MACHINE-LEVEL PROGRAMMING I: BASICS

CS 429H: SYSTEMS I

Instructor:
Emmett Witchel

Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move

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.

Intel x86 Evolution: Milestones

Name	Date	Transistors	MHz
8086	1978	29K	5-10

- First 16-bit processor. Basis for IBM PC & DOS
- 1MB address space

Name	Date	Transistors	MHz
386	1985	275K	16-33

- First 32 bit processor , referred to as IA32
- Added "flat addressing"
- Capable of running Unix
- 32-bit Linux/gcc uses no instructions introduced in later models

Name	Date	Transistors	MHz
Pentium 4F	2004	125M	2800-3800

- First 64-bit processor, referred to as x86-64

Name	Date	Transistors	MHz
Core i7	2008	731M	2667-3333

Intel x86 Processors: Overview

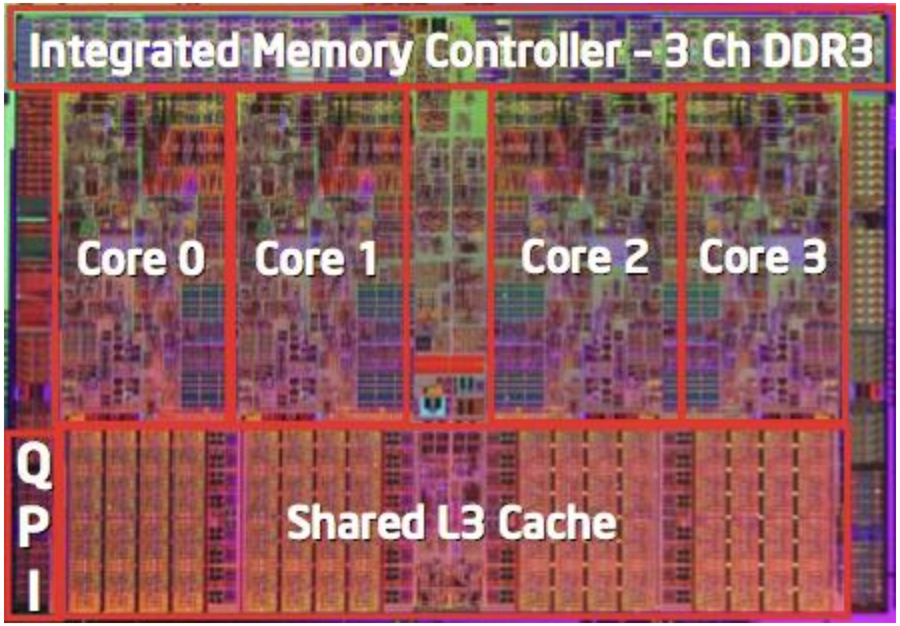
Architectures	Processors
X86-16	**8086**
	286
X86-32/IA32	**386**
	486
	Pentium
MMX	**Pentium MMX**
SSE	**Pentium III**
SSE2	**Pentium 4**
SSE3	**Pentium 4E**
X86-64 / EM64t	**Pentium 4F**
	Core 2 Duo
SSE4	**Core i7**

time

IA: often redefined as latest Intel architecture

Intel x86 Processors, contd.

- Machine Evolution
  - 386 1985
  - Pentium 1993
  - Pentium/MMX 1997
  - PentiumPro 1995
  - Pentium III 1999
  - Pentium 4 2001
  - Core 2 Duo 2006
  - Core i7 2008



- Added Features
  - Instructions to support multimedia operations
    - Parallel operations on 1, 2, and 4-byte data, both integer & FP
  - Instructions to enable more efficient conditional operations
- Linux/GCC Evolution
  - Two major steps: 1) support 32-bit 386. 2) support 64-bit x86-64

x86 Clones: Advanced Micro Devices (AMD)

- Historically
  - AMD has followed just behind Intel
  - A little bit slower, a lot cheaper
- Then
  - Recruited top circuit designers from Digital Equipment Corp. and other downward trending companies
  - Built Opteron: tough competitor to Pentium 4
  - Developed x86-64, their own extension to 64 bits

Intel's 64-Bit

- Intel Attempted Radical Shift from IA32 to IA64
  - Totally different architecture (Itanium)
  - Executes IA32 code only as legacy
  - Performance disappointing
- AMD Stepped 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

Our Coverage

- IA32
  - The traditional x86
- x86-64/EM64T
  - The emerging standard
- Presentation
  - Book presents IA32 in Sections 3.1—3.12
  - Covers x86-64 in 3.13

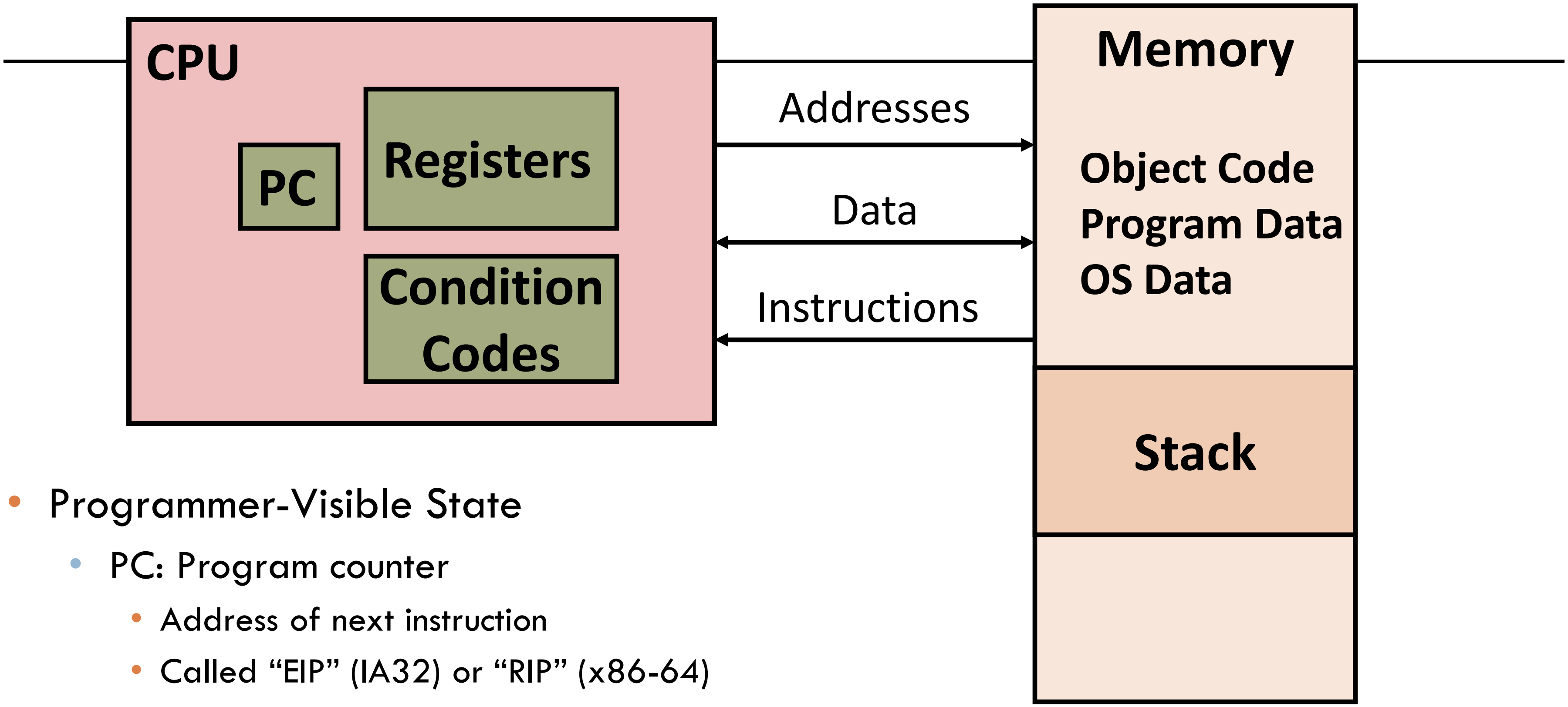
Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move

