x86 basics

ISA context and x86 history
Translation tools: C --> assembly <---> machine code

x86 Basics:
- Registers
- Data movement instructions
- Memory addressing modes
- Arithmetic instructions
Instruction Set Architecture (HW/SW Interface)

**Instructions**
- Names, Encodings
- Effects
- Arguments, Results

**Local storage**
- Names, Size
- How many

**Large storage**
- Addresses, Locations

**Computer**

**Processor**
- Instruction Logic
- Registers

**Memory**
- Encoded Instructions
- Data
**a brief history of x86**

<table>
<thead>
<tr>
<th>Word Size</th>
<th>ISA</th>
<th>First</th>
<th>Year</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>8086</td>
<td>Intel 8086</td>
<td>1978</td>
<td>16-bit processor. Basis for IBM PC &amp; DOS. 1MB address space.</td>
</tr>
<tr>
<td>32</td>
<td>IA32</td>
<td>Intel 386</td>
<td>1985</td>
<td>32-bit ISA. Flat addressing, improved OS support.</td>
</tr>
<tr>
<td>64</td>
<td>x86-64</td>
<td>AMD Opteron</td>
<td>2003*</td>
<td>Slow AMD/Intel conversion, slow adoption. Not actually x86-64 until few years later. Mainstream only after ~10 years.</td>
</tr>
</tbody>
</table>

*2015: most laptops, desktops, servers.*
Turning C into Machine Code

C Code

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

gcc -O1 -S code.c
code.s

Generated IA32 Assembly Code

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

code.o

Object Code

```
0101010110001001101111001010100
0010101100100101000111000101100
00110010010100011100010010010
01011101100010111111110000111

```

code.o

Linker: create full executable
Resolve references between object files, libraries, (re)locate data
Disassembling Object Code (objdump)

Disassembled by objdump

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

Disassembler: objdump -d p

Disassembler

01010101100010011110010110
00101101000101000011100000
00110100010100001000100010
01111011000101111011100011

code.o

Disassembling Object Code (objdump)
Disassembling Object Code (gdb)

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

Disassembled by GDB

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x401040</td>
<td>push %ebp</td>
</tr>
<tr>
<td>0x401041</td>
<td>mov %esp,%ebp</td>
</tr>
<tr>
<td>0x401043</td>
<td>mov 0xc(%ebp),%eax</td>
</tr>
<tr>
<td>0x401046</td>
<td>add 0x8(%ebp),%eax</td>
</tr>
<tr>
<td>0x401049</td>
<td>mov %ebp,%esp</td>
</tr>
<tr>
<td>0x40104b</td>
<td>pop %ebp</td>
</tr>
<tr>
<td>0x40104c</td>
<td>ret</td>
</tr>
</tbody>
</table>

> gdb p
(gdb) disassemble sum
(disassemble function)
(gdb) x/13b sum
(examine the 13 bytes starting at sum)
Integer Registers (IA32)

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

Some have special uses for particular instructions

Origin (mostly obsolete)
accumulate
counter
data
base
source
index
destination
index
stack
pointer
base
pointer

32-bits wide
Integer Registers (historical artifacts)

- %eax %ax %ah %al
- %ecx %cx %ch %cl
- %edx %dx %dh %dl
- %ebx %bx %bh %bl
- %esi
- %edi
- %esp %sp
- %ebp %bp

16-bit virtual registers (backwards compatible)

- accumulate
- counter
- data
- base
- source
- index
- destination
- index
- stack
- pointer
- base
- pointer
IA32: Three Basic Kinds of Instructions

1. **Data movement** between memory and register
   - **Load** data from memory into register
     \[
     \%\text{reg} = \text{Mem}[\text{address}]
     \]
   - **Store** register data into memory
     \[
     \text{Mem}[\text{address}] = \%\text{reg}
     \]

2. **Arithmetic/logic** on register or memory data
   - \( c = a + b; \)
   - \( z = x \ll y; \)
   - \( i = h \& g; \)

3. **Comparisons and Control flow** to choose next instruction
   - Unconditional jumps to/from procedures
   - Conditional branches

Memory is an array[] of bytes!
Data movement instructions

`movx Source, Dest`

- x is one of \{b, w, l\}
- gives size of data

`movl Source, Dest`:
- Move 4-byte “long word”

`movw Source, Dest`:
- Move 2-byte “word”

`movb Source, Dest`:
- Move 1-byte “byte”

historical terms from the 16-bit days not the current machine word size

%eax  
%ecx  
%edx  
%ebx  
%esi  
%edi  
%esp  
%ebp
Data movement instructions

**movl Source, Dest:**

**Operand Types:**

- **Immediate:** Literal integer data
  
  Examples: \$0x400, \$-533

- **Register:** One of 8 integer registers
  
  Examples: %eax, %edx

- **Memory:** 4 consecutive bytes in memory, at address held by register
  
  Simplest example: (%eax)

