# Computer Architecture and Operating System

## Computer Arithmetic

Prof. Indranil Sengupta

Department of Computer Science and Engineering

IIT Kharagpur

# Computer Arithmetic

## Introduction

- Computers are built using tiny electronic switches.
  - Typically made up of MOS transistors.
  - The state of the switches are typically expressed in binary (ON/OFF).
- To design arithmetic circuits for use in computers, we need to work with *binary numbers*.
  - How to carry out various arithmetic operations in binary?
  - How to implement them efficiently in hardware?



## Addition / Subtraction

# Addition of Multi-bit Binary Numbers

```
  0 0 1 0 1 1 0   <- Carry
  0 1 0 1 0 1 1   <- Number A
+ 0 0 0 1 0 0 1   <- Number B
-----------------
  0 1 1 0 1 0 0   <- Sum S
```

```
  1 1 1 1 1 1 0   <- Carry
  0 1 1 1 1 1 1   <- Number A
+ 0 0 0 0 0 0 1   <- Number B
-----------------
  1 0 0 0 0 0 0   <- Sum S
```

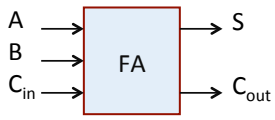
- At every bit position (stage), we require to add 3 bits:
  - 1 bit for number A
  - 1 bit for number B
  - 1 carry bit coming from the previous stage

**WE NEED A FULL ADDER**



## Full Adder

| Inputs | | | Outputs | |
|---|---|---|---|---|
| A | B | $C_{in}$ | S | $C_{out}$ |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |




$S = A'.B'.C_{in} + A'.B.C_{in}' + A.B'C_{in}' + A.B.C$

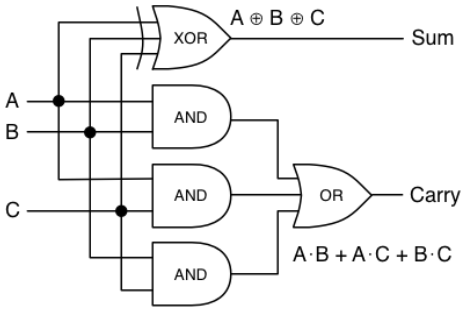
$= A \oplus B \oplus C_{in}$

$C_{out} = B.C_{in} + A.C_{in} + A.B + A.B.C_{in}$

$= A.B + B.C_{in} + A.C_{in}$

- **Delay of a full adder:**
  - Assume that the delay of all basic gates (AND, OR, NAND, NOR, NOT) is **δ**.
  - Delay for Carry = **2δ**
  - Delay for Sum = **3δ**

    (AND-OR delay plus one inverter delay)






## Parallel Adder Design

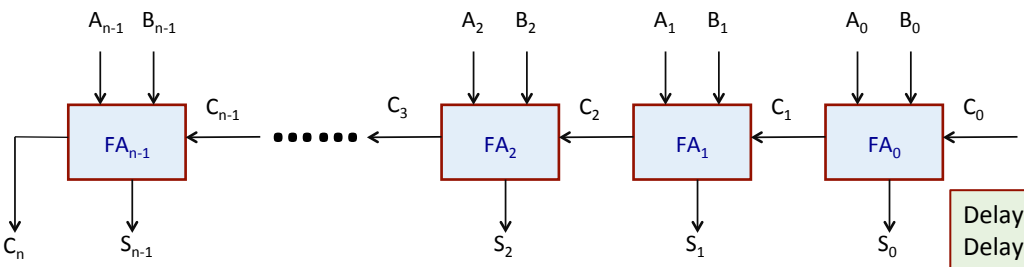
- We shall look at two designs of an **n**-bit parallel adder.
  a) Ripple carry adder
  b) Carry look-ahead adder

# Ripple Carry Adder

- Cascade *n* full adders to create a *n*-bit parallel adder.
- Carry output from stage-*i* propagates as the carry input to stage-(*i*+1).
- In the worst-case, carry ripples through all the stages.

```
  1 1 1 1 1 1 0   <- Carry
  0 1 1 1 1 1 1   <- Number A
+ 0 0 0 0 0 0 1   <- Number B
-----------------
  1 0 0 0 0 0 0   <- Sum S
```






Delay for $C_1$ = 2δ
Delay for $C_2$ = 4δ
Delay for $C_{n-1}$ = 2(n-1)δ
Delay for $C_n$ = 2nδ

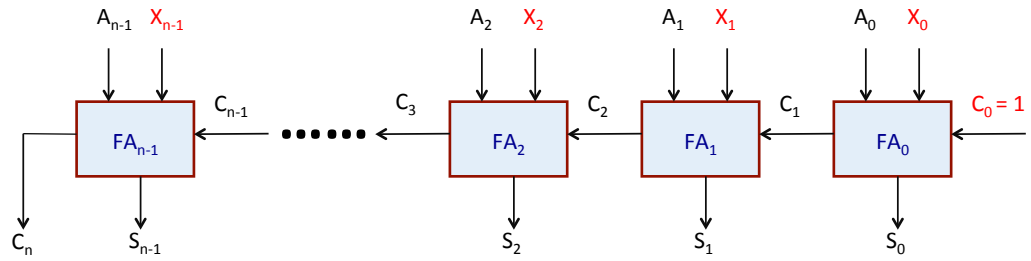
Two numbers: $A_{n-1}…A_2A_1A_0$ and $B_{n-1}…B_2B_1B_0$
Input carry: $C_0$
Sum: $S_{n-1}…S_2S_1S_0$
Output carry: $C_n$

