CSSE132
Introduction to Computer Systems

13 : Machine level programming

March 25, 2013

Today: Machine Level Programming

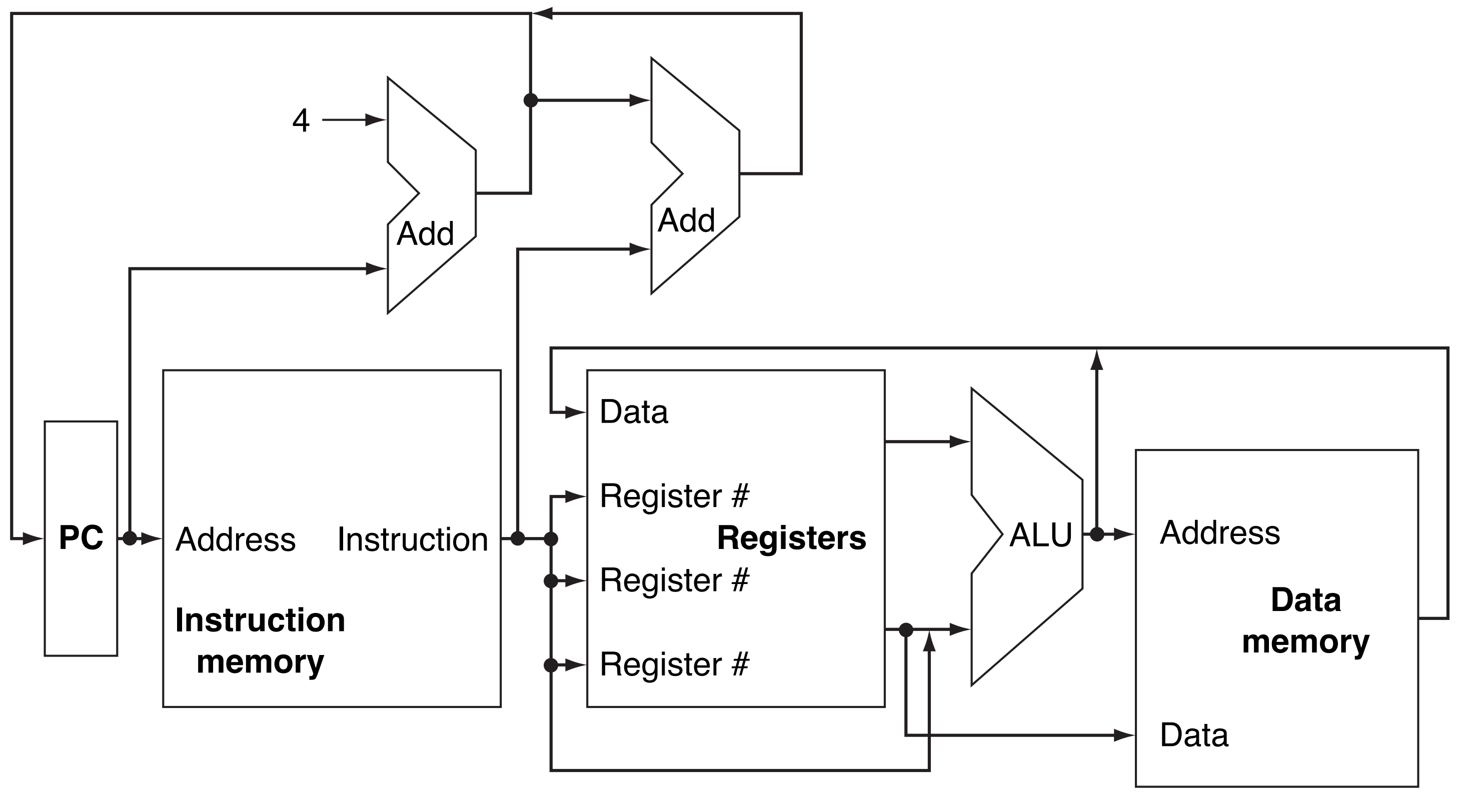
- **Review**
- **History of Intel processors**
- **Assembly programming**
  - GCC demo
- **Intel architecture**
  - Data sizes
  - Registers
  - Operands
- **Data movement instructions**

Review

- **First week**
  - Bit, bytes, and hexadecimal
  - Two’s complement and signed numbers
  - Boolean logic and bitwise operations
  - Integer arithmetic
- **Second week**
  - Floating point representation
  - Boolean algebra
- **Third week**
  - K-maps
  - 1 bit and larger adders
  - Flipflops & registers
  - ALU design

Review

- **Computational model**
  - CPU components : registers, memory, ALU



Today: Machine Level Programming

- **Review**
- **History of Intel processors**
- **Assembly programming**
  - GCC demo
- **Intel architecture**
  - Data sizes
  - Registers
  - Operands
- **Data movement instructions**

Intel x86 Processors

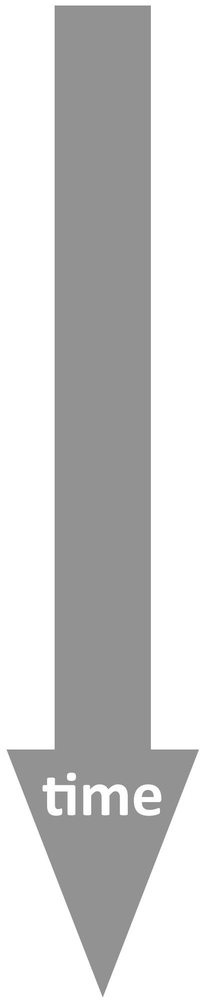
- **Totally dominate laptop/desktop/server market**

- **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 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



