CSCI-UA.0201

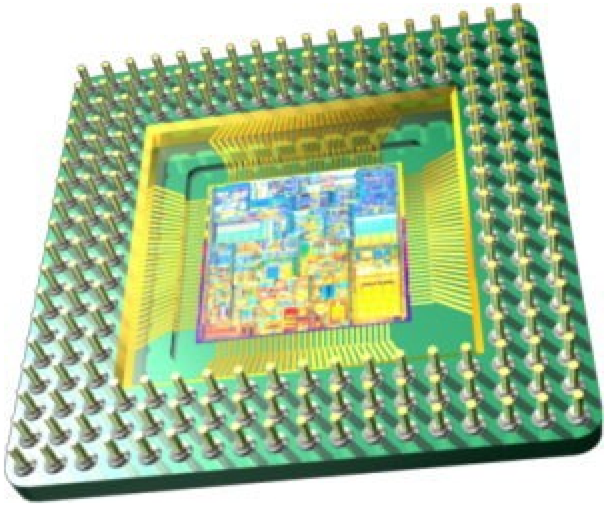
Computer Systems Organization

Machine-Level Programming I

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

Some slides adapted (and slightly modified) from:

- Clark Barrett
- Jinyang Li
- Randy Bryant
- Dave O'Hallaron



Intel x86 Processors

- Evolutionary design
  - Backwards compatible up until 8086, introduced in 1978

- Complex instruction set computer (CISC)
  - Many instructions, many formats
  - By contrast, ARM architecture (in most cell phones and tablets) is RISC

Intel x86 Evolution: Milestones

Name	*Transistors*	*MHz*
8086 (1978)	29K	5-10
386 (1985)	275K	16-33
Pentium 4F (2004)	125M	2800-3800
Core i7 (2008)	731M	2667-3333
Xeon E7 (2011)	2.2B	~2400

- 8086 (1978)
  - First 16-bit processor. Basis for IBM PC & DOS
  - 1MB address space
- 386 (1985)
  - First 32 bit processor , referred to as **IA32**
  - Capable of running Unix
- Pentium 4F (2004)
  - First 64-bit processor, referred to as **x86-64**

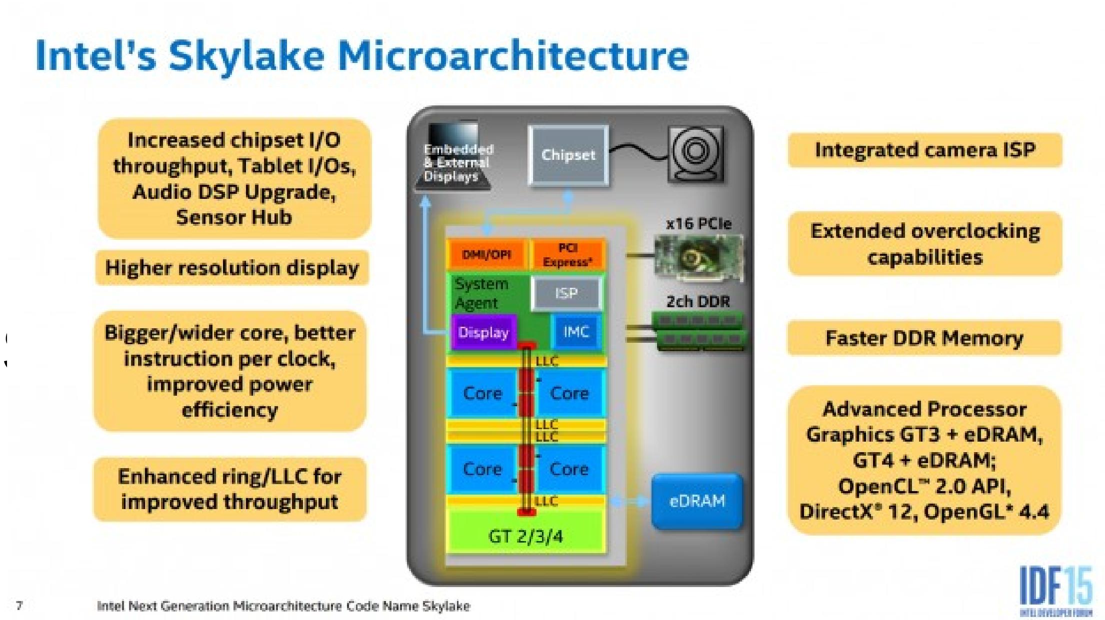
Instruction Set Architecture (ISA)

We will cover x86-64.

