CS 110 (CS)
Intro to Computer Architecture,
Hofstra University

Syllabus

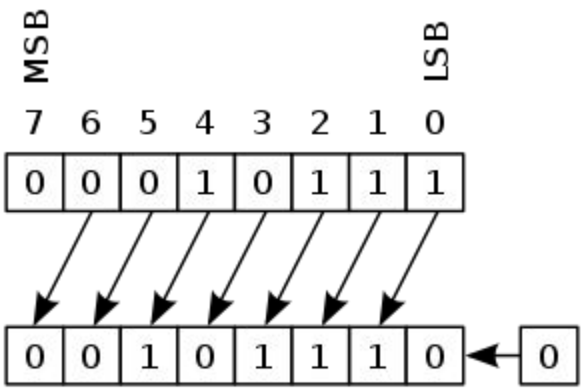
Objectives

- Multiplication, Division, and Booth Algorithm
- Understand the fundamental concepts of floating-point representation.
- Gain familiarity with the most popular character codes.
- Understand the concepts of error detecting and correcting codes

SHIFT operations

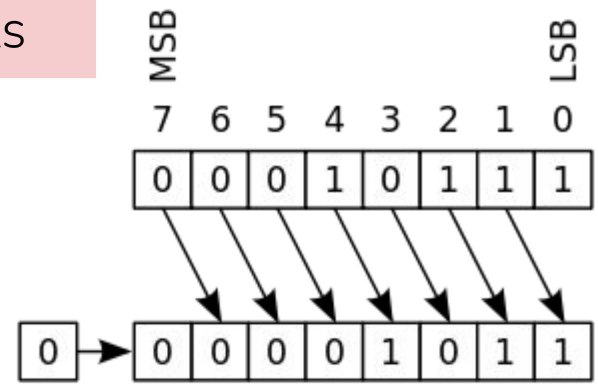
- **Logical shift:** A bitwise operation that shifts all the bits of its operand. The two base variants are the **logical left shift (LLS)** and the **logical right shift (LRS)**. This is further modulated by the number of bit positions a given value shall be shifted, such as shift left by 1 or shift right by n

LLS



Anything special that you observed?

LRS



The LLS multiplies the quantity by 2 while LRS divides (integer division) the quantity by 2.

Would the equivalence of shifts operations with multiplication and division hold on all possible cases? For example, with all signed and unsigned numbers?

SHIFT operations

Would the equivalence of shifts operations with multiplication and division hold on all possible cases? For example, with all signed and unsigned numbers?

Logical LEFT shifts (**unsigned numbers**)

00010111 (23) << 1 (left shift) => 00101110 (46)
00010011 (19) << 1 (left shift) => 00100110 (38)

...

n left shifts mean multiplication by 2^n

Does it hold in case of 00000000, or 11111111?

00000000 (0) << 1 (left shift) => 00000000 (0)

11111111 (255) << 1 (left shift)
=> 1111110 (254)

The last one did not work!!! Its an arithmetic overflow, anyway!

Logical RIGHT shifts (**unsigned numbers**)

00010111 (23) >> 1 (right shift) => 00001011 (11)
00010011 (19) >> 1 (right shift) => 00001001 (9)

...

n RIGHT shifts mean division by 2^n

Does it hold in case of 00000000, or 11111111?

00000000 (0) >> 1 (right shift) => 00000000

11111111 (255) >> 1 (right shift)
=> 01111111 (127)

SHIFT operations

Would the equivalence of shifts operations with multiplication and division hold on all possible cases? For example, with all signed and unsigned numbers?

Logical LEFT shifts (**signed numbers**)

10000011 (-125) << 1 (left shift) => 00000110 (6)
10001000 (-120) << 1 (left shift) => 00010000 (16)

...

n left shifts DOES NOT mean multiplication by 2^n
in case of signed integers

Logical RIGHT shifts (**signed numbers**)

10000011 (-125) >> 1 (right shift) => 01000001 (65)
10001000 (-120) >> 1 (right shift) => 01000100 (68)

...

