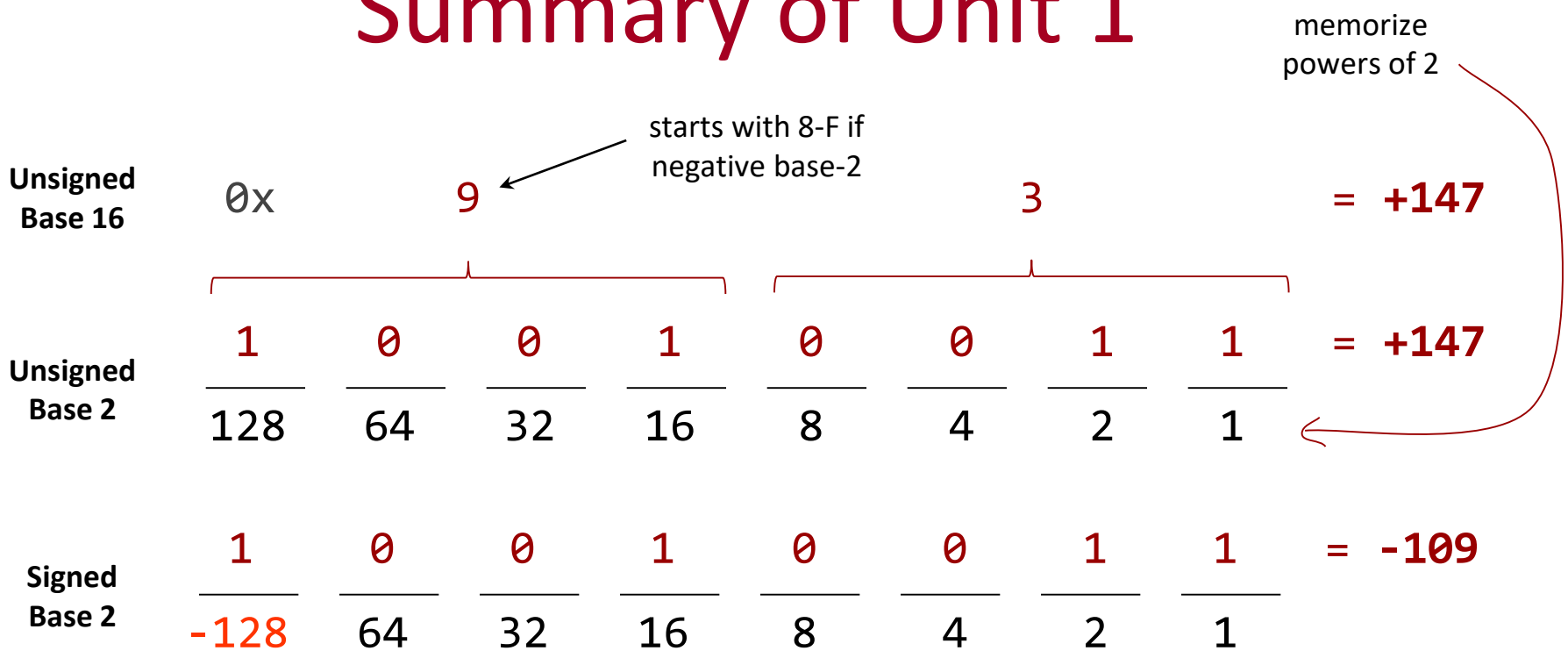
Unit 2

Integer Operations

Summary of Unit 1



- *n* bits, 2^n choices: **[0, 2^n-1]** unsigned and **[-2^{n-1}, 2^{n-1}-1]** signed range
- Unsigned MIN (**00..00**) and MAX (**11..11**)
- Signed MIN (**10..00**), -1 (**11..11**), 0 (**00..00**), 1 (**00..01**), MAX (**01..11**)
- Signed casts replicate the MSB (unsigned casts add 0's)
- C integer types: [**`unsigned`**] **`char`**, **`short`**, **`int`**, **`long`** (1, 2, 4, 8 bytes)

Skills & Outcomes

- You should master (understand + apply)
  - "+" and "-" in unsigned and 2's complement
  - Overflow detection
  - Bitwise operations
  - Logic and arithmetic shifts
    (and how to use them for multiplication/division)
  - Arithmetic in binary and hex

Binary Arithmetic

- Arithmetic operations + - * / on binary numbers
- Can use **same algorithms** as decimal arithmetic
  - **Carry when sum is 2** or more rather than 10 or more
  - **Borrow 2's** not 10's from other columns
- Recorded lecture on binary arithmetic:

```
  0 1 1 1 (7)
+ 0 0 1 1 (3)
-----------

  8 4 2 1
```

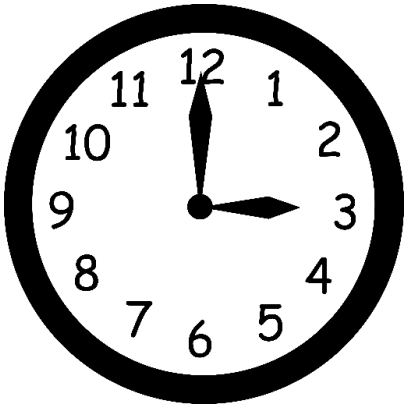
```
  1 0 1 0 (10)
- 0 1 0 1  (5)
-----------

  8 4 2 1
```

"Adding the 2's complement"

SUBTRACTION THE EASY WAY (Or why is -1 encoded as 111…11?)

Modulo Arithmetic

- _____ **precision of computers**
  Primary difference between how humans and computers perform arithmetic
  - Humans can use more digits (precision) as needed
  - Computers can only use a finite number of bits
    - Much like the odometer on your car: once you go too many miles the values will wrap from 999999 to 000000
    - Essentially all computer arithmetic is _____ **arithmetic**
    - If we have *n* bits, then all operations are **modulo** ____
- This leads to alternate approaches to arithmetic
  - Example: Consider how to change the clock time from 5 pm to 3 pm if you can't *subtract* hours, but only *add*



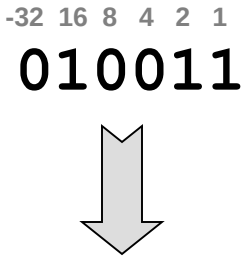
Taking the Negative

CS:APP 2.3.3

- **Question**: Given a number ***x* in 2's complement**, how do we find its **negative -*x*** ?
- **Answer**: By "taking the 2's complement"
  - Operation defined as:
    - _____
    - _____
      (i.e., finish with the _____ # of bits as we started with)
  - Example
    - 6 == 4 + 2 == **0110**
    - Flip all the bits: _____
    - Add 1: _____ == -8 + 2 == -6

Taking the 2's Complement

- Invert/flip each bit (1's complement)
  - 1's become 0's
  - 0's become 1's
- Add 1 (drop final carry-out, if any)



Original number = +19

Bit flip is called the 1's complement of a number

Resulting number = -19

Important: Taking the 2's complement is equivalent to taking the negative (negating)

Taking the 2's Complement

1

-32 16 8 4 2 1

101010

Original number = -22

Take the 2's complement yields the negative of a number

Resulting number = +22

Taking the 2's complement again yields the original number (the operation is symmetric)

101010

Back to original = -22

2

0000

Original # = 0

Take the 2's complement

2's comp. of 0 is __

3

1000

Original # = -8

Take the 2's complement

Negative of -8 is __

(no positive equivalent: overflow!)