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

Today: Machine Level Programming

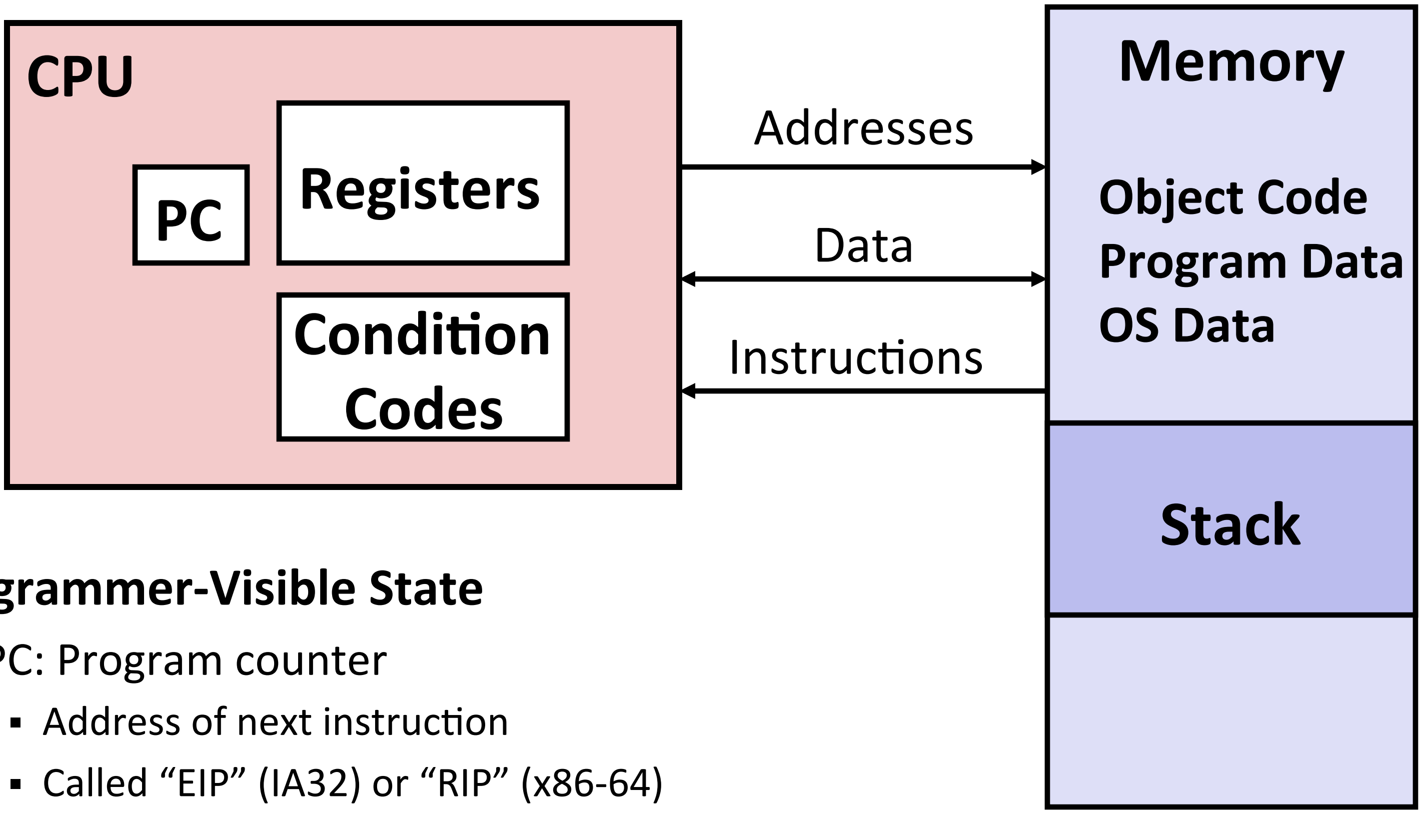
- **Review**
- **History of Intel processors**
- **Assembly programming**
  - GCC demo
- **Intel architecture**
  - Data sizes
  - Registers
  - Operands
- **Data movement instructions**

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

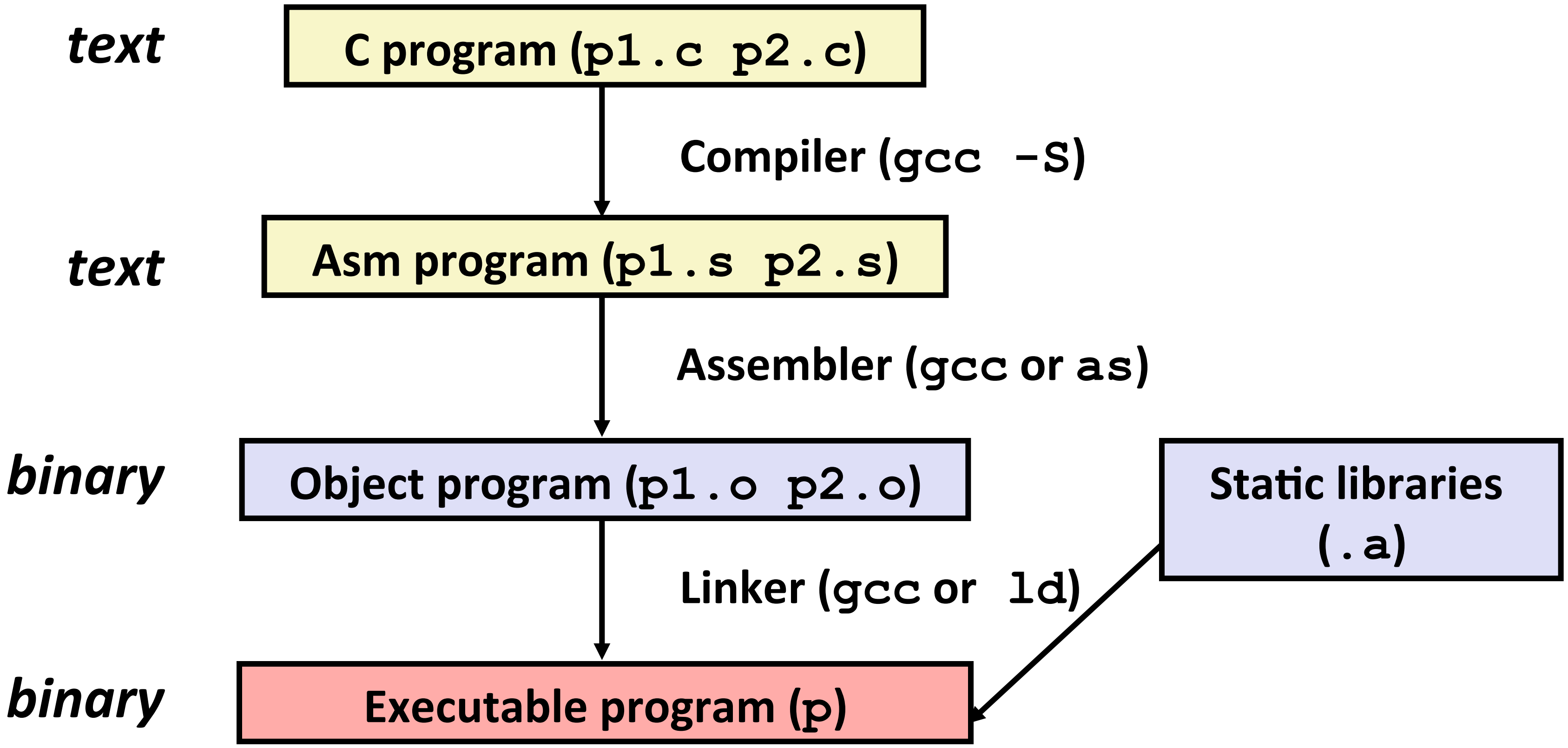


- **Programmer-Visible State**
  - PC: Program counter
    - Address of next instruction
    - Called "EIP" (IA32) or "RIP" (x86-64)
  - Register file
    - Heavily used program data
  - Condition codes
    - Store status information about most recent arithmetic operation
    - Used for conditional branching

- **Memory**
  - Byte addressable array
  - Code, user data, (some) OS data
  - Includes stack used to support procedures

