Unit 2

Integer Operations

(Arithmetic, Overflow, Bitwise Logic, Shifting)

Skills & Outcomes

- You should know and be able to apply the following skills with confidence
  - Perform addition & subtraction in unsigned & 2's complement system
  - Determine if overflow has occurred
  - Perform bitwise operations on numbers
  - Perform logic and arithmetic shifts and understand how they can be used for multiplication/division
  - Understand arithmetic in binary and hex

UNSIGNED BINARY ARITHMETIC

Binary Arithmetic

- Can perform all arithmetic operations (+,-,*,÷) on binary numbers
- Can use same methods as in decimal
  - Still use carries and borrows, etc.
  - Only now we carry when sum is 2 or more rather than 10 or more (decimal)
  - We borrow 2's not 10's from other columns
- Easiest method is to add bits in your head in decimal (1+1 = 2) then convert the answer to binary ($2_{10} = 10_2$)

Binary Addition

- In decimal addition we carry when the sum is 10 or more
- In binary addition we carry when the sum is 2 or more
- Add bits in binary to produce a sum bit and a carry bit

```
    0        0        1       1
                              1
  + 0      + 1      + 0     + 1
  ---      ---      ---     ---
   00       01       01      10
```

Example	Left (red) bit	Right bit
0 + 0 = 00	no need to carry	sum bit
0 + 1 = 01	no need to carry	sum bit
1 + 0 = 01	no need to carry	sum bit
1 + 1 = 10	carry 1 into next column of bits	sum bit

Binary Addition & Subtraction

$$
\begin{array}{cccccc}
 & 1 & 1 & 1 & & \\
 & 0 & 1 & 1 & 1 & (7) \\
+ & 0 & 0 & 1 & 1 & (3) \\
\hline
 & 1 & 0 & 1 & 0 & (10) \\
 & 8 & 4 & 2 & 1 &
\end{array}
\qquad\qquad
\begin{array}{cccccc}
 & 0 & & 0 & & \\
 & \cancel{1} & {}^{1}0 & \cancel{1} & {}^{1}0 & (10) \\
- & 0 & 1 & 0 & 1 & (5) \\
\hline
 & 0 & 1 & 0 & 1 & (5) \\
 & 8 & 4 & 2 & 1 &
\end{array}
$$

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.

$$
\begin{array}{rrrl}
1 & 1 & & \\
 & 4 & D_{16} & \\
+ & B & 5_{16} & \\
\hline
1 & 0 & 2_{16} &
\end{array}
$$

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

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

Binary Multiplication

- Like decimal multiplication, find each partial product and shift them, then sum them up
- **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

Binary Division

- Use the same long division techniques as in decimal
- **Dividing** **two *n*-bit numbers** **may yield an n-bit quotient and n-bit remainder**

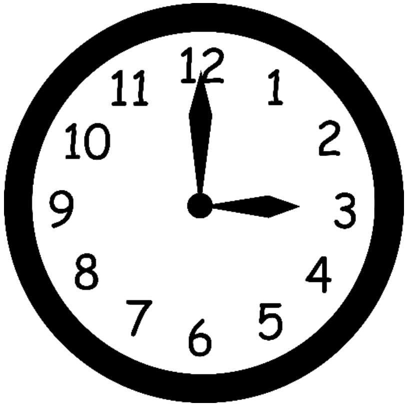
```
                 0 1 0 1 r.1  (5 r.1)10
                ________
(2)10      10 | 1 0 1 1          (11)10
               -1 0
               ----
                  0 1
                 -0 0
                 ----
                    1 1
                   -1 0
                   ----
                    0 1
```

"Taking the 2's complement"

SUBTRACTION THE EASY WAY

Modulo Arithmetic

- The primary difference between how humans and computers perform arithmetic is the finite precision of computers
  - As humans we can use more digits (precision) as needed
  - Computers can only used a finite set 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 modulo arithmetic
    - If we have a width of w bits, then all operations are module 2^w
- This leads to alternate approaches to arithmetic
  - Example: Consider how you could change the clock time from 5 p.m. to 3 p.m. if you can't subtract hours