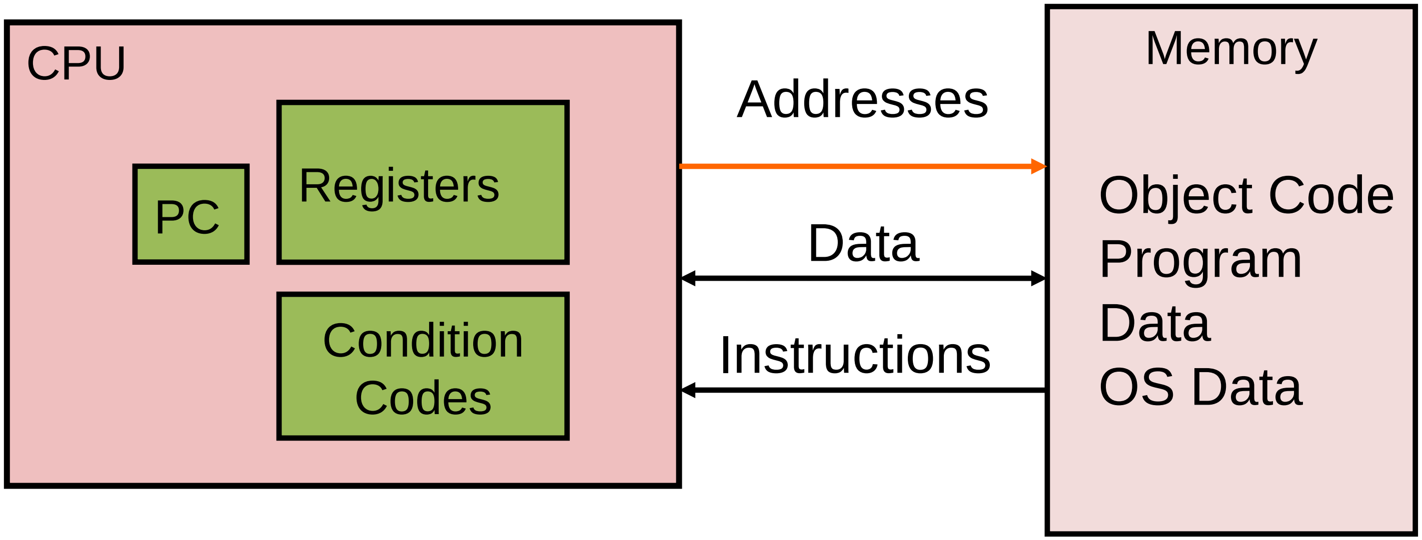
Example from 2015

– Intel Skylake

- 4-8 cores
- Integrated graphics
- 2.4-4.0 GHz
- Integrated I/O
- ~35W-95W



Assembly Programmer's View



- Execution context
  - PC: Program counter
    - Address of next instruction
    - Called "RIP" (x86-64)
  - Registers
    - Heavily used program data
  - Condition code registers
    - 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

