X86-64: Data Access and Operations

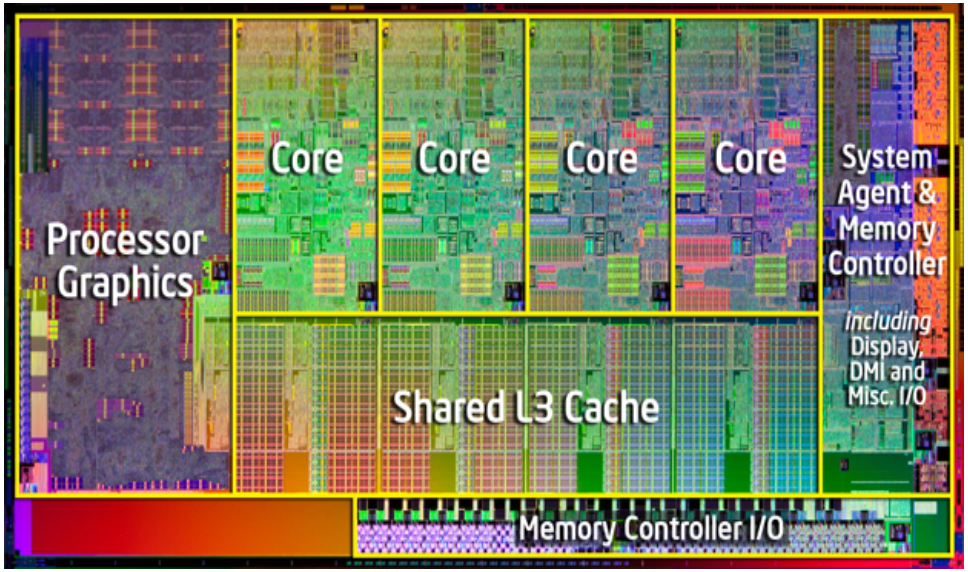
Instruction Set Architecture (ISA)

Intel 8086

Intel Core i7

A very complex ISA

We will look at a subset of it



2015 State of the Art

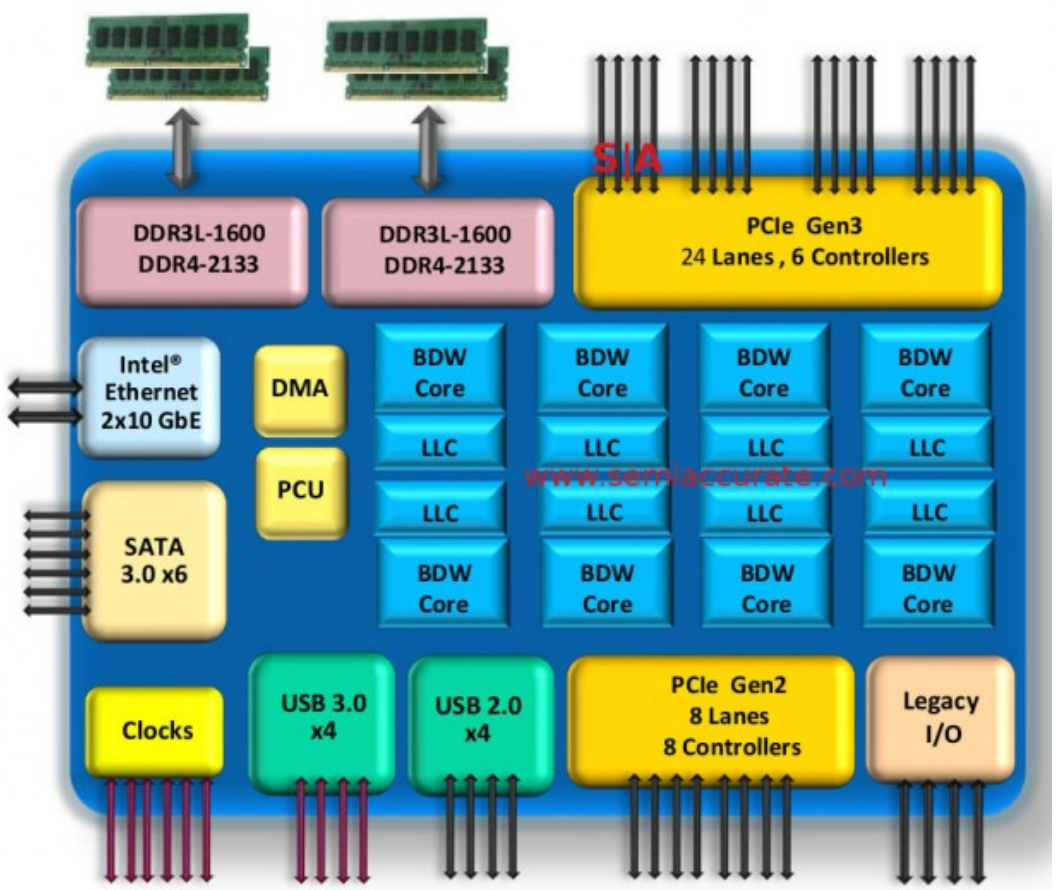
Core i7 Broadwell

Desktop Model

- 4 cores
- Integrated graphics
- 3.3-3.8 GHz
- 65W

Server Model

- 8 cores
- Integrated I/O
- 2-2.6 GHz
- 45W



Instruction Set Architecture (ISA)

How is data represented?

- Previous lectures

How are programs represented?

- Instruction Set Architecture (ISA)
- Encoding is architecture dependent

x86-64

- Evolutionary design starting in 1978
- Many features added over time
  - 8086 → 286 → i386 → Pentium → Core / misc
    - 8 bit → 16 bit → 32 bit → 64 bit
- Backward compatibility → Ugly “legacy baggage”

Complex Instruction Set Computer (CISC)

- Many different instructions with many different formats

How did we program computers?

Initially, no compilers or assemblers

Machine code generated by hand!

- **Error-prone**
- **Time-consuming**
- **Hard to read and write**
- **Hard to debug**

Address	Instruction
0000 0000	0000 1111
0000 0001	1000 0101
0000 0010	0011 1000
0000 0011	0000 1110
0000 0100	1000 0101
0000 0101	0011 0000
0000 0110	1011 0000
0000 0111	0000 1110
0000 1000	1000 0000
0000 1001	0010 0000
0000 1010	0000 1011
0000 1011	0001 0111
0000 1100	1000 1000
0000 1101	0001 0100
0000 1110	1110 1000
0000 1111	0000 0000
0001 0000	0001 0100
0001 0001	0000 1010
0001 0010	1000 0001
0001 0011	0001 0000
0001 0100	1010 1110

An ADD *instruction*

An Assembly Code Program

```
MemoryZero:
            load    [r15+4],r1          ! r1 = arg1 (p)
            load    [r15+8],r2          ! r2 = arg2 (byteCount)
mzLoop1:                                ! LOOP:
            cmp     r2,0                !   if byteCount <= 0 exit
            ble     mzLoop2Test         !   .
            and     r1,0x00000003,r3    !   tmp = p % 4
            cmp     r3,0                !   if tmp == 0 exit
            be      mzLoop2Test         !   .
            storeb  r0,[r1]             !   *p = 0x00
            add     r1,1,r1             !   p = p + 1
            sub     r2,1,r2             !   byteCount = byteCount - 1
            jmp     mzLoop1             ! ENDLOOP
```

assembly code

comments

(Not Intel)

The Assembler Tool

```
000e34           MemoryZero:
000e34 8b1f0004             load   [r15+4],r1          ! r1 = arg1 (p)
000e38 8b2f0008             load   [r15+8],r2          ! r2 = arg2 (byteCount)
000e3c           mzLoop1:                              ! LOOP:
000e3c 81020000             cmp    r2,0                !   if byteCount <= 0 exit
000e40 a500002c             ble    mzLoop2Test         !   .
000e44 88310003             and    r1,0x00000003,r3    !   tmp = p % 4
000e48 81030000             cmp    r3,0                !   if tmp == 0 exit
000e4c a2000020             be     mzLoop2Test         !   .
000e50 70010000             storeb r0,[r1]             !   *p = 0x00
000e54 80110001             add    r1,1,r1             !   p = p + 1
000e58 81220001             sub    r2,1,r2             !   byteCount = byteCount - 1
000e5c a1ffffe0             jmp    mzLoop1             ! ENDLOOP
```

memory address

machine code

assembly code

comments

(Not Intel)

Assemblers

Assign mnemonics to machine code

- Use assembly language for specifying machine instructions
- Names for the machine instructions and registers

```
movq %rax,%rcx
```

- There is no standard for x86 assemblers
  - Intel assembly language
  - AT&T assembler
  - GNU uses AT&T style with its assembler `gas`

Even with the advent of compilers, assembly still used

- Early compilers made big, slow code
- Operating Systems were written mostly in assembly
  into the 1980s
- Accessing new hardware features before compiler has a chance to incorporate them

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 clock frequency.

Code Forms:

Machine Code: The byte-level program that a processor executes

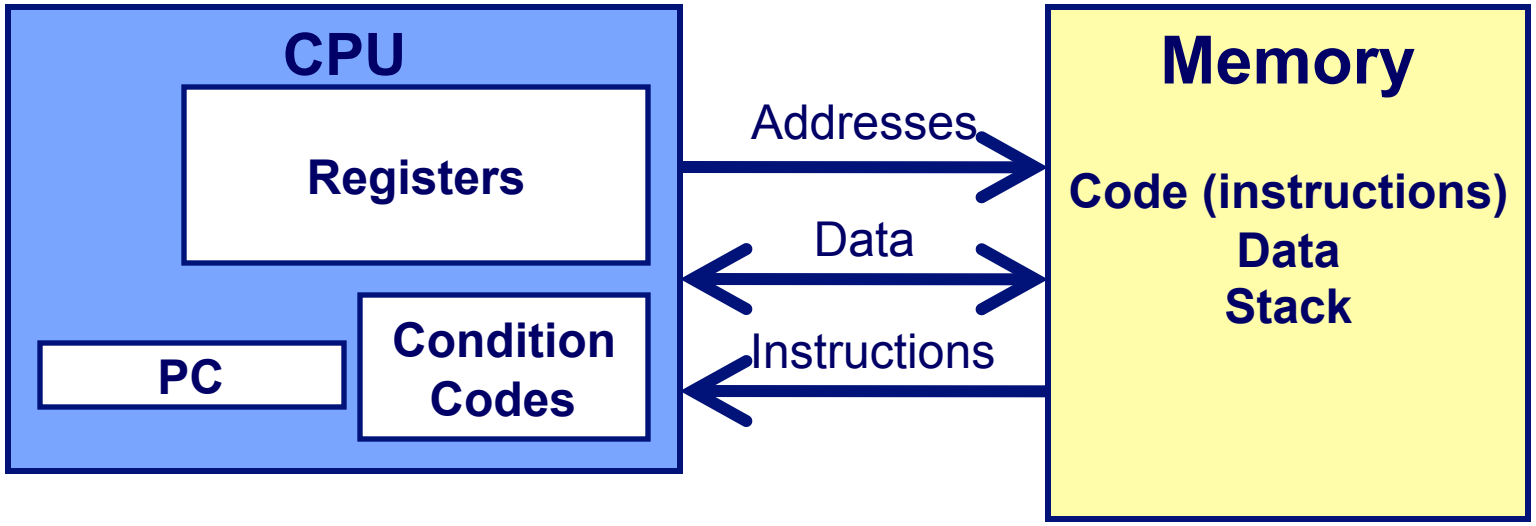
Assembly Code: A text representation of machine code

Example ISAs:

Intel: x86, IA32, Itanium, x86-64

ARM: Used in almost all mobile phones

Assembly Programmer's View



CPU (Programmer-Visible State)

- The Program Counter (PC)
  - RIP (Register Instruction Pointer)
    - The address of next instruction
- Register File
  - Heavily used program data
- Condition Codes
  - Store status information about most recent arithmetic operation
  - Used for conditional branching

Memory

- Byte addressable array
- Code, user data, OS data
- Includes stack used to support procedures

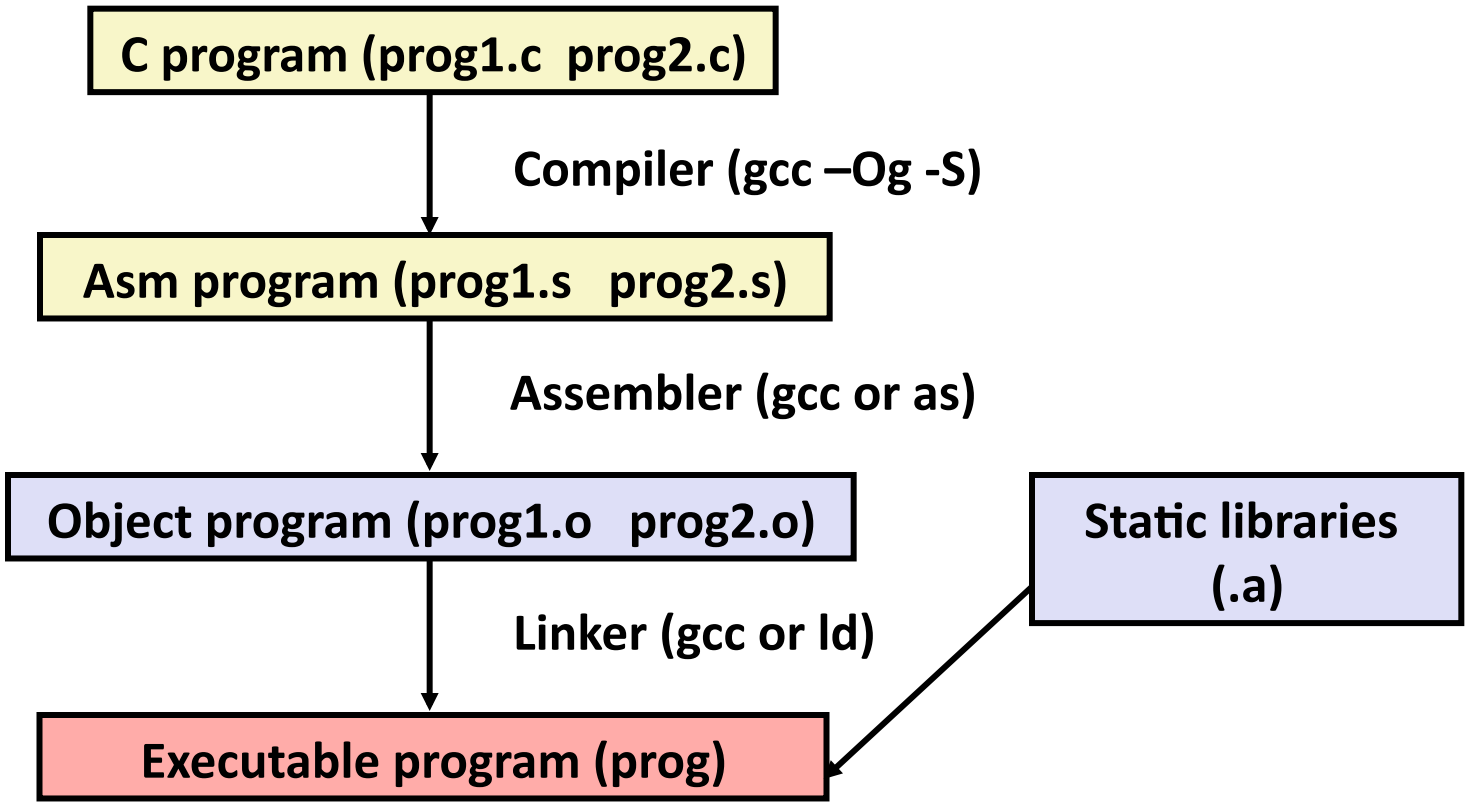
Turning C into Object Code

Code in files `prog1.c prog2.c`

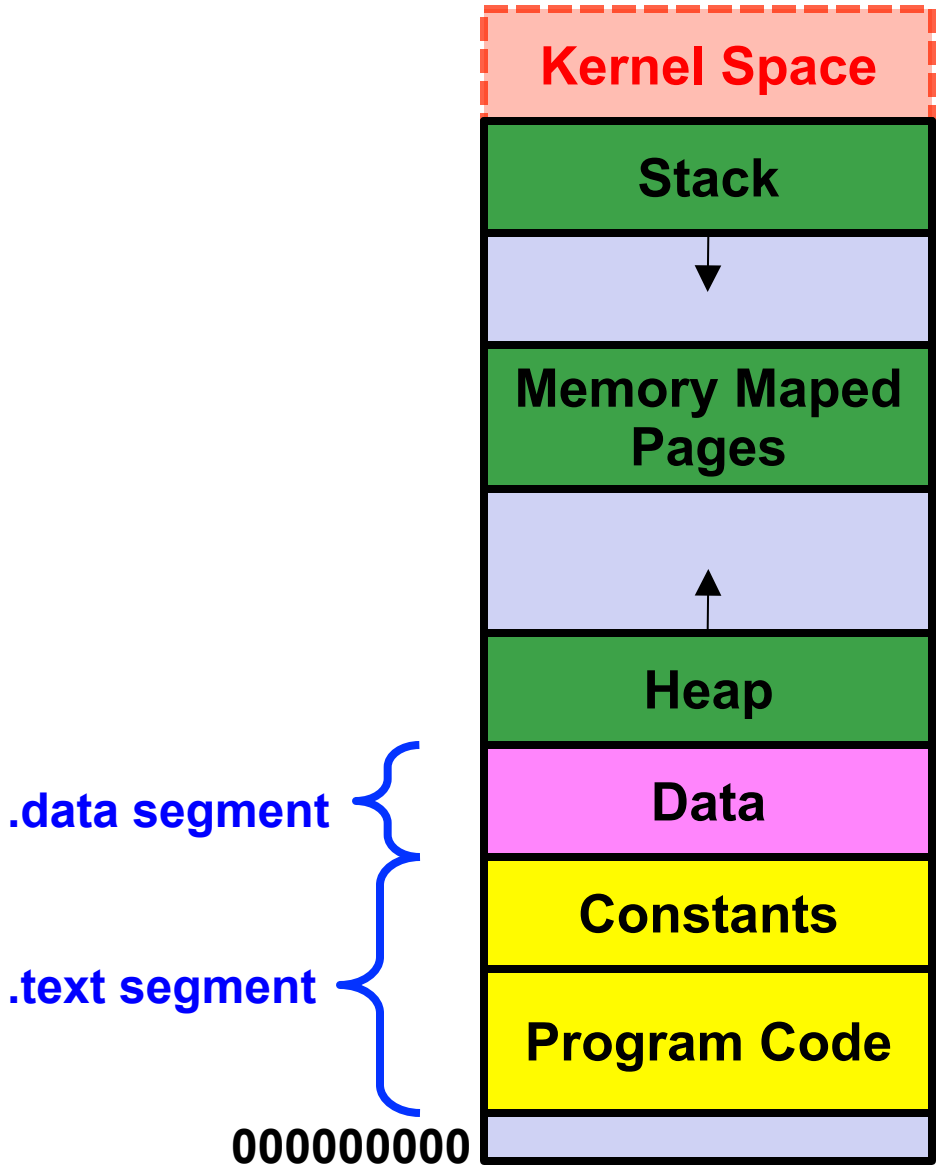
Compile with command: `gcc –Og prog1.c prog2.c -o prog`