  Various other “address modes”

<table>
<thead>
<tr>
<th>%eax</th>
<th>%ecx</th>
<th>%edx</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ebx</td>
<td>%esi</td>
<td>%edi</td>
</tr>
<tr>
<td>%esp</td>
<td>%ebp</td>
<td></td>
</tr>
</tbody>
</table>
### movl Operand Combinations

<table>
<thead>
<tr>
<th>Source</th>
<th>Dest</th>
<th>Src,Dest</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imm</td>
<td>Reg</td>
<td>movl $0x4,%eax</td>
<td>var_a = 0x4;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl $-147,(%eax)</td>
<td>*p_a = -147;</td>
</tr>
<tr>
<td>Reg</td>
<td>Reg</td>
<td>movl %eax,%edx</td>
<td>var_d = var_a;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl %eax,(%edx)</td>
<td>*p_d = var_a;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl (%eax),%edx</td>
<td>var_d = *p_a;</td>
</tr>
</tbody>
</table>

**Cannot do memory-memory transfer with a single instruction.**

**How would you do it?**
Basic Memory Addressing Modes

Indirect  
(R)  
Mem[Reg[R]]

Register R specifies the memory address

\[
\text{movl \ (\%ecx), \%eax}
\]

Displacement  
D(R)  
Mem[Reg[R]+D]

Register R specifies a memory address

(e.g. the start of an object)

Constant displacement D specifies the offset from that address

(e.g. a field in the object)

\[
\text{movl \ 8(\%ebp), \%edx}
\]
Using Basic 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 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

Set Up

Body

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

\[
\begin{array}{cccc}
\%eax & | & \%edx & | \\
\%ecx & | & 0x120 & | \\
\%ebx & | & \%esi & | \\
\%edi & | & \%esp & | \\
%%%% & | & 0x104 & |
\end{array}
\]

\[
\begin{array}{cccc}
123 & | & 0x124 & |\\
456 & | & 0x120 & |\\
 & | & 0x11c & |\\
 & | & 0x118 & |\\
 & | & 0x114 & |\\
 & | & Return addr & |\\
 & | & 0x108 & |\\
 & | & 0x104 & |\\
 & | & 0x100 & |
\end{array}
\]

\[
\begin{align*}
\text{movl } 12(\%ebp),\%ecx & \quad \# \, \text{ecx} = \text{yp} \\
\text{movl } 8(\%ebp),\%edx & \quad \# \, \text{edx} = \text{xp} \\
\text{movl } (\%ecx),\%eax & \quad \# \, \text{eax} = *\text{yp} \, (t1) \\
\text{movl } (\%edx),\%ebx & \quad \# \, \text{ebx} = *\text{xp} \, (t0) \\
\text{movl } \%eax,(\%edx) & \quad \# \, *\text{xp} = \text{eax} \\
\text{movl } \%ebx,(\%ecx) & \quad \# \, *\text{yp} = \text{ebx}
\end{align*}
\]
Understanding Swap

<table>
<thead>
<tr>
<th>%eax</th>
<th>%edx</th>
<th>%ecx</th>
<th>%ebx</th>
<th>%esi</th>
<th>%edi</th>
<th>%esp</th>
<th>%ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x124</td>
<td>0x120</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>%eax</th>
<th>%edx</th>
<th>%ecx</th>
<th>%ebx</th>
<th>%esi</th>
<th>%edi</th>
<th>%esp</th>
<th>%ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>456</td>
<td>0x124</td>
<td>0x120</td>
<td></td>
<td></td>
<td></td>
<td>0x124</td>
<td>0x124</td>
</tr>
</tbody>
</table>

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 |

<table>
<thead>
<tr>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x124</td>
</tr>
<tr>
<td>0x120</td>
</tr>
<tr>
<td>0x11c</td>
</tr>
<tr>
<td>0x118</td>
</tr>
<tr>
<td>0x114</td>
</tr>
<tr>
<td>0x10c</td>
</tr>
<tr>
<td>0x108</td>
</tr>
<tr>
<td>0x104</td>
</tr>
<tr>
<td>0x100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>yp</td>
</tr>
<tr>
<td>xp</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>%ebp</td>
</tr>
</tbody>
</table>

