# Machine-Level Programming – Introduction


## Today

- Assembly programmer's exec model
- Accessing information
- Arithmetic operations

## Next time

- More of the same

Fabián E. Bustamante, Spring 2007

# IA32 Processors

- Totally dominate computer market
- Evolutionary design
  - Starting in 1978 with 8086
  - Added more features as time goes on
  - Backward compatibility: able to run code for earlier version
- Complex Instruction Set Computer (CISC)
  - Many different instructions with many different formats
    - But, only small subset encountered with Linux programs
  - Hard to match performance of Reduced Instruction Set Computers (RISC)
  - But, Intel has done just that!
- X86 evolution clones: Advanced Micro Devices (AMD)
  - Historically followed just behind Intel – a little bit slower, a lot cheaper

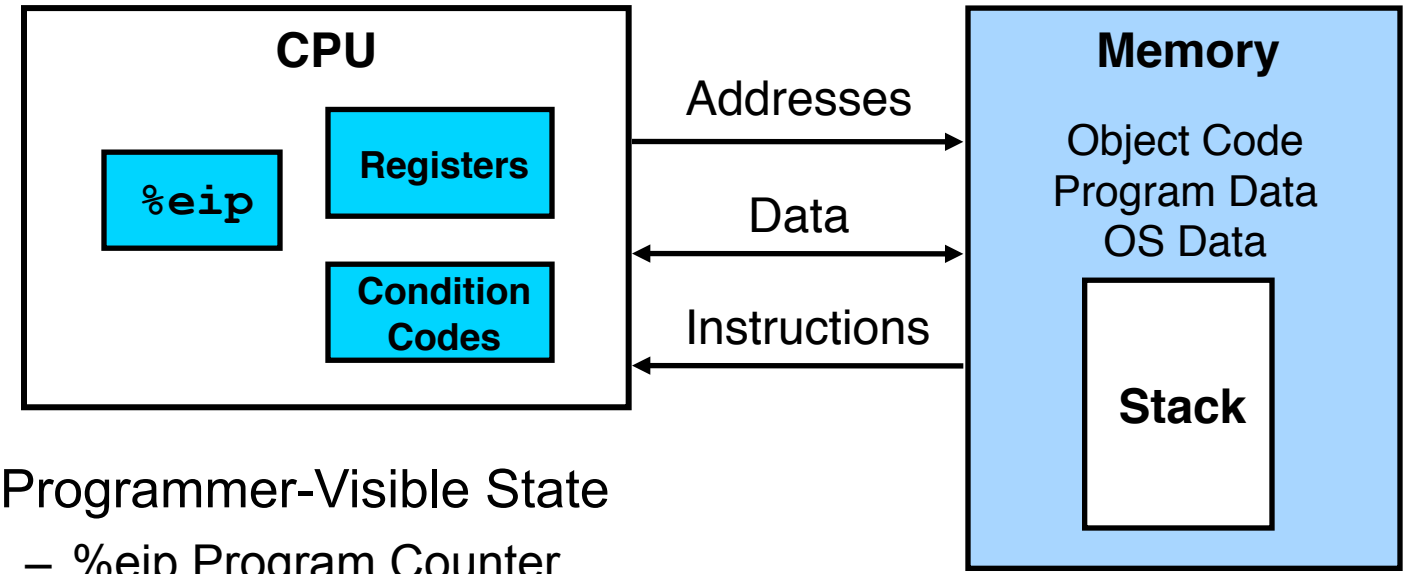
# X86 Evolution: Programmer's view

| Name | Date | Transistors | Comments |
|---|---|---|---|
| 8086 | 1978 | 29k | 16-bit processor, basis for IBM PC & DOS; limited to 1MB address space |
| 80286 | 1982 | 134K | Added elaborate, but not very useful, addressing scheme; basis for IBM PC AT and Windows |
| 386 | 1985 | 275K | Extended to 32b, added "flat addressing", capable of running Unix, Linux/gcc uses |
| 486 | 1989 | 1.9M | Improved performance; integrated FP unit into chip |
| Pentium | 1993 | 3.1M | Improved performance |
| PentiumPro (P6) | 1995 | 6.5M | Added conditional move instructions; big change in underlying microarchitecture |
| Pentium/ MMX | 1997 | 6.5M | Added special set of instructions for 64-bit vectors of 1, 2, or 4 byte integer data |
| Pentium II | 1997 | 7M | Merged Pentium/MMZ and PentiumPro implementing MMX instructions within P6 |
| Pentium III | 1999 | 8.2M | Instructions for manipulating vectors of integers or floating point; later versions included Level2 cache |
| Pentium 4 | 2001 | 42M | 8 byte ints and floating point formats to vector instructions |

# X86 Evolution: Programmer's view

| Name | Date | Transistors | Comments |
|---|---|---|---|
| Pentium 4E | 2004 | 125M | Hyperthreading (execute 2 programs on one processor), EM64T 64-bit extension |
| Core 2 | 2006 | 291M | first multi-core; similar to P6; no hyperthreading |
| Core i7 | 2008 | 781M | multi-core with hyperthreading; 2 programs on each core, up to 4 cores per chip; |

