Chapter 3:

Basic x86 Assembly Language Programming

Topics

- Move operations to/from memory
- Addressing modes
- Arithmetic operations

Announcements

- Data Lab is due Friday Feb 3 by 11:55 pm
  - Grading interview time slots released probably Friday – sign up for 12-minute slots that are spread over next week M-F
- Bomb Lab #2 released, due Friday Feb 24
- Next Assembly Quiz will be released ~Friday, due ~Mon Feb 13
- Read Chapter 3.1-3.12 (except 3.11) and do practice problems

Example of Simple Addressing Modes

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

```
swap:
   movq    (%rdi), %rax
   movq    (%rsi), %rdx
   movq    %rdx, (%rdi)
   movq    %rax, (%rsi)
   ret
```

C Pointers – A Quick Recap

- **int count=1;**
  - Declare an integer named count
  - This allocates 4 bytes in memory for the variable count
  - Initialize count to the value 1
- **char *p1;**
  - Declare p1 as a *pointer* to a char, i.e. the value of p1 is interpreted as a memory address (4 bytes wide on 32-bit systems)
  - The pointer is allocated space in memory (4 bytes, not 1)
- **int *p2 = &count;**
  - Declare p2 as a pointer to an integer
  - Allocates 4 bytes in memory for the pointer (32-bit)
  - Initializes its value to the memory address of the count variable

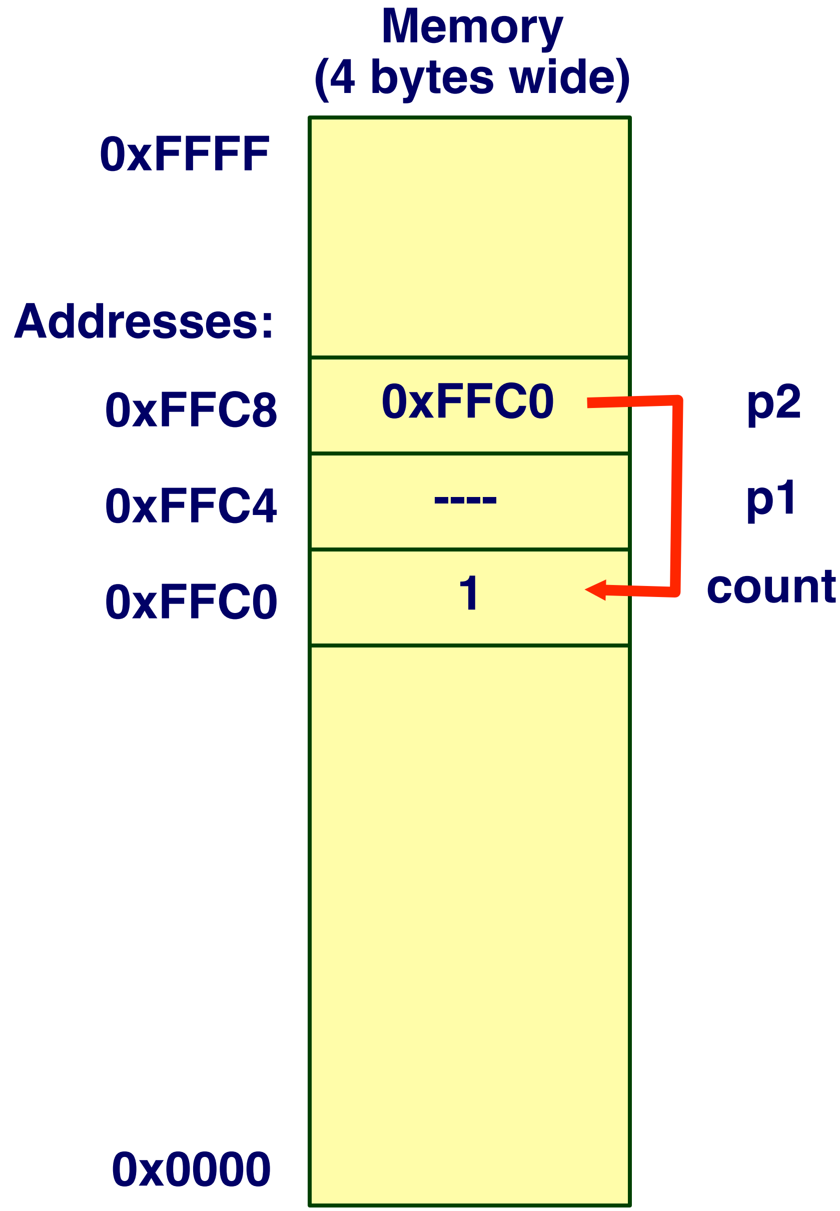
C Pointers (2)

```
int count=1;

char *p1;

int *p2 = &count;
```

