Machine Level Programming: Basics

Computer Systems Organization (Spring 2017)
CSCI-UA 201, Section 2

Instructor: Joanna Klukowska

Slides adapted from
Randal E. Bryant and David R. O'Hallaron (CMU)
Mohamed Zahran (NYU)

Why do we look at machine code?

- understanding how the high-level programming language instructions are executed on a processor
- understanding how optimizing high-level program affects instructions executed in practice
- understanding security flaws of programs
- understanding things that are not handled at the high-level programming language

We will be working with the machine code for x86-64 processors.

A Bit of History

Intel x86 Processors

- Totally 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.

We will just scratch the surface of the available instructions.

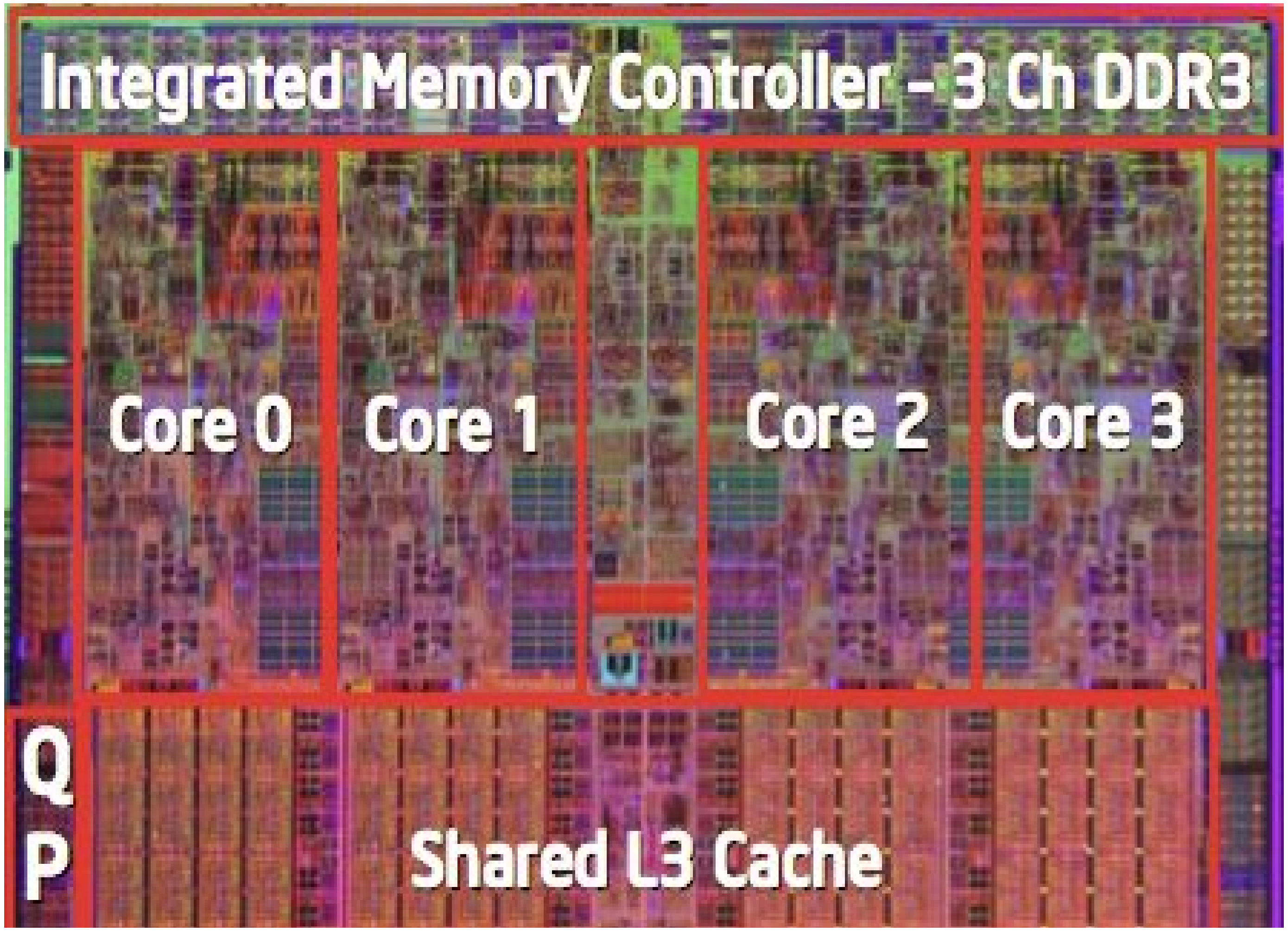
The **hertz** (symbol **Hz**) is the unit of frequency in the International System of Units (SI) and is defined as one cycle per second. Ex:
10^6 Hz = 1 **MHz =** 10^6 cycles/repetitions per second

Intel x86 Evolution

Name	Date	Transistors	MHz
8086	1978	29K	5-10
○ First 16-bit processor. Basis for IBM PC & DOS			
○ 1MB address space			
386	1985	275K	16-33
○ First 32 bit processor, referred to as IA32			
○ Capable of running Unix			
○ 32-bit Linux/gcc uses no instructions introduced in later models			
Pentium 4F	2004	125M	2800-3800
○ First 64-bit processor, referred to as x86-64			
Core i7	2008	731M	2667-3333

Notice that the speed is not increasing as much any more.

