

Machine Level Programming: Basics

CSCI 237: Computer Organization
10th Lecture, Friday, Oct. 2

Kelly Shaw

Administrative Details

- Lab #2b due Wednesday at 11pm
- Suggested reading CSAPP 3.1-3.4
- CS Colloquium today at 2:35pm in Wege

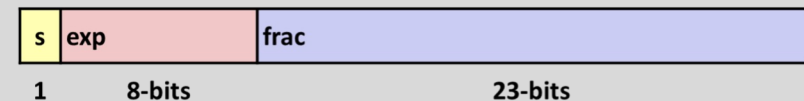
Last Time: Floating Point (part II)

- IEEE FP standard (normalized and denormalized values)
- Tiny Floating Point Example
- Floating point in C
- Summary

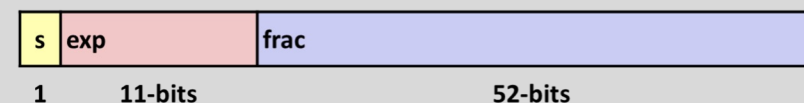
Summary

- IEEE Floating Point has clear mathematical properties
- Represents numbers of form $M \times 2^E$
- One can reason about operations independent of implementation
 - As if computed with perfect precision and then rounded
- Not the same as real arithmetic
 - Violates associativity/distributivity
 - Makes life difficult for compilers & serious numerical applications programmers

Single precision: 32 bits

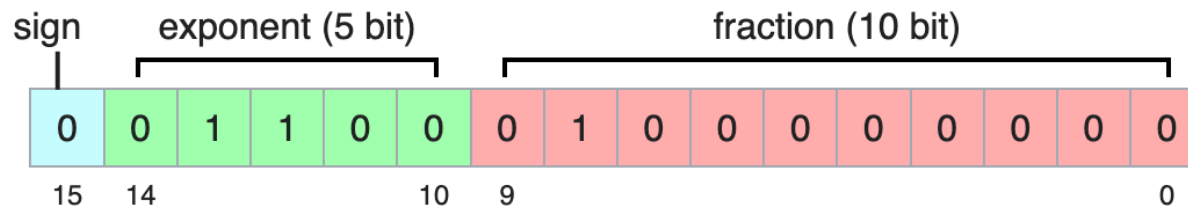


Double precision: 64 bits

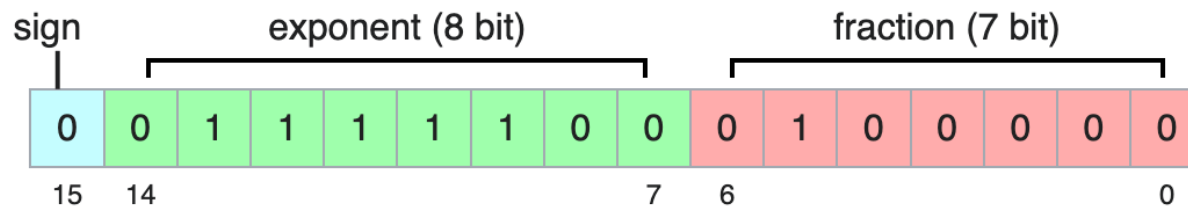


In-class Problem

IEEE half-precision 16-bit float

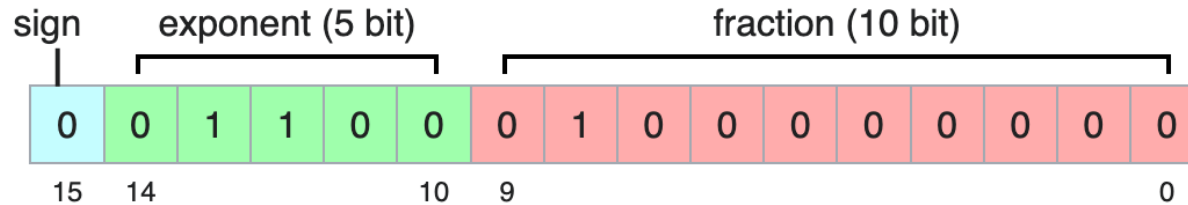


bfloat16

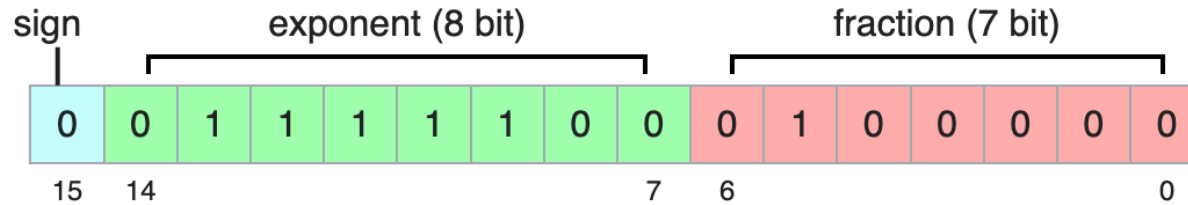


- What are the largest positive normalized numbers that can be represented by the 2 floating point number formats?
- What is the largest distance between any two adjacent numbers in the two number formats?

IEEE half-precision 16-bit float



bfloat16



- Bias = $2^{(\text{exp_width}-1)} - 1$
- Exp values for normalized numbers: 00...001 to 1111...10

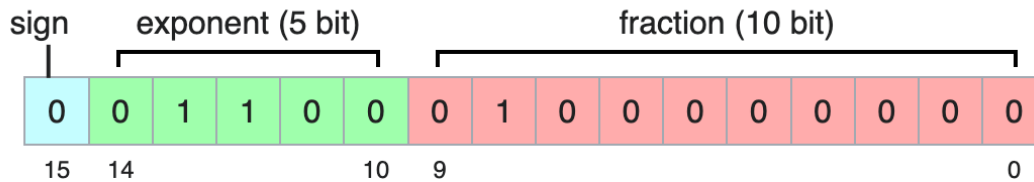
Half-precision

- Bias = $2^{(5-1)} - 1 = 15$
- Max exp = 11110 = 30
- Max E = 30 - 15 = 15
- Max pos = $1.1111111111 \cdot 2^{15}$
= $11111111111 \cdot 2^5$
= $2047 \cdot 32 = 65504$

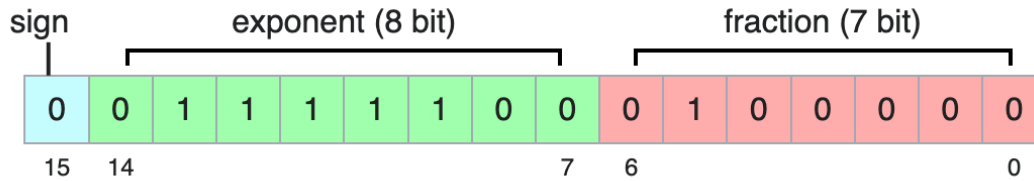
bfloat16

- Bias = $2^{(8-1)} - 1 = 127$
- Max exp = 11111110 = 254
- Max E = 254 - 127 = 127
- Max pos = $1.1111111 \cdot 2^{127}$
= $11111111 \cdot 2^{120}$
= $255 \cdot 2^{120} = 3.39 \cdot 10^{38}$

