# IV. Design of Arithmetic Circuits

# 4. Design of Arithmetic Circuits

4.1 Introduction to Arithmetic Circuits

4.2 Number Systems and Codes

4.3 Design of Fast Adders

4.4 Design of Fast Multipliers

4.5 Design of Fast divider

4.6 Floating Point Arithmetic

# 4.1 Introduction to Arithmetic Circuits

- A large proportion of data that is collected is numerical data which must be proceeded arithmetically
- Efficient high-speed hardware implementations of arithmetic operations are desirable
- Some arithmetic circuits are implemented in combinational logic : addition, subtraction, comparing
- Other arithmetic circuits are too bulky in Combinational logic : multiplier Div, Sqt, filtering
- Regular, iterative structure: a recurring theme in the design of arithmetic circuits.

# 4.2 Number Systems and Codes

4.2.1 Positional Number Systems

4.2.2 Octal and Hexadecimal Numbers

4.2.3 General Positional-Number-System Conversions

4.2.4 Addition and Subtraction of Non-decimal Numbers

4.2.5 Representation of Negative Numbers

4.2.6 Two’s Complement Addition and Subtraction

4.2.7 One’s Complement Addition and Subtraction

4.2.8 Binary Multiplication

4.2.9 Binary Division

# 4.2.1 Positional Number Systems

- Traditional number system
- A number is represented by a string of digits, each digit position has an associated weight.
  - 1734=1*1000+7*100+3*10+4
  - 585.55=5*100+8*10+5*1+5*0.1+5*0.01
  - 10 is called base or radix
- General : $D=d_{p-1}d_{p-2}\ldots d_1d_0d_{-1}d_{-2}\ldots d_{-n}$
- Binary : $B=b_{p-1}b_{p-2}\ldots b_1b_0b_{-1}b_{-2}\ldots b_{-n}$

$$D=\sum_{i=-n}^{p-1} d_i r^i \qquad B=\sum_{i=-n}^{p-1} b_i 2^i$$

- $100.001_2=(?)_{10}$

# 4.2.2 Octal and Hexadecimal Numbers

- Radix 10 is important
- Radix 2 is important
- Radix 8 & 16 ? (When you are 40?)

| B | Decimal | Octal | Hex |
|---|---|---|---|
| 0000 | 0 | 0 | 0 |
| 0001 | 1 | 1 | 1 |
| 0010 | 2 | 2 | 2 |
| 0011 | 3 | 3 | 3 |
| 0100 | 4 | 4 | 4 |
| 0101 | 5 | 5 | 5 |
| 0110 | 6 | 6 | 6 |
| 0111 | 7 | 7 | 7 |
| 1000 | 8 | 10 | 8 |
| 1001 | 9 | 11 | 9 |
| 1010 | 10 | 12 | A |
| 1011 | 11 | 13 | B |
| 1100 | 12 | 14 | C |
| 1101 | 13 | 15 | D |
| 1110 | 14 | 16 | E |
| 1111 | 15 | 17 | F |

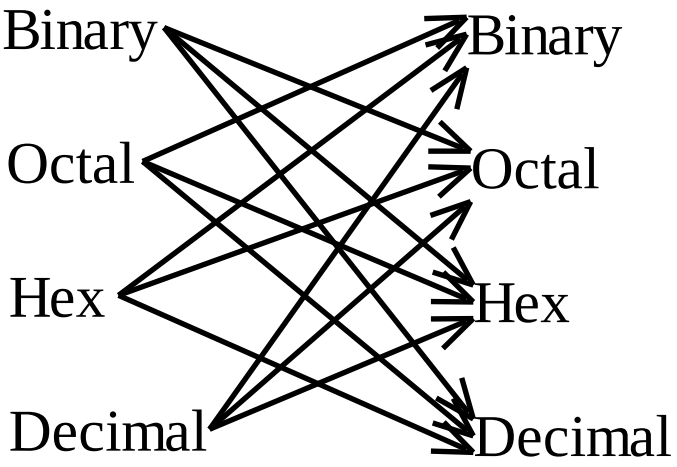
# 4.2.3 General Positional-Number-System Conversions

- Radix r to decimal conversion

$$D = \sum_{i=-n}^{p-1} d_i r^i$$

  - The value of the number can be found by converting each digit of the number to its radix-10 equivalent and expanding the formula using radix –10 arithmetic
  - $436.5_8$=

- $D=d_{p-1}d_{p-2}\ldots d_1 d_0 d_{-1} d_{-2}\ldots d_{-n}$
- $=(\ldots((dp-1)*r+dp-2)*r+\ldots)r+d1$



# 4.2.4 Addition and Subtraction of Nondecimal Numbers

- To add two binary numbers X and Y, we add together the least significant bits with an initial carry (cin) of 0, producing carry (cout) and sum (s) bits. We continue processing bits from right to left, adding the carry out of each column into the next column's sum. (Examples)
- Binary subtraction is performed similarly, using borrow instead of carries between steps, and producing a difference bit d. (Examples)

# 4.2.5 Representation of Negative Numbers

## 1) Signed Magnitude Representation

- The signed magnitude system is applied to binary numbers by using an extra bit position to represent the sign.
- To build a adder and subtractor: disadvantage and advantage.

## 2) Complement Number Systems

- Radix complement system, the complement of an n-digit number is obtained by subtracting it from $r^n$
- Example (10's complement, 9's complement)
- Two's complements
  - MSB serves as the sign bit
  - Examples

# 4.2.5 Representation of Negative Numbers

| Decimal | Two's complement | One's complement | Signed magnitude |
|---|---|---|---|
| -8 | 1000 | | |
| -7 | 1001 | 1000 | 1111 |
| -6 | 1010 | 1001 | 1110 |
| -5 | 1011 | 1010 | 1101 |
| -4 | 1100 | 1011 | 1100 |
| -3 | 1101 | 1100 | 1011 |
| -2 | 1110 | 1101 | 1010 |
| -1 | 1111 | 1110 | 1001 |
| 0 | 0000 | 1111 or 0000 | 1000 or 0000 |
| 1 | 0001 | 0001 | 0001 |
| 2 | 0010 | 0010 | 0010 |
| 3 | 0011 | 0011 | 0011 |
| 4 | 0100 | 0100 | 0100 |
| 5 | 0101 | 0101 | 0101 |
| 6 | 0110 | 0110 | 0110 |
| 7 | 0111 | 0111 | 0111 |

# 4.2.6 Two's Complement Addition and Subtraction

- Addition
  - The results will always be the correct sum as long as the range of the number system is not exceeded.
  - Overflow detector :
  - $V=a_n b_n r_n' + a_n' b_n' r_n$ (example)
- Subtraction
  - Negate the subtrahend by taking its two's complement, and then add it to the minuend using the normal rules for addition
  - Example

# 4.2.7 One's Complement Addition and Subtraction

- Addition
  - If we start at 1000 (-7) and count up, we obtain each successive one one's complement number by adding 1 to the previous one, except at the transition from 1111 (0) to 0001 (1)
  - Suggestion: perform a standard binary addition, but add an extra 1 whenever we count past 1111
  - Examples
- Subtraction
  - Complement all bits of the subtrahend and proceed as in addition

# 4.2.8 Binary Multiplication

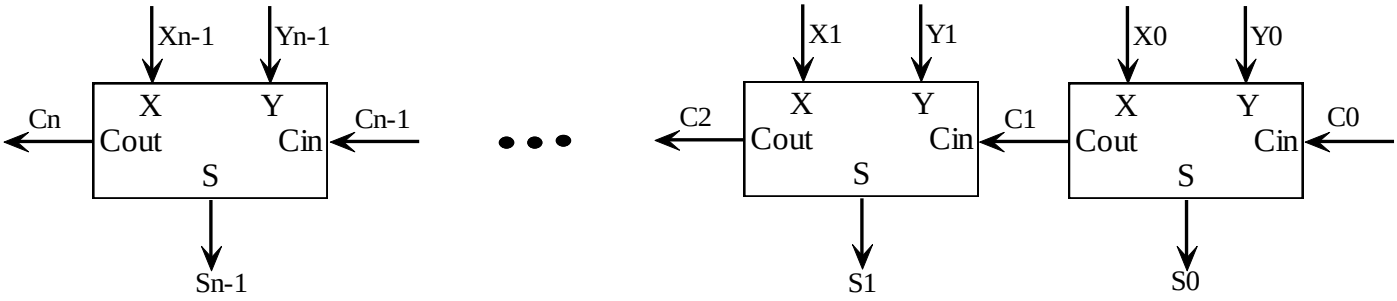
- Unsigned multiplication
  - Shift and Add Multiplication
  - Examples
- Signed multiplication
- Two’s complement multiplication
  - Example

# 4.2.9 Binary Division

- Unsigned division
  - Shift and subtract
- Signed division
  - Using unsigned division
  - Make the quotient positive if the operands had the same sign, negative if they has different signs
  - The reminder should be given the same sign as the division

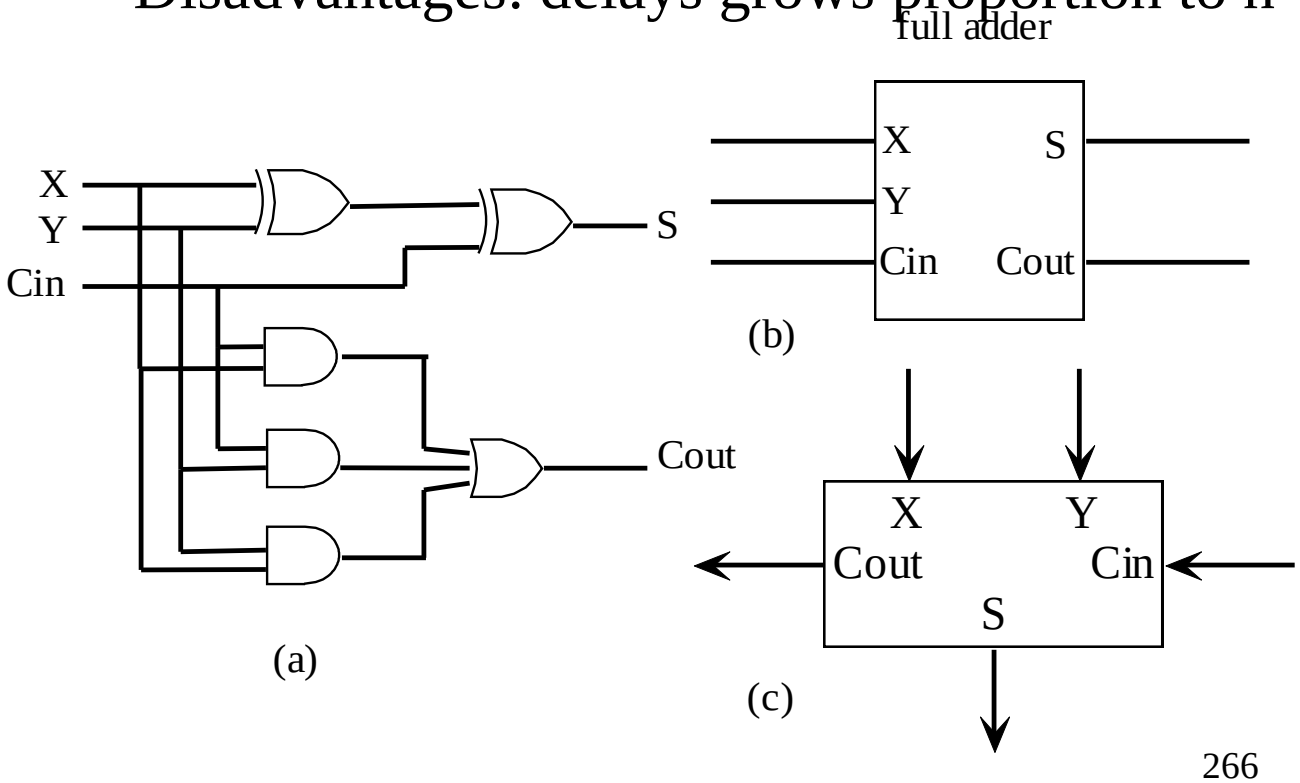
# 4.3 Design of Fast Adders

- Ripple carry adders
  - Decompose addition into bit-wise operations
  - Use the same circuit design for each bit
  - Get simple regular structure
- Carry signal ripples from LSB to MSB