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 -m32 -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)
  - Word size is 2 bytes (16 bits)

- **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

Machine Instruction Example

```
int t = x+y;
```

```
addl 8(%ebp),%eax
```

Similar to expression:

```
x += y
```

More precisely:

```
int eax;
int *ebp;
eax += ebp[2]
```

```
0x80483ca:   03 45 08
```

- **C Code**
  - Add two signed integers
- **Assembly**
  - Add 2 4-byte integers
    - "Long" words in GCC parlance
    - Same instruction whether signed or unsigned
  - Operands:

`x:`	Register	**`%eax`**
`y:`	Memory	**`M[%ebp+8]`**
`t:`	Register	**`%eax`**

    – Return function value in **`%eax`**

- **Object Code**
  - 3-byte instruction
  - Stored at address **`0x80483ca`**

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

GCC demo

- **Generate assembly code**
  - gcc -m32 -O1 -S code.c　　　#output assembler in code.s
- **Generate object code**
  - gcc -m32 -O1 -c code.c　　　#output object code in code.o
- **Using a debugger**
  - gdb code.o
    - disassemble sum　　　#disassemble code in memory
    - x/17xb sum　　　#inspect machine code in memory
    - quit　　　#quit debugger
- **Converting to object code**
  - objdump -d code.o　　　#disassemble code

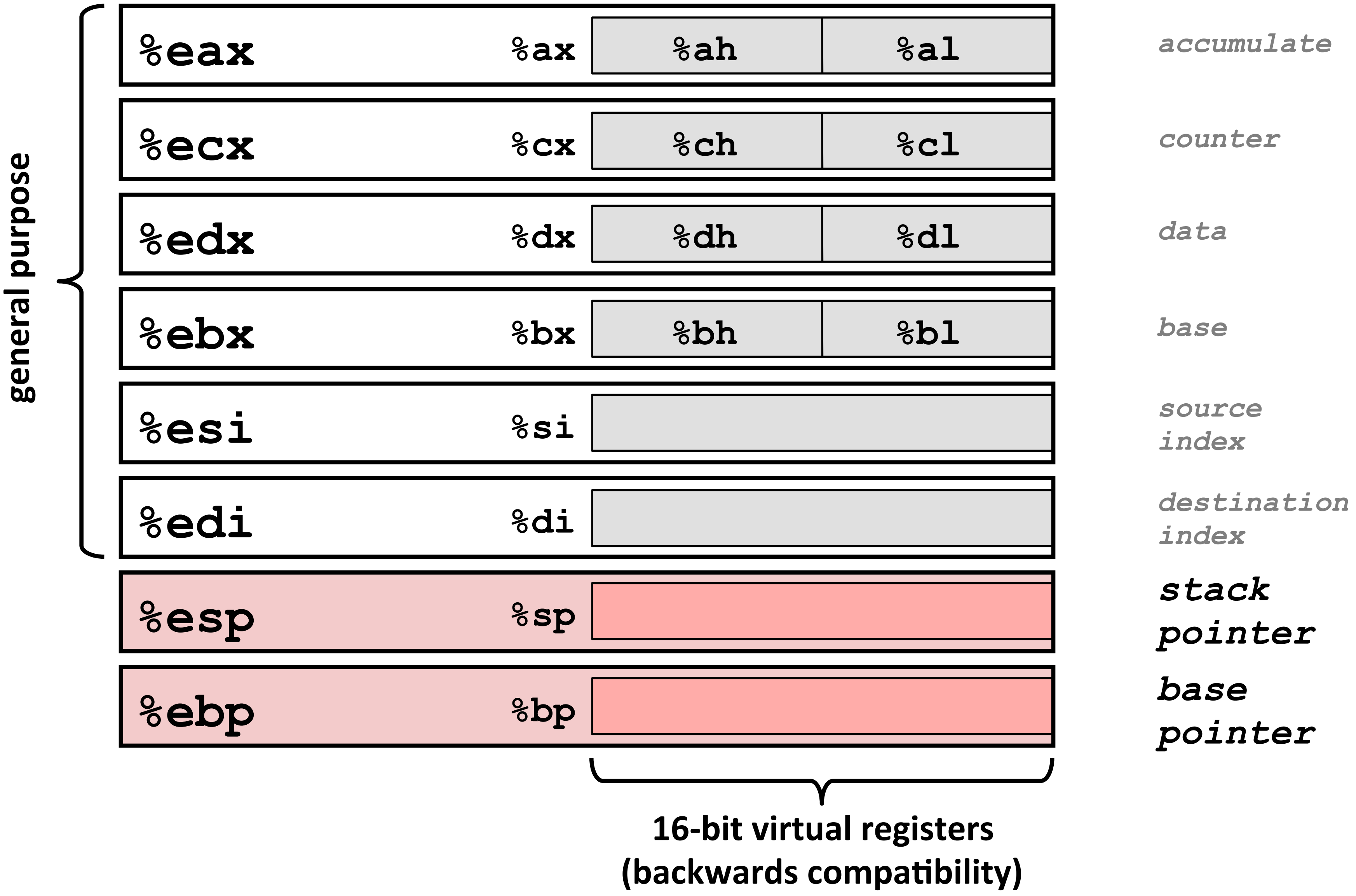
Today: Machine Level Programming

- **Review**
- **History of Intel processors**
- **Assembly programming**
  - GCC demo
- **Intel architecture**
  - Data sizes
  - Registers
  - Operands
- **Data movement instructions**

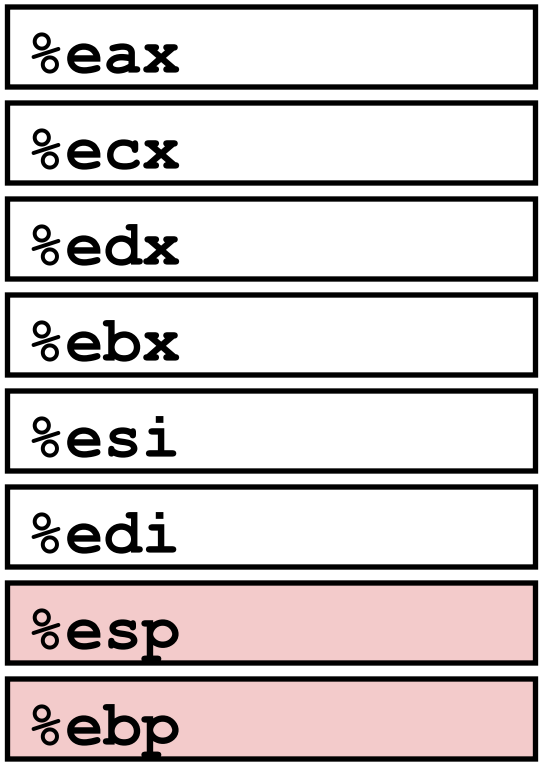
IA32 data sizes

C declaration	Intel data type	Asm code suffix	Size
`char`	Byte	`b`	1
`short`	Word	`w`	2
`int`	Double word	`l`	4
`long int`	Double word	`l`	4
`long long int`	—	—	4
`char *`	Double word	`l`	4
`float`	Single precision	`s`	4
`double`	Double precision	`l`	8
`long double`	Extended precision	`t`	10/12