4

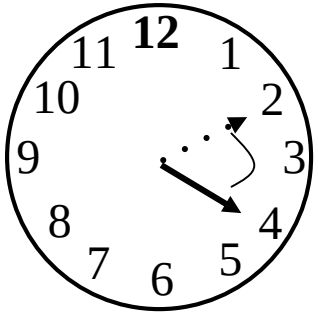
0001 => _____

2's comp. of 1 is -1

The same algorithms regardless of unsigned or signed!

ADDITION AND SUBTRACTION

Radix Complement

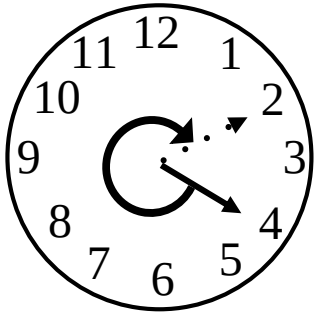


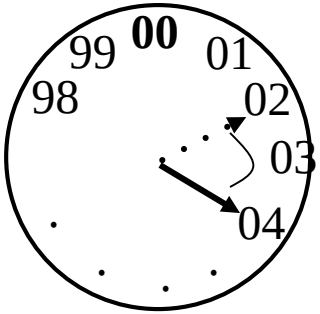
Clock Analogy

4 - 2 =

4 + (12 - 2) =

4 + 10



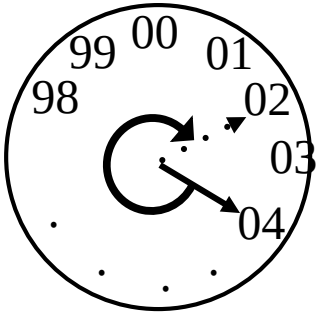


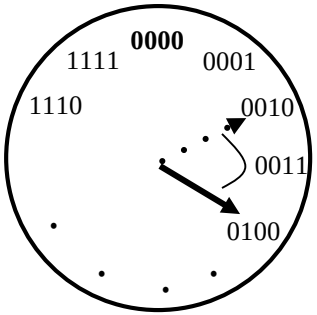
10's complement

04 - 02 =

04 + (100 - 02) =

04 + 98



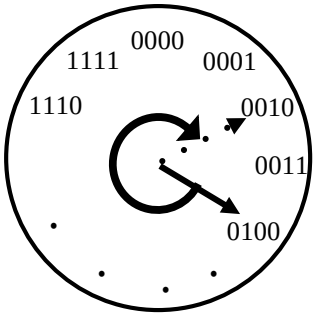


2's complement

0100 - 0010 =

0100 + (10000 - **0010**) =

0100 + **1110**



When using **modulo arithmetic**, **subtraction** can always be converted to **addition.**

2's Complement Addition/Subtraction

CS:APP 2.3.1
CS:APP 2.3.2

- **Addition**
  - Signs of the numbers do not matter
  - Add _____
  - Drop any final carry-out
    - The secret to modulo arithmetic
- **Subtraction**
  - Any subtraction (A-B) can be **converted to addition** (_____) by taking the 2's complement of B
  - (A-B) becomes (_____) *(used in DataLab)*
  - Drop any carry-out
    - The secret to modulo arithmetic

2's Complement Addition

- No matter the sign of the operands just add as normal
- Drop any extra carry out

```
  0000                0000
   0011 (3)            1101 (-3)
 + 0010 (2)          + 0010  (2)
 ------              ------
   0101 (5)            1111 (-1)


   0011  (3)           1101 (-3)
 + 1110 (-2)         + 1110 (-2)
 ------              ------
```

Unsigned and Signed Addition

- Addition process is the **same** for both unsigned and signed numbers!
  - Add columns right to left
- Examples:

	If unsigned	If signed
1001		
+ 0011		

2's Complement Subtraction

- Take the **2's complement of the subtrahend** (bottom #) and add to the original minuend (top #)
- Drop any extra carry out

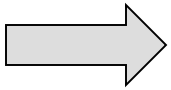
```
   0011 (+3)
 - 0010 (+2)
 ----------
```

```
   1101 (-3)
 - 1110 (-2)
 ----------
```

Unsigned and Signed Subtraction

- Subtraction process is the same for both unsigned and signed numbers!
  - Convert A – B to A + (2's complement of B)
  - Drop any final carry out
- Examples:

	If unsigned	If signed
1100	(12)	(-4)
- 0010	(2)	(2)



If unsigned If signed

Important Note

- Almost all computers use 2's complement because…
  - The same addition and subtraction _____ can be used on unsigned and 2's complement (signed)
  - Thus we only need _____ (HW component) to perform operations on both unsigned and signed numbers

MAX + 1 = MIN

OVERFLOW

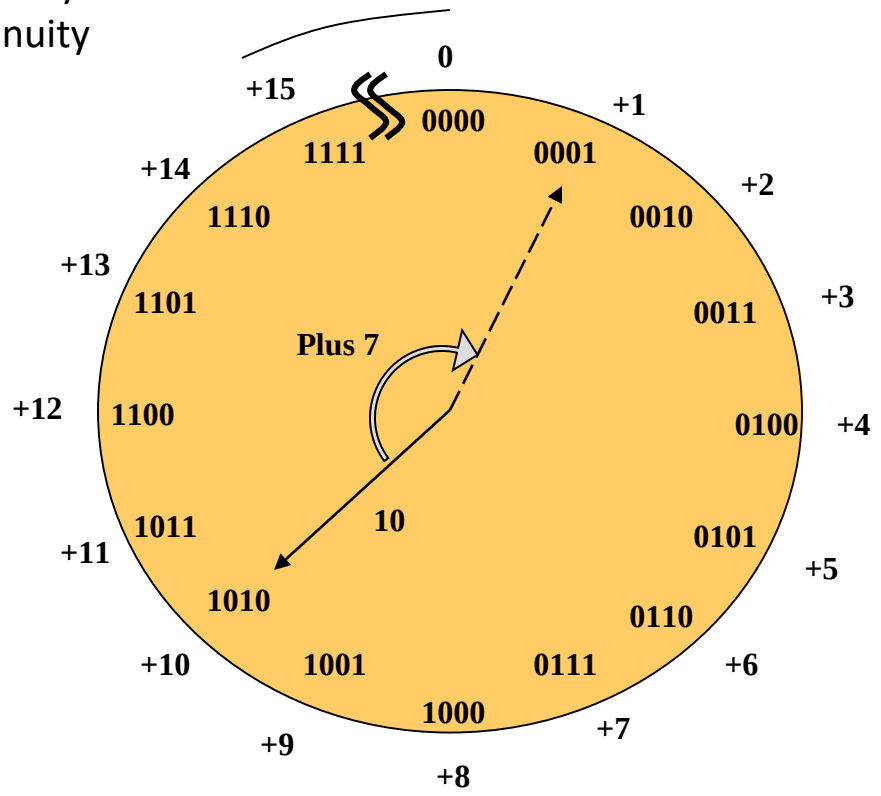
Overflow

CS:APP 2.3.1
CS:APP 2.3.2

- When the result of an arithmetic op. is too large / too small to be represented with the given number of bits
- Different algorithms for detecting overflow based on unsigned or signed
  - _____ / _____ (unsigned)
  - _____ / _____ (signed)

Unsigned Overflow

Overflow occurs when you cross this discontinuity



10 + 7 = 17

With 4-bit *unsigned* numbers we can only represent 0 – 15. Thus, we say overflow has occurred.

