# CISC260 Machine Organization and Assembly Language

## Data representation & arithmetic in binary

How do we count?

once we have developed the notion of numbers, how do we represent them, in an *economical* way?

In contrast to the unary representation, we can try to invent a unique symbol for each number. But there are infinite many numbers, we will run out of symbols.

Let's use compound symbols made from a finite set of atom symbols.

| unary | hex | decimal | binary |
|---|---|---|---|
|  | 0 | 0 | 0 |
| * | 1 | 1 | 1 |
| ** | 2 | 2 | 10 |
| *** | 3 | 3 | 11 |
| **** | 4 | 4 | 100 |
| ***** | 5 | 5 | 101 |
| ****** | 6 | 6 | 110 |
| ******* | 7 | 7 | 111 |
| ******** | 8 | 8 | 1000 |
| ********* | 9 | 9 | 1001 |
| ********** | A | 10 | 1010 |
| *********** | B | 11 | 1011 |
| ************ | C | 12 | 1100 |
| ************* | D | 13 | 1101 |
| ************** | E | 14 | 1110 |
| *************** | F | 15 | 1111 |

# How economical?

Size of number 1000 in unary = 1000

Size of number 1000 in decimal =    4

Size of number 1000 in binary =   10

Size of number $N$ in $b$-nary = $\log_b (N)$

Remember the big O you learned in CISC220!

So, both decimal and binary has "complexity" O( log N).

$$(9038)_{ten} = 9 \cdot 10^3 + 0 \cdot 10^2 + 3 \cdot 10^1 + 8 \cdot 10^0 = 9038$$

$$(10011)_{two} = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 19$$

$$(x_n x_{n-1} \ldots x_0)_b = \sum_{i=0}^{n} x_i \cdot b^i$$

# Positional notation (or place-value) numbering system (to support arithmetic operations)

Two bits can encode 4 numbers, why we have to map

00 ->0, 01-> 1, 10 -> 2, 11 -> 3

 can we map

00->0, 10 -> 1, 01 ->3, 11 -> 2 ?

Probably yes. But when you mess up the meaning, it will require more twisted rules in order to make the arithmetic operations right (i.e., for input, operators, and output to be consistent with what they mean, as dictated by the mapping)

Use addition as example

Addition can be done by following a set of simple rules position by position.
Addition table

```
A   B  | Sum   carry
-------|--------------
0   0  | 0       0
0   1  | 1       0
1   0  | 1       0
1   1  | 0       1
```

Addition table in decimal will be 10x10 = 100 rows
Tradeoff between efficiency and simplicity
Decimal: more efficient (requires fewer digits) but also more complicated (more rules)
Binary: less efficient (requires more digits) but also less complicated (fewer rules)

# Converting between hex and binary

- Hex to binary

  e.g., 0x8F7A93 = 100011110111101010010011

- Binary to Hex

  e.g., binary 1011011110011100 = 0xB79C

| hex | binary |
| --- | --- |
| 0 | 0000 |
| 1 | 0001 |
| 2 | 0010 |
| 3 | 0011 |
| 4 | 0100 |
| 5 | 0101 |
| 6 | 0110 |
| 7 | 0111 |
| 8 | 1000 |
| 9 | 1001 |
| A | 1010 |
| B | 1011 |
| C | 1100 |
| D | 1101 |
| E | 1110 |
| F | 1111 |

**Conversion between decimal and binary**

- Binary to decimal
    - Trivial.  Use the formula:

$$(x_n x_{n-1}...x_0)_b = \sum_{i=0}^{n} x_i \cdot b^i$$

    example:

$$(10011)_{two} = 1\cdot 2^4 + 0\cdot 2^3 + 0\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 = 19$$

-Decimal to binary
    (A decimal number, dividing successively by 2  and prefixing the remainder to the result until a quotient of zero is obtained)
    e.g., decimal number 11

    11 / 2          =           5 remains 1   → 1   (LSB)
    5/2             =           2 remains 1   → 1
    2/2             =           1 remains 0   → 0
    ½              = 0         0 remains 1   → 1   (MSB)
    So the binary number is 1011