# Full Adder Circuit

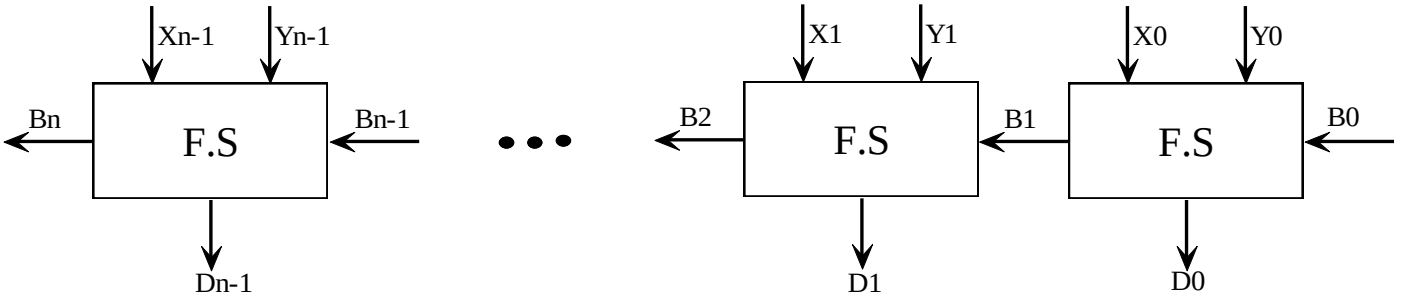
- $C_{i+1}$=Ci*ai+Ci*bi+ai*bi  $S_i = a_i \oplus b_i \oplus c_i$
- Gate delay from $C_i$ to $C_{i+1}$, Ci to Si?
- Critical path analysis:
  - Critical path: the signal path that limits the speed of the circuit
- Delay for n bit ripple carry adder ?
- Advantages: simple,iterative.
- Disadvantages: delays grows proportion to n

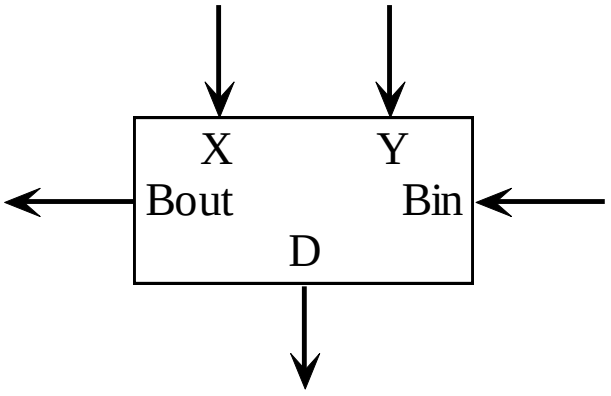


# Full Subtractor Circuit

Gate delay from $B_i$ to $B_{i+1}$, Bi to Di?

- Delay for n bit ripple carry adder ?
- Advantages & Disadvantages: same as ripple carry adder




XiYi

| Bi | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | | | | |
| 1 | | | | |

Di

XiYi

| Bi | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | | | | |
| 1 | | | | |

Bi

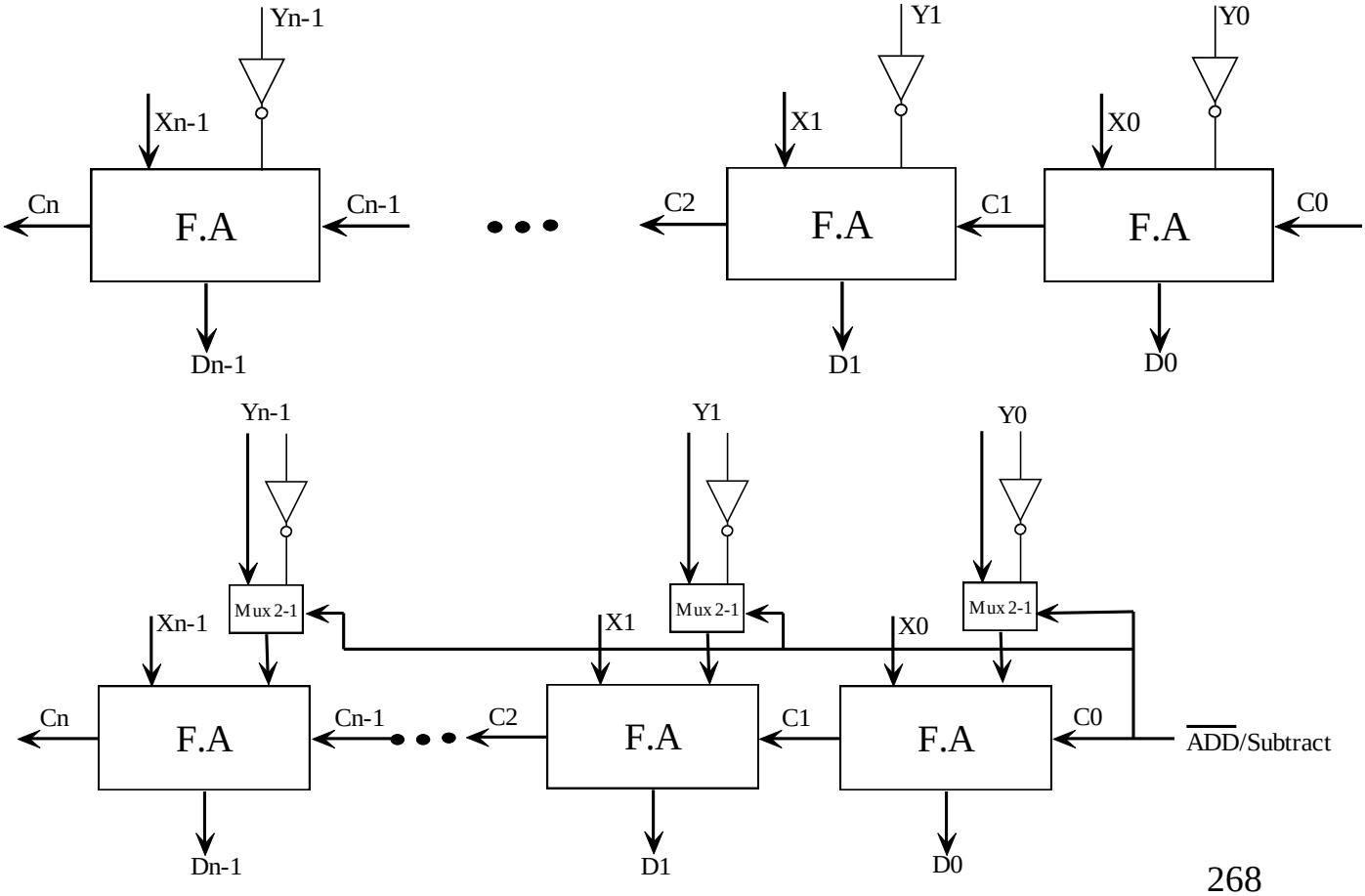
# 2’s Complement subtractors

Gate delay ?

Fast adder solution:

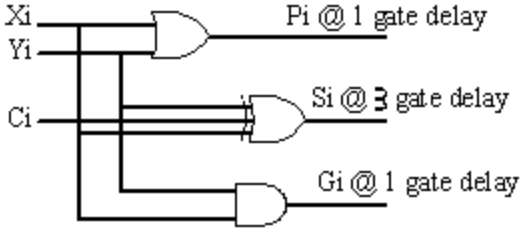
#1 PLA delay?