Integer Registers (IA32)



Moving Data: IA32



- **Moving Data**

  `movl` ***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 8 integer registers
    - Example: **`%eax, %edx`**
    - But **`%esp`** and **`%ebp`** reserved for special use
    - Others have special uses for particular instructions
  - ***Memory:*** 4 consecutive bytes of memory at address given by register
    - Simplest example: **`(%eax)`**
    - Various other "address modes"

`movl` Operand Combinations

	Source	Dest	Src,Dest	C Analog
`movl`	*Imm*	*Reg*	`movl $0x4,%eax`	`temp = 0x4;`
		Mem	`movl $-147,(%eax)`	`*p = -147;`
	Reg	*Reg*	`movl %eax,%edx`	`temp2 = temp1;`
		Mem	`movl %eax,(%edx)`	`*p = temp;`
	Mem	*Reg*	`movl (%eax),%edx`	`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

```
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

Body

Finish

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
```

Set Up

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

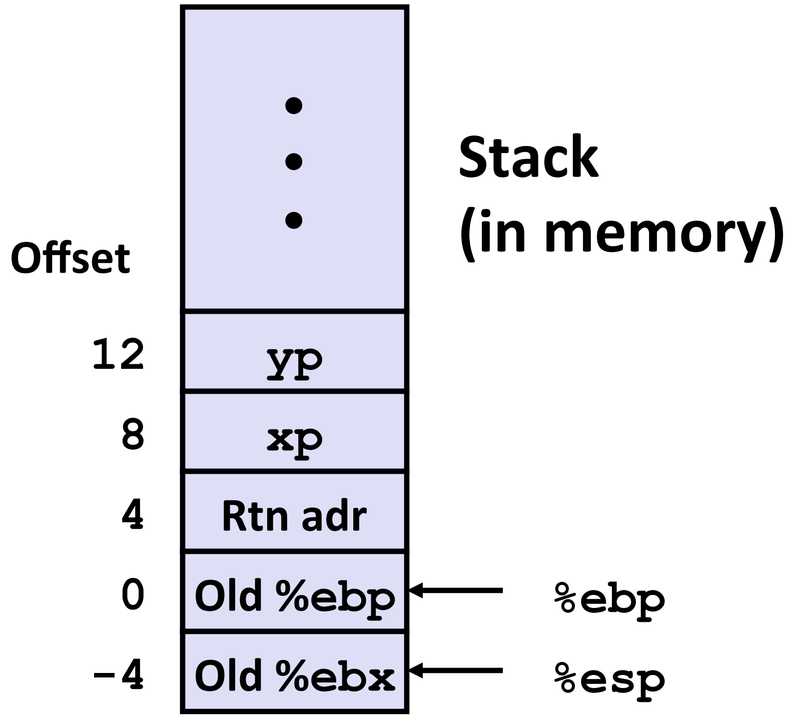
Body

```
  popl   %ebx
  popl   %ebp
  ret
```