n RIGHT shifts DOES NOT mean division by 2^n

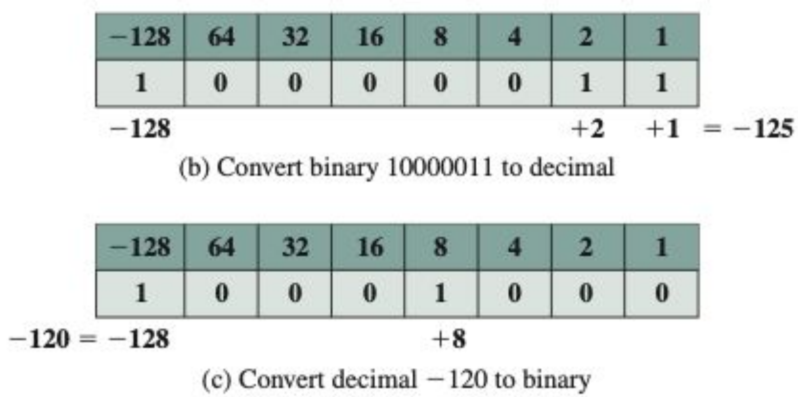
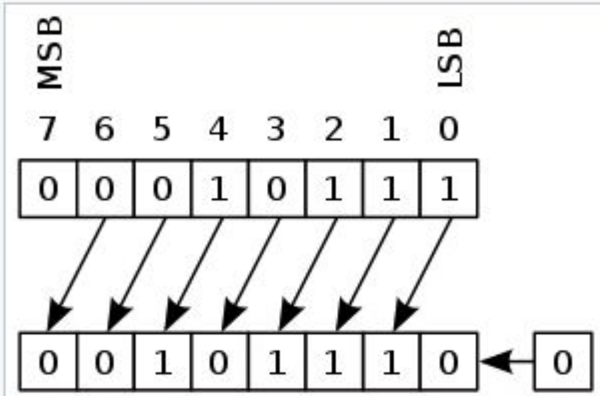
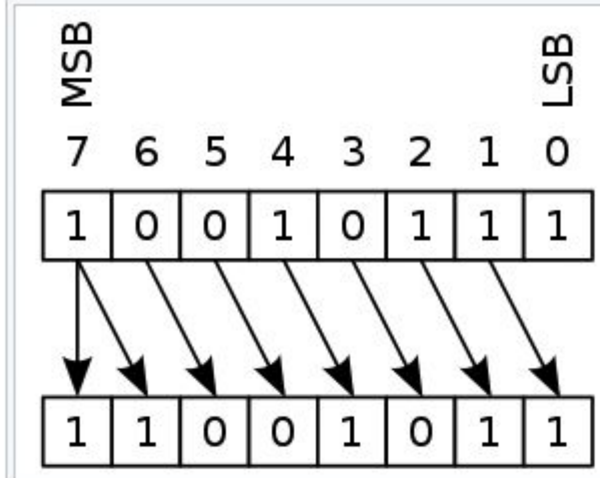


Figure 10.2 Use of a Value Box for Conversion between Twos Complement Binary and Decimal

Arithmetic SHIFT operations



A left arithmetic shift of a binary number by 1. The empty position in the least significant bit is filled with a zero.



A right arithmetic shift of a binary number by 1. The empty position in the most significant bit is filled with a copy of the original MSB.

Arithmetic left shifts are equivalent to multiplication by a power of 2 for binary numbers. They are also equivalent to Logical shift operations.

- **Arithmetic right shifts** are major traps for the unwary, specifically in treating rounding of negative integers. For example, in the usual two's complement representation of negative integers, −1 is represented as all 1's. For an 8-bit signed integer this is 1111 1111. An arithmetic right-shift by 1 (or 2, 3, ..., 7) yields 1111 1111 again, which is still −1.

- **It is frequently stated** that arithmetic right shifts are equivalent to division by a power of 2 for binary numbers, and hence that division by a power of the radix can be optimized by implementing it as an arithmetic right shift.

- Large number of 1960s and 1970s programming handbooks, manuals, and other specifications from companies and institutions such as DEC, IBM, Data General, and ANSI **make such incorrect statements.**

The suggestion: You can make use of shift operations to multiply or divide by 2^n but you should be mindful if you are working with signed numbers, you may not get the desired result.

Arithmetic SHIFT operations

EXAMPLE 2.28 Multiply the value 11 (expressed using 8-bit signed two's complement representation) by 2.

We start with the binary value for 11:

0 0 0 0 1 0 1 1

and we shift left one place, resulting in:

0 0 0 1 0 1 1 0

which is decimal $2 = 11 \times 2$. No overflow has occurred, so the value is correct.

EXAMPLE 2.29 Multiply the value 12 (expressed using 8-bit signed two's complement representation) by 4.

We start with the binary value for 12:

0 0 0 0 1 1 0 0

and we shift left two places (each shift multiplies by 2, so two shifts is equivalent to multiplying by 4), resulting in:

0 0 1 1 0 0 0 0

which is decimal $48 = 12 \times 4$. No overflow has occurred, so the value is correct.

EXAMPLE 2.30 Multiply the value 66 (expressed using 8-bit signed two's complement representation) by 2.