Assembly Data Types

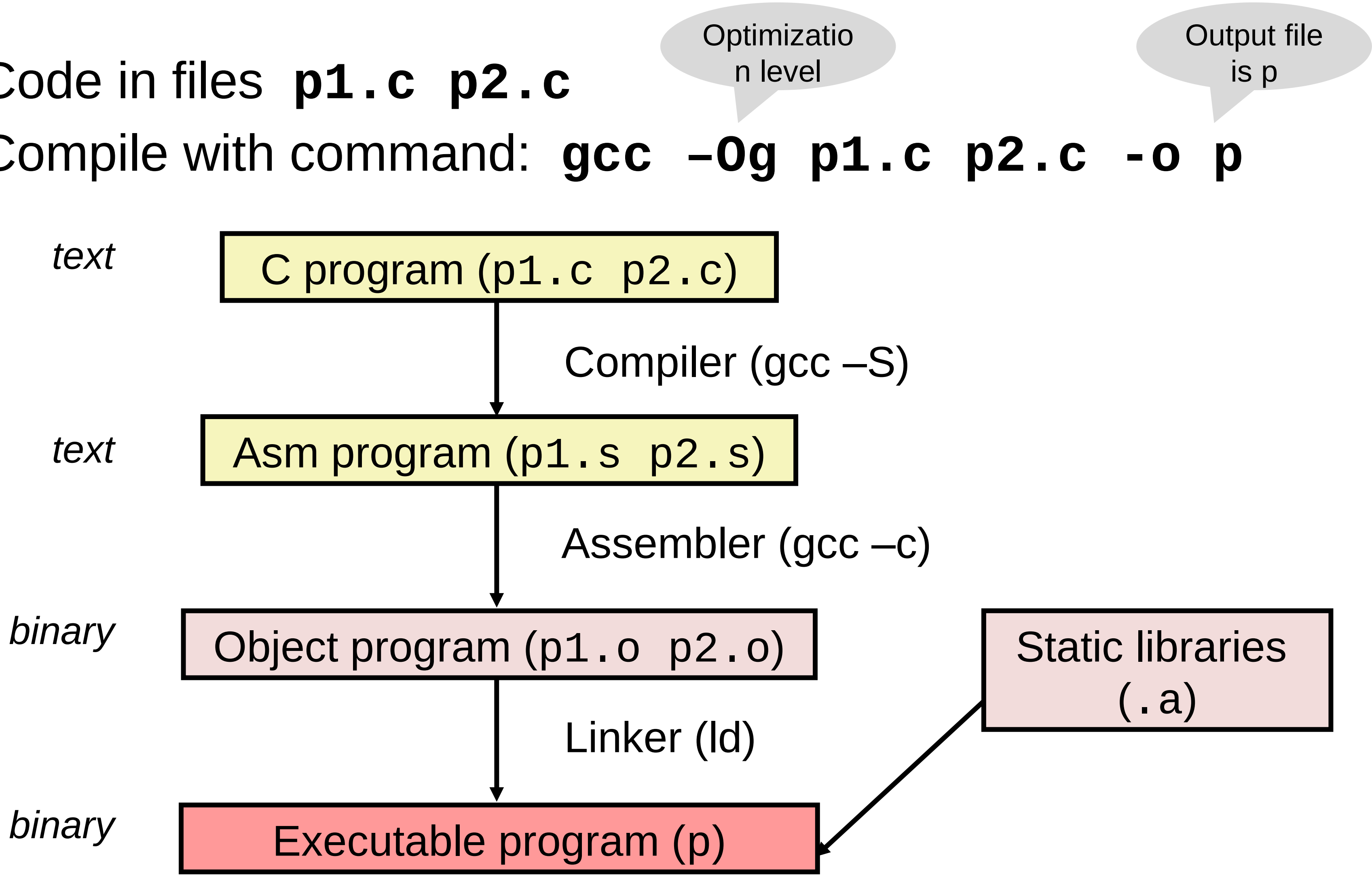
- “Integer” data of 1, 2, or 4 bytes
  - Represent either data value
  - or address
- Floating point data of 4, 8, or 10 bytes
- Code: Byte sequences encoding series of instructions
- No arrays or structures

3 Kind of Assembly Operations

- Perform arithmetic on register or memory data
  - Add, subtract, multiplication…
- 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

Turning C into Object Code

- Code in files **`p1.c p2.c`**
- 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 different machines due to different versions of gcc and different compiler settings.

