System I

Computational Operations & Units

Haifeng Liu

Zhejiang University

Overview

- Addition
- Subtraction
- Arithmetic logic unit (ALU)
- Multiplication
- Division
- Floating number operations

Overview

- Addition
- Subtraction
- Arithmetic logic unit (ALU)
- Multiplication
- Division
- Floating number operations

Iterative Combinational Circuits

- Arithmetic functions
  - Operate on binary vectors
  - Use the same subfunction in each bit position
- Can design functional block for subfunction and repeat to obtain functional block for overall function
- *Cell* - subfunction block
- *Iterative array* - a array of interconnected cells
- An iterative array can be in a single dimension (1D) or multiple dimensions

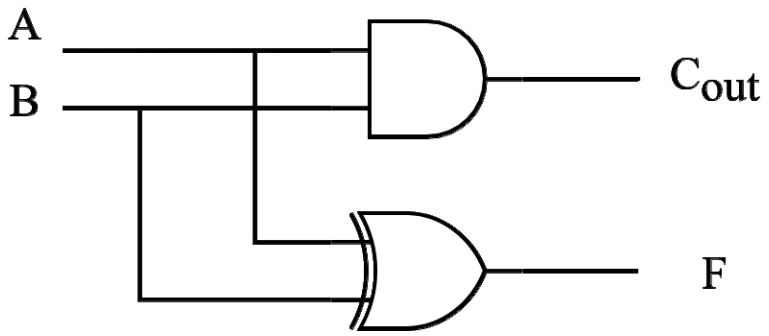
1-bit adder: Half-Adder

- A 2-input, 1-bit width binary adder that performs the following computations:

A	**0**	**0**	**1**	**1**
+ B	**+ 0**	**+ 1**	**+ 0**	**+ 1**
C F	**0 0**	**0 1**	**0 1**	**1 0**

- A half adder adds two bits to produce a two-bit sum
- The sum is expressed as a sum bit , F and a carry bit, C
- The half adder can be specified as a truth table for F and C $\Rightarrow$

A	B	F	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



1-bit adder: Full Adder

- A full adder is similar to a half adder, but includes a carry-in bit from lower stages.
- Like the half-adder, it computes a sum bit **F** and a carry bit C_{in}
  - For a carry-in (C_{in}) of 0, it is the same as the half-adder:

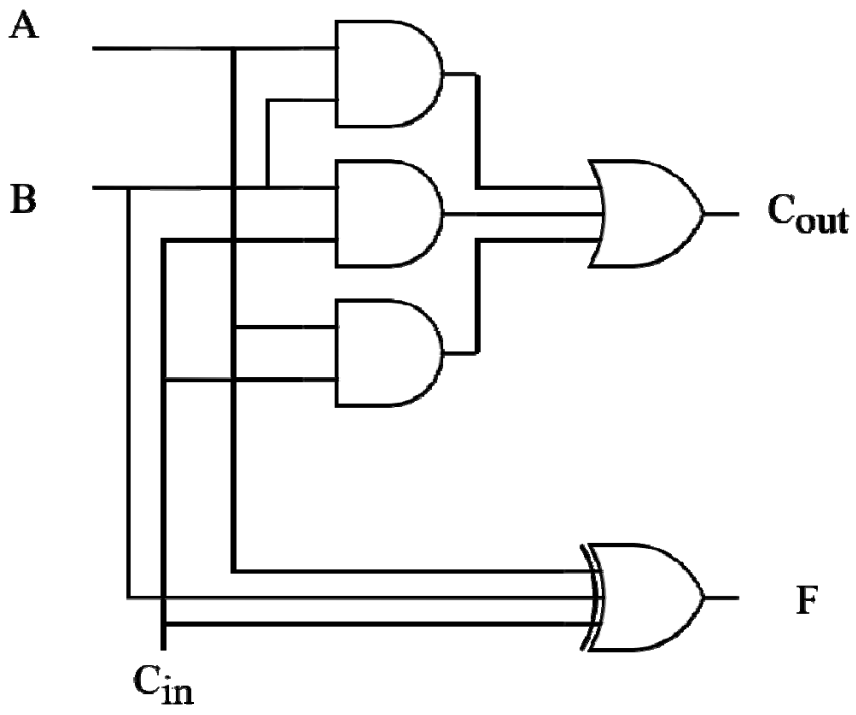
C_{in}	0	0	0	0
A	0	0	1	1
+ B	+ 0	+ 1	+ 0	+ 1
C_{out} **F**	0 0	0 1	0 1	1 0

  - For a carry-in (C_{in}) of 1:

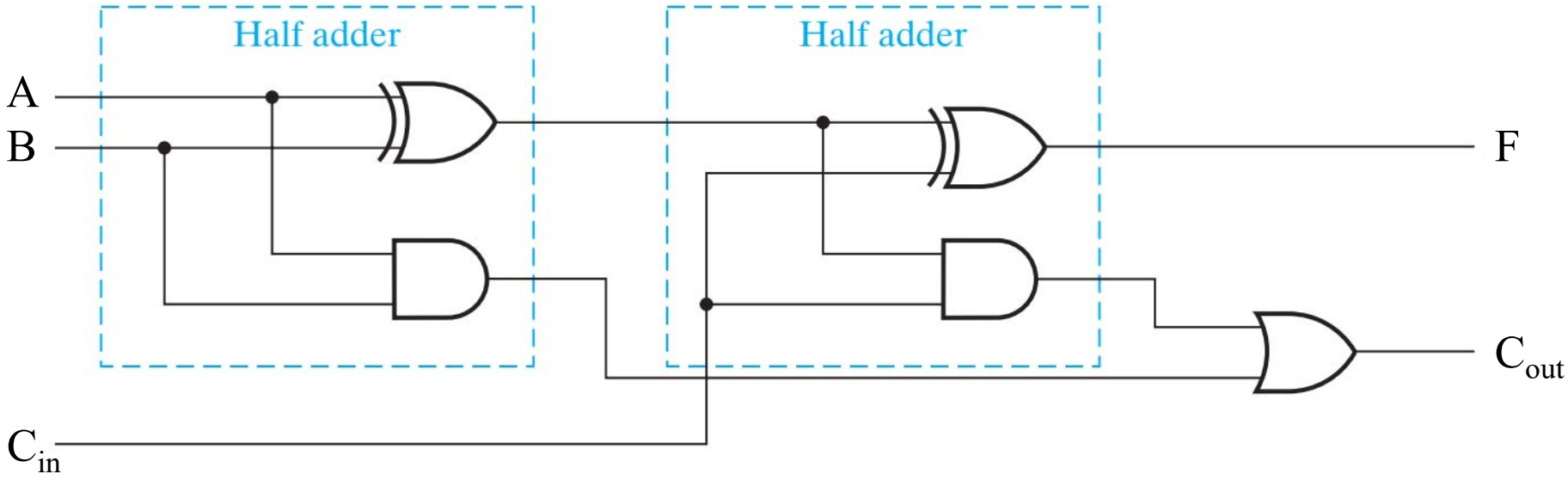
C_{in}	1	1	1	1
A	0	0	1	1
+ B	+ 0	+ 1	+ 0	+ 1
C_{out} **F**	0 1	1 0	1 0	1 1

Classic Designs: Full Adder (cont'd)

Input			Output	
A	B	C_{in}	F	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

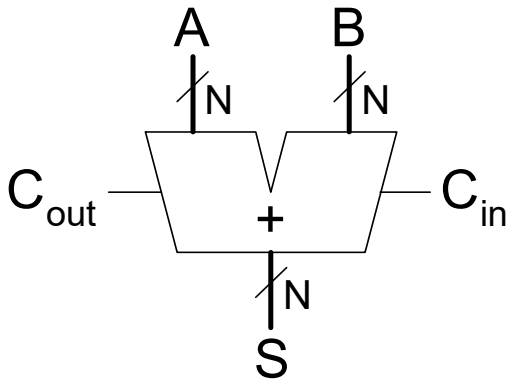


Classic Designs: Full Adder (cont'd)



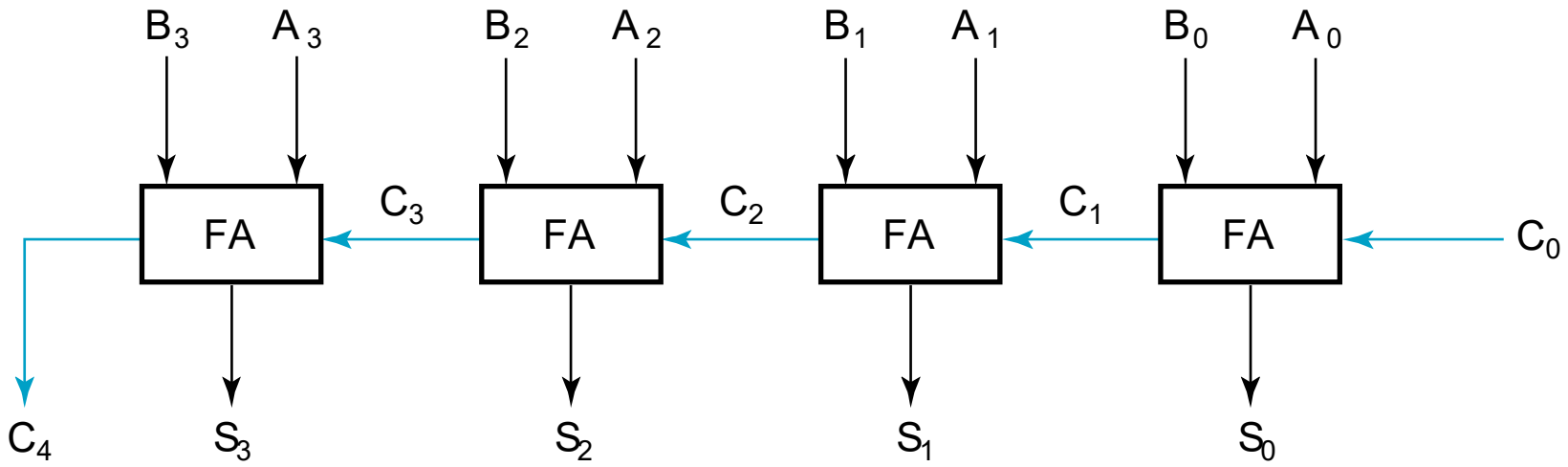
Multibit Carry Propagate Adders (CPA)

- Types of CPA
  - Ripple-carry (slow)
  - Carry Skip
  - Carry Select
  - Carry-lookahead (fast)
  - Prefix (fast)
- Faster adders require more hardware



Ripple-Carry Adder (RCA)

- Chain 1-bit adders together
- Carry ripples through entire chain
- Disadvantage: **slow**



Ripple-Carry Adder Delay

$$t_{ripple} = Nt_{FA}$$

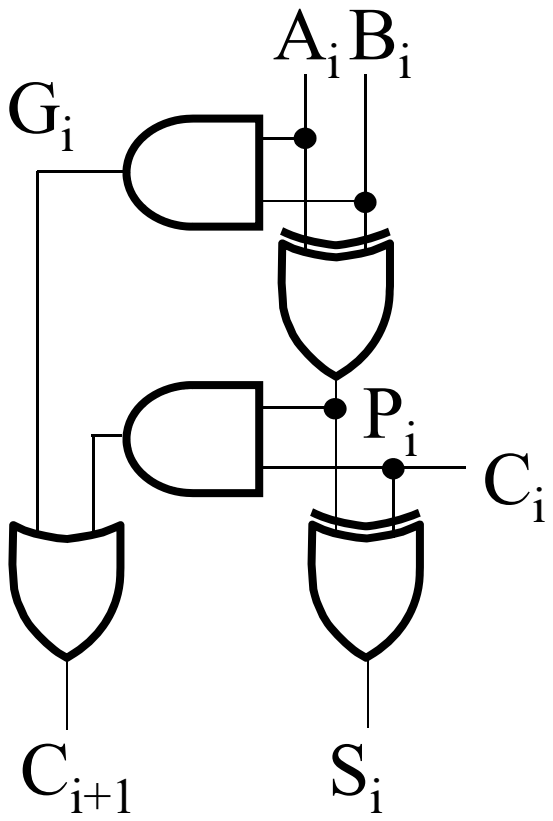
t_{FA}: delay of a 1-bit full adder

Carry Lookahead Adder

$$P_i = A_i \oplus B_i \qquad G_i = A_i B_i$$

$$S_i = P_i \oplus C_i \qquad C_{i+1} = G_i + P_i C_i$$

- For a given i-bit binary adder,
  - If $A_i = B_i =$ "1" and whatever C_i is, we have carry out as 1, that is $C_{i+1} = 1$
  - If the output of half adder is 1 and we have carry in as 1, we have carry out as 1， that is $C_{i+1} = 1$
- These two conditions of setting carry out as 1 is called *generate* (G_i) and *propagate* (P_i).



Carry Lookahead Development

- $C_{i+1} = G_i + P_i C_i$ can be removed from the cells and used to derive a set of carry equations spanning multiple cells.
- Beginning at the cell 0 with carry in C_0:

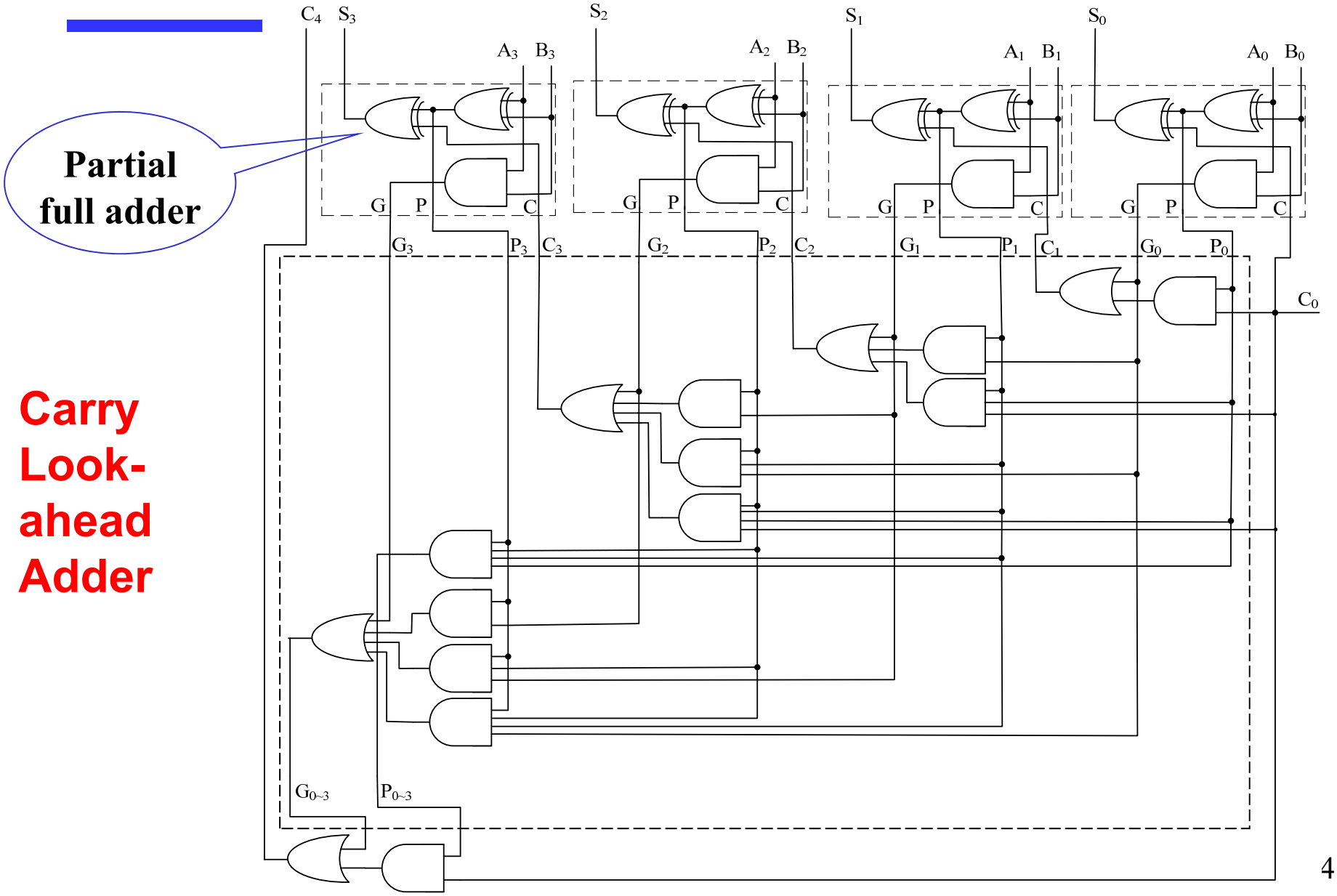
$$C_1 = G_0 + P_0 C_0$$

$$C_2 = G_1 + P_1 C_1 = G_1 + P_1(G_0 + P_0 C_0)$$
$$= G_1 + P_1G_0 + P_1P_0 C_0$$

$$C_3 = G_2 + P_2 C_2 = G_2 + P_2(G_1 + P_1G_0 + P_1P_0 C_0)$$
$$= G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0 C_0$$

$$C_4 = G_3 + P_3 C_3$$
$$= G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0 C_0$$

4-Bit CLA


Partial full adder
Carry Look-ahead Adder
C4 S3
S2
S1
S0
A3 B3
A2 B2
A1 B1
A0 B0
G P C
G3 P3 C3
G2 P2 C2
G1 P1 C1
G0 P0
C0
G0~3 P0~3

Group Carry Lookahead Logic

- Figure in the previous slide shows the implementation of these equations for four bits. This could be extended to more than four bits; in practice, due to limited gate fan-in, such extension is not feasible.
- $C_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0\ C_0$
- Instead, the concept is extended another level by considering *group generate* ($G_{0\text{-}3}$) and *group propagate* ($P_{0\text{-}3}$) functions:

$$\mathbf{G_{0\sim3} = G_3 + P_3\,G_2 + P_3\,P_2\,G_1 + P_3\,P_2\,P_1\,P_0\,G_0}$$
$$\mathbf{P_{0\sim3} = P_3\,P_2\,P_1\,P_0}$$

- Using these two equations:

$$\mathbf{C_4 = G_{0\sim3} + P_{0\sim3}\,C_0}$$

- Thus, it is possible to have four 4-bit adders use one of the same carry lookahead circuit to speed up 16-bit addition

Group Carry Lookahead Logic (Cont.)

- $C_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0 = G_{0\sim3} + P_{0\sim3}C_0$
- $C_8 = G_7 + P_7G_6 + P_7P_6G_5 + P_7P_6P_5G_4 + P_7P_6P_5P_4C_4 = G_{4\sim7} + P_{4\sim7}C_4$
- $C_{12} = G_{11} + P_{11}G_{10} + P_{11}P_{10}G_9 + P_{11}P_{10}P_9G_8 + P_{11}P_{10}P_9P_8C_8 = G_{8\sim11} + P_{8\sim11}C_8$
- $C_{16} = G_{15} + P_{15}G_{14} + P_{15}P_{14}G_{13} + P_{15}P_{14}P_{13}G_{12} + P_{15}P_{14}P_{13}P_{12}C_{12} = G_{12\sim15} + P_{12\sim15}C_{12}$

Group Carry Lookahead Logic (Cont.)

