Machine-Level Programming I: Basics

CS 485G-006: Systems Programming

Lectures 4 and 5: 25–27 Jan 2016

Adapted from slides by R. Bryant and D. O'Hallaron (http://csapp.cs.cmu.edu/3e/instructors.html)

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 2	**2006**	**291M**	**1060-3500**
Core i7	**2008**	**731M**	**1700-3900**

- First 16-bit Intel processor. Basis for IBM PC & DOS
- 1MB address space
- First 32 bit Intel processor , referred to as IA32
- Added “flat addressing”, capable of running Unix
- First 64-bit Intel x86 processor, referred to as x86-64
- Based on AMD’s architecture (specs 2000, Opteron released in 2003).
- First multi-core Intel processor

Our Coverage

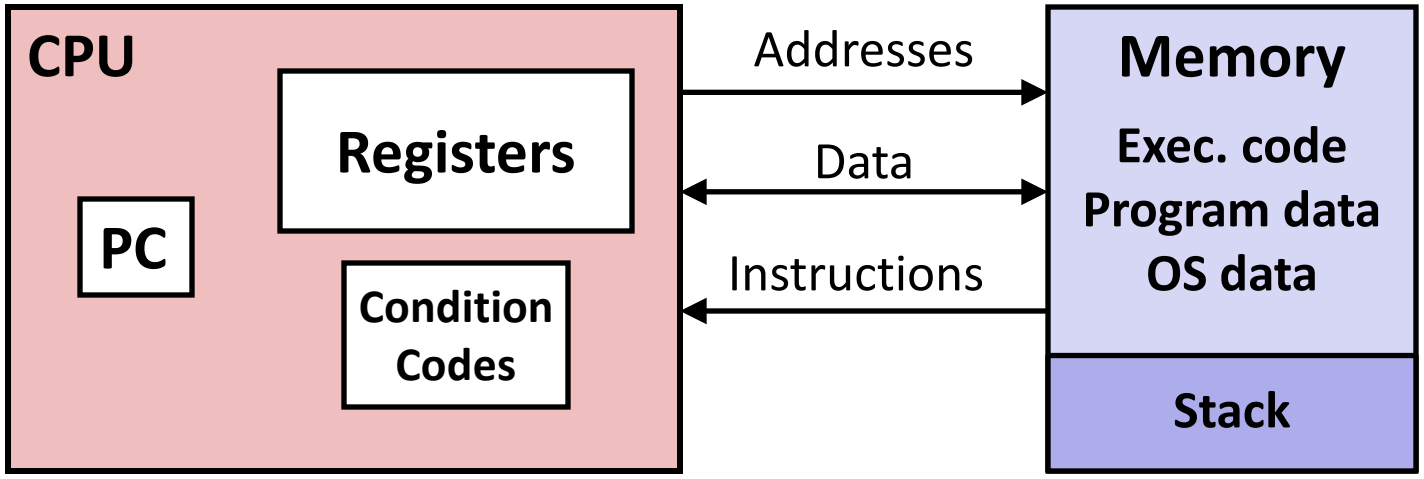
- **x86-64**
  - 64-bit architecture (since 2003): most modern machines and VMs.
  - 3rd ed. of the book.
  - `gcc hello.c`
  - This is what we'll cover in lecture and what you are responsible for knowing.

- **IA32**
  - 32-bit architecture (since 1985): common until rather recently.
  - 2nd ed. of the book, "web aside" on 3rd edition site
  - `gcc –m32 hello.c`
  - I'll occasionally mention how x86-64 differs from IA32

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: IA32, Itanium, x86-64
  - ARM: Used in almost all mobile phones
  - MIPS, AVR32 (embedded), Power (IBM mainframes), SPARC, …