(`-Og` is Nnew to recent versions of GCC)

Put resulting binary in file `prog`



Virtual Address Space



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
```

To compile:

`gcc -Og -S sum.c`

Produces file `sum.s`

Warning:

Will get different results on different machines

(Linux, Mac OS-X, …)

Different versions of gcc and different compiler settings.

Object Code

What goes into memory:

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

- Total of 14 bytes
- Each instruction 1, 3, or 5 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

C Code

```
*dest = t;
```

Store value `t` where designated by `dest`

Assembly

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

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

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

3-byte instruction

Stored at address `0x40059e`

Disassembling Object Code

`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

Output from objdump:

```
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
```

Disassembling Object Code

`objdump -d sum`

- Useful tool for examining
- Analyzes bit pattern of ser
- Produces approximate ren
- Can be run on either `a.ou`

Original Assembly Code:

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

Output from objdump:

```
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
```

Disassembly within gdb

"x" Output:

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

Within gdb

```
gdb sum
```

Disassemble command

```
disass sumstore
```

Examine the 14 bytes starting at `sumstore`

```
x/14xb sumstore
```

"disass" Output:

```
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
```

Disassembly within gdb

"x" Output:

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

Within gdb

```
gdb sum
```

Disassemble

```
disass s
```

Examine the

```
x/14xb sumstore
```

Original Assembly Code:

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

"disass" Output:

```
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
```

What Can be Disassembled?

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

What Can be Disassembled?

Anything that can be interpreted as executable code

Disassembler examines bytes and reconstructs assembly source

```
% 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

Registers

Located Directly on the CPU

- Instructions operate on data in registers
- Used to store frequently-used data
- Very fast
- Not much storage capacity
  - 16 registers, 64-bits each
- Different from main memory
  - Main memory: e.g., 2 billion × 8 bits

CPU does not directly operate on data in memory

- Data needs to be loaded into registers for CPU
- Once in a register, processor can operate on it
- Typically, data is loaded into registers, manipulated or used, and then written back to memory

Instruction Suffixes

C Type	Size in bits	Instruction Suffix	Also Used
char	8	**b**	byte
short	16	**w** (word)	halfword
int	32	**l** (longword)	word
long long	64	**q** (quadword)	doubleword

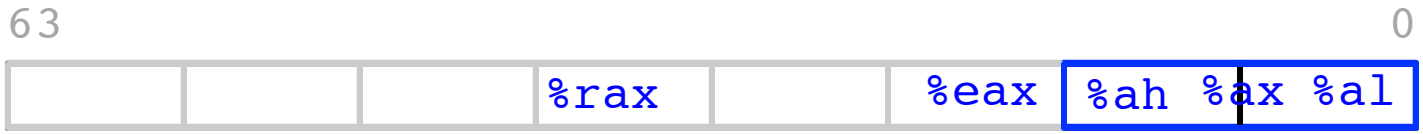
```
movq   %rax,…      copy 64 bits
movl   %eax,…      copy 32 bits
movw   %ax,…       copy 16 bits
movb   %al,…       copy 8 bits
movb   %ah,…       copy 8 bits
```

x86-64 Integer Registers (8 bytes)

%rax	%r8
%rbx	%r9
%rcx	%r10
%rdx	%r11
%rsi	%r12
%rdi	%r13
%rsp	%r14
%rbp	%r15

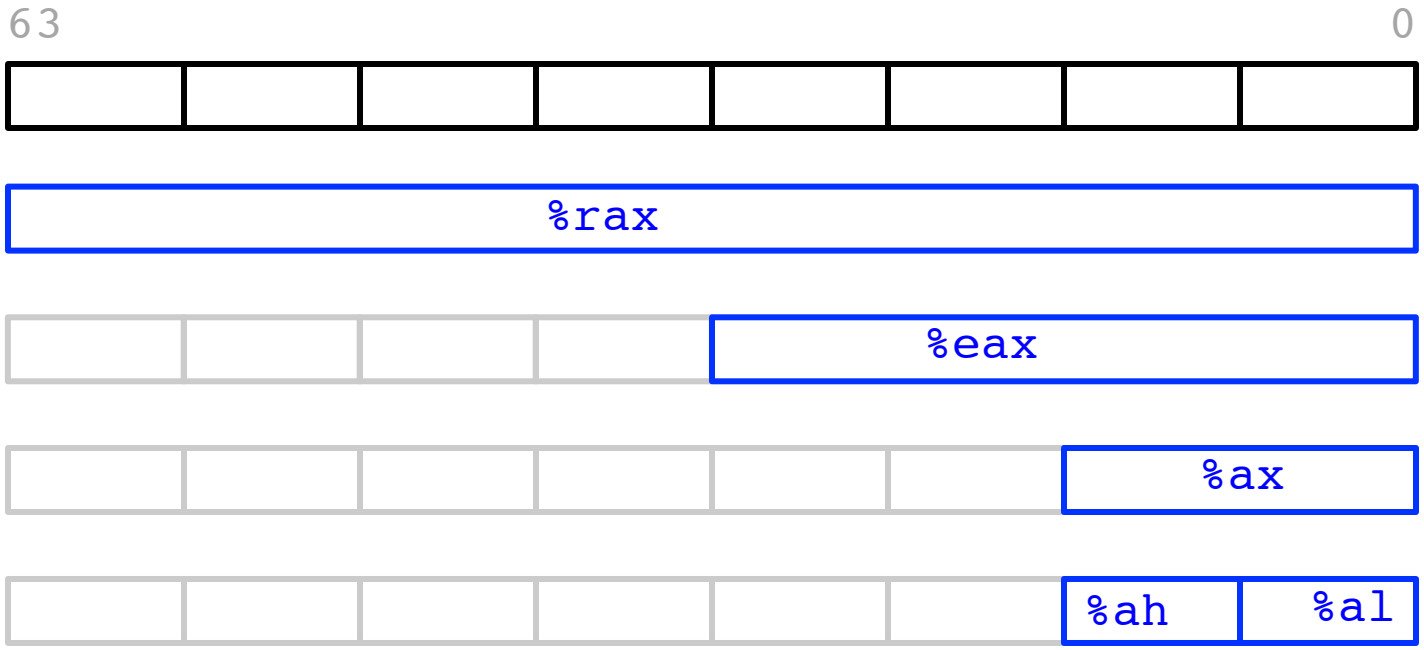
Can also reference low-order 1, 2, and 4 bytes

%rax




```
movq   %rax,…      copy 64 bits
movl   %eax,…      copy 32 bits
movw   %ax,…       copy 16 bits
movb   %al,…       copy 8 bits
movb   %ah,…       copy 8 bits
```

%rax



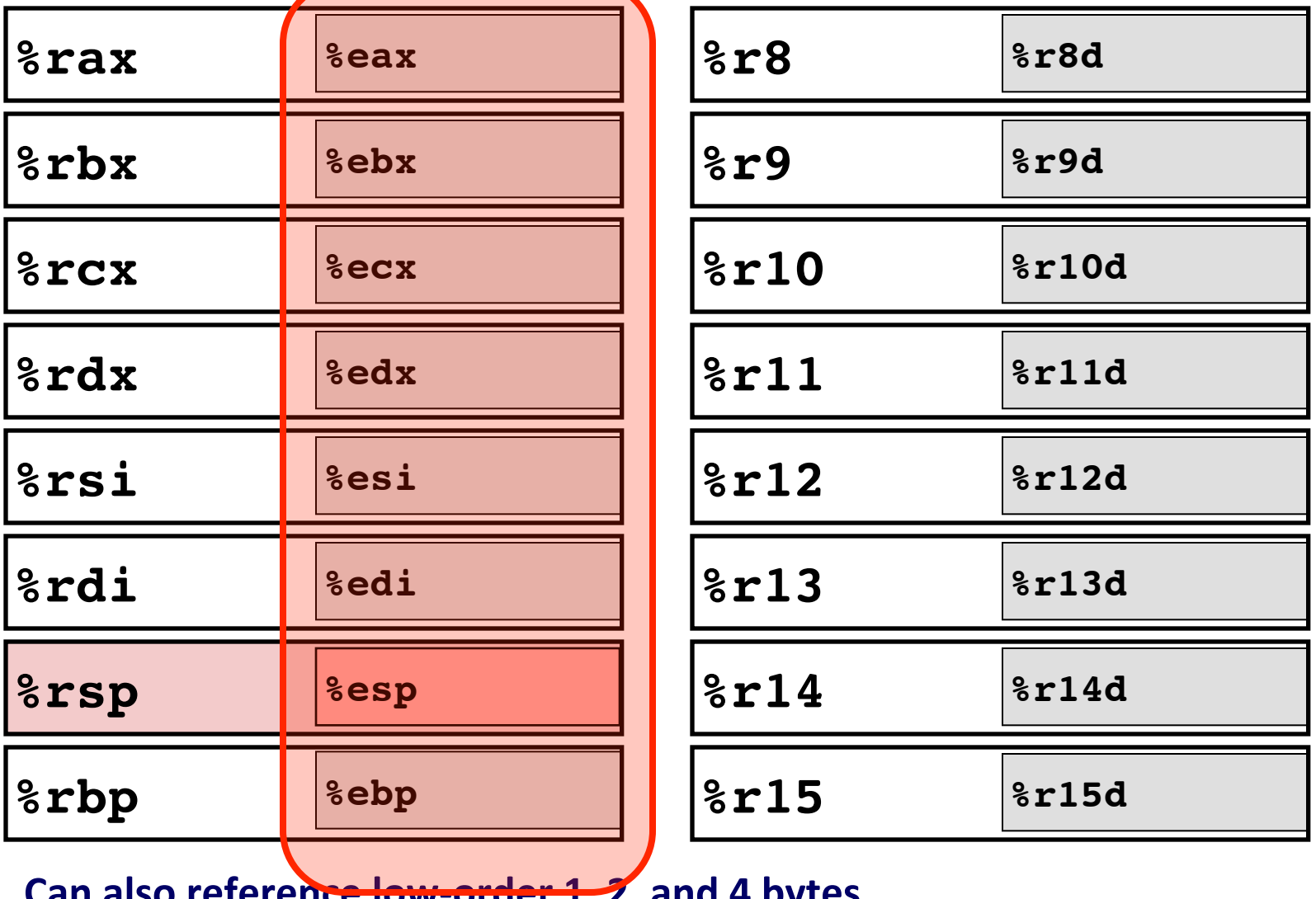

```
movq   %rax,…      copy 64 bits
movl   %eax,…      copy 32 bits
movw   %ax,…       copy 16 bits
movb   %al,…       copy 8 bits
movb   %ah,…       copy 8 bits
```

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

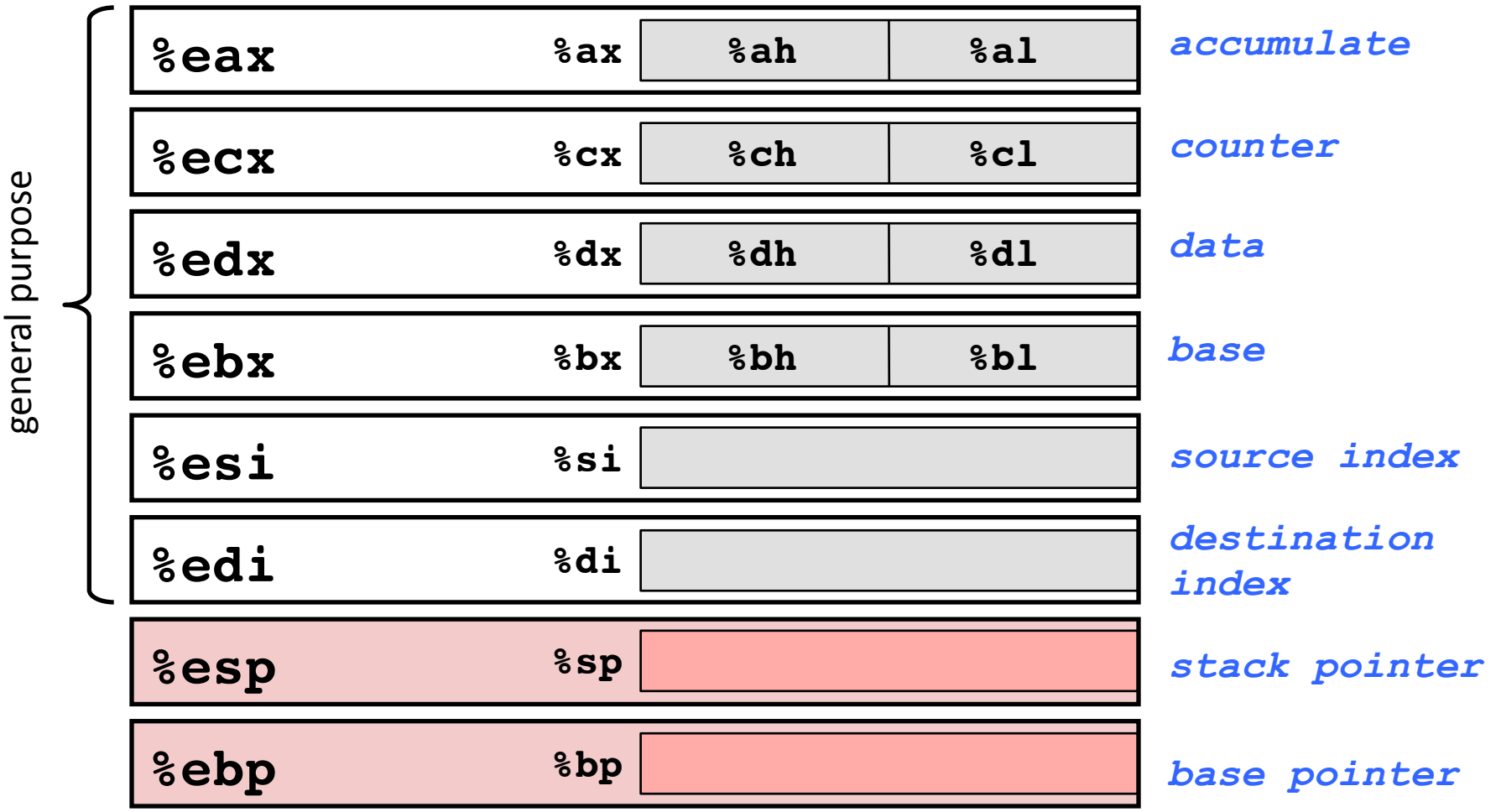
Can also reference low-order 1, 2, and 4 bytes

x86-64 Integer Registers



Can also reference low-order 1, 2, and 4 bytes

Some History: IA32 Registers



Register Naming - History

Originally

`%a, %b, %c, %d`

`%sp` "stack pointer"

`%bp` "base pointer"

