Machine-Level Programming: Basics & Arithmetic

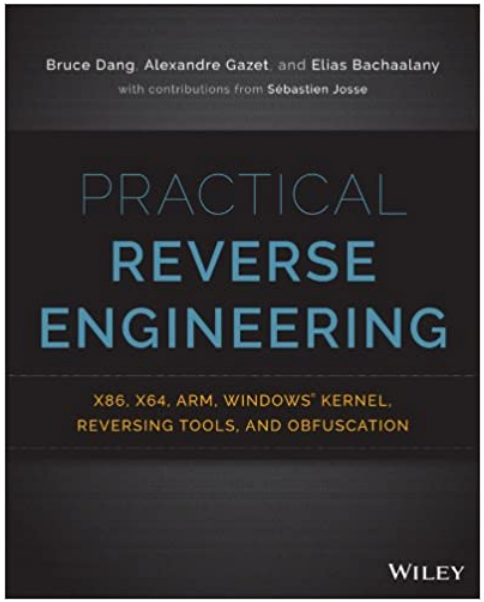
Notes

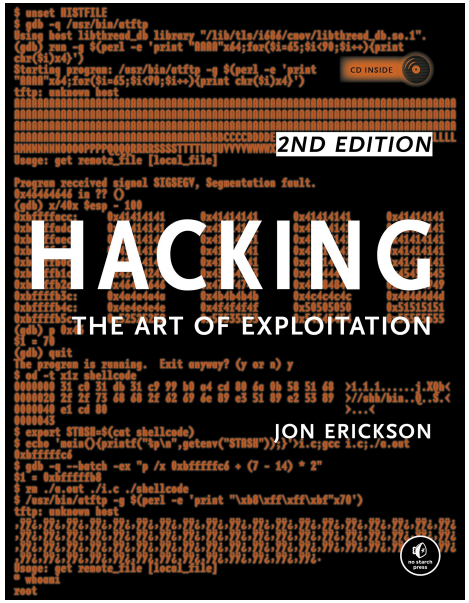
- Lectures will become more and more challenging
- Low level textbook details are “good to know” but not required
- In the next lecture we will have some practice problems

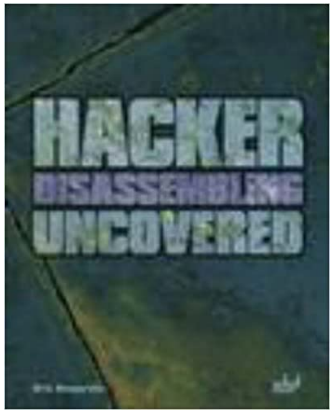
Why Learn Assembly?

Assembly is Important for Security

- System attacks
- Reverse engineering







Today: Machine Programming I: Basics

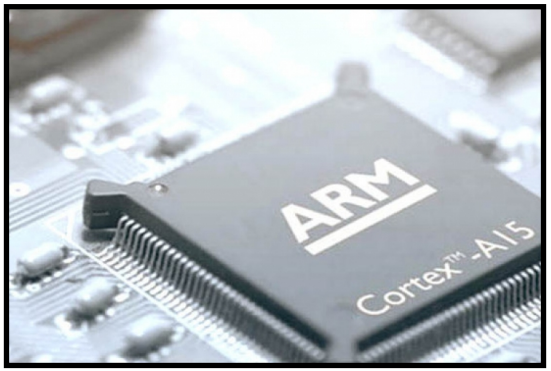
- **History of Intel processors and architectures**
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations

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

Other Processors

- Which CPU company dominates mobile/smartphones?



- Sold 160 billion chips!

Types of Instruction Sets: CISC and RISC

- **Intel CPUs: Complex instruction set computer (CISC)**
  - Complex, specialized instructions
  - Too many of these complex instructions
    - But, only small subset encountered with Linux programs

- **ARM CPUs: Reduced instruction set computer (RISC)**
  - Small number of basic instructions
  - These instructions are composed to create complex functionalities
  - It is hard for CISC to match performance of RISC
  - But, Intel has done just that!
    - In terms of speed. Less so for low power.