Assembler Programmer's View

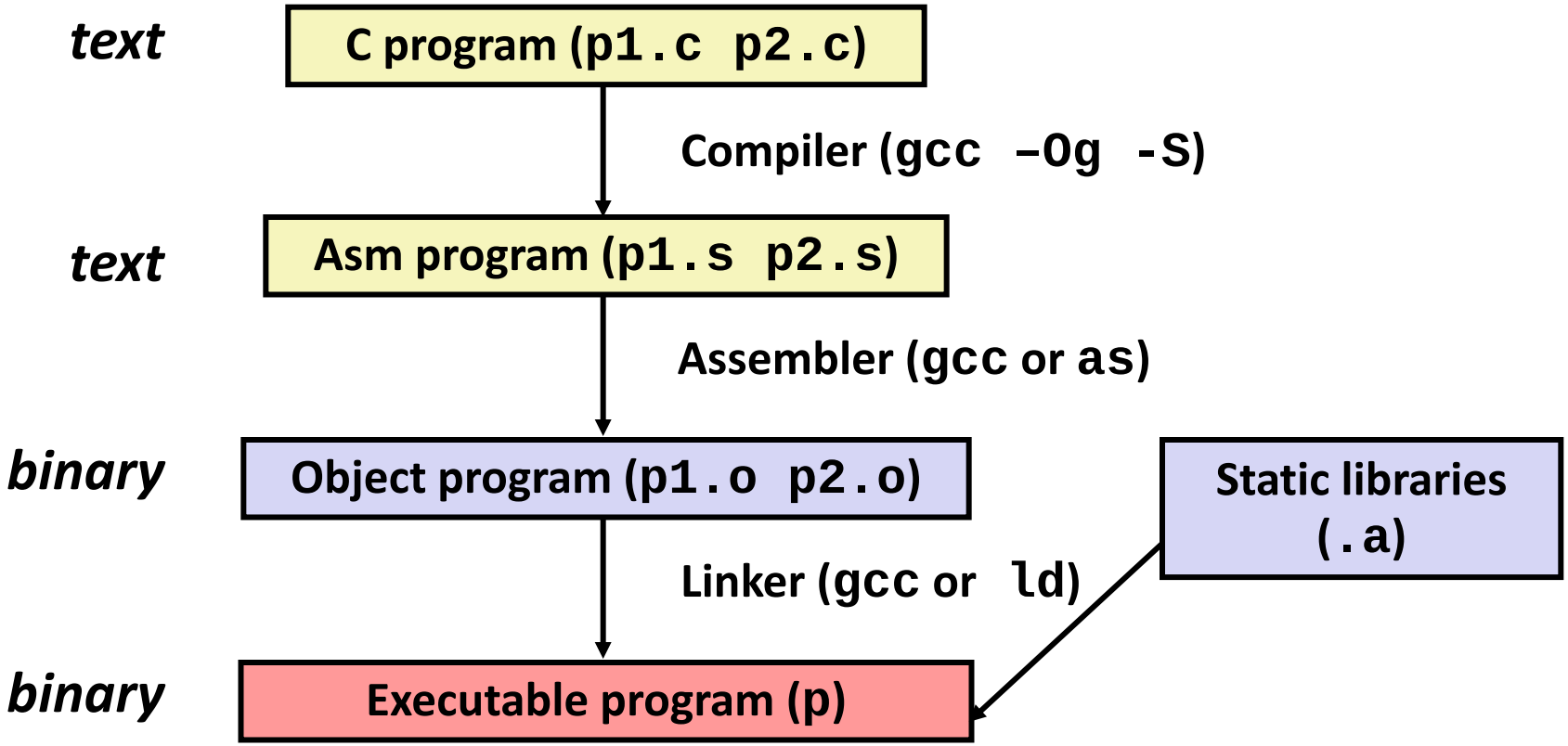


Programmer-Visible State

- **PC: Program counter**
  - Address of next instruction
  - Called "RIP" (x86-64) ("EIP" in IA32)
- **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
  - Executable code, user (program) data, some OS 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 that don't make debugging harder (`-Og`)
  - Put resulting binary in file `p`



Compiling Into Assembly

C Code (sum.c)

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

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

Generated x86-64 Assembly

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

Obtain (on VMs) with command

```
gcc –Og –S sumstore.c
```

Produces file `sumstore.s`

Warning: Will get very different results on other machines 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

Object Code

Code for `sumstore`

```
0x0400562:
    0x53
    0x48
    0x89
    0xd3
    0xe8
    0xf2
    0xff
    0xff
    0xff
    0x48
    0x89
    0x03
    0x5b
    0xc3
```

- Total of 14 bytes
- Each instruction (here) 1, 3, or 5 bytes
- Starts at address `0x0400562`

- **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*
    - Another linking step occurs when program begins execution.

