EE 357 Unit 1

Fixed Point Systems and Arithmetic

© Mark Redekopp, All rights reserved

Learning Objectives

- Understand the size and systems used by the underlying HW when a variable is declared in a SW program
- Understand and be able to find the decimal value of numbers represented in various systems in either binary or hex
- Perform the various arithmetic and logic operations that the HW needs to perform
- Be able to determine when overflow has occurred in an arithmetic operation

© Mark Redekopp, All rights reserved

Connecting C & EE 101

- This slide package is meant to review the basic data representation and operations that we learned in EE 101 but now in the context of a programming language like C
- We will show how the code you write in C directs the compiler to control the HW in specific ways

© Mark Redekopp, All rights reserved

Unsigned

2's Complement

Sign and Zero Extension

Hexadecimal Representation

SIGNED AND UNSIGNED SYSTEMS

© Mark Redekopp, All rights reserved

Binary Representation Systems

- Integer Systems
  - Unsigned
    - Unsigned (Normal) binary
  - Signed
    - Signed Magnitude
    - 2's complement
    - *Excess-N**
    - *1's complement**
- Floating Point
  - For very large and small (fractional) numbers

- Codes
  - Text
    - ASCII / Unicode
  - Decimal Codes
    - BCD (Binary Coded Decimal) / (8421 Code)

* = Not fully covered in this class

© Mark Redekopp, All rights reserved

Data Representation

- In C/C++ variables can be of different types and sizes
  - Integer Types (signed by default…unsigned with leading keyword)

C Type	Bytes	Bits	Coldfire Name	MIPS Name
[unsigned] char	1	8	byte	byte
[unsigned] short [int]	2	16	word	half-word
[unsigned] long [int]	4	32	longword	word
[unsigned] long long [int]	8	64	-	double word

  - Floating Point Types

C Type	Bytes	Bits	Coldfire/MIPS Name
float	4	32	single
double	8	64	double

© Mark Redekopp, All rights reserved

C examples

int x = -2;	Allocates a 4-byte (32-bit) chunk of memory Any operation involving x will use signed operations, if necessary
char c = 0xfa;	Allocates a 1-byte (8-bit) chunk of memory
unsigned char d = 10;	Allocates a 1-byte chunk of memory Any operation involving d will use unsigned operations, if necessary
float f = 3.1;	Allocates a 4-byte (32-bit) chunk of memory Any operation involving f will use floating point HW
double g = -1.5;	Allocates an 8-byte (64-bit) chunk of memory
unsigned long long y;	Allocates an 8-byte (64-bit) chunk of memory
short z = -1;	Allocates a 2-byte (16-bit) chunk of memory

© Mark Redekopp, All rights reserved

Unsigned and Signed Variables

- Unsigned variables use unsigned binary (normal power-of-2 place values) to represent numbers

1	0	0	1	0	0	1	1	= +147
128	64	32	16	8	4	2	1	

- Signed variables use the 2's complement system (Neg. MSB weight) to represent numbers

1	0	0	1	0	0	1	1	= -109
-128	64	32	16	8	4	2	1	

© Mark Redekopp, All rights reserved

2’s Complement System

- MSB has negative weight
- MSB determines sign of the number
  - 1 = negative
  - 0 = positive
- To take the negative of a number (e.g. -7 => +7 or +2 => -2), requires taking the complement
  - 2’s complement of a # is found by flipping bits and adding 1

1001	x = -7
0110	Bit flip (1’s comp.)
+ 1	Add 1
0111	-x = -(-7) = +7

© Mark Redekopp, All rights reserved

Zero and Sign Extension

- Extension is the process of increasing the number of bits used to represent a number without changing its value

Unsigned = Zero Extension (Always add leading 0’s):

111011 = 00111011

Increase a 6-bit number to 8-bit number by zero extending

2’s complement = Sign Extension (Replicate sign bit):

pos. 011010 = 00011010

neg. 110011 = 11110011

Sign bit is just repeated as many times as necessary

© Mark Redekopp, All rights reserved

Zero and Sign Truncation

- Truncation is the process of decreasing the number of bits used to represent a number without changing its value

Unsigned = Zero Truncation (Remove leading 0’s):

~~00~~111011 = 111011

Decrease an 8-bit number to 6-bit number by truncating 0’s. Can’t remove a ‘1’ because value is changed

2’s complement = Sign Truncation (Remove copies of sign bit):

pos. ~~00~~011010 = 011010

neg. ~~11~~110011 = 10011

Any copies of the MSB can be removed without changing the numbers value. Be careful not to change the sign by cutting off ALL the sign bits.

