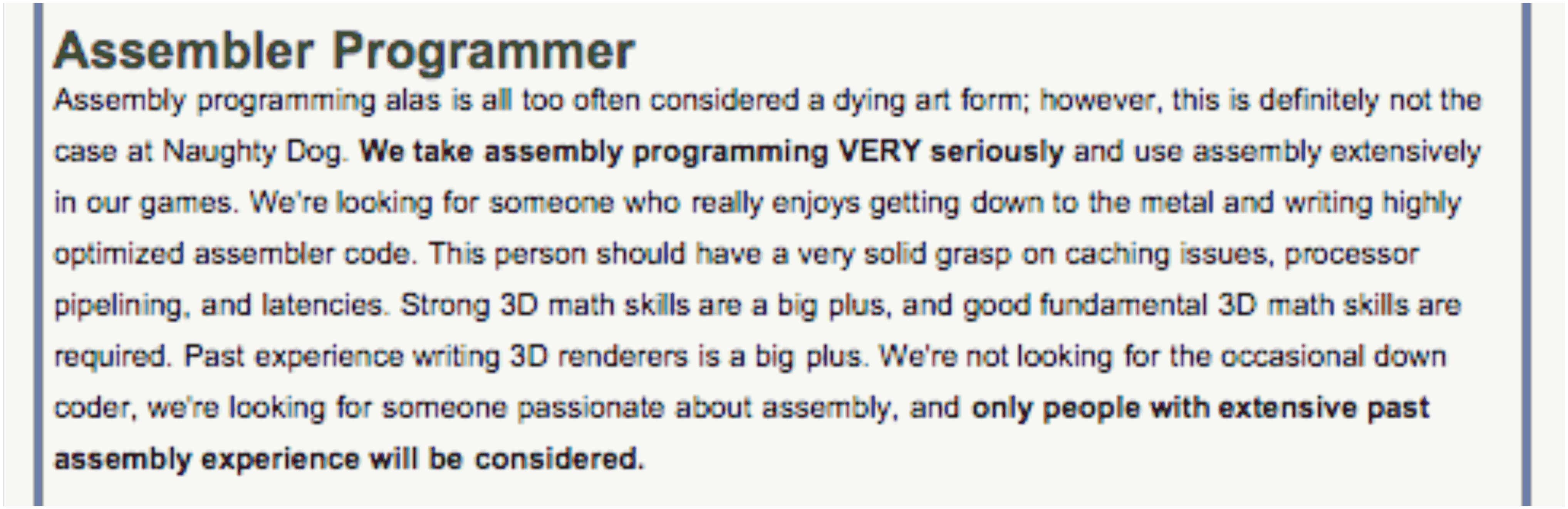
A job ad at a game programming company

WELCOME TO
NAUGHTY DOG ®



Assembler Programmer

Assembly programming alas is all too often considered a dying art form; however, this is definitely not the case at Naughty Dog. **We take assembly programming VERY seriously** and use assembly extensively in our games. We're looking for someone who really enjoys getting down to the metal and writing highly optimized assembler code. This person should have a very solid grasp on caching issues, processor pipelining, and latencies. Strong 3D math skills are a big plus, and good fundamental 3D math skills are required. Past experience writing 3D renderers is a big plus. We're not looking for the occasional down coder, we're looking for someone passionate about assembly, and **only people with extensive past assembly experience will be considered.**

Assembly Programming

- Why do they take assembly programming “very seriously”?

Assembly Programming

- Why do they take assembly programming “very seriously”?
  - Compilers don’t always generate the best possible code
  - Especially for computationally-intensive code
    - Like graphics, signal processing, physical simulation, etc.

  - An assembly programmer can use application/domain knowledge
    - Knowledge that some variables won’t change during computation
    - Knowledge of what precision is required
    - Knowledge that operations can be reordered/pipelined

  - There is often not a good mapping from C to some ISA features
  - Good programmers are more creative than compilers (holistic)

- Generally only works for “small” pieces of code
  - Humans are easily overwhelmed (our caches thrash)

RISC vs. CISC

- SPARC, PowerPC, and ARM are all very similar to MIPS, so you should have no problem learning them on your own, if needed.
- Today, we’ll look at x86, which has some significant differences of which you should be aware.

RISC vs. CISC

- SPARC, PowerPC, and ARM are all very similar to MIPS, so you should have no problem learning them on your own, if needed.
- Today, we'll look at x86, which has some significant differences of which you should be aware.

Comparing x86 and MIPS

- Much more is similar than different.
  - Both use registers and have byte-addressable memories
  - Same basic types of instructions (arithmetic, branches, memory)

- Differences

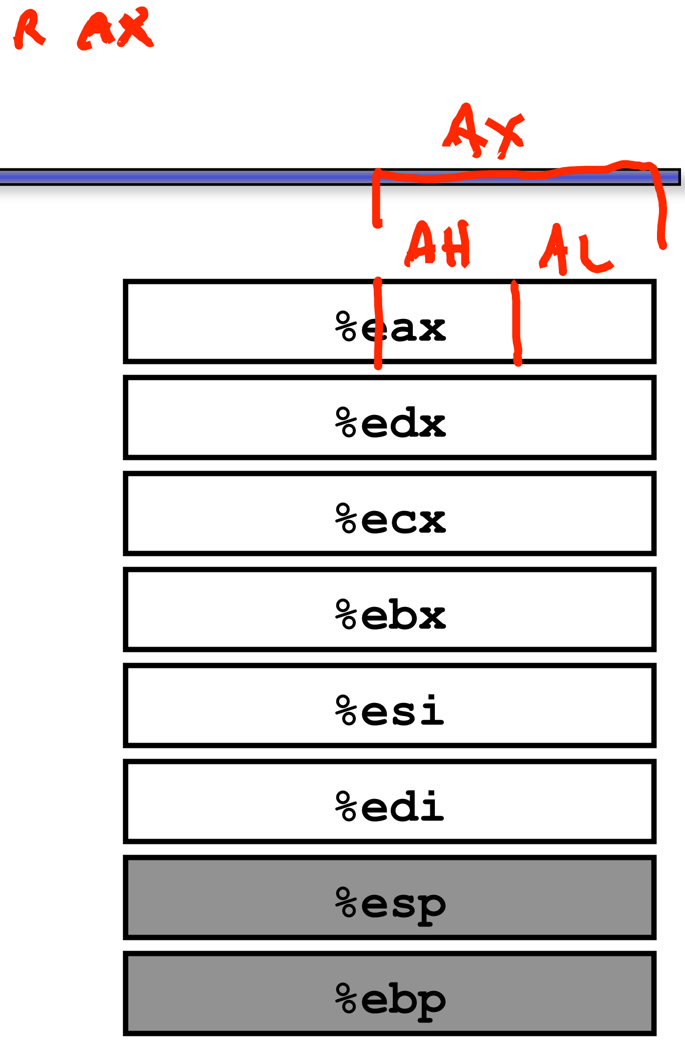
  %eax 64b ⇒ 16 register

  - Fewer (8) registers, different names
  - Two register formats (x86) vs. three (MIPS)
  - Greater reliance on the stack, which is part of the architecture
  - x86 arithmetic supports (register + memory) -> (register) format
  - x86 has additional addressing modes
  - x86 branches use condition codes
  - different instruction names and variable-length encodings

