ECE 250 / CS 250
Computer Architecture

“C to Binary: Memory & Data Representations”

Benjamin Lee

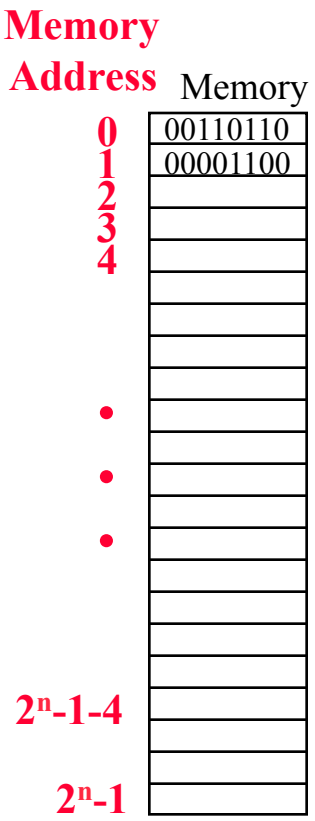
Slides based on those from Alvin Lebeck, Daniel Sorin, Andrew Hilton, Amir Roth, Gershon Kedem

Administrivia

- What did you learn last week?
  - Pointers are hard!
- Today
  - More pointers / memory
  - Data representations
- Resources (from course web page)
  - Video snippets by Prof Drew Hilton in ECE
    - Videos don't work with Firefox (use Safari or Chrome)
  - MIT Open Course

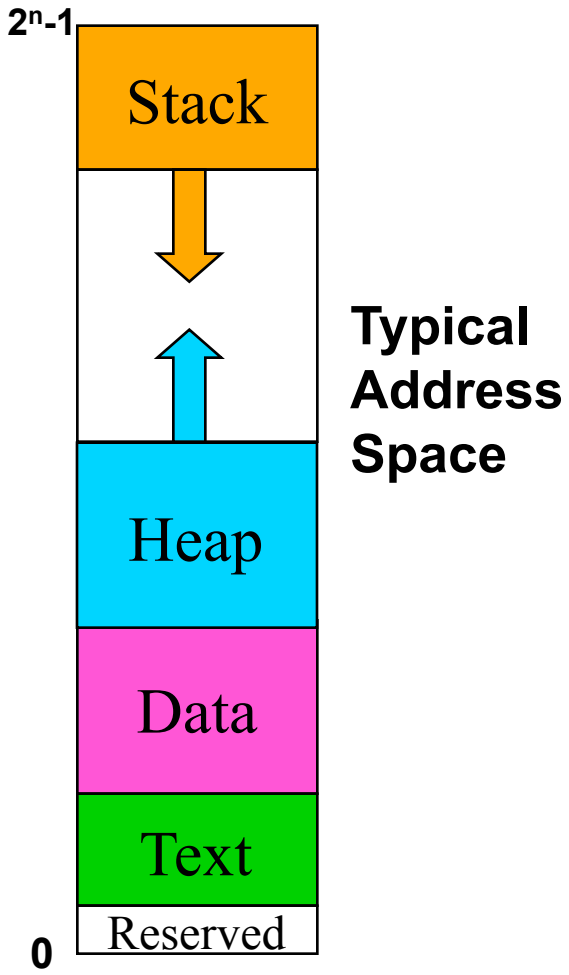
A Program's View of Memory

- What is Memory?
  - A large linear array of bits
- Find things by indexing into array
  - memory address (unsigned integer)
  - read to and write from address
- Processor issues commands to read/write specific locations
  - Read from memory location 0x1400
  - Write 0xff to memory location 0x8675309
- Array of …
  - Bytes? 32-bit ints? 64-bit ints?



