Chapter 7

Design of Arithmetic Circuits

Arithmetic Circuits are Important

- A large proportion of the data that is handled by digital systems is ***numerical data*** that must be processed ***arithmetically***.
- ***High-speed*** and ***area-efficient hardware implementations*** of arithmetic operations are thus of great importance in digital system design (e.g., in CPUs, digital signal processors, special-purpose custom hardware accelerators, etc.)
- Some arithmetic operations are best implemented as ***purely combinational circuits***.
- Other arithmetic operations would require purely combinational circuits that are uneconomically large. In these cases, ***sequential circuits*** are used instead.
- ***Regular***, ***iterative structure*** is a recurring theme in the design of arithmetic circuits. Such structure simplifies both the implementation and analysis of arithmetic circuits.
- Regular structure also simplifies the application of ***pipelining*** and ***parallelism*** to speed up the operation of arithmetic circuits.

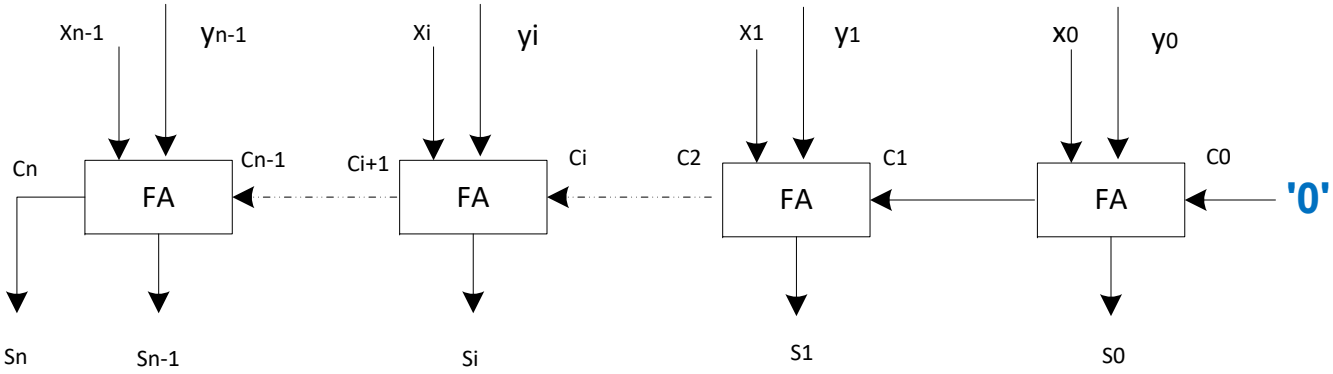
Addition of Unsigned Integers

$01001_2 + 11101_2 = ?$

01001 -> x	← ***Addend***
+ 11101 -> y	← ***Augend***
(**1**1001) -> c	← ***Carry-out bits***
100110 –> s	← ***Sum***

Note: The ***most significant bit (MSB)*** of the *sum* is the MSB of the *carry-out*.

An *n*-bit ***ripple-carry adder (RCA):***



Note: The ***carry-in*** bit into the ***least significant bit (LSB)*** of the adder is a '0'.

Subtraction of Unsigned Integers

Minuend - Subtrahend

$10100110_2 - 01001010_2 = ?$

(0 1 0 1 1 0 0 0) -> b ***Borrow outs*** - can be 0 or 1

1 0 1 0 0 1 1 0 -> x

\- 0 1 0 0 1 0 1 0 -> y

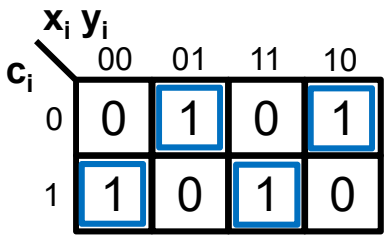
0 1 0 1 1 1 0 0 -> d ***Difference***

Adder/Subtractor Truth Table

x_i	y_i	$\overline{y_i}$	c_i /b_i	$\overline{b_i}$	s_i	c_{i+1}	d_i	b_{i+1}	$\overline{b_{i+1}}$
0	0	1	0	1	0	0	0	0	1
0	0	1	1	0	1	0	1	1	0
0	1	0	0	1	1	0	1	1	0
0	1	0	1	0	0	1	0	1	0
1	0	1	0	1	1	0	1	0	1
1	0	1	1	0	0	1	0	0	1
1	1	0	0	1	0	1	0	0	1
1	1	0	1	0	1	1	1	1	0

Output Equations for Unsigned Addition and Subtraction

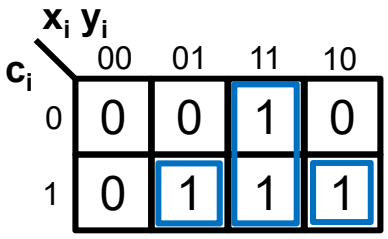
Simplified equations for unsigned addition:



s_i

$$s_i = x_i \oplus y_i \oplus c_i$$

3-input XOR

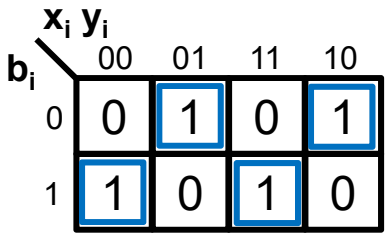


c_{i+1}

$$c_{i+1} = x_i y_i + x_i \overline{y}_i c_i + \overline{x}_i y_i c_i$$
$$= x_i y_i + (x_i \oplus y_i) c_i$$
$$= x_i y_i + c_i x_i + c_i y_i$$

2-level logic

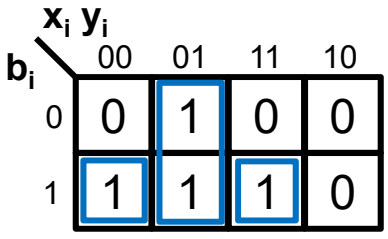
Simplified equations for unsigned subtraction:



d_i

$$d_i = x_i \oplus y_i \oplus b_i$$

3-input XOR



b_{i+1}

3-level logic

$$b_{i+1} = \overline{x}_i y_i + x_i y_i b_i + \overline{x}_i \overline{y}_i b_i$$
$$= \overline{x}_i y_i + \overline{(x_i \oplus y_i)} b_i$$

Adder/Subtractor of Unsigned Integers

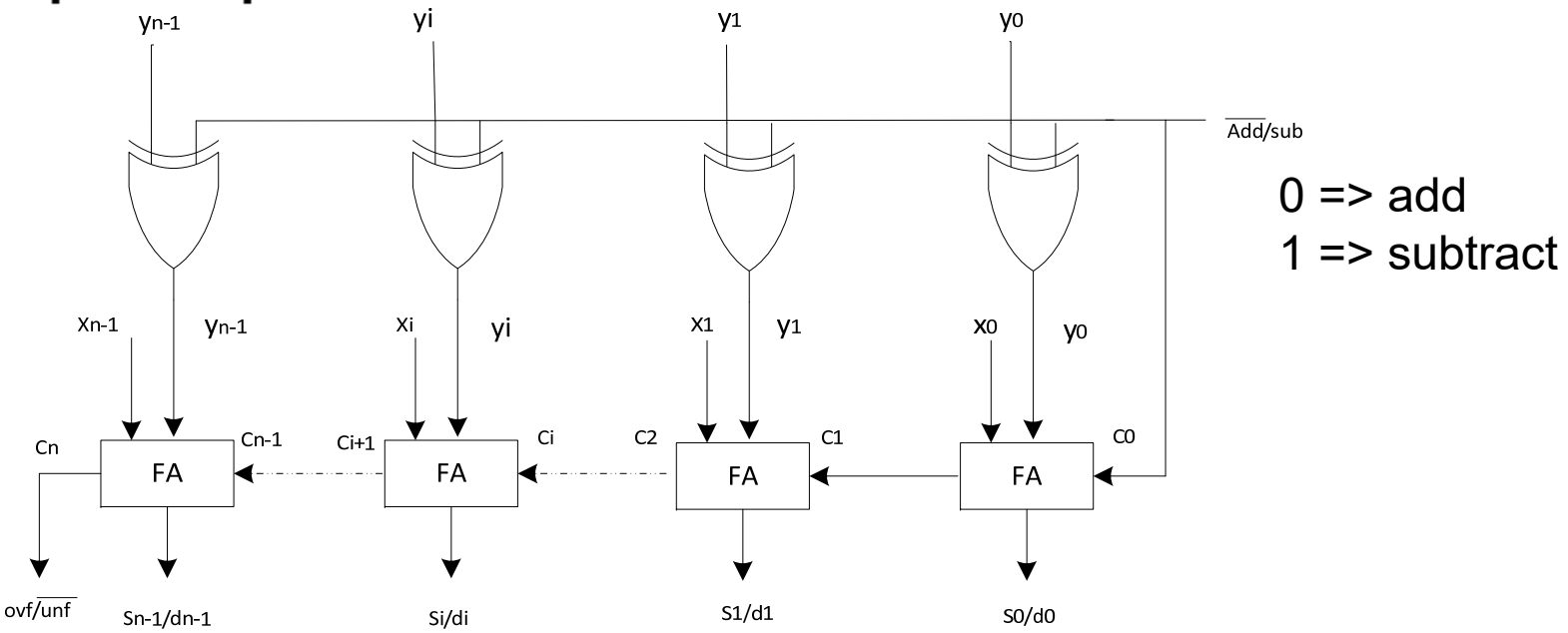
Addition: $S_i = (x_i \oplus y_i) \oplus C_i$ and $C_{i+1} = x_i \cdot y_i + (x_i \oplus y_i) \cdot c_i$, with C_0=0.

Subtraction: $d_i = (x_i \oplus y_i) \oplus b_i$ and $b_{i+1} = \overline{x_i} \cdot y_i + \overline{x_i \oplus y_i} \cdot b_i$, with b_0=0.

Or, $d_i = (x_i \oplus \overline{y_i}) \oplus \overline{b_i}$ and $\overline{b_{i+1}} = x_i \cdot \overline{y_i} + (x_i \oplus \overline{y_i}) \cdot \overline{b_i}$, with $\overline{b_0}$=1.

How to get a subtractor from an adder? Add, but use neg. logic for y_i, b_i & b_{i+1}.

Adder adapted to perform both addition and subtraction:



Representations of Signed Binary Integers (1)

X = the integer to be represented
n = number of bits available to represent X

For nonnegative (unsigned) integers X, a ***radix-2 positional representation*** is used:

$$X = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2 + x_0 = \sum_{i=0}^{n-1} x_i 2^i$$

For negative integers, there are ***several signed representations*** in common use:

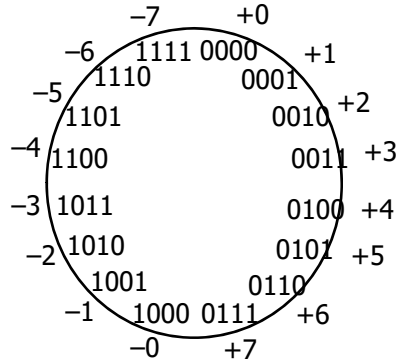
+X	Positive Integers	-X	Sign & Magnitude	2's Com-plement	1's Com-plement
+0	0000	-0	1000	-	1111
+1	0001	-1	1001	1111	1110
+2	0010	-2	1010	1110	1101
+3	0011	-3	1011	1101	1100
+4	0100	-4	1100	1100	1011
+5	0101	-5	1101	1011	1010
+6	0110	-6	1110	1010	1001
+7	0111	-7	1111	1001	1000
		-8	-	1000	-

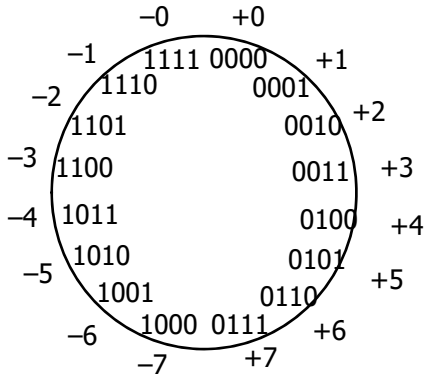
Representations of Signed Binary Integers (2)

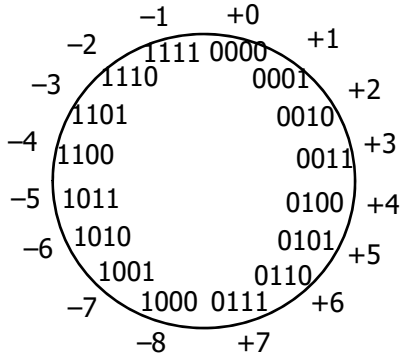
1) ***Sign-and-magnitude***
   - Arithmetic operations tend to be awkward to implement (special cases, two types of zero, etc.)
   - To form -N from N: *Complement the sign bit*.

2) ***1's complement***
   - Arithmetic operations are easier to implement.
   - To form -N from N: *Complement all of the bits.*

3) ***2's complement*** *– the most common choice*
   - Arithmetic operations are easier to implement.
   - There is only one zero, & one more negative num.
   - To form -N from N: *Complement all of the bits and then add 1.*







2's Complement Numbers

- For an *n*-bit representation of 2's complement:

  a negative value = 2^n – the positive value

  Example: *n* = 4

  $-4 = 2^4 - 4 = 16_{10} - 4 = 10000_2 - 00100_2 = 1100_2$

- To compute a 2's complement number:

$$x = -x_{n-1} \cdot 2^{n-1} + x_{n-2} \cdot 2^{n-2} + \cdots + x_0 \cdot 2^0$$

- The "most" negative number is -2^{n-1} and the "most" positive number: 2^{n-1}-1 (not symmetric, x_{n-1} is the sign bit).

- Example: 2's complement numbers

$00110101_2 = 1x2^5 + 1x2^4 + 1x2^2 + 1x2^0 = 32 + 16 + 4 + 1 = 53$

$10110101_2 = -1x2^7 + 1x2^5 + 1x2^4 + 1x2^2 + 1x2^0 = 32 + 16 + 4 + 1$

$= -128 + 32 + 16 + 4 + 1$

$= -75$

-1 ×

Addition and Subtraction in 2's Complement

- ***Addition:***
  - Perform normal unsigned binary addition, including adding in the carry-out bit into the sign position
  - Discard any carry-out bit from the addition at the sign bit position.
  - The answer is correct except when an "overflow" occurs
- ***Overflow*** or ***arithmetic overflow*** is the situation when the correct answer cannot be represented in the given number of bits. Overflow can occur in other kinds of arithmetic operations as well as addition & subtraction.
- ***Subtraction:***
  - Find the 2's complement representation of the ***subtrahend × (-1)***, and then add the result to the ***minuend***.

    e.g. $A - B = A + (- B)$
  - The ***difference*** is correct except when an overflow occurs
- Overflow can be avoided by making the hardware bigger to handle more bits, but sometimes the problem can be avoided by restructuring a calculation to prevent the magnitude of intermediate results from getting too big.

Addition of Signed Integers

Addition of two non-negative numbers:

72: 01001000	72: 01001000
49: 00110001	105: 01101001
(00000000)	(01001000)
01111001	10110001
Sign bit 0, no overflow	sign bit is 1, overflow

Addition of two negative numbers:

-63: 11000001	-63: 11000001
-32: 11100000	-96: 10100000
(11000000)	(10000000)
10100001	01100001
Sign 1, no overflow	sign 0, overflow

Addition of a positive and a negative number:

-42: 11010110	42: 00101010
8: 00001000	-8: 11111000
(00000000)	(11111000)
11011110	00100010
Sign 1, no overflow	sign 0, no overflow
(sum of positive offsets < 2^{n-1})	(sum of positive offsets ≥ 2^{n-1})

Overflow is indicated by the ***difference*** in the carry-in c_{in} into the sign position and the carry-out c_{out} out from the sign (the MSB) position.

Examples

1)

3	*0011*
+ 4	+ *0100*
7	*0111*

Correct answer

2)

5	*0101*
+ 6	+ *0110*
11	*1011*

Overflow for 4-bit signed integers

3)

5	*0101*
- 6	+ *1010*
- 1	*1111*

Correct answer

4)

6	*0110*
- 5	+ *1011*
1	*0001*

Correct answer

5)

- 3	*1101*
- 4	+ *1100*
-7	*1001*

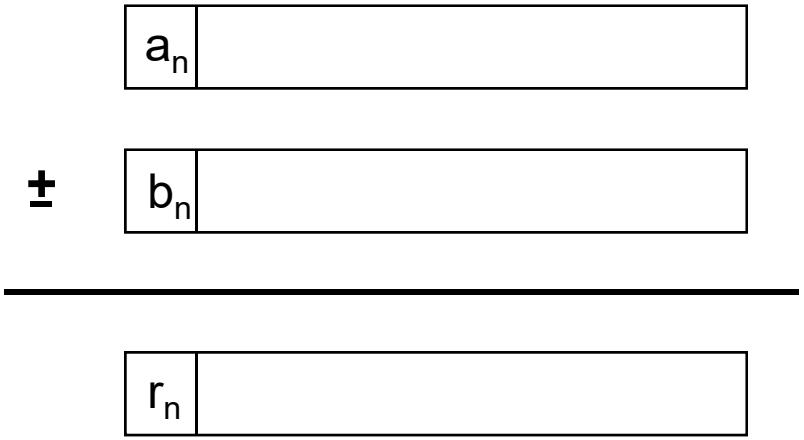
Correct answer

6)

- 5	*1011*
- 6	+ *1010*
- 11	*0101*

Overflow for 4-bit signed integers

Overflow Detection for 2's Complement Add./Sub.



Overflow status bit, V (commonly present in a microprocessor's status register)

V = 1 if overflow occurred
= 0 if overflow did not occur

Algebraically,

$$V = a_n\, b_n\, \overline{r_n} + \overline{a_n}\, \overline{b_n}\, r_n$$

$$= (c_{in} \text{ into } r_n) \text{ xor } (c_{out} \text{ out from } r_n)$$

Timing Analysis in Arithmetic Circuits

- ***Critical path***: The signal path through a circuit that has the longest delay (usually because the path travels through the largest possible number of logic gates).
- ***Worst-case timing analysis*** has two major steps:

  (1) ***Determine the critical path.*** Find the signal path, from an input to an output, that passes through the greatest possible number of logic gates.

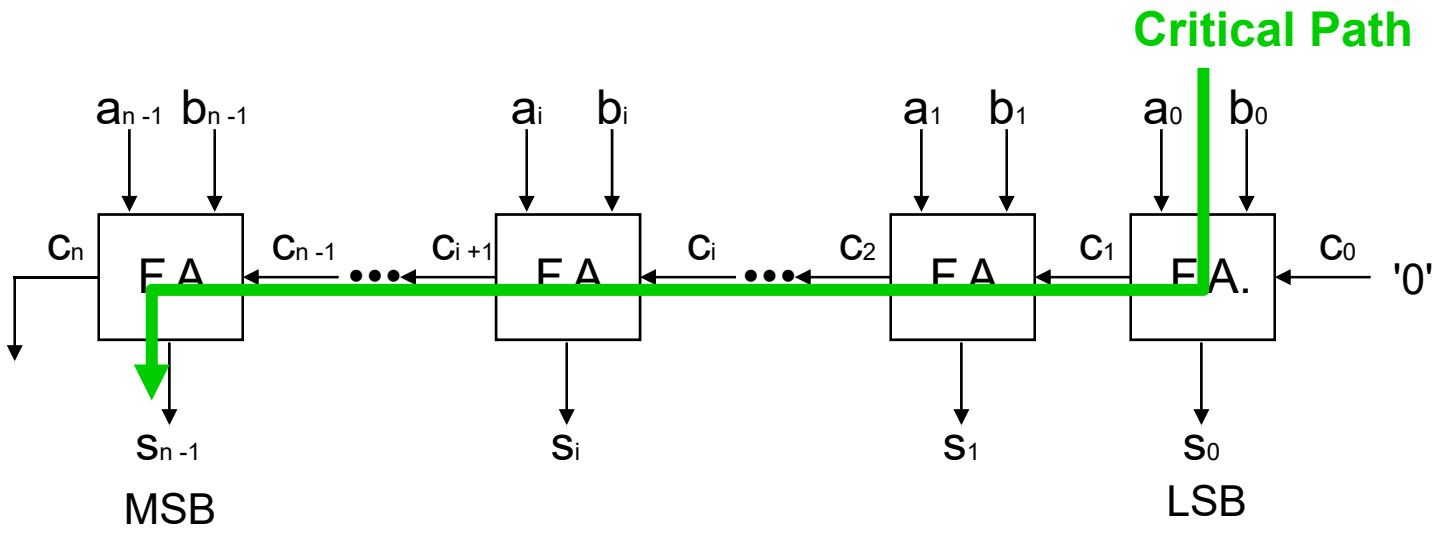
  Note: There may be multiple critical paths with roughly the same length. This will often be the case in a well-optimized design.

  (2) ***Determine the propagation delay along that critical path*** (e.g., in terms of typical gate delays, or in terms of time units using a CAD tool). This is the worst-case delay through the circuit, which must be taken into account to ensure correct operation within some maximum allowed timing delay (such as a target system clock period).

Ripple-Carry Adder (RCA)

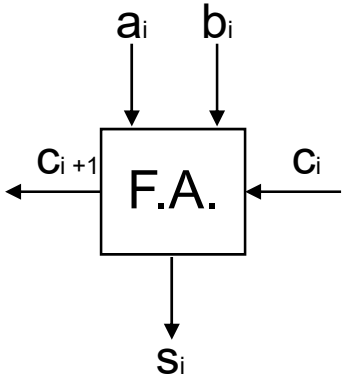
Main Ideas:

- decompose binary addition into bit-wise operations
- re-use the same circuit design for each bit position
- get a simple, regular, and extendable structure



Note: During an addition, the carry signal "ripples" from the LSB to the MSB.

Design of a 1-bit Full Adder "Slice"



c_i \ a_ib_i	00	01	11	10
0	0	1	0	1
1	1	0	1	0

$$s_i = a_i \oplus b_i \oplus c_i$$

c_i \ a_ib_i	00	01	11	10
0	0	0	1	0
1	0	1	1	1

$$c_{i+1} = a_i b_i + a_i c_i + b_i c_i$$

***Critical Path Analysis*:**

Recall: A critical path is a signal path that limits the speed of the circuit.

Worst case addition delay through FA to s_i = 3 gate delays = $3\,\tau_g$

Worst case addition delay through FA to c_{i+1} = 2 gate delays = $2\,\tau_g$

Worst-case addition delay for *n* bits = $2\,\tau_g\,(n\text{-}1) + 3\,\tau_g = (2n + 1)\,\tau_g$

***Summary for the Ripple-Carry Adder*:**

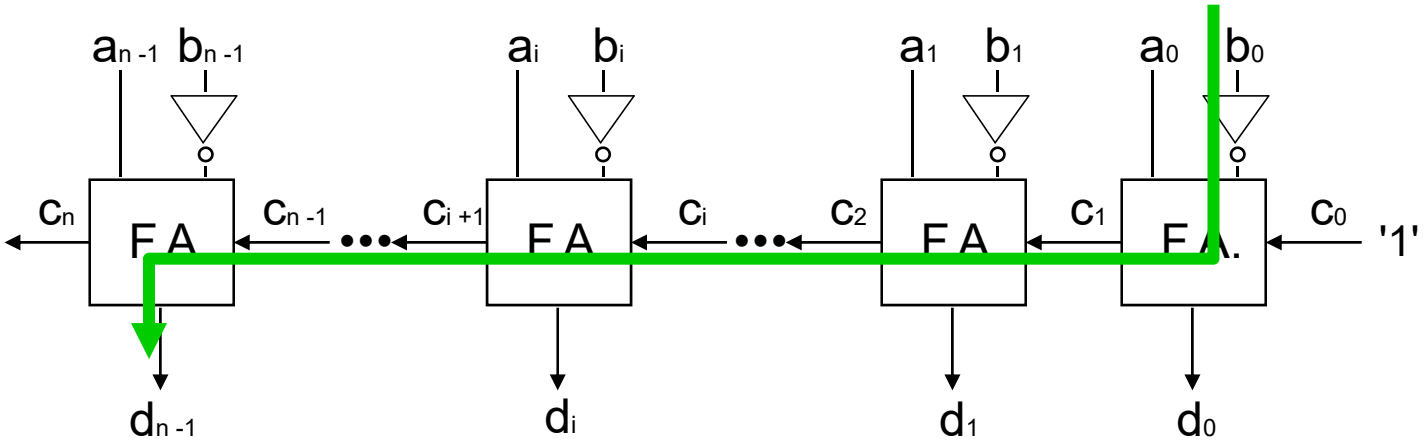
Advantages: simple, iterative, unidirectional structure

Disadvantages: delay grows linearly in proportion to *n*.

