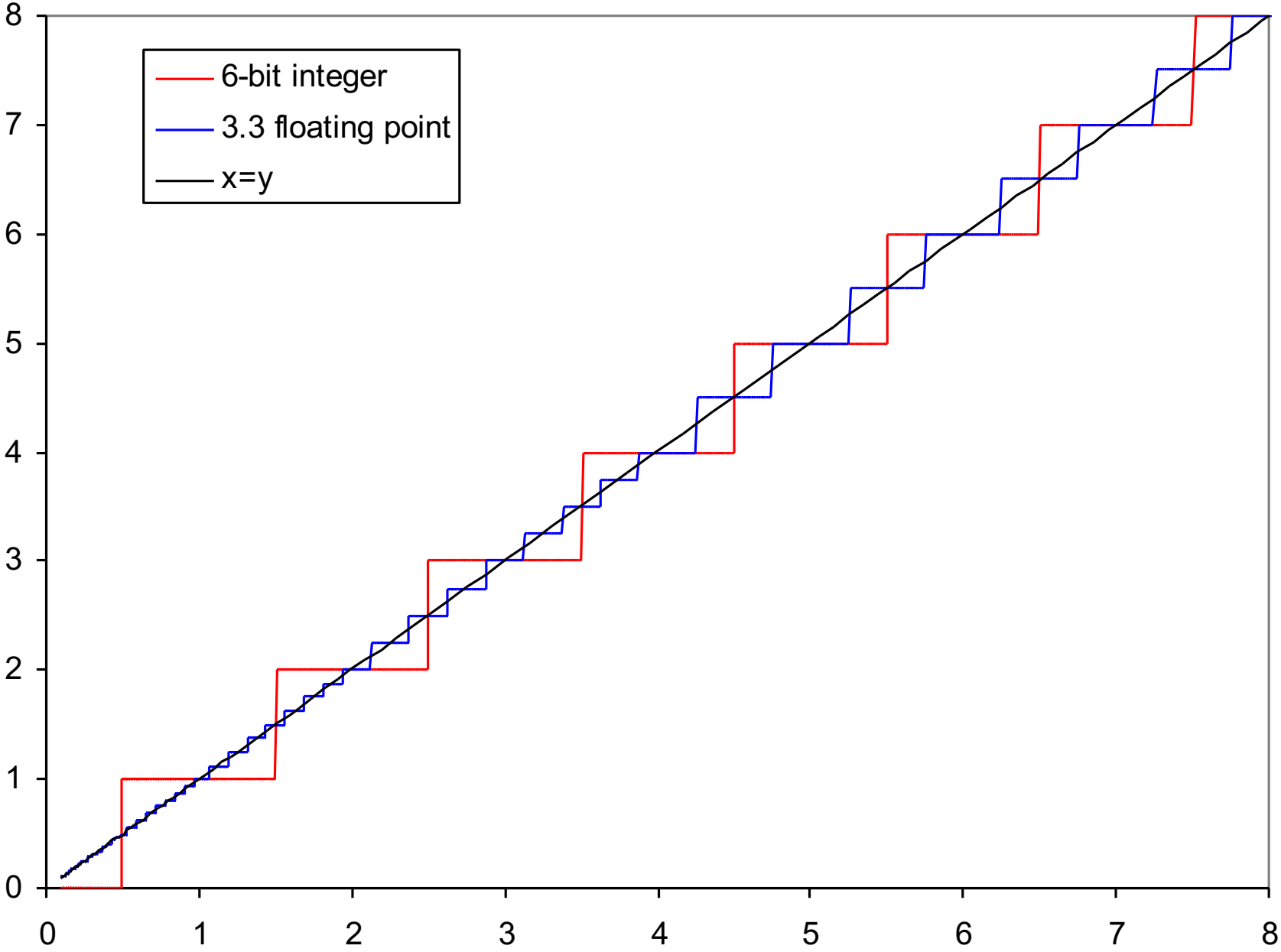
- Suppose you need to represent temperatures from 0 to 100C to 0.1C
- What representation would you use?
  - What is the resolution of this representation?
  - What is its maximum absolute error?

Example

- Suppose you need to represent distances from 1mm to 1m with an accuracy of 0.1%
- How many bits are required to do this with a binary fixed-point number?
- How many bits are needed with binary floating point?
- What floating-point representation is needed?

				6.4 FP				
Value	10-bit intege	Error	%	Exp	Mantissa	Value	Error	%
477.49	0111011101	0.49	0.10%	1110	1110111	476	1.49	0.31%
47.749	0000110000	0.251	0.53%	1011	1011111	47.5	0.249	0.52%
4.7749	0000000101	0.2251	4.71%	1000	1001100	4.75	0.0249	0.52%
0.4775	0000000000	0.47749	100.00%	0100	1111010	0.476563	0.0009275	0.19%
0.0477	0000000000	0.047749	100.00%	0001	1100010	0.047852	0.000102562	0.21%
	Shift mantissa by exp-12							
	Binary point is shifted by exp-5 from left of implied one							


6-bit integer
3.3 floating point
x=y
64
56
48
40
32
24
16
8
0
0
8
16
24
32
40
48
56
64


6-bit integer
3.3 floating point
x=y
0
1
2
3
4
5
6
7
8

Summary of Lecture 12

- Arithmetic units
  - Multipliers
  - Dividers
- Number representation
  - accuracy and resolution
  - Fixed and floating point

- Book (complimentary to the slides):
  - 10.4,10.5, 11-11.3.1

- Next Lecture 13:
  - Guest Lecture:

  Jan Andersson, coding in VHDL in COBHAM Gaisler AB