# The Hardware/Software Interface

CSE351 Winter 2013

## x86 Programming I

# Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

**Assembly language:**

```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

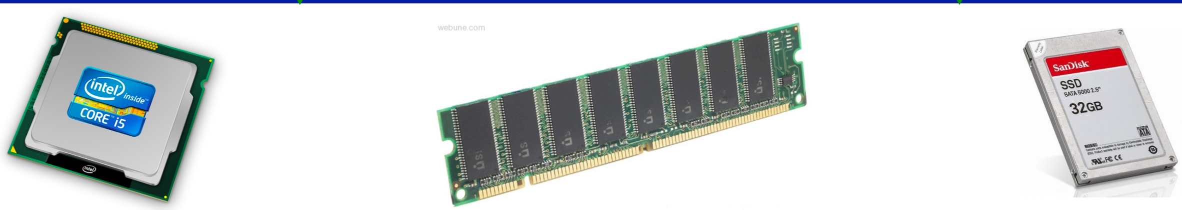
**Machine code:**

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

**OS:**

Windows 8 Mac

**Computer system:**



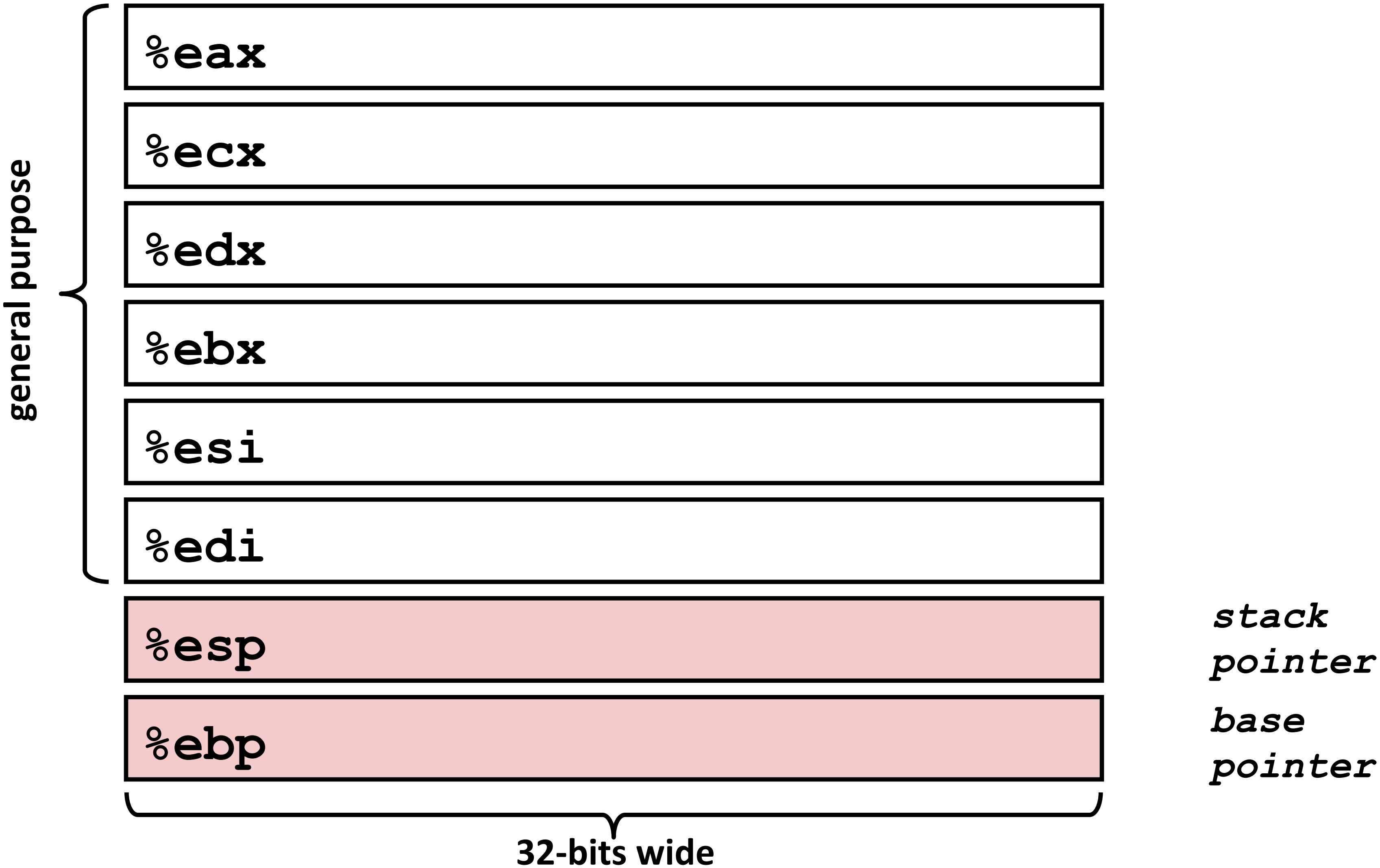
# Today

- Move instructions, registers, and operands
- Memory addressing modes
- `swap` example: 32-bit vs. 64-bit
- Arithmetic operations

# Why learn assembly language?

- **Not to be able to write programs directly in assembly**
  - Compilers do that for you
- **But to be able to *understand* the code generated by compilers, so that you can then:**
  - Optimize performance of critical sections of code
  - Investigate unexpected or even buggy behavior
  - Understand how security vulnerabilities arise, and how to protect against them

# Integer Registers (IA32)



# Three Basic Kinds of Instructions

- **Transfer data between memory and register**
  - *Load* data from memory into register
    - %reg = Mem[address]
  - *Store* register data into memory
    - Mem[address] = %reg

Remember:
memory is indexed
just like an array[]!

- **Perform arithmetic function on register or memory data**
  - c = a + b;
- **Transfer control**
  - Unconditional jumps to/from procedures
  - Conditional branches

# Moving Data: IA32

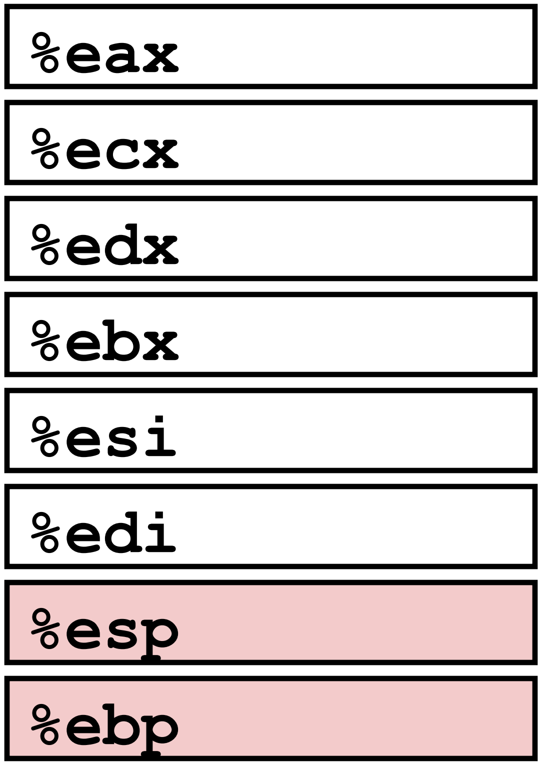
- **Moving Data**
  - **`movx`** ***Source*, *Dest***
  - **`x`** is one of {**`b, w, l`**}