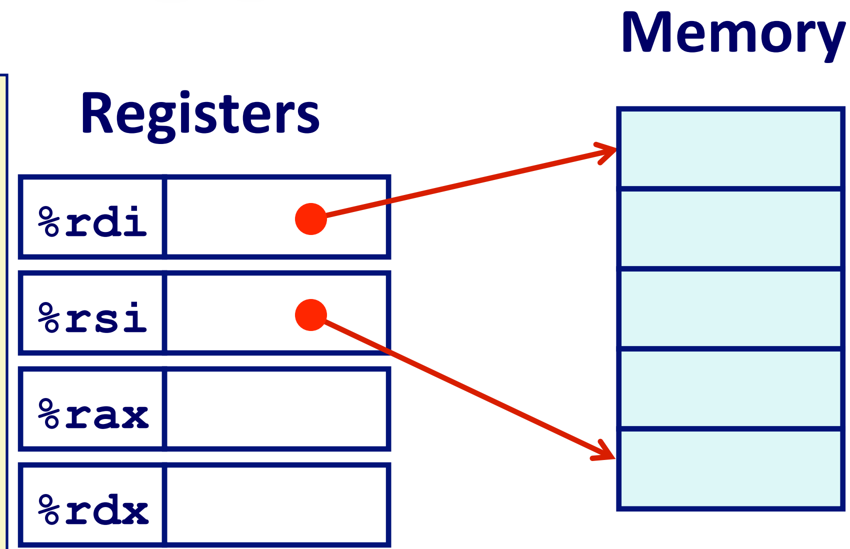
- Assume the variables are laid out in memory as shown
- We see p2 storing the memory address of count, i.e. p2 is *pointing at* count
- p1 is uninitialized and not yet pointing at any character



For brevity, the two most significant bytes of address are not shown

Understanding Swap()

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



Register	Value
%rdi	xp
%rsi	yp
%rax	t0
%rdx	t1

```
swap:
    movq    (%rdi), %rax   # t0 = *xp
    movq    (%rsi), %rdx   # t1 = *yp
    movq    %rdx, (%rdi)   # *xp = t1
    movq    %rax, (%rsi)   # *yp = t0
    ret
```