Memory Partitions

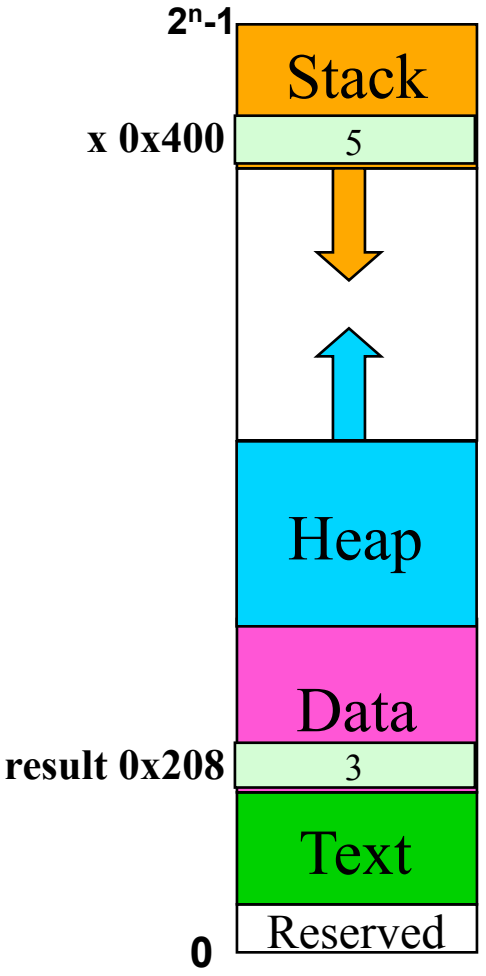
- Text for instructions
  - add dest, src1, src2
  - mem[dest] = mem[src1] + mem[src2]
- Data
  - static (constants, global variables)
  - dynamic (heap, new allocated)
  - grows up
- Stack
  - local variables
  - grows down
- Variables are names for memory locations
  - int x;



A Simple Program's Memory Layout

```
...
int result; // global var
main()
{
   int x;
   ...
   result = x + result;
   ...
}
```

mem[0x208] = mem[0x400] + mem[0x208]



Pointers

- "address of" operator &
  - don't confuse with bitwise AND operator (later)

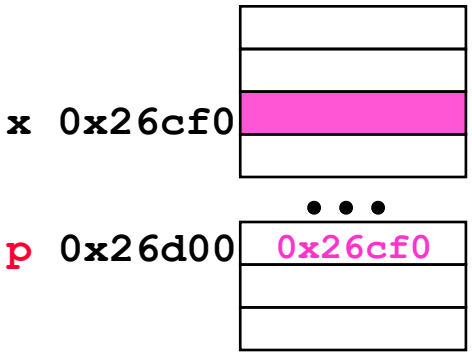
Given

```
int x; int* p;  // p points to an int
p = &x;
```

Then

```
*p = 2;  and x = 2; produce the same result
 Note: p is a pointer, *p is an int
```

