CSC 252:
Data Representation

Bits, Bytes, and Integers

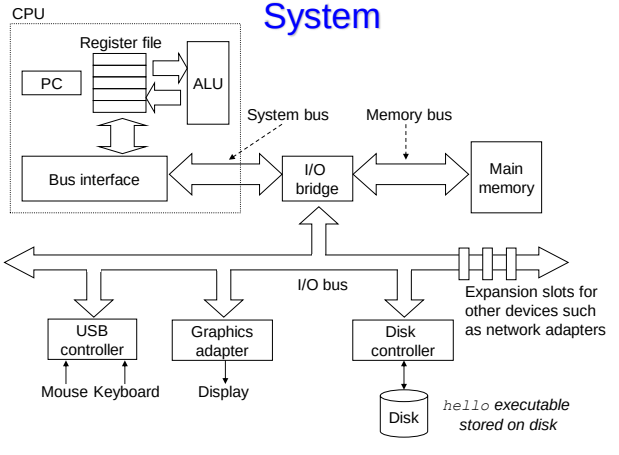
Last Week's Action Items

- Get a CSUG account
  - at https:// accounts.csug.rochester.edu/
  - cycle1.csug.rochester.edu (or cycle2, cycle3)
  - Get familiar with using Linux and C
  - Attend an office hour this week!
- Accept the Academic Honesty Policy on blackboard
- Introduce yourself: "meet your classmates" forum
- Acquire the textbook for the course
  - Read Chapter 1, start reading Chapter 2
- Finish Quiz 0

This Week's Action Items

- Read Chapter 2
- Finish Quiz 1 on Blackboard
- Start on Assignment 1
  - Finish Pre-Assignment 1 on Blackboard
    - Due Date: Thursday September 3 at noon

Hardware Organization of a Typical System



Hardware Components of a Computer System

- Processor
  - Datapath
  - Control
- Memory
- Input and Output devices

5

5

The Principle of Abstraction

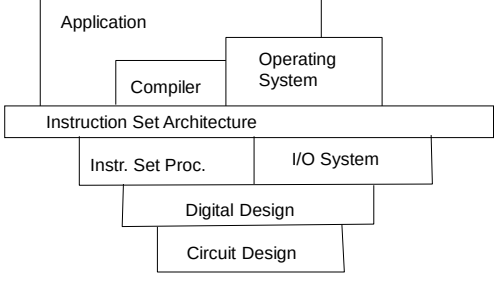
- Grouping principle
  - Levels/layers of abstraction by which each layer only needs to understand that immediately above and below it

6

6

What is Computer Architecture?

- Coordination of levels of abstraction under a set of rapidly changing forces



7

7

Topics to be covered:

- Data representation and computer arithmetic
- Assembly-level programs and instruction-set architectures
- Processor architectures
- Memory and storage hierarchies
- Performance optimization
- Exceptional control flow
- I/O devices
- Concurrency

8

8

Data Representation

- Memory: a large single-dimensional, conventionally byte-addressable, untyped array
- Byte ordering – big versus little endian
- Possible common interpretations
  - Instruction
  - Integer
  - Floating point
  - character

9

9

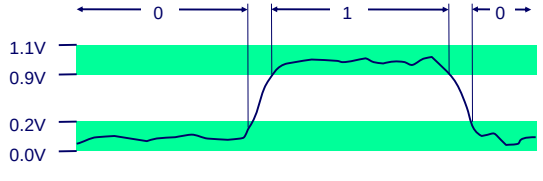
Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- Representations in memory, pointers, strings

10

Everything is bits

- Each bit is 0 or 1
- By encoding/interpreting sets of bits in various ways
  - Computers determine what to do (instructions)
  - … and represent and manipulate numbers, sets, strings, etc…
- Why bits? Electronic Implementation
  - Easy to store with bistable elements
  - Reliably transmitted on noisy and inaccurate wires



11

Number Representation

An n digit number can be represented in any base as

MSD … LSD

n-1 … 0

The value of the *i*th digit d is d x base^{i}, where *i* starts at 0 and increases from right to left

Decimal (base 10) is the natural human representation, binary (base 2) is the natural computer representation

E.g. $1100_2 = 1x2^3 + 1x2^2 + 0x2^1 + 0x2^0 = 12_{10}$

12

12

For example, can count in binary

- Base 2 Number Representation
  - Represent 15213_{10} as
  - Represent 1.20_{10} as
  - Represent 1.5213×10^4 as

