MOV instruction

Most common instruction is data transfer instruction

- Mov SRC, DEST: Move source into destination
- SRC and DEST are operands
- DEST is a register or a location
- SRC can be the contents of register, memory location, constant, or a label.
- If you use gcc, you will see movl <src>, <dest>
- All the instructions in x86 are 32-bit

Used to copy data:

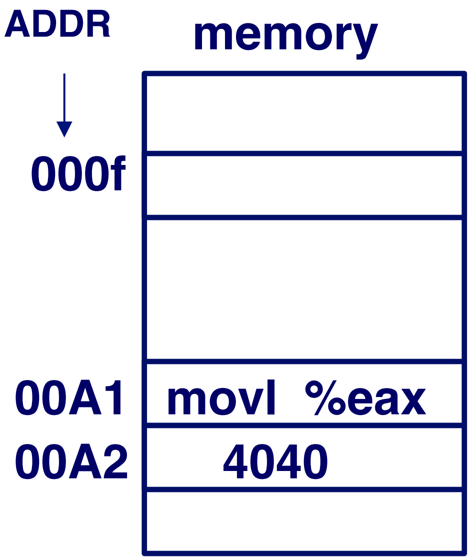
- Constant to register (immediate)
- Memory to register
- Register to memory
- Register to register

Cannot copy memory to memory in a single instruction

Immediate Addressing

Operand is immediate

- Operand value is found immediately following the instruction
- Encoded in 1, 2, or 4 bytes
- $ in front of immediate operand
- E.g., movl $0x4040, %eax



Register Mode Addressing

Use % to denote register

- E.g., %eax

Source operand: use value in specified register

Destination operand: use register as destination for value

Examples:

- movl %eax, %ebx
  - Copy content of %eax to %ebx
- movl $0x4040, %eax → immediate addressing
  - Copy 0x4040 to %eax
- movl %eax, 0x0000f → Absolute addressing
  - Copy content of %eax to memory location 0x0000f

Indirect Mode Addressing

Content of operand is an address

- Designated as parenthesis around operand

Offset can be specified as immediate mode

Examples:

- movl (%ebp), %eax
  - Copy value from memory location whose address is in ebp into eax
- movl -4(%ebp), %eax
  - Copy value from memory location whose address is -4 away from content of ebp into eax

Indexed Mode Addressing

Add content of two registers to get address of operand

- movl (%ebp, %esi), %eax
  - Copy value at (address = ebp + esi) into eax
- movl 8(%ebp, %esi),%eax
  - Copy value at (address = 8 + ebp + esi) into eax

Useful for dealing with arrays

- If you need to walk through the elements of an array
- Use one register to hold base address, one to hold index
  - E.g., implement C array access in a for loop
- Index cannot be ESP

Scaled Indexed Mode Addressing

Multiply the second operand by the scale (1, 2, 4 or 8)

- movl 0x80 (%ebx, %esi, 4), %eax
  - Copy value at (address = ebx + esi*4 + 0x80) into eax