- $C_4 = G_{0\sim3} + P_{0\sim3} C_0$
- $C_8 = G_{4\sim7} + P_{4\sim7} C_4$
- $C_{12} = G_{8\sim11} + P_{8\sim11} C_8$
- $C_{16} = G_{12\sim15} + P_{12\sim15} C_{12}$

$$C_1 = G_0 + P_0 C_0$$

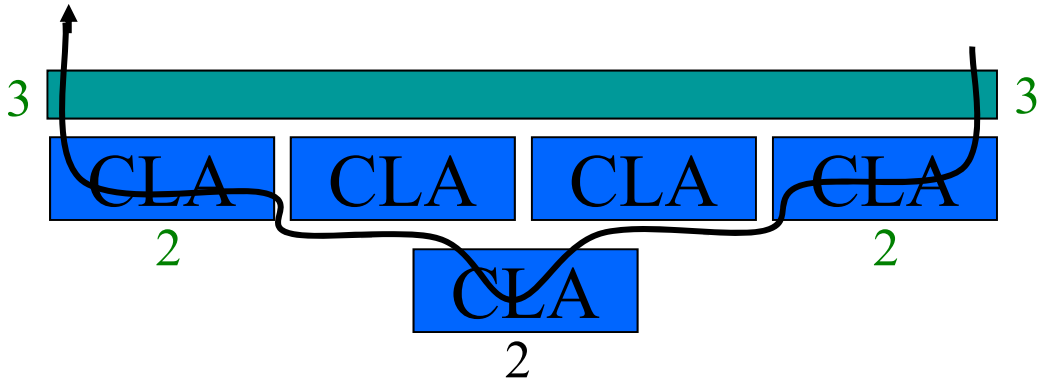
$$C_2 = G_1 + P_1 C_1$$

$$C_3 = G_2 + P_2 C_2$$

$$C_4 = G_3 + P_3 C_3$$

Carry Lookahead Example

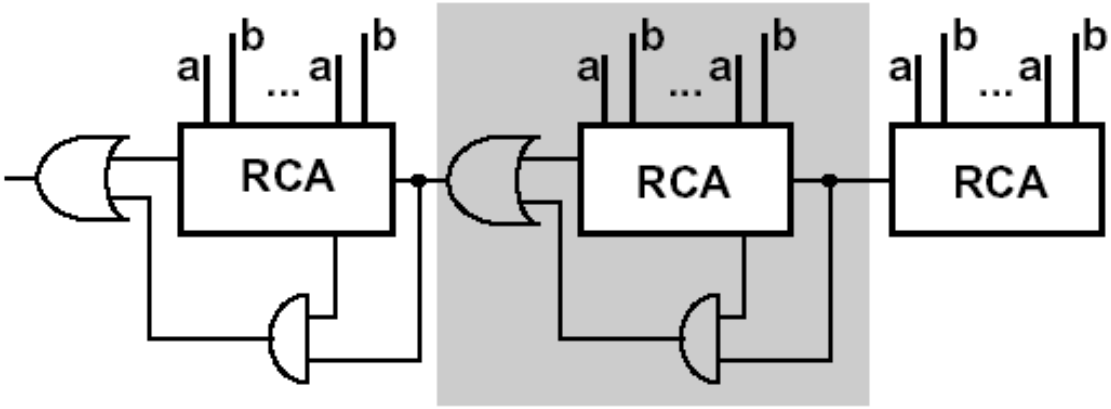
- Specifications:
  - 16-bit CLA
  - Delays:
    - NOT = 1
    - XOR = Isolated AND = 3
    - AND-OR = 2



- Longest Delays:
  - Ripple carry adder* = $3 + 15 \times 2 + 3 = 36$
  - CLA = $3 + 3 \times 2 + 3 = 12$

Carry skip adder

- Accelerating the carry by skipping the interior blocks
- Optimal speed with no-equal distribution of block length



Carry skip adder

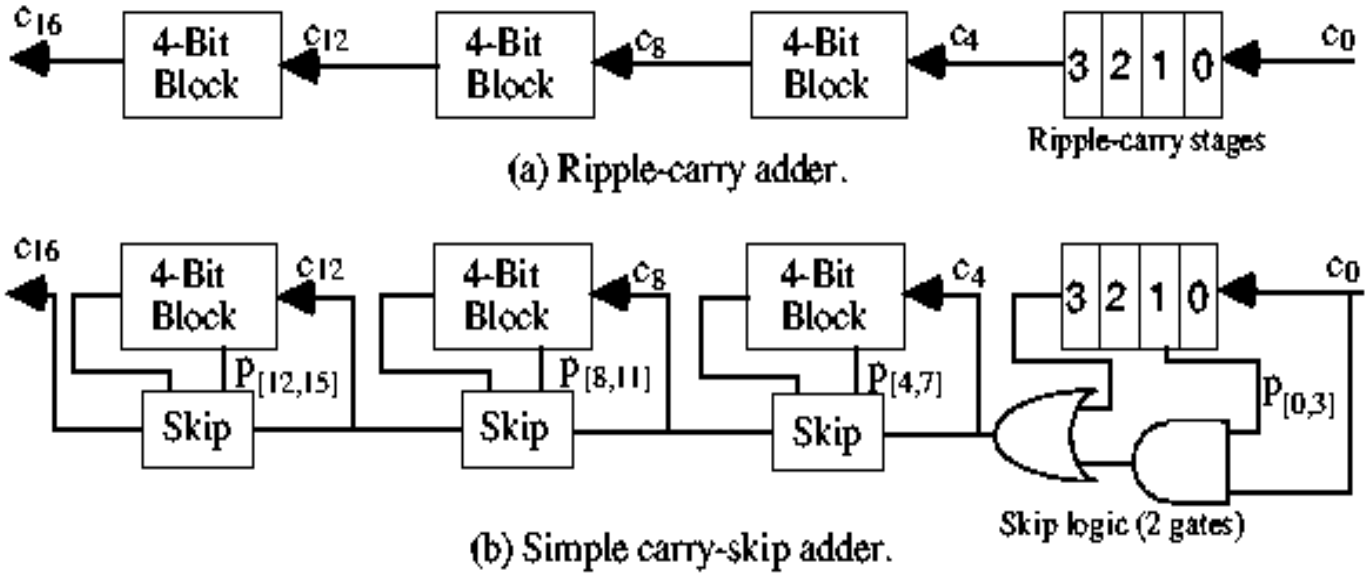
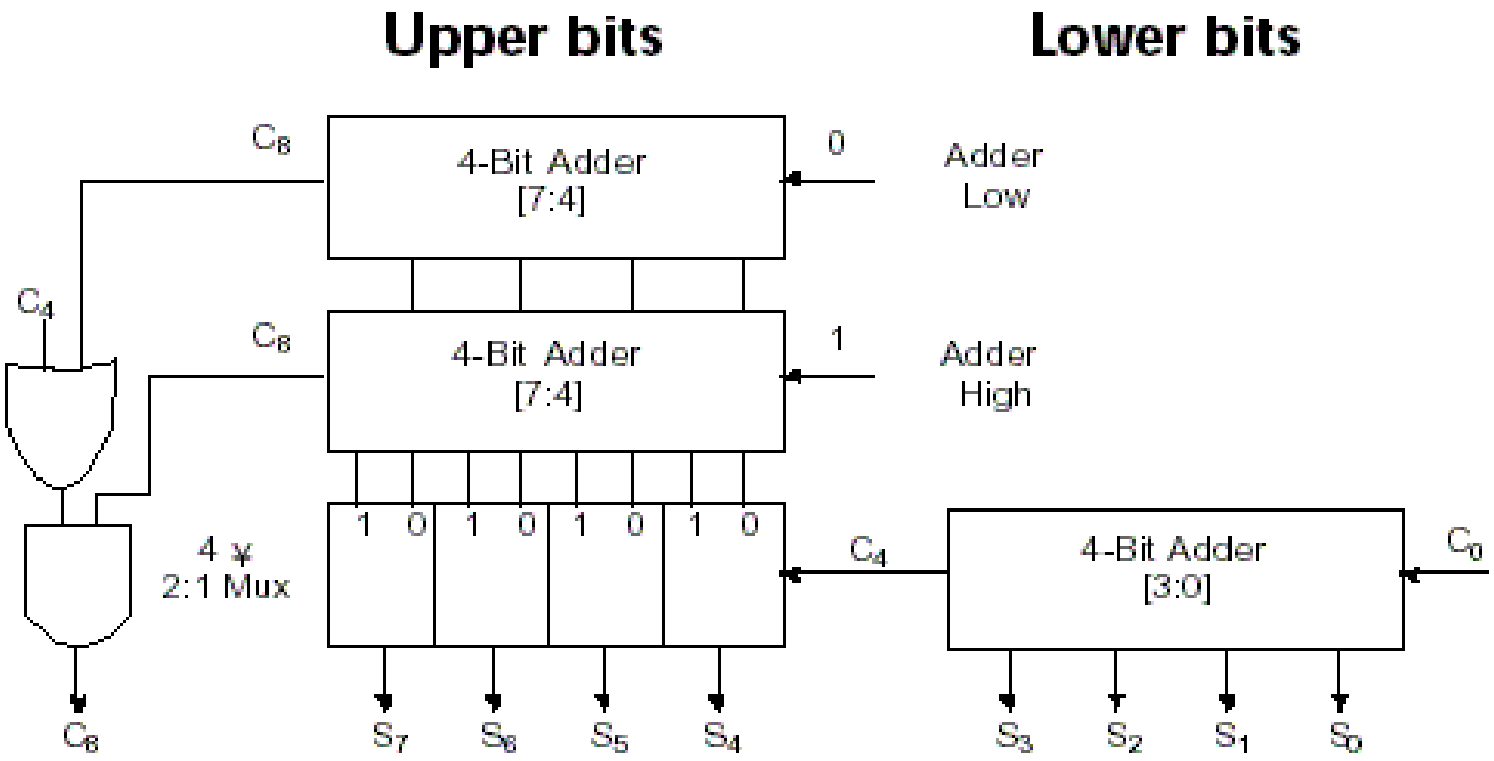


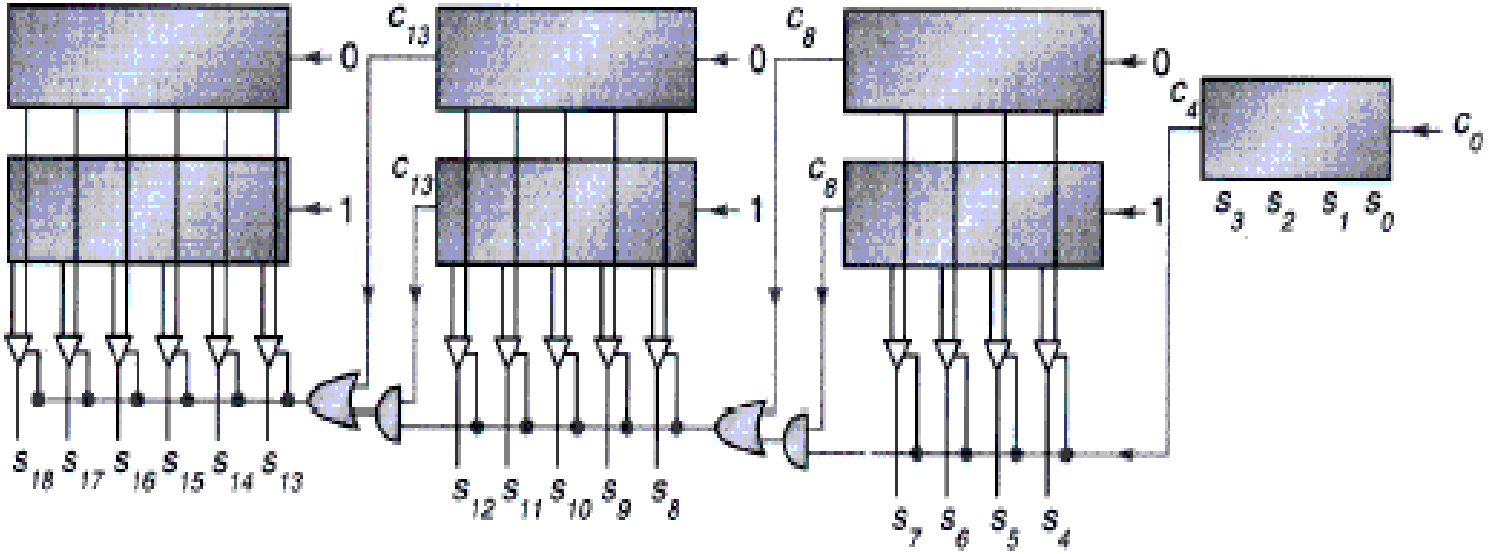
Fig. 7.1 **Converting a 16-bit ripple-carry adder into a simple carry-skip adder with 4-bit skip blocks.**

Carry select adder (CSA)



Carry select adder

- Carry selection by nibbles



Prefix Adder

- Computes carry in (C_{i-1}) for each column, then computes sum:

$$\mathbf{S_i = (A_i \oplus B_i) \oplus C_{i-1}}$$

- Computes G and P for 1-, 2-, 4-, 8-bit blocks, etc. until all G_i (carry in) known

- $\log_2 N$ stages

Prefix Adder (cont'd)

- Carry in either *generated* in a column or *propagated* from a previous column.
- Column -1 holds C_{in}, so

  $\mathbf{G_{-1} = C_{in}, P_{-1} = 0}$

- Carry in to column i == carry out of column i-1:

  $\mathbf{C_{i-1} = G_{i-1:-1}}$

  $G_{i-1:-1}$: generate signal spanning columns i-1 to -1

- Sum equation:

  $\mathbf{S_i = (A_i \oplus B_i) \oplus G_{i-1:-1}}$

- **Goal**: Quickly compute $G_{0:-1}$, $G_{1:-1}$, $G_{2:-1}$, $G_{3:-1}$, $G_{4:-1}$, $G_{5:-1}$, … (called ***prefixes***)

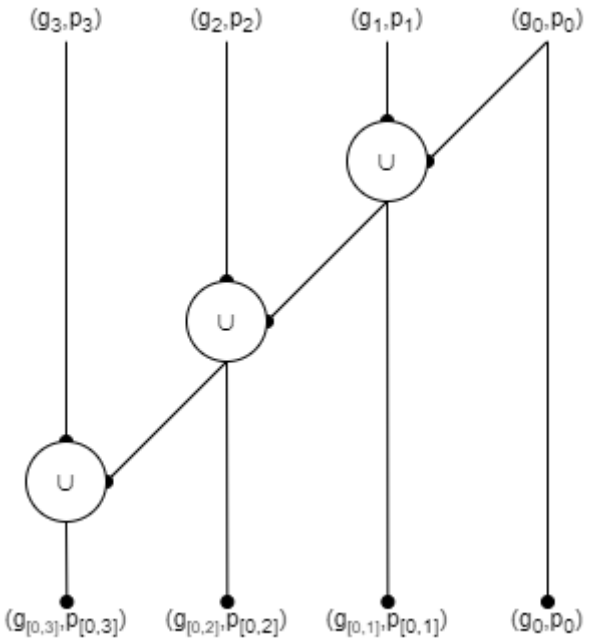
Prefix Adder (cont'd)

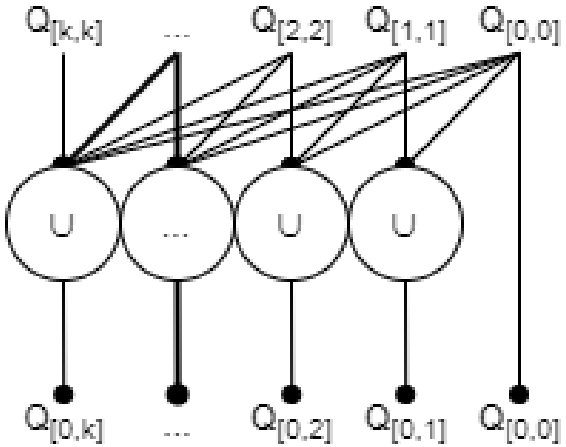
- Generate and propagate signals for a block spanning bits i:j:

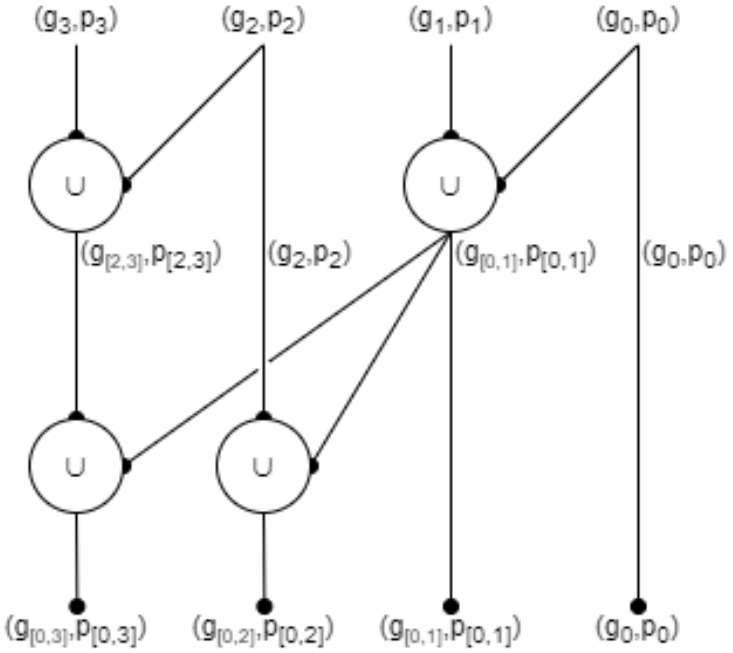
$$G_{i:j} = G_{i:k} + P_{i:k}\, G_{k-1:j}$$

$$P_{i:j} = P_{i:k} P_{k-1:j}$$

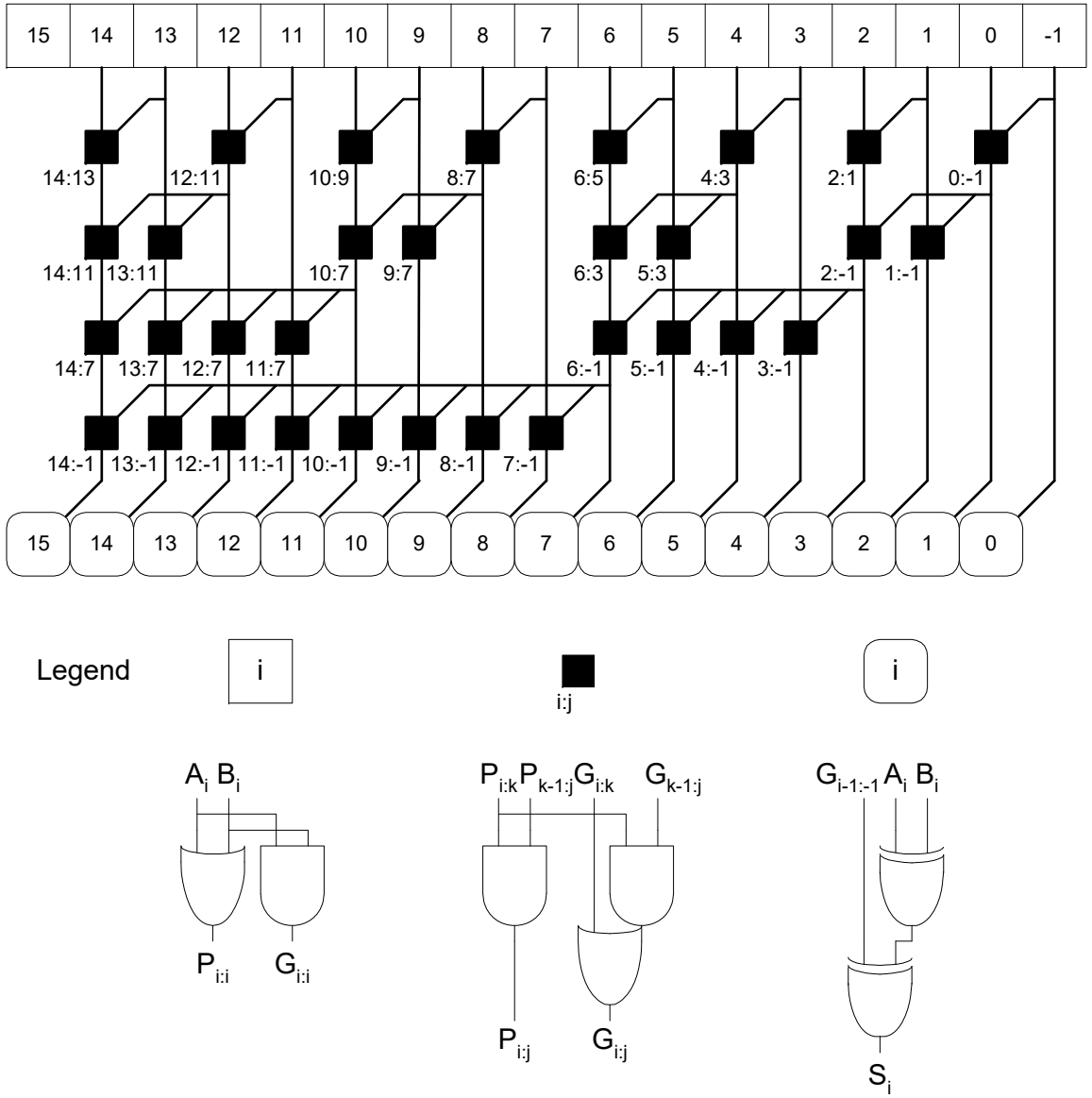
- In words:
  - **Generate**: block i:j will generate a carry if:
    - upper part (i:k) generates a carry or
    - upper part propagates a carry generated in lower part (k-1:j)
  - **Propagate**: block i:j will propagate a carry if *both* the upper and lower parts propagate the carry