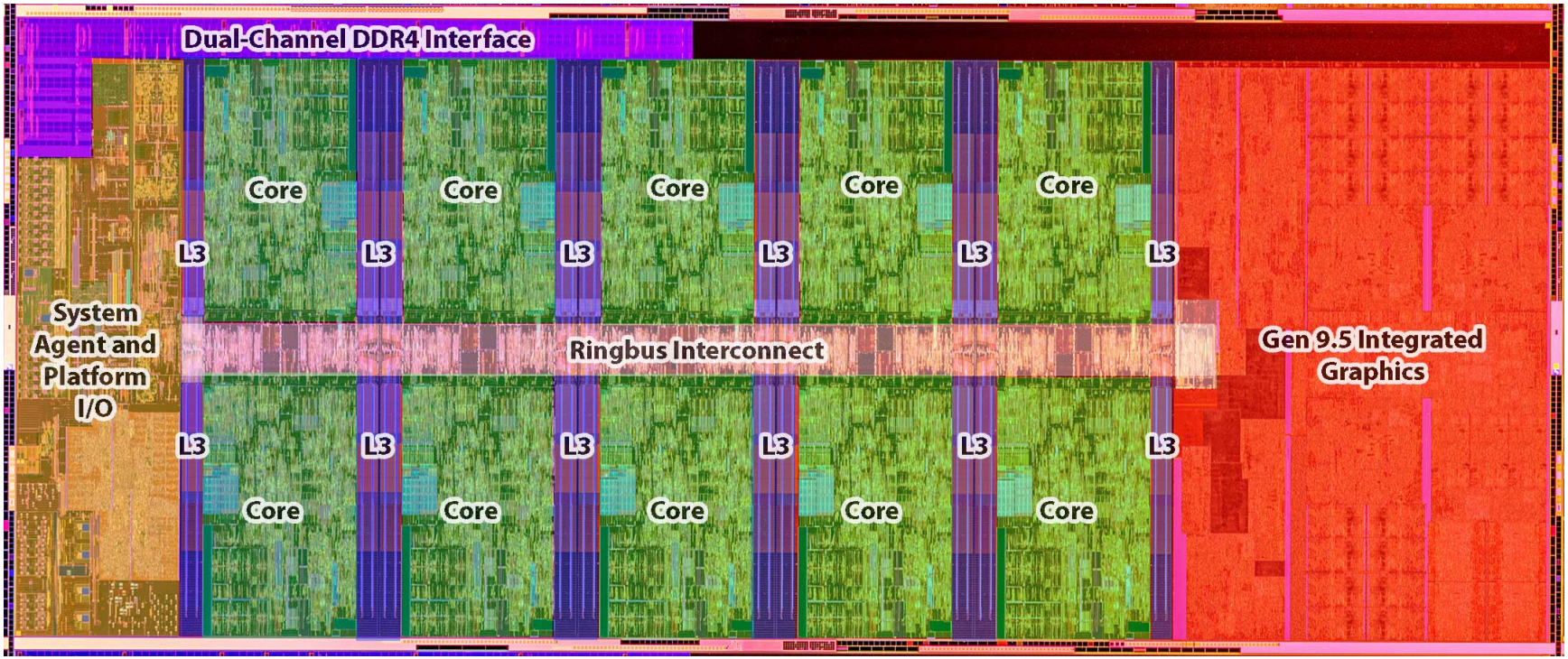
Intel x86 Evolution: Milestones

Name	*Date*	*Transistors*	*MHz*
8086	**1978**	**29K**	**5-10**
386	**1985**	**275K**	**16-33**
Pentium 4E	**2004**	**125M**	**2800-3800**
Core i9 10'th	**2020**		**3600**

- First 16-bit Intel processor. Basis for IBM PC & DOS
- 1MB address space
- First 32 bit Intel processor , referred to as IA32
- First 64-bit Intel x86 processor, referred to as x86-64
- 10 cores

Intel x86 Processors, cont.

- Core i9 10’th generation



Desktop Model

- 10 cores
- Integrated graphics
- 3.6-5.3 GHz
- 125 W

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

Our Coverage

- **x86-64**
  - The standard now
- **We will only cover x86-64**

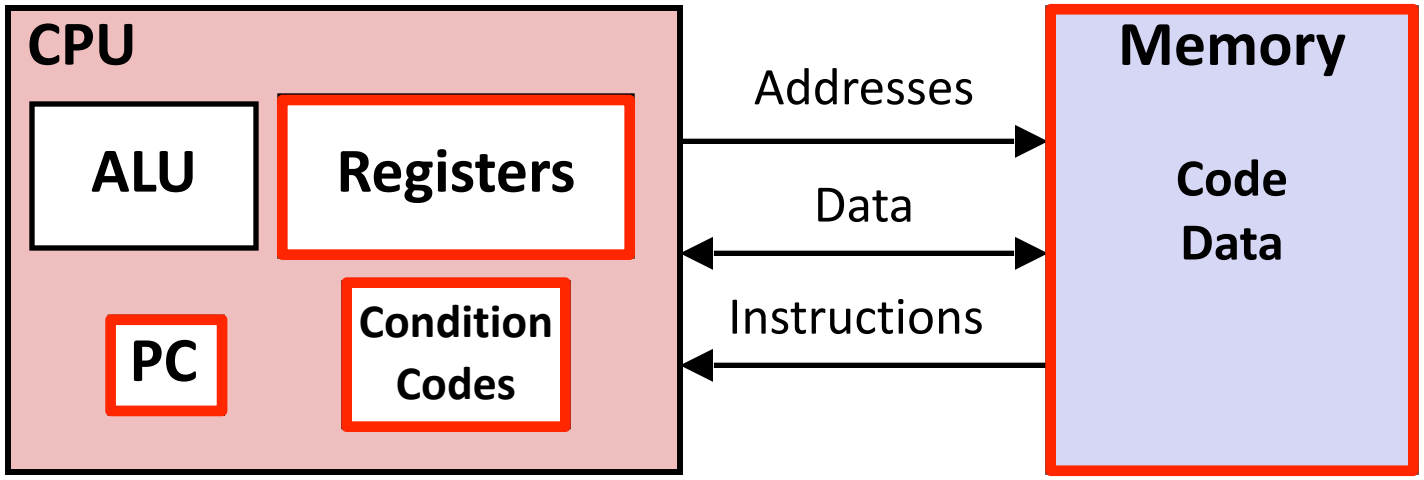
Today: Machine Programming I: Basics

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

Definitions

- **Architecture (ISA: Instruction Set Architecture)**
  - Abstract CPU design that one needs to understand to write assembly
  - 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