- I'll walk you through the tricky parts

x86 Registers

- Few, and special purpose
  - 8 integer registers
  - two generally used only for stack
  - Not all instructions can use any register
- Little room for temporary values
  - x86 uses “two-address code”
  - **op x, y** # y = y op x
- Rarely can the compiler fit everything in registers
  - Stack is used much more heavily



%eax
%edx
%ecx
%ebx
%esi
%edi
%esp
%ebp

x86 Stack is Architected! (Not just a convention)

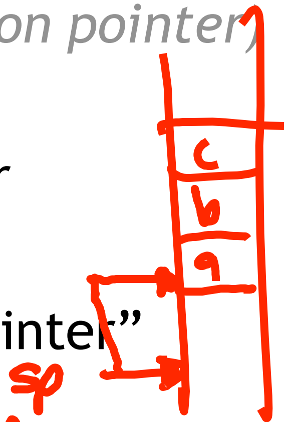
- The **esp** register _is_ the stack pointer

- x86 includes explicit **push** and **pop** instructions
  - **push %eax** # M[ESP - 4] = EAX; ESP = ESP - 4
  - **pop %ecx** # ESP = ESP + 4; ECX = M[ESP - 4]
  - *It can be seen that, like MIPS, the x86 stack grows down*

- **call** instructions (x86 equivalent to **jal**) push the return address on stack
  - **call label** # push next EIP; EIP = label *(EIP = instruction pointer)*

- Stack also used for passing arguments, pushed in reverse order

f(a,b,c) push(c), push(b), push(a)



- Because **esp** is constantly changing, use **ebp** as stack "base pointer"
  - Keeps track of the top of the current stack frame
    - Same as the bottom of the previous stack frame
  - Doesn't move, so can be used throughout the function

A Simple Example

```
int main() {
   int one = 123, two = 456;
   swap(&one, &two);
   ...
}
```

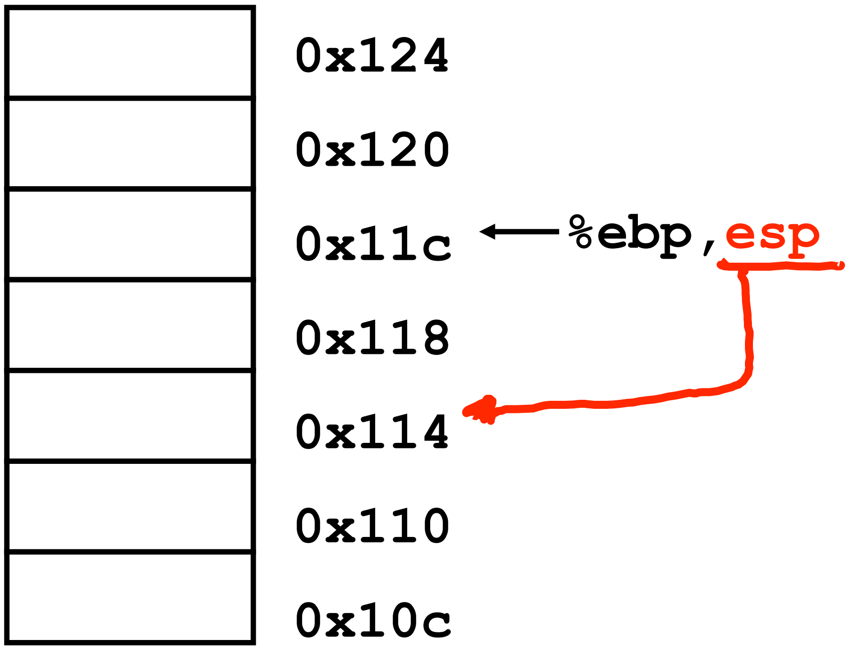
one = 456 , two = 123

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

A Simple Example

```
int main() {
  int one = 123, two = 456;
  swap(&one, &two);
  ...
}
```

```
…
subl     $8, %esp
movl     $123, -8(%ebp)
movl          $456, -4(%ebp)
leal          -4(%ebp), %eax
pushl    %eax
leal          -8(%ebp), %eax
pushl    %eax
call          swap
...
```



Key:

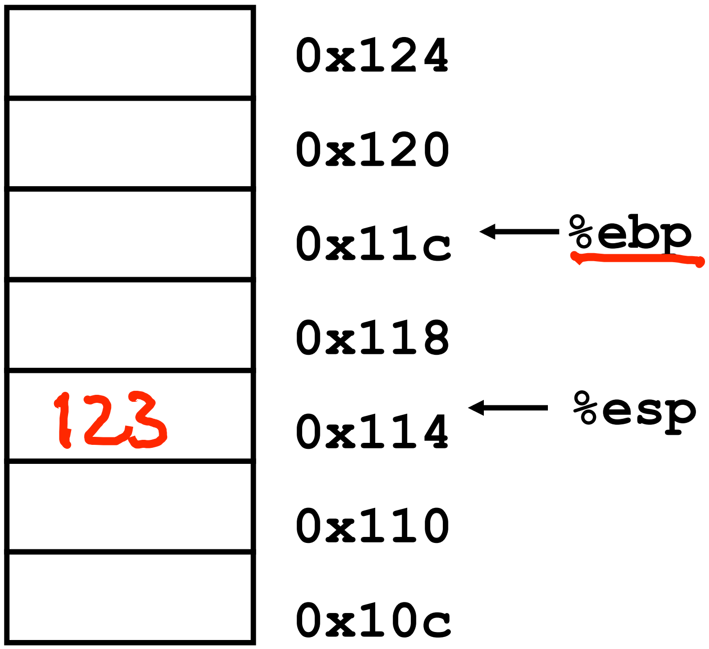
sub	= subtract
l	= long (32-bit operation)
$8	= literal 8
%esp	= stack pointer register

ESP = ESP - 8

A Simple Example

```
int main() {
   int one = 123, two = 456;
   swap(&one, &two);
   ...
}
```

```
…
subl     $8, %esp
movl     $123, -8(%ebp)
movl           $456, -4(%ebp)
leal           -4(%ebp), %eax
pushl    %eax
leal           -8(%ebp), %eax
pushl    %eax
call           swap
...
```



Key:

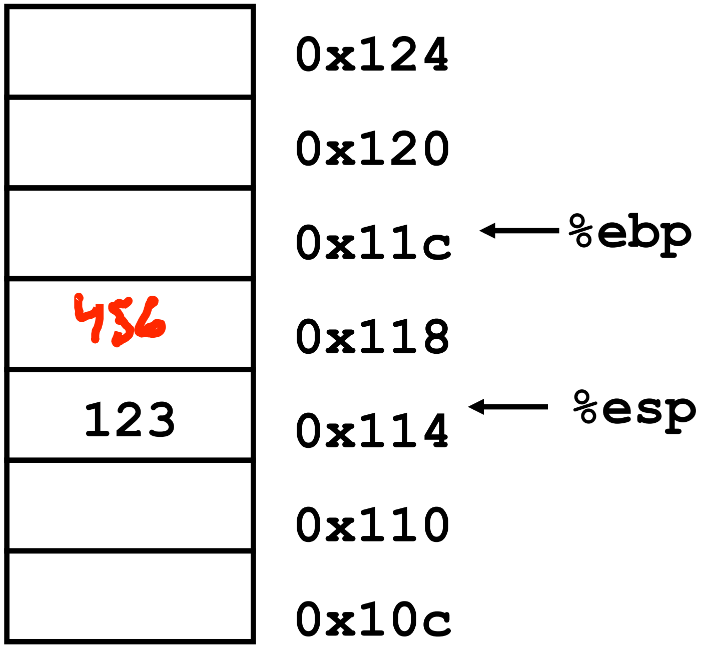
mov	= data movement
l	= long (32-bit operation)
$123	= literal 123
-8(%ebp)	= base + offset addressing