16-bit Registers

`ax` The 'x' means "extended" (from 8 bits)

8-bit Registers

`al` The 'l' means "low-order byte"

`ah` The 'h' means "high-order byte"

32-bit Registers

`eax` The 'e' means "extended" (from 16 bits)

64-bit Registers

`rax`

Moving Data

The "mov" Instruction

`movq` *Source, Destination*

Operand Types

Immediate: Constant data

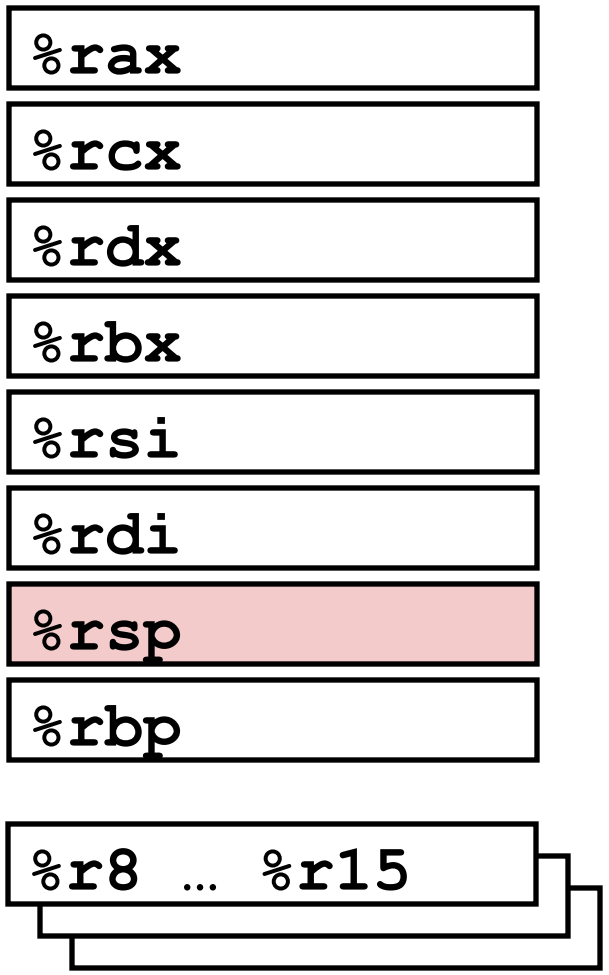
- Example: `$0x400`, `$-533`
- Like C constant, but prefixed with `'$'`
- Encoded with 1, 2, or 4 bytes

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 (reg contains address)

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



Operand Combinations for "mov"

	Source	Dest	Example	C Analog
movq	*Imm*	*Reg*	`movq $0x4,%rax`	`x = 0x4;`
		Mem	`movq $-147,(%rax)`	`*p = -147;`
	Reg	*Reg*	`movq %rax,%rdx`	`x = y;`
		Mem	`movq %rax,(%rdx)`	`*p = x;`
	Mem	*Reg*	`movq (%rax),%rdx`	`x = *p;`

Cannot do memory-to-memory transfer with a single instruction

Instruction Variations

A typical instruction acts on 2 or more *operands* of a particular width

```
addl    %ecx,%edx
```

Adds the contents of %ecx to %edx

```
addq    %rcx,%rdx
```

Adds the contents of %rcx to %rdx

Size of the operand denoted in instruction

Why "long word" for 32-bit registers? Baggage from 16-bit processors

So we have these instructions:

addb: 8 bits = byte

addw: 16 bits = word !!!

addl: 32 bits = double or long word !!!

addq: 64 bits = quad word

Immediate mode

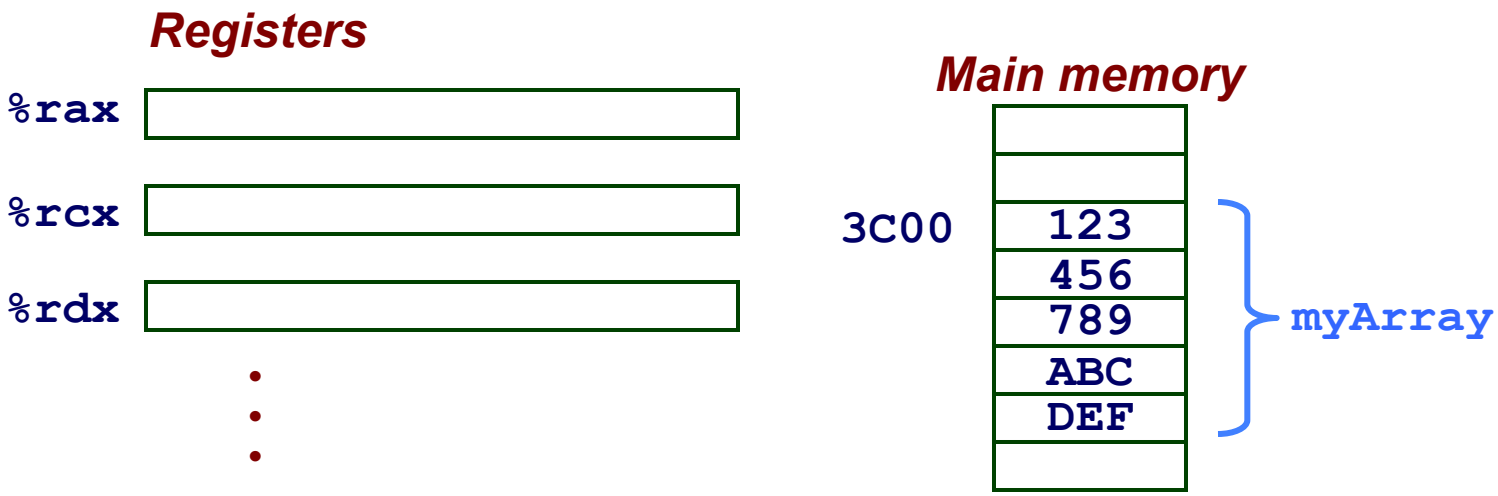
Immediate has only one mode

Form: `$ImmediateValue`

Examples:

```
movl    $0x8000,%eax
movq    $myArray,%rax
```

```
int myArray[5];    /* global variable stored at 0x3C00 */
```



Immediate mode

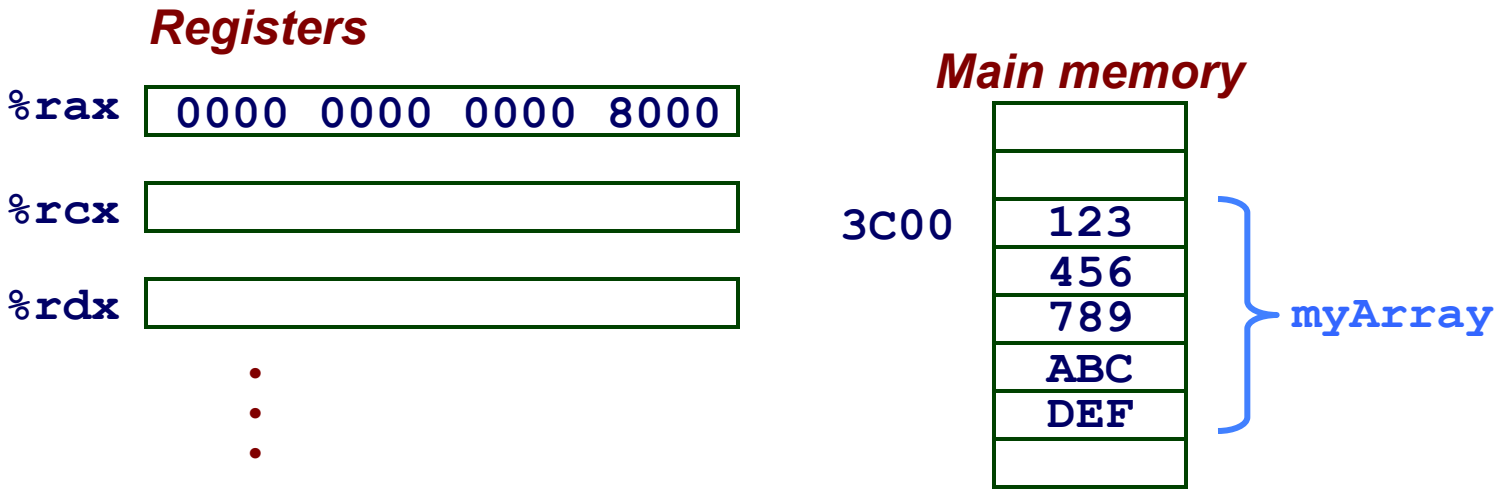
Immediate has only one mode

Form: `$ImmediateValue`

Examples:

```
movl    $0x8000,%eax
movq    $myArray,%rax
```

```
int myArray[5];    /* global variable stored at 0x3C00 */
```



Immediate mode

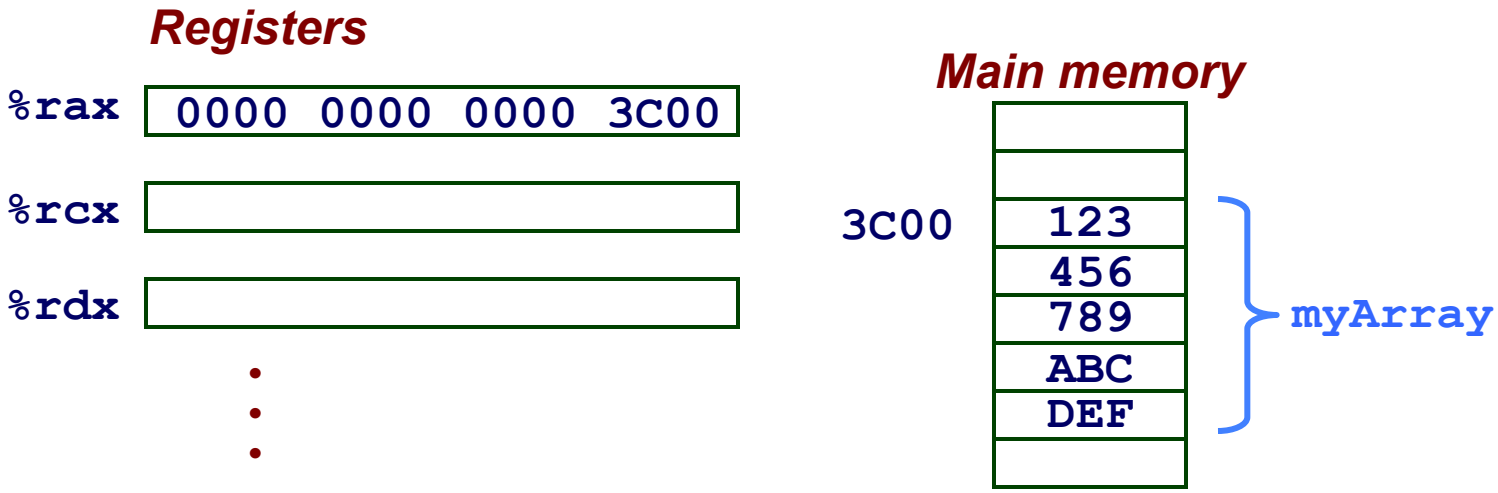
Immediate has only one mode

Form: `$ImmediateValue`

Examples:

```
movl    $0x8000,%eax
movq    $myArray,%rax
```

```
int myArray[5];    /* global variable stored at 0x3C00 */
```



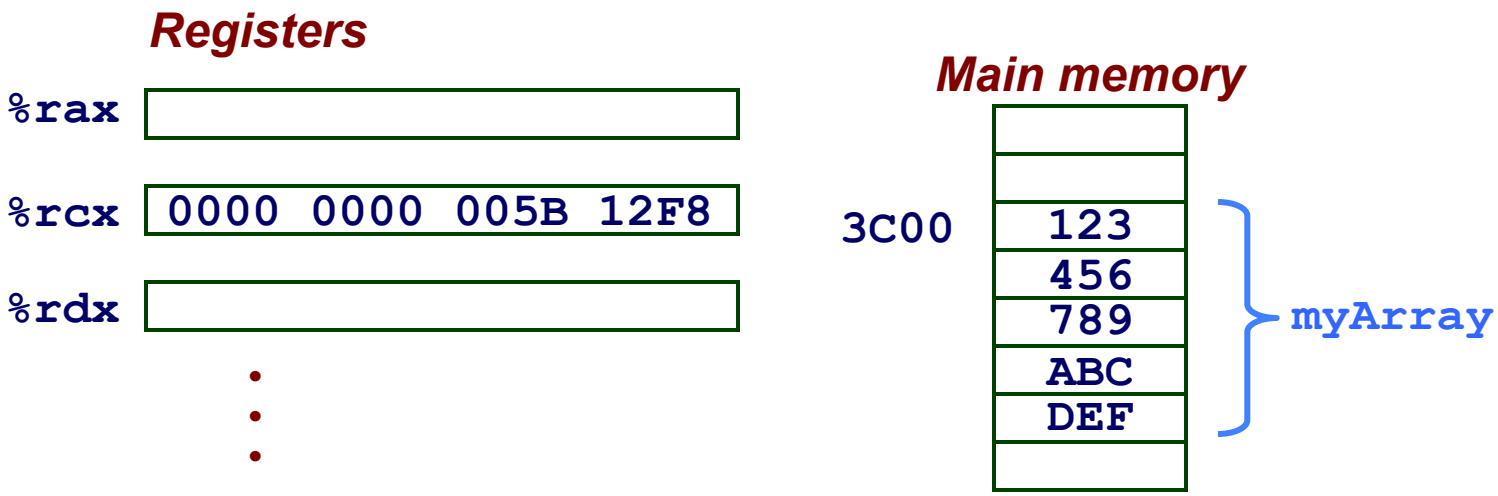
Register mode

Register has only one mode

Form: E_a

Operand value: $R[E_a]$

```
movq    %rcx,%rax
```



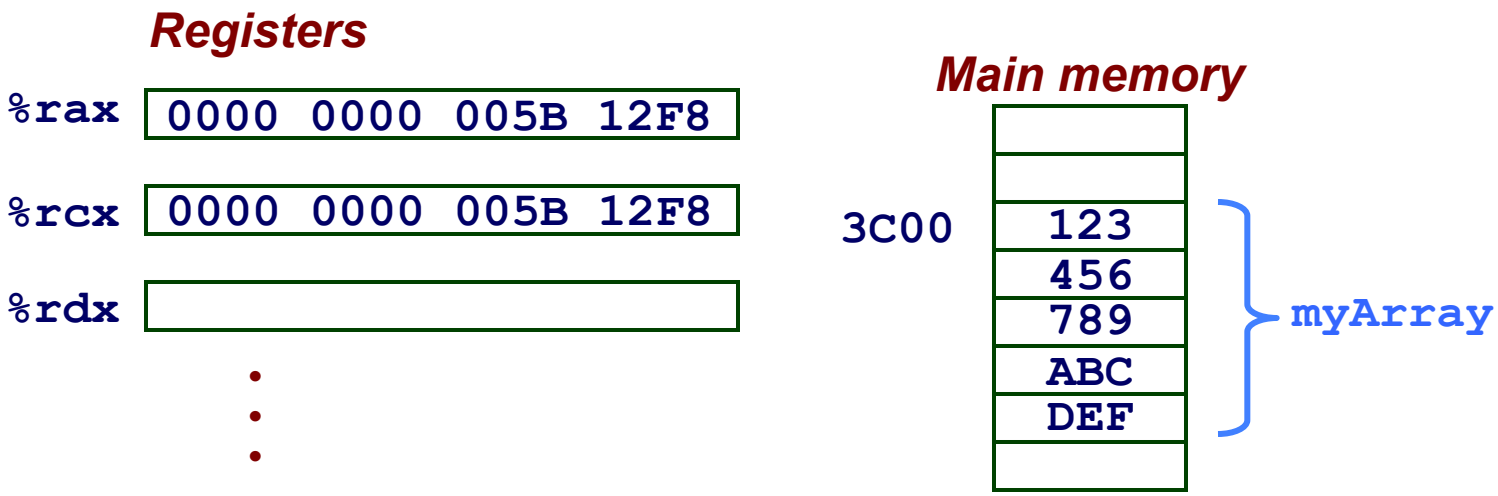
Register mode

Register has only one mode

Form: E_a

Operand value: $R[E_a]$

```
movq    %rcx,%rax
```



Memory Modes

Absolute

Specify the address of the data

```
movl    %eax,myArray
```

Indirect

Use register to calculate address

```
movl    %eax,(%rbx)
```

Base + displacement

Use register plus absolute address to calculate address

```
movl    %eax,32(%rbx)
```

Indexed

Indexed: Add contents of an index register

```
movl    %eax,32(%rbx,%rcx)
```

Scaled index: Add contents of an index register scaled by a constant

```
movl    %eax,32(%rbx,%rcx,4)
```

