# Computer Arithmetic

By,

Ajinkya Karande

Adarsh Yoga

# Introduction

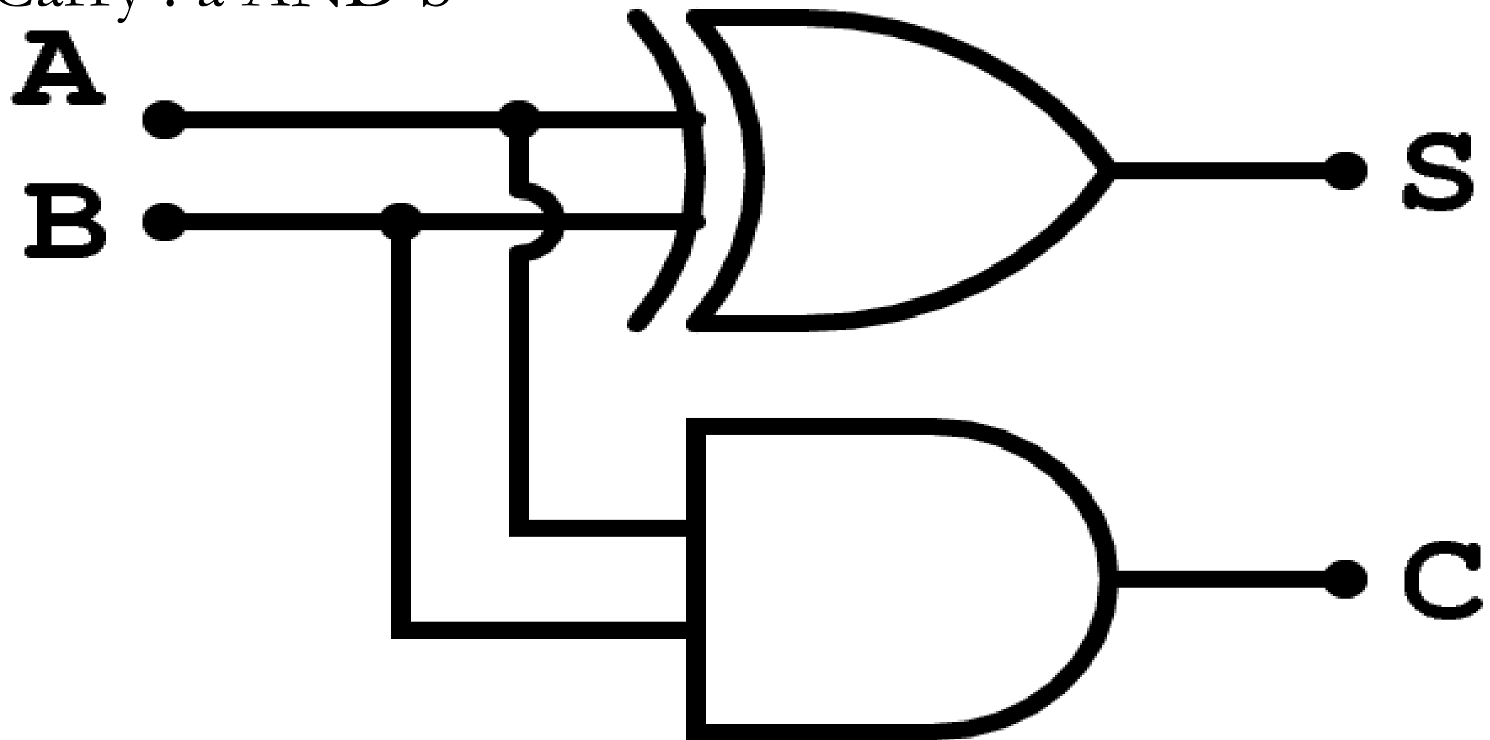
- Early computer designers believed saving computer time and memory were more important than programmer time.
- Bug in the divide algorithm used in Intel chips.
- Basics of Integer/Floating point Addition, Subtraction, Multiplication and Division algorithms
- Refinements and variations on these algorithms