M[EBP - 8] = 123

A Simple Example

```
int main() {
  int one = 123, two = 456;
  swap(&one, &two);
  ...
}
```

```
…
subl    $8, %esp
movl    $123, -8(%ebp)
movl         $456, -4(%ebp)
leal         -4(%ebp), %eax
pushl   %eax
leal         -8(%ebp), %eax
pushl   %eax
call         swap
...
```



0x124

0x120

0x11c ← %ebp

456 0x118

123 0x114 ← %esp

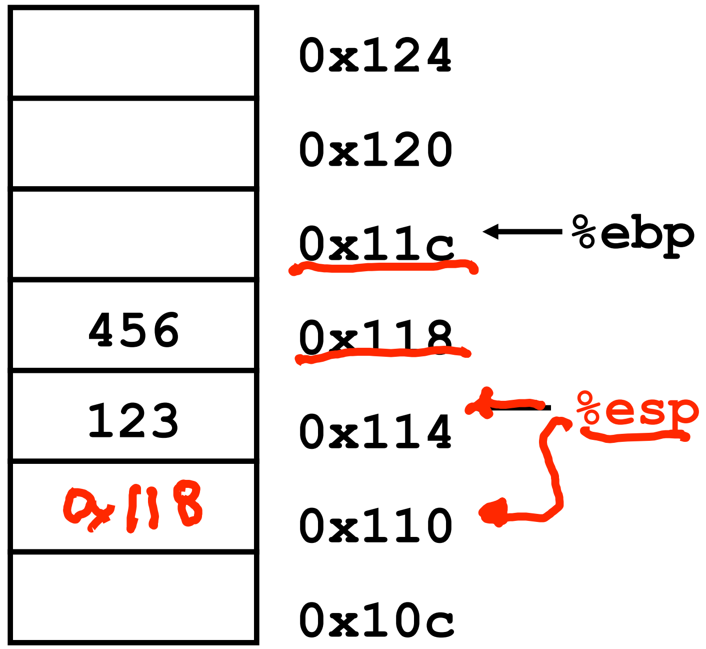
0x110

0x10c

A Simple Example

```
int main() {
  int one = 123, two = 456;
  swap(&one, &two);
  ...
}
```

```
…
subl     $8, %esp
movl     $123, -8(%ebp)
movl          $456, -4(%ebp)
leal          -4(%ebp), %eax
pushl    %eax
leal          -8(%ebp), %eax
pushl    %eax
call          swap
...
```



Key:

(push arguments in reverse order)

lea = load effective address

(don't do a load, just compute addr.)

EAX = EBP - 4

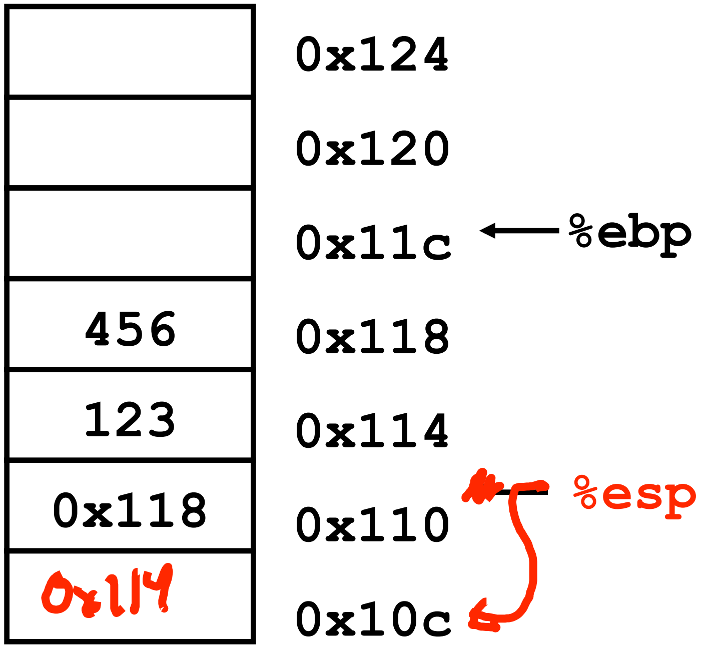
M[ESP - 4] = EAX

ESP = ESP - 4

A Simple Example

```
int main() {
  int one = 123, two = 456;
  swap(&one, &two);
  ...
}
```

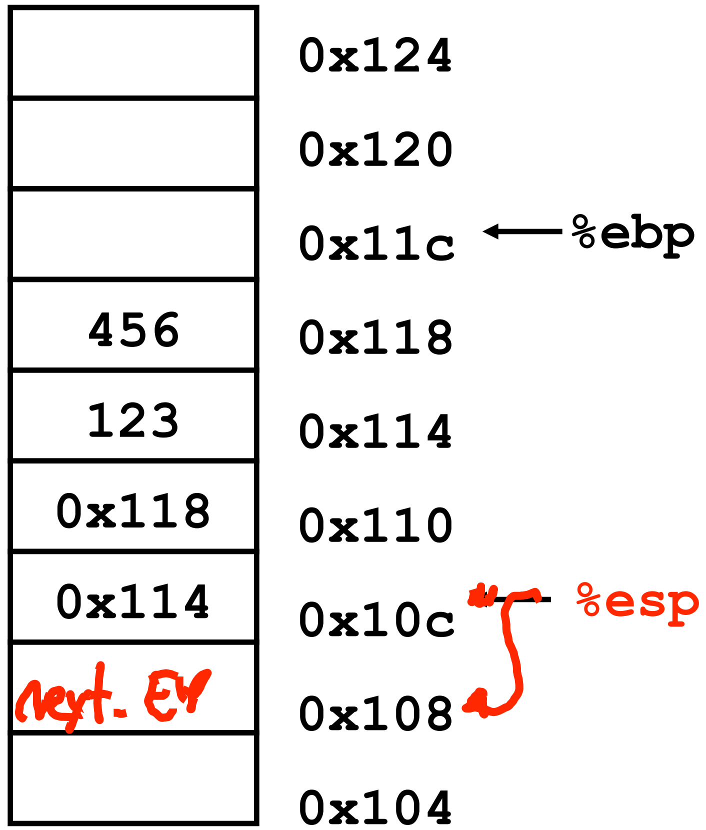
```
…
subl     $8, %esp
movl     $123, -8(%ebp)
movl          $456, -4(%ebp)
leal          -4(%ebp), %eax
pushl    %eax
leal          -8(%ebp), %eax
pushl    %eax
call          swap
...
```



A Simple Example

```
int main() {
  int one = 123, two = 456;
  swap(&one, &two);
  ...
}
```

```
…
subl    $8, %esp
movl    $123, -8(%ebp)
movl         $456, -4(%ebp)
leal         -4(%ebp), %eax
pushl   %eax
leal         -8(%ebp), %eax
pushl   %eax
call         swap
...
```