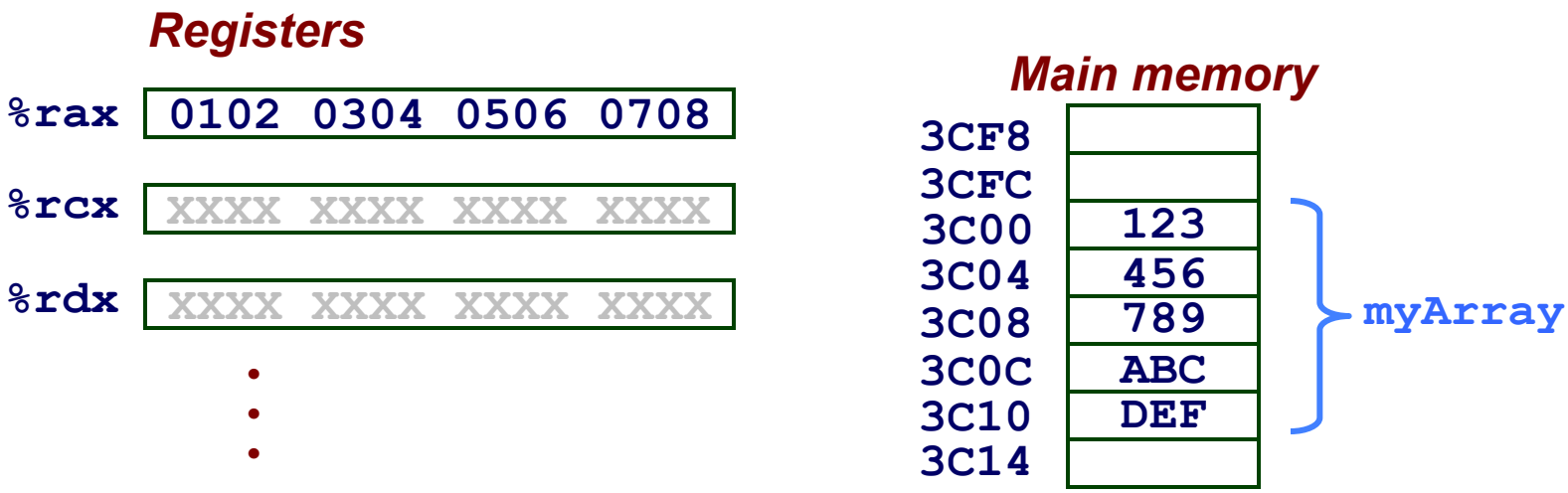
Absolute Memory Mode

Form: `Imm`

Operand value: `M[Imm]`

```
movl 0x3C04,%eax
movl myArray,%edx
```

```
int myArray[30];   /* global variable stored at 0x3C00 */
```



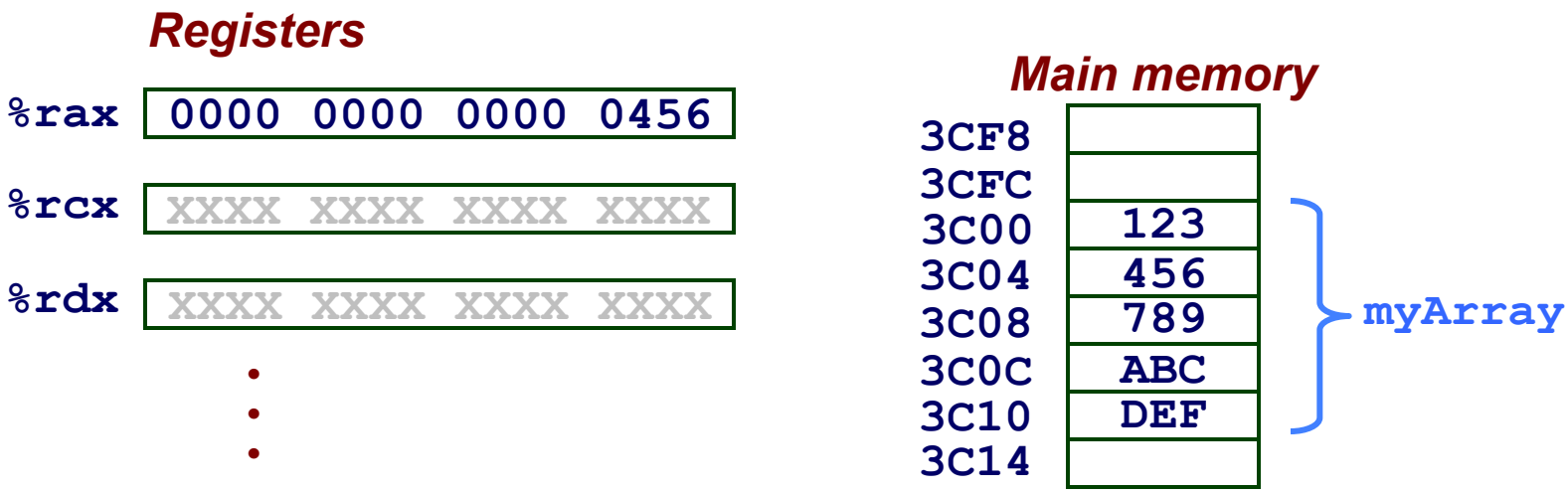
Absolute Memory Mode

Form: `Imm`

Operand value: `M[Imm]`

```
movl 0x3C04,%eax
movl myArray,%edx
```

```
int myArray[30];   /* global variable stored at 0x3C00 */
```



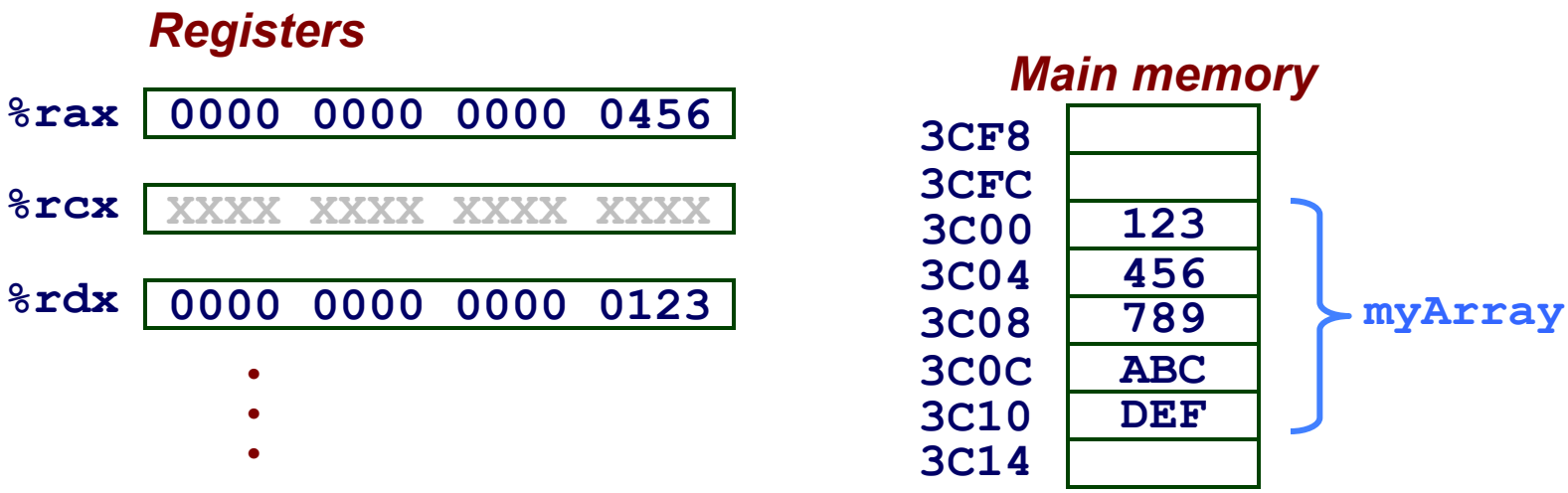
Absolute Memory Mode

Form: `Imm`

Operand value: `M[Imm]`

```
movl 0x3C04,%eax
movl myArray,%edx
```

```
int myArray[30];   /* global variable stored at 0x3C00 */
```



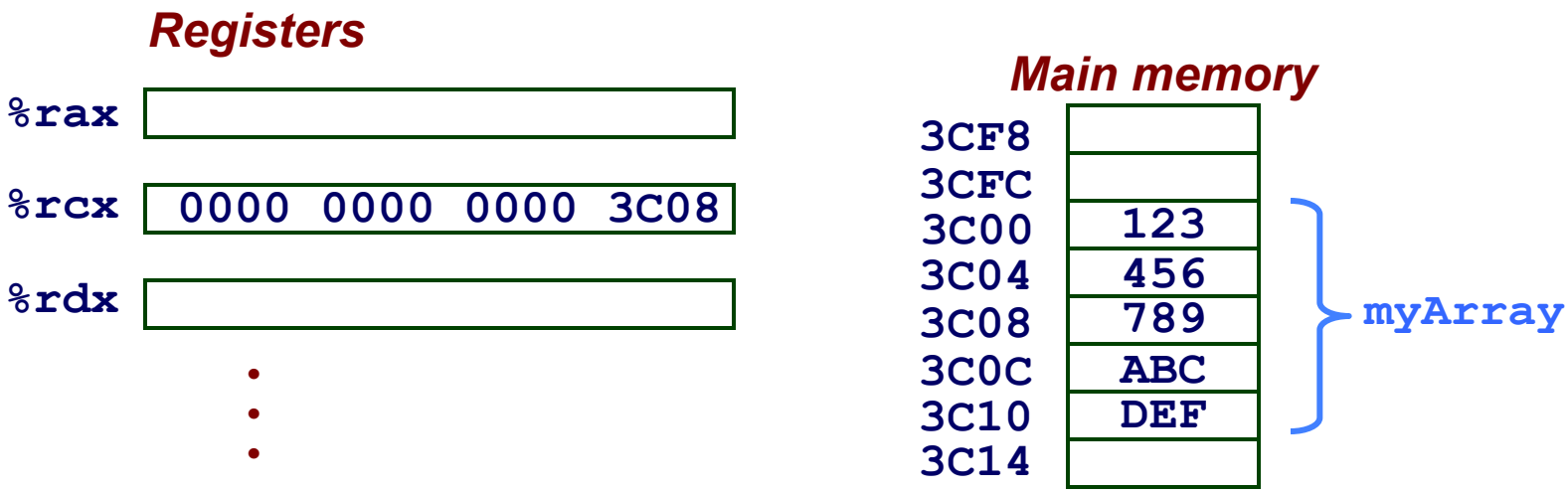
Indirect Memory Mode

Form: (E_a)

Operand value: $M[R[E_a]]$

Register E_a specifies the memory address

```
movl (%rcx),%eax
```



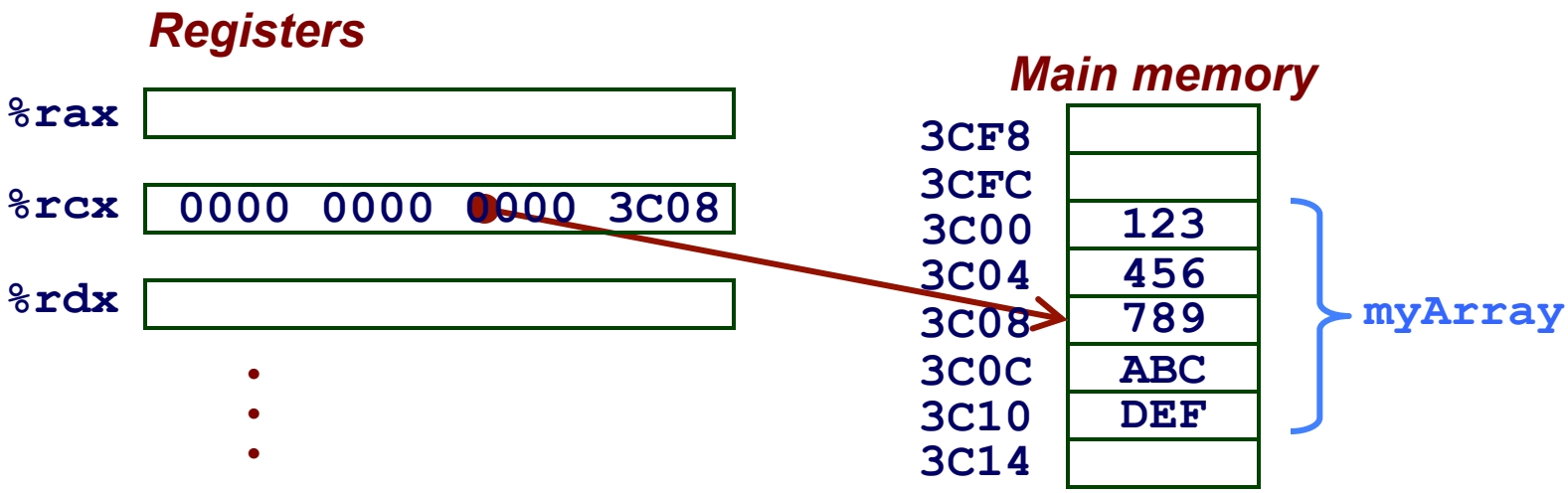
Indirect Memory Mode

Form: (E_a)

Operand value: $M[R[E_a]]$

Register E_a specifies the memory address

```
movl (%rcx),%eax
```



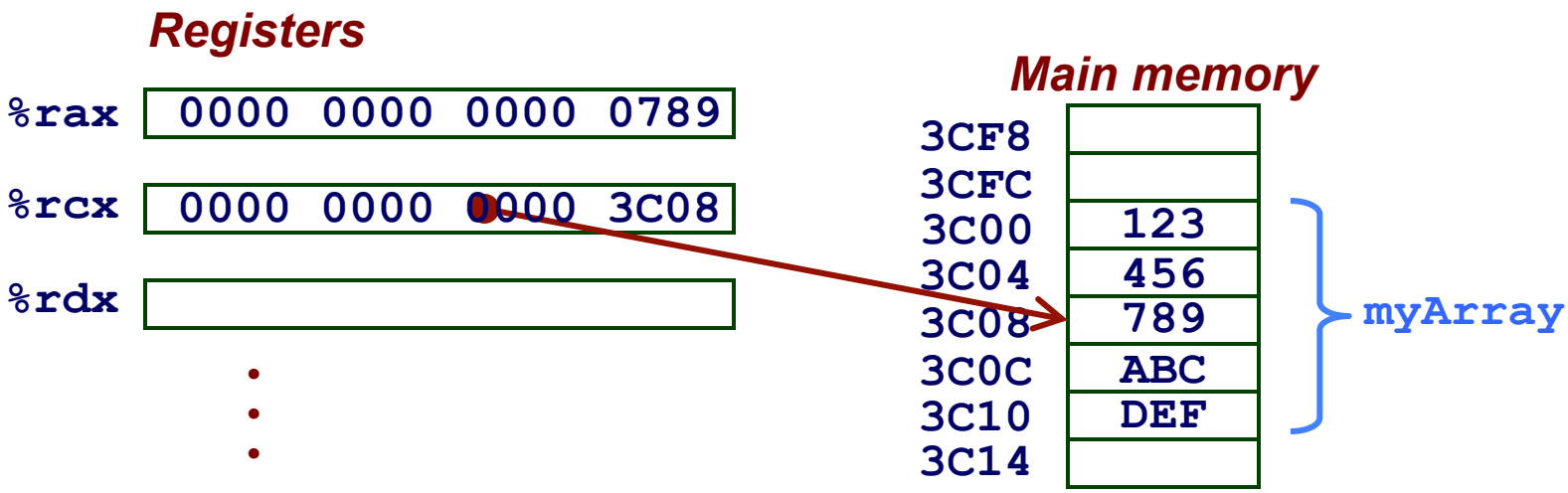
Indirect Memory Mode

Form: (E_a)

Operand value: $M[R[E_a]]$

Register E_a specifies the memory address

```
movl (%rcx),%eax
```



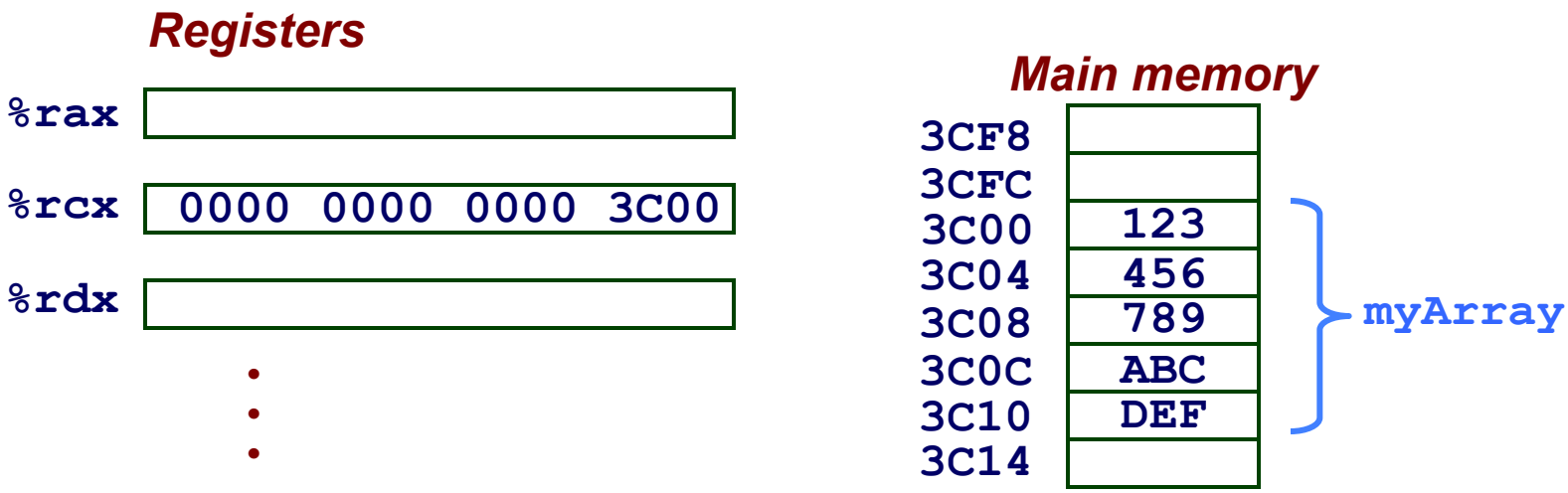
Memory Mode: Base + Displacement

Form: `Imm(E_b)`

Operand value: `M[Imm+R[E_b]]`

- Register E_b specifies start of memory region
- `Imm` specifies the offset/displacement

```
movl 8(%rcx),%eax
```