Taking the Negative

CS:APP 2.3.3

- **Question**: Given a number in 2's complement how do we find its negative (i.e. -1 * X)
- **Answer**: By "taking the 2's complement"
  - 0110 = +6 => -6 = 1010
  - Operation defined as:
    1. Flip/invert/not all the bits (1's complement)
    2. Add 1 and drop any carry (i.e. finish with the same # of bits as we start with)
  - See next slides for example

Taking the 2's Complement

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

```
-32 16  8  4  2  1
  0  1  0  0  1  1

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

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

```
  010101
+      1
--------
  010110
```

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

Resulting number = +22

```
  101001
+      1
--------
  101010
```

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

Back to original = -22

2

```
  0000
```

Original # = 0

```
  1111
+    1
------
  0000
```

Take the 2's complement

2's comp. of 0 is 0

3

```
  1000
```

Original # = -8

```
  0111
+    1
------
  1000
```

Take the 2's complement

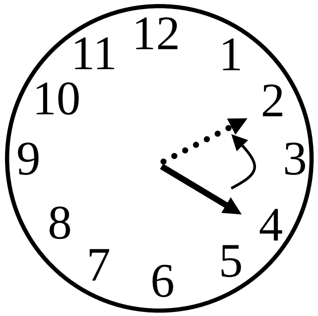
Negative of -8 is -8

(i.e. no positive equivalent, but this is not a huge problem)

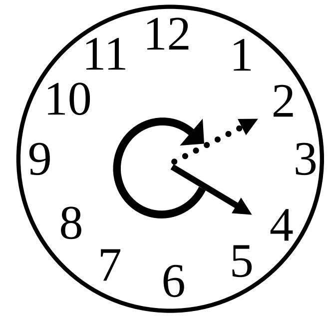
The same algorithms regardless of unsigned or signed

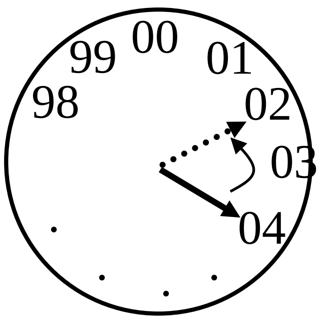
ADDITION AND SUBTRACTION

Radix Complement

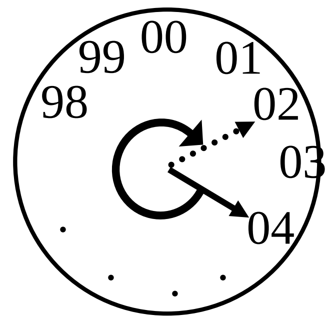


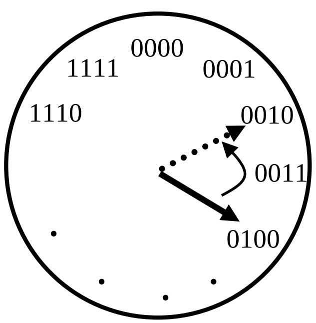
Clock Analogy
4-2 = 4+10



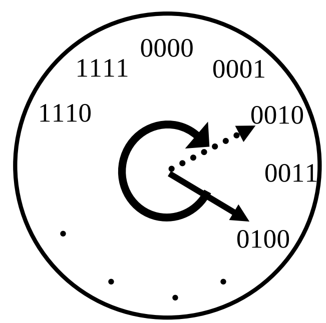


10's complement
04-02 = 04 + 98





2's complement
0100 - 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
  - Sign of the numbers do not matter
  - Add column by column
  - Drop any final carry-out
    - The secret to modulo arithmetic
- Subtraction
  - Any subtraction (A-B) can be converted to addition (A + -B) by taking the 2's complement of B
  - (A-B) becomes (A + ~B + 1)
  - 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)

Drop final carry-out

~~1~~110	~~1~~100
0011 (3)	1101 (-3)
+ 1110 (-2)	+ 1110 (-2)
0001 (1)	1011 (-5)

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
1 1 (carries)		
1001	(9)	(-7)
+ 0011	(3)	(3)
1100	(12)	(-4)

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)        1101 (-3)
  - 0010 (+2)      - 1110 (-2)
  ---------        ---------
```