(g_3,p_3)
(g_2,p_2)
(g_1,p_1)
(g_0,p_0)
$\cup$
$\cup$
$\cup$
$(g_{[0,3]},p_{[0,3]})$
$(g_{[0,2]},p_{[0,2]})$
$(g_{[0,1]},p_{[0,1]})$
(g_0,p_0)


$Q_{[k,k]}$
...
$Q_{[2,2]}$
$Q_{[1,1]}$
$Q_{[0,0]}$
$\cup$
...
$\cup$
$\cup$
$Q_{[0,k]}$
...
$Q_{[0,2]}$
$Q_{[0,1]}$
$Q_{[0,0]}$


(g_3,p_3)
(g_2,p_2)
(g_1,p_1)
(g_0,p_0)
$\cup$
$\cup$
$(g_{[2,3]},p_{[2,3]})$
(g_2,p_2)
$(g_{[0,1]},p_{[0,1]})$
(g_0,p_0)
$\cup$
$\cup$
$(g_{[0,3]},p_{[0,3]})$
$(g_{[0,2]},p_{[0,2]})$
$(g_{[0,1]},p_{[0,1]})$
(g_0,p_0)

Prefix Adder Schematic



Prefix Adder Delay

$$t_{PA} = t_{pg} + log_2N(t_{pg\_prefix}) + t_{XOR}$$

t_{pg}: delay to produce P_iG_i (AND or OR gate)

t_{pg_prefix}: delay of black prefix cell (AND-OR gate)

Overview

- Addition
- Subtraction
- Arithmetic logic unit (ALU)
- Multiplication
- Division
- Floating number operations

Unsigned Subtraction

- For n-digit, unsigned numbers M and N, find M – N in base 2:
  - Add the 2's complement of the subtrahend N to the minuend M:

    $$M + (2^n - N) = M - N + 2^n$$

  - If $M \geq N$, the sum produces end carry r^n which is discarded; from above, M – N remains.
  - If $M < N$, the sum does not produce an end carry and, from above, is equal to $2^n - (N - M)$, the 2's complement of (N – M).
  - To obtain the result – (N – M) , take the 2's complement of the sum and place a – to its left.

Unsigned 2's Complement Subtraction Example 1

- Find $01010100_2 - 01000011_2$

$$\begin{array}{rlrl} & 01010100 & & 1\ 01010100 \\ - & \underline{01000011} & \xrightarrow{\textbf{2's comp}} & + \underline{10111101} \\ & & & 00010001 \end{array}$$

- The carry of 1 indicates that no correction of the result is required.

Unsigned 2's Complement Subtraction Example 2

- Find $01000011_2 - 01010100_2$

$$
\begin{array}{rlll}
 & 01000011 & & \mathbf{0}\,01000011 \\
- & \underline{01010100} & \xrightarrow{\textbf{2's comp}} & + \underline{10101100} \\
 & & & 11101111 \xrightarrow{\textbf{2's comp}} \\
 & & & 00010001
\end{array}
$$

- The carry of 0 indicates that a correction of the result is required.
- Result = – (00010001)

Signed 2's Complement Arithmetic Examples

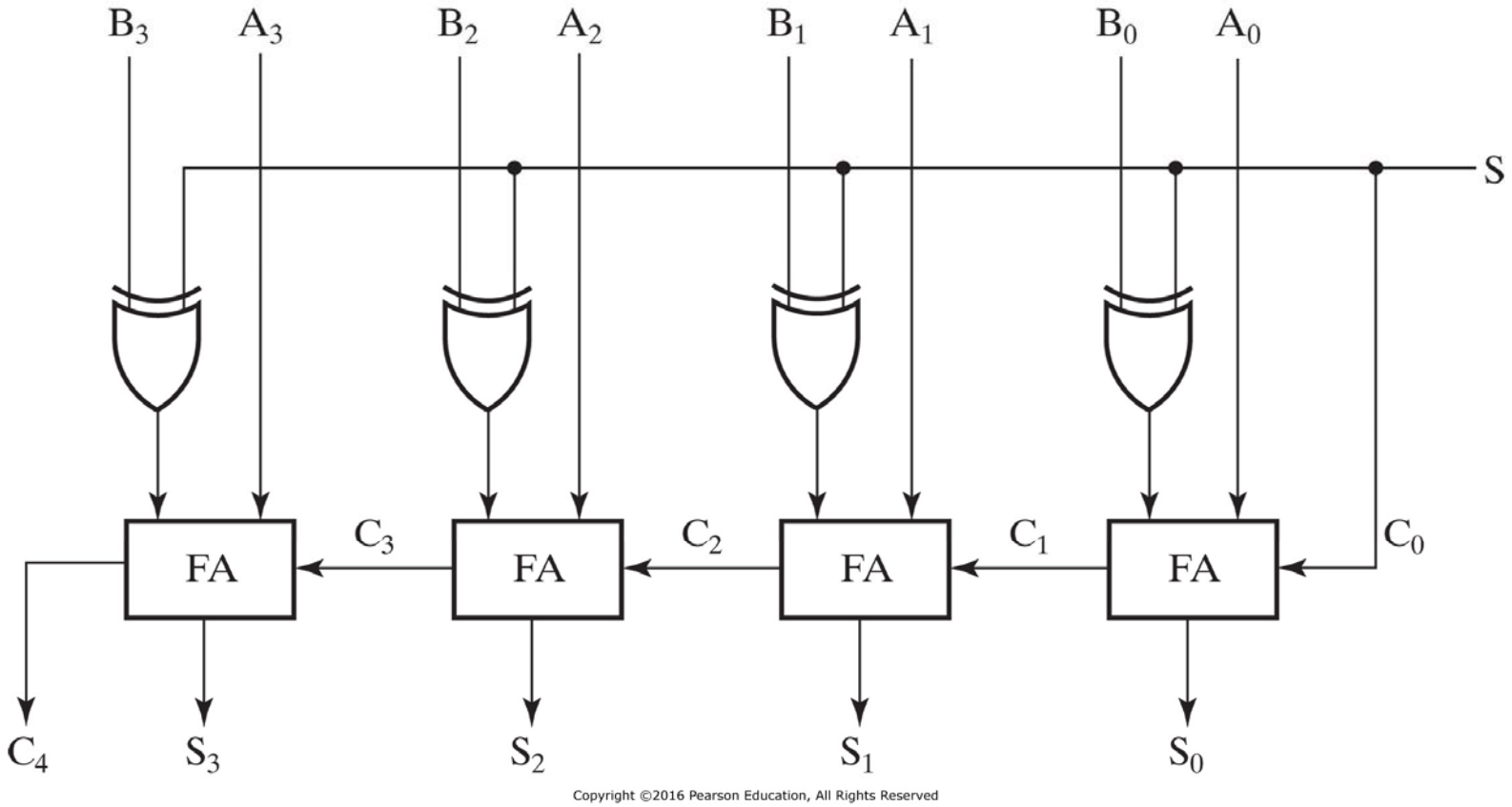
- Signed binary addition using 2s complement

+ 6	00000110	− 6	11111010	+ 6	00000110	− 6	11111010
+ 13	00001101	+ 13	00001101	− 13	11110011	− 13	11110011
+ 19	00010011	+ 7	00000111	− 7	11111001	− 19	11101101

- Signed binary subtraction using 2s complement

− 6	11111010	11111010	+ 6	00000110	00000110
− (−13)	− 11110011	+ 00001101	− (–13)	− 11110011	+ 00001101
+ 7		00000111	+ 19		00010011

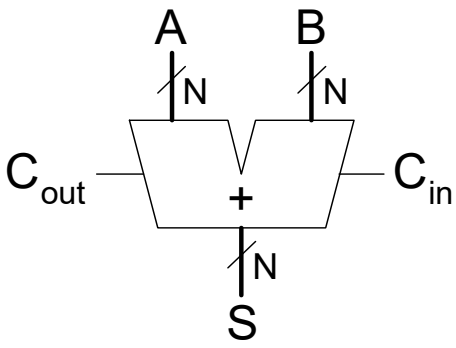
4-Bit Binary Adder-Subtractors

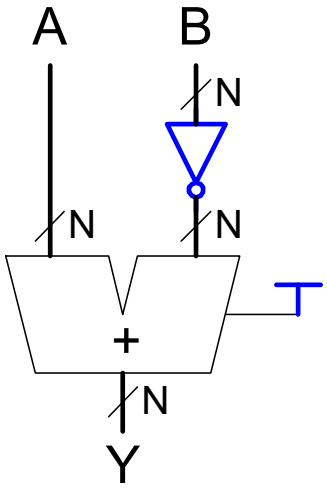


- When S=0: Addition (A+B)
- When S=1: Subtraction (A+2's complement of B)
- Can be used to add/subtract unsigned numbers and signed 2's complement numbers

Addition/Subtraction

- Both can be handled by using 2's complement representation
- Can achieve a unified implementation
  - Addition vs. subtraction
  - Unsigned vs. signed





- Corner cases shall be considered
  - Some important flags

Carry & Overflow

- Carry is important when…
  - Adding or subtracting ***unsigned integers***
  - Indicates that the unsigned sum is out of range
  - Either < 0 or > maximum unsigned n-bit value
- Overflow is important when…
  - Adding or subtracting ***signed integers***
  - Indicates that the signed sum is out of range

- Overflow occurs when?

Signed Overflow

- With two's complement and a 4-bit adder, for example, the largest representable decimal number is +7, and the smallest is -8.
- What if you try to compute 4 + 5, or (-4) + (-5)?

```
  0100   (+4)          1100   (-4)
+ 0101   (+5)        + 1011   (-5)
-------              -------
 01001   (-7)         10111   (+7)
```

- We cannot just include the carry out to produce a five-digit result, as for unsigned addition. If we did, (-4) + (-5) would result in +23!
- Also, unlike the case with unsigned numbers, the carry out cannot be used to detect overflow, by itself
  - In the example on the left, the carry out is 0 but there is overflow.
  - Conversely, there are situations where the carry out is 1 but there is no overflow.

How to Detect Signed Overflow?

- The impact of carry and overflow

Expression	Result	Carry?	Overflow?	Correct Result?
0100(+4)+0010(+2)	0110(+6)	No	No	Yes
0100(+4)+0110(+6)	1010(-6)	No	Yes	No
1100(-4)+1110(-2)	1010(-6)	Yes	No	Yes
1100(-4)+1010(-6)	0110(+6)	Yes	Yes	No

Examples of four *signed* additions

- The easiest way to detect signed overflow is to look at all the sign bits.

```
  0100   (+4)        1100   (-4)
+ 0101   (+5)      + 1011   (-5)
-------            -------
 01001   (-7)       10111   (+7)
```

Detecting Signed Overflow

- Overflow occurs only in the two situations:
  - Adding two positive numbers and the sum is negative
  - Adding two negative numbers and the sum is positive
  - Can happen because of the fixed number of sum bits
- Overflow cannot occur if you add a positive number to a negative number. Do you see why?

- In two's complement addition/subtraction
  - If the two numbers have the same sign bit and the sum/difference has a different sign bit, then overflow
  - Or, if the carry out flags of the sign bit and the highest value bit are different

Overflow Detection

- For unsigned number
  - Add
    - The carry of 1 indicates overflow
  - Subtraction
    - The carry of 1 indicates that no correction of the result is required
    - The carry of 0 indicates that a correction of the result is required

- For signed number
  - $V = C_n \oplus C_{n\text{-}1}$

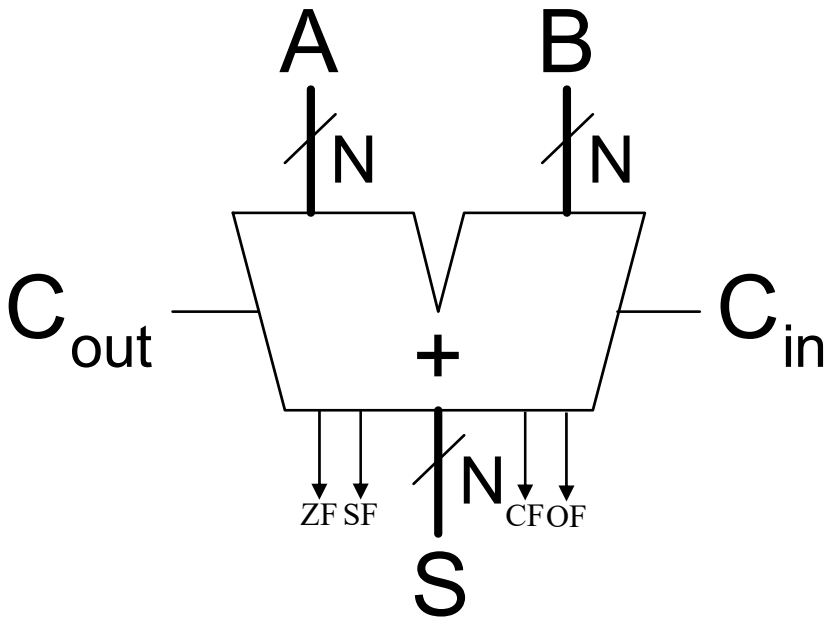
Important Flags

- Zero flag (ZF)
  - ZF = 1 means the result is 0
  - Valid for both unsigned and signed operations
- Sign flag (SF/NF)
  - The sign of the result, i.e., S_{n-1}
  - Valid for signed operations
- Carry/borrow flag (CF)
  - If CF = 1
    - Carry for addition, i.e., C_{out}
    - Borrow for subtraction, i.e., $\sim C_{out}$
  - Valid for unsigned operations
- Overflow flag (OF)
  - Valid for signed operations

Adders with Flags

- ZF, SF, CF and OF

Symbol

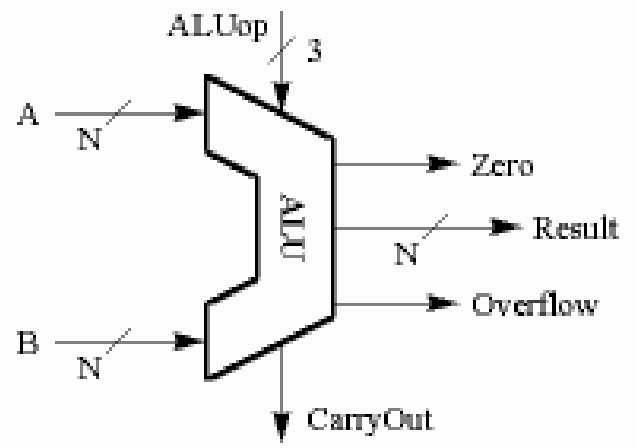


Overview

- Addition
- Subtraction
- Arithmetic logic unit (ALU)
- Multiplication
- Division
- Floating number operations

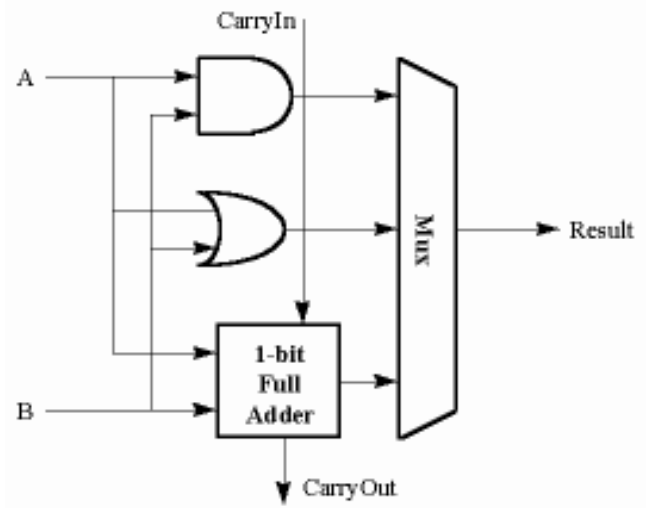
What is ALU?