| ASCII value | Character | Control character | ASCII value | Character | ASCII value | Character | ASCII value | Character |
|---|---|---|---|---|---|---|---|---|
| 000 | (null) | NUL | 032 | (space) | 064 | @ | 096 | |
| 001 | ☺ | SOH | 033 | ! | 065 | A | 097 | a |
| 002 | ☻ | STX | 034 | " | 066 | B | 098 | b |
| 003 | ♥ | ETX | 035 | # | 067 | C | 099 | c |
| 004 | ♦ | EOT | 036 | $ | 068 | D | 100 | d |
| 005 | ♣ | ENQ | 037 | % | 069 | E | 101 | e |
| 006 | ♠ | ACK | 038 | & | 070 | F | 102 | f |
| 007 | (beep) | BEL | 039 | ' | 071 | G | 103 | g |
| 008 | ◘ | BS | 040 | ( | 072 | H | 104 | h |
| 009 | (tab) | HT | 041 | ) | 073 | I | 105 | i |
| 010 | (line feed) | LF | 042 | * | 074 | J | 106 | j |
| 011 | (home) | VT | 043 | + | 075 | K | 107 | k |
| 012 | (form feed) | FF | 044 | , | 076 | L | 108 | l |
| 013 | (carriage return) | CR | 045 | - | 077 | M | 109 | m |
| 014 | ♫ | SO | 046 | . | 078 | N | 110 | n |
| 015 | ☼ | SI | 047 | / | 079 | O | 111 | o |
| 016 | ► | DLE | 048 | 0 | 080 | P | 112 | p |
| 017 | ◄ | DC1 | 049 | 1 | 081 | Q | 113 | q |
| 018 | ↕ | DC2 | 050 | 2 | 082 | R | 114 | r |
| 019 | ‼ | DC3 | 051 | 3 | 083 | S | 115 | s |
| 020 | π | DC4 | 052 | 4 | 084 | T | 116 | t |
| 021 | § | NAK | 053 | 5 | 085 | U | 117 | u |
| 022 | ▬ | SYN | 054 | 6 | 086 | V | 118 | v |
| 023 | ↨ | ETB | 055 | 7 | 087 | W | 119 | w |
| 024 | ↑ | CAN | 056 | 8 | 088 | X | 120 | x |
| 025 | ↓ | EM | 057 | 9 | 089 | Y | 121 | y |
| 026 | → | SUB | 058 | : | 090 | Z | 122 | z |
| 027 | ← | ESC | 059 | ; | 091 | [ | 123 | { |
| 028 | (cursor right) | FS | 060 | < | 092 | \ | 124 | | |
| 029 | (cursor left) | GS | 061 | = | 093 | ] | 125 | } |
| 030 | (cursor up) | RS | 062 | > | 094 | ∧ | 126 | ~ |
| 031 | (cursor down) | US | 063 | ? | 095 | _ | 127 | ⌂ |

|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 |   | ☺ | ☻ | ♥ | ♦ | ♣ | ♠ |   |   |   |   | ♂ | ♀ |   | ♫ | ☼ |
| 1 | ► | ◄ | ↕ | ‼ | ¶ | § | ▬ | ↨ | ↑ | ↓ | → | ← | ∟ | ↔ | ▲ | ▼ |
| 2 |   | ! | " | # | $ | % | & | ' | ( | ) | * | + | , | - | . | / |
| 3 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | : | ; | < | = | > | ? |
| 4 | @ | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 5 | P | Q | R | S | T | U | V | W | X | Y | Z | [ | \ | ] | ^ | _ |
| 6 | ` | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o |
| 7 | p | q | r | s | t | u | v | w | x | y | z | { | \| | } | ~ | ⌂ |
| 8 | Ç | ü | é | â | ä | à | å | ç | ê | ë | è | ï | î | ì | Ä | Å |
| 9 | É | æ | Æ | ô | ö | ò | û | ù | ÿ | Ö | Ü | ¢ | £ | ¥ | ₧ | ƒ |
| A | á | í | ó | ú | ñ | Ñ | ª | º | ¿ | ⌐ | ¬ | ½ | ¼ | ¡ | « | » |
| B | ░ | ▒ | ▓ | │ | ┤ | ╡ | ╢ | ╖ | ╕ | ╣ | ║ | ╗ | ╝ | ╜ | ╛ | ┐ |
| C | └ | ┴ | ┬ | ├ | ─ | ┼ | ╞ | ╟ | ╚ | ╔ | ╩ | ╦ | ╠ | ═ | ╬ | ╧ |
| D | ╨ | ╤ | ╥ | ╙ | ╘ | ╒ | ╓ | ╫ | ╪ | ┘ | ┌ | █ | ▄ | ▌ | ▐ | ▀ |
| E | α | ß | Γ | π | Σ | σ | µ | τ | Φ | θ | Ω | δ | ∞ | φ | ε | ∩ |
| F | ≡ | ± | ≥ | ≤ | ⌠ | ⌡ | ÷ | ≈ | ° | · | · | √ | ⁿ | ² | ■ |   |

- Viewing a binary file as text file

>more

- Bits are just bits (no inherent meaning)
  conventions define relationship between bits and numbers
- Binary numbers (base 2)
  0000 0001 0010 0011 0100 0101 0110 0111 1000 1001...
  decimal:  0...$2^n$-1

- Of course it gets more complicated:
  numbers are finite (overflow)
  fractions and real numbers
  negative numbers

- How do we  represent negative numbers?
  i.e., which bit patterns will represent which numbers?

