Floating Representation and Machine Programming

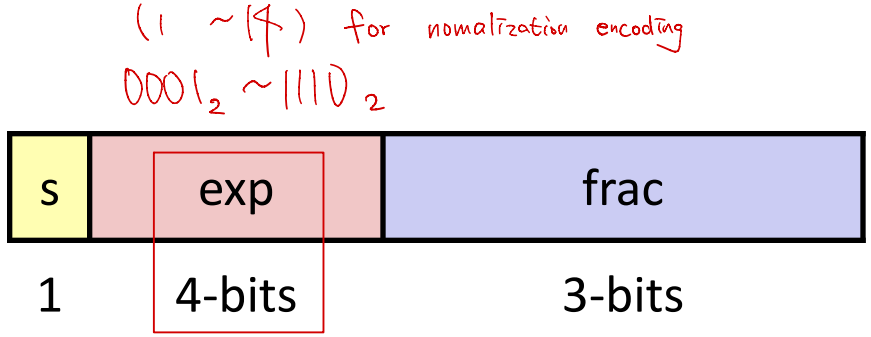
Prof. Hyuk-Yoon Kwon
http://bigdata.seoultech.ac.kr

Most parts are based on slides written by Brayant and O'Hallaon, CMU
(http://csapp.cs.cmu.edu/3e/instructors.html)

Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- **Example and properties**
- Rounding, addition, multiplication
- Floating point in C
- Summary

Tiny Floating Point Example



8-bit Floating Point Representation

- the sign bit is in the most significant bit
- the next four bits are the exponent, with a bias of 7 $= 2^{4-1}-1$
- the last three bits are the **`frac`**

Same general form as IEEE Format

- normalized, denormalized
- representation of 0, NaN, infinity

Dynamic Range (Positive Only)

$$v = (-1)^s M 2^E$$

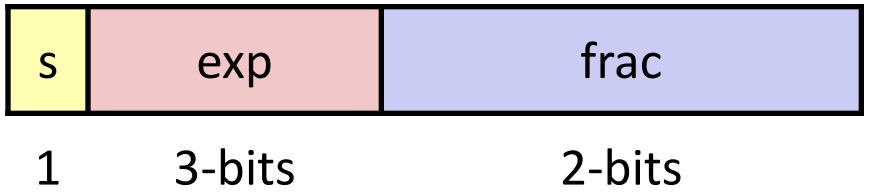
n: E = Exp – Bias

d: E = 1 – Bias

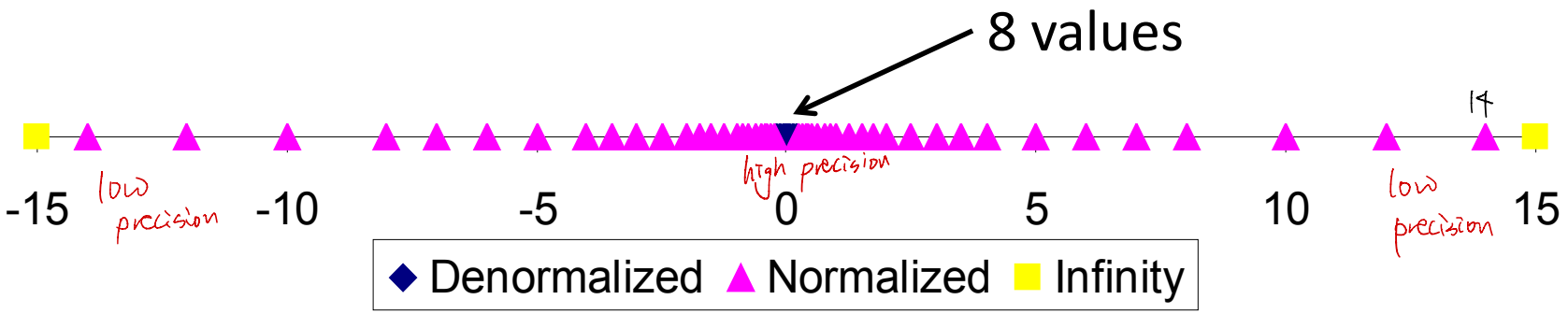
	s	exp	frac	E	Value	
Denormalized numbers	0	0000	000	-6	0	
	0	0000	001	-6	1/8*1/64 = 1/512	closest to zero
	0	0000	010	-6		
	…				…	
	0	0000	110	-6	Q1	
	0	0000	111	-6		largest denorm
Normalized numbers	0	0001	000	-6	8/8*1/64 = 8/512	smallest norm
	0	0001	001	-6		
	…					
	0	0110	110	-1		
	0	0110	111	-1		closest to 1 below
	0	0111	000	0	8/8*1 = 1	
	0	0111	001	0		closest to 1 above
	0	0111	010	0		
	…					
	0	1110	110	7		
	0	1110	111	7	Q2	largest norm
	0	1111	000	n/a	inf	

Distribution of Values

- **6-bit IEEE-like format**
  - e = 3 exponent bits
  - f = 2 fraction bits
  - Bias is $2^{3-1}-1 = 3$



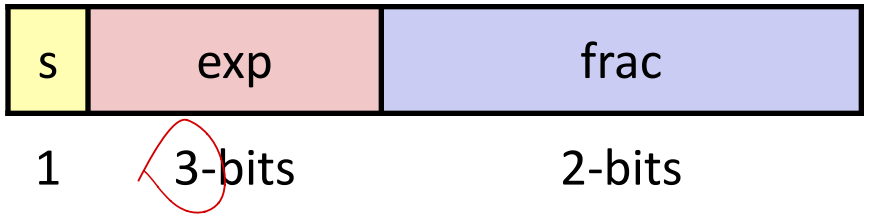
- **Notice how the distribution gets denser toward zero.**