2's Complement Ripple Subtractor

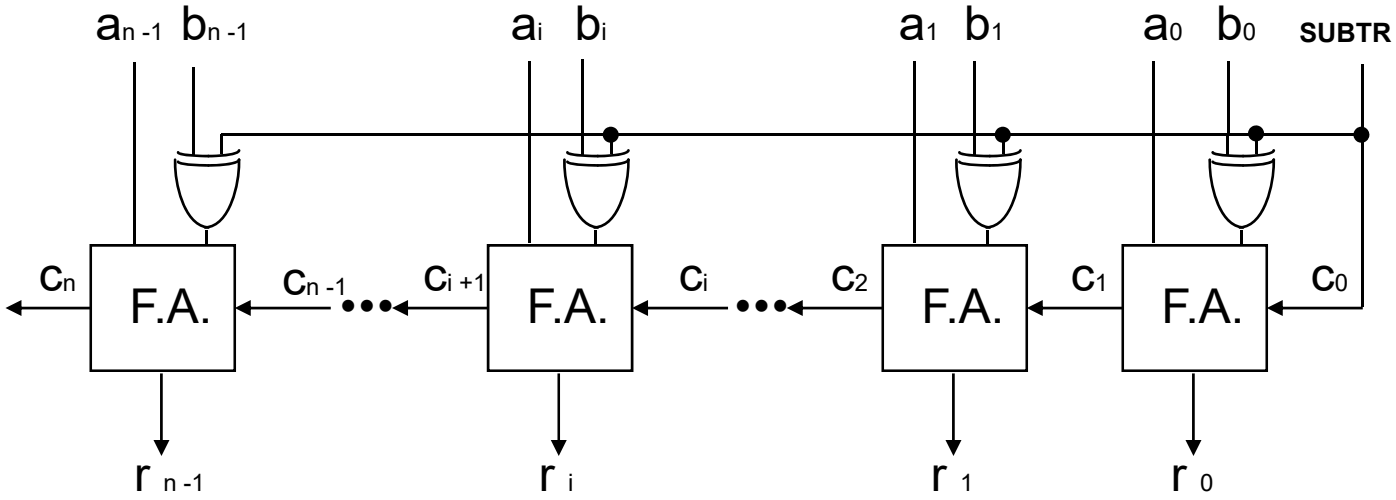
***Main Ideas*:**

- Want a simple regular structure that performs subtraction correctly, regardless of the sign of the two input operands.
- Most common solution: use 2's complement arithmetic



Worst case delay = $3\,\tau_g + (n-2)\,2\tau_g + 3\,\tau_g = (2n+2)\,\tau_g$

2's-Complement Adder/Subtractor



Note: The control signal SUBTR is asserted low to 0 to cause the addition A+B, and is asserted high to 1 to cause the subtraction A - B.

Subtraction: $x - y = x + (-y) = x + \bar{y} + 1$

Fast Binary Adders

***Strategy #1*:** Implement the *n*-bit adder as a two-level network

Advantages: The worst case delay $\geq 3\ \tau_g$

Disadvantages: The size of the combinational circuit grows very rapidly (exponentially in gate count) with the operand width *n*.

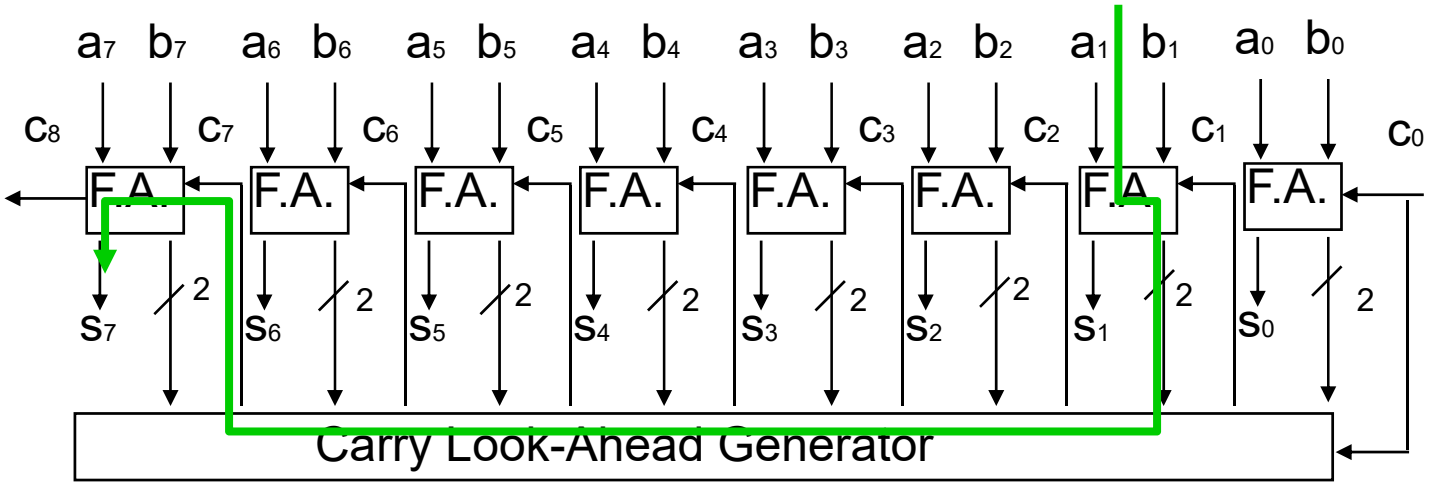
***Strategy #2*:** Find a compromise circuit design (use a tree-structured network perhaps) that offers sufficient speed with acceptable cost.

Goals:

- faster operation than a ripple-carry adder
- has a circuit structure that is not overly complicated

Fast Carry Signal Generation

- The limiting factor in the ripple-carry adder is the critical path that propagates the carry signals across the full width of the adder.
- Perhaps there is a faster way of generating the required carry signals.
- Use a special circuit, a ***carry look-ahead generator (CLG)*** to rapidly produce the carry signals that are needed at each bit position.



Propagate and Generate Signals

Recall: $c_1 = a_0 b_0 + a_0 c_0 + b_0 c_0$

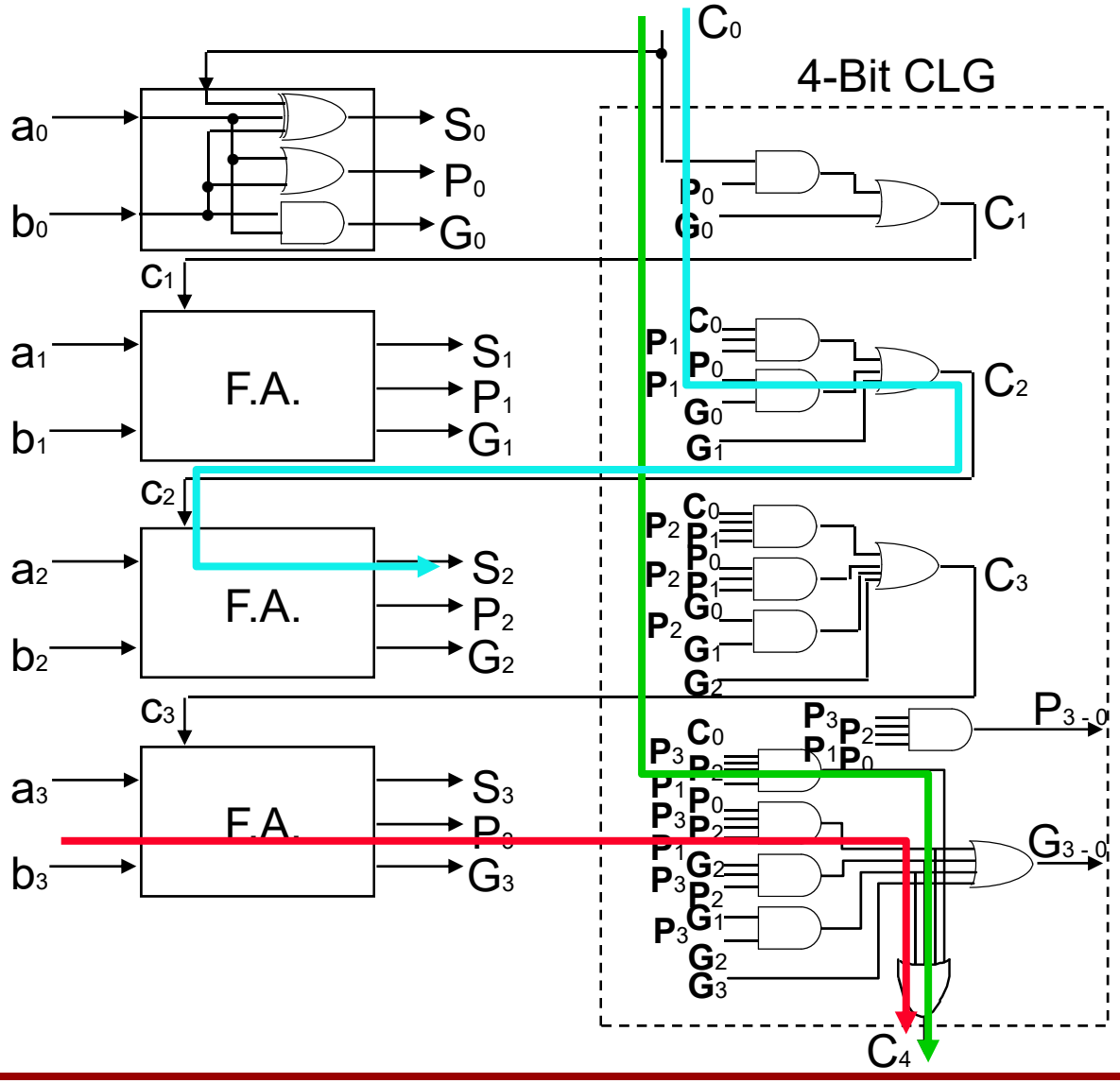
Define ***generate*** $G_i = a_i b_i$ and ***propagate*** $P_i = a_i + b_i$

Then we can calculate the carries directly from the generate and propagate signals as follows:

$$
\begin{aligned}
c_1 &= G_0 + c_0 P_0, \\
c_2 &= G_1 + G_0 P_1 + c_0 P_0 P_1, \\
c_3 &= G_2 + G_1 P_2 + G_0 P_1 P_2 + c_0 P_0 P_1 P_2, \\
c_4 &= G_3 + G_2 P_3 + G_1 P_2 P_3 + G_0 P_1 P_2 P_3 + c_0 P_0 P_1 P_2 P_3
\end{aligned}
$$

A ripple-carry adder can then be modified and sped up by using a CLG circuit, of suitable width, to calculate the required carry-in signals and then input those signals into the appropriate full-adder slices.

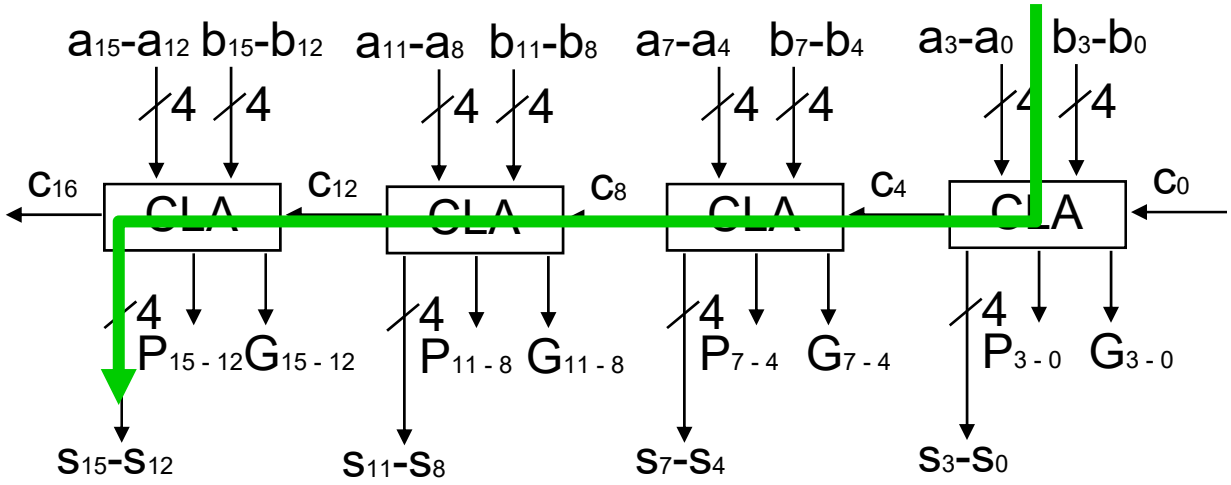
4-Bit Carry Look-ahead Adder



Critical Path Analysis of the 4-Bit Fast Adder

- The critical path still involves the carry signal, but we can now take advantage of the ***greater parallelism***.
- All of the individual generate and propagate signals can be formed ***at the same time*** (in parallel) from the corresponding "a" and "b" inputs. This calculation requires only 1 gate delay.
- The carry signals can now be formed from all of the generate and propagate signals (2 more gate delays).
- The carries are now input to the full adder slices, and this allows the final output bits in the sum to be formed (3 more gate delays).
- Thus a 4-bit "fast" addition takes only 6 gate delays as opposed to the 9 gate delays for a 4-bit ripple carry adder.
- However, a practical problem is encountered when such a fast adder is made wider: the CLG requires gates with greater and greater fan-in, and such gates become prohibitively large and slow.

Cascaded 4-Bit Carry Look-ahead Adders



Timing analysis in the case of four cascaded 4-bit CLAs:

$$\text{Worst-case delay} = (1+2)\tau_g + 2\tau_g + 2\tau_g + (2+3)\tau_g = 12\tau_g$$

General expression for worst case delay through $n/4$ cascaded 4-bit CLAs

$$\text{Worst-case delay} = (4 + n/2)\ \tau_g$$

where n is a multiple of 4.

Group Generates and Group Propagates (1)

Recall: $c_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0c_0$

Define: $G_{3\text{-}0} = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0$ (*group generate*, $2\tau_g$)

$P_{3\text{-}0} = P_3P_2P_1P_0$ (*group propagate*, τ_g)

Then: $c_4 = (G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0) + (P_3P_2P_1P_0)c_0$

$c_4 = G_{3\text{-}0} + P_{3\text{-}0}c_0$ (a delay of just over $2\tau_g$)

Then: $c_4 = G_{3\text{-}0} + P_{3\text{-}0}c_0$

$c_5 = G_4 + P_4G_{3\text{-}0} + P_4P_{3\text{-}0}c_0$ (still a delay of just over $2\tau_g$)

$c_6 = G_5 + P_5G_4 + P_5P_4G_{3\text{-}0} + P_5P_4P_{3\text{-}0}c_0$

$c_7 = G_6 + P_6G_5 + P_6P_5G_4 + P_6P_5P_4G_{3\text{-}0} + P_6P_5P_4P_{3\text{-}0}c_0$

$c_8 = G_7 + P_7G_6 + P_7P_6G_5 + P_7P_6P_5G_4 +$

Etc. $P_7P_6P_5P_4G_{3\text{-}0} + P_7P_6P_5P_4P_{3\text{-}0}c_0$

Group Generate and Group Propagate (2)

Define: $G_{7-4} = G_7 + P_7G_6 + P_7P_6G_5 + P_7P_6P_5G_4$ (group generate, 1st level)

$P_{7-4} = P_7P_6P_5P_4$ (group propagate, 1st level)

Then: $c_4 = G_{3-0} + P_{3-0}c_0$ (G_{3-0} + P_{3-0} are also from the 1st level)

$c_5 = G_4 + P_4G_{3-0} + P_4P_{3-0}c_0$

$c_6 = G_5 + P_5G_4 + P_5P_4G_{3-0} + P_5P_4P_{3-0}c_0$

$c_7 = G_6 + P_6G_5 + P_6P_5G_4 + P_6P_5P_4G_{3-0} + P_6P_5P_4P_{3-0}c_0$

$c_8 = (G_7 + P_7G_6 + P_7P_6G_5 + P_7P_6P_5G_4) + (P_7P_6P_5P_4)G_{3-0} + (P_7P_6P_5P_4)P_{3-0}c_0$

$= G_{7-4} + P_{7-4}G_{3-0} + P_{7-4}P_{3-0}c_0$

$c_9 = G_8 + P_8G_{7-4} + P_8P_{7-4}G_{3-0} + P_8P_{7-4}P_{3-0}c_0$

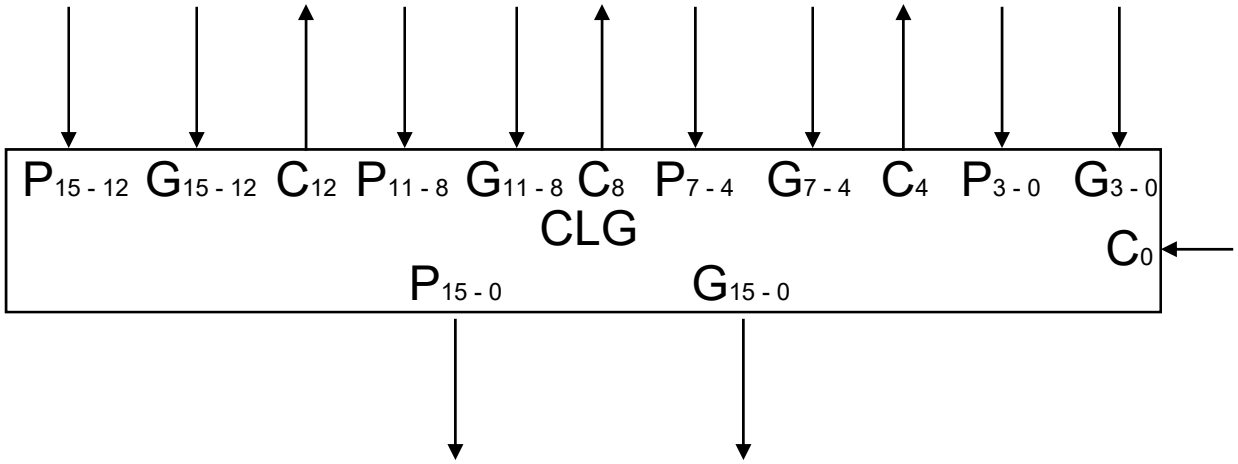
$c_{10} = G_9 + P_9G_8 + P_9P_8G_{7-4} + P_9P_8P_{7-4}G_{3-0} + P_9P_8P_{7-4}P_{3-0}c_0$

$c_{11} = (G_{10} + P_{10}G_9 + P_{10}P_9G_8) + (P_{10}P_9P_8)G_{7-4} + (P_{10}P_9P_8)P_{7-4}G_{3-0} + (P_{10}P_9P_8)P_{7-4}P_{3-0}c_0$

$c_{12} = G_{11-8} + P_{11-8}G_{7-4} + P_{11-8}P_{7-4}G_{3-0} + P_{11-8}P_{7-4}P_{3-0}c_0$

etc.

Carry Lookahead Generator (CLG)



Carry outputs:

$c_4 = G_{3\text{-}0} + P_{3\text{-}0}\, c_0$

$c_8 = G_{7\text{-}4} + P_{7\text{-}4}G_{3\text{-}0} + P_{7\text{-}4}P_{3\text{-}0}c_0$

$c_{12} = G_{11\text{-}8} + P_{11\text{-}8}G_{7\text{-}4} + P_{11\text{-}8}P_{7\text{-}4}G_{3\text{-}0} + P_{11\text{-}8}P_{7\text{-}4}P_{3\text{-}0}c_0$

Group propagate and generate outputs:

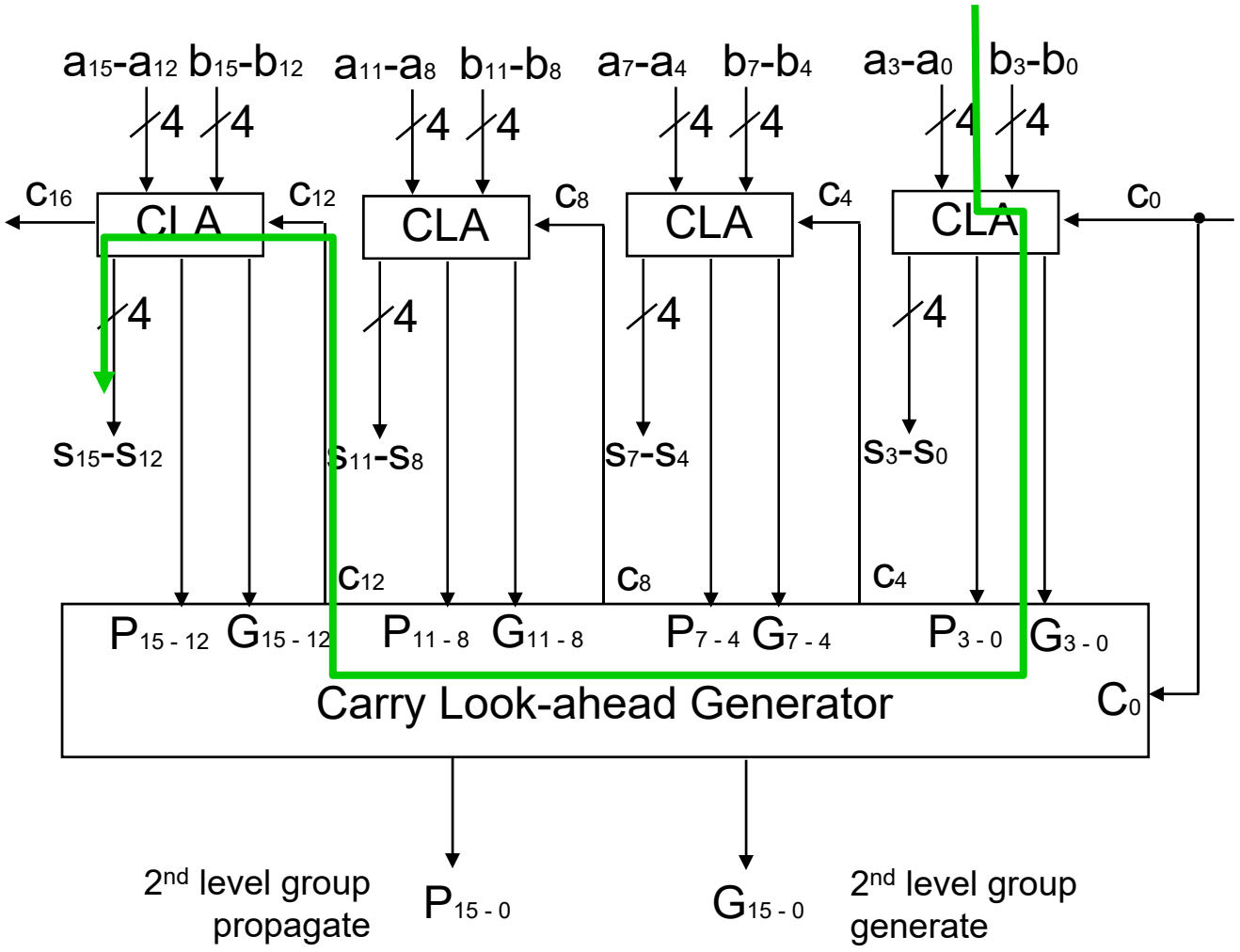
$P_{15\text{-}0} = P_{15\text{-}12}P_{11\text{-}8}P_{7\text{-}4}P_{3\text{-}0}$

$G_{15\text{-}0} = G_{15\text{-}12} + P_{15\text{-}12}G_{11\text{-}8} + P_{15\text{-}12}P_{11\text{-}8}G_{7\text{-}4} + P_{15\text{-}12}\, P_{11\text{-}8}P_{7\text{-}4}G_{3\text{-}0}$

Timing analysis: The critical path(s) to the G's and C's have a delay of $2\tau_g$.

The delay to the $P_{15\text{-}0}$ output is slightly faster, τ_g.