- `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

```assembly
 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

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>456</td>
</tr>
<tr>
<td>%edx</td>
<td>0x124</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x120</td>
</tr>
<tr>
<td>%ebx</td>
<td>123</td>
</tr>
<tr>
<td>%esi</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>yp</td>
<td>12</td>
</tr>
<tr>
<td>xp</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td>%ebp</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>-4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x124</td>
</tr>
<tr>
<td>0x120</td>
</tr>
<tr>
<td>0x11c</td>
</tr>
<tr>
<td>0x118</td>
</tr>
<tr>
<td>0x114</td>
</tr>
<tr>
<td>0x110</td>
</tr>
<tr>
<td>0x124</td>
</tr>
<tr>
<td>0x10c</td>
</tr>
<tr>
<td>0x108</td>
</tr>
<tr>
<td>0x104</td>
</tr>
<tr>
<td>0x100</td>
</tr>
</tbody>
</table>

```
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
```
Complete Memory Addressing Modes

General Form:

\[
D(Rb, Ri, S) \quad \text{Mem}[\text{Reg}[Rb] + S*\text{Reg}[Ri] + D]
\]

D: Literal “displacement” value represented in 1, 2, or 4 bytes

Rb: Base register: Any register

Ri: Index register: Any except %esp; %ebp unlikely

S: Scale: 1, 2, 4, or 8 (why these numbers?)

Special Cases: can use any combination of D, Rb, Ri and S

\[
(Rb, Ri) \quad \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]] \quad (S=1, D=0)
\]

\[
D(Rb, Ri) \quad \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]+D] \quad (S=1)
\]

\[
(Rb, Ri, S) \quad \text{Mem}[\text{Reg}[Rb]+S*\text{Reg}[Ri]] \quad (D=0)
\]
## Address Computation Examples

<table>
<thead>
<tr>
<th>Register contents</th>
<th>Addressing modes</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%edx 0xf000</td>
<td>(Rb,Ri)</td>
<td>Mem[Reg[Rb]+Reg[Ri]]</td>
<td></td>
</tr>
<tr>
<td>%ecx 0x100</td>
<td>D(Ri,S)</td>
<td>Mem[S*Reg[Ri]+D]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(Rb,Ri,S)</td>
<td>Mem[Reg[Rb]+S*Reg[Ri]]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>D(Rb)</td>
<td>Mem[Reg[Rb] +D]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8 (%edx)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(%edx,%ecx)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(%edx,%ecx,4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x80 (,%edx,2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
leal *Src,Dest*  
*load effective address*

*Src* is address mode expression

Set *Dest* to address computed by expression

Example: leal (%edx,%ecx,4), %eax

**DOES NOT ACCESS MEMORY**

**Uses**

Computing addresses, *e.g.*, translation of \( p = \&x[i] \);

Computing arithmetic expressions of the form \( x + k \times i \)

\( k = 1, 2, 4, \text{ or } 8 \)
Arithmetic Operations

Two-operand instructions:

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>addl</td>
<td>Dest = Dest + Src</td>
</tr>
<tr>
<td>subl</td>
<td>Dest = Dest - Src</td>
</tr>
<tr>
<td>imull</td>
<td>Dest = Dest * Src</td>
</tr>
<tr>
<td>shll</td>
<td>Dest = Dest &lt;&lt; Src</td>
</tr>
<tr>
<td>sarl</td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td>shrl</td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td>xorl</td>
<td>Dest = Dest ^ Src</td>
</tr>
<tr>
<td>andl</td>
<td>Dest = Dest &amp; Src</td>
</tr>
<tr>
<td>orl</td>
<td>Dest = Dest</td>
</tr>
</tbody>
</table>

argument order

No distinction between signed and unsigned int (why?)
except arithmetic vs. logical shift right
Arithmetic Operations

One-operand (unary) instructions

incl  Dest  Dest = Dest + 1  increment
decl  Dest  Dest = Dest - 1  decrement
negl  Dest  Dest = -Dest  negate
notl  Dest  Dest = ~Dest  bitwise complement

See CSAPP 3.5.5 for more: mull, cltd, idivl, divl
The function `arith` calculates the sum of `x` and `y`, adds `z` to the result, adds 4 to `x`, multiplies `y` by 48, adds the results, and returns the product of the sum and the total result. The code is written in x86 assembly language for Intel Architecture 32-bit (IA32) processors.
Understanding `arith (IA32)`