13

For example, can count in binary

- Base 2 Number Representation
  - Represent 15213_{10} as 11101101101101_2
  - Represent 1.20_{10} as $1.0011001100110011[0011]\ldots_2$
  - Represent 1.5213×10^4 as $1.1101101101101_2 \times 2^{13}$

14

Encoding Byte Values

- Byte = 8 bits
  - Binary 00000000_2 to 11111111_2
  - Decimal: 0_{10} to 255_{10}
  - Hexadecimal 00_{16} to FF_{16}
    - Base 16 number representation
    - Use characters '0' to '9' and 'A' to 'F'
    - Write $FA1D37B_{16}$ in C as
      - 0xFA1D37B
      - 0xfa1d37b

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

15

Bit-Level Operations in C

- Logical
  - || && !
- Bitwise
  - | & ^ ~ >> <<
  - Arithmetic versus logical right shift

16

16

Example Data Representations

C Data Type	Typical 32-bit	Typical 64-bit	x86-64
`char`	1	1	1
`short`	2	2	2
`int`	4	4	4
`long`	4	8	8
`float`	4	4	4
`double`	8	8	8
`long double`	–	–	10/16
pointer	4	8	8

17

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- Representations in memory, pointers, strings

18

Boolean Algebra

- Developed by George Boole in 19th Century
  - Algebraic representation of logic
    - Encode "True" as 1 and "False" as 0

And

- **A&B = 1 when both A=1 and B=1**

&	0	1
0	0	0
1	0	1

Or

- **A|B = 1 when either A=1 or B=1**

|	0	1
0	0	1
1	1	1

Not

- **~A = 1 when A=0**

~	
0	1
1	0

Exclusive-Or (Xor)

- **A^B = 1 when either A=1 or B=1, but not both**

^	0	1
0	0	1
1	1	0

19

Algebraic Laws for Logical Expressions

- AND and OR are commutative
- AND and OR are associative
- AND is distributive over OR; OR is distributive over AND
- TRUE is the identity for AND; FALSE is the identity for OR
- FALSE annihilates AND; TRUE annihilates OR
- AND and OR are idempotent (p AND p ≡ p OR p ≡ p)
- Subsumption
  - (p OR (p AND q)) ≡ (p AND (p OR q)) ≡ p
- DeMorgan's Laws:
  - NOT(p AND q) ≡ (NOT p) OR (NOT q)
  - NOT(p OR q) ≡ (NOT p) AND (NOT q)

20

20

General Boolean Algebras

- Operate on Bit Vectors
  - Operations applied bitwise

```
  01101001     01101001     01101001
& 01010101   | 01010101   ^ 01010101   ~ 01010101
  01000001     01111101     00111100     10101010
```

- All of the Properties of Boolean Algebra Apply

Example: Representing & Manipulating Sets

- Representation
  - Width w bit vector represents subsets of {0, …, w–1}
  - $a_j = 1$ if $j \in A$
    - 01101001 { 0, 3, 5, 6 }
    - *76543210*
    - 01010101 { 0, 2, 4, 6 }
    - *76543210*
- Operations

– &	Intersection	01000001	{ 0, 6 }
– |	Union	01111101	{ 0, 2, 3, 4, 5, 6 }
– ^	Symmetric difference	00111100	{ 2, 3, 4, 5 }
– ~	Complement	10101010	{ 1, 3, 5, 7 }

Bit-Level Operations in C

- Operations &, |, ~, ^ Available in C
  - Apply to any "integral" data type
    - long, int, short, char, unsigned
  - View arguments as bit vectors
  - Operations applied bit-wise
- Examples (Char data type)
  - ~0x41 -> 0xBE
    - $\sim 01000001_2 \rightarrow 10111110_2$
  - ~0x00 -> 0xFF
    - $\sim 00000000_2 \rightarrow 11111111_2$
  - 0x69 & 0x55 -> 0x41
    - $01101001_2 \;\&\; 01010101_2 \rightarrow 01000001_2$
  - 0x69 | 0x55 -> 0x7D
    - $01101001_2 \mid 01010101_2 \rightarrow 01111101_2$

