# COMPUTER ORGANIZATION AND ASSEMBLY LANGUAGE

Computer Arithmetic

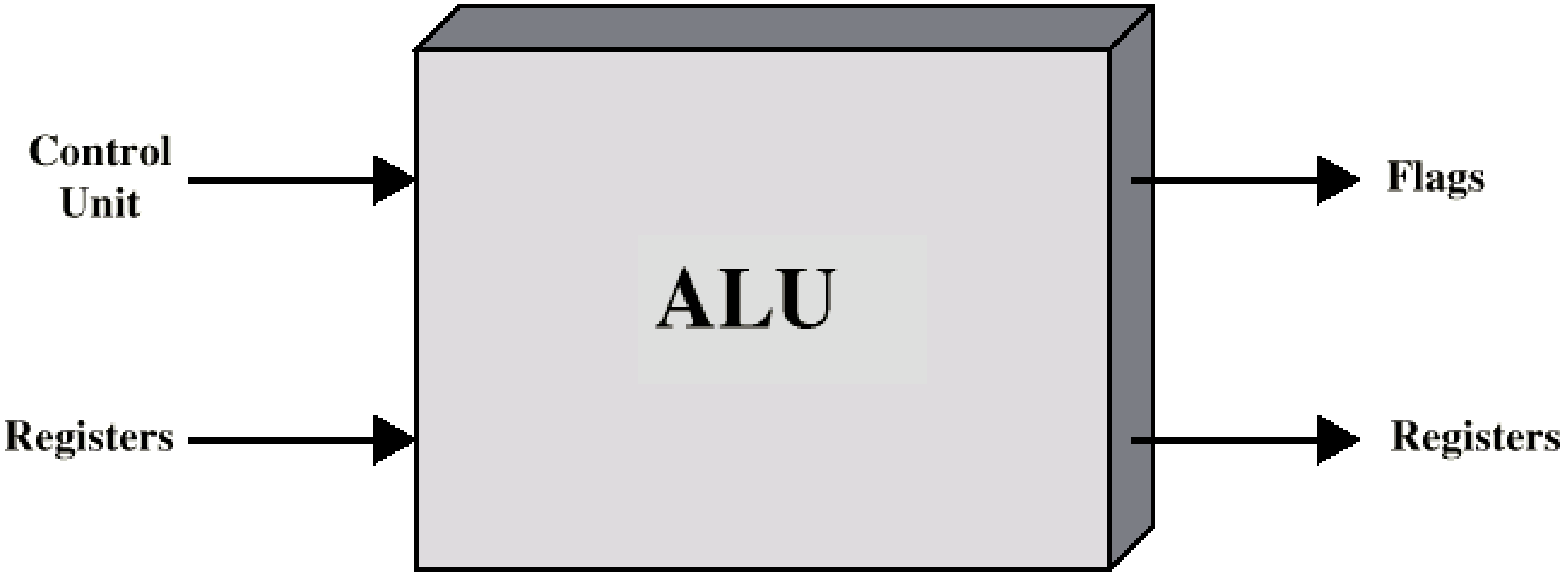
Integer Representation, Integer Arithmetic

Course Instructor: Klarence M. Baptista, MIT

# Arithmetic & Logic Unit

- Does the calculations
- Everything else in the computer is there to service this unit
- Handles integers
- May handle floating point (real) numbers
- May be separate FPU (maths co-processor)
- May be on chip separate FPU (486DX +)

# ALU Inputs and Outputs



# Integer Representation

- Only have 0 & 1 to represent everything
- Positive numbers stored in binary
  - e.g. 41=00101001
- No minus sign
- No period
- Sign-Magnitude
- Two's compliment

# Sign-Magnitude

- The simplest form of representation that employs a sign bit is the sign-magnitude representation
- In an *n*-bit word, the rightmost *n-1* bits hold the magnitude of the integer
- +18 = 00010010
- -18 = 10010010