The result is 1 (17 mod 16), not 17!

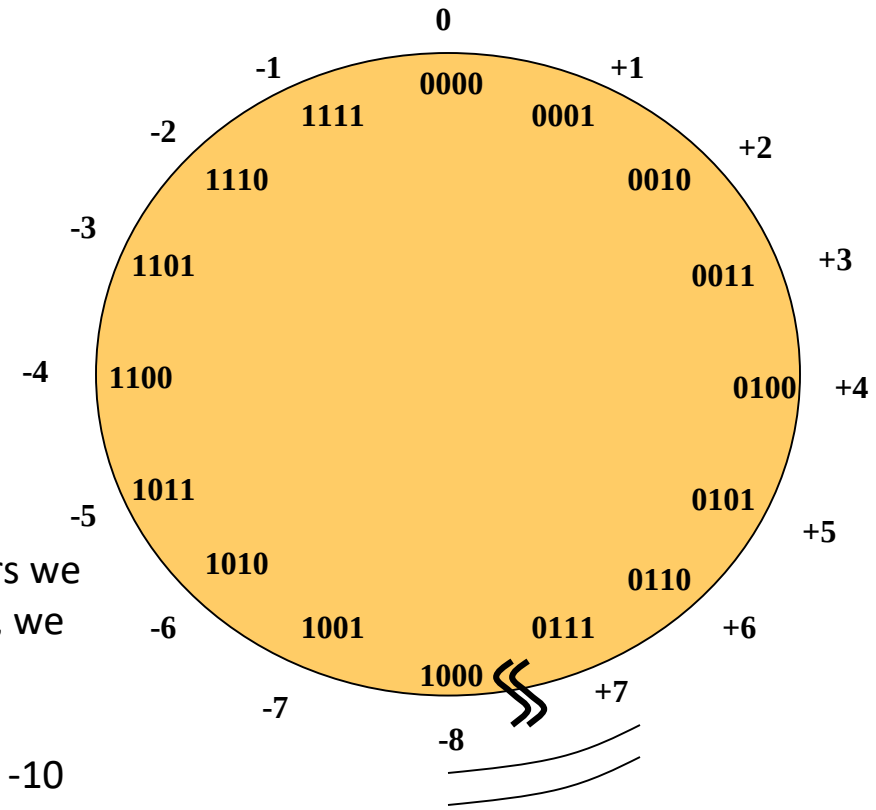
Signed Overflow

5 + 7 = +12

-6 + -4 = -10

With 4-bit *2's complement* numbers we can only represent -8 to +7. Thus, we say overflow has occurred.

The result is -4, not 12; and 6, not -10



Overflow occurs when you cross this discontinuity

Overflow in Addition

- Overflow occurs when the result of the addition cannot be represented with the given # of bits
- Tests for overflow:
  - Unsigned: if _____ [result smaller than inputs]
  - Signed: if **p+p=n** or **n+n=p** [result has inappropriate sign]

Cout		If unsigned	If signed
1 1			
	1101	(13)	(-3)
+	0100	(4)	(4)
	0001	(17)	(+1)
		Overflow	No Overflow
		Cout = 1	n + p

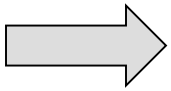
Cout		If unsigned	If signed
0 1			
	0110	(6)	(6)
+	0101	(5)	(5)
	1011	(11)	(-5)
		No Overflow	Overflow
		Cout = 0	p + p = n

Overflow in Subtraction

- Overflow occurs when the result of the subtraction cannot be represented with the given # of bits
- Tests for overflow:
  - Unsigned: if _____ [expect negative result]
  - Signed: if **p+p=n** or **n+n=p** [result has inappropriate sign]

	If unsigned	If signed
0111	(7)	(7)
− 1000	(8)	(−8)
	(−1)	(15)

Desired Results



Cout		
0111		
0111	A	
0111	1's comp. of B	
+ 1	Add 1	
1111	(15)	(−1)
	If unsigned Overflow Cout = 0	**If signed Overflow p + p = n**

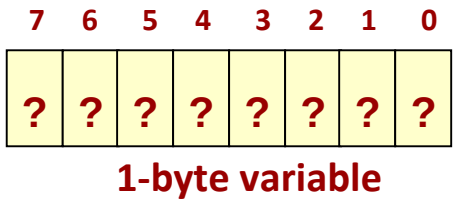
4-bit range: [0,15] or [-8,7]

BITWISE & LOGIC OPERATIONS

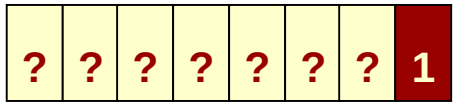
Modifying Individual Bits

CS:APP 2.1.7

- Suppose we want to **change only a single bit** (or a few bits) in a variable (i.e., a `char v`) _____ the other bits
  - Set the LSB of v to 1 w/o affecting other bits
    - Would this work? `v = 1;`
  - Set the upper 4 bits of v to 1111 w/o affecting other bits
    - Would this work? `v = 0xf0;`
  - Clear the lower 2 bits of v to 00 w/o affecting other bits
    - Would this work? `v = 0;`
  - _____**!!! Assignment changes _____ bits in a variable**
- The smallest unit of data in computers is usually a _____, so **manipulating individual bits requires bitwise operations**
  - **BITWISE AND = __** (different from "&&")
  - **BITWISE OR = __** (different from "||")
  - **BITWISE XOR = __** (different from "!=")
  - **BITWISE NOT = __** (different from "!")

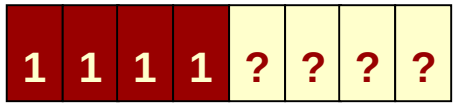


1-byte variable



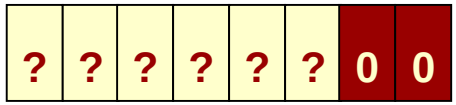
Desired v

(change LSB to 1)



Desired v

(change upper 4 bits to 1111)



Desired v

(change lower 2 bits to 00)

Bitwise operations change bits

- **ANDs** can be used to ____ **a bit** (make it '_') or leave it unchanged
- **ORs** can be used to ___ **a bit** (make it '_') or leave it unchanged
- **XORs** can be used to _____ **a bit** (_____ it) or leave it unchanged

mask

X	Y	AND
0	0	0
0	1	0
1	0	0
1	1	1

Force '0'
Pass

0 AND y = 0
1 AND y = y
y AND y = y

mask

X	Y	OR
0	0	0
0	1	1
1	0	1
1	1	1

Pass
Force '1'

0 OR y = y
1 OR y = 1
y OR y = y

mask

X	Y	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Invert
Pass

0 XOR y = y
1 XOR y = NOT y
y XOR y = 0

Identity	**1 AND Y = Y**	**0 OR Y = Y**
Null Ops	**0 AND Y = 0**	**1 OR Y = 1**
Idempotency	**Y AND Y = Y**	**Y OR Y = Y**

