LECTURE 10: MORE ON NUMBER SYSTEMS

NUMBER SYSTEM CONVERSIONS

Remember that, in each number system, 10 is the number of different digits that system has:

	binary	octal	decimal	hex
decimal value of 10	$2^1 = 2$	$8^1 = 8$	$10^1 = 10$	$16^1 = 16$

This is because each place value corresponds to the base of the numbering system to some power.

The LSB place is always 1 because $n^0 = 1$

The next place value is n^1 and so on.

CONVERTING DECIMAL TO BINARY

Divide the decimal number by 2 and write down the remainder. Repeat.

The first remainder is the least significant bit.

Example: Convert 117_{10} to an 8-bit binary number.

117	1	LSB	(117 / 2 = 58 r 1)
58	0		(58 / 2 = 29 r 0)
29	1		(29 / 2 = 14 r 1)
14	0		(14 / 2 = 7 r 1)
7	1		(7 / 2 = 3 r 1)
3	1		(3 / 2 = 1 r 1)
1	1		(1 / 2 = 0 r 1)
0	0	MSB	(add leading 0)

Answer: 01110101

CONVERTING BINARY TO DECIMAL

Treat each bit position n as adding 2^n to the value.

If a bit position n contains a 1, add 2^n to the value.	If a bit position contains a 0, do not add anything to the value.

Example: Convert 00101101_2 to decimal.

bit			running total	place value
bit 0	(LSB)	1	0 + 1 = 1	$2^0 = 1$
bit 1		0		
bit 2		1	+ 4 = 5	$2^2 = 4$
bit 3		1	+ 8 = 13	$2^3 = 8$
bit 4		0		
bit 5		1	+ 32 = 45	$2^5 = 32$
bit 6		0		
bit 7	(MSB)	0		
total:			45	

CONVERTING DECIMAL TO HEX

	Method 1
1.	Convert the decimal number to binary: 117_{10} to 01110101_2
2.	Group the binary digits into sets of 4: 0111 0101
3.	Convert the 4-bit binary numbers into hex digits: 75_{16}

	Method 2
	Divide the number successively by 16 and write the remainder as a hex digit.
1.	117 / 16 = 7 r 5 (LSB)
2.	7 / 16 = 0 r 7 (MSB)
	0x75

CONVERTING HEX TO DECIMAL

Treat each digit as adding:
(that digit's value * 16^n) to the value

Example: Convert 0x100b20 to decimal.

bit			running total		place value
digit 0	(LSB)	0	0 + 0	= 0	$2^0 = 1$
digit 1		2	+ 32	= 32	$2 * 16^1 = 32$
digit 2		b	+ 2816	= 2848	$11 * 16^2 = 2816$
digit 3		0			$2^3 = 8$
digit 4		0			
digit 5		1	+ 1048576	= 1051424	$1 * 16^5 = 1048576$
digit 6		0			
digit 7	(MSB)	0			
total:				1051424	

LECTURE 10: CHAR/INT CONSTANTS, SIGNED/UNSIGNED (K&R § 2.7)

DESIGNATING BASE OF INTEGER CONSTANTS

If a constant begins with	Then it is
0x	hex with a-f as hex digits.
0X	hex with A-F as hex digits.
0	octal
[nothing]	decimal

DESIGNATING BASE OF CHARACTER CONSTANTS

If a constant has the form	Then it is
'\x[number]'	hex with a-f as hex digits.
'\X[number]'	hex with A-F as hex digits.
'\0[number]'	octal
Otherwise, it is the ASCII code for a character	
'a'	

ASIDE: TWO'S COMPLEMENT NUMBER REPRESENTATION

two's complement	Can refer to two things: 1. A system of storing integers. 2. An operation on binary numbers.
This system says, for a bit length of n:	
for zero:	Represent with n 0s (i.e., 4-bit 0 is 0000).
for positive integers:	Count up, with a maximum of $2^{(n-1)}-1$. i.e., for 4-bits: 1 is 0001 2 is 0010 [...] 7 is 0111
for negative integers:	Same as positive, but reverse role of 0s and 1s (and start from 1111 = -1). i.e., for 4-bits: -1 is 1111 -2 is 1110 [...] -8 is 1000
This explains the value ranges we saw for signed integers: the reason there is one more negative than positive is that 0000 is used for 0.	
Using this system, the MSB gets the role of sign bit:	
1	negative number
0	non-negative number (zero and positive)
Notice that this system allows you to get 0 when adding positive and negative:	e.g., 5 + -5 = 0 0101 (5) + 1011 (-5) 10000 drop the carry bit 0000 zero
This system also avoids the problem of having two representations for 0 (a positive 0 and a negative 0).	

Note:	C does not specify whether variables of type char are signed or unsigned. Often chars are signed by default, as in our machines at UMB and in examples that follow.

CHAR AND INTEGER CONSTANTS ARE SIGNED

sign bit not set (i.e., sign bit is 0)		
'\x55'	character constant. equals an 8-bit quantity:	01010101
	when casted to an int, it equals:	0000...01010101
0x55	integer constant. equals a 32-bit quantity:	0000...01010101

sign bit set (i.e., sign bit is 1)		
'\xaa'	character constant. equals an 8-bit quantity:	10101010
	when casted to an int, it equals:	1111...01010101
0xaa	integer constant. equals a 32-bit quantity:	0000...10101010

Note:	None of the ASCII codes (x00 - x7f) have the sign bit set.