Delay for $S_0$ = 3δ
Delay for $S_1$ = 2δ + 3δ = 5δ
Delay for $S_2$ = 4δ + 3δ = 7δ
Delay for $S_{n-1}$ = 2(n-1)δ + 3δ = (2n+1) δ

**Delay is proportional to n**

# How to Design a Parallel Subtractor?

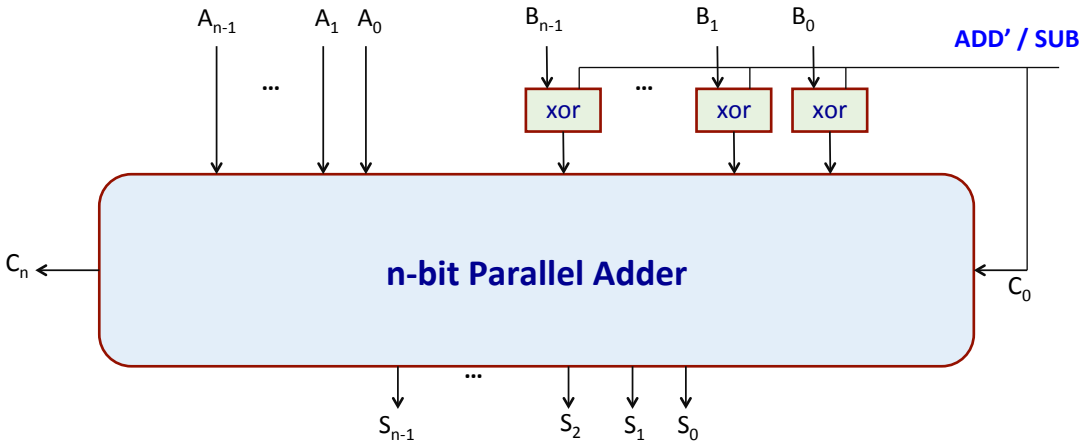
- **Observation:**
  - Computing *A - B* is the same as adding the 2's complement of *B* to *A*.
  - 2's complement is equal to 1's complement plus 1.
  - Let $X_i = B_i'$.






# A Parallel Adder/Subtractor

xor gate is a controlled inverter



## Carry Look-ahead Adder

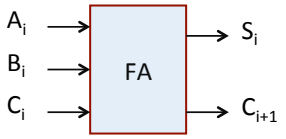
- The propagation delay of an *n*-bit ripple carry order has been seen to be proportional to *n*.
  - Due to the rippling effect of carry sequentially from one stage to the next.
- One possible way to speedup the addition.
  - Generate the carry signals for the various stages in parallel.
  - Time complexity reduces from O(*n*) to O(1).
  - Hardware complexity increases rapidly with *n*.



- Consider the i-th stage in the addition process.
- We define the *carry generate* and *carry propagate* functions as:

  $G_i = A_i \,.\, B_i$

  $P_i = A_i \oplus B_i$
- $G_i = 1$ represents the condition when a carry is generated in stage-i independent of the other stages.
- $P_i = 1$ represents the condition when an input carry $C_i$ will be propagated to the output carry $C_{i+1}$.