Understanding `Swap()`

Registers

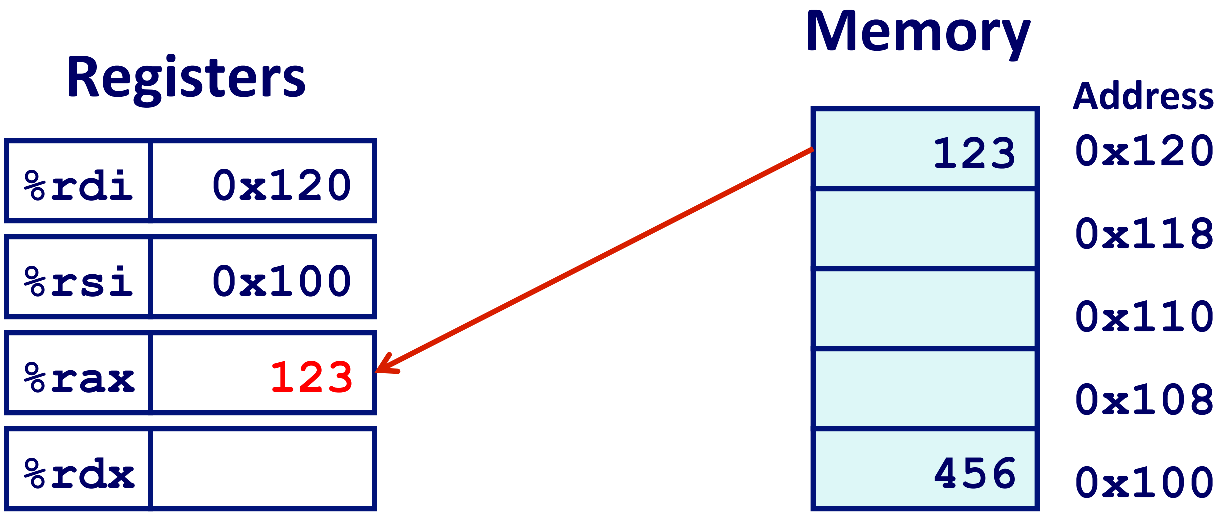
%rdi	0x120
%rsi	0x100
%rax	
%rdx	

Memory

	Address
123	0x120
	0x118
	0x110
	0x108
456	0x100

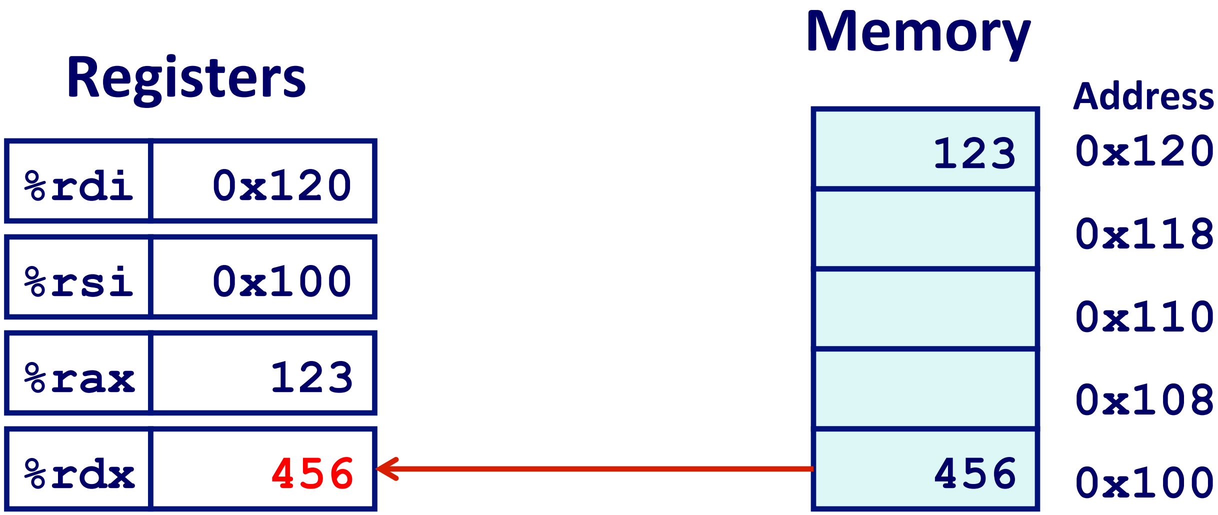
```
swap:
    movq    (%rdi), %rax   # t0 = *xp
    movq    (%rsi), %rdx   # t1 = *yp
    movq    %rdx, (%rdi)   # *xp = t1
    movq    %rax, (%rsi)   # *yp = t0
    ret
```

Understanding Swap()