  - **`movl`** ***Source*, *Dest***:
    Move 4-byte "long word"
  - **`movw`** ***Source*, *Dest***:
    Move 2-byte "word"
  - **`movb`** ***Source*, *Dest***:
    Move 1-byte "byte"

- **Lots of these in typical code**

| |
|---|
| **`%eax`** |
| **`%ecx`** |
| **`%edx`** |
| **`%ebx`** |
| **`%esi`** |
| **`%edi`** |
| **`%esp`** |
| **`%ebp`** |

# Moving Data: IA32




- **Moving Data**

  `movl` ***Source*, *Dest***:

- **Operand Types**
  - ***Immediate:*** Constant integer data
    - Example: **`$0x400, $-533`**
    - Like C constant, but prefixed with **`'$'`**
    - Encoded with 1, 2, or 4 bytes
  - ***Register:*** One of 8 integer registers
    - Example: **`%eax, %edx`**
    - But **`%esp`** and **`%ebp`** reserved for special use
    - Others have special uses for particular instructions
  - ***Memory:*** 4 consecutive bytes of memory at address given by register
    - Simplest example: **`(%eax)`**
    - Various other "address modes"

# movl Operand Combinations

| | Source | Dest | Src,Dest | C Analog |
|---|---|---|---|---|
| movl | *Imm* | *Reg* | `movl $0x4,%eax` | `var_a = 0x4;` |
| | | *Mem* | `movl $-147,(%eax)` | `*p_a = -147;` |
| | *Reg* | *Reg* | `movl %eax,%edx` | `var_d = var_a;` |
| | | *Mem* | `movl %eax,(%edx)` | `*p_d = var_a;` |
| | *Mem* | *Reg* | `movl (%eax),%edx` | `var_d = *p_a;` |

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

# Memory vs. registers

- **Why both?**

# Memory Addressing Modes: Basic

- **Indirect (R) Mem[Reg[R]]**
  - Register R specifies the memory address