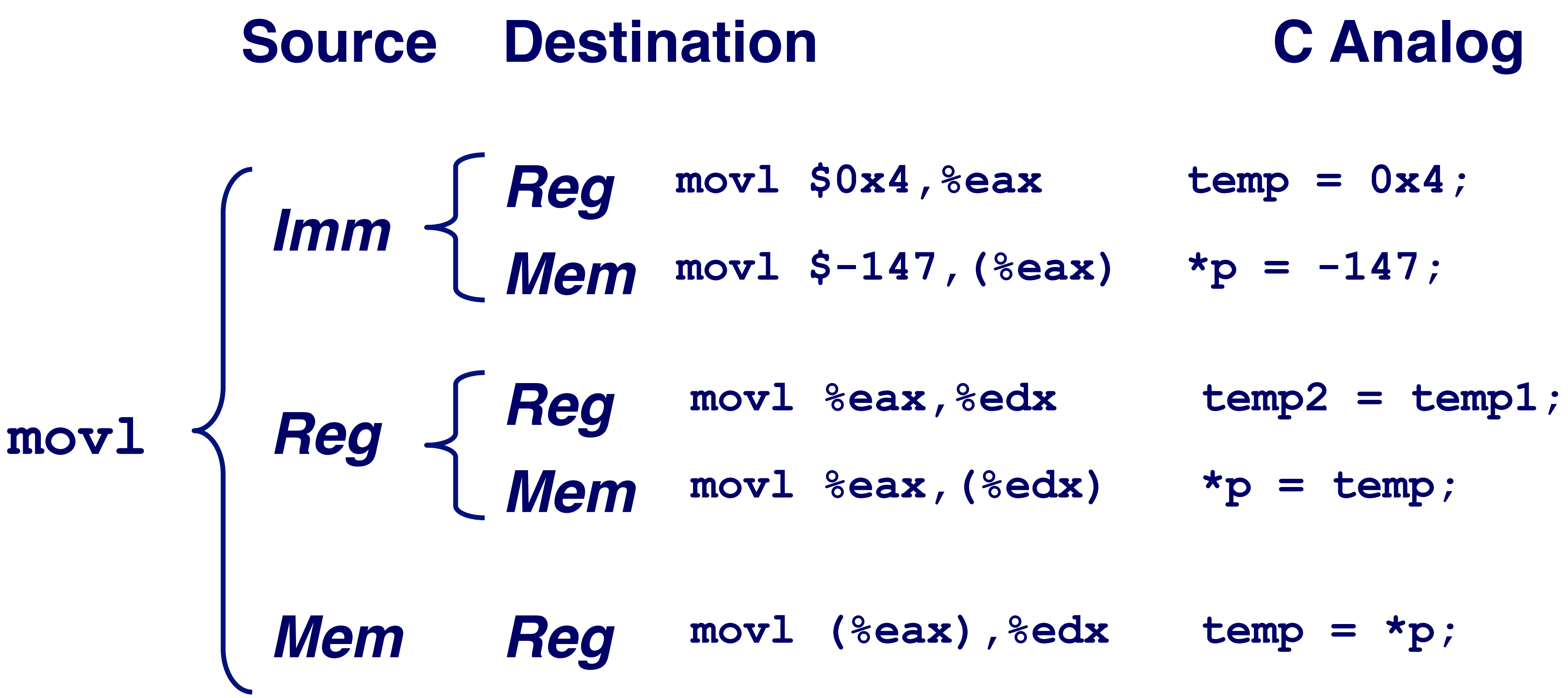
Where is it useful?

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`

`movl` Operand Combinations



	Source	Destination		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 transfers with single instruction

Stack Operations

By convention, %esp is used to maintain a stack in memory

- Used to support C function calls

%esp contains the address of top of stack

Instructions to push (pop) content onto (off of) the stack

- pushl %eax
  - esp = esp – 4
  - Memory[esp] = eax
- popl %ebx
  - ebx = Memory[esp]
  - esp = esp + 4

Where does the stack start? We'll discuss later

Using Simple Addressing Modes

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

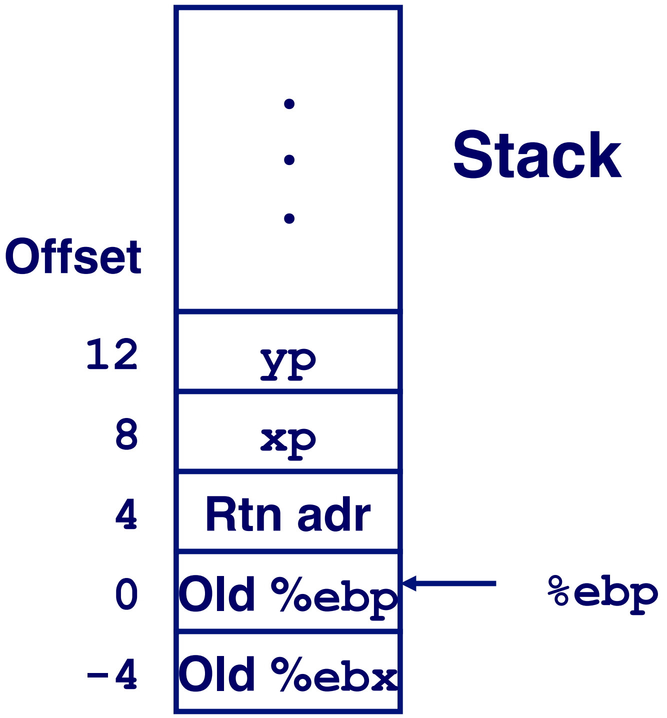
```
swap:
    pushl %ebp
    movl %esp,%ebp              Set
    pushl %ebx                  Up

    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
```

