معماری کامپیوتر

فصل سه

مدارهای محاسباتی

Computer Architecture

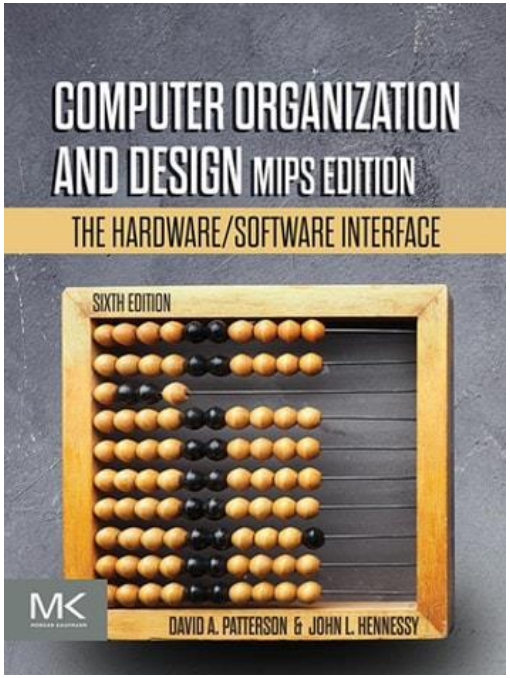
Chapter Three

Arithmetic Circuits

Copyright Notice

The slides of this lecture are adopted from:

- D. Patterson & J. Hennessey, “Computer Organization & Design, The Hardware/Software Interface”, 6th Ed., MK publishing, 2020

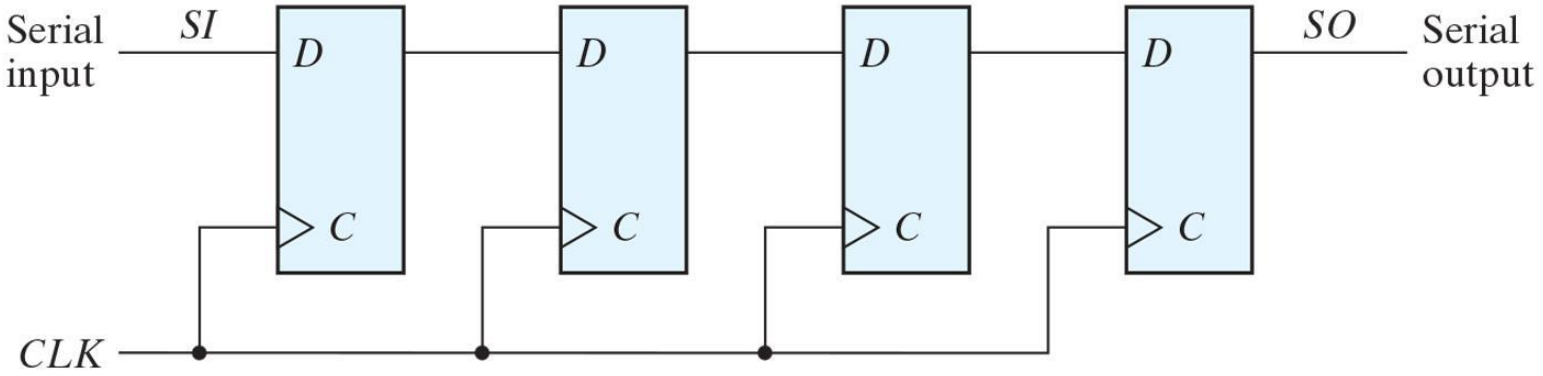


Outlines

- Shift
- Addition/Subtraction
  - Ripple-Carry Adders
  - Carry Look-ahead Adders
  - Carry-Select Adders
- Multiplication
  - Shift-Add Multiplier
  - Combinational Multiplier
  - Carry-Save Adder Multiplier
- Division
- Floating Point
  - Representation
  - Arithmetic