# Assembly programmer's view

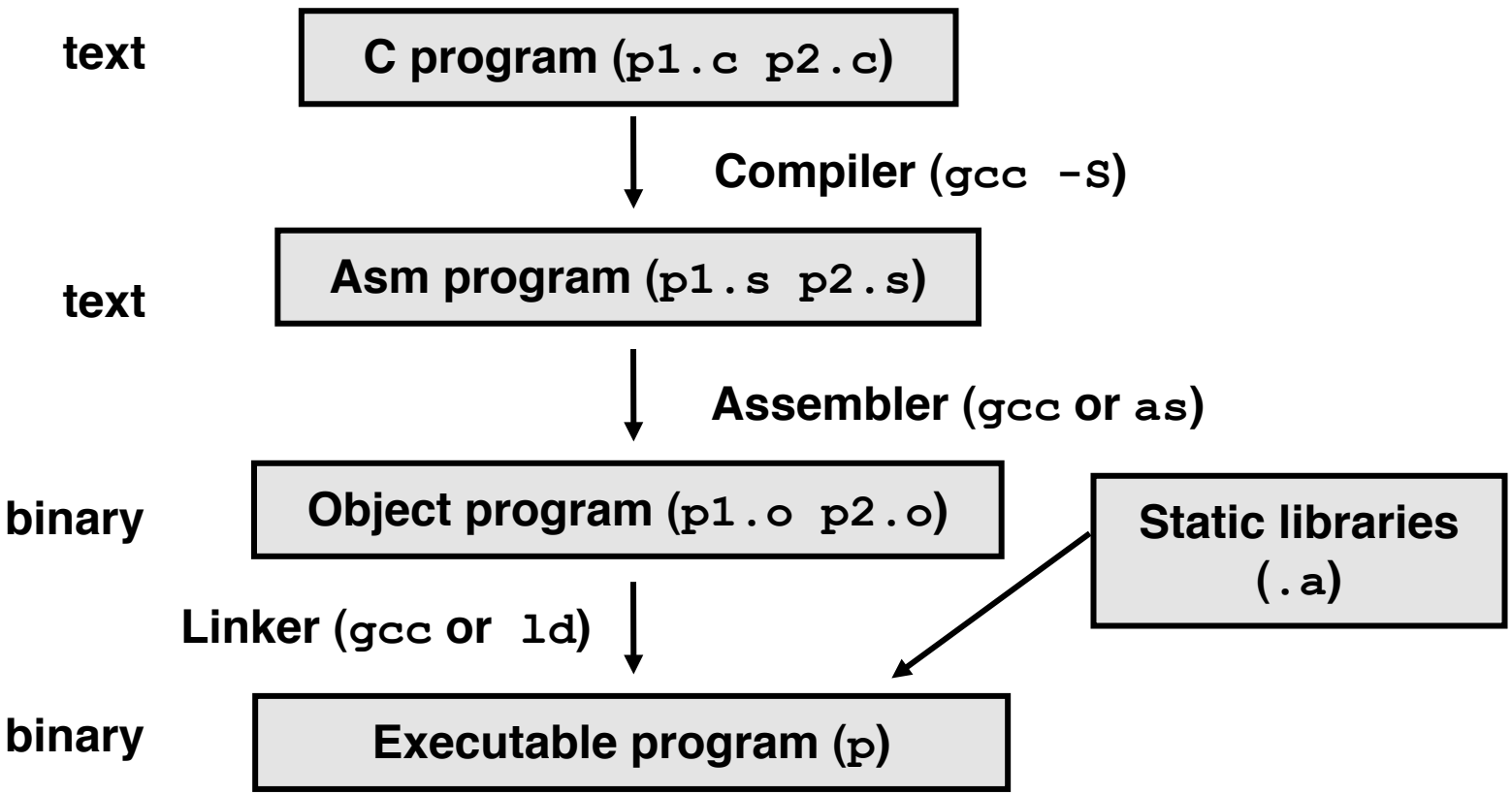



- Programmer-Visible State
  - %eip Program Counter
    - %rip in 64bit
    - Address of next instruction
  - Register file (8x32bit)
    - Heavily used program data
  - Condition codes
    - Store status information about most recent arithmetic operation
    - Used for conditional branching
  - Floating point register file

- Memory
  - Byte addressable array
  - Code, user data, (some) OS data
  - Includes stack used to support procedures

# Turning C into object code

- Code in files p1.c p2.c
- Compile with command: `gcc -O2 p1.c p2.c -o p`
  - Use level 2 optimizations (-O2); put resulting binary in file p



# Compiling into assembly

## code.c (C source)

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

```
gcc -S code.c -O1
```

Text

## code.s (GAS Gnu Assembler)

ordinary text file

might see "leave"

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

# Assembly characteristics

- gcc default target architecture: I386 (flat addressing)
- Minimal data types
  - "Integer" data of 1, 2, or 4 bytes
    - Data values or addresses
  - Floating point data of 4, 8, or 10 bytes
  - No aggregate types such as arrays or structures
    - Just contiguously allocated bytes in memory
- Primitive operations
  - Perform arithmetic function on register or memory data
  - Transfer data between memory and register
    - Load data from memory into register
    - Store register data into memory
  - Transfer control
    - Unconditional jumps to/from procedures
    - Conditional branches

# Object code

## Code for `sum`

```
0x401040
<sum>:      0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x89
    0xec
    0x5d
    0xc3
```

gcc -c code.c -O1

- Assembler
  - Translates `.s` into `.o`
  - Binary encoding of each instruction
  - Nearly-complete image of exec code
  - But unresolved linkages between code in different files, such as function calls

- **Total of 13 bytes**
- **Each instruction 1, 2, or 3 bytes**
- **Starts at address `0x401040`**

# Getting an executable

- To generate an executable requires the linker
  - resolves references between files, e.g., function calls, including to library functions like `printf()`
  - dynamic linking leaves references for resolution at run-time
  - checks that there is one and only one `main()` function

```
gcc -o code.o main.c -O1
```

```
int main()
{
  return sum(1, 3);
}
```

# Machine instruction example

```
int t = x+y;
```

```
addl 8(%ebp),%eax
```

**Similar to C expression**
**`x += y`**

```
0x401046: 03 45 08
```

- C Code
  - Add two signed integers
- Assembly
  - Add 2 4-byte integers
    - "Long" words in GCC parlance
    - Same instruction whether signed or unsigned
  - Operands:
    - `x`: Register `%eax`
    - `y`: Memory M[`%ebp+8`]
    - `t`: Register `%eax`
      - Return function value in `%eax`
- Object code
  - 3-byte instruction
  - Stored at address `0x401046`

# Disassembling object code

**Disassembled**

```
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
   d:      8d 76 00             lea    0x0(%esi),%esi
```

- Disassembler
  - `objdump -d code` (`otool -tV` on MacOS X)
  - Useful tool for examining object code
  - Analyzes bit pattern of series of instructions
  - Produces approximate rendition of assembly code
  - Can be run on either a.out (complete executable) or .o file