$$C_{i+1} = G_i + P_i.C_i$$

# Design of 4-bit CLA Adder

$C_4 = G_3 + G_2P_3 + G_1P_2P_3 + G_0P_1P_2P_3 + C_0P_0P_1P_2P_3$

$C_3 = G_2 + G_1P_2 + G_0P_1P_2 + C_0P_0P_1P_2$

$C_2 = G_1 + G_0P_1 + C_0P_0P_1$

$C_1 = G_0 + C_0P_0$

$S_0 = A_0 \oplus B_0 \oplus C_0 = P_0 \oplus C_0$

$S_1 = P_1 \oplus C_1$

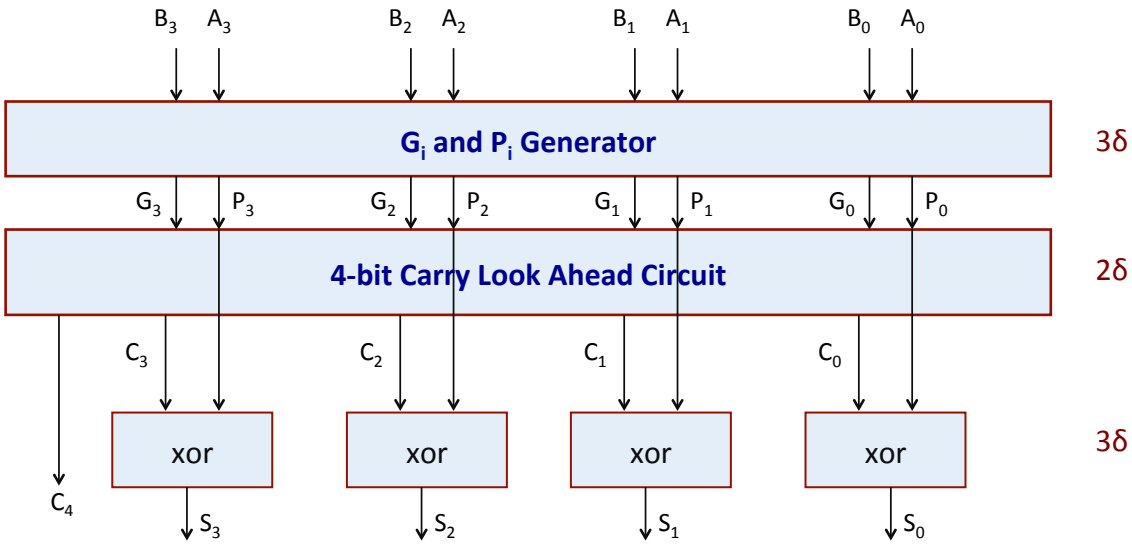
$S_2 = P_2 \oplus C_2$

$S_3 = P_3 \oplus C_3$

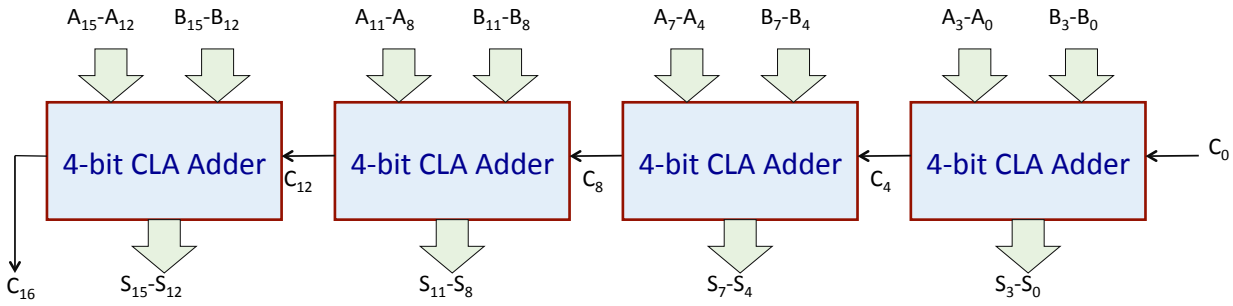
4 AND2 gates
3 AND3 gates
2 AND4 gates
1 AND5 gate
1 OR2, 1 OR3, 1 OR4 and
1 OR5 gate

4 XOR2 gates





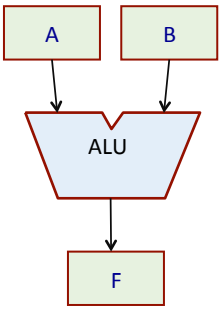
## 16-bit Adder Using 4-bit CLA Modules



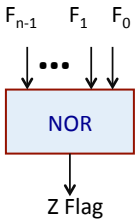


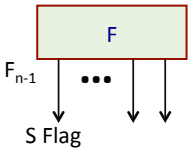

## Generating the Status Flags

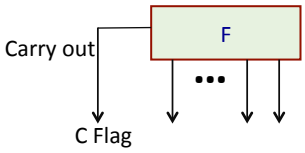
- Many contemporary processors have a flag register that contains the status of the last arithmetic / logic operation.
  - **Zero (Z)**: tells whether the result is zero.
    - Can be used for both arithmetic and logic operations.
  - **Sign (S)**: tells whether the result is positive (=0) or negative (=1).
    - Can be used for both arithmetic and logic operations.
  - **Carry (C)**: tells whether there has been a carry out of the most significant stage.
    - Used only for arithmetic operations.
  - **Overflow (V)**: tells whether the result is too large to fit in the target register.
    - Used only for arithmetic operations (addition and subtraction).




Assume A, B and F are n-bit registers




- Overflow can occur during addition when the sign of the two operands are the same.
  - Sign of the result becomes different from the sign of the operand(s).

$V = A_{n-1}.B_{n-1}.F_{n-1}' + A_{n-1}'.B_{n-1}'.F_{n-1}$

$V = F_{n-1} \oplus Carry\_out$



# Multiplication

## Multiplication of Unsigned Numbers

- Multiplication requires substantially more hardware than addition.
- Multiplication of two *n*-bit number generates a 2*n*-bit product.
- We can use shift-and-add method.
  - Repeated additions of shifted versions of the multiplicand.

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

**Multiplicand M (10)**
**Multiplier Q (13)**
**Product P (130)**

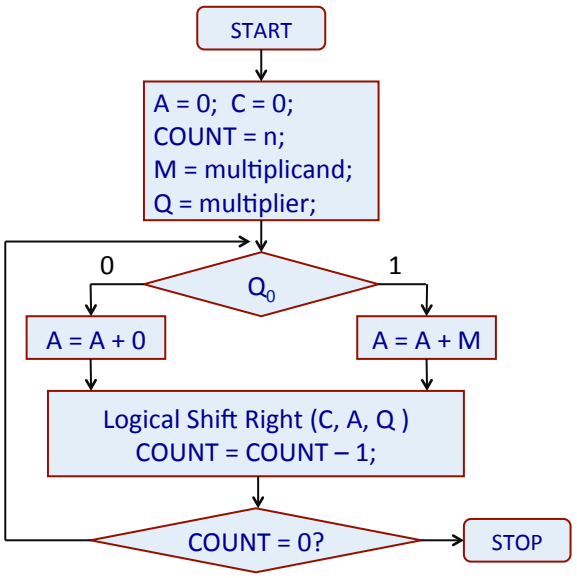


## Unsigned Sequential Multiplication

- Requires much less hardware, but requires several clock cycles to perform multiplication of two *n*-bit numbers.
  - Typical hardware complexity: O(*n*).
  - Typical time complexity: O(*n*).
- In the "*hand multiplication*" that we have seen:
  - If the *i*-th bit of the multiplier is 1, the multiplicand is shifted left by *i* bit positions, and added to the partial product.
  - The relative position of the partial products do not change; it is the multiplicand that gets shifted left.

- In the "*shift-and-add*" multiplication that we discuss now, we make the following modifications.
  - We do not shift the multiplicand (i.e., keep its position fixed).
  - We right shift an 2*n*-bit partial product at every step.






M: n-bit multiplicand

Q: n-bit multiplier

A: n-bit temporary register

C: 1-bit carry out from adder

**Example 1**: (10) x (13)

Assume 5-bit numbers.