Definitions

- Architecture: (also instruction set architecture: ISA) The parts of a processor design that one needs to understand to write assembly code.
  - Examples: instruction set specification, registers.
- Microarchitecture: Implementation of the architecture.
  - Examples: cache sizes and core frequency.
- Example ISAs (Intel): x86, IA, IPF

Assembly Programmer's View



- Programmer-Visible State
  - PC: Program counter
    - Address of next instruction
    - Called "EIP" (IA32) or "RIP" (x86-64)
  - 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 stack used to support procedures

Program to Process

- We write a program in e.g., C.
- A compiler turns that program into an instruction list.
- The CPU interprets the instruction list (which is more a graph of basic blocks).

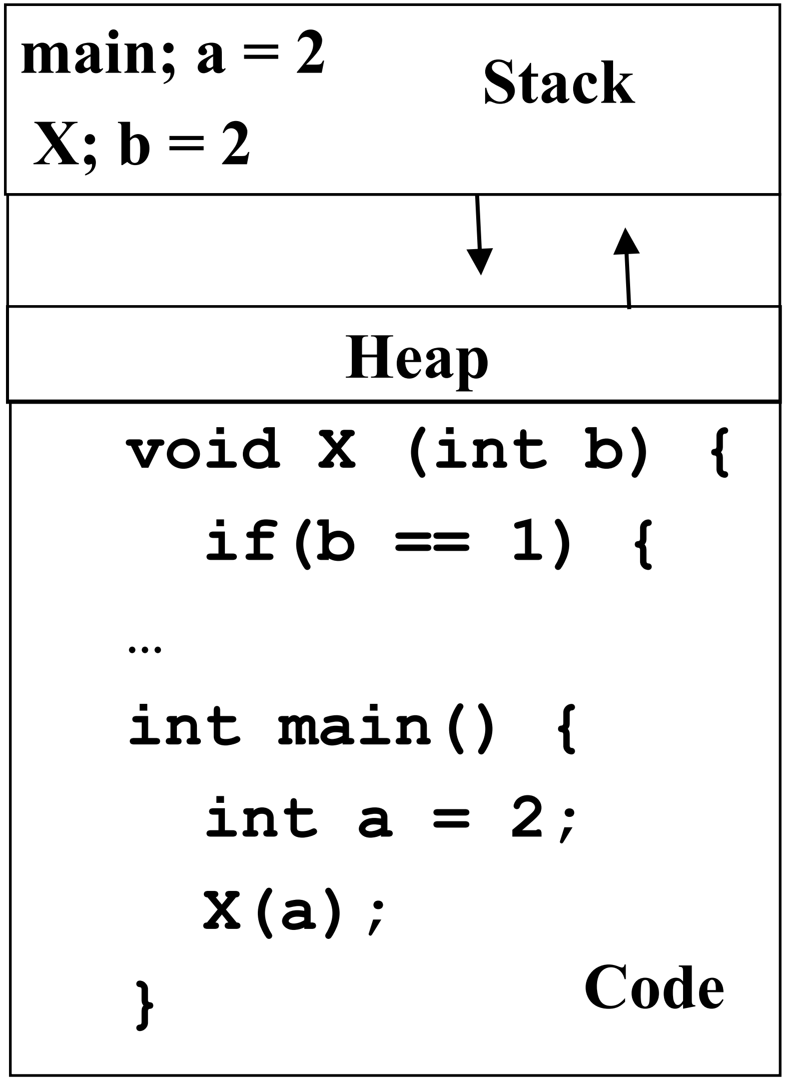
```
void X (int b) {
    if(b == 1) {
...
int main() {
   int a = 2;
   X(a);
}
```

Process in Memory

- What is in memory.