Schematic of an Intel Processor



64-bit History

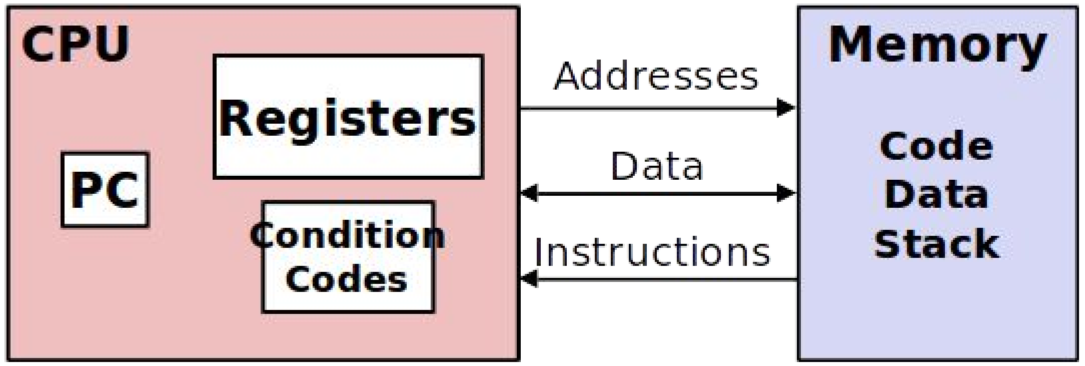
- 2001: Intel Attempts Radical Shift from IA32 to IA64
  - Totally different architecture (Itanium)
  - Executes IA32 code only as legacy
  - Performance disappointing
- 2003: AMD Steps in with Evolutionary Solution
  - x86-64 (now called “AMD64”)
- Intel Felt Obligated to Focus on IA64
  - Hard to admit 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

C, assembly, machine code

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.
    - Target of the compiler
- 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 devices, Raspberry Pi

Assembly/Machine Code View of a Computer



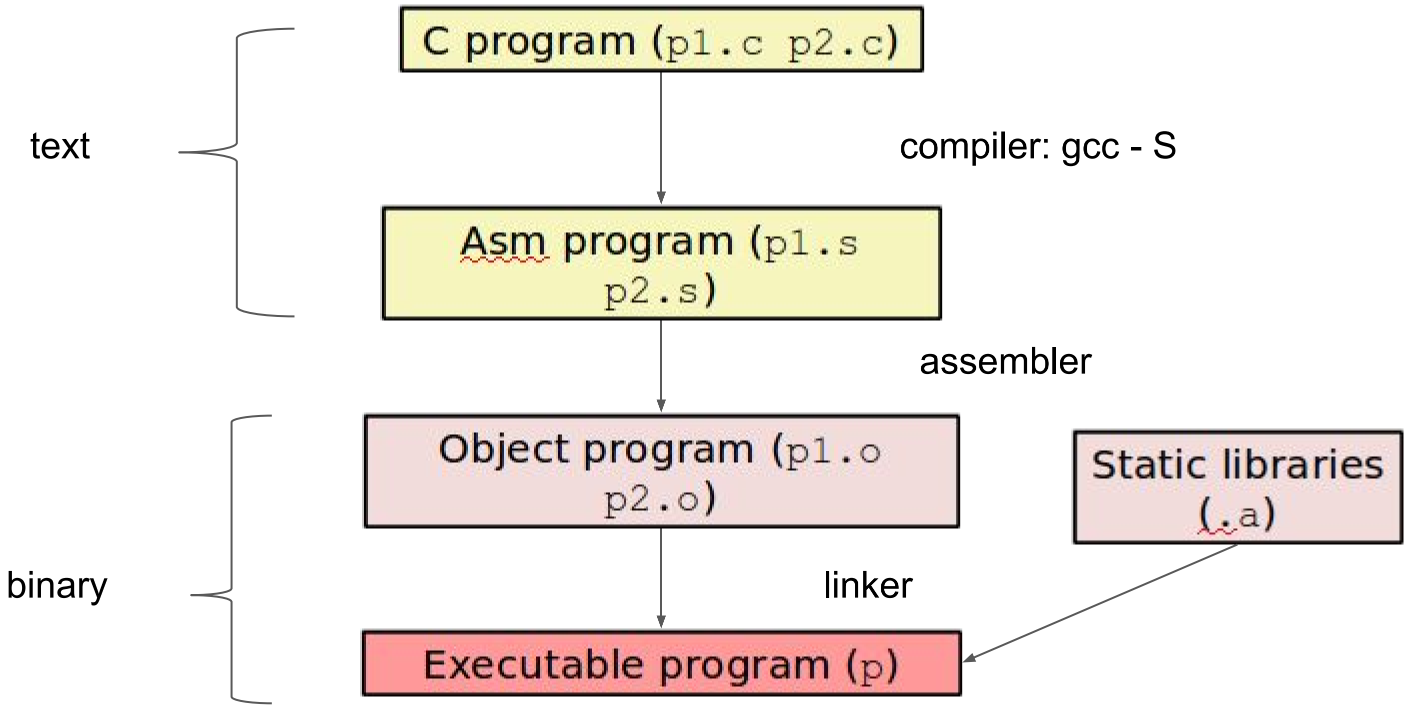
Programmer-Visible State

- PC: Program counter / instruction pointer
    - 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