# Possible Representations

| Sign Magnitude: | One's Complement | Two's Complement |
|---|---|---|
| 000 = +0 | 000 = +0 | 000 = +0 |
| 001 = +1 | 001 = +1 | 001 = +1 |
| 010 = +2 | 010 = +2 | 010 = +2 |
| 011 = +3 | 011 = +3 | 011 = +3 |
| 100 = -0 | 100 = -3 | 100 = -4 |
| 101 = -1 | 101 = -2 | 101 = -3 |
| 110 = -2 | 110 = -1 | 110 = -2 |
| 111 = -3 | 111 = -0 | 111 = -1 |

- Issues:  balance, number of zeros, ease of operations

Issues with sign-magnitude.
- Two zero's
- Arithmetic operations are cumbersome to implement.
- E.g. when adding two numbers, if the second operand is negative, ordinary addition will give wrong answer:

       2 + (-3)                    5 + (-5) = 0

       010                         1101
       111                         0101
    -----------                 -----------
     1|001                        10010

- There is an overflow;
- The answer is wrong

Therefore, depending on the signs of the operands, different rules are needed to ensure the correct arithmetic operations.

Issues with one's complement.
- Arithmetic operations are cumbersome to implement.
- E.g. when adding two numbers, if the second operand is negative, ordinary addition may give wrong answer:

2 + (-1)

```
  010
  110
-----------
 1|000
```

1), there is an overflow; 2) the answer is wrong
Therefore, depending on the signs of the operands, different rules are needed to ensure the correct arithmetic operations.

```
  subtraction:
      000    (0)
-)  001    (1)
--------------------
      111    (-0)
```

Motivation to two's complement.
What has led to negative numbers?     Subtraction. Let's see the effect of borrowing.

```
     0000
  -  0001
 ------------
     1111
```

If we can borrow "out of the range", this gives us negative one.
So 1111 here actually represents -1, as the result of 0-1.  This is two's complement.

So when we add two integers, a + b, if b is negative, it is equivalent to

$$a - |b| = a + (0 - |b|) = a + b",$$

where b" stands for b in two's complement.

This shows:
 1. addition can be done as usual for integers, both positive and negative, when represented in two's complement.
 2. subtraction can be done by negation (which is a simpler operation) and addition.

# Now look at two's complement

Two's Complement

      011 = +3

      010 = +2

      001 = +1

      000 = 0

      111 = -1

      110 = -2

      101 = -3

      100 = -4       => left-most bit is sign bit but also contributes a value

$$A = d_2 (-2^2) + d_1 2^1 + d_0 2^0$$

Starting at 000, keep subtracting 1 to get negative numbers, and keep adding 1 to get positive numbers.

- Sign bit contribution: $(-2^2)$ in this example, and $-2^{N-1}$ for N-bit integers.

- There is one zero.

- Unbalanced: the most negative integer has no positive counterpart.

- Look at pairs (1, -1), (2, -2) … to discover patterns for negation.

- Check the correctness of arithmetic operations.