```
movl (%ecx),%eax
```

- **Displacement D(R) Mem[Reg[R]+D]**
  - Register R specifies a memory address
    - (e.g. the start of some memory region)
  - Constant displacement D specifies the offset from that address

```
movl 8(%ebp),%edx
```

# Using Basic Addressing Modes

```
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
```

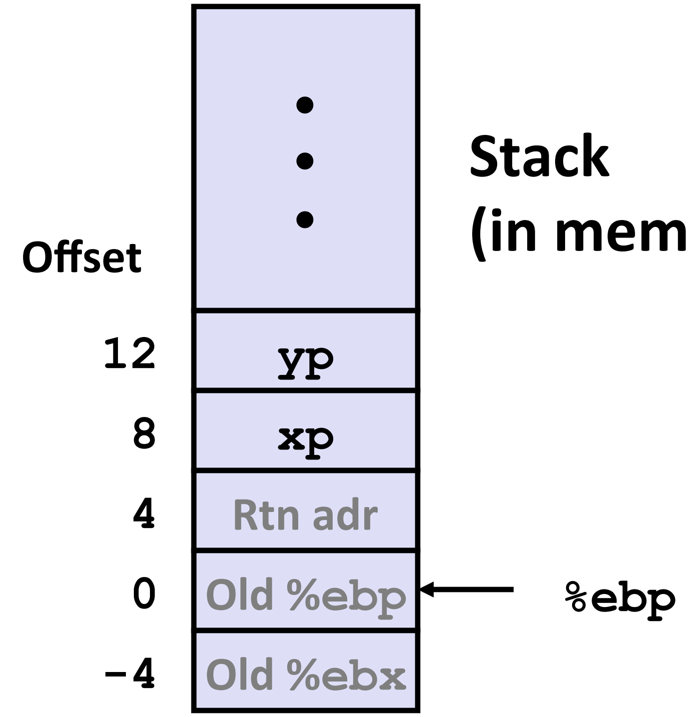
Set Up

Body

Finish

# Understanding Swap

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




Stack
(in memory)

| Register | Value |
|---|---|
| %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

| Register | Value |
|---|---|
| %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

| Register | Value |
| --- | --- |
| %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

| 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

| | |
|---|---|
| %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

| Register | Value |
|---|---|
| %eax | 456 |
| %edx | 0x124 |
| %ecx | 0x120 |
| %ebx | 123 |
| %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

| | |
|---|---|
| %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
```

