Unit 1

Integer Representation

Skills & Outcomes

- You should know and be able to apply the following skills with confidence
  - Convert an unsigned binary number to and from decimal
  - Understand the finite number of combinations that can be made with n bits
  - Convert a signed (2's complement system) binary number to and from decimal
  - Convert bit sequences to and from hexadecimal
  - Predict the outcome & perform casting operations

DIGITAL REPRESENTATION

Information Representation

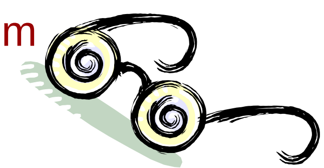
- All information in a computer system is represented as bits
  - Bit = (Binary digit) = 0 or 1
- A single bit is can only represent 2 values so to represent a wider variety of options we use a sequence of bits (e.g. 11001010)
  - Commonly sequences are 8-bits (aka a "byte"), 16-, 32- or 64-bits
- Kinds of information
  - Numbers, text, code/instructions, sound, images/videos

Interpreting Binary Strings

- Given a sequence of 1's and 0's, you need to know the *representation system* being used, before you can understand the value of those 1's and 0's.
- Information (value) = Bits + Context (System)

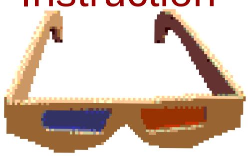
01000001 = ?

Unsigned Binary system



65 decimal

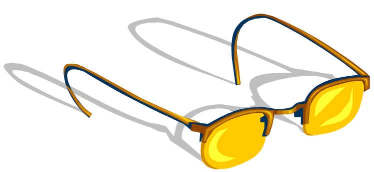
x86 Assembly Instruction



`inc %ecx`

(Add 1 to the ecx register)

ASCII system



'A'$_{\text{ASCII}}$

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 covered in this class

Data Representation

- In C/C++ variables can be of different types and sizes
  - Integer Types on 32-bit (64-bit) architectures

C Type (Signed)	C Type (Unsigned)	Bytes	Bits	x86 Name
char	unsigned char	1	8	byte
short	unsigned short	2	16	word
int / int32_t †	unsigned / uint32_t †	4	32	double word
long	unsigned long	4 (8)	32 (64)	double (quad) word
long long / int64_t †	unsigned long long / uint64_t †	8	64	quad word
char*	-	4 (8)	32 (64)	double (quad) word
int*	-	4 (8)	32 (64)	double (quad) word

† = defined in stdint.h

  - Floating Point Types

C Type	Bytes	Bits	x86 Name
float	4	32	single
double	8	64	double

OVERVIEW

Using power-of-2 place values

UNSIGNED BINARY TO DECIMAL

Number Systems

- Unsigned binary follows the rules of positional number systems
- A positional number systems consist of
  1. A base (radix) r
  2. r coefficients [0 to r-1]
- Humans: Decimal (Base 10): 0,1,2,3,4,5,6,7,8,9
- Computers: Binary (Base 2): 0,1
- Human systems for working with computer systems (shorthand for human to read/write binary)
  - Octal (Base 8): 0,1,2,3,4,5,6,7
  - Hexadecimal (Base 16): 0-9,A,B,C,D,E,F (A thru F = 10 thru 15)

Anatomy of a Decimal Number

- A number consists of a string of explicit coefficients (digits).
- Each coefficient has an implicit place value which is a power of the base.
- The value of a decimal number (a string of decimal coefficients) is the sum of each coefficient times it place value

radix (base)

$$(934)_{10} = 9*10^2 + 3*10^1 + 4*10^0 = 934$$

Explicit coefficients

Implicit place values

$$(3.52)_{10} = 3*10^0 + 5*10^{-1} + 2*10^{-2} = 3.52$$

Anatomy of an Unsigned Binary Number

- Same as decimal but now the coefficients are 1 and 0 and the place values are the powers of 2

Most Significant Digit (MSB)