Fast 16-bit Adder Using 4-bit CLAs and a CLG



Timing analysis:

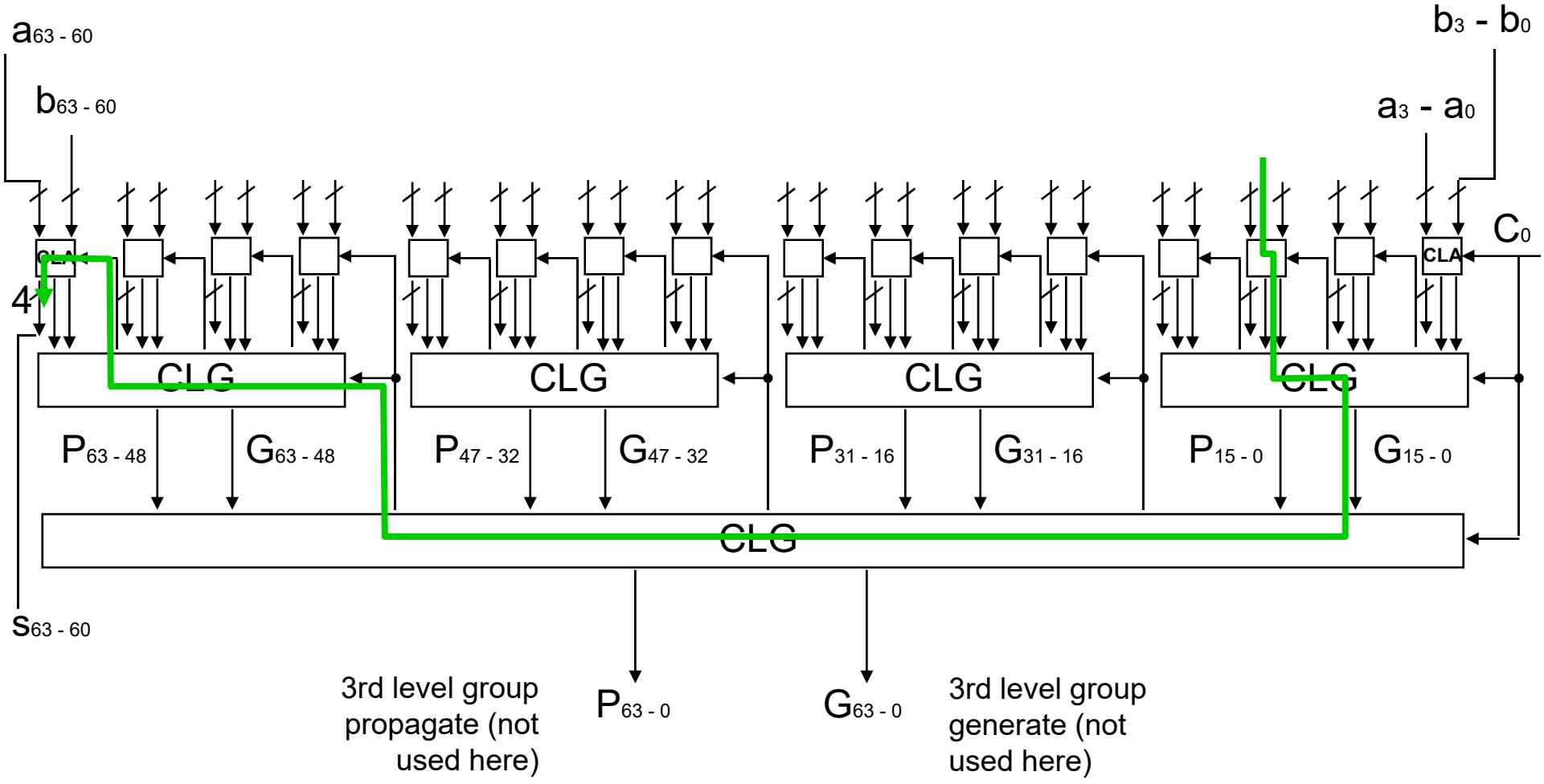
a_{3-0}, b_{3-0} to G_{3-0} : $3\ \tau_g$

G_{3-0} to C_{12} : $2\ \tau_g$

C_{12} to C_{15} : $2\ \tau_g$

C_{15} to S_{15} : $3\ \tau_g$

64-Bit Carry Look-ahead Adder (1)



64-Bit Carry Look-ahead Adder (2)

- A and B inputs to first-level group P's and G's: (1 + 2) τ_g
- First-level group P's and G's to second-level group P's and G's: 2 τ_g
- Second-level group P's and G's to C48: 2 τ_g

  C48 to C60: 2 τ_g

  C60 to C63: 2 τ_g

  C63 to S63: 3 τ_g

- Total delay for the 64-bit *carry look-ahead adder* = **14 τ_g** (9.2x faster)
- Total delay for the 16 *cascaded 4-bit CLAs* = **36 τ_g** (3.6x faster)
- Total delay for 64-bit *ripple-carry adder* = **129 τ_g**

64-Bit Carry Look-ahead Adder (3)

Question: What is the delay for a 128-bit CLA?

Answer: 18 τ_g

Generalize to arbitrary *n*:

The delay for an *n*-bit carry look-ahead adder constructed using a tree of 4-bit CLG's is:

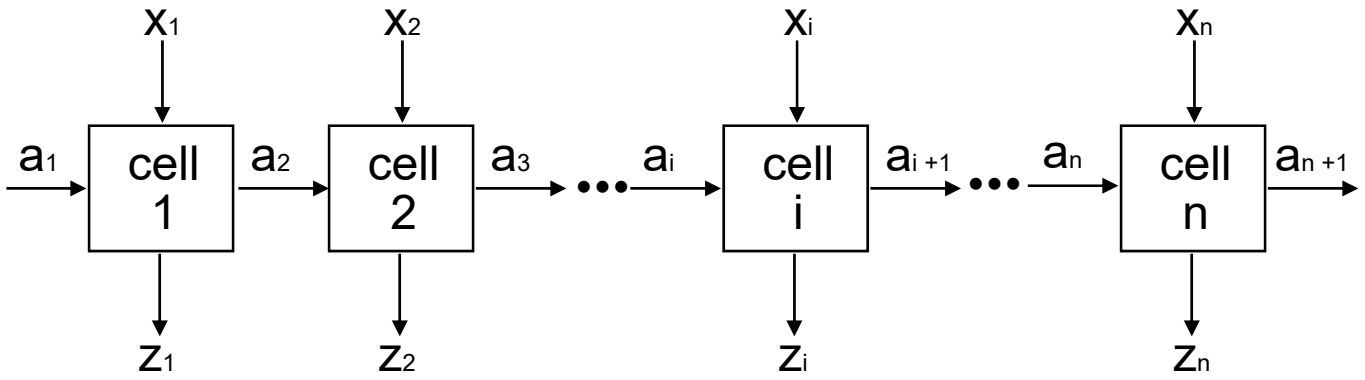
$$(2 + 4 \lceil \tfrac{1}{2} \log_2 n \rceil) \tau_g$$

Versus $(2n + 1) \tau_g$ for an *n*-bit ripple-carry adder.

Note: For the CLA, the size *n* lies within a slow-growing $\log_2$ function.

Unidirectional 1-D Iterative Networks (1)

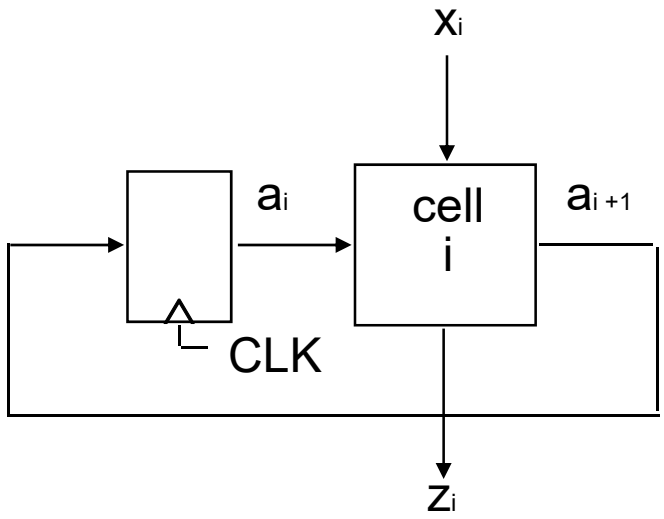
Many arithmetic operations can be decomposed into a series of locally interacting operations involving corresponding bit positions.



- ***Repetitions in space*** in one direction.
- Output Z_i from cell i depends on the cell input X_i and information a_i obtained from X_1 .... X_{i-1} and a_1.
- ***Speed is limited by the propagation delay of the 1-D ripple path***, which forms most of the critical path.

Unidirectional 1-D Iterative Networks (2)

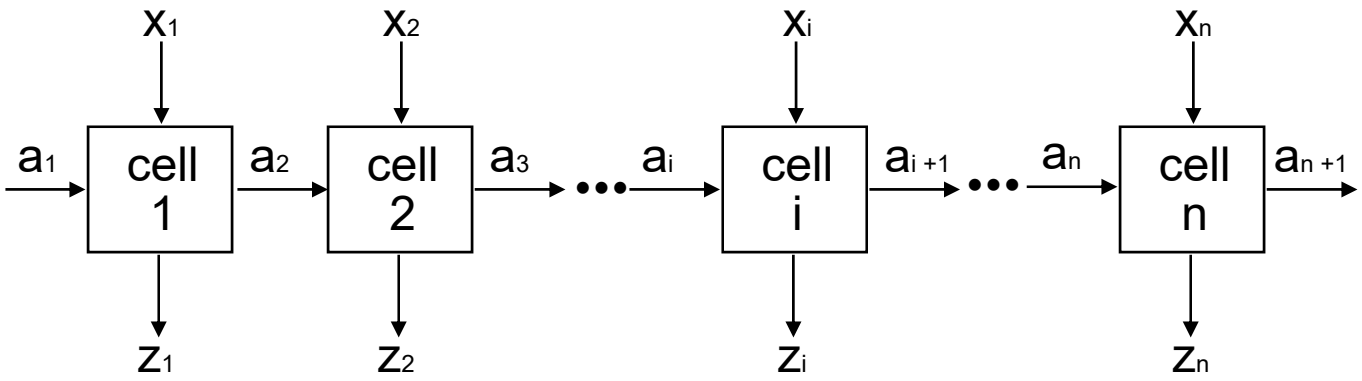
- There is a relationship between a 1-D-iterative network and a sequential finite state machine (FSM).
- The corresponding FSM has ***repetitions in time*** in the time dimension.
- The output z_i at time i depends on the present input X_i and the memory determined by previous inputs $X_1 \ldots X_{i-1}$ and the initial state a_0.
- ***Speed is limited by n x (clock period), where n is the number of iterations.***



Design of a 1-D Iterative (in Space) Network

- Use a design procedure similar to that used when designing synchronous sequential circuits. There is still iteration in the calculation.

Example: Design a combinational circuit that compares two n-bit binary numbers, $a = a_1 \cdots a_n$ and $b = b_1 \cdots b_n$, and produces outputs corresponding to $a < b$, $a = b$, and $a > b$.



- Each cell i inputs a corresponding pair, a_i and b_i, of input bits.
- There are three possible states of a cell: (1) $a < b$; (2) $a = b$; (3) $a > b$, where the comparison involves all bits at position i and higher.
- Two state bits, L and M, are sufficient to record the "state" in each cell.

Design of a 1-D Iterative (in Time) Network

L M	$a_i b_i$ = 00	01	11	10
(a=b) 00	00	10	00	01
(a>b) 01	01	01	01	01
11	XX	XX	XX	XX
(a<b) 10	10	10	10	10

$L^+ M^+$

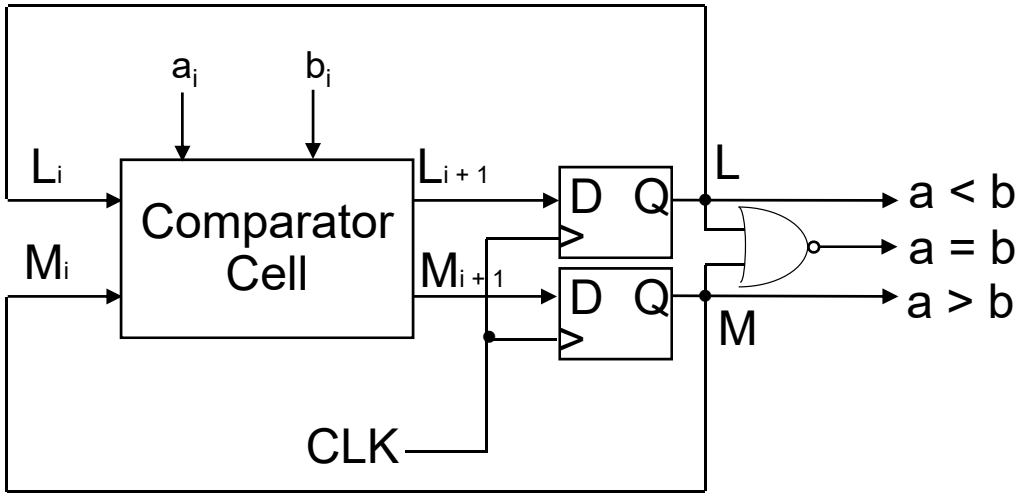
L M \ $a_i b_i$	00	01	11	10
00	0	1	0	0
01	0	0	0	0
11	X	X	X	X
10	1	1	1	1

$L^+ = L + (M' a_i' b_i)$

L M \ $a_i b_i$	00	01	11	10
00	0	0	0	1
01	1	1	1	1
11	X	X	X	X
10	0	0	0	0

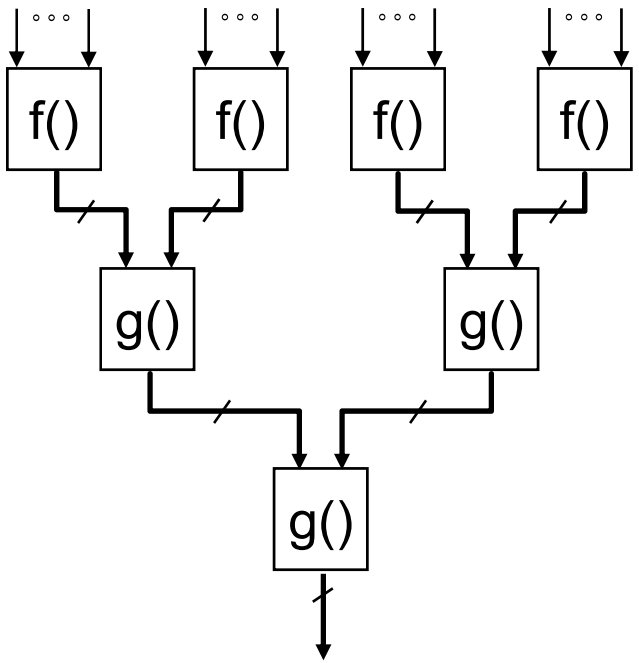
$M^+ = M + (L' a_i b_i')$

Sequential (Bit-Serial) Comparator



Note: The state bits need to be cleared just before the start of each bit-serial vector comparison.

Tree-Structured Networks



- Many functions have a ***recursive structure***, i.e., the function calls itself one or more times.

$$f(x_1, \ldots, x_k, x_{k+1}, \ldots, x_{2k}) = g(f(x_1, \ldots, x_k), f(x_{k+1}, \ldots, x_{2k}))$$

Here function g( ) combines the solutions from two calls to smaller instances of the original function f( ).

- Recursive structure can often be exploited in a *tree-structured* implementation (*divide-and-conquer*).

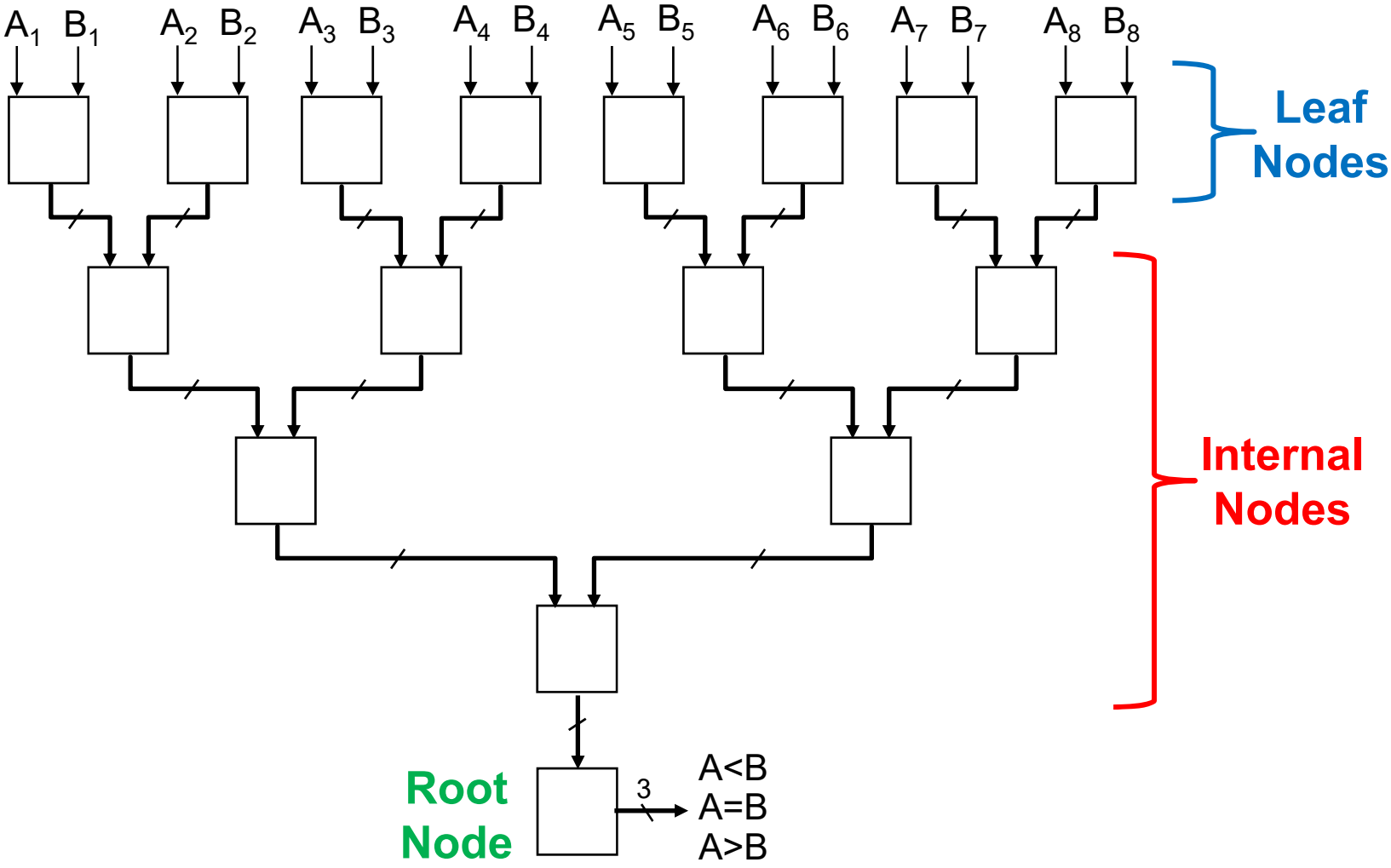
***Advantages of tree-structured networks*:**

--- can be faster than linear networks
--- can trade off fan-in factor against tree height
--- tree height growths with the log of the width

***Disadvantages*:**

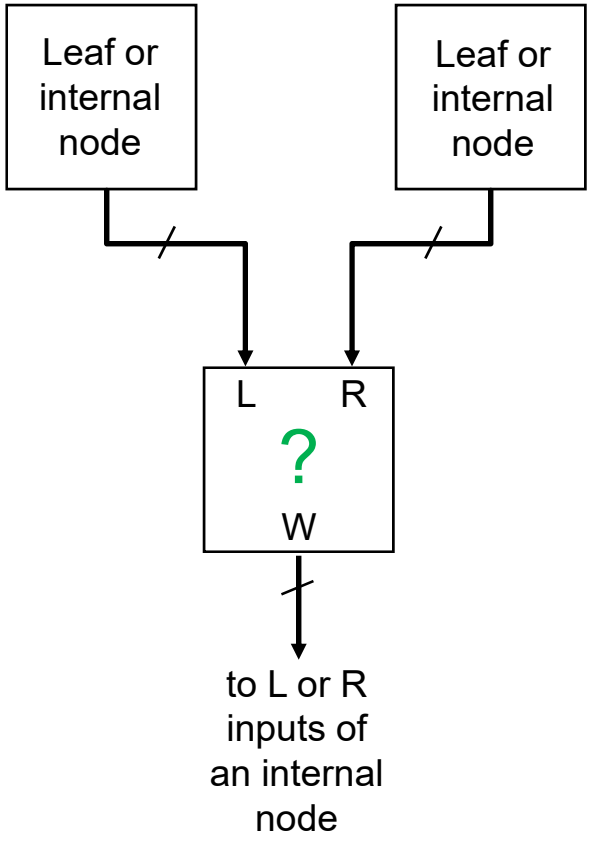
--- more complex structure
--- usually more hardware is required

Example: A Tree-Structured Comparator (1)



Example: A Tree-Structured Comparator (2)

Consider one ***internal node*** in the tree:



L_1L_2	R_1R_2	W_1W_2
(a=b)00	00	00
(a=b)00	01	01
(a=b)00	10	10
(a>b)01	XX	01
(a<b)10	XX	10
11	XX	XX
XX	11	XX

L_1L_2 \ R_1R_2	00	01	11	10
00	0	0	X	1
01	0	0	X	0
11	X	X	X	X
10	1	1	X	1

$W_1 = L_1 + L_2'R_1$

L_1L_2 \ R_1R_2	00	01	11	10
00	0	1	X	0
01	1	1	X	1
11	X	X	X	X
10	0	0	X	0

$W_2 = L_2 + L_1'R_2$

Example: A Tree-Structured Comparator (3)

Input logic in each ***leaf node*** in the tree (i.e. a first-level node) must implement the following initial calculation:

a_i	b_i	W_1	W_2
0	0	0	0
0	1	1	0
1	0	0	1
1	1	0	0

$W_1 = a_i' \, b_i$

$W_2 = a_i \, b_i'$

Output logic following the ***root node*** in the tree must implement the following final output calculation:

$(a < b) = W_1$

$(a > b) = W_2$

$(a = b) = (W_1 + W_2)'$

Timing analysis for a tree-structured comparator with n-bit inputs:

The critical path(s) go from the inputs through the leaf nodes ($2\tau_g$), then through $\lceil \log_2 n \rceil$ levels of internal and root nodes ($3\tau_g$ each), and then through the output logic ($2\tau_g$).

The worst-case delay along the critical path is therefore: $4\tau_g + 3\lceil \log_2 n \rceil \tau_g$

Unsigned Array Multiplier

- Multiplication is another important arithmetic operation in digital systems.
- Multiplication involves the ***repeated addition of partial product terms*** to form the product.
- The repeated additions can be performed as ***iterations in space*** (forming a combinational array multiplier) or as ***iterations in time*** (forming a sequential multiplier).
- Multiplication may operate on signed or unsigned operands, and different implementations are available for signed and unsigned multiplication.

Ex. of unsigned multiplication:

```
   1011   <-- Multiplicand
  x0101   <-- Multiplier
  ------
   1011
  0000    <-- Partial products
 1011
+0000
--------
00110111  <-- Product
```

Ex. of signed multiplication:

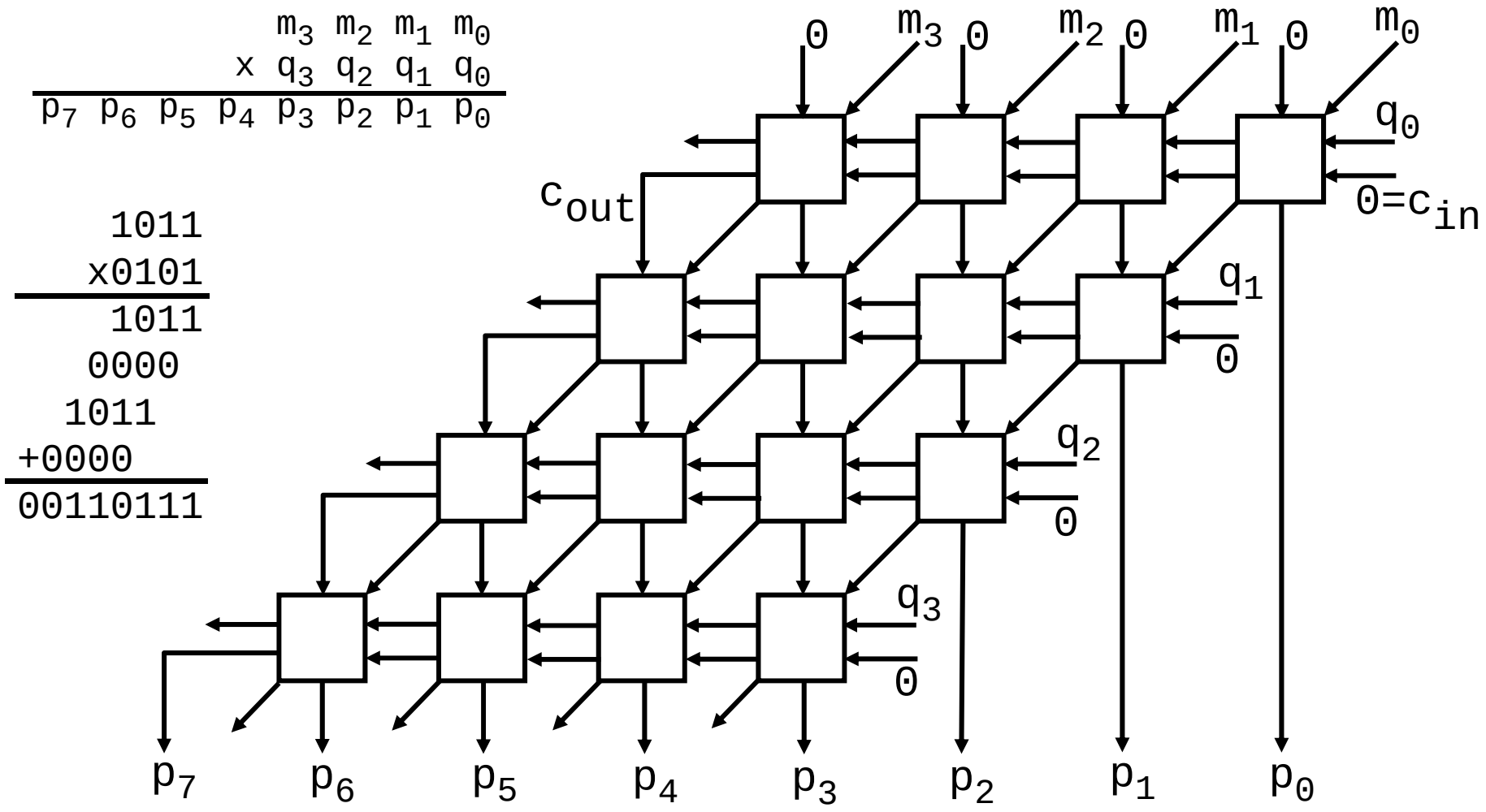
```
  1011
  x0101
  ------
   011
  000
 +011
 ------
 1001111
```

Note: Sign & magnitude operands assumed here.