Drop final carry-out

```
   ~~1~~111_              1_
    0011               1101
    1101  Bit flip of +2   0001  Bit flip of -2
  +    1  Add 1          +    1  Add 1
  ------               ------
    0001               1111
```

Unsigned and Signed Subtraction

- Subtraction process is the same for both unsigned and signed numbers
  - Convert A – B to A + Comp. of B
  - Drop any final carry out
- Examples:

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

```
  11_1_
   1100   A
   1101   ~B
+     1   Add 1
   ----
   1010   (10)    (-6)
          If unsigned   If signed
```

Important Note

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

OVERFLOW

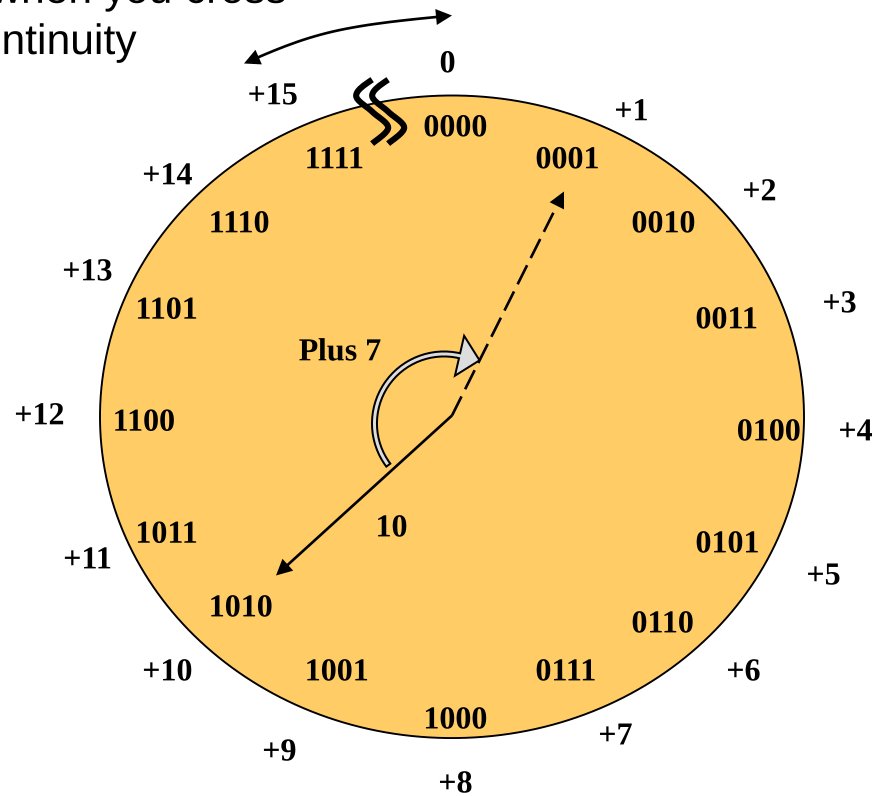
Overflow

CS:APP 2.3.1
CS:APP 2.3.2

- Overflow occurs when the result of an arithmetic operation is too large to be represented with the given number of bits
- Conditions and tests to determine overflow depend on the system being used
  - Different algorithms for detecting overflow based on unsigned or signed