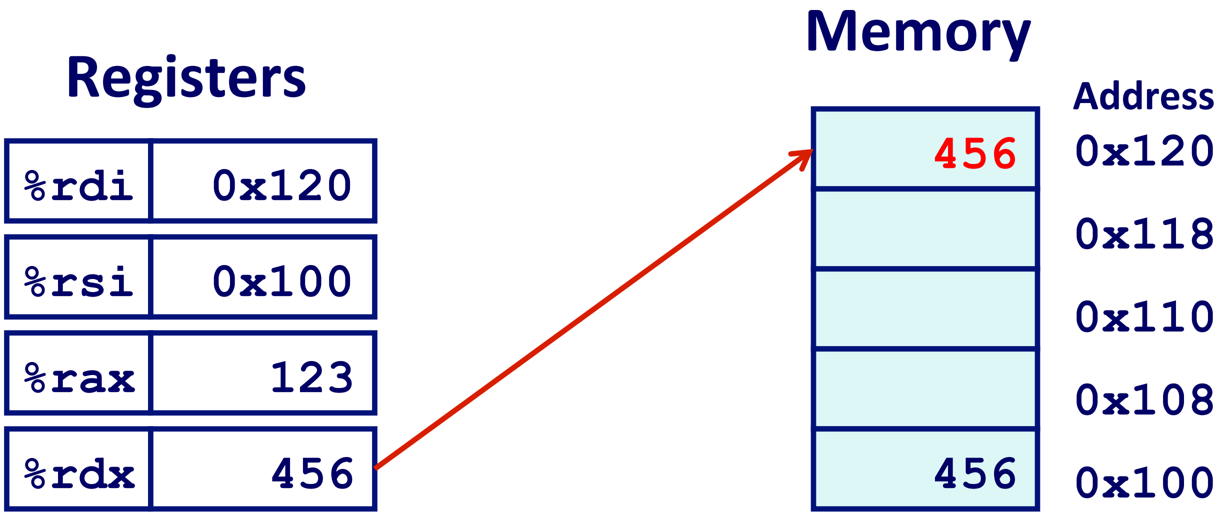

```
swap:
    movq     (%rdi), %rax   # t0 = *xp
    movq     (%rsi), %rdx   # t1 = *yp
    movq     %rdx, (%rdi)   # *xp = t1
    movq     %rax, (%rsi)   # *yp = t0
    ret
```

Understanding Swap()



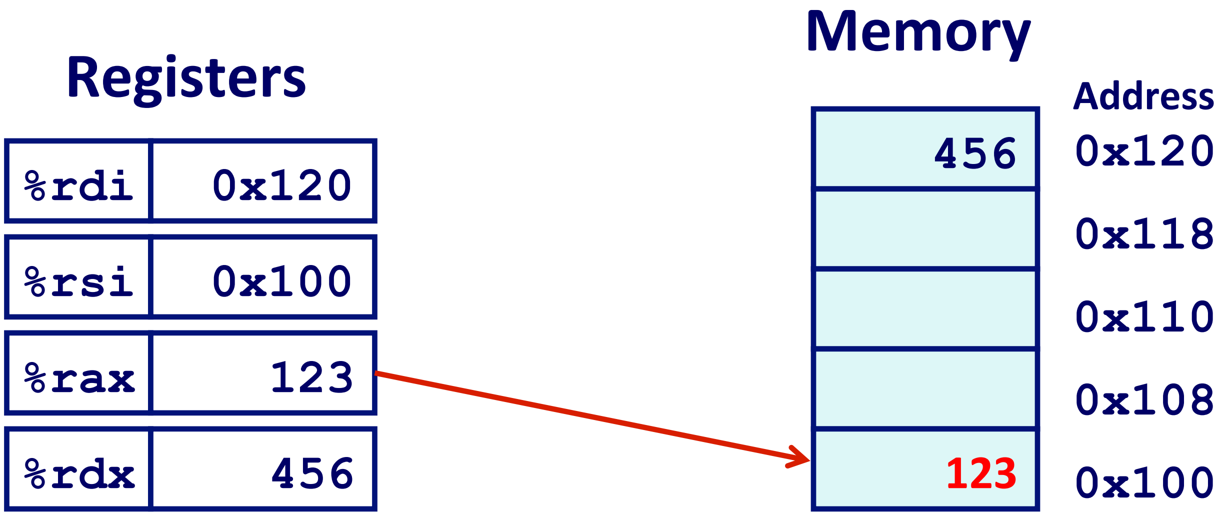

```
swap:
    movq     (%rdi), %rax    # t0 = *xp
    movq     (%rsi), %rdx    # t1 = *yp
    movq     %rdx, (%rdi)    # *xp = t1
    movq     %rax, (%rsi)    # *yp = t0
    ret
```