- An arithmetic-logic unit, or ALU, performs many different arithmetic and logic operations. The ALU is the “heart” of a processor—you could say that everything else in the CPU is there to support the ALU.
- Two methods constitute the ALU
  - extended the adder
  - Parallel redundant select



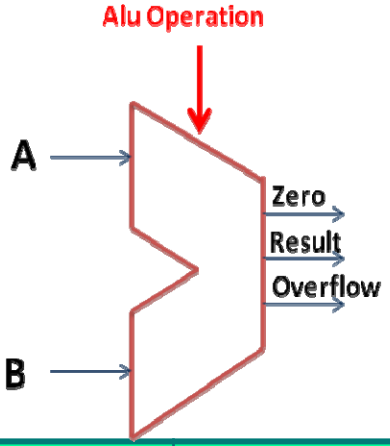
A Simple Example: 1-Bit ALU with 3 OPs

$F_{1:0}$	Function
00	A & B
01	A | B
10	A + B
11	not used

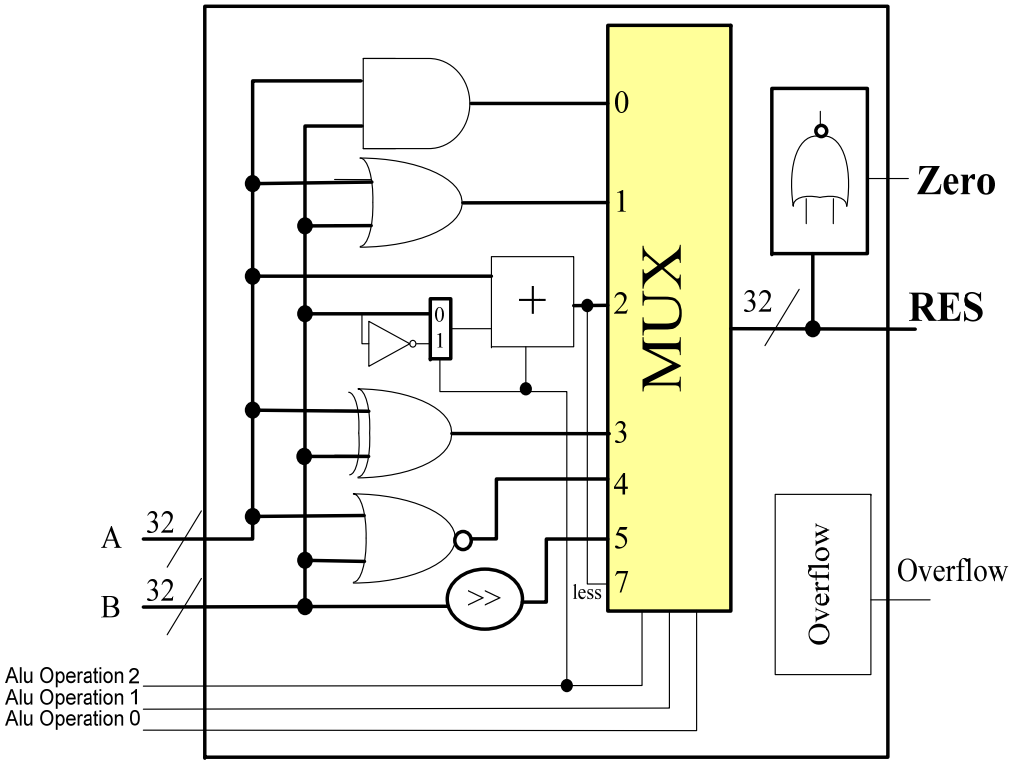



```
module ALU(
    input [3:0] a, b,
    input [1:0] aluop,
    input cin,
    output [3:0] f,
    output OF, SF, ZF, CF,
    output cout
);
    wire [3:0] sum;
    CLA_FLAGS(a, b, cin, sum, OF, SF, ZF,
CF, cout);
    always @(*) begin
        case(aluop)
            2'b00: f = a & b;
            2'b01: f = a | b;
            2'b10: f = sum;
            default: f = 0;
        endcase
    end
endmodule
```

N-Bit ALU with 8 OPs

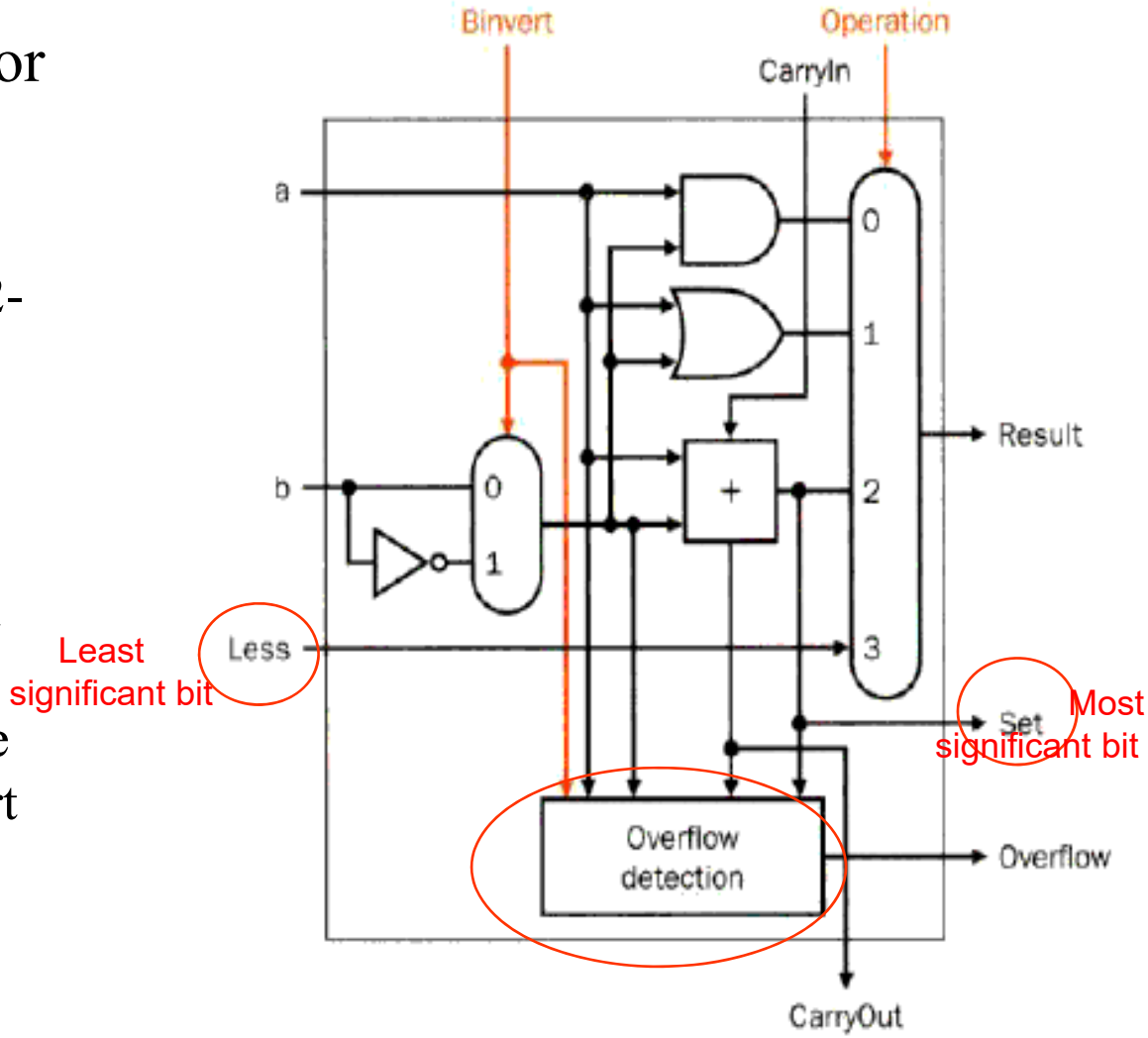


ALU Control Lines	Function
000	And
001	Or
010	Add
110	Sub
111	SLT
100	nor
101	srl
011	xor



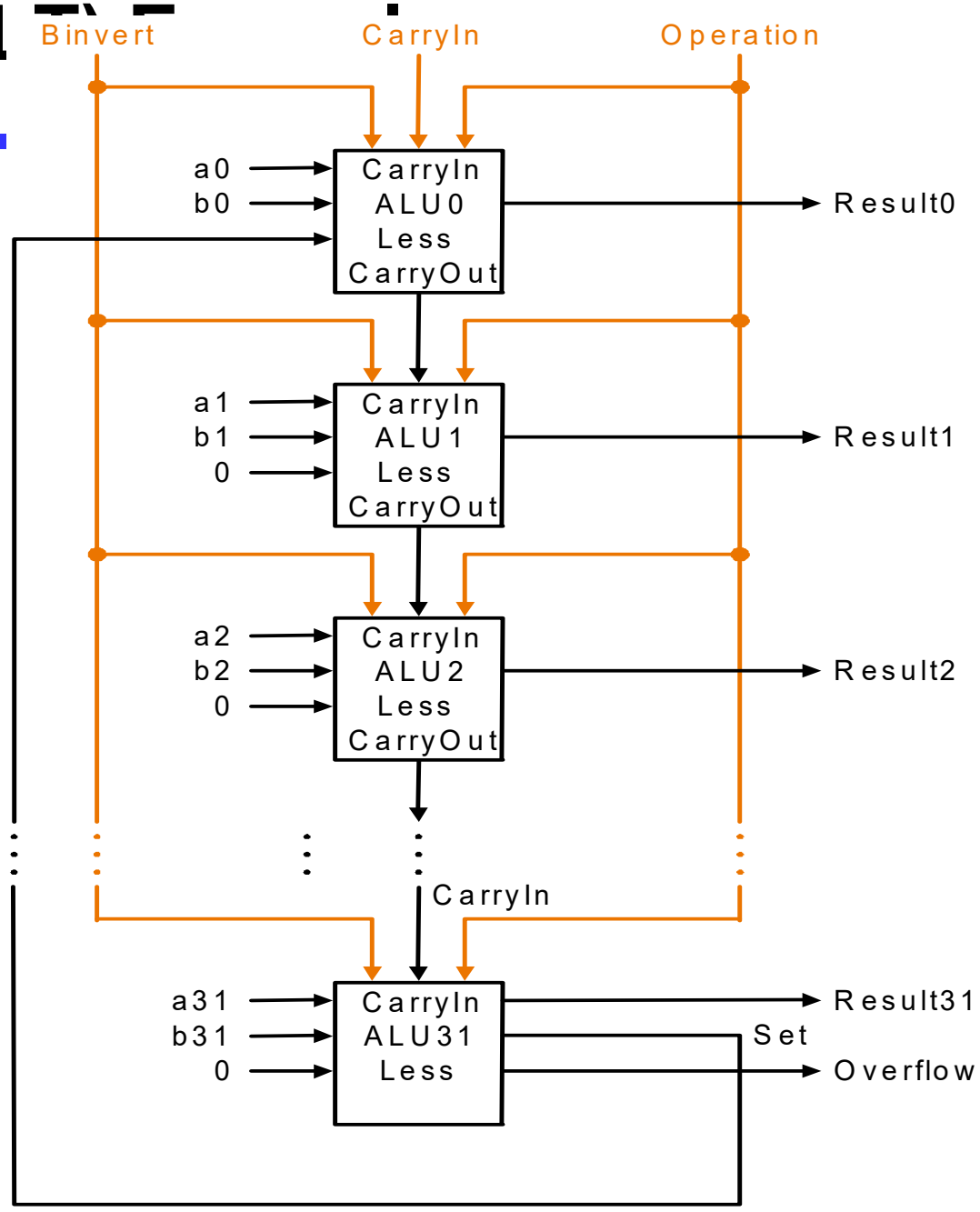
Set Less Than (SLT) Example

- Configure 32-bit ALU for SLT operation: A = 25 and B = 32
  - A < B, so Y should be 32-bit representation of 1 (0x00000001)
  - slt rd,rs,rt
  - If rs < rt, rd=1, else rd=0
  - For rd, all bits = 0 except the least significant
  - Subtraction (rs - rt), if the result is negative→ rs < rt
  - **Use of sign bit as indicator**



Set Less Than (SLT)

- Configure 32-bit ALU for SLT operation: A = 25 and B = 32
  - A < B, so Y should be 32-bit representation of 1 (0x00000001)
  - $F_{2:0} = 111$
    - $F_2 = 1$ (adder acts as subtracter), so 25 - 32 = -7
    - -7 has 1 in the most significant bit ($S_{31} = 1$)
    - $F_{1:0} = 11$ multiplexer selects $Y = S_{31}$ (zero extended) = 0x00000001.



Various Use of Shifters

- Logical shifter: shifts value to left or right and fills empty spaces with 0's
  - 11001 >> 2 = 00110
  - 11001 << 2 = 00100
- Arithmetic shifter: same as logical shifter, but on right shift, fills empty spaces with the old most significant bit (MSB).
  - 11001 >>> 2 = 11110
  - 11001 <<< 2 = 00100
- Rotator: rotates bits in a circle, such that bits shifted off one end are shifted into the other end
  - 11001 ROR 2 = 01110
  - 11001 ROL 2 = 00111

Shifters as Multipliers and Dividers

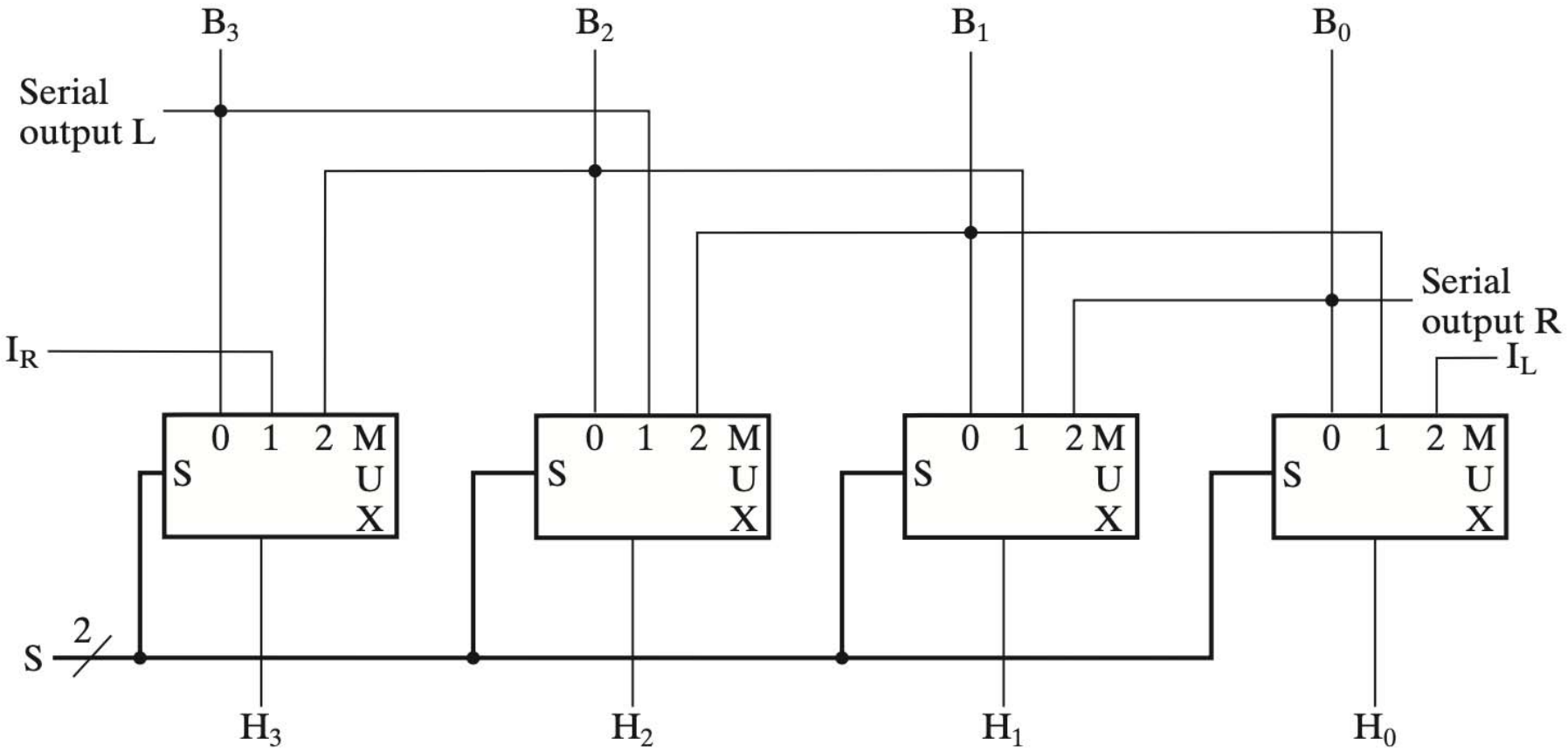
- $A << N = A \times 2^N$
  - E.g., 00001 << 2 = 00100 $(1 \times 2^2 = 4)$
  - E.g., 11101 << 2 = 10100 $(-3 \times 2^2 = -12)$

- $A >>> N = A \div 2^N$
  - E.g., 01000 >>> 2 = 00010 $(8 \div 2^2 = 2)$
  - E.g., 10000 >>> 2 = 11100 $(-16 \div 2^2 = -4)$

Recall the Design of A Shifter

- Bidirectional shift registers with parallel load
  - Data from Bus B can be transferred to the register in parallel and then shifted to the right, the left, or not at all.
  - A clock pulse loads the output of Bus B into the shift register, and a second clock pulse performs the shift.
  - Finally, a third clock pulse transfers the data from the shift register to the selected destination register.
- Alternatively, the transfer from a source register to a destination register can be done using *only one clock pulse* if the shifter is implemented ***as a combinational circuit***
  - Because of the faster operation that results from the use of one clock pulse instead of three, this is the preferred method.
  - In a combinational shifter, the signals propagate through the gates without the need for a clock pulse.
  - Hence, the only clock needed for a shift in the datapath is for loading the data from Bus H into the selected destination register.

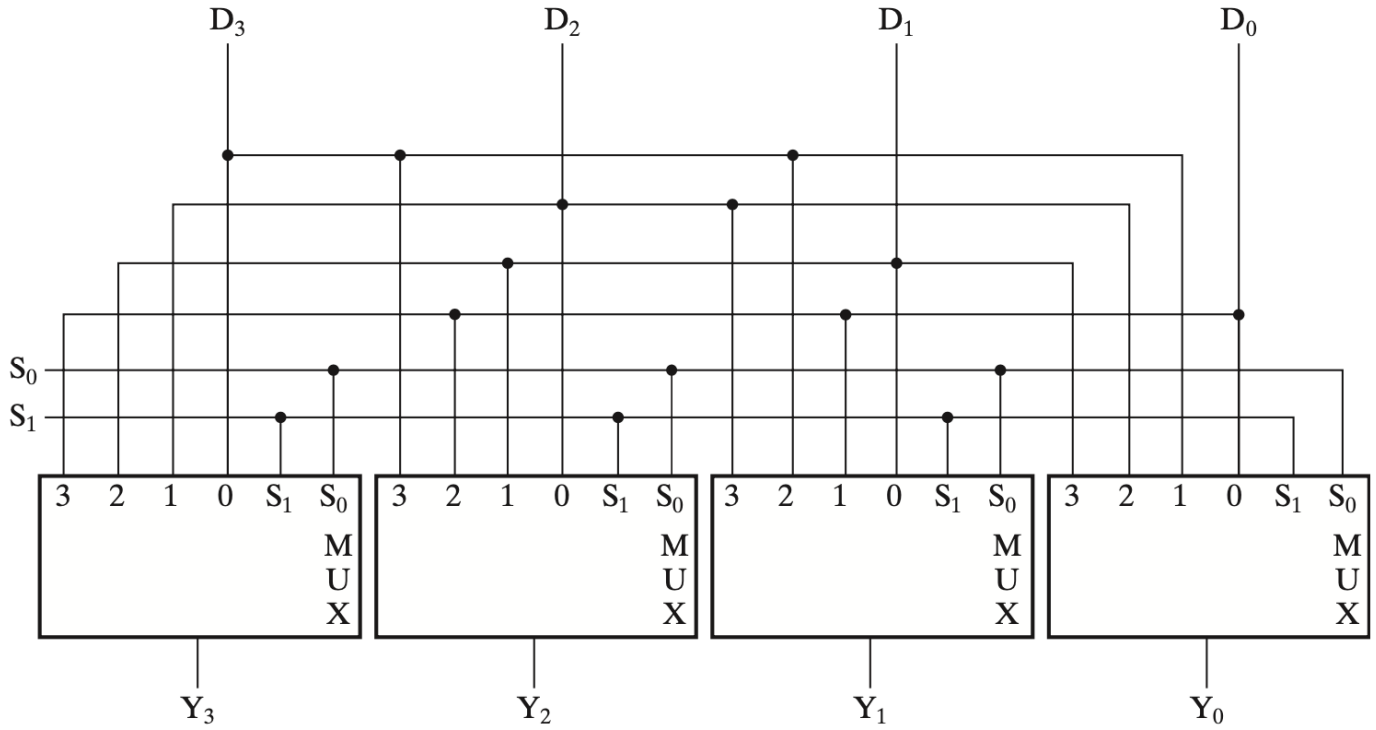
Shifter Design as A Combinational Circuit



4-Bit Barrel Shifter (ROL)

Function Table for 4-Bit Barrel Shifter

Select		Output				
S_1	S_0	Y_3	Y_2	Y_1	Y_0	Operation
0	0	D_3	D_2	D_1	D_0	No rotation
0	1	D_2	D_1	D_0	D_3	Rotate one position
1	0	D_1	D_0	D_3	D_2	Rotate two positions
1	1	D_0	D_3	D_2	D_1	Rotate three positions