# 2's Complement Number "line": N = 5
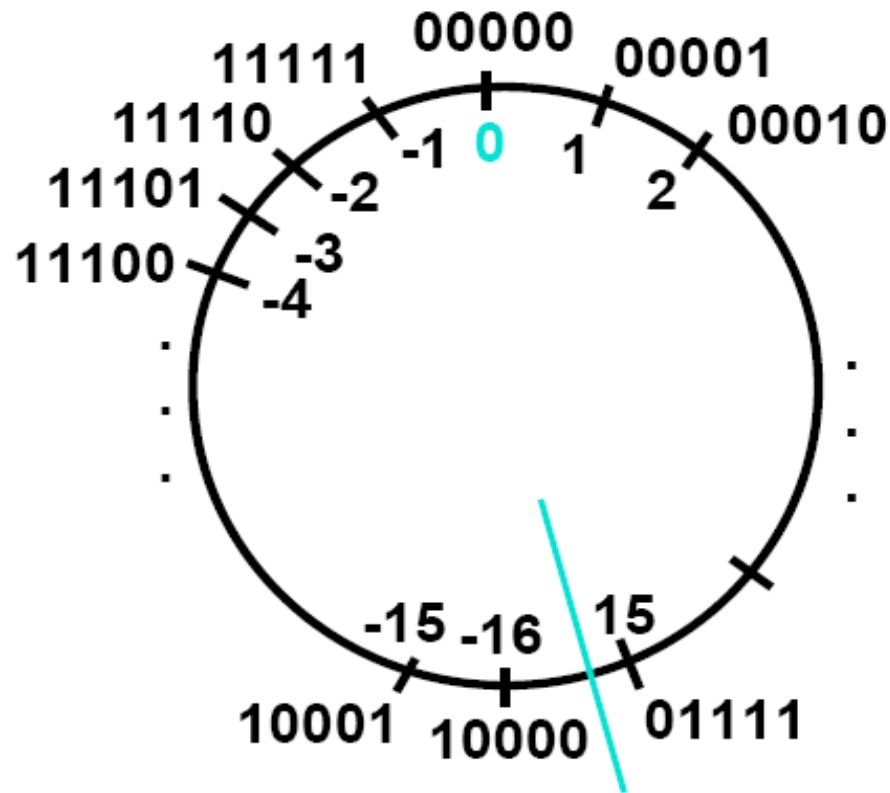


- $2^{N-1}$ non-negatives
- $2^{N-1}$ negatives
- one zero
- how many positives?

# Generalized to 32 bit (used in ARM)

32 bit signed numbers:

```
0000 0000 0000 0000 0000 0000 0000 0000two =    0ten
0000 0000 0000 0000 0000 0000 0000 0001two = +  1ten
0000 0000 0000 0000 0000 0000 0000 0010two = +  2ten
...
0111 1111 1111 1111 1111 1111 1111 1110two = + 2,147,483,646ten
0111 1111 1111 1111 1111 1111 1111 1111two = + 2,147,483,647ten
1000 0000 0000 0000 0000 0000 0000 0000two = − 2,147,483,648ten
1000 0000 0000 0000 0000 0000 0000 0001two = − 2,147,483,647ten
1000 0000 0000 0000 0000 0000 0000 0010two = − 2,147,483,646ten
...
1111 1111 1111 1111 1111 1111 1111 1101two = −  3ten
1111 1111 1111 1111 1111 1111 1111 1110two = −  2ten
1111 1111 1111 1111 1111 1111 1111 1111two = −  1ten
```

*maxint*

*minint*

# Two's Complement Formula

° Can represent positive <u>and negative</u> numbers in terms of the bit value times a power of 2:

$$d_{31} \times \boxed{-2^{31}} + d_{30} \times 2^{30} + ... + d_2 \times 2^2 + d_1 \times 2^1 + d_0 \times 2^0$$

° Example: $1111\ 1100_{two}$

$$= 1 \times -2^7 + 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$$

$$= -2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 0 + 0$$

$$= -128 + 64 + 32 + 16 + 8 + 4$$

$$= -128 + 124$$

$$= -4_{ten}$$

# Two's Complement Operations

- Negating a two's complement number: invert all bits and add 1.

Why?

For a number X, let X' be its invert, then $X + X' = 11...1 = -1_{ten}$

Therefore, $X' + 1 = -X$

E.g., X = 5 in 8-bit

| | | |
|---|---|---|
| X | = | 0000 0101 |
| X' | = | 1111 1010 |
| X' +1 | = | 1111 1011 |

- Special case1: negate 0 should still give you 0 (by ignoring carry out at the leftmost bit)
- Special case2: negate the most negative number, say 1111 in 4-bit, you still get the same number since the most negative number does not have a positive counterpart in two's complement – its range is asymmetric!
- remember: for numbers in two's complement, "negate" and "invert" are quite different!

# Addition & Subtraction

- Just like in grade school  (carry/borrow 1s)

```
  0111 (7)          0111 (7)          0110 (6)
+ 0110 (6)        - 0110 (6)        - 0111 (7)
  ────────          ────────          ────────
  1101              0001              1111 (-1)
```

- Two's complement operations easy

  - subtraction using addition of negative numbers, and unlike the above example, here the sign is automatically taken care of.

```
  0111 (7)              0110     (6)
+ 1010 (-6)           + 1001     (-7)
  ──────────            ────────────
  0001                  1111
```

More examples

Overflow is not indicated by spilling over a bit out of the range; rather it is indicated by sign bit flipping: both operands have the same sign bit, but the result gets an opposite sign bit.