Unsigned Overflow

Overflow occurs when you cross this discontinuity



10 + 7 = 17

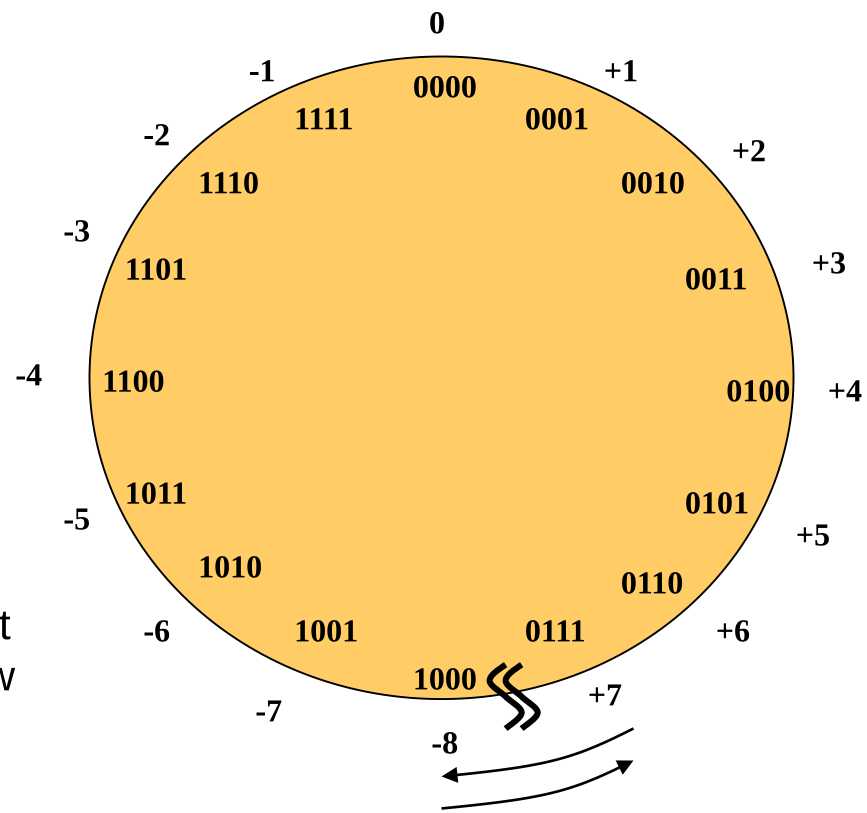
With 4-bit *unsigned* numbers we can only represent 0 – 15. Thus, we say overflow has occurred.

2’s Complement 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.



Overflow occurs when you cross this discontinuity

Overflow in Addition

- Overflow occurs when the result of the addition cannot be represented with the given number of bits.
- Tests for overflow:
  – Unsigned: if Cout = 1 [result smaller than inputs]
  – Signed: if p+p=n or n+n=p [result has inappropriate sign]

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

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

Overflow in Subtraction

- Overflow occurs when the result of the subtraction cannot be represented with the given number of bits.
- Tests for overflow:
  - Unsigned: if Cout = 0 [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

```
  0111_
   0111   A
   0111   1's comp. of B
+     1   Add 1
-------
   1111  (15)      (-1)
```

If unsigned	If signed
Overflow	Overflow
Cout = 0	p + p = n

MULTIPLICATION AND DIVISION

Binary Multiplication

CS:APP 2.3.4

- **Multiplying two *n*-bit numbers yields at most a *2*n*-bit product**
- Multiplication operations on a modern processor can take **3-5 times** longer than addition operations

```
          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

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

```
                 0 1 0 1 r.1  (5 r.1)10
               ________
(2)10      10 | 1 0 1 1         (11)10
               -1 0
               ----
                  0 1
                 -0 0
                 ----
                    1 1
                   -1 0
                   ----
                    0 1
```

Unsigned Multiplication Review

- Same rules as decimal multiplication
- Multiply each bit of Q by M shifting as you go
- An m-bit * n-bit mult. produces an m+n bit result
- Notice 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 2n bits)

Without Sign Extension… Wrong Answer!	**With Sign Extension… Correct Answer!**

```
Without Sign Extension:
      1001   = -7
   *  0110   = +6
   --------
      0000
     1001_
    1001__
 +  0000___
 ----------
  00110110   = +54

With Sign Extension:
      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
  - If MSB=1, multiply by -1 (i.e. take 2's comp. of multiplicand)

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 by -1

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

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

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. `char v;`] without changing 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;`
  - No!!! Assignment changes ALL bits in a variable
- **Because the smallest unit of data in computers is usually a byte, manipulating individual bits requires us to use BITWISE OPERATIONS.**
  - AND = &
  - OR = |
  - XOR = ^
  - NOT = ~

Bit:	7	6	5	4	3	2	1	0
	?	?	?	?	?	?	?	?

1-byte variable

?	?	?	?	?	?	?	1

Desired v

(change LSB to 1)

1	1	1	1	?	?	?	?