Overview

- Addition
- Subtraction
- Arithmetic logic unit (ALU)
- Multiplication
- Division
- Floating number operations

Multiplication

- More complicated than addition
  - A straightforward implementation will involve shifts and adds
- More complex operation can lead to
  - More area (on silicon) and/or
  - More time (multiple cycles or longer clock cycle time)
- Let's begin from a simple, straightforward method

Decimal vs. Binary Multiplication

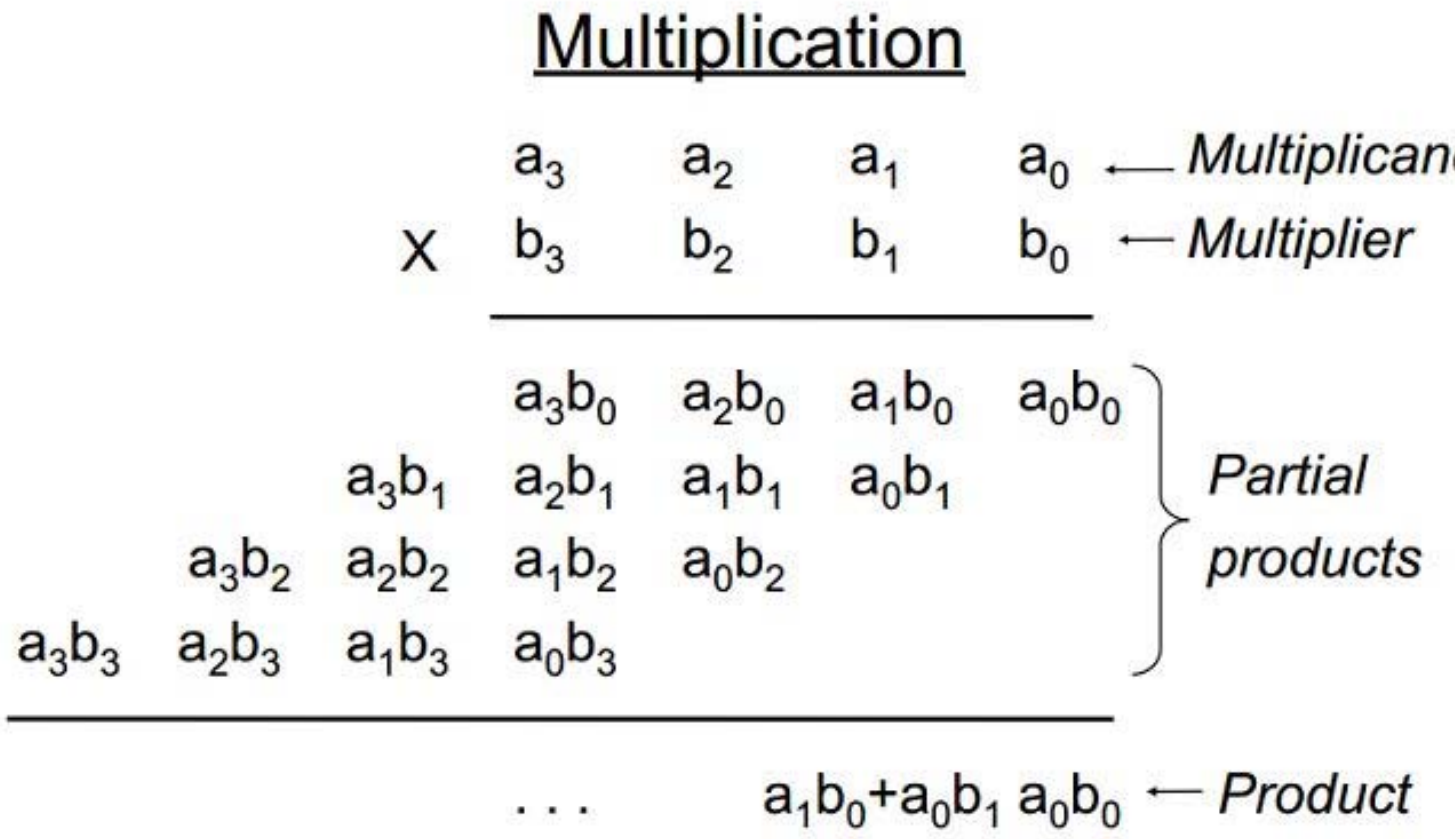
- Partial products formed by multiplying a single digit of the multiplier with multiplicand
- Shifted partial products summed to form result

Decimal		Binary
230	multiplicand	0101
x 42	multiplier	x 0111
460	partial products	0101
+ 920		0101
9660		0101
		+ 0000
	result	0100011

230 x 42 = 9660

5 x 7 = 35

Unsigned Multiplication: 4-bit X 4-bit



Unsigned Multiplication: M-bit X N-bit

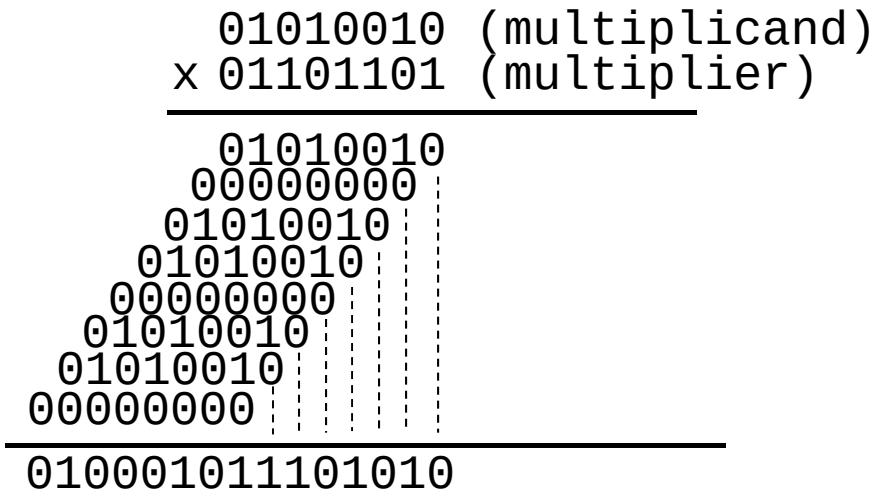
$$A = a_{m-1} \ldots a_1 a_0 \qquad B = b_{n-1} \ldots b_1 b_0$$

$$a = \sum_{i=0}^{m-1} a_i 2^i \qquad b = \sum_{j=0}^{n-1} b_j 2^j$$

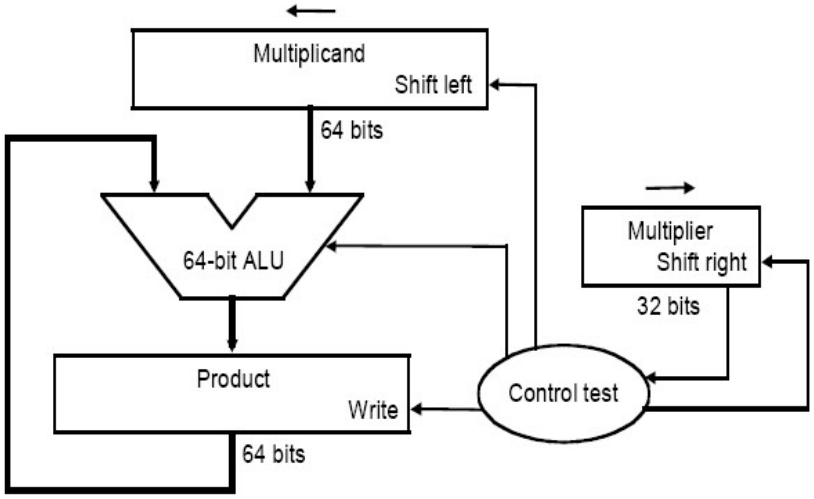
$$p = ab = \left(\sum_{i=0}^{m-1} a_i 2^i\right)\left(\sum_{j=0}^{n-1} b_j 2^j\right) = \sum_{i=0}^{m-1}\sum_{j=0}^{n-1} (a_i b_j) 2^{i+j} = \sum_{k=0}^{m+m+1} p_k 2^k$$

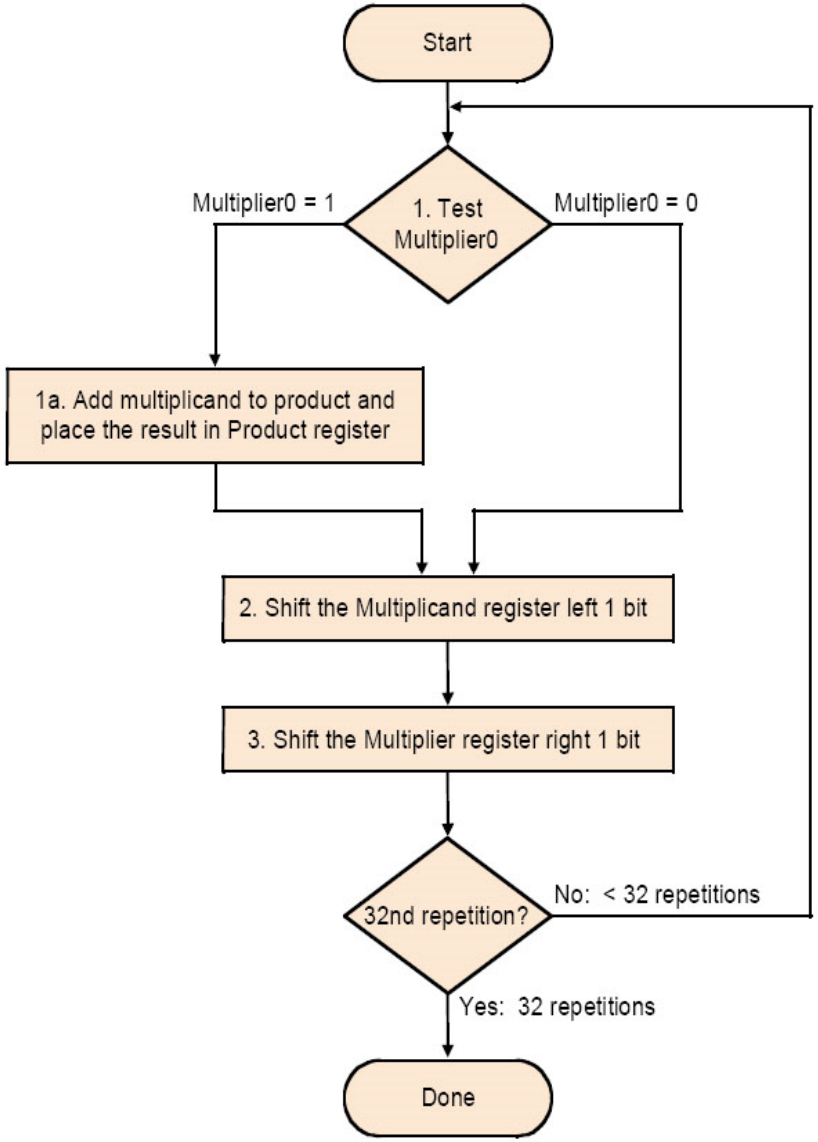
$$\begin{array}{rccccccc}
 & a_{m-1} & a_{m-2} & \cdots & a_1 & a_0 & = A \\
\times) & & b_{n-1} & \cdots & b_1 & b_0 & = B \\
\hline
 & a_{m-1}b_0 & a_{m-2}b_0 & \cdots & a_1b_0 & a_0b_0 & \\
 & a_{m-1}b_1 & a_{m-2}b_1 & \cdots & a_1b_1 & a_0b_1 & \\
 & \vdots & \vdots & & \vdots & \vdots & \\
+) & a_{m-1}b_{n-1} & a_{m-2}b_{n-1} & \cdots & a_1b_{n-1} & a_0b_{n-1} & \\
\hline
p_{m+n-1} & p_{m+n-2} & p_{m+n-3} & \cdots & p_{n-1} & \cdots & p_1 \quad p_0 = P
\end{array}$$

Straightforward Algorithm



Implementation 1





Example of Implementation 1

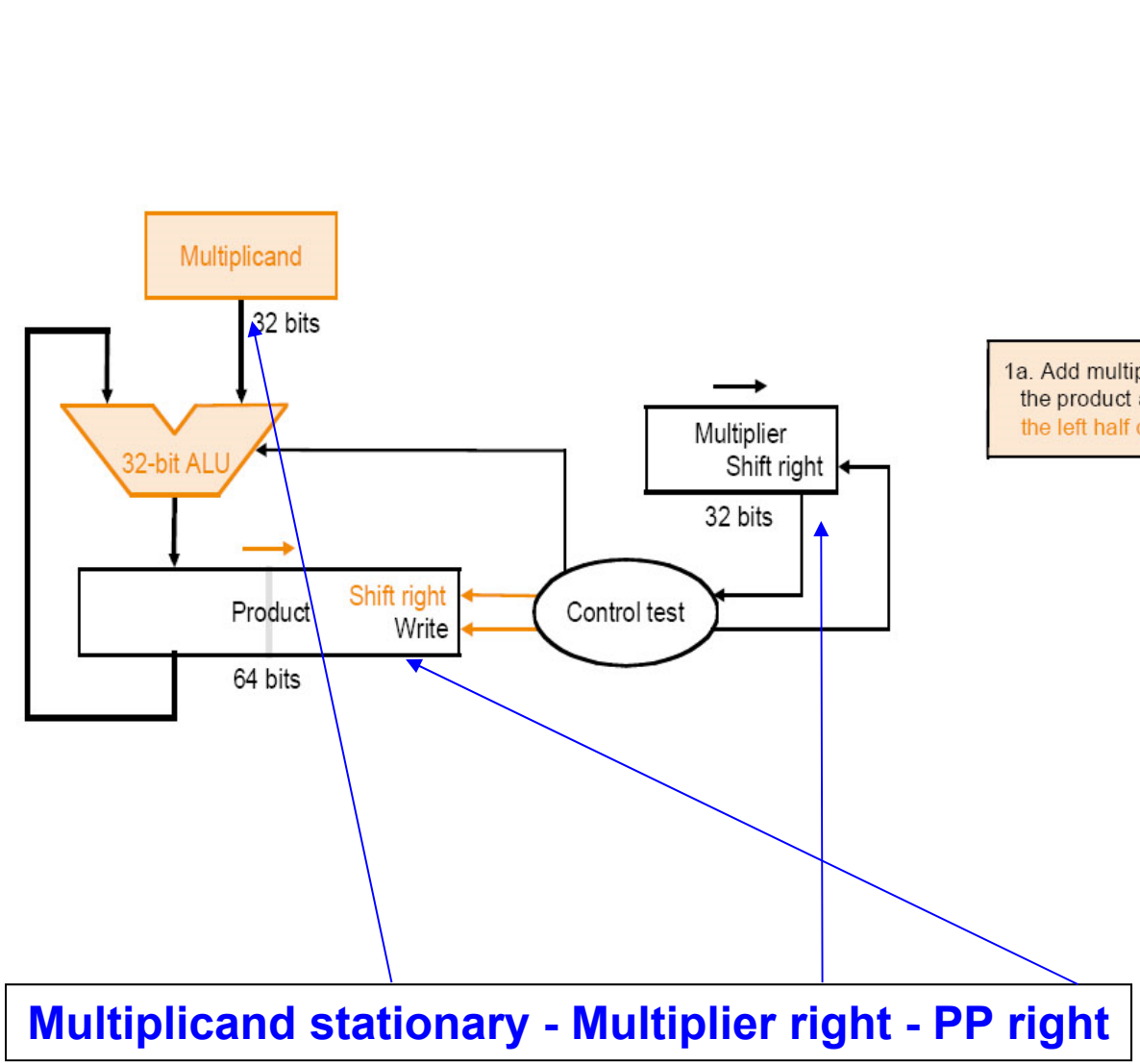
- Let's do 0010 x 0110 (2 x 6), unsigned

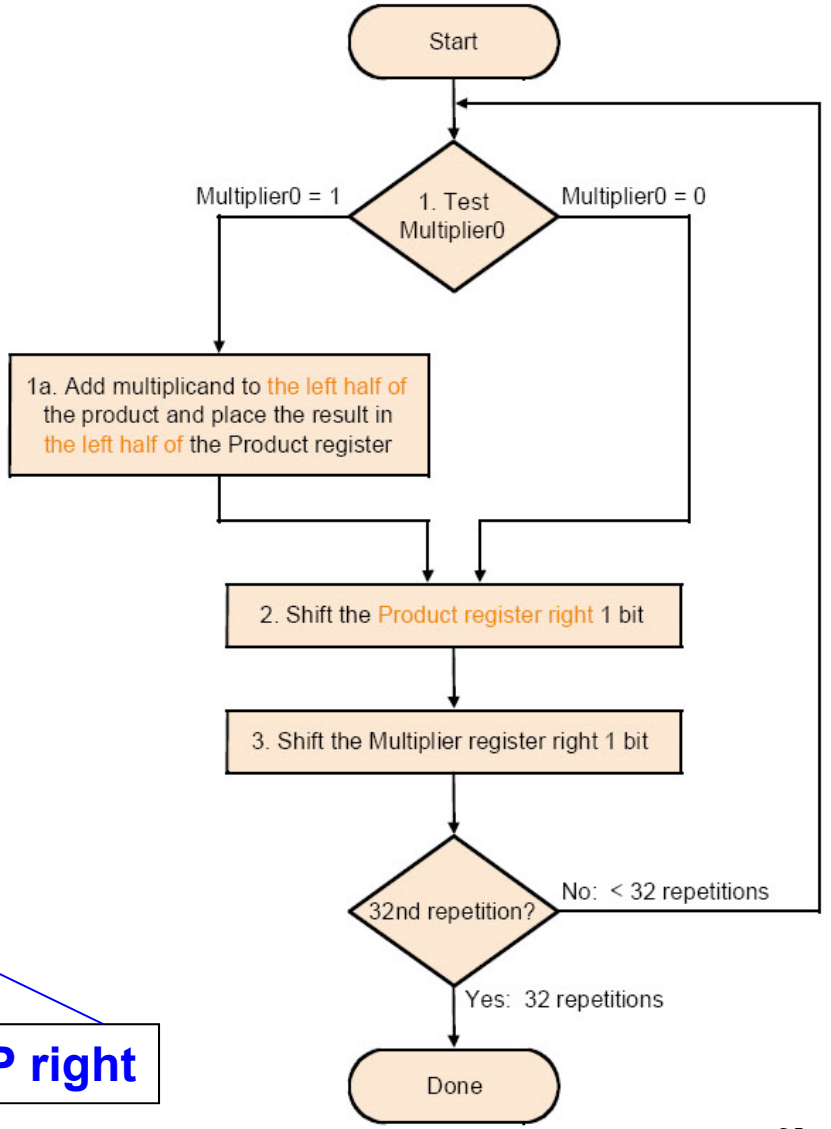
Iteration	Implementation 1			
	Step	Multiplier	Multiplicand	Product
0	initial values	0110	0000 0010	0000 0000
1	1: 0 -> no op	0110	0000 0010	0000 0000
	2: Multiplier shift right/ Multiplicand shift left	011	0000 0100	0000 0000
2	1: 1 -> product = product + multiplicand	011	0000 0100	0000 0100
	2: Multiplier shift right/ Multiplicand shift left	01	0000 1000	0000 0100
3	1: 1 -> product = product + multiplicand	01	0000 1000	0000 1100
	2: Multiplier shift right/ Multiplicand shift left	0	0001 0000	0000 1100
4	1: 0 -> no op	0	0001 0000	0000 1100
	2: Multiplier shift right/Multiplicand shift left		0010 0000	