# Alternate disassembly

## Object

```
0x401040:
    0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x89
    0xec
    0x5d
    0xc3
```

## Disassembled

```
0x401040 <sum>:      push   %ebp
0x401041 <sum+1>:    mov    %esp,%ebp
0x401043 <sum+3>:    mov    0xc(%ebp),%eax
0x401046 <sum+6>:    add    0x8(%ebp),%eax
0x401049 <sum+9>:    mov    %ebp,%esp
0x40104b <sum+11>:   pop    %ebp
0x40104c <sum+12>:   ret
0x40104d <sum+13>:   lea    0x0(%esi),%esi
```

- Within gdb debugger
  - Once you know the length of sum using the dissambler
  - Examine the 13 bytes starting at `sum`

```
gdb code.o
x/13b sum
```

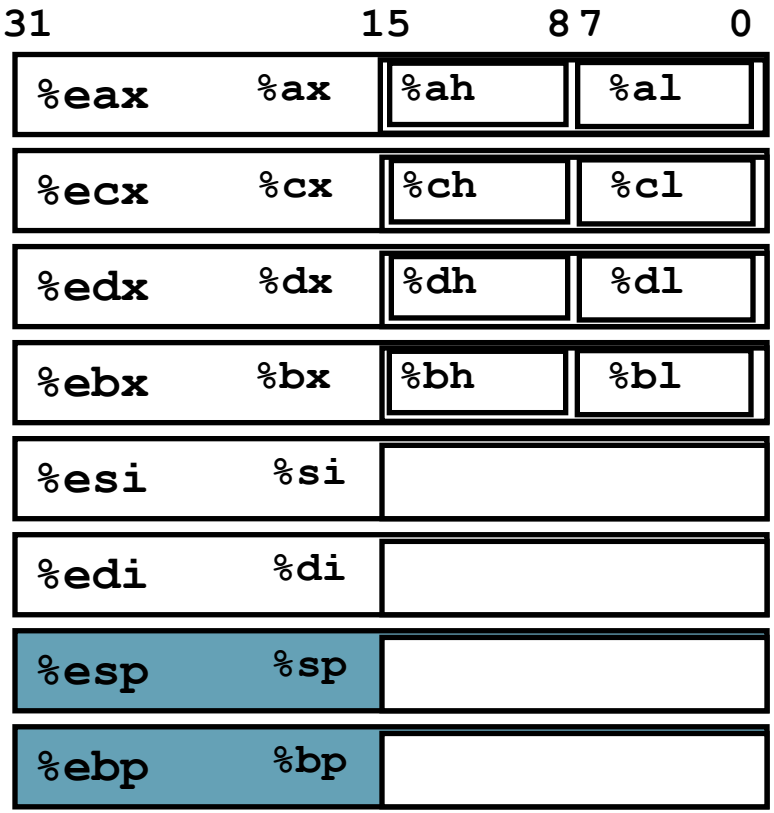
# Data formats

- "word" – (Intel) 16b data type (historical)
  - 32b – double word
  - 64b – quad words
- In GAS, operator suffix indicates word size involved.
- The overloading of "l" (long) OK because FP involves different operations & registers

| C decl | Intel data type | GAS suffix | Size (bytes) |
|---|---|---|---|
| char | Byte | b | 1 |
| short | Word | w | 2 |
| int, unsigned, long int, unsigned long, char * | Double word | l | 4 |
| float | Single precision | s | 4 |
| double | Double precision | l | 8 |
| long double | Extended precision | t | 10/12 |

# Registers

- Eight 32bit registers
- First six mostly general purpose
- Last two used for process stack
- First four also support access to low order bytes and words



# Instruction formats

- Most instructions have 1 or 2 operands
  - operator [source[, destination]]
  - Operand types:
    - Immediate – constant, denoted with a “$” in front
    - Register – either 8 or 16 or 32bit registers
    - Memory – location given by an effective address
  - Source: constant or value from register or memory
  - Destination: register or memory

# Operand specifiers

- Operand forms
  - *Imm* means a number
  - $E_a$ means a register form, e.g., %eax
  - *s* is 1, 2, 4 or 8 (called the scale factor)
  - Memory form is the most general; subsets also work, e.g.,
    - Absolute: *Imm* ⇒ M[*Imm*]
    - Base + displacement: *Imm*($E_b$) ⇒ M[*Imm* + R[$E_b$]]
- Operand values
  - R[$E_a$] means "value in register"
  - M[*loc*] means "value in memory location *loc*"

| Type | Form | Operand value | Name |
|---|---|---|---|
| Immediate | $*Imm* | *Imm* | Immediate |
| Register | $E_a$ | R[$E_a$] | Register |
| Memory | *Imm* ($E_b$, $E_i$, *s*) | M[*Imm* + R[$E_b$] + R[$E_i$] * *s*] | Scaled indexed |

# Operand specifiers

- Memory form has many subsets
  - Don't confuse $*Imm* with *Imm*, or $E_a$ with $(E_a)$

| Type | Form | Operand value | Name |
|---|---|---|---|
| Memory | *Imm* | M[*Imm*] | Absolute |
| Memory | $(E_a)$ | M[R[$E_b$]] | Indirect |
| Memory | *Imm* $(E_b)$ | M[*Imm* + R[$E_b$]] | Base + displacement |
| Memory | $(E_b, E_i)$ | M[R[$E_b$] + R[$E_i$]] | Indexed |
| Memory | *Imm* $(E_b, E_i)$ | M[*Imm* + R[$E_b$] + R[$E_i$]] | Indexed |
| Memory | $(, E_i, s)$ | M[R[$E_i$] * *s*] | Scaled indexed |
| Memory | *Imm* $(, E_i, s)$ | M[*Imm* + R[$E_i$] * *s*] | Scaled indexed |
| Memory | $(E_b, E_i, s)$ | M[R[$E_b$] + R[$E_i$] * *s*] | Scaled indexed |
| Memory | *Imm* $(E_b, E_i, s)$ | M[*Imm* + R[$E_b$] + R[$E_i$] * *s*] | Scaled indexed |