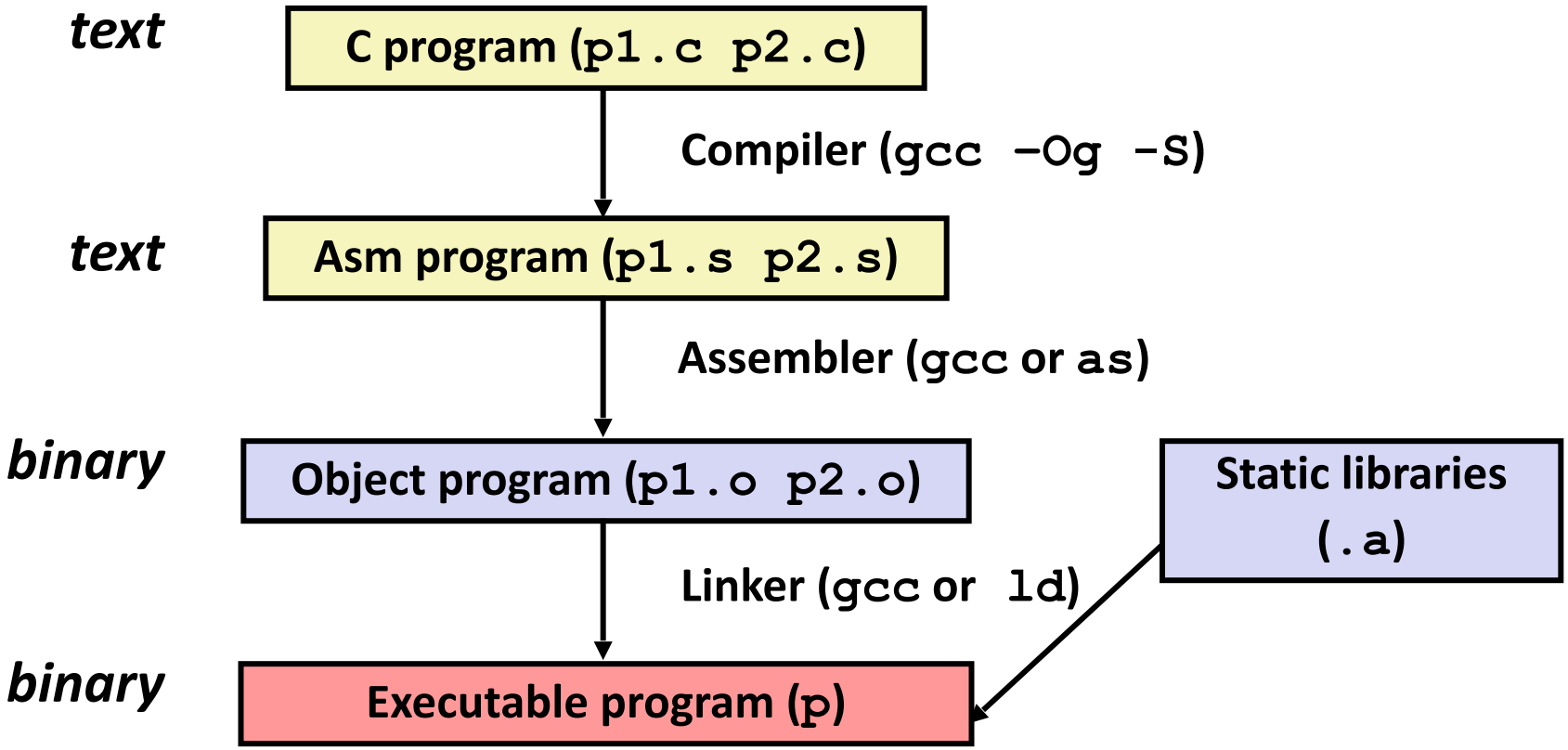
Programmer-Visible State

- **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

Turning C into Object Code

- Use basic optimizations (**-Og**) [New to recent versions of GCC]
  - Compile with command: **`gcc –Og p1.c p2.c -o p`**



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 on your machine (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 or 8 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
    0x48
    0x89
    0x03
    0x5b
    0xc3
```

- **Total of 14 bytes**
- **Starts at address `0x0400595`**

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;
```

C Code

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

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

Assembly

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

  t: Register **%rax**

  dest: Register **%rbx**

  ***dest:** Memory **M[%rbx]**

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

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 rendition of assembly code
- Can be run on either `a.out` (complete executable) or `.o` file