We start with the binary value for 66:

0 1 0 0 0 0 1 0

and we shift left one place, resulting in:

1 0 0 0 0 1 0 0

but the sign bit has changed, so overflow has occurred ($66 \times 2 = 132$, which is too large to be expressed using 8 bits in signed two's complement notation).

Arithmetic SHIFT operations

A **right arithmetic shift** moves all bits to the right, but carries (copies) the sign bit from bit b_{n-1} to b_{n-2}. Because we copy the sign bit from right to left, overflow is not a problem. However, division by 2 may have a remainder of 1; division using this method is strictly integer division, so the remainder is not stored in any way. Consider the following examples:

EXAMPLE 2.31 Divide the value 12 (expressed using 8-bit signed two's complement representation) by 2.

We start with the binary value for 12:

0 0 0 0 1 1 0 0

and we shift right one place, copying the sign bit of 0, resulting in:

0 0 0 0 0 1 1 0

which is decimal $6 = 12 \div 2$.

EXAMPLE 2.32 Divide the value 12 (expressed using 8-bit signed two's complement representation) by 4.

We start with the binary value for 12:

0 0 0 0 1 1 0 0

and we shift right two places, resulting in:

0 0 0 0 0 0 1 1

which is decimal $3 = 12 \div 4$.

EXAMPLE 2.33 Divide the value −14 (expressed using 8-bit signed two's complement representation) by 2.

We start with the two's complement representation for −14:

1 1 1 1 0 0 1 0

and we shift right one place (carrying across the sign bit), resulting in:

1 1 1 1 1 0 0 1

which is decimal $-7 = -14 \div 2$.

Note that if we had divided −15 by 2 (in Example 2.33), the result would be 11110001 shifted one to the left to yield 11111000, which is −8. Because we are doing integer division, −15 divided by 2 is indeed equal to −8.

Range extension:

Generally for compatibility we need to extend the range of integers. For example, n n-bit integer and store it in m bits, where m > n. This expansion of bit length is referred to as range extension, because the range of numbers that can be expressed is extended by increasing the bit length.

In sign-magnitude notation, this is easily accomplished: simply move the sign bit to the new leftmost position and fill in with zeros.

+18	=	00010010	(sign magnitude, 8 bits)
+18	=	0000000000010010	(sign magnitude, 16 bits)
−18	=	10010010	(sign magnitude, 8 bits)
−18	=	1000000000010010	(sign magnitude, 16 bits)

This procedure will not work for twos complement negative integers. Using the same example,

+18	=	00010010	(twos complement, 8 bits)
+18	=	0000000000010010	(twos complement, 16 bits)
−18	=	11101110	(twos complement, 8 bits)
−32,658	=	1000000001101110	(twos complement, 16 bits)

The next to last line is easily seen using the value box of Figure 10.2. The last line can be verified using Equation (10.2) or a 16-bit value box.

Instead, the rule for two's complement integers **is to move the sign bit to the new leftmost position and fill in with copies of the sign bit.** For positive numbers, fill in with zeros, and for negative numbers, fill in with ones. This is called sign extension.

−18	=	11101110	(twos complement, 8 bits)
−18	=	1111111111101110	(twos complement, 16 bits)

Multiplication in unsigned binary representation:

```
    1011      Multiplicand (11)
   ×1101      Multiplier (13)
   -----
    1011  ⎫
   0000   ⎬  Partial products
  1011    ⎪
 1011     ⎭
--------
10001111      Product (143)
```

Figure 10.7 Multiplication of Unsigned Binary Integers

Critical observations:

- Multiplication involves **the generation of partial products**, one for each digit in the multiplier. These partial products are then summed to produce the final product.
- **The partial products:** when the multiplier bit is 0, the partial product is 0. When the multiplier is 1, the partial product is the multiplicand.
- **The total product** is produced by summing the partial products. For this operation, each successive partial product is shifted one position to the left relative to the preceding partial product.
- The multiplication of two n-bit binary integers results in a product of up to 2n bits in length (e.g., 11 * 11 = 1001).
- All the partial products need not be stored, eliminates the requirement to have so many registers.
- For each 1 on the multiplier, an add and a shift operation are required; but for each 0, only a shift is required.