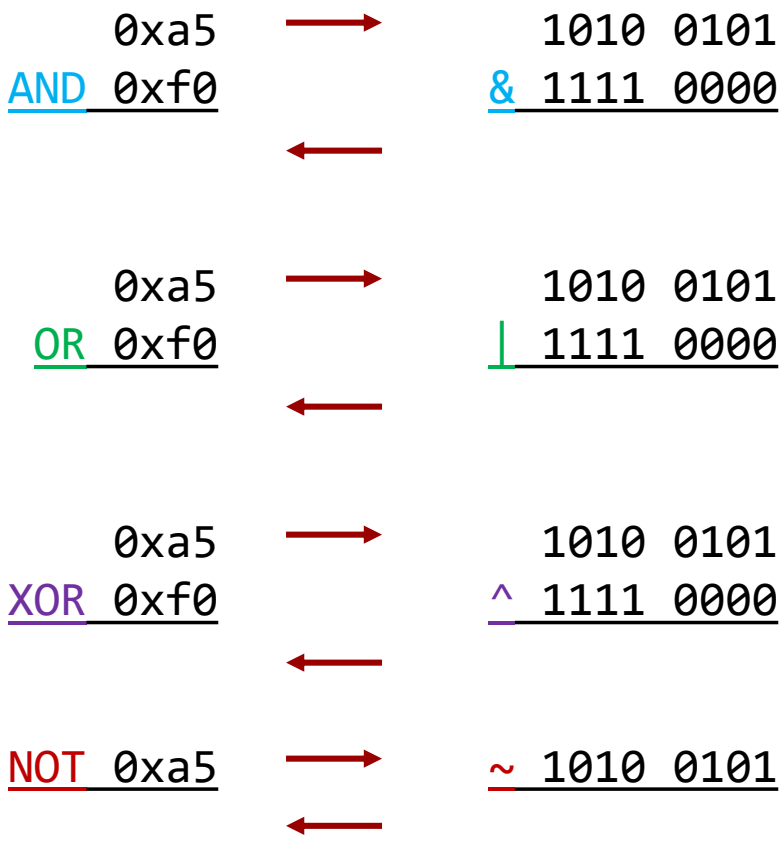
0 XOR Y = Y
1 XOR Y = ~Y
Y AND Y = 0

(x XOR y) XOR y = x
(x XOR y) = 1 when when bits are different

Bitwise Operations

CS:APP 2.1.7

- In C, bitwise AND, OR, XOR, NOT perform the same operation on each pair of bits of 2 numbers




```c
#include <stdio.h> // C-Library
                   // for printf()

int main()
{
  char a = 0xa5;
  char b = 0xf0;

  printf("a & b = %x\n", a & b);
  printf("a | b = %x\n", a | b);
  printf("a ^ b = %x\n", a ^ b);
  printf("~a = %x\n",    ~a);
  return 0;
}
```

C bitwise operators:

& = AND
| = OR
^ = XOR
~ = NOT

Logical vs. Bitwise Operations

CS:APP 2.1.8

- The C language has two types of logic operations
  - Logical and Bitwise
- **Logical Operators** (&&, ||, !=, !)
  - Interpret entire value as ____ (**non-zero**) or ____ (**zero**), **return 1 or 0**
- **Bitwise Operators** (&, |, ^, ~)
  - Apply the logical operation on each pair of bits of the inputs

```
#include <stdio.h>
int main() {
  int x = 1, y = 2;
  int z1 = x && y;  // 1
  int z2 = x & y;   // 0
  printf("z1=%d, z2=%d\n",z1,z2);

  char z = 1;
  if(!z) { printf("L1\n"); } // F
  if(~z) { printf("L2\n"); } // T
  return 0;
}
```

```
  0000 0001=T
&& 0000 0010=T
```

```
  0000 0001
& 0000 0010
```

```
! 0000 0001=T
  0000 0000=F
```

```
~ 0000 0001
```

```
!! 0101 0111=T
   0000 0001=T
```

Important Note: Since !(non-zero) = 0; and !0 = 1
we have !!35=1 and !!-109=1 … !!(x) == (x != 0)

Application: Swapping via XORs

- Swapping variables can be done with a temporary variable
- For bitwise swapping, XORs can be used (classic programming interview question)

```
#include <stdio.h>

int main() {
  int x = 0x59;
  int y = 0xd3;
  int temp = x;   // save x
  x = y;          // overwrite x
  y = temp;       // overwrite y
}
```

Traditional swap with 'temp'

XOR swap

```
#include <stdio.h>
int main() {
  int x = 0x59;
  int y = 0xd3;
  x = x ^ y;  // x = bitwise diff
  y = x ^ y;  // flip y if different
  x = x ^ y;  // flip y if flipped
}
```

```
  0101 1001=x
```

```
  1101 0011=y
```

```
  0101 1001=x
^ 1101 0011=y
```

```
  1000 1010=x
^ 1101 0011=y
```

```
  1000 1010=x
^ 0101 1001=y
```

Exercises

- Check if an integer is odd (without % operator)

```
int isOdd(int x) {
  /* Isolate the lowest bit */
  ____________________;
}
```

- Check if an integer is a multiple of 4 (without % operator)

```
int isMultOf4(int x) {
  /* Check if 2 LSBs are both 0 */
  ____________________;
}
```

Arithmetic and Logical Shifts

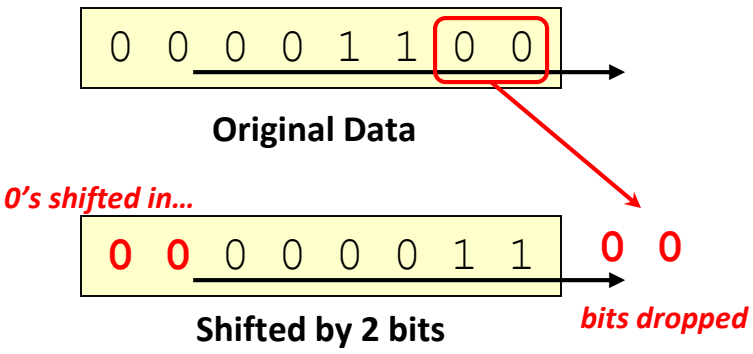
SHIFT OPERATIONS

Shift Operations

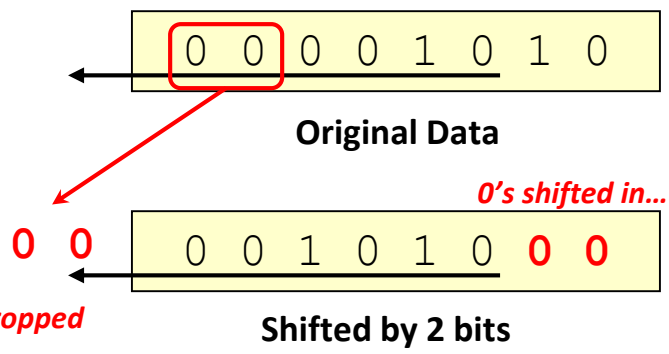
CS:APP 2.1.9

- Shifts data bits either **left** or **right**
  - Bits shifted out and **dropped on one side**
  - **Usually (but not always) 0's are shifted in** on the other side
- Shifting is equivalent to multiplying or dividing by _____
- 2 kinds of shifts
  - _____ shifts (used for **unsigned** numbers)
  - _____ shifts (used for **signed** numbers, replicate MSB)