- 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



Compiling into Assembly

```
long plus(long x, long y) {
    return x + y;
}

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

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

Assembly code generated using

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

output written to `sum.s` file.

Note1: the assembly code will be different for different versions of gcc and different compiler settings. The generated code should be equivalent in terms of what it does, though.

Note2: for now we ignore all instructions in the .s file that start with a dot - they are not really part of the assembly.

Assembly Characteristics: Data Types

- “Integer” data of 1, 2, 4, or 8 bytes
  - Data values (it does not matter if it is signed or not at the level of assembly)
  - Addresses (untyped pointers)
- Floating point data of 4, 8, or 10 bytes
  - we will not really go into floating point numbers at the level of assembly
- Code: Byte sequences encoding series of instructions
- No aggregate types such as arrays or structures, just contiguously allocated bytes in memory

Assembly 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

Very limited in what can be done in one instruction - does only one thing: move data, single simple arithmetic operation, memory dereference.

Object Code

```
gcc -Og -c sum.c
objdump -d sum.o
```

```
0000000000000005 <sumstore>:
   5:   53                       push   %rbx
   6:   48 89 d3                 mov    %rdx,%rbx
   9:   e8 00 00 00 00           callq  e <sumstore+0x9>
   e:   48 89 03                 mov    %rax,(%rbx)
  11:   5b                       pop    %rbx
  12:   c3                       retq
```

Total of 14 bytes. Each instruction can use a different number of bytes. Stats at location 0x4005a2.

- 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 Instructions - Example

- C Code
  - Store value t where designated by dest

```
*dest = t;
```

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

t:	register	%rax
dest:	register	%rbx
*dest:	memory	M[%rbx]

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

- Object Code
  - 3-byte instruction
  - Stored at address 0x40059e

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

Disassembling Object Code

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

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

```
0000000004005a2 <sumstore>:
 4005a2:    53                     push   %rbx
 4005a3:    48 89 d3               mov    %rdx,%rbx
 4005a6:    e8 f2 ff ff ff         callq  40059d <plus>
 4005ab:    48 89 03               mov    %rax,(%rbx)
 4005ae:    5b                     pop    %rbx
 4005af:    c3                     retq
```

Alternate Disassembly

Within gdb Debugger

```
gdb sum
```

- `disassemble sumstore`

  Disassemble procedure

- `x/14xb sumstore`

  Examine the 14 bytes starting at sumstore

```
Dump of assembler code for function sumstore:
   0x00000000004005a2 <+0>:     push   %rbx
   0x00000000004005a3 <+1>:     mov    %rdx,%rbx
   0x00000000004005a6 <+4>:     callq  0x40059d <plus>
   0x00000000004005ab <+9>:     mov    %rax,(%rbx)
   0x00000000004005ae <+12>:    pop    %rbx
   0x00000000004005af <+13>:    retq
End of assembler dump.
```