Finish

Understanding Swap

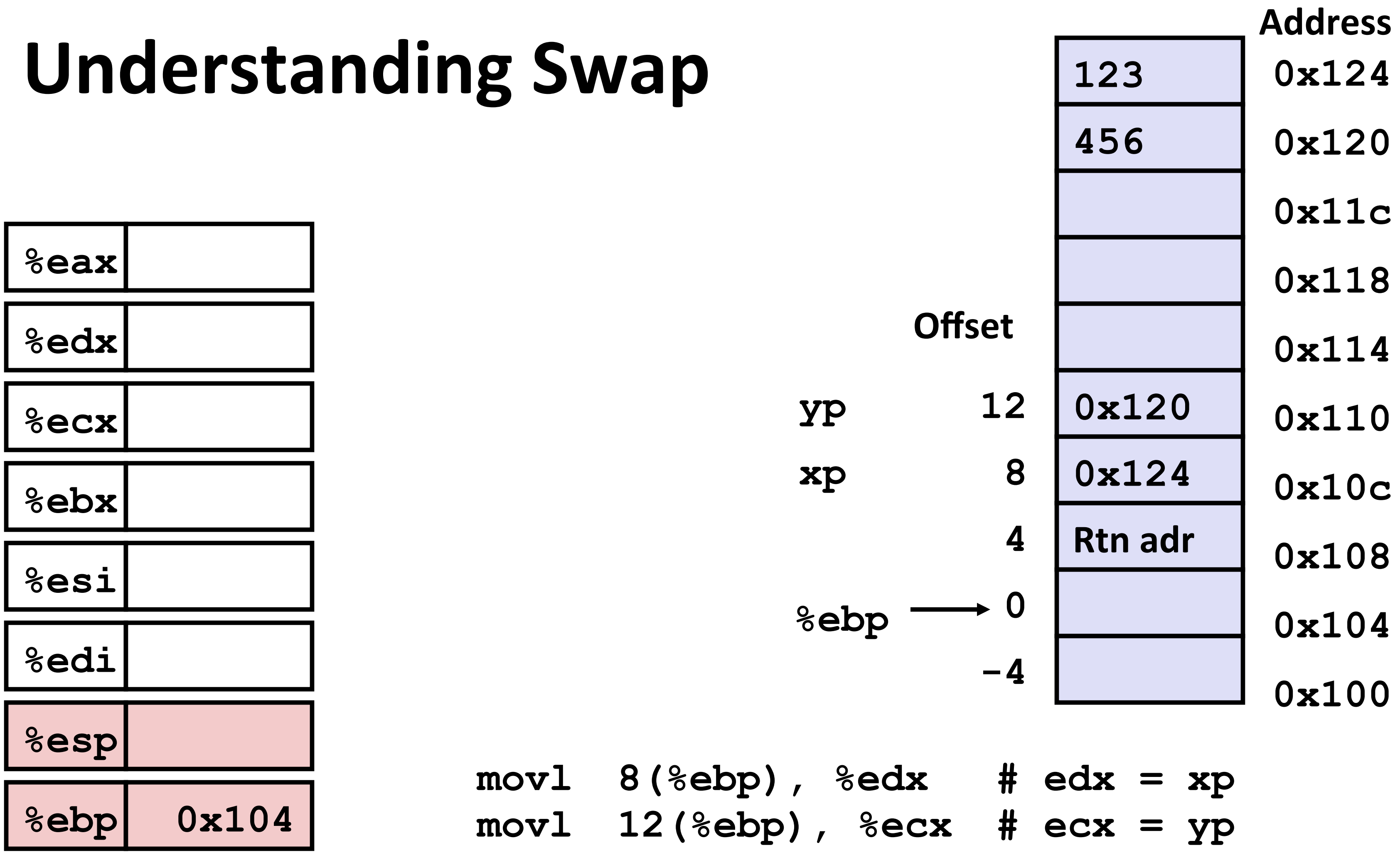
```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



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

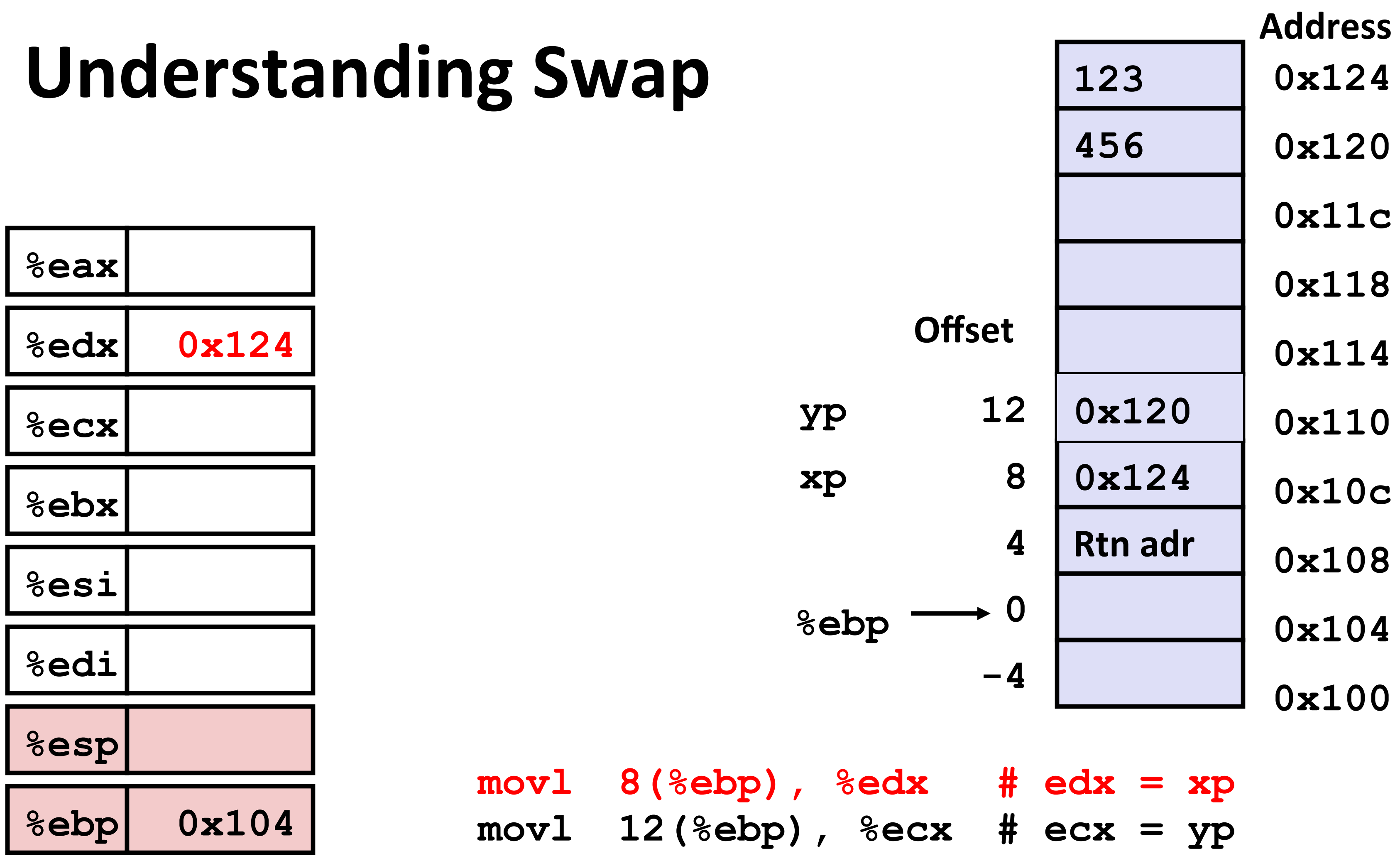
```
movl  8(%ebp), %edx    # edx = xp
movl  12(%ebp), %ecx   # ecx = yp
movl  (%edx), %ebx     # ebx = *xp (t0)
movl  (%ecx), %eax     # eax = *yp (t1)
movl  %eax, (%edx)     # *xp = t1
movl  %ebx, (%ecx)     # *yp = t0
```

Understanding Swap



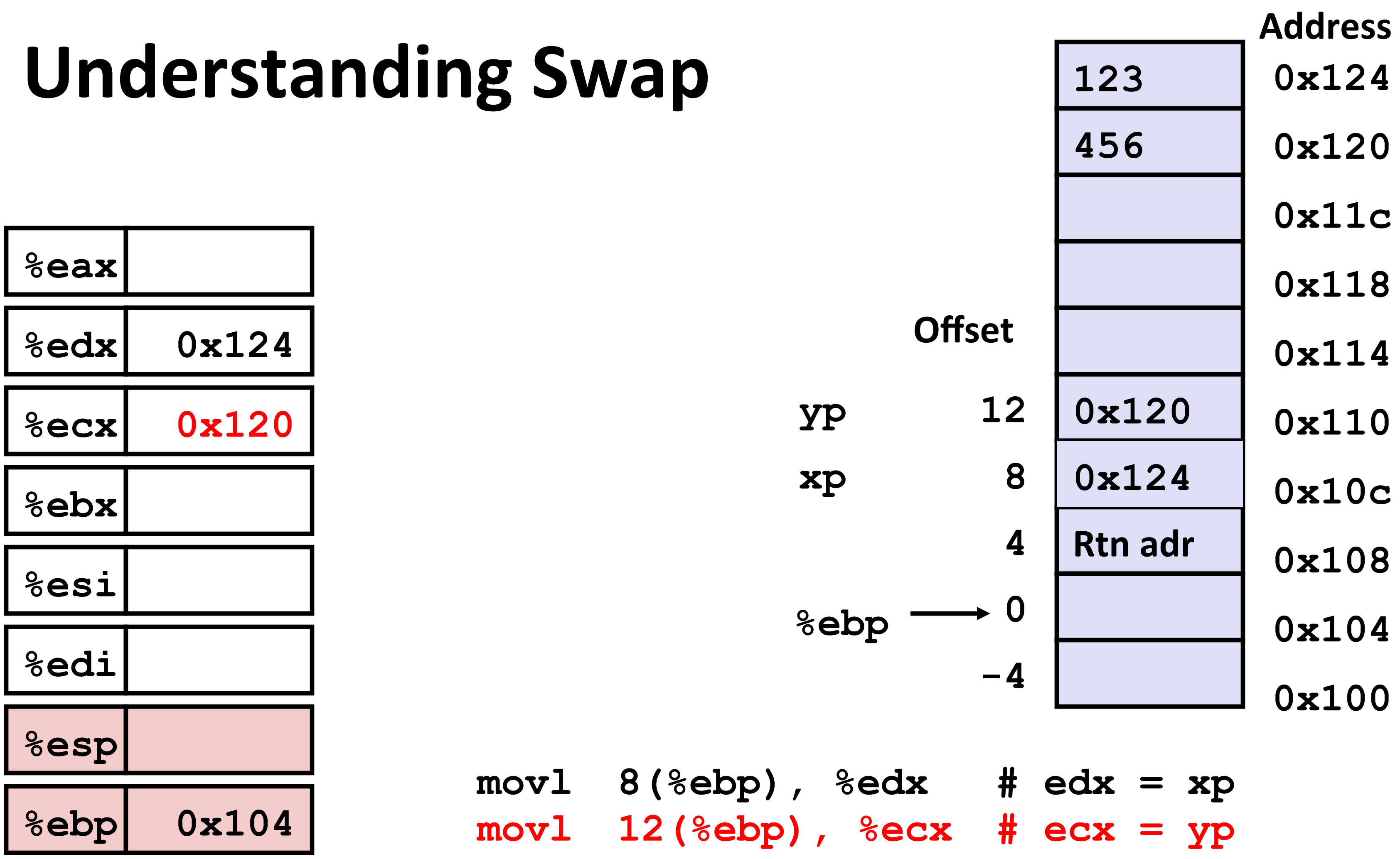
```
movl  8(%ebp), %edx    # edx = xp