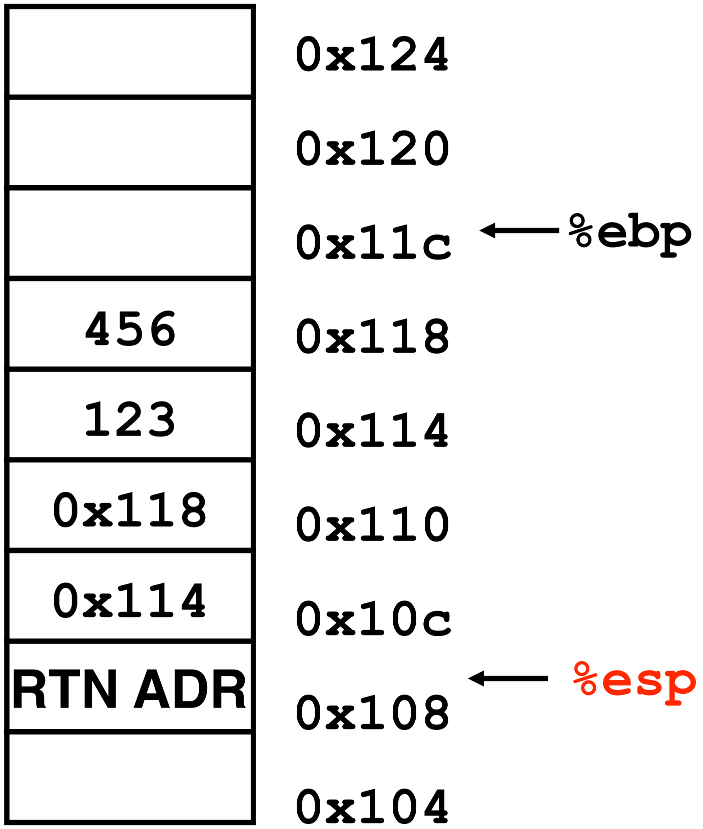
	0x124
	0x120
	0x11c ← %ebp
456	0x118
123	0x114
0x118	0x110
0x114	0x10c ← %esp
next. EIP	0x108
	0x104

Key:
M[ESP - 4] = next_EIP
ESP = ESP - 4
EIP = swap

A Simple Example

```
int main() {
  int one = 123, two = 456;
  swap(&one, &two);
  ...
}
```

```
…
subl     $8, %esp
movl     $123, -8(%ebp)
movl          $456, -4(%ebp)
leal          -4(%ebp), %eax
pushl    %eax
leal          -8(%ebp), %eax
pushl    %eax
call          swap
...
```



	0x124
	0x120
	0x11c ← %ebp
456	0x118
123	0x114
0x118	0x110
0x114	0x10c
RTN ADR	0x108 ← %esp
	0x104

The “swap” function

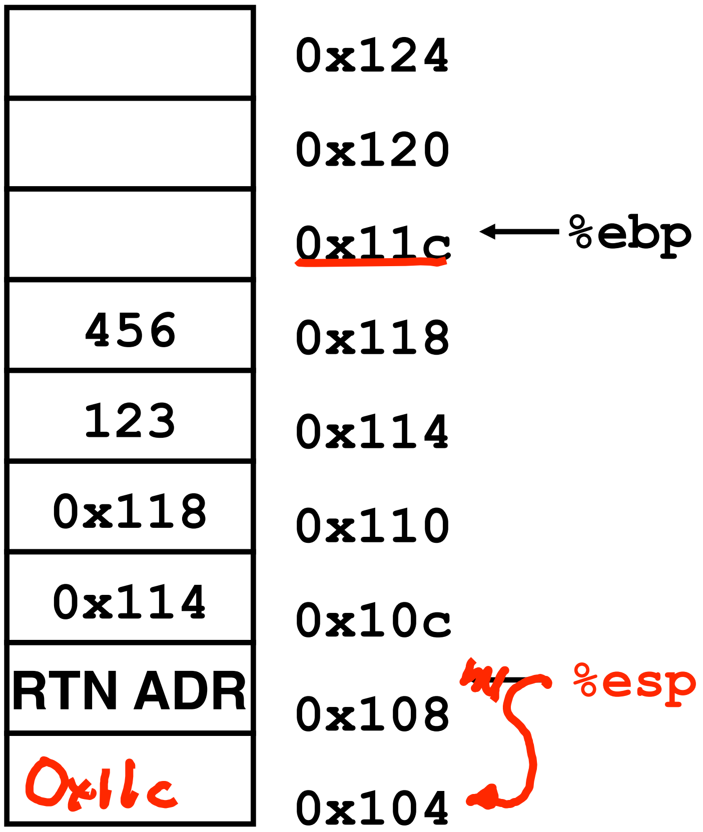
```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

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

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

   movl  -4(%ebp),%ebx
   movl  %ebp,%esp                Finish
   popl  %ebp
   ret
```

Function Prologue

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

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



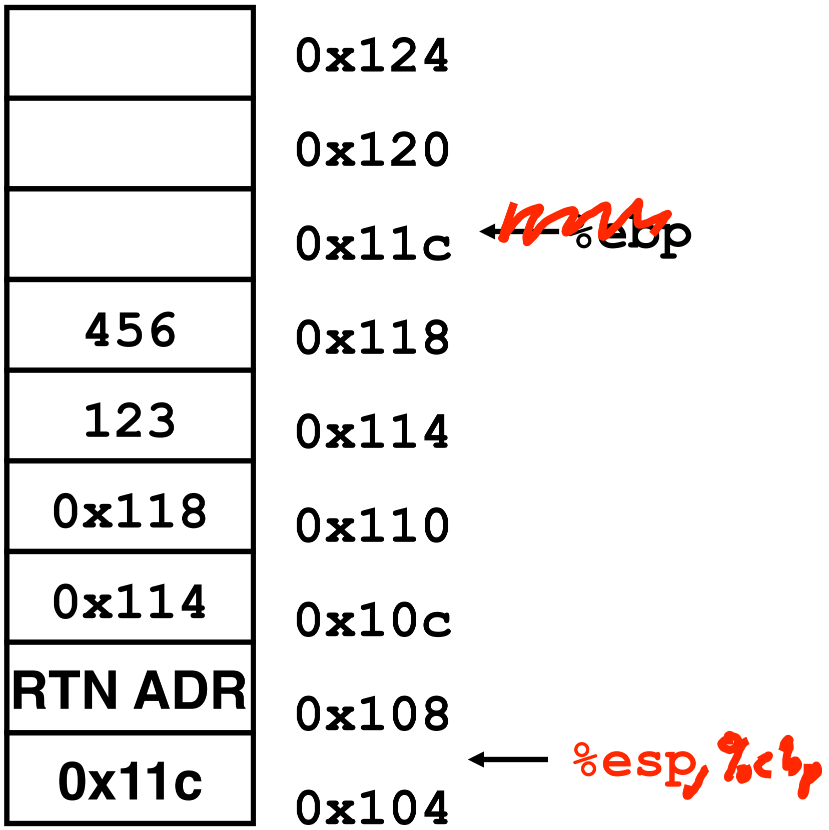
	0x124
	0x120
	0x11c ← %ebp
456	0x118
123	0x114
0x118	0x110
0x114	0x10c
RTN ADR	0x108 %esp
0x11c	0x104