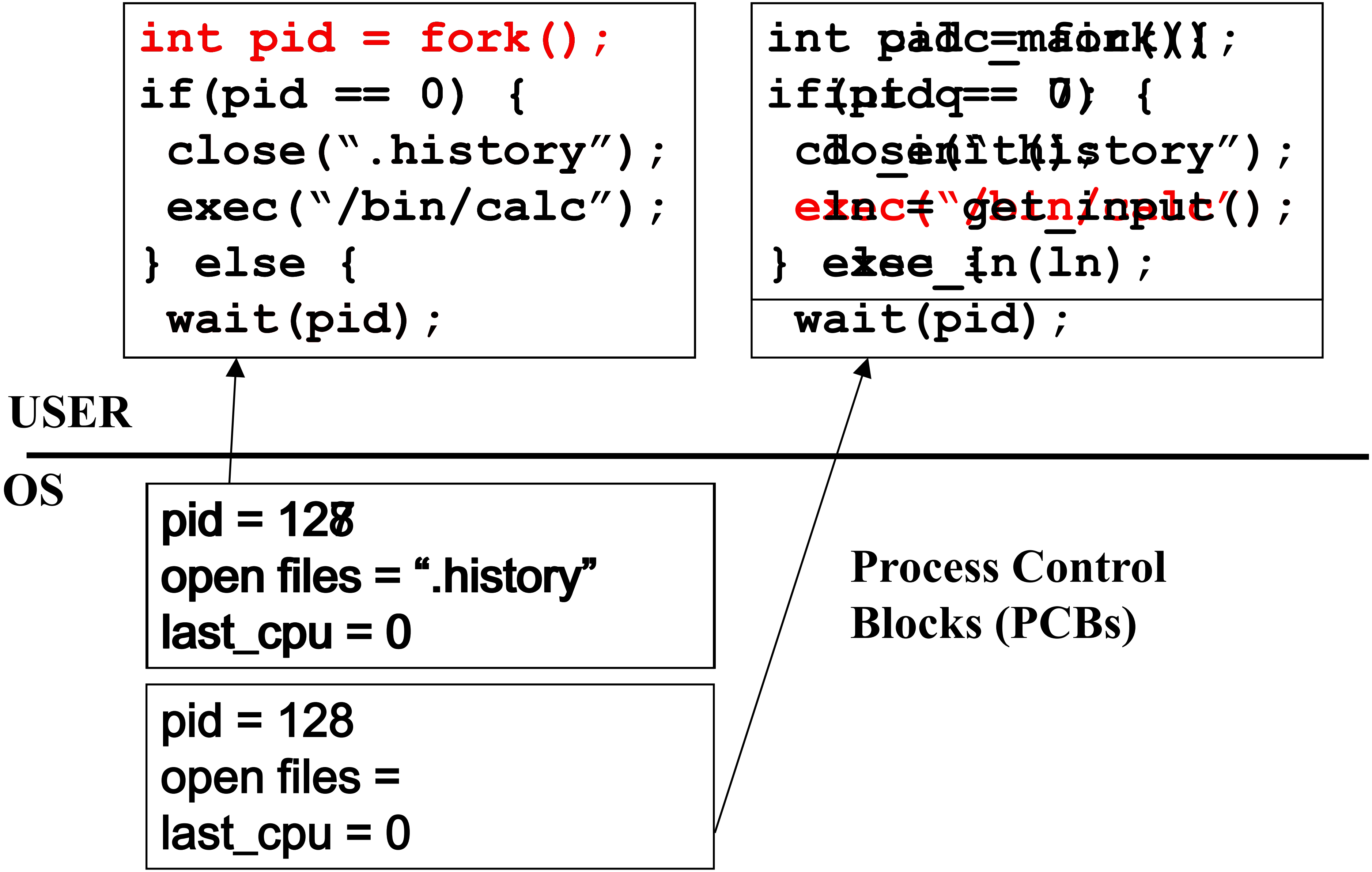
- Program to process.
  - What you wrote

```
void X (int b) {
   if(b == 1) {
…
int main() {
   int a = 2;
   X(a);
}
```

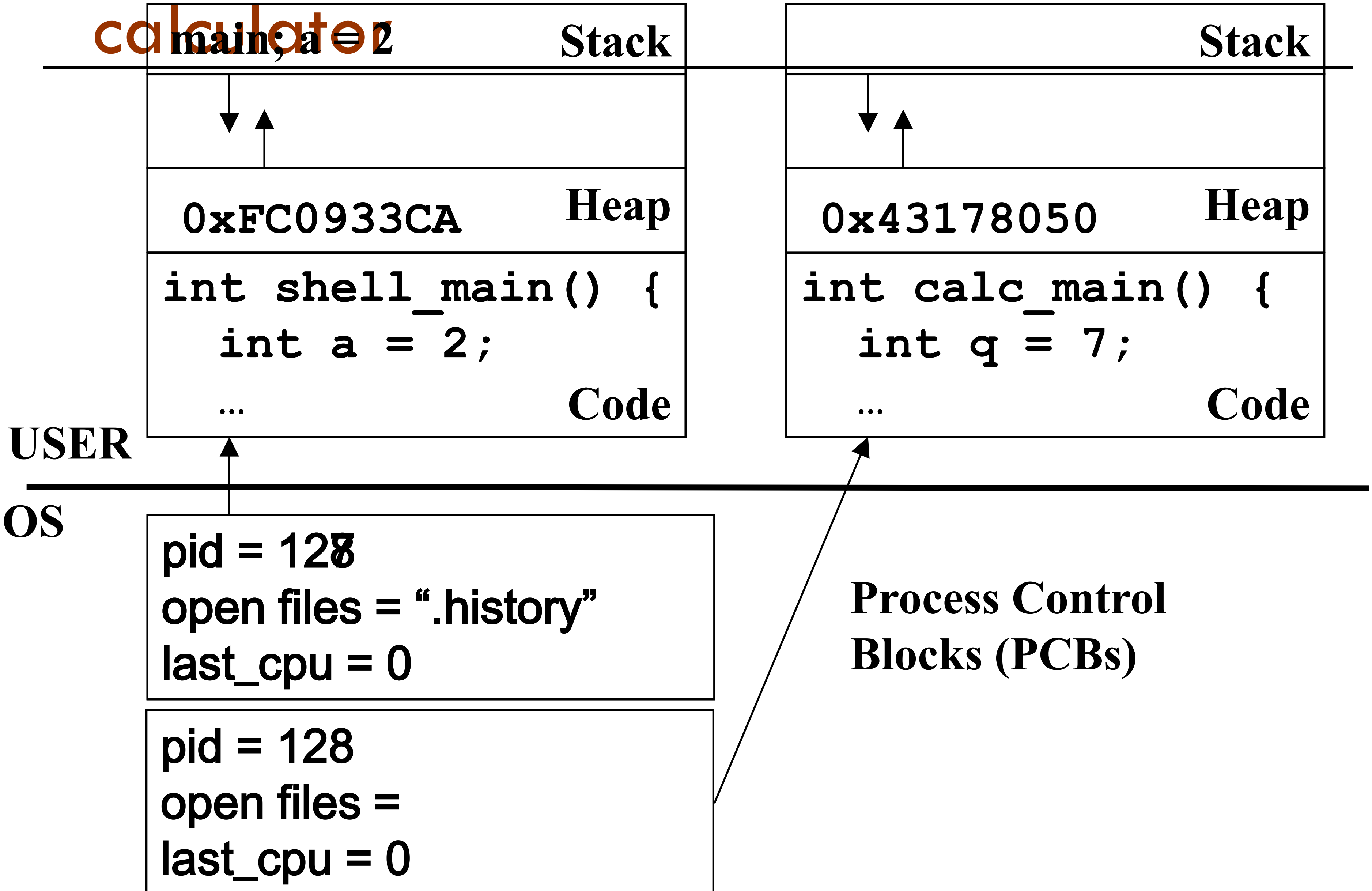


- What must the OS track for a process?