movl  12(%ebp), %ecx   # ecx = yp
movl  (%edx), %ebx     # ebx = *xp (t0)
movl  (%ecx), %eax     # eax = *yp (t1)
movl  %eax, (%edx)     # *xp = t1
movl  %ebx, (%ecx)     # *yp = t0
```

Understanding Swap



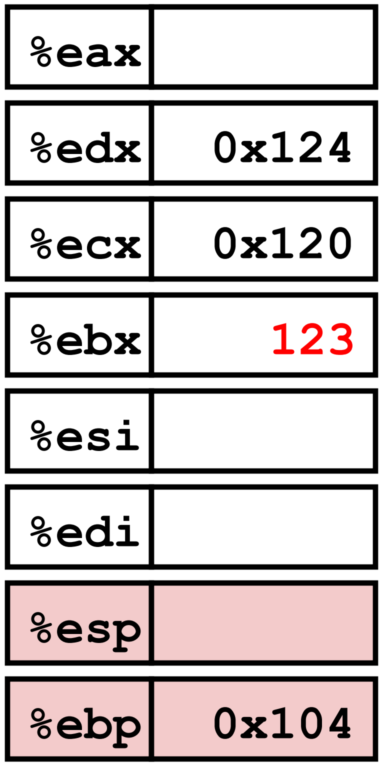
```
movl   8(%ebp), %edx     # edx = xp
movl   12(%ebp), %ecx    # ecx = yp
movl   (%edx), %ebx      # ebx = *xp (t0)
movl   (%ecx), %eax      # eax = *yp (t1)
movl   %eax, (%edx)      # *xp = t1
movl   %ebx, (%ecx)      # *yp = t0
```

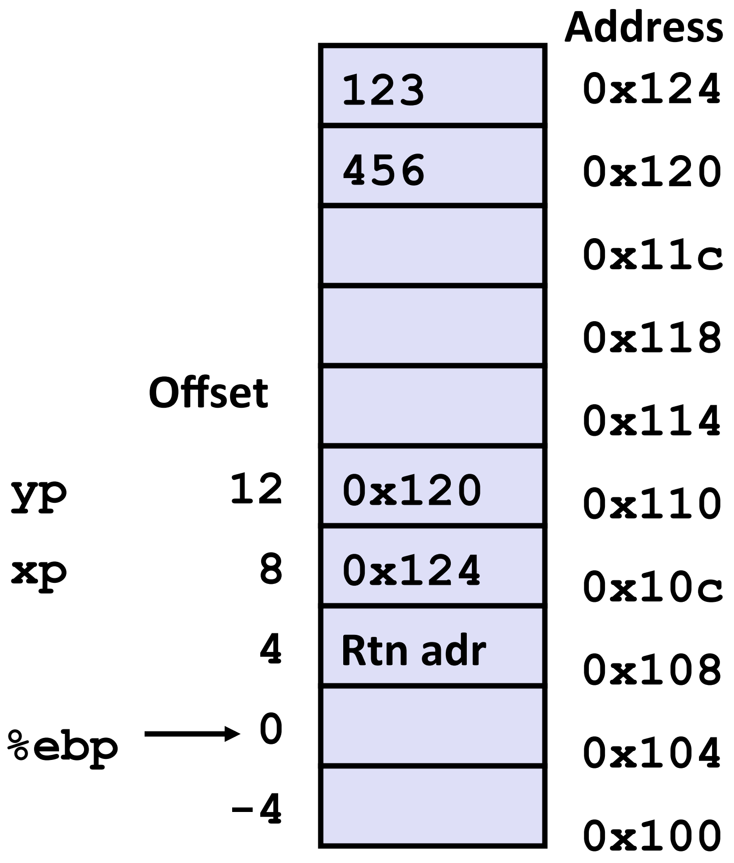
Understanding Swap