Drawbacks

- The ALU is twice as wide as necessary
- The multiplicand register takes twice as many bits as needed
- The product register won't need 2n bits till the last step
  - Being filled
- The multiplier register is being emptied during the process

Implementation 2





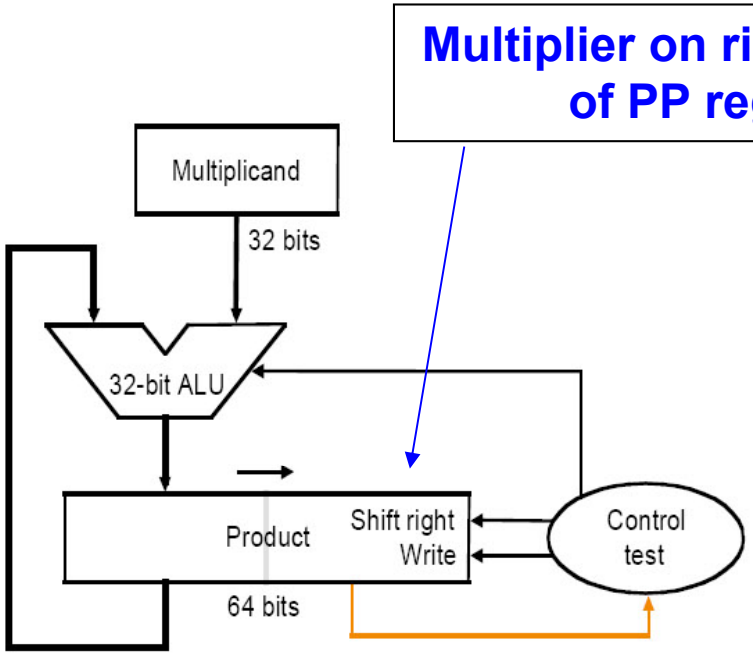
Multiplicand stationary - Multiplier right - PP right

Example of Implementation 2

- Let's do 0010 x 0110 (2 x 6), unsigned

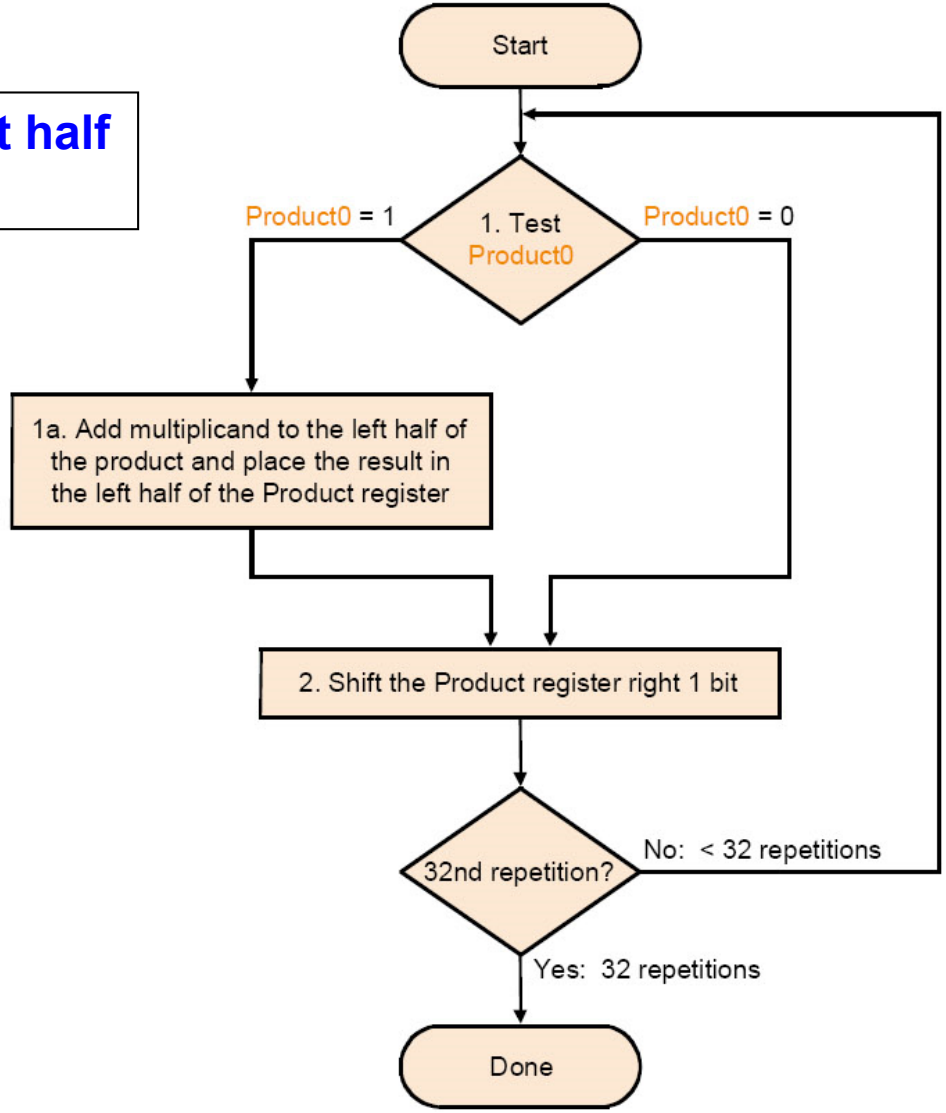
Iteration	Implementation 2			
	Step	Multiplier	Multiplicand	Product
0	initial values	0110	0010	0000 $_{\times\times\times\times}$
1	1: 0 -> no op	0110	0010	0000 $_{\times\times\times\times}$
	2: Multiplier shift right/ Product shift right	$_{\times}$011	0010	0000 0$_{\times\times\times}$
2	1: 1 -> product = product + multiplicand	$_{\times}$011	0010	0010 0$_{\times\times\times}$
	2: Multiplier shift right/ Product shift right	$_{\times\times}$01	0010	0001 00$_{\times\times}$
3	1: 1 -> product = product + multiplicand	$_{\times\times}$01	0010	0011 00$_{\times\times}$
	2: Multiplier shift right/ Product shift right	$_{\times\times\times}$0	0010	0001 100$_{\times}$
4	1: 0 -> no op	$_{\times\times\times}$0	0010	0001 100$_{\times}$
	2: Multiplier shift right/ Product shift right	$_{\times\times\times\times}$	0010	0000 1100

Implementation 3



Multiplier on right half of PP reg

Multiplicand stationary - Multiplier right - PP right



Example of Implementation 3

- Let's do 0010 x 0110 (2 x 6), unsigned

Iteration	Implementation 3			
	Step	Multiplier	Multiplicand	Product|Multiplier
0	initial values	0110	0010	0000 0110
1	1: 0 -> no op	0110	0010	0000 0110
	2: Multiplier shift right/ Product shift right	×011	0010	0000 0011
2	1: 1 -> product = product + multiplicand	×011	0010	0010 0011
	2: Multiplier shift right/ Product shift right	××01	0010	0001 0001
3	1: 1 -> product = product + multiplicand	××01	0010	0011 0001
	2: Multiplier shift right/ Product shift right	××00	0010	0001 1000
4	1: 0 -> no op	×××0	0010	0001 1000
	2: Multiplier shift right/ Product shift right	××××	0010	0000 1100

Signed Multiplication

- Basic approach:
  - Store the signs of the operands
  - Convert signed numbers to unsigned numbers (most significant bit (MSB) = 0)
  - Perform multiplication
  - If sign bits of operands are equal
    sign bit = 0, else
    sign bit = 1
- Improved method:
  Booth's Algorithm

Assumption: addition and subtraction are available

Principle -- Decomposable multiplication

- Assumes： $Z = y \times 10111100$

$Z = y(10000000 + 111100 + 100 - 100)$

$= y(1 \times 2^7 + 1000000 - 100)$

$= y(1 \times 2^7 + 1 \times 2^6 - 2^2)$

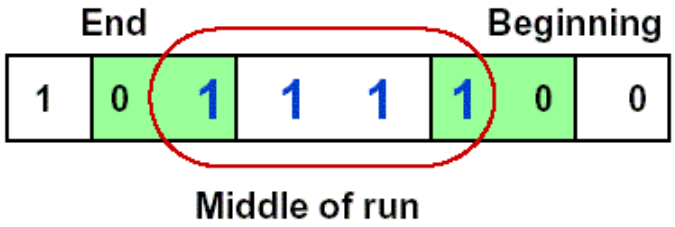
$= y(1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 - 1 \times 2^2)$

$= y(1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 - 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0)$

$= y \times 2^7 + y \times 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 - y \times 2^2 + 0 \times 2^1 + 0 \times 2^0)$

Booth's Algorithm

- Idea: If you have a sequence of '1's
  - subtract at first '1' in multiplier
  - shift for the sequence of '1's
  - add where prior step had last '1‘



- Result:
  - Possibly less additions and more shifts
  - Faster, if shifts are faster than additions

Booth's Algorithm

Current bit	Bit to right	Explanation	Example	Operation
1	0	Begins run of '1'	00001111000	Subtract
1	1	Middle of run of '1'	00001111000	Nothing
0	1	End of a run of '1'	00001111000	Add
0	0	Middle of a run of '0'	00001111000	Nothing

Example 1 of Booth's Algorithm

- Let's do 0010 x 1101 (2 x -3)

Iteration	Implementation 3		
	Step	Multiplicand	Product
0	initial values	0010	0000 1101 0
1	10 -> product = product – multiplicand	0010	1110 1101 0
	shift right		1111 0110 1
2	01 -> product = product + multiplicand	0010	0001 0110 1
	shift right		0000 1011 0
3	10 -> product = product – multiplicand	0010	1110 1011 0
	shift right		1111 0101 1
4	11 -> no op	0010	1111 0101 1
	shift right		1111 1010 1

Example 2 of Booth's Algorithm

- Negative multiplicand:
  - -6 x 6 = -36
  - 1010 x 0110, 0110 in Booth's encoding is +0-0
  - Hence:

1111 1010	x 0	0000 0000
1111 0100	x –1	0000 1100
1110 1000	x 0	0000 0000
1101 0000	x +1	1101 0000
	Final Sum:	1101 1100 (-36)

Example 3 of Booth's Algorithm

- Negative multiplier:
  - -6 x -2 = 12
  - 1010 x 1110, 1110 in Booth's encoding is 00-0
  - Hence:

1111 1010	x 0	0000 0000
1111 0100	x –1	0000 1100
1110 1000	x 0	0000 0000
1101 0000	x 0	0000 0000
	Final Sum:	0000 1100 (12)

Summary

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

Benefit: Reducing the number of partial products

Take away: Booth encoding

Exercise

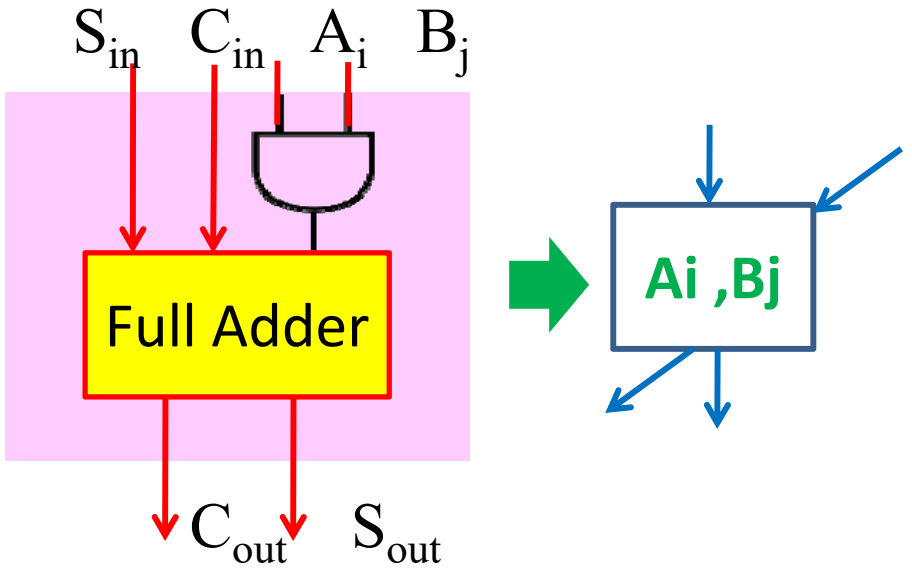
Unsigned Multiplication: x=5, y=3

Signed Multiplication: x=5, y=-7

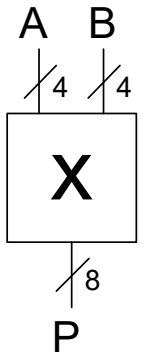
Yet Another Option: Array Multiplier

			1	0	0	0	
		x	1	0	0	1	
			1	0	0	0	
		0	0	0	0		
	0	0	0	0			
1	0	0	0				
1	0	0	1	0	0	0	

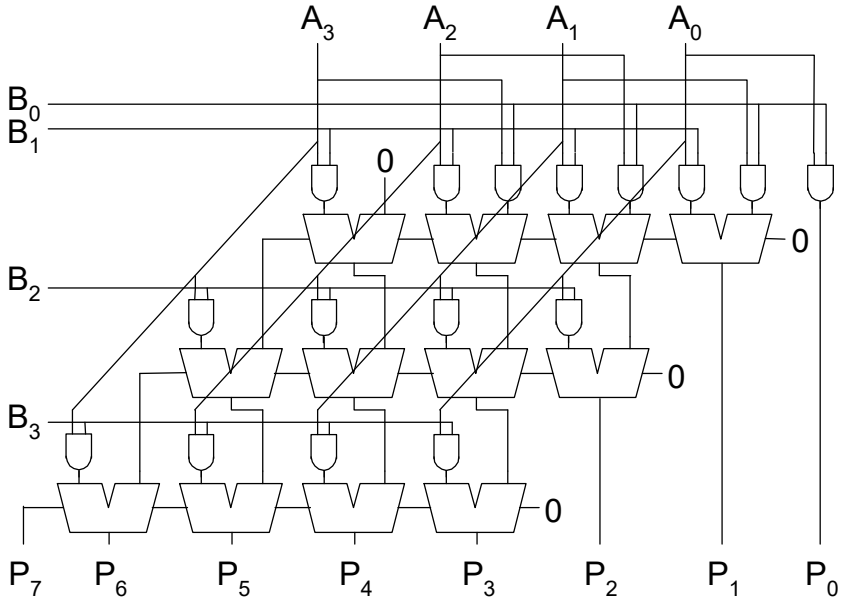
Adding all partial products simultaneously using an array of basic cells



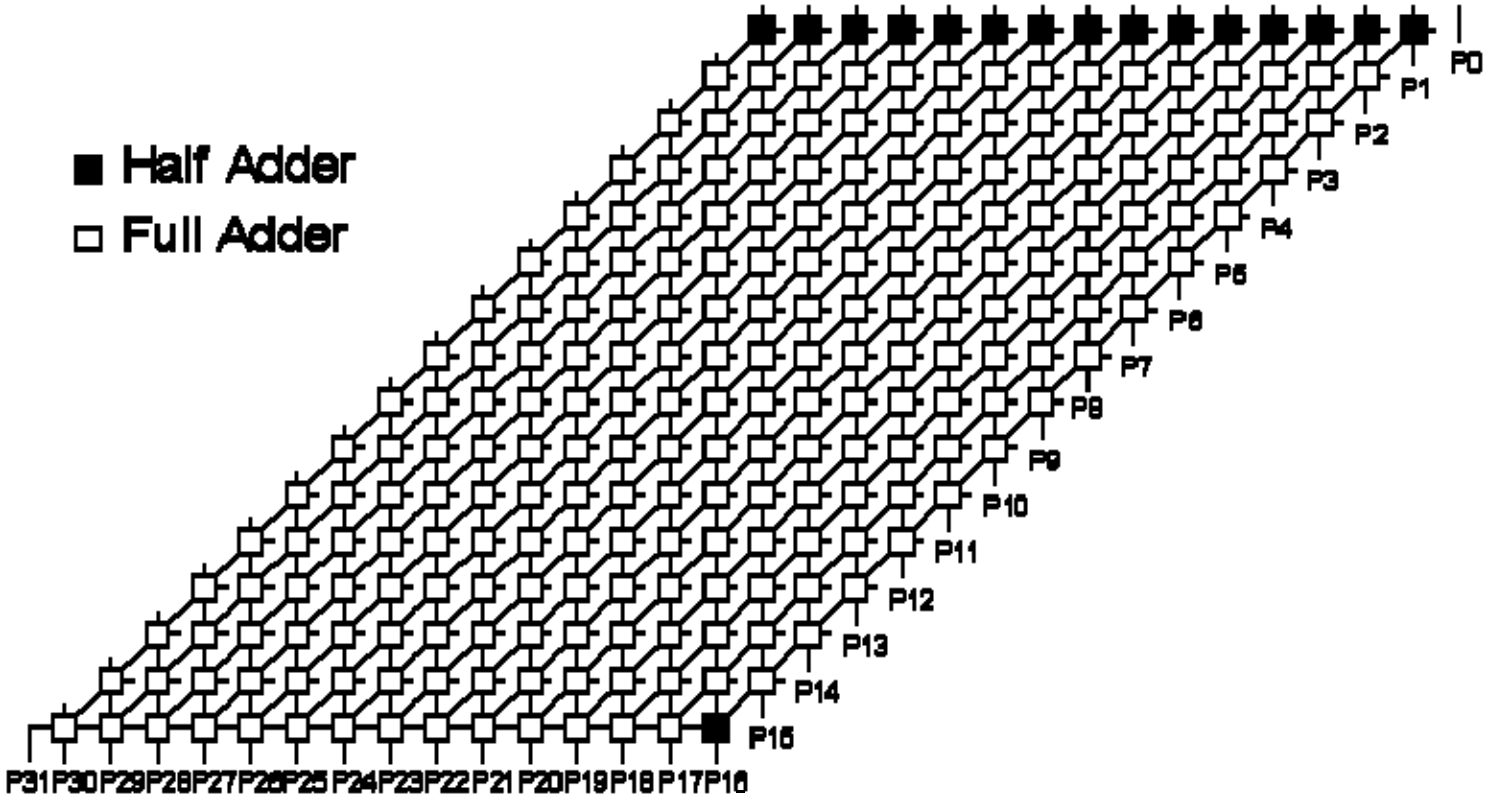
4 x 4 Array Multiplier



$$
\begin{array}{rcccccccc}
 & & & A_3 & A_2 & A_1 & A_0 & & \\
\times & & & B_3 & B_2 & B_1 & B_0 & & \\
\hline
 & & & A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 & & \\
 & & A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 & & & \\
 & A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 & & & & \\
+ & A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 & & & & \\
\hline
P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0
\end{array}
$$



16-bit Array Multiplier



Conceptually straightforward

Fairly expensive hardware, integer multiplies relatively rare

Most used in array address calc: replace with shifts

Overview

- Addition
- Subtraction
- Arithmetic logic unit (ALU)
- Multiplication
- Division
- Floating number operations