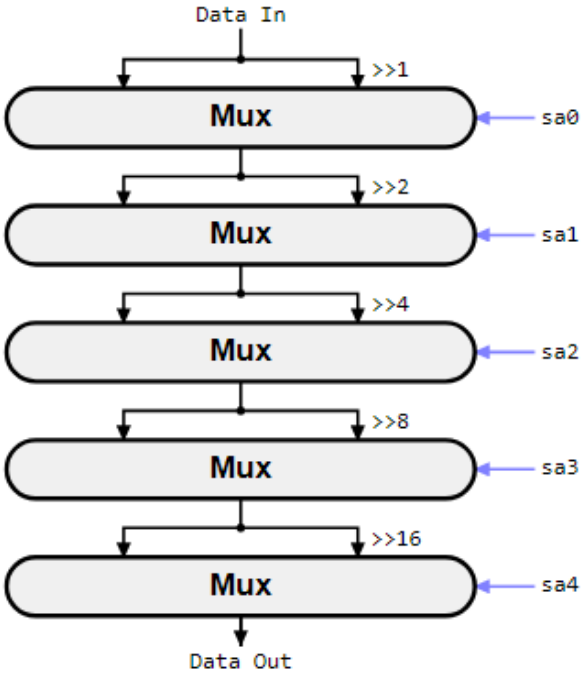
Shift Registers

Sequential Circuits



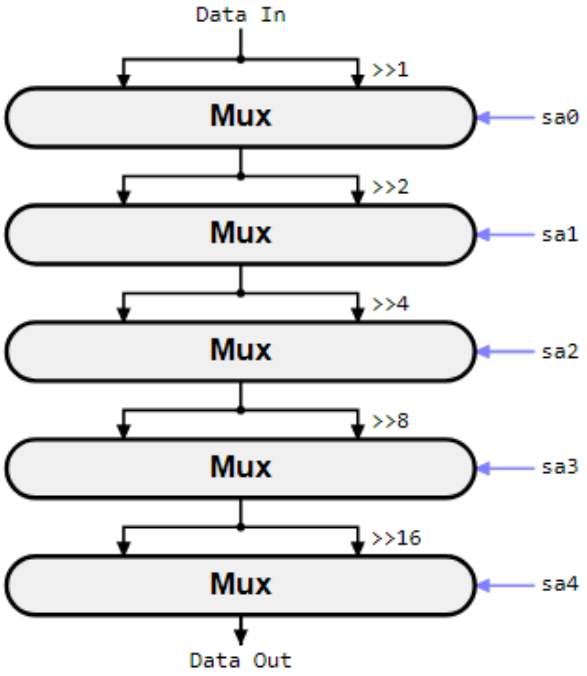
Barrel Shifter

Combinational Circuits

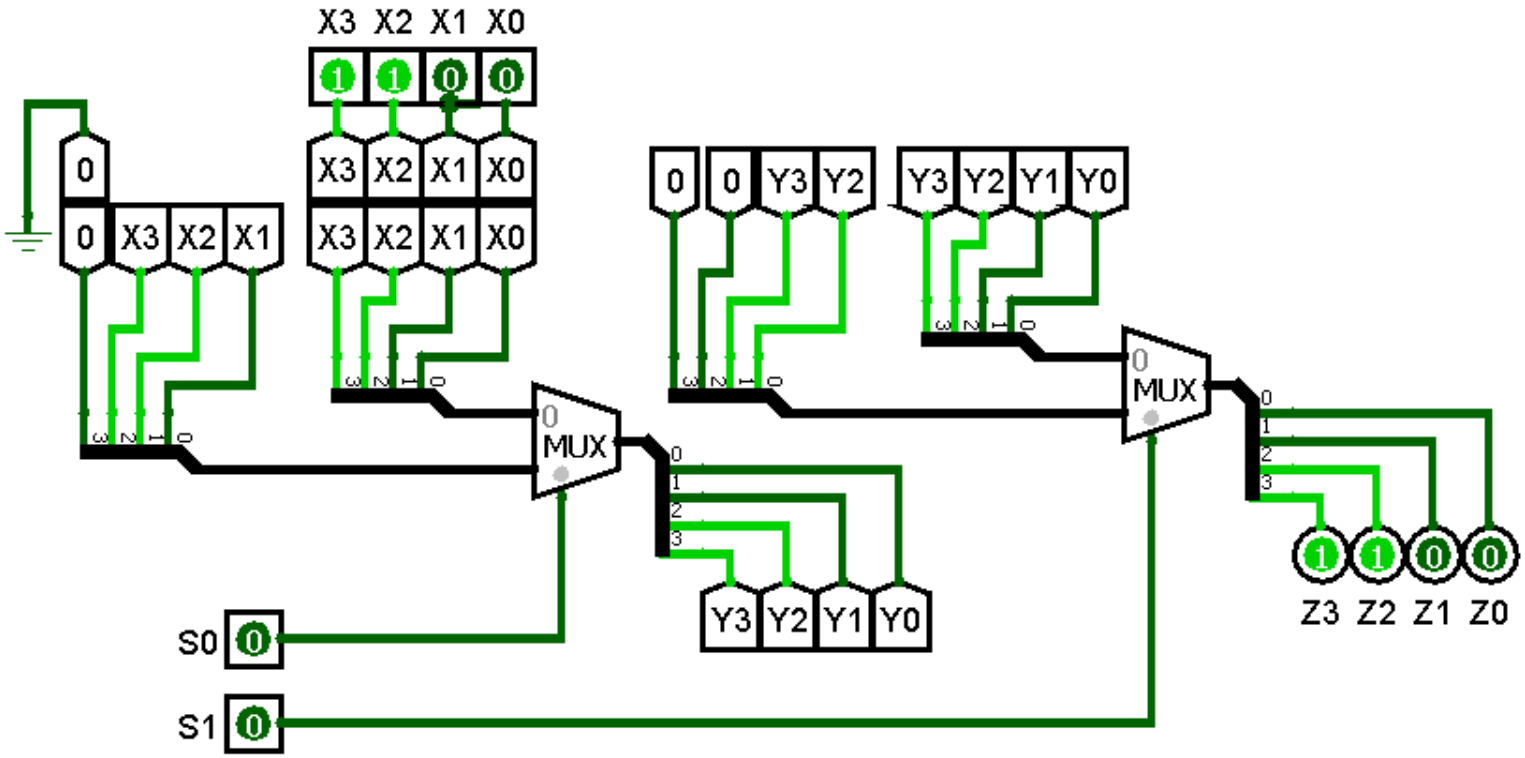


Barrel Shifter

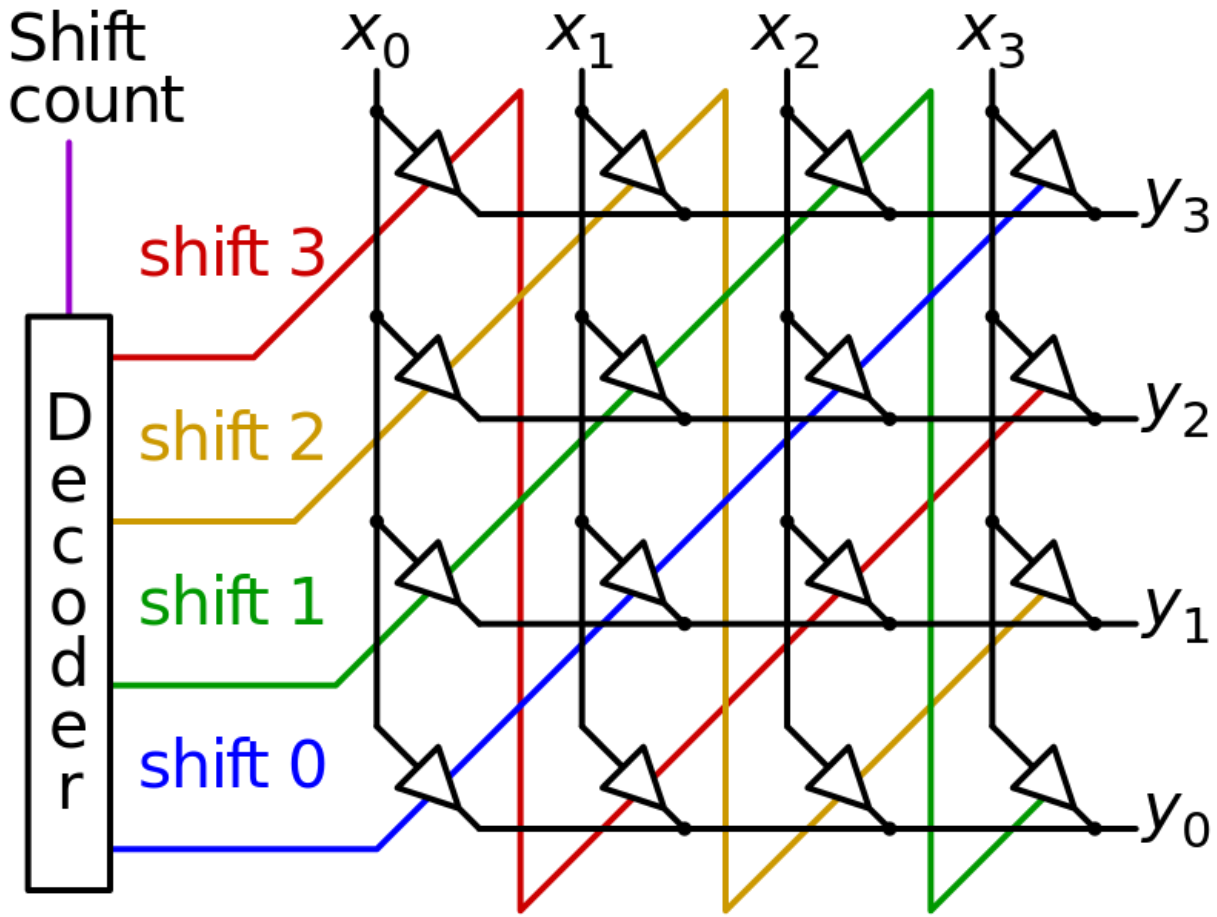
- Shift an optional no of bits
- With a sequence of multiplexers
  - each shifting a word by 2^k bit positions for different values of k



Mux Implementation



Crossbar (Circular) Barrel Shifter



Outlines

- Shift
- Addition/Subtraction
  - Ripple-Carry Adders
  - Carry Look-ahead Adders
  - Carry-Select Adders
- Multiplication
  - Shift-Add Multiplier
  - Combinational Multiplier
  - Carry-Save Adder Multiplier
- Division
- Floating Point
  - Representation
  - Arithmetic

Integer Addition / Subtraction

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 +        64
  0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0      + 42
-------------------------------------------------
  0 0 0 0 0 0 0 0 0 0 1 1 0 1 0 1 0        106