Least Significant Bit (LSB)

$$(1011)_2 = 1*2^3 + 0*2^2 + 1*2^1 + 1*2^0$$

radix (base)

coefficients

place values = powers of 2

Binary Examples

$$(\underline{1}\underline{0}\underline{0}\underline{1}.\underline{1})_2 = 8 + 1 + 0.5 = 9.5_{10}$$

8 4 2 1 .5

$$(\underline{1}0\underline{1}\underline{1}000\underline{1})_2 = 128 + 32 + 16 + 1 = 177_{10}$$

128 32 16 1

General Conversion From Unsigned Base r to Decimal

- An unsigned number in base r has place values/weights that are the powers of the base
- Denote the coefficients as: a_i

$$(a_3a_2a_1a_0.a_{-1}a_{-2})_r = a_3*r^3 + a_2*r^2 + a_1*r^1 + a_0*r^0 + a_{-1}*r^{-1} + a_{-2}*r^{-2}$$

Left-most digit = Most Significant Digit (MSD)

Right-most digit = Least Significant Digit (LSD)

$$N_r \Rightarrow \Sigma_i(a_i*r^i) \Rightarrow D_{10}$$

Number in base r

Decimal Equivalent

Examples

$$(746)_8 = 7*8^2 + 4*8^1 + 6*8^0$$
$$= 448 + 32 + 16 = 486_{10}$$

$$(1A5)_{16} = 1*16^2 + 10*16^1 + 5*16^0$$
$$= 256 + 160 + 5 = 421_{10}$$

$$(AD2)_{16} = 10*16^2 + 13*16^1 + 2*16^0$$
$$= 2560 + 208 + 2 = (2770)_{10}$$

"Making change"

UNSIGNED DECIMAL TO BINARY

Decimal to Unsigned Binary

- To convert a decimal number, *x,* to binary:
  - Only coefficients of 1 or 0. So simply find place values that add up to the desired values, starting with larger place values and proceeding to smaller values and place a 1 in those place values and 0 in all others

25_{10} =

0	1	1	0	0	1
32	16	8	4	2	1

For 25_{10} the place value 32 is too large to include so we include 16. Including 16 means we have to make 9 left over. Include 8 and 1.

Decimal to Unsigned Binary

73_{10}=	0	1	0	0	1	0	0	1
	128	64	32	16	8	4	2	1
87_{10}=	0	1	0	1	0	1	1	1
145_{10}=	1	0	0	1	0	0	0	1

0.625_{10}=	1	0	1	0	0
	.5	.25	.125	.0625	.03125

Decimal to Another Base

- To convert a decimal number, *x*, to base r:
  - Use the place values of base r (powers of r). Starting with largest place values, fill in coefficients that sum up to desired decimal value without going over.

75_{10} =	0	4	B	hex
	256	16	1	

USC Viterbi School of Engineering

The 2^n rule

UNIQUE COMBINATIONS

Powers of 2

$2^0 = 1$

$2^1 = 2$

$2^2 = 4$

$2^3 = 8$

$2^4 = 16$

$2^5 = 32$

$2^6 = 64$

$2^7 = 128$

$2^8 = 256$

$2^9 = 512$

$2^{10} = 1024$

___	___	___	___	___	___	___	___	___	___	___
1024	512	256	128	64	32	16	8	4	2	1

Unique Combinations

- Given *n* digits of base *r*, how many unique numbers can be formed? r^n
  - What is the range? [0 to r^n-1]

2-digit, decimal numbers (r=10, n=2) ____ ____ (0-9, 0-9) — **100 combinations: 00-99**

3-digit, decimal numbers (r=10, n=3) ____ ____ ____ — **1000 combinations: 000-999**

4-bit, binary numbers (r=2, n=4) ____ ____ ____ ____ (0-1, 0-1, 0-1, 0-1) — **16 combinations: 0000-1111**

6-bit, binary numbers (r=2, n=6) ____ ____ ____ ____ ____ ____ — **64 combinations: 000000-111111**

Main Point: Given n digits of base r, **r^n** unique numbers can be made with the range [0 - (r^n-1)]