Understanding Swap

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



Stack

Offset	
	• • •
12	yp
8	xp
4	Rtn adr
0	Old %ebp ← %ebp
-4	Old %ebx

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

%eax	
%edx	
%ecx	
%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

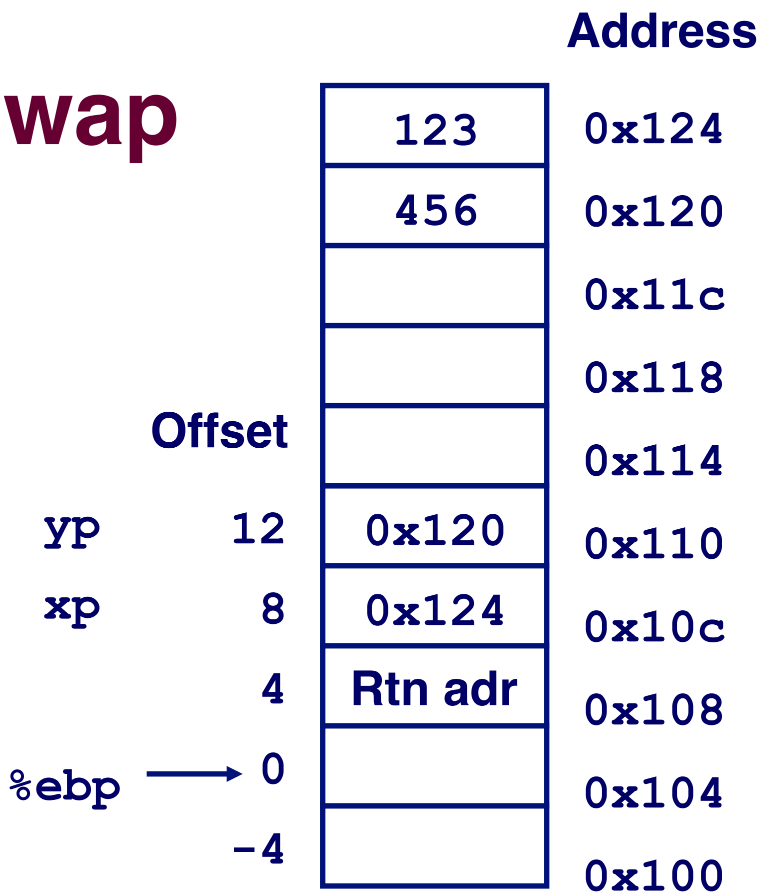
%eax	
%edx	
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