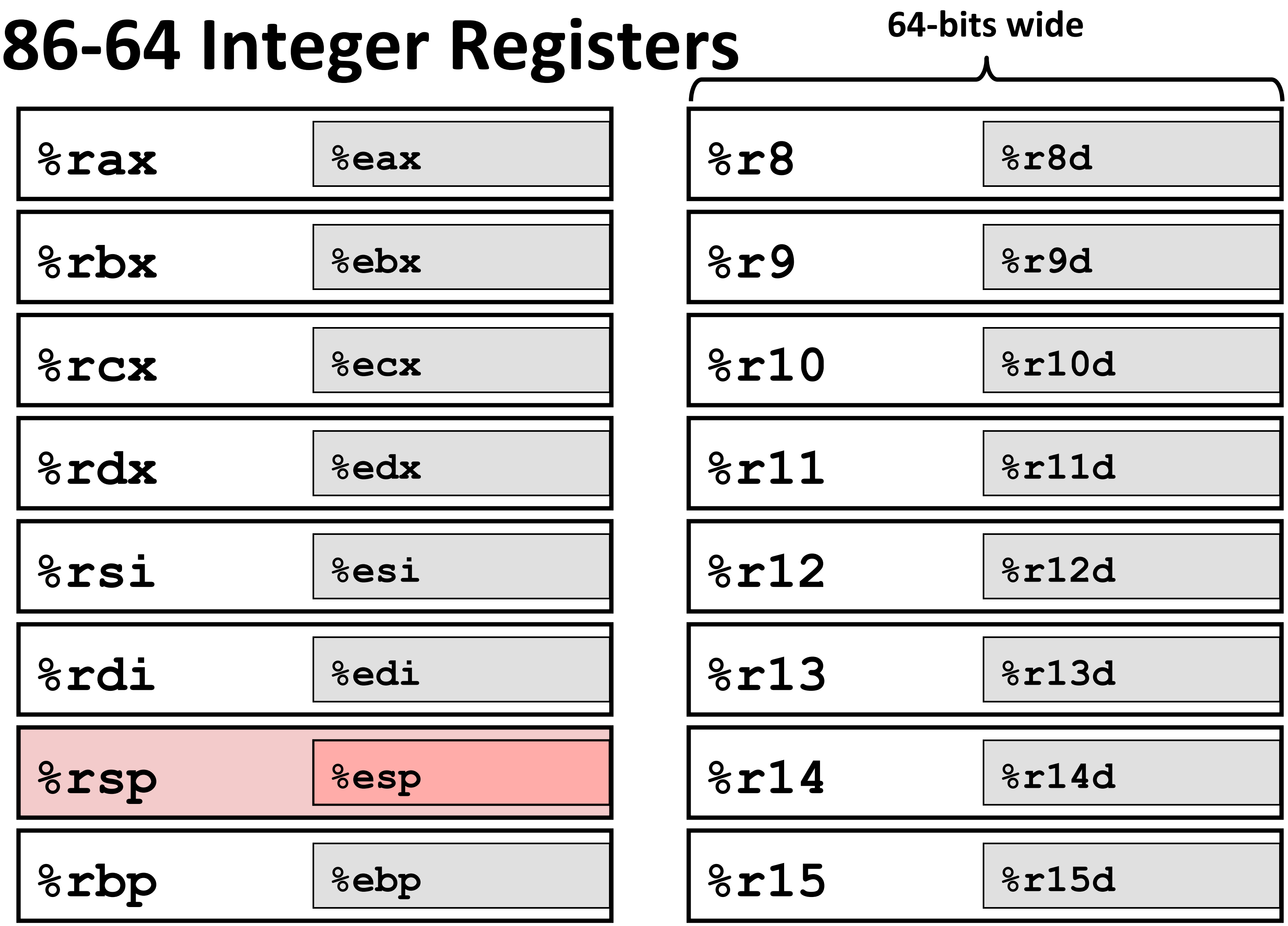
# Understanding Swap

| Register | Value |
|---|---|
| %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
```

# x86-64 Integer Registers




- Extend existing registers, and add 8 new ones; *all* accessible as 8, 16, 32, 64 bits.

# 32-bit vs. 64-bit operands

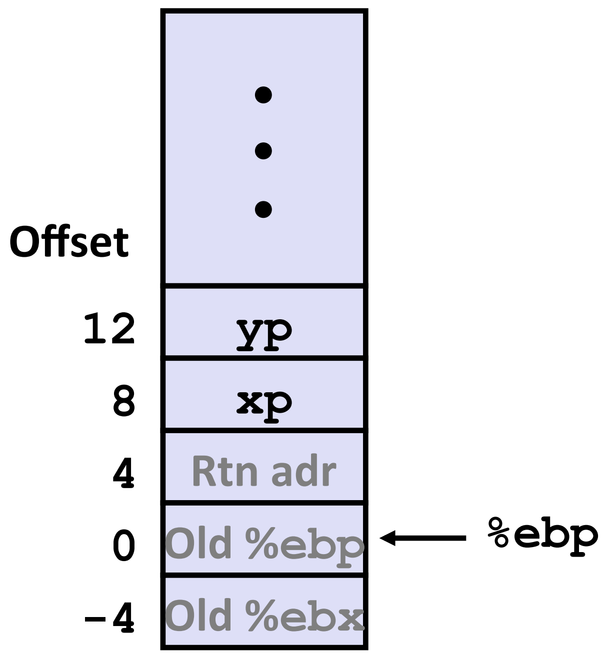
- **Long word `l` (4 Bytes) ↔ Quad word `q` (8 Bytes)**

- **New instruction forms:**
  - `movl` → `movq`
  - `addl` → `addq`
  - `sall` → `salq`
  - etc.

- **x86-64 can still use 32-bit instructions that generate 32-bit results**
  - Higher-order bits of destination register are just set to 0
  - Example: `addl`

# Swap Ints in 32-bit Mode

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




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

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

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

# Swap Ints in 64-bit Mode

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

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

- **Arguments passed in registers (why useful?)**
  - First (**`xp`**) in **`%rdi`**, second (**`yp`**) in **`%rsi`**
  - 64-bit pointers
- **No stack operations required**
- **32-bit data**
  - Data held in registers **`%eax`** and **`%edx`**
  - **`movl`** operation (the **`l`** refers to data width, not address width)

# Swap Long Ints in 64-bit Mode

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

```
swap_l:
    movq    (%rdi), %rdx
    movq    (%rsi), %rax
    movq    %rax, (%rdi)
    movq    %rdx, (%rsi)
    retq
```

- **64-bit data**
  - Data held in registers **`%rax`** and **`%rdx`**
  - **`movq`** operation
  - "q" stands for quad-word

# Complete Memory Addressing Modes

- **Remember, the addresses used for accessing memory in `mov` (and other) instructions can be computed in several different ways**