# Integer Arithematic

Half Adder

Sum : a XOR b

Carry : a AND b

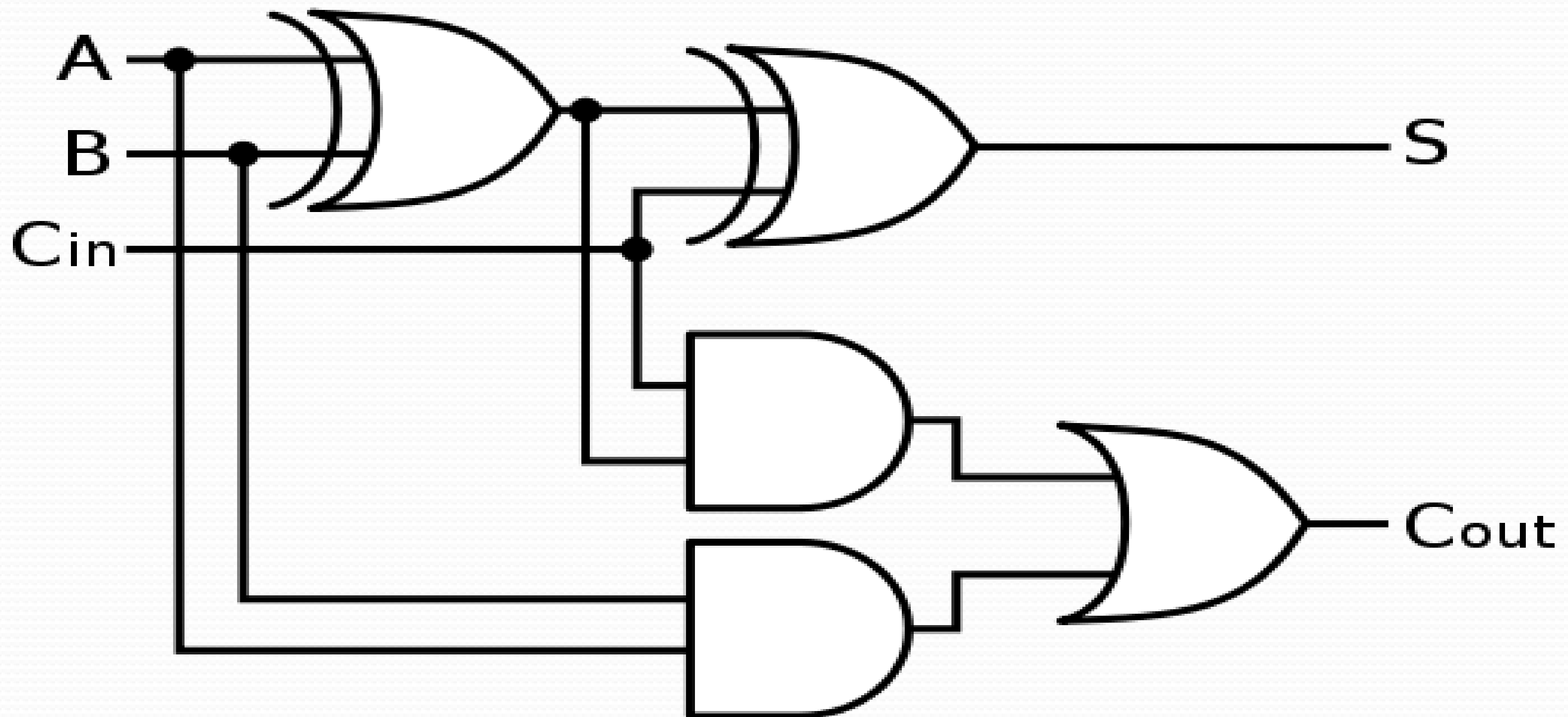


# Integer Arithematic

Full Adder

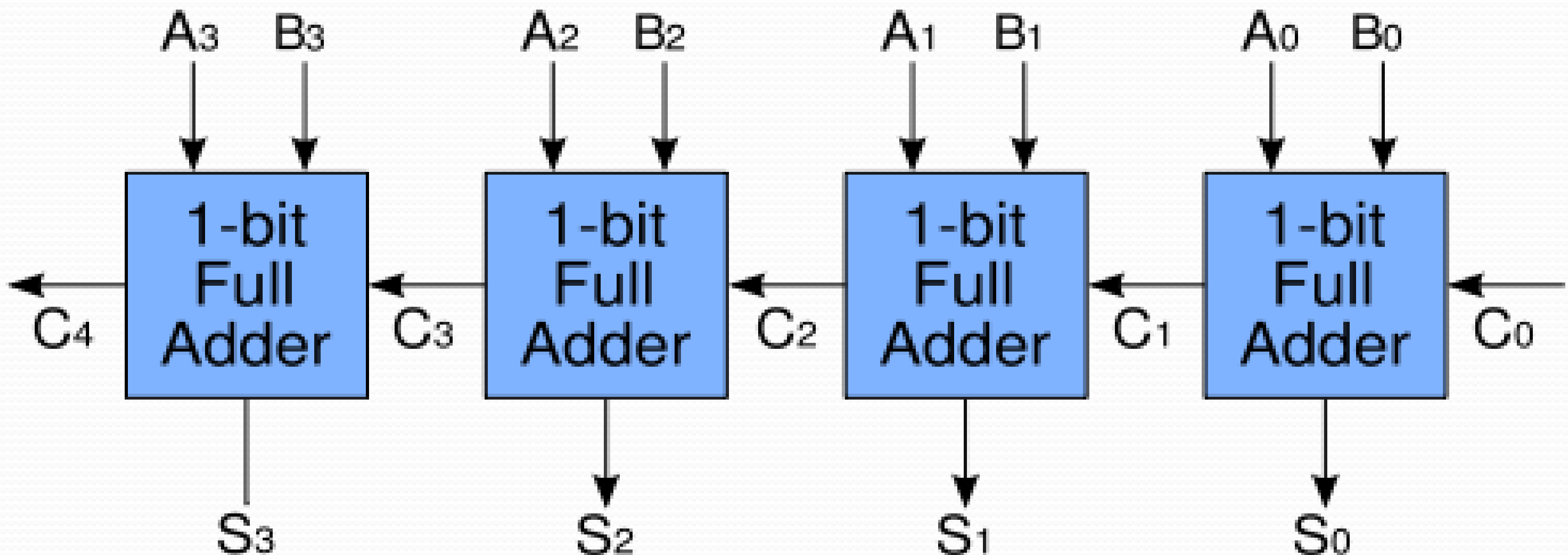
Sum : a¬b¬c + ¬ab¬c +a¬bc + abc

Carry : (a XOR b)c + ab

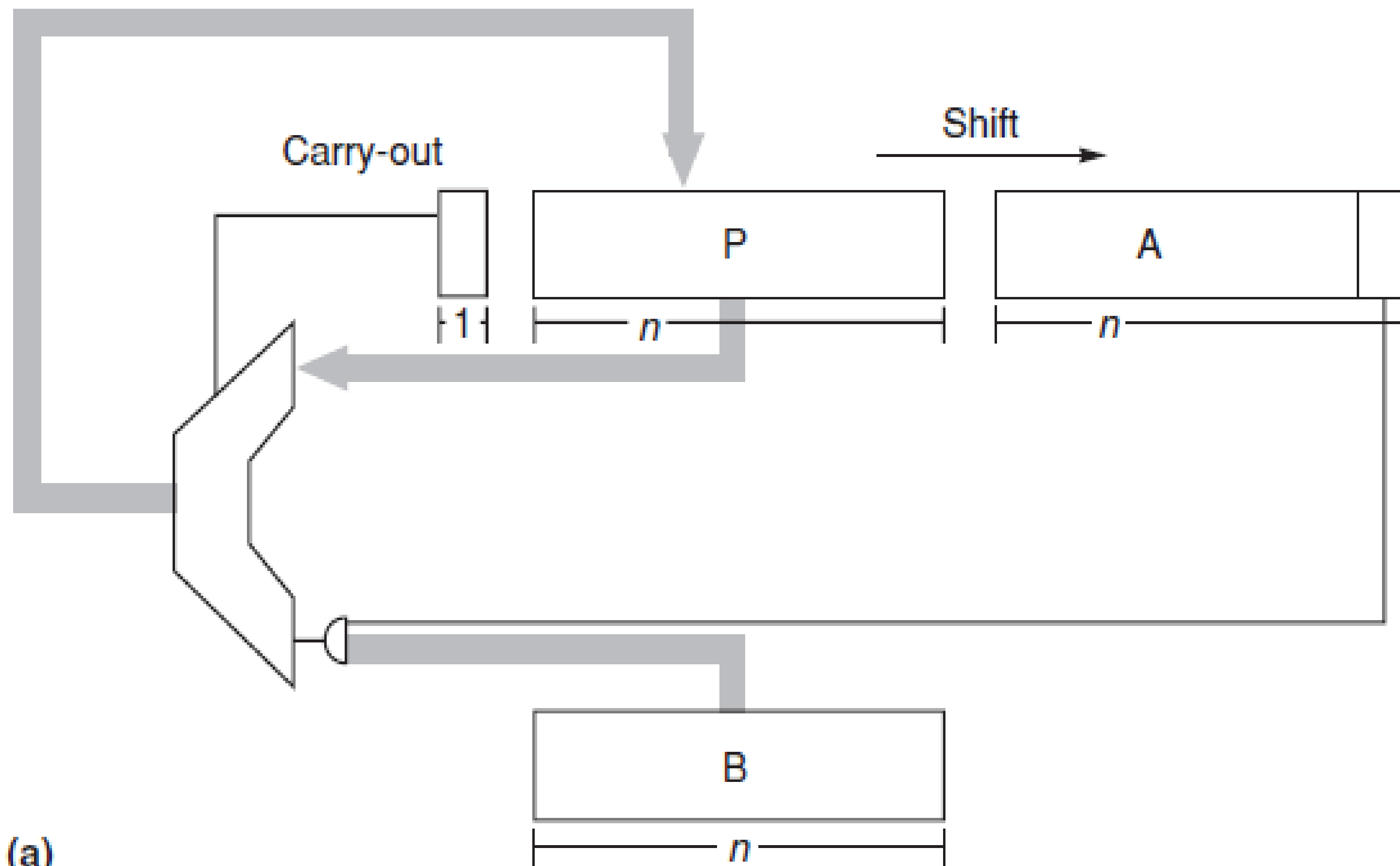


# Integer Arithematic

Ripple Carry Adder



# Integer Multiplication



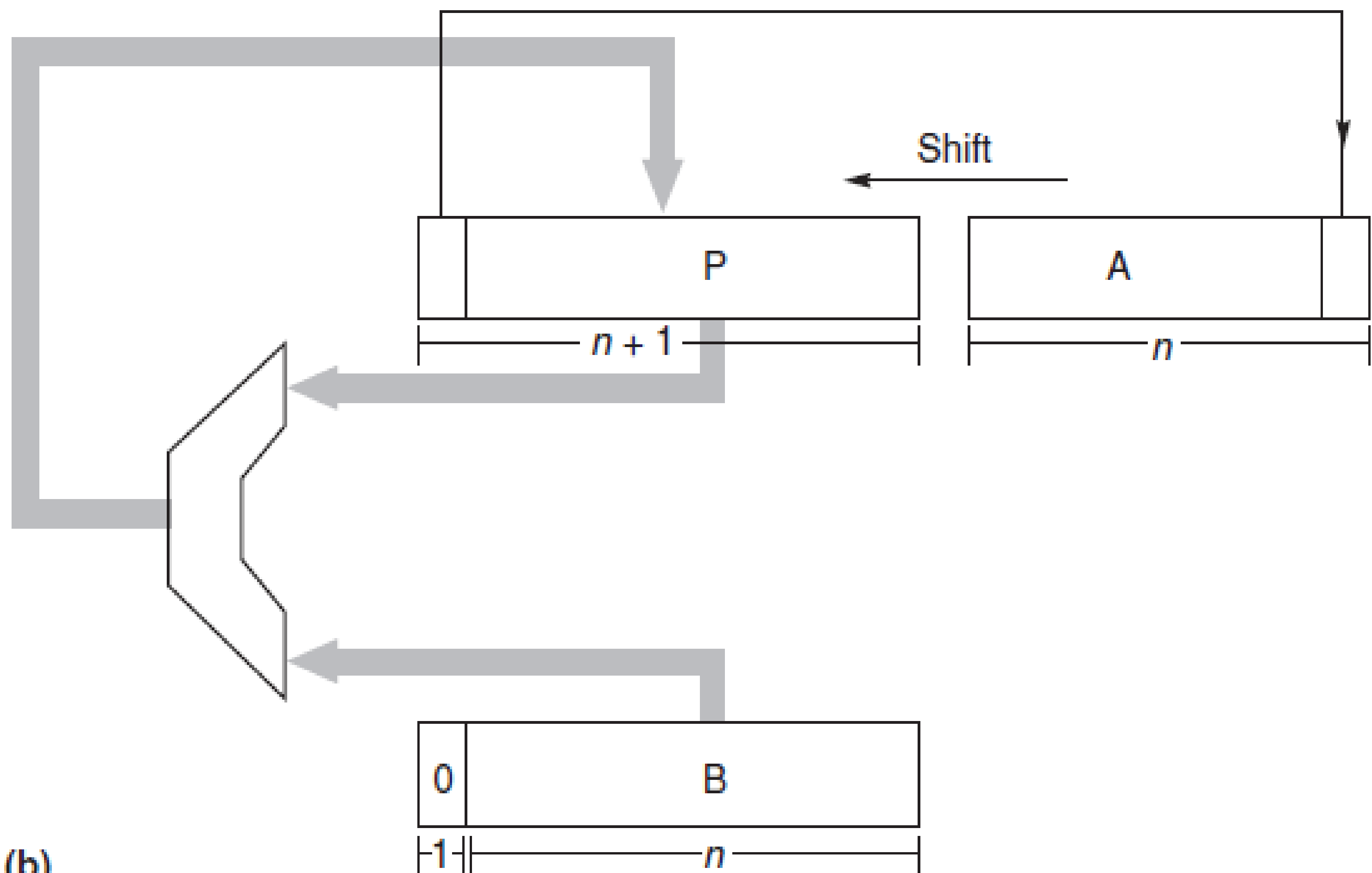
# Integer Multiplication

Unsigned Multiplication:

1. LSB of A is 1, add B to P

2. Shift registers P and A right with carry-out of the sum being moved into the high-order bit of P, the low-order bit of P being moved in register A.

# Integer Division



# Integer Division

Restoring Division

1. Shift register pair (P,A) one bit left.
2. Subtract content of B from P
3. If result in step 2. is negative, set Ao to zero else to 1
4. If result in step 2. is negative, restore the old value of P by adding the contents of B back in P

# Integer Division

Non – Restoring Division

If P is negative

1.a Shift (P,A) pair one bit left

2.a Add the contents of register B to P

Else,

1.b Shift (P,A) pair one bit left

2.b Subtract the contents of register B from P

3. If P is –ve, set the low order bit of A to zero else set it to one.

# Signed Numbers

Four ways to represent signed numbers:

- Sign Magnitude
- One’s Complement
- Biased
- Two’s Complement
    - Formula for Two’s Complement of a number:

$$-a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \ldots + a_1 2^1 + a_0$$

# Booth Recoding

- Examines the bits of a from the LSB to the MSB
- For example, if a=7=0111 then the step 1 of integer multiplication will successively add B, add B, add B and add 0.
- Booth recoding "recodes" 7 as 8-1=1000-0001=$100\bar{1}$
- Works equally well for positive and negative numbers

# Booth Recoding

If the initial content of register A is $a_{n-1} \ldots a_0$ then the multiplication algorithm for signed numbers,

1. If $A_i = 0$ and $A_{i-1} = 0$ add 0 to P
2. If $A_i = 0$ and $A_{i-1} = 1$ add B to P
3. If $A_i = 1$ and $A_{i-1} = 0$ subtract B from P
4. If $A_i = 1$ and $A_{i-1} = 1$ add 0 to P

Works because

$$\sum_{i=0}^{n-1} b(a_{i-1} - a_i)2^i = b(-a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \ldots + a_1 2 + a_0) + ba_{-1}$$

# System Issues

Overflow:

- Three approaches to detect overflows in signed arithmetic,
  - Set bit on overflow.
  - Trap on overflow.