Right Shift by 2 bits:



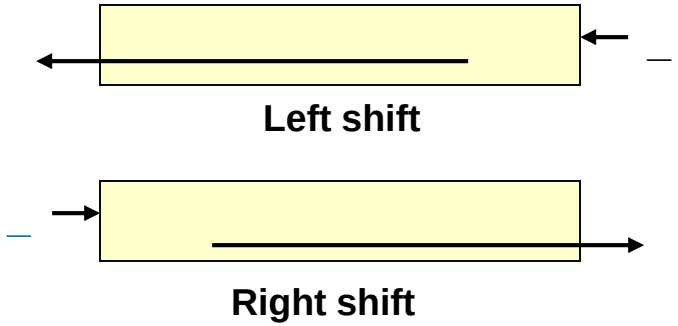
Left Shift by 2 bits:

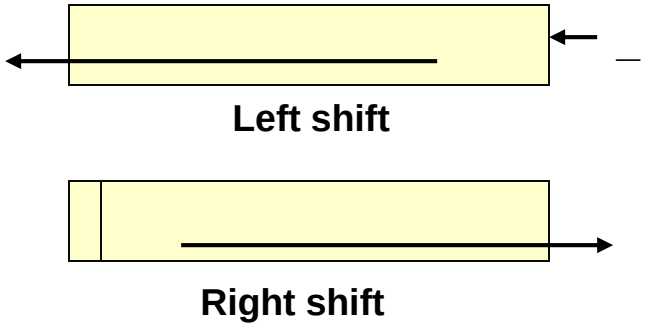


Logical Shift vs. Arithmetic Shift

- **Logical Shift**
  - Use for _____ or non-numeric data
  - Will always shift in ___'s (left and right shift)

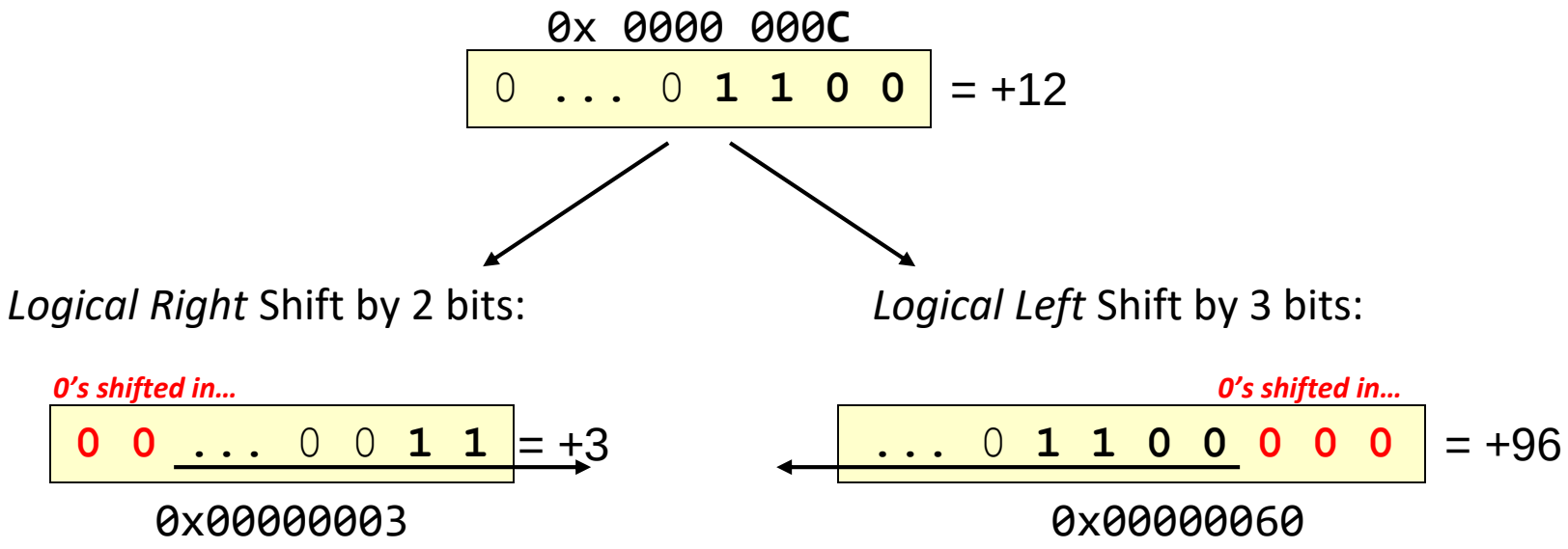
- **Arithmetic Shift**
  - Use for **signed** data
  - Left shift will shift in 0's
  - **Right shift will** _____ (replicate the ____ bit) rather than shift in 0's
    - If negative number…stays negative by shifting in __'s
    - If positive…stays positive by shifting in __'s





