CSE 523S: Systems Security

Computer & Network Systems Security

Fall 2024

Prof. Patrick Crowley

Week 3 Q&A

Nobody has responded yet.

Hang tight! Responses are coming in.

Remember the Big Picture

- The last high-level discussion before we dive into the details later in this lecture

- It is very easy to get lost in the details!!
  - Remember the high-level motivation

- Our goal is to develop security awareness by looking at known vulnerabilities, exploits and mitigation
  - What does it mean?

Let’s Start with Definitions

- **Vulnerabilities**: A weaknesses or flaws in the system or network
  - Intended or not…

- **Exploits**: programs or code designed to leverage flaws or weaknesses, and cause unintended effects.

“If a vulnerability is an open window into the system, an exploit is the rope or ladder the thief uses to reach the open window.”

Hacker Quadrant (In 523S we are ethical & legal)

	Ethical	Unethical
Legal		*Grey area, usually bad (e.g., stalking)*
Illegal	*Grey area, usually good (e.g., dissidents)*	*Criminals*

Vulnerabilities & Exploits

- Some vulnerabilities & exploits include
  - Stack overflow
  - Format String
  - Race Condition
  - Dirty Cow
  - Shellshock
  - SQL injection
  - Integer Overflow
  - Heap Overflow
  - Use After Free
  - ...
- We will cover some and also different types of exploits and fuzzing
  - Tools, scripts, self-constructed payloads

WHY ARE OUR COMPUTER SYSTEMS VULNERABLE?

Computers are Vulnerable

- Because we **write** **our own** **software**
  - Did we mistakenly/intentionally add vulnerabilities?
- Because we **choose** **our own** **software**
  - Can we know if it has vulnerabilities?
- Because **software** **requires input**
  - Can inputs be used to trigger a vulnerability?

Which one is Vulnerable?






- Write SW?
- Choose SW?
- Provide input?

How can I execute <u>my</u> code on <u>your</u> system?

- I can give you the program, and have you **<u>execute</u>** it for me
  - E.g.: *Email: Please download and run this attachment*
  - *E.g.: Web: download dodgy sw from the internet*

- I can gain access to your machine and **<u>execute</u>** it myself
  - E.g.: Exploit a system vulnerability to gain access
  - E.g.: Steal credentials to gain access

Execute?

- CSE 361 Reminders

Let's review how code gets executed

- Adopt this mindset
  - We write our code into memory, and give a starting address to the CPU
  - The CPU executes a simple machine language
  - **Assembly code is nothing to fear**

- We will be looking at binaries throughout the semester, so let's start from the beginning
- Note: x86 comes in two flavors, Intel assembly syntax and AT&T syntax (we use the latter)