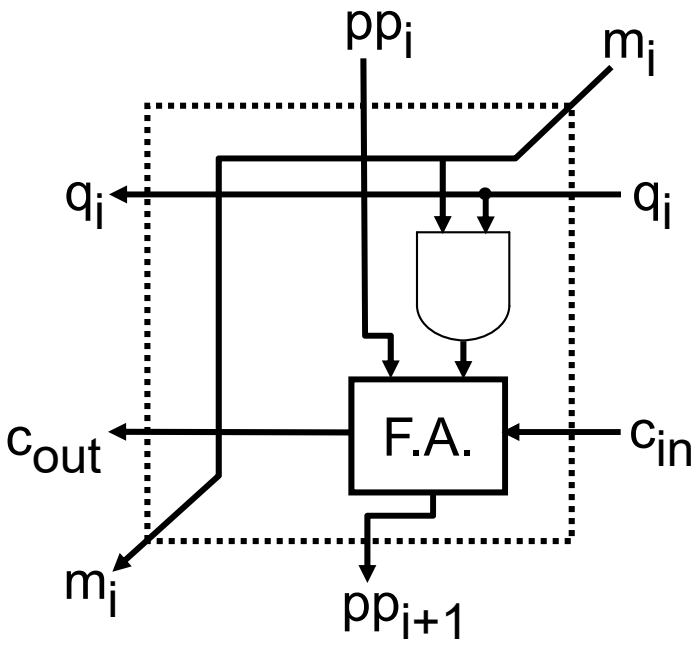
What is the Width of the Binary Product?

- Consider the product of two ***unsigned N-bit inputs***, where $N \geq 2$.
- What is the bit width of the largest possible product of two such numbers?
- The decimal value of the largest unsigned N-bit binary number (all 1s) is $2^N - 1$.
- Thus the largest possible product is $(2^N - 1) \times (2^N - 1) = 2^{2N} - 2^{N+1} + 1$.
- This largest possible product requires exactly **$2N$ bits** to represent in binary.
- Note that $2^{2N} - 2^{N+1} + 1 = (2^{2N} - 1) - (2^{N+1} - 1) + 1$
- For $N = 2$: $(2^4-1)-(2^3-1)+1 = 1111_2 - 111_2 + 1 = 1001_2$ ($2N = 4$ bits wide)
- For $N = 3$: $(2^6-1)-(2^4-1)+1 = 111111_2 - 1111_2 + 1 = 110001_2$ ($2N = 6$ bits wide)
- For $N = 4$: $(2^8-1)-(2^5-1)+1 = 11111111_2-11111_2+1 = 11100001_2$ ($2N = 8$ bits wide)
- etc.
- ***Claim***: The product of two ***signed*** N-bit numbers requires only **2N – 1** bits.
- Why? With sign-and-magnitude representation, remove the sign bit from the two input operands. The ***magnitude*** of the product requires $2(N-1) = 2N - 2$ bits. When the sign bit is appended to the product, we confirm the claimed $2N$ -1 bits.

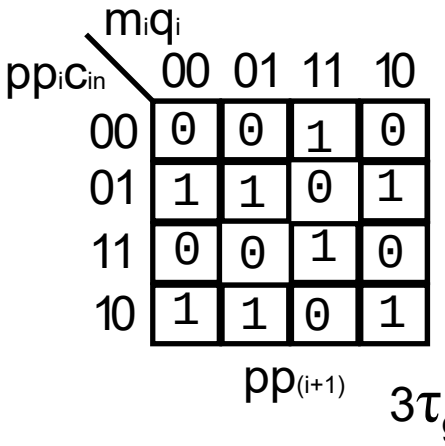
Purely Combinational Unsigned Multiplier (1)

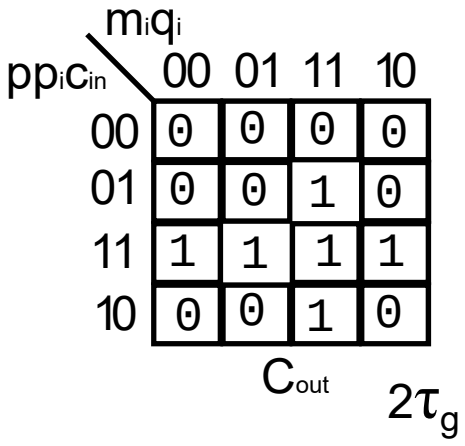


Purely Combinational Unsigned Multiplier (2)



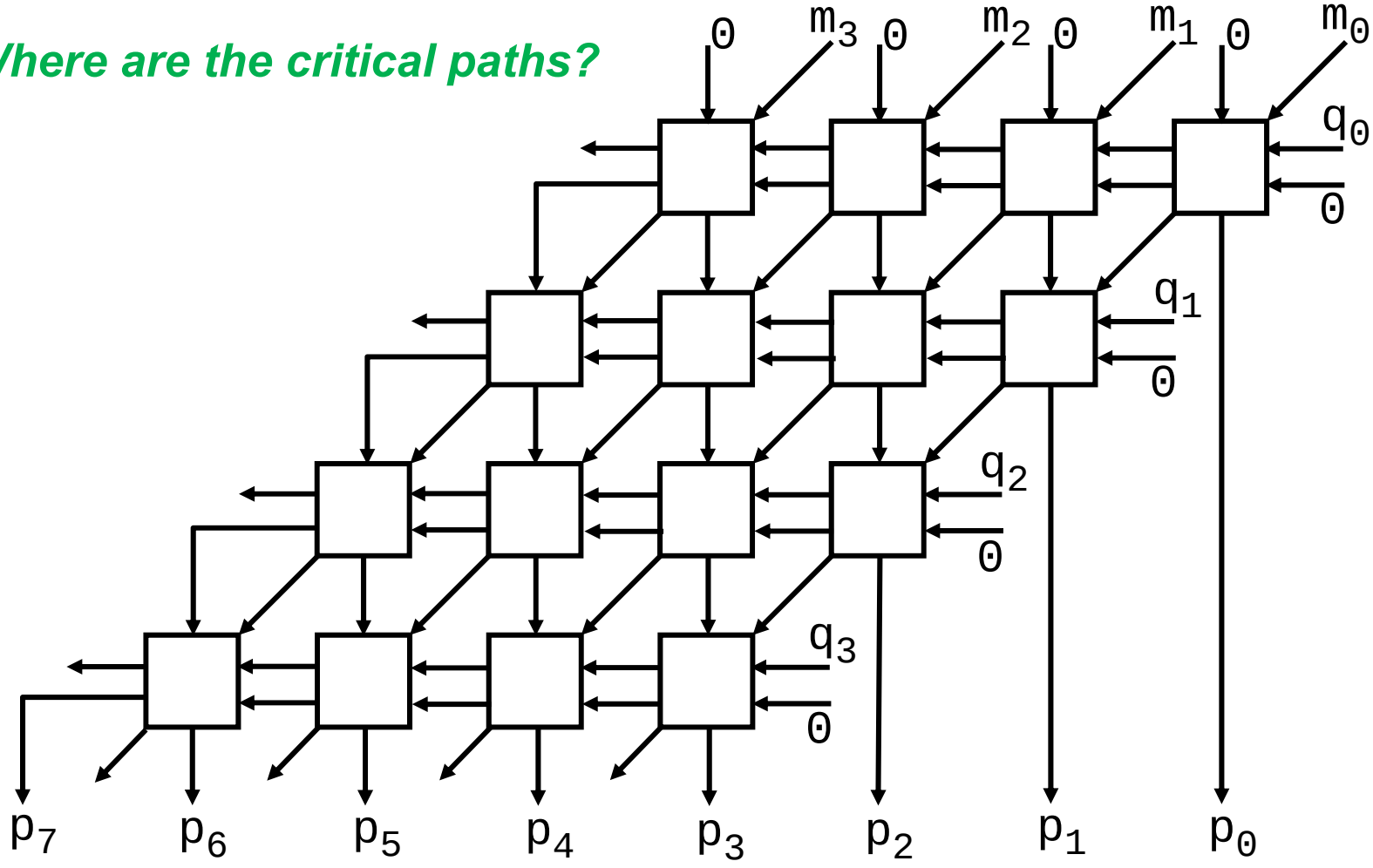
m_i	q_i	pp_i	c_{in}	pp_{i+1}	c_{out}
0	0	0	0	0	0
0	0	0	1	1	0
0	0	1	0	1	0
0	0	1	1	0	1
0	1	0	0	0	0
0	1	0	1	1	0
0	1	1	0	1	0
0	1	1	1	0	1
1	0	0	0	0	0
1	0	0	1	1	0
1	0	1	0	1	0
1	0	1	1	0	1
1	1	0	0	1	0
1	1	0	1	0	1
1	1	1	0	0	1
1	1	1	1	1	1





Purely Combinational Unsigned Multiplier (2)

Where are the critical paths?



Purely Combinational Unsigned Multiplier (3)

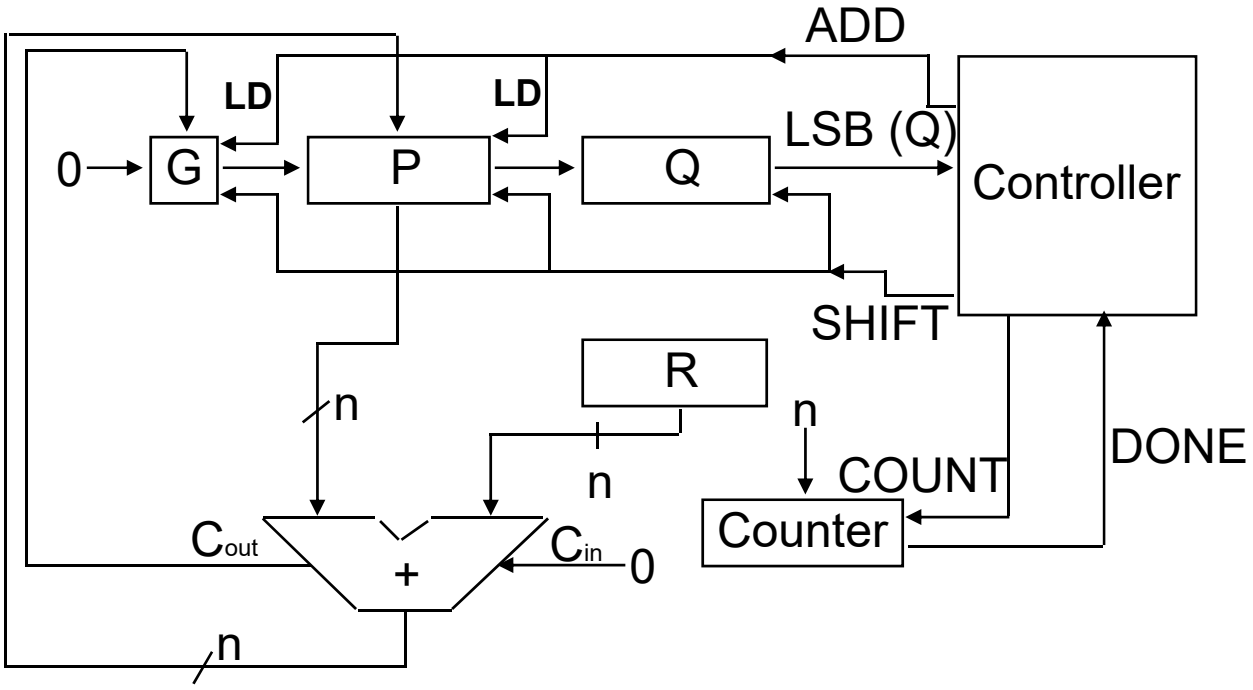
- There are many critical paths through the multiplier, which all terminate at the most significant bit position (exiting the sum-out circuit, not the slightly faster carry-out). In the example, p_6 is the slowest product bit.
- In general, for an *n*-bit combinational multiplier, all critical paths must pass out through (2*n*-2) carry-out outputs (each of delay $2\tau_g$), and *n* sum outputs (each of delay $3\tau_g$).
- Worst-case delay = $(2n\text{-}2)2\tau_g + n3\tau_g = (7n\text{-}4)\tau_g$ This is between 3 to 4 times slower than an *n*-bit addition using a ripple carry adder.
- How to speed up the multiplier?
  - Use fast adders to sum up the partial products

    Delay (roughly) = $3 \times (n - 2)\tau_g + 2 \times (2 + 4\lceil \tfrac{1}{2} \log_2 n \rceil)\tau_g$
  - Sum up the partial products using a binary tree of 2*n*-bit adders

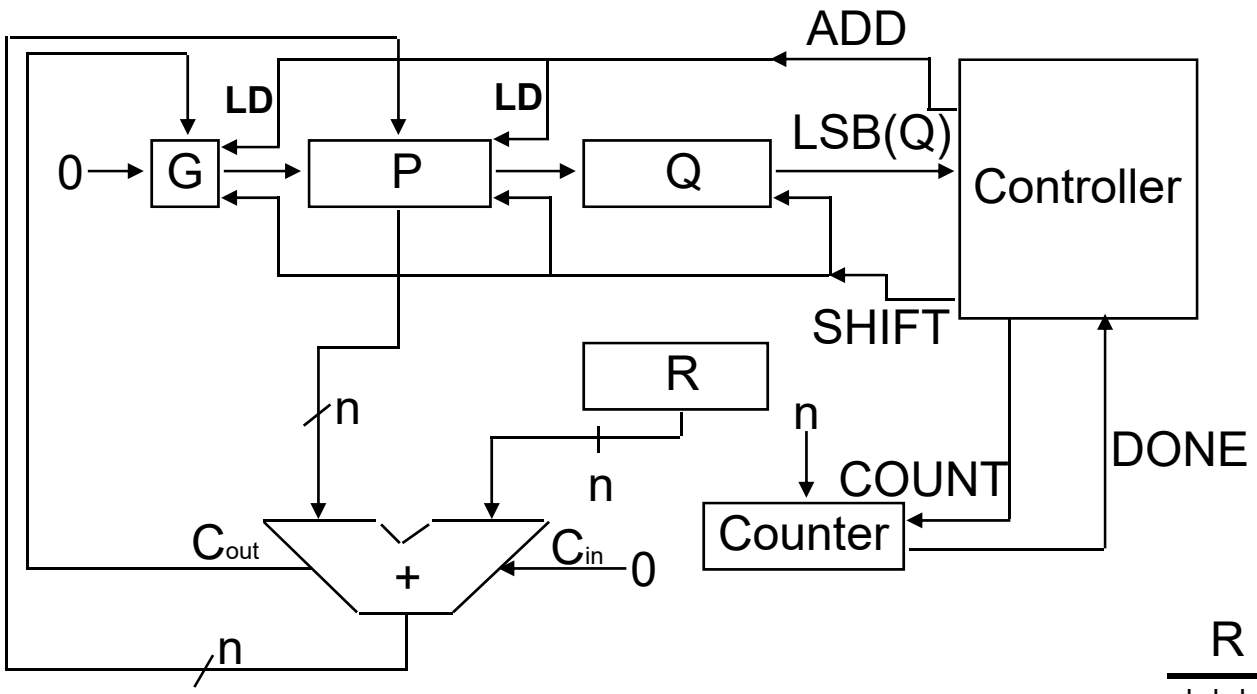
    Delay (roughly) = $\tau_g + \lceil \log_2 n \rceil \times (2 + 4\lceil \tfrac{1}{2} \log_2 2n \rceil)\tau_g$

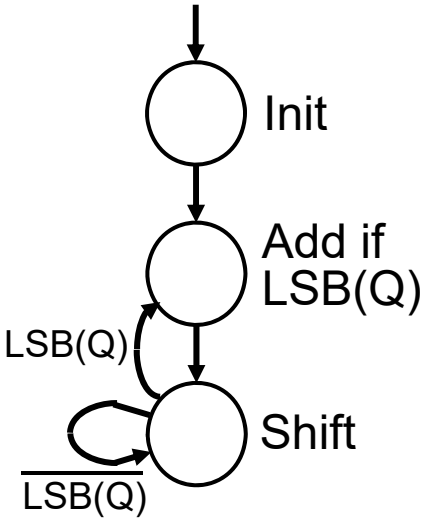
Sequential Multipliers

- The hardware size of the combinational multiplier designs may be too big, and so a variety of more compact sequential multiplier designs can be considered.
- Sequential multiplier designs allow hardware elements to be re-used over multiple clock cycles, thus reducing the required hardware size.



Unsigned Sequential Multiplier






```
  1101
 x1011
--------
 10001111
```

R = multiplicand

Q = multiplier, which is consumed

G = extra bit of precision for additions

P,Q = product, which is produced

R	G	P	Q	Operation
ϕϕϕϕ	ϕ	ϕϕϕϕ	ϕϕϕϕ	
1101	0	0000	1011	Initialize
1101	0	1101	1011	Add
1101	0	0110	1101	Shift
1101	1	0011	1101	Add
1101	0	1001	1110	Shift
1101	0	0100	1111	Shift
1101	1	0001	1111	Add
1101	0	1000	1111	Shift

Signed Multiplication Using Sign Extension

- Until now, we have considered unsigned multiplication, where both the operands are taken to be positive. What if one of the two operands is negative?
- One approach is to convert both operands to positive integers, then perform an unsigned multiplication, determine the required sign of the product from the original operands, and then adjust the sign of the product. However, negating 2's comple. numbers implies cost.
- There are more elegant multiplication algorithms that operate on 2's complement numbers directly.
- Consider negative multiplicands.
- Extending the sign of the partial products allows negative multi-plicands to be handled, but negative multipliers require special treatment.

```
      10011   = -13(10)
    x01011   = +11(10)
   ----------
   111 10011
   111 10011
   000 00000
     1 10011
 + 00000
 ----------
 10 101110001   = -143(10)
```

Extended signs shown in bold.

Why does sign extension work?

Case 1 (positive partial product):

Adding 0's to the left of a positive partial product to extend a number to $m > n$ bits will have no effect on the final product, assuming the standard positional representation.

Case 2 (negative partial product):

Let P be the magnitude of a negative partial product.
Given n bits, -P is represented as $[(2^n - 1)-P] +1$.
Consider increasing the bit width from n to $m > n$.
Adding 1's to the left of the sign bit is equivalent to adding $(2^m-1) - (2^n -1)$ in the positional representation.
But $[(2^n-1)-P] +1+ [(2^m-1)-(2^n-1)] = [(2^m-1)-P]+1$
= 2's complement representation of -P in m bits.

Conclusion: Extending the sign (0 or 1) of a partial product does not change the value of the partial product.

Booth Recoded Multipliers

- Negative multipliers can be readily handled if they are first "recoded" from 2's-complement numbers to **ternary** vectors

x_i	x_{i-1}	y_i	Meaning
0	0	0	No string of 1's in window
0	1	+1	Start of a string of 1's
1	0	-1	End of a string of 1's
1	1	0	Within a string of 1's

- At the rightmost end of the input vector, the next digit to the right is assumed to be a 0.

Examples:

```
0 1 0 1 0       =>   +1 -1 +1 -1  0
1 0 0 1 1 0     =>   -1  0 +1  0 -1  0
1 0 0 1 1       =>   -1  0 +1  0 -1
0 1 1 0 1 0 1   =>
```

The Booth Multiplication Algorithm (1)

- Works for arbitrary combinations of positive and negative integers in 2's complement representation.
- ***Step 1***: Recode the multiplier.
- ***Step 2***: Form and store the 2's complement of the multiplicand.
- ***Step 3***: Perform the binary multiplication, taking care to sign-extend the multiplicand +M (or its 2's complement negation -M) when the partial products are summed.

```
  1 0 1 1
x 1 1 0 1

  (-5)
x(-3)
```

```
   1  0  1  1
x  0 -1 +1 -1

+M = 1 0 1 1
-M = 0 1 0 1
```

```
                1  0  1  1
             x  0 -1 +1 -1
-------------------------------
  0  0  0  0  1  0  1
  1  1  1  0  1  1
  0  0  1  0  1
  0  0  0  0
-------------------------------
1 | 0  0  0  1  1  1  1 |

= +15
```

The Booth Multiplication Algorithm (2)

Another example:

(45)	0 1 0 1 1 0 1
× (-30)	× 1 1 0 0 0 1 0

+M = 0 1 0 1 1 0 1
-M = 1 0 1 0 0 1 1

						0	1	0	1	1	0	1
						x 0	-1	0	0	+1	-1	0
0	**0**	**0**	**0**	**0**	**0**	0	0	0	0	0	0	0
1	**1**	**1**	**1**	**1**	1	0	1	0	0	1	1	
0	**0**	**0**	**0**	0	1	0	1	1	0	1		
0	**0**	**0**	0	0	0	0	0	0	0			
0	**0**	0	0	0	0	0	0	0				
1	1	0	1	0	0	1	1					
0	0	0	0	0	0	0						
1	1	1	0	1	0	1	0	1	1	1	0	1	0

= -1350

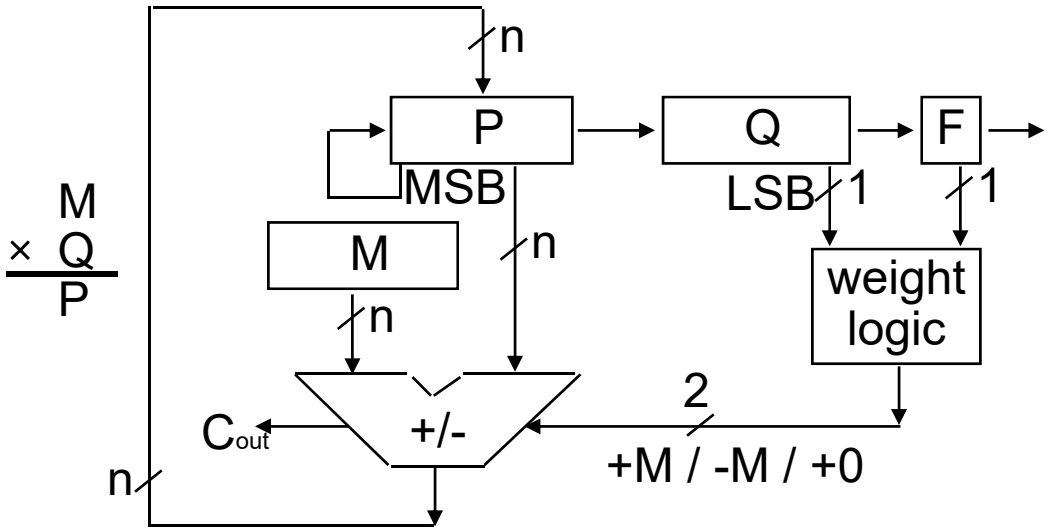
The Booth Multiplication Algorithm (3)

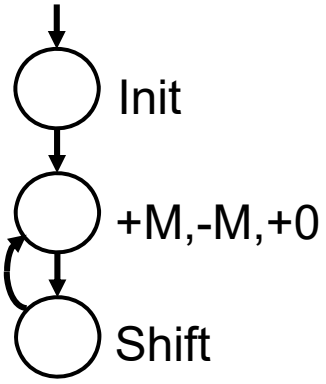
Two more examples:

13	0 1 1 0 1
×(- 6)	× 1 1 0 1 0

(- 44)	1 0 1 0 1 0 0
× (- 19)	× 1 1 0 1 1 0 1

Sequential Implementation of Booth Multiplication





Note: shift register P and flip-flop F are cleared at initialization.