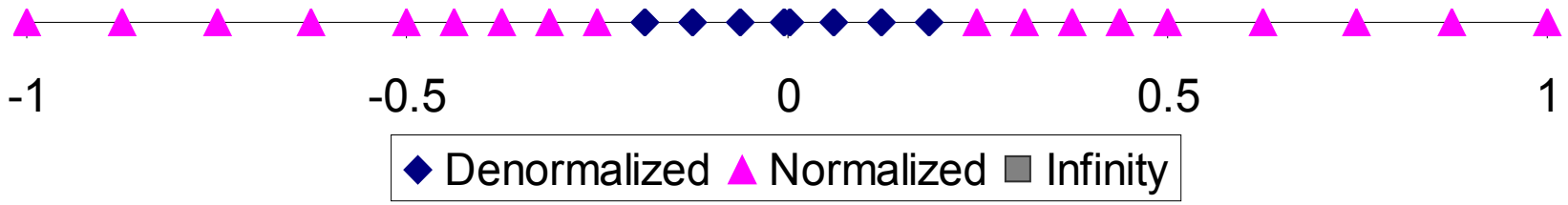


Distribution of Values (close-up view)

6-bit IEEE-like format

- e = 3 exponent bits
- f = 2 fraction bits
- Bias is 3 $(2^{3-1}-1)$





Special Properties of the IEEE Encoding

FP Zero Same as Integer Zero

- All bits = 0

Can (Almost) Use Unsigned Integer Comparison

- Must first compare sign bits
- Must consider −0 = 0
- NaNs problematic
  - Will be greater than any other values
  - What should comparison yield? Not determined.
- Otherwise OK
  - Denorm vs. normalized
  - Normalized vs. infinity

Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- **Rounding, addition, multiplication**
- Floating point in C
- Summary

Floating Point Operations: Basic Idea

- $x +_f y = \text{Round}(x + y)$

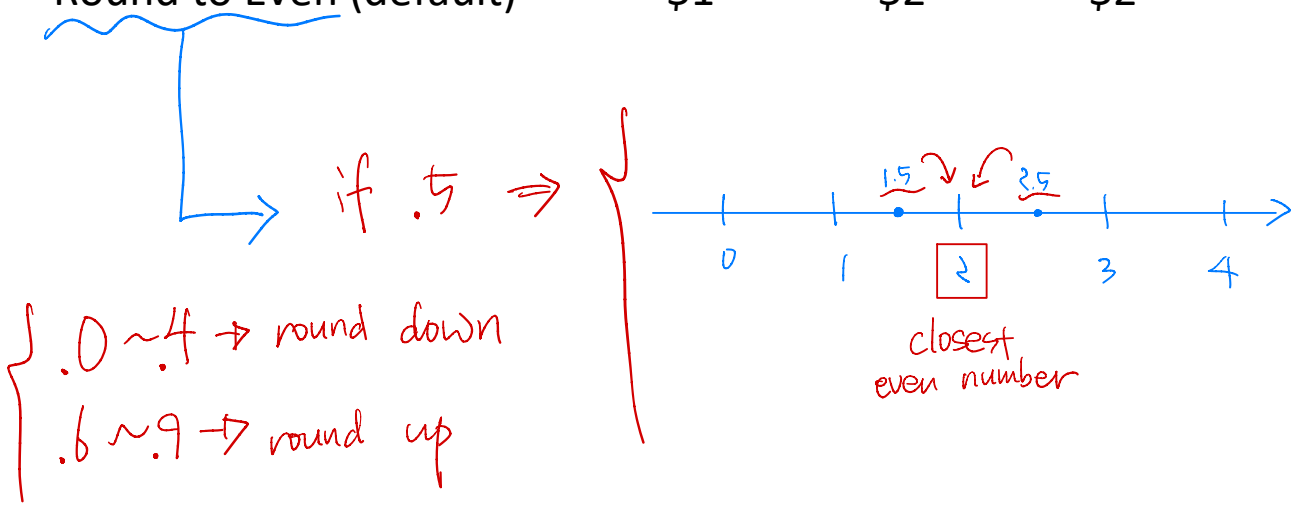
- $x \times_f y = \text{Round}(x \times y)$

- **Basic idea**
  - First compute exact result
  - Make it fit into desired precision
    - Possibly overflow if exponent too large
    - Possibly round to fit into `frac`

Rounding

- **Rounding Modes (illustrate with $ rounding)**

	$1.40	$1.60	$1.50	$2.50	–$1.50
Towards zero (truncate)	$1	$1	$1	$2	**Q1** −1
Round down (−∞) (smaller)	$1	$1	$1	$2	**Q2** −2
Round up (+∞) (bigger)	$2	$2	$2	$3	**Q3** −1
Round to Even (default)	$1	$2	$2	$2	**Q4** −2



Closer Look at Round-To-Even

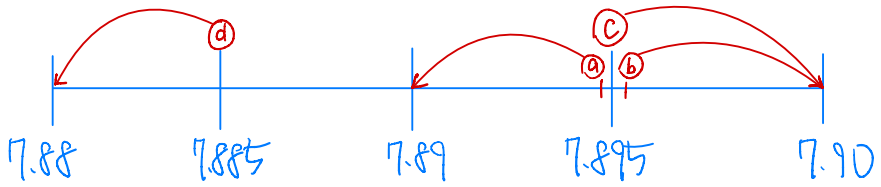
Default Rounding Mode

- Hard to get any other kind without dropping into assembly
- All others are statistically biased
  - Sum of set of positive numbers will consistently be over- or under- estimated