# Checkpoint

# Moving data

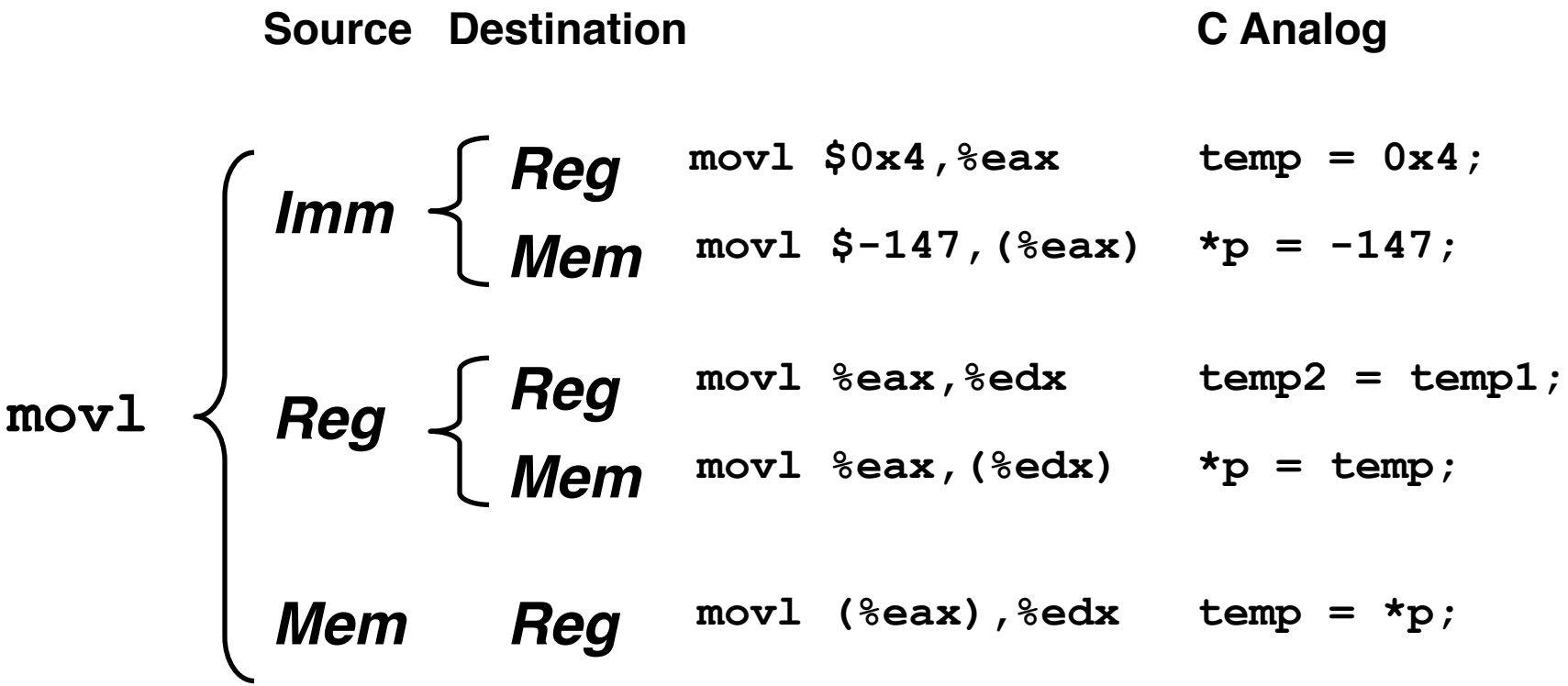
- Among the most common instructions
- IA32 restriction – cannot move from one memory location to another with one instruction
- Note the differences between `movb, movsbl and movzbl`
- Last two work with the stack

```
pushl %ebp        =        subl $4, %esp
                           movl %ebp, (%esp)
```

- Since stack is part of program mem, you can really access all

| Instruction | Effect | Description |
|---|---|---|
| `mov{l,w,b}` S,D | D ← S | Move double word, word or byte |
| `movsbl` S,D | D ← SignExtend(S) | Move sign-extended byte |
| `movzbl` S,D | D ← ZeroExtend(S) | Move zero-extended byte |
| `pushl` S | R[%esp] ← R[%esp] – 4;<br>M[R[%esp]] ← S | Push S onto the stack |
| `popl` D | D ← M[R[%esp]]<br>R[%esp] ← R[%esp] + 4; | Pop S from the stack |