  - Do nothing.
- No traps unsigned arithmetic since they are primarily used in manipulating addresses.

# System Issues

- Should the result of multiplication be a 2n bit result? (or just return the lower order n bits?)
- Against: Virtually in all high level languages, the result is of the same type
- For: Can be used to substantially speed up the multiplication.

# System Issues

Three possible system-level approaches to speeding up multiplication

- Single-cycle multiply step.
- Do integer multiplication in FPU.
- Have an autonomous unit in the CPU do the multiplication.

# System Issues

Division and remainder for negative numbers:

- Built-in hardware instructions should correspond to what high-level languages specify.
- No agreement among existing programming languages.
- Eg: -5 MOD 3, -5 DIV 3.

# Floating Point Arithmetic

- IEEE standard for FPA specifies single precision (32 bits) and double precision (64 bits) floating point numbers.
- Consists of an exponent and a significand.
- If e is the biased exponent and f is the value of the fraction, then the number is $1.f*2^{e-127}$.
- Special values NaN, ∞, -∞.
- Denormal numbers to represent result$<1.0*2^{Emin}$.

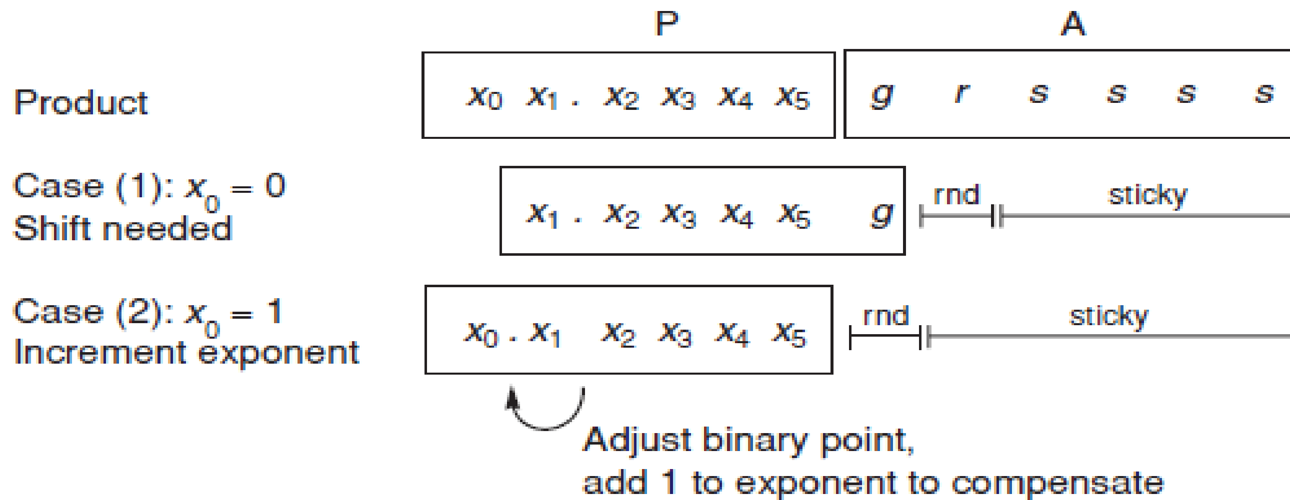
# Floating Point Multiplication

$$(s_1 \times 2^{e1}) \bullet (s_2 \times 2^{e2}) = (s_1 \bullet s_2) \times 2^{e1+e2}$$

FP multiplication consists of three steps

- Multiplies the significands using ordinary integer multiplication.
- Rounds the result to take care of the precision.
- Computes the new exponent.

# Rounding Algorithm




- If MSB of P is 0, shift P left by 1 bit.
- If MSB of P is 1, then set s=s OR r and r=g where r is the round bit, g is the guard bit and s is the sticky bit.

# Rounding Algorithm

| Rounding mode | Sign of result ≥ 0 | Sign of result < 0 |
|---|---|---|
| $-\infty$ | | $+1$ if $r \vee s$ |
| $+\infty$ | $+1$ if $r \vee s$ | |
| 0 | | |
| Nearest | $+1$ if $r \wedge p_0$ or $r \wedge s$ | $+1$ if $r \wedge p_0$ or $r \wedge s$ |

**Figure I.11 Rules for implementing the IEEE rounding modes.** Let $S$ be the magnitude of the preliminary result. Blanks mean that the $p$ most-significant bits of $S$ are the actual result bits. If the condition listed is true, add 1 to the $p$th most-significant bit of $S$. The symbols $r$ and $s$ represent the round and sticky bits, while $p_0$ is the $p$th most-significant bit of $S$.