Object Code

Code for `sumstore`

```
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

```
*dest = t;
```

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

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

- C Code (look at sum.c 2 slides back)
  - Store value **`t`** where designated by **`dest`**
- Assembly
  - Move 8-byte value to memory
    - Quad words in x86-64 parlance
  - Operands:

    `t:` Register **`%rax`**

    `dest:` Register **`%rbx`**

    `*dest:` Memory **`M[%rbx]`**
- Object Code
  - 3-byte instruction
  - Stored at address **`0x40059e`**

Disassembling Object Code

Disassembled

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

- Disassembler

  objdump –d sum

  – Useful tool for examining object code

  – Analyzes bit pattern of series of instructions

  – Produces approximate rendition of assembly code

  – Can be run on either `a.out` (complete executable) or `.o` file

Alternate Disassembly

Object

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

Disassembled

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

- Within gdb Debugger

  `gdb sum`

  `disassemble sumstore`

  – Disassemble procedure

  `x/14xb sumstore`

  – Examine the 14 bytes starting at `sumstore`

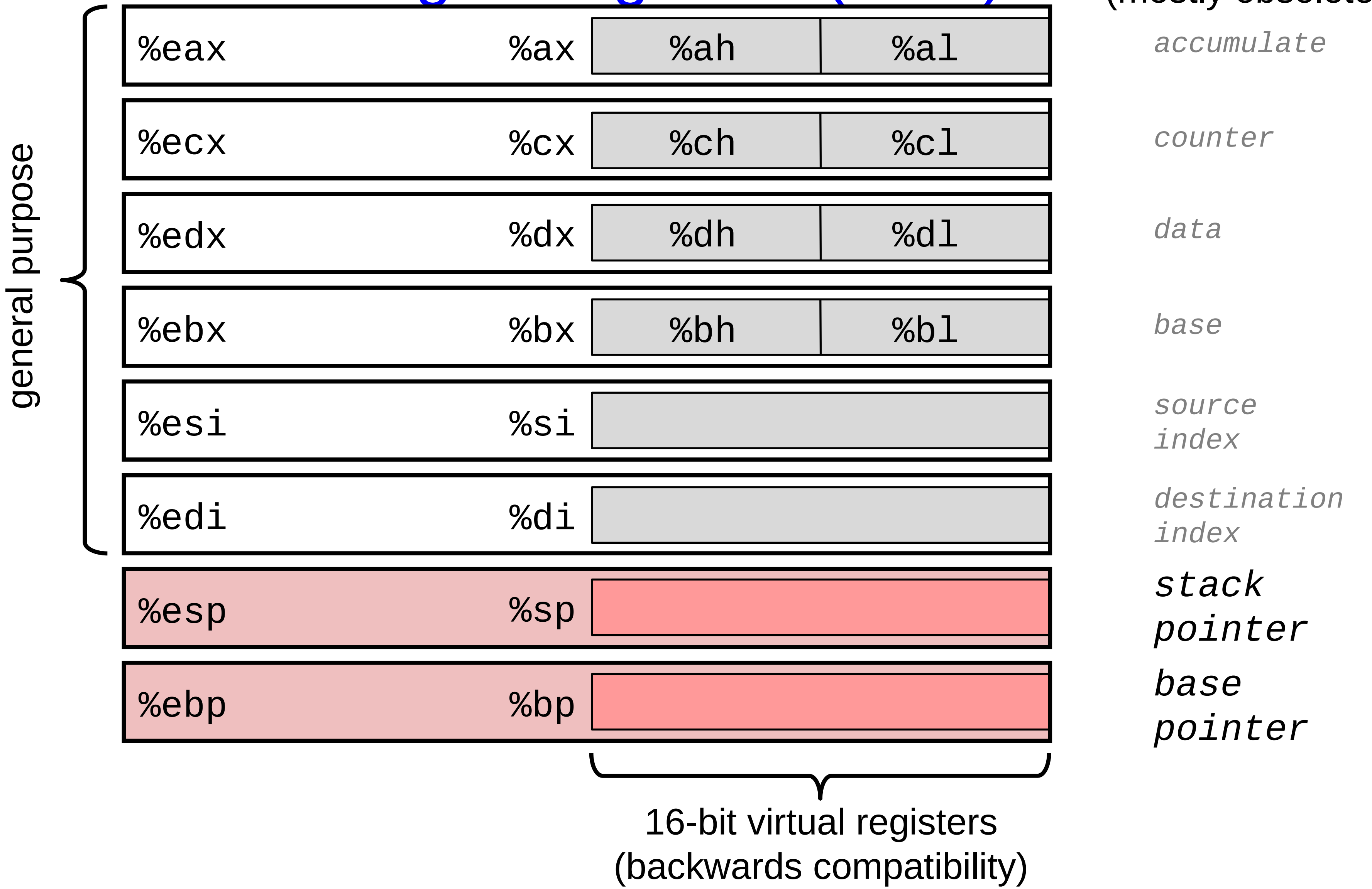
x86-64 Integer Registers

%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: Integer Registers (IA32)

Origin
(mostly obsolete)



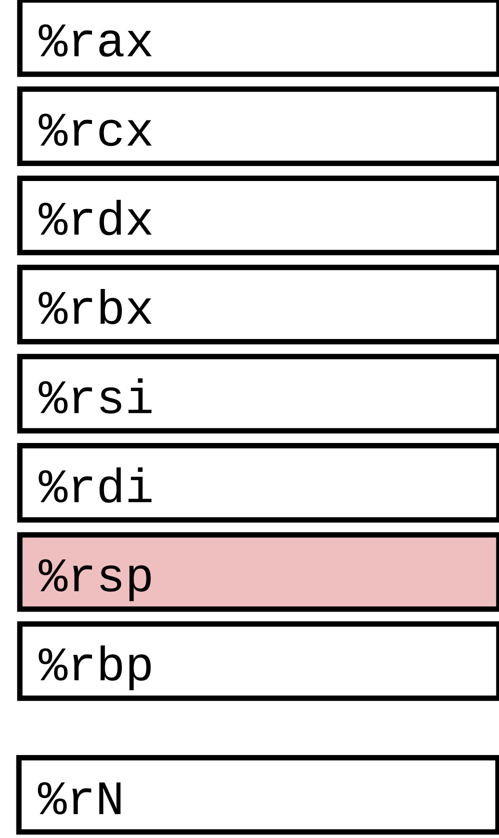
Moving Data

Moving Data

- Moving Data

  movq ***Source*, *Dest***

- Operand Types
  - ***Immediate:*** Constant integer data
    - Example: **$0x400**, **$-533**
    - Like C constant, but prefixed with **'$'**
  - ***Register:*** One of 16 integer registers
    - Example: **%rax, %r13**
    - But **%rsp** reserved for special use
    - Others have special uses for particular instructions (later on that)
  - ***Memory:*** 8 consecutive bytes of memory at address given by register
    - Simplest example: **(%rax)**
    - We will see various other "address modes" later.



`movq` Operand Combinations

	Source	Dest	Src,Dest	C Analog
`movl`	*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;`

No memory-to-memory instruction

movq

C Declaration	Intel Data Type	Assembly code suffix	Size (bytes)
Char	Byte	b	1
Short	Word	w	2
Int	Double Word	l	4
Long	Quad Word	q	8
Pointer	Quad Word	q	8

Special Type of mov

- movz S, R □ R = ZeroExtend(S)
  - movzbw
  - movzbl
  - movzbq
  - movzwl
  - movzwq
- movs S, R □ R = SignExtend(S)
  - movsbw
  - movsbl
  - movsbq
  - movswl
  - movswq
  - movslq
- S: memory or register R: register

Yet Another Special Case

- You can have movq $num, %register
  – num is an immediate number (signed)
  – register is any 64-bit register
  – num cannot exceed 32 bits
  – The rest is sign-extended
- movabsq $num, %register
  – Means move num absolute to register
  – num is 64-bit

Simple Memory Addressing Modes

- Normal (R) □ Mem[Reg[R]]
  - Register R specifies memory address