Understanding `Swap()`




```
swap:
    movq    (%rdi), %rax   # t0 = *xp
    movq    (%rsi), %rdx   # t1 = *yp
    movq    %rdx, (%rdi)   # *xp = t1
    movq    %rax, (%rsi)   # *yp = t0
    ret
```

Understanding Swap()




```
swap:
    movq     (%rdi), %rax    # t0 = *xp
    movq     (%rsi), %rdx    # t1 = *yp
    movq     %rdx, (%rdi)    # *xp = t1
    movq     %rax, (%rsi)    # *yp = t0
    ret
```

Moving different word sizes

- `movq %rax, %rdx`
  - Move a "quad" word (4*16 = 64 bits = 8 bytes) from register %rax to register %rdx
- `movl %eax, %edx`
  - Move a "long" word (2*16 = 32 bits = 4 bytes) from register %eax to register %edx
- `movw %ax, %dx`
  - Move a word (16 bits = 2 bytes) from register %ax to register %dx
- `movb %al, %dl`
  - Move a byte from register %al to register %dl

Indexed Addressing Modes

```
movq 24(%rdi,%rsi,4), %rax
```

- **This means:**
  - Move a quad word (8 bytes) from the memory location %rdi + 4*%rsi + 24 to register %rax
- **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 16 integer registers
  - Ri: Index register: Any, except for `%rsp`
  - S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

Indexed Addressing Modes (2)

- **Special Cases**
  - **(Rb,Ri)** **Mem[Reg[Rb]+Reg[Ri]]**
    - `movq (%rax,%rbx), %rdx`
  - **D(Rb,Ri)** **Mem[Reg[Rb]+Reg[Ri]+D]**
    - `movq %rdx, 12(%rax,%rbx)`
  - **(Rb,Ri,S)** **Mem[Reg[Rb]+S*Reg[Ri]]**
    - `movq (%rax,%rbx,8), %rdx`
  - **(Rb)** **Mem[Reg[Rb]+S*Reg[Ri]]**
    - `movq %rdx, (%rax)`

Address Computation Examples

%rdx	0xf000
%rcx	0x0100

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

C operators – Assembly Equivalents?

Operators

```
()   []   ->   .
!   ~   ++   --   +   -   *   & (type) sizeof
*   /   %
+   -
<<   >>
<   <=   >   >=
==   !=
&
^
|
&&
||
?:
= += -= *= /= %= &= ^= != <<= >>=
,
```

Many of these C operators have direct x86 assembly equivalents

Some Arithmetic Operations

- **Two Operand Instructions:**

Format		Computation	
`addq`	Src,Dest	Dest = Dest + Src	
`subq`	Src,Dest	Dest = Dest - Src	
`imulq`	Src,Dest	Dest = Dest * Src	
`salq`	Src,Dest	Dest = Dest << Src	Also called shlq
`sarq`	Src,Dest	Dest = Dest >> Src	Arithmetic
`shrq`	Src,Dest	Dest = Dest >> Src	Logical
`xorq`	Src,Dest	Dest = Dest ^ Src	
`andq`	Src,Dest	Dest = Dest & Src	
`orq`	Src,Dest	Dest = Dest | Src	

- **Watch out for argument order!**
- **No distinction between signed and unsigned int (why?)**

Some Arithmetic Operations

- **One Operand Instructions**

`incq`	Dest	Dest = Dest + 1
`decq`	Dest	Dest = Dest - 1
`negq`	Dest	Dest = - Dest
`notq`	Dest	Dest = ~Dest