- What happens for `p = 2`?
- On 32-bit machine, p is 32-bits



C Memory Allocation

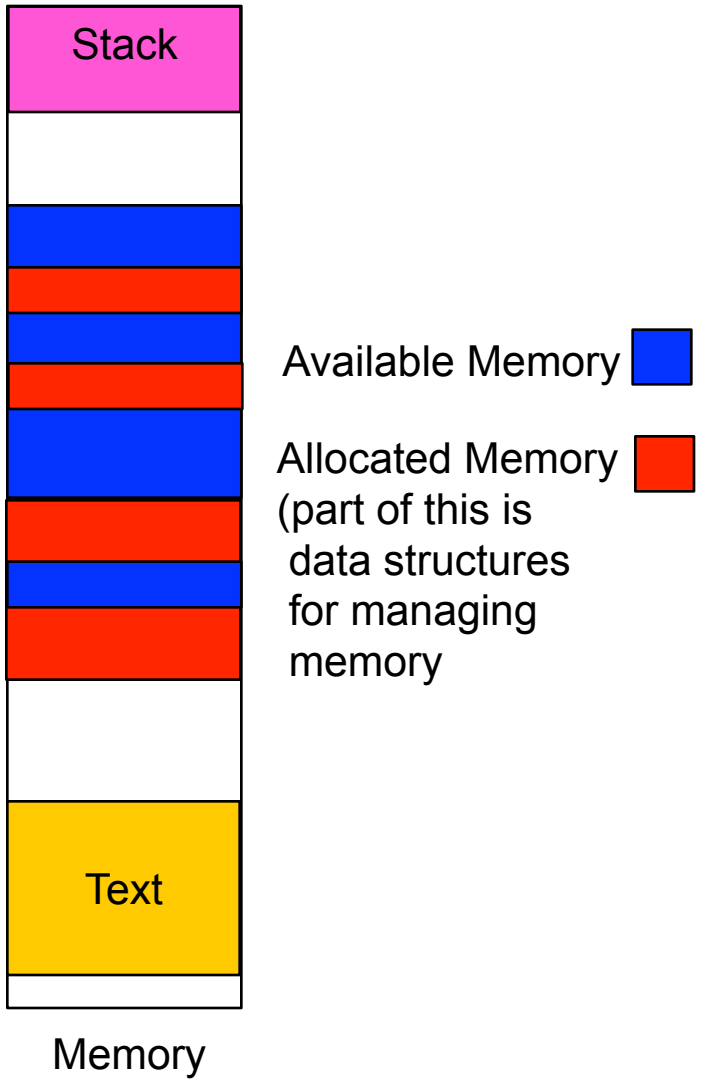
- **How do you allocate an object in Java?**
  - What do you do when you are finished with an object?
  - Garbage collection
  - Counts references to objects, when == 0 can reuse

- **C does not have garbage collection**
  - Must explicitly manage memory

- **void * malloc(nbytes)**
  - Obtain storage for your data (like new in Java)
  - Use *sizeof(type)*, which returns bytes needed for type
  - Cast return value into appropriate type(int) malloc(sizeof(int));
- **free(ptr)**
  - Return the storage when you are finished (no Java equivalent)
  - ptr must be a value previously returned from malloc

Memory Manager (Heap Manager)

- Malloc & free are library routines that handle memory management for the heap (allocation / deallocation)
- Java has garbage collection
- C must use free
  - else memory leak -> no more available memory
- Write a Heap Manager in Compsci 310



Linked List (two nodes)

```
#include <stdio.h>
#include <stdlib.h>
struct list_ent {
        int id;
        struct list_ent *next;
};
main()
{
  struct list_ent *head, *ptr;
  head = (struct list_ent *)
        malloc(sizeof(struct list_ent));
  head->id = 66;
  head->next = NULL;

  ptr = (struct list_ent *)
        malloc(sizeof(struct list_ent));
  ptr->id = 23;
  ptr->next = NULL;
  head->next = ptr;

  printf("head id: %d, next id: %d\n",
        head->id,head->next->id);

  ptr = head;
  head = ptr->next;

  printf("head id: %d, next id: %d\n",
        head->id, ptr->id);
  free(head);
  free(ptr);
}
```

Back to C: Command Line Arguments

- Parameters to main (int argc, char *argv[])
  - argc = number of arguments (0 to argc-1)
  - argv is array of strings (i.e., array of character pointers)
  - argv[0] = program name

```
main(int argc, char *argv[]) {
  int i;
  printf("%d arguments\n", argc);
  for (i=0; i< argc; i++)
  printf("argument %d: %s\n", i, argv[i]);
}
```

C Summary

- C Language is lower level than Java
- Many things are similar
  - Data types
  - Control flow
- Some important differences
  - No objects!
  - Explicit memory allocation/deallocation
- Create and compile a program
- Intro to Memory & Pointers
- Up Next:
  - So what are those chars, ints, floats?