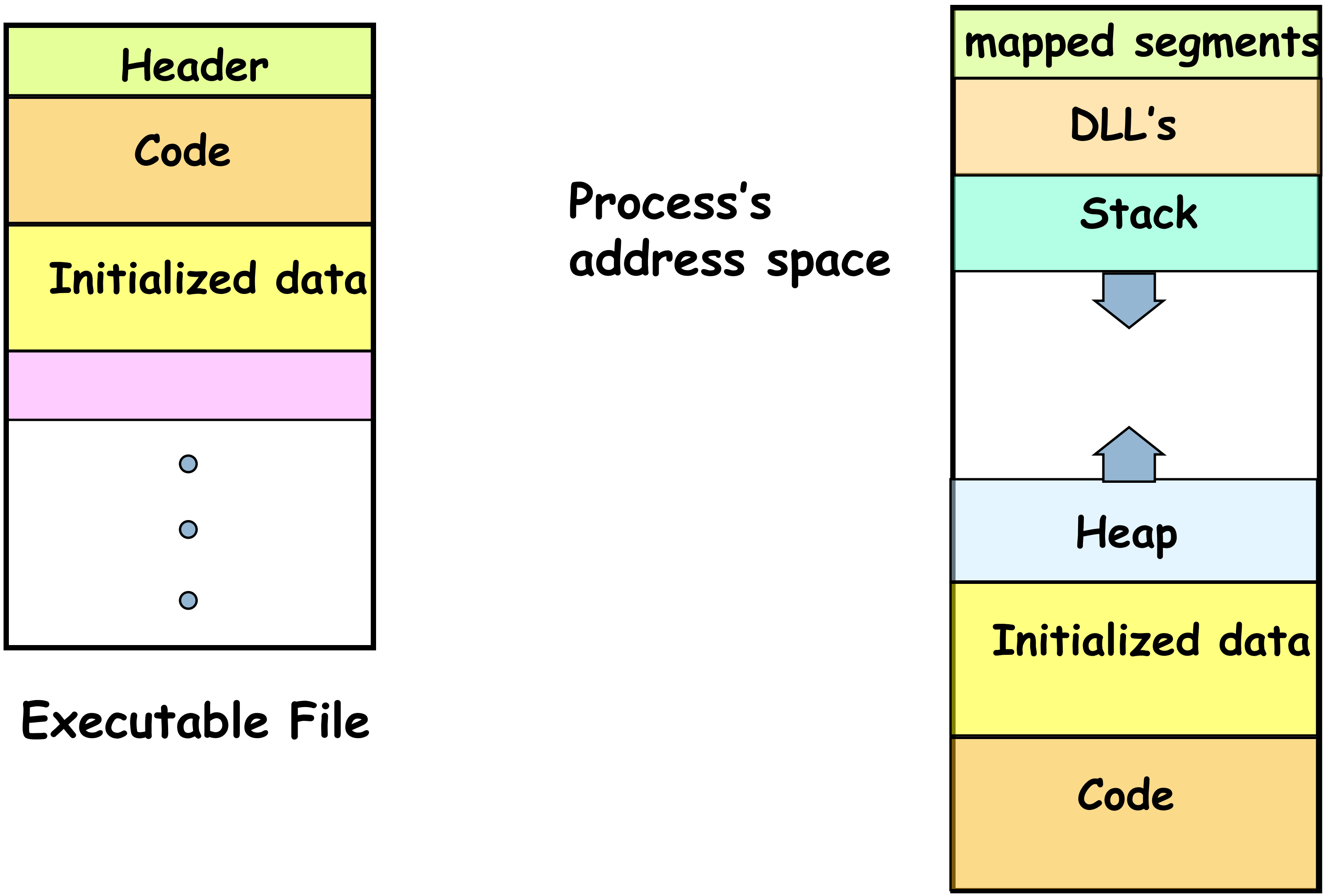
A shell forks and execs a calculator



A shell forks and then execs a calculator

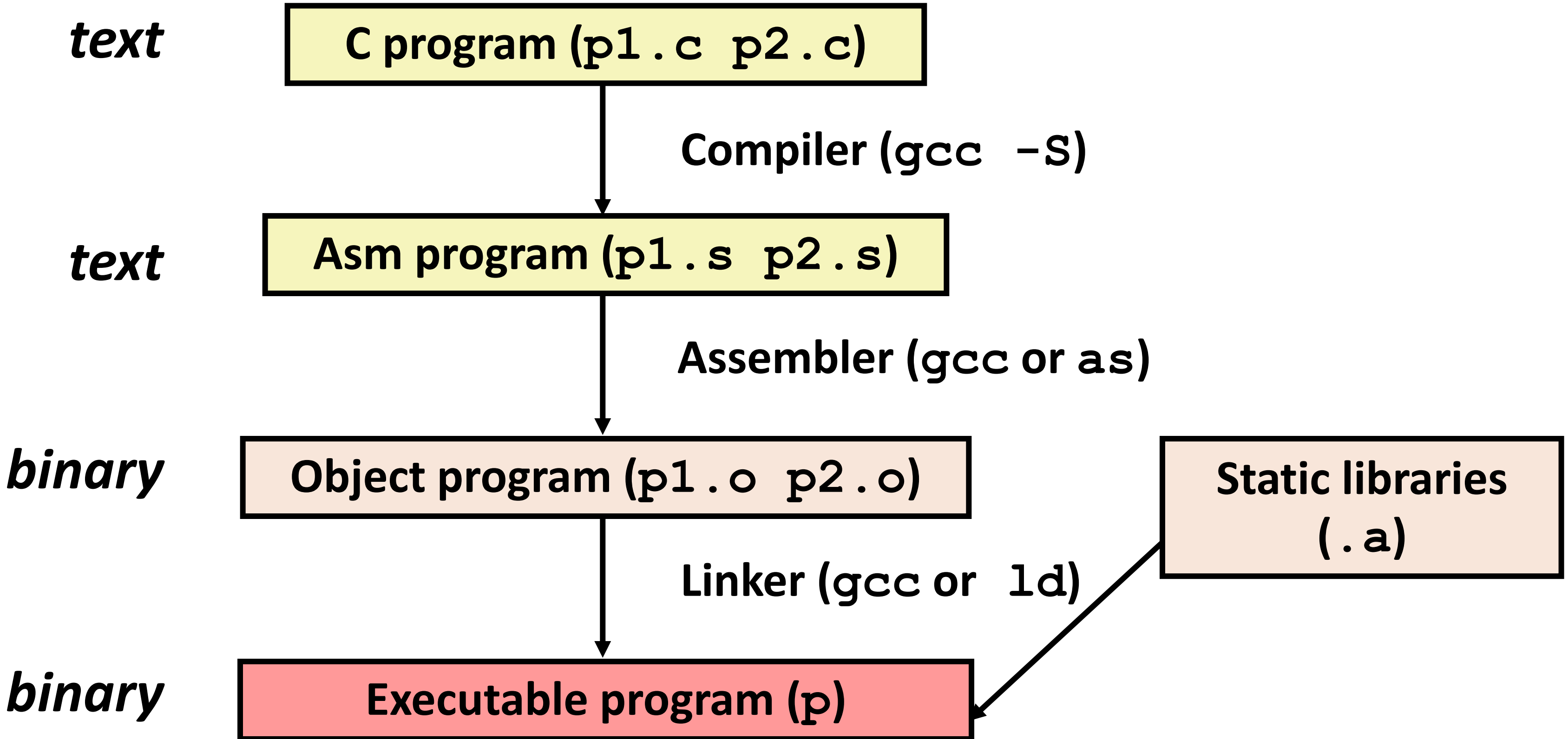


Anatomy of a Process



Turning C into Object Code

- Code in files `p1.c p2.c`
- Compile with command: `gcc –O1 p1.c p2.c -o p`
  - Use basic optimizations (`-O1`)
  - Put resulting binary in file `p`



Compiling Into Assembly

C Code

```
int sum(int x, int y)
{
   int t = x+y;
   return t;
}
```

Generated IA32 Assembly

```
sum:
    pushl %ebp
    movl %esp,%ebp
    movl 12(%ebp),%eax
    addl 8(%ebp),%eax
    popl %ebp
    ret
```

Some compilers use instruction "`leave`"

Obtain with command

`/usr/local/bin/gcc –O1 -S code.c`

Produces file `code.s`

Assembly Characteristics: Data Types

- "Integer" data of 1, 2, or 4 bytes
  - Data values
  - Addresses (untyped pointers)

- Floating point data of 4, 8, or 10 bytes

- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory

Assembly Characteristics: 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

Object Code

Code for **`sum`**

```
0x401040 <sum>:
    0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x5d
    0xc3
```

- **Total of 11 bytes**
- **Each instruction 1, 2, or 3 bytes**
- **Starts at address `0x401040`**

- 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

Disassembling Object Code

Disassembled

```
080483c4 <sum>:
 80483c4:   55          push    %ebp