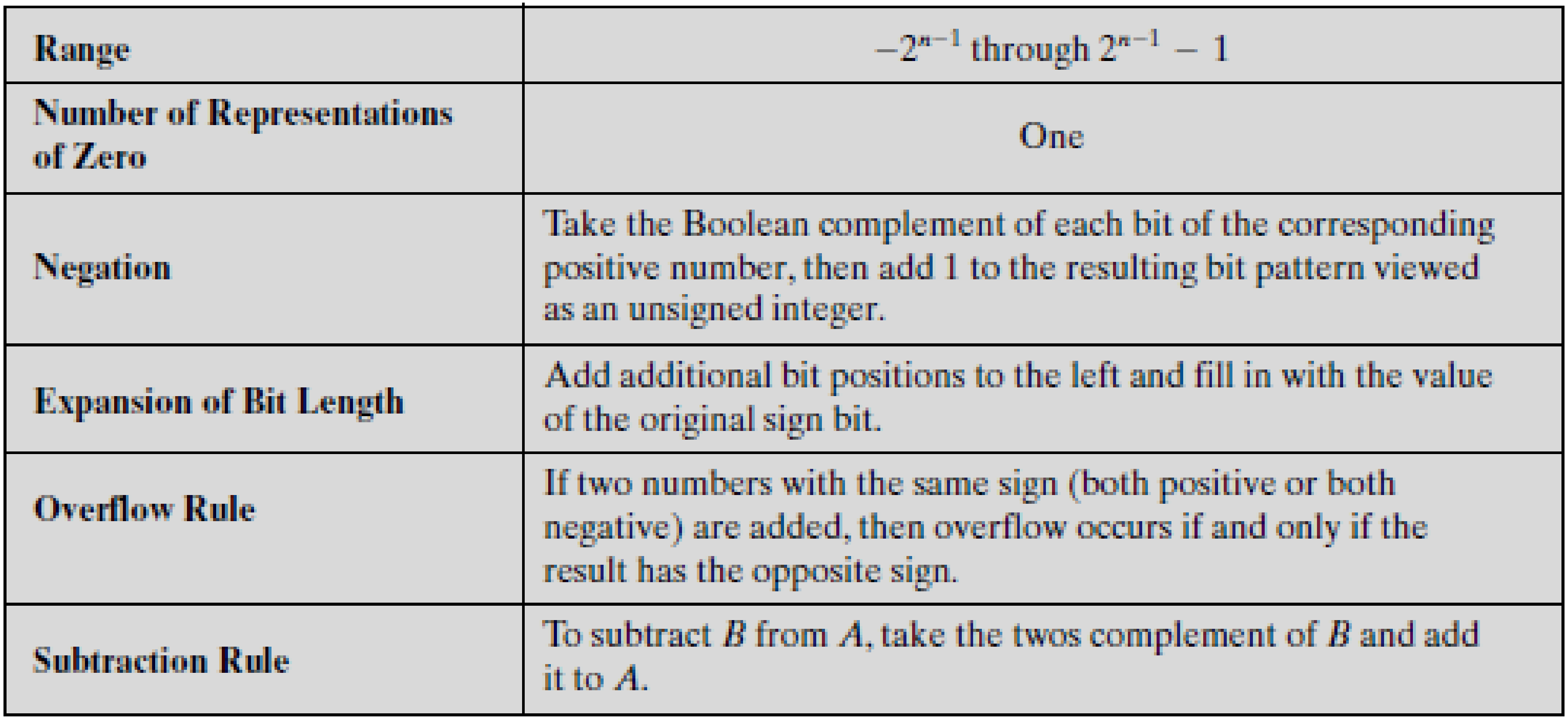
# Sign-Magnitude

- Problems
  - Need to consider both sign and magnitude in arithmetic
  - Two representations of zero (+0 and -0)
- Inconvenient because it is slightly more difficult to test for 0 (an operation performed frequently on computers) than if there were a single representation
- Sign-magnitude representation is rarely used in implementing the integer portion of the ALU
- Instead, the most common scheme is twos complement representation

# Two's Compliment

- Like sign magnitude, twos complement representation uses the most significant bit as a sign bit, making it easy to test whether an integer is positive or negative
- It differs from the use of the sign-magnitude representation in the way that the other bits are interpreted