Demo

Alternate Disassembly

Object

```
0x0400595:
    0x53
    0x48
    0x89
    0xd3
    0xe8
    0xf2
    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

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

Today: Machine Programming I: Basics

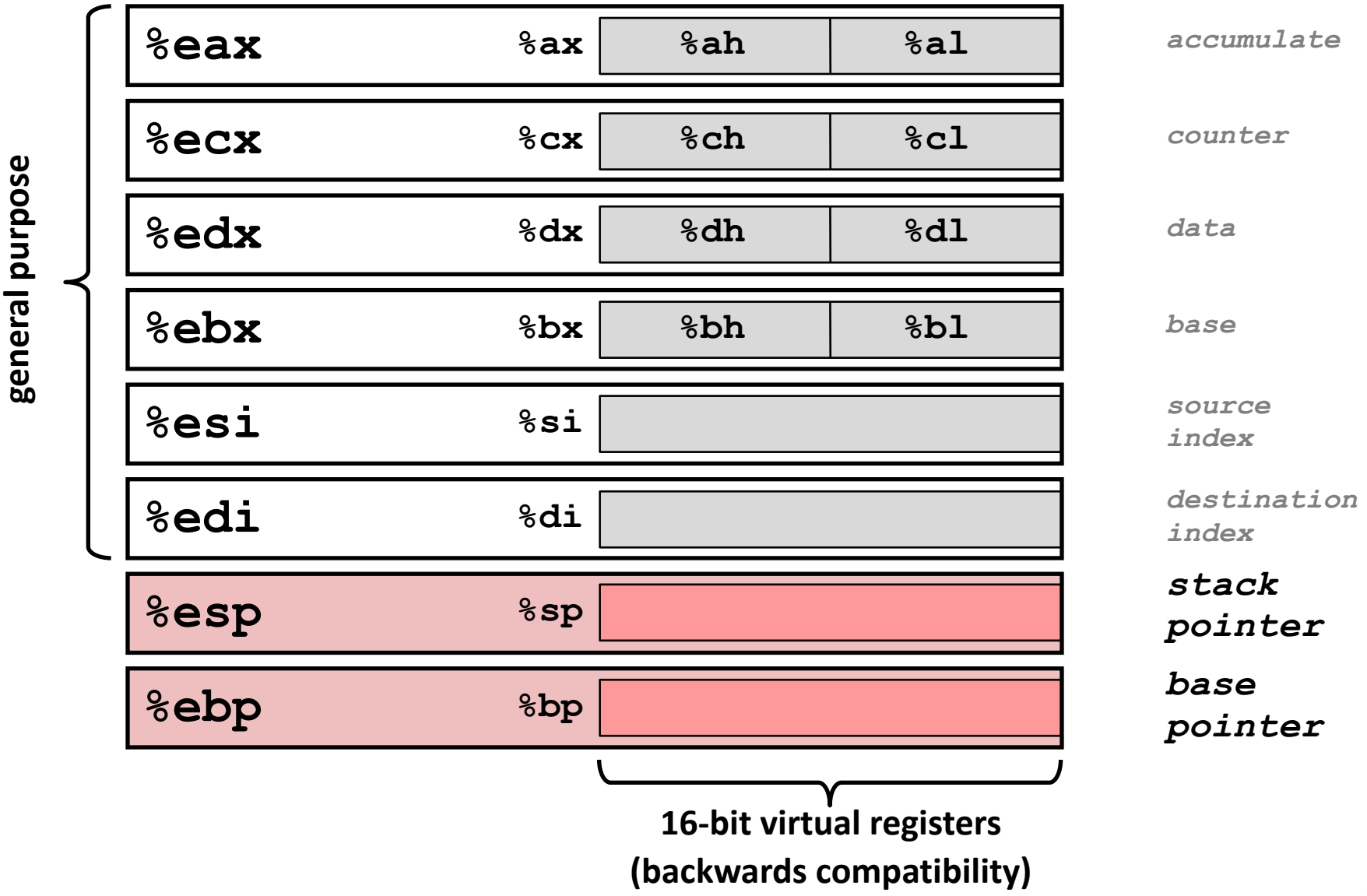
- History of Intel processors and architectures
- C, assembly, machine code
- **Assembly Basics: Registers, operands, move**
- Arithmetic operations

x86-64 Integer Registers

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)

Some History: IA32 Registers

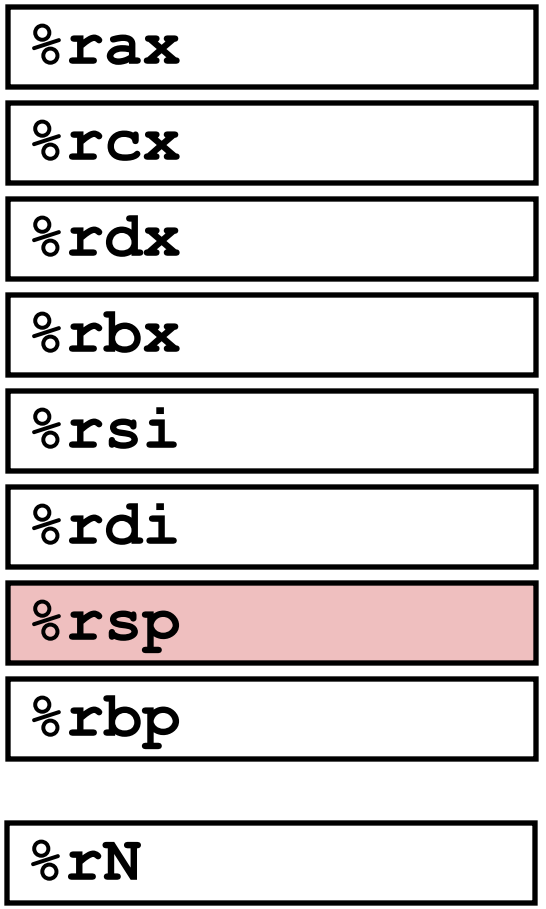


Moving Data

- **Moving Data**

  `movq` *Source, Dest*

- **Operand Types**
  - ***Immediate:*** Constant integer data
    - Example: **`$0x400`**
  - ***Register:*** One of 16 integer registers
    - Example: **`%rax`**
  - ***Memory:***
    - Example: **`(%rax)`**



`movq` Operand Combinations

	Source	Dest	Src,Dest	C Analog
`movq`	*Imm*	*Reg*	`movq $0x4,%rax`	`temp = 0x4;`
		Mem	`movq $-147,(%rax)`	`*p = -147;`
	Reg	*Reg*	`movq %rax,%rdx`	`temp2 = temp1;`
		Mem	`movq %rax,(%rdx)`	`*p = temp;`
	Mem	*Reg*	`movq (%rax),%rdx`	`temp = *p;`

Cannot do memory-memory transfer with a single instruction

Practice

- Write the corresponding assembly code

```
long t1 = t0;                    //t0 is stored in %rax
long t2 = *xp;                   //xp is stored in %rdx
```

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

Practice

- Which ones are correct?