# `movl` operand combinations



# Using simple addressing modes

Declares xp as being a pointer to an int

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

Read value stored in location xp and store it in t0

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

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

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

Set Up

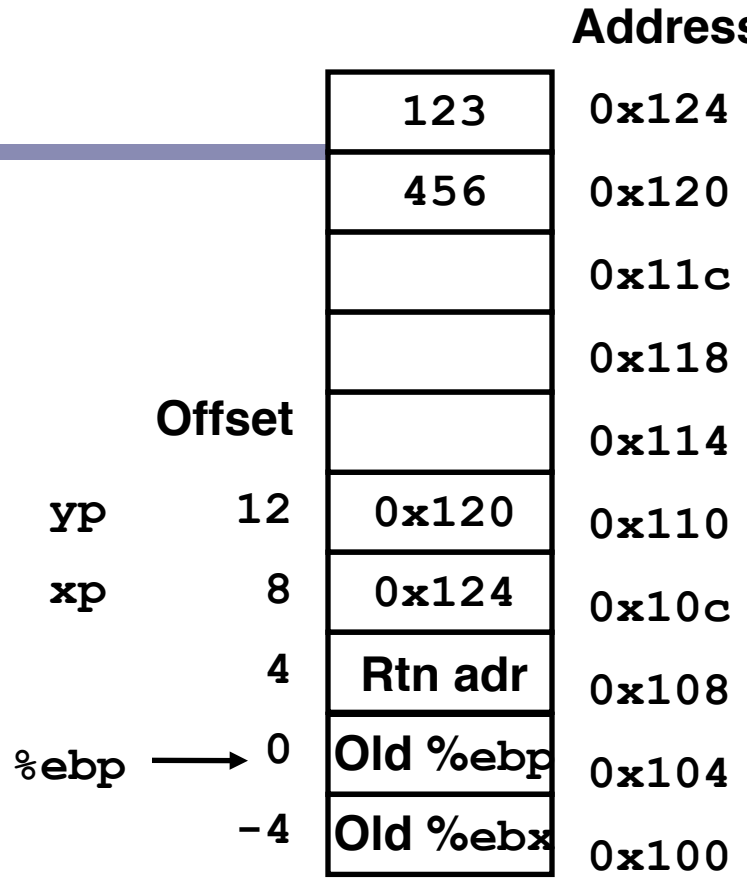
Body

Finish

# Understanding swap

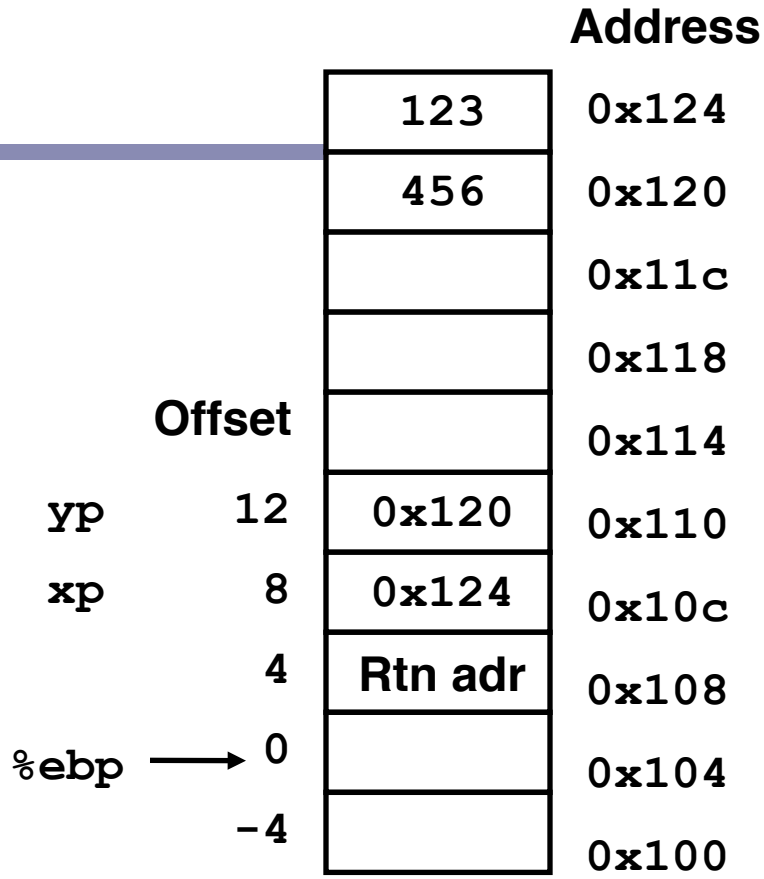
```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

| Register | Variable |
|---|---|
| %ecx | yp |
| %edx | xp |
| %eax | t1 |
| %ebx | t0 |



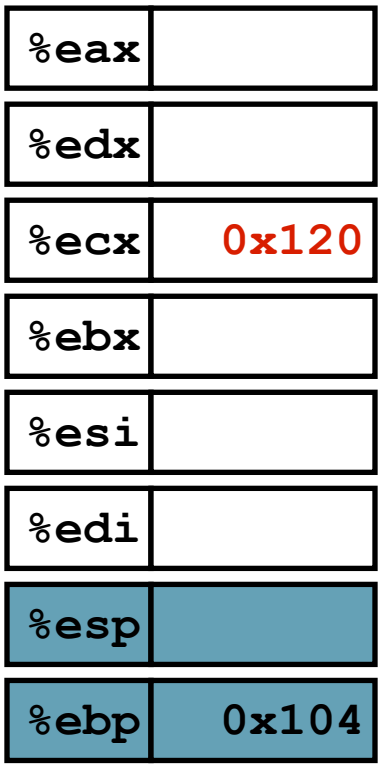

```
movl 12(%ebp),%ecx      # ecx = yp
movl 8(%ebp),%edx       # edx = xp
movl (%ecx),%eax        # eax = *yp (t1)
movl (%edx),%ebx        # ebx = *xp (t0)
movl %eax,(%edx)        # *xp = eax
movl %ebx,(%ecx)        # *yp = ebx
```

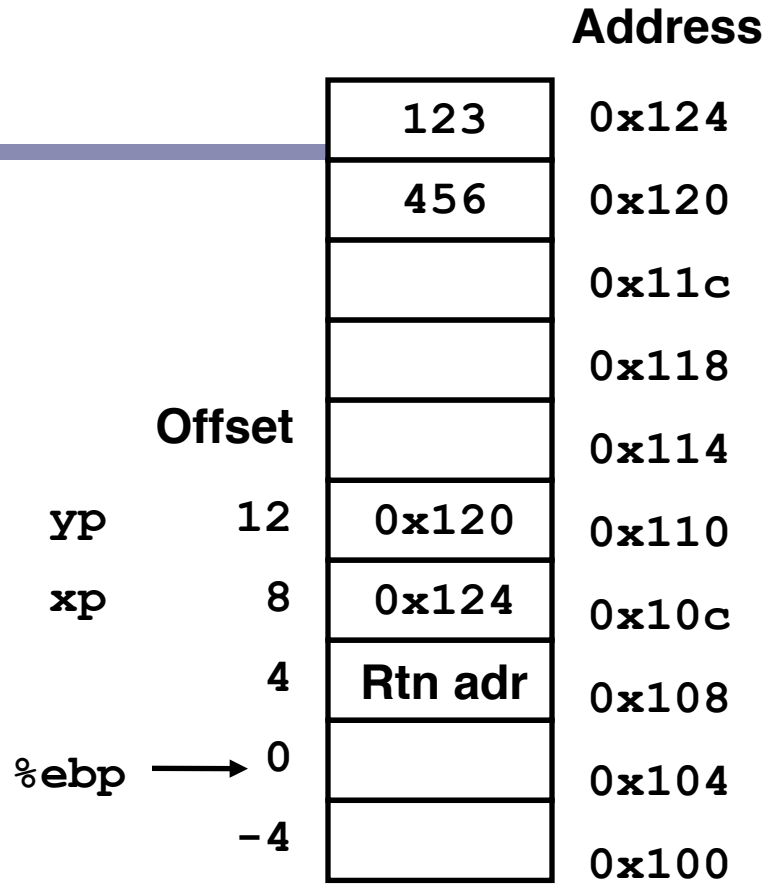
# Understanding swap