© Mark Redekopp, All rights reserved

Representation Range

- Given an n-bit system we can represent 2^n unique numbers
  - In unsigned systems we use all combinations to represent positive numbers [0 to 2^n-1]
  - In 2's complement we use half for positive and half for negative [-2^{n-1} to $+2^{n-1}-1$]

n	2^n
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512

© Mark Redekopp, All rights reserved

Approximating Large Powers of 2

- Often need to find decimal approximation of a large powers of 2 like 2^{16}, 2^{32}, etc.
- Use following approximations:
  - $2^{10} \approx 10^3$ (1 thousand)
  - $2^{20} \approx 10^6$ (1 million)
  - $2^{30} \approx 10^9$ (1 billion)
- For other powers of 2, decompose into product of 2^{10} or 2^{20} or 2^{30} and a power of 2 that is less than 2^{10}
- See examples

$2^{16} = 2^6 * 2^{10}$
$\approx 64 * 10^3 = 64{,}000$

$2^{24} = 2^4 * 2^{20}$
$\approx 16 * 10^6 = 16{,}000{,}000$

$2^{28} = 2^8 * 2^{20}$
$\approx 256 * 10^6 = 256{,}000{,}000$

$2^{32} = 2^2 * 2^{30}$
$\approx 4 * 10^9 = 4{,}000{,}000{,}000$

© Mark Redekopp, All rights reserved

Hexadecimal Representation

- Since values in modern computers are many bits, we use hexadecimal as a shorthand notation (4 bits = 1 hex digit)
  - 11010010 = D2 hex
  - 0111011011001011 = 76CB hex
- To interpret the value of a hex number, you must know what underlying binary system is assumed (unsigned, 2's comp. etc.)

© Mark Redekopp, All rights reserved

Translating Hexadecimal

- Hex place values (16^2, 16^1, 16^0) can ONLY be used if the number is positive.
- If hex represents unsigned binary simply apply hex place values
  - B2 hex = $11*16^1 + 2*16^0 = 178_{10}$
- If hex represents signed value (2's comp.)
  - First determine the sign to be pos. or neg.
    - Convert the MS-hex digit to binary to determine the MSB (e.g. for B2 hex, B=1011 so since the MSB=1, B2 is neg.)
    - In general, hex values starting 0-7 = pos. / 8-F = neg.
  - If pos., apply hex place values (as if it were unsigned)
  - If neg., take the **16's complement** and apply hex place values to find the neg. number's magnitude

© Mark Redekopp, All rights reserved

Taking the 16's Complement

- Taking the 2's complement of a binary number yields its negative and is accomplished by finding the 1's complement (bit flip) and adding 1
- Taking the 16's complement of a hex number yields its negative and is accomplished by finding the 15's complement and adding 1
  - 15's complement is found by subtracting each digit of the hex number from F_{16}

Original value B2:	FF	
	- B2	Subtract each digit from F
	4D	15's comp. of B2
	+ 1	Add 1
16's comp. of B2:	4E	16's comp. of B2

© Mark Redekopp, All rights reserved

Translating Hexadecimal

- Given 6C hex
  - If it is unsigned, apply hex place values
    - 6C hex = $6*16^1 + 12*16^0 = 108_{10}$
  - If it is signed…
    - Determine the sign by looking at MSD
      - 0-7 hex has a 0 in the MSB [i.e. positive]
      - 8-F hex has a 1 in the MSB [i.e. negative]
      - Thus, 6C (start with 6 which has a 0 in the MSB is positive)
    - Since it is positive, apply hex place values
      - 6C hex = $6*16^1 + 12*16^0 = 108_{10}$

© Mark Redekopp, All rights reserved

Translating Hexadecimal

- Given FE hex
  - If it is unsigned, apply hex place values
    - FE hex = $15*16^1 + 14*16^0 = 254_{10}$
  - If it is signed…
    - Determine sign => Negative
    - Since it is negative, take 16’s complement and then apply place values
      - 16’s complement of FE = 01 + 1 = 02 and apply place values = 2
      - Add in sign => -2 = FE hex

© Mark Redekopp, All rights reserved

Finding the Value of Hex Numbers

- B2 hex representing a signed (2's comp.) value
  - Step 1: Determine the sign: Neg.
  - Step 2: Take the 16's comp. to find magnitude

    FF - B2 + 1 = 4E hex
  - Step 3: Apply hex place values ($4E_{16} = +78_{10}$)
  - Step 4: Final value: B2 hex = -78_{10}
- 7C hex representing a signed (2's comp.) value
  - Step 1: Determine the sign: Pos.
  - Step 2: Apply hex place values ($7C_{16} = +124_{10}$)
- 82 hex representing an unsigned value
  - Step 1: Apply hex place values ($82_{16} = +130_{10}$)