```
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
```

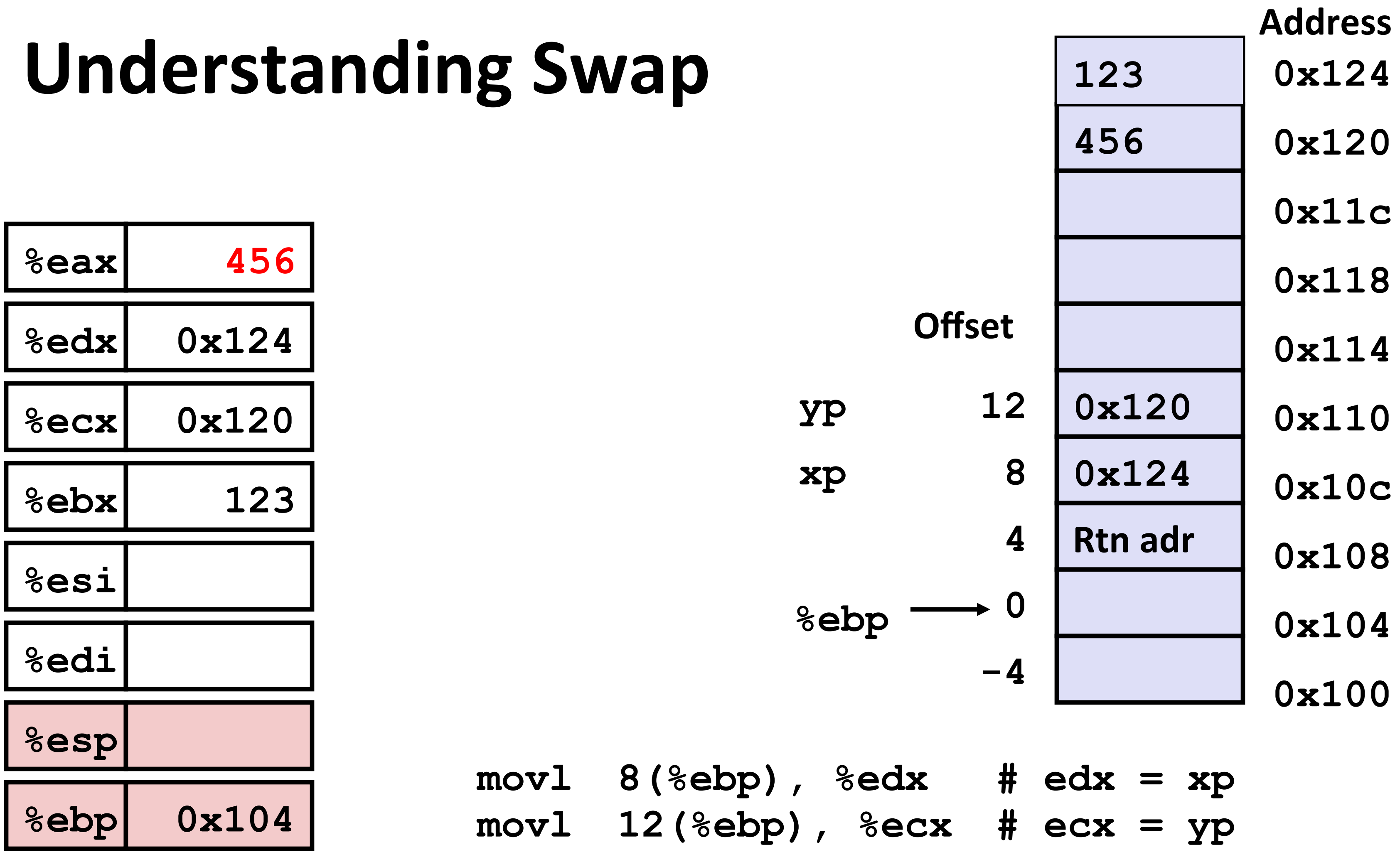
Understanding Swap





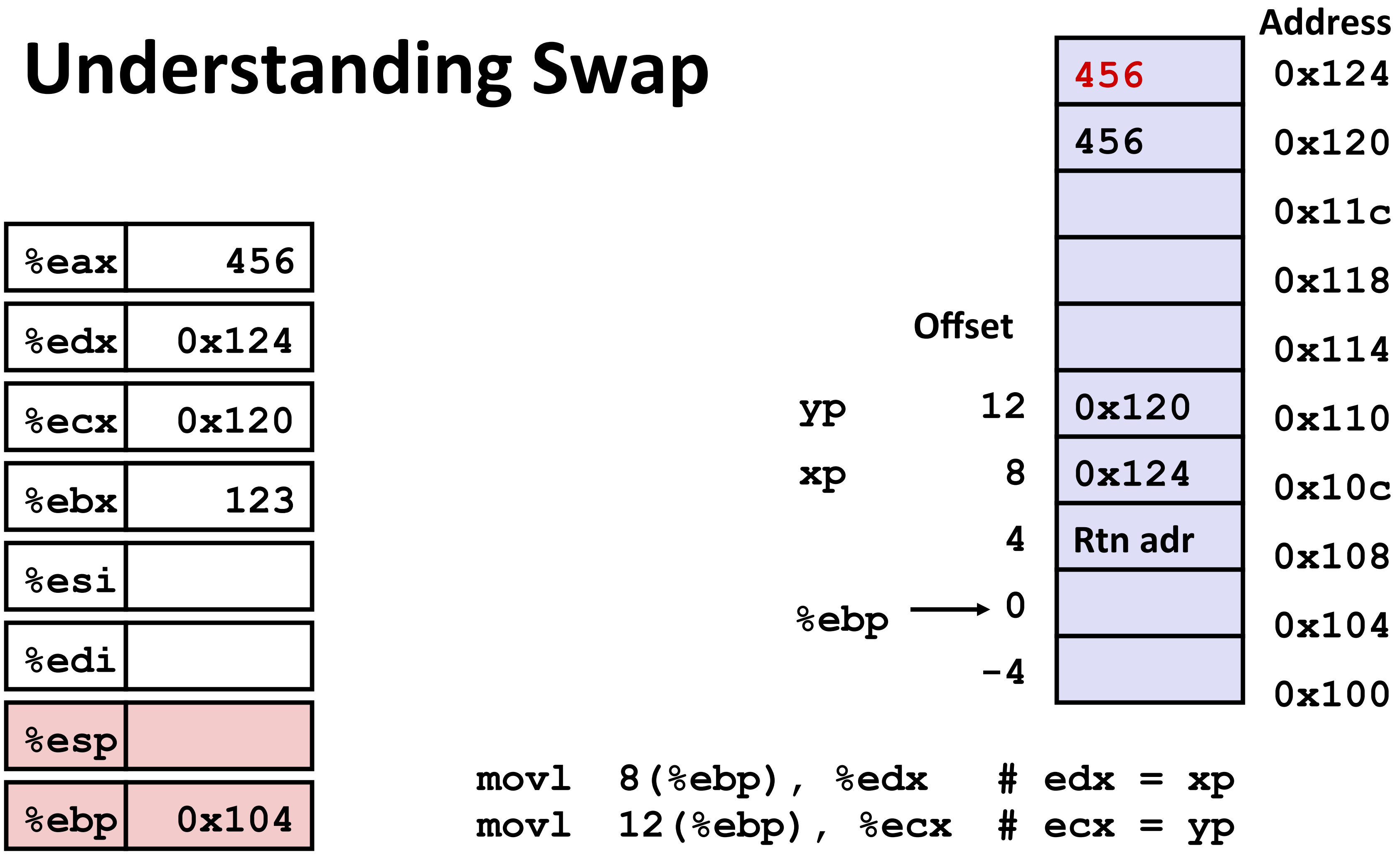
```
movl   8(%ebp), %edx    # edx = xp
movl   12(%ebp), %ecx   # ecx = yp
movl   (%edx), %ebx     # ebx = *xp (t0)
movl   (%ecx), %eax     # eax = *yp (t1)
movl   %eax, (%edx)     # *xp = t1
movl   %ebx, (%ecx)     # *yp = t0
```

Understanding Swap



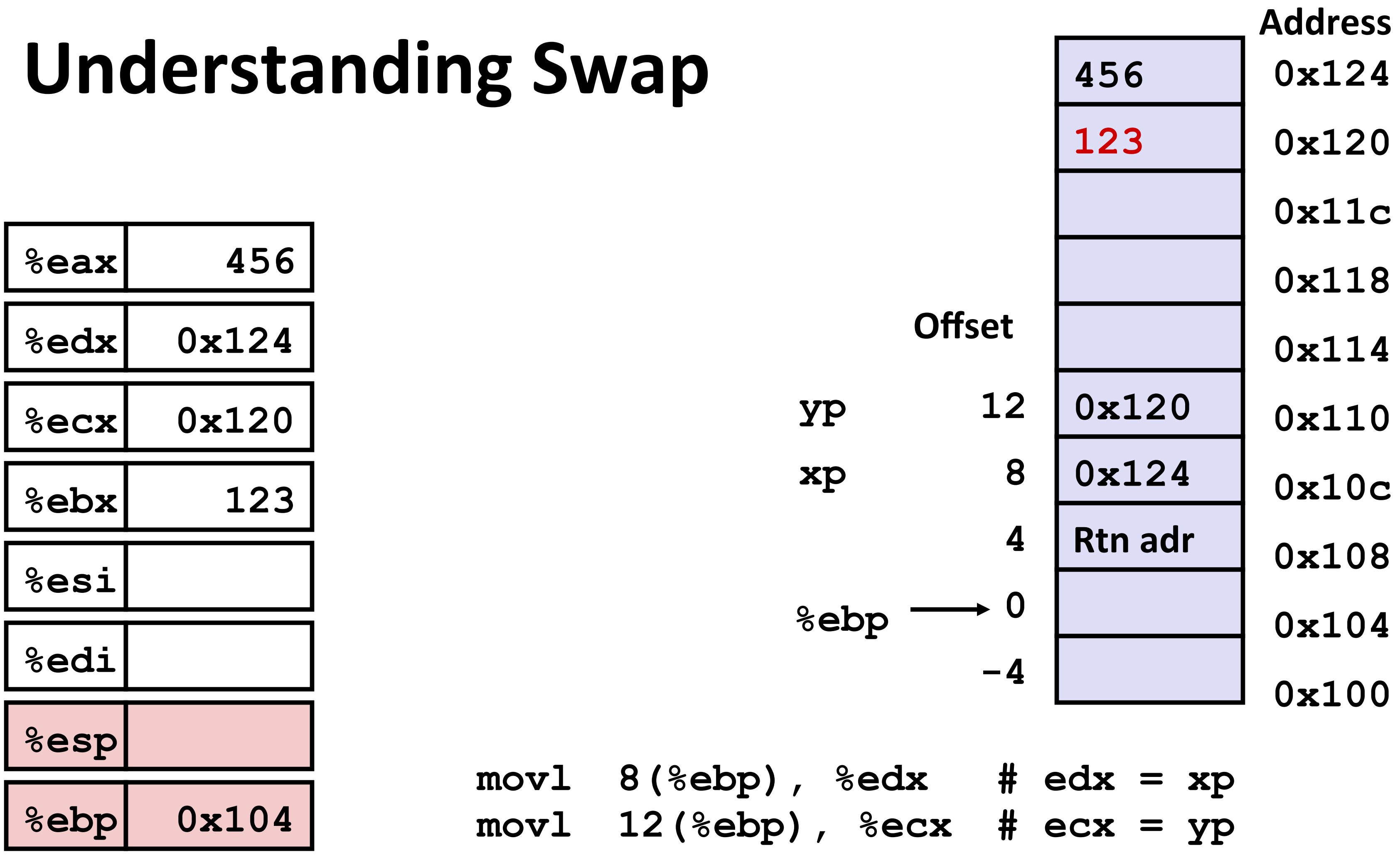
```
movl   8(%ebp), %edx     # edx = xp
movl   12(%ebp), %ecx    # ecx = yp
movl   (%edx), %ebx      # ebx = *xp (t0)
movl   (%ecx), %eax      # eax = *yp (t1)
movl   %eax, (%edx)      # *xp = t1
movl   %ebx, (%ecx)      # *yp = t0
```

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
```