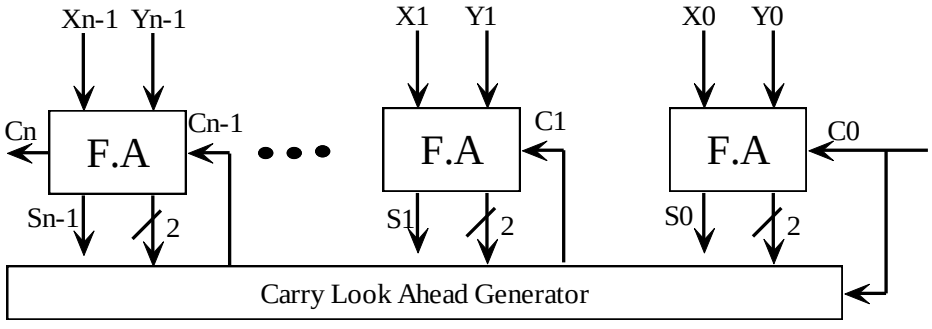
#2 Find a Compromise Circuit Design



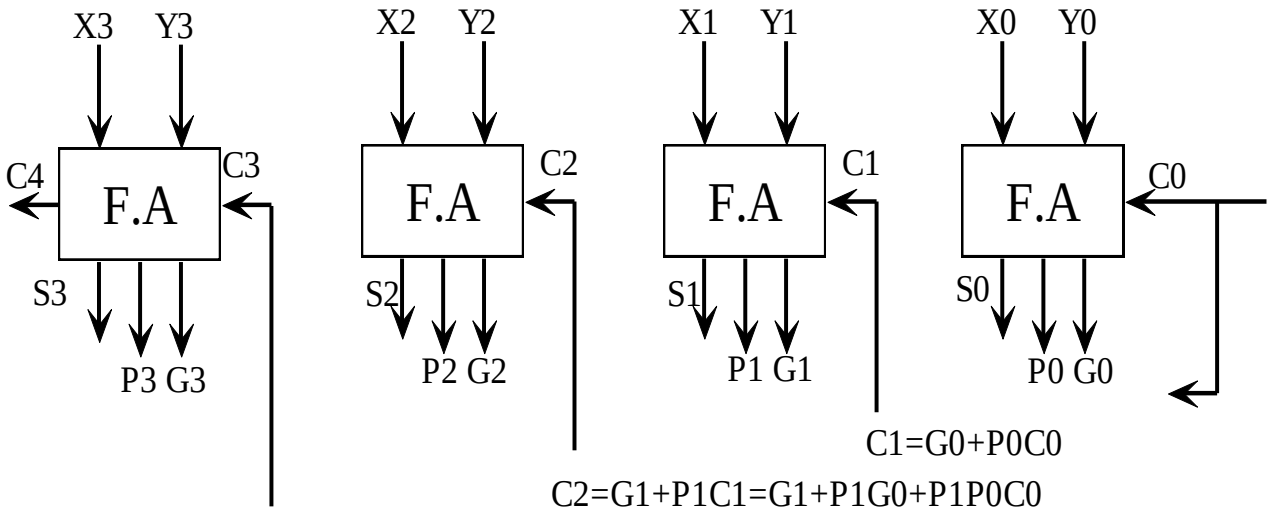
# Carry Look Ahead Logic

- In the 4-bit "ripple" adder, the carry out of each stage, Ci+1, is expressed as a function of Xi, Y, and Ci. The basic idea of carry look ahead logic is to express each Ci in terms of Xi, Xi-1,…X0, Yi, Yi-1, …Y0, and C0 directly. Delay from C0 to Cn-1?
- Recall C1=X0Y0+X0C0+Y0C0
- Define: generate Gi=XiYi
- propagate Pi=Xi+Yi
- C1= X0Y0+X0C0+Y0C0
- =X0Y0+(X0+Y0)C0=G0+P0C0
- C2=G1+P1C1=G1+P1G0+P0P1C0
- C3=G2+P2C2=G2+P2(G1+P1G0+P0P1C0)
- =G2+P2G1+P2P1G0+P2P1P0C0
- …



# 4-bit Carry Look- Ahead Adder (CLA)

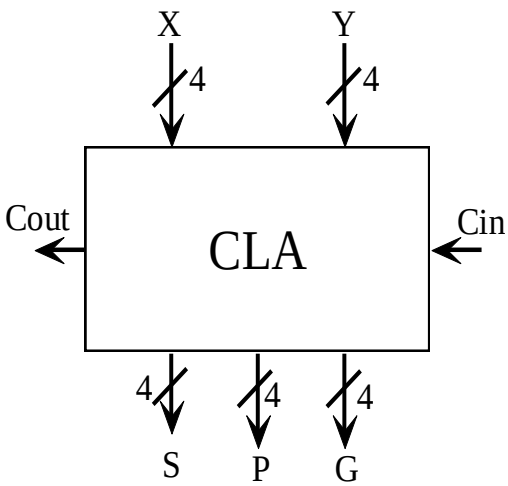



C1=G0+P0C0

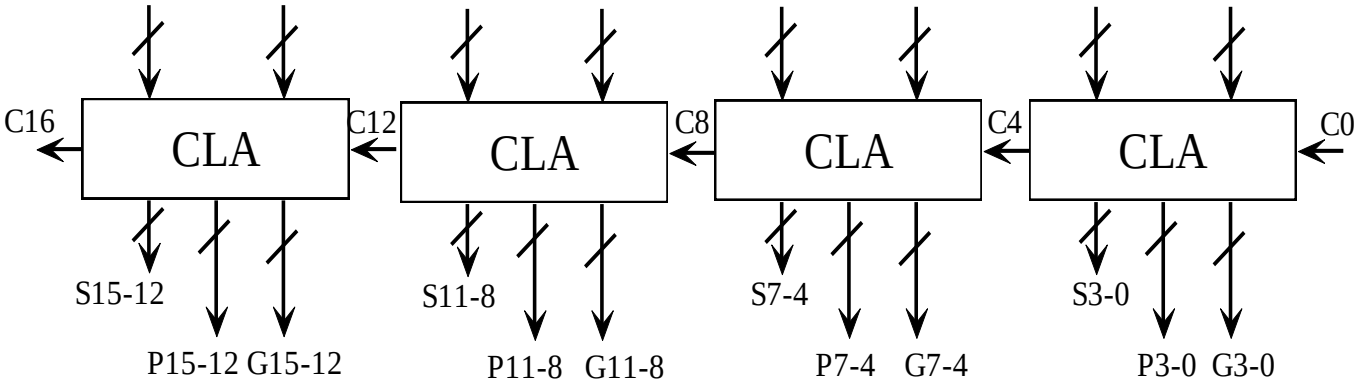
C2=G1+P1C1=G1+P1G0+P1P0C0

C3=G2+P2C2=G2+P2G1+P2P1G0+P2P1P0C0

C4=G3+P3C3=G3+P3G2+P3P2G1+P3P2P1G0+P3P2P1P0C0



# Cascade 4-bit CLAs

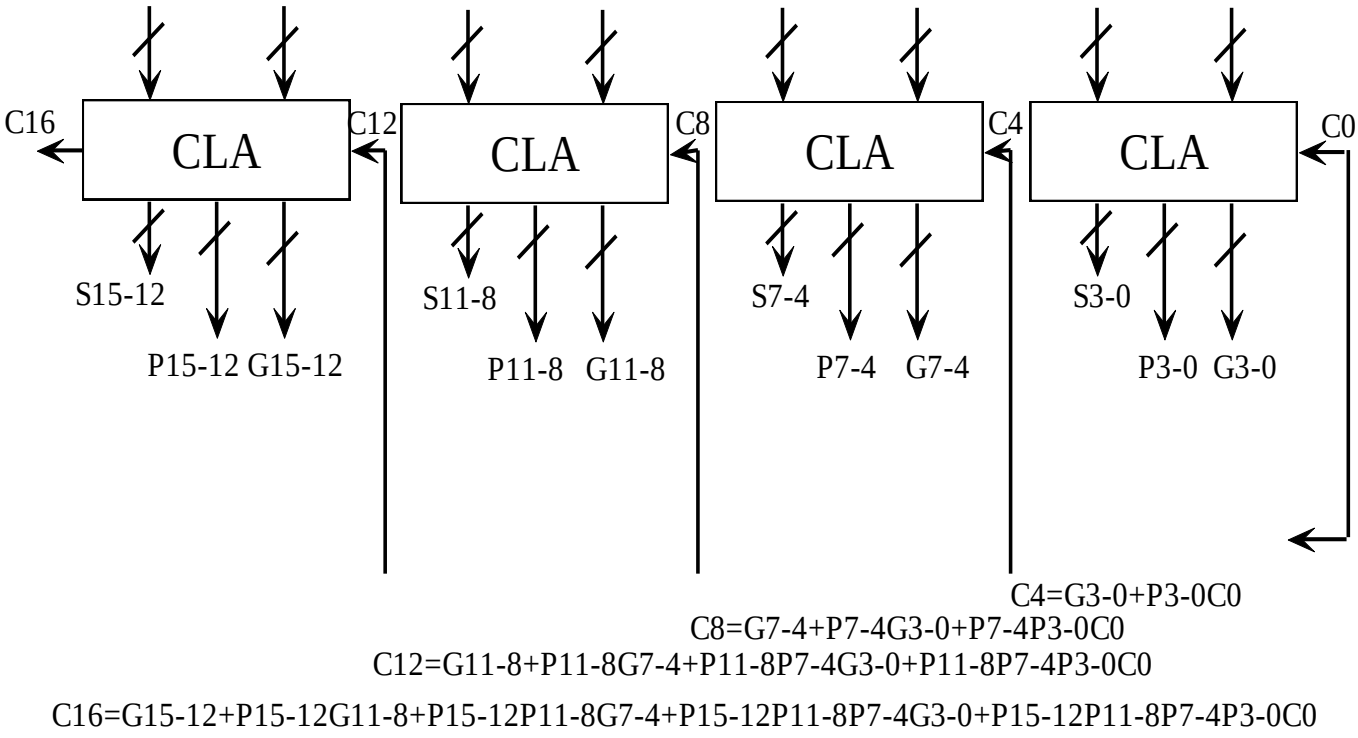



- Time Analysis:
- C0 → C4, C4→C8, C8→C12, C12 → C15, C15 →S15

Worst case delay:

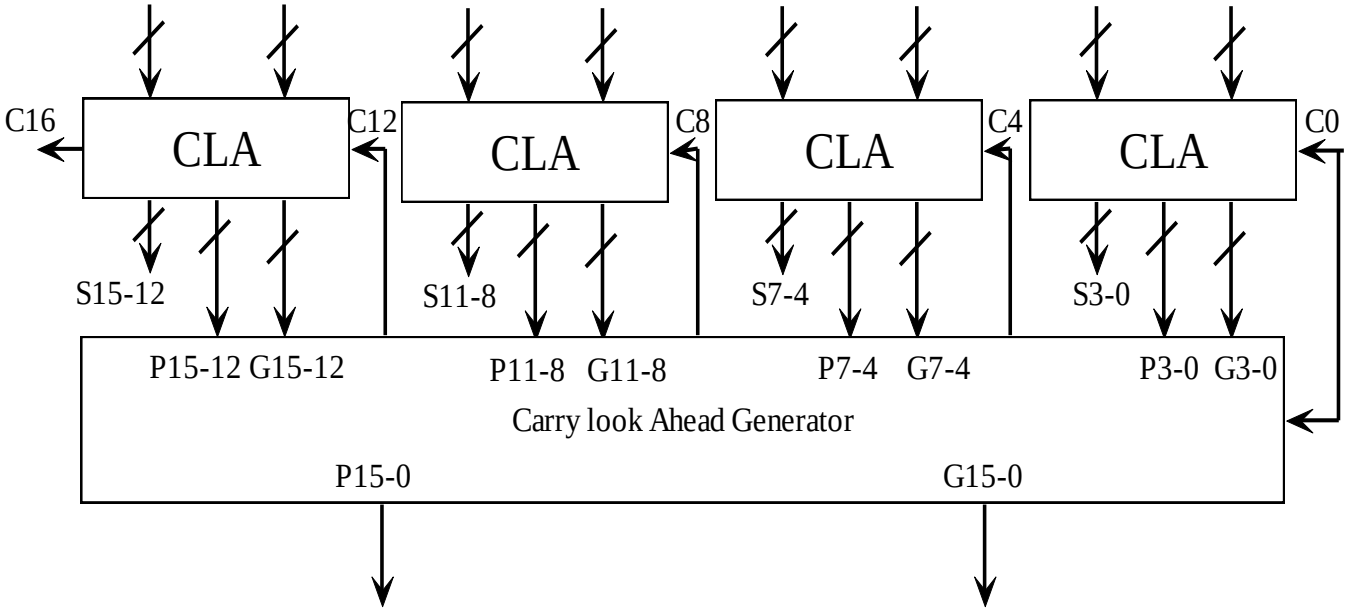
- 16-bit ripple carry adder worst case delay:
- General 4-bit CLAs delay:

# Cascade look Ahead Generator



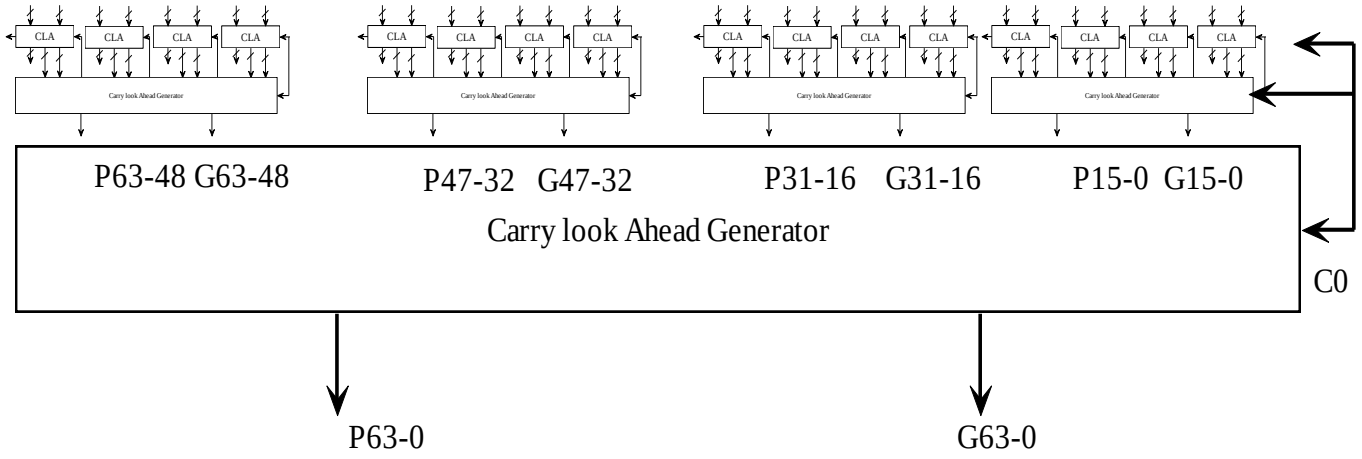

- G3-0=G3+P3G2+P3P2G1+P3P2P1G0
- P3-0=P3P2P1P0
- G7-4=G7+P7G6+P7P6G5+P7P6P5G4
- P7-4=P7P6P5P4
- G11-8=G11+P11G10+P11P10G9+P11P10P9G8
- P11-8=P11P10P9P8
- G15-12=G15+P15G14+P15P14G13+P15P14P13G12
- P3-0=P15P14P13P12

# Fast 16 bit CLA and CLG



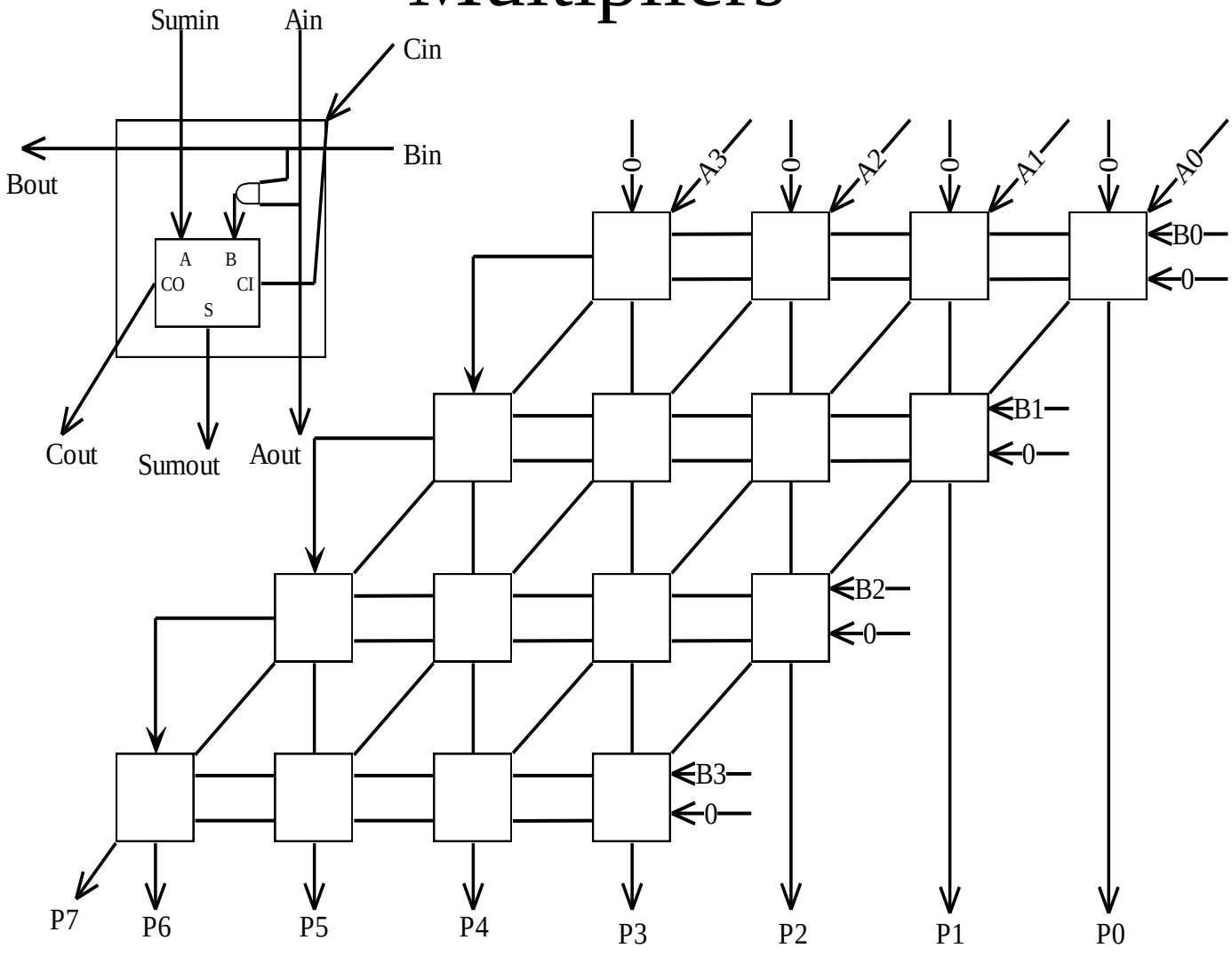

- G15-0=G15-12+P15-12G11-8+P15-12P1-8G7-4+P15-12P11-8P7-4G3-0
- P15-0=P15-12P11-8P7-4P3-0
- Time Analysis:
- C0 → G3-0, G3-0→C12, C12→C15
- C15→S15
- Worst case delay:
- Cascade CLA delay:
- Ripple carry adder delay:

# Fast 64 Bit Adder Based on CLA and CLG



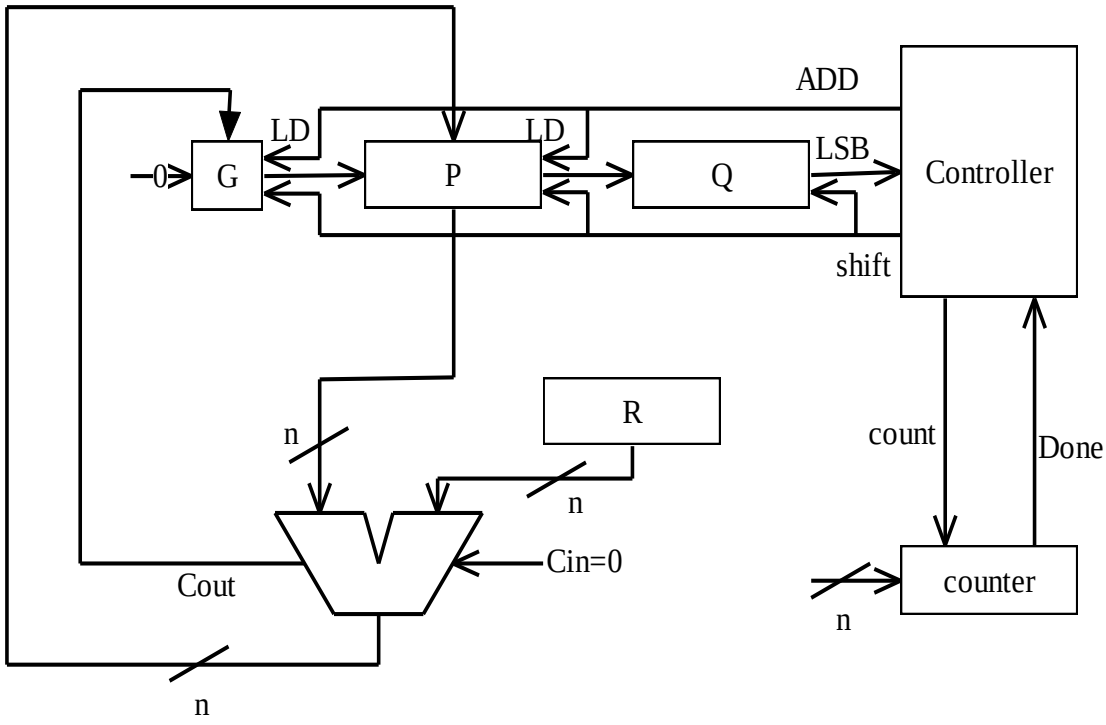

- Time Analysis:
- C0 → G3-0, G3-0→G15-0, G15→C48, C48→C60, C60→C63, C63→S63
- Worst case delay:
- Cascade CLA delay:
- Ripple carry adder delay:
- 128 bit CLA +CLG delay:
- General :

# 4.4.1 Combinational Multipliers




- Time Analysis:
- Cin → Cout, Cin→Sumout,
  - Worst case delay: 2(n-1)+3(n-1)+2(n-1)+3

# 4.4.2 Sequential Add/Shift Unsigned Multiplier




| R | | | | G | P | | | | Q | | | | operation |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | **0** | **0** | **0** | **0** | 0 | 0 | 0 | 0 | |
| 1 | 0 | 1 | 1 | 0 | **0** | **0** | **0** | **0** | 1 | 1 | 0 | 1 | initialize |
| 1 | 0 | 1 | 1 | 0 | **1** | **0** | **1** | **1** | 1 | 1 | 0 | 1 | add |
| 1 | 0 | 1 | 1 | 0 | **0** | **1** | **0** | **1** | 1 | 1 | 1 | 0 | shift |
| 1 | 0 | 1 | 1 | 0 | **0** | **0** | **1** | **0** | 1 | 1 | 1 | 1 | shift |
| 1 | 0 | 1 | 1 | 0 | **1** | **1** | **0** | **1** | 1 | 1 | 1 | 1 | add |
| 1 | 0 | 1 | 1 | 0 | **0** | **1** | **1** | **0** | 1 | 1 | 1 | 1 | shift |
| 1 | 0 | 1 | 1 | 1 | **0** | **0** | **0** | **1** | 1 | 1 | 1 | 1 | add |
| 1 | 0 | 1 | 1 | 0 | **1** | **0** | **0** | **0** | 1 | 1 | 1 | 1 | shift |