# Characteristics of Twos Complement Representation and Arithmetic

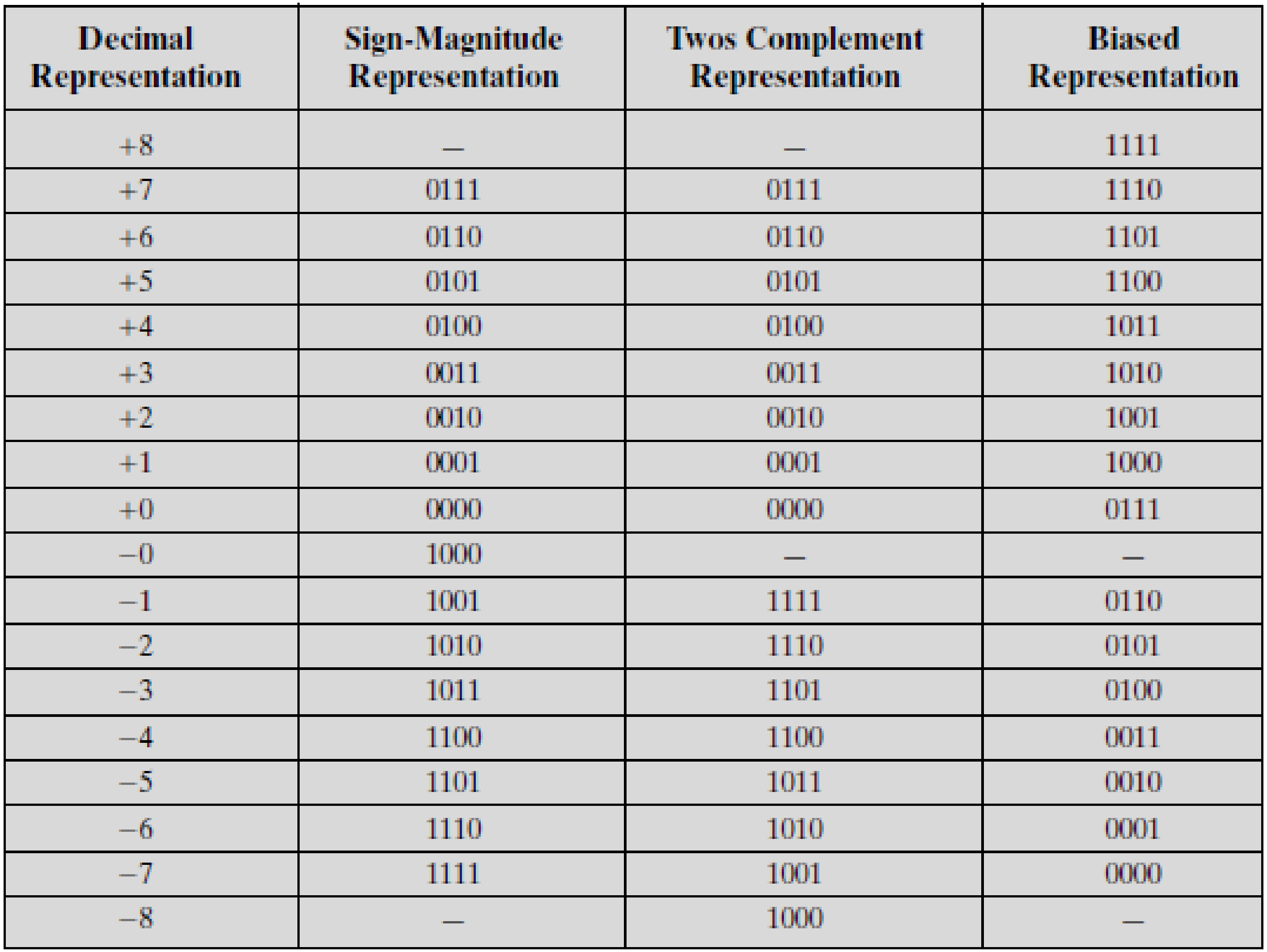


| | |
|---|---|
| **Range** | $-2^{n-1}$ through $2^{n-1} - 1$ |
| **Number of Representations of Zero** | One |
| **Negation** | Take the Boolean complement of each bit of the corresponding positive number, then add 1 to the resulting bit pattern viewed as an unsigned integer. |
| **Expansion of Bit Length** | Add additional bit positions to the left and fill in with the value of the original sign bit. |
| **Overflow Rule** | If two numbers with the same sign (both positive or both negative) are added, then overflow occurs if and only if the result has the opposite sign. |
| **Subtraction Rule** | To subtract *B* from *A*, take the twos complement of *B* and add it to *A*. |

# Two's Compliment

- +3 = 00000011
- +2 = 00000010
- +1 = 00000001
- +0 = 00000000
- -1 = 11111111
- -2 = 11111110
- -3 = 11111101

# Alternative Representations for 4-Bit Integers



| Decimal Representation | Sign-Magnitude Representation | Twos Complement Representation | Biased Representation |
|---|---|---|---|
| +8 | — | — | 1111 |
| +7 | 0111 | 0111 | 1110 |
| +6 | 0110 | 0110 | 1101 |
| +5 | 0101 | 0101 | 1100 |
| +4 | 0100 | 0100 | 1011 |
| +3 | 0011 | 0011 | 1010 |
| +2 | 0010 | 0010 | 1001 |
| +1 | 0001 | 0001 | 1000 |
| +0 | 0000 | 0000 | 0111 |
| −0 | 1000 | — | — |
| −1 | 1001 | 1111 | 0110 |
| −2 | 1010 | 1110 | 0101 |
| −3 | 1011 | 1101 | 0100 |
| −4 | 1100 | 1100 | 0011 |
| −5 | 1101 | 1011 | 0010 |
| −6 | 1110 | 1010 | 0001 |
| −7 | 1111 | 1001 | 0000 |
| −8 | — | 1000 | — |

# Benefits

- One representation of zero
- Arithmetic works easily
- Negating is fairly easy
    - 3 = 00000011
    - Boolean complement gives 11111100
    - Add 1 to LSB 11111101

# Negation Special Case 1

- 0 = 00000000
- Bitwise not 11111111
- Add 1 to LSB +1
- Result 1 00000000
- Overflow is ignored, so:
- - 0 = 0

# Negation Special Case 2

- 128 = 10000000
- bitwise not 01111111
- Add 1 to LSB +1
- Result 10000000
- So:
- -128 = 128
- Monitor MSB (sign bit)
- It should change during negation

# Range of Numbers

- 8 bit 2s compliment
  - +127 = 01111111 = $2^7$ -1
  - -128 = 10000000 = -$2^7$
- 16 bit 2s compliment
  - +32767 = 011111111 11111111 = $2^{15}$ - 1
  - -32768 = 100000000 00000000 = -$2^{15}$