1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 +        64
  1 1 1 1 1 1 1 1 1 1 1 0 1 0 1 1 0      - 42
-------------------------------------------------
  0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 0        22
```

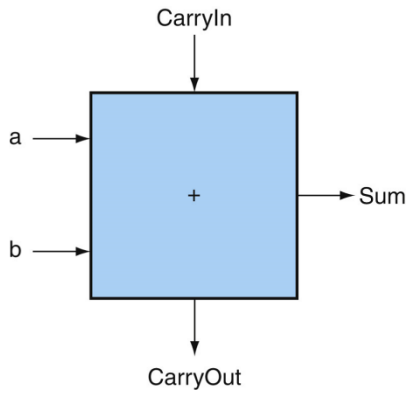
2's complement

Overflow Conditions for Add/Sub

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

- While adding signed numbers, an overflow occurs when
  - Both operands have the same sign,
  - but the result has the opposite sign
- the carry into and out of the MSB differ

One-bit Full Adder



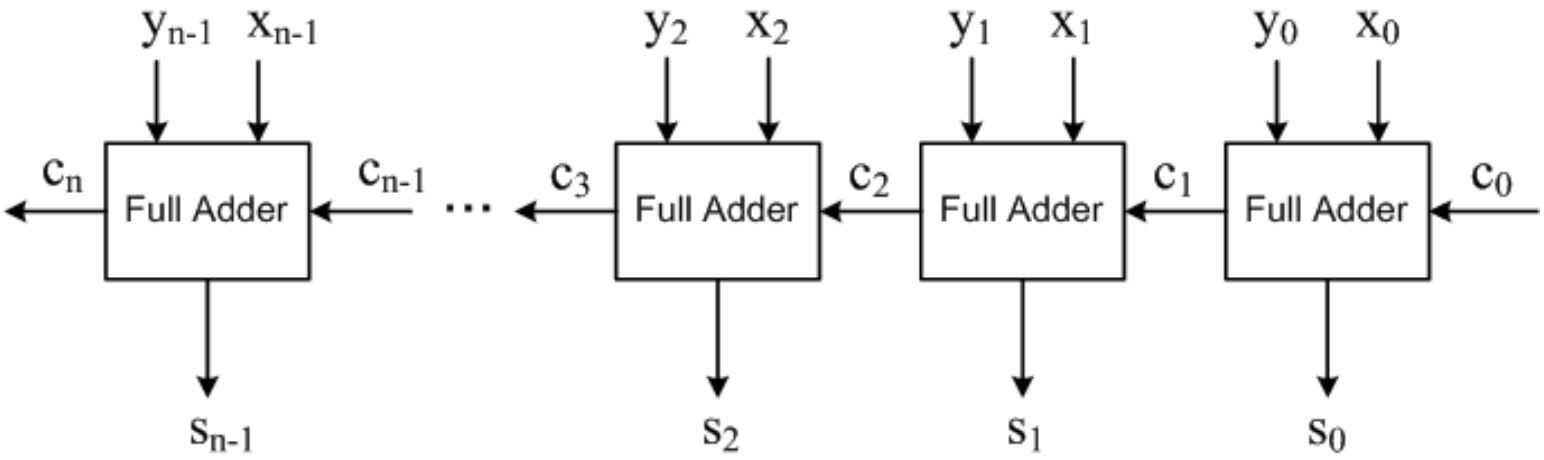
Input and output specification for a 1-bit adder

Inputs			Outputs		
a	b	CarryIn	CarryOut	Sum	Comments
0	0	0	0	0	$0 + 0 + 0 = 00_{two}$
0	0	1	0	1	$0 + 0 + 1 = 01_{two}$
0	1	0	0	1	$0 + 1 + 0 = 01_{two}$
0	1	1	1	0	$0 + 1 + 1 = 10_{two}$
1	0	0	0	1	$1 + 0 + 0 = 01_{two}$
1	0	1	1	0	$1 + 0 + 1 = 10_{two}$
1	1	0	1	0	$1 + 1 + 0 = 10_{two}$
1	1	1	1	1	$1 + 1 + 1 = 11_{two}$

$$Sum = a \oplus b \oplus CarryIn$$

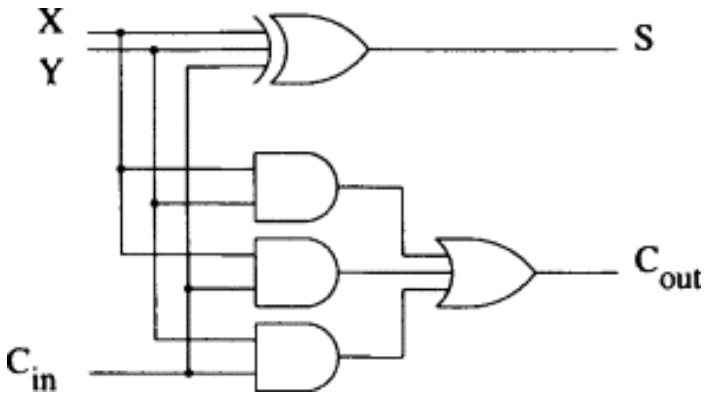
$$CarryOut = a.b + a.CarryIn + b.CarryIn$$

Ripple-Carry Adder



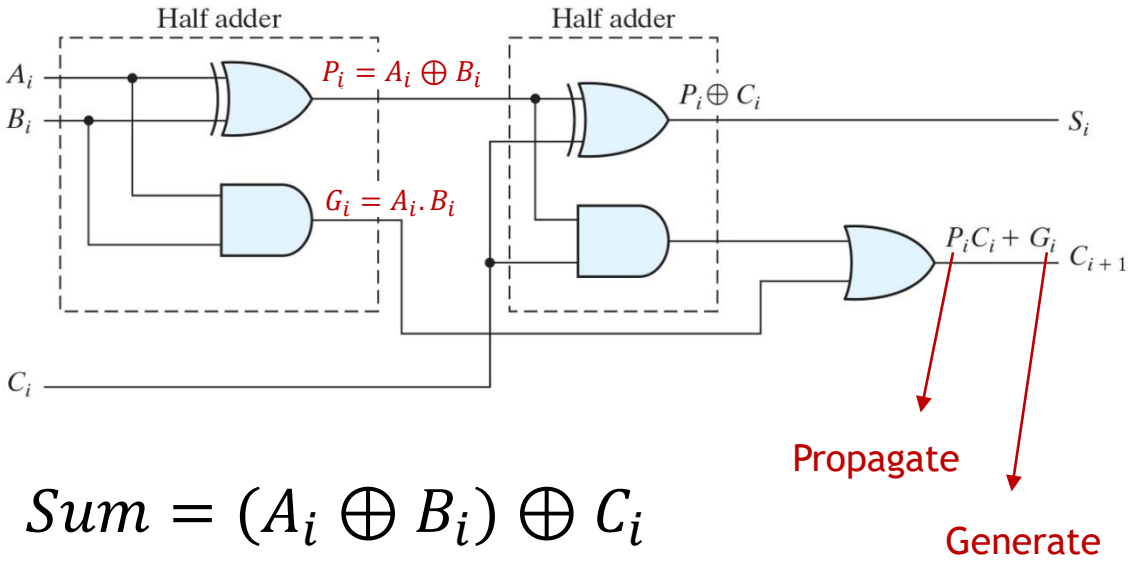
Analysis (n-bit Ripple-Carry Adder)

- Cost: O(n)
  - n × F.A.
- Critical Path Delay: O(n)