# Floating Point Addition

- Takes two inputs of p bits and returns a p-bit result
- Ideal algorithm would first perform the addition and then round the result to p bits

# Floating Point Addition

- Consider, two 6 bit numbers 1.10011 and $1.10001 * 2^{-5}$

$$
\begin{array}{r l}
 & 1.10011 \\
+ & .000011 0001 \\
\hline
\end{array}
$$

- Discarding the lower order bits of the second addend, we get

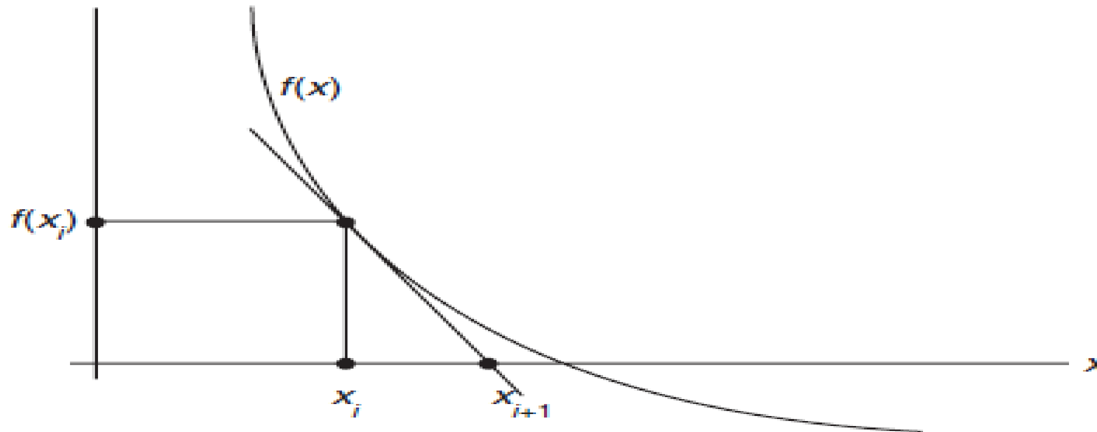
$$
\begin{array}{r l}
 & 1.10011 \\
+ & .00001 \\
\hline
 & 1.10100
\end{array}
$$

# Floating Point Addition

- Designate from the discarded bits, the MSB as g (guard bit), the next MSB as r (round bit) and the logical OR of the remaining bits as s(sticky bit)
- Round the sum using the Rounding Algorithm specified for multiplication.
- In case of subtraction 2's complement taken and the sum is computed as above.

# Floating Point Division

- Approach to compute division converges to the quotient using an iterative technique called Newton's iteration.




- Cast the problem as finding the zero of a function.

# Floating Point Division

- If $x_i$ is the guess at zero,

$$y - f(x_i) = f'(x_i)(x - x_i)$$

- The equation has zero at

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

- By recasting zero as finding zero of a function, $f(x)=x^{-1}-b$. Hence $f'(x)=-1/x^2$

$$x_{i+1} = x_i - \frac{1/x_i - b}{-1/x_i^2} = x_i + x_i - x_i^2 b = x_i(2 - x_i b)$$

# Floating Point Division

Now a/b is computed by

- Scale b in the range $1 \leq b < 2$ and get an approximate value of 1/b.
- Iterate $x_i+1=x_i(2-x_ib)$ until it is accurate enough ($x_n$).
- Compute $ax_n$ and reverse the scaling done.

Iterate until, $|(x_i-1/b)/(1/b)|=2^{-p}$.

# More on Floating Point Arithmetic

Fused Multiply-Add

- Motivated by IBM RS/6000 which has an instruction that computes ab+c the fused multiply-add.
- Used to implement efficient floating-point multiple precision packages.

# Exceptions

Five types

1. Underflow
2. Overflow
3. Divide by zero
4. Inexact
5. Invalid

# Exceptions

Underflow:

- Underflow occurs when there is loss of accuracy
- It is set whenever the delivered result is different from what would be delivered in a system with the same fraction size.

Overflow:

- Occurs when the result of an arithmetic operation is too large to hold.

# Exceptions

Divide by Zero:

- Occurs when the divisor is zero

Inexact:

- If there was a result that couldn't be represented exactly and had to be rounded, inexact is signaled.

Invalid:

- If the arithmetic operation results in NAN

# Speeding Up Integer Addition

Carry-Lookahead

The ith sum will be

$$s_i = a_i \overline{b}_i \overline{c}_i + \overline{a}_i b_i \overline{c}_i + \overline{a}_i \overline{b}_i c_i + a_i b_i c_i$$

Rewriting $c_i$ in terms of $a_i$ and $b_i$

$$c_i = g_{i-1} + p_{i-1} c_{i-1}, \quad g_{i-1} = a_{i-1} b_{i-1}, \quad p_{i-1} = a_{i-1} + b_{i-1}$$

$$c_i = g_{i-1} + p_{i-1} g_{i-2} + p_{i-1} p_{i-2} g_{i-3} + \cdots + p_{i-1} p_{i-2} \cdots p_1 g_0 + p_{i-1} p_{i-2} \cdots p_1 p_0 c_0$$

# Speeding Up Integer Addition

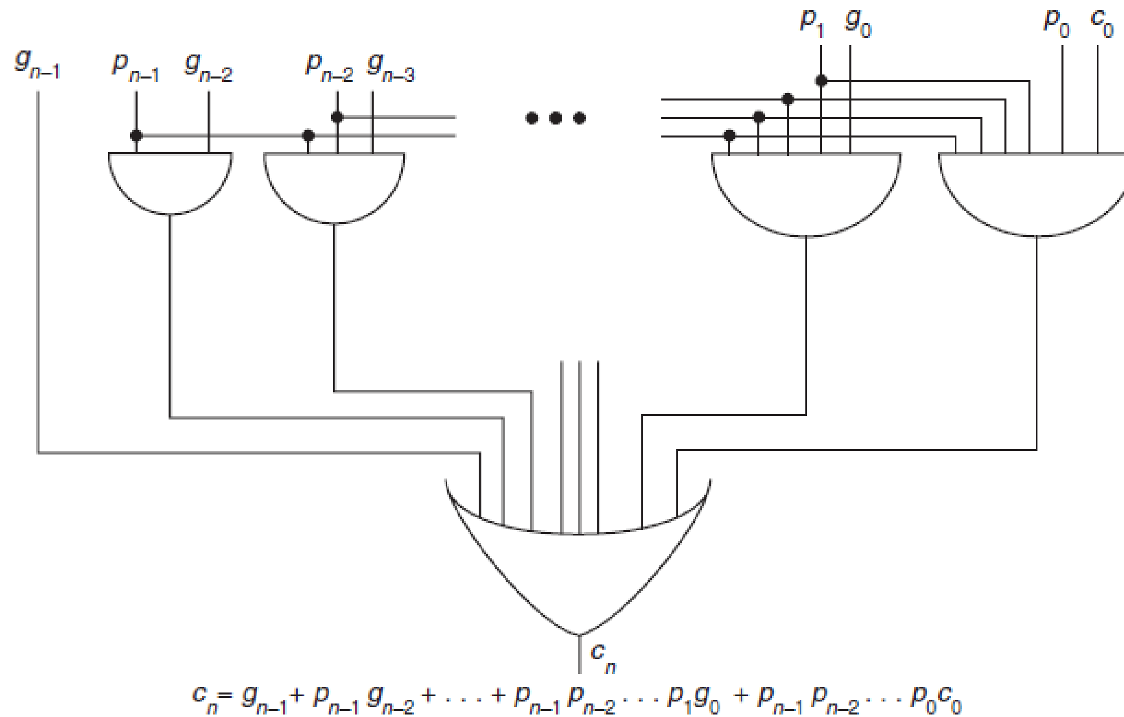



**Figure I.14** Pure carry-lookahead circuit for computing the carry-out $c_n$ of an $n$-bit adder.

# Speeding Up Integer Addition

It takes one logic level to form p and g, two levels to form the carries and two for the sum in all five logic levels.

Carry Lookahead idea is used to build an adder that has $\log_2 n$ logic levels.