# Conversion Between Lengths

- Positive number pack with leading zeros
- +18 = 00010010
- +18 = 00000000 00010010
- Negative numbers pack with leading ones
- -18 = 10010010
- -18 = 11111111 10010010
- i.e. pack with MSB (sign bit)

# Addition and Subtraction

- Normal binary addition
- Monitor sign bit for overflow

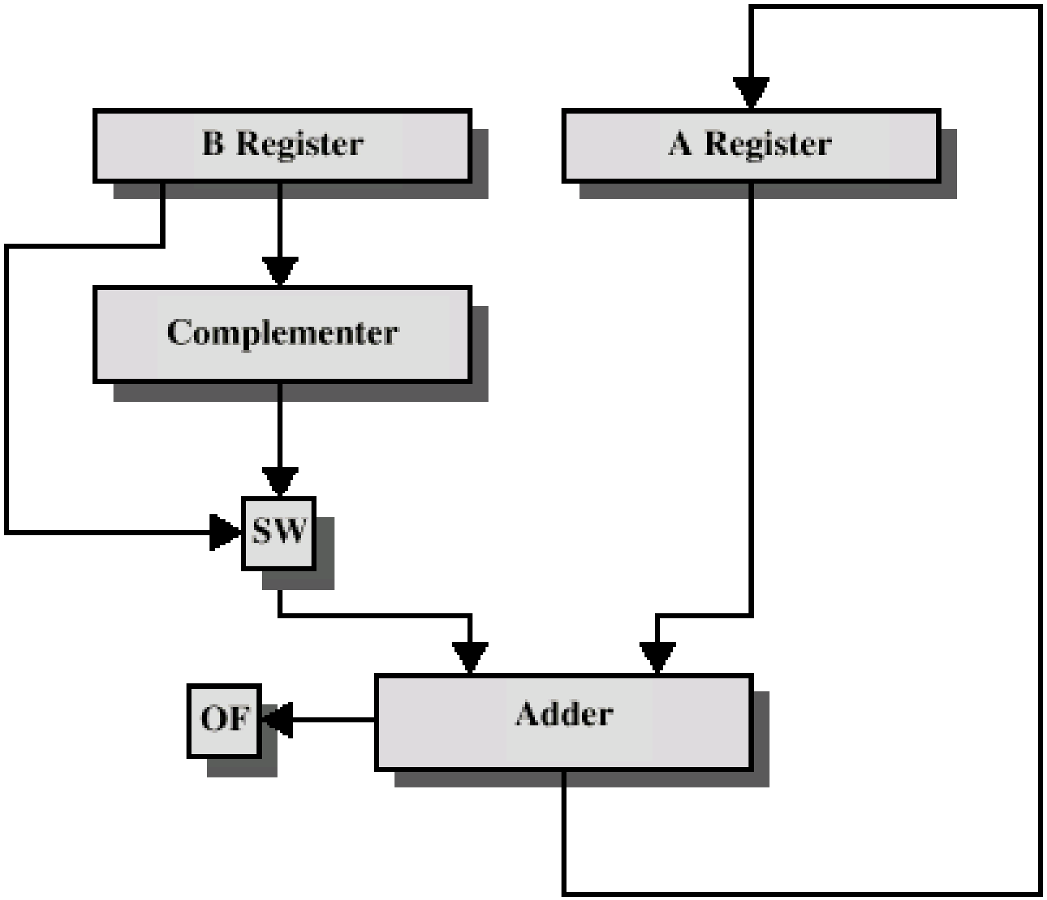
- Take twos compliment of substahend and add to minuend
  - i.e. a - b = a + (-b)

- So we only need addition and complement circuits

# Overflow

- For unsigned integers, **overflow** occurs when there is a carry out of the msb.
- 1000 (8) +1001 (9) ----------- 1 0001 (1)
- For 2's complement integers, **overflow** occurs when the signs of the addends are the same, and the sign of the result is different
- 0011 (3) + 0110 (6) ---------- 1001 (-7)
- (note that a correct answer would be 9, but 9 cannot be represented in 4-bit 2's complement)

# Hardware for Addition and Subtraction



OF = overflow bit
SW = Switch (select addition or subtraction)

# Problems

- Represent the following decimal numbers in both binary sign/magnitude and twos complement using 16 bits:

I. +512

II. -29

- Represent the following twos complement values in decimal:

I. 1101011

II. 0101101.

# Problem

□ Assume numbers are represented in 8-bit twos complement representation. Show the calculation of the following:

I. 6+13

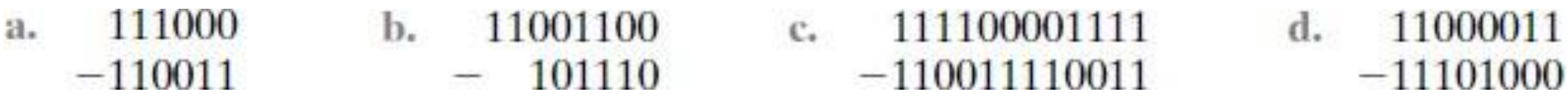
II. -6+13

III. 6-13

IV. -6-13

# Problem

- □ Find the following differences using twos complement arithmetic:



a. $\begin{array}{r} 111000 \\ -110011 \end{array}$

b. $\begin{array}{r} 11001100 \\ -\quad 101110 \end{array}$

c. $\begin{array}{r} 111100001111 \\ -110011110011 \end{array}$

d. $\begin{array}{r} 11000011 \\ -11101000 \end{array}$

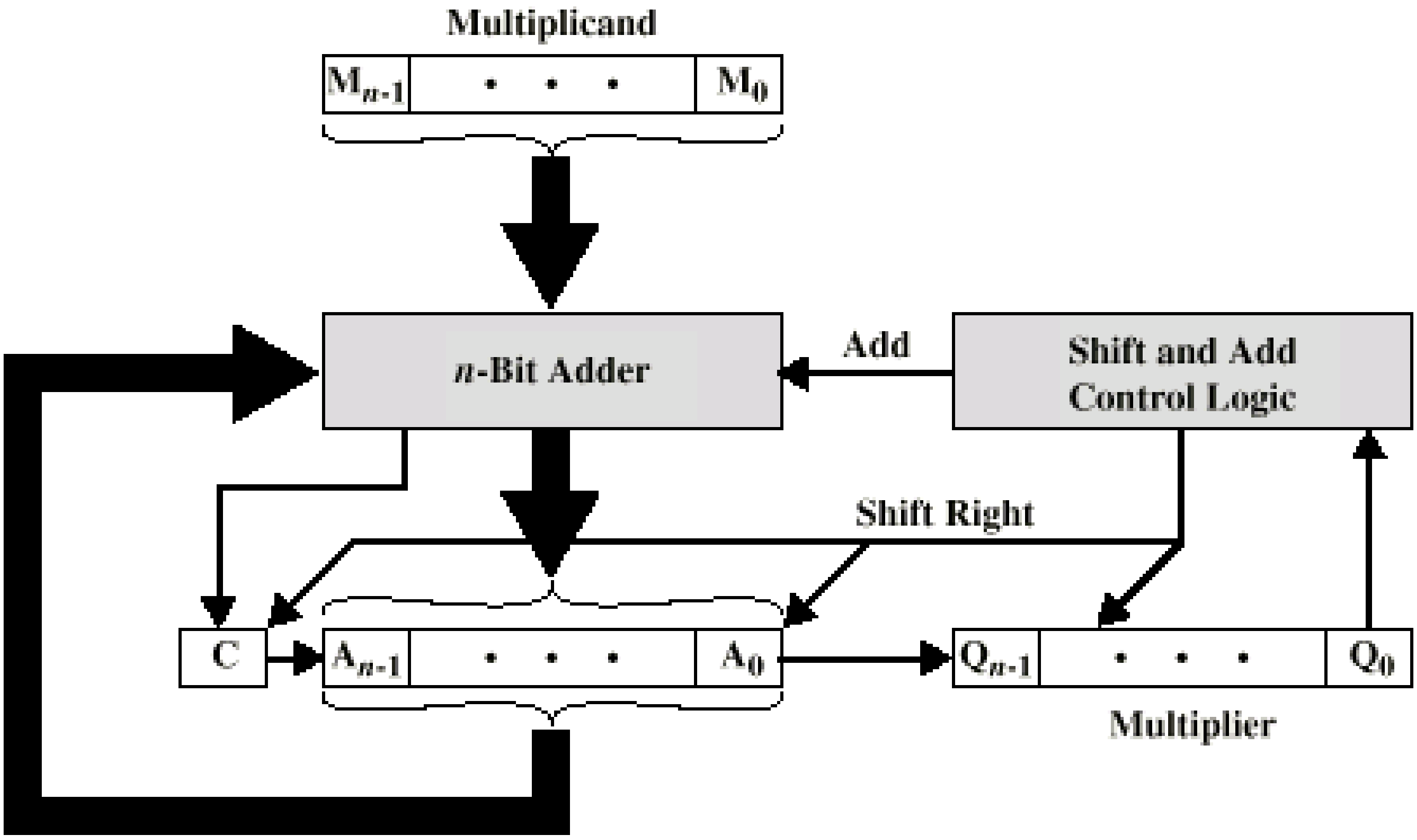
# Multiplication

- Complex
- Work out partial product for each digit
- Take care with place value (column)
- Add partial products