 80483c5:   89 e5       mov     %esp,%ebp
 80483c7:   8b 45 0c    mov     0xc(%ebp),%eax
 80483ca:   03 45 08    add     0x8(%ebp),%eax
 80483cd:   5d          pop     %ebp
 80483ce:   c3          ret
```

- Disassembler

  `objdump -d p`
  - 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

```
0x401040:
    0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x5d
    0xc3
```

Disassembled

```
Dump of assembler code for function sum:
0x080483c4 <sum+0>:      push   %ebp
0x080483c5 <sum+1>:      mov    %esp,%ebp
0x080483c7 <sum+3>:      mov    0xc(%ebp),%eax
0x080483ca <sum+6>:      add    0x8(%ebp),%eax
0x080483cd <sum+9>:      pop    %ebp
0x080483ce <sum+10>:     ret
```

- Within gdb Debugger

  `gdb p`

  `disassemble sum`

  - Disassemble procedure

  `x/11xb sum`

  - Examine the 11 bytes starting at `sum`

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:   55              push   %ebp
30001001:   8b ec           mov    %esp,%ebp
30001003:   6a ff           push   $0xffffffff
30001005:   68 90 10 00 30 push   $0x30001090
3000100a:   68 91 dc 4c 30 push   $0x304cdc91
```

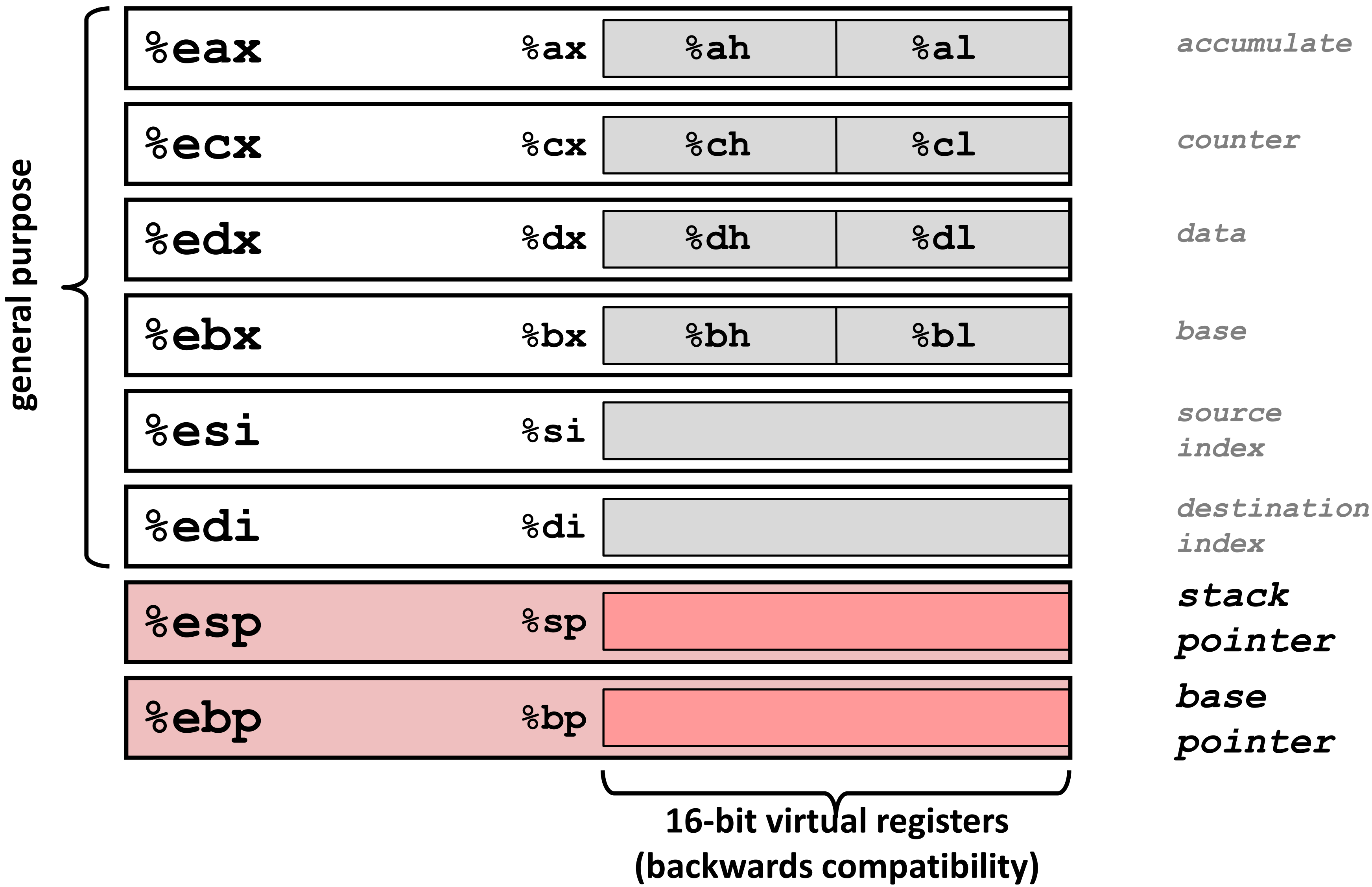
- Anything that can be interpreted as executable code
- Disassembler examines bytes and reconstructs assembly source

Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move

Integer Registers (IA32)

Origin (mostly obsolete)



Simple Memory Addressing Modes

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