Multiplication in unsigned binary numbers:

1011	**Multiplicand (11)**
×1101	**Multiplier (13)**
1011	
0000	
1011	**Partial products**
1011	
10001111	**Product (143)**

Figure 10.7 Multiplication of Unsigned Binary Integers

1011	
× 1101	
00001011	$1011 \times 1 \times 2^0$
00000000	$1011 \times 0 \times 2^1$
00101100	$1011 \times 1 \times 2^2$
01011000	$1011 \times 1 \times 2^3$
10001111	

Figure 10.10 Multiplication of Two Unsigned 4-Bit Integers Yielding an 8-Bit Result

Multiplication in unsigned binary numbers:

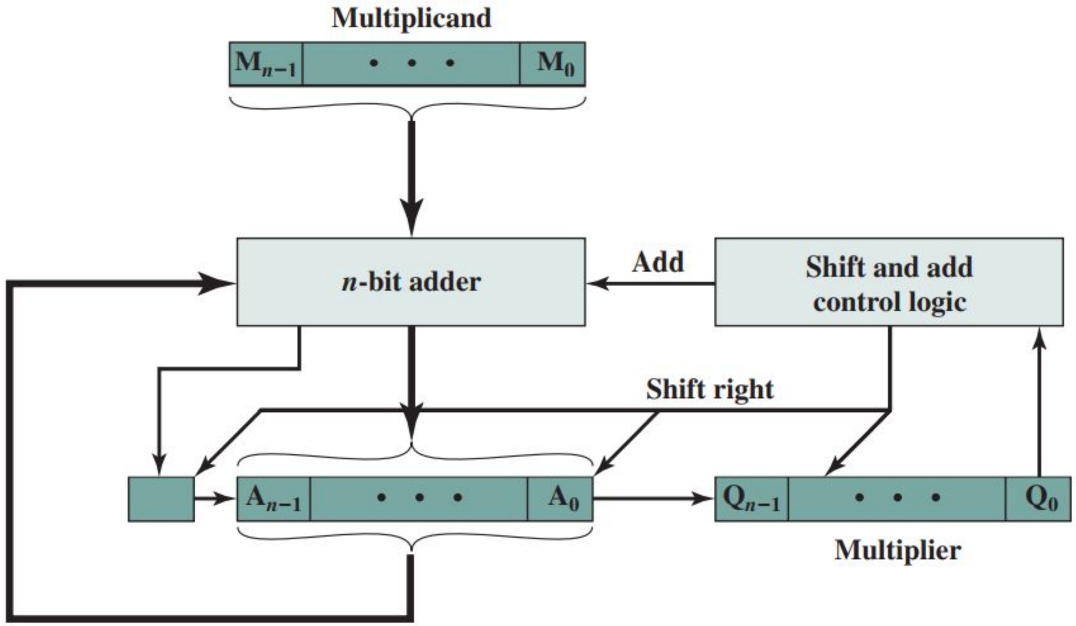
Try multiplying the following two numbers in binary:

```
     14
x    13
---------
    182
---------
```

Hardware for multiplication of unsigned binary numbers:

Setup:

- The multiplier and multiplicand are loaded into two registers (**Q** and **M**).
- A third register, the **A** register, is also needed and is **initially set to 0.**
- There is also a **1-bit C register**, initialized to 0, which holds a potential **carry bit** resulting from addition.



(a) Block diagram

Explanation:

1. **Control logic reads the bits of the multiplier one at a time.**
2. If Q0 is 1,
   - then the multiplicand is added to the A register and the result is stored in the A register, with the C bit used for overflow.
   - then all of the bits of the C, A, and Q registers are shifted to the right one bit, so that the C bit goes into An-1, A0 goes into Qn-1, and Q0 is lost.
3. If Q0 is 0,
   - then no addition is performed, just the shift.
4. Step2-4 are repeated for each bit of the original multiplier.
5. The resulting 2n-bit product is contained in the A and Q registers.

How it works? See the flow chart on the next page and verify!

Flowchart for multiplication of unsigned binary numbers:

Explanation:

1. **Control logic reads the bits of the multiplier one at a time.**
2. If Q0 is 1,
   - then the multiplicand is added to the A register and the result is stored in the A register, with the C bit used for overflow.
   - then all of the bits of the C, A, and Q registers are shifted to the right one bit, so that the C bit goes into An-1, A0 goes into Qn-1, and Q0 is lost.
3. If Q0 is 0,
   - then no addition is performed, just the shift.
4. Step2-4 are repeated for each bit of the original multiplier.
5. The resulting 2n-bit product is contained in the A and Q registers.