IEEE half-precision 16-bit float



bfloat16



	<u>s</u>	<u>exp</u>	<u>frac</u>	<u>E</u>	<u>Value</u>
Denormalized numbers	0	0000	000	-6	0
	0	0000	001	-6	$1/8 * 1/64 = 1/512$
	0	0000	010	-6	$2/8 * 1/64 = 2/512$
	...				
	0	0000	110	-6	$6/8 * 1/64 = 6/512$
	0	0000	111	-6	$7/8 * 1/64 = 7/512$
	0	0001	000	-6	$8/8 * 1/64 = 8/512$
	0	0001	001	-6	$9/8 * 1/64 = 9/512$
	...				
	0	0110	110	-1	$14/8 * 1/2 = 14/16$
Normalized numbers	0	0110	111	-1	$15/8 * 1/2 = 15/16$
	0	0111	000	0	$8/8 * 1 = 1$
	0	0111	001	0	$9/8 * 1 = 9/8$
	0	0111	010	0	$10/8 * 1 = 10/8$
	...				
	0	1110	110	7	$14/8 * 128 = 224$
	0	1110	111	7	$15/8 * 128 = 240$
	0	1111	000	n/a	inf

■ Largest space between numbers at high end of normalized

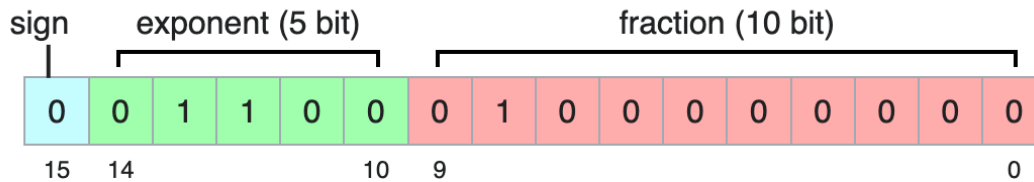
Half-precision

- 2nd to largest: ?
- Largest: ?

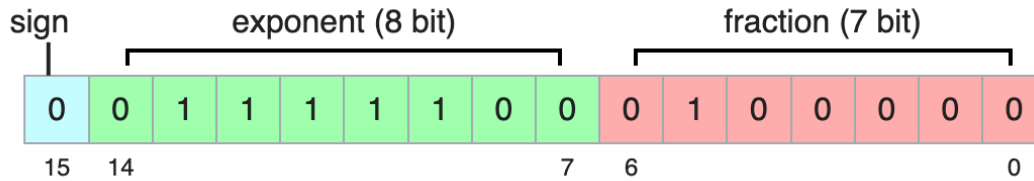
Bfloat

- 2nd to largest: ?
- Largest: :

IEEE half-precision 16-bit float



bfloat16



	<u>s</u>	<u>exp</u>	<u>frac</u>	<u>E</u>	<u>Value</u>
Denormalized numbers	0	0000	000	-6	0
	0	0000	001	-6	$1/8 * 1/64 = 1/512$
	0	0000	010	-6	$2/8 * 1/64 = 2/512$
	...				
	0	0000	110	-6	$6/8 * 1/64 = 6/512$
	0	0000	111	-6	$7/8 * 1/64 = 7/512$
Normalized numbers	0	0001	000	-6	$8/8 * 1/64 = 8/512$
	0	0001	001	-6	$9/8 * 1/64 = 9/512$
	...				
	0	0110	110	-1	$14/8 * 1/2 = 14/16$
	0	0110	111	-1	$15/8 * 1/2 = 15/16$
	0	0111	000	0	$8/8 * 1 = 1$
	0	0111	001	0	$9/8 * 1 = 9/8$
	0	0111	010	0	$10/8 * 1 = 10/8$
	...				
	0	1110	110	7	$14/8 * 128 = 224$
	0	1110	111	7	$15/8 * 128 = 240$
	0	1111	000	n/a	inf

■ Largest space between numbers at high end of normalized

Half-precision

- 2nd to largest: 0 11110 1111111110
- Largest: 0 11110 1111111111
- Difference: $0.0000000001 * 2^{(15)}$
 $= 1 * 2^5$

Bfloat

- 2nd to largest: 0 11111110 1111110
- Largest: 0 11111110 1111111
- Difference: $0.0000001 * 2^{(127)}$
 $= 1 * 2^{120}$

Today: Machine-Level Programming: Basics

- History of Intel processors and architectures
- Instruction Set Architecture (ISA)

Major shift:
Data representation (Ch. 2)
to
Machine instructions and assembly programming (Ch. 3)