| | |
|---|---|
| ```1001 = −7```<br>```+0101 =    5```<br>```1110 = −2```<br><br>(a) (−7) + (+5) | ```1100 = −4```<br>```+0100 =    4```<br>```10000 =    0```<br><br>(b) (−4) + (+4) |
| ```0011 = 3```<br>```+0100 = 4```<br>```0111 = 7```<br><br>(c) (+3) + (+4) | ```1100 = −4```<br>```+1111 = −1```<br>```11011 = −5```<br><br>(d) (−4) + (−1) |
| ```0101 = 5```<br>```+0100 = 4```<br>```1001  = Overflow```<br><br>(e) (+5) + (+4) | ```1001 = −7```<br>```+1010 = −6```<br>```10011 = Overflow```<br><br>(f) (−7) + (−6) |

# Overflow for addition and subtraction of signed integers

| Operation | Operand A | Operand B | Result indicating overflow |
|:---:|:---:|:---:|:---:|
| $A + B$ | $\geq 0$ | $\geq 0$ | $< 0$ |
| $A + B$ | $< 0$ | $< 0$ | $\geq 0$ |
| $A - B$ | $\geq 0$ | $< 0$ | $< 0$ |
| $A - B$ | $< 0$ | $\geq 0$ | $\geq 0$ |

**FIGURE 3.2 Overflow conditions for addition and subtraction.**

**Sign extension**: If you get larger size to represent numbers, say from 16bit to 32 bit, how do you migrate? It is simple, you just pad these new bits with zeros or with ones as decided by the sign bit. This is sign extension.

Proof of the sign extension for negative numbers:

In b+1 bit representation

$$Z = -1 \times (2)^b + \underbrace{z_{b-1} \times (2)^{b-1} + \dots + z_0}_{y}$$

$$= -(2)^b + y$$

Extended to B+1 bit with B > b:

$$Z' = -1 \times (2)^B + (2)^{B-1} + \dots (2)^{b+1} + (2)^b + y.$$

It is easy to see that

$$Z' - Z = -1 \times (2)^B + (2)^{B-1} + \dots (2)^{b+1} + (2)^b + y + (2)^b - y$$

$$= -1 \times (2)^B + (2)^{B-1} + \dots (2)^{b+1} + \underbrace{(2)^b + (2)^b}$$

$$(2)^{b+1}$$

Starting from right end, the two neighboring terms are identical and summing them will give a new term identical the third term. Repeating this process till the n

# Multiplication

- More complicated than addition
  - accomplished via shifting and addition
- More time
- Let's look at a grade school (bit-shifting) algorithm

```
      0010    (multiplicand)
   x  1011    (multiplier)
```
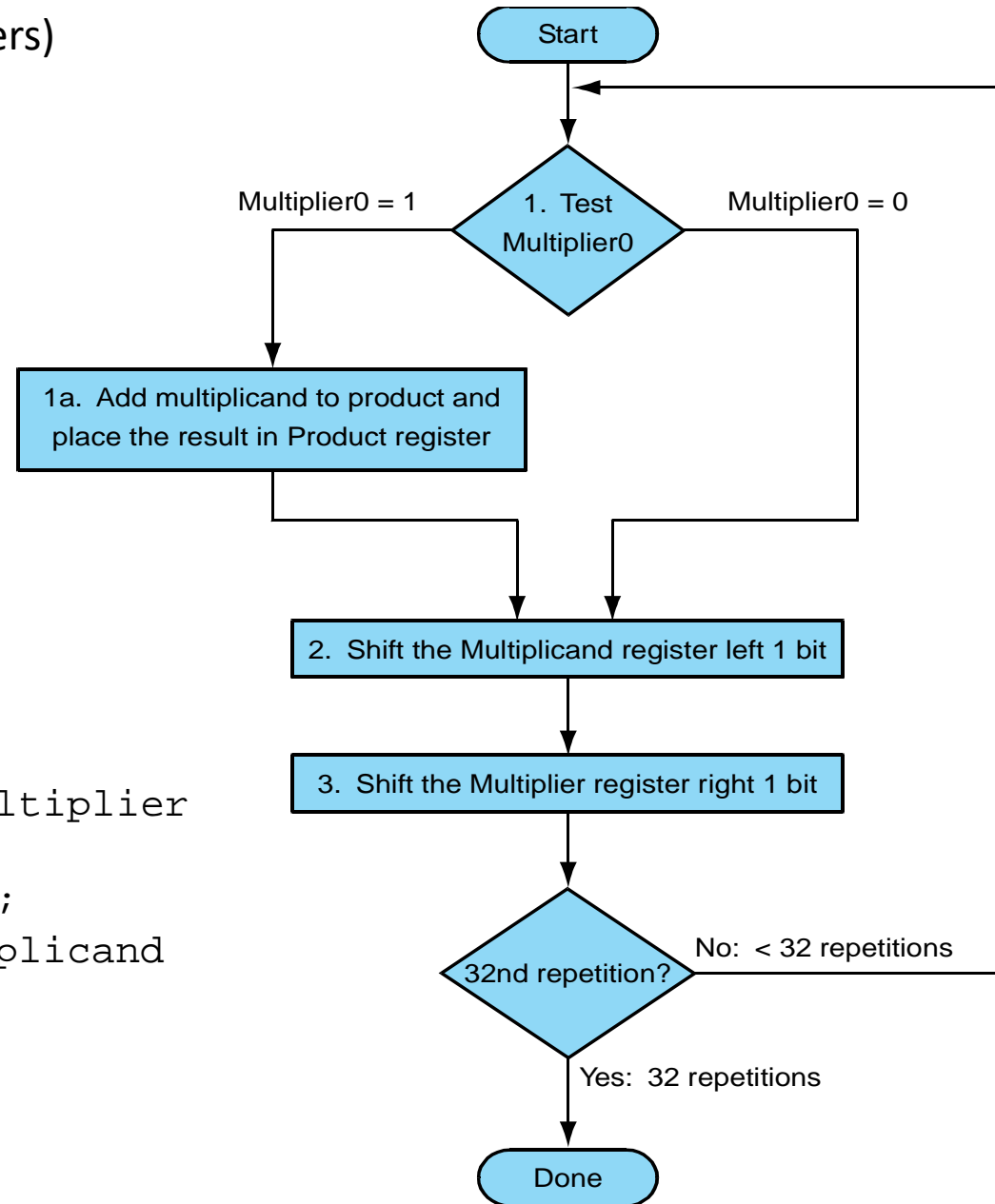
# Multiplication (for **unsigned** integers)

```
      0010   (multiplicand)
   x  1011   (multiplier)
      0010
     0010
    0000
   0010___
   0010110  (product)
```

```
product  = 0
while (b != 0) { // b is multiplier
    if(b & 0x01) == 1
        product = product + a;
               // a is multiplicand
    a = a << 1;
    b = b >> 1;
}
return product;
```