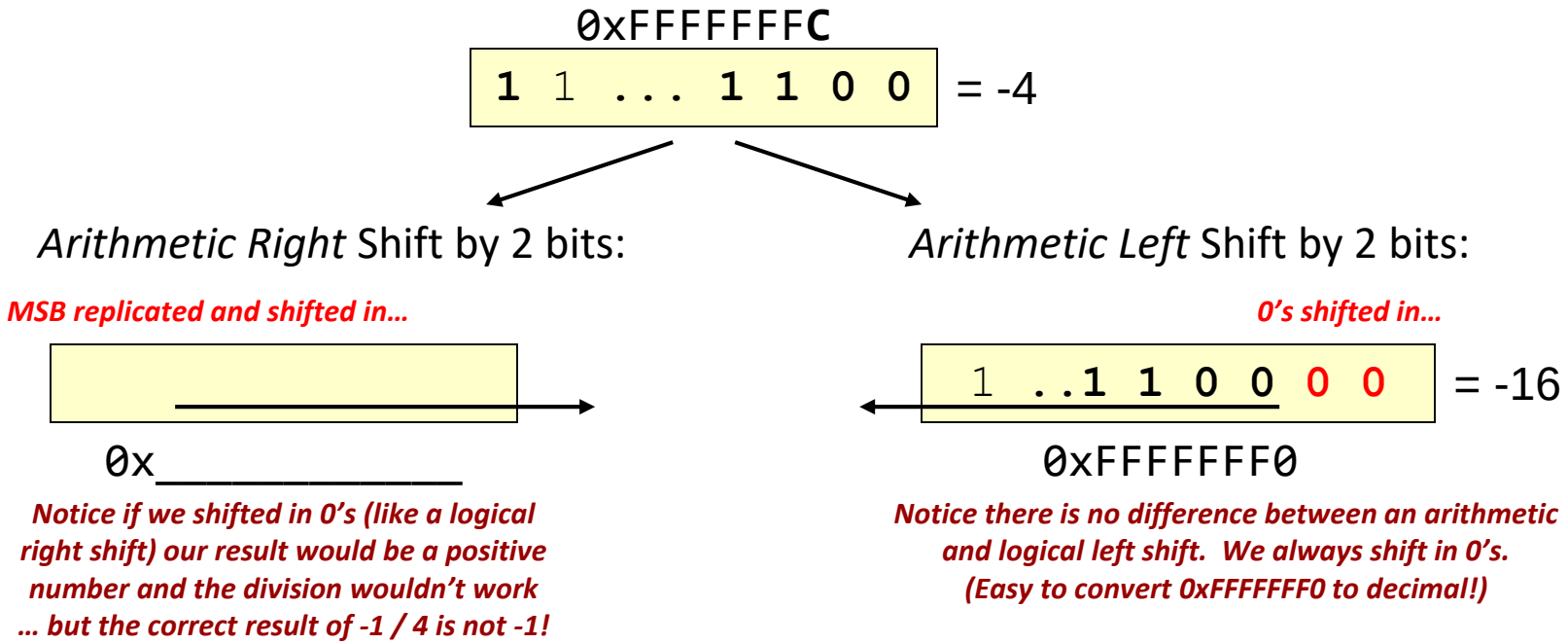
Logical Shift

- 0's shifted in
- Only use for operations on **unsigned** data
  - Right shift by *n* bits = **Dividing by 2^n**
  - Left shift by *n* bits = **Multiplying by 2^n**



Arithmetic Shift

- Use for operations on *signed* data
- Arithmetic Right Shift – replicate MSB
  - Right shift by *n* bits = Dividing by 2^n (**may need biasing**)
- Arithmetic Left Shift – shifts in 0's
  - Left shift by *n* bits = Multiplying by 2^n



Multiplying by Non-Powers of 2

CS:APP 2.3.6

- Left shifting by *n*-bits allow us to multiply by 2^n
- But what if I have to multiply a number by a *non-power* of 2 (e.g., 17*x). Can we still use shifting?
  - _____! Break constant into a _____ using power of 2 coefficients:
    17x = _____
- Exercise: How many adds/shift are needed to compute 14*x ?
  - _____ (3 shifts, 2 adds)
  - _____ (2 shift and 1 add)

$$17 = \underset{16}{1}\ \underset{8}{0}\ \underset{4}{0}\ \underset{2}{0}\ \underset{1}{1}_2$$

```
int mul17(int x) {
  return 17*x;
}
```

Written Code

```
 sall    $4, %edx
 addl    %edx, %eax
```

```
int mul17(int x) {


}
```

Optimized Assembly
(Equivalent C)

Compiler will determine when shifts / adds become faster than constant multiplication

Integer Division By Shifting

CS:APP 2.3.7

- What is 5/2?
  - +2
- Is 5/2 = (5 >> 1)?
  - Yes

5 = 0 1 0 1

-8 4 2 1

5>>1 = 0 0 1 0 . 1

-8 4 2 1 0.5

- What is -5/2?
  - -2
- Is -5/2 = (-5 >> 1)?
  - _____

-5 = 1 0 1 1

-8 4 2 1

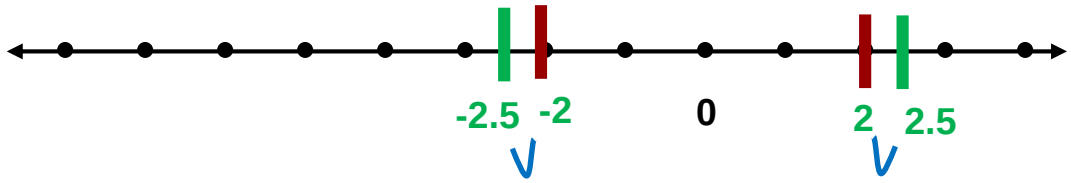
-5>>1 = 1 1 0 1 . 1

-8 4 2 1 0.5

Main Point: Rounding _____ when using shifting to divide a _____ number.

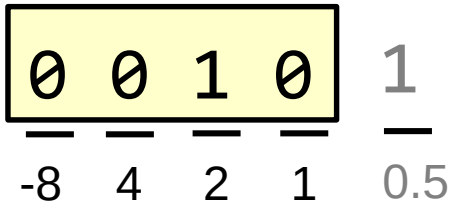
Dividing Negative Numbers

Traditional integer rounding

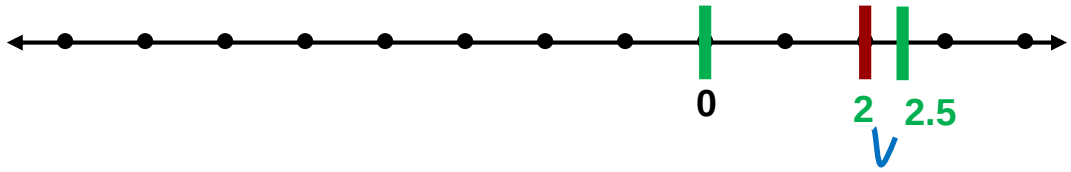


Traditional integer division rounds toward 0 (i.e., drops fractional portion)

+5>>1

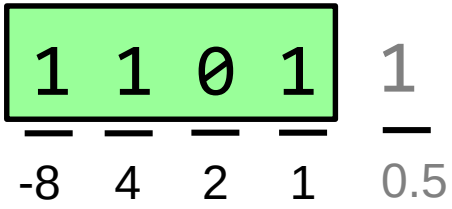


5 >> 1

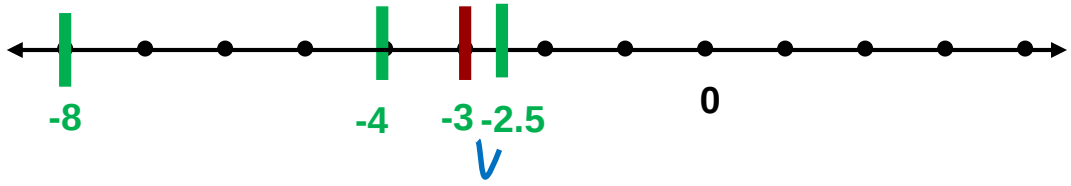


Rounding (by dropping fractional portion)

-5>>1



-5 >> 1



Rounding (by dropping fractional portion) decreases the value

Main Point: Dividing numbers in the 2's complement system causes rounding to the next smallest integer, not toward 0 as desired

Biasing

- Dividing by 2^k with **x >> k**
  - Works if $x \geq 0$ OR ($x < 0$ AND x is a multiple of 2^k)
  - Doesn't work if $x < 0$ AND x is NOT a multiple of 2^k
- Idea to solve the problem:
  - Add some value (aka a ______ value) to x __________ shifting that will correct for the rounding issue
  - Add ____, i.e., sequence of _ ones (no effect on multiples of 2^k)

```
   -4 = 1 1 0 0
-4>>1 = 1 1 1 0   -2

   -5 = 1 0 1 1
-5>>1 = 1 1 0 1   -3

  -5     1 0 1 1
  +_   + _______

_____ = 1 1 1 0   -2
```

More Examples

- (-8 / 4) == (-8 >> 2)
  - Bias by _____
  - (-8 + __) >> 2

bias not needed
(but doesn't hurt)

```
  -8 = 1 0 0 0
-8>>2 = 1 1 1 0  -2

  -8     1 0 0 0
  +__    +_______
```