- **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 the 8/16 integer registers
  - Ri:    Index register: Any, except for **`%esp`** or **`%rsp`**
    - Unlikely you'd use **`%ebp`**, either
  - S:    Scale: 1, 2, 4, or 8 (*why these numbers?*)
- **Special Cases: can use any combination of D, Rb, Ri and S**

  **(Rb,Ri)    Mem[Reg[Rb]+Reg[Ri]]**

  **D(Rb,Ri)    Mem[Reg[Rb]+Reg[Ri]+D]**

  **(Rb,Ri,S)    Mem[Reg[Rb]+S*Reg[Ri]]**

# Address Computation Examples

| | |
|---|---|
| `%edx` | `0xf000` |
| `%ecx` | `0x100` |

| | |
|---|---|
| (Rb,Ri) | Mem[Reg[Rb]+Reg[Ri]] |
| D(,Ri,S) | Mem[S*Reg[Ri]+D] |
| (Rb,Ri,S) | Mem[Reg[Rb]+S*Reg[Ri]] |
| D(Rb) | Mem[Reg[Rb] +D] |

| Expression | Address 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` |

# Address Computation Instruction

- **`leal`** ***Src,Dest***
  - *Src* is address mode expression
  - Set *Dest* to address computed by expression
    - (lea stands for *load effective address)*
  - Example: **`leal (%edx,%ecx,4), %eax`**

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

# Some Arithmetic Operations

- **Two Operand (Binary) Instructions:**

| *Format* | | *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* | ***Also called shll*** |
| `sarl` | *Src,Dest* | *Dest = Dest >> Src* | ***Arithmetic*** |
| `shrl` | *Src,Dest* | *Dest = Dest >> Src* | ***Logical*** |
| `xorl` | *Src,Dest* | *Dest = Dest ^ Src* | |
| `andl` | *Src,Dest* | *Dest = Dest & Src* | |
| `orl` | *Src,Dest* | *Dest = Dest \| Src* | |

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

# Some Arithmetic Operations

- **One Operand (Unary) Instructions**

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

- **See textbook section 3.5.5 for more instructions: `mull`, `cltd`, `idivl`, `divl`**

# 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;
}
```