```
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
```

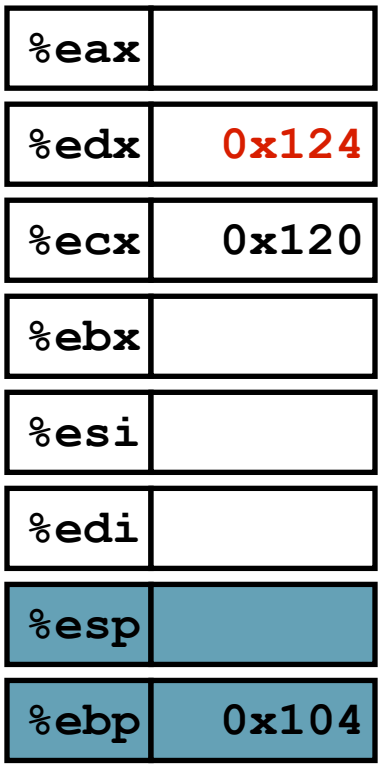
# Understanding swap

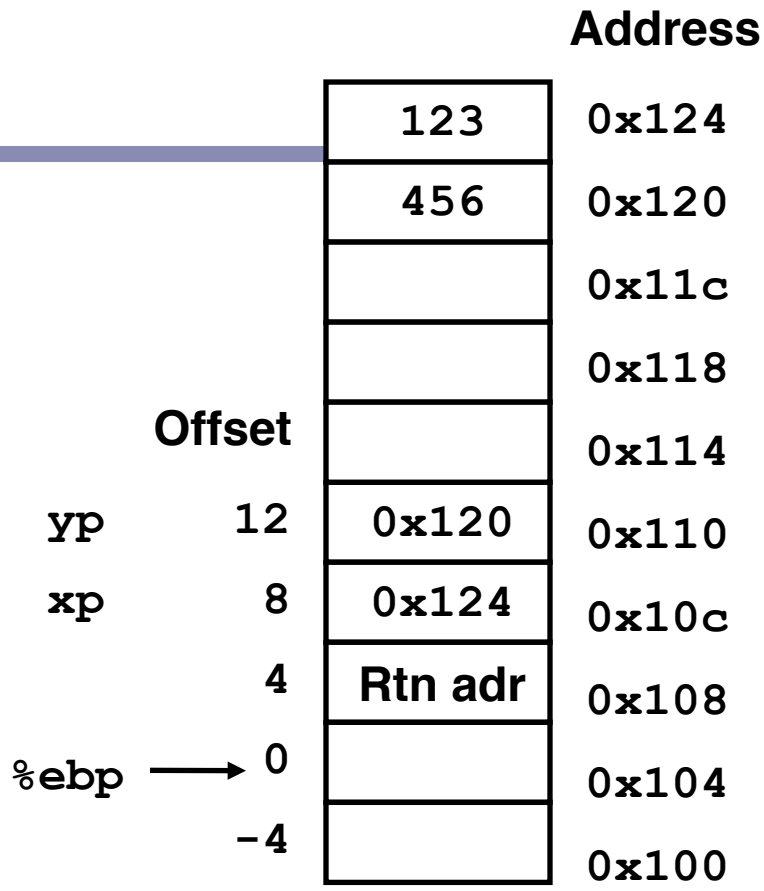