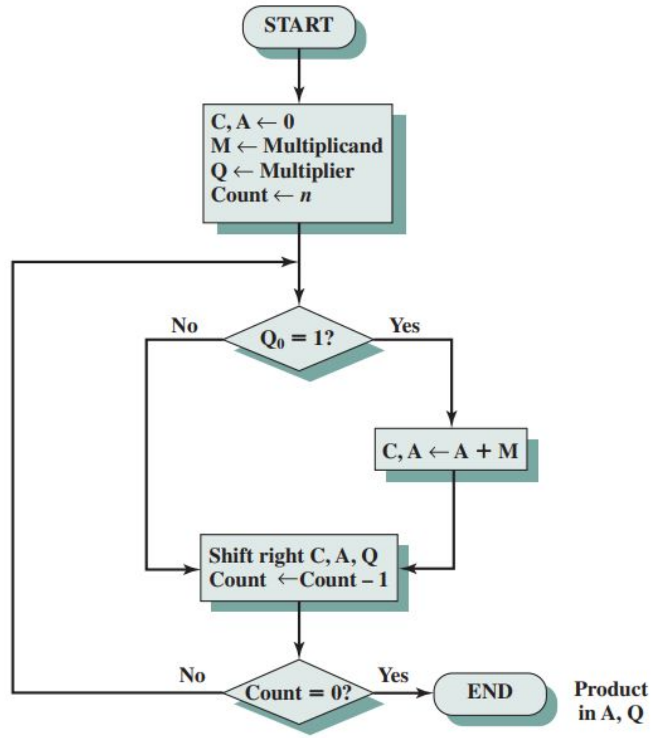


Figure 10.9 Flowchart for Unsigned Binary Multiplication

C	A	Q	M		
0	0000	1101	1011	**Initial values**	
0	1011	1101	1011	**Add**	**First**
0	0101	1110	1011	**Shift**	**cycle**
0	0010	1111	1011	**Shift**	**Second cycle**
0	1101	1111	1011	**Add**	**Third**
0	0110	1111	1011	**Shift**	**cycle**
1	0001	1111	1011	**Add**	**Fourth**
0	1000	1111	1011	**Shift**	**cycle**

Fill in the values and actions

Multiplication of signed binary numbers

```
    1011      Multiplicand (11)
   ×1101      Multiplier (13)
    1011   ⎫
   0000    ⎪
  1011     ⎬  Partial products
 1011      ⎭
10001111      Product (143)
```

Figure 10.7 Multiplication of Unsigned Binary Integers

```
    1011
  × 1101
00001011   1011 × 1 × 2^0
00000000   1011 × 0 × 2^1
00101100   1011 × 1 × 2^2
01011000   1011 × 1 × 2^3
10001111
```

Figure 10.10 Multiplication of Two Unsigned 4-Bit Integers Yielding an 8-Bit Result

Bummer! The above process won't work for multiplication of signed numbers:

- We multiplied 11 (1011) by 13 (1101) to get 143 (10001111).
- If we interpret these as two's complement numbers, we have - 5(1011) times - 3 (1101) equals - 113 (10001111).
- This example demonstrates t**hat straightforward multiplication will not work** if both the multiplicand and multiplier are negative.
- In fact, it will not work if **either the multiplicand or the multiplier is negative**. Can you verify that?

Multiplication of signed binary numbers

Why?

- The problem is that **each contribution of the negative multiplicand as a partial product must be a negative number** on a 2n-bit field; the sign bits of the partial products must line up.
- This is demonstrated in Figure 10.11, which shows that multiplication of 1001 by 0011. If these are treated as unsigned integers, the multiplication of 9 * 3 = 27 proceeds simply.
- However, if 1001 is interpreted as the two's complement value -7, then each partial product must be a negative two's complement number of 2n (8) bits, as shown in Figure 10.11b.
- Note that this is accomplished by padding out each partial product to the left with binary 1s

1011	
$\times$ 1101	
00001011	$1011 \times 1 \times 2^0$
00000000	$1011 \times 0 \times 2^1$
00101100	$1011 \times 1 \times 2^2$
01011000	$1011 \times 1 \times 2^3$
10001111	

Figure 10.10 Multiplication of Two Unsigned 4-Bit Integers Yielding an 8-Bit Result

1001	(9)
$\times$ 0011	(3)
00001001	1001×2^0
00010010	1001×2^1
00011011	(27)

(a) Unsigned integers

1001	(−7)
$\times$ 0011	(3)
11111001	$(-7) \times 2^0 = (-7)$
11110010	$(-7) \times 2^1 = (-14)$
11101011	(−21)

(b) Twos complement integers

Figure 10.11 Comparison of Multiplication of Unsigned and Twos Complement Integers

Multiplication of signed binary numbers

- If **the multiplier is negative**, straightforward multiplication will not work.
- **A typical approach:**
    - One would be to **convert both multiplier and multiplicand to positive numbers, perform the multiplication**, and then take the two's complement of the result if and only if the sign of the two original numbers differed.
- Implementers have preferred to use techniques **that do not require this final transformation step.**