```
0x4005a2 <sumstore>:    0x53  0x48  0x89  0xd3  0xe8  0xf2  0xff  0xff
0x4005aa <sumstore+8>:  0xff  0x48  0x89  0x03  0x5b  0xc3
```

What Can be Disassembled?

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

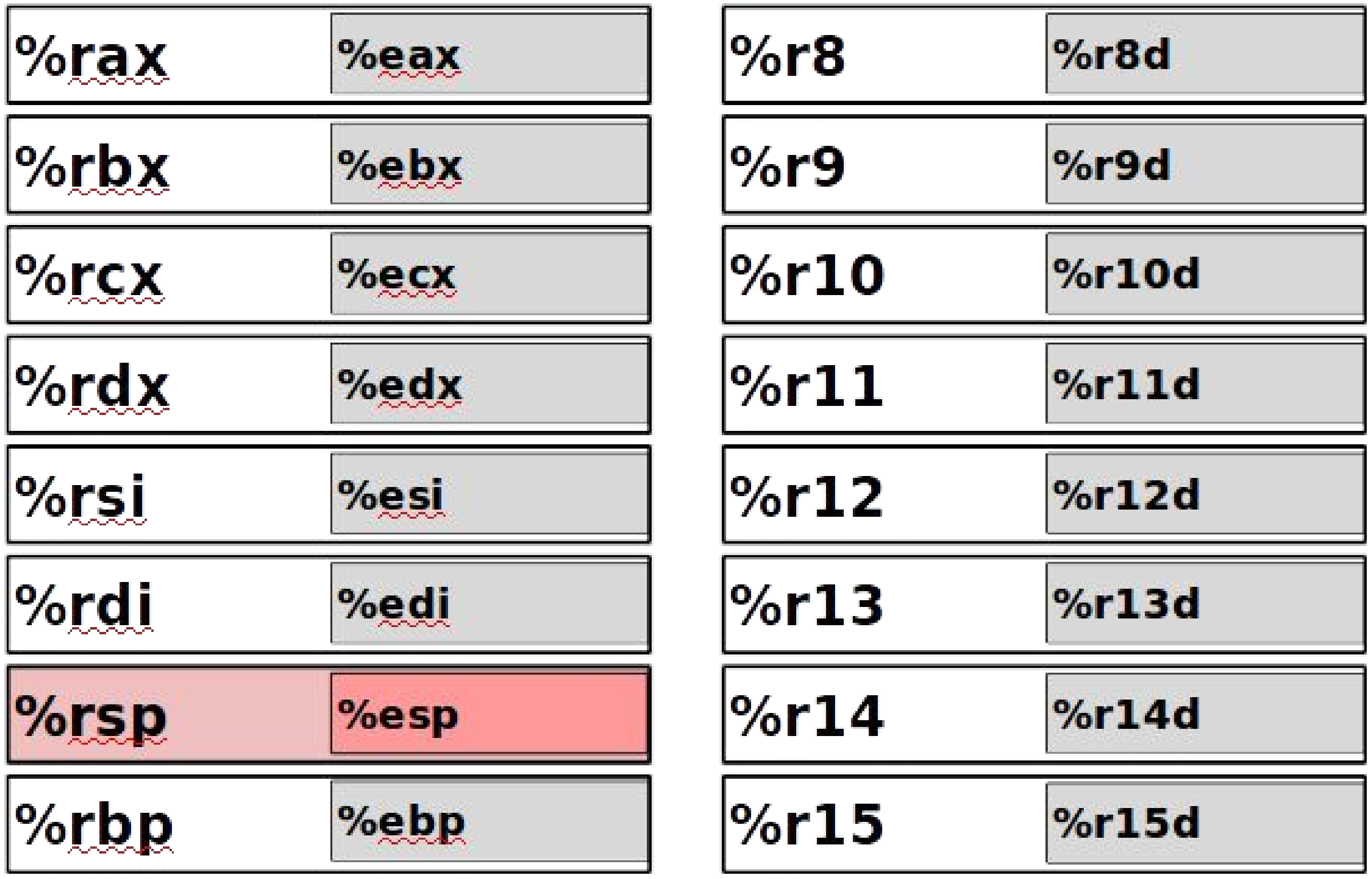
BUT:

The end user license agreement for some software

forbids reverse engineering of code.

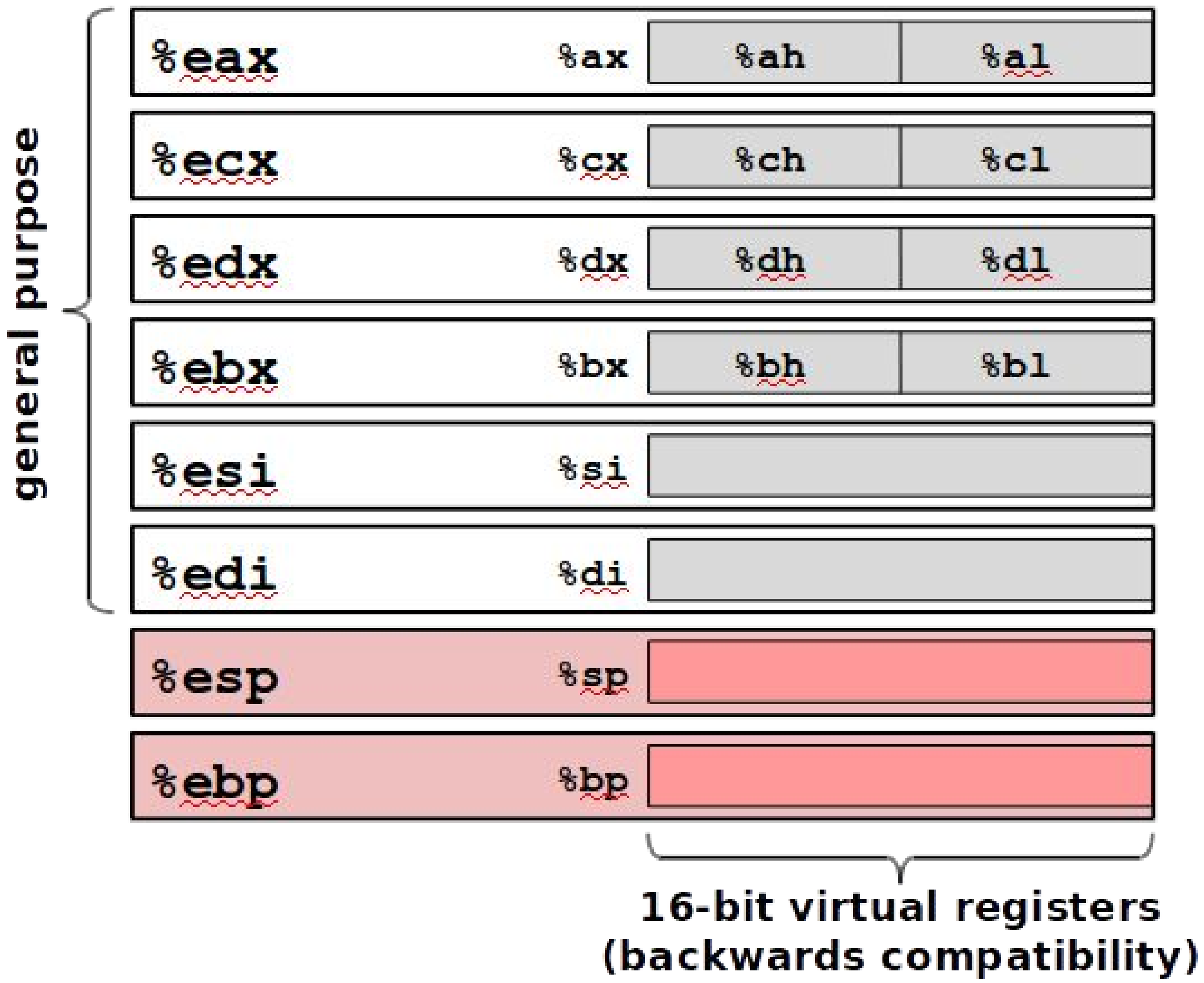
Assembly Basics: Registers, Operands, Move

x86-64 Integer Registers



Can reference low-order 4 bytes (also low-order 1 & 2 bytes), see p. 180 in the book

IA32 Registers (History)



Origin (mostly obsolete)

accumulate

counter

data

base

source index

destination index

stack pointer

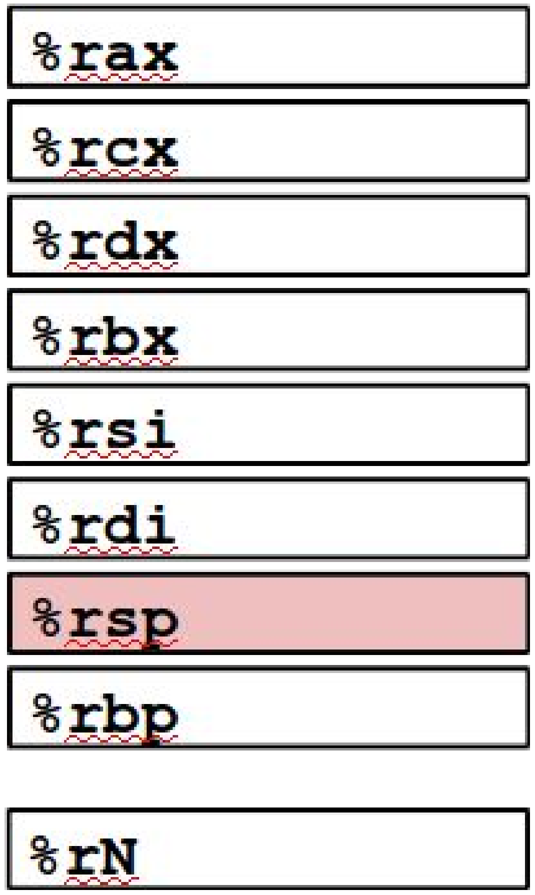
base pointer

Moving Data

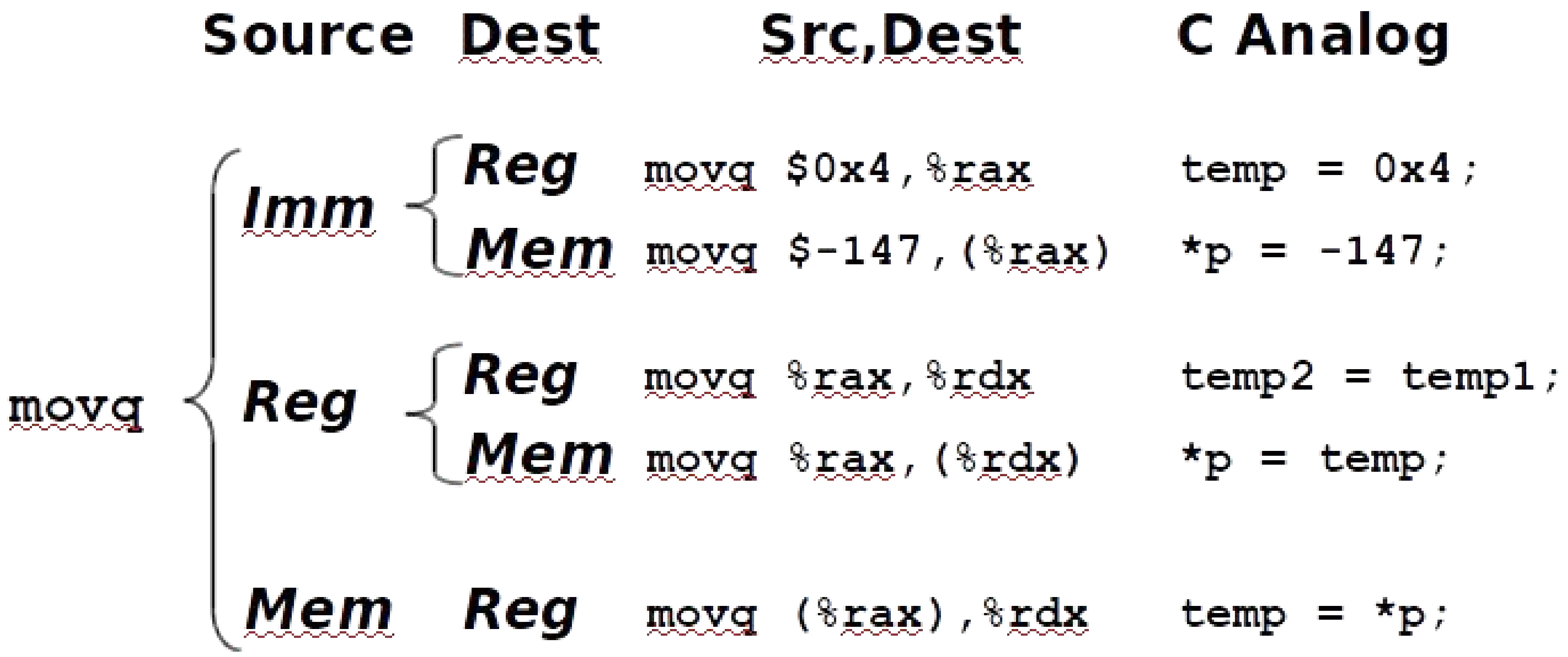
- Moving Data

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



`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

