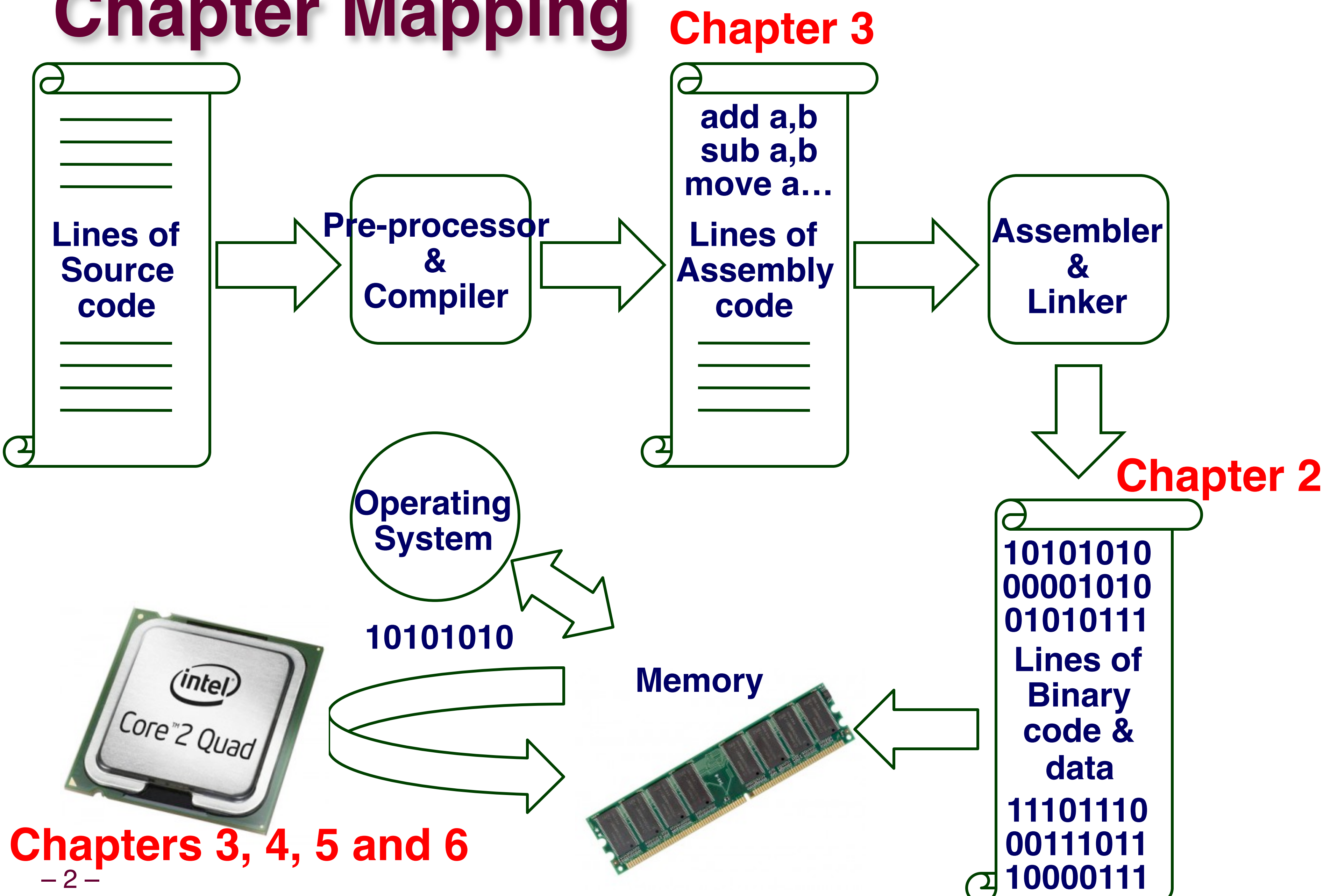
Chapter 3:

Assembly Language Programming I

Topics

- Assembly Programmer's Execution Model
- Accessing Information
  - Registers
  - Memory