  - $n \times D_{FA} = 2n \times D_{gate}$



Carry Look-Ahead Adders

Is there any way to calculate the final carry-out in just two levels of logic?



$$Sum = (A_i \oplus B_i) \oplus C_i$$

$$C_{i+1} = A_i B_i + C_i (A_i \oplus B_i)$$

Carry Look-Ahead Adders

Is there any way to calculate the final carry-out in just two levels of logic?

$$c_1 = g_0 + p_0.c_0$$

$$c_2 = g_1 + p_1.g_0 + p_1.p_0.c_0$$

$$c_3 = g_2 + p_2.g_1 + p_2.p_1.g_0 + p_2.p_1.p_0.c_0$$

$$c_4 = g_3 + p_3.g_2 + p_3.p_2.g_1 + p_3.p_2.p_1.g_0 + p_3.p_2.p_1.p_0.c_0$$

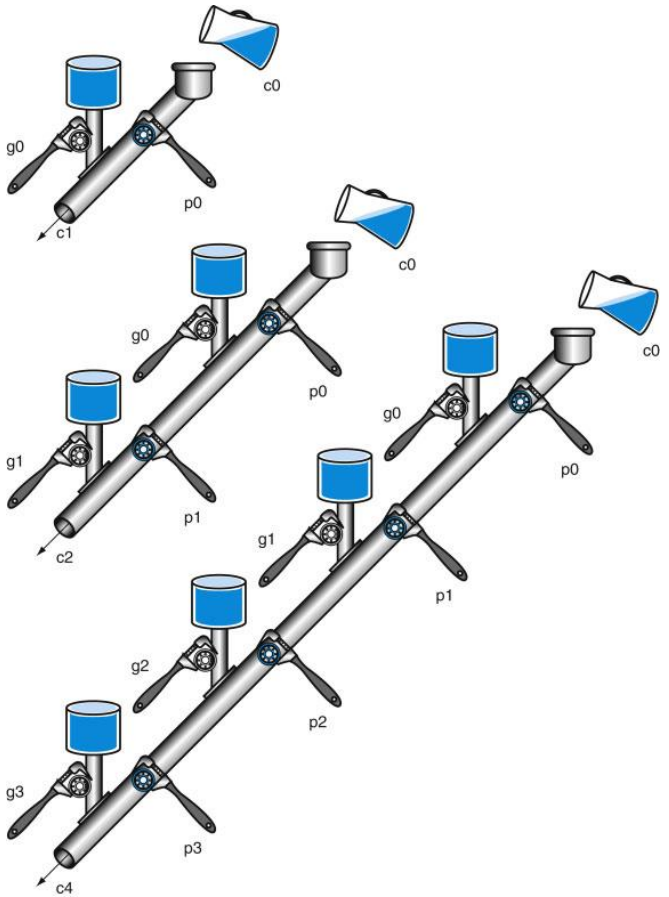
1st Level of Abstraction

$$c_1 = g_0 + p_0 . c_0$$

$$c_2 = g_1 + p_1 . g_0 + p_1 . p_0 . c_0$$

$$c_3 = g_2 + p_2 . g_1 + p_2 . p_1 . g_0 + p_2 . p_1 . p_0 . c_0$$

$$c_4 = g_3 + p_3 . g_2 + p_3 . p_2 . g_1 + p_3 . p_2 . p_1 . g_0 + p_3 . p_2 . p_1 . p_0 . c_0$$



2-Level Abstraction

$$c_4 = (g_3 + p_3.g_2 + p_3.p_2.g_1 + p_3.p_2.p_1.g_0) + (p_3.p_2.p_1.p_0).c_0$$

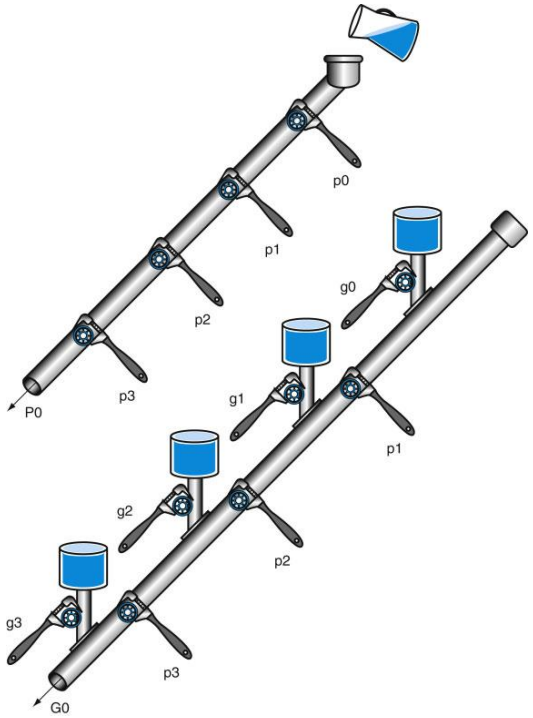
c_4 → C1; $(g_3 + p_3.g_2 + p_3.p_2.g_1 + p_3.p_2.p_1.g_0)$ → G0; $(p_3.p_2.p_1.p_0)$ → P0; c_0 → C0

$$C1 = G0 + P0.C0$$

$$C2 = G1 + P1.C1$$

$$C3 = G2 + P2.C2$$

$$C4 = G3 + P3.C3$$



2-Level Abstraction

$C1 = G0 + P0.C0$

$C2 = G1 + P1.G0 + P1.P0.C0$