00 → +0
01 → +M
10 → -M
11 → +0

13
(- 6)

+M = 0 1 1 0 1
-M = 1 0 0 1 1

P	Q	F	Operation
ϕϕϕϕϕ	ϕϕϕϕϕ	ϕ	
00000	11010	0	Initialize
00000	11010	0	+ 0
00000	01101	0	Shift right
10011	01101	0	- M
11001	10110	1	Shift right
00110	10110	1	+ M
00011	01011	0	Shift right
10110	01011	0	- M
11011	00101	1	Shift right
11011	00101	1	+ 0
11101	10010	1	Shift right

-78

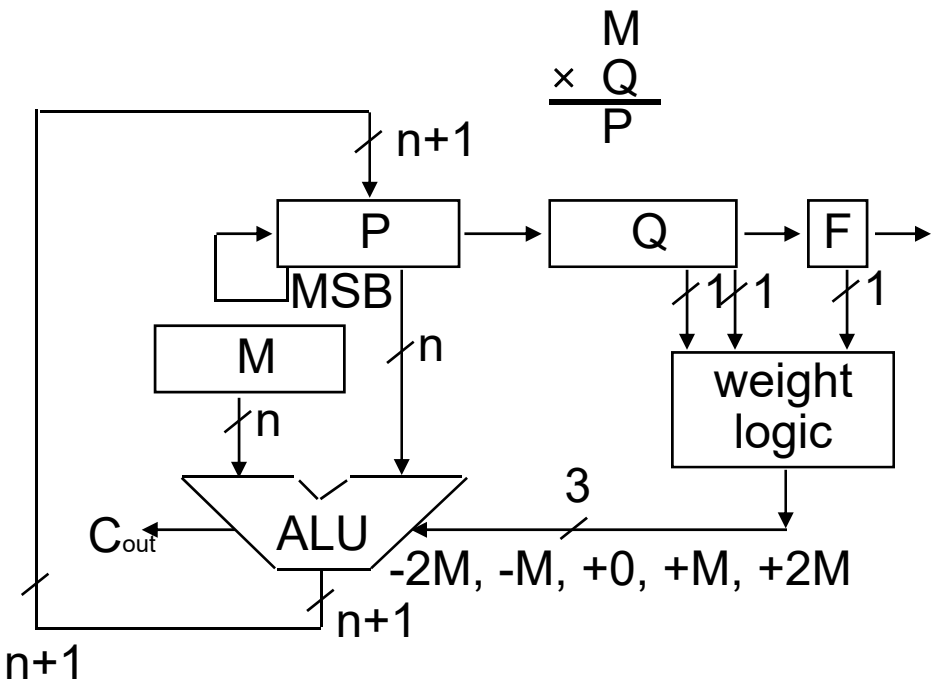
How to Speed Up Booth Multiplication?

- The Booth algorithm can be sped up if the 1's in the multiplier are bunched together since such strings cause all-0 partial products that could be skipped over.
- However, this method produces data-dependent speed-up, which is awkward to schedule in a larger digital system.
- Another idea is to add groups of partial products.
- Specifically, we can extend the Booth recoding to encode two Booth bits at a time over a sliding window of three multiplier bits (instead of two multiplier bits).
- When two Booth bits are combined, the more significant bit is weighted by a factor of two.
- The resulting Booth bits have values +2, +1, 0, -1 and -2. The factors of two are easily handled by left-shifts by one bit position.
- ***Fast multiplication***, with two multiplier bits processed together, is also called ***radix-4 multiplication*** because pairs of bits have ***four*** values.

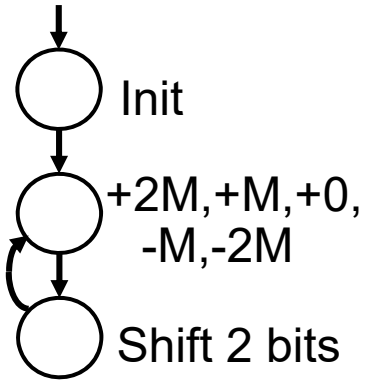
Booth Recoding for Fast (Radix-4) Multiplication

x_{i+1}	x_i	x_{i-1}	Booth Bits		Combined Booth Bit y_i	
0	0	0	0	0	0	= 2(0)+0
0	0	1	0	+1	+1	= 2(0)+1
0	1	0	+1	-1	+1	= 2(+1)-1
0	1	1	+1	0	+2	= 2(+1)+0
1	0	0	-1	0	-2	= 2(-1)+0
1	0	1	-1	+1	-1	= 2(-1)+1
1	1	0	0	-1	-1	= 2(0)-1
1	1	1	0	0	0	= 2(0)+0

Sequential Implementation of Fast Multiplication



000 → +0
001 → +M
010 → +M
011 → +2M
100 → -2M
101 → -M
110 → -M
111 → +0



13
× (- 6)

+M = 0 0 1 1 0 1
+2M = 0 1 1 0 1 0
-M = 1 1 0 0 1 1
-2M = 1 0 0 1 1 0

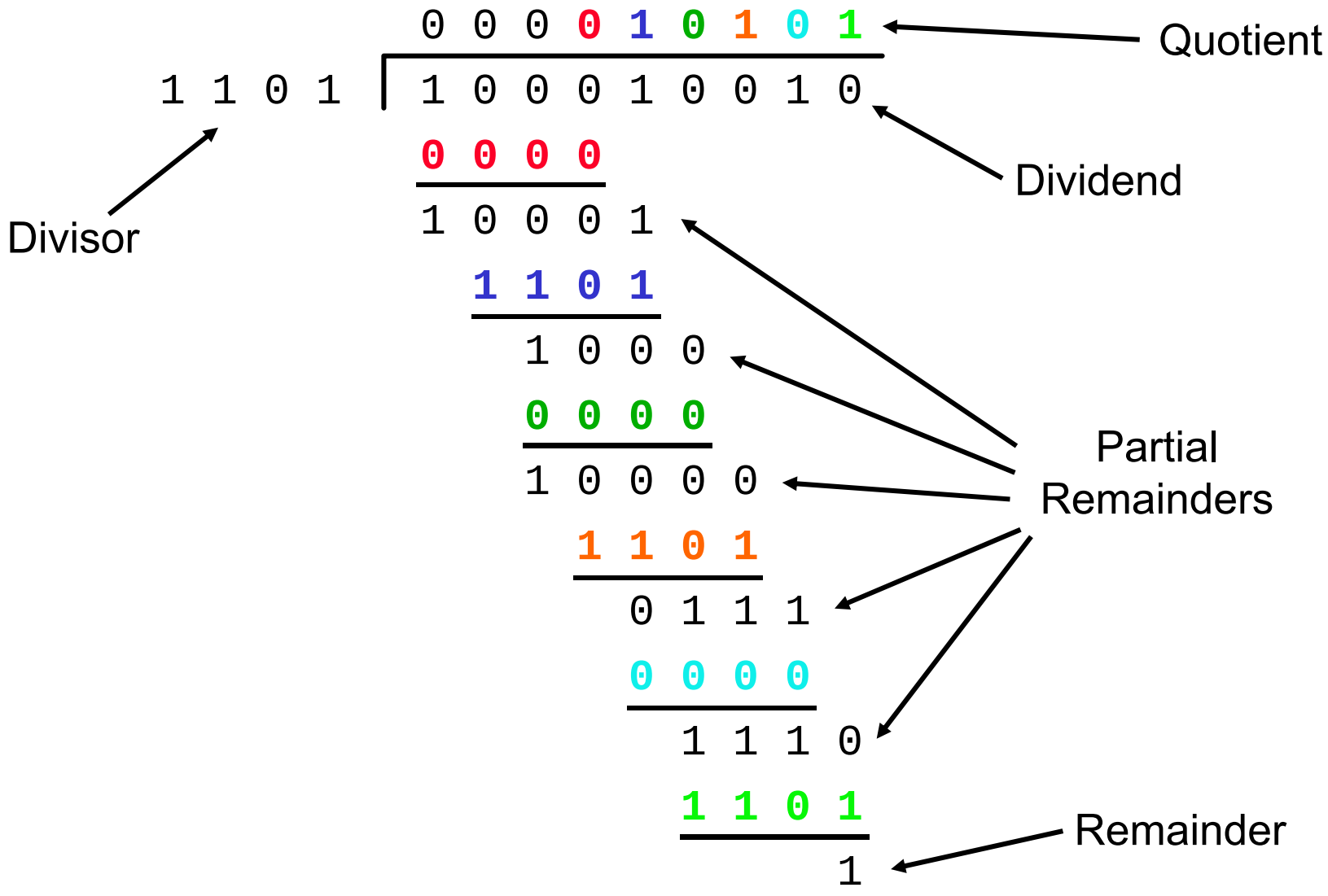
P	Q	F	Operation
ϕϕϕϕϕϕϕ	ϕϕϕϕϕϕ	ϕ	
0000000	111010	0	Initialize
1100110	111010	0	-2M
1111001	101110	1	Shift 2 bits
1101100	101110	1	-M
1111011	001011	1	Shift 2 bits
1111011	001011	1	+0
1111110	110010	1	Shift 2 bits

-78_{10}

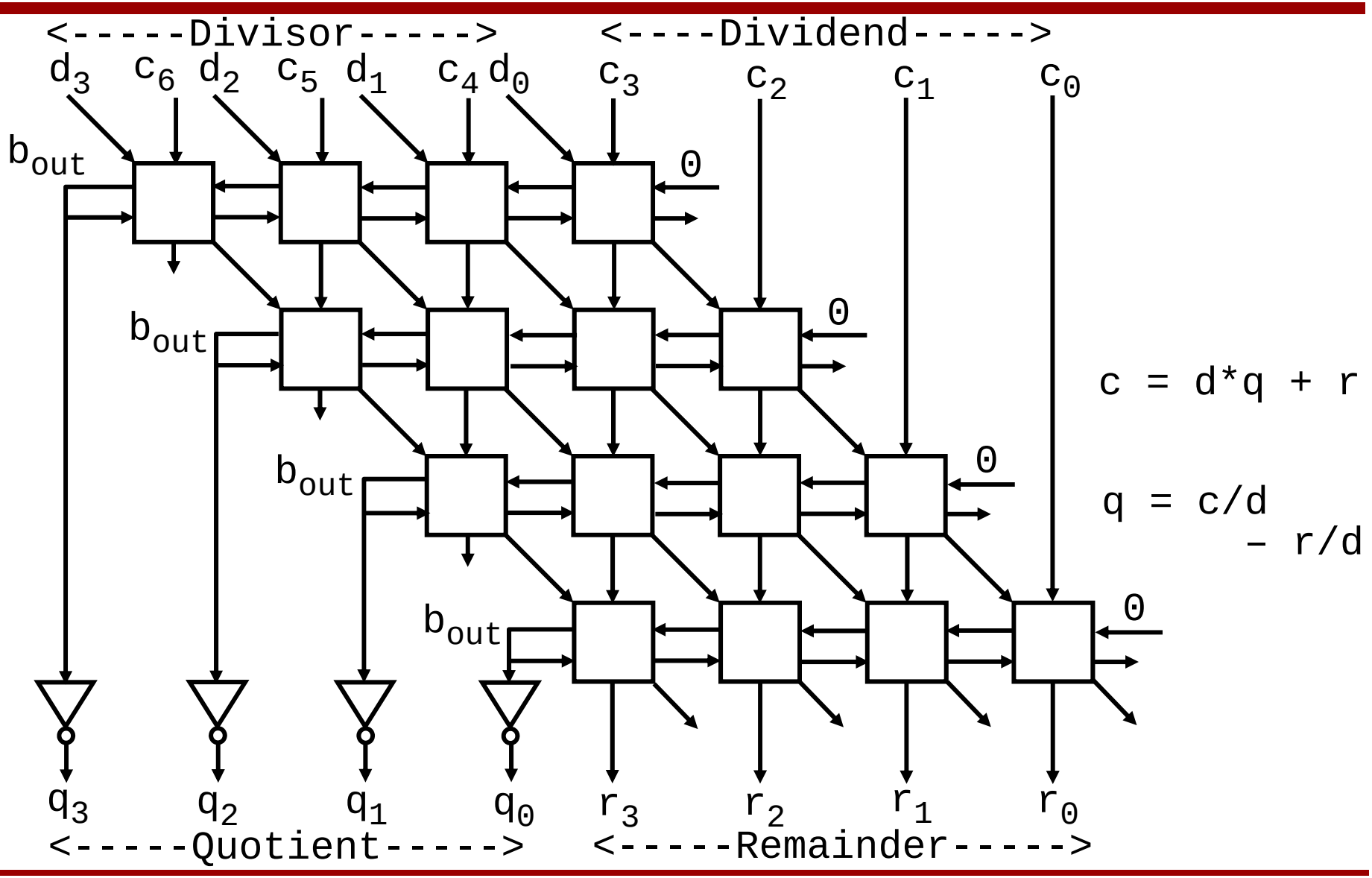
Unsigned Binary Division

- Binary multiplication involves shift and addition operations; similarly, binary division involves ***shifts*** and ***conditional subtractions***.
- The presence of conditional subtractions, however, makes division a slower operation than multiplication. Because of this, it is common to attempt to restructure numerical algorithms to minimize the number of required divisions.
- Division terminology: ***dividend*** = (***divisor*** × ***quotient***) + ***remainder***
- Division is sometimes implemented as a reciprocal operation on the divisor followed by a multiplication by the dividend. This especially makes sense when multiple divisions are to be made using the same divisor with multiple dividends (e.g., A_1/B, A_2/B, A_3/B, implemented as B^{-1} followed by $A_1 \times B^{-1}$, $A_2 \times B^{-1}$, $A_3 \times B^{-1}$, etc.).
- As with binary multiplication, binary division can be implemented using an array architecture motivated by the structure of the familiar hand division algorithm.

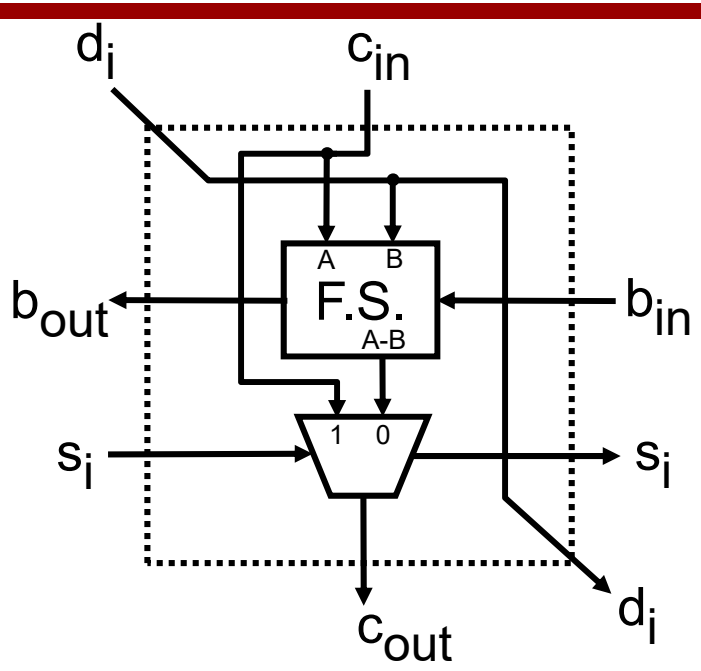
Binary Division Using the Hand Algorithm



Purely Combinational Array Divider



Combinational Divider Cell



c_{in}	d_i	b_{in}	s_i	A-B	c_{out}	b_{out}
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	0	1	1	1
0	0	1	1	1	0	1
0	1	0	0	1	1	1
0	1	0	1	1	0	1
0	1	1	0	0	0	1
0	1	1	1	0	0	1
1	0	0	0	1	1	0
1	0	0	1	1	1	0
1	0	1	0	0	0	0
1	0	1	1	0	1	0
1	1	0	0	0	0	0
1	1	0	1	0	1	0
1	1	1	0	1	1	1
1	1	1	1	1	1	1

c_{out}

$c_{in}d_i$ \ $b_{in}s_i$	00	01	11	10
00	0	0	0	1
01	1	0	0	0
11	0	1	1	1
10	1	1	1	0

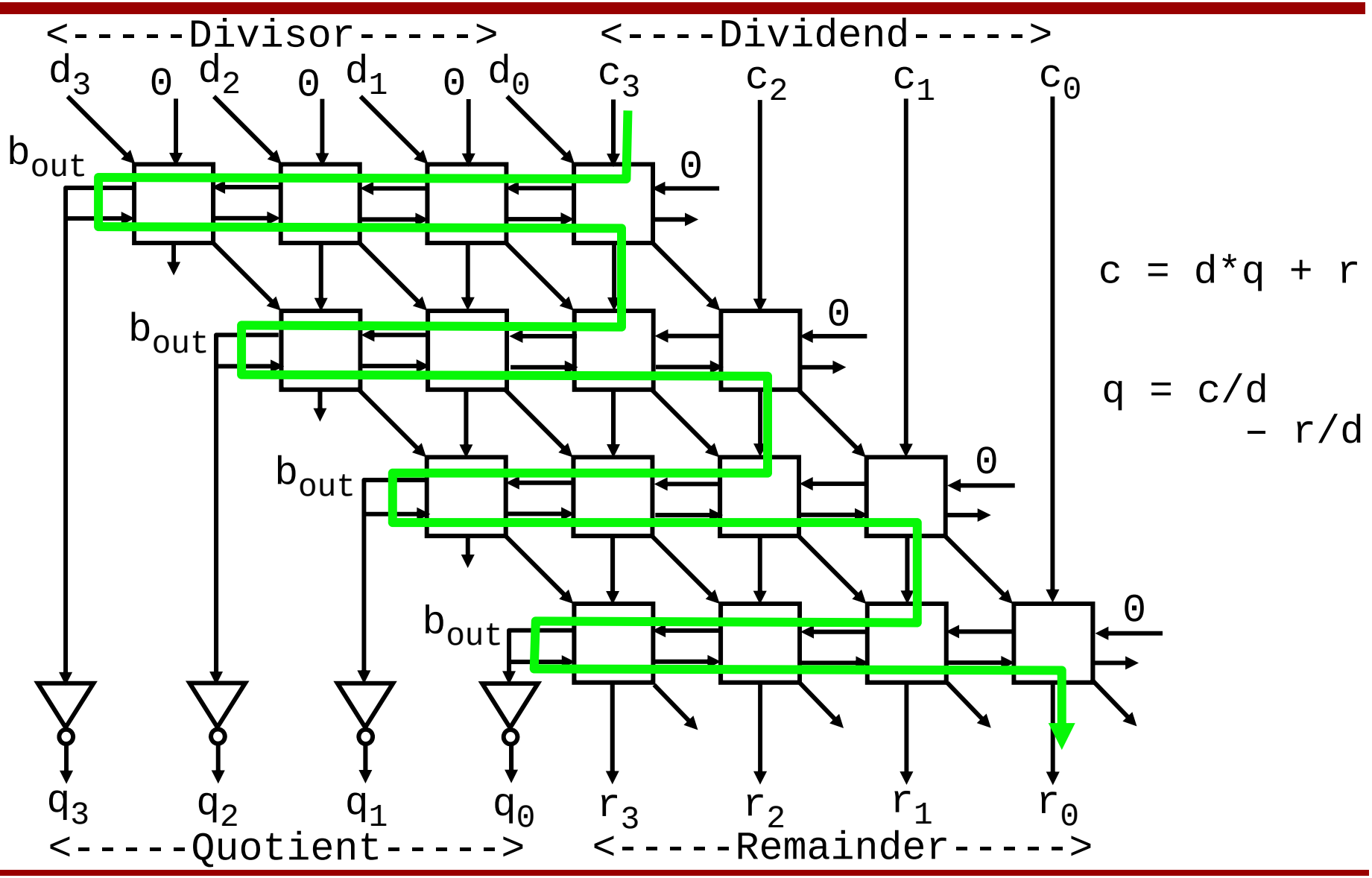
$3\tau_g$

b_{out}

$c_{in}d_i$ \ $b_{in}s_i$	00	01	11	10
00	0	0	1	1
01	1	1	1	1
11	0	0	0	0
10	0	0	1	1

$3\tau_g$

Purely Combinational Array Divider



d_3 0 d_2 0 d_1 0 d_0 c_3 c_2 c_1 c_0

b_{out} 0

b_{out} 0

b_{out} 0

b_{out} 0

$c = d*q + r$

$q = c/d$
$\quad - r/d$

q_3 q_2 q_1 q_0 r_3 r_2 r_1 r_0

<-----Quotient-----> <-----Remainder----->

Critical Path Analysis

- ***Initial Observation***: The critical path passes through all of the cells in the worst case. Thus the worst-case delay will be of order $O(n^2)$.
- Delay for the first conditional subtraction:

$$\text{Delay} = (\text{divisor\_width})(3\tau_g) + 2\tau_g = (3n + 2)\tau_g$$

- Delay for all the remaining (dividend_width – 1) conditional subtractions:

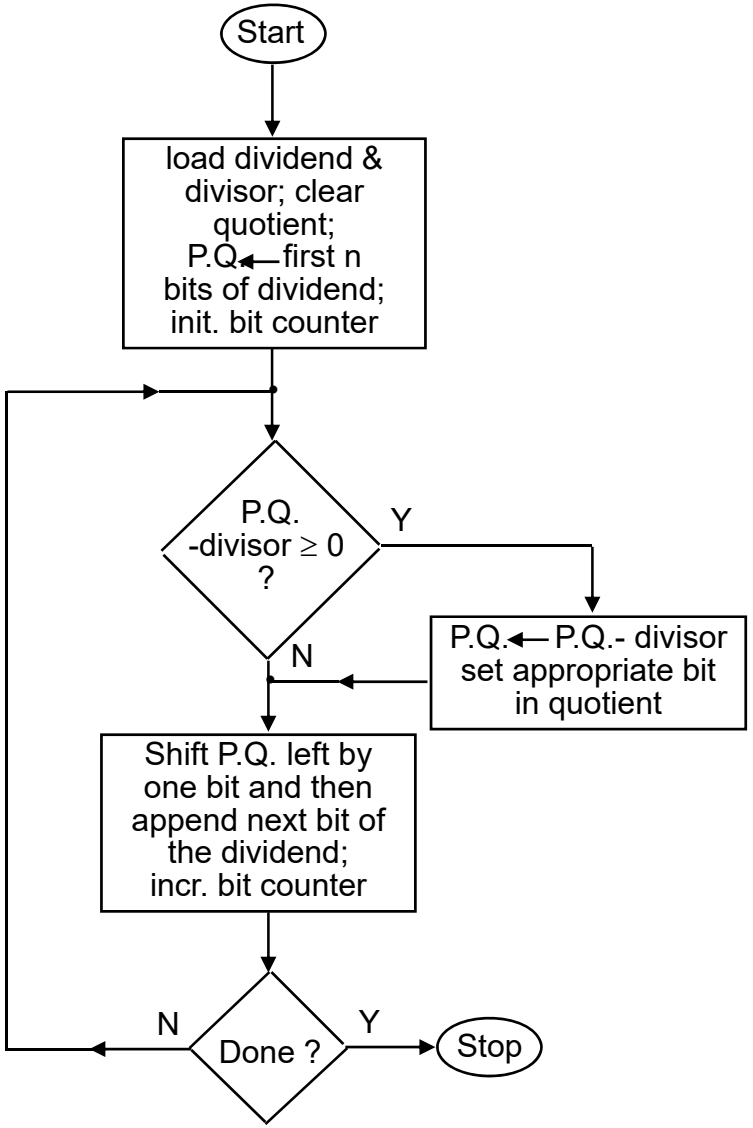
$$\text{Delay} = (\text{divisor\_width} - 1)(3\tau_g) + 2\tau_g = (3n - 1)\tau_g$$

- Assuming divisor_width = dividend_width = n, then the critical path delay is:

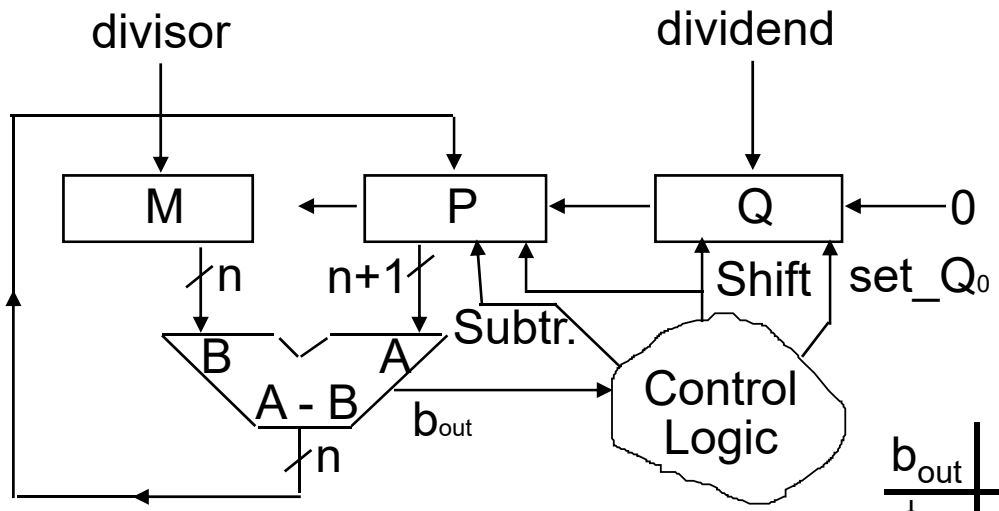
$$\text{Total delay} = (3n + 2)\tau_g + (n\text{-}1)(3n - 1)\tau_g = (3n^2 - n + 3)\tau_g$$

- The worst-case delay is indeed of order $O(n^2)$.