Unsigned Division

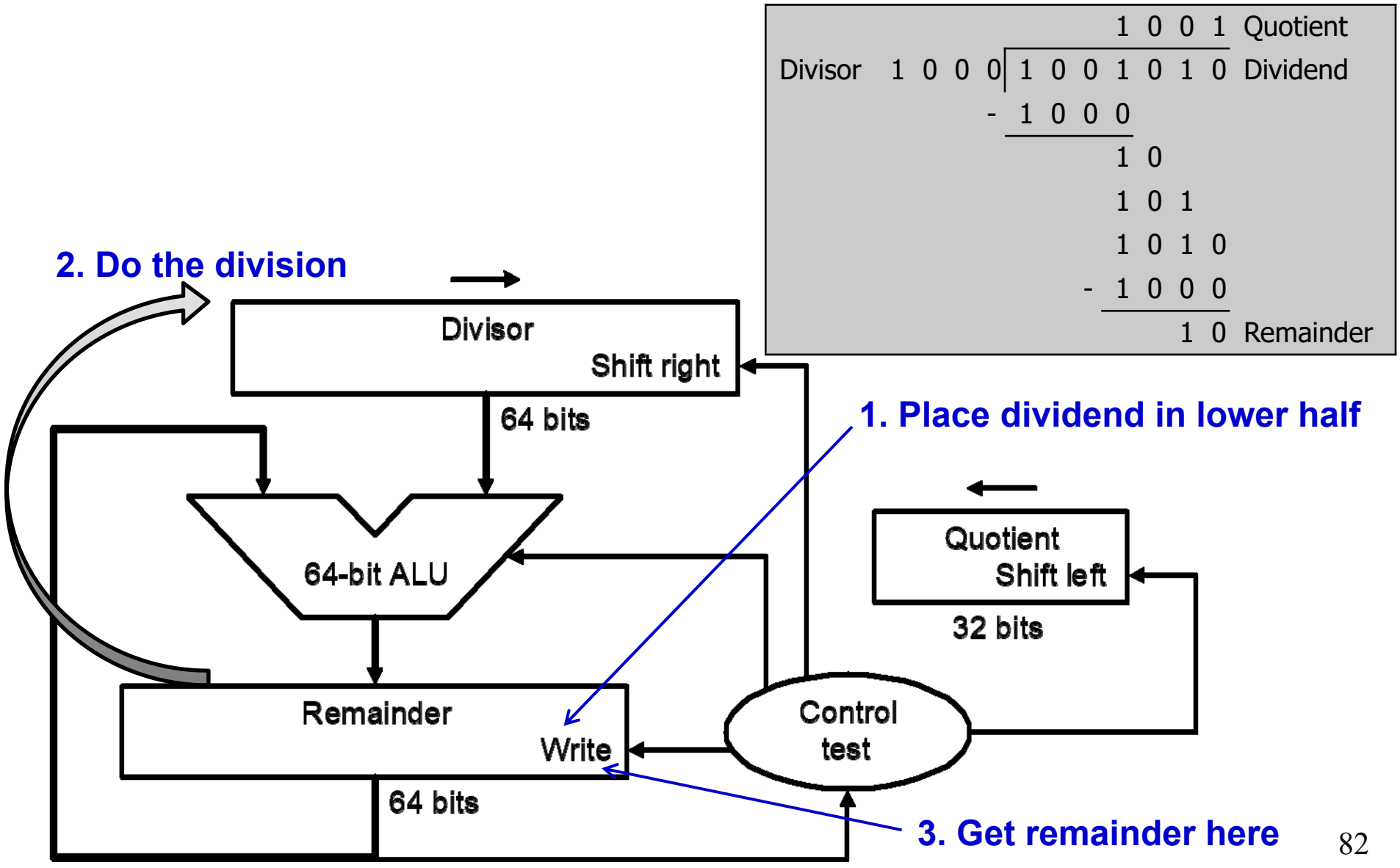
- Again, back to 3rd grade (74 ÷ 8 = 9 rem 2)

							1	0	0	1	Quotient
Divisor	1	0	0	0	1	0	0	1	0	1	0 Dividend
				-	1	0	0	0			
								1	0		
								1	0	1	
								1	0	1	0
							-	1	0	0	0
										1	0 Remainder

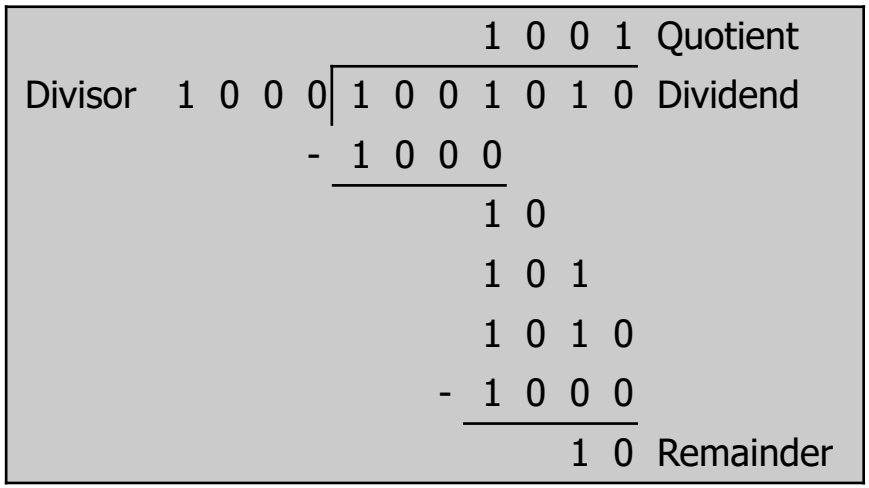
Unsigned Division

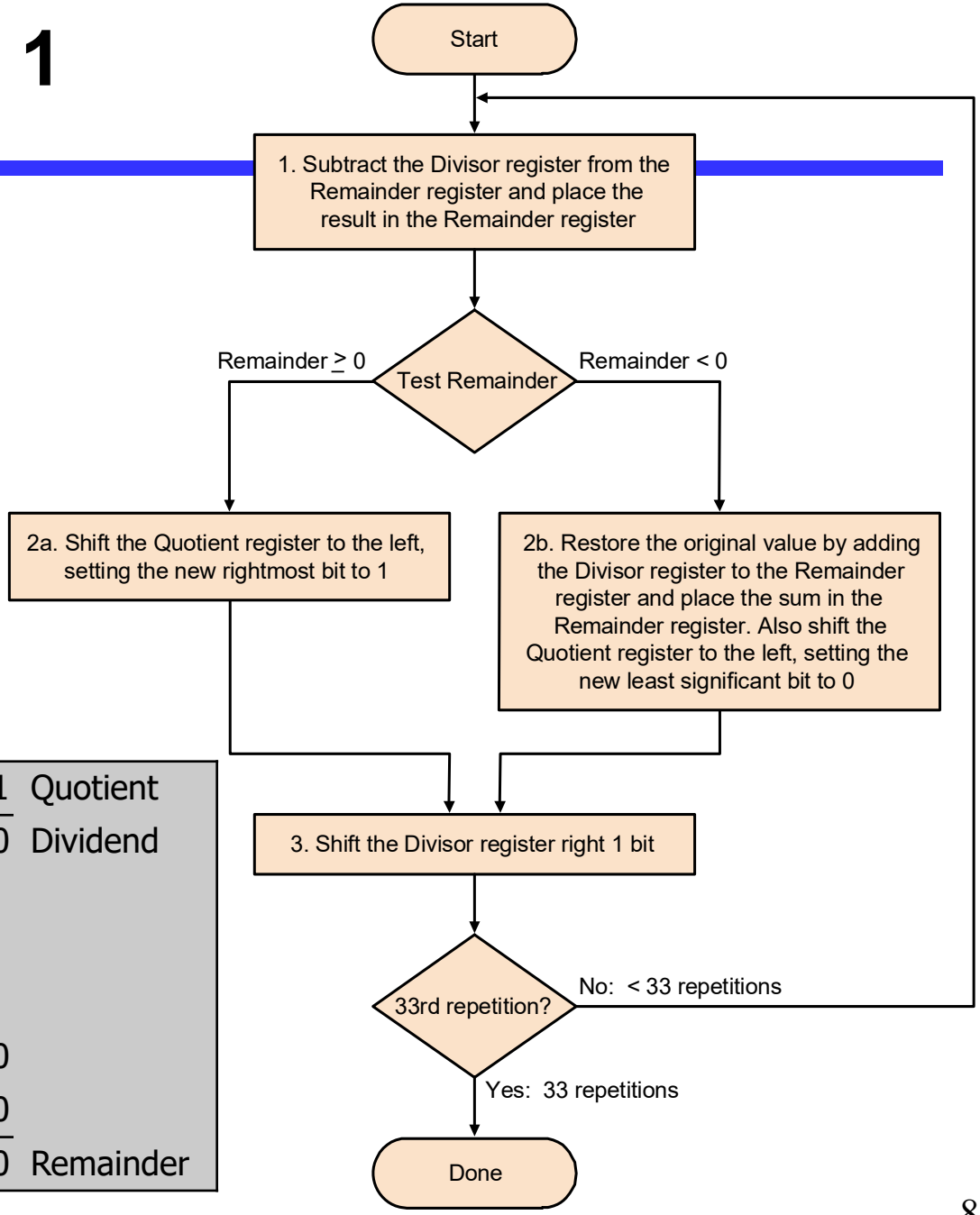
- How does hardware know if division fits?
  - Condition: if remainder ≥ divisor
  - Use subtraction: (remainder – divisor) ≥ 0
- OK, so if it fits, what do we do?
  - $\text{Remainder}_{n+1} = \text{Remainder}_{n} - \text{divisor}$
- What if it doesn't fit?
  - Have to restore original remainder
- Called **restoring division**

Implementation 1



Implementation 1





Example of Implementation 1

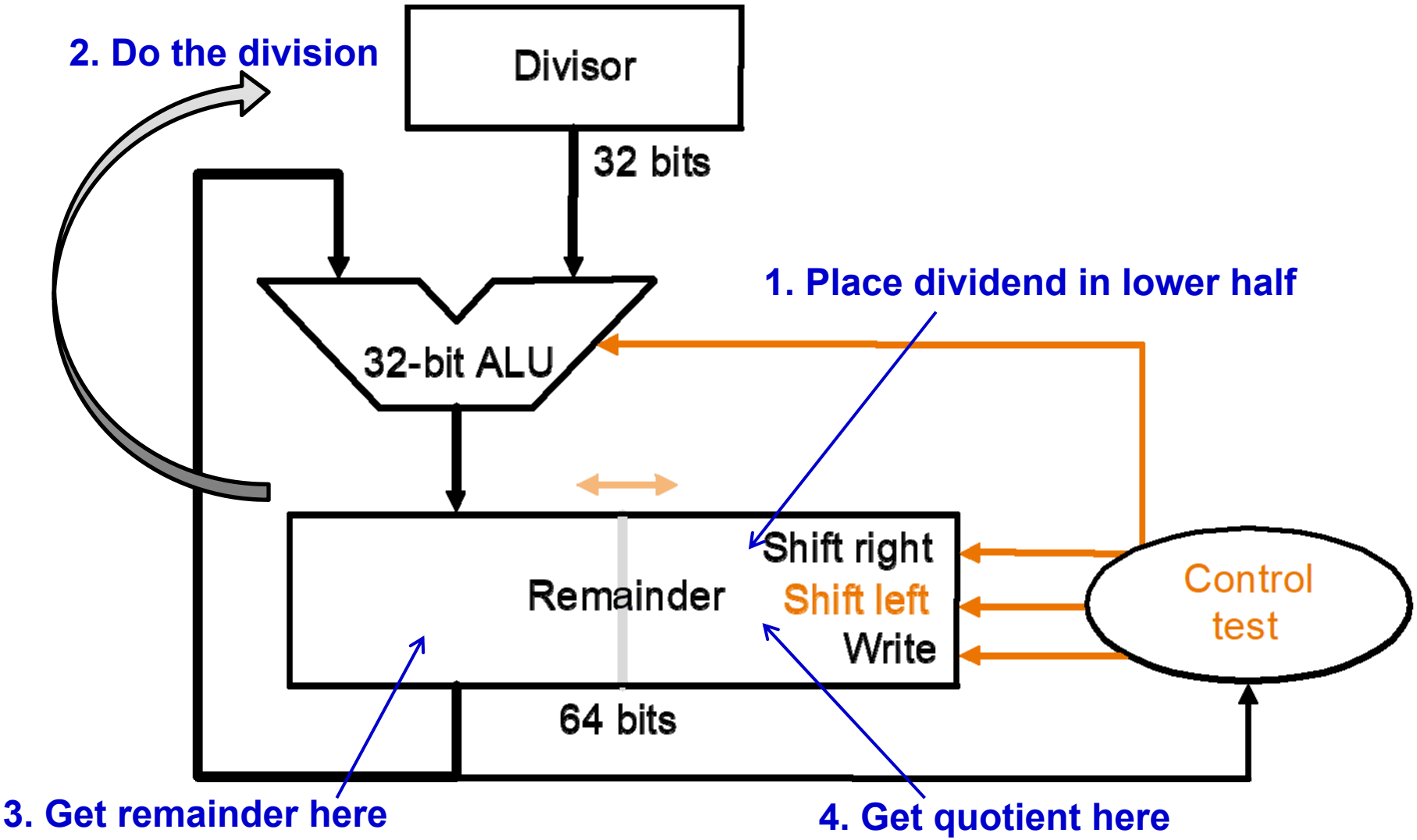
- Let's do 7 ÷ 2

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem=Rem-Div	0000	0010 0000	1110 0111
	2b: Rem<0=>+Div, sll Q, Q_0=0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem=Rem-Div	0000	0001 0000	1111 0111
	2b: Rem<0=>+Div, sll Q, Q_0=0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem=Rem-Div	0000	0000 1000	1111 1111
	2b: Rem<0=>+Div, sll Q, Q_0=0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem=Rem-Div	0000	0000 0100	0000 0011
	2a: Rem≥0=> sll Q, Q_0=1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem=Rem-Div	0001	0000 0010	0000 0001
	2a: Rem≥0=> sll Q, Q_0=1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

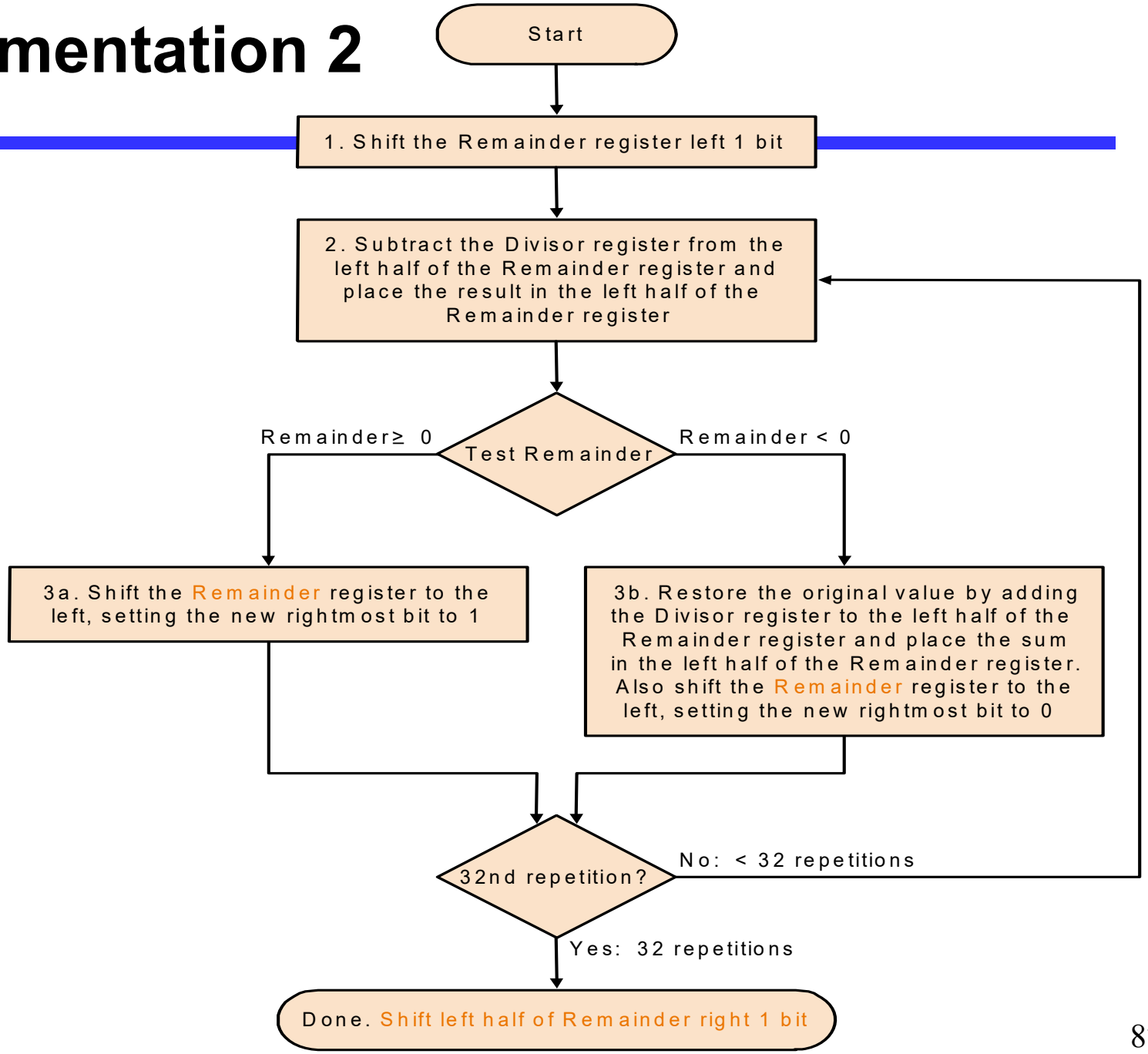
Division Improvements

- Skip first subtract
  - Can't shift '1' into quotient anyway
  - Hence shift first, then subtract
    - Undo extra shift at end
- Hardware similar to multiplier
  - Can store quotient in remainder register
  - Only need 32-bit ALU
    - Shift remainder left vs. divisor right

Implementation 2



Implementation 2



Restoring Division

Iteration	Divisor	Divide algorithm	
		Step	Remainder
0	0010	Initial values	0000 0111
	0010	Shift Rem left 1	0000 1110
1	0010	2: Rem = Rem - Div	1110 1110
	0010	3b: Rem < 0 ⇒ + Div, sll R, R0 = 0	0001 1100
2	0010	2: Rem = Rem - Div	1111 1100
	0010	3b: Rem < 0 ⇒ + Div, sll R, R0 = 0	0011 1000
3	0010	2: Rem = Rem - Div	0001 1000
	0010	3a: Rem ≥ 0 ⇒ sll R, R0 = 1	0011 0001
4	0010	2: Rem = Rem - Div	0001 0001
	0010	3a: Rem ≥ 0 ⇒ sll R, R0 = 1	0010 0011
Done	0010	shift left half of Rem right 1	0001 0011

Further Improvements

- Division still takes:
  - 2 ALU cycles per bit position
    - 1 to check for divisibility (subtract)
    - One to restore (if needed)
- Can reduce to 1 cycle per bit
  - Called **non-restoring division**
  - Avoids restore of remainder when test fails

Non-Restoring Division

- Consider remainder to be restored:

  $$\mathbf{R_i = 2 \text{ x } R_{i-1} - d < 0}$$

  - Since R_i is negative, we must restore it, right?
  - Well, maybe not. Consider next step i+1:

  $$\mathbf{R_{i+1} = 2 \text{ x } (R_i) - d = 2 \text{ x } (R_i - d) + d}$$

- Hence, we can compute R_{i+1} by not restoring R_i, and adding d instead of subtracting d
  - Same value for R_{i+1} results
- Throughput of 1 bit per cycle

Non-Restoring Division

Iteration	Step	Divisor	Remainder
0	Initial values	0010	0000 0111
	Shift rem left 1	0010	0000 1110
1	2: Rem = Rem - Div	0010	1110 1110
	3b: Rem < 0 (add next), sll 0	0010	1101 1100
2	2: Rem = Rem + Div	0010	1111 1100
	3b: Rem < 0 (add next), sll 0	0010	1111 1000
3	2: Rem = Rem + Div	0010	0001 1000
	3a: Rem > 0 (sub next), sll 1	0010	0011 0001
4	Rem = Rem – Div	0010	0001 0001
	Rem > 0 (sub next), sll 1	0010	0010 0011
	Shift Rem right by 1	0010	0001 0011

What About Signed Division?