$C3 = G2 + P2.G1 + P2.P1.G0 + P2.P1.P0.C0$

$C4 = G3 + P3.G2 + P3.P2.G1 + P3.P2.P1.G0 + P3.P2.P1.P0.C0$

$P0 = p_3.p_2.p_1.p_0$

$P1 = p_7.p_6.p_5.p_4$

$P2 = p_{11}.p_{10}.p_9.p_8$

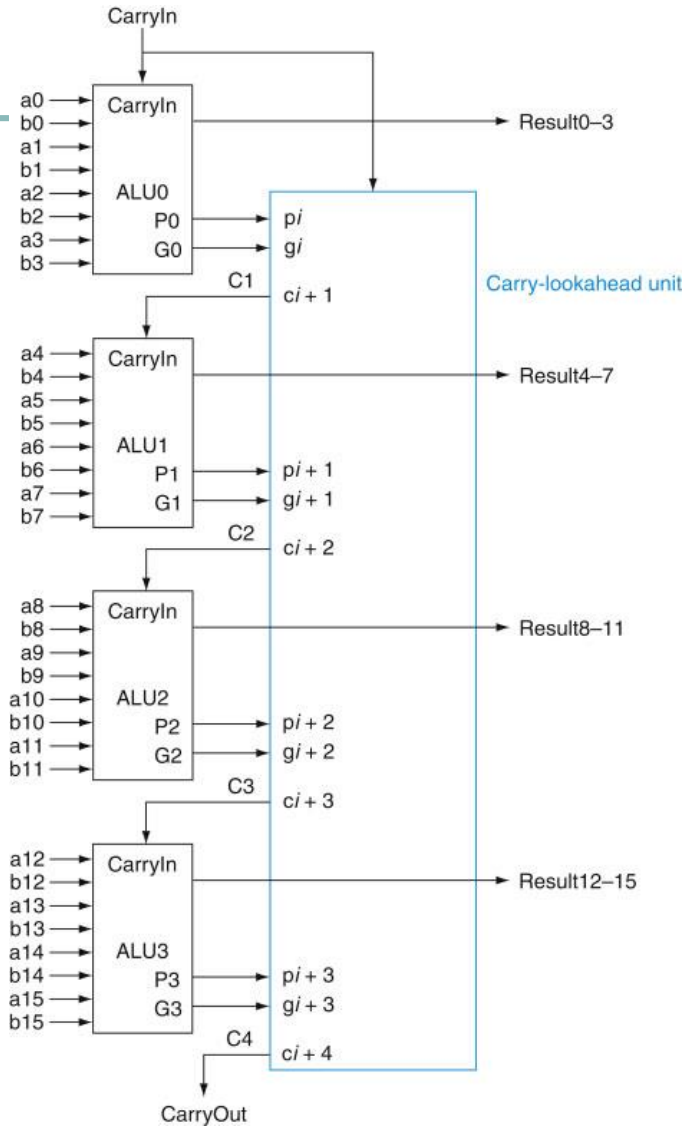
$P3 = p_{15}.p_{14}.p_{13}.p_{12}$

$G0 = g_3+p_3.g_2+p_3.p_2.g_1+p_3.p_2.p_1.g_0$

$G1 = g_7+p_7.g_6+p_7.p_6.g_5+p_7.p_6.p_5.g_4$

$G2 = g_{11}+p_{11}.g_{10}+p_{11}.p_{10}.g_9+p_{11}.p_{10}.p_9.g_8$

$G3 = g_{15}+p_{15}.g_{14}+p_{15}.p_{14}.g_{13}+p_{15}.p_{14}.p_{13}.g_{12}$



Analysis (16-bit 2-level CLA)

- Cost:
  - $O(n^2)$
- Critical Path Delay:
  - 1 D_{gate} for $a_i, b_i \rightarrow g_i, p_i$
  - 2 D_{gate} for $g_i, p_i \rightarrow G_i, P_i$
  - 2 D_{gate} for $G_i, P_i \rightarrow C_i$
  - 2 D_{gate} for $C_i \rightarrow c_i$
  - 1 D_{gate} for $c_i \rightarrow S_i$

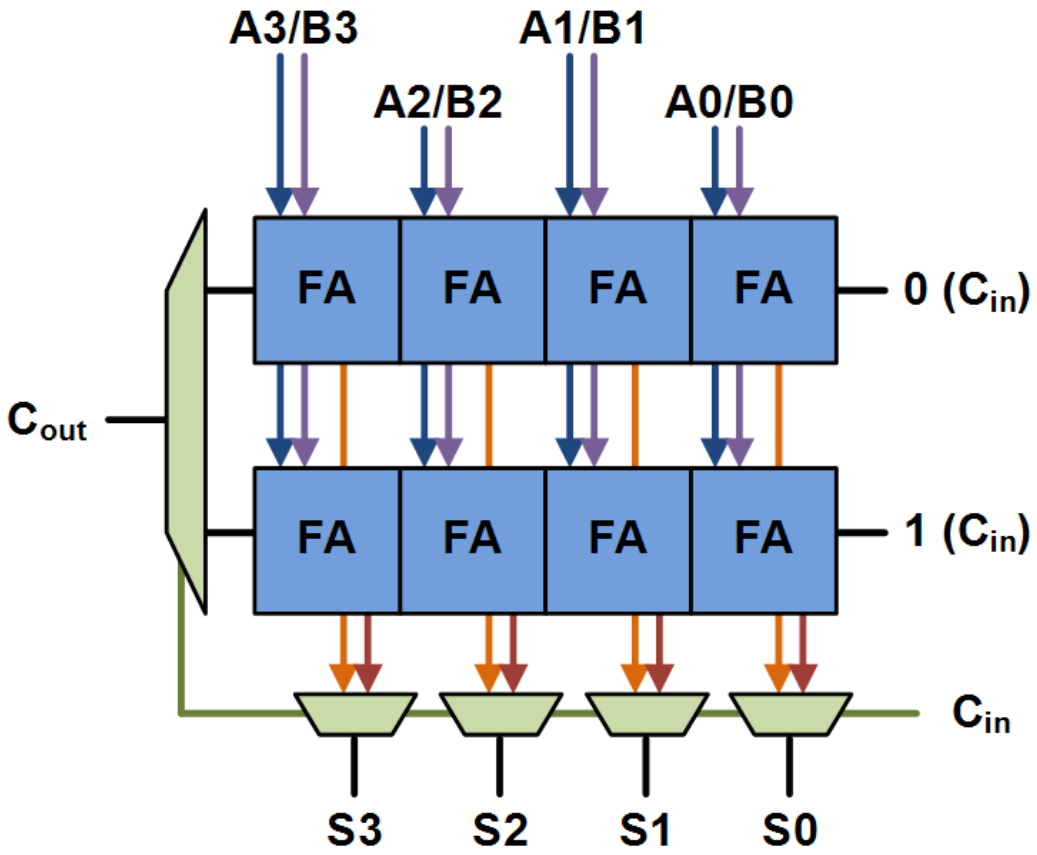
Check Yourself

- What is the relative performance of a ripple carry 8-bit add vs. a 64-bit add using carry look-ahead logic?
  - A 64-bit carry-lookahead adder is faster
  - They are about the same speed, since 64-bit adds need more levels of logic in the 16-bit adder
  - 8-bit adds are faster than 64 bits, even with carry look-ahead

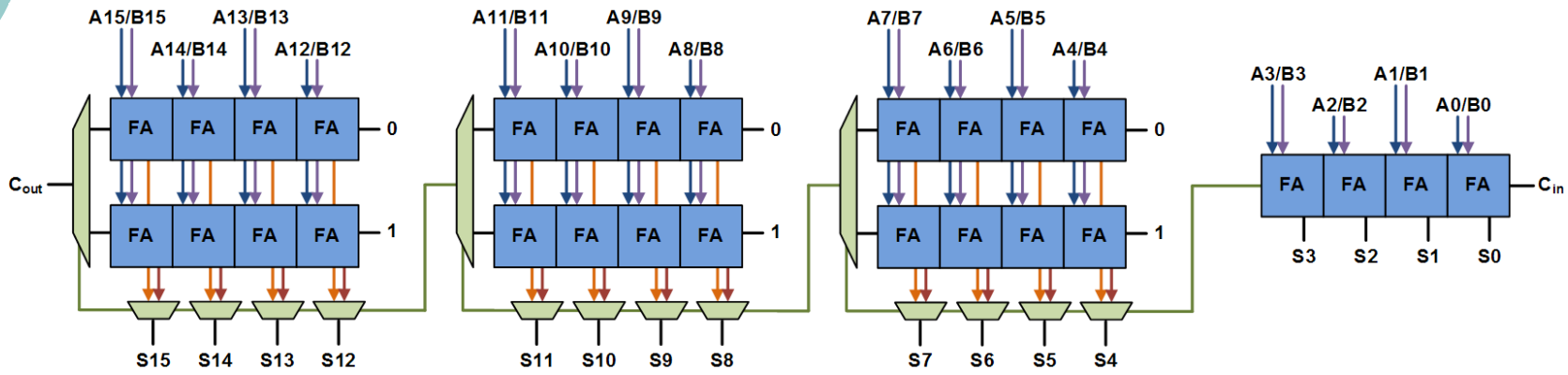
Carry-Select Adders