$c_1 = g_0 + c_0 p_0$

$c_2 = G_{01} + c_0 P_{01}$

$G_{01} = g_1 + p_1 g_0$

$P_{01} = p_1 p_0$

# Speed Up Integer Addition

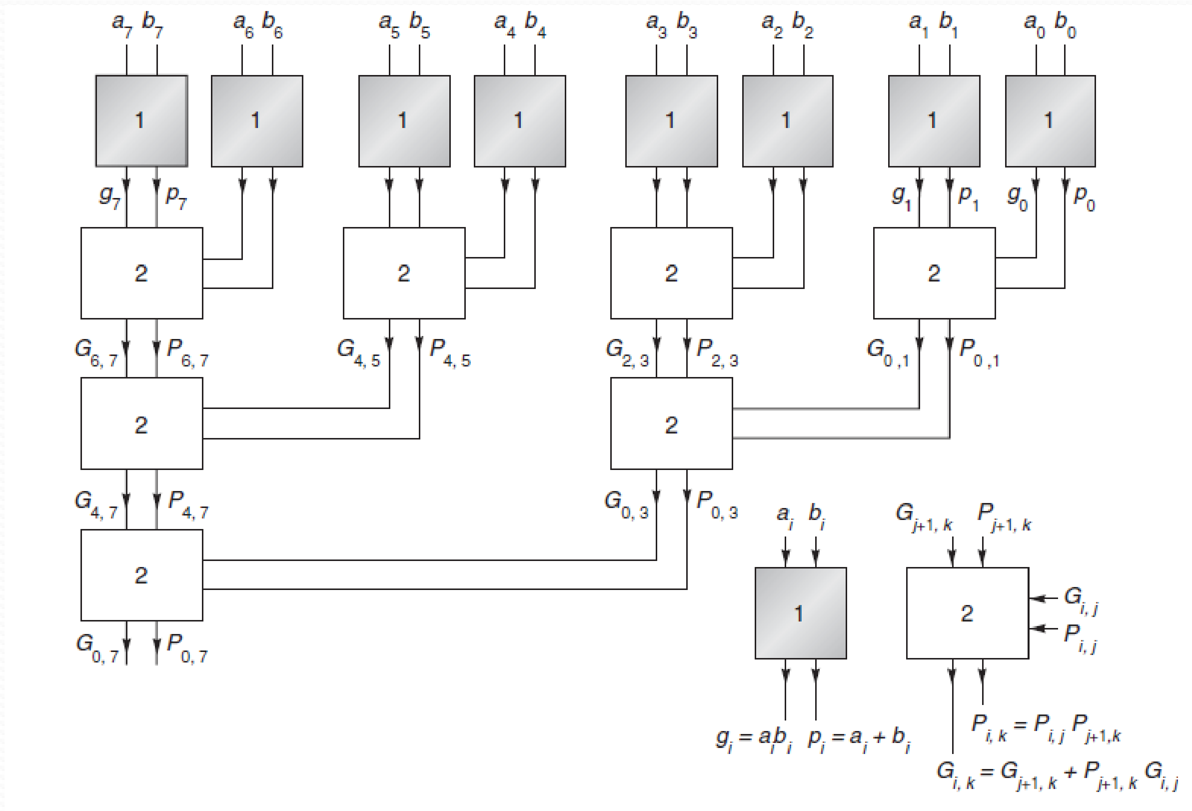



**Figure I.15 First part of carry-lookahead tree.** As signals flow from the top to the bottom, various values of $P$ and $G$ are computed.

# Speed Up Integer Addition

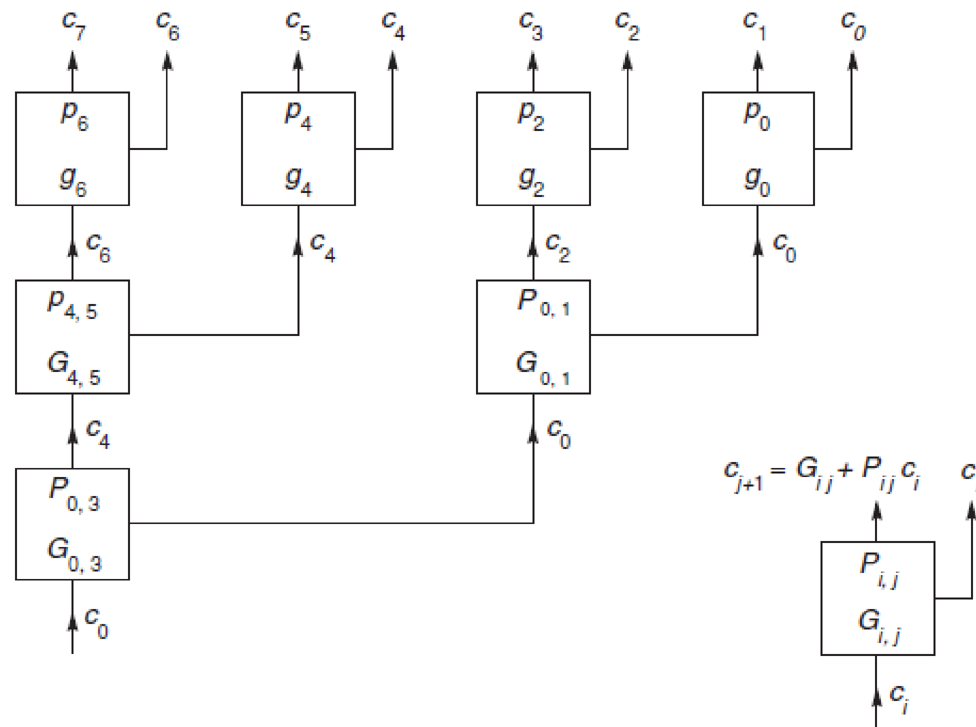



**Figure I.16 Second part of carry-lookahead tree.** Signals flow from the bottom to the top, combining with *P* and *G* to form the carries.