Intel x86 Processors

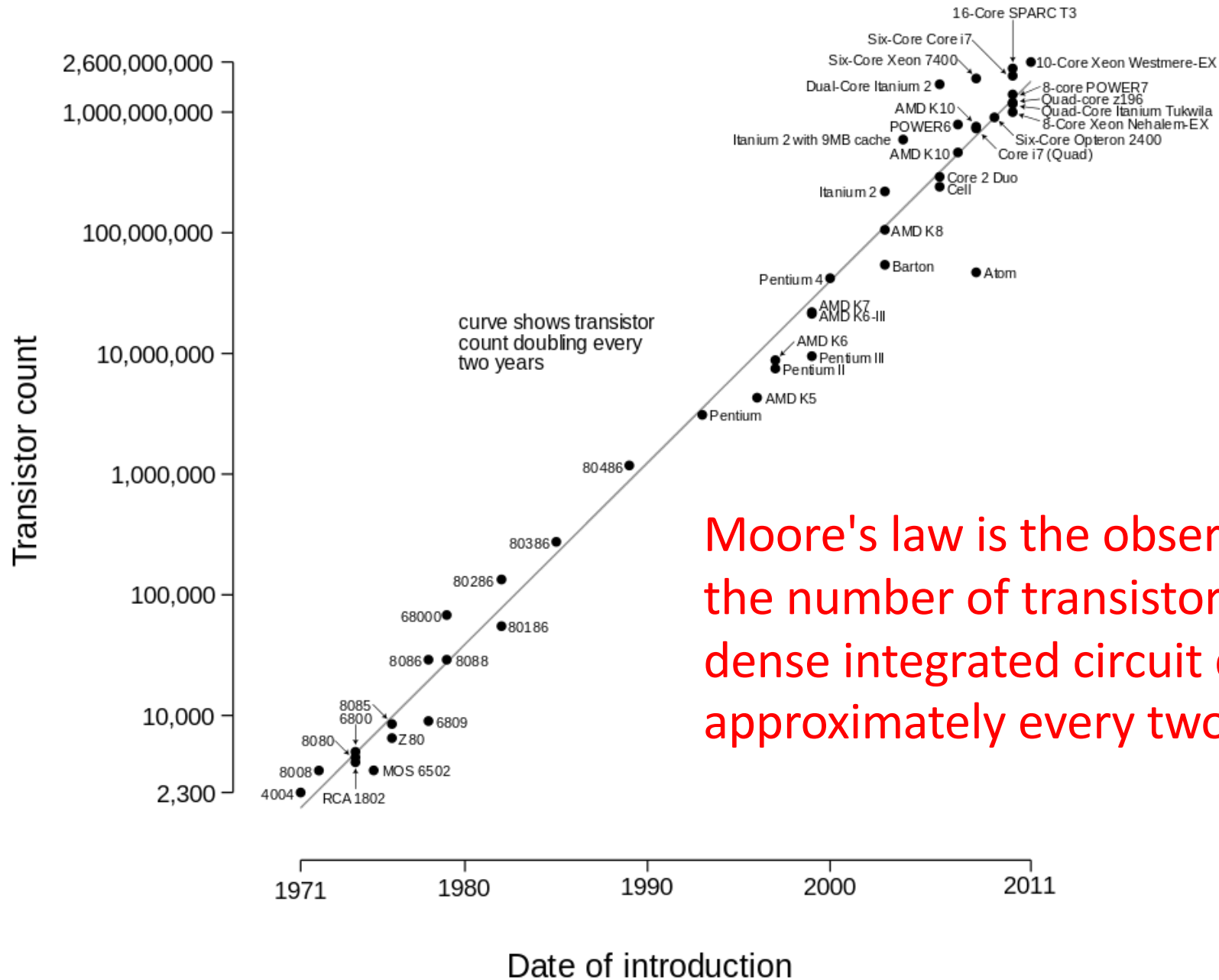
- Dominate laptop/desktop/server market (for now)
- Evolutionary design
 - Backwards compatible up until 8086, introduced in 1978
 - Added more features as time goes on
- Complex instruction set computer (CISC)
 - Many different instructions with many different formats
 - But, only small subset encountered with Linux programs
 - Hard to match performance of Reduced Instruction Set Computers (RISC)
 - But, Intel has done just that!
 - In terms of speed. Less so for low power.



Intel x86 Evolution: Milestones

<i>Name</i>	<i>Date</i>	<i>Transistors</i>	<i>MHz</i>
■ 8086	1978	29K	5-10
■ First 16-bit Intel processor. Basis for IBM PC & DOS ■ 1MB address space			
■ 386	1985	275K	16-33
■ First 32 bit Intel processor , referred to as IA32 ■ Added “flat addressing”, capable of running Unix			
■ Pentium 4E	2004	125M	2800-3800
■ First 64-bit Intel x86 processor, referred to as x86-64			
■ Core 2	2006	291M	1060-3500
■ First multi-core Intel processor (core = CPU)			
■ Core i7	2008	731M	1700-3900
■ Four cores			

Microprocessor Transistor Counts 1971-2011 & Moore's Law



Intel x86 Processors, cont.

■ Past Generations

■ 1 st Pentium Pro	1995	600 nm
■ 1 st Pentium III	1999	250 nm
■ 1 st Pentium 4	2000	180 nm
■ 1 st Core 2 Duo	2006	65 nm

■ Recent Generations

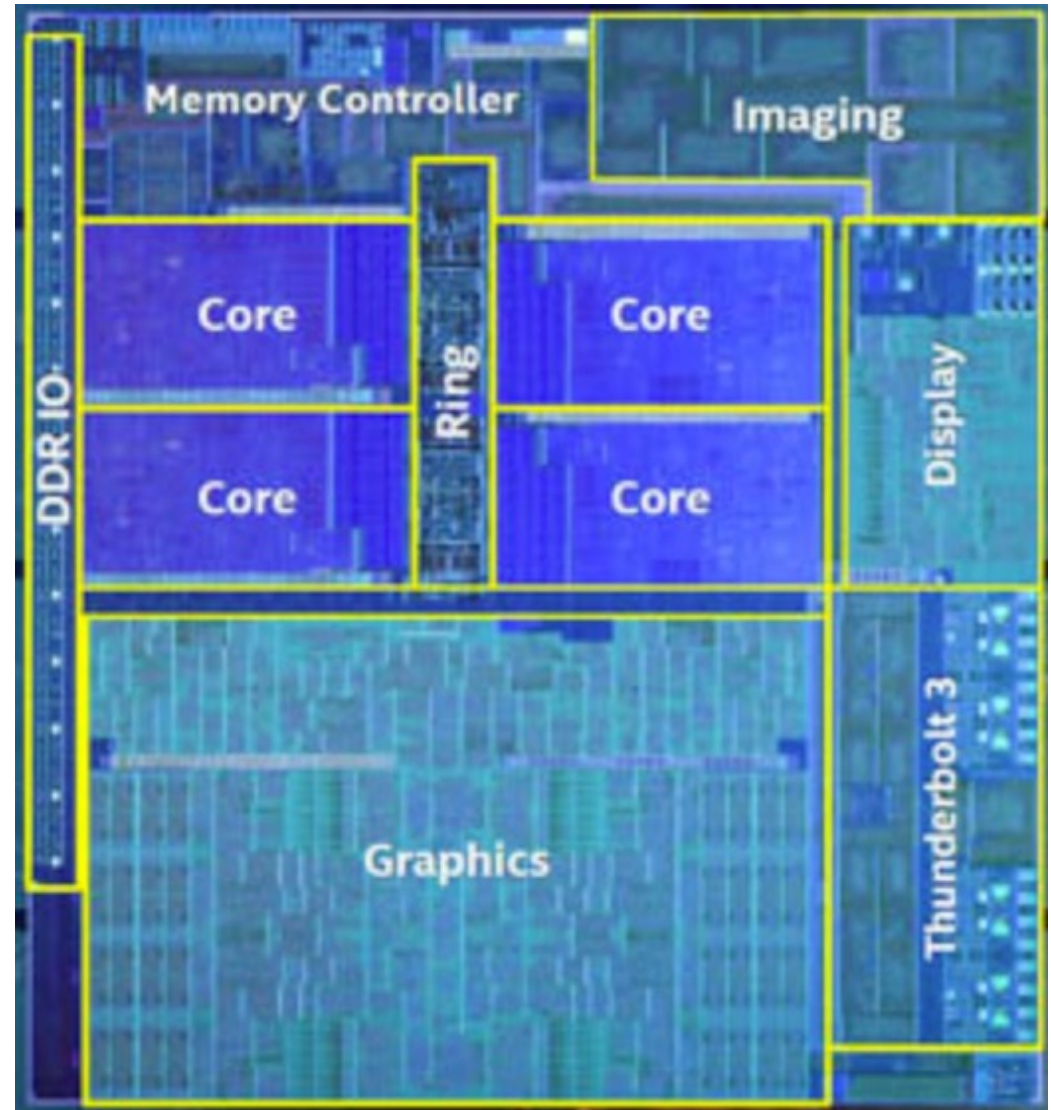
1.	Nehalem	2008	45 nm
2.	Sandy Bridge	2011	32 nm
3.	Ivy Bridge	2012	22 nm
4.	Haswell	2013	22 nm
5.	Broadwell	2014	14 nm
6.	Skylake	2015	14 nm
7.	Kaby Lake	2016/7	14 nm
8.	Coffee Lake	2017	14nm
9.	C.L. Refreshed	2018	14nm
10.	Ice Lake	2019	10nm