M: $(0\ 1\ 0\ 1\ 0)_2$
Q: $(0\ 1\ 1\ 0\ 1)_2$

Product = 130
= $(0\ 0\ 1\ 0\ 0\ 0\ 0\ 1\ 0)_2$

| | C | A | Q | | |
|---|---|---|---|---|---|
| | 0 | 0 0 0 0 0 | 0 1 1 0 (1) | Initialization | |
| Step 1 | 0 | 0 1 0 1 0 | 0 1 1 0 1 | A = A + M | |
| | 0 | 0 0 1 0 1 | 0 0 1 1 (0) | Shift | |
| Step 2 | 0 | 0 0 1 0 1 | 0 0 1 1 0 | A = A + 0 | |
| | 0 | 0 0 0 1 0 | 1 0 0 1 (1) | Shift | |
| Step 3 | 0 | 0 1 1 0 0 | 1 0 0 1 1 | A = A + M | |
| | 0 | 0 0 1 1 0 | 0 1 0 0 (1) | Shift | |
| Step 4 | 0 | 1 0 0 0 0 | 0 1 0 0 1 | A = A + M | |
| | 0 | 0 1 0 0 0 | 0 0 1 0 (0) | Shift | |
| Step 5 | 0 | 0 1 0 0 0 | 0 0 1 0 0 | A = A + 0 | |
| | 0 | 0 0 1 0 0 | 0 0 0 1 0 | Shift | |



**Example 2**: (29) x (21)

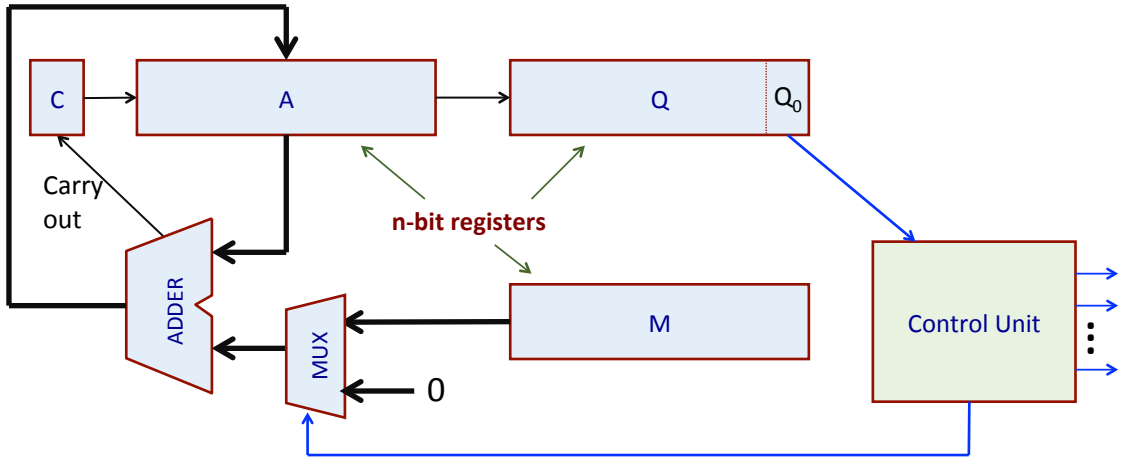
Assume 5-bit numbers.

M: $(1\ 1\ 1\ 0\ 1)_2$
Q: $(1\ 0\ 1\ 0\ 1)_2$

Product = 609
= $(1\ 0\ 0\ 1\ 1\ 0\ 0\ 0\ 1)_2$

| | C | A | Q | | |
|---|---|---|---|---|---|
| | 0 | 0 0 0 0 0 | 1 0 1 0 (1) | Initialization | |
| Step 1 | 0 | 1 1 1 0 1 | 1 0 1 0 1 | A = A + M | |
| | 0 | 0 1 1 1 0 | 1 1 0 1 (0) | Shift | |
| Step 2 | 0 | 0 1 1 1 0 | 1 1 0 1 0 | A = A + 0 | |
| | 0 | 0 0 1 1 1 | 0 1 1 0 (1) | Shift | |
| Step 3 | 1 | 0 0 1 0 0 | 0 1 1 0 1 | A = A + M | |
| | 0 | 1 0 0 1 0 | 0 0 1 1 (0) | Shift | |
| Step 4 | 0 | 1 0 0 1 0 | 0 0 1 1 0 | A = A + 0 | |
| | 0 | 0 1 0 0 1 | 0 0 0 1 (1) | Shift | |
| Step 5 | 1 | 0 0 1 1 0 | 0 0 0 1 1 | A = A + M | |
| | 0 | 1 0 0 1 1 | 0 0 0 0 1 | Shift | |

## Data Path for Shift-and-Add Multiplier






## Signed Multiplication

- We can extend the basic shift-and-add multiplication method to handle signed numbers.
- One important difference:
  - Require to sign-extend all the partial products before they are added.
  - Recall that for 2's complement representation, sign extension can be done by replicating the sign bit any number of times.

| |
|---|
| 0101 = 0000 0101 = 0000 0000 0000 0101 = 0000 0000 0000 0000 0000 0000 0000 0101 |
| 1011 = 1111 1011 = 1111 1111 1111 1011 = 1111 1111 1111 1111 1111 1111 1111 1011 |