Machine Instruction Example

```
*dest = t;
```

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

```
0x400562:  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 **0x400562**

Disassembling Object Code

Disassembled

```
0000000000400562 <sumstore>:
  400562:  53                   push   %rbx
  400563:  48 89 d3             mov    %rdx,%rbx
  400566:  e8 f2 ff ff ff       callq  40055d <plus>
  40056b:  48 89 03             mov    %rax,(%rbx)
  40056e:  5b                   pop    %rbx
  40056f:  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

Alternate Disassembly

Object

```
0x0400562:
    0x53
    0x48
    0x89
    0xd3
    0xe8
    0xf2
    0xff
    0xff
    0xff
    0x48
    0x89
    0x03
    0x5b
    0xc3
```

Disassembled

```
Dump of assembler code for function sumstore:
 0x0000000000400562 <+0>: push   %rbx
 0x0000000000400563 <+1>: mov    %rdx,%rbx
 0x0000000000400566 <+4>: callq  0x400590 <plus>
 0x000000000040056b <+9>: mov    %rax,(%rbx)
 0x000000000040056e <+12>:pop    %rbx
 0x000000000040056f <+13>:retq
```

- **Within gdb Debugger**

  `gdb sum`

  `disassemble sumstore`

  - Disassemble procedure

  `x/14xb sumstore`

  - Examine the 14 bytes starting at `sumstore`

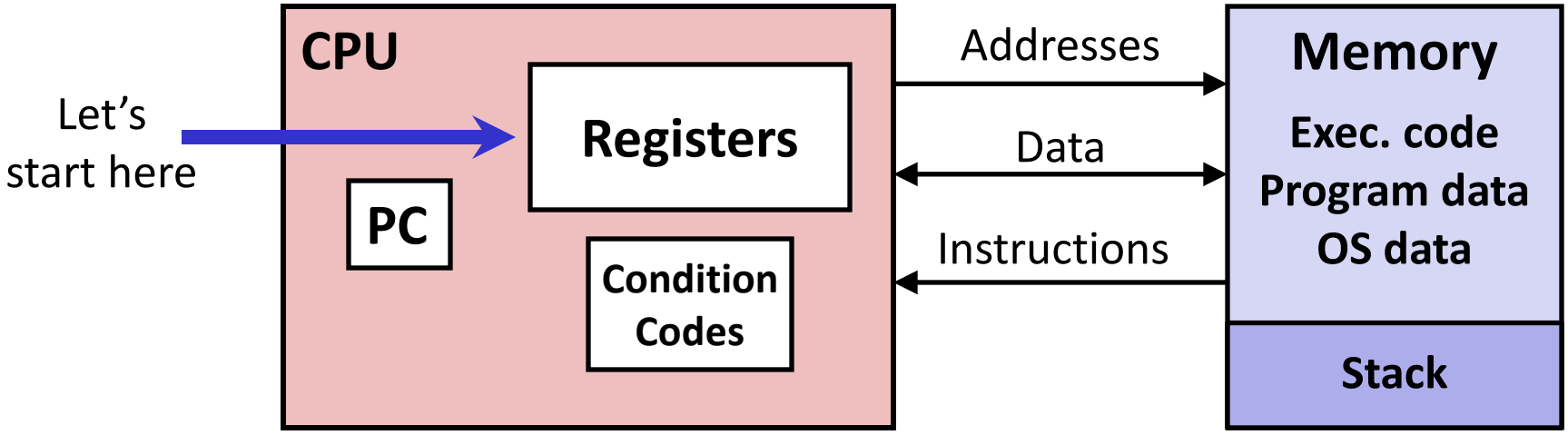
What Can be Disassembled?

```
% objdump -d WINWORD.EXE