**Process technology dimension
= width of narrowest wires
(10 nm $\approx$ 100 atoms wide)**



2019 State of the Art: Ice Lake

- Mobile Device: Core i7
 - 1-2.3 GHz
 - Turbo (3.8-4.1 GHz)
 - 9-28 W
 - Integrated Intel Gen 11 GPU
 - 2-4 CPUs



Our Coverage

■ IA32

- The traditional x86

■ x86-64

- The standard
- `> gcc hello.c`
- `> gcc -m64 hello.c`

■ Presentation

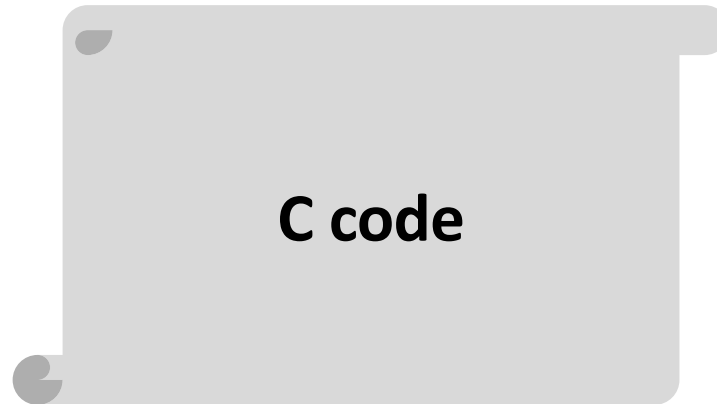
- Book covers x86-64
- Web aside on IA32
- We will only cover x86-64

Today: Machine-Level Programming: Basics

- History of Intel processors and architectures
- Instruction Set Architecture (ISA)

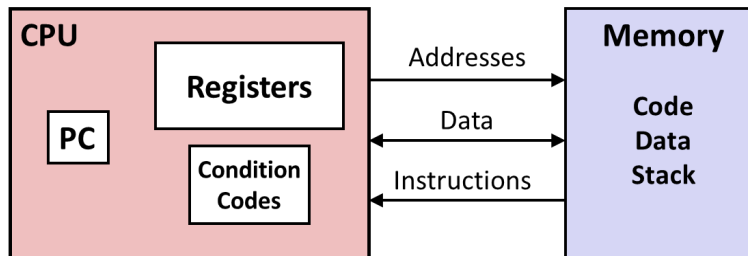
Levels of Abstraction

C programmer



**Computer
scientists love
layers...**

Assembly programmer

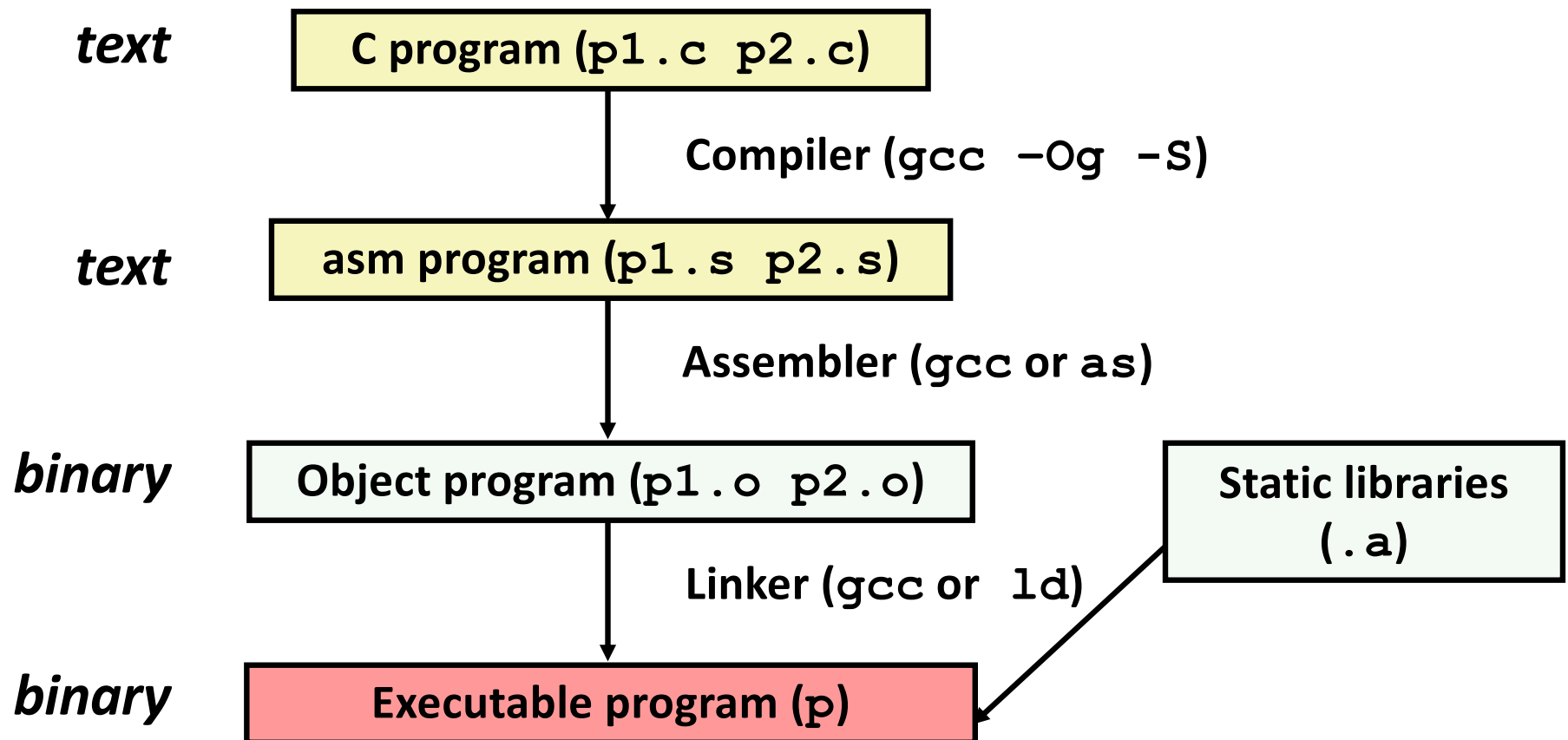


Computer Designer

Caches, clock freq, layout, ...