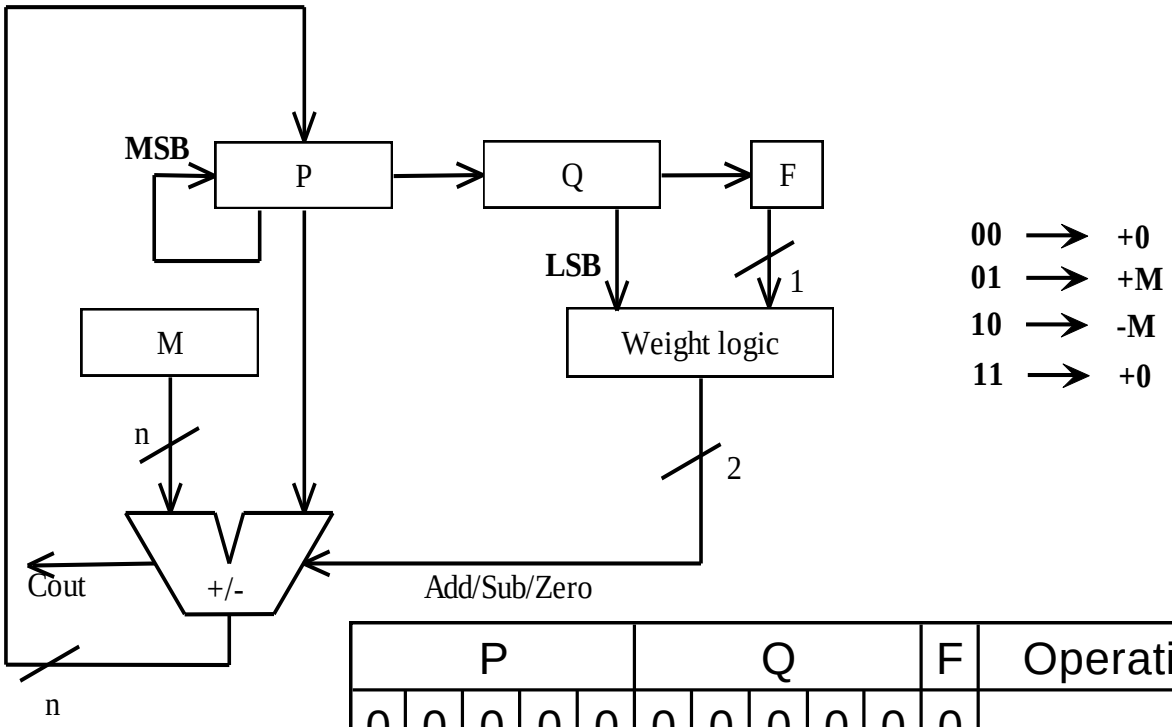
# 4.4.3Booth Re-coded Multipliers

- Used in the booth multiplication algorithm
- Consider adjacent bits as follows:
- 0 0  0 1  1 0  1 1
- 0 0  +1 1  -1 0  0 1
- Examples: (MSB is sign bit)
- 01010  100110
- 10011  0110101

# 4.4.3Booth Re-coded Multipliers (cont'd)

- The Booth Multiplication Algorithm
- Works for arbitrary combinations of positive and negative in 2's complement
  - Step1 recode the (positive or negative) multiplier
  - Step2: carry out the multiplication taking into account the sign of each nonzero multiplier bit
  - Examples: 45*30, and 13*(-16)
  - Example (-44)*(-19)

# 4.4.3Booth Multipliers (implementation)




**-12**

**-7**

---

**M= 10100**

**M*=01100**

**Q=11001**

**M**

**Q**

---

| P | | | | | Q | | | | | F | Operation |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | initialize |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | -M |
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | shift right |
| 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | +M |
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | shift right |
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | +0 |
| 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | shift right |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | -M |
| 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | shift right |
| 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | +0 |
| **0** | **0** | **0** | **1** | **0** | **1** | **0** | **1** | **0** | **0** | 1 | shift right |

# 4.4.3Fast Multipliers

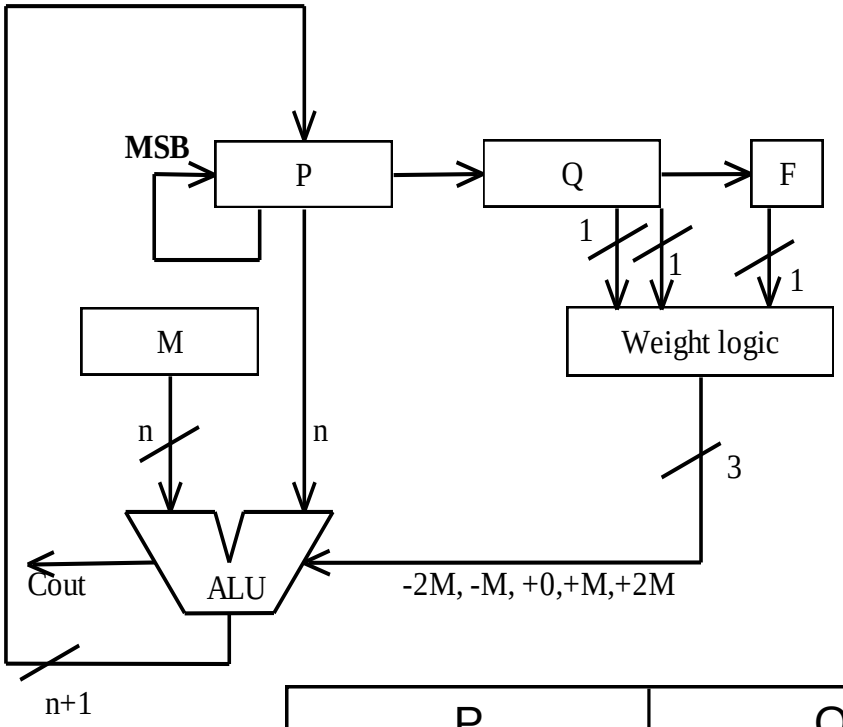
- The booth algorithm can save effort only when …
- Worst case ?
- Fast multiplication: a modified booth algorithm that guarantees that no more than 0.5 n partial products are required
- Idea: consider two booth bits at a time.

| multiplier bit pair | | next multiplier bit at right | booth bits | equivalent multiplicand |
|---|---|---|---|---|
| qi+1 | qi | qi-1 | | |
| 0 | 0 | 0 | +0+0 | 0*M |
| 0 | 0 | 1 | +0+1 | +1*M |
| 0 | 1 | 0 | +1-1 | +1*M |
| 0 | 1 | 1 | +1+0 | +2*M |
| 1 | 0 | 0 | -1+0 | -2*M |
| 1 | 0 | 1 | -1+1 | -1*M |
| 1 | 1 | 0 | +0-1 | -1*M |
| 1 | 1 | 1 | +0+0 | 0*M |

# 4.4.3Fast Multipliers(cont'd)

- Example (-24)*(-19)=101000*101101
  - Booth multiplication
  - Fast Multiplication

# 4.4.3Fast Multipliers Implementation




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

**-24 (101000)**
**-19 (101101)**

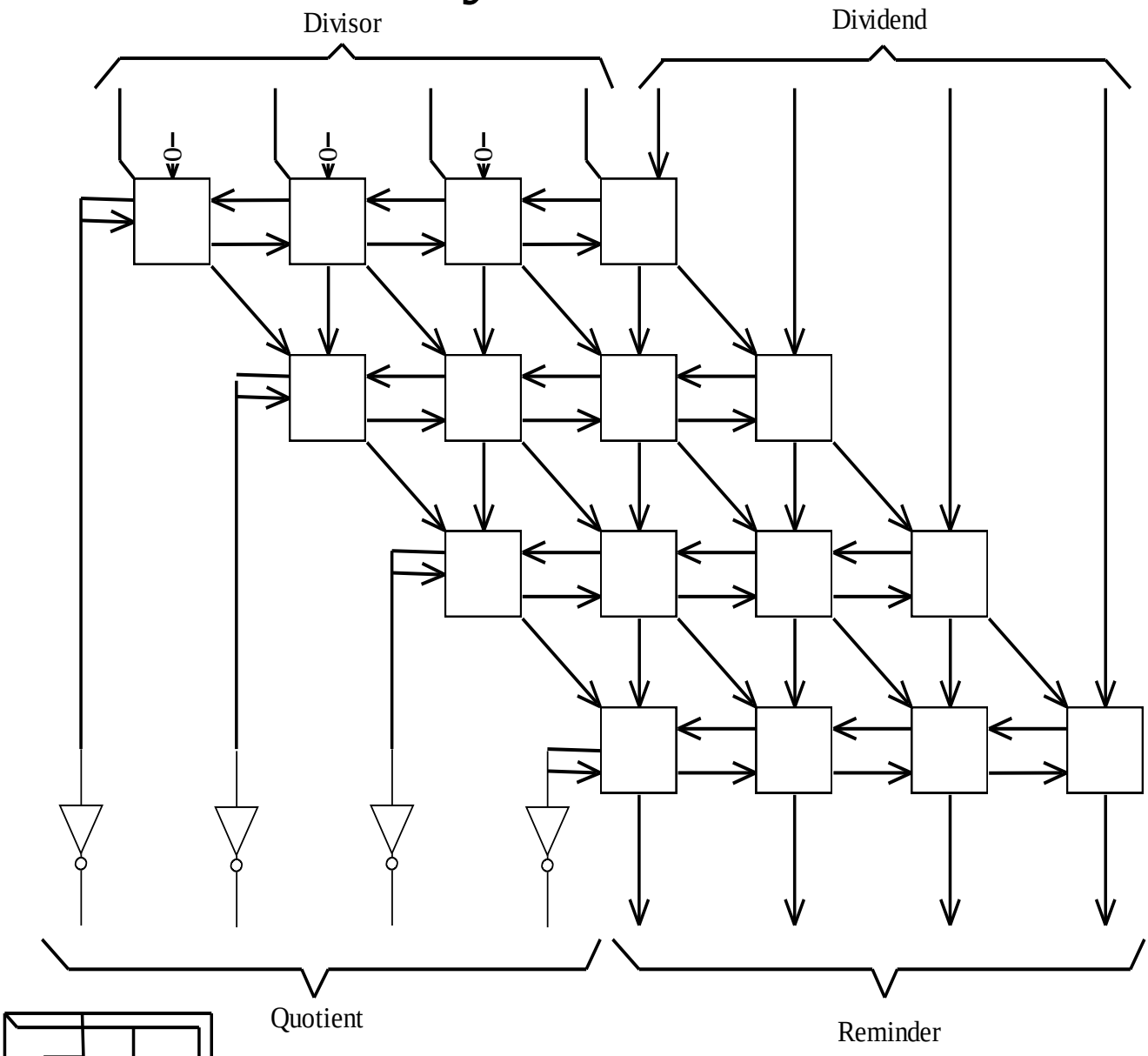
**M=101000**
**2M=1010000**
**-M=011000**
**-2M=0110000**