```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```

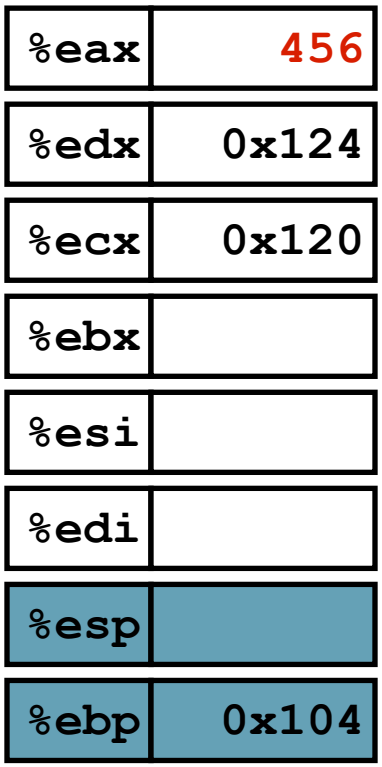
# Understanding swap

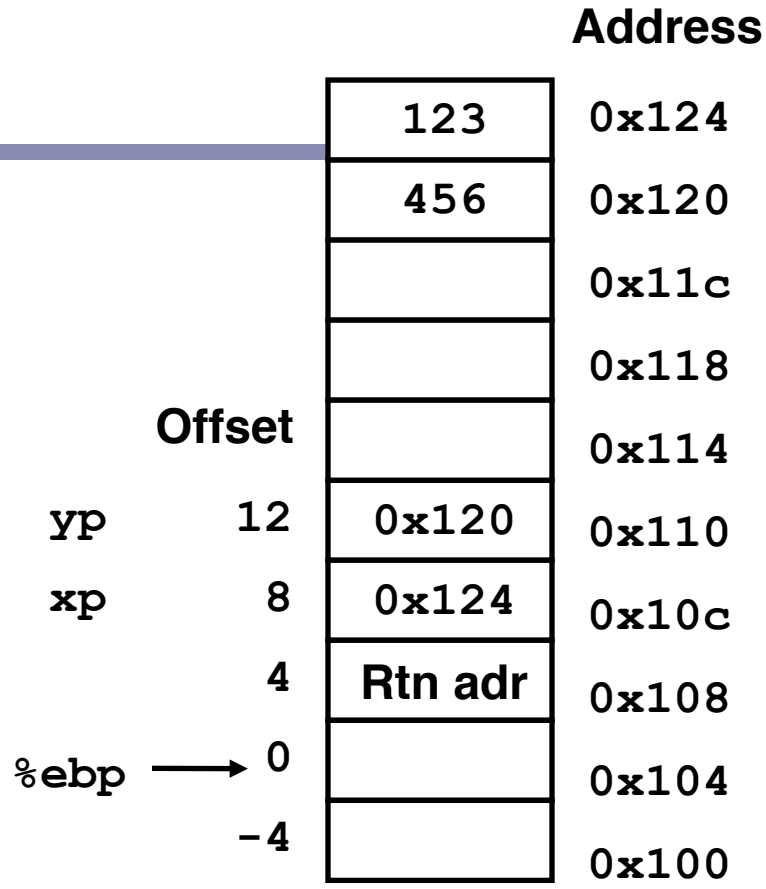



```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```

# Understanding swap

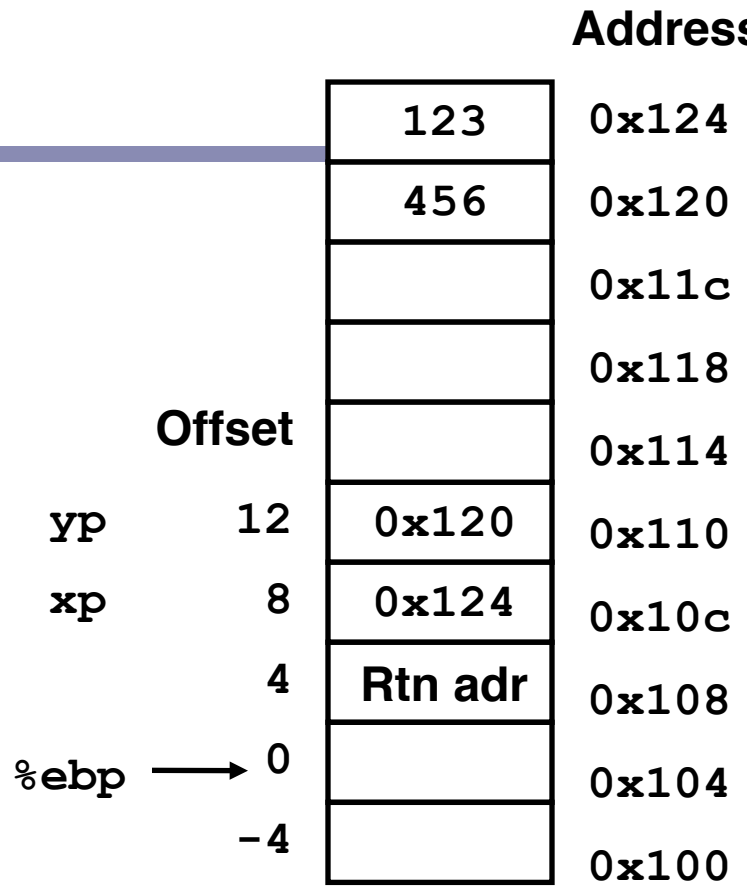



```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```

# Understanding swap

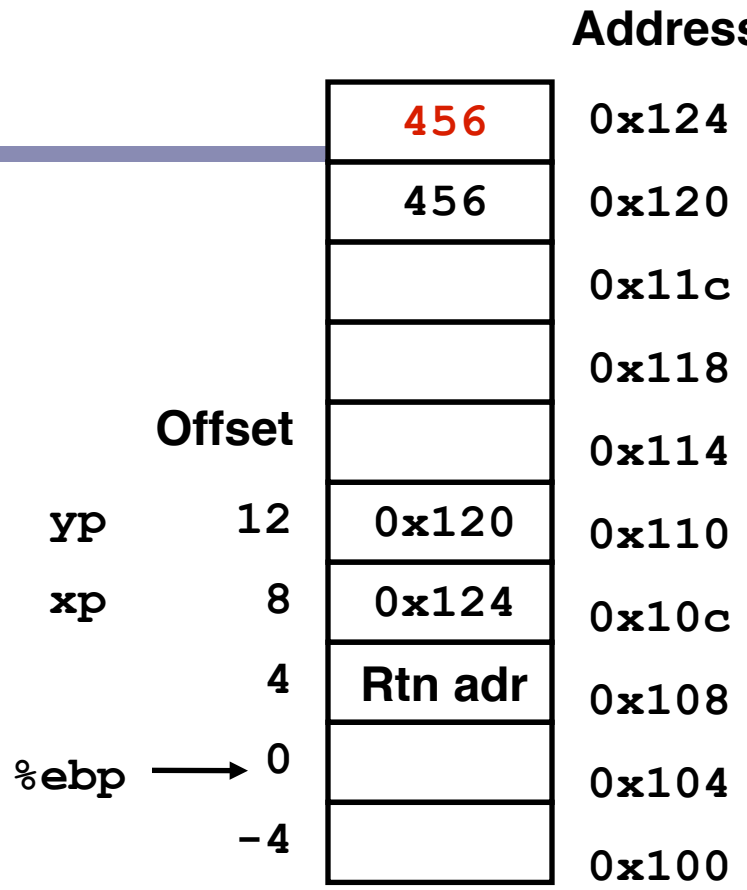
| | |
|---|---|
| %eax | 456 |
| %edx | 0x124 |
| %ecx | 0x120 |
| %ebx | 123 |
| %esi | |
| %edi | |
| %esp | |
| %ebp | 0x104 |




```
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
```