```
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
```

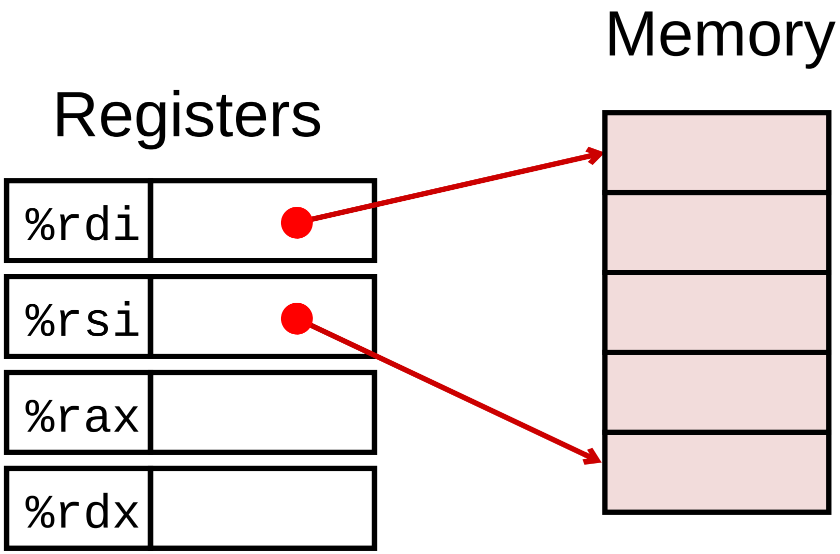
Example of Simple Addressing Modes

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