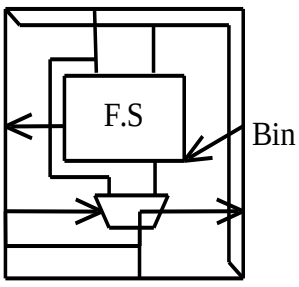
| P | | | | | | | | Q | | | | | | | | F | Operation |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | initialize |
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | +M |
| 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | shift 2 bits |
| 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | -M |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | shift 2 bits |
| 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | -M |
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | shift 2 bits |
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |
| **0** | **0** | **0** | **0** | **0** | **0** | **0** | **1** | **1** | **1** | **0** | **0** | **1** | **0** | **0** | **0** | 1 | shift 2 bits |

# 4.5.1 Unsigned Binary Division

- Division is more complicated to implement in hardware than multiplication.
- Division by hand involves conditional subtraction and shifting
- Hardware implementations of division are usually sequential approximations of the hand division algorithm.
- Example : 100010010(dividend)
- 1101(divisor)

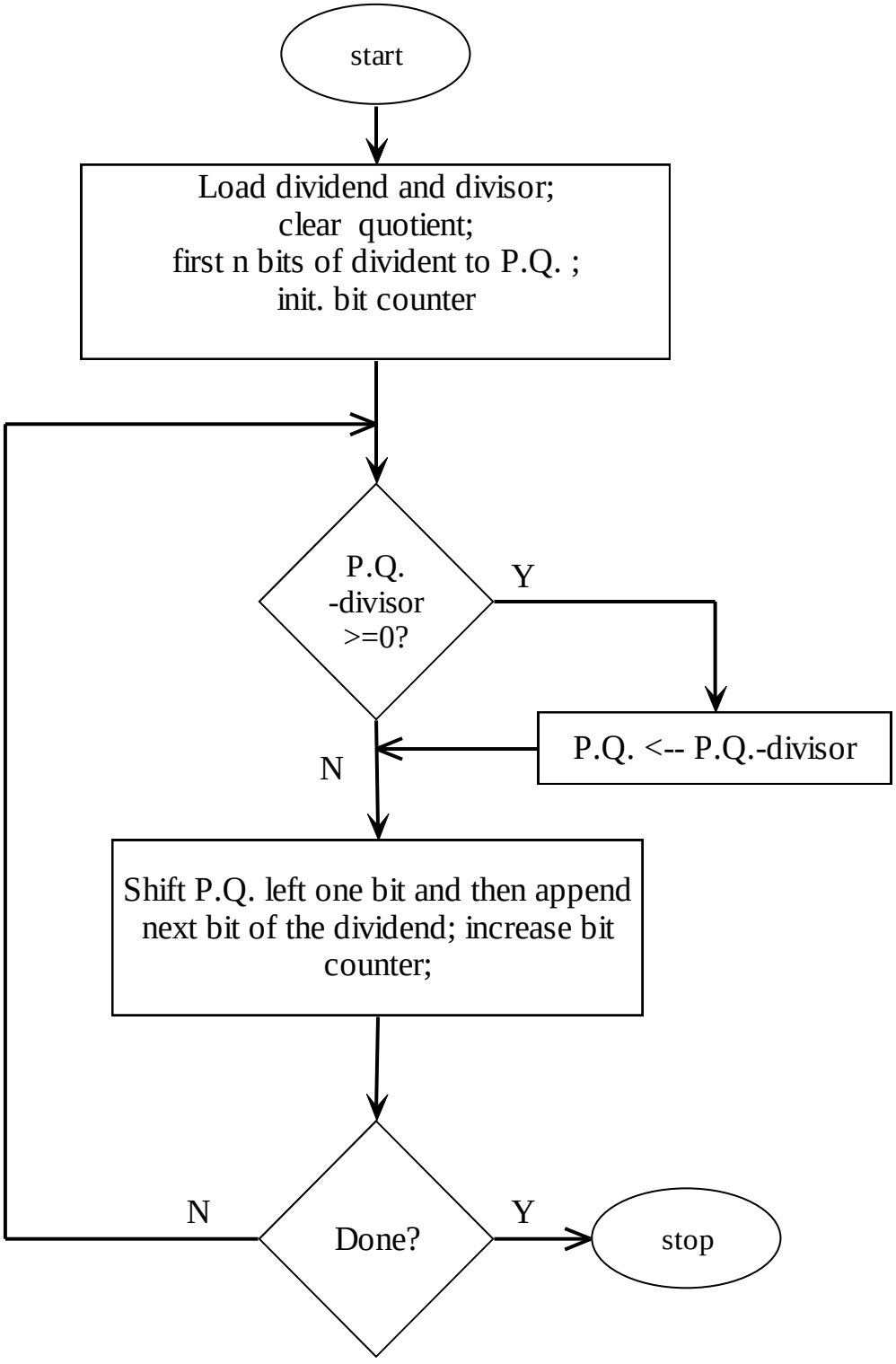
# 4.5.2 Purely Combinational Array Divider