Applying to Other Decimal Places / Bit Positions

- When exactly halfway between two possible values
  - Round so that least significant digit is even
- E.g., round to nearest hundredth 1/100

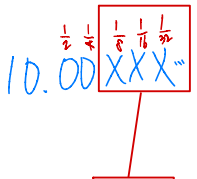
(a)	7.8949999	7.89	(Less than half way)
(b)	7.8950001	7.90	(Greater than half way)
(c)	7.8950000	7.90	(Half way—round up)
(d)	7.8850000	7.88	(Half way—round down)



Rounding Binary Numbers

Binary Fractional Numbers

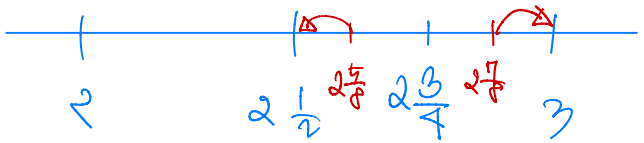
- "Even" when least significant bit is 0
- "Half way" when bits to right of rounding position = $100\ldots_2$



Examples

- Round to nearest 1/4 (2 bits right of binary point)

Value	Binary	Rounded	Action	Rounded Value
2 3/32	10.00011_2	10.00_2	(<1/2—down)	2
2 3/16	10.00110_2	10.01_2	(>1/2—up)	2 1/4
2 7/8	10.11100_2	11.00_2	(1/2—up)	3
2 5/8	10.10100_2	10.10_2	(1/2—down)	2 1/2



FP Multiplication

- $(-1)^{s1}\ M1\ \ 2^{E1}\ \ \times\ \ (-1)^{s2}\ M2\ \ 2^{E2}$

s_1 \ s_2	even (0)	odd (1)
even (0)	even (0)	odd (1)
odd (1)	odd (1)	even (0)

- **Exact Result:** $(-1)^{s}\ M\ \ 2^{E}$
  - Sign *s*: $s1 \wedge s2$ (XOR)
  - Significand *M*: $M1 \times M2$
  - Exponent *E*: $E1 + E2$

$(1.0101_2 \times 2^6) \times (1.11_2 \times 2^3)$

147

- **Fixing**
  - If $M \geq 2$, shift *M* right, increment *E*
  - Round *M* to fit `frac` precision
  - If *E* out of range, overflow — drop upper bits
    - $E > bias$ or $E < -bias+1$
- **Implementation**
  - Biggest chore is multiplying significands

```
 1,0101
    1,11
 --------
    10101
   10101
  10101
 --------
 1,0010011
```

$1 \times 10.0100111_2 \times 2^9$

$= 1 \times 1.0010011_2 \times 2^{10}$

$E = 10_{10}$

$bias = 127_{10}$

$exp = 137_{10} = 10001011_2$

$frac = 0010011_2$

0 10001011 00100110…

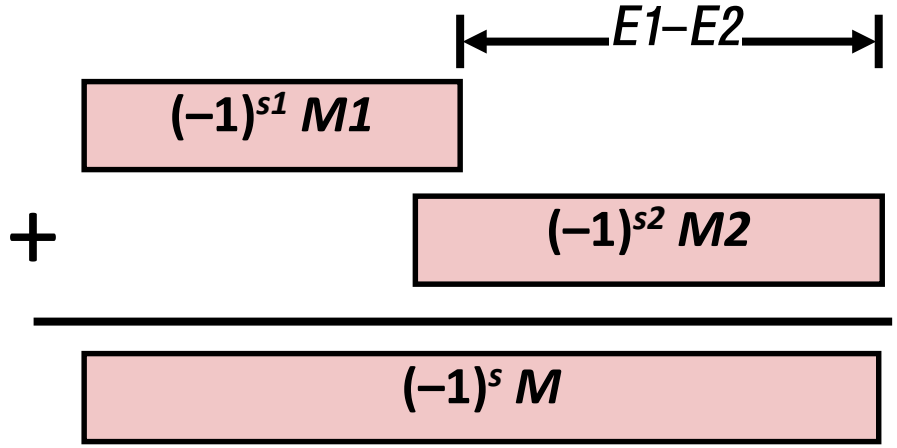
Floating Point Addition

- $(-1)^{s1}\ M1\ \ 2^{E1}\ \ +\ \ (-1)^{s2}\ M2\ \ 2^{E2}$
  - Assume *E1* > *E2*

- **Exact Result:** $(-1)^{s}\ M\ \ 2^{E}$
  - Sign *s*, significand *M*:
    - Result of signed align & add
  - Exponent *E*: *E1*

- **Fixing**
  - If *M* ≥ 2, shift *M* right, increment *E*
  - if *M* < 1, shift *M* left *k* positions, decrement *E* by *k*
  - Round *M* to fit `frac` precision
  - Overflow if *E* out of range

Get binary points lined up



$0.1101_2 \times 2^3 \times 2^2 \times 2^{-7}$

$+\ 0.1111_2 \times 2^5$

$= 0.0011101_2 \times 2^5 + 0.1111_2 \times 2^5$

$= (-1)^0 \times 1.0010001 \times 2^5$ E=5 bias=127 exp=132

0 10000100 00100010000000000000000

Mathematical Properties of FP Add

Compare to Mathematical Properties

- Closed under addition? — ***Almost***
  - But may generate infinity or NaN
- Commutative? — ***Yes***
- Associative? — ***No***
  - Overflow and inexactness of rounding
  - `(3.14+1e10)-1e10 = 0, 3.14+(1e10-1e10) = 3.14`