	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

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

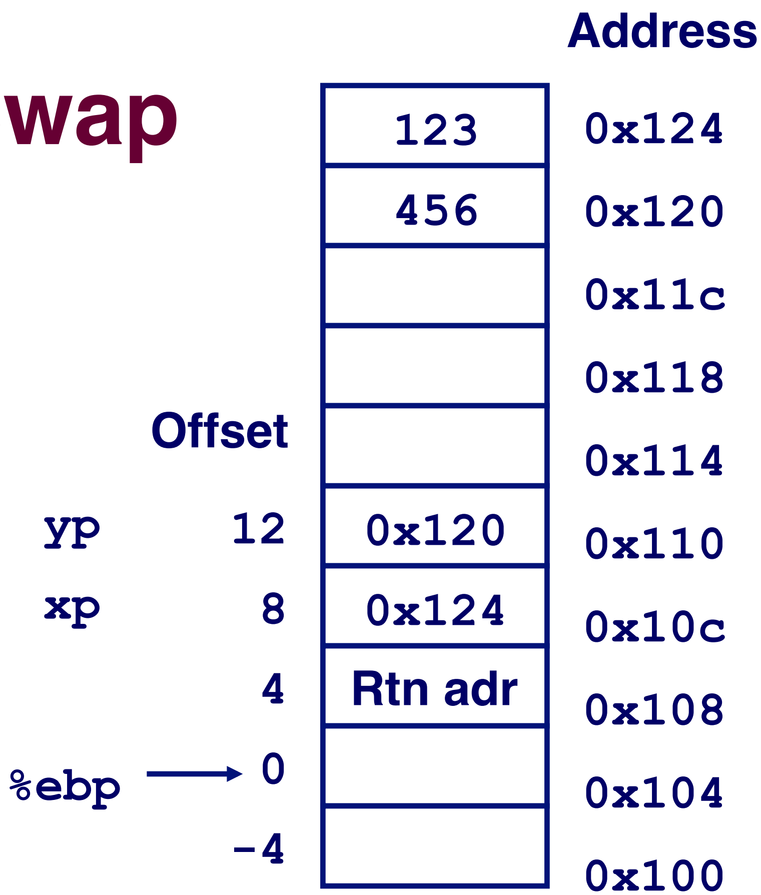


	Offset		Address
		123	0x124
		456	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp →	0		0x104
	-4		0x100

```
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

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

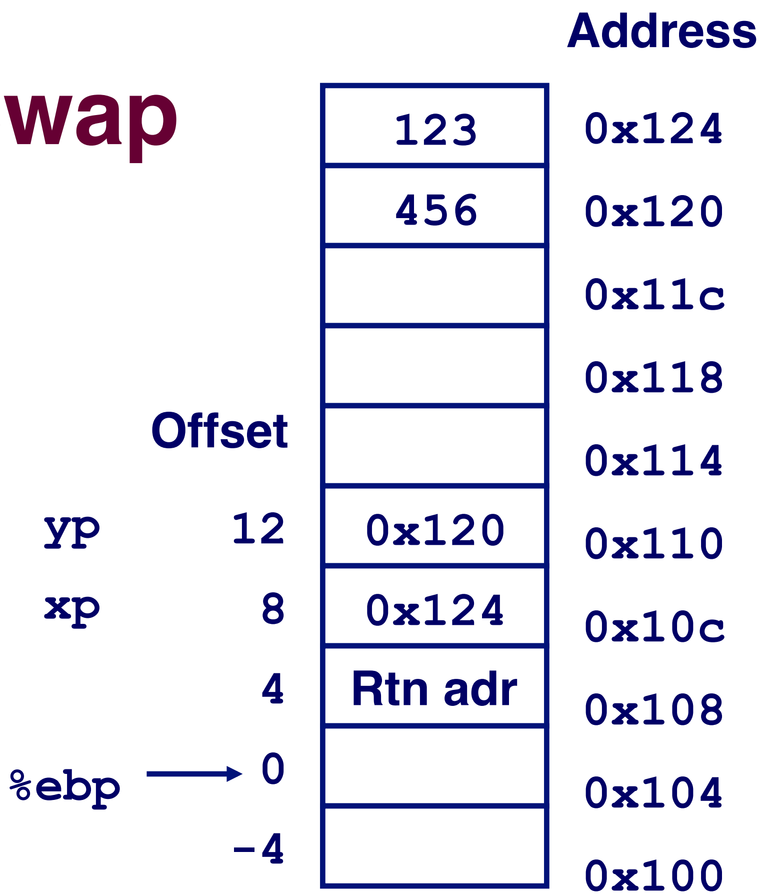


	Offset		Address
		123	0x124
		456	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp →	0		0x104
	-4		0x100

```
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

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%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

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

	Offset		Address
		456	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

%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
```

Swap in x86-64: 64-bit Registers

rax	eax
rcx	ecx
rdx	edx
rbx	ebx
rsp	esp
rbp	ebp
rsi	esi
rdi	edi

r8
r9
r10
r11
r12
r13
r14
r15

Swap in x86-64 bit