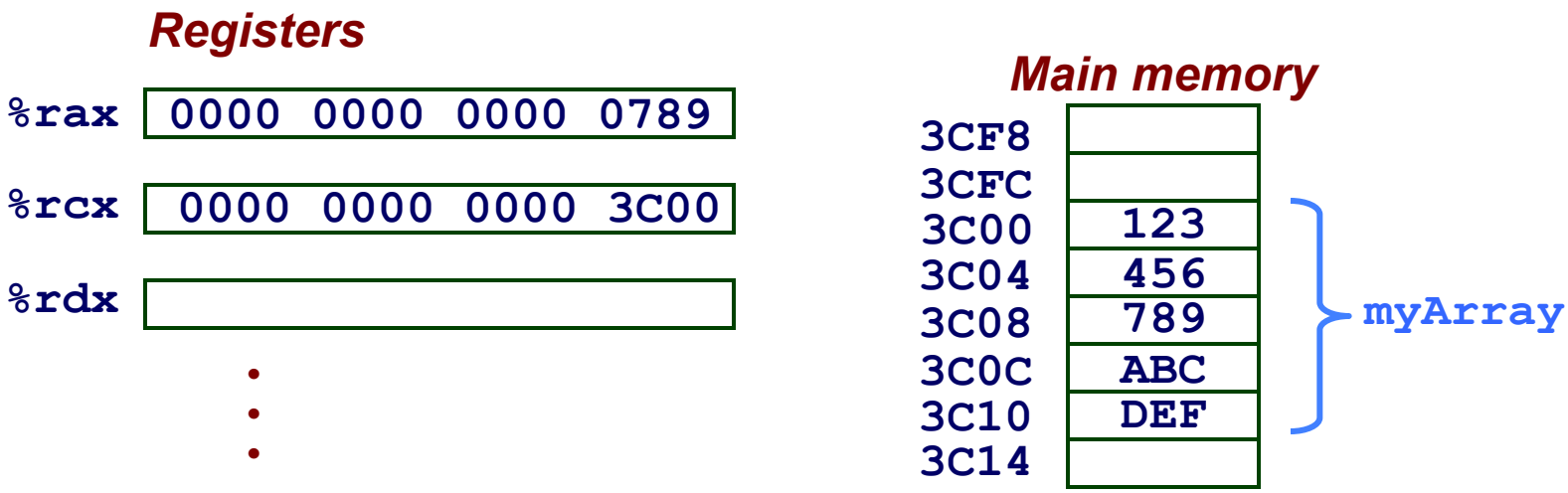
Memory Mode: Base + Displacement

Form: `Imm(E_b)`

Operand value: `M[Imm+R[E_b]]`

- Register E_b specifies start of memory region
- `Imm` specifies the offset/displacement

```
movl 8(%rcx),%eax
```



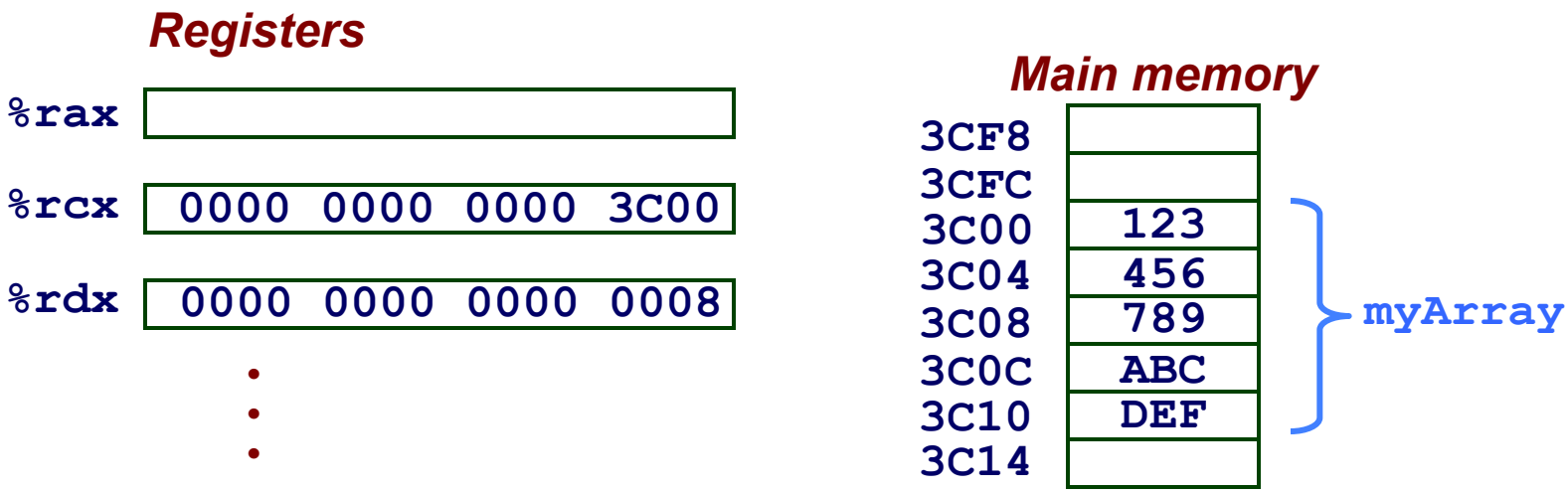
Indexed Memory Mode

Form: `Imm(E_b,E_i)`

Operand value: `M[Imm+R[E_b]+Reg[E_i]]`

- Register E_b specifies start of memory region
- Register E_i holds index

```
movl 4(%rcx,%rdx),%eax
```



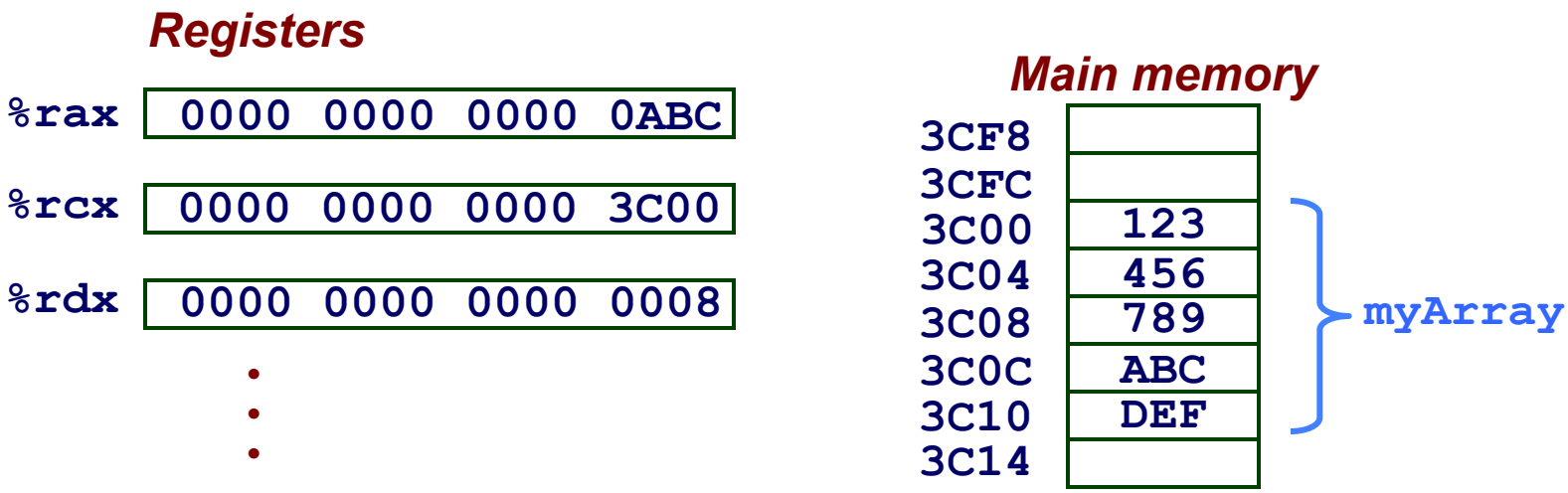
Indexed Memory Mode

Form: `Imm(E_b,E_i)`

Operand value: `M[Imm+R[E_b]+Reg[E_i]]`

- Register E_b specifies start of memory region
- Register E_i holds index

```
movl 4(%rcx,%rdx),%eax
```



Memory Mode: Scaled Indexed

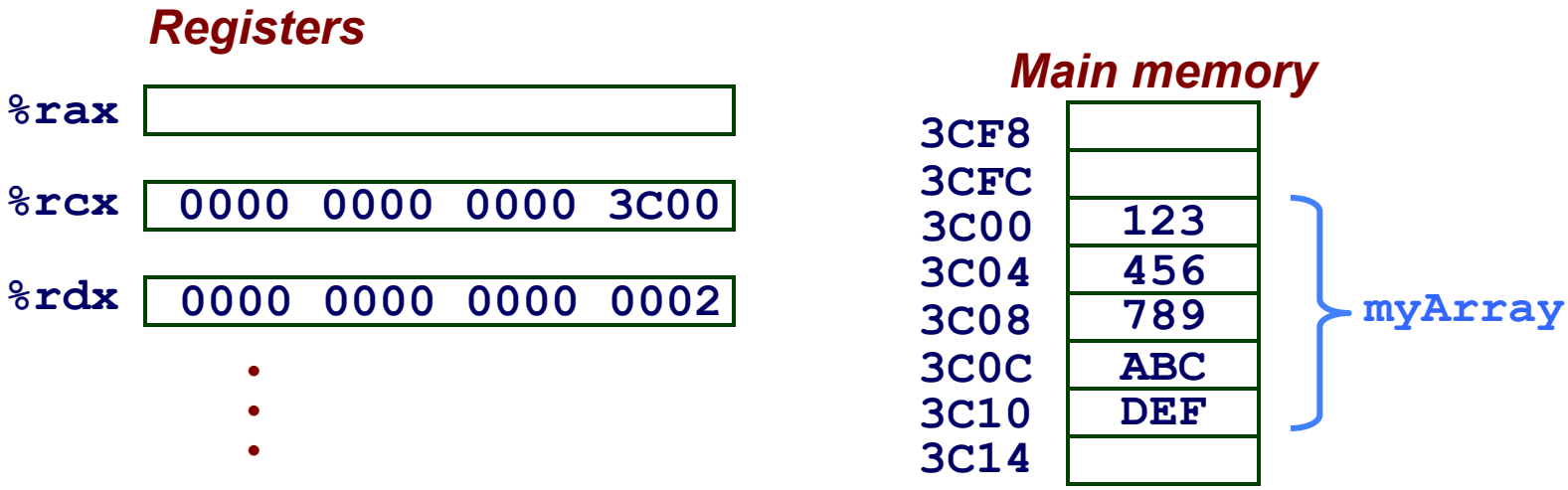
The most general format

Form: `Imm(E_b,E_i,S)`

Operand value: `M[Imm+R[E_b]+S*R[E_i]]`

- Register E_b specifies start of memory region
- E_i holds index
- S is integer scale (1,2,4,8)

```
movl 8(%rcx,%rdx,4),%eax
```



Memory Mode: Scaled Indexed

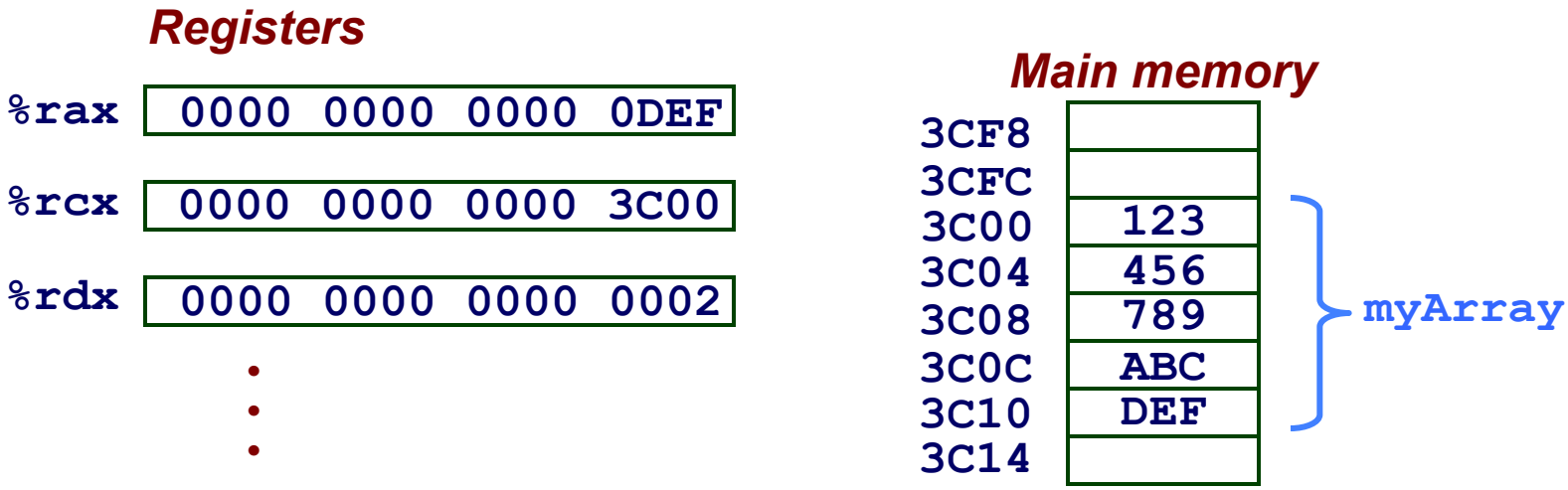
The most general format

Form: `Imm(E_b,E_i,S)`

Operand value: `M[Imm+R[E_b]+S*R[E_i]]`

- Register E_b specifies start of memory region
- E_i holds index
- `S` is integer scale (1,2,4,8)

```
movl 8(%rcx,%rdx,4),%eax
```



Most General Form: Scaled indexed

Absolute, indirect, base+displacement, and indexed are simply special cases of scaled indexed.