Monotonicity — ***Almost***

- $a \geq b \Rightarrow a+c \geq b+c$?
  - Except for infinities & NaNs

Mathematical Properties of FP Mult

Compare to Mathematical Properties

- Closed under multiplication? — ***Almost***
  - But may generate infinity or NaN
- Multiplication Commutative? — ***Yes***
- Multiplication is Associative? — ***No***
  - Possibility of overflow, inexactness of rounding
  - Ex: `(1e20*1e20)*1e-20= inf, 1e20*(1e20*1e-20)= 1e20`
- Multiplication distributes over addition? — ***No***
  - Possibility of overflow, inexactness of rounding
  - `1e20*(1e20-1e20)= 0.0, 1e20*1e20 – 1e20*1e20 = NaN`

Monotonicity — ***Almost***

- $a \geq b \;\&\; c \geq 0 \Rightarrow a * c \geq b * c$?
  - Except for infinities & NaNs

Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- **Floating point in C**
- Summary

Floating Point in C

C Guarantees Two Levels

- **`float`** single precision
- **`double`** double precision

Conversions/Casting

- Casting between **`int`**, **`float`**, and **`double`** changes bit representation
- **`double`**/**`float`** → **`int`**
  - Truncates fractional part
  - Like rounding toward zero
  - Not defined when out of range or NaN: Generally sets to TMin
- **`int`** → **`double`**
  - Exact conversion, as long as **`int`** has ≤ 53 bit word size
- **`int`** → **`float`**
  - Will round according to rounding mode

$2^{23+1}+1$ 向下舍入 rounding

Practice: Floating Point Puzzles

- **For each of the following C expressions, either:**
  - Argue that it is true for all argument values
  - Explain why not true

```
int x = …;
float f = …;
double d = …;
```

Assume neither
`d` nor `f` is NaN

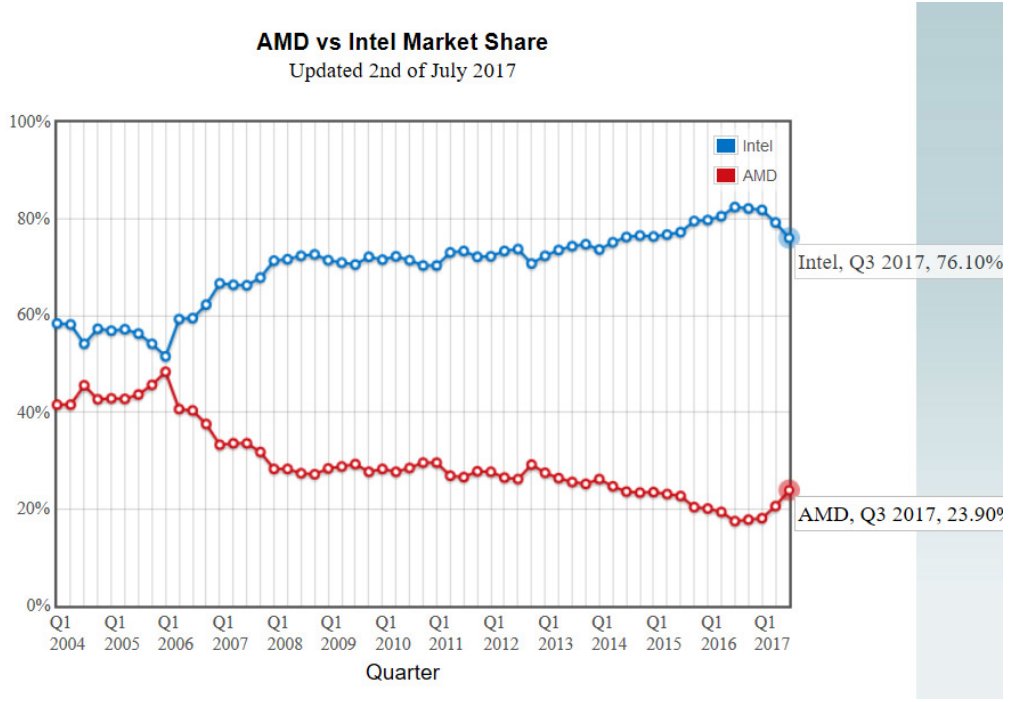
- `x == (int)(float) x` false (loss)
- `x == (int)(double) x` true
- `f == (float)(double) f` true
- `d == (double)(float) d` false (loss)
- `f == -(-f);` true (only affects sign bit!)
- `(d+f)-d == f` false (can overflow)

Today: Machine Programming I: Basics

- **History of Intel processors and architectures**
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations

Intel x86 Processors

- **Dominate laptop/desktop/server market**

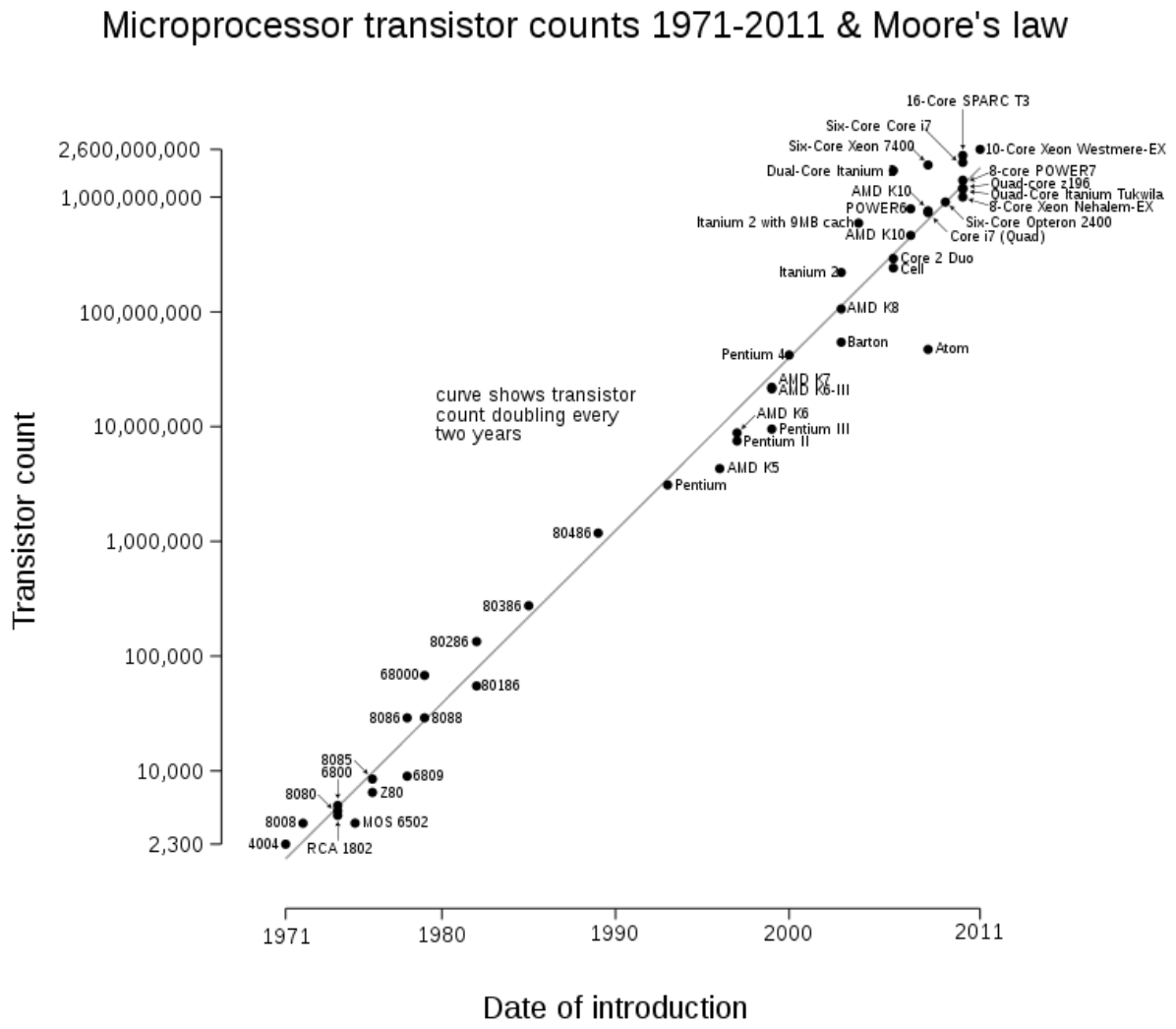


- **Evolutionary design**
  - Backwards compatible up until 8086, introduced in 1978
  - Added more features as time goes on

- **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

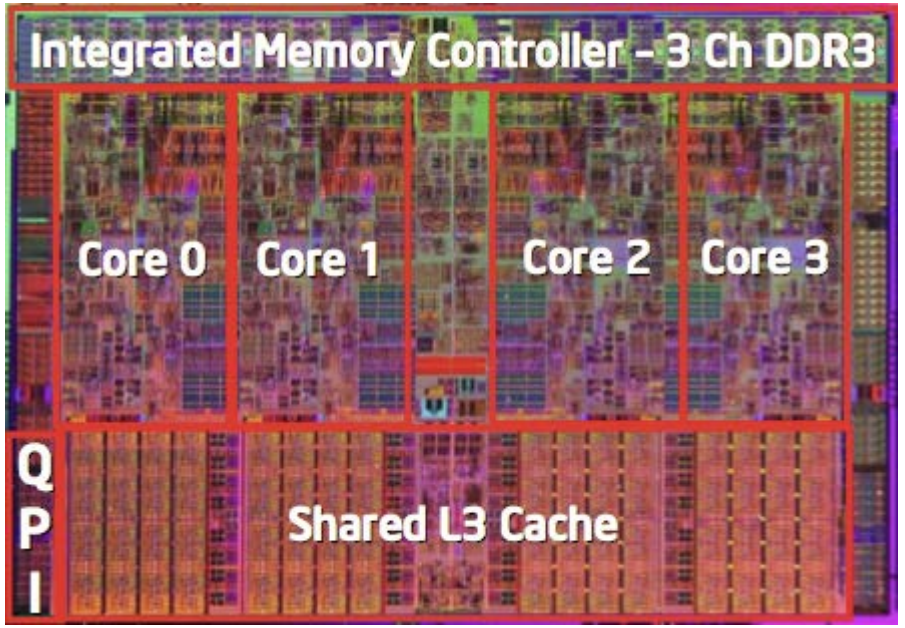
Name	Date	Trans
8086	**1978**	**29K**
• First 16-bit Intel processor. Basis for IBM I		
• 1MB address space		
386	**1985**	**275K**
• First 32 bit Intel processor , referred to as		
Pentium 4E	**2004**	**125M**
• First 64-bit Intel x86 processor, referred tc		
Core 2	**2006**	**291M**
• First multi-core Intel processor		
Core i7	**2008**	**731M**
• Four cores		



Intel x86 Processors, cont.