# Speed Up Integer Addition

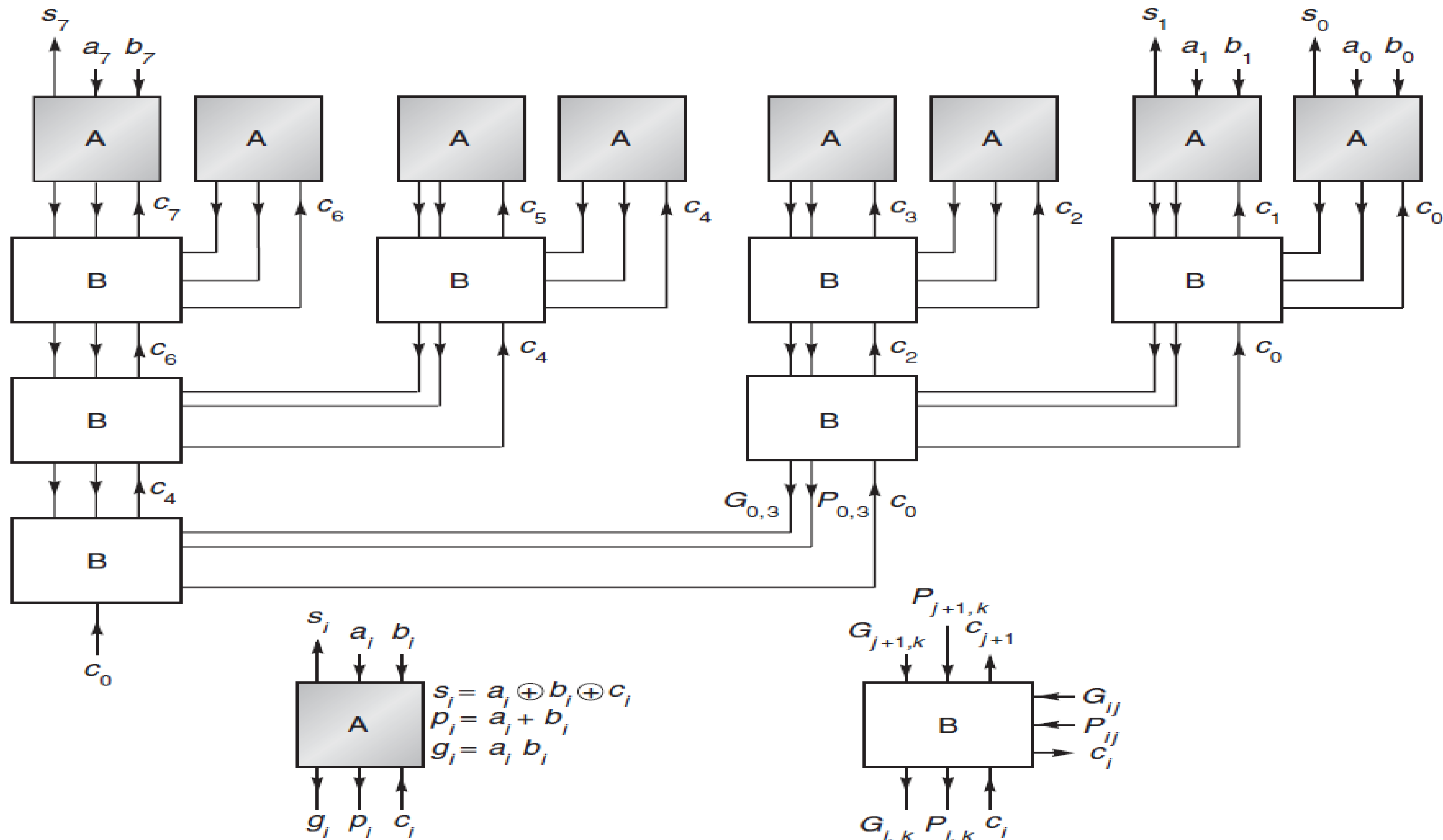



**Figure I.17 Complete carry-lookahead tree adder.** This is the combination of Figures I.15 and I.16. The numbers to be added enter at the top, flow to the bottom to combine with $c_0$, and then flow back up to compute the sum bits.

# Speed Up Integer Addition

Carry Skip Adder

- If any block generates carry, the carry-out of a becomes true
- The carry-out from each least-significant block is fed to AND gate with the P signal
- If carry-out and P signals both are true, then carry skips the block
- Practical only if carry-in signals are cleared at the start of each operations

# Speed Up Integer Addition

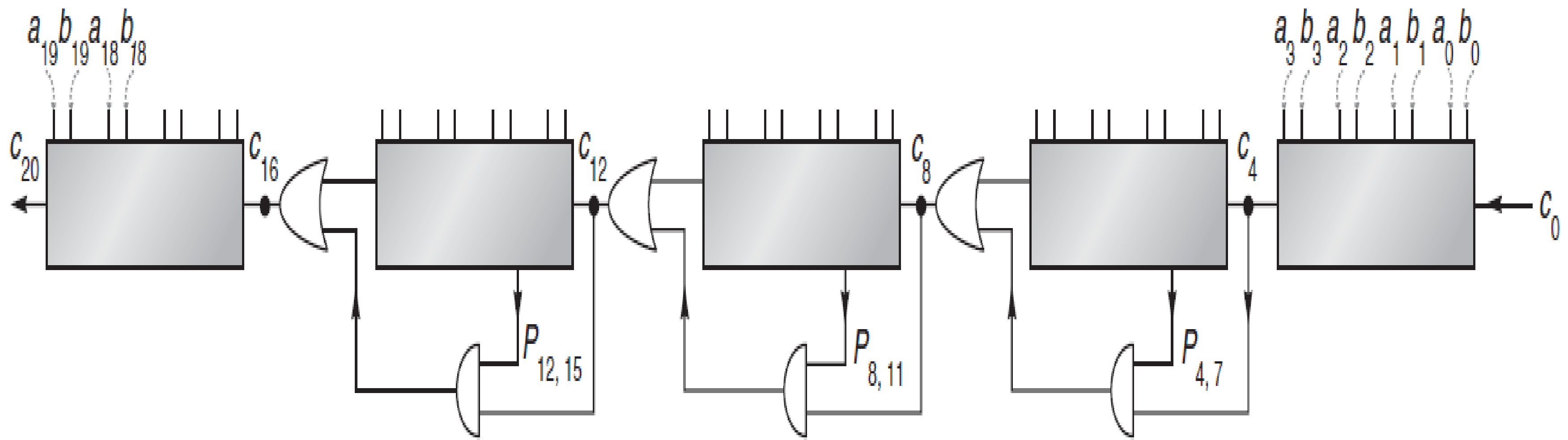



**Figure I.18 Carry-skip adder.** This is a 20-bit carry-skip adder ($n = 20$) with each block 4-bits wide ($k = 4$).

# Speed Up Integer Addition

Carry-Select Adder

- Performs two additions in parallel, with carry-in as zero and carry-in as one
- On determining final carry-in correct sum is selected

# Speeding Up Integer Addition

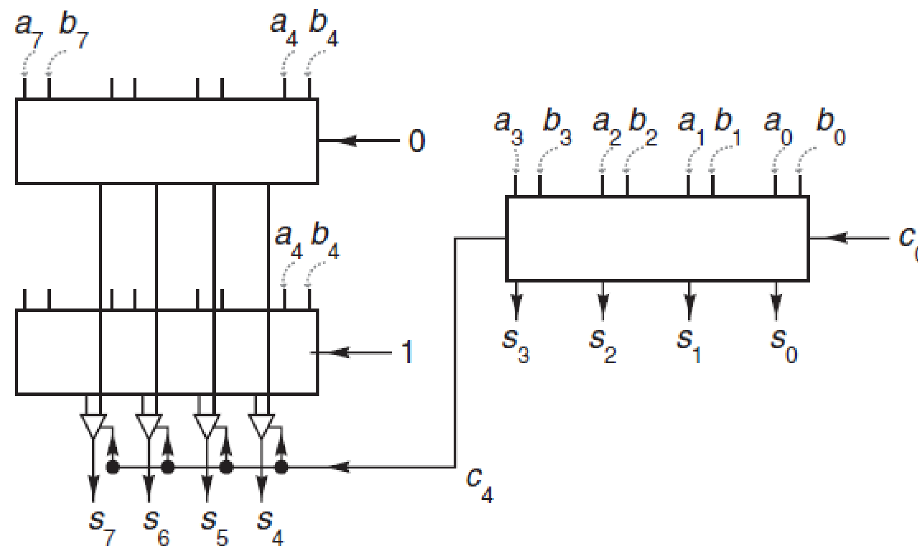



**Figure I.20 Simple carry-select adder.** At the same time that the sum of the low-order 4 bits is being computed, the high-order bits are being computed twice in parallel: once assuming that $c_4 = 0$ and once assuming $c_4 = 1$.

# Speeding Up Integer Addition

**Figure I.21 Carry-select adder.** As soon as the carry-out of the rightmost block is known, it is used to select the other sum bits.

# Asymptotic time and space requirements

| Adder | Time | Space |
|---|---|---|
| Ripple | $0(n)$ | $0(n)$ |
| CLA | $0(\log n)$ | $0(n \log n)$ |
| Carry-skip | $0(\sqrt{n})$ | $0(n)$ |
| Carry-select | $0(\sqrt{n})$ | $0(n)$ |

# Speeding Up Integer Multiplication and Division

SRT Division

1. If B has k leading zeros, shift all registers left by k bits

For i = 0 to n-1

2.a. If top 3 bits of P are equal, Qi = 0, shift (P,A) one bit left

2.b. If top 3 bits of P are not all equal and P is negative, set Qi = -1, shift (P,A) one bit left, add B

2.c. Otherwise set Qi = 1, shift (P,A) one bit left, subtract B

End loop

3. If remainder is –ve, correct by adding B and correct quotient by subtracting 1 from Qo. Shift the remainder k bits right

# Speeding Up Integer Multiplication and Division

| P | A | |
|---|---|---|
| 00000 | 1000 | Divide 8 = 1000 by 3 = 0011. B contains 0011. |
| 00010 | 0000 | Step 1: B had two leading 0s, so shift left by 2. B now contains 1100. |
| | | Step 2.1: Top three bits are equal. This is case (a), so |
| 00100 | 0000 | set $q_0 = 0$ and shift. |
| | | Step 2.2: Top three bits not equal and P ≥ 0 is case (c), so |
| 01000 | 0001 | set $q_1 = 1$ and shift. |
| + 10100 | | Subtract B. |
| 11100 | 0001 | Step 2.3: Top bits equal is case (a), so |
| 11000 | 0010 | set $q_2 = 0$ and shift. |
| | | Step 2.4: Top three bits unequal is case (b), so |
| 10000 | $010\bar{1}$ | set $q_3 = -1$ and shift. |
| + 01100 | | Add B. |
| 11100 | | Step 3. Remainder is negative so restore it and subtract 1 from $q$. |
| + 01100 | | |
| 01000 | | Must undo the shift in step 1, so right-shift by 2 to get true remainder. |
| | | Remainder = 10, quotient = $010\bar{1} - 1 = 0010$. |

**Figure I.23 SRT division of $1000_2/0011_2$.** The quotient bits are shown in bold, using the notation $\bar{1}$ for −1.