Desired v

(change upper 4 bits to 1111)

?	?	?	?	?	?	0	0

Desired v

(change lower 2 bits to 00)

Using Bitwise Ops to Change Bits

- **ANDs** can be used to **clear a bit** (make it '0') or leave it unchanged
- **ORs** can be used to **set a bit** (make it '1') or leave it unchanged
- **XORs** can be used to **invert a bit** (flip it) or leave it unchanged

X	Y	AND	
0	0	0	Force '0'
0	1	0	
1	0	0	Pass
1	1	1	

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

X	Y	OR	
0	0	0	Pass
0	1	1	
1	0	1	Force '1'
1	1	1	

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

X	Y	XOR	
0	0	0	Pass
0	1	1	
1	0	1	Invert
1	1	0	

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

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

Bitwise Operations

CS:APP 2.1.7

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

```
    0xa5   →     1010 0101
AND 0xf0       & 1111 0000
    0x30   ←     1010 0000

    0xa5   →     1010 0101
 OR 0xf0       | 1111 0000
    0xfc   ←     1111 0101

    0xa5   →     1010 0101
XOR 0xf0       ^ 1111 0000
    0x55   ←     0101 0101

NOT 0xa5   →   ~ 1010 0101
    0x5a   ←     0101 1010
```

```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 either True (non-zero) or False (zero)
- Bitwise Operators (&, |, ^, ~)
  - Applies 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;
  int z2 = x & y;
  printf("z1=%d, z2=%d\n",z1,z2);

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

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

```
    0000 0001
  & 0000 0010
F = 0000 0000
```

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

```
    ~ 0000 0001
T   = 1111 1110
```

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

Important Note: Since !(non-zero) = 0; and !0 = 1
So !!35=1. And !!-109=1

Application: Swapping via XORs

- Swapping variables can be done with a 3rd 'temp' variable
- For bitwise swapping, XORs can be used

```
#include <stdio.h>
int main()
{
   int x = 0x59, y = 0xd3;
   int temp = x;
   x = y;
   y = temp;

   return 0;
}
```

Traditional swap with 'temp'

XOR swap

```
#include <stdio.h>
int main()
{
   int x = 0x59, y = 0xd3;
   x = x ^ y;
   y = x ^ y;
   x = x ^ y;

   return 0;
}
```

```
  0101 1001=x
```

```
  1101 0011=y
```

```
  0101 1001=x
^ 1101 0011=y
  1000 1010=x
```

```
  1000 1010=x
^ 1101 0011=y
  0101 1001=y
```

```
  1000 1010=x
^ 0101 1001=y
  1101 0011=x
```

Exercises

- Determine if an integer is odd (w/o % operator).

```
bool isOdd(int x)
{
  /* Isolate the lowest bit */
  return x&1;
}
```

- Determine if an integer is a multiple of 4 (w/o % operator).