```
movq    %rax, $0x010
movq    %rax, (%rbx)
movq    (%rdx), (%rcx)
```

- Answers

```
movq    %rax, $0x010
        --> Incorrect: immediate can't be a destination

movq    %rax, (%rbx)
        --> Correct

movq    (%rdx), (%rcx)
        --> Incorrect: can't move from a memory address
        to a memory address
```

Simple Memory Addressing Modes

- **Normal (R) Mem[Reg[R]]**
  - Register R specifies memory address
  - Aha! Pointer dereferencing in C

```
movq (%rcx),%rax
```

- **Displacement D(R) —> (D+R) —> Mem[Reg[R]+D]**
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

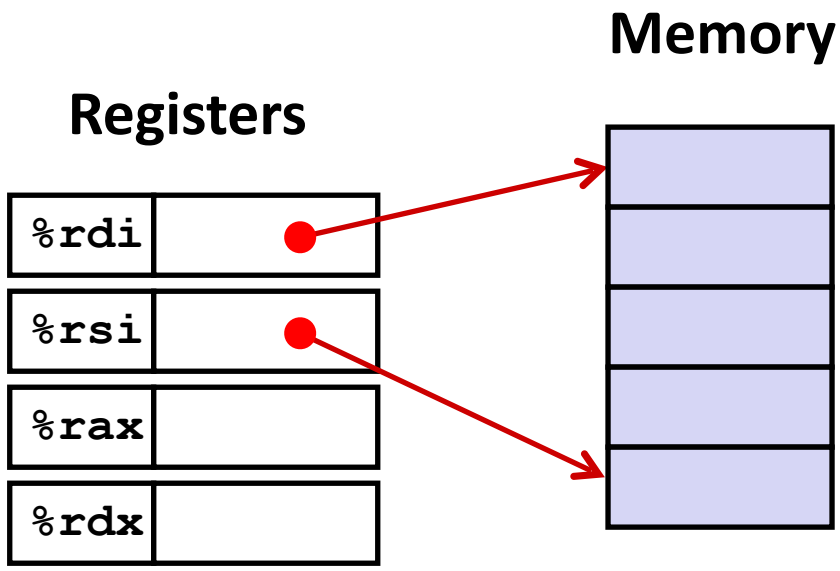
```
movq 2(%rcx),%rdx
```

Example of Simple Addressing Modes

```
void swap
    (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

Understanding Swap()

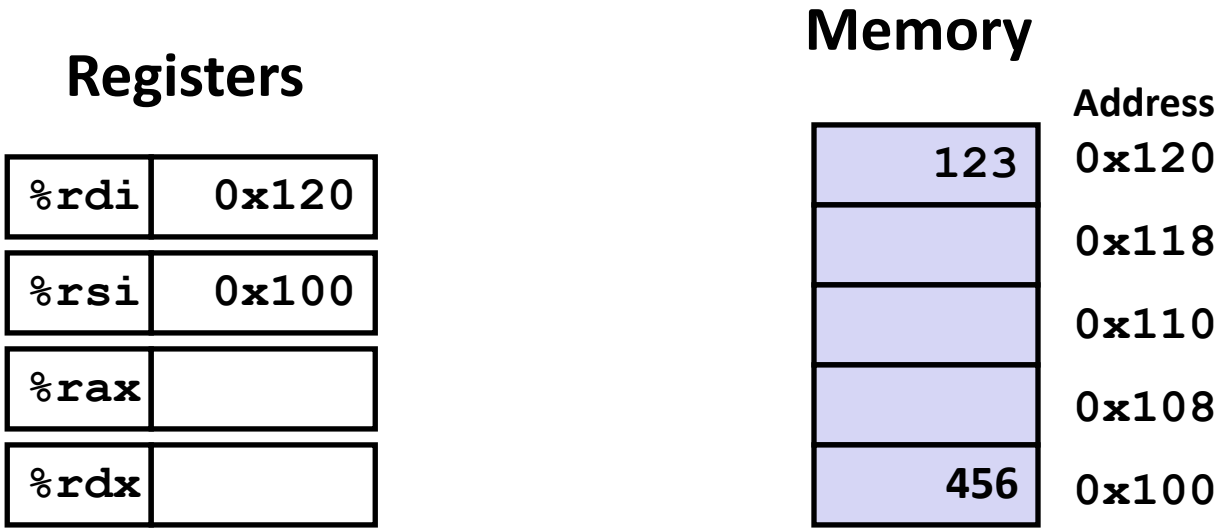
```
void swap
    (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



Register	Value
%rdi	xp
%rsi	yp
%rax	t0
%rdx	t1

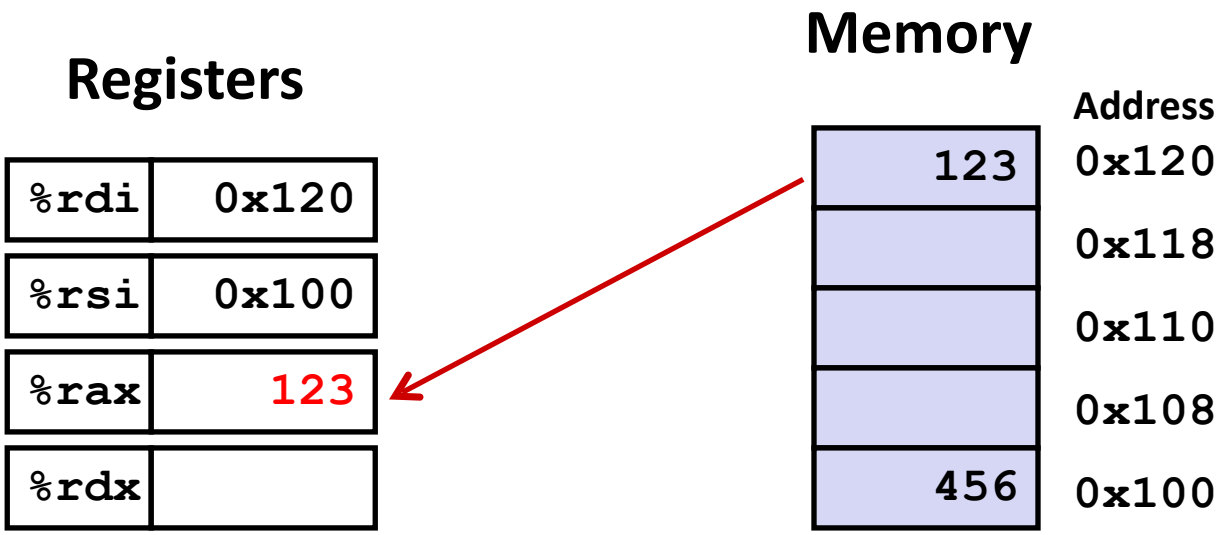
```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding Swap()