# Difference Between nonrestoring division and SRT division

- ALU decision rule: In non-restoring it is determined by sign of P, in SRT it is determined by two most significant bits of P
- Final quotient: In non-restoring it is immediate from successive signs of P, in SRT there are three quotient digits (0,1,-1) and the final quotient is computed in full n-bit adder
- Speed: SRT division will be faster on operands that produce zero quotient bits

# Speeding Up Integer Multiplication using single Adder

- Load sum and carry bits of P to zero
- Shift lower order sum bit of P into A
- n-1 high order bits of P are shifted to next lower-adder in next cycle
- Two Drawbacks to carry-save adders

  i. Requires more hardware

  ii. After last step higher order word must be fed into an ordinary adder to combine the sum and carry parts
- Speedup without using extra adder is to examine k low order bits of A at each step than just one bit

# Speeding Up Integer Multiplication using single Adder

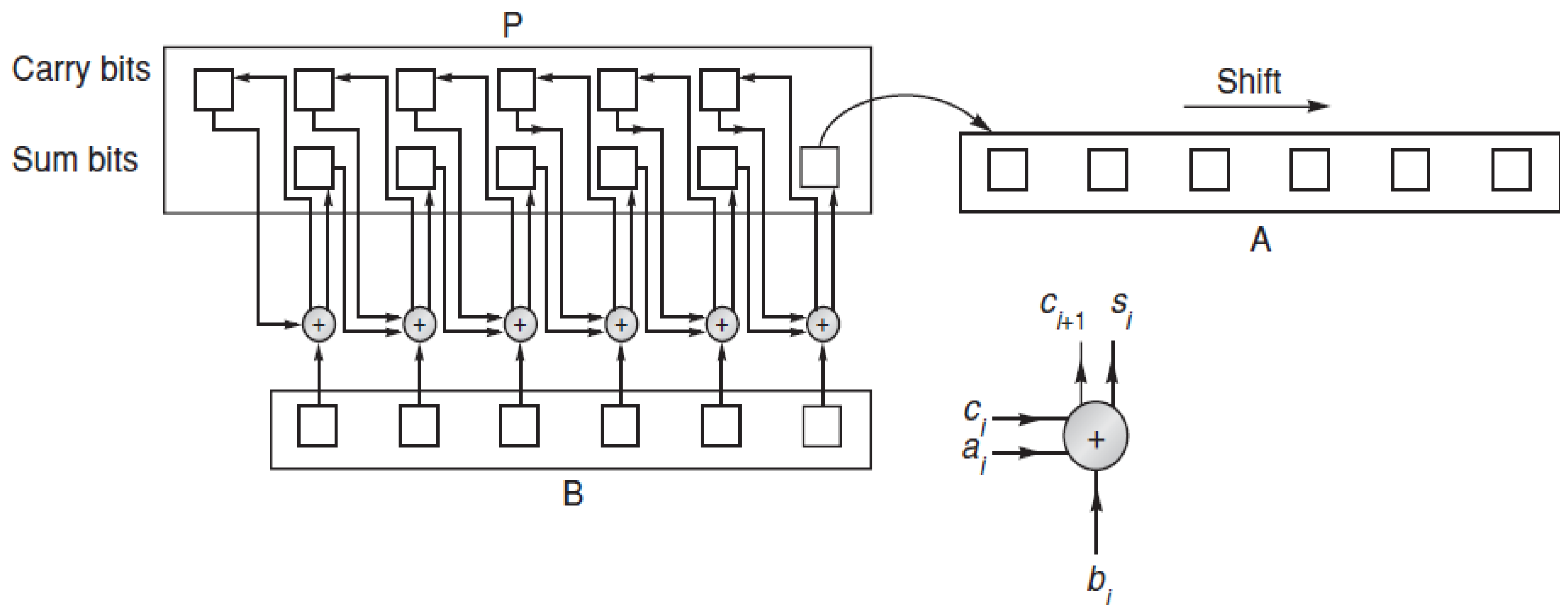


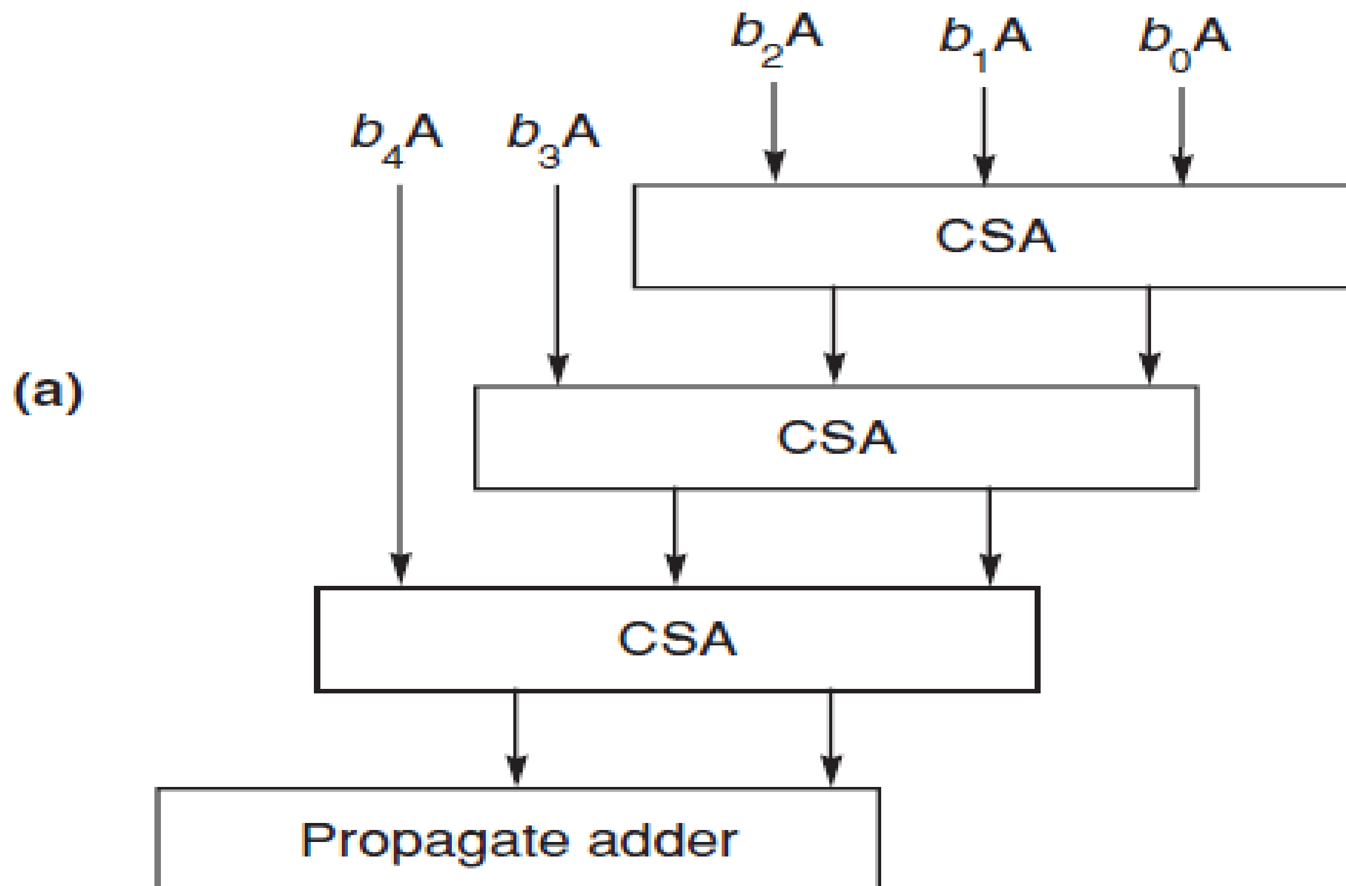

**Figure I.24 Carry-save multiplier.** Each circle represents a (3,2) adder working independently. At each step, the only bit of P that needs to be shifted is the low-order sum bit.