© Mark Redekopp, All rights reserved

C examples

int x = -2;	Allocates a 4-byte (32-bit) chunk of memory Any operation involving x will use signed operations, if necessary
char c = 0xfa;	Allocates a 1-byte (8-bit) chunk of memory
unsigned char d = 10;	Allocates a 1-byte chunk of memory Any operation involving d will use unsigned operations, if necessary
float f = 3.1;	Allocates a 4-byte (32-bit) chunk of memory Any operation involving f will use floating point HW
double g = -1.5;	Allocates an 8-byte (64-bit) chunk of memory
unsigned long long y;	Allocates an 8-byte (64-bit) chunk of memory
short z = -1;	Allocates a 2-byte (16-bit) chunk of memory
x = z;	Implicit ‘cast’ of 2-byte z to 4-byte x by sign extending
y = d;	Implicit ‘cast’ of 1-byte d to 8-byte y by zero extending
c = (char) x;	Explicit ‘cast’ of 4-byte x to 1-byte c by sign truncation (keep LSB’s)
x = (int) f;	Explicit ‘cast’ of 4-byte floating point format to 4-byte integer format
g = g + (double) z;	Explicit ‘cast’ of 2-byte integer format z to 8-byte FP format

© Mark Redekopp, All rights reserved

Arithmetic Operations and Overflow

Bitwise Logic Operations

Bit Shift Operations

OPERATIONS

© Mark Redekopp, All rights reserved

Arithmetic and Logic Instructions

C operator	CF Assembly	Notes
+	ADD.s src1,src2/dst	
-	SUB.s src1,src2/dst	Order: src2 – src1
&	AND.s src1,src2/dst	
|	OR.s src1,src2/dst	
^	EOR.s src1,src2/dst	
~	NOT.s src/dst	
-	NEG.s src/dst	Performs 2's complementation
* (signed)	MULS src1,src2/dst	Implied size .W
* (unsigned)	MULU src1, src2/dst	Implied size .W
/ (signed)	DIVS src1,src2/dst	Implied size .W
/ (unsigned)	DIVU src1, src2/dst	Implied size .W
<< (signed)	ASL.s cnt, src/dst	Arithmetic (signed) Left shift
<< (unsigned)	LSL.s cnt, src/dst	Logical (uns.) Left Shift
>> (signed)	ASR.s cnt, src/dst	Arithmetic (signed) Right shift
>> (unsigned)	LSR.s cnt, src/dst	Logical (uns.) Right Shift
==, <, >, <=, >=, != (src2 ? src1)	CMP.s src1, src2	Order: src2 – src1

© Mark Redekopp, All rights reserved

Unsigned and Signed Addition

- Addition process is the same for both unsigned and signed numbers
  - – Add columns right to left
  - – Drop any final carry out
- Examples:

	If unsigned	**If signed**
1 1		
1001	(9)	(-7)
+ 0011	(3)	(3)
1100	(12)	(-4)

© Mark Redekopp, All rights reserved

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
  - Signed: if p + p = n   or   n + n = p

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

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

© Mark Redekopp, All rights reserved

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	1's comp. of B		
+ 1	Add 1		
1010	(10)	(-6)	
	If unsigned	If signed	

© Mark Redekopp, All rights reserved

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
  - Signed: if addition is p + p = n or n + n = p

	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**

© Mark Redekopp, All rights reserved

Hex Addition and Overflow

- Same rules as in binary
  - Add left to right
  - Drop any carry (carry occurs when sum > F_{16})
- Same addition overflow rules
  - Unsigned: Check if final Cout = 1
  - Signed: Check signs of inputs and result

```
  1  1
    7AC5
+   C18A
--------
    3C4F
```

If unsigned

Overflow
Cout = 1

If signed

No Overflow
p + n

```
  0 1  1
    6C12
+   549F
--------
    C0B1
```

If unsigned

No Overflow
Cout = 0

If signed

Overflow
p + p = n

© Mark Redekopp, All rights reserved

Hex Subtraction and Overflow

- Same rules as in binary
  - Convert A – B to A + Comp. of B
  - Drop any final carry out
- Same subtraction overflow rules
  - Unsigned: Check if final Cout = 0
  - Signed: Check signs of addition inputs and result

```
                 1
  B1ED          B1ED
- 76FE    ⇨     8901
------        +    1
              ------
                3AEF
```