Intel "x86" Processors

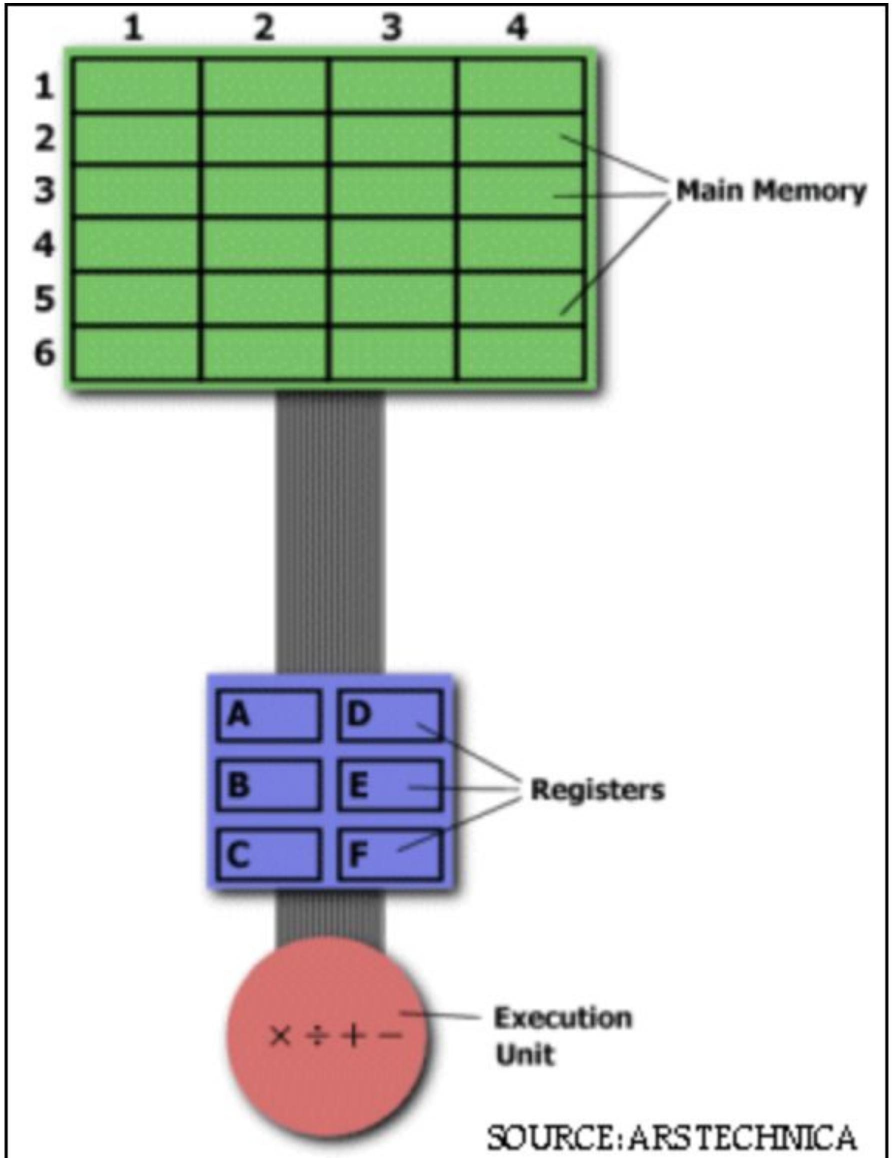
- Dominate computer market but share is shrinking

- Evolutionary Design
  – Starting in 1978 with 8086
  – Add more features as time goes on
  – Still support old features, although obsolete

Many of following slides taken from CSE 361, based on Computer Systems, by Bryant & O'Hallaron

- Complex Instruction Set Computer (CISC)
  – Many different instructions with many different formats
    • But, only small subset encountered with Linux programs
  – People thought that it would be hard to match performance of Reduced Instruction Set Computer (RISC)
    • But, Intel has done just that!

CISC vs. RISC



Multiplying Two Numbers in Memory

CISC APPROACH:

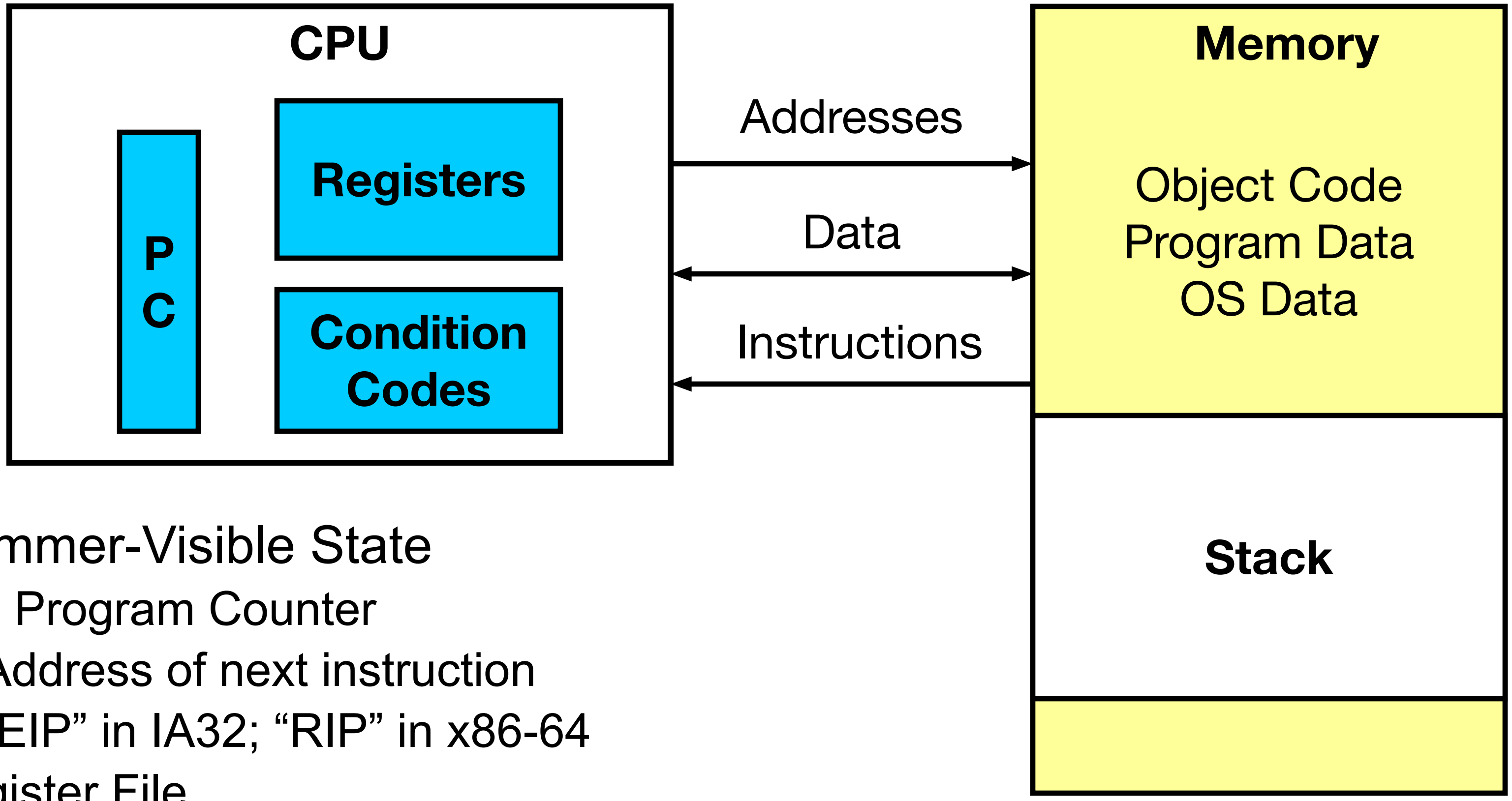
```
MULT 2:3, 5:2
```

RISC APPROACH:

```
LOAD A, 2:3
LOAD B, 5:2
PROD A, B
STORE 2:3, A
```

example taken from:
https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/

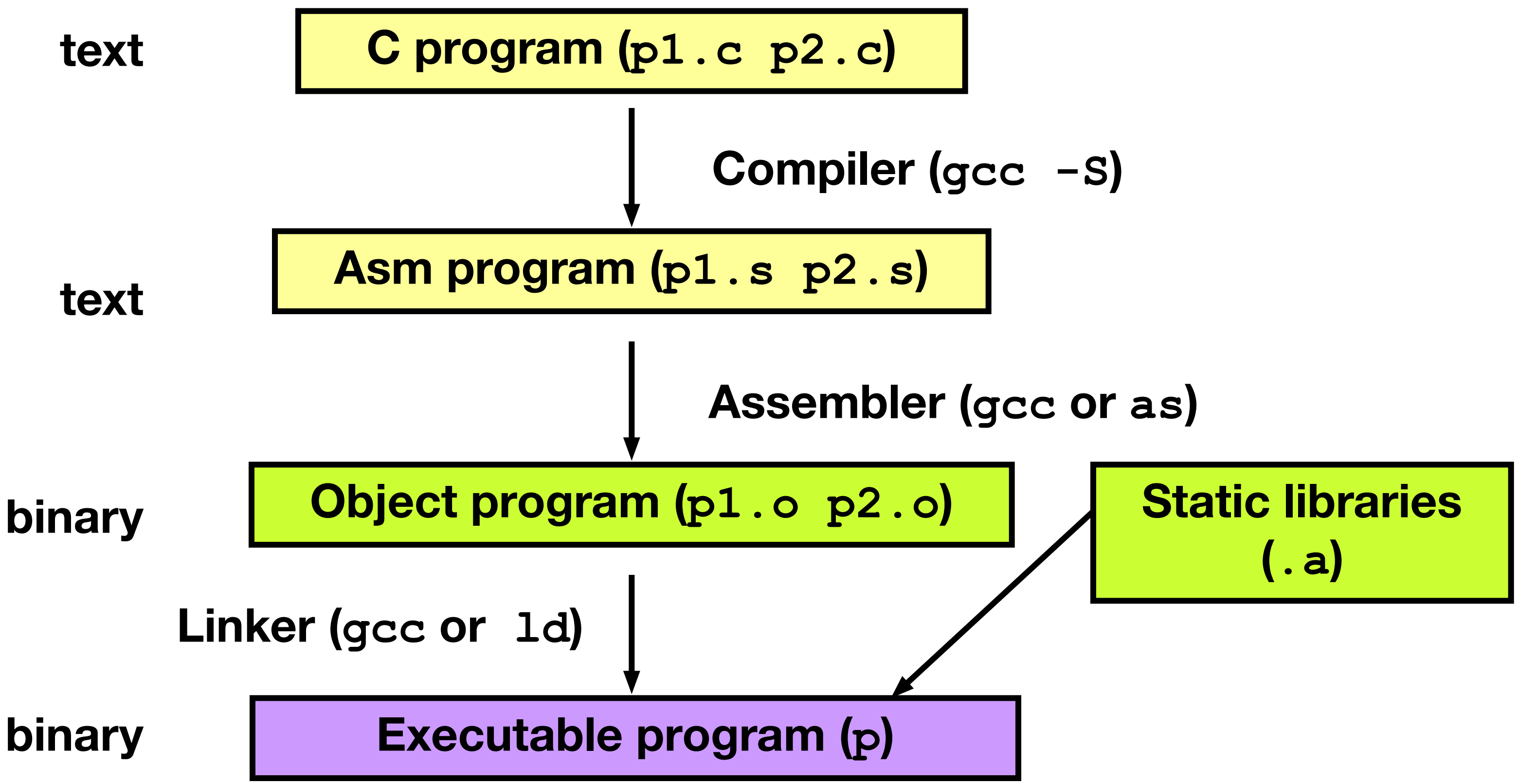
Assembly Programmer's View



- Programmer-Visible State
  - PC: Program Counter
    - Address of next instruction
    - "EIP" in IA32; "RIP" in 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 -O p1.c p2.c -o p`
  - Use optimizations (`-O`)
  - Put resulting binary in file `p`



Compiling Into Assembly

C Code

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

Generated Assembly

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

Obtain with command

```
gcc -O -S code.c
```

Produces file `code.s because of -S`

`Using -O will produce optimized results`

`Try and compare:`

```
gcc -S code.c
```

`Are we using 32-bit or 64-bit instructions?`

`Try -m32 and -m64 to see differences`

`One more thing: compilers change`

`Exact .s results might vary depending on version of gcc`

Object Code

Code for `sum`

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

- **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;
```

- C Code
  - Add two signed integers

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

Similar to expression:

```
x += y
```

Or

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

- 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`

```
0x401046: 03 45 08
```

- Object Code
  - 3-byte instruction
  - Stored at address `0x401046`

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 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 w/ gdb

Object

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

Disassembled

```
0x401040 <sum>:  push   %ebp
0x401041 <sum+1>:   mov    %esp,%ebp
0x401043 <sum+3>:   mov    0xc(%ebp),%eax
0x401046 <sum+6>:   add    0x8(%ebp),%eax
0x401049 <sum+9>:   mov    %ebp,%esp
0x40104b <sum+11>:  pop    %ebp
0x40104c <sum+12>:  ret
0x40104d <sum+13>:  lea    0x0(%esi),%esi
```

- Within gdb Debugger

  `gdb p`

  `disassemble sum`

  – Disassemble procedure

  `x/13b sum`

  – Examine the 13 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
```