- (-7 / 4) != (-7 >> 2)
  - Bias by _____
  - _____

bias needed

```
__>>2 = 1 1 1 0  -2

  -7 = 1 0 0 1
-7>>2 = 1 1 1 0  -2
```

- (-20 / 16) != (-20 >> 4)
  - Bias by _____
  - _____

bias needed

```
  -7     1 0 0 1
  +__    +_______

__>>2 = 1 1 1 1  -1
```

CS:APP Practice 2.43 (tweaked)

```
#define M /* mystery number 1 */
#define N /* mystery number 2 */

int arith(int x, int y) {
  int result = x*M + y/N;
  return result;
}

/* Translation of assembled code for
   a given value of M and N */

int optarith(int x, int y) {
  int t = x;
  x <<= 5;
  x -= t;
  if (y < 0) y += 3;
  y >>= 2;
  return x + y;
}
```

What were M and N when the code was compiled? (M = ____, N = ___)

Recorded Lecture

Arithmetic Operations in Binary

Carry is 2, not 10

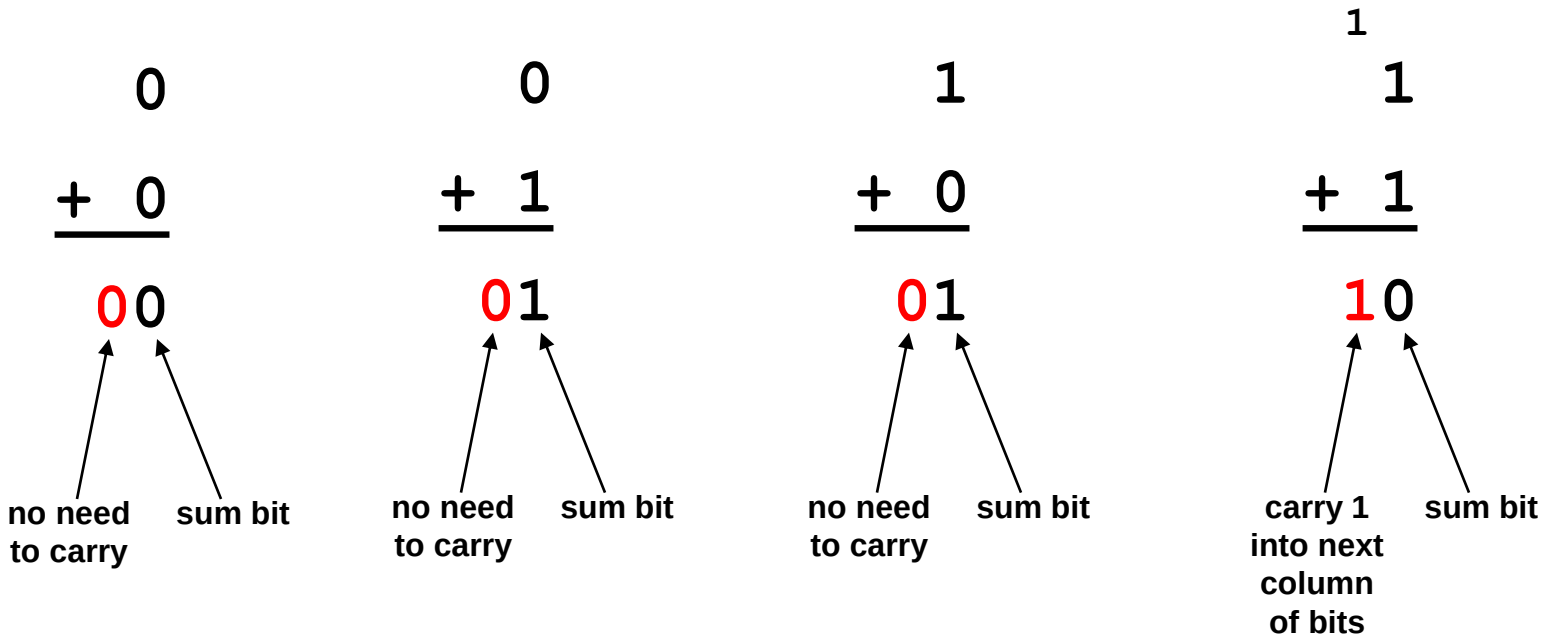
UNSIGNED BINARY ARITHMETIC

Binary Arithmetic

- Arithmetic operations + - * / on binary numbers
- Can use **same algorithms** as decimal arithmetic
  - Use carries and borrows
  - **Carry when sum is 2** or more rather than 10 or more
  - **Borrow 2's** not 10's from other columns
- Easiest method: add bits in your head in decimal (1+1 = 2) then convert the answer to binary ($2_{10} = 10_2$)

Binary Addition

- Decimal addition: Carry when the sum is 10 or more
- Binary addition: Carry when the sum is 2 or more
- Add bits in binary to produce a sum bit and a carry bit



Binary Addition & Subtraction

```
  1  1  1
  0  1  1  1 (7)
+ 0  0  1  1 (3)
-------------
  1  0  1  0 (10)
  8  4  2  1
```

```
    0     0
  ~~1~~ 1 0 ~~1~~ 1 0 (10)
- 0  1  0  1   (5)
-------------
  0  1  0  1   (5)
  8  4  2  1
```

Binary Addition: Try it!

```
   0110 (6)
   8421
 + 0111 (7)
 ---------
        (   )
```

Binary Addition

```
   110
   0110 (6)
   8 4 2 1
 + 0111 (7)
 ---------
   1101 (13)
```

Binary Addition

1

```
  0
 0110 (6)
+ 0111 (7)
-----------
 1101 (13)
```

```
  0
+ 1
----
 01
```

carry bit — sum bit

2

```
 10
 0110 (6)
+ 0111 (7)
-----------
 1101 (13)
```

```
  0
  1
+ 1
----
 10
```

carry bit — sum bit

3

```
 110
 0110 (6)
+ 0111 (7)
-----------
 1101 (13)
```

```
  1
  1
+ 1
----
 11
```

carry bit — sum bit

4

```
 110
 0110 (6)
+ 0111 (7)
-----------
 1101 (13)
```

```
  1
  0
+ 0
----
 01
```

carry bit — sum bit

Hexadecimal Arithmetic

- Same style of operations
  - Carry when sum is 16 or more, etc.

$$\begin{array}{r} {\scriptstyle 1\ \ 1\ \ \ } \\ 4\ D_{16} \\ +\ B\ 5_{16} \\ \hline 1\ 0\ 2_{16} \\ {\scriptstyle 256\ \ 16\ \ \ 1} \end{array}$$

$13+5 = 18_{10} = \underline{1}\ \underline{2}_{16}$
16 1

$1+4+11 = 16_{10} = \underline{1}\ \underline{0}_{16}$
16 1

MULTIPLICATION AND DIVISION

Binary Multiplication

- Like decimal multiplication, compute partial products, shift them, then tally up
  - 3-5x more expensive on modern CPUs)
- **Multiplying two *n*-bit numbers yields at most a (2**n*)-bit product**

```
        0 1 1 0 (6)
    *   0 1 0 1 (5)
  -----------------
        0 1 1 0
      0 0 0 0
    0 1 1 0
+ 0 0 0 0
  -----------------
  0 0 1 1 1 1 0
```