Chapter Mapping



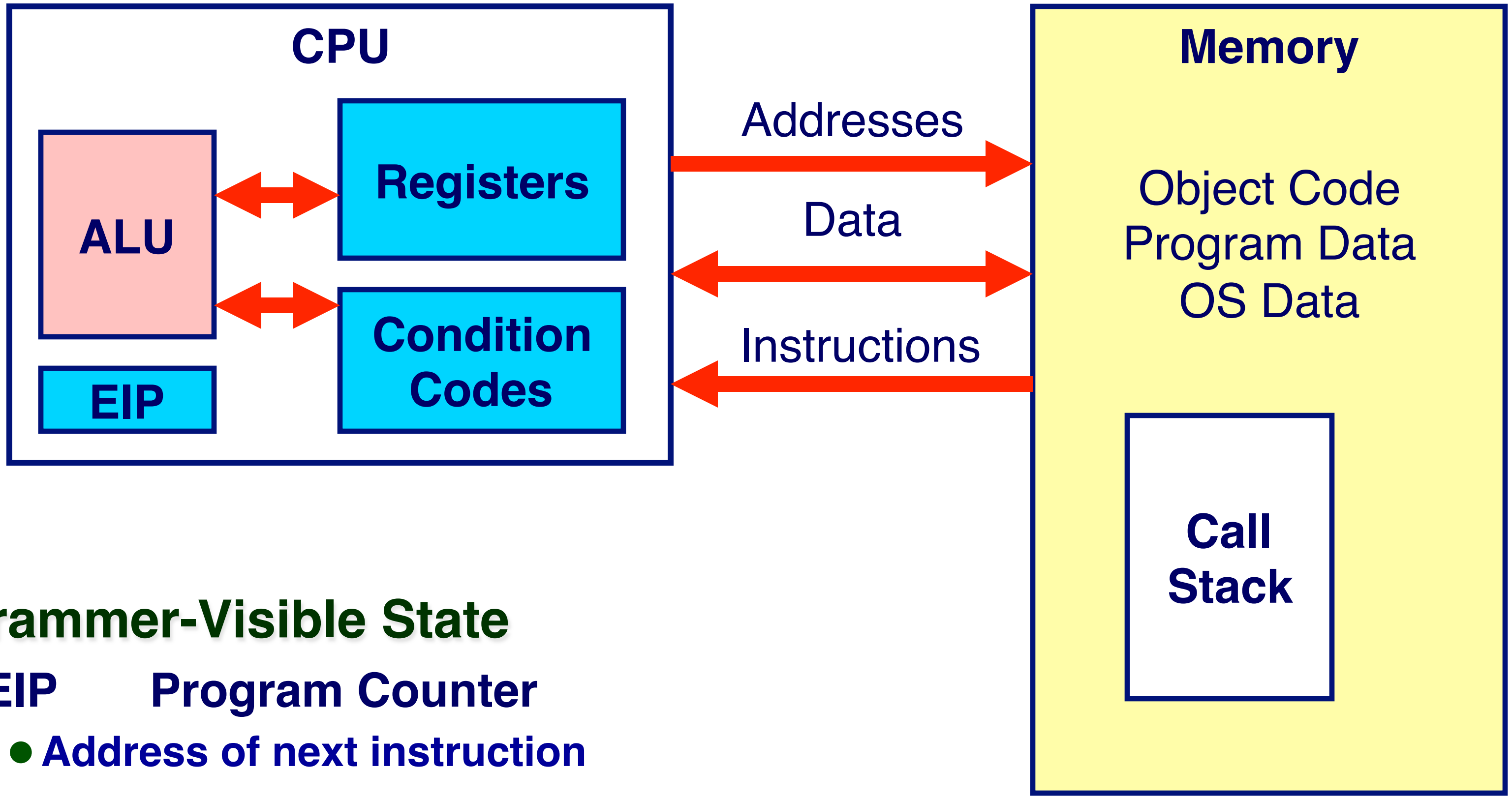
Intel x86 Processors

- Totally dominate PC computer market
- Evolutionary design
  - Backwards compatible up until 8086 16-bit CPU, introduced in 1978
  - Then 80286, 80386, 80486, Pentium, …, Intel Core i7 – hence the name x86
  - 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!