Start

Multiplier0 = 1      1. Test      Multiplier0 = 0
                     Multiplier0

1a.  Add multiplicand to product and
place the result in Product register

2.  Shift the Multiplicand register left 1 bit

3.  Shift the Multiplier register right 1 bit

32nd repetition?      No:  < 32 repetitions

Yes:  32 repetitions

Done

# What if either operant is negative?

- Two's complement works well for addition and subtraction—how about multiplication?

  **Example** (numbers interpreted in two's complement):

$$\begin{array}{r}
1\,0\,1\,1 \quad (-5) \\
\times\,0\,1\,1\,1 \quad (7) \\
\hline
1\,0\,1\,1 \\
1\,0\,1\,1 \\
1\,0\,1\,1 \\
+\,0\,0\,0\,0 \\
\hline
0\,1\,0\,0\,1\,1\,0\,1 \quad (77\,?) \\
\end{array}$$

  – This multiplication scheme fails if either operand is negative

Multiplication of signed integers

- Covert the multiplicand and multiplier to positive numbers and remember the original signs

- Run multiplication algorithm (for 31 iterations) leaving the signs out of the calculation

- Negate the product if the original signs disagree (sign extension may be involved)

## Multiplication (for signed integers)

```
product  = 0
sign_a = 1;
sign_b = 1;
if(a < 0) sign_a = -1; a = -a;
if(b < 0) sign_b = -1; b = -b;
while (b != 0) { // b is multiplier
    if(b & 0x01) == 1
       product = product + a;
                   // a is multiplicand
    a = a << 1;
    b = b >> 1;
}
If (sign_a != sign_b) product = -product;
return product;
```

# Multiplication with two's complement?

- Why does multiplication fail?

- Remember: the multiplication of two $n$-bit numbers requires $2n$-bit additions to add the partial products

  - This is what actually happens:

$$
\begin{array}{r}
1\,0\,1\,1 \quad (-5) \\
\times\,0\,1\,1\,1 \quad (7) \\
\hline
0\,0\,0\,0\,1\,0\,1\,1 \\
0\,0\,0\,1\,0\,1\,1\,0 \\
0\,0\,1\,0\,1\,1\,0\,0 \\
+\,0\,0\,0\,0\,0\,0\,0\,0 \\
\hline
0\,1\,0\,0\,1\,1\,0\,1 \quad (77) \\
\end{array}
$$