- K stages of n/K bit ripple carry adders can be used to add n-bit numbers
- The numbers are added twice in each stage
  - One time assuming carry-in is zero
  - Other time assuming the carry-in is one
- The correct sum and carry-out is selected by a multiplexer, when the correct carry-in is known

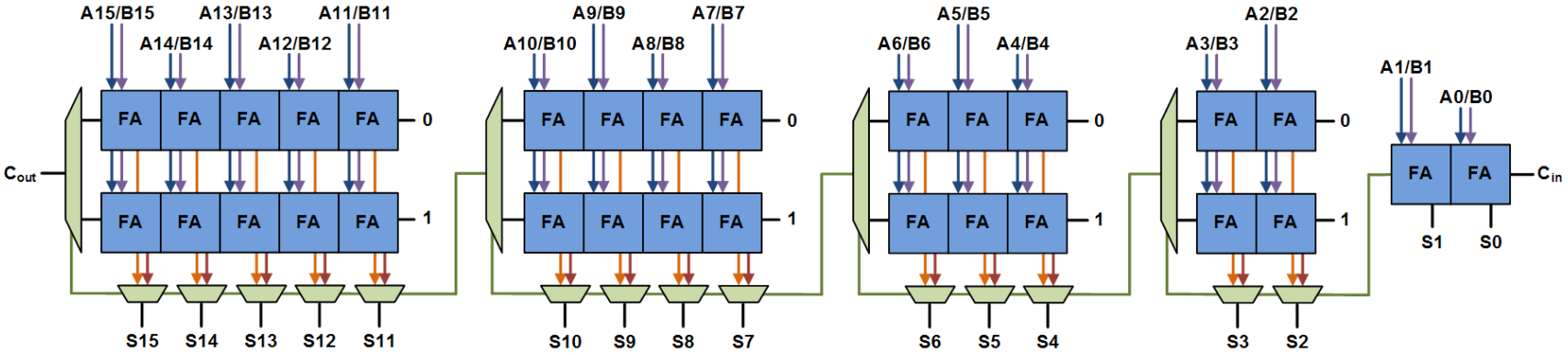
Building Block



Uniform-Sized Adder

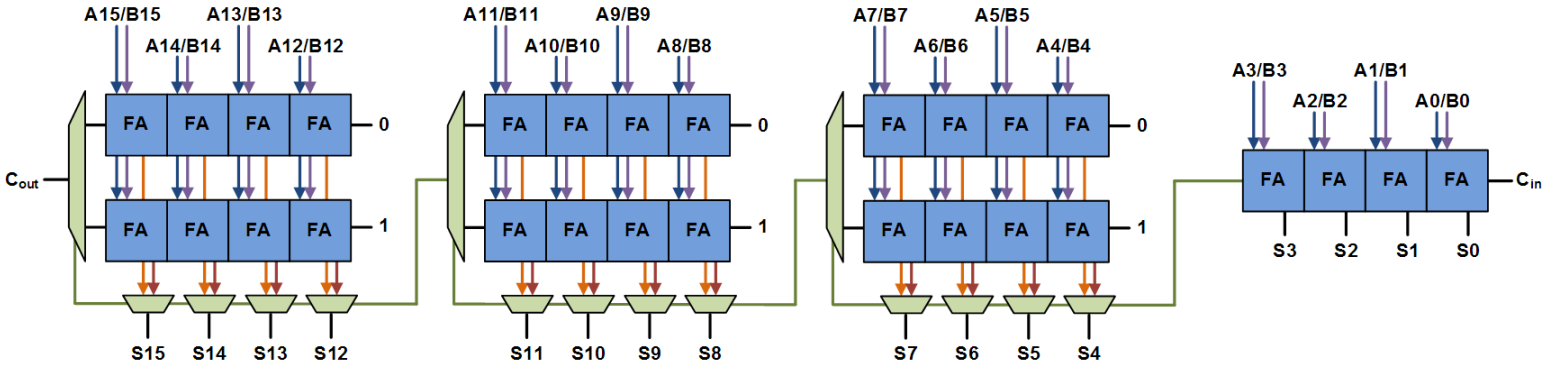


Variable-Sized Adder



Analysis (n-bit k-stage CSA)

- Cost:
  - Full Adders: $n/K \times (2K-1)$
  - Mux: $(K-1)(n/K+1)$
- Critical Path Delay:
  - $n/K \times D_{FA} + (K-1) \times D_{MUX}$



Outlines

- Shift
- Addition/Subtraction
  - Ripple-Carry Adders
  - Carry Look-ahead Adders
  - Carry-Select Adders
- Multiplication
  - Shift-Add Multiplier
  - Combinational Multiplier
  - Carry-Save Adder Multiplier
- Division
- Floating Point
  - Representation
  - Arithmetic

Multiplication Approach (1st ver)

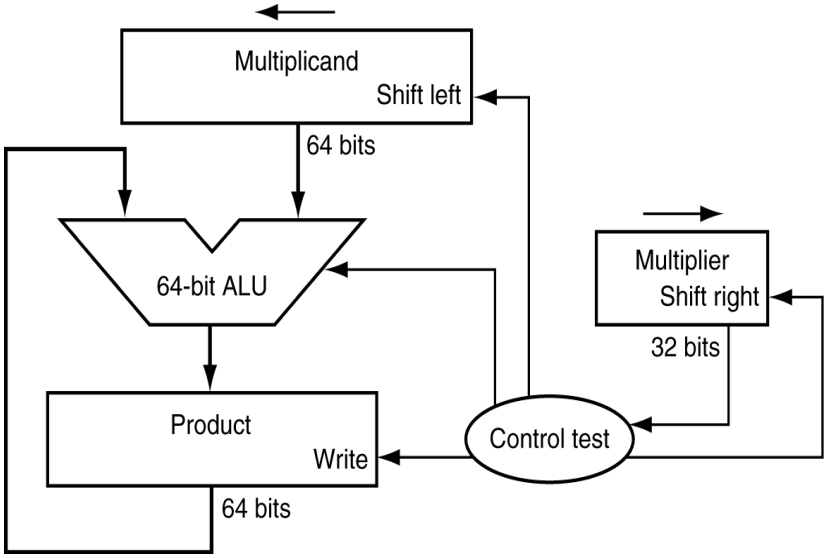
مضروب (multiplicand)

مضروب‌فیه (multiplier)

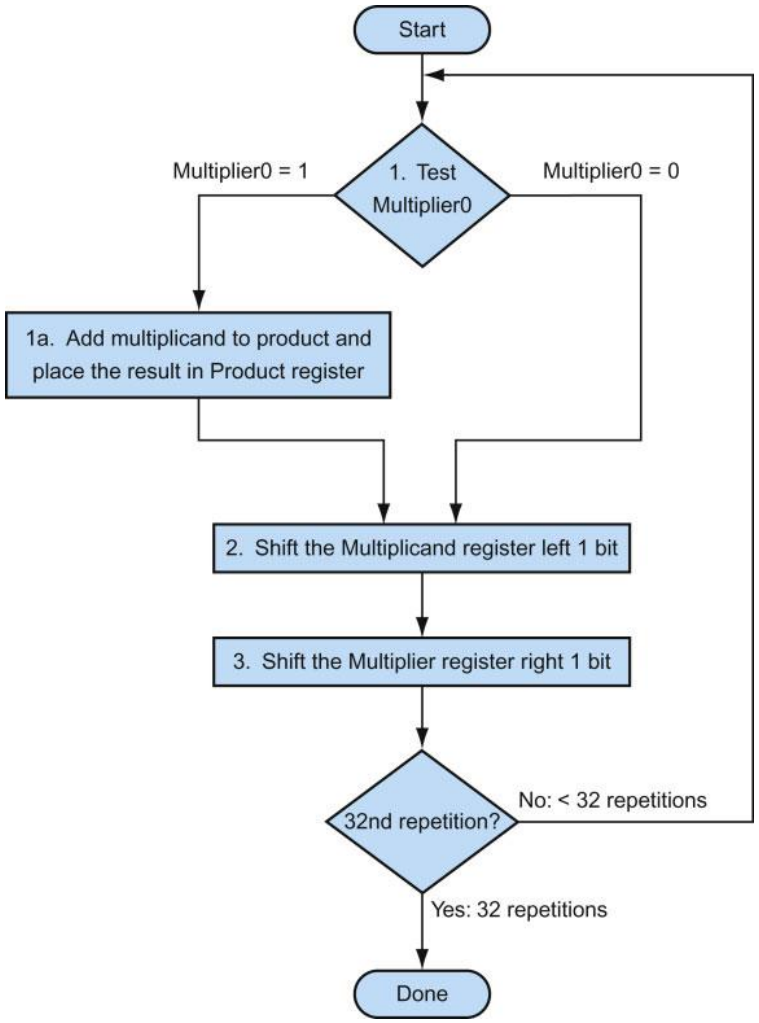
حاصل‌ضرب (product)

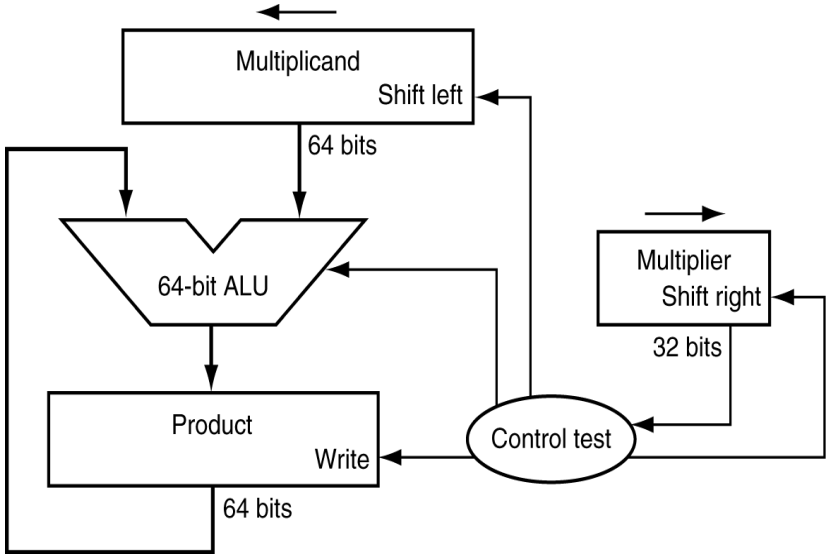
```
1000 ×
1001
------
1000 +
0000
0000
1000
-------
1001000
```

Length of product is the sum of operand lengths

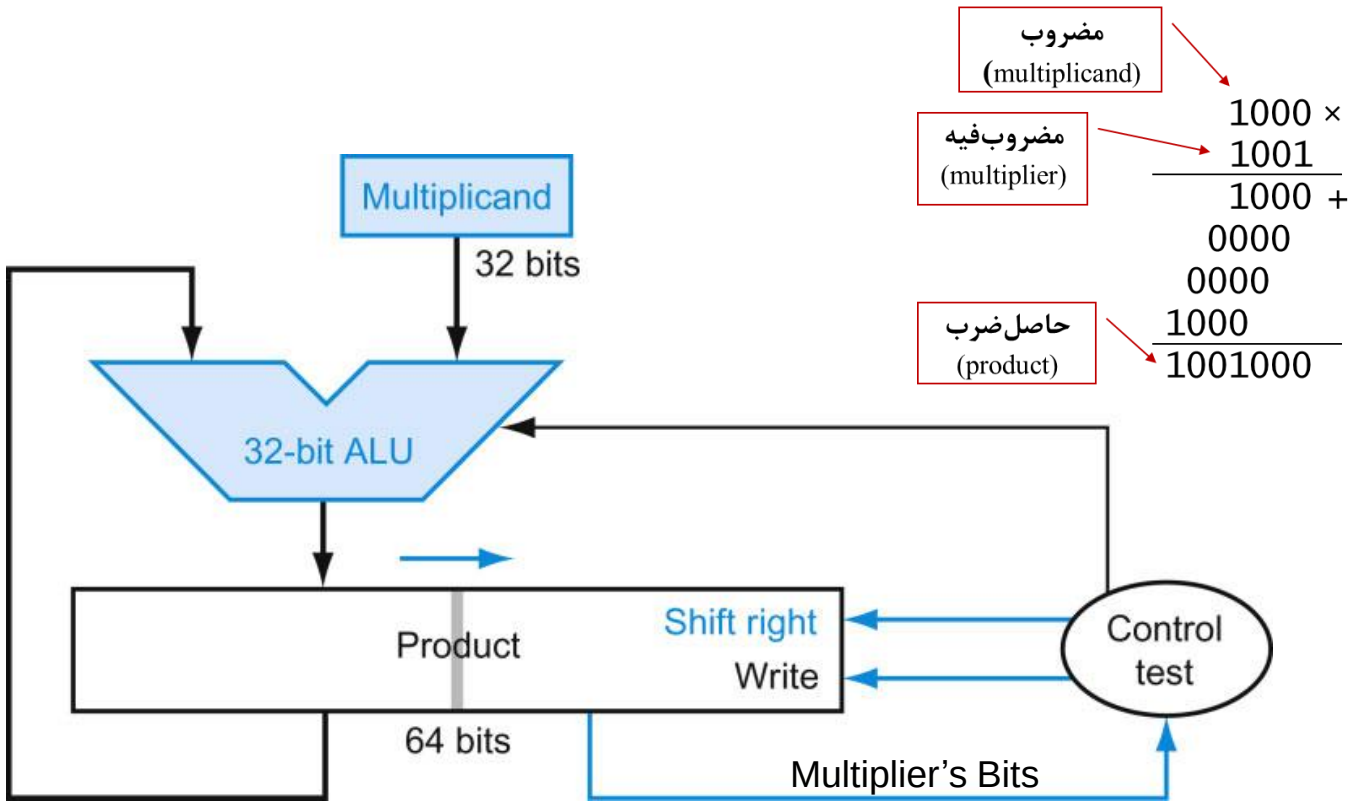


Multiplication Algorithm (1st ver)