Representations: Thought Experiment

- Do this at home with friends…
- Using only the three symbols @ # $ specify a representation for the following:
  - All integers from 0 to 10
  - Commands to 1) walk, 2) turn, 3) sit, 4) raise right arm, 5) raise left arm

- Using only your representation write down series of commands & integers (if appropriate, e.g., raise left arm-3, turn-2)
  - Must have at least 5 commands
- Give someone your representations from above and the series of commands, see if they can execute the commands.

Data Representation

- Compute two hundred twenty nine minus one hundred sixty seven divided by twelve
- Compute XIX - VII + IV
- We reason about numbers many different ways
- Computers store variables (data)
  - Typically numbers, characters or combination of these
- Computers have instructions (operations like add)
- The key is to use a representation that is "efficient"

Number Systems

- A number is a mathematical concept
  - 10
- Many ways to represent a number
  - 10, ten, 2x5, X, 100/10, ||||| |||||
- Symbols are used to create a representation

- Which representation is best for counting?
- Which representation is best for addition and subtraction?
- Which representation is best for multiplication and division?

More Number Systems

- Humans use decimal (base 10)
  - digits 0-9 are composed to make larger numbers

$$11 = 1*10^1 + 1*10^0$$

  - weighted positional notation
- Addition and Subtraction are straightforward
  - carry and borrow (today called regrouping)
- Multiplication and Division less so
  - can use logarithms and then do adds and subtracts

- The key is to use a representation that is "efficient"

Number Systems for Computers

- Today's computers are built from transistors
- Transistor is either off or on
- Need to represent numbers using only off and on
  - two symbols
- off and on can represent the digits 0 and 1
  - BIT is Binary Digit
  - A bit can have a value of 0 or 1

- Everything in a computer is represented using Bits!

Representing High Level Things in Binary

- Computers represent everything using binary (0 or 1)
- Instructions are specified using binary

- Instructions must be able to describe
  - Operation types (add, subtract, shift, etc.)
  - Data objects (integers, decimals, characters, etc.)
  - Memory locations

- Example:

```
int x, y;              // Where are x and y?  How to represent an int?
bool decision;         // How do we represent a bool?  Where is it?
y = x + 7;             // How do we specify "add"?  How to represent 7?
decision=(y>18);  //  Etc.
```

Representing Operation Types

- How do we tell computer to add? Shift? Read from memory? Etc.
- Arbitrarily! ☺

- Each Instruction Set Architecture (ISA) has its own binary encodings for each operation type

- E.g., in MIPS:
  - Integer add is: 00000 010000
  - Read from memory (load) is: 010011
  - Etc.
- More on Instruction Sets later this semester

Representing Data Types

- How do we specify an integer? A character? A floating point number? A bool? Etc.
- Same as before: binary!

- Key Idea: the same 32 bits might mean one thing if interpreted as an integer but another thing if interpreted as a floating point number, and yet another if interpreted as an instruction, etc.

Basic Data Types

Bit (bool): 0, 1

Bit String: sequence of bits of a particular length
- 4 bits is a nibble
- 8 bits is a byte
- 16 bits is a half-word
- 32 bits is a word
- 64 bits is a double-word
- 128 bits is a quad-word

Integers (int, long):
- "2's Complement" (32-bit or 64-bit representation)

Floating Point (float, double):
- Single Precision (32-bit representation)
- Double Precision (64-bit representation)
- Extended (Quad) Precision (128-bit representation)

Character (char):
- ASCII 7-bit code

Binary, Octal and Hexadecimal numbers

- Computers can input and output decimal numbers but they convert them to internal binary representation.
- Binary is good for computers, hard for us to read
  - Use numbers easily computed from binary
- Binary numbers use only two different digits: {0,1}
  - Example: $1200_{10} = 0000010010110000_2$
- Octal numbers use 8 digits: {0 - 7}
  - Example: $1200_{10} = 02260_8$
- Hexadecimal numbers use 16 digits: {0-9, A-F}
  - Example: $1200_{10} = 04B0_{16} = 0x04B0$
  - does not distinguish between upper and lower case

Issues for Binary Representation of Numbers

- There are many ways to represent numbers in binary
  - Binary representations are encodings → many encodings possible
  - What are the issues that we must address?