General Form

$Imm(E_b,E_i,S)$	M[Imm + R[E_b] + R[E_i]*S]

Examples

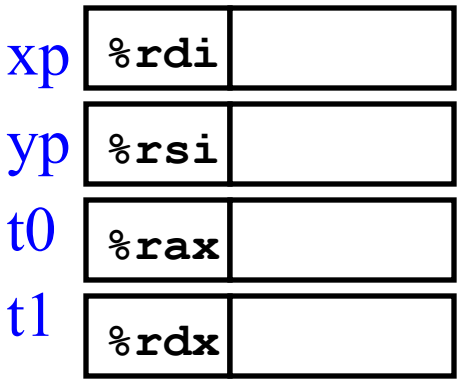
Imm	M[Imm]
$Imm(E_b)$	M[Imm + R[E_b]]
(E_b,E_i,S)	M[R[E_b] + R[E_i]*S]
(E_b,E_i)	M[R[E_b] + R[E_i]]
$(,E_i,S)$	M[R[E_i]*S]
$Imm(,E_i,S)$	M[Imm + R[E_i]*S]

Understanding Swap()

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

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

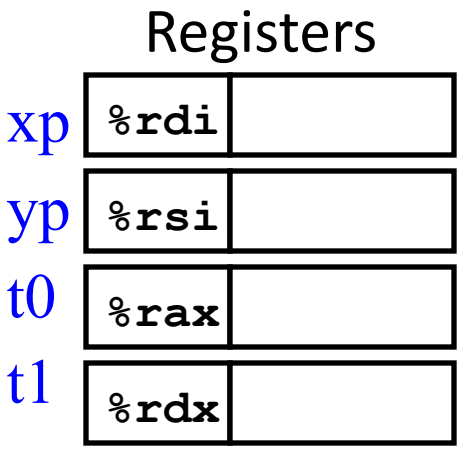
Registers

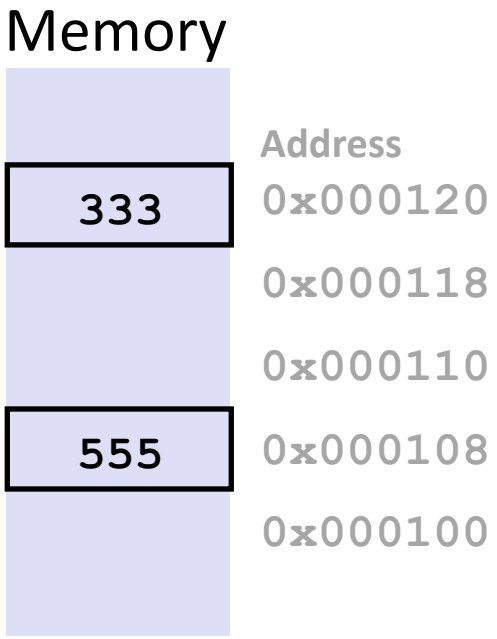


Understanding Swap()

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

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

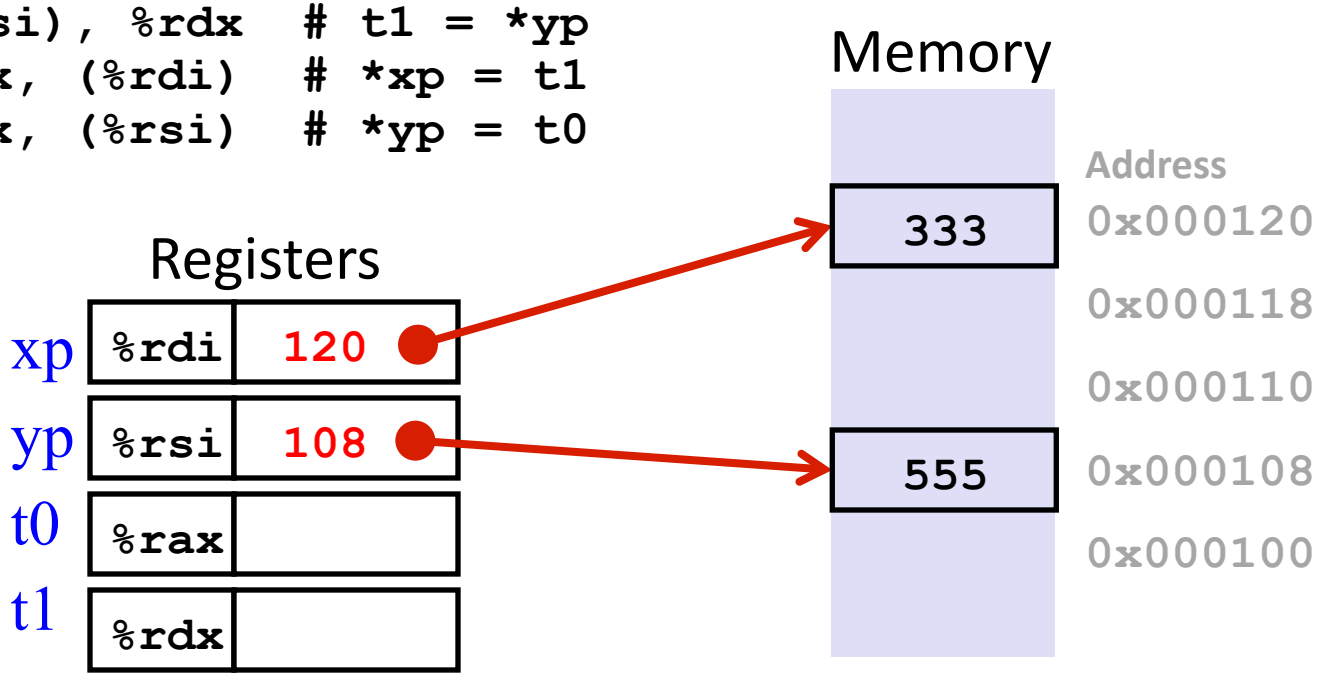




Understanding Swap()

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

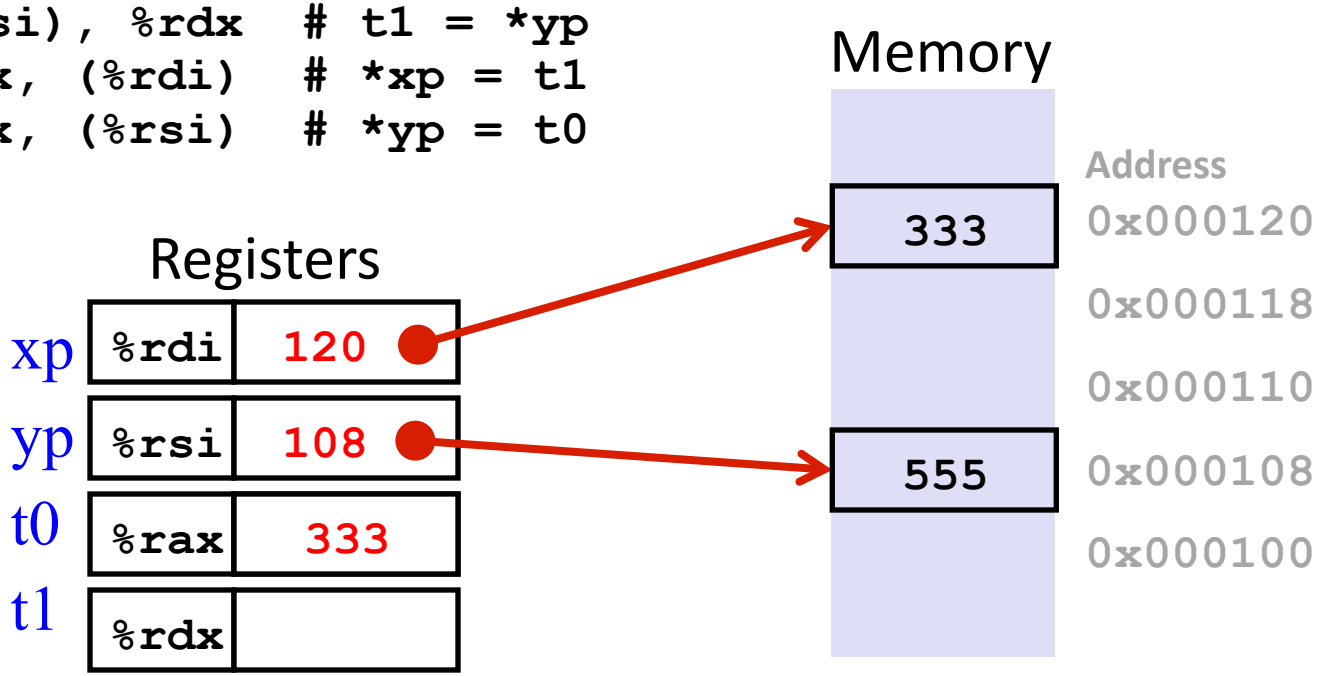
```
swap:
    movq     (%rdi), %rax  # t0 = *xp
    movq     (%rsi), %rdx  # t1 = *yp
    movq     %rdx, (%rdi)  # *xp = t1
    movq     %rax, (%rsi)  # *yp = t0
    ret
```



Understanding Swap()

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

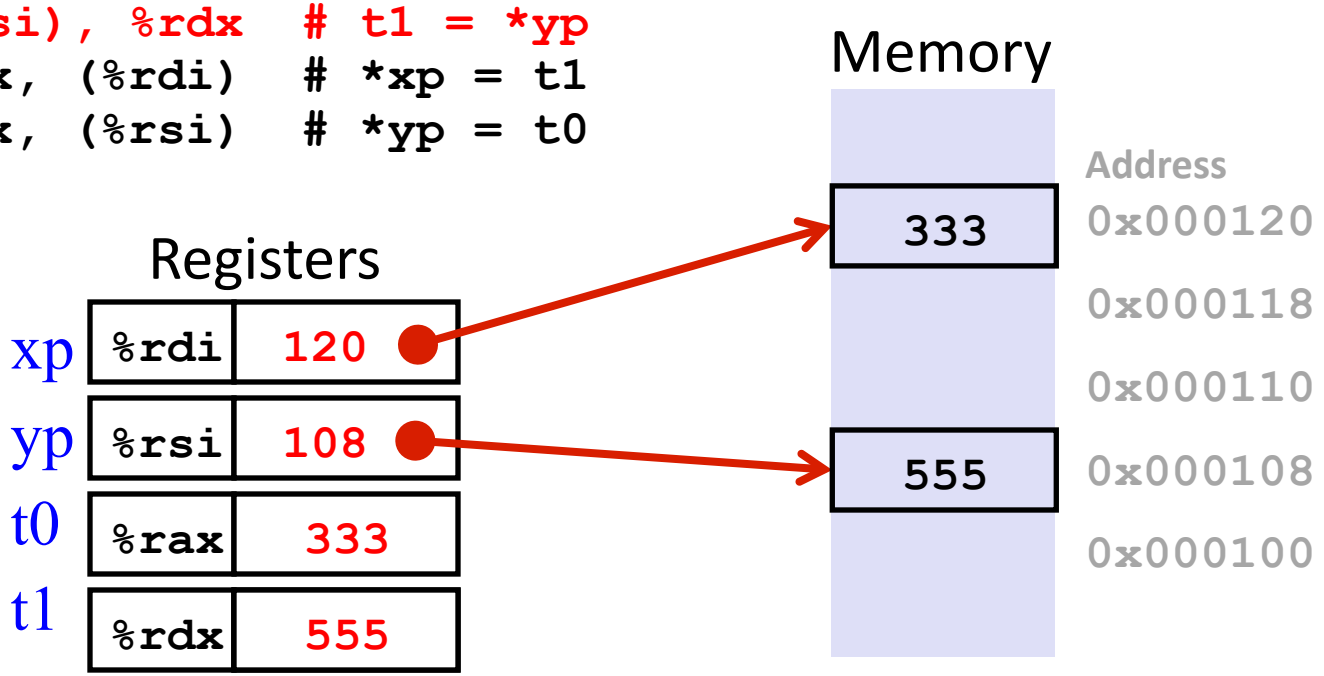
```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```



Understanding Swap()

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

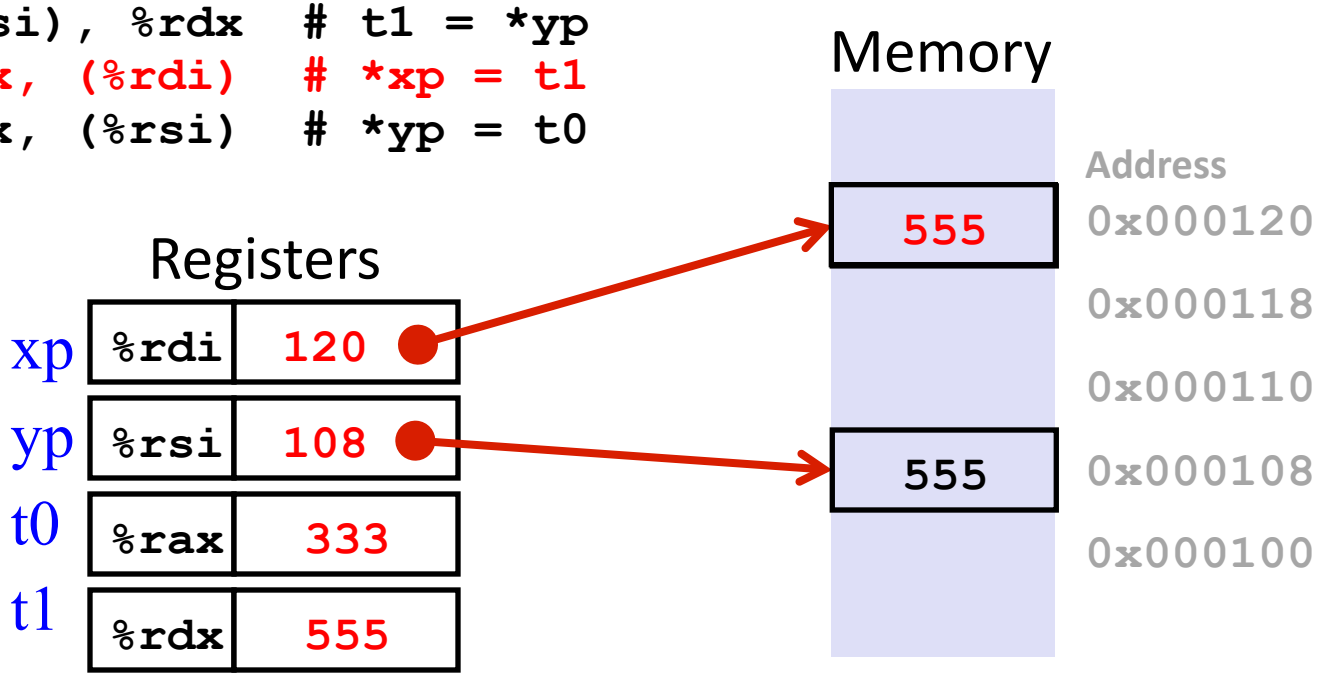
```
swap:
    movq     (%rdi), %rax   # t0 = *xp
    movq     (%rsi), %rdx   # t1 = *yp
    movq     %rdx, (%rdi)   # *xp = t1
    movq     %rax, (%rsi)   # *yp = t0
    ret
```



Understanding Swap()

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

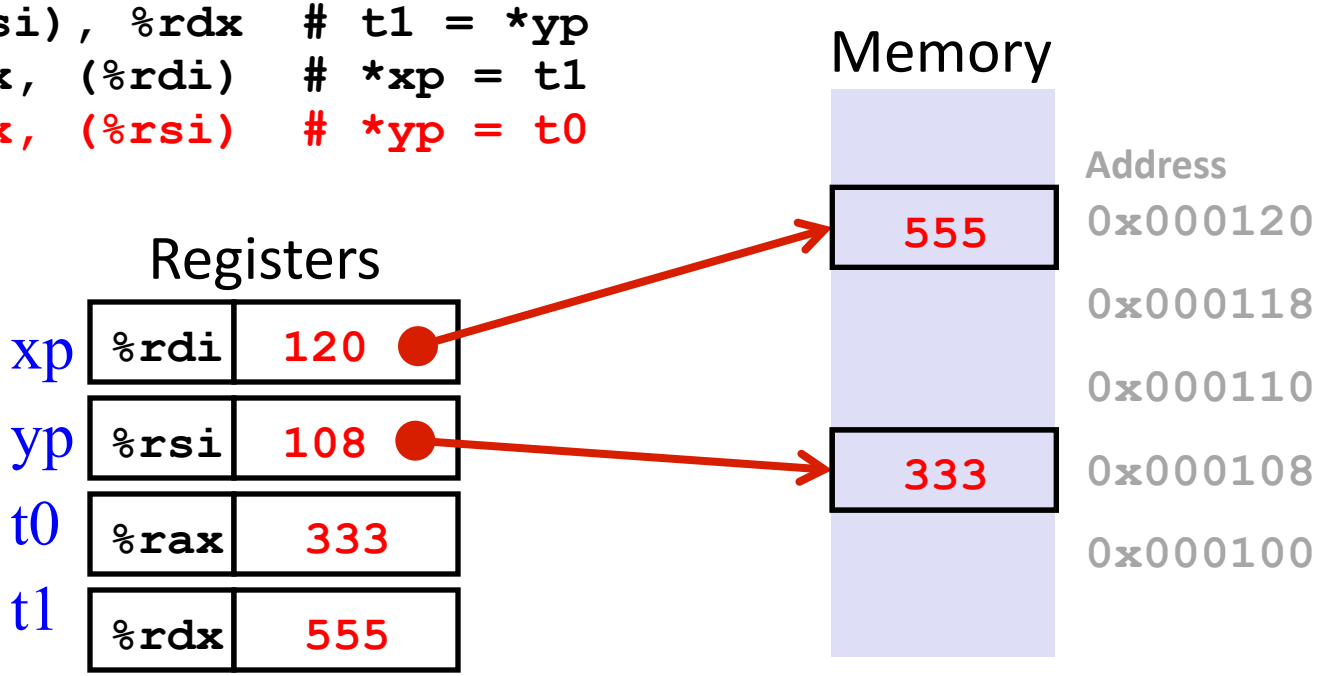
```
swap:
    movq     (%rdi), %rax  # t0 = *xp
    movq     (%rsi), %rdx  # t1 = *yp
    movq     %rdx, (%rdi)  # *xp = t1
    movq     %rax, (%rsi)  # *yp = t0
    ret
```



Understanding Swap()

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

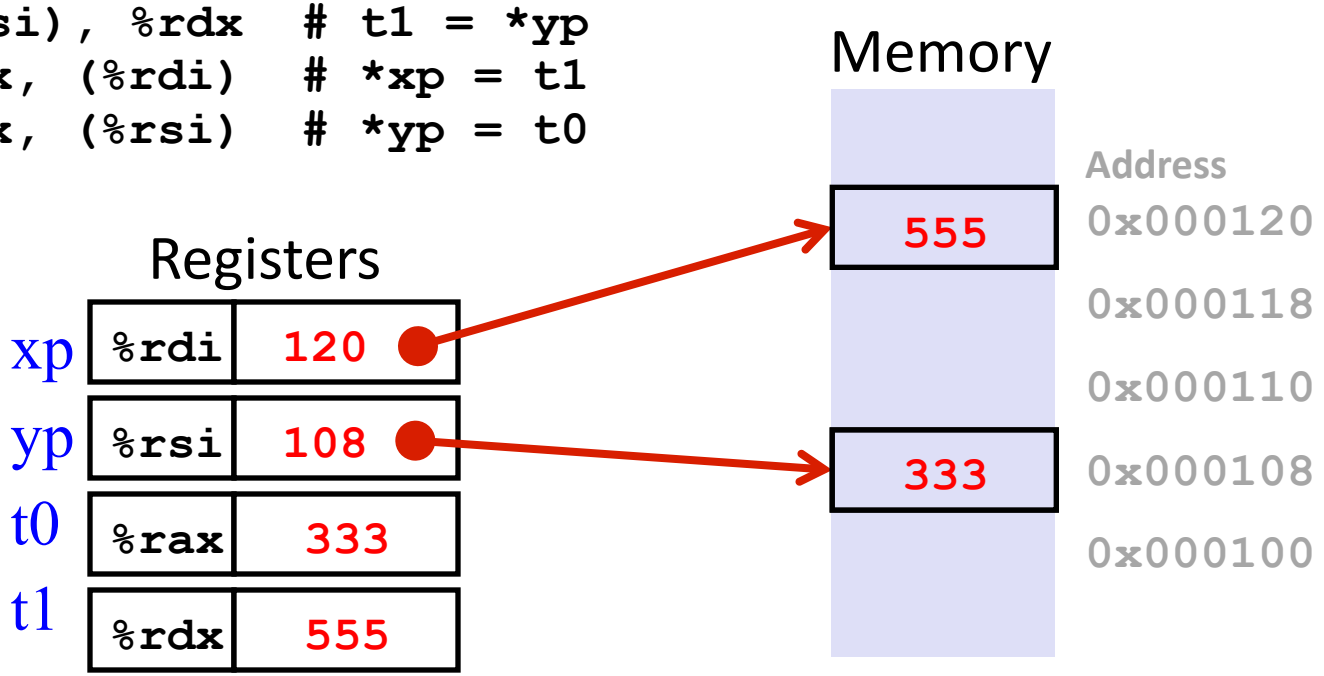
```
swap:
    movq     (%rdi), %rax  # t0 = *xp
    movq     (%rsi), %rdx  # t1 = *yp
    movq     %rdx, (%rdi)  # *xp = t1
    movq     %rax, (%rsi)  # *yp = t0
    ret
```



Understanding Swap()

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

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



Addressing Mode Examples

`addq 12(%rbp),%rcx`	Add the long-long at address rbp+12 to rcx
`movb (%rax,%rcx),%dl`	Load the byte at address rax+rcx into dl
`subl %edx,(%rcx,%rax,4)`	Subtract edx from the int at address rcx+(4*rax)
`incw 0xA(,%rcx,8)`	Increment the short at address 0xA+(8*rcx)

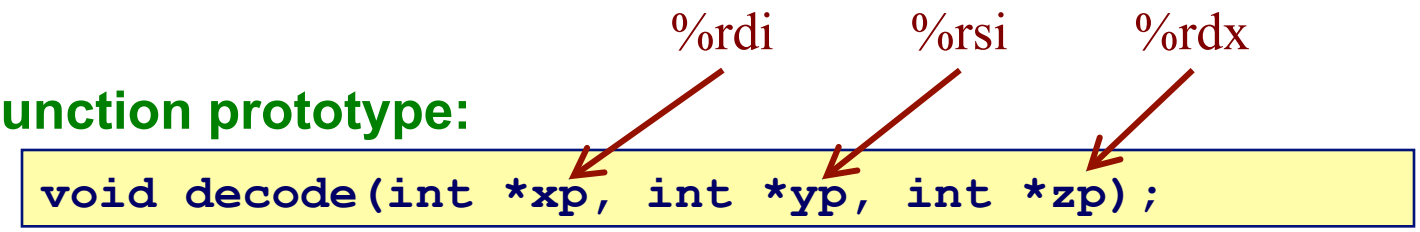
Practice Problems

Register	Value
%rax	0x00000100
%rcx	0x00000001
%rdx	0x00000003

Address	Value
0x0100	0xFF
0x0104	0xAB
0x0108	0x13
0x010C	0x11

Operand	Value
%rax	0x100
0x104	0xAB
$0x108	0x108
(%rax)	0xFF
4(%rax)	0xAB
9(%rax, %rdx)	0x11
260(%rcx, %rdx)	0x13
0xFC(,%rcx,4)	0xFF
(%rax,%rdx,4)	0x11

Practice Problem



Function prototype:

```
void decode(int *xp, int *yp, int *zp);
```

Assembly code:

```
1   decode:
2       movq (%rdi),%r8      t1 = *xp
3       movq (%rsi),%rcx     t2 = *yp
4       movq (%rdx),%rax     t3 = *zp
5       movq %r8,(%rsi)      *yp = t1
6       movq %rcx,(%rdx)     *zp = t2
7       movq %rax,(%rdi)     *xp = t3
8       ret
```

```
void decode(…) {
    long x=*xp;
    long y=*yp;
    long z=*zp;
    *yp=x;
    *zp=y;
    *xp=z;
}
```

Write C code for this function

Practice Problem

Suppose an array in C is declared as a global variable:

```
int myArray[34];
```

Write some assembly code that:

- **sets `%rsi` to the address of myArray**
- **sets `%rbx` to the constant 9**
- **loads `myArray[9]` into register `%eax`.**

```
movq $myArray,%rsi
movq $0x9,%rbx
movl (%rsi,%rbx,4),%eax
```

Use scaled index memory mode

Alternate mov instructions

Not all move instructions are equivalent

- **There are three byte move instructions and each produces a different result**

  movb only changes specific byte

  movsbl does sign extension

  movzbl sets other bytes to zero

Assumptions: %dh = 0x8D, %eax = 0x98765432

`movb %dh, %al`	%eax = 0x9876548D
`movsbl %dh, %eax`	%eax = 0xFFFFFF8D
`movzbl %dh, %eax`	%eax = 0x0000008D

Data Movement Instructions

Instruction		Effect	Description
`movl`	`S,D`	D ← S	Move double word
`movw`	`S,D`	D ← S	Move word
`movb`	`S,D`	D ← S	Move byte
`movsbl`	`S,D`	D ← SignExtend(S)	Move sign-extended byte
`movzbl`	`S,D`	D ← ZeroExtend(S)	Move zero-extended byte

Arithmetic and Logical Operations

LEA: Load Effective Address

`movq` *Source, Dest*

Dest ← M[*Source*]

`leaq` *Source, Dest*

Dest ← &*Source*

```
leaq (%rax),%rdx
movq %rax,%rdx
```

} Equivalent

Destination must be a register

Commonly used by compiler to do simple arithmetic

If variable "k" is in %rbx…

```
leaq 7(%rbx,%rbx,4),%rax
```

%rax ← (5 × k) + 7

Performs a multiply and add all in one instruction!

Practice Problem

%rax = x, %rcx = y

Instruction	Result in %rdx?
`leaq 6(%rax), %rdx`	x+6
`leaq (%rax, %rcx), %rdx`	x+y
`leaq (%rax, %rcx, 4), %rdx`	x+4y
`leaq 7(%rax, %rax, 8), %rdx`	9x+7
`leaq 0xA(, %rcx, 4), %rdx`	4y+10
`leaq 9(%rax, %rcx, 2), %rdx`	x+2y+9
`leaq (%rcx, %rax, 4), %rdx`	4x+y
`leaq 1(, %rax, 2), %rdx`	2x+1

Unary Operations

Unary ⇒ one operand

```
inc = increment     ⇒ D ← D + 1
dec = decrement     ⇒ D ← D - 1
neg = negate        ⇒ D ← -D
not = complement    ⇒ D ← ~D
```

Examples

```
incl (%rsp)
```

Increment 32-bit quantity at top of stack

```
notl %eax
```

Complement 32-bit quantity in register %eax

Binary Operations

A little bit tricky

- The second operand is used as both a source and destination
- A bit like C operators +=, -=, etc.

Format

`<op> Src,Dest` ⇒ Dest = Dest <op> Src

Can be confusing

`subl X,Y` ⇒ Y = Y – X

Be careful! Not X – Y

Examples

```
add S,D   ⇒ D = D + S
sub S,D   ⇒ D = D - S
xor S,D   ⇒ D = D ^ S
or  S,D   ⇒ D = D | S
and S,D   ⇒ D = D & S
```

addb, addw, addl, addq

subb, subw, subl, subq

etc.

Integer Multiply

`imul` = signed multiply

`mul` = unsigned multiply

Binary form

```
imul Src,Dest
```

dest ← dest × src

Multiple forms based on 8-bit, 16-bit, 32-bit, 64-bit

imulb, imulw, imull, imulq,

mulb, mulw, mull, mulq

Unary form (for 128-bit result)

```
mulq %r13        imulq %rbx
```

Other operand is assumed to be in `%rax`

128-bit result in %rdx:%rax`

```
%rdx:%rax ← %r??? × %rax
```

Integer Divide

`idiv` = signed divide

`div` = unsigned divide

Binary form

Not present (?)

Unary form

```
divq  %r13
idivq %rbx
```

Computes both division and remainder at once!

Other operand is assumed to be in `%rax`

`%rax` ← `%r???` ÷ `%rax`

`%rdx` ← `%r???` mod `%rax`

R[%edx]:R[%eax] *is viewed as a 64-bit quad word*

Instruction	Effect	
imull S	R[%edx]:R[%eax] ← S x R[%eax]	signed
mull S	R[%edx]:R[%eax] ← S x R[%eax]	unsigned
cltd	R[%edx]:R[%eax] ← SignExtend(R[%eax])	
idivl S	R[%edx] ← R[%edx]:R[%eax] mod S R[%eax] ← R[%edx]:R[%eax] ÷ S	signed
divl S	R[%edx] ← R[%edx]:R[%eax] mod S R[%eax] ← R[%edx]:R[%eax] ÷ S	unsigned

Practice Problem

Address	Value
0x100	0xFF
0x108	0xAB
0x110	0x13
0x118	0x11

Register	Value
%rax	0x100
%rcx	0x1
%rdx	0x3

Instruction	Destination address	Result
addq %ecx, (%eax)	0x100	0x100
subw %edx, 4(%eax)	0x108	0xA8
imulq $16, (%eax,%edx,4)	0x118	0x110
incq 8(%eax)	0x110	0x14
decq %ecx	%rcx	0x0
subq %edx, %eax	%rax	0xFD

Logical Operators in C

Bitwise operators → assembly instructions

```
&   → and    b/w/l/q
|   → or     b/w/l/q
^   → xor    b/w/l/q
~   → not    b/w/l/q
```

Short-circuit logical operators give "true" or "false" result

```
    &&    ||     !
```

`cmpb/w/l/q, testb/w/l/q` set the condition code registers

Condition Codes (each is one bit):

- **CF** Carry Flag
- **ZF** Zero Flag
- **SF** Sign Flag
- **OF** Overflow Flag

Can be tested by other subsequent instructions

Shift Operations

Arithmetic and logical shifts are possible

<op> amount value

```
sal k,D     ⇒ D = D << k
shl k,D     ⇒ D = D << k
```

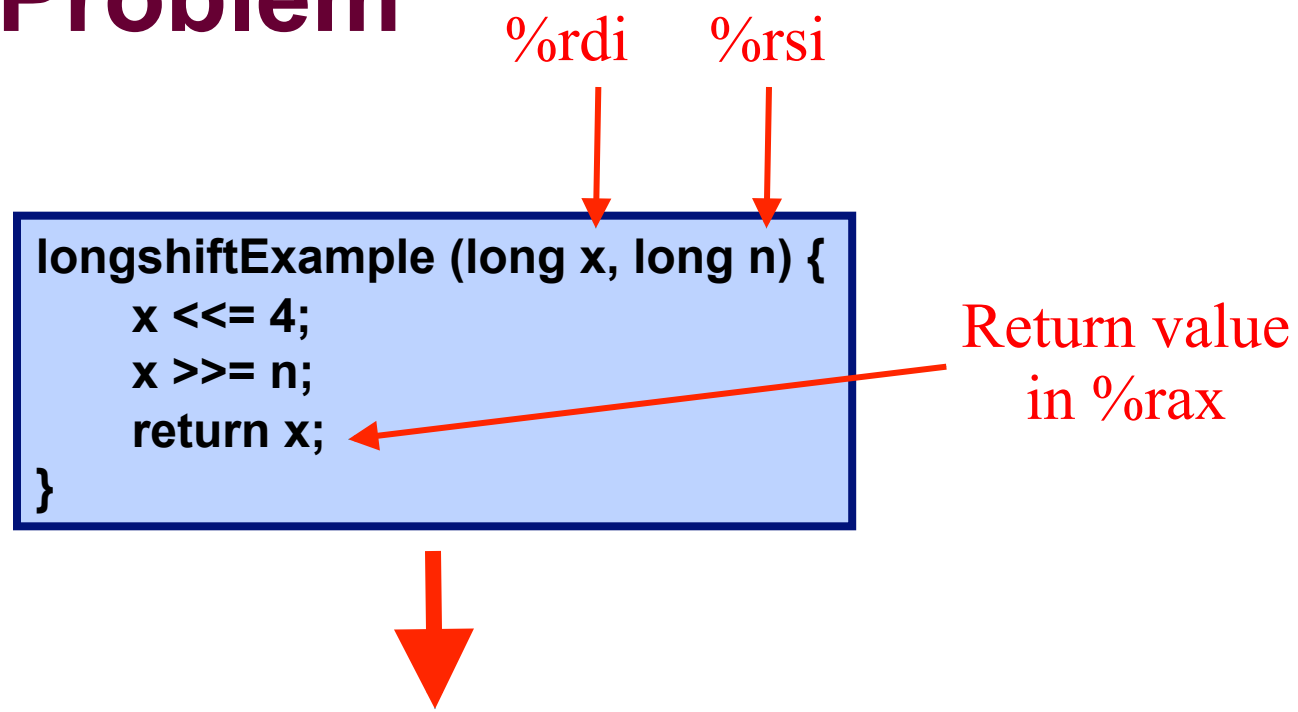
} same (i.e., same machine code)

```
sar k,D     ⇒ D = D >> k, sign extend
shr k,D     ⇒ D = D >> k, zero fill
```

Max shift is 64 bits, so k is either an immediate byte, or register (e.g. `%cl`)

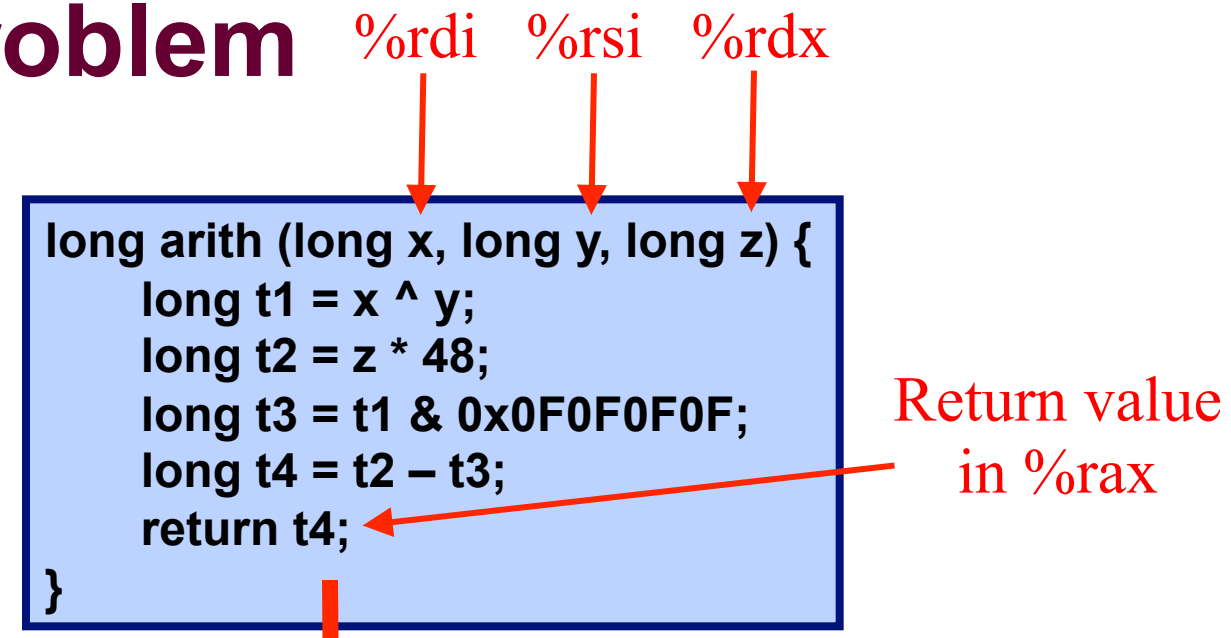
- `%cl` **is byte 0 of register** `%rcx`

Practice Problem