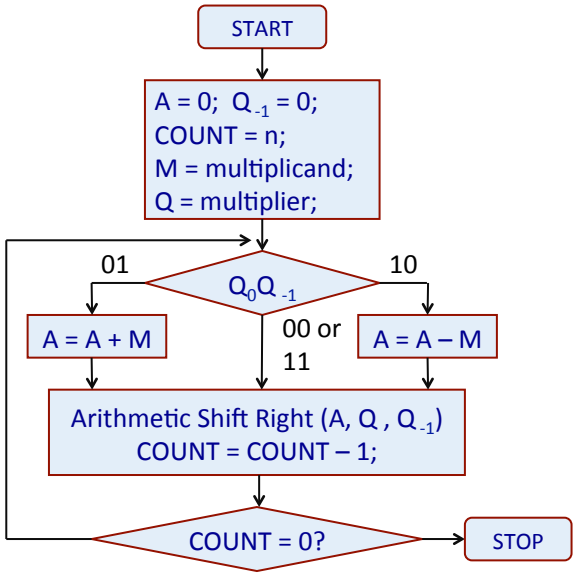
## Booth's Algorithm for Signed Multiplication

- In the conventional shift-and-add multiplication as discussed, for *n*-bit multiplication, we iterate *n* times.
  - Add either 0 or the multiplicand to the 2*n*-bit partial product (depending on the next bit of the multiplier).
  - Shift the 2*n*-bit partial product to the right.
- Essentially we need *n additions and n shift operations*.
- Booth's algorithm is an improvement whereby we can avoid the additions whenever consecutive 0's or 1's are detected in the multiplier.
  - Makes the process faster.



## Basic Idea Behind Booth's Algorithm

- We inspect two bits of the multiplier ($Q_i$, $Q_{i-1}$) at a time.
  - If the bits are same (00 or 11), we only shift the partial product.
  - If the bits are 01, we do an addition and then shift.
  - If the bits are 10, we do a subtraction and then shift.
- Significantly reduces the number of additions / subtractions.
  - For encoding the least significant bit $Q_0$, we assume $Q_{-1} = 0$.




M: n-bit multiplicand

Q: n-bit multiplier

A: n-bit temporary register

$Q_{-1}$: 1-bit flip-flop

**Skips over consecutive 0's and 1's of the multiplier Q.**



**Example 1**: (-10) x (13)

Assume 5-bit numbers.

M: $(1\ 0\ 1\ 1\ 0)_2$
-M: $(0\ 1\ 0\ 1\ 0)_2$
Q: $(0\ 1\ 1\ 0\ 1)_2$

Product = -130
= $(1\ 1\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 0)_2$

| A | Q | $Q_{-1}$ | | |
|---|---|---|---|---|
| 0 0 0 0 0 | 0 1 1 0 | 1 | 0 | Initialization | |
| 0 1 0 1 0 | 0 1 1 0 | 1 | 0 | A = A - M | Step 1 |
| 0 0 1 0 1 | 0 0 1 1 | 0 | 1 | Shift | |
| 1 1 0 1 1 | 0 0 1 1 | 0 | 1 | A = A + M | Step 2 |
| 1 1 1 0 1 | 1 0 0 1 | 1 | 0 | Shift | |
| 0 0 1 1 1 | 1 0 0 1 | 1 | 0 | A = A - M | Step 3 |
| 0 0 0 1 1 | 1 1 0 0 | 1 | 1 | Shift | |
| 0 0 0 0 1 | 1 1 1 1 | 0 | 1 | Shift | Step 4 |
| 1 0 1 1 1 | 1 1 1 0 | 0 | 1 | A = A + M | Step 5 |
| 1 1 0 1 1 | 1 1 1 1 | 0 | 0 | Shift | |

**Example 2**:

(-31) x (28)

Assume 6-bit numbers.

M: $(1\,0\,0\,0\,0\,1)_2$
-M: $(0\,1\,1\,1\,1\,1)_2$
Q: $(0\,1\,1\,1\,0\,0)_2$

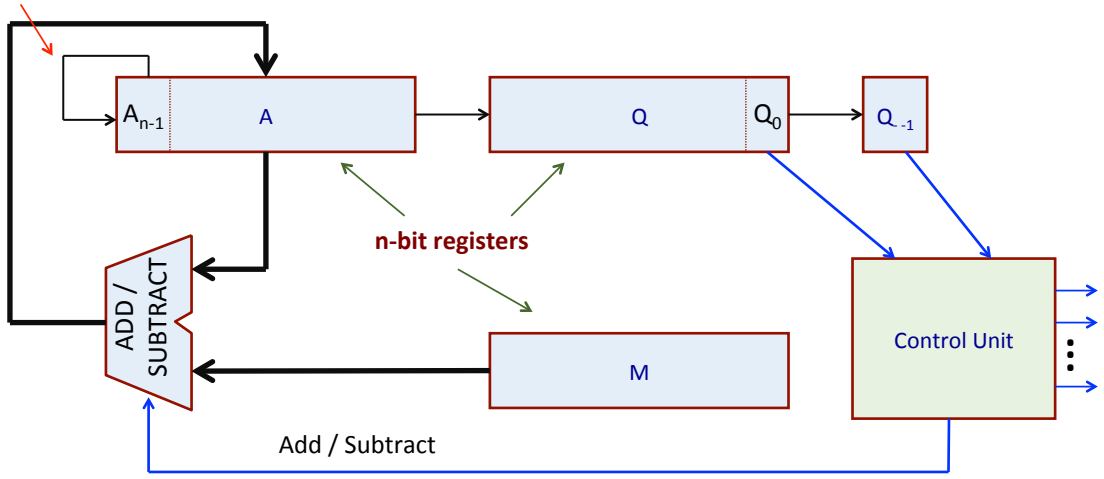
Product = -868
= $(1\,1\,0\,0\,1\,0\;\; 0\,1\,1\,1\,0\,0)_2$

| A | Q | $Q_{-1}$ | | |
|---|---|---|---|---|
| 0 0 0 0 0 0 | 0 1 1 1 0 | 0 | Initialization | |
| 0 0 0 0 0 0 | 0 0 1 1 1 | 0 | Shift | Step 1 |
| 0 0 0 0 0 0 | 0 0 0 1 1 | 1 | Shift | Step 2 |
| 0 1 1 1 1 1 | 0 0 0 1 1 | 1 | A = A - M | Step 3 |
| 0 0 1 1 1 1 | 1 0 0 0 1 | 1 | Shift | |
| 0 0 0 1 1 1 | 1 1 0 0 0 | 1 | Shift | Step 4 |
| 0 0 0 0 1 1 | 1 1 1 0 0 | 0 | Shift | Step 5 |
| 1 0 0 1 0 0 | 1 1 1 0 0 0 | 1 | A = A + M | Step 6 |
| 1 1 0 0 1 0 | 0 1 1 1 0 0 | 0 | Shift | |