WINWORD.EXE:   file format pei-i386

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

Recall: Assembler Programmer's View



Programmer-Visible State

- **PC: Program counter**
  - Address of next instruction
  - Called "RIP" (x86-64) ("EIP" in IA32)
- **Register file**
  - Heavily used program data
- **Condition codes**
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching

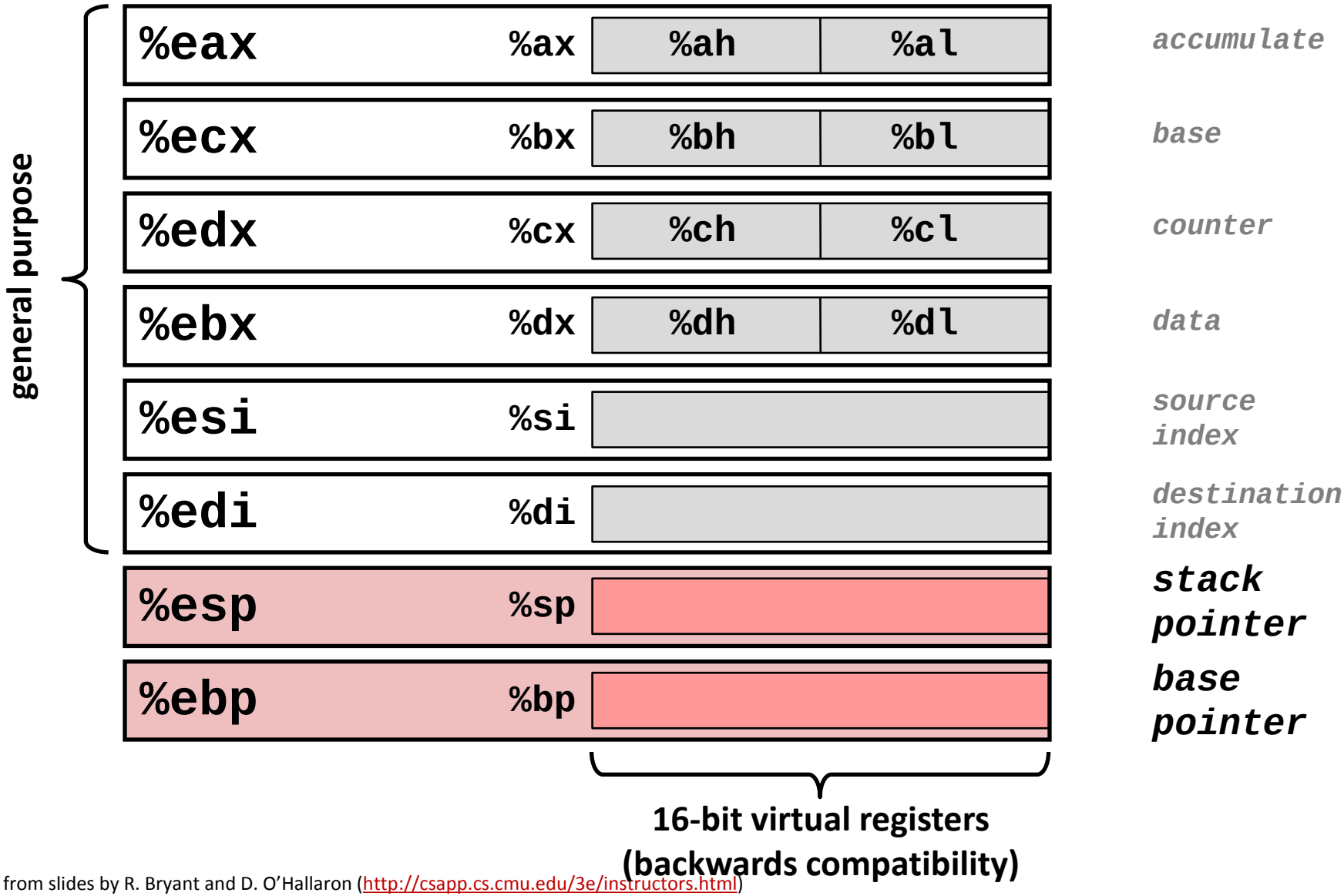
- **Memory**
  - Byte addressable array
  - Executable code, user (program) data, some OS data
  - Stack to support procedures

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***

  The "q" means it's moving 64 bits (a quadword)

- **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: **`(%rax)`**
    - Various other "address modes"

`%rax`
`%rcx`
`%rdx`
`%rbx`
`%rsi`
`%rdi`
`%rsp`
`%rbp`
`%rN` *(8…15)*

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

Simple Memory Addressing Modes

- **Normal (R) Mem[Reg[R]]**
  - Register R specifies memory address
  - Pointer dereferencing in C
  - **`movq (%rcx),%rax`**
- **Displacement D(R) Mem[Reg[R]+D]**
  - Register R specifies start of memory region
  - Constant displacement D specifies offset
  - Array indexing, members of a struct/class, …
  - **`movq 8(%rbp),%rdx`**

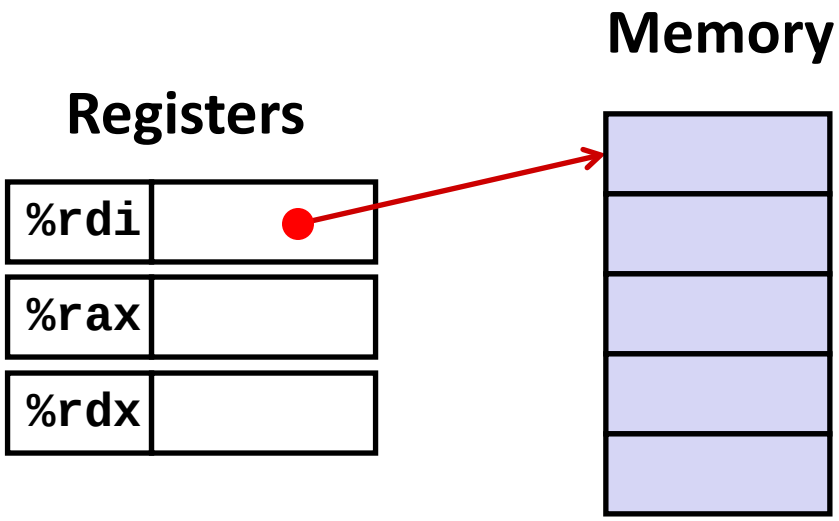
Example of Simple Addressing Modes