Machine Evolution

386	1985	0.3M
Pentium	1993	3.1M
Pentium/MMX	1997	4.5M
PentiumPro	1995	6.5M
Pentium III	1999	8.2M
Pentium 4	2001	42M
Core 2 Duo	2006	291M
Core i7	2008	731M



Added Features

- Instructions to support multimedia operations
- Instructions to enable more efficient conditional operations
- Transition from 32 bits to 64 bits
- More cores parallel programming

2015 State of the Art

Core i7 Broadwell 2015

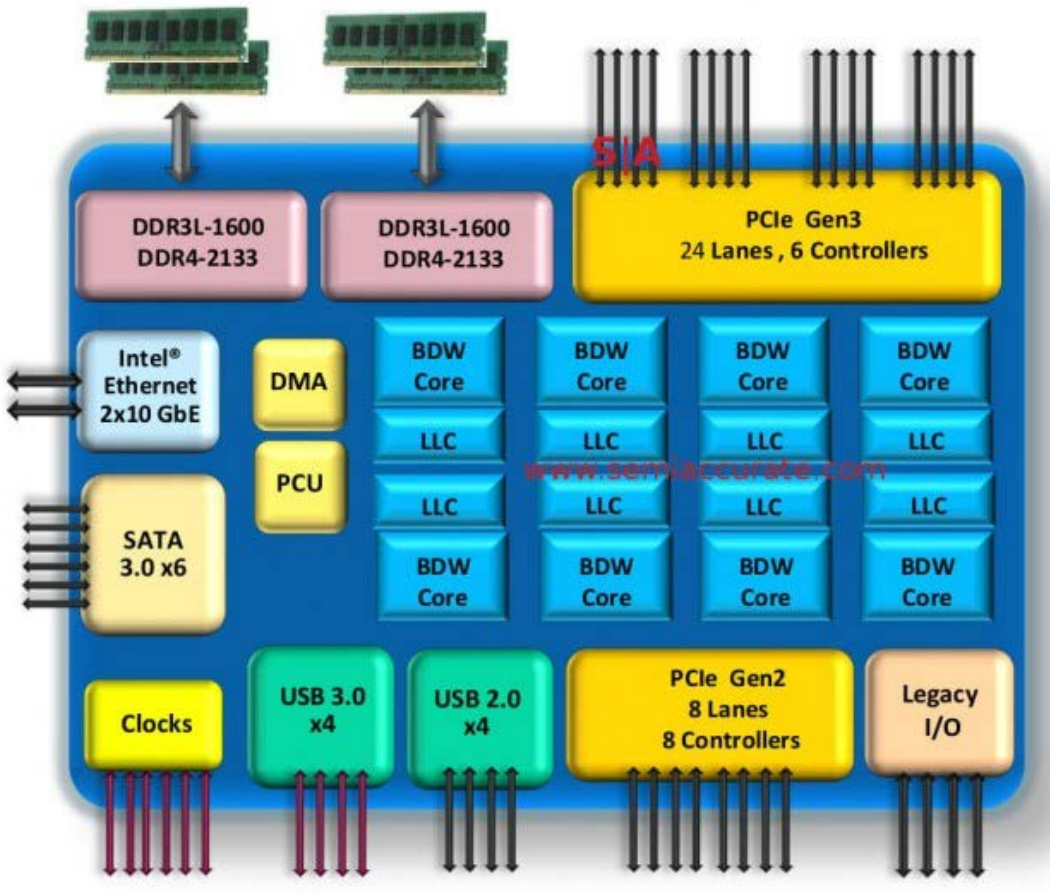
- Skylake, Aug 2015
- Kabylake, Jan. 2017

Desktop Model

- 4 cores
- Integrated graphics
- 3.3-3.8 GHz

Server Model

- 8 cores
- Integrated I/O
- 2-2.6 GHz



x86 Clones: Advanced Micro Devices (AMD)

Historically

- AMD has followed just behind Intel
- A little bit slower, a lot cheaper

Then

- Recruited top circuit designers from Digital Equipment Corp. and other downward trending companies
- Built Opteron: tough competitor to Pentium 4
- Developed x86-64, their own extension to 64 bits

Recent Years

- Intel got its act together
  - Leads the world in semiconductor technology
- AMD has fallen behind
  - Relies on external semiconductor manufacturer

Intel's 64-Bit History

- **2001: Intel Attempts Radical Shift from IA32 to IA64**
  - Totally different architecture (Itanium)
  - Executes IA32 code only as legacy backward compatibility
  - Performance disappointing
- **2003: AMD Steps in with Evolutionary Solution**
  - x86-64 (now called "AMD64")
- **Intel Felt Obligated to Focus on IA64**
  - Hard to admit mistake or that AMD is better
- **2004: Intel Announces EM64T extension to IA32**
  - Extended Memory 64-bit Technology
  - Almost identical to x86-64!
- **All but low-end x86 processors support x86-64**
  - But, lots of code still runs in 32-bit mode

Our Coverage

- **x86-64**
  - The standard
  - `linux> gcc hello.c`
  - `linux> gcc -m64 hello.c` 64-bits
  - `linux> gcc -m32 hello.c` 32-bits
    - For generating executable for x86 architecture

```
osboxes@osboxes:~/work/Lab/Cachelab/grade/cachelab$ gcc -Q --help=target
The following options are target specific:
  -m128bit-long-double                  [disabled]
  -m16                                  [disabled]
  -m32                                  [disabled]
  -m3dnow                               [disabled]
  -m3dnowa                              [disabled]
  -m64                                  [enabled]
  -m80387                               [enabled]
  -m8bit-idiv                           [disabled]
  -m96bit-long-double                   [enabled]
  -mabi=                                sysv
  -mabm                                 [disabled]
  -maccumulate-outgoing-args            [disabled]
  -maddress-mode=                       short
  -madx                                 [disabled]
  -maes                                 [disabled]
```