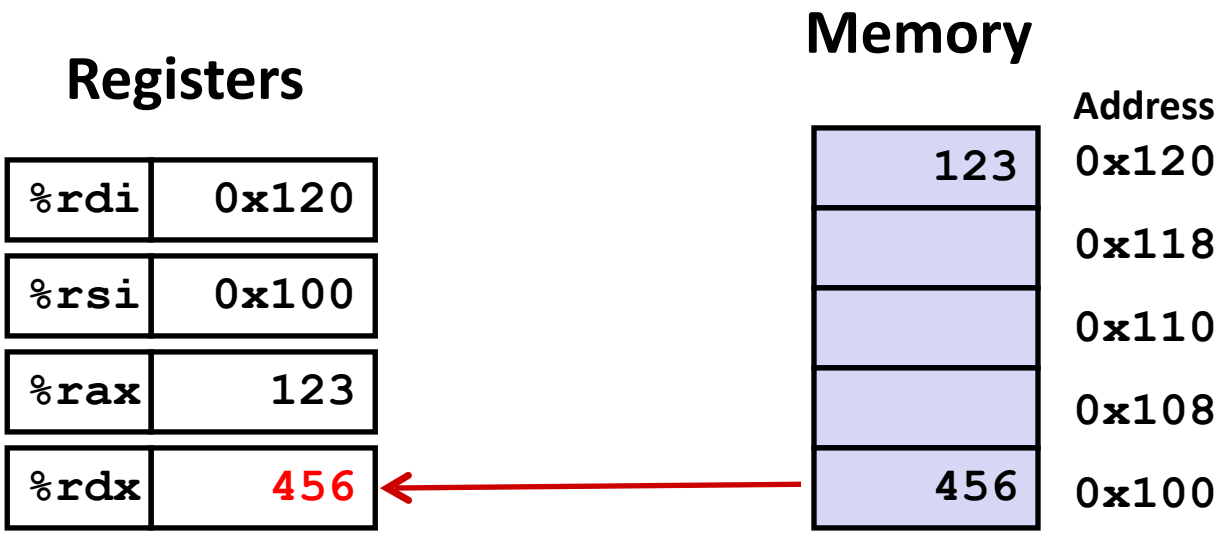

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding Swap()



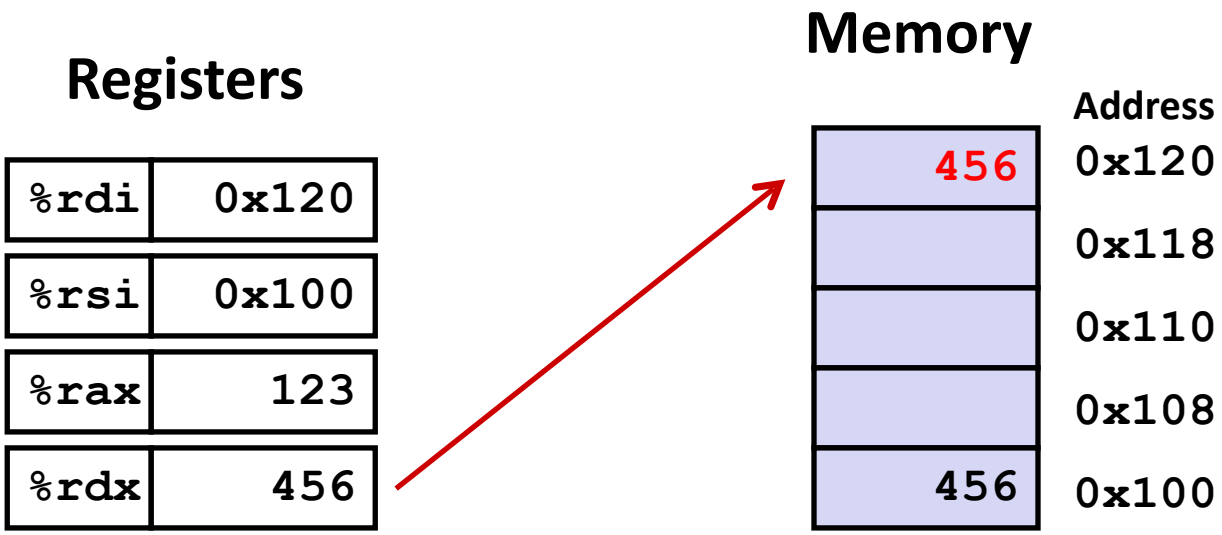

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding Swap()