```c
int arith(int x, int y, int z){
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

```
movl 8(%ebp),%eax  # eax = x
movl 12(%ebp),%edx  # edx = y
leal (%edx,%eax),%ecx  # ecx = x+y (t1)
leal (%edx,%edx,2),%edx  #
sall $4,%edx  #
addl 16(%ebp),%ecx  #
leal 4(%edx,%eax),%eax  #
imull %ecx,%eax  #
```
int arith(int x, int y, int z) {
    int t1 = x + y;
    int t2 = z + t1;
    int t3 = x + 4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}

movl 8(%ebp),%eax  # eax = x
movl 12(%ebp),%edx  # edx = y
leal (%edx,%eax),%ecx  # ecx = x+y (t1)
leal (%edx,%edx,2),%edx  # edx = y + 2*y = 3*y
sall $4,%edx  # edx = 48*y (t4)
addl 16(%ebp),%ecx  # ecx = z+t1 (t2)
leal 4(%edx,%eax),%eax  # eax = 4+t4+x (t5)
imull %ecx,%eax  # eax = t5*t2 (rval)
Understanding `arith (IA32)`

```c
int arith(int x, int y, int z){
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

```
movl 8(%ebp),%eax         # eax = x
movl 12(%ebp),%edx        # edx = y
leal (%edx,%eax),%ecx     # ecx = x+y  (t1)
leal (%edx,%edx,2),%edx    # edx = y + 2*y = 3*y
sall $4,%edx              # edx = 48*y  (t4)
addl 16(%ebp),%ecx        # ecx = z+t1  (t2)
leal 4(%edx,%eax),%eax    # eax = 4+t4+x  (t5)
imull %ecx,%eax            # eax = t5*t2  (rval)
```
Understanding `arith` (IA32)

```c
int arith(int x, int y, int z){
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

```assembly
movl 8(%ebp),%eax          # eax = x
movl 12(%ebp),%edx         # edx = y
leal (%edx,%eax),%ecx      # ecx = x+y (t1)
leal (%edx,%edx,2),%edx     # edx = y + 2*y = 3*y
sall $4,%edx               # edx = 48*y (t4)
addl 16(%ebp),%ecx         # ecx = z+t1 (t2)
leal 4(%edx,%eax),%eax      # eax = 4+t4+x (t5)
imull %ecx,%eax            # eax = t5*t2 (rval)
```
Observations about arith

```c
int arith(int x, int y, int z){
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

- Instructions in different order from C code
- Some expressions require multiple instructions
- Some instructions cover multiple expressions
- Same x86 code by compiling:
  
  \[(x+y+z) \times (x+4+48\times y)\]

```asm
movl 8(%ebp),%eax  # eax = x
movl 12(%ebp),%edx # edx = y
leal (%edx,%eax),%ecx # ecx = x+y (t1)
leal (%edx,%edx,2),%edx # edx = y + 2\times y = 3\times y
sall $4,%edx # edx = 48\times y (t4)
addl 16(%ebp),%ecx # ecx = z+t1 (t2)
leal 4(%edx,%eax),%eax # eax = 4+t4+x (t5)
imull %ecx,%eax # eax = t5\times t2 (rval)
```
Another Example (IA32)

```c
int logical(int x, int y){
    int t1 = x^y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

```
logical:
    pushl %ebp
    movl %esp,%ebp

    movl 8(%ebp),%eax
    xorl 12(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax

    movl %ebp,%esp
    popl %ebp
    ret
```

Offset Stack

<table>
<thead>
<tr>
<th>Offset</th>
<th></th>
<th>Offset</th>
<th></th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>y</td>
<td>8</td>
<td>x</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>0</td>
<td>Old %ebp</td>
<td>0</td>
</tr>
</tbody>
</table>
Another Example (IA32)

```c
int logical(int x, int y){
    int t1 = x^y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

```assembly
logical:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    xorl 12(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax
    movl %ebp,%esp
    popl %ebp
    ret
```
Another Example (IA32)

```c
int logical(int x, int y){
    int t1 = x^y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

```assembly
logical:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    xorl 12(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

- Set Up
- Body
- Finish

```assembly
movl 8(%ebp),%eax        ; eax = x
xorl 12(%ebp),%eax       ; eax = x^y (t1)
sar $17,%eax             ; eax = t1>>17 (t2)
andl $8185,%eax          ; eax = t2 & 8185
```
Another Example (IA32)

```c
int logical(int x, int y) {
    int t1 = x^y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

**logical:**

```
pushl %ebp
movl %esp,%ebp
movl 8(%ebp),%eax
xorl 12(%ebp),%eax
sarl $17,%eax
andl $8185,%eax
movl %ebp,%esp
popl %ebp
ret
```

Set Up

Body

Finish

$2^{13} = 8192,$
$2^{13} - 7 = 8185$

...0010000000000000, ...0001111111111001

**compiler optimization**

```
movl 8(%ebp),%eax
xorl 12(%ebp),%eax
sarl $17,%eax
andl $8185,%eax
```

$eax = x$

$eax = x^y \quad (t1)$

$eax = t1 >> 17 \quad (t2)$

$eax = t2 \ & \ 8185$