Save the old base pointer on the stack

Function Prologue

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

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



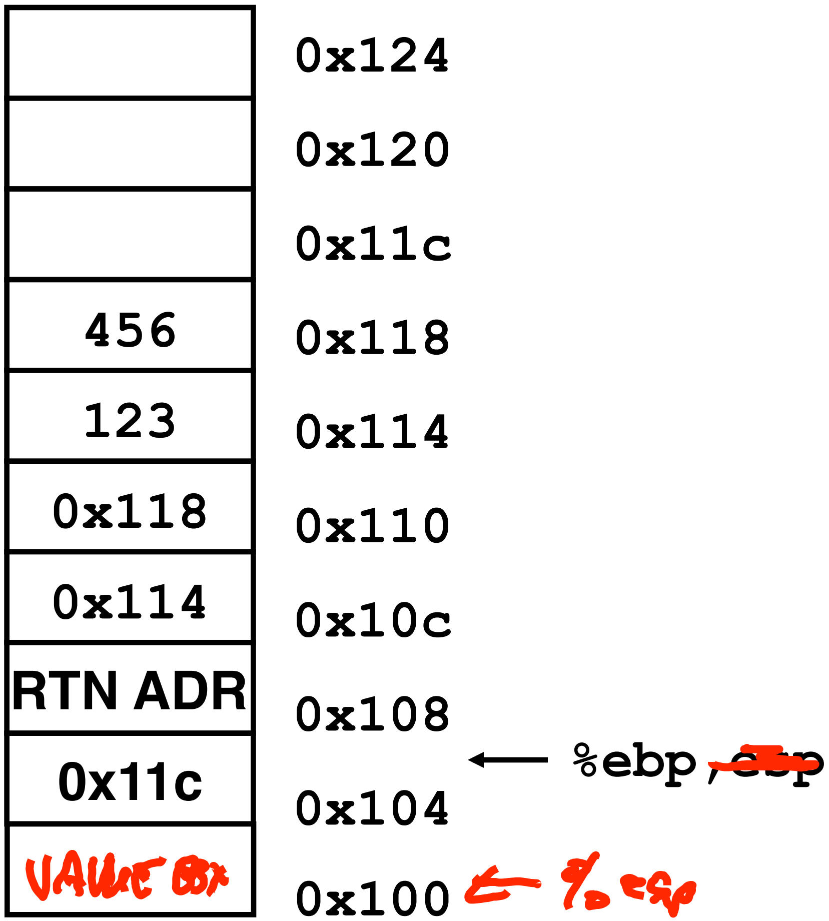
	0x124
	0x120
	0x11c ← ~~%ebp~~
456	0x118
123	0x114
0x118	0x110
0x114	0x10c
RTN ADR	0x108
0x11c	0x104 ← %esp, %ebp

Old stack pointer becomes new base pointer.

Function Prologue

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

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

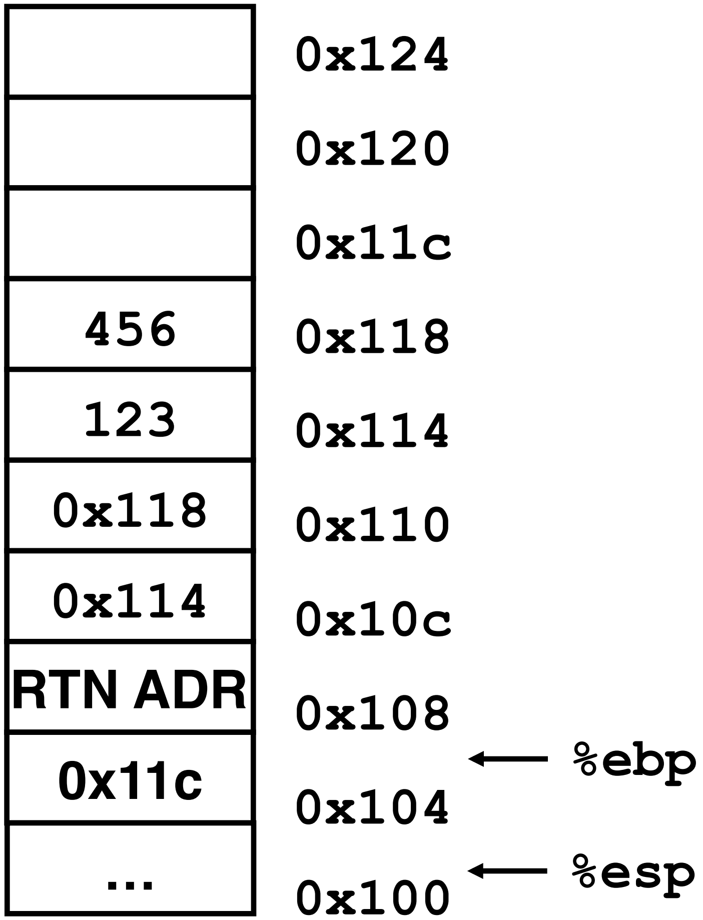


*Save register **ebx**, which is callee saved.*

Function Prologue

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

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



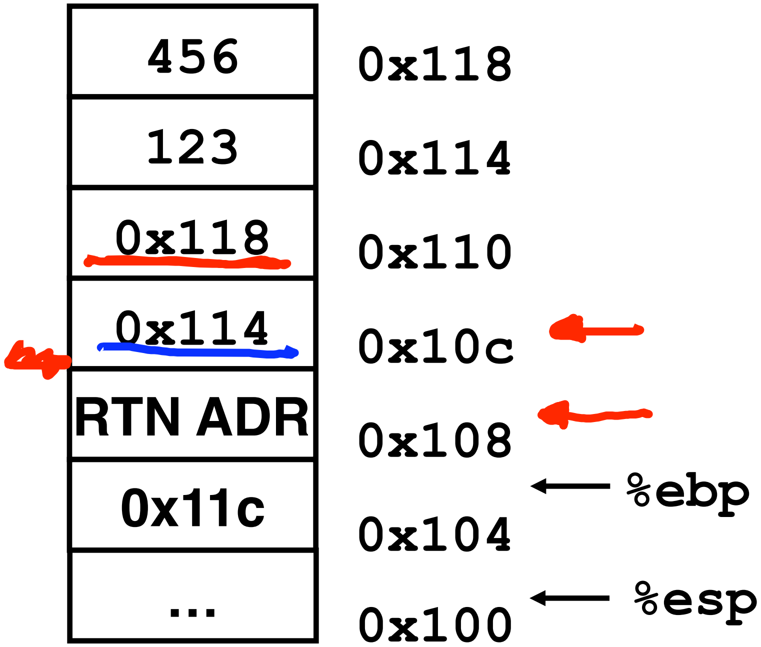
	0x124	
	0x120	
	0x11c	
456	0x118	
123	0x114	
0x118	0x110	
0x114	0x10c	
RTN ADR	0x108	
0x11c	0x104	← %ebp
...	0x100	← %esp