Turning C into Machine Code

- Code in files `p1.c` `p2.c`
- Compile with command: `gcc -Og p1.c p2.c -o p`
 - Use basic optimizations (`-Og`)
 - Put resulting binary in file `p`



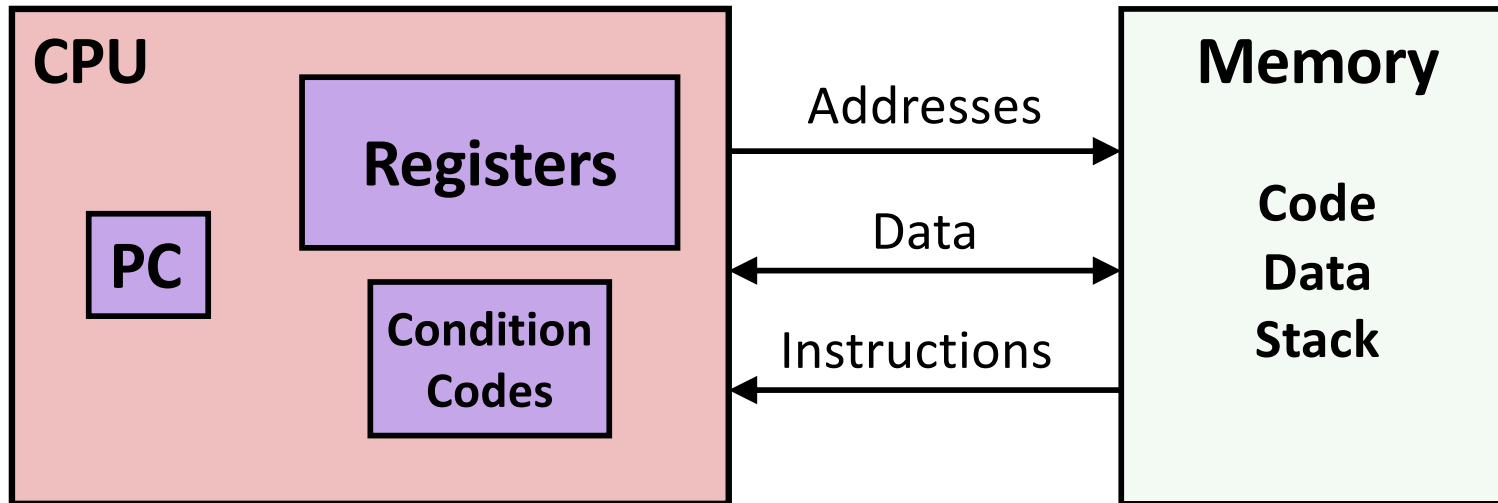
Variables: Source Code → Compiler Internal Representation → Executable

- Source code variable type
 - Determines how many bytes needed to represent variables
 - Determines how bits in bytes should be interpreted
- Compiler's internal representation
 - For each variable, keeps track of variable type and name
 - For each variable, determines # of bytes needed and determines memory locations
- Compiler generates executable that is all memory locations/registers and machine language instructions
 - Names and types are lost. Everything is just memory addresses and registers.
 - Information about size and types is implied by the size of bytes and what versions of instructions are used (different machine instructions for different variable types) by compiler in generating code

Definitions

- **Architecture:** (also ISA: instruction set architecture) The parts of a processor design that one needs to understand for writing assembly/machine code.
 - Examples: instruction set specification, registers
- **Microarchitecture:** Implementation of the architecture
 - Examples: cache sizes and core frequency
- **Code Forms:**
 - **Machine Code:** The byte-level programs that a processor executes
 - **Assembly Code:** A text representation of machine code
- **Example ISAs:**
 - Intel: x86, IA32, Itanium, x86-64
 - ARM: Used in almost all mobile phones

Assembly/Machine Code View



Programmer-Visible State

- **PC: Program counter**

- Address of next instruction
- Called “RIP” (x86-64)

- **Register file**

- Heavily used program data

- **Condition codes**

- Store status information about most recent arithmetic or logical operation
- Used for conditional branching

- **Memory**

- Byte addressable array
- Code and user data
- Stack to support procedures

Assembly Characteristics: Data Types

- “Integer” data of 1, 2, 4, or 8 bytes
 - Data values
 - Addresses (untyped pointers)
- Floating point data of 4, 8, or 10 bytes
- Code: Byte sequences encoding series of instructions
- No aggregate types such as arrays or structures
 - Just contiguously allocated bytes in memory

x86-64 Integer Registers

%rax	%eax
%rbx	%ebx
%rcx	%ecx
%rdx	%edx
%rsi	%esi
%rdi	%edi
%rsp	%esp
%rbp	%ebp

%r8	%r8d
%r9	%r9d
%r10	%r10d
%r11	%r11d
%r12	%r12d
%r13	%r13d
%r14	%r14d
%r15	%r15d

- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)
- Not part of memory (or cache)

Some History: IA32 Registers

Origin
(mostly obsolete)

general purpose	%eax	%ax	%ah	%al	<i>accumulate</i>
	%ecx	%cx	%ch	%cl	<i>counter</i>
	%edx	%dx	%dh	%dl	<i>data</i>
	%ebx	%bx	%bh	%bl	<i>base</i>
	%esi	%si			<i>source index</i>
	%edi	%di			<i>destination index</i>
	%esp	%sp			<i>stack pointer</i>
	%ebp	%bp			<i>base pointer</i>
16-bit virtual registers (backwards compatibility)					

Assembly Characteristics: Operations

- Transfer data between memory and register
 - Load data from memory into register
 - Store register data into memory
- Perform arithmetic function on register or memory data
- Transfer control
 - Unconditional jumps to/from procedures
 - Conditional branches

Moving Data (Ch 3.4)

■ Moving Data

`movq Source, Dest`

■ Operand Types

- **Immediate:** Constant integer data
 - Example: `$0x400`, `$-533`
 - Like C constant, but prefixed with ``$'`
 - Encoded with 1, 2, or 4 bytes
- **Register:** One of 16 integer registers
 - Example: `%rax`, `%r13`
 - But `%rsp` reserved for special use
 - Others have special uses for particular instructions
- **Memory:** 8 consecutive bytes of memory at address given by register
 - Simplest example (notice parentheses): `(%rax)`
 - Various other “addressing modes”