```
movl (%ecx),%eax
```

- Displacement D(R) Mem[Reg[R]+D]
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

```
movl 8(%ebp),%edx
```

Using Simple Addressing Modes

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

```
swap:
   pushl %ebp
   movl  %esp,%ebp
   pushl %ebx

   movl  8(%ebp), %edx
   movl  12(%ebp), %ecx
   movl  (%edx), %ebx
   movl  (%ecx), %eax
   movl  %eax, (%edx)
   movl  %ebx, (%ecx)

   popl  %ebx
   popl  %ebp
   ret
```

Set Up (pushl %ebp, movl %esp,%ebp, pushl %ebx)

Body (movl 8(%ebp), %edx through movl %ebx, (%ecx))

Finish (popl %ebx, popl %ebp, ret)

Using Simple Addressing Modes

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

```
swap:
   pushl %ebp
   movl  %esp,%ebp
   pushl %ebx

   movl  8(%ebp), %edx
   movl  12(%ebp), %ecx
   movl  (%edx), %ebx
   movl  (%ecx), %eax
   movl  %eax, (%edx)
   movl  %ebx, (%ecx)

   popl  %ebx
   popl  %ebp
   ret
```

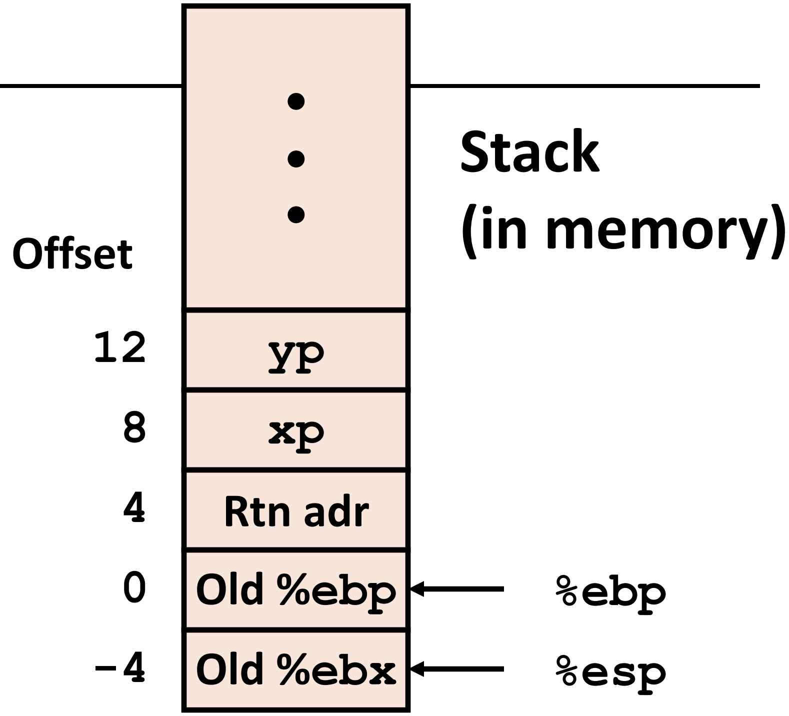
Set Up

Body

Finish

Understanding Swap

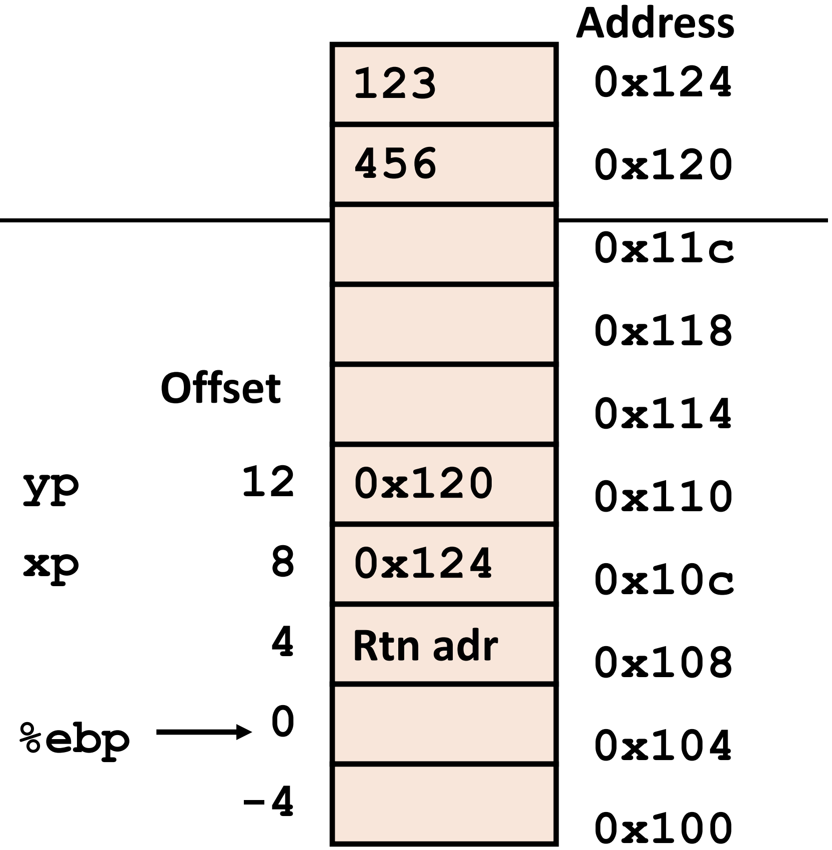
```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