# Multiplication Example

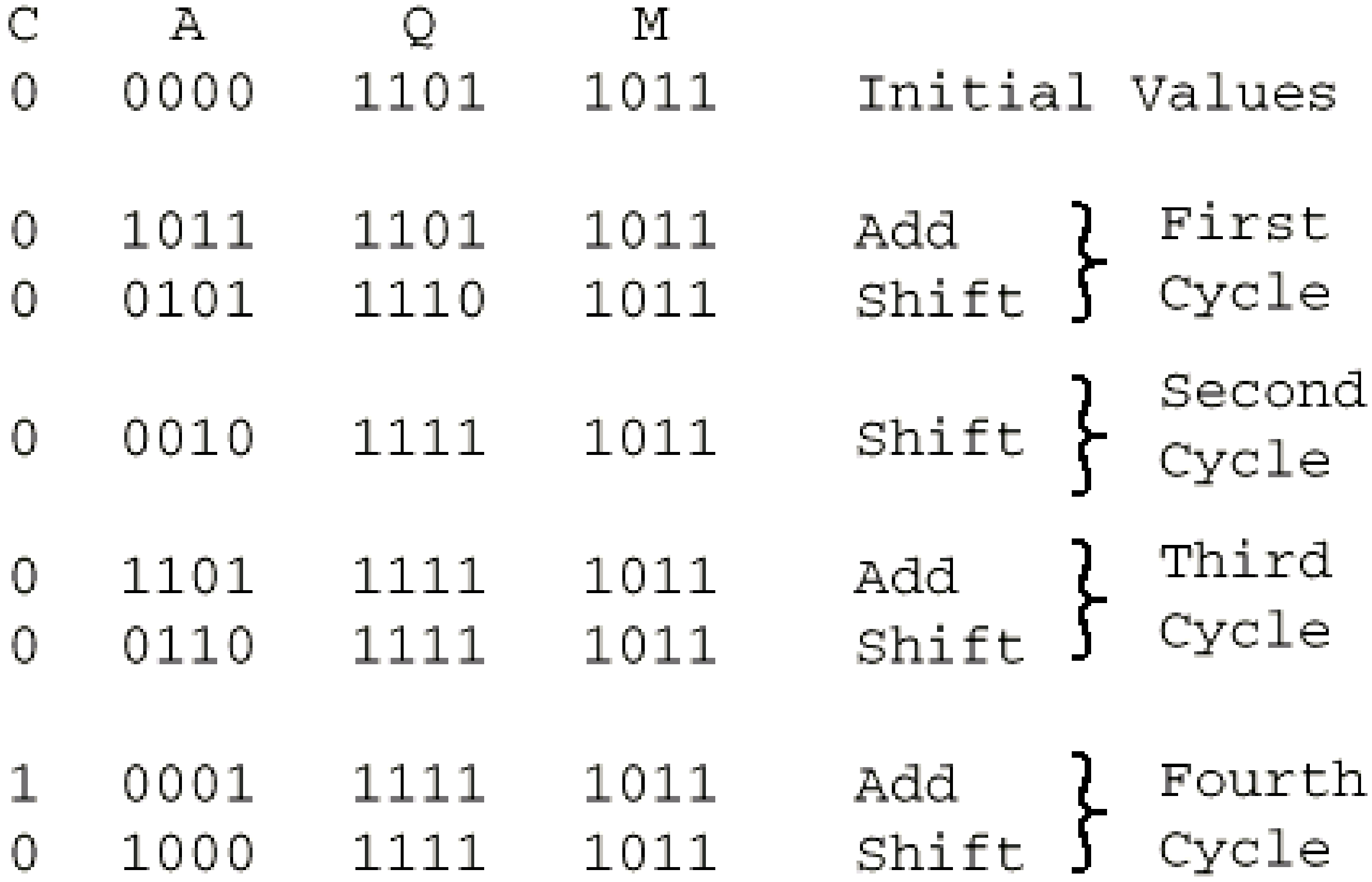
- 1011 Multiplicand (11 dec)
- x 1101 Multiplier (13 dec)
- 1011 Partial products
- 0000 Note: if multiplier bit is 1 copy
- 1011 multiplicand (place value)
- 1011 otherwise zero
- 10001111 Product (143 dec)
- Note: need double length result