```
swap:
    movl (%rdi), %edx
    movl (%rsi), %eax
   movl  %eax, (%rdi)
   movl  %edx, (%rsi)
   retq
```

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

Arguments passed in registers

- First, xp in rdi and yp in rsi
- 64-bit pointers, data values are 32-bit ints, so uses eax/edx

No stack operations

What happens with long int?

Address Computation Instruction

leal: compute address using addressing mode without accessing memory

leal src, dest

- src is address mode expression
- Set dest to address specified by src

Use

- Computing address without doing memory reference
  - E.g., translation of p = &x[i];

Example:

- leal 7(%edx, %edx, 4), %eax
  - eax = 4*edx + edx + 7 = 5*edx + 7

Some Arithmetic Operations

Instruction	Computation
`addl` *Src,Dest*	*Dest* = *Dest* + *Src*
`subl` *Src,Dest*	*Dest* = *Dest* – *Src*
`imull` *Src,Dest*	*Dest* = *Dest* * *Src*
`sall` *Src,Dest*	*Dest* = *Dest* << *Src (left shift)*
`sarl` *Src,Dest*	*Dest* = *Dest* >> *Src (right shift)*
`xorl` *Src,Dest*	*Dest* = *Dest* ^ *Src*
`andl` *Src,Dest*	*Dest* = *Dest* & *Src*
`orl` *Src,Dest*	*Dest* = *Dest* | *Src*

Some Arithmetic Operations

Instruction	Computation
`incl` *Dest*	*Dest* = *Dest* + 1
`decl` *Dest*	*Dest* = *Dest* - 1
`negl` *Dest*	*Dest* = - *Dest*
`notl` *Dest*	*Dest* = ~ *Dest*

Using `leal` for Arithmetic Expressions

```
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;
}
```

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

Set Up