Partial Products

Sum of the partial products

Unsigned Multiplication Review

- Same rules as decimal multiplication
- Multiply each bit of Q by M shifting as you go
- An (*m*-bit) x (*n*-bit) mult. produces an (*m*+*n*) bit result
- Each partial product is a shifted copy of M or 0 (zero)

```
      1010   M (Multiplicand)
    * 1011   Q (Multiplier)
    ------
      1010
     1010_   PP(Partial
     0000__     Products)
  +  1010___
    ---------
    01101110   P (Product)
```

Signed Multiplication Techniques

- When multiplying signed (2's comp.) numbers, some new issues arise
- Must **sign extend partial products** (out to 2*n* bits)

Without Sign Extension…
Wrong Answer!

```
    1001  = -7
  * 0110  = +6
  ------
    0000
   1001_
  1001__
+ 0000___
  ------
  00110110  = +54
```

With Sign Extension…
Correct Answer!

```
     1001  = -7
   * 0110  = +6
   -------
  00000000
  1111001_
  111001__
+ 00000___
  -------
  11010110  = -42
```

Signed Multiplication Techniques

- Also, must worry about **negative multiplier**
  - MSB of multiplier has negative weight
  - **For MSB=1, take 2's comp. of multiplicand** (and extend to *n* bits)

With Sign Extension but w/o consideration of MSB…
Wrong Answer!

```
    1100   =  -4
  * 1010   =  -6
  --------
  00000000
  1111100_
  000000__
+ 11100___
  --------
  11011000   =  -40
```

With Sign Extension and w/ consideration of MSB…
Correct Answer!

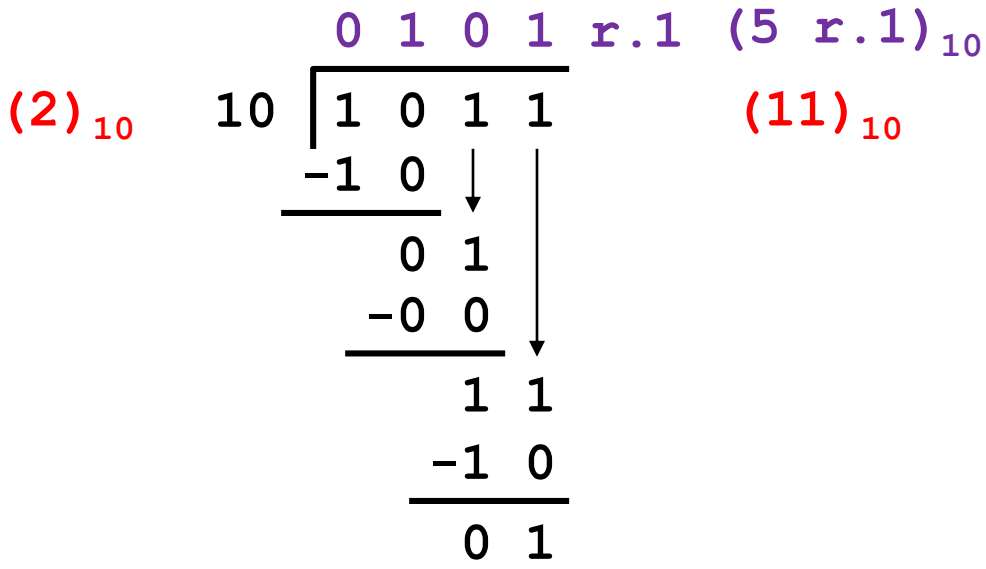
Place Value: -8
Multiply -4 by -1

```
    1100   =  -4
  * 1010   =  -6
  --------
  00000000
  1111100_
  000000__
+ 00100___
  --------
  00011000   =  +24
```

Main Point: Signed and Unsigned Multiplication require different techniques… Thus different assembly instructions.

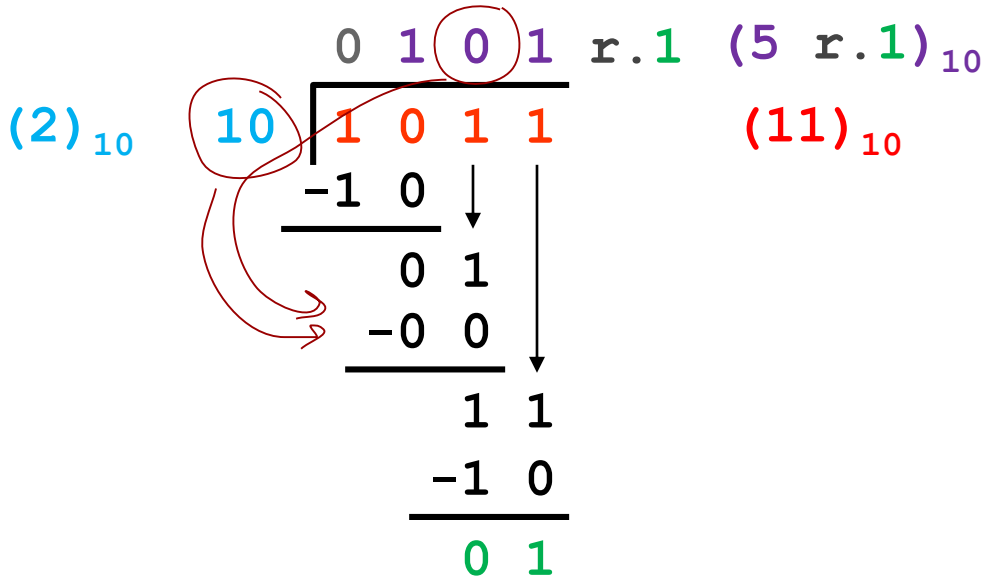
Binary Division

- **Dividing two *n*-bit numbers may yield an *n*-bit quotient and *n*-bit remainder**
- Division operations on a modern processor can take **17-41 times** longer than addition operations



Binary Division

- Use the same algorithm as in decimal
- Divide 11 by 2:



Binary Division

10 (2) goes into 1, 0 times. Since it doesn't, bring in the next bit.

```
      0
   ┌─────
10 │ 1 0 1 1
```

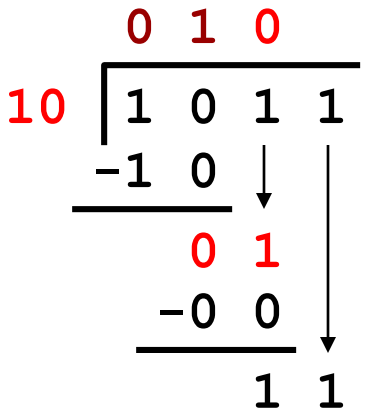
Binary Division

10 (2) goes into 10, 1 time. Multiply, subtract, and bring down the next bit.

```
        0  1
      ┌─────────
10    │ 1  0  1  1
      -1  0  ↓
      ──────
         0  1
```

Binary Division

10 (2) goes into 01, 0 times. Multiply, subtract, and bring down the next bit.



Binary Division

10 (2) goes into 11, 1 time. Multiply and subtract. The remainder is 1.

```
          0  1  0  1  r.1
       ┌──────────────
  10   │ 1  0  1  1
      -1  0  ↓
      ──────
          0  1  ↓
         -0  0  ↓
         ──────
             1  1
            -1  0
            ─────
             0  1
```