# Unsigned Binary Multiplication



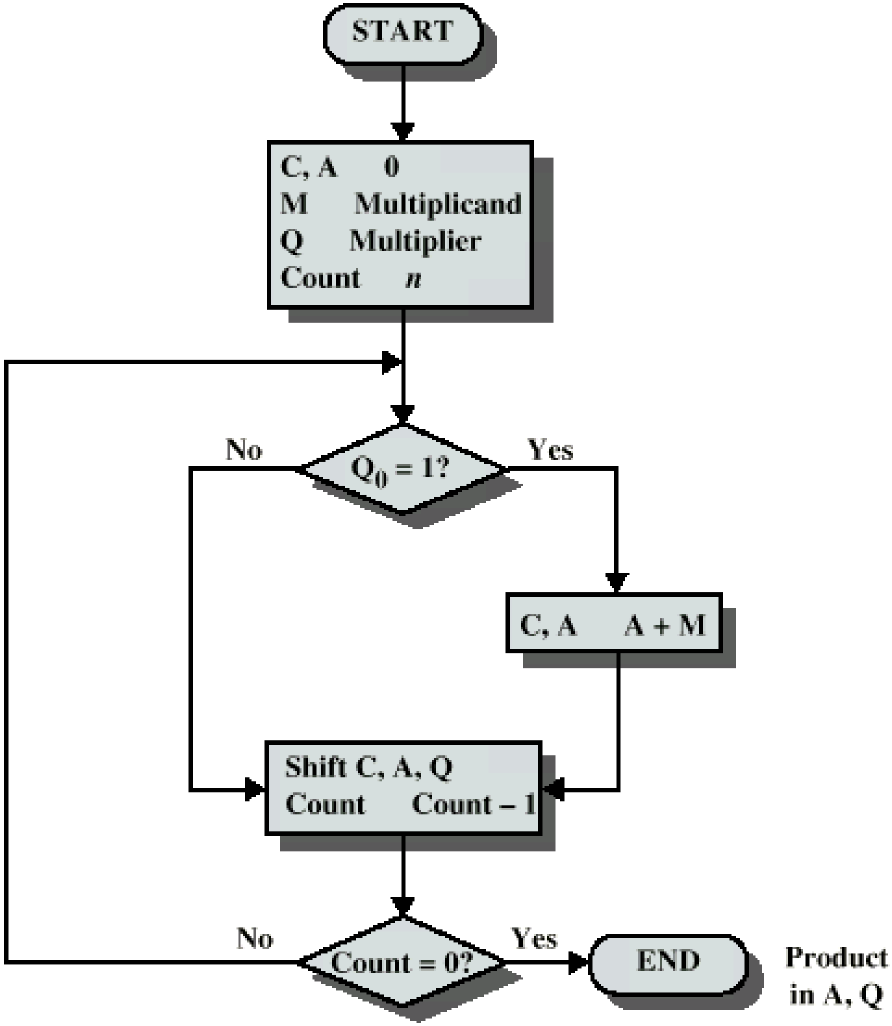
(a) Block Diagram

# Execution of Example



| C | A | Q | M | | |
|---|---|---|---|---|---|
| 0 | 0000 | 1101 | 1011 | Initial Values | |
| 0 | 1011 | 1101 | 1011 | Add | First Cycle |
| 0 | 0101 | 1110 | 1011 | Shift | |
| 0 | 0010 | 1111 | 1011 | Shift | Second Cycle |
| 0 | 1101 | 1111 | 1011 | Add | Third Cycle |
| 0 | 0110 | 1111 | 1011 | Shift | |
| 1 | 0001 | 1111 | 1011 | Add | Fourth Cycle |
| 0 | 1000 | 1111 | 1011 | Shift | |

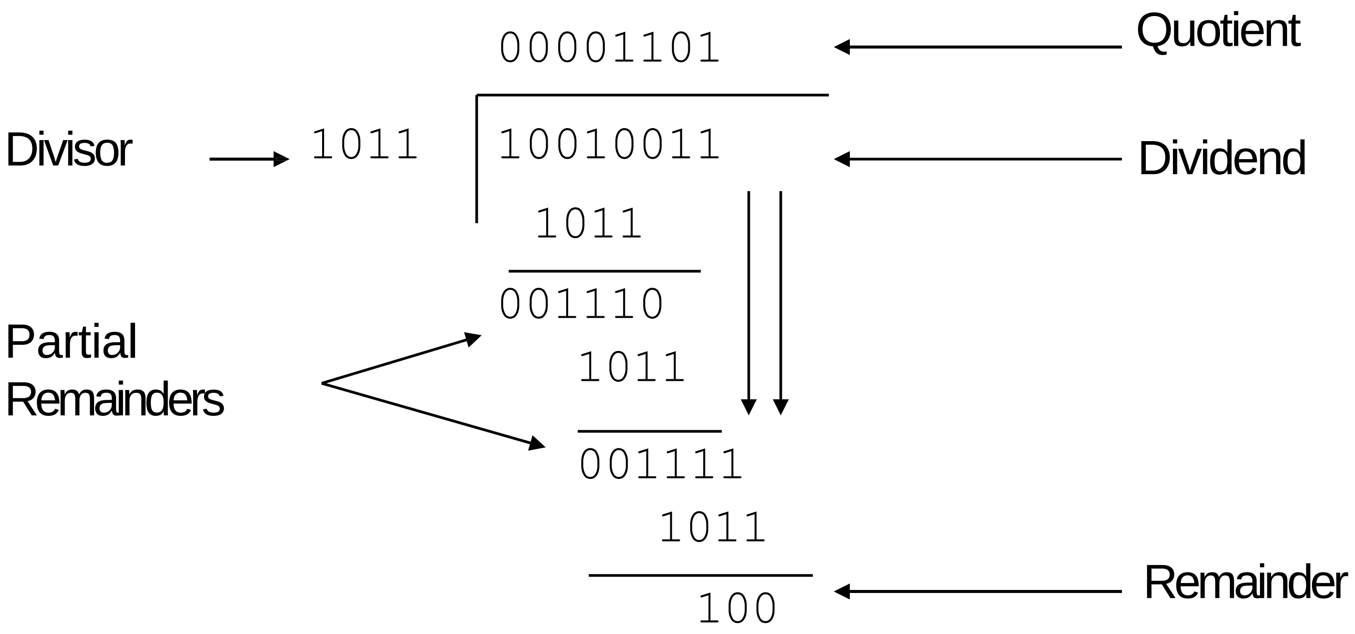
# Flowchart for Unsigned Binary Multiplication