```
swap:
   … some setup code
   movq    (%rdi), %rax
   movq    (%rsi), %rdx
   movq    %rdx, (%rdi)
   movq    %rax, (%rsi)
   … wrap-up code
   ret
```

Understanding Swap()

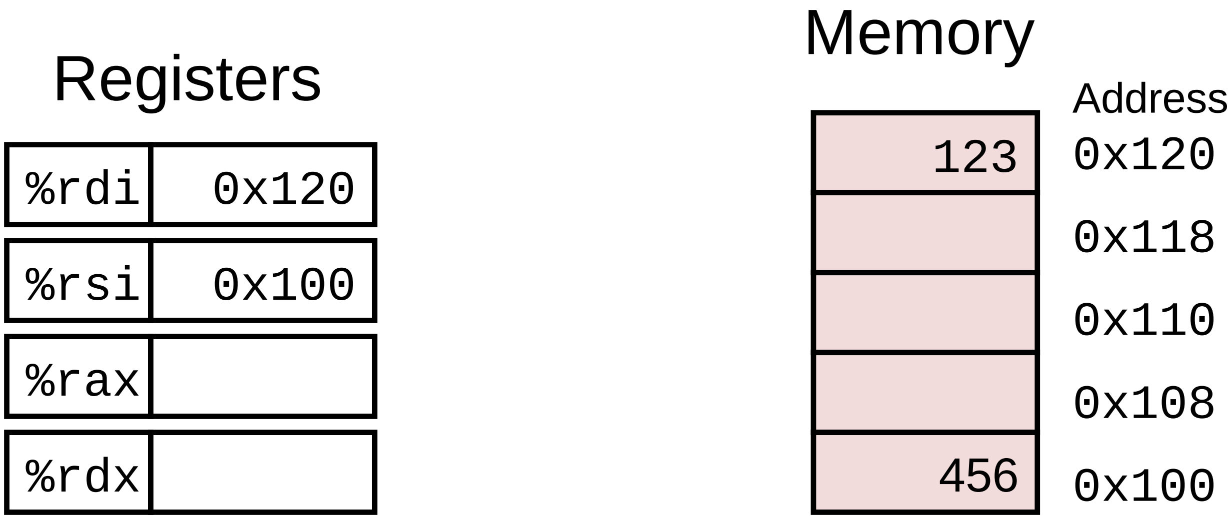
```
void swap
   (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



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

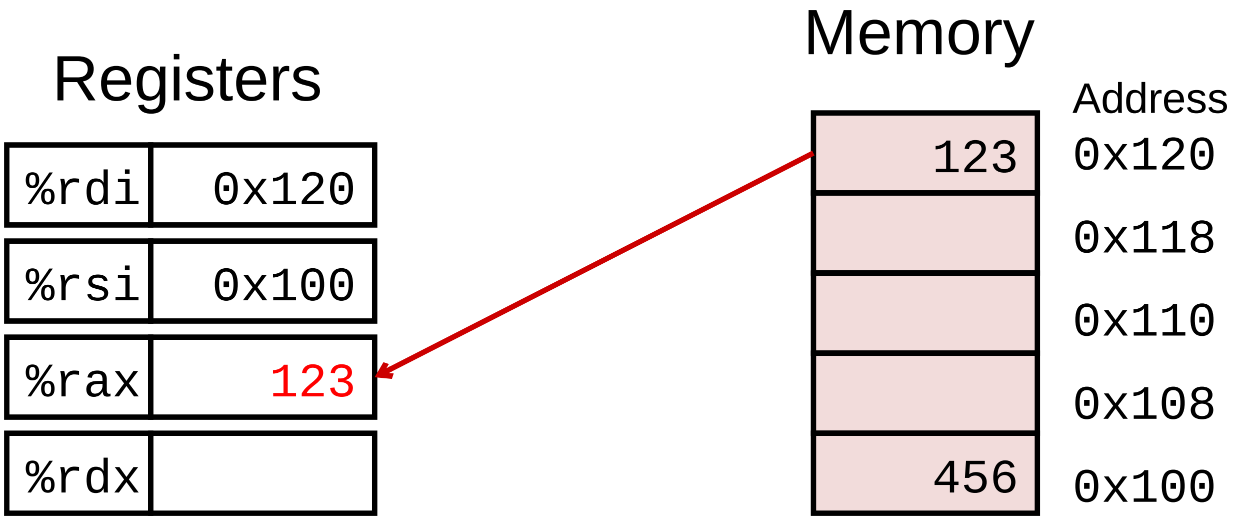
```
swap:
   movq    (%rdi), %rax   # t0 = *xp
   movq    (%rsi), %rdx   # t1 = *yp
   movq    %rdx, (%rdi)   # *xp = t1
   movq    %rax, (%rsi)   # *yp = t0
   ret
```