```
void swap(long xs[2])
{
  long t0 = xs[0]; // or *xs
  long t1 = xs[1];
  xs[0] = t1;
  xs[1] = t0;
}
```

```
swap:
    movq    (%rdi), %rax
    movq    8(%rdi), %rdx
    movq    %rdx, (%rdi)
    movq    %rax, 8(%rdi)
    ret
```

Understanding Swap()

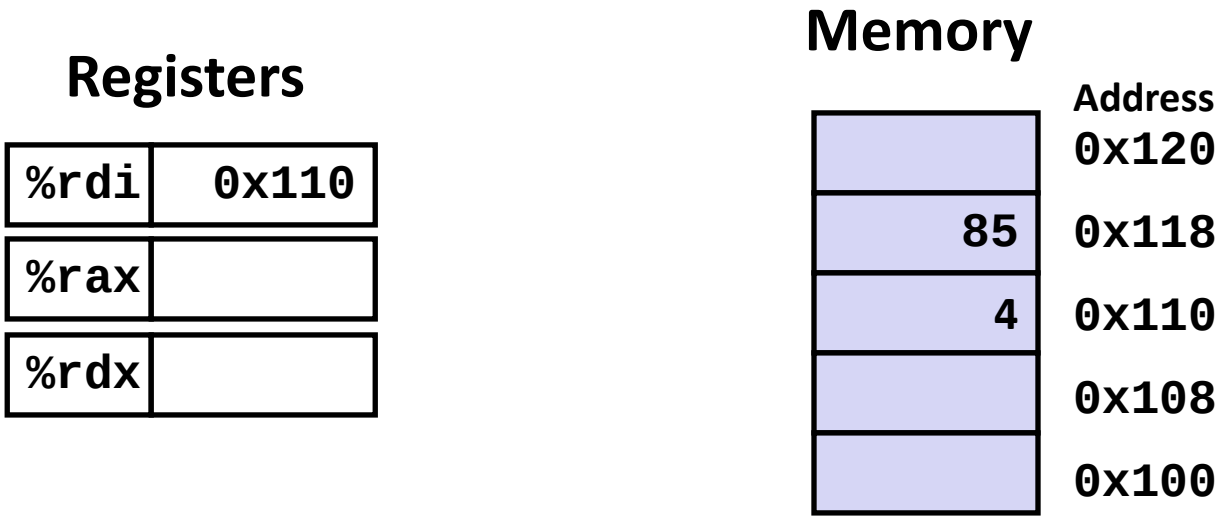
```
void swap(long xs[2])
{
  long t0 = xs[0];
  long t1 = xs[1];