*Save register **ebx**, which is callee saved.*

The swap itself

```
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



456	0x118
123	0x114
0x118	0x110
0x114	0x10c
RTN ADR	0x108
0x11c	0x104
...	0x100

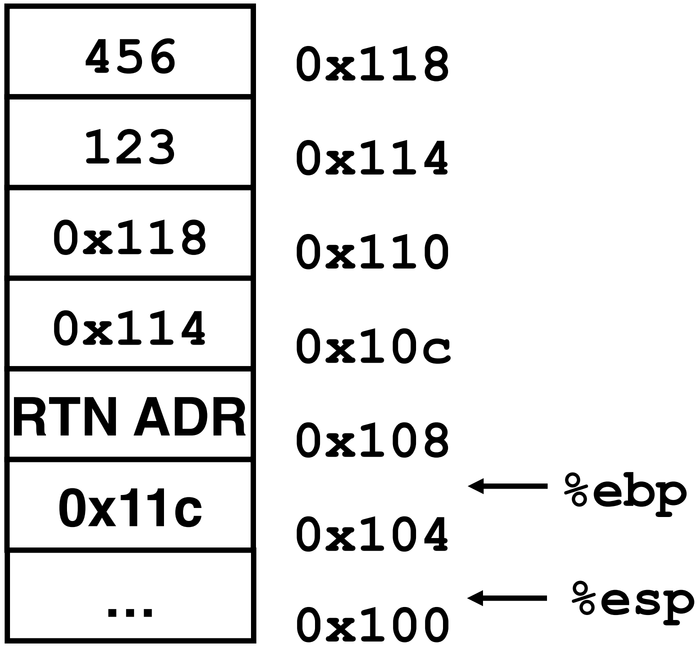
```
movl 12(%ebp),%ecx     # ecx = yp
movl 8(%ebp),%edx      # edx = xp
movl (%ecx),%eax       # eax = *yp (t1)
movl (%edx),%ebx       # ebx = *xp (t0)
movl %eax,(%edx)       # *xp = eax
movl %ebx,(%ecx)       # *yp = ebx
```

%eax	
%edx	%0x114
%ecx	0x118
%ebx	

The swap itself

```
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



456	0x118	
123	0x114	
0x118	0x110	
0x114	0x10c	
RTN ADR	0x108	
0x11c	0x104	%ebp
...	0x100	%esp

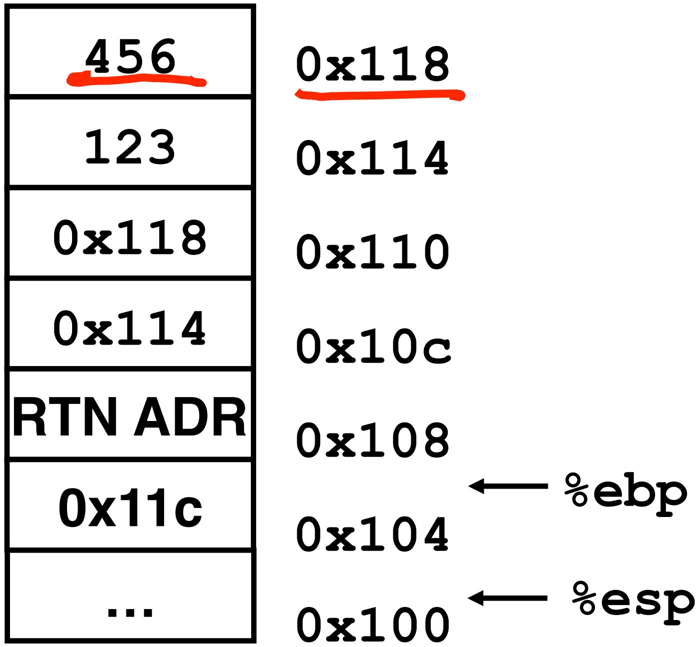
```
movl 12(%ebp),%ecx      # ecx = yp
movl 8(%ebp),%edx       # edx = xp
movl (%ecx),%eax        # eax = *yp (t1)
movl (%edx),%ebx        # ebx = *xp (t0)
movl %eax,(%edx)        # *xp = eax
movl %ebx,(%ecx)        # *yp = ebx
```

%eax	
%edx	0x114
%ecx	0x118
%ebx	

The swap itself

```
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



456	0x118	
123	0x114	
0x118	0x110	
0x114	0x10c	
RTN ADR	0x108	
0x11c	0x104	← %ebp
...	0x100	← %esp

```
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
```

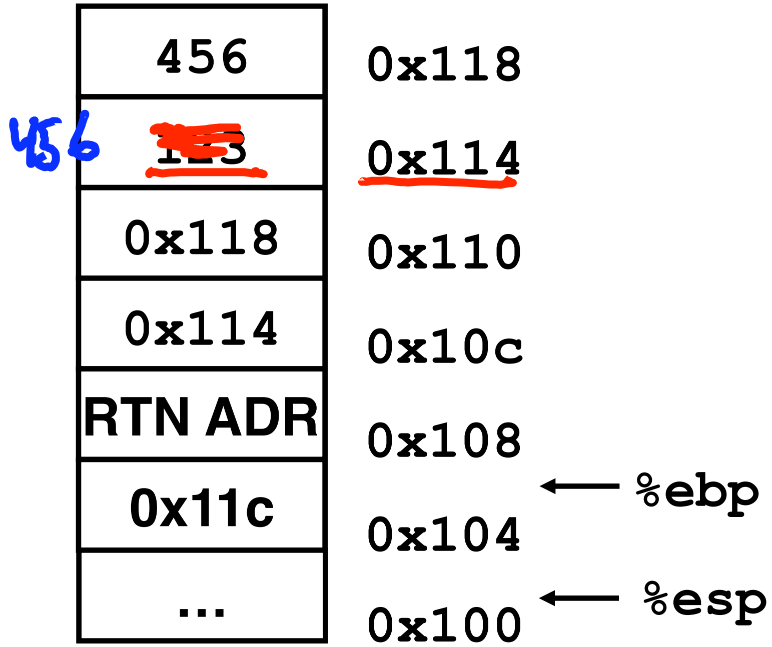
%eax	456
%edx	0x114
%ecx	0x118
%ebx	

The swap itself

```
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
```



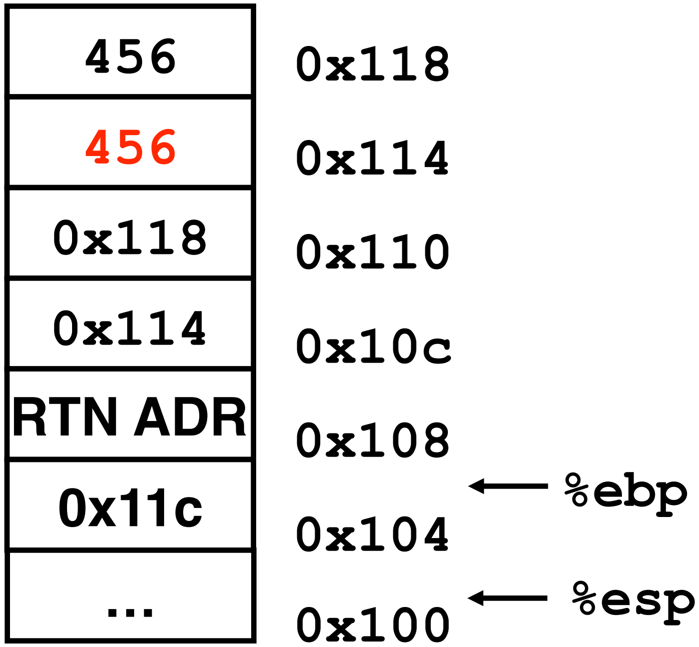
456	0x118
456 ~~123~~	0x114
0x118	0x110
0x114	0x10c
RTN ADR	0x108
0x11c	0x104
...	0x100

%eax	456
%edx	0x114
%ecx	0x118
%ebx	123

The swap itself