# Division

- More complex than multiplication
- Negative numbers are really bad!
- Based on long division

# Division of Unsigned Binary Integers



Quotient

Divisor

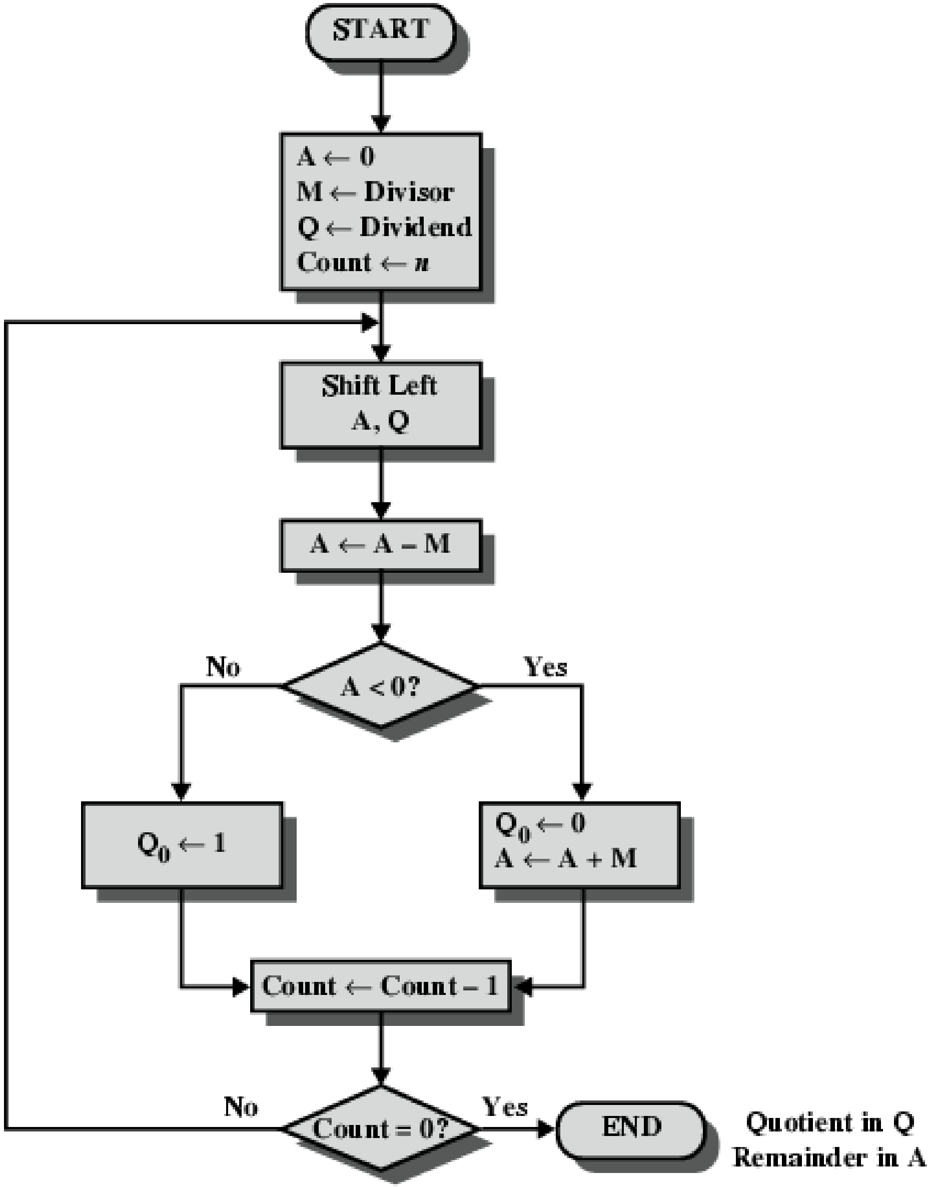
Dividend

Partial
Remainders

Remainder

```
            00001101
     1011 ) 10010011
              1011
            001110
               1011
               001111
                  1011
                    100
```

# Flowchart for Unsigned Binary Division


START
A ← 0
M ← Divisor
Q ← Dividend
Count ← n
Shift Left
A, Q
A ← A – M
A < 0?
No
Yes
Q0 ← 1
Q0 ← 0
A ← A + M
Count ← Count – 1
Count = 0?
No
Yes
END
Quotient in Q
Remainder in A