Range of C Data Types

- For a given integer data type we can find its range by raising 2 to the n, 2^n (where n = number of bits of the type)
  - For signed representations we break the range in half with half negative and half positive (0 is considered a positive number by common integer convention)

Bytes	Bits	Type	Unsigned Range	Signed Range
1	8	[unsigned] char	0 to 255	-128 to +127
2	16	[unsigned] short	0 to 65535	-32768 to +32767
4	32	[unsigned] int	0 to 4,294,967,295	-2,147,483,648 to +2,147,483,648
8	8	[unsigned] long long	0 to 18,446,744,073,709,551,615	-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807
4 (8)	32 (64)	char*	0 to 18,446,744,073,709,551,615	

- How will I ever remember those ranges?
  - I wish I had an easy way to approximate those large numbers!

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) = 1 Kilo-
  - $2^{20} \approx 10^6$ (1 million) = 1 Mega-
  - $2^{30} \approx 10^9$ (1 billion) = 1 Giga-
  - $2^{40} \approx 10^{12}$ (1 trillion) = 1 Tera-
- 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}
  - 16-bit word: 64K numbers
  - 32-bit dword: 4G numbers
  - 64-bit qword: 16 million trillion numbers

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

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

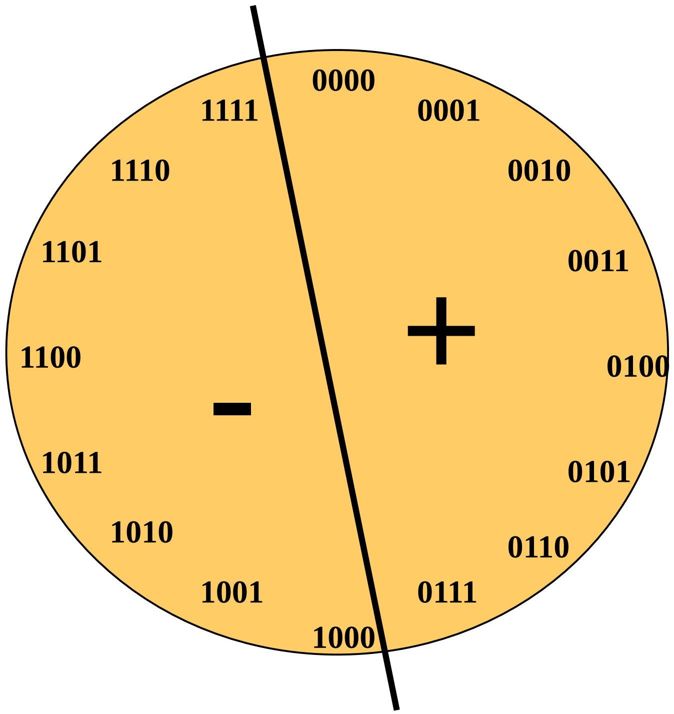
$2^{28} = 2^8 * 2^{20}$
$\approx 256 * 10^6 = 256{,}000{,}000$

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

CONVERTING SIGNED NUMBERS TO DECIMAL

Signed numbers

- Systems used to represent signed numbers split the possible binary combinations in half (half for positive numbers / half for negative numbers)
- Generally, positive and negative numbers are separated using the MSB
  - MSB=1 means negative
  - MSB=0 means positive



2’s Complement System

- Normal binary place values except MSB has negative weight
  - MSB of 1 = -2^{n-1}

4-bit Unsigned → **0 to 15**

Bit 3	Bit 2	Bit 1	Bit 0
8	4	2	1

4-bit 2’s complement → **-8 to +7**

Bit 3	Bit 2	Bit 1	Bit 0
-8	4	2	1

8-bit 2’s complement → **-128 to +127**

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
-128	64	32	16	8	4	2	1

2's Complement Examples

4-bit 2's complement

1	**0**	**1**	**1**	**= -5**
-8	4	2	1	
0	**0**	**1**	**1**	**= +3**
-8	4	2	1	
1	**1**	**1**	**1**	**= -1**
-8	4	2	1	