# Speeding Up Multiplication using single Adder

| P | A | L | |
|---|---|---|---|
| 00000 | 1001 | | Multiply –7 = 1001 times –5 = 1011. B contains 1011. |
| + 11011 | | | Low-order bits of A are 0, 1; L = 0, so add B. |
| 11011 | 1001 | | |
| 11110 | 1110 | 0 | Shift right by two bits, shifting in 1s on the left. |
| + 01010 | | | Low-order bits of A are 1, 0; L = 0, so add –2b. |
| 01000 | 1110 | 0 | |
| 00010 | 0011 | 1 | Shift right by two bits. |
| | | | Product is 35 = 0100011. |

**Figure I.26 Multiplication of –7 times –5 using radix-4 Booth recoding.** The column labeled L contains the last bit shifted out the right end of A.

# Faster Multiplication with many Adders



# Faster Multiplication with many Adders

An array Multiplier

- Latency similar to carry-save adder
- Multiplication can be pipelined, increasing total throughput
- Space available may not hold the array to multiply two double precision numbers

# Faster Multiplication with many Adders

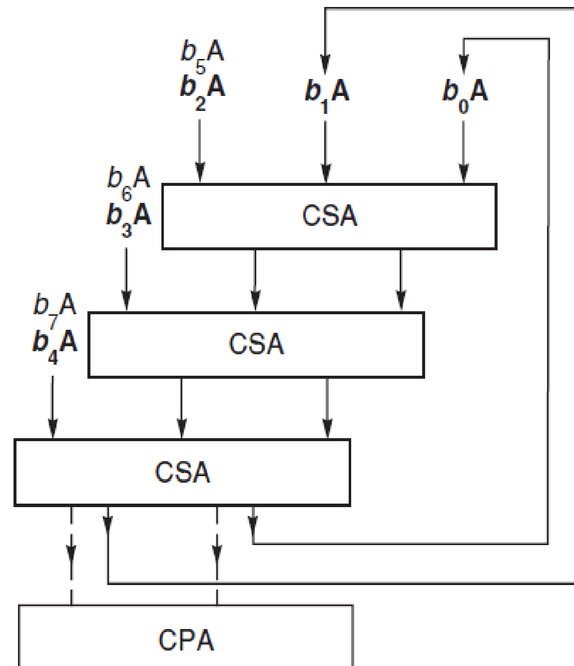



**Figure I.28 Multipass array multiplier.** Multiplies two 8-bit numbers with about half the hardware that would be used in a one-pass design like that of Figure I.27. At the end of the second pass, the bits flow into the CPA. The inputs used in the first pass are marked in bold.

# Faster Multiplication with many Adders

- The above design cannot be pipelined
- Array has smaller latency than using single adder, array is combinational circuit, signal flows directly without being clocked
- Even/odd array design for speedup
- The notion of running two adds in parallel

# Faster Multiplication with many Adders

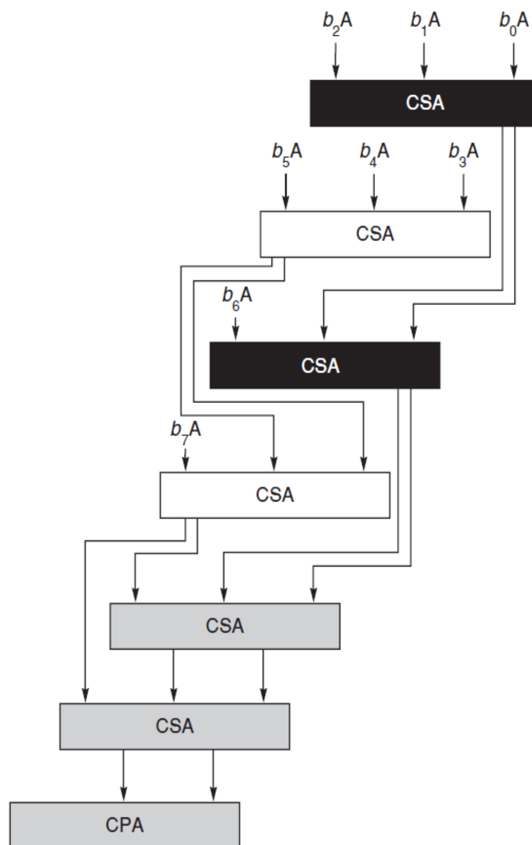



**Figure I.29 Even/odd array.** The first two adders work in parallel. Their results are fed into the third and fourth adders, which also work in parallel, and so on.

# Faster Multiplication with many Adders

- Even/odd array has time of O(n)
- Can be reduced to logn using tree
- Tree that uses full adders known as Wallace tree
- Wallace tree was designed of choice for high speed multipliers
- Due to irregular structure of Wallace trees, not used in VLSI designs

# Faster Multiplication with many Adders

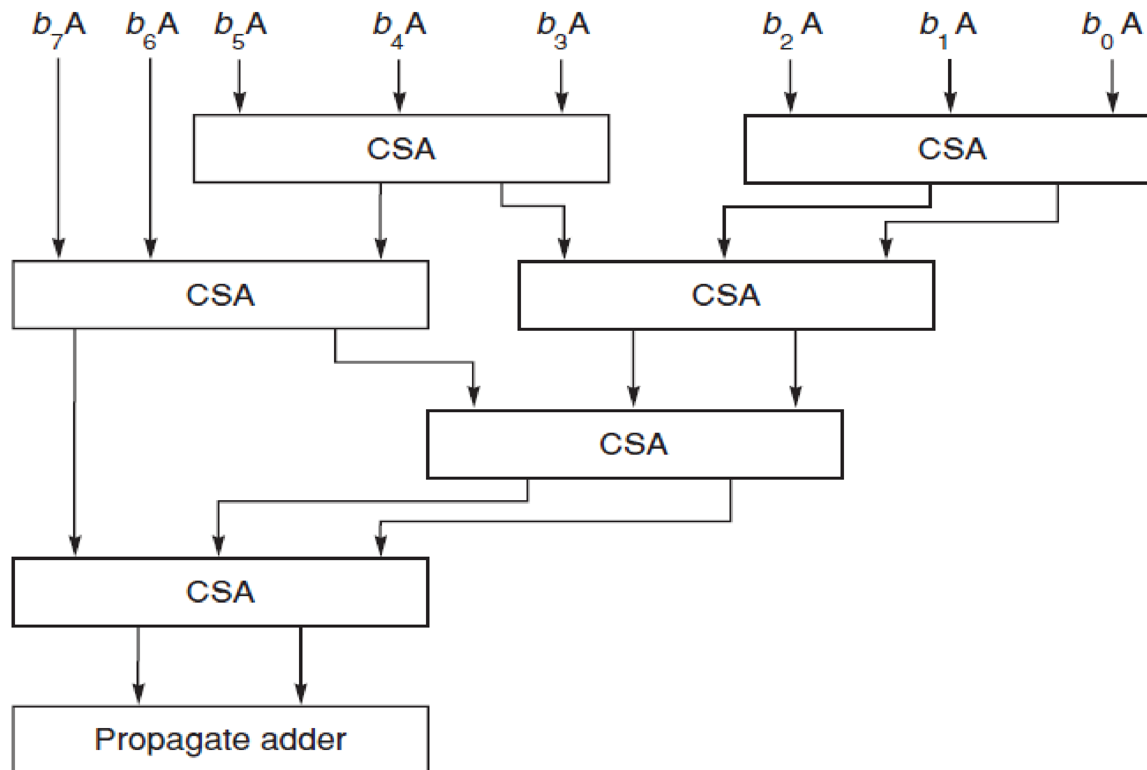
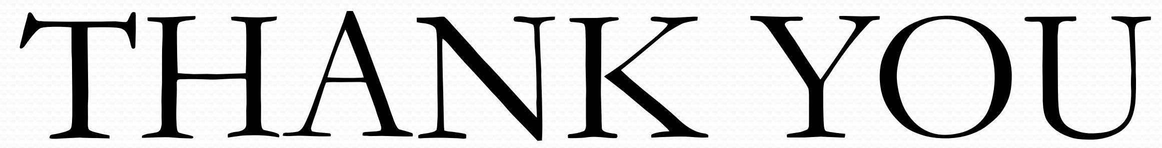



**Figure I.30 Wallace tree multiplier.** An example of a multiply tree that computes a product in 0(log *n*) steps.

# References

- Computer Architecture - John L Hennessy, David Patterson
- Computer Organization Architecture – William Stallings


THANK YOU