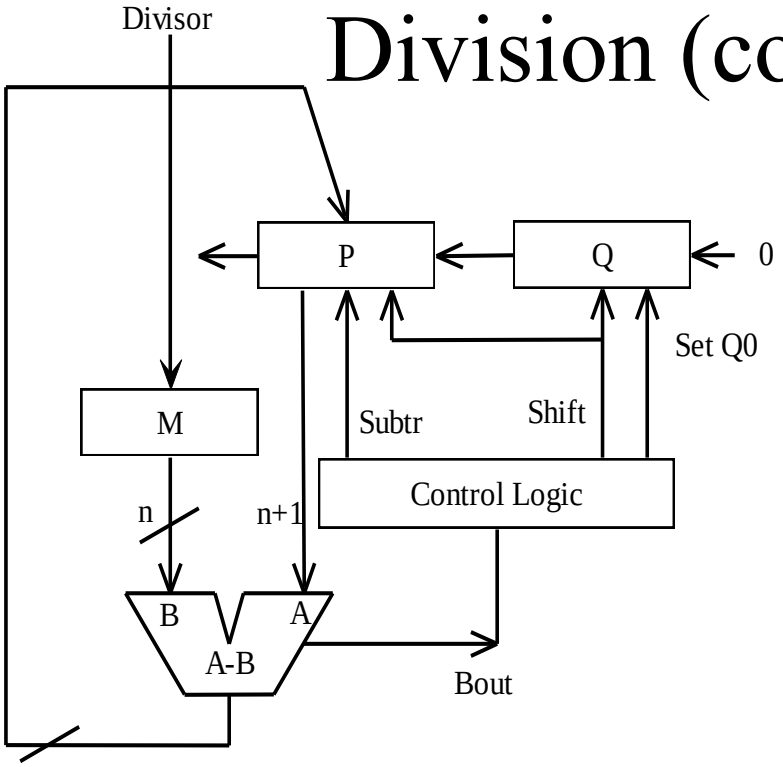




- Time delay 3 $n^2$ tg

# 4.5.3 Comparison Method Division



# 4.5.3 Comparison Method Division (cont'd)

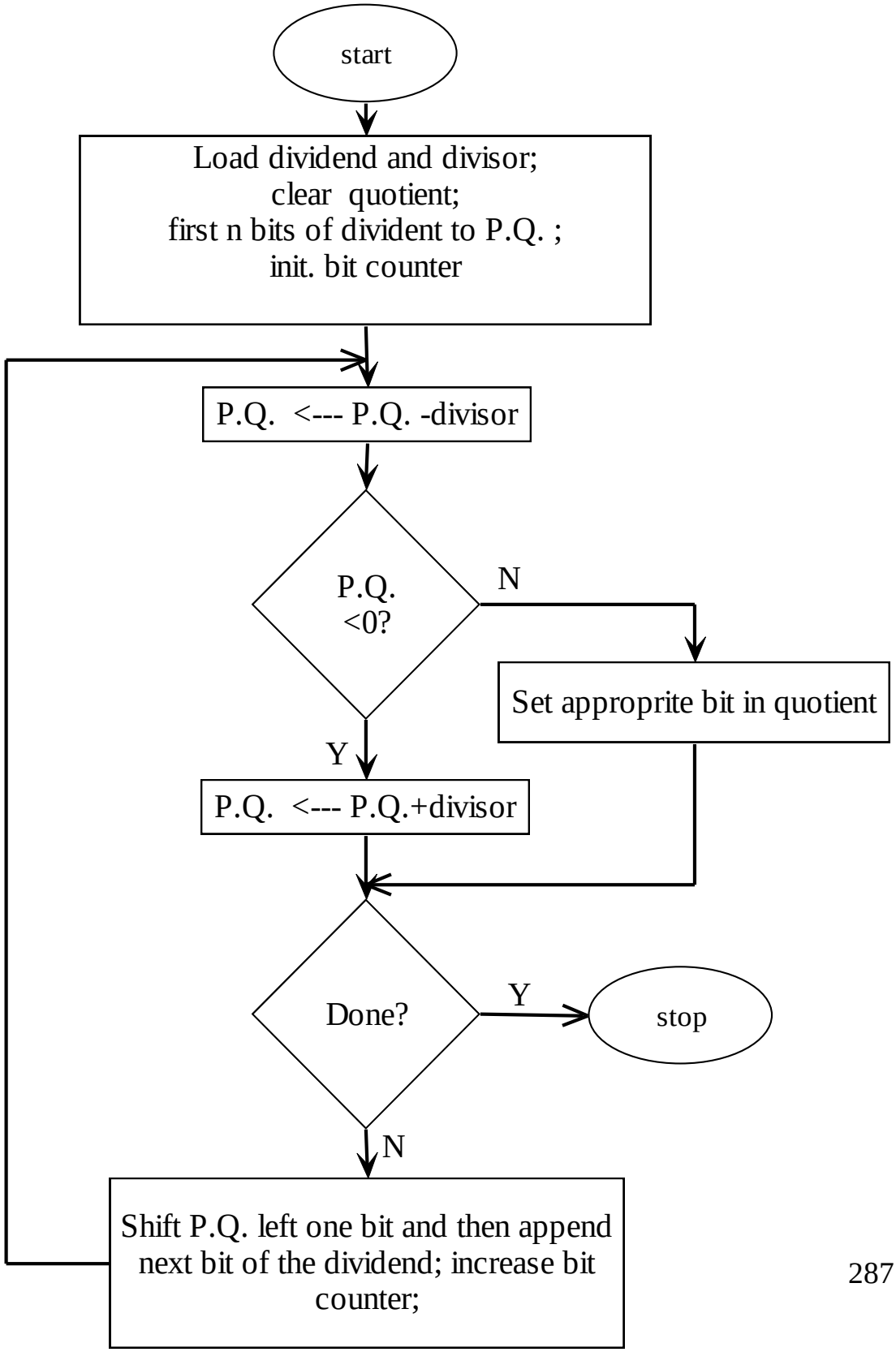



**01011 | 100010010**

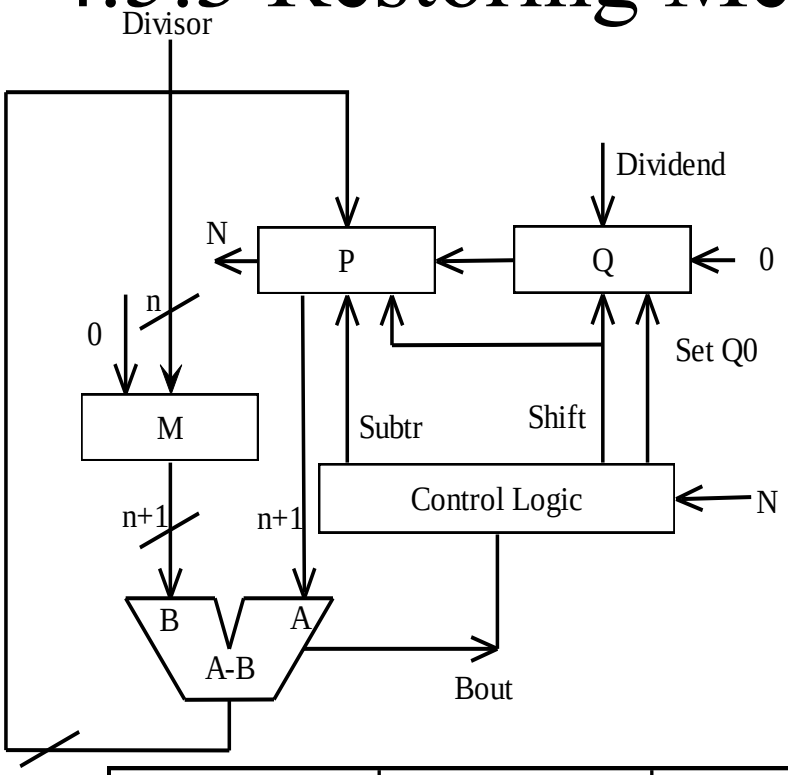
**M=01011**

| B | P | | | | | Q | | | | | | | | | operation |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | load M,Q, clear P |
| 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | shift P.Q |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | shift |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | shift |
| 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | shift |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | subtr.+set Q0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | shift |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | subtr. +set Q0 |
| 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | shift |
| 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | shift |
| 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | shift |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | |

# 4.5.3 Restoring Method for Division



# 4.5.3 Restoring Method cont'd

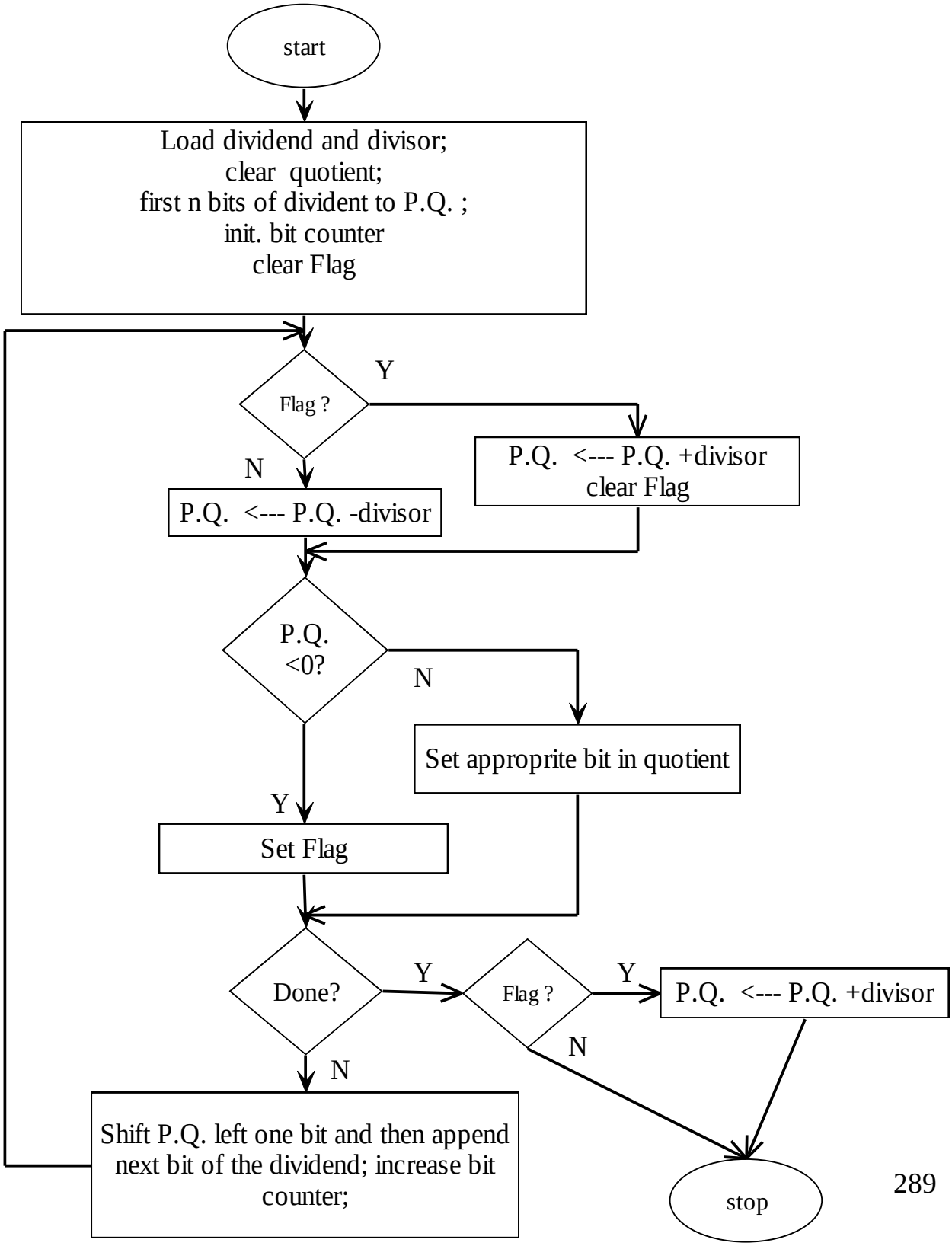



**0100 | 1001**

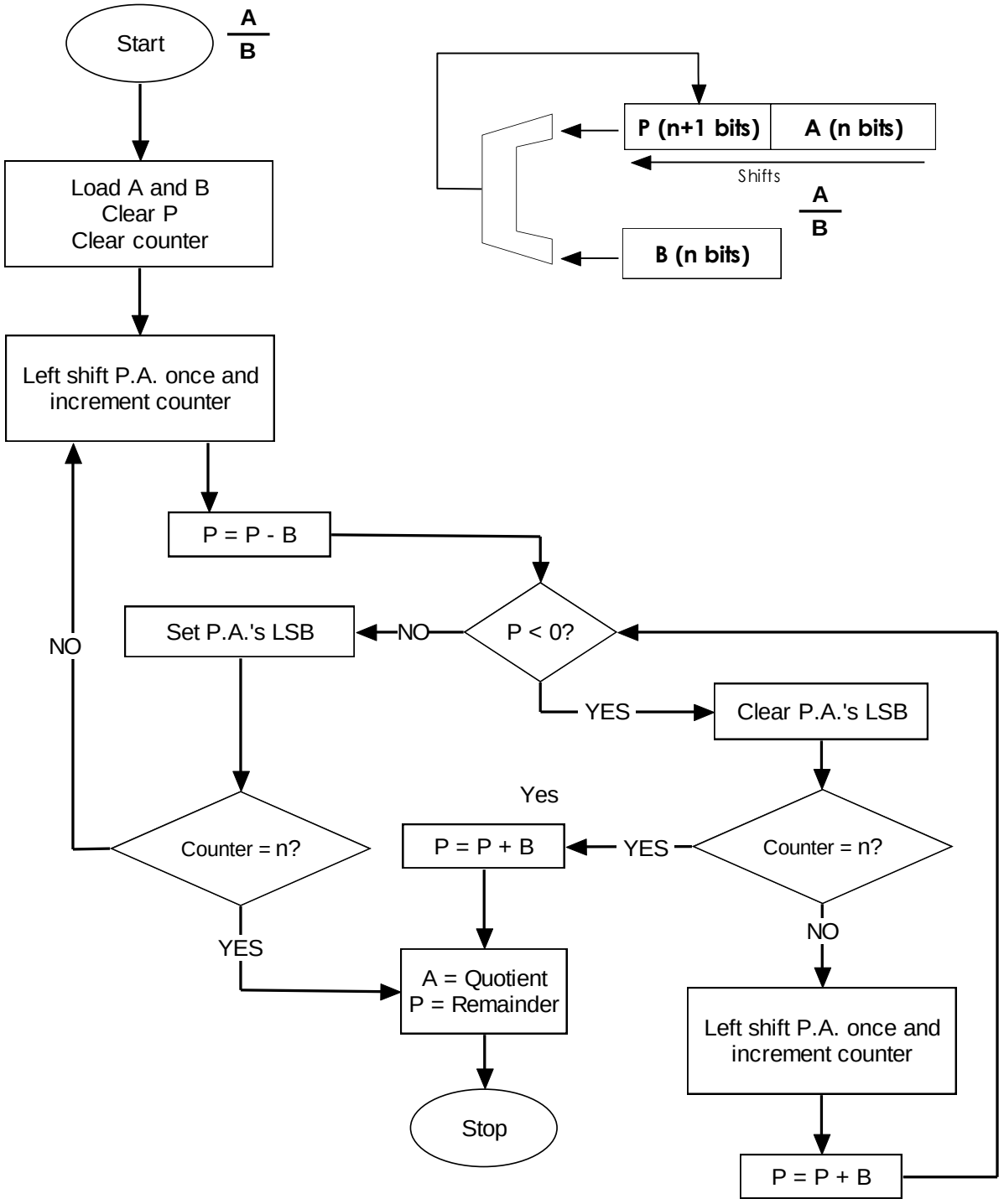
**M=0100**

| M | | | | | P | | | | | Q | | | | | operation |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | load M,Q, clear P |
| | | | | | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | shift left |
| | | | | | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | P <--P-M |
| | | | | | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | P<--P+M |
| | | | | | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | shift left |
| | | | | | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | P<--P-M |
| | | | | | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | P<--P+M |
| | | | | | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | shift left |
| | | | | | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | P<--P-M |
| | | | | | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | P<--P+M |
| | | | | | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | shift left |
| | | | | | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | P<--P-M |
| | | | | | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | shift left |
| | | | | | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | P<--P-M |
| | | | | | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | P<--P+M |