  - Note: interpreted as 8-bit numbers, the shifted numbers in the lines are *positive* (and not *negative* as expected)

# Multiplication with two's complement?

- To **extend** a $n$-bit two's complement number to $m$ bits ($m > n$), pad the number on the left; the new $m - n$ bits have the **same value as the sign bit**

    **Example** (extend 4-bit to 8-bit number, two's complement):

    $$-5_{10} \quad = \quad 1011_2 \quad = \quad \underbrace{1111}_{new}1011_2$$

- 4-bit multiplication (corrected 8-bit extension):

$$
\begin{array}{r}
1\,0\,1\,1 \quad (-5) \\
\times\,0\,1\,1\,1 \quad (7) \\
\hline
1\,1\,1\,1\,1\,0\,1\,1 \\
1\,1\,1\,1\,0\,1\,1\,0 \\
1\,1\,1\,0\,1\,1\,0\,0 \\
+\,0\,0\,0\,0\,0\,0\,0\,0 \\
\hline
1\,1\,0\,1\,1\,1\,0\,1 \quad (-35)
\end{array}
$$

Do an example when the multiplier is also negative and show how the sign bit should be treated specially: the intermediate product from the sign bit should be subtracted from the total. With the consecutive-one approach, the sign-bit is taken care of automatically.

Multiplication (for signed integers in two's complement)
Example when multiplier is negative

```
      0010   (2)
    x 1011   (-5)
  00000010
  00000100
  00000000
+)  00010000
  00010110   (22)
```

```
      0010   (2)
    x 1011   (-5)
  00000010
  00000100
  00000000
+) -00010000
```

This last row is contributed by the leftmost bit (i.e., sign bit) of the multiplier (-5), which is negative, and therefore should be subtracted rather added to the final product.

This subtraction can be implemented by negation first and addition.

-00010000  = 11110000

```
      0010   (2)
    x 1011   (-5)
  00000010
  00000100
  00000000
+)  11110000
  11110110   (-10)
```

Consecutive one

```
   aaaaaaaa   (A)
  x 00111100   (B)
   00000000
   00000000
  aaaaaaaa
 aaaaaaaa
 aaaaaaaa
 aaaaaaaa
00000000
+)00000000_          __
```
$A \times (2^5 + 2^4 + 2^3 + 2^2)$     (C)

$C = A \times B$

$76543210$

$B = 00111100$

$= 2^5 + 2^4 + 2^3 + 2^2$

$B + 2^2 = 2^5 + 2^4 + 2^3 + 2^2 + 2^2$

$= 2^5 + 2^4 + 2^3 + 2^3$

$= 2^5 + 2^4 + 2^4$

$= 2^5 + 2^5$

$= 2^6$

$B = 2^6 - 2^2$

$C = A \times (2^6 - 2^2)$

```
   aaaaaaaa   (A)
  x 00111100   (B)
   00000000
   00000000
  -aaaaaaaa
  00000000
 00000000
 00000000
 aaaaaaaa
+)  00000000
```
$A2^6 - A2^2$     (C)

Observation: A left shifted 6 positions and then added to C
A left shifted 2 positions and then subtracted from C
Rule: Enter a "consecutive-one" block, subtract the shifted A from product
Within the block, just shift A (without adding to C)
Exit the block, add the shifted A to the product

"consecutive-one" taking care of the sign bit for multiplier automatically.

C = A x B

```
  76543210
B = 10111100
  = -2^7 +2^5 + 2^4 + 2^3 +2^2
```

$B = -2^7 +2^5 + 2^4 + 2^3 +2^2$

$B = -2^7 + (2^6 - 2^2)$
$C = A \times (-2^7) + A \times (2^6 - 2^2)$

Ask: What if this bit is also 1? Then the whole consecutive-block is just sign ext of a multiplier 100.