  xs[0] = t1;
  xs[1] = t0;
}
```



Register	Value
%rdi	xs = &xs[0]
%rax	t0
%rdx	t1

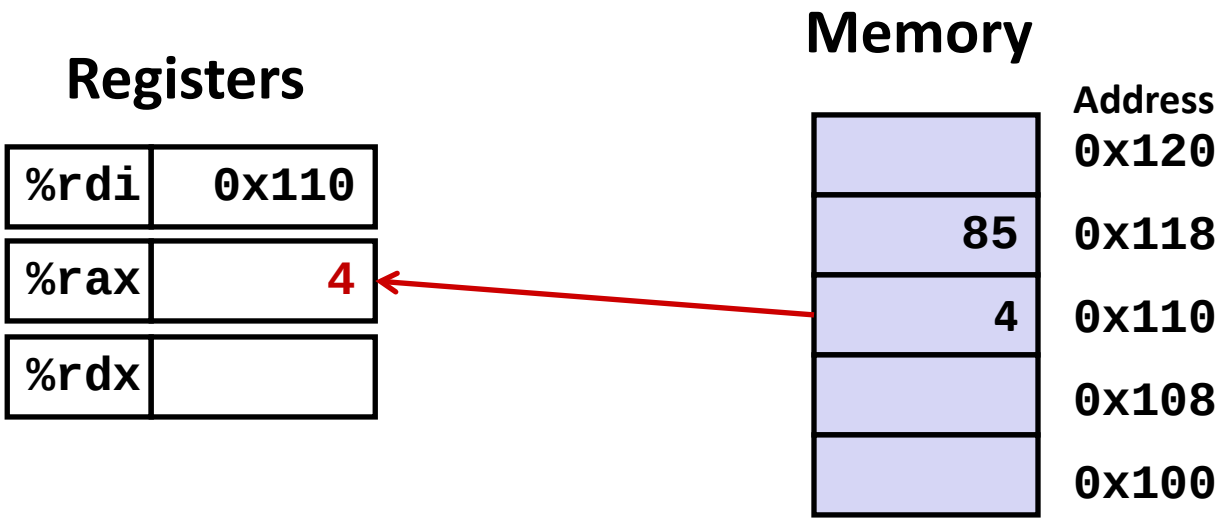
```
swap:
    movq    (%rdi), %rax    # t0 = *xs
    movq    8(%rdi), %rdx   # t1 = xs[1]
    movq    %rdx, (%rdi)    # xs[0] = t1
    movq    %rax, 8(%rdi)   # xs[1] = t0
    ret
```

Understanding Swap()



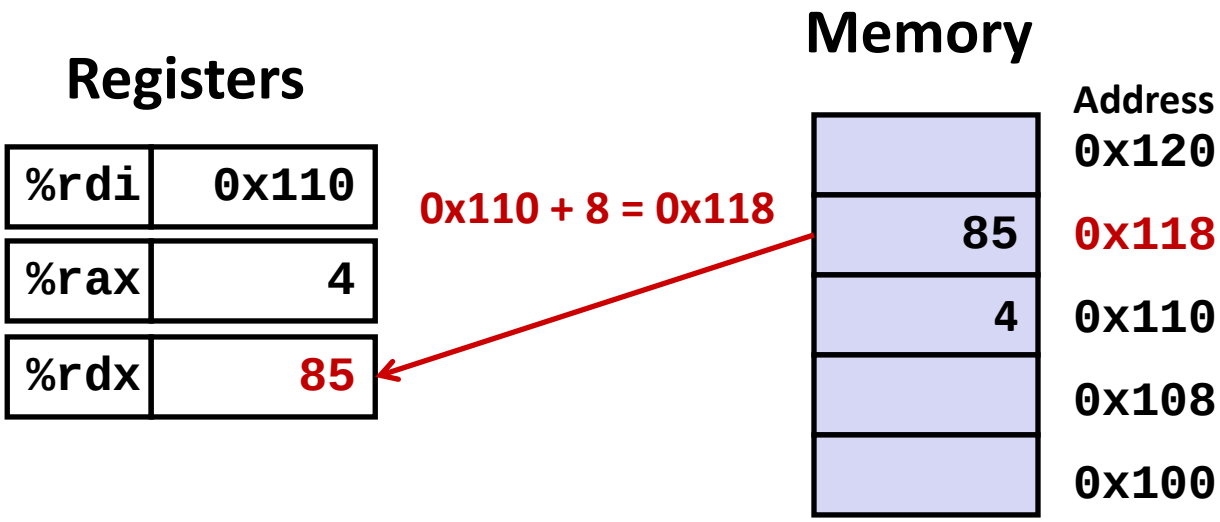