```
bool isMultOf4(int x)
{
  /* Check if 2 LSBs are both 0 */
  return !(x&3);
}
```

Arithmetic and Logical Shifts

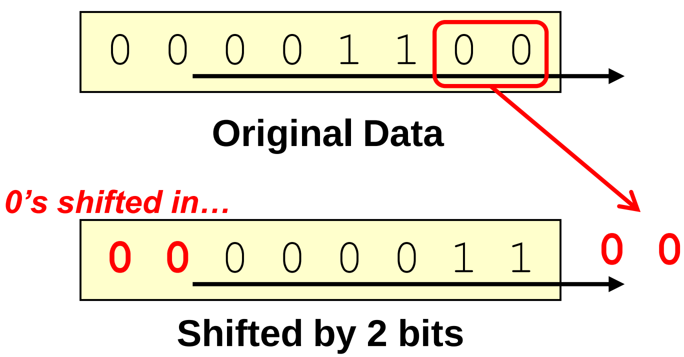
SHIFT OPERATIONS

Shift Operations

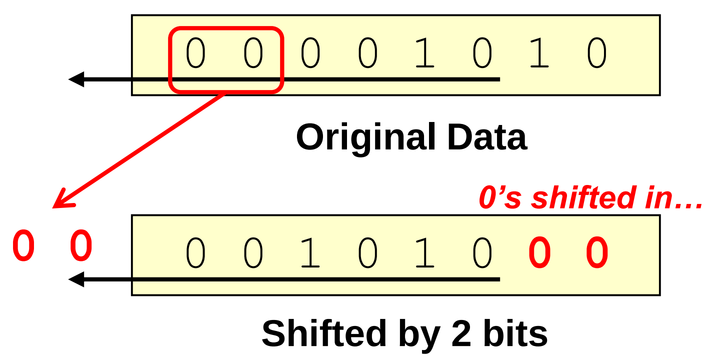
CS:APP 2.1.9

- 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 powers of 2
- 2 kinds of shifts
  - Logical shifts (used for unsigned numbers)
  - Arithmetic shifts (used for signed numbers)

Right Shift by 2 bits:



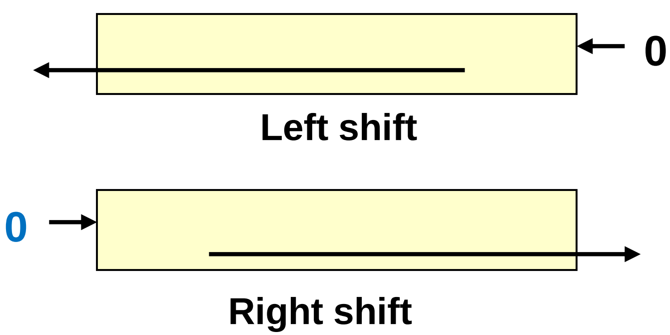
Left Shift by 2 bits:

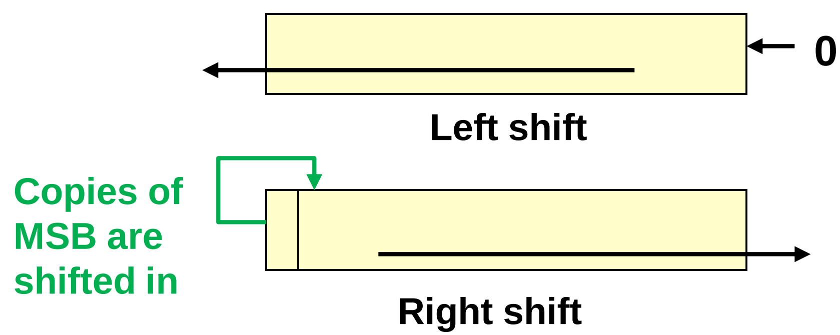


Logical Shift vs. Arithmetic Shift

- Logical Shift
  - Use for unsigned or non-numeric data
  - Will always shift in 0's whether it be a left or right shift

- Arithmetic Shift
  - Use for signed data
  - Left shift will shift in 0's
  - Right shift will sign extend (replicate the sign bit) rather than shift in 0's
    - If negative number…stays negative by shifting in 1's
    - If positive…stays positive by shifting in 0'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

0x0000000C

0	...	0	1	1	0	0

= +12

Logical Right Shift by 2 bits:

0's shifted in…

0	**0**	...	0	0	1	1

= +3

0x00000003

Logical Left Shift by 3 bits:

0's shifted in…

...	0	1	1	0	0	**0**	**0**	**0**

= +96

0x00000060

Arithmetic Shift

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

0xFFFFFFFC

```
1 1 ... 1 1 0 0
```

= -4

Arithmetic Right Shift by 2 bits:

MSB replicated and shifted in…

```
1 1 1 ... 1 1 1
```

= -1

0xFFFFFFFF

Notice if we shifted in 0's (like a logical right shift) our result would be a positive number and the division wouldn't work

Arithmetic Left Shift by 2 bits:

0's shifted in…

```
1 ... 1 0 0 0 0
```

= -16

0xFFFFFFF0

Notice there is no difference between an arithmetic and logical left shift. We always shift in 0's.

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 (i.e. 17*x). Can we still use shifting?
  - Yes. Break constant into a sum using power of 2 coefficients
  - 17x = 16x + 1x
- Exercise: How many adds/shift would be needed to compute 14*x
  - 8x + 4x + 2x = 3 shifts, 2 adds OR
  - 16x - 2x = 2 shift and 1 add

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

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

Written Code

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

```
int mul17(int x)
{
  int x16 = x << 4;
  return x16 + 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)
  - No