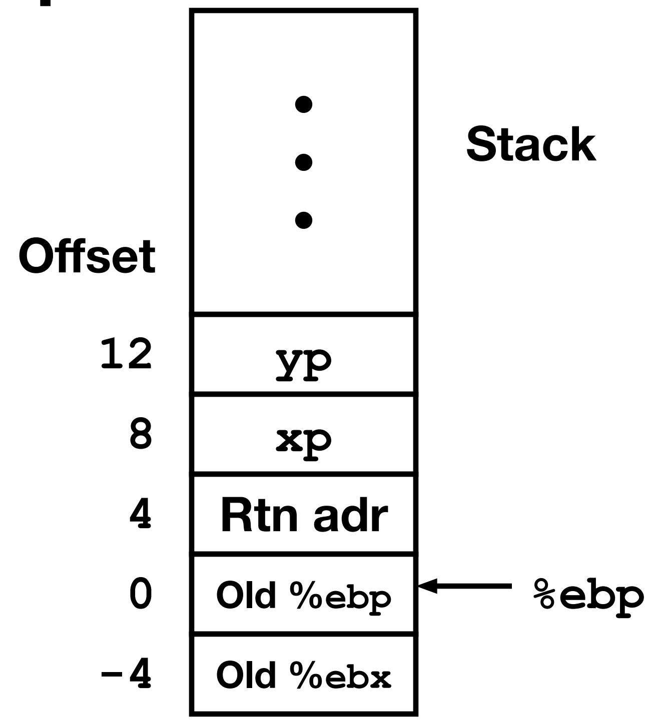
- Anything that can be interpreted as executable code
- Disassembler examines bytes and reconstructs assembly source

- BUT be careful, reverse engineering forbidden by Microsoft end user license agreement!

Integer Registers (IA32)