Sequential Binary Division Algorithm



Implementation of Sequential Binary Division



M
ϕϕϕϕϕ
01101

Example:

1 1 0 1 | 1 0 0 0 1 0 0 1 0

b_{out}	P	Q	Operations
ϕ	ϕϕϕϕϕ	ϕϕϕϕϕϕϕϕϕϕ	Load M & Q, clear P
1	00000	1000100100	Shift P, Q
1	00001	0001001000	Shift P, Q
1	00010	0010010000	Shift P, Q
1	00100	0100100000	Shift P, Q
1	01000	1001000000	Shift P, Q
0	10001	0010000000	Subtr. + set Q_0
1	00100	0010000001	Shift P, Q
1	01000	0100000010	Shift P, Q
0	10000	1000000100	Subtr. + set Q_0
1	00011	1000000101	Shift P, Q
1	00111	0000001010	Shift P, Q
0	01110	0000010100	Subtr. + set Q_0
1	00001	0000010101	

Possible Exceptions in Binary Division

1) ***Divide by zero*:**

- The result is undefined mathematically.
- Unusual answers can be produced by division hardware. What's worse, those answers are potentially hazardous to use in further calculations.

2) ***Divide Overflow*:**

- Often the divisor occupies one word (*n* bits) and the dividend occupies two words (2*n* bits).
- However, it is possible that the resulting quotient will occupy *n*+1 bits when only *n* bits can be stored in a word.
- This condition is called *divide overflow*. For example:

  11001010 ÷ 1010 = 10100, remainder = 10

***Solutions*:**

- Depend on software to avoid exceptions before they occur (check for a zero divisor just before starting every division).
- Add special hardware that detects exceptions. If an exception is detected, a hardware interrupt can then be triggered. Recover in software.

Floating-Point Representation of Numbers

Floating-Point Notation *(also called Scientific Notation)*:

Ex: 3 155 760 000 = 3.15576×10^9 seconds/century

- The decimal point can be "moved" or "floated" left (or right) to different positions in the *significand* as long as the *exponent* in the power factor is increased (or decreased) accordingly.

- This flexibility allows a much greater range of numbers to be represented with the same number of digits.

Normalized Scientific Notation:

- Same as scientific notation, except that there must be exactly one nonzero digit to the left of the *decimal point*.

- In normalized binary scientific notation, the bit to the left of the *radix point* must be 1 for every nonzero number. Zero is a special case that must be encoded differently.

Issues in Floating-Point Representation

Given: A fixed word width. For example, the width could be set to 16.

Determine:

1) The number of bits to allocate to the significand.
2) The number of bits to allocate to the exponent.
3) How to represent negative numbers.

Trade-off:

Larger Exponent => larger total range of representable numbers

Larger Significand => smaller gap between representable numbers

Exceptions:

Overflow: The desired number has a magnitude greater than the largest representable number.

Underflow: The desired number has a magnitude smaller than the smallest representable number.

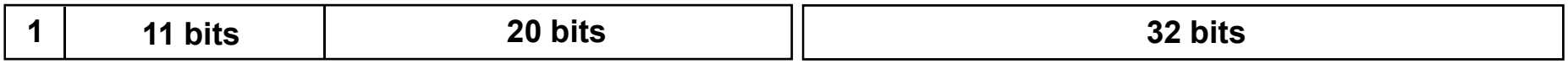
IEEE Std 754 Floating-Point Formats

Single-Precision Format: (32 bits wide)

Sign Biased exponent
e + 127

Significand field
s = 1.xxxxxxxx (The left bit is omitted)

Double-Precision Format: (64 bits wide)



Sign Biased exponent
e + 1023

Significand field
s = 1.xxxxxxxx (The left bit is omitted)

Why use biased exponents?

- The biased exponent field uses unsigned integers.
- This simplifies the sorting of floating-point numbers.

The Range of Single-Precision Numbers

Smallest positive single-precision normalized number

0	00000001	000 . . . 0000

$$1.00000000000000000000000_2 \times 2^{-126} \approx 1.2 \times 10^{-38}$$

Largest positive single-precision normalized number

0	11111110	111 . . . 1111

$$1.11111111111111111111111_2 \times 2^{+127} \approx 3.4 \times 10^{38}$$

The Range of Double-Precision Numbers

Smallest positive double-precision normalized number

0	000...001	000 . . . 0000

$$1.00000000000000000000000_2 \times 2^{-1022} \approx 2.2 \times 10^{-308}$$

Largest positive double-precision normalized number

0	111...110	111 . . . 1111

$$1.11111111111111111111111_2 \times 2^{+1023} \approx 1.8 \times 10^{308}$$

Single-Precision Examples

Example #1: Determine the IEEE 754 single-precision representation of –6.375

$-6.375_{10} = -110.011_2 = -1.10011_2 \times 2^2$

Biased exponent = $2 + 127 = 129_{10} = 10000001_2$

1	10000001	10011000000000000000000

Example #2: Determine what decimal number is represented by the following bits interpreted as a single-precision IEEE 754 value

0	01111100	0100 · · · 0

Special IEEE Std 754 Numbers (1)

1) ***Zero***
   - All 0's in the exponent and significand fields.
   - The sign bit is ignored (it is a don't care).
   - There is no "negative zero".

2) ***Positive infinity*** **(+inf)**
   - 0 in sign bit; all 1's in exponent; all 0's in significand.
   - The result of dividing a positive number by 0.
   - 1 / +inf is defined to have value 0.

3) ***Negative infinity*** **(-inf)**
   - 1 in sign bit; all 1's in exponent; all 0's in significand.
   - The result of dividing a negative number by 0.
   - 1 / -inf is defined to have value 0.

Special IEEE Std 754 Numbers (2)

4) *Not-a-Number* (NaN)

- All 1's in exponent; nonzero significand
- Ex. #1: The result of taking the square root of a negative number (unexpected imaginary number)
- Ex. #2: The result of computing 0/0
- If an argument to a function is NaN, then the result of the function must also be NaN. That is, NaN propagates "safely" through all functions to the end of a calculation.

5) *Denormal* (also called subnormal) numbers

- All 0's in exponent; nonzero significand (hidden bit is now 0).
- Used to implement "gradual underflow" (useful for representing numbers lying in between the smallest normalized number and zero). Not all floating-point hardware will support denormals.
- Example: $0.0001_2 \times 2^{-126}$ is a denormal.
- Some implementations of IEEE Std 754 do not support denormals. Instead, they "flush" denormals down to zero.

IEEE Std 754 Representation Summary

Single-Precision Numbers

Exponent	*Significand*	*Meaning*
0	0	0
0	nonzero	+/- denormal
1-254	any value	+/- floating-point number
255	0	+/- infinity
255	nonzero	NaN

Double-Precision Numbers

Exponent	*Significand*	*Meaning*
0	0	0
0	nonzero	+/- denormal
1-2046	any value	+/- floating-point number
2047	0	+/- infinity
2047	nonzero	NaN

Algorithm for Floating-Point Addition

Step 1: Compare the exponents. Shift the significand of the smaller number to the right (while increasing its exponent) until the two exponents are equalized.
This step is called *alignment shift* or *preshift*.

Step 2: Add / subtract the significands.

Step 3: Normalize the result by either: (a) shifting the significand right while incrementing the exponent, or (b) shifting the significand left while decrementing the exponent.
This step is called *normalization shift* or *postshift*.
Check for overflow or underflow.

Step 4: Round the significand to the available number of bits.
23 + 1 hidden bit in IEEE single-precision format.
52 + 1 hidden bit in IEEE double-precision format.

Step 5: If the result is not normalized, then repeat Steps 3 and 4.

Example of Floating-Point Addition (1)

Decimal	*Fixed-Point Binary*	*Normalized Floating-Point Binary*
0.5_{10}	0.1_2	$(+1) \times 2^{-1} \times 1.00000000000000000000000_2$
$+(-0.4375_{10})$	$+(-0.0111_2)$	$+ (-1) \times 2^{-2} \times 1.11000000000000000000000_2$
0.0625_{10}		

$-1 + 127 = 01111110_2$

$-2 + 127 = 01111101_2$

Single-Precision IEEE

	0	01111110	00000000000000000000000
+	1	01111101	11000000000000000000000

Step 1: Equalize the exponents

$1.0_2 \times 2^{-1}$	exp = -1	1.00000000000000000000000
$+(-0.111_2 \times 2^{-1})$	exp = -1	0.11100000000000000000000

Step 2: Add / subtract the significands

Subtract the significands

exp = -1	0.00100000000000000000000

Example of Floating-Point Addition (2)

Step 3: Normalize the result

Before:	exp = -1	0.00100000000000000000000
After:	exp = -4	1.00000000000000000000

Step 4: Round the significand to the available size

Before:	exp = -4	1.00000000000000000000
After:	exp = -4	1.00000000000000000000

Assume 1 + 23 bits here

Step 5: If the result is not normalized, repeat steps 3 & 4

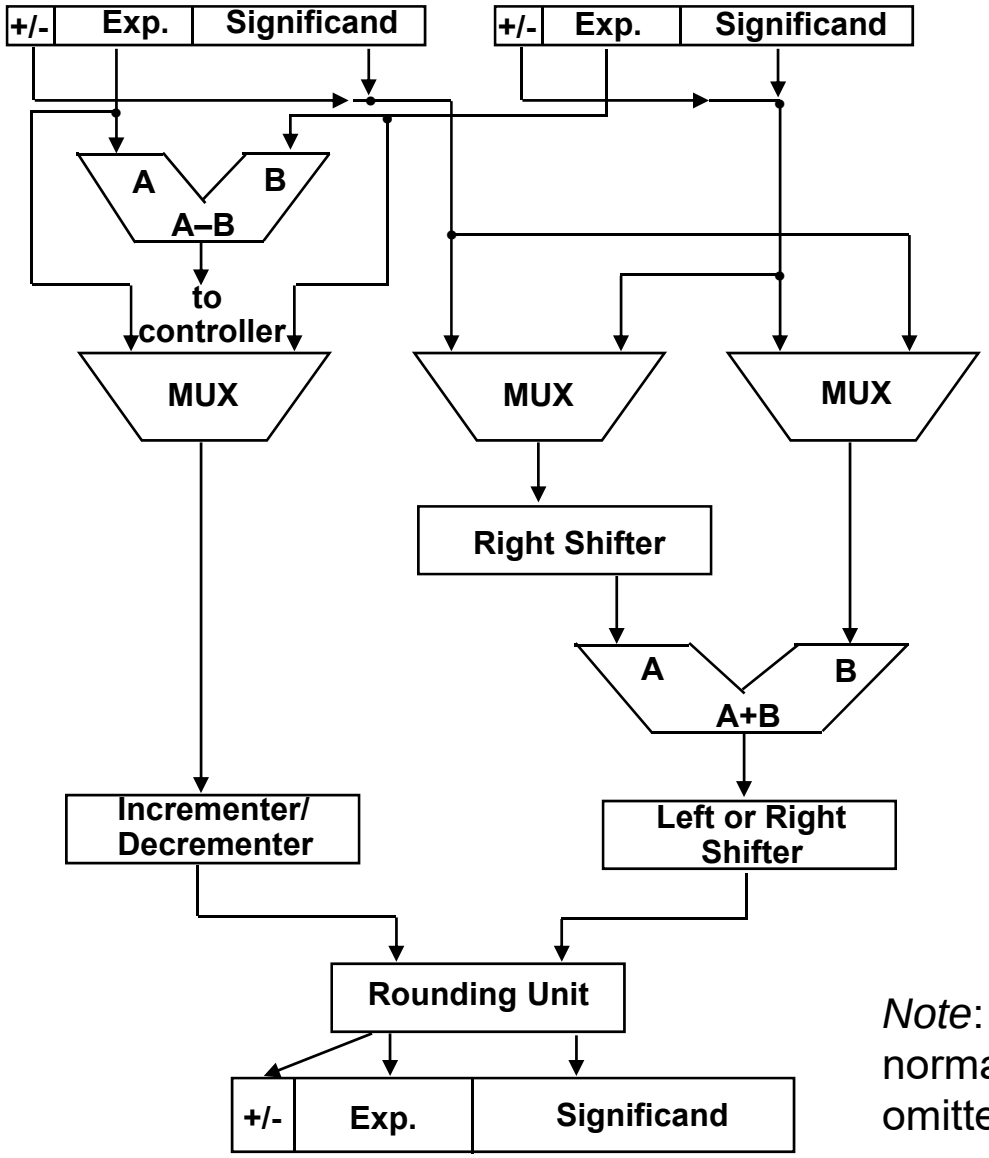
In this example, rounding did not denormalize the result, so there is no need for the second normalization step.

$-4 + 127 = 01111011_2$

Result in Single-Precision IEEE

0	01111011	00000000000000000000

Datapath for Floating-Point Addition



Note: Hardware for a second normalization step has been omitted here.

Floating-Point Multiplication

Step 1: Add the exponents. Subtract the extra bias from the sum (only if biased exponents were used, as in IEEE format).

Step 2: Multiply the significands using fixed-point multiplication.

Step 3: If necessary, normalize the product resulting from Steps 1 & 2. Either: (a) shift the significand right and add to the exponent, or (b) shift the significand left and subtract from the exponent. Check for overflow and underflow.

Step 4: Round the number to the available number of digits. Repeat steps 3 and 4 if the result is not normalized.

Step 5: Obtain the correct sign of the product from the signs of the original operands.

Example of Floating-Point Multiplication (1)

Decimal	*Fixed-Point Binary*	*Normalized Floating-Point Binary*
6.25_{10}	110.01_2	$(+1) \times 2^{+2} \times 1.10010000000000000000000$
$\times (-84.0_{10})$	$\times (-1010100.0_2)$	$\times (-1) \times 2^{+6} \times 1.01010000000000000000000$
-525.0_{10}		

$+2 + 127 = 10000001_2$

$+6 + 127 = 10000101_2$

Single-Precision IEEE

	0	10000001	10010000000000000000000
×	1	10000101	01010000000000000000000

Step 1: Add the exponents. Subtract the extra bias from the sum.

Without bias: $+2 + 6 = +8$

With bias: $(+2 + 127) + (6 + 127) - 127 = (+8 + 127) = 10000111_2$

Step 2: Multiply the (unsigned) significands.

$1.1001_2 \times 1.0101_2 = 10.00001101_2$

Example of Floating-Point Multiplication (2)

Step 3: Normalize the product.

Un-normalized product = $10.00001101_2 \times 2^{+8}$

Normalized product = $1.000001101_2 \times 2^{+9}$

Step 4: Round the result.

No need for rounding in this example.

$+9 + 127 = 10001000_2$

Single-Precision IEEE

?	10001000	00000110100000000000000

Step 5: Obtain the correct sign.

Use the "xor" operation: 0 xor 1 = 1 (negative product).

Single-Precision IEEE

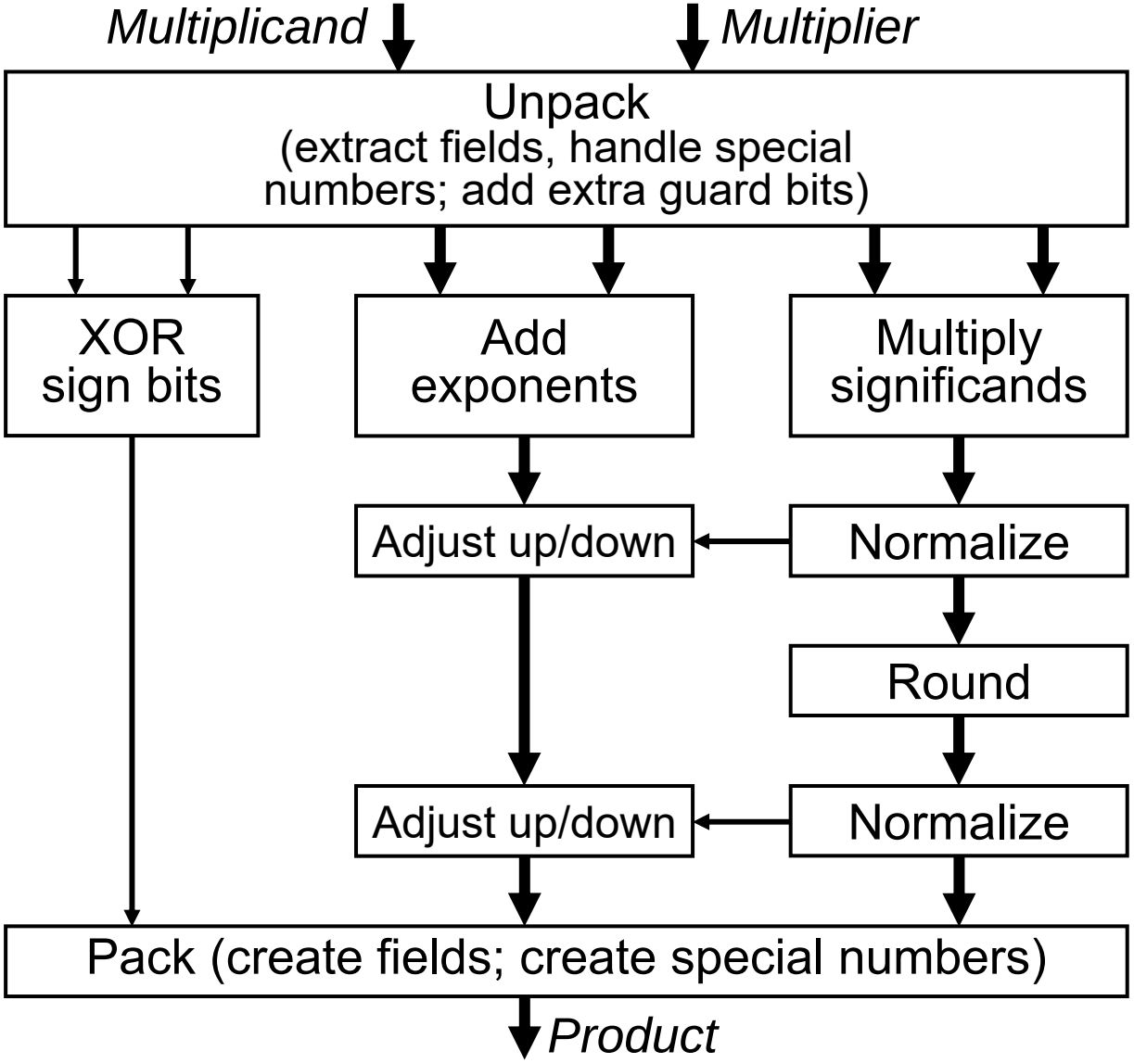
1	10001000	00000110100000000000000

$- 1.000001101_2 \times 2^{+9} = -525_{10}$

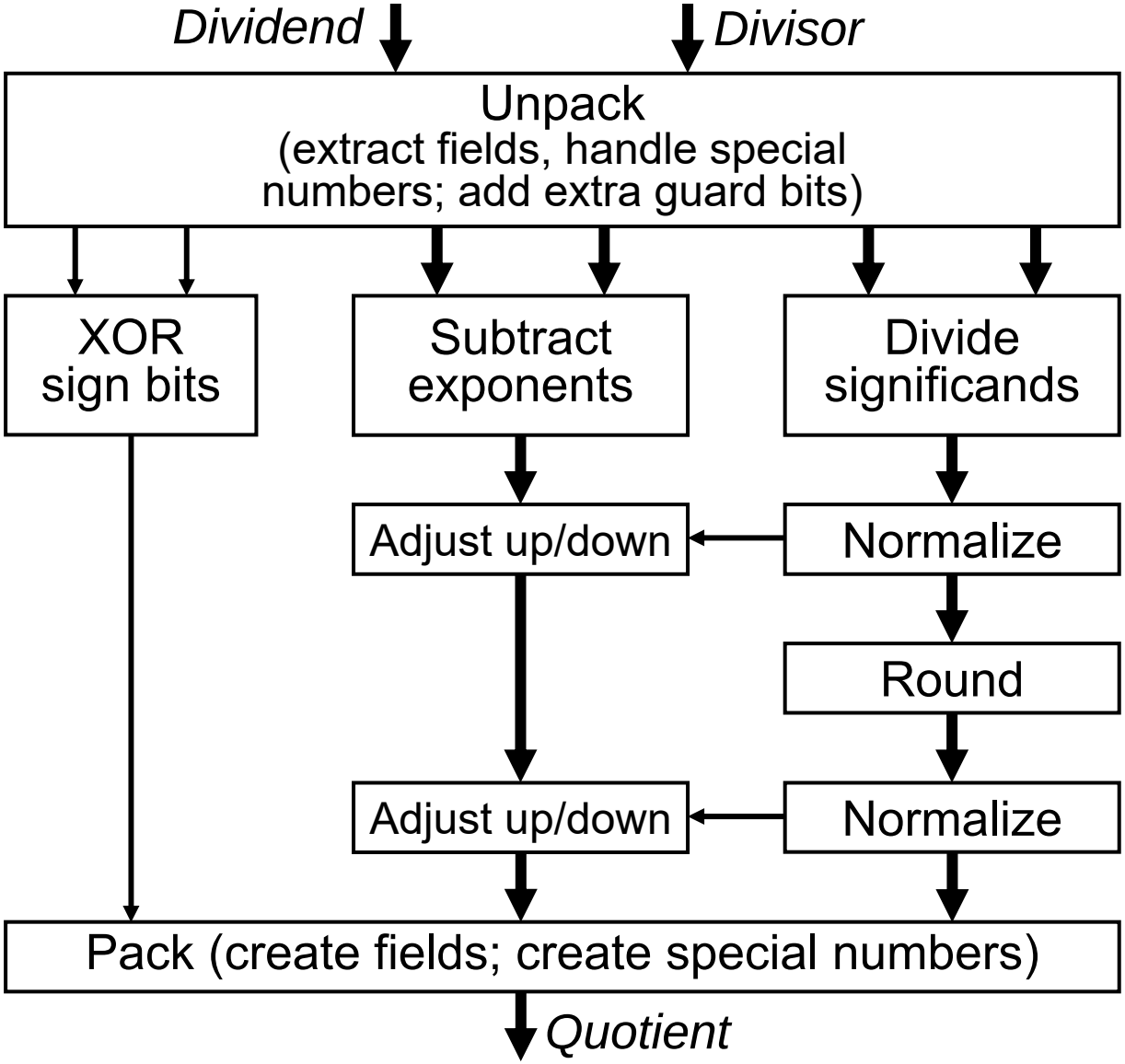
Rounding Schemes

- A ***rounding*** operation is required to convert a higher-precision number into a lower-precision number.
- Rounding by simply truncating the extra bits has different systematic effects on sign+magnitude and 2's-complement numbers:
  - Most sign+magnitude numbers have their magnitude decreased
  - Most 2's-complement numbers are shifted towards neg. infinity.
- Conventional rounding to the nearest integer introduces a slight bias because of the treatment of "midpoint" numbers.
  - Sign+magnitude midpoint numbers all get greater magnitudes
  - 2's-complement midpoint numbers all shift towards positive infinity
- To avoid such systematic shifts and biases in the midpoint values, IEEE Std 754 uses "round to the nearest even number" by default. It is also possible to specify rounding inward towards zero, upward towards positive infinity, and downward towards negative infinity.

Datapath for Floating-Point Multiplication



Datapath for Floating-Point Division



Final Thoughts on Floating-Point Hardware

- Floating-point multiplication is simpler to implement than floating-point addition/subtraction. Floating-point division requires exponent subtraction as well as binary division of the significands.
- Extra bits of significand (called ***guard bits***) must be included at the least significant end of numbers in floating-point hardware to ensure correct rounding. Three extra bits are sufficient to accomplish this.
- Implementing fully-compliant IEEE floating-point hardware may be "overkill" for some hardware designs. More bits of precision may be required in the exponent and/or the significand by the IEEE standard than are really needed in the particular application.
- Using a custom floating-point format allows the width of the exponent and/or the significand to be adjusted to nonstandard values to get a better range/precision trade-off and/or to save hardware (and power). However, many simulation experiments will likely be required to determine safe data widths to use in the intermediate results.

Function Evaluation

- It is often desirable to have efficient hardware implementations for standard mathematical functions such as reciprocal (x^{-1}); square root ($x^{0.5} = \sqrt{x}$); square (x^2); natural exponentiation (e^x); power (x^y); natural logarithm (ln x); sine (sin x); cosine (cos x); etc.
- If the given function has periodicity (e.g., sine and cosine), then only a limited domain of the function may need to be implemented (e.g., one quarter cycle of a sine wave). Function values outside of this domain might be obtained by a suitable remapping of the input argument to an in-range input value that has the same function value.
- Depending on the function, there are trade-offs to be made between compactness, speed and accuracy:
  - For greatest speed (but with greatest memory cost) a function can be stored as a *look-up table*, usually combined with *interpolation*.
  - For the least memory space (but slower speed), a function can sometimes be implemented using a *convergent iterative formula*.
  - Often a *hybrid combination* of look-up table with a convergent iterative formula gives the best design (e.g., reciprocal, x^{-1}).