```
movq 8(%rbp),%rdx
```

Mem - think of as a memory array: Mem[address] means value stores at the particular memory address.

Reg - think of as a register array: Reg[reg_name] means value stored at the particular register

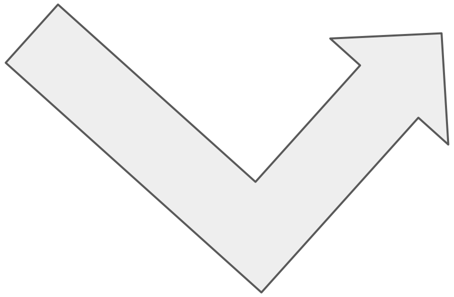
Note: the normal mode is a special case of displacement mode in which D = 0

Simple Addressing Modes - `swap()` Examples

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

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

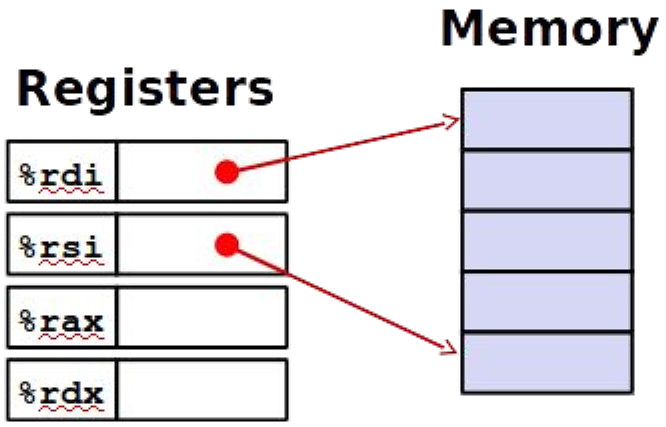
`swap.c` **`swap.s`**



`gcc -S -Og swap.c`

Understanding `swap()`

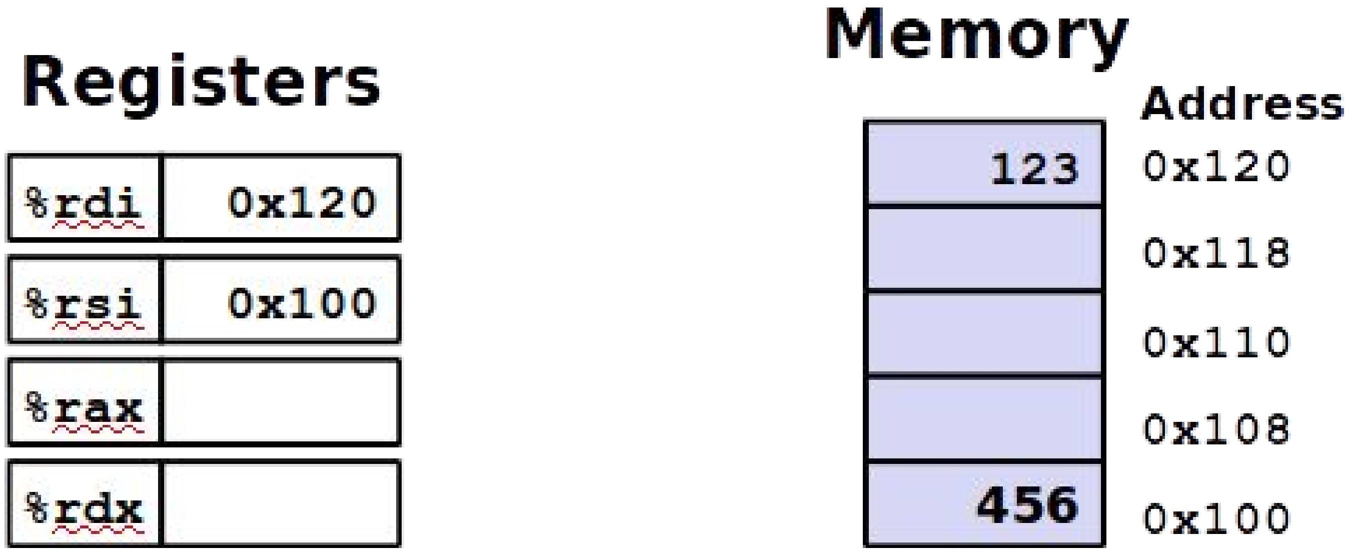
```
void swap  (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