If unsigned
No Overflow
Cout = 1

If signed
Overflow
n + n = p

```
                 0
  0001          0001
- 0002    ⇨     FFFD
------        +    1
              ------
                FFFF
```

If unsigned
Overflow
Cout = 0

If signed
No Overflow
p + n

© Mark Redekopp, All rights reserved

Logical Operations

- Logic operations on numbers means performing the operation on each pair of bits

Initial Conditions: x = 0x000000F0, y = 0x0000003C

① x = x & y;	→	0xF0	→	1111 0000
		AND 0x3C		AND 0011 1100
x = 0x00000030	←	0x30	←	0011 0000
② x = x | y;	→	0xF0	→	1111 0000
		OR 0x3C		OR 0011 1100
x = 0x000000FC	←	0xFC	←	1111 1100
③ x = x ^ y;	→	0xF0	→	1111 0000
		EOR 0x3C		EOR 0011 1100
x = 0x000000CC	←	0xCC	←	1100 1100

© Mark Redekopp, All rights reserved

Logical Operations

- Logic operations on numbers means performing the operation on each pair of bits

Initial Conditions: x = 0xFFFFFFF0, y = 0x0000003C

① x = ~x; → NOT 0xF0 → NOT 1111 0000
x = 0x0000000F ← 0x0F ← 0000 1111

② y = -y; → 2's Comp. 0x3C → 0xffffffc3
\+ 1
y = 0xFFFFFFC4 ← 0xFFFFFFC4 ← 0xffffffc4

© Mark Redekopp, All rights reserved

Logical Operations

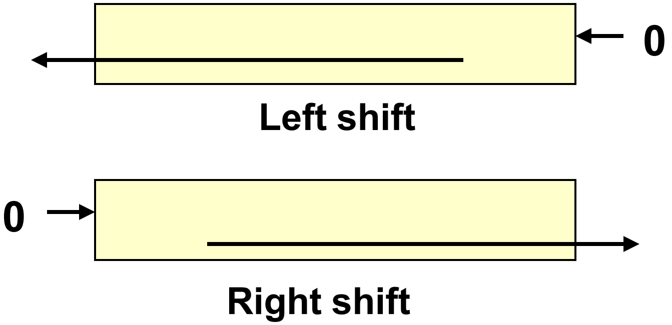
- Logic operations are often used for “bit” fiddling
  - Change the value of 1-bit in a number w/o affecting other bits
  - C operators: & = AND, | = OR, ^ = XOR, ~ = NOT
- Examples (Assume an 8-bit variable, v)
  - Set the LSB to ‘0’ w/o affecting other bits
    - v = v & 0xfe;
  - Set the MSB to ‘1’ w/o affecting other bits
    - v = v | 0x80;
  - Flip the LS 4-bits w/o affecting other bits
    - v = v ^ 0x0f;
  - Check if the MSB = ‘1’ regardless of other bit values
    - if(v & 0x80) { code }