```
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



456	0x118
456	0x114
0x118	0x110
0x114	0x10c
RTN ADR	0x108
0x11c	0x104
...	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
```

%eax	456
%edx	0x114
%ecx	0x118
%ebx	123

The swap itself

```
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
```

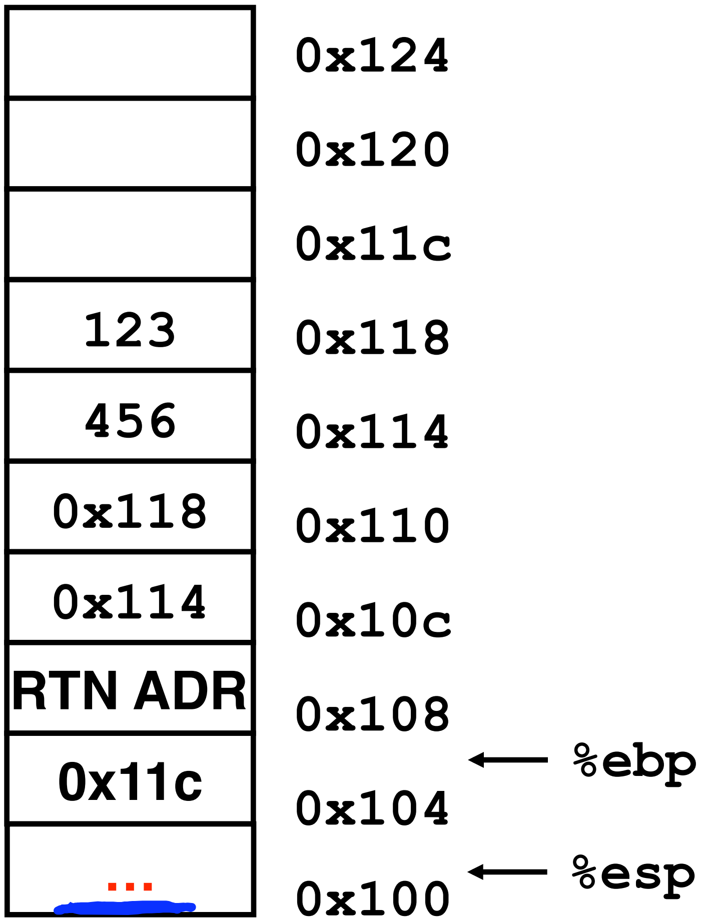
123	0x118	
456	0x114	
0x118	0x110	
0x114	0x10c	
RTN ADR	0x108	
0x11c	0x104	← %ebp
...	0x100	← %esp

%eax	456
%edx	0x114
%ecx	0x118
%ebx	123

Function Epilogue

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

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```



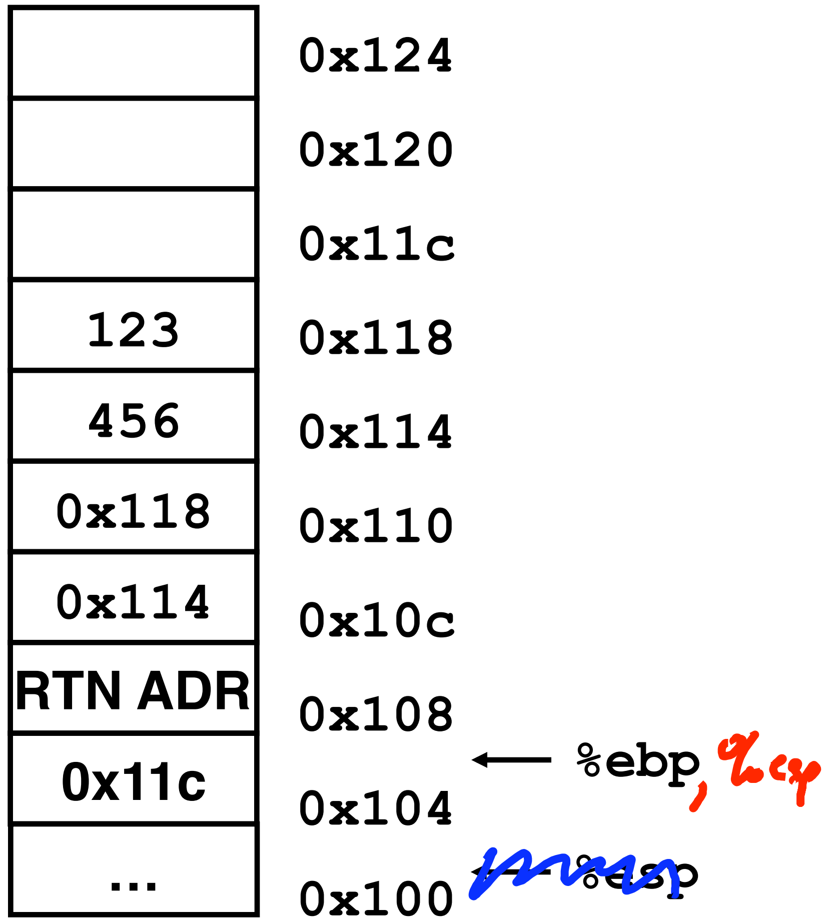
	0x124
	0x120
	0x11c
123	0x118
456	0x114
0x118	0x110
0x114	0x10c
RTN ADR	0x108
0x11c	0x104
...	0x100

Restore register ***ebx***.

Function Epilogue

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

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```



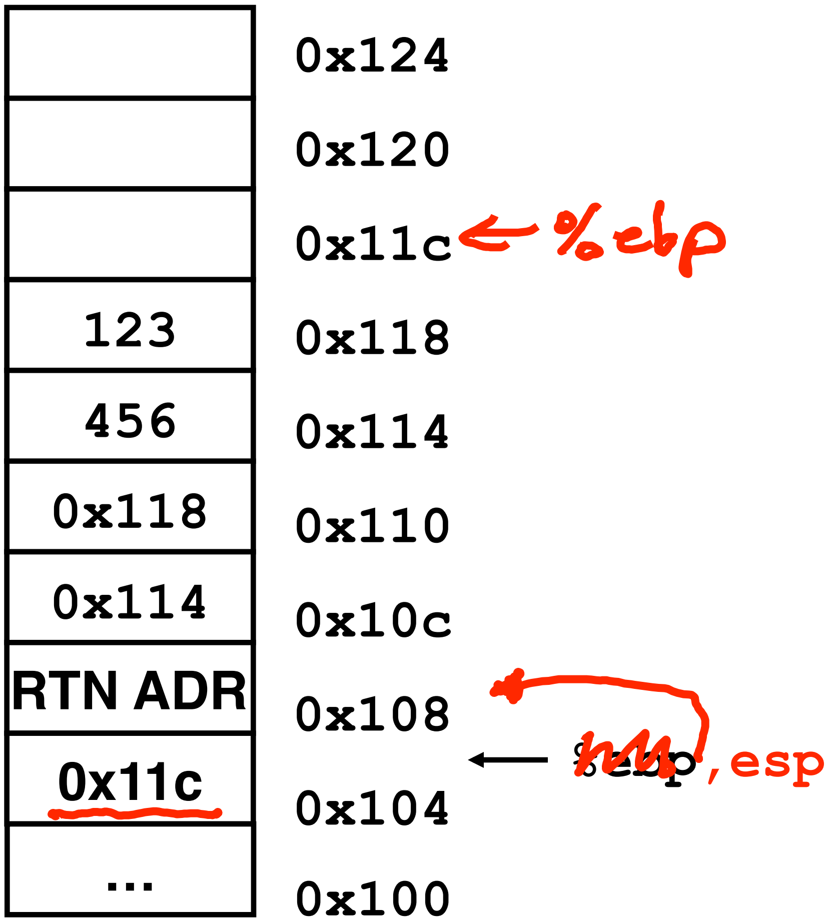
	0x124	
	0x120	
	0x11c	
123	0x118	
456	0x114	
0x118	0x110	
0x114	0x10c	
RTN ADR	0x108	
0x11c	0x104	← %ebp, %esp
...	0x100	~~← %esp~~

Copy the base pointer to the stack pointer.

Function Epilogue

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

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```



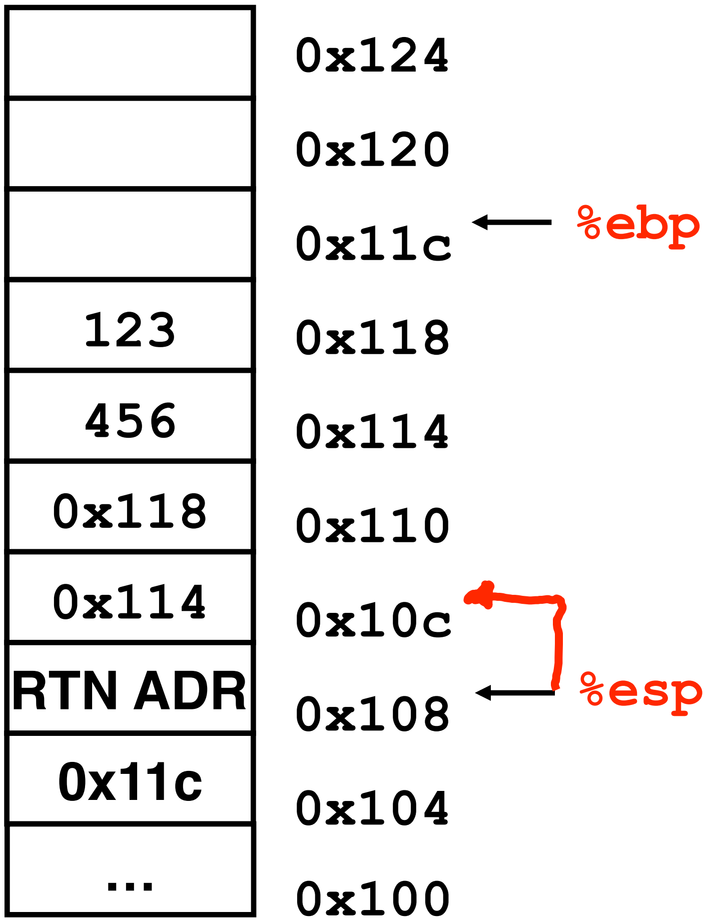
	0x124
	0x120
	0x11c ← %ebp
123	0x118
456	0x114
0x118	0x110
0x114	0x10c
RTN ADR	0x108
0x11c	0x104 ← ~~%ebp~~, esp
...	0x100

Restore the old base pointer.

Function Epilogue

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

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```



	0x124	
	0x120	
	0x11c	%ebp
123	0x118	
456	0x114	
0x118	0x110	
0x114	0x10c	
RTN ADR	0x108	%esp
0x11c	0x104	
...	0x100	

Return, which pops the return address off the stack

Memory Operands

- Most instructions (not just mov) can include a memory operand
  - **addl -8(%ebp), %eax** # EAX = EAX + M[EBP - 8]
  - **incl -8(%ebp)** # M[EBP - 8] = M[EBP - 8] + 1
- More complex addressing modes are supported
  - 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
  - Useful for accessing arrays of scalars (including those within structs)

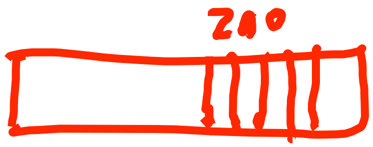
Address Computation Examples

%edx	0xf000
%ecx	0x100

Expression	Computation	Address
0x8(%edx)	0xf000 + 0x8	0xf008
(%edx,%ecx)	0xf000 + 0x100	0xf100
(%edx,%ecx,4)	0xf000 + 4*0x100	0xf400
0x80(,%edx,2)	2*0xf000 + 0x80	0x1e080

Control Flow = Condition Codes

- Conditional control flow is a two step process:
  - Setting a condition code (held in the EFLAGS register)
    - done by most arithmetic operations
  - Branching based on a condition code bit



- Standard sequence involves using the compare (**cmp**) instruction
  - Compare acts like a subtract, but doesn’t write dest. register

test
and