Understanding Swap

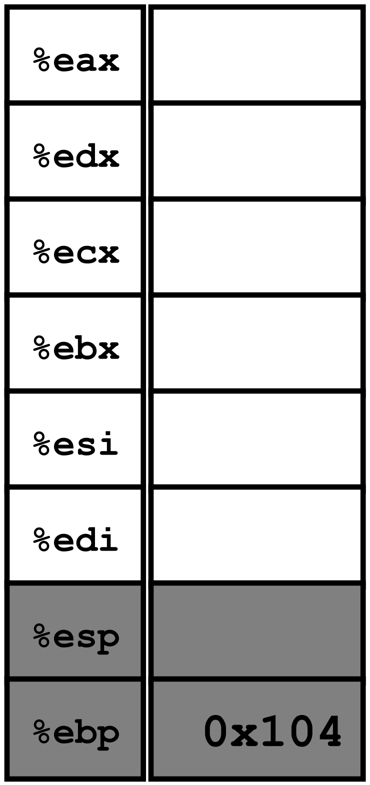
```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

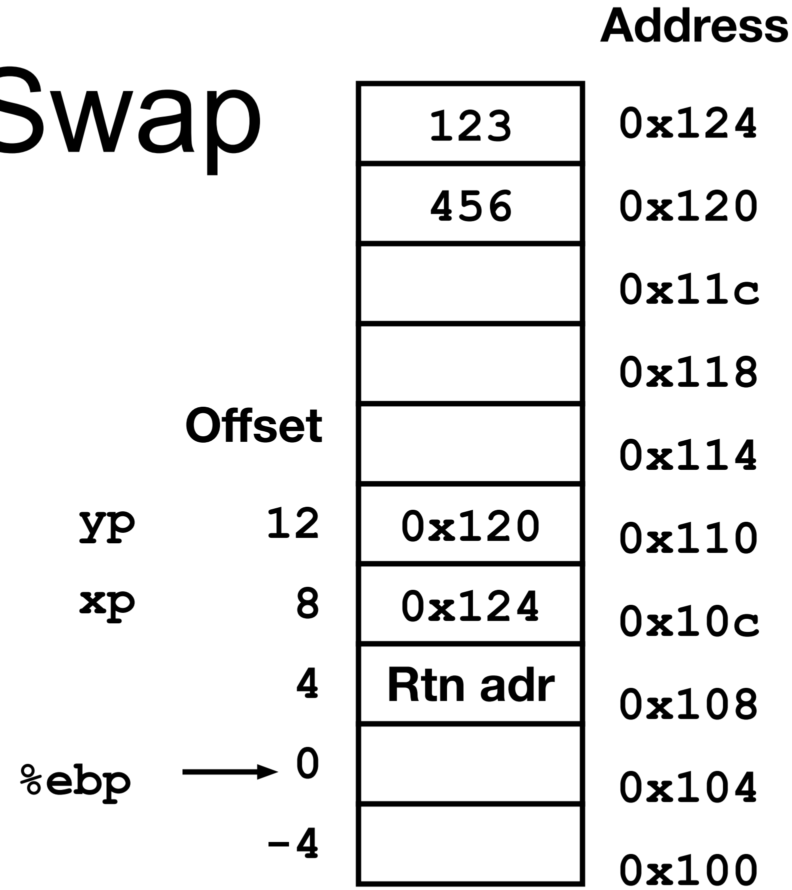


Register	Variable
%ecx	yp
%edx	xp
%eax	t1
%ebx	t0

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

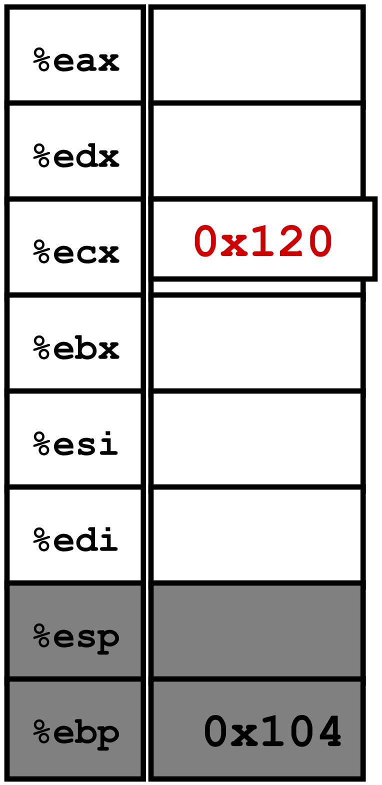
Understanding Swap

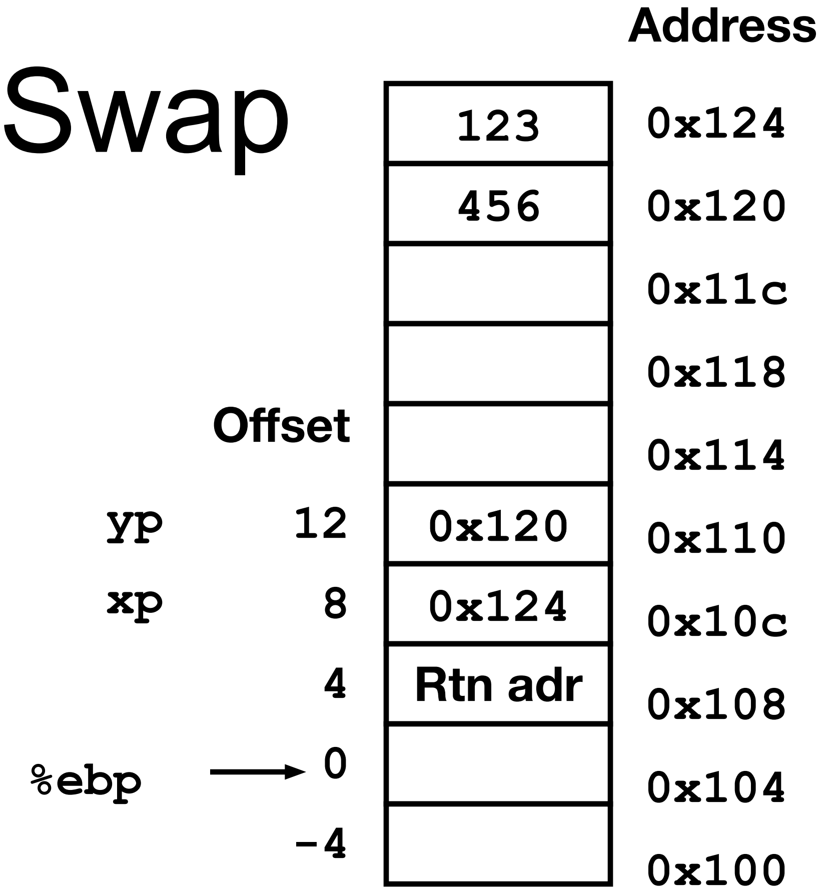





```
movl 12(%ebp),%ecx    # ecx = yp
movl 8(%ebp),%edx     # edx = xp
movl (%ecx),%eax      # eax = *yp (t1)
movl (%edx),%ebx      # ebx = *xp (t0)
movl %eax,(%edx)      # *xp = eax
movl %ebx,(%ecx)      # *yp = ebx
```

Understanding Swap






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

Understanding Swap

Register	Value
%eax	
%edx	0x124
%ecx	0x120
%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 12(%ebp),%ecx    # ecx = yp
movl 8(%ebp),%edx     # edx = xp
movl (%ecx),%eax      # eax = *yp (t1)
movl (%edx),%ebx      # ebx = *xp (t0)
movl %eax,(%edx)      # *xp = eax
movl %ebx,(%ecx)      # *yp = ebx
```