`%rax`

`%rcx`

`%rdx`

`%rbx`

`%rsi`

`%rdi`

`%rsp`

`%rbp`

`%rN`

Warning: Intel docs use
`mov Dest, Source`

Moving Data (Ch 3.4)

■ Moving Data

`movq Source, Dest`

■ Operand Types

■ **Immediate**: Constant integer data

- Example: `$0x400`, `$-533`
- Like C constant, but prefixed with ``$'`
- Encoded with 1, 2, or 4 bytes

■ **Register**: One of 16 integer registers

- Example: `%rax`, `%r13`
- But `%rsp` reserved for special use
- Others have special uses for particular instructions

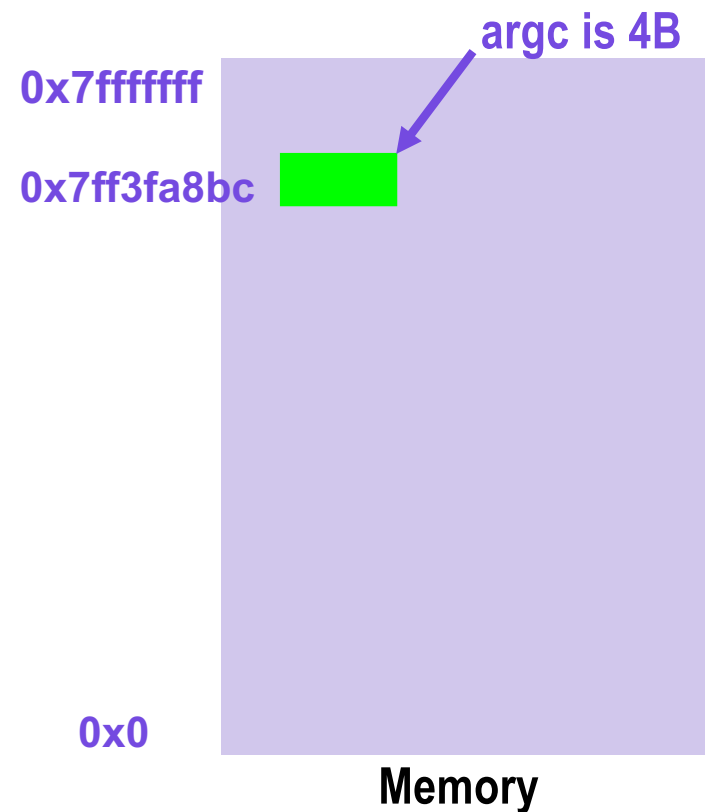
■ **Memory**: 8 consecutive bytes of memory at address given by register

- Simplest example (notice parentheses): `(%rax)`
- Various other “addressing modes”

<code>%rax</code>
<code>%rcx</code>
<code>%rdx</code>
<code>%rbx</code>
<code>%rsi</code>
<code>%rdi</code>
<code>%rsp</code>
<code>%rbp</code>
<code>%rN</code>

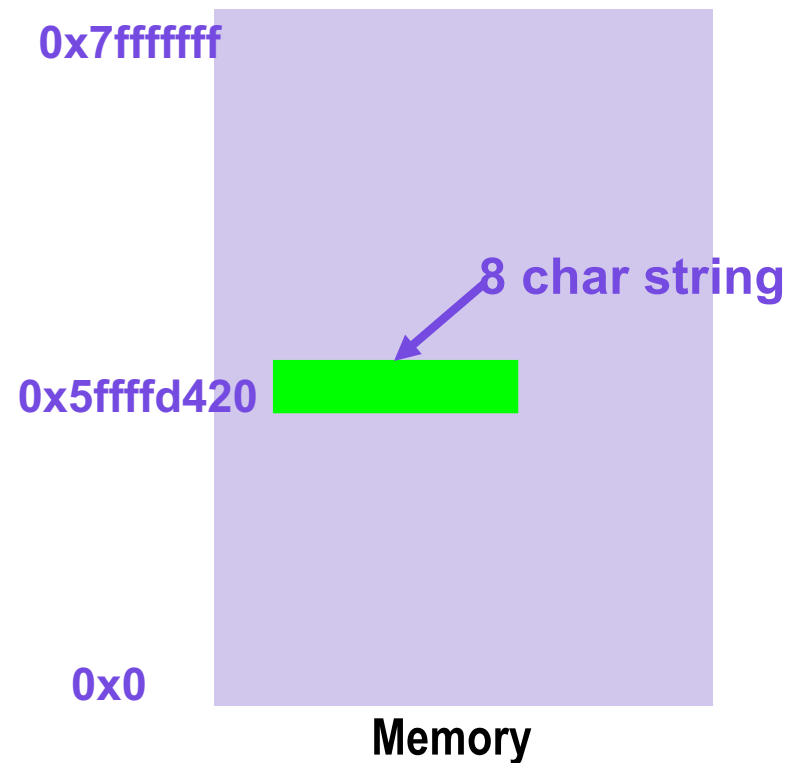
Memory Aside: Data

- Each program has memory associated with it where data is stored
 - # of bits for addresses determines total number of addressable bytes
- That data is just like an array of memory cells
 - Smallest addressable component is a byte
 - Each cell is addressable by its byte location



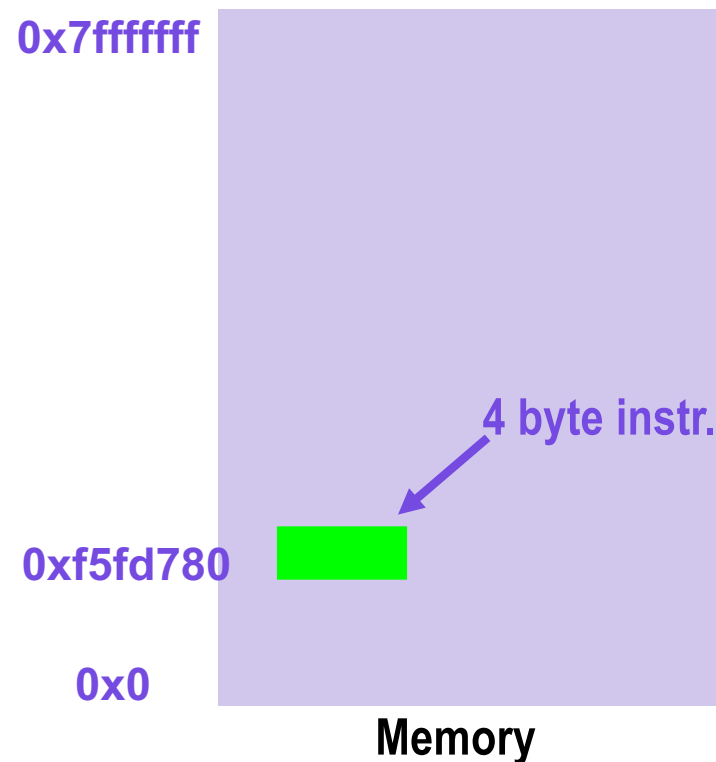
Memory Aside: Data

- Each program has memory associated with it where data is stored
 - # of bits for addresses determines total number of addressable bytes
- That data is just like an array of memory cells
 - Smallest addressable component is a byte
 - Each cell is addressable by its byte location