$B_0\ B_{-1}$

```
        aaaaaaaa    (A)
     x  10111100    (B)
        00000000
        00000000
       -aaaaaaaa
        00000000
        00000000
        00000000
       aaaaaaaa
+    -aaaaaaaa_           __
   A(-2^7)  +   A2^6  -  A2^2     (C)
```
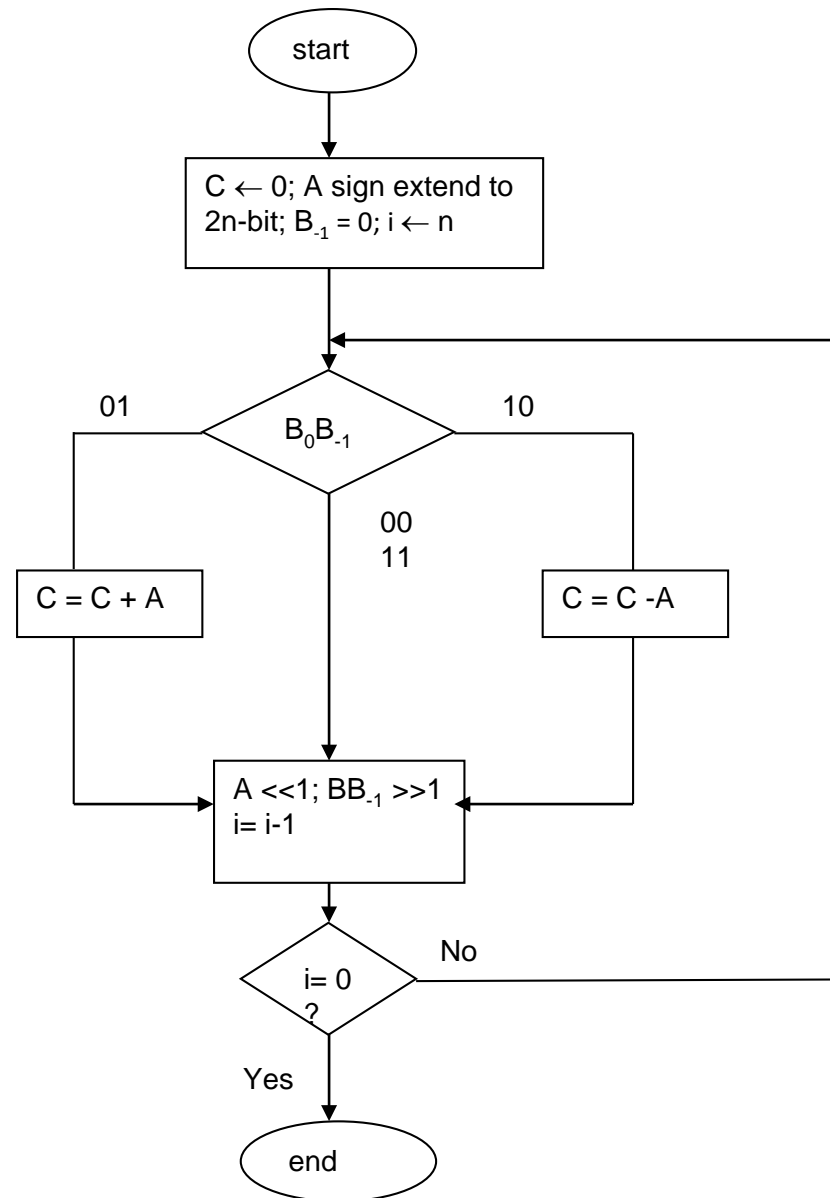
The sign bit for a negative multiplier is taken care of automatically in Booth algorithm.

A x B = C
A: multiplicand, n-bit
B: multiplier, n-bit
C: product, 2n-bit

start

$C \leftarrow 0$; A sign extend to 2n-bit; $B_{-1} = 0$; $i \leftarrow n$

$B_0 B_{-1}$

01

10

00
11

C = C + A

C = C -A

A <<1; $BB_{-1}$ >>1
i= i-1

i= 0
?

No

Yes

end

A = 1011
B = 0111

C = A x B

| i | A | B | $B_{-1}$ | C | actions |
|---|---|---|---|---|---|
| 4 | 1111 1011 | 0111 | 0 | 0000 0000 | $B_0B_{-1}$ = "10" entering c1 block |
| 4 | 1111 1011 | 0111 | 0 | 0000 0101 | C = C-A = 5 |
| 4 | 1111 0110 | 0011 | 1 | 0000 0101 | A <<1, $BB_{-1}$ >>1 |
| 3 | 1111 0110 | 0011 | 1 | 0000 0101 | $B_0B_{-1}$ = "11", inside c1 block |
| 3 | 1110 1100 | 0001 | 1 | 0000 0101 | A <<1, $BB_{-1}$>>1, i-- |
| 2 | 1110 1100 | 0001 | 1 | 0000 0101 | $B_0B_{-1}$ = "11", inside c1 block |
| 2 | 1101 1000 | 0000 | 1 | 0000 0101 | A>>1, $BB_{-1}$ >>1, i-- |
| 1 | 1101 1000 | 0000 | 1 | 1101 1101 | $B_0B_{-1}$ = "01", exiting c1 block, C = C+A = -35 |
| 1 | 1011 0000 | 0000 | 0 | 1101 1101 | A>>1, $BB_{-1}$ >>1, i-- |
| 0 | stop | | | | |