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]]**

Addressing modes

Type	Form	Operand value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	E_a	$\mathbf{R}[\mathrm{E}_a]$	Register
Memory	`Imm`	$\mathbf{M}[Imm]$	Absolute
Memory	(E_a)	$\mathbf{M}[\mathbf{R}[\mathrm{E}_a]]$	Indirect
Memory	$Imm(\mathrm{E}_b)$	$\mathbf{M}[Imm + \mathbf{R}[\mathrm{E}_b]]$	Base + displacement
Memory	$(\mathrm{E}_b, \mathrm{E}_i)$	$\mathbf{M}[\mathbf{R}[\mathrm{E}_b] + \mathbf{R}[\mathrm{E}_i]]$	Indexed
Memory	$Imm(\mathrm{E}_b, \mathrm{E}_i)$	$\mathbf{M}[Imm + \mathbf{R}[\mathrm{E}_b] + \mathbf{R}[\mathrm{E}_i]]$	Indexed
Memory	$(, \mathrm{E}_i, s)$	$\mathbf{M}[\mathbf{R}[\mathrm{E}_i] \cdot s]$	Scaled indexed
Memory	$Imm(, \mathrm{E}_i, s)$	$\mathbf{M}[Imm + +\mathbf{R}[\mathrm{E}_i] \cdot s]$	Scaled indexed
Memory	$(\mathrm{E}_b, \mathrm{E}_i, s)$	$\mathbf{M}[\mathbf{R}[\mathrm{E}_b] + \mathbf{R}[\mathrm{E}_i] \cdot s]$	Scaled indexed
Memory	$Imm(\mathrm{E}_b, \mathrm{E}_i, s)$	$\mathbf{M}[Imm + \mathbf{R}[\mathrm{E}_b] + \mathbf{R}[\mathrm{E}_i] \cdot s]$	Scaled indexed

Data movement

Instruction		Effect	Description
`mov`	S, D	$D \leftarrow S$	Move
`movb`		Move byte	
`movw`		Move word	
`movl`		Move double word	
`movs`	S, D	$D \leftarrow \texttt{SignExtend}(S)$	Move with sign extension
`movsbw`		Move sign-extended byte to word	
`movsbl`		Move sign-extended byte to double word	
`movswl`		Move sign-extended word to double word	
`movz`	S, D	$D \leftarrow \texttt{ZeroExtend}(S)$	Move with zero extension
`movzbw`		Move zero-extended byte to word	
`movzbl`		Move zero-extended byte to double word	
`movzwl`		Move zero-extended word to double word	
`pushl`	S	$\mathbf{R}[\%\texttt{esp}] \leftarrow \mathbf{R}[\%\texttt{esp}] - 4;$ $\mathbf{M}[\mathbf{R}[\%\texttt{esp}]] \leftarrow S$	Push double word
`popl`	D	$D \leftarrow \mathbf{M}[\mathbf{R}[\%\texttt{esp}]];$ $\mathbf{R}[\%\texttt{esp}] \leftarrow \mathbf{R}[\%\texttt{esp}] + 4$	Pop double word