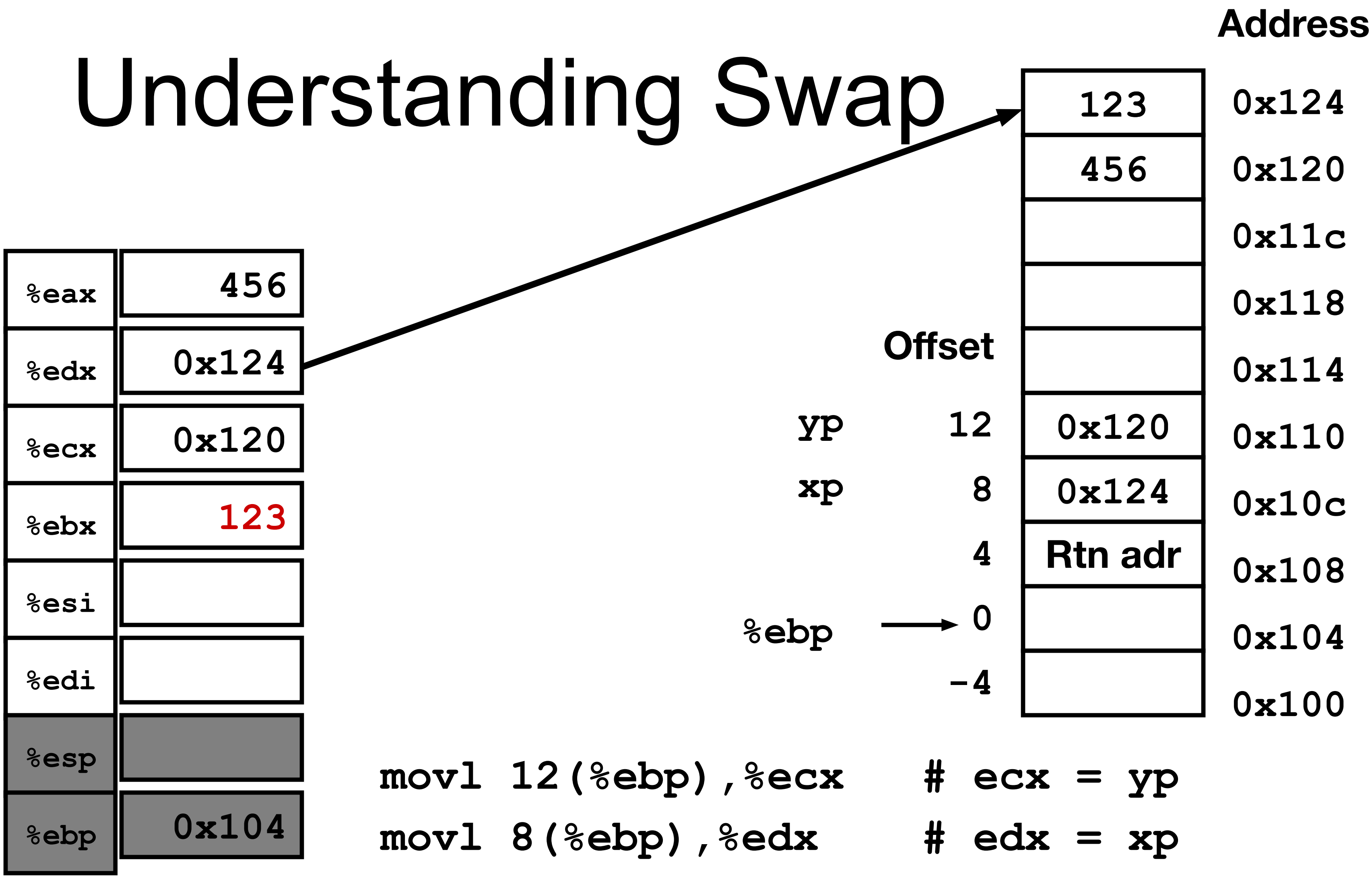
Understanding Swap

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

	Offset	Value	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

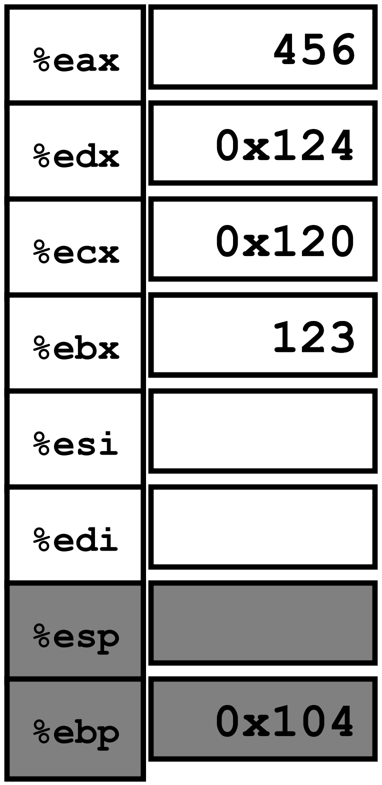
```
movl 12(%ebp),%ecx     # ecx = yp
movl 8(%ebp),%edx      # edx = xp
movl (%ecx),%eax       # eax = *yp (t1)
movl (%edx),%ebx       # ebx = *xp (t0)
movl %eax,(%edx)       # *xp = eax
movl %ebx,(%ecx)       # *yp = ebx
```

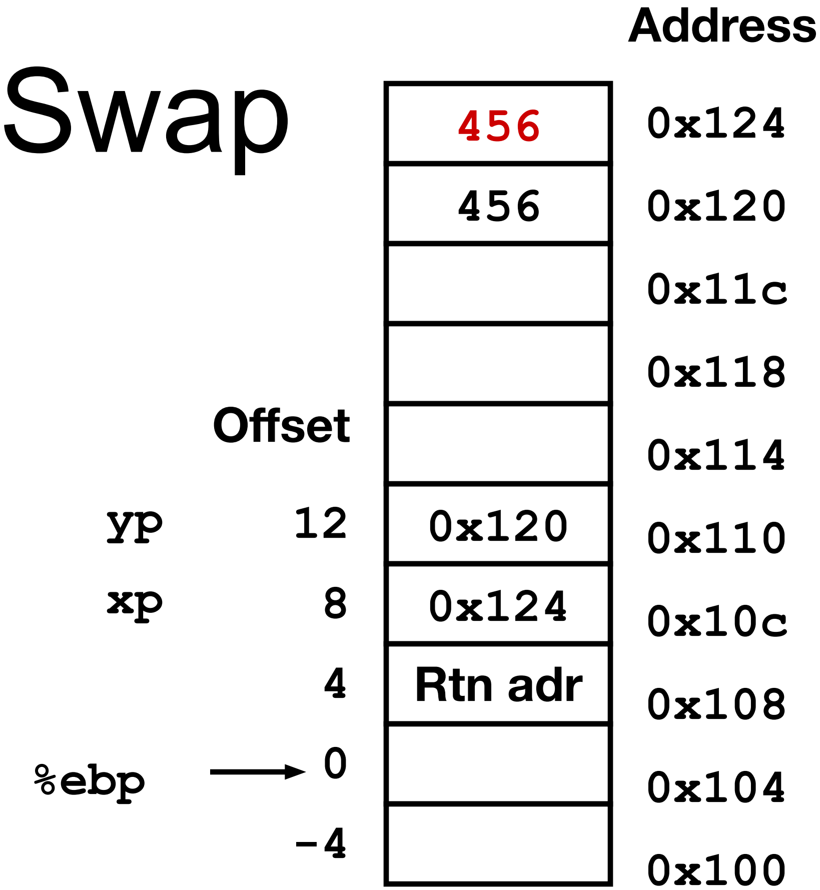
Understanding Swap




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

Understanding Swap






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

Understanding Swap

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

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

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

Experience with x86 assembly and how the stack is used to support procedure calls

I have studied this, and I remember it well	0%
I have studied this, but I will need a refresher	0%
I have never studied this, so I am a beginner	0%

Week 3 Q&A

Nobody has responded yet.

Hang tight! Responses are coming in.