Contrast: Logic Operations in C

- Contrast to Logical Operators
  - &&, ||, !
    - View 0 as "False"
    - Anything nonzero as "True"
    - Always return 0 or 1
    - Early termination
- Examples (char data type)
  - !0x41 -> 0x00
  - !0x00 -> 0x01
  - !!0x41 -> 0x01
  - 0x69 && 0x55 -> 0x01
  - 0x69 || 0x55 -> 0x01

Contrast: Logic Operations in C

- Contrast to Logical Operators
  - &&, ||, !
    - View 0 as "Fal
    - Anyt
    - Alwa
    - Early
- Exampl
  - !0x41 -> 0x00
  - !0x00 -> 0x01
  - !!0x41 -> 0x01
  - 0x69 && 0x55 -> 0x01
  - 0x69 || 0x55 -> 0x01

Watch out for && vs. & (and || vs. |)…
a common bug in C programming

25

Shift Operations

- Left Shift: `x << y`
  - Shift bit-vector **x** left **y** positions
    - Throw away extra bits on left
  - Fill with 0's on right
- Right Shift: `x >> y`
  - Shift bit-vector **x** right **y** positions
    - Throw away extra bits on right
  - Logical shift
    - Fill with 0's on left
  - Arithmetic shift
    - Replicate most significant bit on left
- Undefined Behavior
  - Shift amount < 0 or ≥ word size

Argument x	01100010
<< 3	00010000
Log. >> 2	00011000
Arith. >> 2	00011000

Argument x	10100010
<< 3	00010000
Log. >> 2	00101000
Arith. >> 2	11101000

26

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - **Representation: unsigned and signed**
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- Representations in memory, pointers, strings
- Summary

27

Representing Positive and Negative Integers

- Sign-Magnitude - MSB represents sign (0 for +ve, 1 for -ve)
- One's Complement of $x = 2^n - x - 1$ (complement individual bits)

Problem: Balanced representation, but two values for 0

Solution:

- Two's Complement of $x = 2^n - x$ (radix complement; most common representation)
  - single bit pattern for 0
  - ensures that $x + (-x)$ is 0
  - still keeps 1 in MSB for a -ve number (sign bit)
  - 100... represents the most -ve number
  - E.g. 4-bit 2's complement number $1100_2 = -1x2^3 + 1x2^2 + 0x2^1 + 0x2^0 = -4_{10}$

28

28

Integer Arithmetic

- Normal base 2 2's complement addition works on both positive and negative numbers
- Shortcuts
  - 2's complement = 1s' complement + 1
  - 2's complement representation of n digit number as n+m digit number --- sign extend

29

Encoding Integers

Unsigned

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Two's Complement

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Sign Bit

```
short int x =  15213;
short int y = -15213;
```

- C `short` 2 bytes long

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
y	-15213	C4 93	11000100 10010011

- Sign Bit
  - For 2's complement, most significant bit indicates sign
    - 0 for nonnegative
    - 1 for negative

30

Two-complement Encoding Example (Cont.)

```
x =     15213: 00111011 01101101
y =    -15213: 11000100 10010011
```

Weight	15213		-15213	
1	1	1	1	1
2	0	0	1	2
4	1	4	0	0
8	1	8	0	0
16	0	0	1	16
32	1	32	0	0
64	1	64	0	0
128	0	0	1	128
256	1	256	0	0
512	1	512	0	0
1024	0	0	1	1024
2048	1	2048	0	0
4096	1	4096	0	0
8192	1	8192	0	0
16384	0	0	1	16384
-32768	0	0	1	-32768
Sum		**15213**		**-15213**

31

Numeric Ranges

- Unsigned Values
  - *UMin* = 0
    000…0
  - *UMax* = $2^w - 1$
    111…1
- Two's Complement Values
  - *TMin* = -2^{w-1}
    100…0
  - *TMax* = $2^{w-1} - 1$
    011…1
- Other Values
  - Minus 1
    111…1

Values for $W = 16$

	Decimal	Hex	Binary
UMax	65535	FF FF	11111111 11111111
TMax	32767	7F FF	01111111 11111111
TMin	-32768	80 00	10000000 00000000
-1	-1	FF FF	11111111 11111111
0	0	00 00	00000000 00000000

32

Values for Different Word Sizes

	W			
	8	16	32	64
UMax	255	65,535	4,294,967,295	18,446,744,073,709,551,615
TMax	127	32,767	2,147,483,647	9,223,372,036,854,775,807
TMin	-128	-32,768	-2,147,483,648	-9,223,372,036,854,775,808