Arithmetic shift right

## Data Path for Booth's Algorithm



# Division

## The Process of Integer Division

- In integer division, a *divisor* **M** and a *dividend* **D** are given.
- The objective is to find a third number **Q**, called the *quotient*, such that

  **D = Q x M + R**

  where **R** is the *remainder* such that **0 ≤ R < M**.
- The relationship **D = Q x M** suggests that there is a close correspondence between division and multiplication.
  - Dividend, quotient and divisor correspond to product, multiplicand and multiplier, respectively.
  - Similar algorithms and circuits can be used for multiplication and division.

- One of the simplest division methods is the sequential digit-by-digit algorithm similar to that used in pencil-and-paper methods.

```
                    0 1 1 0        Quotient Q = Q0Q1Q2Q3
Divisor M    1 1 0 | 1 0 0 1 0 1    Dividend D = R0
                    1 1 0          Q0.M      (Does not go; Q0 = 0)
                    -------------
                    1 0 0 1 0 1    R1
                -   1 1 0          Q1.2^-1.M (Does go; Q1 = 1)
                    -------------
                      0 1 1 0 1    R2
                -     1 1 0 Q2.2^-2.M        (Does go; Q2 = 1)
                    -------------
                        0 0 0 1    R3
                          1 1 0    Q3.2^-3.M (Does not go; Q3 = 0)
                    -------------
                          0 0 1    R4 = Remainder R
```

**$D = 37 = (1\ 0\ 0\ 1\ 0\ 1)_2$**
**$M = 6 = (1\ 1\ 0)_2$**
**Quotient Q = 6**
**Remainder R = 1**



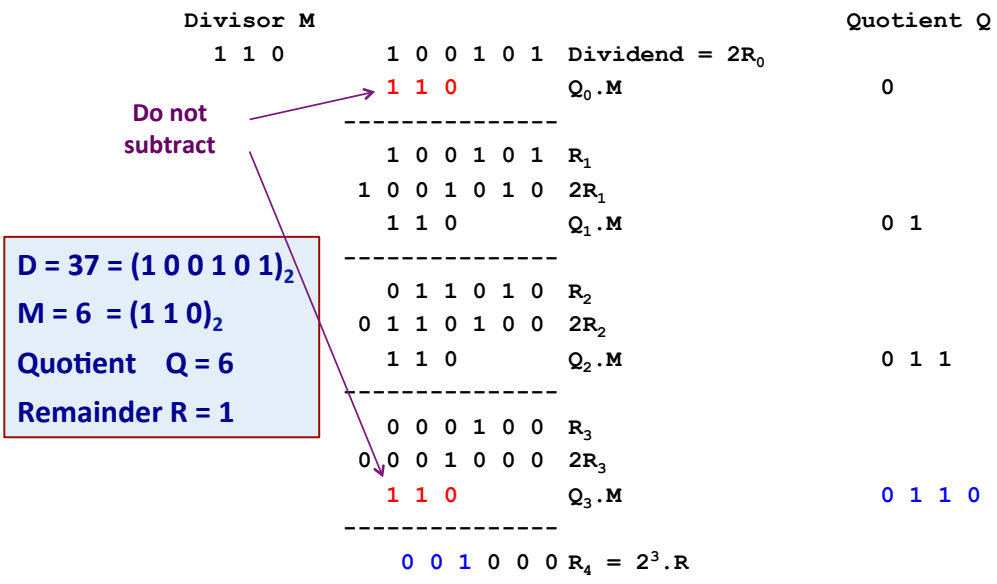
- In the example, the quotient $Q = Q_0Q_1Q_2\ldots$ is computed one bit at a time.
  - At each step *i*, the divisor shifted *i* bits to the right (i.e. $2^{-i}.M$) is compared with the current partial remainder $R_i$.
  - The quotient bit $Q_i$ is set to 0 (1) if $2^{-i}.M$ is greater than (less than) $R_i$ .
  - The new partial remainder $R_{i+1}$ is computed as:

$$R_{i+1} = R_i - Q_i.2^{-i}.M$$

- **Machine implementation:**
  - For hardware implementation, it is more convenient to shift the partial remainder to the left relative to a fixed divisor; thus

    $R_{i+1} = 2R_i - Q_i.M$ *(instead of* $R_{i+1} = R_i - Q_i.2^{-i}.M$*)*
  - The final partial remainder is the required remainder shifted to the left, so that $R = 2^{-3}.R_4$ (see next slide).