Notice that +3 in 2's comp. is the same as in the unsigned system

8-bit 2's complement

1	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**= -127**
-128	64	32	16	8	4	2	1	
0	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**= +25**
-128	64	32	16	8	4	2	1	

Important: Positive numbers have the same representation in 2's complement as in normal unsigned binary

2's Complement Range

- Given n bits...
  - Max positive value = 011...11
    - Includes all n-1 positive place values
  - Max negative value = 100...00
    - Includes only the negative MSB place value

Range with n-bits of 2's complement

$$[\ -2^{n-1} \text{ to } +2^{n-1}-1]$$

- Side note – What decimal value is 111...11?
  - -1_{10}

Unsigned and Signed Variables

- In C, 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	

- In C, 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	

IMPORTANT NOTE

- All computer systems use the **2's complement system** to represent **signed integers**!
- So from now on, if we say an integer is **signed**, we are actually saying it uses the 2's complement system unless otherwise specified
  - Other systems like "signed magnitude" or "1's complement" exist but will not be used for integers

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

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. ~~111~~10011 = 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.

Shortcuts for Converting Binary to Hexadecimal

SHORTHAND FOR BINARY

Binary and Hexadecimal

- Hex is base 16 which is 2^4
- 1 Hex digit $(?)_{16}$ can represent: 0-F $(0\text{-}15)_{10}$
- 4 bits of binary $(? ? ? ?)_2$ can represent: 0000-1111= $0\text{-}15_{10}$
- Conclusion…
  1 Hex digit = 4 bits

Binary to Hex

- Make groups of 4 bits starting from radix point and working outward
- Add 0's where necessary
- Convert each group of 4 to an octal digit

0 0 0 1 0 1 0 0 1 1 1 0 . 1 1 0 0

1 4 E C

= $14E.C_{16}$

0 1 1 0 1 0 1 1 . 1 0 1 0

6 B A

= $6B.A_{16}$

Hex to Binary

- Expand each hex digit to a group of 4 bits

$14E.C_{16}$

0 0 0 1 0100 1110 . 11 0 0

$= 101001110.11_2$

$D93.8_{16}$

110110010011.1000_2

$= 110110010011.1_2$

Hexadecimal Representation

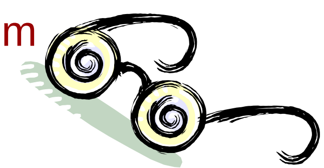
- Since values in modern computers are many bits, we use hexadecimal as a shorthand notation (4 bits = 1 hex digit)
  - 11010010 = D2 hex or 0xD2 if you write it in C/C++
  - 0111011011001011 = 76CB hex or 0x76CB if you write it in C/C++

Interpreting Hex Strings

- What does the following hexadecimal represent?
- Just like binary, you must know the underlying *representation system* being used before you can interpret a hex value
- Information (value) = Hex + Context (System)
  - For now, best be is to convert to binary, then translate

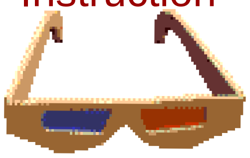
0x41 = ?

Unsigned Binary system



65 decimal

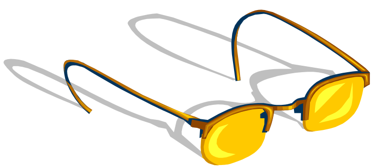
x86 Assembly Instruction



`inc %ecx`

(Add 1 to the ecx register)

ASCII system



$\text{'A'}_{\text{ASCII}}$

Hexadecimal & Sign

- If a number is represented in 2's complement (e.g. 10010110) then the MSB of its binary representation would correspond to:
  - 0 = Positive
  - 1 = Negative
- If that same 2's complement number were viewed as hex (e.g. 0x96) how could we tell if the corresponding number is positive or negative?
  - MSD of 0-7 = Positive
  - MSD of 8-F = Negative

Hex – Binary – Sign
0 = 0000 = Pos.
1 = 0001 = Pos.
2 = 0010 = Pos.
3 = 0011 = Pos.
4 = 0100 = Pos.
5 = 0101 = Pos.
6 = 0110 = Pos.
7 = 0111 = Pos.
8 = 1000 = Neg.
9 = 1001 = Neg.
A = 1010 = Neg.
B = 1011 = Neg.
C = 1100 = Neg.
D = 1101 = Neg.
E = 1110 = Neg.
F = 1111 = Neg.