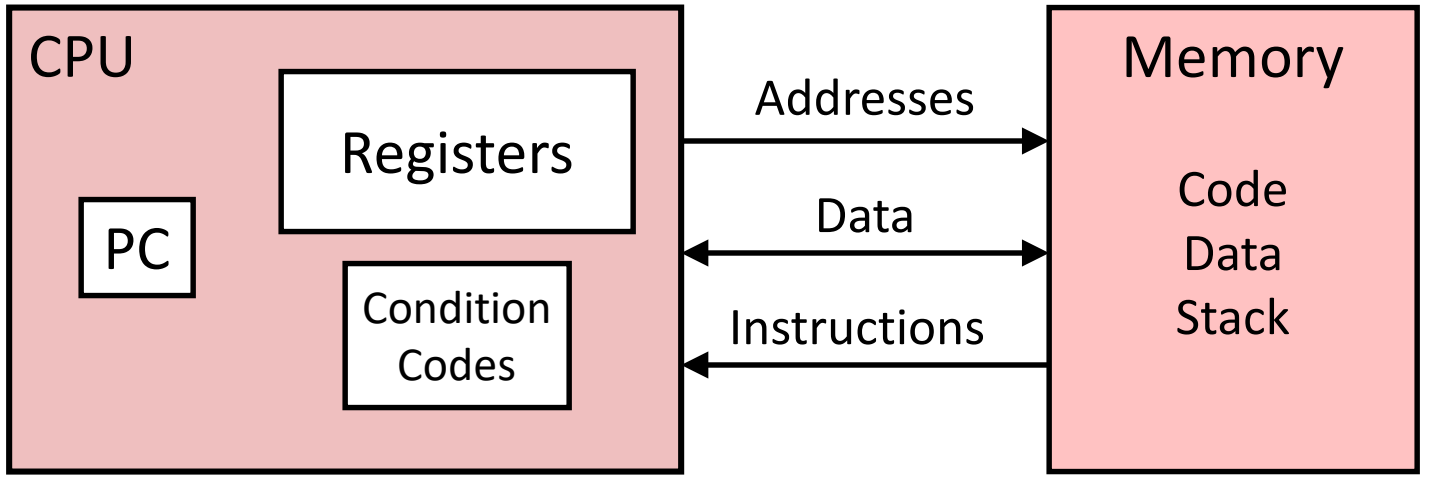
Today: Machine Programming I: Basics

- History of Intel processors and architectures
- **C, assembly, machine code**
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations

Definitions

- **Architecture: (also ISA: instruction set architecture) The parts of a processor design that one needs to understand or write 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



CPU → Memory
(New result)