- The simplest solution is to remember the signs of the divisor and dividend and then negate the quotient if the signs disagree.
- However, (-7) ÷ 2 = ?
  - (+7) ÷ (-2) = ?
  - (-7) ÷ (-2) = ?

- The correctly signed division algorithm *negates the quotient if the signs of the operands are opposite and makes the sign of the nonzero remainder match the dividend*.

Overview

- Addition
- Subtraction
- Arithmetic logic unit (ALU)
- Multiplication
- Division
- Floating number operations

Floating point addition

- Example in decimal: system precision 4 digits

  What is $9.999 \bullet 10^{1} + 1.610 \bullet 10^{-1}$?

- Aligning the two numbers

  $9.999 \bullet 10^{1}$

  $0.01610 \bullet 10^{1} \rightarrow 0.016 \bullet 10^{1}$ Truncation

- Addition

$$\begin{array}{rl} 9.999 & \bullet 10^{1} \\ +\ 0.016 & \bullet 10^{1} \\ \hline 10.015 & \bullet 10^{1} \end{array}$$

- Normalization

  $1.0015 \bullet 10^{2}$

- Rounding

  $1.002 \bullet 10^{2}$

Floating point addition steps

- Alignment
- The proper digits have to be added
- Addition of significands
- Normalization of the result
- Rounding

Example y=0.5+(-0.4375) in binary

- $0.5_{10} = 1.000_2 \times 2^{-1}$
- $-0.4375_2 = -1.110_2 \times 2^{-2}$
- Step1:The fraction with lesser exponent is shifted right until matches

  $-1.110_2 \times 2^{-2} \rightarrow -0.111_2 \times 2^{-1}$
- Step2: Add the significands

  $\mathbf{1.000_2 \times 2^{-1}}$

  $+) \mathbf{-0.111_2 \times 2^{-1}}$

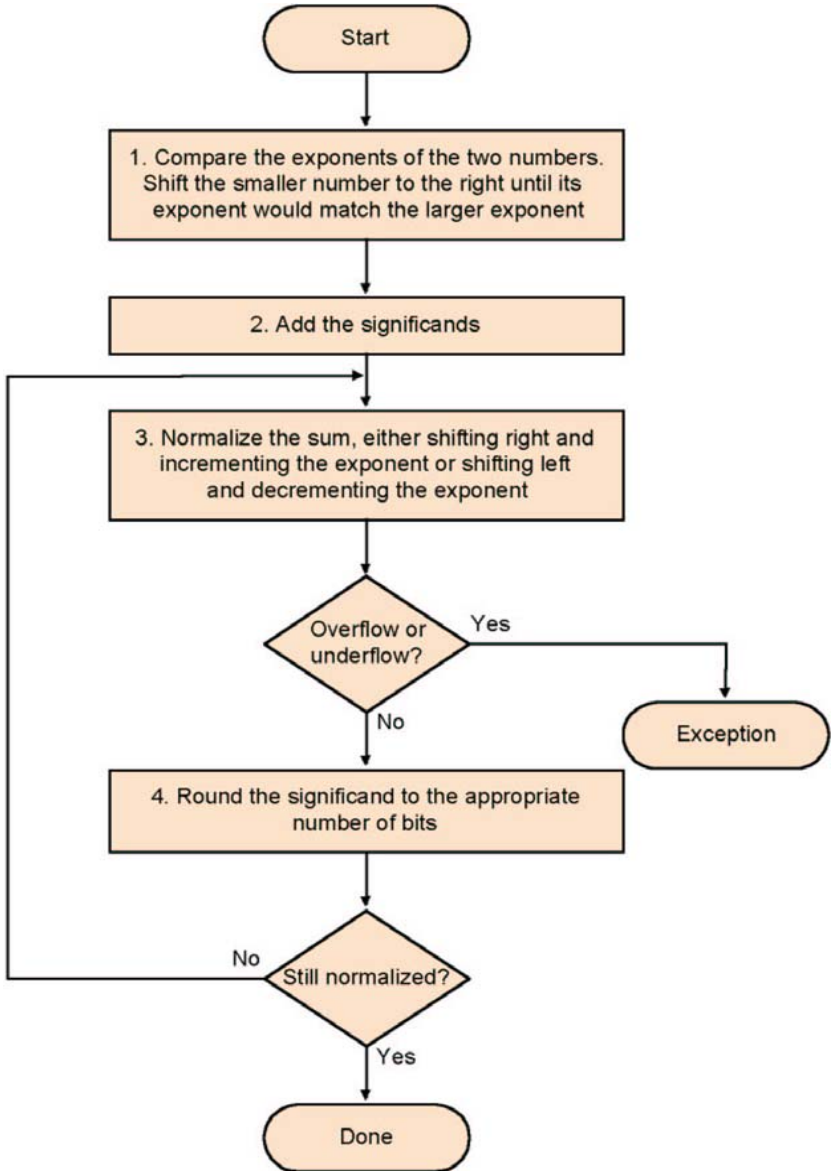
  $\mathbf{0.001_2 \times 2^{-1}}$
- Step3: Normalize the sum and checking for overflow or underflow

  $\mathbf{0.001_2 \times 2^{-1} \rightarrow 0.010_2 \times 2^{-2} \rightarrow 0.100_2 \times 2^{-3} \rightarrow 1.000_2 \times 2^{-4}}$
- Step4: Round the sum

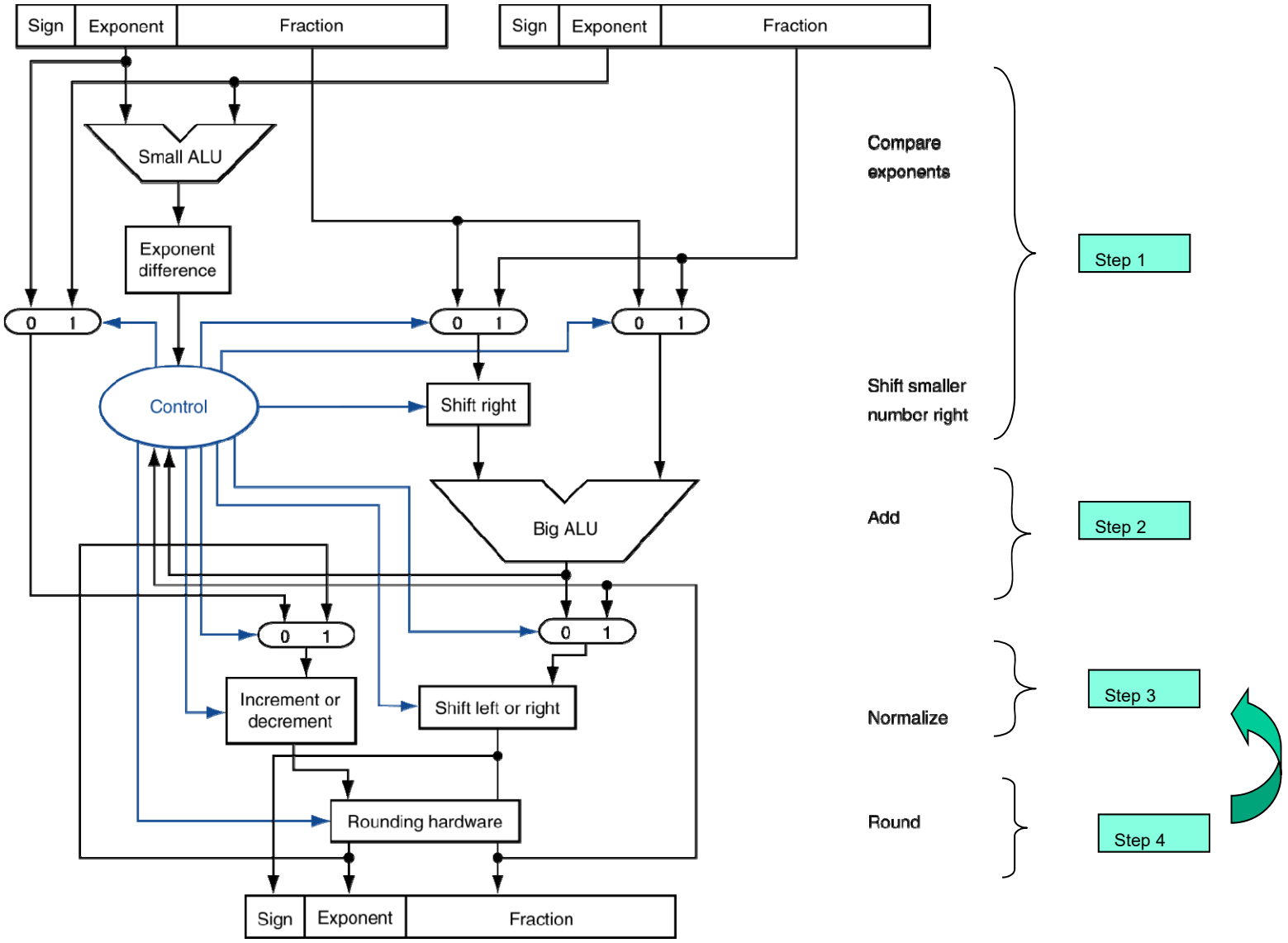
  $\mathbf{1.000_2 \times 2^{-4} = 0.0625_{10}}$

Algorithm

- Normalize Significands
- Add Significands
- Normalize the sum
- Over/underflow
- Rounding
- Normalization



FP Adder Hardware



FP Adder Hardware

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

Floating-Point Multiplication

- Consider a 4-digit decimal example
  - $1.110 \times 10^{10} \times 9.200 \times 10^{-5}$
- 1. Add exponents
  - For biased exponents, subtract bias from sum
  - New exponent = 10 + –5 = 5
- 2. Multiply significands
  - $1.110 \times 9.200 = 10.212 \Rightarrow 10.212 \times 10^{5}$
- 3. Normalize result & check for over/underflow
  - 1.0212×10^{6}
- 4. Round and renormalize if necessary
  - 1.021×10^{6}
- 5. Determine sign of result from signs of operands
  - $+1.021 \times 10^{6}$

Floating Point Multiplication

- Composition of number from different parts

  → separate handling

  $$(s1 \bullet 2^{e1}) \bullet (s2 \bullet 2^{e2}) = (s1 \bullet s2) \bullet 2^{e1+e2}$$

- Example

  1 10000010 000 0000 0000 0000 0000 0000 = -1×2^3

  0 10000011 000 0000 0000 0000 0000 0000 = 1×2^4

- Both significands are 1 → product = 1 →Sign=1

- Add the exponents, bias = 127

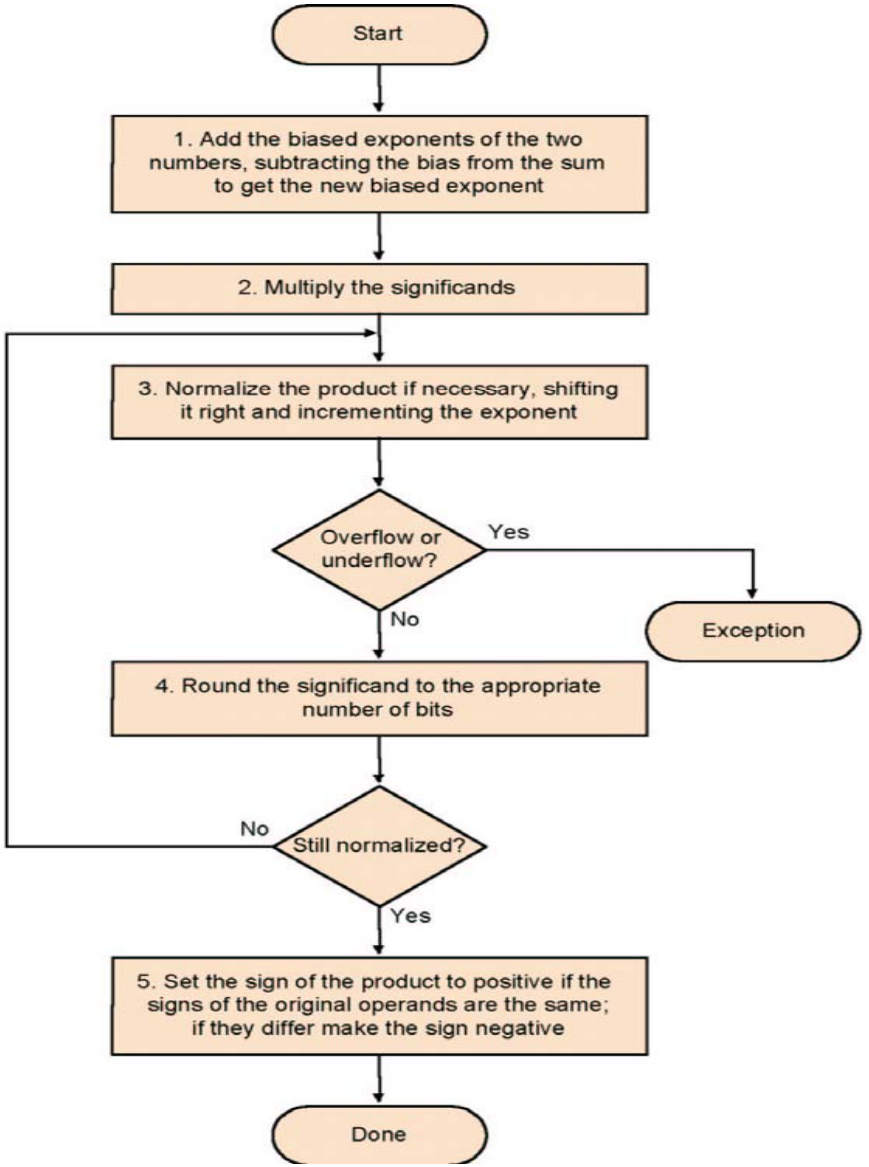
  $$\begin{array}{r} 10000010 \\ +10000011 \\ \hline 110000101 \end{array}$$

  Correction: 110000101-01111111=10000110=134=127+3+4

- The result: 1 10000110 000 0000 0000 0000 0000 0000 = -1×2^7

Multiplication

- Add exponents
- Multiply the significands
- Normalize
- Over- underflow
- Rounding
- Sign

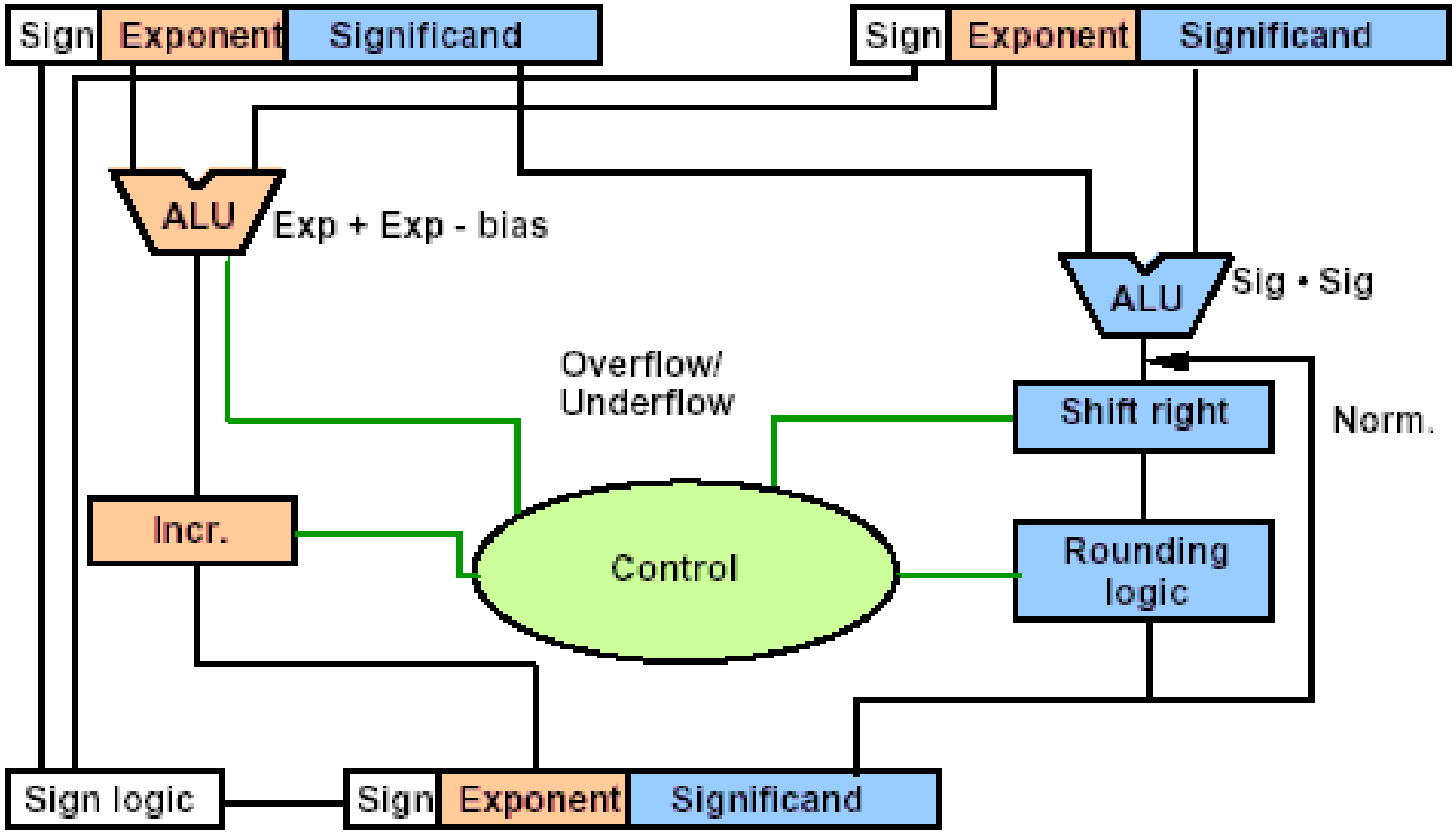


multiplying the numbers 0.5_{ten} and -0.4375_{ten} $\rightarrow 1.000_{two} \times 2^{-1}$ by $-1.110_{two} \times 2^{-2}$

- Step1:Adding the exponents without bias or using the biased
  - -1 + (-2)=-3
  - (-1 + 127) + (-2 + 127) - 127 = (-1 - 2) + (127 + 127-127) =-3+127 = 124
- Step 2. Multiplying the significands
  - $1.110000_{two} \times 2^{-3}$
- Step 3. normalize
  - $127 \geq -3 \geq -126$, no overflow or underflow.
- Step 4. Rounding
  - $1.110_{two} \times 2^{-3}$
- Step 5. sign
  - $-1.110_{two} \times 2^{-3} = -0.21875_{ten}$

$$
\begin{array}{rl}
 & 1.000_{two} \\
\times & 1.110_{two} \\
\hline
 & 0000 \\
 & 1000 \\
 & 1000 \\
 & 1000 \\
\hline
 & 1110000_{two}
\end{array}
$$

FP Multiplier Hardware



FP Arithmetic Hardware

- FP multiplier is of similar complexity to FP adder
  - But uses a multiplier for significands instead of an 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

Division-- Brief

- Subtraction of exponents
- Division of the significands
- Normalization
- Rounding
- Sign

Parallelism and Computer Arithmetic: Associativity

- $x + (y+ z) = (x + y) + z$?
  - $x = -1.5_{ten} \times 10^{38}$, $y= 1.5_{ten} X 10^{38}$, and $z = 1.0$
  - $x + (y + z) = 0.0$
  - $(x+y) + z = 1.0$

END