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

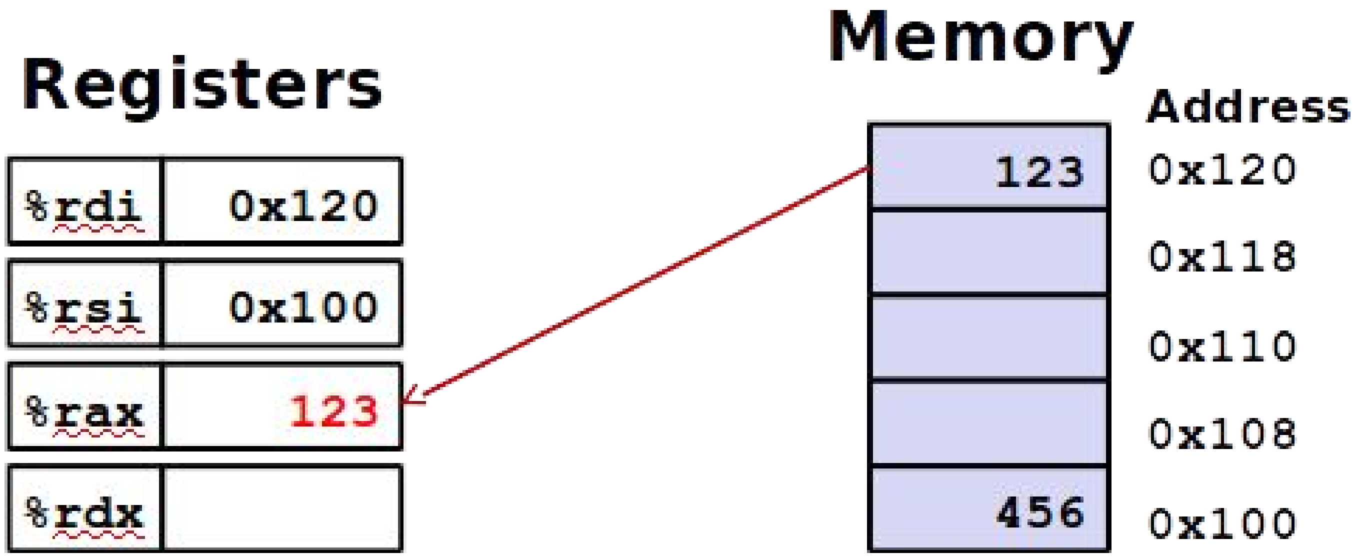
```
swap:
   movq     (%rdi), %rax
   movq     (%rsi), %rdx
   movq     %rdx, (%rdi)
   movq     %rax, (%rsi)
   ret
```