```
swap:
   movq    (%rdi), %rax    # t0 = *xs
   movq    8(%rdi), %rdx   # t1 = xs[1]
   movq    %rdx, (%rdi)    # xs[0] = t1
   movq    %rax, 8(%rdi)   # xs[1] = t0
   ret
```

Understanding Swap()



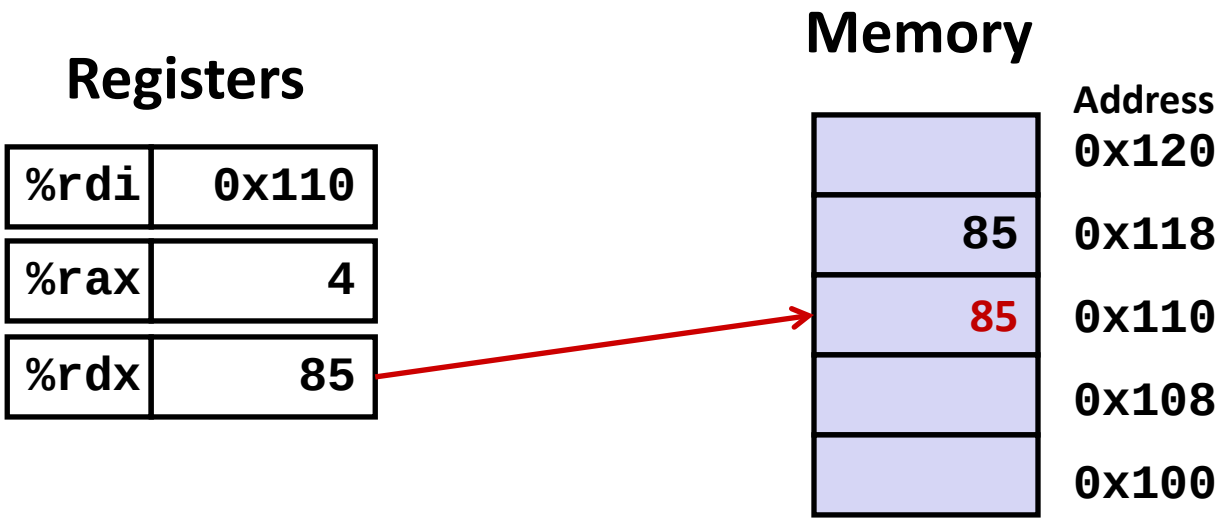