Assembly Language

- **Specific to a CPU**
  - We will be using x86 assembly language
  - ARM processors will have a different assembly language, etc.
  - 32-bit processors will have different assembly language than 64-bit processors
- **Different styles for x86-64 assembly code**
  - We will be using gcc/GNU-style assembly language
  - There is also the Intel style of assembly language
    - Example: switches order of source and destination compared to gcc/GNU

Assembly Programmer's View

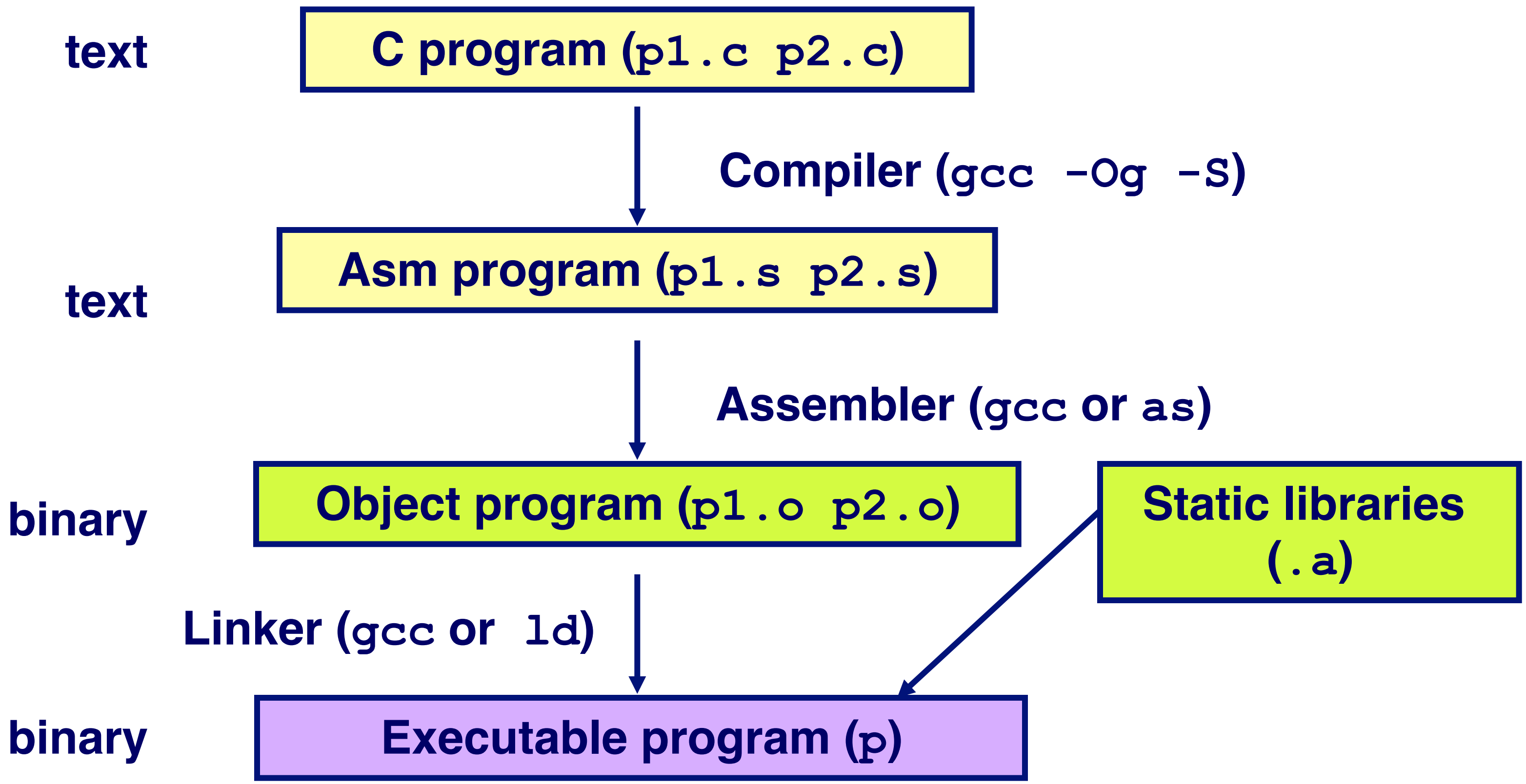


Programmer-Visible State

- EIP   Program Counter
  - 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, (some) OS data
  - Includes call stack used 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

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 (on VM) with command

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

Produces file `sum.s`

Note how assembly maps to C code

Note: May get very different results on other machines, even other Linux machines, due to different versions of gcc and different compiler settings

Object Code sum.o

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

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`

What Can be Disassembled?

```
% objdump -d WINWORD.EXE

WINWORD.EXE:    file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:
30001001:
30001003:
30001005:
3000100a:
```