Understanding `swap()`



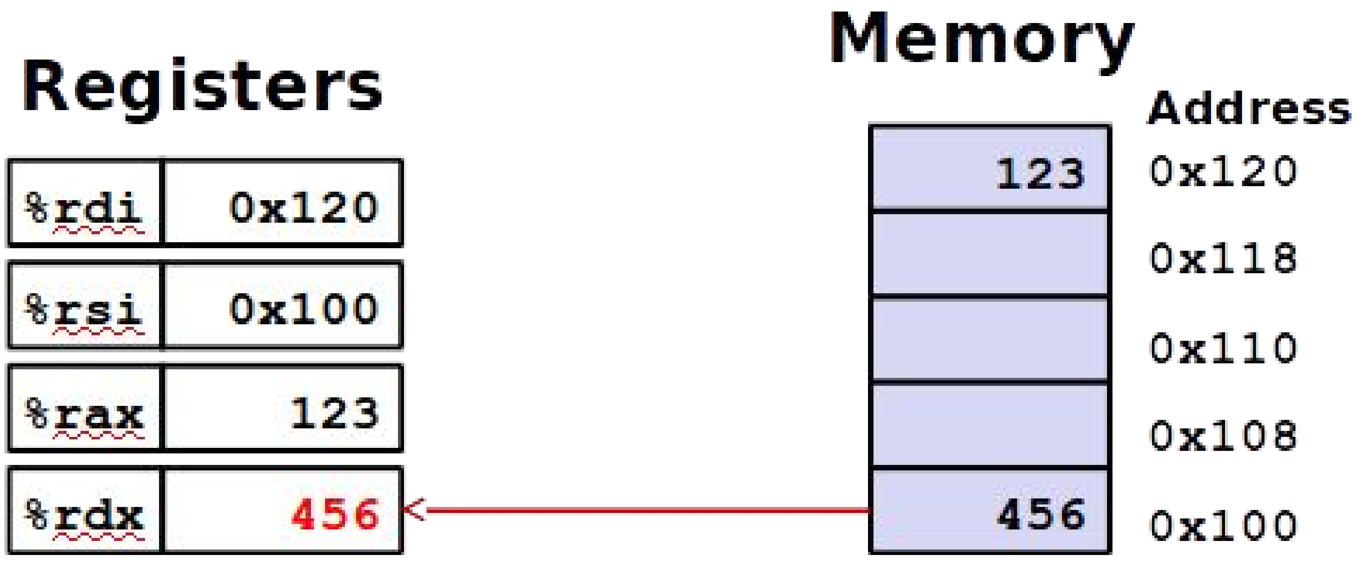
```
swap:
   movq    (%rdi), %rax     # t0 = *xp
   movq    (%rsi), %rdx     # t1 = *yp
   movq    %rdx, (%rdi)     # *xp = t1
   movq    %rax, (%rsi)     # *yp = t0
   ret
```