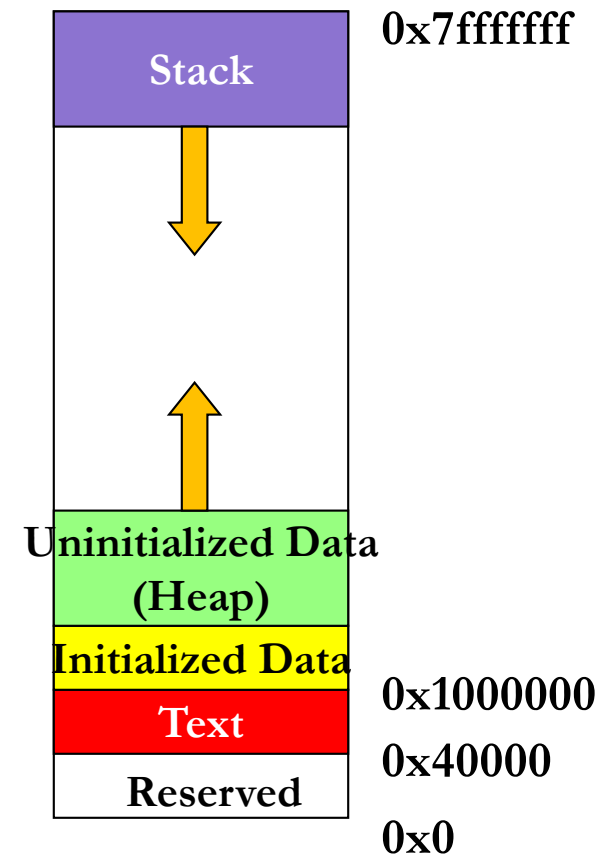
Memory Aside: Instructions

- Each program has memory associated with it where data is stored
 - # of bits for addresses determines total number of addressable bytes
- That data is just like an array of memory cells
 - Smallest addressable component is a byte
 - Each cell is addressable by its byte location
- Instructions are also allocated space in memory



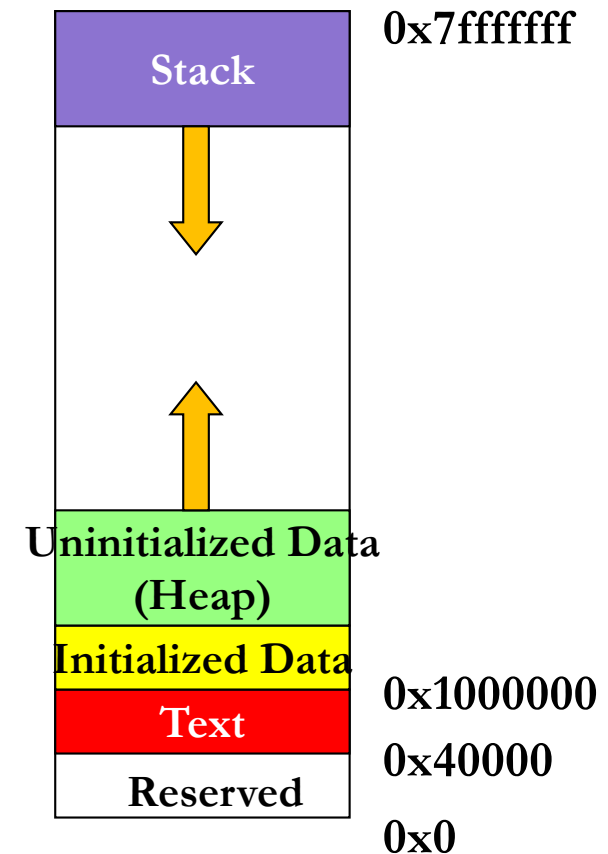
Address Space

- Each process has an address space
- The address space is divided into segments:
 - Text
 - Instructions
 - Initialized Data
 - Globals
 - Uninitialized Data or Heap
 - `new/malloc` allocates space here
 - Stack
 - local variables are given space here



Dynamic Memory Allocation

- Allocate memory in a separate part of address space so it can persist across function calls
 - Heap
- Dynamic memory allocator
 - Software that keeps track of all memory allocated in heap
 - Request a chunk of contiguous memory from allocator
 - `malloc()`
 - Tell allocator when finished with memory so it can reuse that memory
 - `free()`

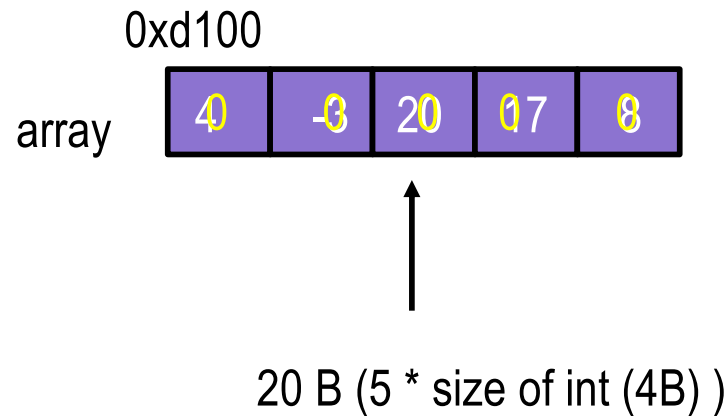


malloc() and free()

- `void *malloc(size_t size);`
 - Specify # of bytes desired as argument
 - Returns address of first byte in contiguous set of bytes allocated
- `void free(void *ptr);`
 - Specify address of first byte of data returned by previous call to malloc
- `void *`
 - Generic pointer since malloc doesn't know the data type the memory is going to be used to store