Register	Value
%edx	xp
%ecx	yp
%ebx	t0
%eax	t1

```
movl   8(%ebp), %edx     # edx = xp
movl   12(%ebp), %ecx    # ecx = yp
movl   (%edx), %ebx      # ebx = *xp (t0)
movl   (%ecx), %eax      # eax = *yp (t1)
movl   %eax, (%edx)      # *xp = t1
movl   %ebx, (%ecx)      # *yp = t0
```

Understanding Swap




```
movl   8(%ebp), %edx     # edx = xp
movl   12(%ebp), %ecx    # ecx = yp
movl   (%edx), %ebx      # ebx = *xp (t0)
movl   (%ecx), %eax      # eax = *yp (t1)
movl   %eax, (%edx)      # *xp = t1
movl   %ebx, (%ecx)      # *yp = t0
```

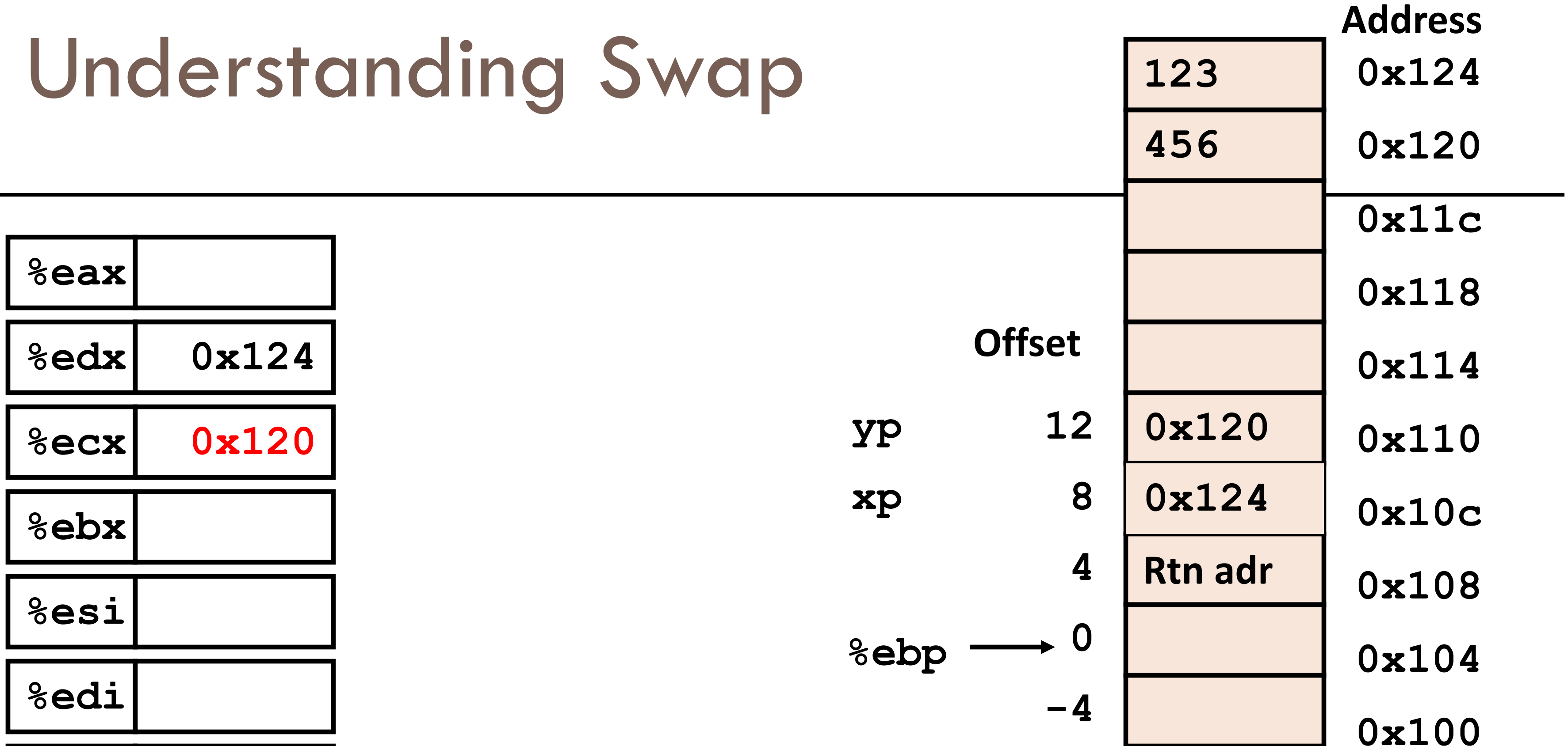
Understanding Swap

%eax	
%edx	0x124
%ecx	
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

	Offset		Address
		123	0x124
		456	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp →	0		0x104
	-4		0x100

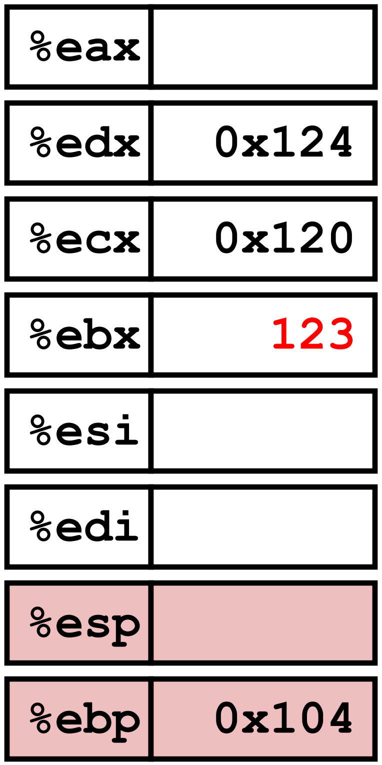
```
movl   8(%ebp), %edx     # edx = xp
movl   12(%ebp), %ecx    # ecx = yp
movl   (%edx), %ebx      # ebx = *xp (t0)
movl   (%ecx), %eax      # eax = *yp (t1)
movl   %eax, (%edx)      # *xp = t1
movl   %ebx, (%ecx)      # *yp = t0
```