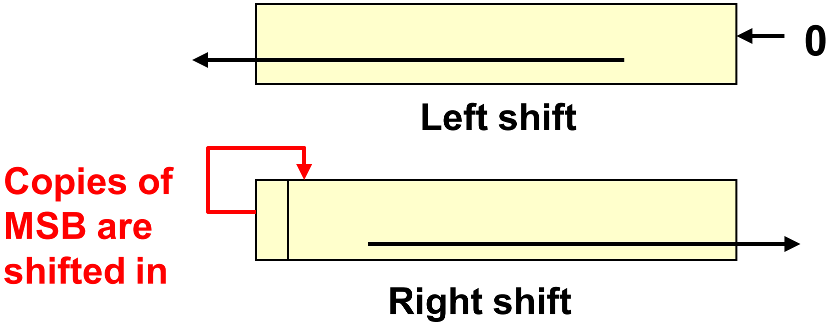
© Mark Redekopp, All rights reserved

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

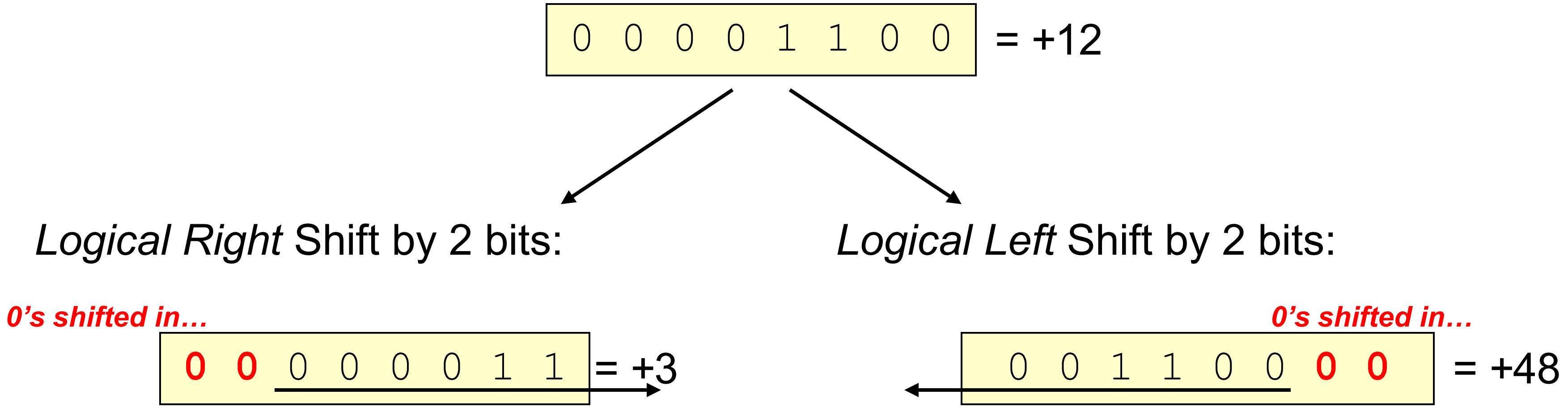




© Mark Redekopp, All rights reserved

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



0 0 0 0 1 1 0 0 = +12

Logical Right Shift by 2 bits:

0's shifted in…

0 0 0 0 0 0 1 1 = +3

Logical Left Shift by 2 bits:

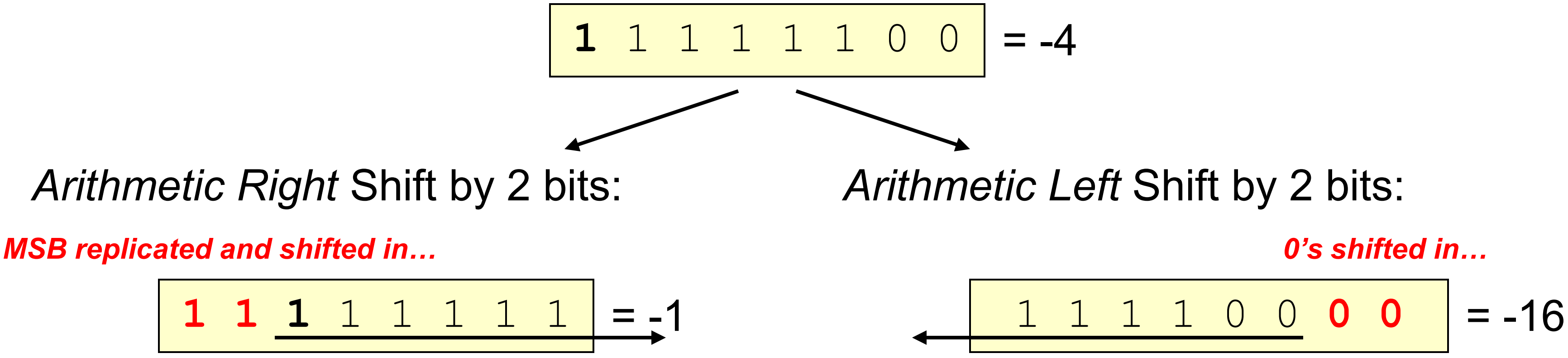
0's shifted in…

0 0 1 1 0 0 **0 0** = +48

© Mark Redekopp, All rights reserved

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 (same as logical)
  - Left shift by n-bits = Multiplying by 2^n



1 1 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 1 = -1

Arithmetic Left Shift by 2 bits:

0's shifted in…

1 1 1 1 0 0 **0 0** = -16

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

© Mark Redekopp, All rights reserved

C examples

int x = 4;	x = 0x00000004
char c = 0x80;	c = 0x80
unsigned char d = 10;	d = 0x0a
short z = -2;	z = 0xfffe
c = c >> x;	Signed (arithmetic) right shift of 0x80 by 4 bit places = 0xf8
d = d << 2;	Unsigned (logical) left shift of 0x0a by 2 bit places = 0x28
d = (unsigned char) c >> x;	Unsigned (logical) right shift of 0x80 by 4 bit places = 0x08
c = c | 0x0f ;	Bitwise OR of 0x80 with 0x0f = 0x8f
z = z & 0x03;	Bitwise AND of 0xfffe and 0x0003 = 0x0002

© Mark Redekopp, All rights reserved