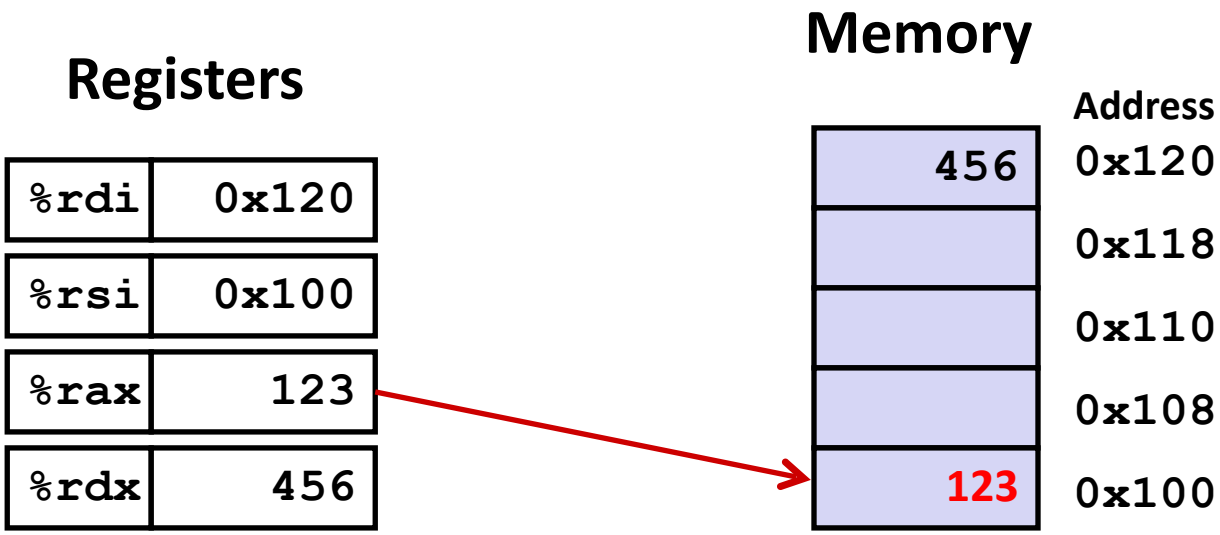

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding Swap()




```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding Swap()




```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Complete Memory Addressing Modes

Most General Form

D(Rb,Ri,S) **(D+Rb+Ri*S)**

- D: Constant "displacement" 1, 2, or 4 bytes
- Rb: Base register: Any of 16 integer registers
- Ri: Index register: Any, except for `%rsp`
- S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

Special Cases

(Rb,Ri)	**(Rb+Ri)**
D(Rb,Ri)	**(D+Rb+Ri)**
(Rb,Ri,S)	**(Rb+Ri*S)**

Complete Memory Addressing Modes

Most General Form

D(Rb,Ri,S) **Mem[Reg[Rb]+S*Reg[Ri]+ D]**

- D: Constant "displacement" 1, 2, or 4 bytes
- Rb: Base register: Any of 16 integer registers
- Ri: Index register: Any, except for `%rsp`
- S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

Special Cases

(Rb,Ri) **Mem[Reg[Rb]+Reg[Ri]]**

D(Rb,Ri) **Mem[Reg[Rb]+Reg[Ri]+D]**

(Rb,Ri,S) **Mem[Reg[Rb]+S*Reg[Ri]]**

Address Computation Examples

%rdx	0xf000
%rcx	0x0100

Expression	Address Computation	Address
0x8(%rdx)	0xf000 + 0x8	0xf008
(%rdx,%rcx)	0xf000 + 0x100	0xf100
(%rdx,%rcx,4)	0xf000 + 4*0x100	0xf400
0x80(,%rdx,2)	2*0xf000 + 0x80	0x1e080

Today: Machine Programming I: Basics

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

Some Arithmetic Operations

- **Two Operand Instructions:**

Format	*Computation*		
`addq`	*Src,Dest*	Dest = Dest + Src	+=
`subq`	*Src,Dest*	Dest = Dest – Src	-=
`imulq`	*Src,Dest*	Dest = Dest * Src	…
`salq`	*Src,Dest*	Dest = Dest << Src	***Also called shlq***
`sarq`	*Src,Dest*	Dest = Dest >> Src	***Arithmetic***
`shrq`	*Src,Dest*	Dest = Dest >> Src	***Logical***
`xorq`	*Src,Dest*	Dest = Dest ^ Src	
`andq`	*Src,Dest*	Dest = Dest & Src	
`orq`	*Src,Dest*	Dest = Dest | Src	

- **Watch out for argument order!**

Some Arithmetic Operations

- **One Operand Instructions**

`incq`	*Dest*	*Dest = Dest + 1*
`decq`	*Dest*	*Dest = Dest – 1*
`negq`	*Dest*	*Dest = – Dest*
`notq`	*Dest*	*Dest = ~Dest*