```
shiftExample:
        movq    %rdi,%rax       ; get x
        salq    $4,%rax         ; x <<= 4;
        movl    %esi,%ecx       ; get n
        sarq    %cl,%rax        ; x >>= n;
        ret
```

Practice Problem




```
arith:
        xorq    %rsi,%rdi               t1 = x ^ y
        leaq    (%rdx,rdx,2),%rax       3×z
        salq    $4,%rax                 t2 = 16×(3×z) = 48×z
        andl    $252645135,%edi         t3 = t1& 0x0F0F0F0F
        subq    %rdi,%rax               return t2-t3
        ret
```

Disassembling Object Code

Disassembled

```
00401040 <_sum>:
   0:  55                   push   %ebp
   1:  89 e5                mov    %esp,%ebp
   3:  8b 45 0c             mov    0xc(%ebp),%eax
   6:  03 45 08             add    0x8(%ebp),%eax
   9:  89 ec                mov    %ebp,%esp
   b:  5d                   pop    %ebp
   c:  c3                   ret
   d:  8d 76 00             lea    0x0(%esi),%esi
```

Disassembler

`objdump -d <object_file>` `elfdump (on Sun)`

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

gcc -Wall -O1 -m32 -Wa,-al foo.c -c

```
extern int x;
int y=456;

int foo3 (int i) {
  x = y + 123;
  return (y*i);
}
```

foo.c

gcc -Wall -O1 -m32 -Wa,-al foo.c -c

```
 1                          .file   "foo.c"
 2                          .text
 3                          .globl  foo3
 4                          .type   foo3, @function
 5                  foo3:
 6                  .LFB0:
 7                          .cfi_startproc
 8 0000 A1000000            movl    y, %eax
 8      00
 9 0005 8D507B              leal    123(%eax), %edx
10 0008 89150000            movl    %edx, x
10      0000
11 000e 0FAF4424            imull   4(%esp), %eax
11      04
12 0013 C3                  ret
13                          .cfi_endproc
14                  .LFE0:
15                          .size   foo3, .-foo3
16                          .globl  y
17                          .data
18                          .align  4
19                          .type   y, @object
20                          .size   y, 4
21                  y:
22 0000 C8010000            .long   456
23                          .ident  "GCC: (Ubuntu 4.8.2-19ubuntu1) 4.8.2"
24                          .section .note.GNU-stack,"",@progbits
```

```
extern int x;
int y=456;

int foo3 (int i) {
  x = y + 123;
  return (y*i);
}
```

foo.c

gcc -Wall -O1 -m32 -Wa,-al foo.c -c

```
 1                      .file   "foo.c"
 2                      .text
 3                      .globl  foo3
 4                      .type   foo3, @function
 5                  foo3:
 6                  .LFB0:
 7                      .cfi_startproc
 8 0000 A1000000        movl    y, %eax
 8      00
 9 0005 8D507B          leal    123(%eax), %edx
10 0008 89150000        movl    %edx, x
10      0000
11 000e 0FAF4424        imull   4(%esp), %eax
11      04
12 0013 C3              ret
13                      .cfi_endproc
14                  .LFE0:
15                      .size   foo3, .-foo3
16                      .globl  y
17                      .data
18                      .align  4
19                      .type   y, @object
20                      .size   y, 4
21                  y:
22 0000 C8010000        .long   456
23                      .ident  "GCC: (Ubuntu 4.8.2-19ubuntu1) 4.8.2"
24                      .section .note.GNU-stack,"",@progbits
```

```
extern int x;
int y=456;

int foo3 (int i) {
  x = y + 123;
  return (y*i);
}
```

foo.c

What is in memory?

```
                    foo3:
0000  A1000000        movl      y, %eax
      00
0005  8D507B          leal      123(%eax), %edx
0008  89150000        movl      %edx, x
      0000
000e  0FAF4424        imull     4(%esp), %eax
      04
0013  C3              ret
```

```
extern int x;
int y=456;

int foo3 (int i) {
  x = y + 123;
  return (y*i);
}
```

foo.c

```
0x??
0x??
0x??
0x??
```

x

```
0xc8
0x01
0x00
0x00
```

y

$456_{10} = 1C8_{16}$

What is in memory?

+0	a1
+1	00
+2	00
+3	00
+4	00
+5	8d
+6	50
+7	7b
+8	89
+9	15
+10	00
+11	00
+12	00
+13	00
+14	0f
+15	af
+16	44
+17	24
+18	04
+19	c3

```
                    foo3:
0000 A1000000       movl    y, %eax
     00
0005 8D507B         leal    123(%eax), %edx
0008 89150000       movl    %edx, x
     0000
000e 0FAF4424       imull   4(%esp), %eax
     04
0013 C3             ret
```

```
0x??
0x??
0x??
0x??
```

x

```
0xc8
0x01
0x00
0x00
```

y

```
extern int x;
int y=456;

int foo3 (int i) {
  x = y + 123;
  return (y*i);
}
```

foo.c

$456_{10} = 1C8_{16}$

What is in memory?

```
08048409:
  +0    a1
  +1    00
  +2    00
  +3    00
  +4    00
  +5    8d
  +6    50
  +7    7b
  +8    89
  +9    15
  +10   00
  +11   00
  +12   00
  +13   00
  +14   0f
  +15   af
  +16   44
  +17   24
  +18   04
  +19   c3
```

```
                    foo3:
0000  A1000000      movl    y, %eax
      00
0005  8D507B        leal    123(%eax), %edx
0008  89150000      movl    %edx, x
      0000
000e  0FAF4424      imull   4(%esp), %eax
      04
0013  C3            ret
```

```
0804a01c:  0x??
      1d:  0x??
      1e:  0x??
      1f:  0x??
      20:  0xc8
      21:  0x01
      22:  0x00
      23:  0x00
```

x

y

```
extern int x;
int y=456;

int foo3 (int i) {
  x = y + 123;
  return (y*i);
}
```

foo.c

$456_{10} = 1C8_{16}$

What is in memory?

```
08048409:
 +0   a1
 +1   20
 +2   a0
 +3   04
 +4   08
 +5   8d
 +6   50
 +7   7b
 +8   89
 +9   15
 +10  1c
 +11  a0
 +12  04
 +13  08
 +14  0f
 +15  af
 +16  44
 +17  24
 +18  04
 +19  c3
```

```
                    foo3:
0000  A1000000      movl    y, %eax
      00
0005  8D507B        leal    123(%eax), %edx
0008  89150000      movl    %edx, x
      0000
000e  0FAF4424      imull   4(%esp), %eax
      04
0013  C3            ret
```

```
0804a01c:  0x??
      1d:  0x??
      1e:  0x??
      1f:  0x??
      20:  0xc8
      21:  0x01
      22:  0x00
      23:  0x00
```

x

y

```
extern int x;
int y=456;

int foo3 (int i) {
  x = y + 123;
  return (y*i);
}
```

foo.c

$456_{10} = 1C8_{16}$

Using gdb to disassemble

```
08048409:
  +0    a1
  +1    20
  +2    a0
  +3    04
  +4    08
  +5    8d
  +6    50
  +7    7b
  +8    89
  +9    15
  +10   1c
  +11   a0
  +12   04
  +13   08
  +14   0f
  +15   af
  +16   44
  +17   24
  +18   04
  +19   c3
```

```
LINUX% gdb foo
(gdb) x/x foo3
0x8048409 <foo3>:      0x04a020a1
(gdb) RETURN
0x804840d <foo3+4>:    0x7b508d08
(gdb) RETURN
0x8048411 <foo3+8>:    0xa01c1589
(gdb) RETURN
0x8048415 <foo3+12>:   0xaf0f0804
(gdb) RETURN
0x8048419 <foo3+16>:   0xc3042444
(gdb)
```

Using gdb to disassemble

```
08048409:
  +0    a1
  +1    20
  +2    a0
  +3    04
  +4    08
  +5    8d
  +6    50
  +7    7b
  +8    89
  +9    15
  +10   1c
  +11   a0
  +12   04
  +13   08
  +14   0f
  +15   af
  +16   44
  +17   24
  +18   04
  +19   c3
```

```
LINUX% gdb foo
(gdb) x/x foo3
0x8048409 <foo3>:     0x04a020a1
(gdb) RETURN
0x804840d <foo3+4>:   0x7b508d08
(gdb) RETURN
0x8048411 <foo3+8>:   0xa01c1589
(gdb) RETURN
0x8048415 <foo3+12>:  0xaf0f0804
(gdb) RETURN
0x8048419 <foo3+16>:  0xc3042444
(gdb) disass foo3
Dump of assembler code for function foo3:
   0x08048409 <+0>:   mov    0x804a020,%eax
   0x0804840e <+5>:   lea    0x7b(%eax),%edx
   0x08048411 <+8>:   mov    %edx,0x804a01c
   0x08048417 <+14>:  imul   0x4(%esp),%eax
   0x0804841c <+19>:  ret
End of assembler dump.
```

Using gdb to disassemble

```
08048409:
  +0    a1
  +1    20
  +2    a0
  +3    04
  +4    08
  +5    8d
  +6    50
  +7    7b
  +8    89
  +9    15
  +10   1c
  +11   a0
  +12   04
  +13   08
  +14   0f
  +15   af
  +16   44
  +17   24
  +18   04
  +19   c3
```

```
LINUX% gdb foo
                    foo3:
0000 A1000000           movl        y, %eax
     00
0005 8D507B             leal        123(%eax), %edx
0008 89150000           movl        %edx, x
     0000
000e 0FAF4424           imull       4(%esp), %eax
     04
0013 C3                 ret
(gdb) disass foo3
Dump of assembler code for function foo3:
   0x08048409 <+0>:   mov    0x804a020,%eax
   0x0804840e <+5>:   lea    0x7b(%eax),%edx
   0x08048411 <+8>:   mov    %edx,0x804a01c
   0x08048417 <+14>:  imul   0x4(%esp),%eax
   0x0804841c <+19>:  ret
End of assembler dump.
```

Where can you look this stuff up?

Is there a manual for assembly language?

Intel Architecture Software Developer's Manual

- Vol 2: Instruction Set Reference
  - A complete reference to the machine instruction set
  - Some indication of the assembly language also
  - Intel publishes an assembly language manual

For Linux & gas assembler

- Go to llinuxassembly.org and follow link to docs
- Start with existing assembly code and modify it

From Intel Manual

ADD—Add

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
04 *ib*	ADD AL, *imm8*	I	Valid	Valid	Add *imm8* to AL.
05 *iw*	ADD AX, *imm16*	I	Valid	Valid	Add *imm16* to AX.
05 *id*	ADD EAX, *imm32*	I	Valid	Valid	Add *imm32* to EAX.
REX.W + 05 *id*	ADD RAX, *imm32*	I	Valid	N.E.	Add *imm32 sign-extended to 64-bits* to RAX.
80 /0 *ib*	ADD *r/m8, imm8*	MI	Valid	Valid	Add *imm8* to *r/m8.*
REX + 80 /0 *ib*	ADD *r/m8**, *imm8*	MI	Valid	N.E.	Add *sign*-extended *imm8* to *r/m64.*
81 /0 *iw*	ADD *r/m16, imm16*	MI	Valid	Valid	Add *imm16* to *r/m16.*
81 /0 *id*	ADD *r/m32, imm32*	MI	Valid	Valid	Add *imm32* to *r/m32.*

SOURCE: Intel.com --- 3,603 pages!!!

From Intel Manual

ADD—Add

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
04 *ib*	ADD AL, *imm8*	I	Valid	Valid	Add *imm8* to AL.
05 *iw*	ADD AX, *imm16*	I	Valid	Valid	Add *imm16* to AX.
05 *id*	ADD EAX, *imm32*	I	Valid	Valid	Add *imm32* to EAX.
REX.W + 05 *id*	ADD RAX, *imm32*	I	Valid	N.E.	Add *imm32 sign-extended to 64-bits* to RAX.
80 /0 *ib*	ADD *r/m8, imm8*	MI	Valid	Valid	Add *imm8* to *r/m8*.
REX + 80 /0 *ib*	ADD *r/m8*, imm8*	MI	Valid	N.E.	Add *sign*-extended *imm8* to *r/m64*.
81 /0 *iw*	ADD *r/m16, imm16*	MI	Valid	Valid	Add *imm16* to *r/m16*.
81 /0 *id*	ADD *r/m32, imm32*	MI	Valid	Valid	Add *imm32* to *r/m32*.
REX.W + 81 /0 *id*	ADD *r/m64, imm32*	MI	Valid	N.E.	Add *imm32 sign-extended to 64-bits* to *r/m64*.
83 /0 *ib*	ADD *r/m16, imm8*	MI	Valid	Valid	Add *sign*-extended *imm8* to *r/m16*.
83 /0 *ib*	ADD *r/m32, imm8*	MI	Valid	Valid	Add *sign*-extended *imm8* to *r/m32*.
REX.W + 83 /0 *ib*	ADD *r/m64, imm8*	MI	Valid	N.E.	Add *sign*-extended *imm8* to *r/m64*.
00 /r	ADD *r/m8, r8*	MR	Valid	Valid	Add *r8* to *r/m8*.
REX + 00 /r	ADD *r/m8*, r8**	MR	Valid	N.E.	Add *r8* to *r/m8*.
01 /r	ADD *r/m16, r16*	MR	Valid	Valid	Add *r16* to *r/m16*.
01 /r	ADD *r/m32, r32*	MR	Valid	Valid	Add r32 to *r/m32*.
REX.W + 01 /r	ADD *r/m64, r64*	MR	Valid	N.E.	Add r64 to *r/m64*.
02 /r	ADD *r8, r/m8*	RM	Valid	Valid	Add *r/m8* to *r8*.
REX + 02 /r	ADD *r8*, r/m8**	RM	Valid	N.E.	Add *r/m8* to *r8*.
03 /r	ADD *r16, r/m16*	RM	Valid	Valid	Add *r/m16* to *r16*.
03 /r	ADD *r32, r/m32*	RM	Valid	Valid	Add *r/m32* to *r32*.
REX.W + 03 /r	ADD *r64, r/m64*	RM	Valid	N.E.	Add *r/m64* to *r64*.

NOTES:
*In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA
I	AL/AX/EAX/RAX	imm8	NA	NA

Description

Adds the destination operand (first operand) and the source operand (second operand) and then stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The ADD instruction performs integer addition. It evaluates the result for both signed and unsigned integer operands and sets the OF and CF flags to indicate a carry (overflow) in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
DEST ← DEST + SRC;
```

Flags Affected

The OF, SF, ZF, AF, CF, and PF flags are set according to the result.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

SOURCE: Intel.com --- 3,603 pages!!!

CONTENTS

SOURCE: Intel.com --- 3,603 pages!!!

From Intel Manual

AND—Logical AND

Opcode	Instruction	Description
24 *ib*	AND AL,*imm8*	AL AND *imm8*
25 *iw*	AND AX,*imm16*	AX AND i*mm16*
25 *id*	AND EAX,*imm32*	EAX AND *imm32*
80 /4 *ib*	AND *r/m8,imm8*	*r/m8* AND *imm8*
81 /4 *iw*	AND *r/m16,imm16*	*r/m16* AND *imm16*
81 /4 *id*	AND *r/m32,imm32*	*r/m32* AND *imm32*
83 /4 *ib*	AND *r/m16,imm8*	*r/m16* AND *imm8 (sign-extended)*
83 /4 *ib*	AND *r/m32,imm8*	*r/m32* AND *imm8 (sign-extended)*
20 /r	AND *r/m8,r8*	*r/m8* AND *r8*
21 /r	AND *r/m16,r16*	*r/m16* AND *r16*
21 /r	AND *r/m32,r32*	*r/m32* AND *r32*
22 /r	AND *r8,r/m8*	*r8* AND *r/m8*
23 /r	AND *r16,r/m16*	*r16* AND *r/m16*
23 /r	AND *r32,r/m32*	*r32* AND *r/m32*

SOURCE: http://download.intel.com/design/intarch/manuals/24319101.pdf

From Intel Manual

Description

This instruction performs a bitwise AND operation on the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. Two memory operands cannot, however, be used in one instruction. Each bit of the instruction result is a 1 if both corresponding bits of the operands are 1; otherwise, it becomes a 0.

Operation

DEST ← DEST AND SRC;

Flags Affected

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.

SOURCE: http://download.intel.com/design/intarch/manuals/24319101.pdf

From Intel Manual

Protected Mode Exceptions

#GP(0)	If the destination operand points to a nonwritable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

SOURCE: http://download.intel.com/design/intarch/manuals/24319101.pdf