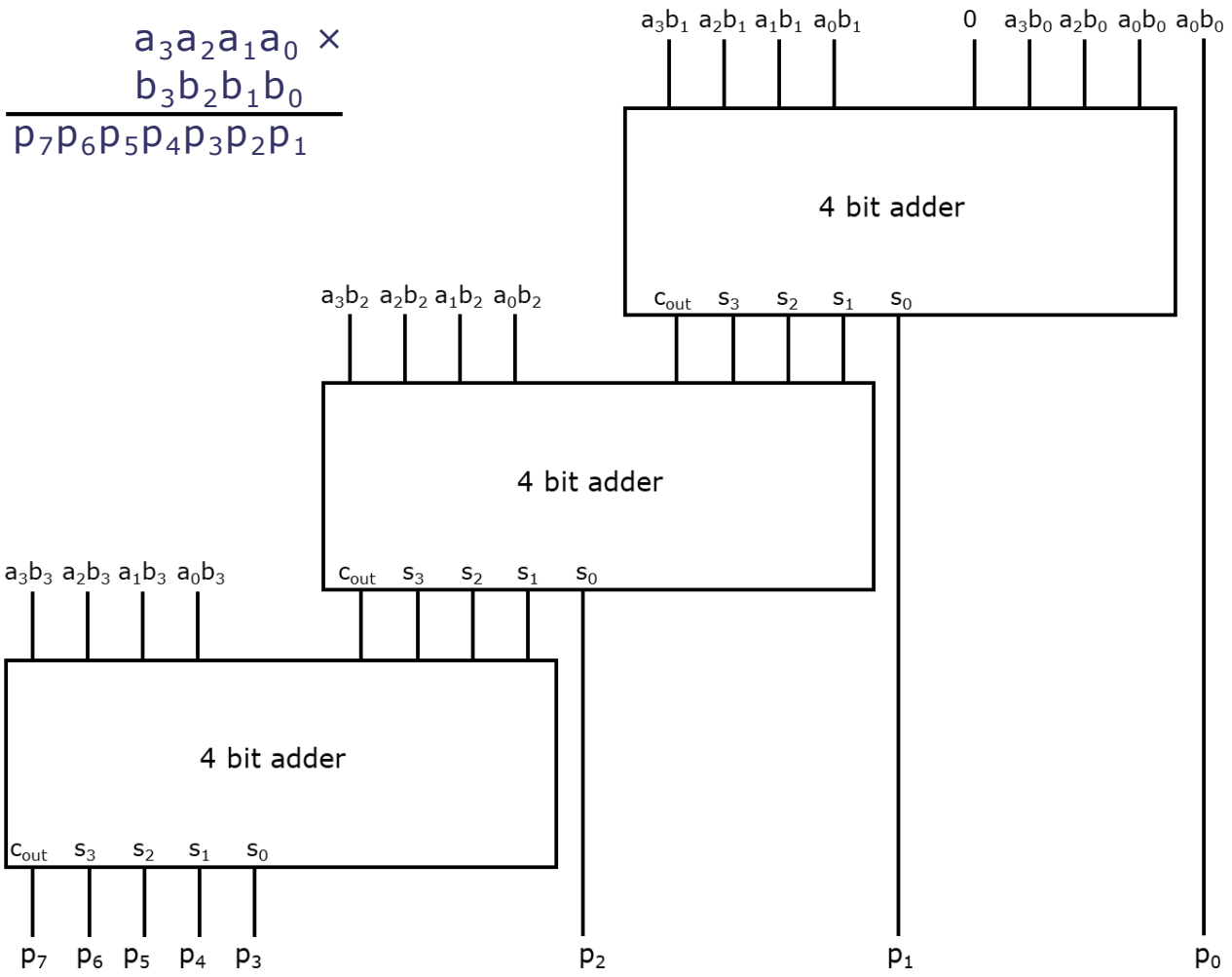


Multiplication (2nd ver)

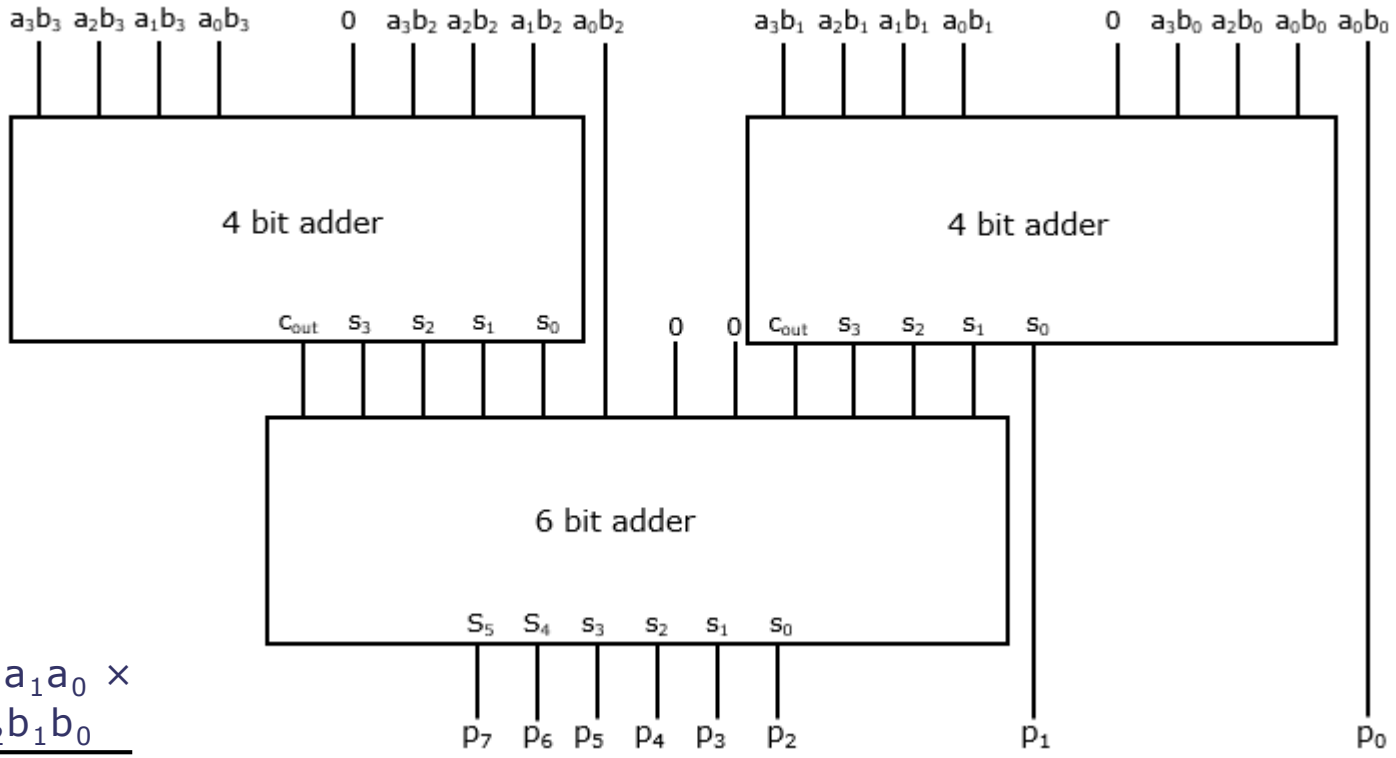


Combinational Multiplier

$$\begin{array}{r} a_3a_2a_1a_0 \times \\ b_3b_2b_1b_0 \\ \hline p_7p_6p_5p_4p_3p_2p_1 \end{array}$$



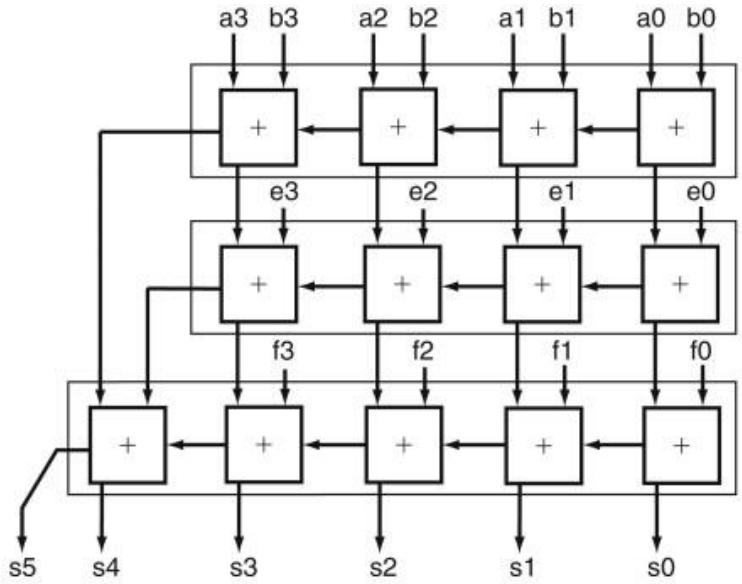
Combinational Multiplier (tree-form)

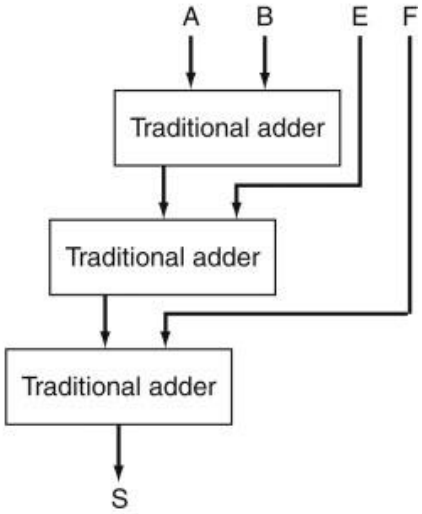


$$\begin{array}{r} a_3a_2a_1a_0 \times \\ b_3b_2b_1b_0 \\ \hline p_7p_6p_5p_4p_3p_2p_1 \end{array}$$

Addition of Multiple Numbers

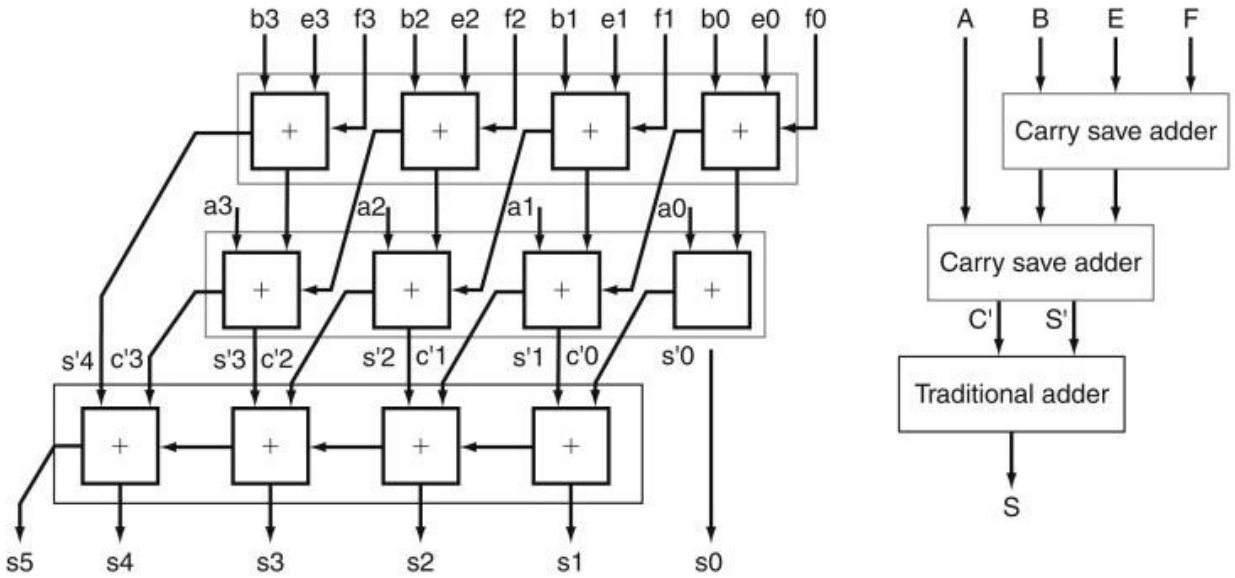
Traditional Approach



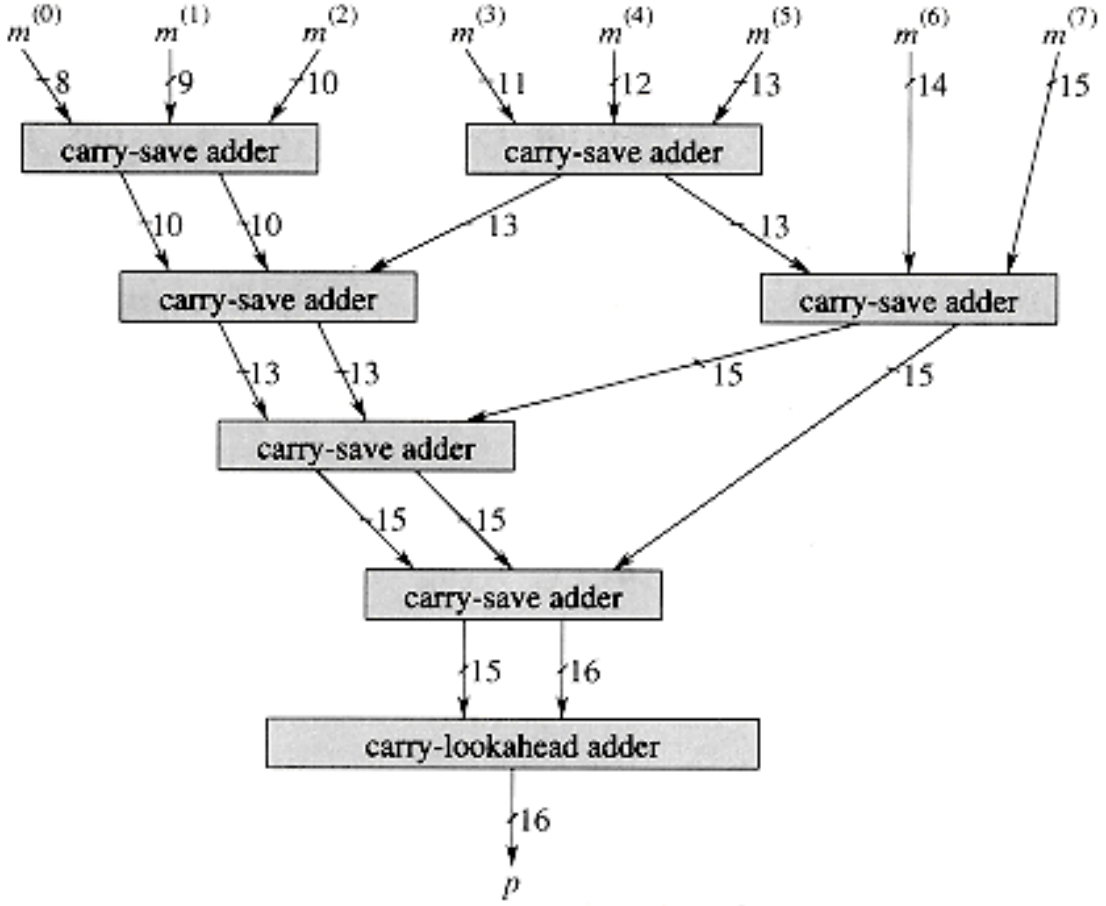


Addition of Multiple Numbers

Applying Carry Save Adders



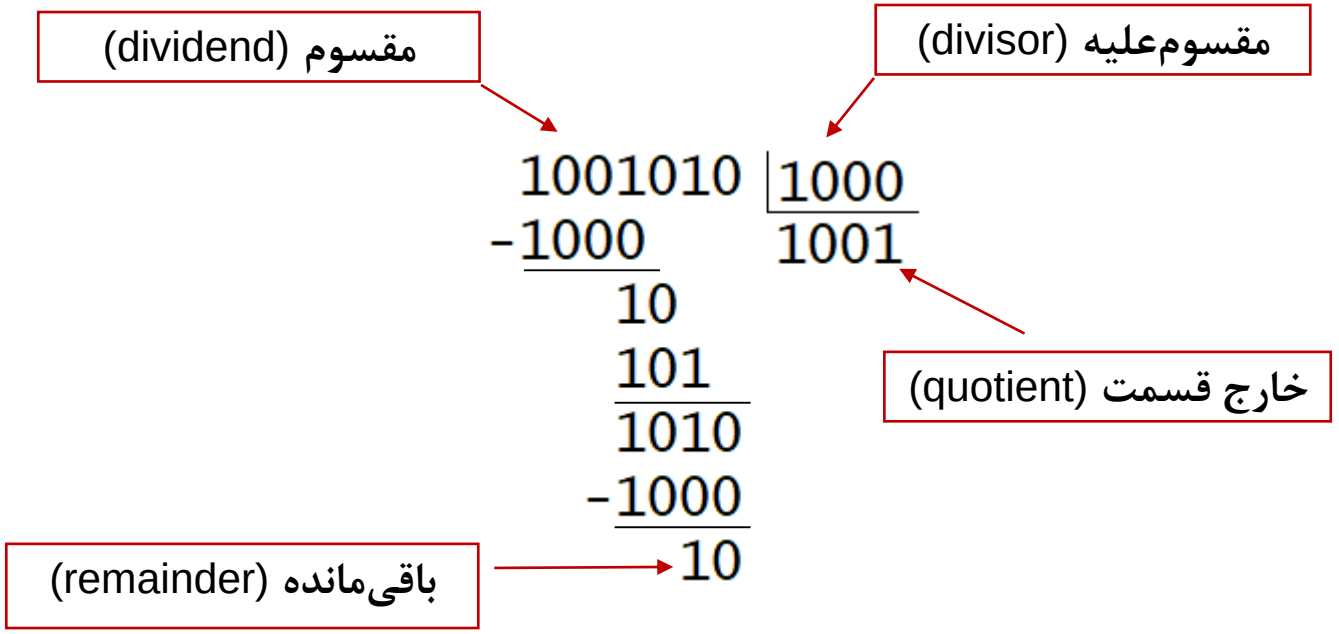
Carry-Save Adder Multiplier



Outlines

- Shift
- Addition/Subtraction
  - Ripple-Carry Adders
  - Carry Look-ahead Adders
  - Carry-Select Adders
- Multiplication
  - Shift-Add Multiplier
  - Combinational Multiplier
  - Carry-Save Adder Multiplier
- Division
- Floating Point
  - Representation
  - Arithmetic

Division

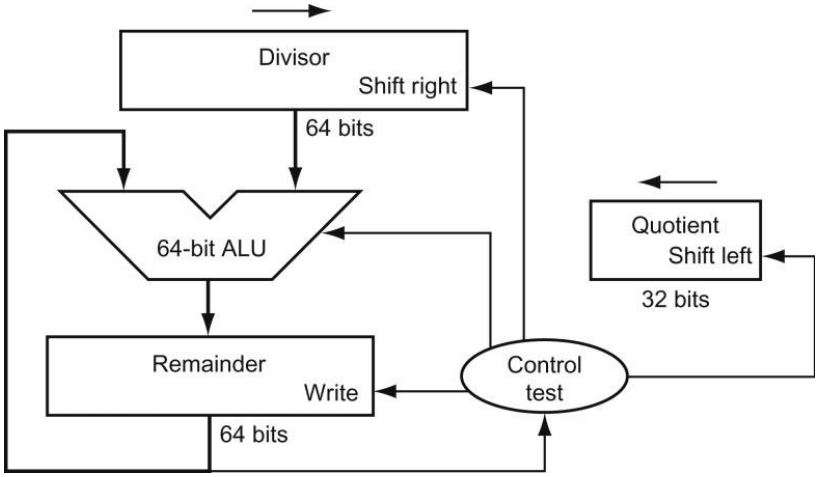


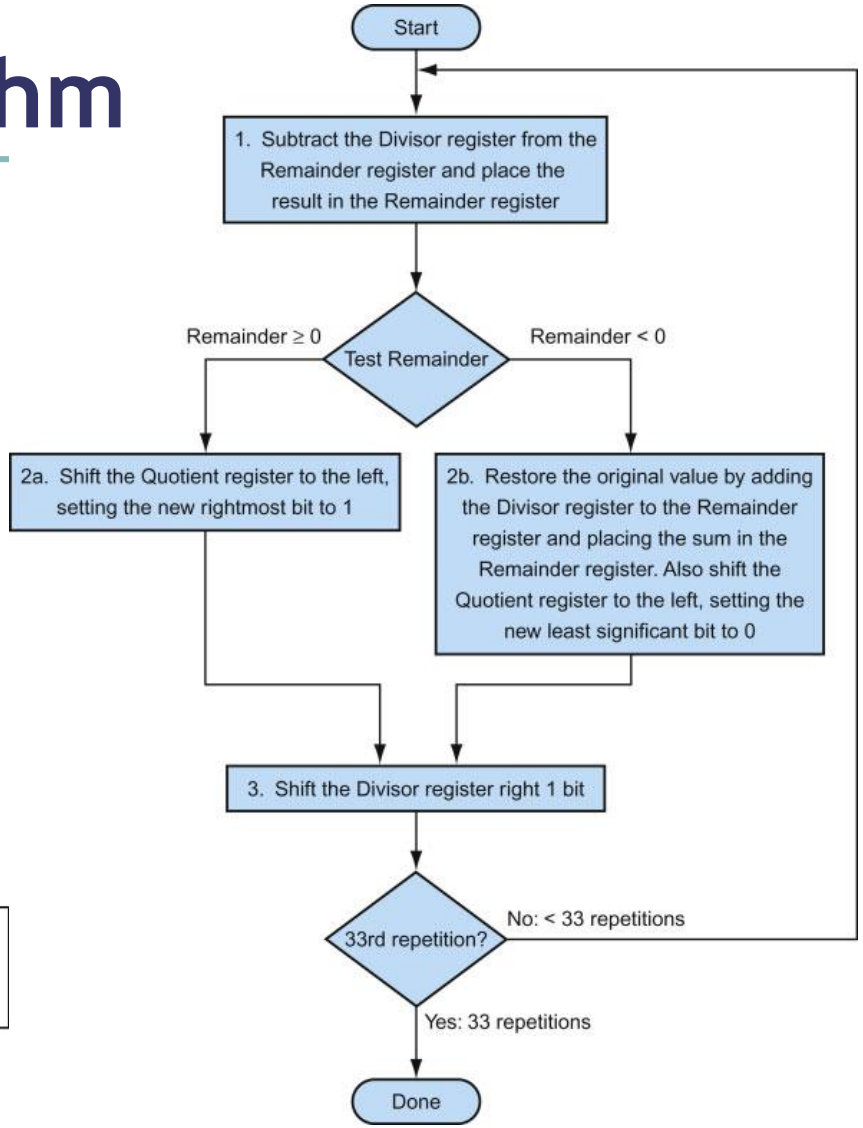
$$Dividend = Quotient \times Divisor + Remainder$$

$$|Remainder| < |Divisor|$$

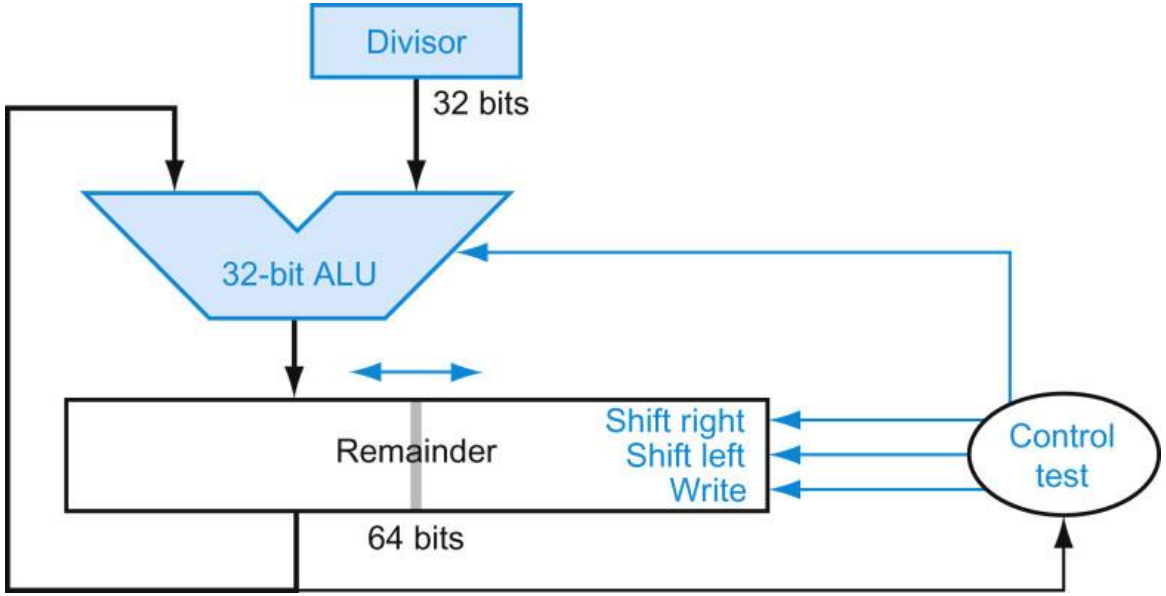
Division Algorithm