Address Computation Instruction

- What do you expect the generated assembly to be?

```
long m12(long x)
{
  return x*12;
}
```

Converted to ASM by compiler:

```
leaq (%rdi,%rdi,2), %rax
salq $2, %rax
```

Address Computation Instruction

- **`leaq` *Src*, *Dst***
  - *Src* is address mode expression
  - Set *Dst* to address denoted by expression
- **`Use`**
  - Computing addresses without a memory reference
    - E.g., translation of `p = &x[i];`
- **`leaq 7(%rdx,%rdx,4), %rax`**
  - *Set register %rax to 5 %rdx + 7*

Address Computation Instruction

- **Used also to do arithmetic operations**
  - Computing arithmetic expressions of the form x + k*y
    - k = 1, 2, 4, or 8

```
long m12(long x)
{
  return x*12;
}
```

Converted to ASM by compiler:

```
leaq (%rdi,%rdi,2), %rax # t <- x+x*2
salq $2, %rax            # return t<<2
```

Arithmetic Expression Example

```
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
  leaq    (%rdi,%rsi), %rax
  addq    %rdx, %rax
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx
  leaq    4(%rdi,%rdx), %rcx
  imulq   %rcx, %rax
  ret
```

Interesting Instructions

- **`leaq`**: address computation
- **`salq`**: shift
- **`imulq`**: multiplication
  - But, only used once

Understanding Arithmetic Expression Example

```
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
   leaq    (%rdi,%rsi), %rax    # t1
   addq    %rdx, %rax           # t2
   leaq    (%rsi,%rsi,2), %rdx
   salq    $4, %rdx             # t4
   leaq    4(%rdi,%rdx), %rcx   # t5
   imulq   %rcx, %rax           # rval
   ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	t1, t2, rval
%rdx	t4
%rcx	t5

Practice

- Calculate the address

`%rdx`	`0xff00`
`%rcx`	`0x0110`

Expression	Address Computation	Address
`0x10(%rdx)`	`0xff00 + 0x10`	`0xff10`
`(%rdx,%rcx)`	`0xff00 + 0x0110`	`0x10010`
`(%rdx,%rcx,4)`	`0xff00 + 4*0x0110`	`0x10340`
`0x80(,%rdx,2)`	`2*0xff00 + 0x80`	`0x1fe80`

Practice

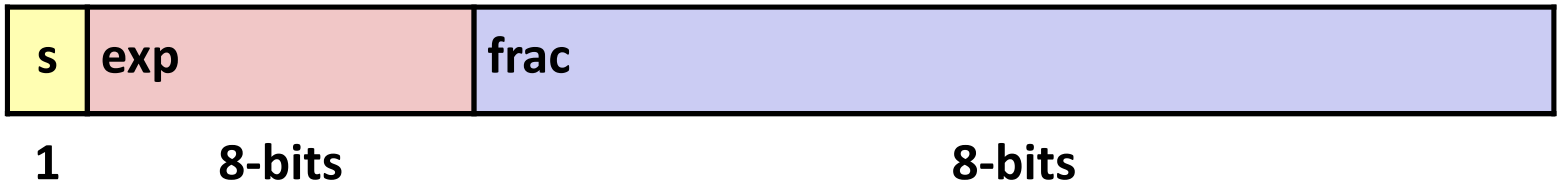
- Represent 7.125 in the normalized encoding of floating points
- Assume 1 bit for sign, 8 bits for exponent and 8 bits for the fractional part

Practice

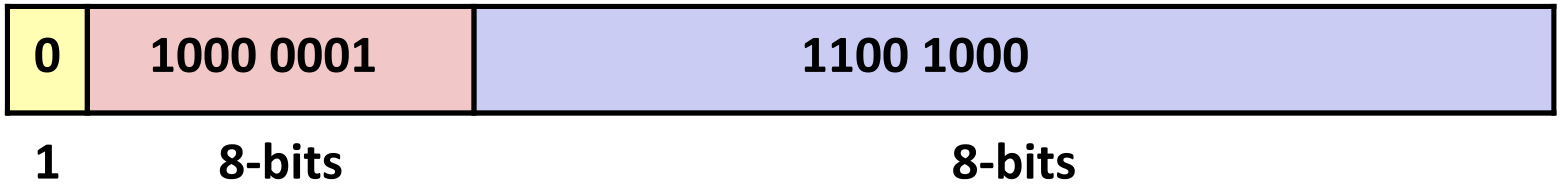
$v = (-1)^s M 2^E$

$E = Exp - Bias$

- $7.125_{10} = 111.001_2$



- $111.001 = 1.\underline{11001} \times 2^2$
  - M = $1.\underline{11001}$ ===> frac = $\underline{11001}$
  - E = 2
    - Exp = E + Bias
    - Bias = $2^{k-1} - 1 = 2^{8-1} - 1 = 2^7 - 1 = 127$
    - ===> Exp = $2 + 127 = 129_{10} = 10000001_2$



Xor properties

- $A \oplus B = ?$

Practice

- What does the following function do?

```
void s(int& a, int& b)
  {
    a = a ^ b;
    b = a ^ b;
    a = a ^ b;
  }
```

- It's a function to swap two variables in place without having to use a temporary variable

Practice

- Write the C analog for the following assembly instructions

Assembly	C Analog
`movq $0x10,%rbx`	`b = 0x10;`
`movq %rbx,(%rax)`	`*a = b;`
`movq $7,(%rcx)`	`*c = 7;`
`movq (%rax),%rdx`	`d = *a;`