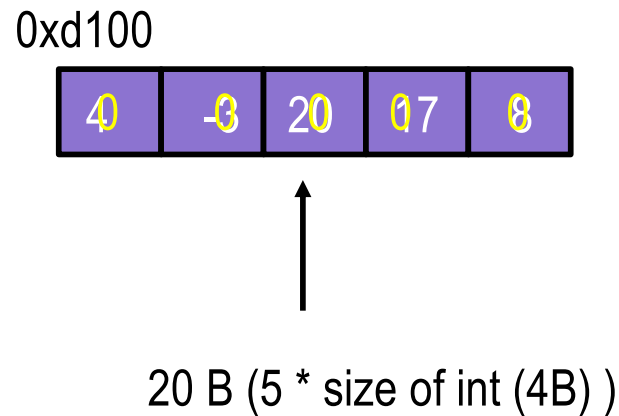
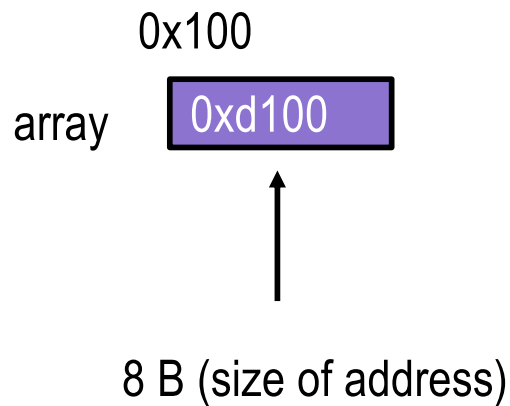
Integer Array example (Static)

```
int array[5];  
  
// Memory contains garbage values,  
// so need to initialize  
for(int i = 0; i < 5; i++){  
    array[i] = 0;  
}
```



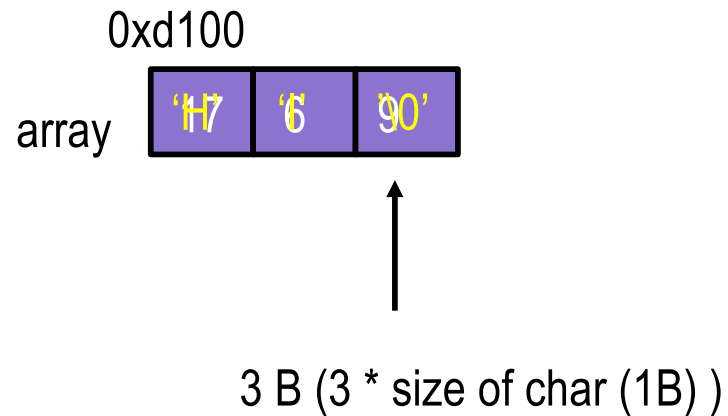
Integer Array example

```
int *array;  
array = (int*)malloc(5 * sizeof(int));  
  
// Memory contains garbage values,  
// so need to initialize  
for(int i = 0; i < 5; i++){  
    array[i] = 0;  
}  
  
free(array);
```



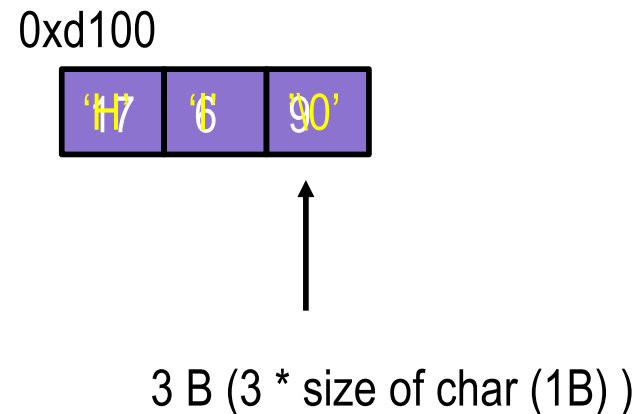
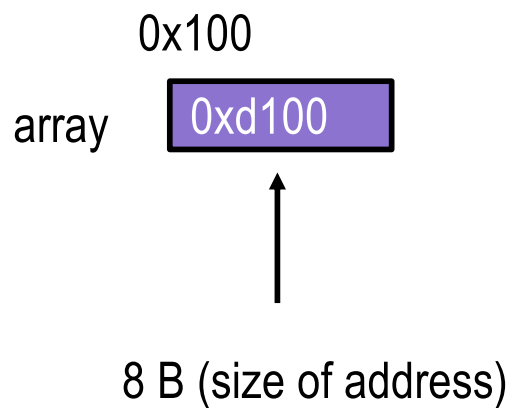
char Array example (Static)

```
char array[3] = "Hi";
```



char Array example

```
char *array;  
array = (char*)malloc( (1+strlen("Hi")) * sizeof(char));  
  
strcpy(array, "Hi");  
  
free(array);
```



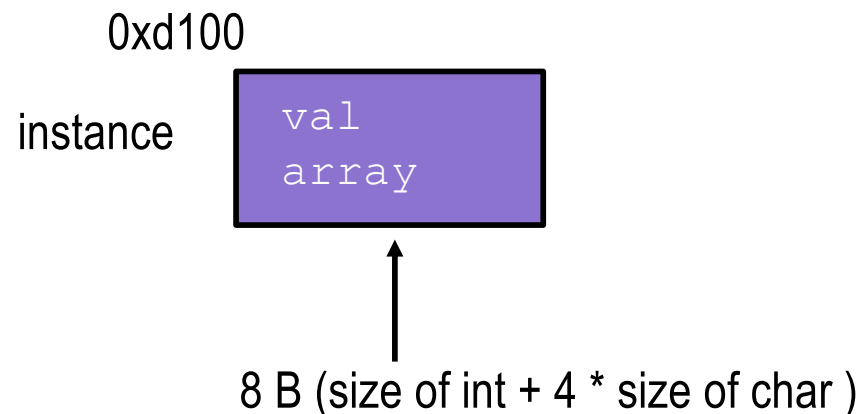
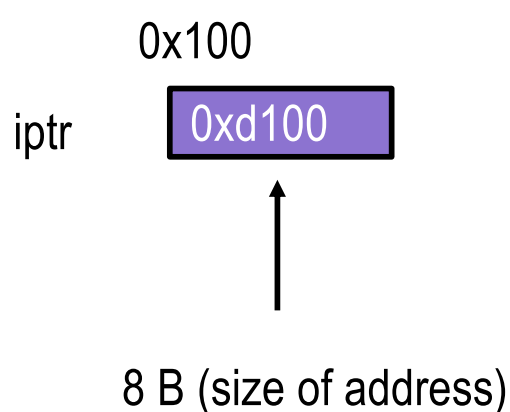
struct example (Static)

```
struct item {  
    int val;  
    char array[4];  
};
```

```
struct item instance;  
struct item *iptr;  
iptr = &instance;
```

```
iptr->val = 10;  
iptr->array[3] = 'a';
```

```
// instance.val = 10;  
// instance.array[3] = 'a';
```



struct example

```
struct item {  
    int val;  
    char array[4];  
};
```

```
struct item *iptr;  
iptr = (struct item*)malloc(sizeof(struct item));
```

```
iptr->val = 10;  
iptr->array[3] = 'a';
```

```
free(iptr);
```