- Observations
  - $|TMin| = TMax + 1$
    - Asymmetric range
  - $UMax = 2 * TMax + 1$

- **C Programming**
  - #include <limits.h>
  - Declares constants, e.g.,
    - ULONG_MAX
    - LONG_MAX
    - LONG_MIN
  - Values platform specific

Unsigned & Signed Numeric Values

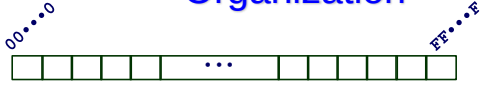
X	B2U(X)	B2T(X)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	–8
1001	9	–7
1010	10	–6
1011	11	–5
1100	12	–4
1101	13	–3
1110	14	–2
1111	15	–1

- Equivalence
  - Same encodings for nonnegative values
- Uniqueness
  - Every bit pattern represents unique integer value
  - Each representable integer has unique bit encoding
- ⇒ Can Invert Mappings
  - $U2B(x) = B2U^{-1}(x)$
    - Bit pattern for unsigned integer
  - $T2B(x) = B2T^{-1}(x)$
    - Bit pattern for two's comp integer

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- Representations in memory, pointers, strings

Byte-Oriented Memory Organization



- Programs refer to data by address
  - Conceptually, envision it as a very large array of bytes
    - In reality, it's not, but can think of it that way
  - An address is like an index into that array
    - and, a pointer variable stores an address
- Note: system provides private address spaces to each "process"
  - Think of a process as a program being executed
  - So, a program can clobber its own data, but not that of others

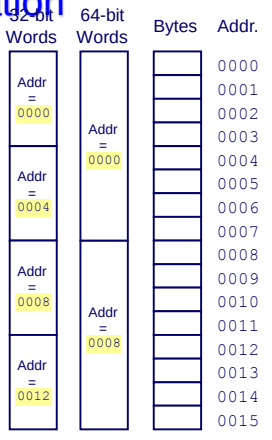
Machine Words

- Any given computer has a "Word Size"
  - Nominal size of integer-valued data
    - and of addresses
  - Until recently, most machines used 32 bits (4 bytes) as word size
    - Limits addresses to 4GB (2^{32} bytes)
  - Increasingly, machines have 64-bit word size
    - Potentially, could have 18 EB (exabytes) of addressable memory
    - That's 18.4 X 10^{18}
  - Machines still support multiple data formats
    - Fractions or multiples of word size
    - Always integral number of bytes

58

Word-Oriented Memory Organization

- Addresses Specify Byte Locations
  - Address of first byte in word
  - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)



59

Example Data Representations

C Data Type	Typical 32-bit	Typical 64-bit	x86-64
`char`	1	1	1
`short`	2	2	2
`int`	4	4	4
`long`	4	8	8
`float`	4	4	4
`double`	8	8	8
`long double`	–	–	10/16
pointer	4	8	8

60

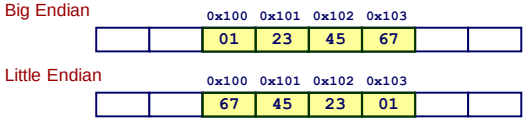
Byte Ordering

- So, how are the bytes within a multi-byte word ordered in memory?
- Conventions
  - Big Endian: Sun, PPC Mac, Internet
    - Least significant byte has highest address
  - Little Endian: x86, ARM processors running Android, iOS, and Windows
    - Least significant byte has lowest address

61

Byte Ordering Example

- Example
  - Variable x has 4-byte value of 0x01234567
  - Address given by &x is 0x100

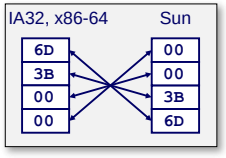


62

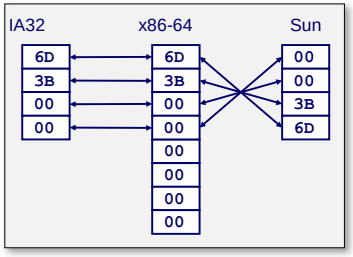
Representing Integers

Decimal:	15213
Binary:	0011 1011 0110 1101
Hex:	3 B 6 D

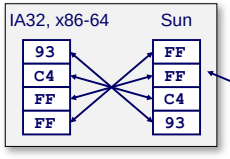
```
int A = 15213;
```




```
long int C = 15213;
```




```
int B = -15213;
```



Two's complement representation

63