```
 1001010 |1000
-1000    |1001
   10
   101
   1010
  -1000
     10
```





Division Algorithm (improved)



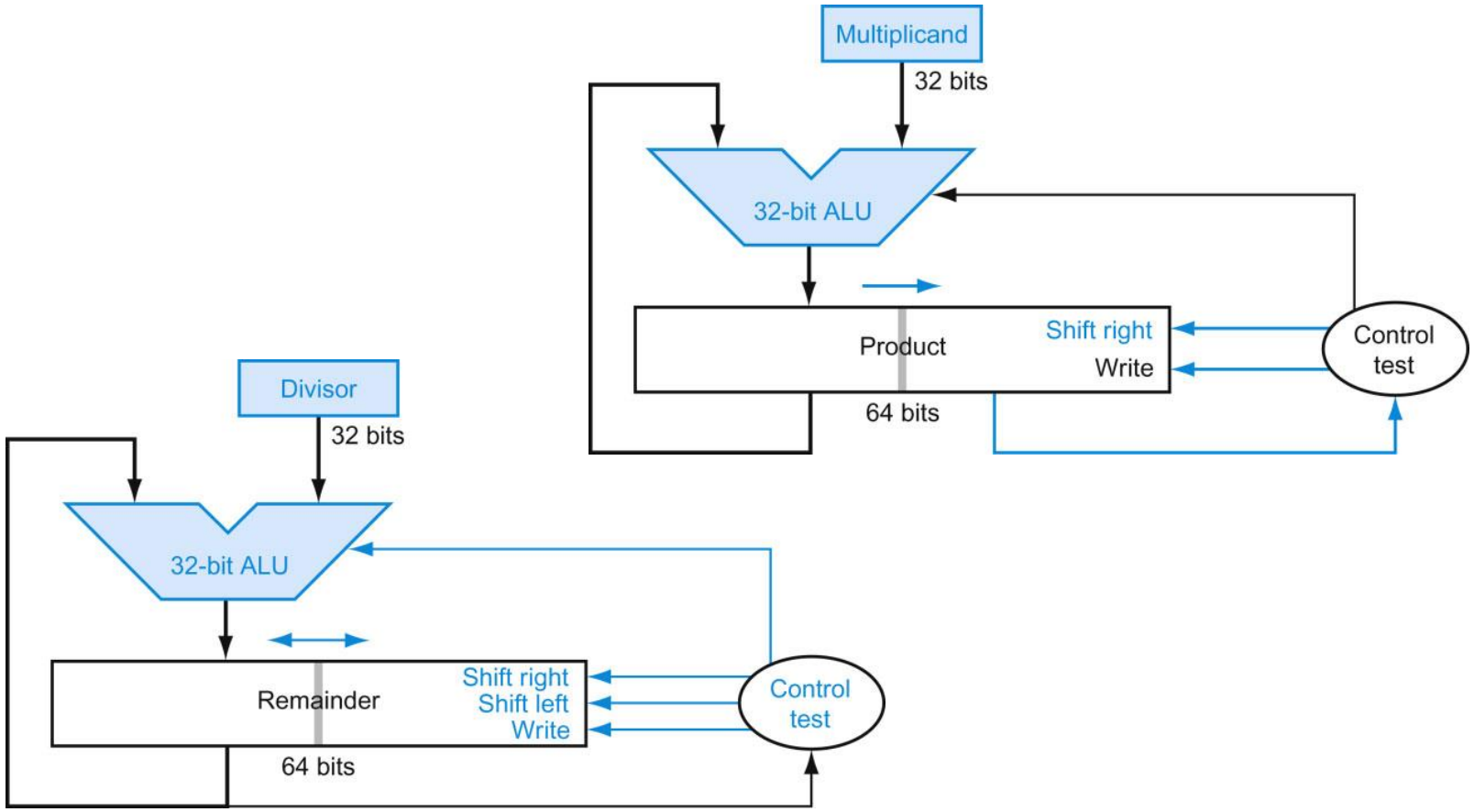
The Divisor register, ALU, and Quotient register are all 32 bits wide.

The ALU and Divisor registers are halved and the remainder is shifted left.

The Quotient register is combined with the right half of the Remainder register.

The Remainder register should really be 65 bits to make sure the carry out of the adder is not lost.

Multiplication vs. Division



Signed Division

- Divide using absolute values, considering:
  - Dividend = Quotient × Divisor + Remainder
- Adjust sign of quotient and remainder as required
  - no change in the absolute value of quotient
  - the dividend and remainder must have the same signs
  - e.g., divide ±7 by ±2

Fallacy

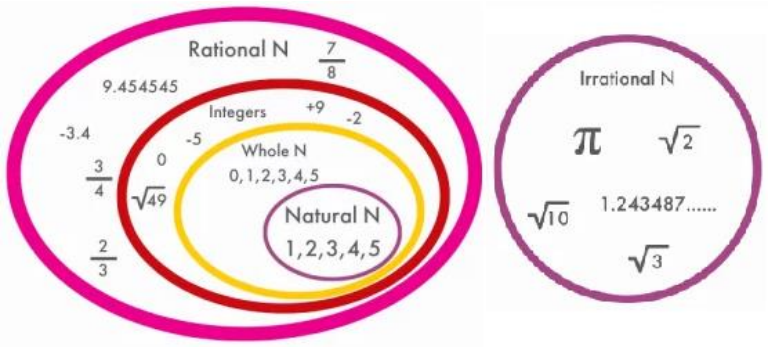
- Just as a left shift instruction can replace an integer multiply by a power of 2, a right shift is the same as an integer division by a power of 2
  - Only true for unsigned integers
  - Even with arithmetic right shift
  - e.g., try to shift right -5 twice

Outlines

- Shift
- Addition/Subtraction
  - Ripple-Carry Adders
  - Carry Look-ahead Adders
  - Carry-Select Adders
- Multiplication
  - Shift-Add Multiplier
  - Combinational Multiplier
  - Carry-Save Adder Multiplier
- Division
- Floating Point
  - Representation
  - Arithmetic

Real Numbers

- Numbers with Fractions:
  - 3.14159…
  - 2.17
  - 0.0000001
  - 12.5×10^{-12}
  - $1.43 \times 10^{+12}$



- Representation in computers:
  - Fixed point
  - Floating point

Fixed-Point Representation

- A real Example:
  - $d_{23}d_{22}...d_1d_0.f_{-1}f_{-2}f_{-3}f_{-4}f_{-5}f_{-6}f_{-7}f_{-8}$
  - 24-bit: integer bits
  - 8-bit: fraction bits
- Application
  - Used in CPUs with no floating-point unit
    - Embedded microprocessors and microcontrollers
  - Digital Signal Processing (DSP) applications

Fixed-Point Representation (cont.)

- Consider 5-Bit Representation
  - $d_2 d_1 d_0 . f_{-1} f_{-2}$
  - $(d_2 \times -2^2)+(d_1 \times 2^1)+(d_0 \times 2^0)+(f_{-1} \times 2^{-1})+(f_{-2} \times 2^{-2})$
- Largest positive number?
- Smallest positive number?
- Largest magnitude negative number?
- Smallest magnitude negative number?

Fixed-Point Representation (cont.)

- Arithmetic:
  - 011.11 + 011.11 = 111.10
  - 010.10 × 000.10 = 000001.0100
  - 000.01 × 000.01 = 000000.0001
  - 011.01 × 011.01 = ?

out of range (overflow)

out of range (underflow)

Fixed-Point Representation (cont.)

- Arithmetic:
  - 011.11 + 011.11 = 111.10
  - 010.10 × 000.10 = 000001.0100
  - 000.01 × 000.01 = 000000.0001
  - 011.01 × 011.01 = 001010.1001

out of range (overflow)

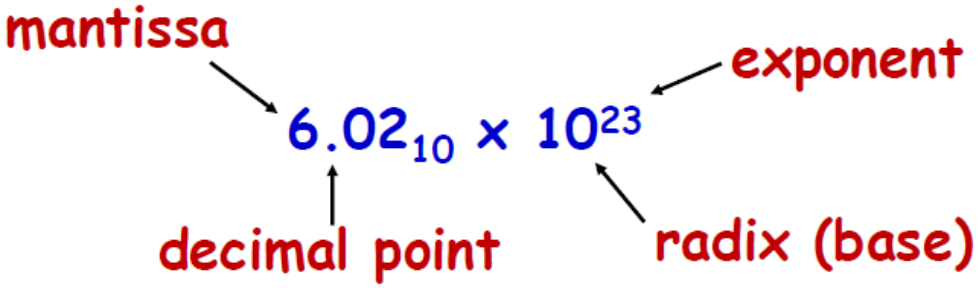
out of range (underflow)

Both overflow & underflow

Fixed-Point Representation (cont.)

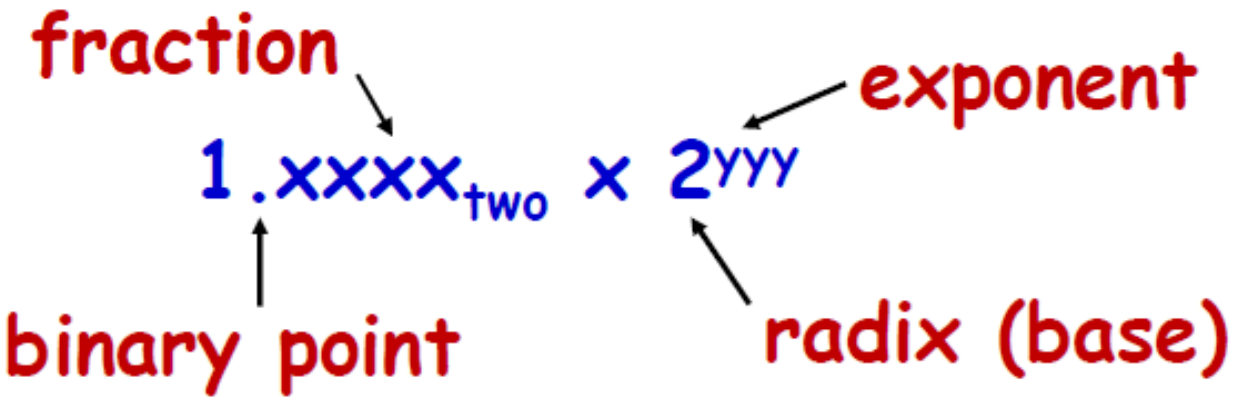
- ☺ Pros
  - Simple hardware
  - Fast computation
  - Different precisions at different applications
    - 24bits/8bits , 18bits/12bits, 8bits/24bits
- ☹ Cons
  - Low precision
  - Small range

Scientific Notation (Decimal)



- Normalized Form:
  - Exactly one non-zero digit to left of decimal point
- Alternatives to representing 0.0000000012:
  - Normalized: 1.2×10^{-9}
  - Not normalized: 0.12×10^{-8}, 12.0×10^{-10}

Normalized Scientific Notation (Binary)



Floating-Point Notation

- Floating Point Notation Consists of:
  - Fraction (F): 23 bits
  - Exponent (E): 8 bits
  - Fraction Sign bit (S)
  - Also called, single precision floating-point
- $N = (-1)^S \times (1+F) \times 2^E$

Floating-Point Notation (cont.)

- Pros (compared to fixed-point)
  - Very Wide Range
  - More precision bits
- Cons (compared to fixed-point)
  - Arithmetic operation more complicated
  - HW more complicated

Floating-Point Notation (cont.)

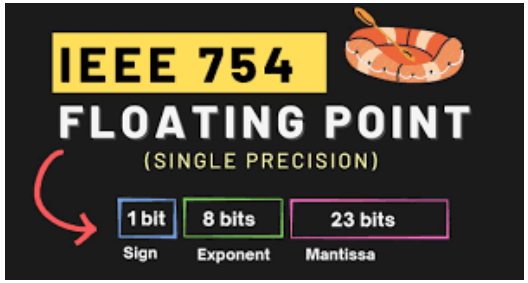
- $N = (-1)^S \times (1 + F) \times 2^E$
- Precision versus Range
  - More precision ➔ smaller range?
  - Wider range ➔ less precision?
- True for fixed-point
  - Not necessarily correct for floating point

Floating-Point Notation (cont.)

- Overflow:
  - Exponent too large to fit in “Exponent” field
- Underflow:
  - Non-zero fraction so small to represent
  - Negative exponent too large to fit

IEEE 754 Floating Point Standard

- There are many reasonable ways to represent floating-point numbers
- For many years, computer manufacturers used incompatible floating-point formats
- Results from one computer could not directly be interpreted by another computer.
- The Institute of Electrical and Electronics Engineers solved this problem by defining the IEEE 754 floating point standard in 1985 defining floating-point numbers



- This floating-point format is now almost universally used

IEEE 754 - Single Precision

- Signed-magnitude notation for mantissa
- Biased (Excess $2^{n-1}-1$) notation for exponent
- E_{min}=00000001
- E_{max}=11111110
- E=00000000 reserved for zero
- E=11111111 reserved for infinity & NaN
- Smallest positive no: 1.17549435 E-38
- Largest positive no: 3.4028235 E38

$$N = (-1)^S * (1 + F) * 2^{E-bias}$$

IEEE 754 - Double Precision

- Two words long (64 bits)
- Reduced chances of overflow/underflow
- Format
  - Sign bit (S)
  - Fraction (F): 52 bits
  - Exponent (E): 11 bits
- A bias of 1023 in Exponential part

More on IEEE 754 Standard

- Single precision (32bits)/Double precision (64bits)
- Normalized/ Denormalized forms
- Standard definitions for zero, infinity, NaN
- Check: https://www.h-schmidt.net/FloatConverter/IEEE754.html

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	± denormalized number
1–254	Anything	1–2046	Anything	± floating-point number
255	0	2047	0	± infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

Denormalized Forms

- An attempt to squeeze every last bit of precision from a floating-point operation
- The smallest positive single precision normalized no:
  - $1.00000000000000000000000 \times 2^{-126}$
- The smallest single precision denormalized no:
  - $0.00000000000000000000001 \times 2^{-126} = 1.0 \times 2^{-149}$

Floating-Point Addition

1. Align binary points
   - Shift number with smaller exponent (why?)
2. Add significands
3. Normalize result & check for over/underflow
4. Round and renormalize if necessary

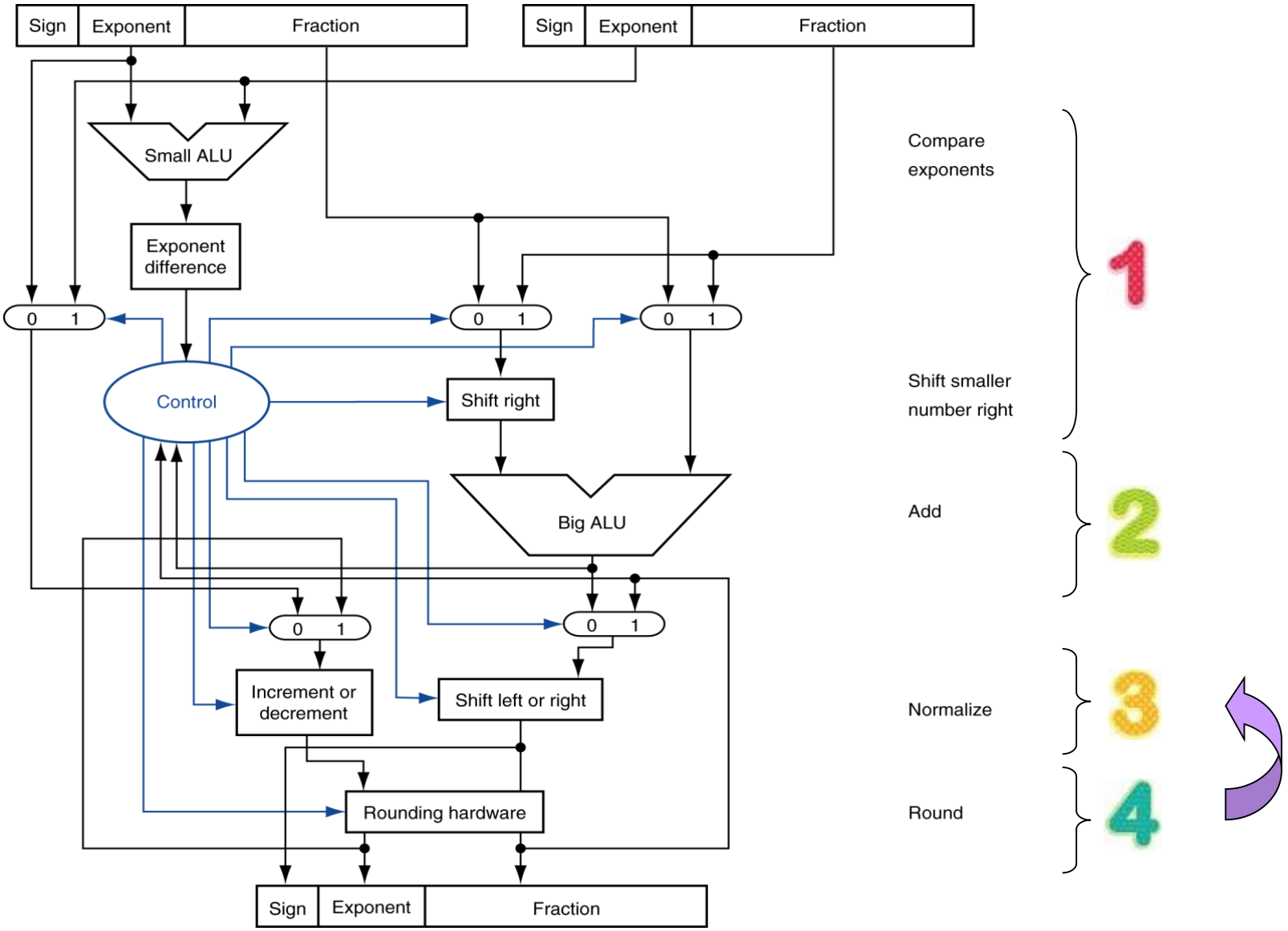
Example

- Consider 0.5 + (-0.4375)
  - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$

1. Align binary points (Shift number with smaller exponent)
   - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
2. Add significands
   - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$
3. Normalize result & check for over/underflow
   - $1.000_2 \times 2^{-4}$, with no over/underflow
4. Round and renormalize if necessary
   - $1.000_2 \times 2^{-4}$ (no change) = 0.0625

FP Adder Hardware

- Much more complex than integer adder
- Doing it in one clock cycle takes too long
  - Much longer than integer operations
  - Slower clock would penalize all instructions
- FP adder usually takes several cycles
  - Can be pipelined


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
1
Shift smaller number right
Add
2
Normalize
3
Round
4

Floating-Point Multiplication

1. Add exponents
   - For biased exponents, subtract bias from sum
2. Multiply significands
3. Normalize result & check for over/underflow
4. Round and renormalize if necessary
5. Determine sign of result from signs of operands

Example

- Consider 0.5 × (-0.4375)
  - $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$

1. Add exponents
   - Biased: $(-1 + 127) + (-2 + 127) = -3 + 254 - 127 = -3 + 127$
2. Multiply significands
   - $1.000_2 \times 1.110_2 = 1.110_2 \Rightarrow 1.110_2 \times 2^{-3}$
3. Normalize result & check for over/underflow
   - $1.110_2 \times 2^{-3}$, with no over/underflow
4. Round and renormalize if necessary
   - $1.110_2 \times 2^{-3} = 0.21875$
5. Determine sign of result from signs of operands
   - $-1.1102 \times 2^{-3} = -0.21875$

FP Arithmetic Hardware

- FP multiplier is of similar complexity to FP adder
  - Uses a multiplier for significands instead of adder
- FP arithmetic hardware usually does
  - Addition, subtraction, multiplication, division, reciprocal, square-root, FP $\leftrightarrow$ integer conversion
- Operations usually takes several cycles
  - Can be pipelined

Concluding Remarks

- Bits have no inherent meaning
  - Interpretation depends on the operations applied
- Computer representations of numbers
  - Finite range and precision
  - Need to account for this in programs
- Bounded range and precision
  - Operations can overflow and underflow