Reverse engineering forbidden by
Microsoft End User License Agreement

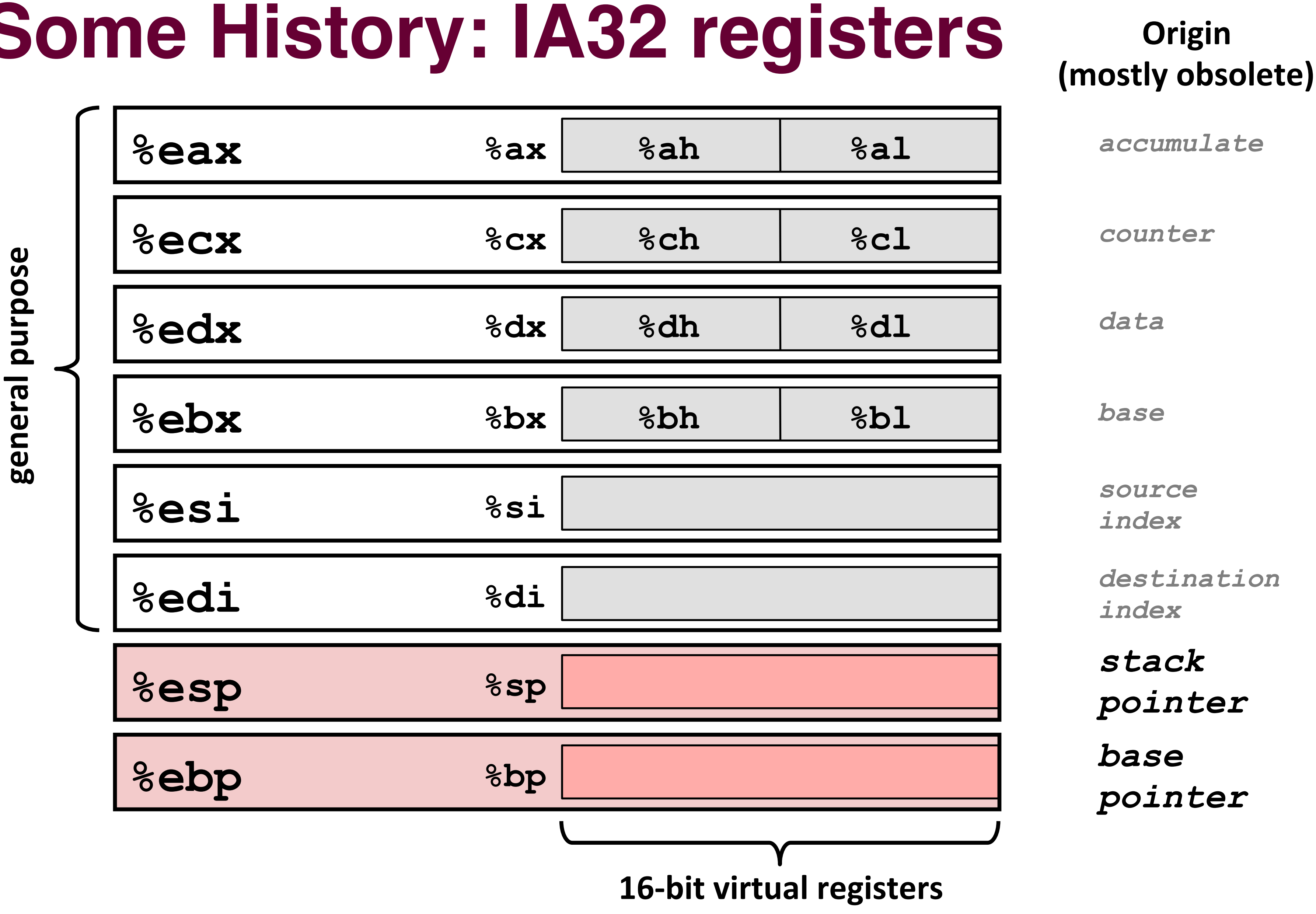
- Anything that can be interpreted as executable code
- Disassembler examines bytes and reconstructs assembly source