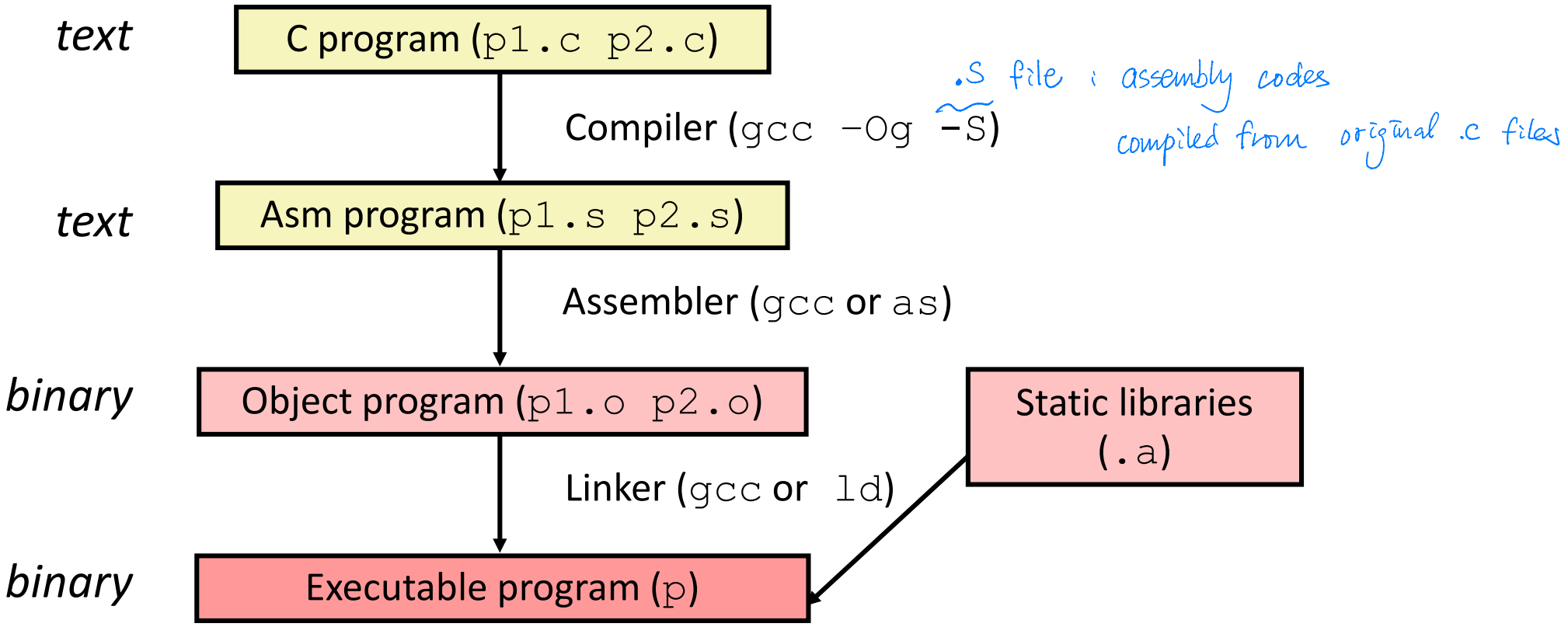
Programmer-Visible State

- **PC: Program counter**
  - Address of next instruction
  - Called "RIP" (x86-64) Register Instruction Pointer.
- **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

Turning C into Object Code

- Code in files **`p1.c p2.c`**
- Compile with command: **`gcc -Og p1.c p2.c -o p`**
  - Use basic optimizations (**`-Og`**) [New to recent versions of GCC]
  - Put resulting binary in file **`p`**

optional



.S file : assembly codes compiled from original .c files

Practice: Compiling Into Assembly

C Code (sum.c)

```
long plus(long x, long y);

void sumstore(long x, long y,
              long *dest)
{
    long t = plus(x, y);
    *dest = t;
}
```

Generated x86-64 Assembly

```
sumstore:
    pushq   %rbx
    movq    %rdx, %rbx
    call    plus
    movq    %rax, (%rbx)
    popq    %rbx
    ret
```

Obtain with command

```
gcc -Og -S sum.c
```

Produces file `sum.s`

Warning: Will get very different results depending on the machine architectures (Andrew Linux, Mac OS-X, …) due to different versions of gcc and different compiler settings.

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

Assembly Characteristics: Operations

- **Perform arithmetic function on register or memory data**

- **Transfer data between memory and register**
  - Load data from memory into register
  - Store register data into memory

- **Transfer control**
  - Unconditional jumps to/from procedures
  - Conditional branches

Object Code

Code for `sumstore`

```
0x0400595:
     0x53
     0x48
     0x89
     0xd3
     0xe8
     0xf2
     0xff
     0xff
     0xff
     0xff
     0x48
     0x89
     0x03
     0x5b
     0xc3
```

- Total of 14 bytes
- Each instruction 1, 3, or 5 bytes
- Starts at address `0x0400595`

contain all necessary information without loading.
be able to execute in any environments!

no dependency on other libraries!

Static linked executable size
∨
Dynamic linked executable size
(not contain the library itself, just information of it)

Static Link → Linker combines the library into the executable during the compilation.
including library.

Dynamic Link → Linker writes information for the library in the executable. Program runs then load the library dynamically

Assembler

- Translates `.s` into `.o`
- Binary encoding of each instruction
- Nearly-complete image of executable code
- Missing linkages between code in different files

Linker

- Resolves references between files
- Combines with static run-time libraries
  - E.g., code for **`malloc`**, **`printf`**
- Some libraries are *dynamically linked*
  - Linking occurs when program begins execution

Machine Instruction Example

```
*dest = t;
```

parenthesis = * in assembly.

```
movq %rax, (%rbx)
```

```
0x40059e:  48 89 03
```

C Code

- Store value **t** where designated by **dest**

Assembly

- Move 8-byte value to memory
  - Quad words in x86-64 parlance
- Operands:

t:	Register	**%rax**
dest:	Register	**%rbx**
***dest:**	Memory	**M[%rbx]**

Object Code

- 3-byte instruction
- Stored at address **0x40059e**

Disassembling Object Code

Disassembled

```
0000000000400595 <sumstore>:
  400595:  53                 |  push   %rbx
  400596:  48 89 d3           |  mov    %rdx,%rbx
  400599:  e8 f2 ff ff ff     |  callq  400590 <plus>
  40059e:  48 89 03           |  mov    %rax,(%rbx)
  4005a1:  5b                 |  pop    %rbx
  4005a2:  c3                 |  retq
```

- **Disassembler**

  `objdump -d sum`

  - Useful tool for examining object code
  - Analyzes bit pattern of series of instructions
  - Produces approximate assembly code
  - Can be run on either `a.out` (complete executable) or `.o` file

Alternate Disassembly

Object

```
0x0400595:
    0x53
    0x48
    0x89
    0xd3
    0xe8
    0xf2
    0xff
    0xff
    0xff
    0xff
    0x48
    0x89
    0x03
    0x5b
    0xc3
```

Disassembled

```
Dump of assembler code for function sumstore:
 0x0000000000400595 <+0>: push   %rbx
 0x0000000000400596 <+1>: mov    %rdx,%rbx
 0x0000000000400599 <+4>: callq  0x400590 <plus>
 0x000000000040059e <+9>: mov    %rax,(%rbx)
 0x00000000004005a1 <+12>:pop    %rbx
 0x00000000004005a2 <+13>:retq
```

- **Within gdb Debugger**

  `gdb sum`

  `disassemble sumstore`

  - Disassemble procedure

  `x/14xb sumstore`

  - Examine the 14 bytes starting at `sumstore`

What Can be Disassembled?

```
% objdump -d WINWORD.EXE

WINWORD.EXE:    file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:
30001001:
30001003:
30001005:
3000100a:
```

Reverse engineering forbidden by
Microsoft End User License Agreement

- Anything that can be interpreted as executable code
- Disassembler examines bytes and reconstructs assembly source