Understanding `swap()`



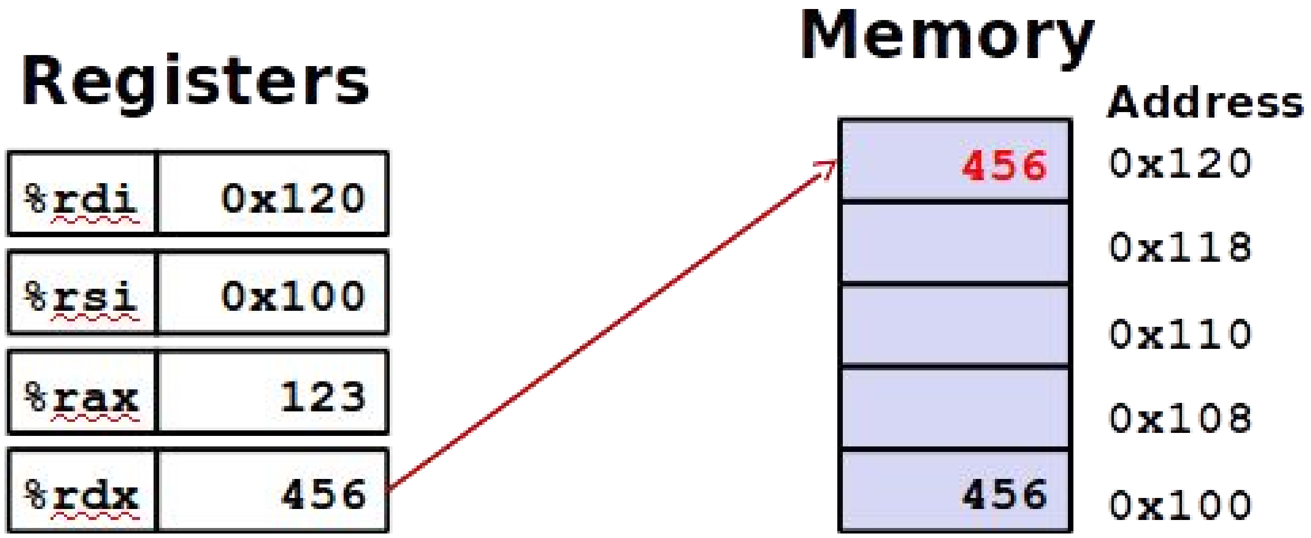
```
swap:
   movq     (%rdi), %rax     # t0 = *xp
   movq     (%rsi), %rdx     # t1 = *yp
   movq     %rdx, (%rdi)     # *xp = t1
   movq     %rax, (%rsi)     # *yp = t0
   ret
```

Understanding `swap()`



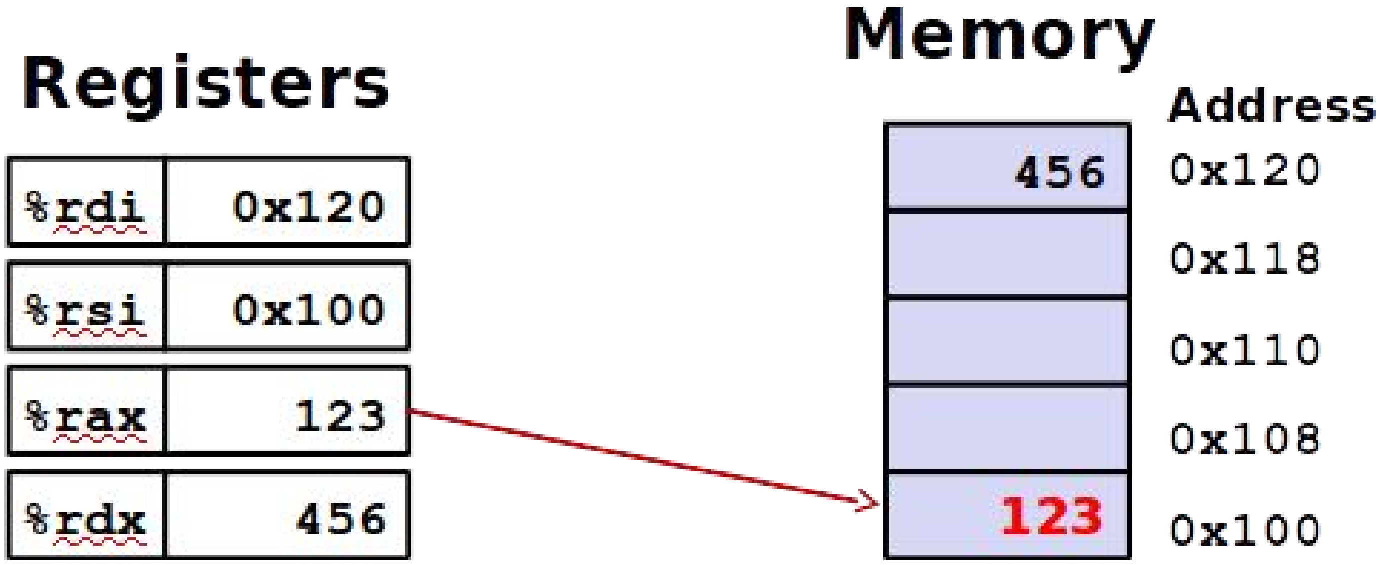
```
swap:
   movq     (%rdi), %rax     # t0 = *xp
   movq     (%rsi), %rdx     # t1 = *yp
   movq     %rdx, (%rdi)     # *xp = t1
   movq     %rax, (%rsi)     # *yp = t0
   ret
```

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
```

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

```
movq 8(%rbp),%rdx
```

Mem - think of as a memory array: Mem[address] means value stores at the particular memory address.

Reg - think of as a register array: Reg[reg_name] means value stored at the particular register

Note: the normal mode is a special case of displacement mode in which D = 0

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

Logical and Arithmetic Operations

Address Computation Instruction

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

```
file: leaq_example.c

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

create object code using
`gcc -Og -S leaq_example.c`

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

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?)
- See page 192 in the book

Arithmetic Operations

- One Operand Instructions:

Format	Computation
`incq Dest`	`Dest = Dest + 1`
`decq Dest`	`Dest = Dest - 1`
`negq Dest`	`Dest = -Dest`
`notq Dest`	`Dest = ~Dest`

- Watch out for argument order!
- No distinction between signed and unsigned int (why?)
- See page 192 in the book

Example: arithmetic expression

```c
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), %rcx
    salq $4, %rcx
    leaq 4(%rdi,%rcx), %rcx
    imulq %rcx, %rax
    ret
```

Example: arithmetic expression

```
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), %rcx
    salq  $4, %rcx             #t4
    leaq  4(%rdi,%rcx), %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`