Understanding Swap()



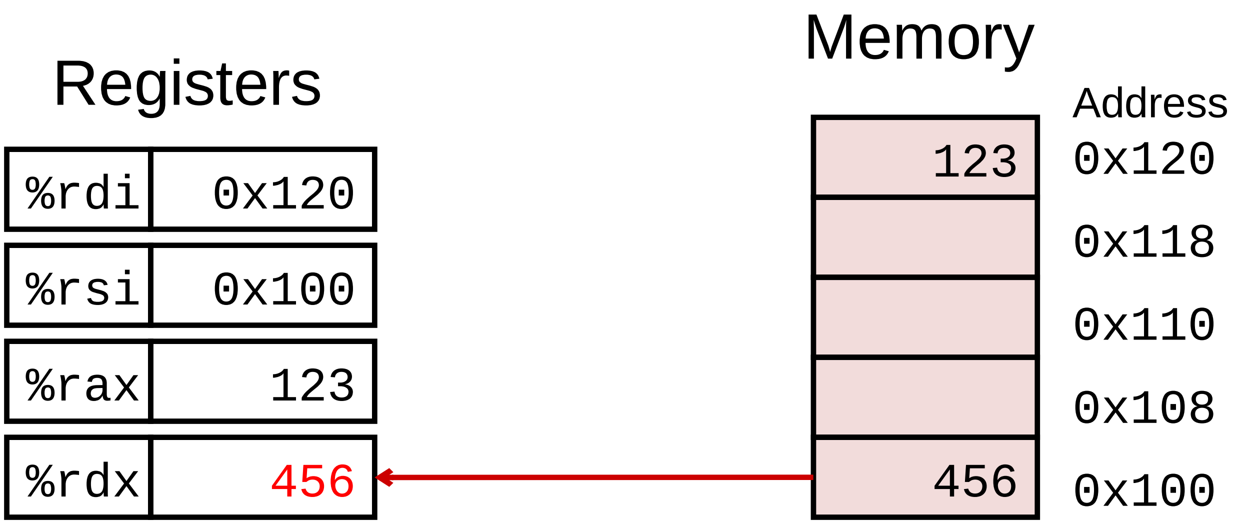

```
swap:
   movq    (%rdi), %rax   # t0 = *xp
   movq    (%rsi), %rdx   # t1 = *yp
   movq    %rdx, (%rdi)   # *xp = t1
   movq    %rax, (%rsi)   # *yp = t0
   ret
```

Understanding Swap()



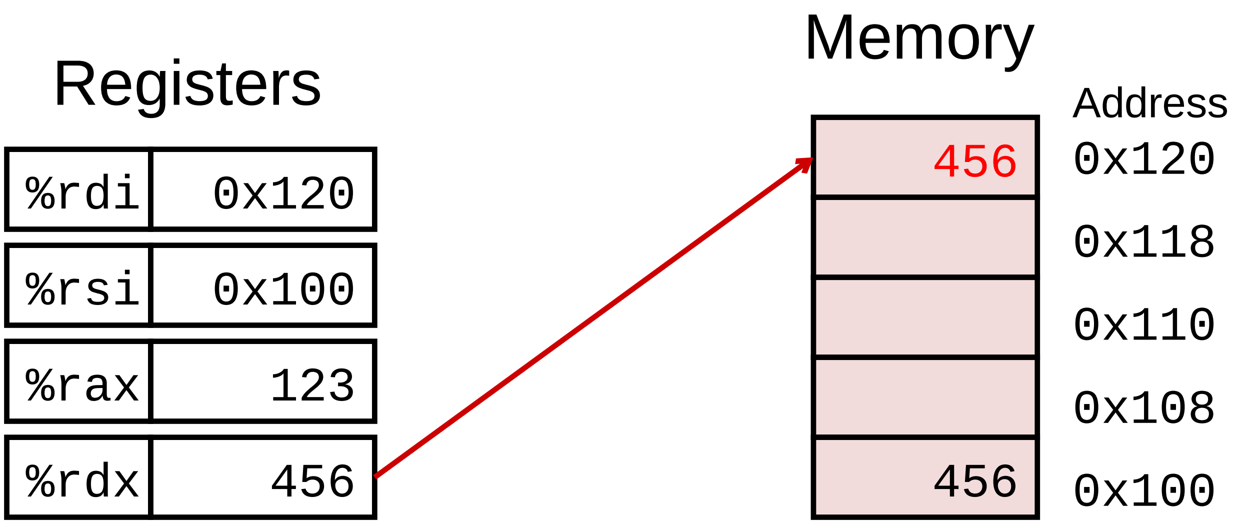

```
swap:
   movq     (%rdi), %rax   # t0 = *xp
   movq     (%rsi), %rdx   # t1 = *yp
   movq     %rdx, (%rdi)   # *xp = t1
   movq     %rax, (%rsi)   # *yp = t0
   ret
```