- Issue #1: Complexity of arithmetic operations
- Issue #2: Negative numbers
- Issue #3: Maximum representable number

- Choose representation that makes these issues easy for machine, even if it's not easy for humans

Unsigned Binary Numbers

- Weighted positional notation using base 2

$$11_{10} = 1*2^3 + 1*2^1 + 1*2^0 = 1011_2$$

$$11_{10} = \quad 8 \quad + \quad 2 \quad + \quad 1$$

- Only positive numbers
  - unsigned int data type

- What is largest number, given 4 bits?

Binary and Hex

- To convert to and from hex: group binary digits in groups of four and convert according to table
- $2^4 = 16$

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

Example:

1100 0010 0110 0111 0100 1111 1101 0101_2

C 2 6 7 4 F D 5_{16}

Sign Magnitude

- Use leftmost bit for + (0) or – (1):
- 6-bit example (1 sign bit + 5 magnitude bits):
- +17 = 010001
- -17 = 110001
- Pros:
  - Conceptually simple
  - Easy to convert
- Cons:
  - Harder to compute (add, subtract, etc.) with
  - Positive and negative 0: 000000 and 100000

1's Complement Representation for Integers

- Use largest positive binary numbers to represent negative numbers
- To negate a number,
  invert ("not") each bit:
  0 → 1
  1 → 0
- Cons:
  - two 0s
  - hard to compute with

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-7
1001	-6
1010	-5
1011	-4
1100	-3
1101	-2
1110	-1
1111	-0

2's Complement Integers

- Use large positives to represent negatives
- $(-x) = 2^n - x$

- This is 1's complement + 1
- $(-x) = 2^n - 1 - x + 1$
- Invert bits and add 1

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

6-bit examples:

$010110_2 = 22_{10}$; $101010_2 = -22_{10}$

$1_{10} = 000001_2$; $-1_{10} = 111111_2$

$0_{10} = 000000_2$; $-0_{10} = 000000_2$ → good!

Pros and Cons of 2's Complement

- Advantages:
  - Only one representation for 0 (unlike 1's comp): 0 = 000000
  - Addition algorithm is much easier than with sign and magnitude
    - Independent of sign bits

- Disadvantage:
  - One more negative number than positive
  - Example: 6-bit 2's complement number
    100000_2 = -32_{10}; but 32_{10} could not be represented

All modern computers use 2's complement for integers

2's Complement Precision Extension

- Most computers today support 32-bit (int) or 64-bit integers
  - Specify 64-bit using gcc C compiler with long long type
- To extend precision, use sign bit extension
  - Integer precision is number of bits used to represent a number

Examples

$14_{10} = 001110_2$ in 6-bit representation.

$14_{10} = 000000001110_2$ in 12-bit representation

$-14_{10} = 110010_2$ in 6-bit representation

$-14_{10} = 111111110010_2$ in 12-bit representation.

Binary Math : Addition

- Suppose we want to add two 8-bit numbers:

```
   00011101
+  00101011
```

- How do we do this?

Binary Math : Addition

- Suppose we want to add two 8-bit numbers:

```
   00011101          695
+  00101011        + 232
```

- How do we do this?
  - Let's revisit decimal addition
  - Think about the process as we do it

Binary Math : Addition

- Suppose we want to add two 8-bit numbers:

```
   00011101          695
+  00101011        + 232
-----------        -----
                      7
```

- First add one's digit 5+2 = 7

Binary Math : Addition

- Suppose we want to add two 8-bit numbers:

```
     00011101                  1
+    00101011                695
                           + 232
                              27
```

- First add one's digit 5+2 = 7
- Next add ten's digit 9+3 = 12 (2 carry a 1)

Binary Math : Addition

- Suppose we want to add two 8-bit numbers:

```
   00011101          695
+  00101011        + 232
----------        -----
                    927
```

- First add one's digit 5+2 = 7
- Next add ten's digit 9+3 = 12 (2 carry a 1)
- Last add hundred's digit 1+6+2 = 9

Binary Math : Addition

- Suppose we want to add two 8-bit numbers:

```
    00011101
+   00101011
```

- Back to the binary:
  - First add 1's digit 1+1 = …?