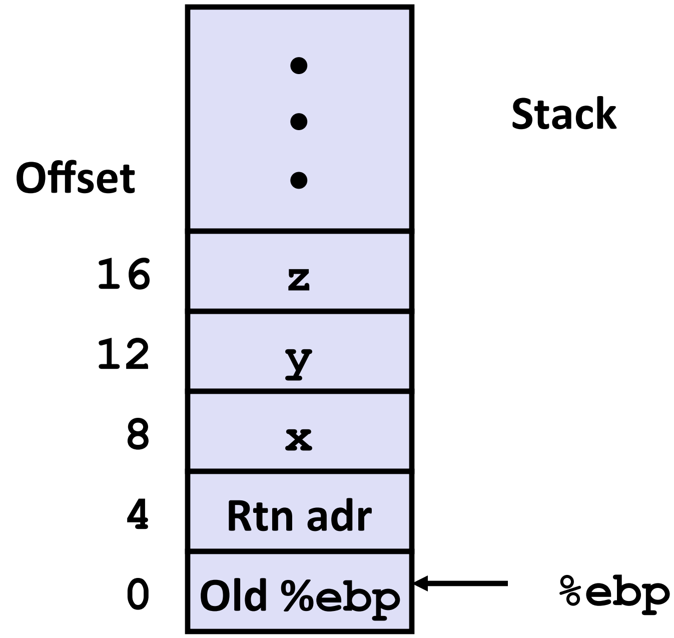
```
arith:
    pushl %ebp                      } Set Up
    movl %esp,%ebp

    movl 8(%ebp),%eax               } Body
    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

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

# Understanding `arith`

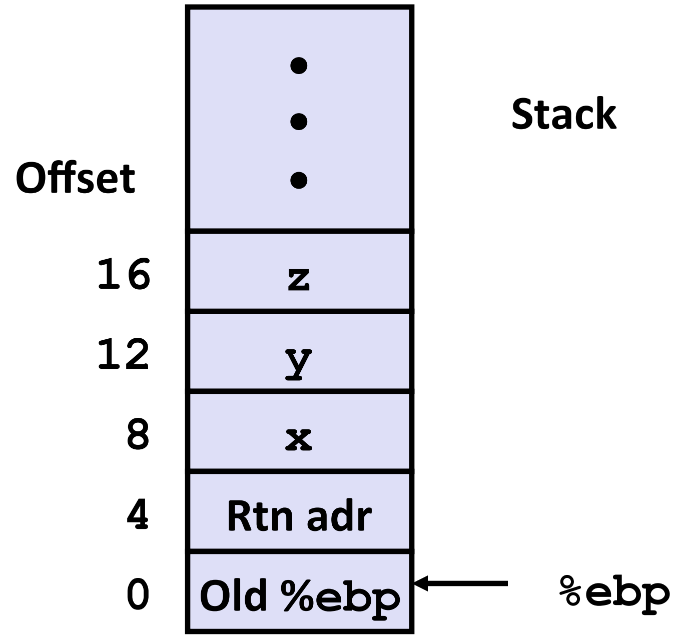
```
int arith
   (int x, int y, int z)
{
   int t1 = x+y;
   int t2 = z+t1;
   int t3 = x+4;
   int t4 = y * 48;
   int t5 = t3 + t4;
   int rval = t2 * t5;
   return rval;
}
```




```
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 = 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`