```
   movl 8(%ebp),%eax
   movl 12(%ebp),%edx
   leal (%edx,%eax),%ecx
   leal (%edx,%edx,2),%edx
   sall $4,%edx
   addl 16(%ebp),%ecx
   leal 4(%edx,%eax),%eax
   imull %ecx,%eax
```

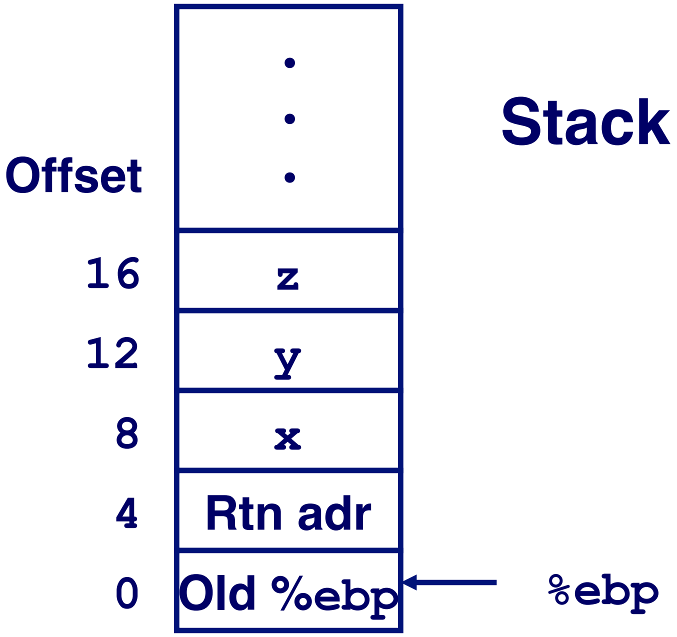
Body

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

Finish

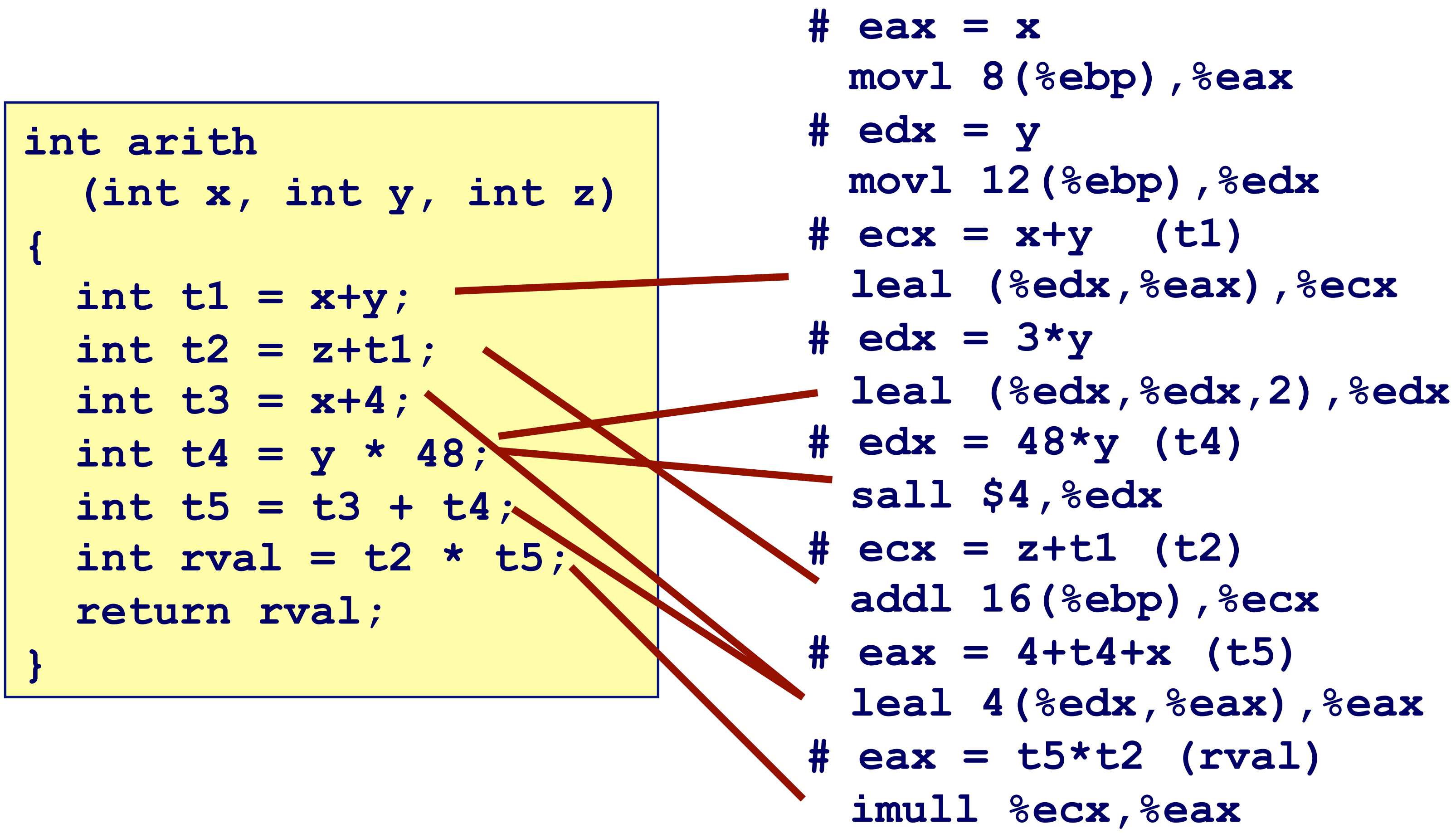
Understanding `arith`

```
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 = 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`



```
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;
}
```

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

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;
}
```

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

```
logical:
    pushl %ebp                 } Set
    movl %esp,%ebp             } Up

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

```
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
```

Mystery Function

What does the following piece of code do?

A. Add two variables

B. Subtract two variables

C. Swap two variables

D. No idea

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

iClicker Quiz 1

```
.globl foo
        .type   foo, @function
foo:
        pushl   %ebp

        movl    %esp, %ebp

        movl    16(%ebp), %eax

        imull   12(%ebp), %eax

        addl    8(%ebp), %eax

        popl    %ebp

        ret
```

A: A function that takes two arguments

B: A function that takes three arguments

C: A function that takes four arguments

D: A function that takes no arguments

What does this function do?

```
.globl foo
        .type   foo, @function
foo:
        pushl   %ebp

        movl    %esp, %ebp

        movl    16(%ebp), %eax

        imull   12(%ebp), %eax

        addl    8(%ebp), %eax

        popl    %ebp

        ret
```

Control Flow/Conditionals

How do we represent conditionals in assembly?

A conditional branch can implement all control flow constructs in higher level language

- Examples: if/then, while, for

A unconditional branch for constructs like break/ continue