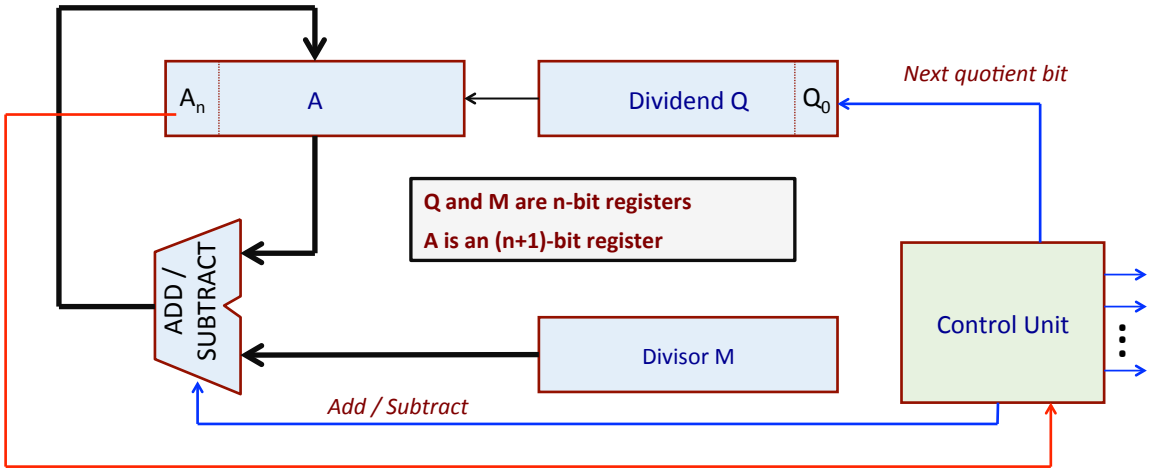


## Two alternative approaches

- We shall discuss two approaches:
  a) Restoring division
  b) Non-restoring division



## (a) Restoring Division: The Data Path



# Basic Steps (Restoring Division)

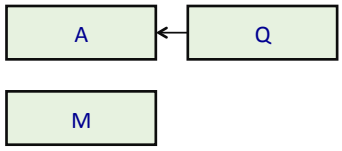
Repeat the following steps *n* times:

a) Shift the dividend left one bit at a time into register *A*.

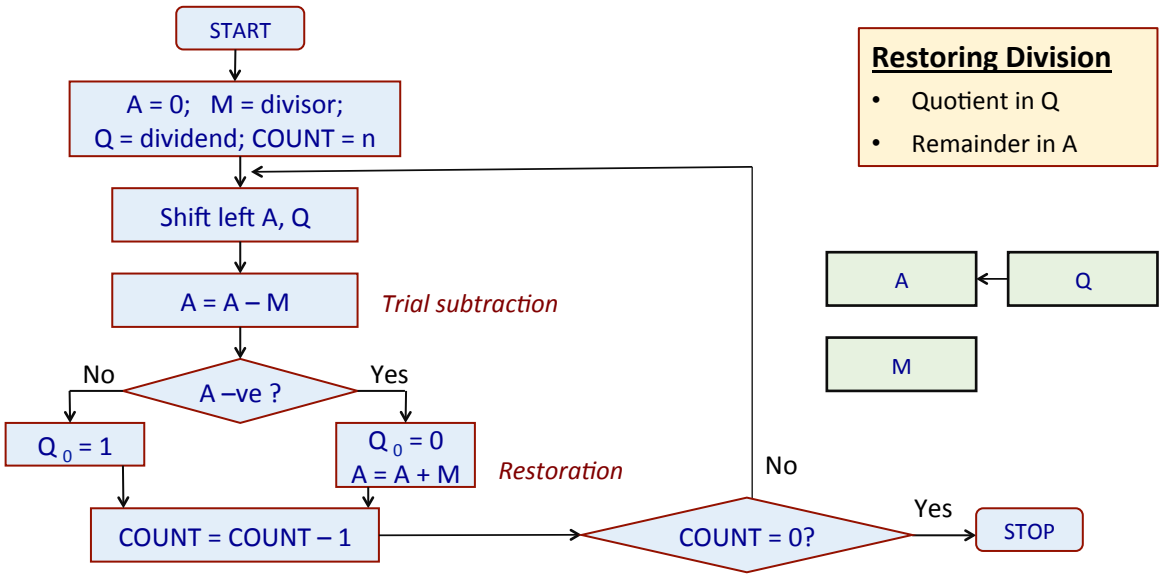
b) Subtract the divisor *M* from this register *A* (*trial subtraction*).

c) If the result is negative (*i.e. not going*):
- Add the divisor *M* back into the register *A* (*i.e. restoring back*).
- Record 0 as the next quotient bit.

d) If the result is positive:
- Do not restore the intermediate result.
- Record 1 as the next quotient bit.





- **Analysis**:
  - For *n*-bit divisor and *n*-bit dividend, we iterate *n* times.
  - Number of trial subtractions: *n*
  - Number of restoring additions: *n/2* on the average
    - Best case: *0*
    - Worst case: *n*



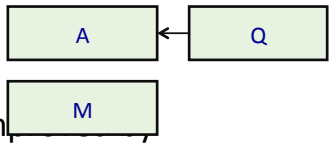
# A Simple Example: 8/3 for 4-bit representation (n=4)

| | | |
|---|---|---|
| Initially: | 0 0 0 0 0 | 1 0 0 0 |
| Shift: | 0 0 0 0 1 | 0 0 0 – |
| Subtract: | 0 0 1 1 | |
| Set $Q_0$: | (1) 1 1 1 0 | |
| Restore: | 0 0 1 1 | |
| | 0 0 0 0 1 | 0 0 0 (0) |
| Shift: | 0 0 0 1 0 | 0 0 0 – |
| Subtract: | 0 0 1 1 | |
| Set $Q_0$: | (1) 1 1 1 1 | |
| Restore: | 0 0 1 1 | |
| | 0 0 0 1 0 | 0 0 0 (0) |

| | | |
|---|---|---|
| Shift: | 0 0 1 0 0 | 0 0 0 – |
| Subtract: | 0 0 1 1 | |
| Set $Q_0$: | (0) 0 0 0 1 | |
| | 0 0 0 0 0 | 0 0 0 (1) |
| Shift: | 0 0 0 1 0 | 0 0 1 – |
| Subtract: | 0 0 1 1 | |
| Set $Q_0$: | (1) 1 1 1 1 | |
| Restore: | 0 0 1 1 | |
| | 0 0 0 1 0 | 0 0 1 (0) |