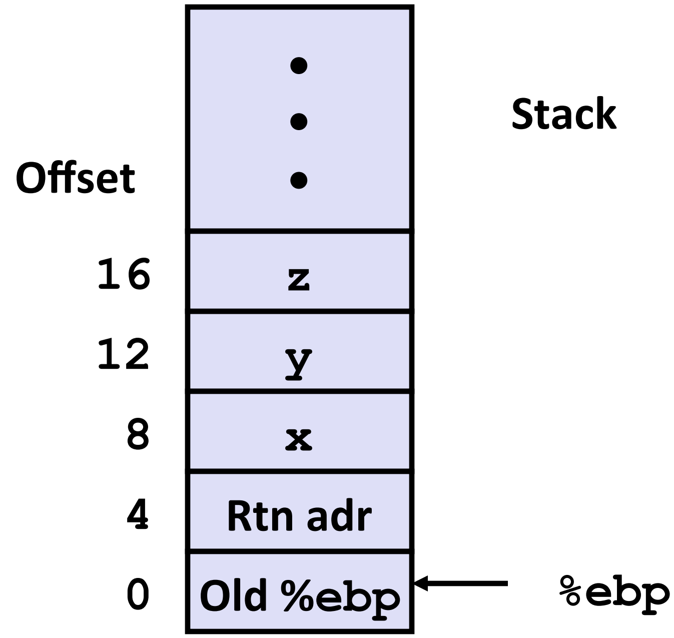
```
int arith
  (int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```




```
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`

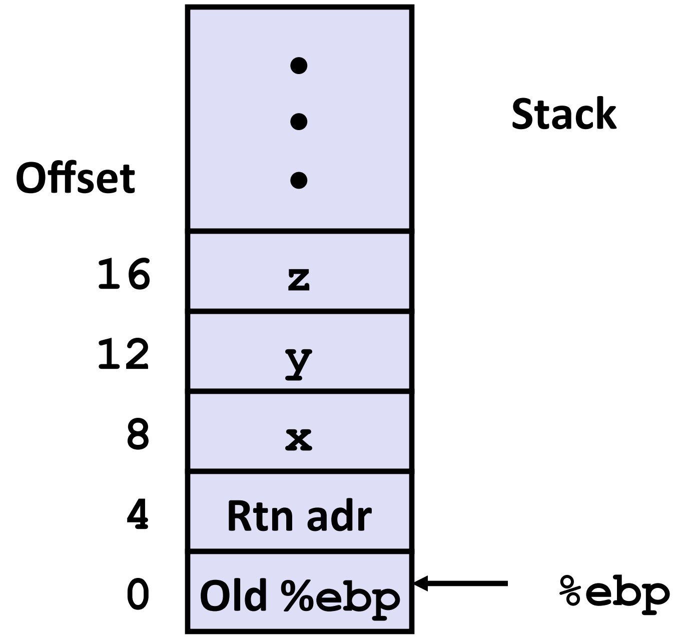
```
int arith
  (int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```




```
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`

```
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)
```

# Observations about `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;
}
```

- Instructions in different order from C code
- Some expressions require multiple instructions
- Some instructions cover multiple expressions
- Get exact same code when compile:
- `(x+y+z)*(x+4+48*y)`

```
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)
```

# 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;
}
```

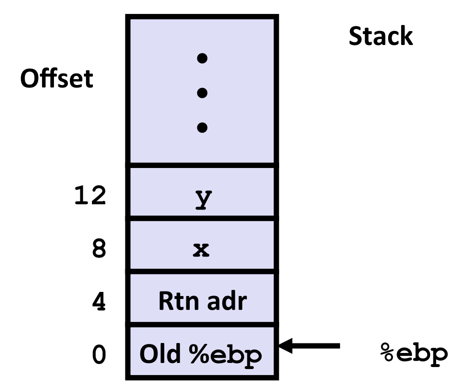
```
logical:
    pushl %ebp
    movl %esp,%ebp

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

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

Set Up (pushl, movl) · Body (movl, xorl, sarl, andl) · Finish (movl, popl, ret)

```
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              } 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
```

```
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
    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
```

# 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
…0010000000000000, …0001111111111001

```
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
```