Table-Based Function Evaluation (1)

- A general strategy for implementing arbitrary functions is to use pre-computed data stored in a look-up table. The table stores a finite number of (input value, function value) pairs covering the expected domain of x values. The basic algorithm has three steps:

  Step 1: Given an input value x, locate the two nearest stored input values, x_i and x_{i+1}, on either side of x.

  Step 2: Two memory accesses then yield $f(x_i)$ and $f(x_{i+1})$.

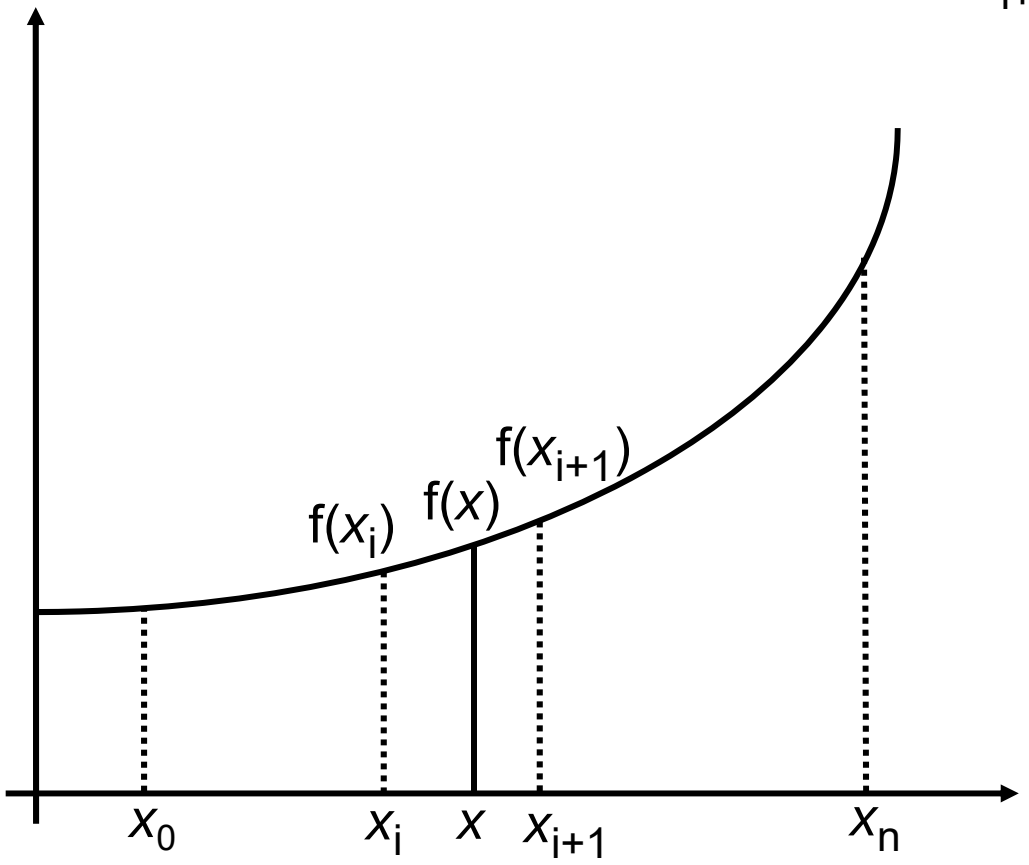
  Step 3: Linear interpolation can be used to compute $f(x)$ as follows:

$$f(x) = f(x_i) + [f(x_{i+1}) - f(x_i)]\frac{x - x_i}{x_{i+1} - x_i}$$

- There is a basic trade-off between table size and function accuracy: A larger table will produce higher function accuracy.
- The stored input values do not have to be equally spaced. For some functions, it may be desirable to have smaller steps in x in regions where $f(x)$ is changing rapidly, and larger steps in x elsewhere.

Table-Based Function Evaluation (2)

$$f(x) = f(x_i) + [f(x_{i+1}) - f(x_i)]\frac{x - x_i}{x_{i+1} - x_i}$$



Speeding Up Table-Based Function Evaluation

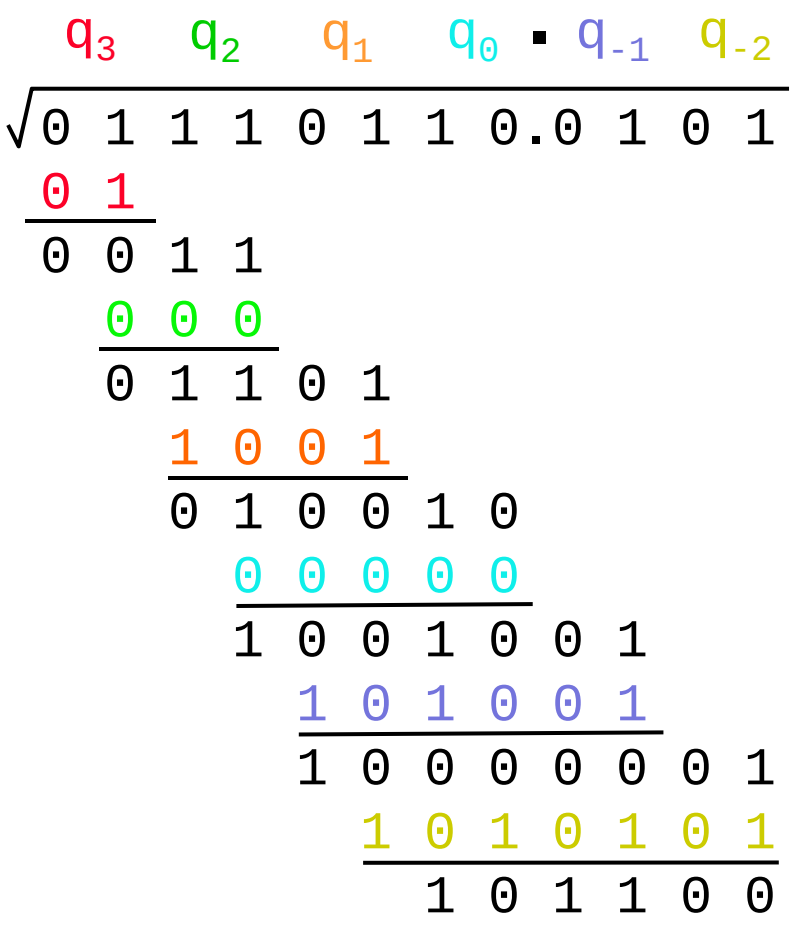
$$f(x) = f(x_i) + [f(x_{i+1}) - f(x_i)]\frac{x - x_i}{x_{i+1} - x_i}$$

- ***Step 1*** (find x_i and x_{i+1}) can be made fast if the stored x values are equally spaced and the address of the stored x values is simply a bit field in x. For example, take the five most significant bits of x, and then set the LSB of this binary field to both 0 and 1 to obtain the memory addresses corresponding to x_i and x_{i+1}.
- ***Step 2*** (table look-up) could be made faster (at the cost of extra space) by accessing two look-up memories in parallel.
- ***Step 3*** (linear interpolation) can be made faster if the difference $x_{i+1} - x_i$ is a power of two. Then the division operation can be accomplished by right shifting.
- Pipelining could be considered if there is a stream of function values that needs to be evaluated.

Iterative Binary Square Root, $x^{0.5}$, Algorithm

1. Group the *radicand*, x, into pairs of bits going out from the radix point. Add a dummy "0" to the left if necessary to ensure an even number of bits. The algorithm produces the bits of $x^{0.5} = q$ going from MSB to LSB. Like in binary subtraction, the radicand is replaced with a sequence of *partial remainders* as bits of the square root q are determined (i.e., as the computed prefix q_{pre} of q grows to the right).
2. Let the first partial remainder be the leftmost two bits of x. Let the first prefix q_{pre} of the square root q be a null string.
3. Attempt to subtract $(4 \times q_{pre})+01_2$ from the current partial remainder.
4. If the result of the subtraction was non-negative, then commit to the subtraction by updating the current partial remainder. Append a "1" to the right end of q_{pre}.
5. If the result of the subtraction was negative, then leave the partial remainder unchanged. Append a "0" to the right end of q_{pre}.
6. Bring down the next two bits of x to form the next partial remainder.
7. If not enough bits of q have been found, then go back to step 3.

Example of the Square Root Algorithm



$= 1010.11$

q_{pre} = "" $(4 \times q_{pre})+1 = 01$ $q_3=1$

q_{pre} = "1" $(4 \times q_{pre})+1 = 101$ $q_2=0$

q_{pre} = "10" $(4 \times q_{pre})+1 = 1001$ $q_1=1$

q_{pre} = "101" $(4 \times q_{pre})+1 = 10101$ $q_0=0$

q_{pre} = "1010" $(4 \times q_{pre})+1 = 101001$ $q_{-1}=1$

q_{pre} = "10101" $(4 \times q_{pre})+1 = 1010101$ $q_{-2}=1$

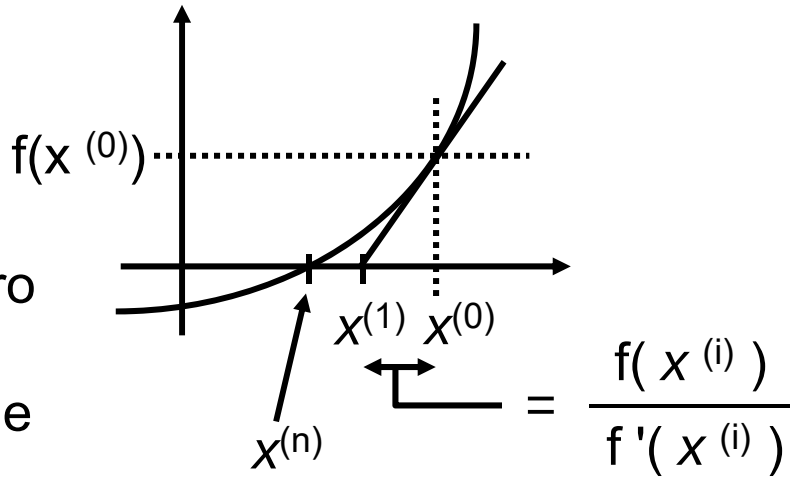
Continuing using trial-and-error algorithm:

1010.11 × 1010.11 = 1110011.1001 (too low, at it should be)
1010.111 × 1010.111 = 1110110.010001 (too low, so $q_{-3}=1$)
1010.1111 × 1010.1111 = 1110110.10100001 (too high, so $q_{-4}=0$)
1010.11101 × 1010.11101 = 1110110.1111001001 (too high, so $q_{-5}=0$)

The Newton-Raphson Method

- The ***Newton-Raphson method*** is a general method for iteratively determining the *root* of a function f(x), that is, the value of x that causes f(x) = 0. In general, a function can have zero or more roots.
- The method involves starting with some initial estimate $x^{(0)}$ of the root, and then iteratively refining the estimate using the recurrence:

$$x^{(i+1)} = x^{(i)} - \frac{f(x^{(i)})}{f'(x^{(i)})}$$



- If f(x) is "well behaved" near the zero and if the initial estimate $x^{(0)}$ is sufficiently close to the root, then the algorithm converges ***quadratically***. That is, the number of accurate bits in $x^{(i)}$ is doubled in every iteration (e.g., 1, 2, 4, 8, etc.).

Reciprocation using Newton-Raphson Iteration

Consider the function $f(x) = x^{-1} - d$

The root of this function occurs when $x = d^{-1}$

Note that $f'(x) = -x^{-2}$.

Thus we obtain the N-R recurrence $x^{(i+1)} = x^{(i)} + [x^{(i)}]^2([x^{(i)}]^{-1} - d)$

But this recurrence simplifies to $x^{(i+1)} = x^{(i)} \times (2 - [d \times x^{(i)}])$

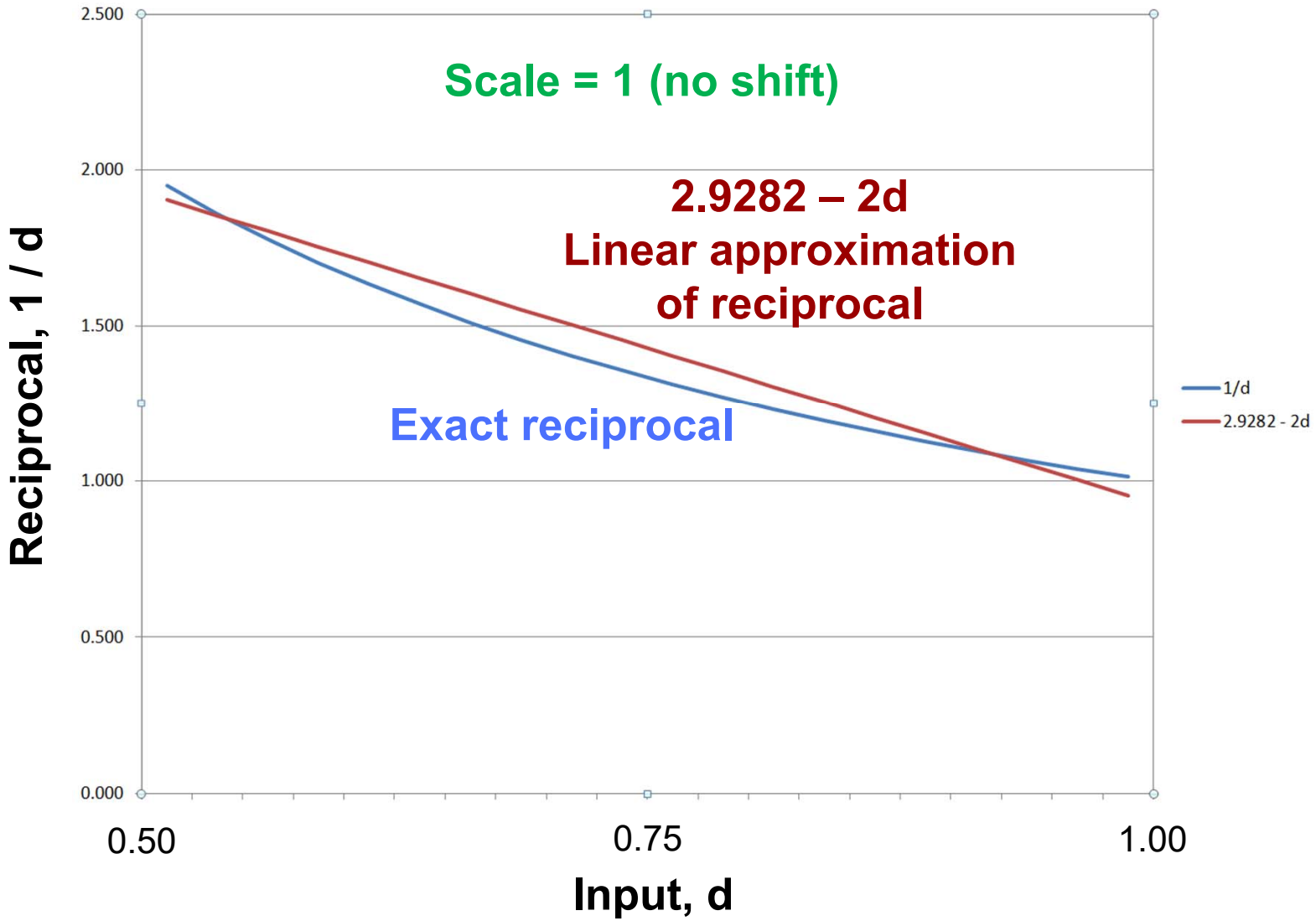
Each iteration requires only one subtraction and two multiplications!

The initial estimate $x^{(0)}$ for d^{-1} can be obtained using a small look-up table.

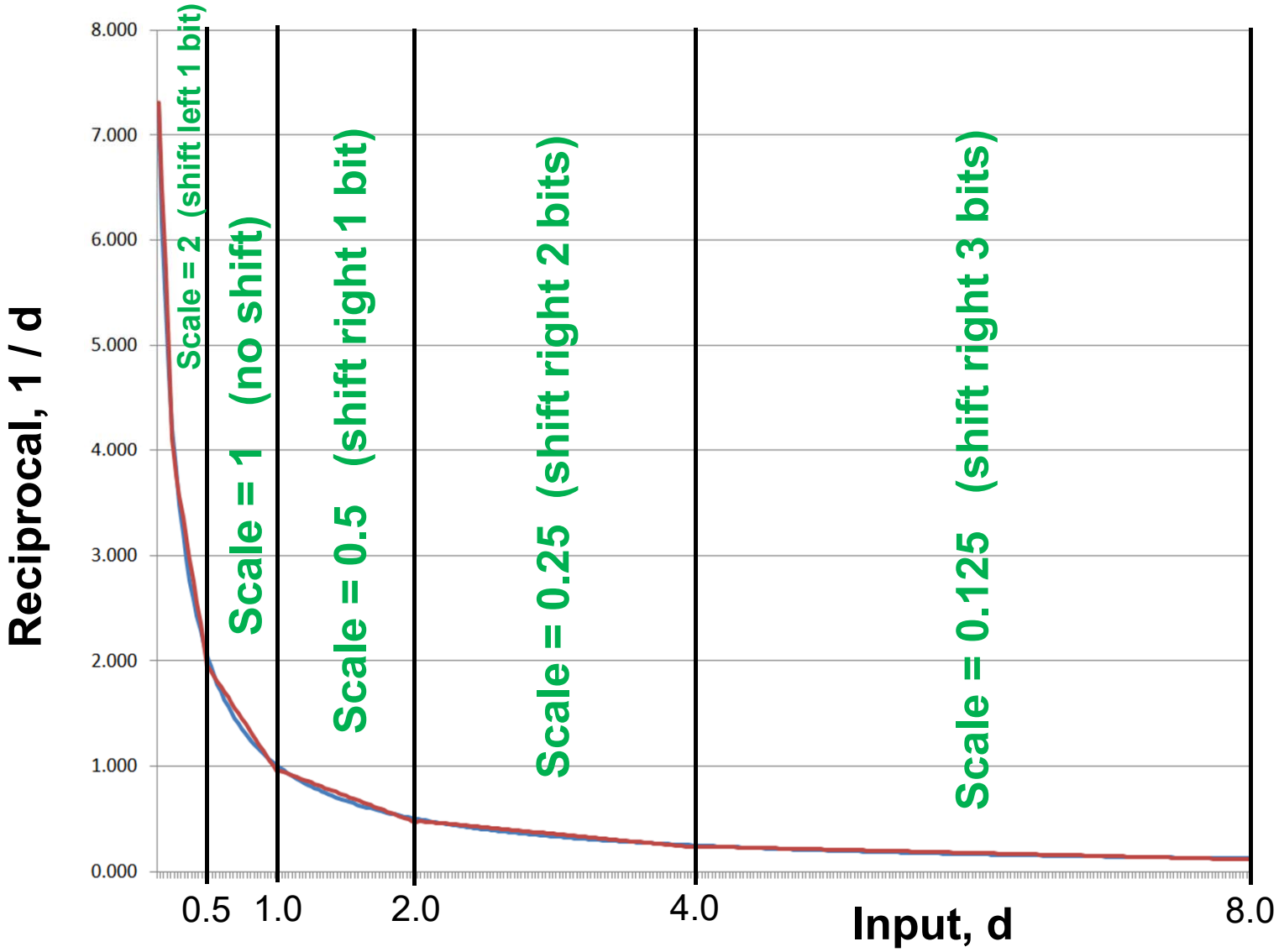
Alternatively, if $0.5 \leq d \leq 1.0$, then $x^{(0)} = 2.9282 - 2d$ is a good estimate.

Note: Any given d can be mapped to a value d' in the domain [0.5,1.0] by shifting **left**(or **right**) some number n of bit places. The desired reciprocal d^{-1} can be obtained by shifting $[d']^{-1}$ **left**(**right**) n places.

Linear Approximation of Reciprocal over (0.5,1.0)



Piecewise Linear Approximation of Reciprocal



Example of the Reciprocal Approximation (in decimal)

Find the reciprocal of d = 3.14159 (It is 0.3183101550488765…)
Scale d by 2^{-2} to map it into the range 0.5 to 1.0
So d' = 3.14159 × 2^{-2} = 0.7853975
(Using a calculator we find that 1/d' = 1.27324062019506 etc.)

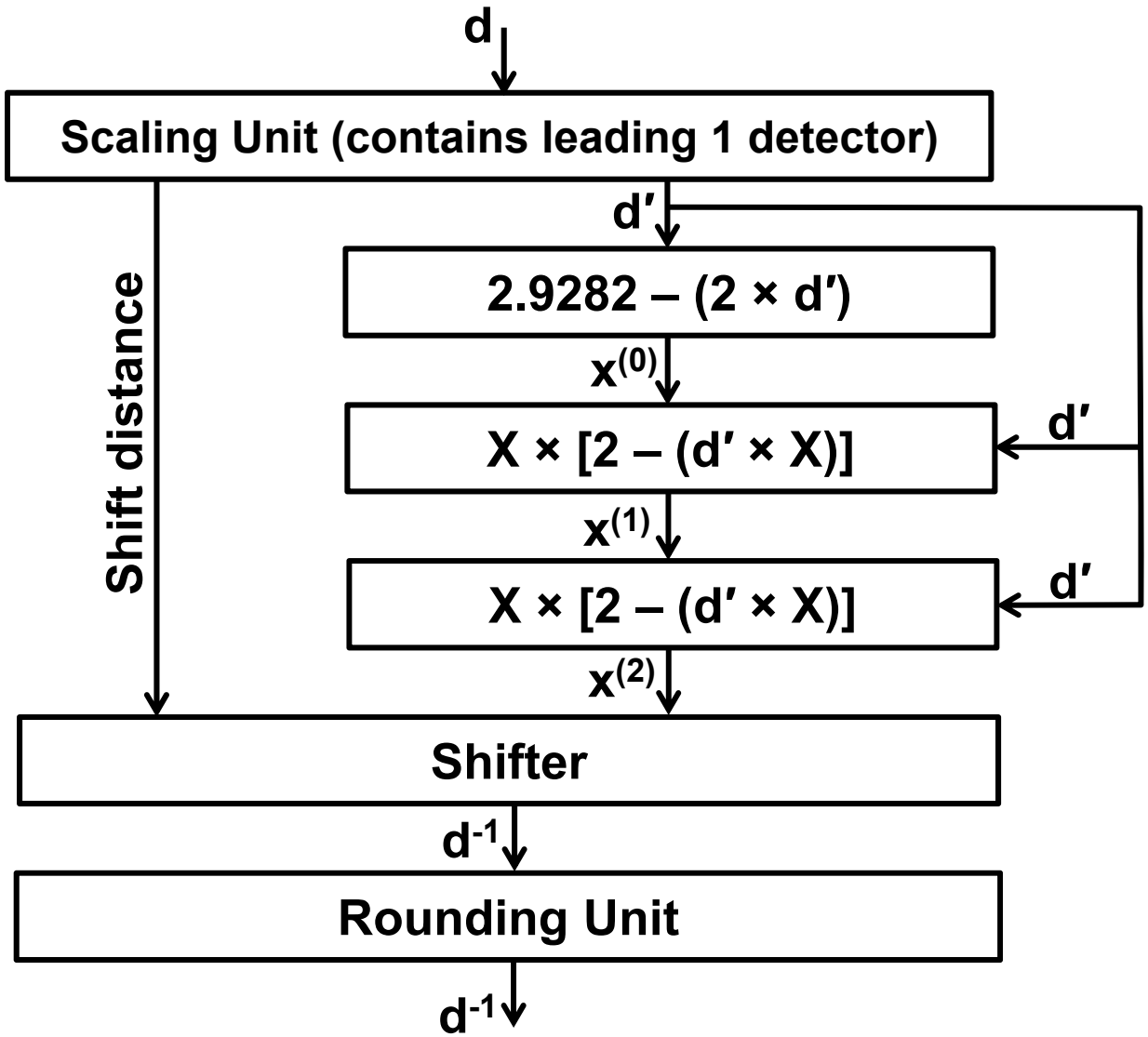
The first estimate $x^{(0)}$ of 1/d' is 2.9282 - 2d' = **1**.357405 (6.61% too high)

The second estimate $x^{(1)}$ of 1/d' is $x^{(0)}$ × [2 – [d′ × $x^{(0)}$]] =
(1.357405)[2 – (0.7853975 × 1.357405)]
= **1.2**676771448276 (0.437% too low) (2 accurate digits)

The third estimate $x^{(2)}$ of 1/d' is $x^{(1)}$ × [2 – [d' × $x^{(1)}$]] =
(1.2676771448276)[2 – (0.7853975 × 1.2676771448276)]
= **1.2732**16310369321 (0.00191% too low) (5 accurate digits)

The corresponding estimate of 1/d is 2^{-2} × 1.27321631036932 1
= **0.3183**040775923303 (0.00191% too low)

Architecture of a Possible Reciprocal Datapath



Big-Oh Complexity Notation for Algorithms

- Computer scientists have developed a ***time complexity*** notation that is useful for making statements concerning the growth in the ***running time*** of an algorithm expressed in terms of the ***size of the problem*** and the required ***number of fundamental operations***.
- Let n be some measure of the size of the problem and let f(n) be a "simple" function of n. An algorithm is said to have complexity O(f(n)) if there exists some positive constant k such that the number C of fundamental operations obeys the ***upper bound*** constraint $C \leq k\ f(n)$.
- For example, in the case of sorting algorithms, the problem size n is the number of data elements to be sorted and the basic operation is the comparison of two elements. The *bubble-sort* algorithm then has complexity $O(n^2)$ while the *heapsort* algorithm has complexity $O(n\log_2 n)$. In the average case, the *quicksort* algorithm has average complexity $O(n\log_2 n)$; however, in the worst case (if the elements are pre-sorted in the wrong order), the complexity is $O(n^2)$.