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

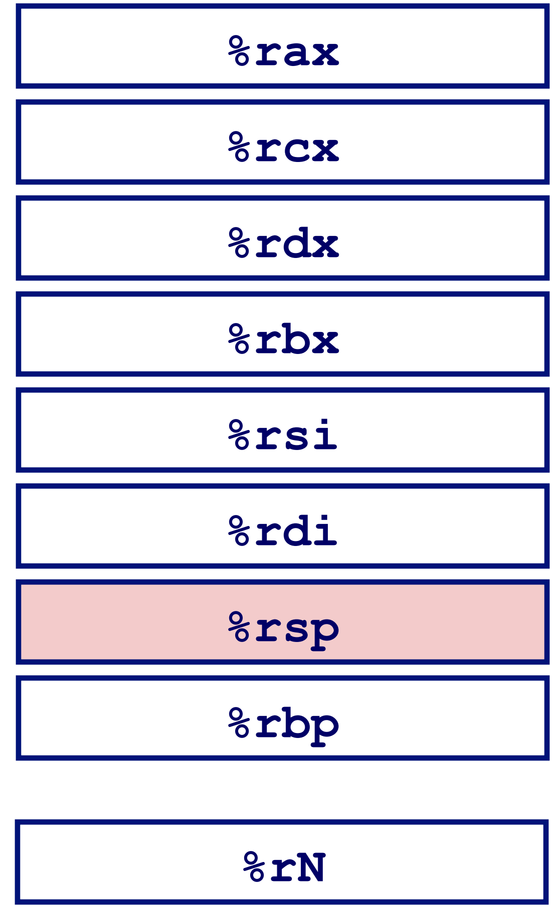
- ## Moving Data

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

  - **Move 8-byte ("quad") word**
  - **Lots of these in typical code**

- ## Operand Types

  - ***Register:* One of 16 integer registers**
    - **Example: `%rax, %r13`**
    - **But `%rsp` reserved for special use**
    - **Others have special uses for particular instructions**