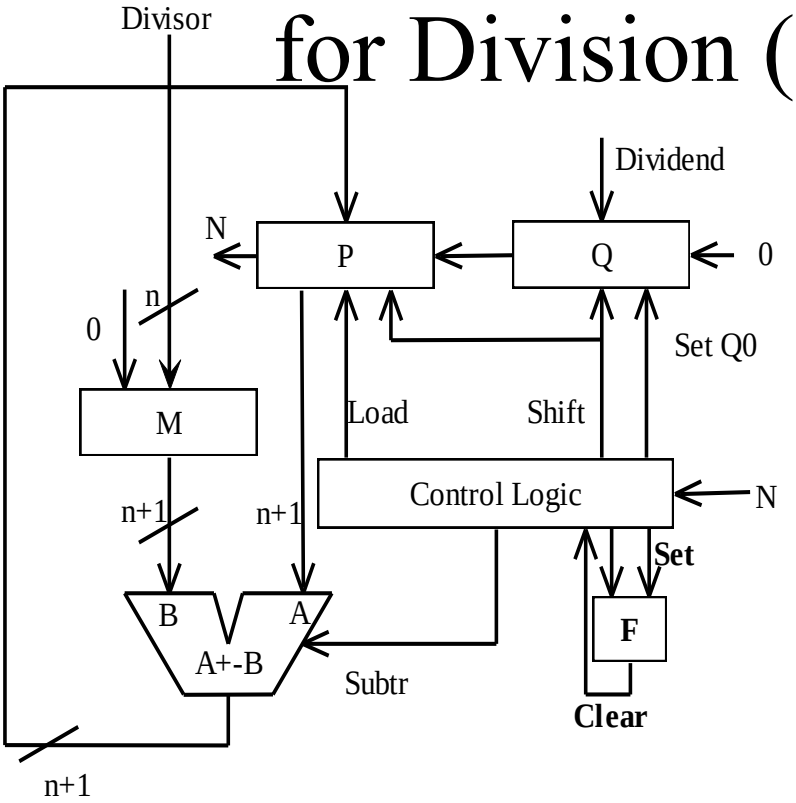
# 4.5.4 Non Restoring Method for Division



# Non-restoring Division Dataflow Diagram



# 4.5.4 Non Restoring Method for Division (cont'd)




**0100** $\overline{\phantom{x}}$ **1001**

**M=0100**

| M | | | | | F | P | | | | | | Q | | | | | operation |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | load M,Q, clear P |
| | | | | | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | | shift left |
| | | | | | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | | P=P-M |
| | | | | | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | | shift left |
| | | | | | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | | P=P+M |
| | | | | | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | | shift left |
| | | | | | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | | P=P+M |
| | | | | | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | | shift |
| | | | | | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | | P=P+M |
| | | | | | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | | shift |
| | | | | | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | | P=P+M* |
| | | | | | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | | P=P+M |

# 4.5.5 Possible Exception Conditions in Division

- Divide by zero
  - Undefined results
- Divide overflow
  - Divisor one word (n bit) , dividend two words (2n bit)
  - Possible results that the resulting quotient occupies n+1 bits when only n bits can be stored in a word
- Solution
  - Using software to avoid exceptions
  - Add special hardware to detect exceptions

# 4.6 Floating Point Arithmetic

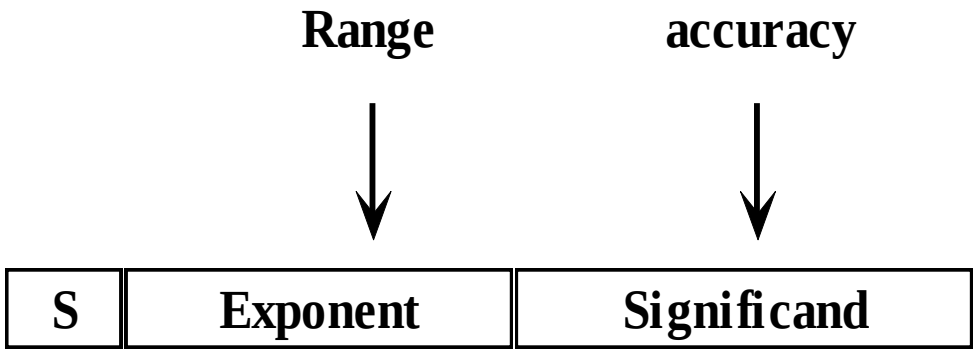
- Scientific Notation:
- Example : 123456789

- Normalized scientific notation
- 123456789

  - Same as scientific notation, except that there must be exactly one nonzero digit to the left of the radix point.
  - In normalized binary scientific notation, the bit the left of the radix point must be 1, or else the number is 0.

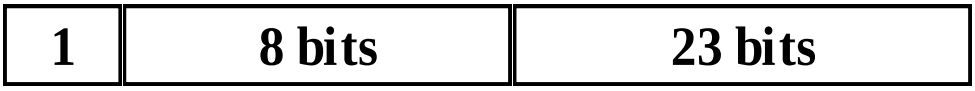
# 4.6.1 Hardware Representations of Scientific Notation

- Given: fixed word width
- Need to find out:
  - Number if bits in significant
  - Number of bits in exponent
  - Representation of negative numbers
- Problem:
  - Number of bits in exponent vs. number of bits in significand



# 4.6.1 IEEE 754 Floating Point Standard

- Single precision format (32 bits Wide)




- Double precision format: (64 Bit wide)


- Biased exponents? (logic and computer design fundamentals)
  - The exponent representation employed in most computers is known as a biased exponent. The bias is an excess number added to the exponent so that, internally, all exponents become positive.
- For single precision format offset=127
- For double precision format ?

# 4.6.1 IEEE 754 Floating Point Standard (cont'd)

- Example #1: Determine the IEEE 754 single precision representation of
- -4.125

| 1 | 10000001 | 00001 |
|---|---|---|

- 8.25

| | | |
|---|---|---|

- Example #2 determine what decimal number is represented by following bits interpreted as a single precision IEEE 754 Value

| 1 | 10110110 | 1100 |
|---|---|---|

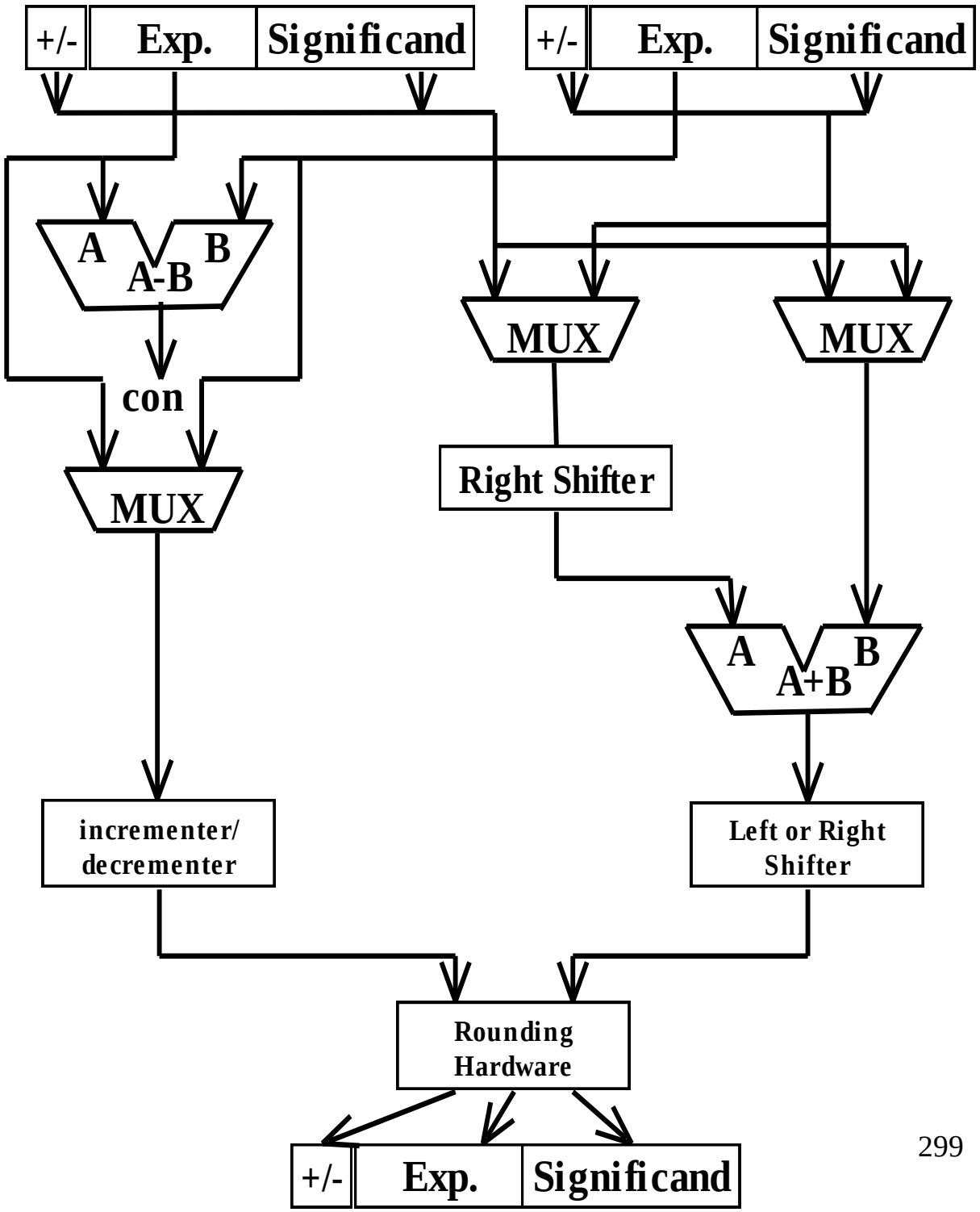
# 4.6.2 Floating Point Addition

I. Compare the exponents
    I. Shift the significant of the smaller number to the right to equalize the exponents
II. Add/subtract the significands
III. Normalize the results by either:
    I. Shifting right and incrementing exponent
    II. Shifting left and decrementing exponent
    III. Check for overflow or underflow
IV. Round the significand to the available number of bits
V. If results is not normalized, then repeat steps III and IV

# 4.6.2 Floating Point Addition(cont'd)

Example $0.625_{10}$-$0.4375_{10}$

# 4.6.2 Floating Point Addition Data Path



# 4.6.3 Floating Point Multiplication

- Add the exponents
  - Subtract the bias (if any) from the sum
- Multiply the significands
- If necessary, normalize the product resulting from step I and step 2
  - Either shift significand right and add to exponent
  - Or shift significand left and subtract from exponent
  - Check the overflow and underflow
- Round the number to the available number of digits
  - Repeat step III and step IV if not normalized
- Obtain the correct sign of the product from the signs of the original operands

# 4.6.3 Floating Point Multiplication (cont'd)

- Example 7.25*-6.125