SIGN EXTENSION

When casting to a larger data type, how the excess bits are set is dependent on your machine. Sign extension will duplicate the MSB in this case, potentially converting positive values to negative values.

signed default behavior	
int i;	signed by default
char c;	signed by default
i = 0xaa;	== 0000 00aa
i = '\xaa';	== ffff ffaa (sign extends!)
<---------sign bit extension--------	
char	10101010
int	11111111 11111111 11111111 10101010
c = '\xaa';	== aa
i = c;	== ffff ffaa (sign extends!)

unsigned default behavior	
unsigned int i;	must specify unsigned if wanted
unsigned char c;	must specify unsigned if wanted
i = 0xaa;	== 0000 00aa
i = '\xaa';	== ffff ffaa (sign extends!) (char constant is signed by default)
c = '\xaa';	== aa
i = c;	== 0000 00aa (sign extends!)
sign does not extend because c is unsigned	
char	10101010
int	00000000 00000000 00000000 10101010

LECTURE 10: BITWISE OPERATIONS (K&R § 2.9)

BITWISE OPERATIONS

&	bitwise AND	<<	left shift
|	bitwise inclusive OR	>>	right shift
^	bitwise exclusive OR	~	one's complement
		-	two's complement

ONE'S COMPLEMENT (~)

Takes the logical NOT of each bit in the operand.

This means it flips the value of each bit in the operand's value.

	zeros become ones	ones become zeros
Example:	~'\xaa' == '\x55' ~10101010 == 01010101	10101010 01010101

TWO'S COMPLEMENT (-)

The two's complement operator is the negative sign.

It's a unary operator that performs the following steps:

1. take the one's complement
2. add 1

The two's complement generates the negative of the original value.

Example:	-'\x55' == '\xab' -01010101 == 10101011	1.	~01010101 == 10101010
		2.	10101010 + 1 == 10101011

Two Special Case Values	
char 0 (or zero of any length)	-00000000 == 11111111 + 1 00000000
char -2^7 (or -2^{n-1} for any length n)	-10000000 == 01111111 + 1 10000000

AND (&)

Takes the logical AND of the bits in each position of two numbers in binary form.

This means that both bits have to be on for the result to be on.

Example:	01001000 & 10111000 00001000	Check each bit position. LSB to MSB (R to L):	0	0 ∧ 0 == 0
			1	0 ∧ 0 == 0
			2	0 ∧ 0 == 0
			3	1 ∧ 1 == 1
			4	0 ∧ 1 == 0
			5	0 ∧ 1 == 0
			6	1 ∧ 0 == 0
			7	0 ∧ 1 == 0

Note:	A number anded with its one's complement == 0 10101010 & 01010101 == 00000000

inclusive OR (|)

Takes the logical OR of the bits in each position of two numbers in binary form.

This means that if either bit is on for the result is on.

Example:	01001000 | 10111000 11111000	Check each bit position. LSB to MSB (R to L):	0	0 ∨ 0 == 0
			1	0 ∨ 0 == 0
			2	0 ∨ 0 == 0
			3	1 ∨ 1 == 1
			4	0 ∨ 1 == 1
			5	0 ∨ 1 == 1
			6	1 ∨ 0 == 1
			7	0 ∨ 1 == 1

exclusive OR (^)

Takes the logical OR of the bits in each position of two numbers in binary form, but not if both are on.

This means that the result is on if exactly one bit is on.

Example:	01001000 ^ 10111000 11110000	Check each bit position. LSB to MSB (R to L):	0	0 v 0 == 0
			1	0 v 0 == 0
			2	0 v 0 == 0
			3	1 V 1 == 0
			4	0 V 1 == 1
			5	0 V 1 == 1
			6	1 V 0 == 1
			7	0 V 1 == 1

LEFT SHIFT (<<)

Form:	number << units-to-shift	
	number	a char or int
	units-to-shift	an integer (how far to shift)

Left shift moves bits to the left.

Bits that fall off the left side (MSB) disappear.

0s are filled in on the right side (LSB).

Example:	01010101 << 3							
	7	6	5	4	3	2	1	0
	0	1	0	1	0	1	0	1
	we add 3 to each bit position							
	was 4 new 7	was 3 new 6	was 2 new 5	was 1 new 4	was 0 new 3	new 2	new 1	new 0
	1	0	1	0	1	0	0	0

Left-shifting has the effect of multiplying by a power of 2.	In the above example, we shift 3, which multiplies by 2^3 == 8. Note that we have overflow in this example.

RIGHT SHIFT (>>)

Like left shift, but moves bits to the right.

Bits that fall off the right side (LSB) disappear.

How bits are filled in on the left side (MSB) depends on whether the variable you are shifting is signed or unsigned:

unsigned	0s are filled in on the left side (LSB).	
signed	implementation-defined (see K&R page 49)	
	arithmetic shift (common)	fills in with copies of sign bit (sign extension)
	logical shift	fills in with 0s

Example:	(unsigned) 01010101 >> 1							
	7	6	5	4	3	2	1	0
	0	1	0	1	0	1	0	1
	we subtract 1 from each bit position							
	new 7	was 7 new 6	was 6 new 5	was 5 new 4	was 4 new 3	was 3 new 2	was 2 new 1	was 1 new 0
	0	0	1	0	1	0	1	0

Right-shifting unsigned variables has the effect of dividing by a power of 2.

BINARY LOGIC TABLES

shaded boxes = operands
unshaded boxes = results

NOT	0	1	
	1	0	

AND		0	1
	0	0	0
	1	0	1

OR		0	1
	0	0	1
	1	1	1

XOR		0	1
	0	0	1
	1	1	0

ADD		0	1
	0	0	1
	1	1	0 carry 1