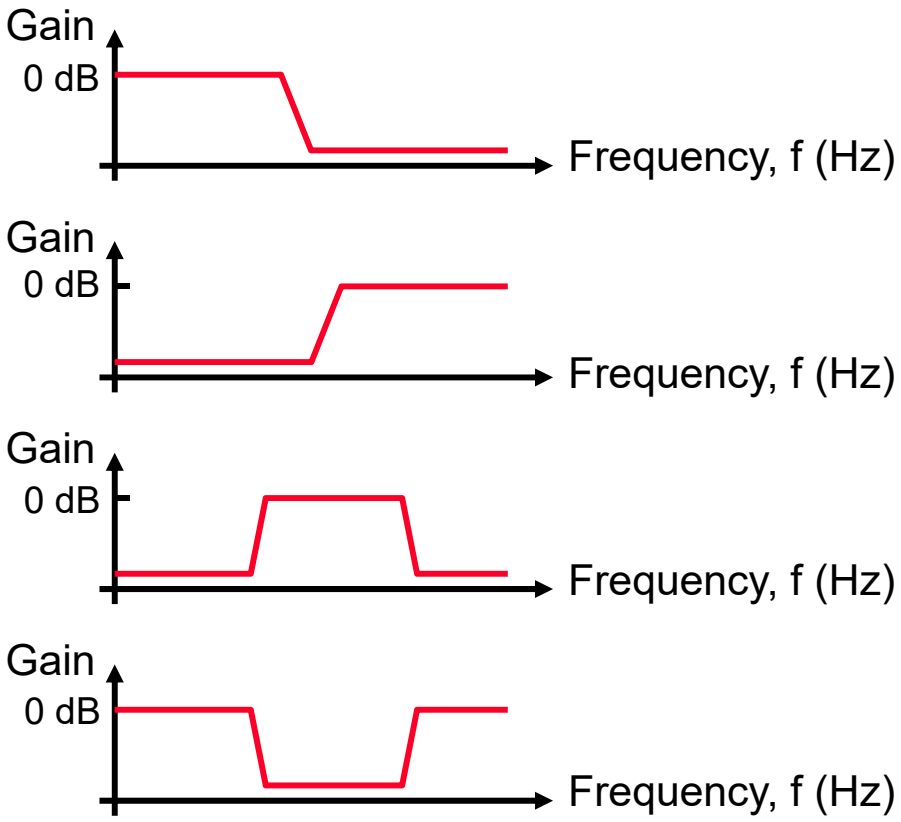
Big-Oh Complexity Notation for Circuits

- Big-Oh notation can be extended to hardware circuits to express both the ***worst-case delay*** through a circuit and a circuit's ***hardware cost***.

- In the case of worst-case delay, the fundamental unit is the gate delay τ_g. The ripple-carry adder/subtractor and the linear comparator all have linear delays of order O(n). Cascaded carry look-ahead adders also have linear delay O(n), but the constant k is smaller. A fast adder with a fully tree-structured carry look-ahead generator and a tree-structured comparator both have delays of order O($\log_2 n$). A purely combinational multiplier with RCAs has a linear delay of order O(n), but using faster adders can reduce the delay to O(($\log_2 n$)^2).
- In the case of hardware cost, the fundamental unit could be a single gate (e.g., 7-input NAND/OR), a configurable logic block (for FPGAs), some repeated cluster of gates (e.g., full adder circuit), and/or the number of memory bits. For example, the ripple-carry adder/subtractor has hardware cost O(n) and the purely combinational multiplier has cost O(n^2).

Digital Filters

- Digital filters operate on sampled, discrete-time signals to perform many useful signal processing operations.
- Common filtering operations include:
  - *low-pass filter* (LPF)
  - *high-pass filter* (HPF)
  - *band-pass filter* (BPF)
  - *band-stop filter*



Specification of Digital Filters

- Digital filters are usually specified as a sequence of frequency bands with required gains or attenuations. For example:
  - *stop bands* with minimum attenuation (given in dB) between given upper and lower frequencies.
  - *pass bands* with specified gain (typically 0 dB gain) with maximum specified *ripple* (i.e., variation in gain, given in dB) about a nominal pass gain between given upper and lower frequencies
  - *transition bands* between given stop and pass bands.
- Standard algorithms (e.g., functions in the Filter Design Toolbox in Matlab) can be used to generate *linear transfer functions* in the Z-transform frequency domain that satisfy given filter specifications.
- The coefficients of the transfer functions can be used to implement digital filters that involve binary multiplications and additions on an incoming discrete-time stream of digitized samples.

The Linear Constant-Coefficient Difference Eqn.

- The behaviour of a linear, time-invariant digital filter is usually given in the Z-transform domain as a *rational transfer function* as follows:

$$H(z) = \frac{B(z)}{A(z)} = \frac{\sum_{k=0}^{M} b_k z^{-k}}{1 + \sum_{k=1}^{N} a_k z^{-k}}$$

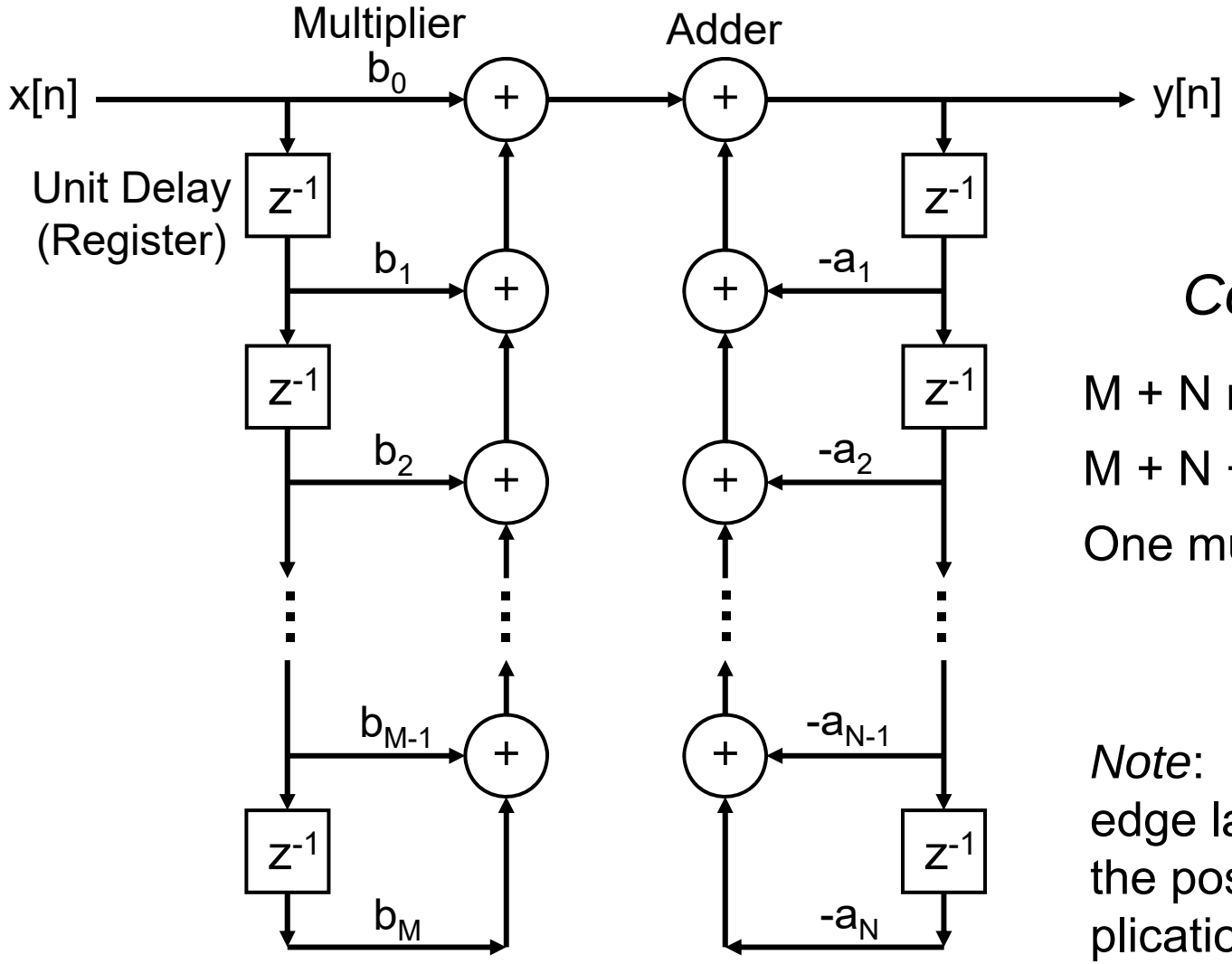
- In the Z-transform domain we have Y(z) = X(z) H(z), where X(z) and Y(z) are the Z-transforms of the sequence of the discrete-time filter inputs x(n) and filter outputs y(n), respectively.
- By taking the inverse Z-transform of H(z) we obtain the *linear constant-coefficient difference equation*, which can be used to implement the filter as multiplications and additions in discrete time.

$$y[n] = \sum_{k=0}^{M} b_k x[n-k] - \sum_{k=1}^{N} a_k y[n-k]$$

Considerations in Digital Filter Implementation

- *Computational complexity*: How many additions, multiplications, divisions, table look-ups, etc. are required per output sample?
- *Memory requirements*: How many of bits of memory storage are required during filter operation?
- *Finite wordlength effects*: If too few bits of precision are used to represent numbers, then the filter's transfer function may be inaccurate with respect to the intended specification. If too many bits of precision are used, then unnecessary hardware is present.
- *Filter stability*: Certain filter structures (e.g., recursive infinite impulse response filters) may be numerically unstable leading to very inaccurate filter output.
- *Suitability for parallel processing*: Filter structures tend to have a regular structure that can be readily mapped to parallel hardware to increase the throughput and/or allow the operating voltage to be reduced to reduce the power consumption.
- *Ability to pipeline*: Similar issues as in parallelizability.

Direct Form I Digital Filter Structure



Complexity

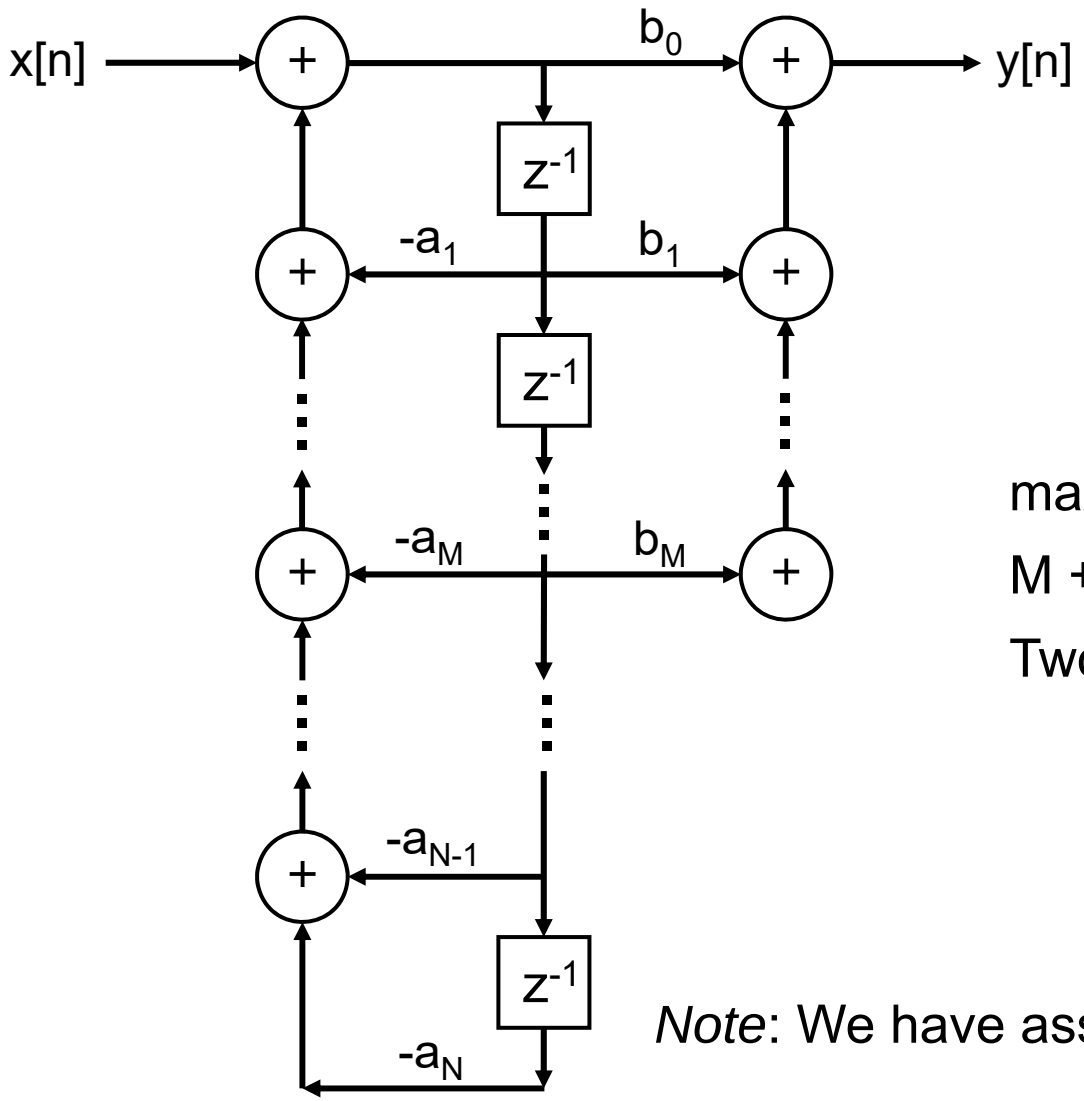
M + N registers

M + N + 1 multiplications

One multi-input addition

Note: The "a" and "b" edge labels indicate the positions of multiplication operations.

Direct Form II Digital Filter Structure



Complexity

max{ M, N } registers

M + N + 1 multiplications

Two multi-input additions

Note: We have assumed here that N>M.

Infinite Impulse Response (IIR) Filters

- The *impulse response* of a filter is the output that is produced when a *unit impulse* $\delta[n]$ is applied to the input. The function $\delta[n]$ has value 1 when $n = 0$ and has value 0 when $n \neq 0$.
- The Direct Form I and II digital filters are called *Infinite Impulse Response* (IIR) filters because if an impulse function is applied to the input x[n] of an IIR, the signal at the filter output y[n] can take an arbitrarily long time to stabilize at a constant zero.
- The finite precision of the filter coefficients, input samples and intermediate results can lead to instability in IIR filters. For this reason, IIR filters are usually implemented as a cascade of order-2 IIR stages.
- Note: The *order of a filter* is the maximum of M and N.
- Order-2 IIR filters (also known as *biquads*) can be more easily designed to be stable, so general IIR filters are usually designed as a cascade of smaller biquad filter stages.

Weighted Moving Average Filter

- If $a_k = 0$ for k = 1, . . . , N then the transfer function has a denominator of 1, which makes the filter inherently stable (no "poles"; only "zeros").

$$H(z) = \frac{B(z)}{1} = \sum_{k=0}^{M} b_k z^{-k}$$

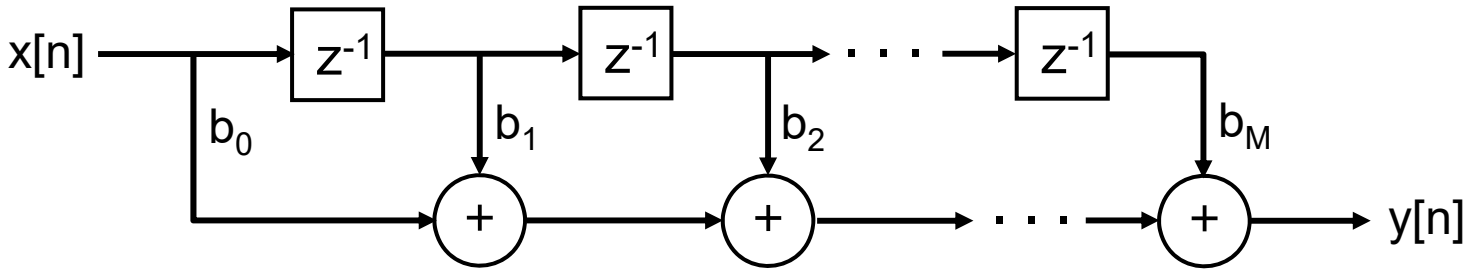
- The corresponding linear constant-coefficient difference equation is a filter whose output y[n] is the ***weighted average*** of the M + 1 most recent input samples.

$$y[n] = \sum_{k=0}^{M} b_k x[n-k]$$

- The impulse response of the weighted moving average filter is the finite-length sequence b[0], b[1], . . . , b[M]. Thus these filters are commonly called ***Finite Impulse Response*** (FIR) filters.
- The inherent stability of FIR filters makes them a common choice in implementations, but the stability comes at the cost of much more hardware compared to an IIR with the same filter specifications.

Direct Form FIR Filter Architecture

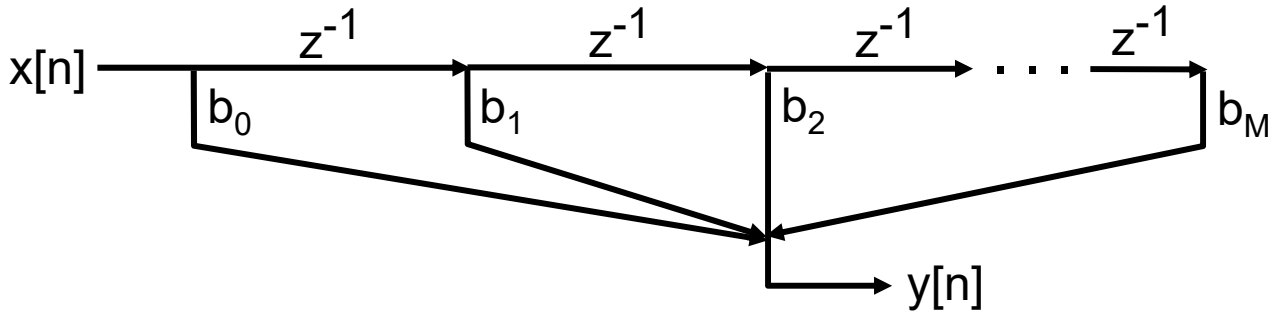
- Also called the "Transversal" or "Tapped Delay Line" FIR architecture.
- A straightforward implementation of the FIR difference equation.



- *Arithmetic Complexity*: M+1 multiplications and M additions per sample
- *Hardware Complexity*: M registers, M+1 multipliers, M adders
- *Latency*: Output y[n] appears within one clock cycle of input x[n]
- *Critical Path*: One multiplier delay + M adder delays. The number of adder delays could be reduced to $\lceil \log_2 M \rceil$ with a tree-structured adder.

Transposition Theorem

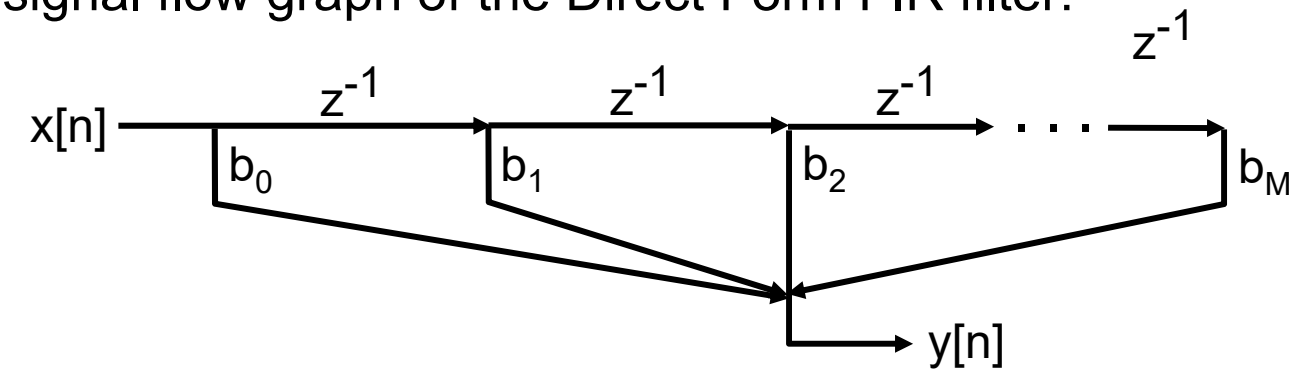
- Also called the "Flow Graph Reversal Theorem"
- A *signal flow graph* is obtained from a filter block diagram by replacing all adders with simple nodes and replacing all delay elements with edge labels.
- For example, the Direct Form FIR filter has this signal flow graph:



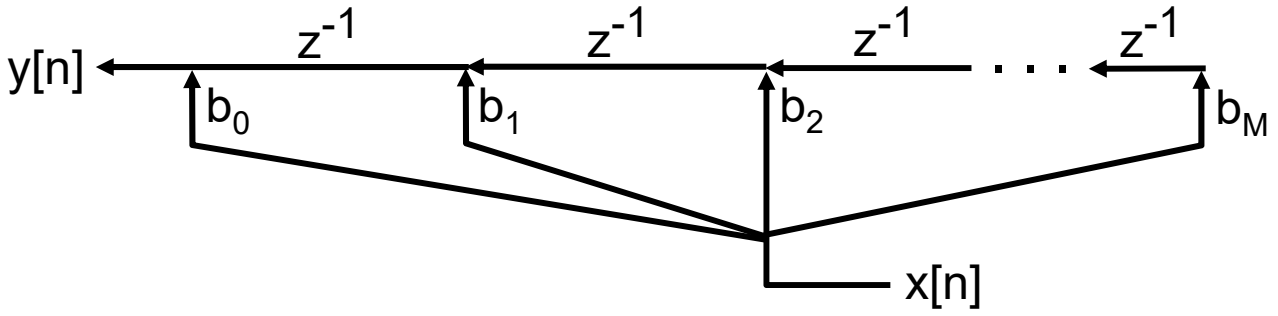
- The *Transposition Theorem* states that if we take a signal flow graph and (1) reverse the direction of all arrows and (2) interchange the input and output, the resulting *transposed flow graph* implements the same function as the original flow graph. Note: Nodes with incoming flows will turn into adders in the transformed filter structure.

Example of the Transposition Theorem

- The signal flow graph of the Direct Form FIR filter:

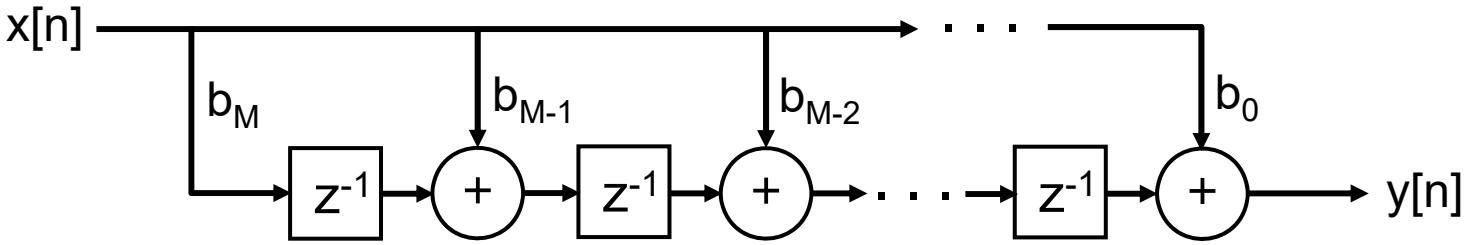


- The corresponding transposed signal flow graph is:



Transposed Direct Form FIR Filter Architecture

- Also called the **"Data Broadcast" FIR architecture**.
- Mathematically equivalent to the Direct Form (ignoring numerical errors) because of the Transposition Theorem.



- *Arithmetic Complexity*: M+1 multiplications and M additions per sample
- *Hardware Complexity*: M registers, M+1 multipliers, M (wider) adders
- *Latency*: Output y[n] appears within one clock cycle of x[n]
- *Critical Path*: One multiplier delay + one adder delay. This can be significantly faster than the Direct Form FIR filter architecture.