**Remainder 00010 = 2** **Quotient 0010 = 2**

## (b) Non-Restoring Division

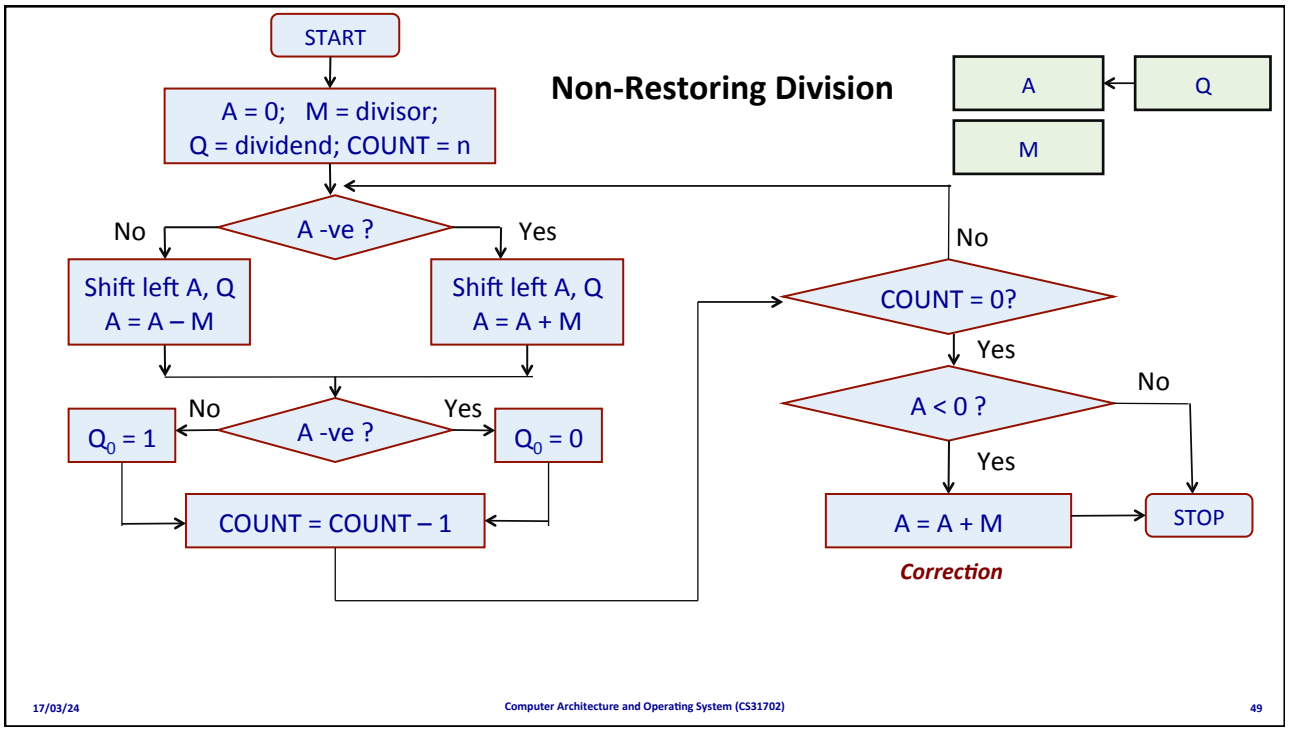



- The performance of restoring division algorithm can be im[illegible] exploiting the following observation.
- In restoring division, what we do actually is:
  - If *A* is positive, we shift it left and subtract *M*.
    - That is, we compute 2A – M.
  - If *A* is negative, we restore is by doing A+M, shift it left, and then subtract *M*.
    - That is, we compute 2(A + M) – M = 2A + M.
- We can accordingly modify the basic division algorithm by eliminating the restoring step. → ***NON-RESTORING DIVISION***

Shift left means multiplying by 2.



- **Basic steps in non-restoring division:**
  a) Start by initializing register *A* to 0, and repeat steps (b)-(d) *n* times.
  b) If the value in register *A* is positive,
     - Shift *A* and *Q* left by one bit position.
     - Subtract *M* from *A*.
  c) If the value in register *A* is negative,
     - Shift *A* and *Q* left by one bit position.
     - Add *M* to *A*.
  d) If *A* is positive, set $Q_0 = 1$; else, set $Q_0 = 0$.
  e) If *A* is negative, add *M* to *A* as a final corrective step.






# A Simple Example: 8/3 for n=4

```
Initially:    0 0 0 0 0    1 0 0 0

Shift:        0 0 0 0 1    0 0 0 –
Subtract:   - 0 0 1 1
Set Q0:       (1) 1 1 1 0  0 0 0 (0)
Shift:        1 1 1 0 0    0 0 0 –
Add:          0 0 1 1
Set Q0:       (1) 1 1 1 1  0 0 0 (0)
Shift:        1 1 1 1 0    0 0 0 –
Add:          0 0 1 1
Set Q0:       (0) 0 0 0 1  0 0 0 (1)
```

```
Shift:        0 0 0 1 0    0 0 1 –
Subtract:   - 0 0 1 1
Set Q0:       (1) 1 1 1 1  0 0 1 (0)
Correction Add:
              1 1 1 1 1
              0 0 0 1 1
              0 0 0 1 0
```

Quotient
0010 = 2

Remainder
00010 = 2

## Data Path for Non-Restoring Division