- **See book for more instructions**

`lea` Instruction for Address Computation

`lea` = "Load effective address"

`leaq` ***Src*,*Dest***

- ***Src*** **is indexed address mode expression**
- **Set *Dest* (must be register) to value denoted by expressio**

- **Example:**

```
leaq 10(%rdx, %rdx, 4), %rax
```

%rdx + 4*%rdx + 10
= 5*%rdx + 10

Therefore"%rax = 5 * %rdx + 10"

- **Compare to:**

```
movq 10(%rdx, %rdx, 4), %rax
```

means "%rax = Mem[5*%rdx + 10]

`lea` Instruction for Address Computation

Uses

- Computing arithmetic expressions of the form x + k*y
  - k = 1, 2, 4, or 8.
- Computing address without doing memory reference
  - E.g., translation of `p = &x[i];`

Example

```
long m12(long x)
{
  return x*12;
}
```

Converted to ASM by compiler:

```
leaq (%rdi,%rdi,2), %rax # t <- x+x*2
salq $2, %rax            # return t<<2
```

Arithmetic Expression Example

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

```
arith:
  leaq    (%rdi,%rsi), %rax
  addq    %rdx, %rax
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx
  leaq    4(%rdi,%rdx), %rcx
  imulq   %rcx, %rax
  ret
```

Interesting Instructions

- `leaq`: address computation
- `salq`: shift
- `imulq`: multiplication
  - But, only used once

Understanding Arithmetic Expression Example

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

```
arith:
  leaq    (%rdi,%rsi), %rax     # t1
  addq    %rdx, %rax            # t2
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx              # t4
  leaq    4(%rdi,%rdx), %rcx    # t5
  imulq   %rcx, %rax            # rval
  ret
```

Register	Use(s)
`%rdi`	Argument **x**
`%rsi`	Argument **y**
`%rdx`	Argument **z**
`%rax`	`t1`, `t2`, `rval`
`%rdx`	`t4`
`%rcx`	`t5`

Supplementary Slides

Another Example

```
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
```

$\left.\right\}$ Set Up

$\left.\right\}$ Body

$\left.\right\}$ Finish

```
movl 8(%ebp),%eax     # eax = x
xorl 12(%ebp),%eax    # eax = x^y
sarl $17,%eax         # eax = t1>>17
andl $8185,%eax       # eax = t2 & 8185
```

Another Example

```
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
```

$\left.\begin{matrix}\texttt{pushl \%ebp}\\ \texttt{movl \%esp,\%ebp}\end{matrix}\right\}$ Set Up

$\left.\begin{matrix}\texttt{movl 8(\%ebp),\%eax}\\ \texttt{xorl 12(\%ebp),\%eax}\\ \texttt{sarl \$17,\%eax}\\ \texttt{andl \$8185,\%eax}\end{matrix}\right\}$ Body

$\left.\begin{matrix}\texttt{movl \%ebp,\%esp}\\ \texttt{popl \%ebp}\\ \texttt{ret}\end{matrix}\right\}$ Finish

```
movl 8(%ebp),%eax     # eax = x
xorl 12(%ebp),%eax    # eax = x^y   (t1)
sarl $17,%eax         # eax = t1>>17    (t2)
andl $8185,%eax       # eax = t2 & 8185
```

Another Example

```
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

```
movl 8(%ebp),%eax       # eax = x
xorl 12(%ebp),%eax      # eax = x^y   (t1)
sarl $17,%eax           # eax = t1>>17    (t2)
andl $8185,%eax         # eax = t2 & 8185
```

Another Example

```
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              } Set Up
    movl %esp,%ebp

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

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

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

```
movl 8(%ebp),%eax     # eax = x
xorl 12(%ebp),%eax    # eax = x^y    (t1)
sarl $17,%eax         # eax = t1>>17   (t2)
andl $8185,%eax       # eax = t2 & 8185
```

Note how compiler combines 2 source code lines into 1