Binary Math : Addition

- Suppose we want to add two 8-bit numbers:

```
          1
    00011101
+   00101011
-------------
         00
```

- Back to the binary:
  - First add 1's digit 1+1 = 2 (0 carry a 1)

Binary Math : Addition

- Suppose we want to add two 8-bit numbers:

```
          11
    00011101
+   00101011
-------------
          00
```

- Back to the binary:
  - First add 1's digit 1+1 = 2 (0 carry a 1)
  - Then 2's digit: 1+0+1 =2 (0 carry a 1)
- You all finish it out….

Binary Math : Addition

- Suppose we want to add two 8-bit numbers:

```
    111111
   00011101    = 29
 + 00101011    = 43
 ------------------
   01001000    = 72
```

- Can check our work in decimal

Binary Math : Addition

- What about this one:

```
    01011101
 +  01101011
```

Binary Math : Addition

- What about this one:

```
    1111111
    01011101    =  93
+   01101011    = 107
-------------------
    11001000    = -56
```

- But… that can't be right?
  - What do you expect for the answer?
  - What is your expected answer in 8-bit signed 2's complement?

Software Implication! Integer Overflow

- Answer should be 200
  - Not representable in 8-bit signed representation
  - No right answer
- Call Integer Overflow
- Real problem in programs

Subtraction

- 2's complement makes subtraction easy:
  - Remember: A - B = A + (-B)
  - And: -B = ~B + 1

    ↑ that means flip bits ("not")
  - So we just flip the bits and start with CI = 1
  - Later: No new circuits to subtract (re-use add)

```
                                  1
   0110101        ->        0110101
-  1010010                + 0101101
```

What About Non-integer Numbers?

- There are infinitely many real numbers between two integers
- Many important numbers are real
  - Speed of light ~= $3x10^8$
  - Pi = 3.1415…
- Fixed number of bits limits range of integers
  - Can't represent some important numbers
- Humans use Scientific Notation
  - $1.3x10^4$

Option 1: Fixed Point Representation

- Decimal fixed point (6 digits)
  - 102832
  - 1028.32
  - Same digits, different decimal point
  - Right of decimal point weight is $1/10^{-i}$ (i starts at 1 -> 1/10, 1/100, …)
- Binary Point (6 bits)
  - 001010
  - 0010.10
  - Right of binary point weight is $1/2^{-i}$ (i starts at 1 -> ½, ¼, 1/8,…)
- General Fixed Point representation specifies
  - Total number of bits (i.e., width)
  - Fixed point position (e.g., fixed<6,0> fixed<6,2>)
- It's all just bits…what decimal values are above binary numbers?

Fixed Point Representation

- Pros:
  - Addition/subtraction just like integers (“free”)
- Cons:
  - Mul/div require renormalizing (divide by 64K)
  - Range limited (no good rep for large + small)
- Can be good in specific situations

Can we do better?

- Think about scientific notation for a second:
- For example:

  $6.82 * 10^{23}$

- Real number, but comprised of ints:
  - 6 generally only 1 digit here
  - 82 any number here
  - 10 always 10 (base we work in)
  - 23 can be positive or negative
- Can we do something like this in binary?

Option 2: Floating Point Representation

- How about:
  - +/- X.YYYYYY * $2^{+/-N}$

- Big numbers: large positive N
- Small numbers (<1): negative N
- Numbers near 0: small N

- This is “floating point” : most common way