Implicit and Explicit

APPLICATION: CASTING

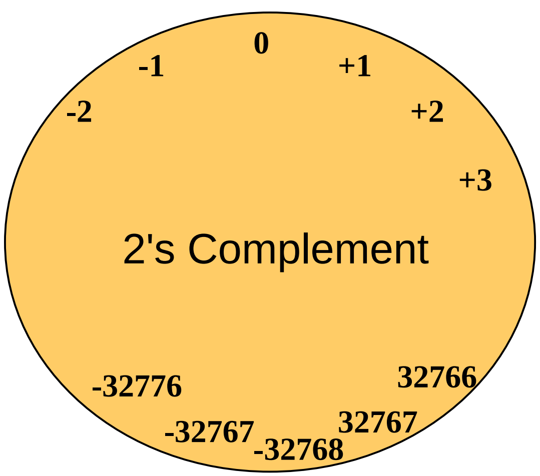
Implicit and Explicit Casting

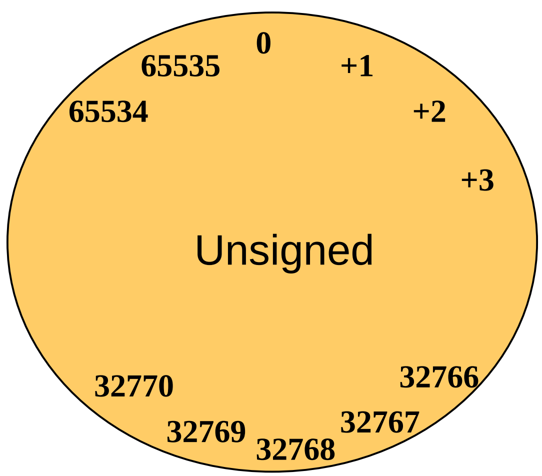
- Use your understanding of unsigned and 2's complement to predict the output
- Notes:
  - unsigned short range: 0 to 65535
  - signed short range: -32768 to +32768

```
int main()
{
  short int v = -10000; /* 0xd8f0 */
  unsigned short uv = (unsigned short) v;
  printf("v = %d, uv = %u\n", v, uv);
  return 0;
}
```

Expected Output:

```
v = -10000, uv = 55536
```






```
int main()
{
  unsigned u = 4294967295u;  /* UMax */
  int tu = (int) u;
  printf("u = %u, tu = %d\n", u, tu);
  return 0;
}
```

Expected Output:

```
u = 4294967295, tu = -1
```

Implicit and Explicit Casting

- Use your understanding of zero and sign extension to predict the output

```
int main()
{
  short int v = 0xcfc7; /* -12345 */
  unsigned short uv = 0xcfc7; /* 53191 */
  int vi = v; /* ??? */
  unsigned uvi = uv; /* ??? */
  printf("vi = %x, uvi = %x\n", vi, uvi);
  return 0;
}
```

Expected Output:

```
vi = ffffcfc7, uvi = cfc7
```

```
int main()
{
  int x = 53191; /* 0xcfc7 */
  short sx = x;
  int y = sx;
  char z = x;

  printf("sx = %d, y = %d ", sx, y);
  printf("z = %d\n", z);
  return 0;
}
```

Expected Output:

```
sx = -12345, y = -12345, z = -57
```

Advice

- Casting can be done implicitly and explicitly
- Casting from one system to another applies a new "interpretation" (pair of glasses) on the same bits
- Casting from one size to another will perform extension or truncation (based on the system)