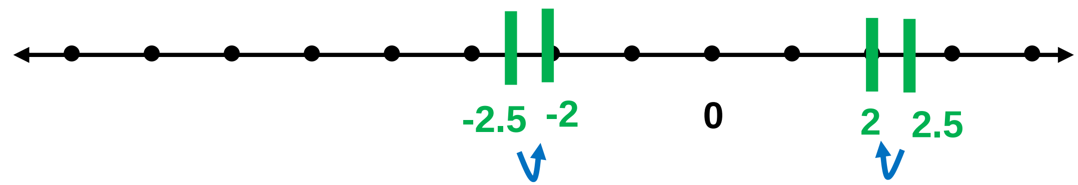
```
   -5  = 1 0 1 1
         -8 4 2 1

-5>>1  = 1 1 0 1  1
         -8 4 2 1 0.5
```

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

Dividing Negative Numbers

Traditional integer rounding

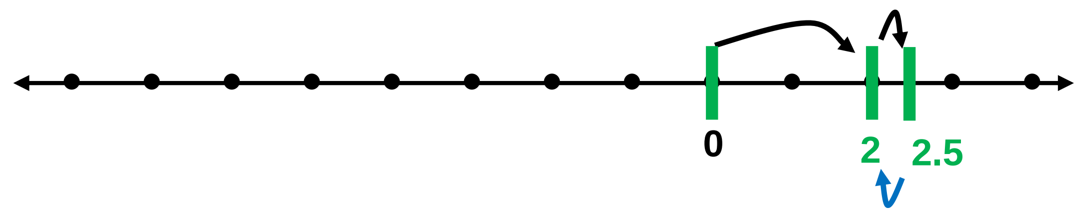


Traditional integer division rounds toward 0 (i.e. drops fractional portion)

+5>>1

0	0	1	0	1
-8	4	2	1	0.5

5 >> 1

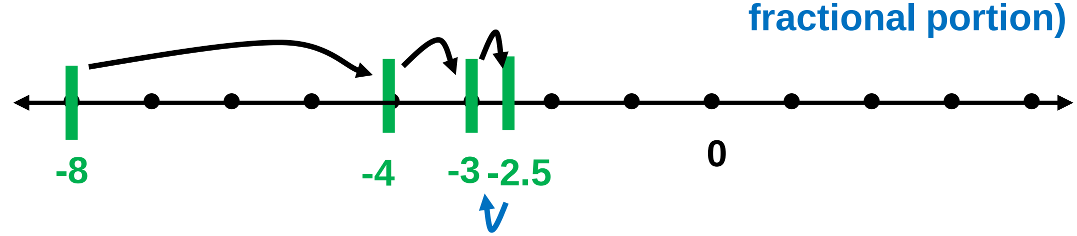


Rounding (by dropping fractional portion)

-5>>1

1	1	0	1	1
-8	4	2	1	0.5

-5 >> 1



Rounding (by dropping fractional portion)

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

Biasing

- Summary: Dividing $x / 2^k$ by performing $(x >> k)$...
  - Works when $x \geq 0$ OR when $x < 0$ & x is a multiple of 2^k
  - Doesn't work when $x < 0$ and x is NOT a multiple of 2^k
- Idea to solve the problem:
  - Add some value (aka a **bias** value) to x before shifting that will correct for the rounding issue
  - Add 2^k-1 (i.e. k ones)

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

More Examples

- $-8 / 4 = (-8 >> 2)$
  - Bias by $2^2-1 = 3$
  - $(-8 + 3) >> 2$
- $-7 / 4 = (-7 >> 2)$
  - Bias by $2^2-1 = 3$
  - $(-7 + 3) >> 2$
- $-20 / 16 = (-20 >> 4)$
  - Bias by $2^4-1 = 15$
  - $(-20 + 15) >> 4$

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

  -8    1 0 0 0
  +3  +     1 1
  --  -----------
  -5    1 0 1 1
-5>>2 = 1 1 0 0    -2

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

  -7    1 0 0 1
  +3  +     1 1
  --  -----------
  -4    1 1 0 0
-4>>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 = 31, N = 4)