```
swap:
    movq    (%rdi), %rax    # t0 = *xs
    movq    8(%rdi), %rdx   # t1 = xs[1]
    movq    %rdx, (%rdi)    # xs[0] = t1
    movq    %rax, 8(%rdi)   # xs[1] = t0
    ret
```

Understanding Swap()



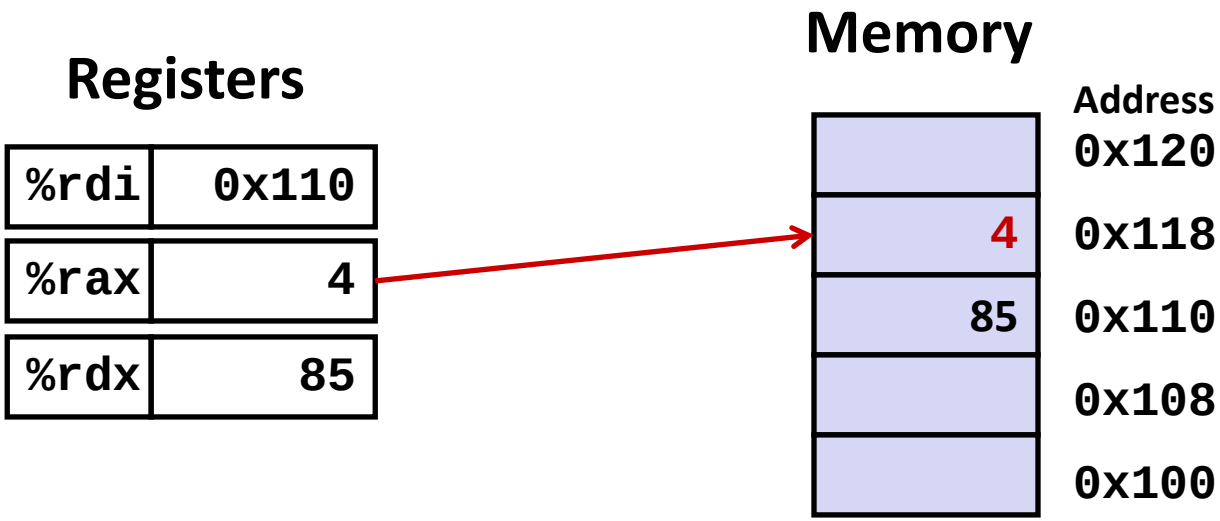

```
swap:
    movq    (%rdi), %rax    # t0 = *xs
    movq    8(%rdi), %rdx   # t1 = xs[1]
    movq    %rdx, (%rdi)    # xs[0] = t1
    movq    %rax, 8(%rdi)   # xs[1] = t0
    ret
```

Understanding Swap()




```
swap:
    movq    (%rdi), %rax    # t0 = *xs
    movq    8(%rdi), %rdx   # t1 = xs[1]
    movq    %rdx, (%rdi)    # xs[0] = t1
    movq    %rax, 8(%rdi)   # xs[1] = t0
    ret
```

Understanding Swap()




```
swap:
    movq    (%rdi), %rax    # t0 = *xs
    movq    8(%rdi), %rdx   # t1 = xs[1]
    movq    %rdx, (%rdi)    # xs[0] = t1
    movq    %rax, 8(%rdi)   # xs[1] = t0
    ret
```

Simple Memory Addressing Modes

- **Normal (R) Mem[Reg[R]]**
  - Register R specifies memory address
  - Pointer dereferencing in C
  - **`movq (%rcx),%rax`**
- **Displacement D(R) Mem[Reg[R]+D]**
  - Register R specifies start of memory region
  - Constant displacement D specifies offset
  - Array indexing, members of a struct/class, …
  - **`movq 8(%rbp),%rdx`**

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

Address Computation Instruction

- **`leaq` *Src*, *Dst***
  - *Src* is address mode expression
  - Set *Dst* to address denoted by expression
  - "Load effective address (quadword)"
- **Uses**
  - Computing addresses without a memory reference
    - E.g., translation of **`p = &x[i];`**
  - Computing arithmetic expressions of the form x + k*y
    - k = 1, 2, 4, or 8
- **Example**

```
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
```

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!**
- **No distinction between signed and unsigned int (why?)**

Some Arithmetic Operations

- **One-Operand Instructions**

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

- **See book for more, or "Intel 64 developer's manual"**

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