```
cmp        8(%ebx), %eax     # set flags based on (EAX - M[EBX + 8])
jg         branch_target     # taken if (EAX > M[EBX + 8])
```

Control Flow Example

```
int sum(int n) {
  int i, sum = 0;
  for (i = 1 ; i <= n ; ++ i) {
    sum += i;
  }
  return sum;
}
```

Control Flow Example

```
int sum(int n) {
  int i, sum = 0;
  for (i = 1 ; i <= n ; ++ i) {
    sum += i;
  }
  return sum;
}
```

```
sum:    pushl    %ebp
        movl     %esp, %ebp
        movl     8(%ebp), %ecx  # n (was argument)
        movl     $1, %edx       # i = 1
        xorl     %eax, %eax     # sum = 0
        cmpl     %ecx, %edx     # (i ? n), sets cond. codes
        jg       .L8            # branch if (i > n)
.L6:
        addl     %edx, %eax     # sum += i
        incl     %edx           # i += 1
        cmpl     %ecx, %edx     # (i ? n)
        jle      .L6            # branch if (i <= n)
.L8:
```

Variable Length Instructions

```
08048344 <sum>:
 8048344:       55                        push    %ebp
 8048345:       89 e5                     mov     %esp,%ebp
 8048347:       8b 4d 08                  mov     0x8(%ebp),%ecx
 804834a:       ba 01 00 00 00            mov     $0x1,%edx
 804834f:       31 c0                     xor     %eax,%eax
 8048351:       39 ca                     cmp     %ecx,%edx
 8048353:       7f 0a                     jg      804835f
 8048355:       8d 76 00                  lea     0x0(%esi),%esi
   ...
 804835f:       c9                        leave
 8048360:       c3                        ret
```

- Instructions range in size from 1 to 17 bytes
  - Commonly used instructions are short (think compression)
    - In general, x86 has smaller code than MIPS
- Many different instruction formats, plus pre-fixes, post-fixes
  - Harder to decode for the machine (more on this later)

Why did Intel win?

x86 won because it was the first 16-bit chip by two years.

- IBM put it in PCs because there was no competing choice
- Rest is inertia and “financial feedback”
  - x86 is most difficult ISA to implement for high performance, but
  - Because Intel sells the most processors ...
  - It has the most money ...
  - Which it uses to hire more and better engineers ...
  - Which is uses to maintain competitive performance ...
  - And given equal performance, compatibility wins ...
  - So Intel sells the most processors.