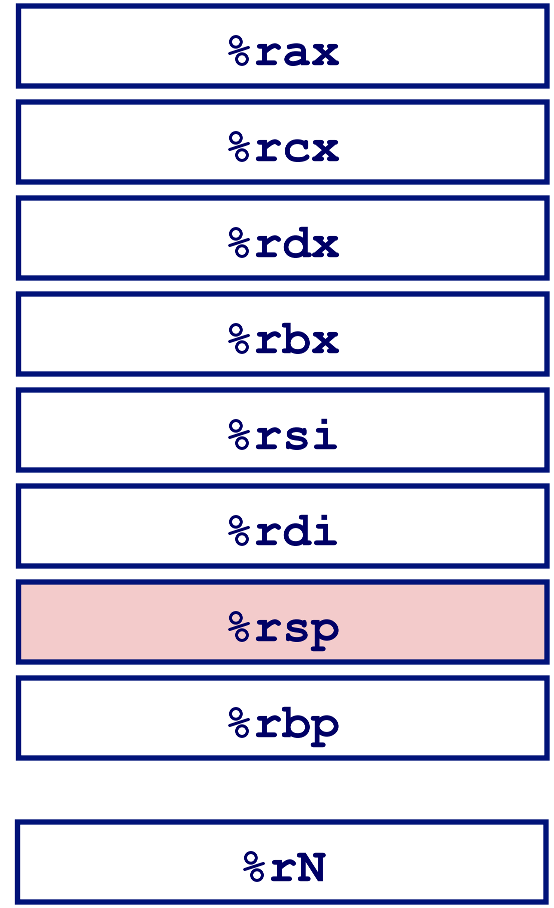


Moving Data

- **Moving Data**

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

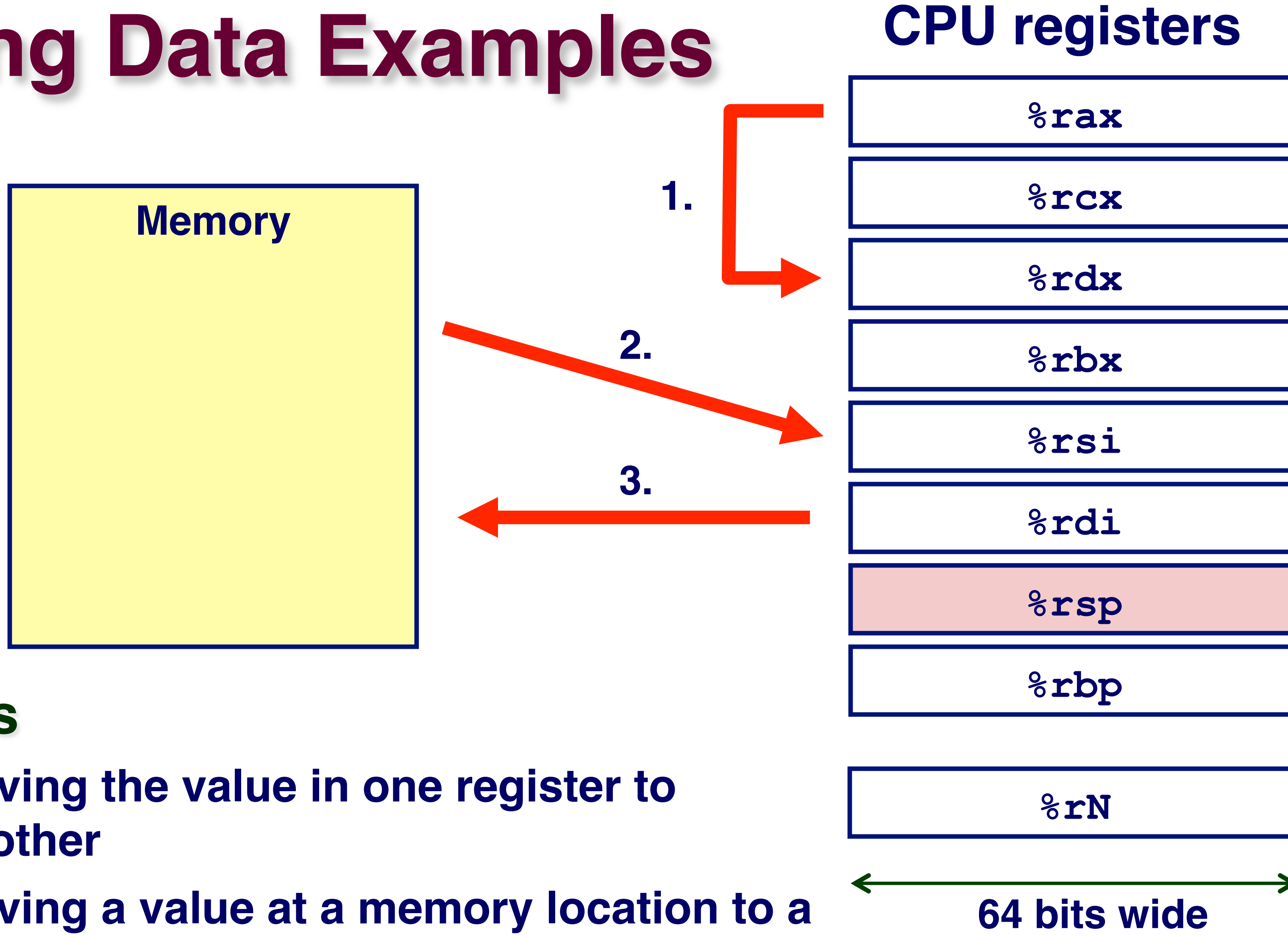
- **Operand Types**
  - ***Immediate:*** **Constant integer data**
    - **Example: `$0x400`, `$-533`**
    - **Like C constant, but prefixed with `'$'`**
    - **Encoded with 1, 2, or 4 bytes**
  - ***Memory:*** **8 consecutive bytes of memory at address given by register**
    - **Simplest example: `(%rax)`**
    - **Various other "address modes"**



Representing Instructions

- For historical reasons (16-bit processors), Intel terminology considers a "word" to be 16 bits long
  - '`movw %ax, %dx`', where the '`w`' in `movw` implies a 16 bit quantity is about to be moved
  - '`movl %eax, %edx`', the 'l' in `movl` implies a "long" 32-bit quantity is about to be moved.
  - See text for more `mov` instructions: `movb, movw, movl, movsbw, movsbl, movswl, movzbw, movzbl, movzwl`
- Does the width of a C 'long' int == an x86 assembly language 'long' word?
  - Yes, they're both 32-bits or 4 bytes on a 32-bit x86 machine
  - No, a C 'long' is 64-bits on a 64-bit x86 machine, while an x86 assembly language 'long' is still 32-bits wide
    - i.e. a `movl` on a 64-bit machine is still going to move a 32-bit quantity

Moving Data Examples



Examples

1. Moving the value in one register to another
2. Moving a value at a memory location to a register
3. Moving a register value to a memory location

`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 transfers with single instruction
  - i.e. can't do: `movq (%rax), (%rdx)`

Simple 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
```

Go to memory address %rbp+8
and fetch the data located there

Supplementary Slides