Understanding Swap()



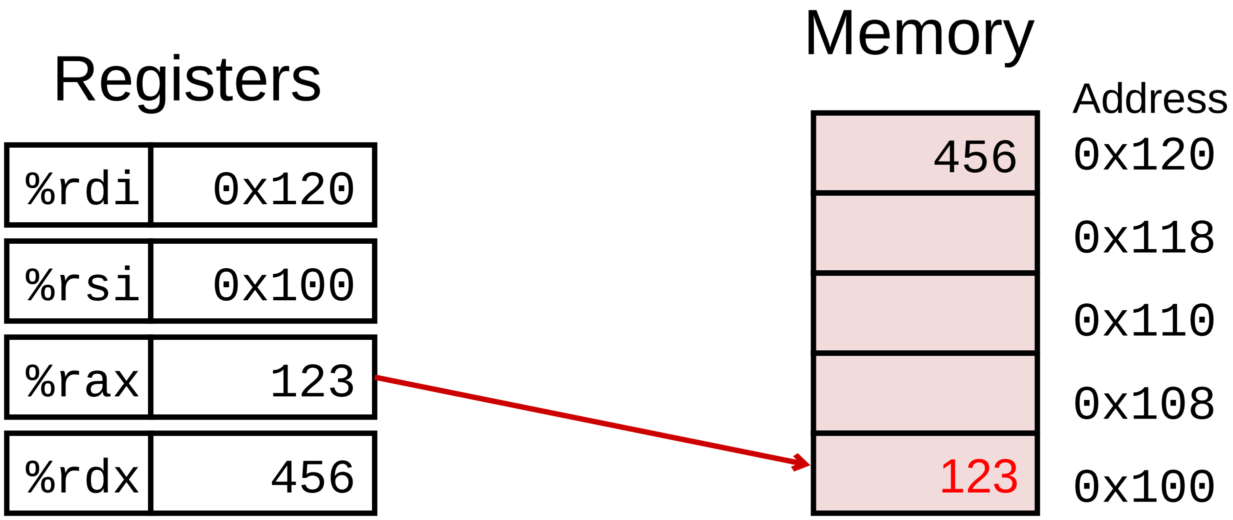

```
swap:
    movq    (%rdi), %rax   # t0 = *xp
    movq    (%rsi), %rdx   # t1 = *yp
    movq    %rdx, (%rdi)   # *xp = t1
    movq    %rax, (%rsi)   # *yp = t0
    ret
```

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
```

General Memory Addressing Modes

- Most General Form

D (Rb, Ri, S)

Constant displacement (cannot be bigger than 4 bytes)

Base register (any of the 16 registers)

Index register (any if the 16 registers except %rsp)

Scale (1,2,4,8)

Mem[Reg[Rb]+S*Reg[Ri]+ D]

- 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
0x80(,%rdx,2)		

Address Computation Instruction

- leaq Src, Dst
  - Src is address mode expression
  - Set Dst to address calculated for src
- Uses
  - Computing addresses <u>without a memory access</u>
    - 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
```

Conclusions

- History of Intel processors and architectures
  - Evolutionary design leads to many quirks and artifacts
- C, assembly, machine code
  - Compiler must transform statements, expressions, procedures into low-level instruction sequences
- Assembly Basics: Registers, operands, move
  - The x86 move instructions cover wide range of data movement forms