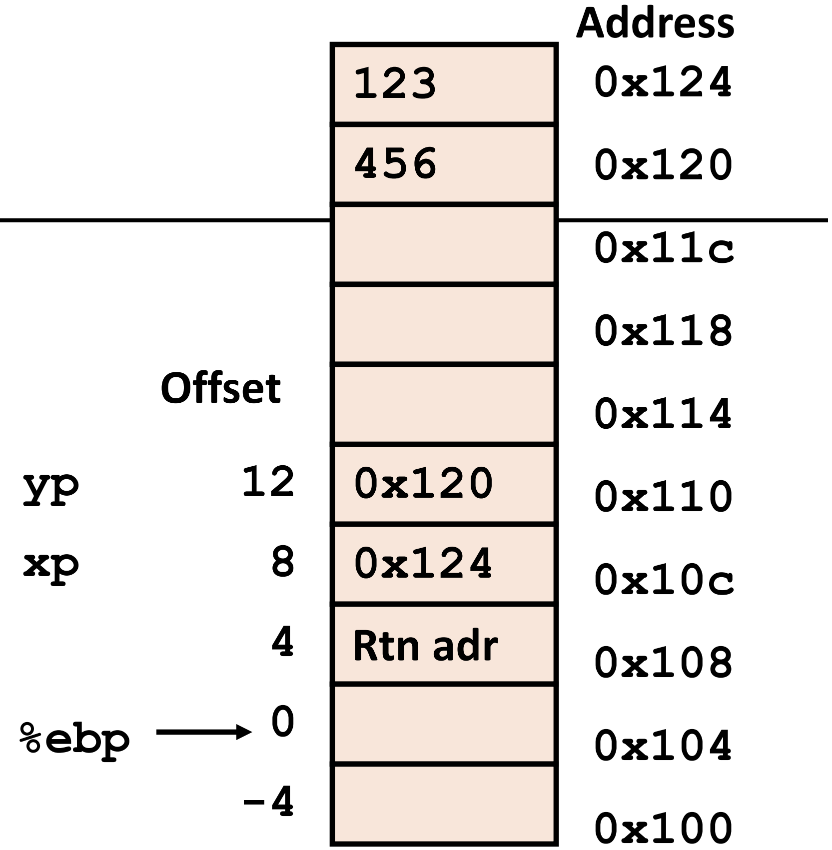
Understanding Swap




```
movl  8(%ebp), %edx    # edx = xp
movl  12(%ebp), %ecx   # ecx = yp
movl  (%edx), %ebx     # ebx = *xp (t0)
movl  (%ecx), %eax     # eax = *yp (t1)
movl  %eax, (%edx)     # *xp = t1
movl  %ebx, (%ecx)     # *yp = t0
```

Understanding Swap






```
movl   8(%ebp), %edx     # edx = xp
movl   12(%ebp), %ecx    # ecx = yp
movl   (%edx), %ebx      # ebx = *xp (t0)
movl   (%ecx), %eax      # eax = *yp (t1)
movl   %eax, (%edx)      # *xp = t1
movl   %ebx, (%ecx)      # *yp = t0
```

Understanding Swap

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

	Offset		Address
		123	0x124
		456	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp →	0		0x104
	-4		0x100

```
movl   8(%ebp), %edx     # edx = xp
movl   12(%ebp), %ecx    # ecx = yp
movl   (%edx), %ebx      # ebx = *xp (t0)
movl   (%ecx), %eax      # eax = *yp (t1)
movl   %eax, (%edx)      # *xp = t1
movl   %ebx, (%ecx)      # *yp = t0
```

Understanding Swap

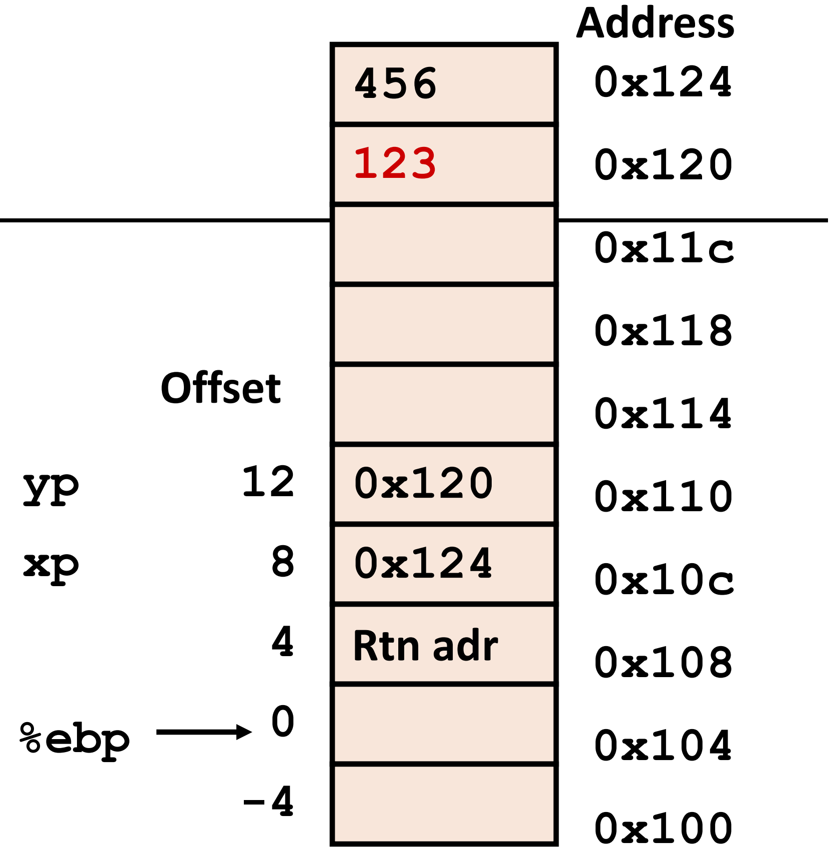
Register	Value
%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

	Offset		Address
		456	0x124
		456	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp →	0		0x104
	-4		0x100

```
movl   8(%ebp), %edx     # edx = xp
movl   12(%ebp), %ecx    # ecx = yp
movl   (%edx), %ebx      # ebx = *xp (t0)
movl   (%ecx), %eax      # eax = *yp (t1)
movl   %eax, (%edx)      # *xp = t1
movl   %ebx, (%ecx)      # *yp = t0
```

Understanding Swap

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104




```
movl   8(%ebp), %edx     # edx = xp
movl   12(%ebp), %ecx    # ecx = yp
movl   (%edx), %ebx      # ebx = *xp (t0)
movl   (%ecx), %eax      # eax = *yp (t1)
movl   %eax, (%edx)      # *xp = t1
movl   %ebx, (%ecx)      # *yp = t0
```

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 8 integer registers
  - Ri: Index register: Any, except for `%esp`
    - Unlikely you'd use `%ebp`, either
  - 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]]

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

- Extend existing registers. Add 8 new ones.
- Make **%ebp/%rbp** general purpose