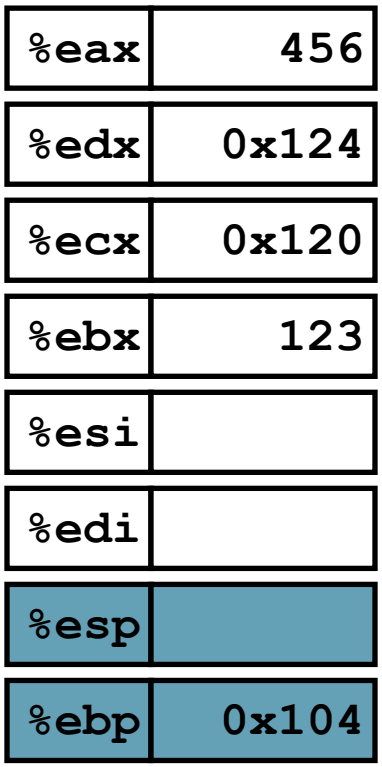
# Understanding swap

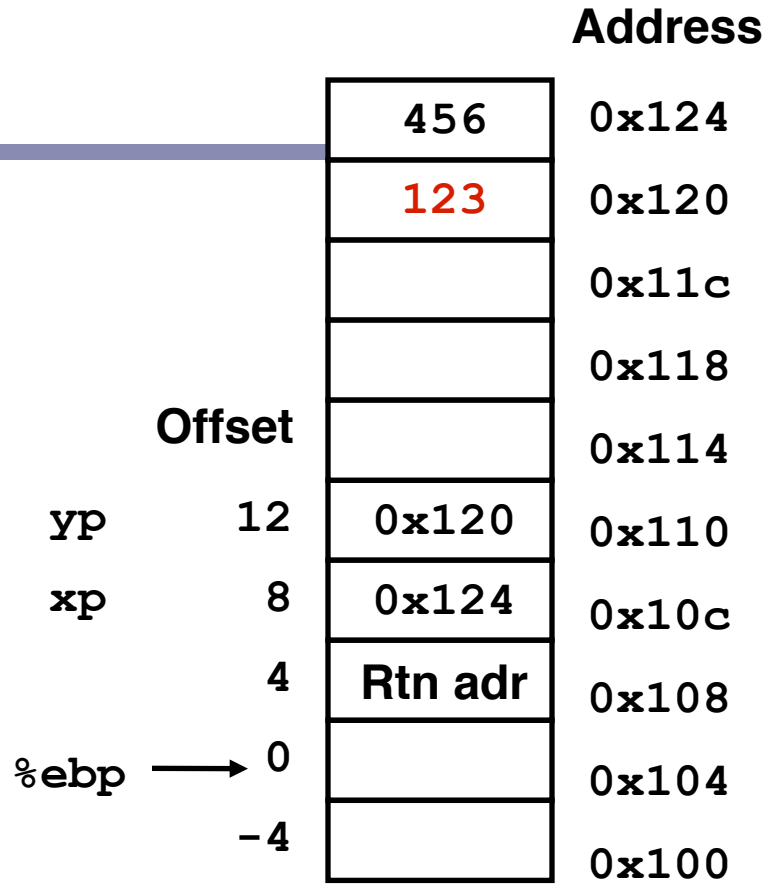
| | |
|---|---|
| %eax | 456 |
| %edx | 0x124 |
| %ecx | 0x120 |
| %ebx | 123 |
| %esi | |
| %edi | |
| %esp | |
| %ebp | 0x104 |




```
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
```

# Checkpoint

# Address computation instruction

- `leal S,D` D ← &S
  - `leal` = Load Effective Address
  - `S` is address mode expression
  - Set `D` to address denoted by expression
- Uses
  - Computing address w/o doing memory reference
    - E.g., translation of p = &x[i];
  - Computing arithmetic expressions of form x + k*y

    k = 1, 2, 4, or 8.

    ```
    leal 7(%edx,%edx,4), %eax
    ```

    - when `%edx=x`, `%eax` becomes 5x+7

# Checkpoint

# Some arithmetic operations

| Instruction | Effect | Description |
|---|---|---|
| `incl` D | D ← D + 1 | Increment |
| `decl` D | D ← D – 1 | Decrement |
| `negl` D | D ← -D | Negate |
| `notl` D | D ← ~D | Complement |
| `addl` S,D | D ← D + S | Add |
| `subl` S,D | D ← D – S | Subtract |
| `imull` S,D | D ← D * S | Multiply |
| `xorl` S,D | D ← D ^ S | Exclusive or |
| `orl` S,D | D ← D \| S | Or |
| `andl` S,D | D ← D & S | And |
| `sall` k,D | D ← D << k | Left shift, 0 ≤ k ≤ 31, *Imm* or %cl |
| `shll` k,D | D ← D << k | Left shift (same as sall) |
| `sarl` k,D | D ← D >> k | Arithmetic right shift |
| `shrl` k,D | D ← D >> k | Logical right shift |

# Checkpoint

# 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

    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

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

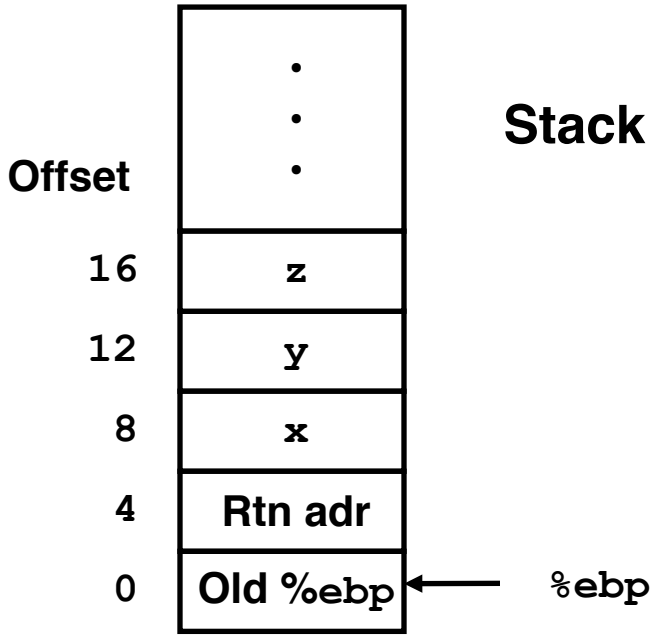
**Set Up**

**Body**

**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)
```

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

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

Set Up

Body

Finish

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

```
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
```

# CISC Properties

- Instruction can reference different operand types
  - Immediate, register, memory
- Arithmetic operations can read/write memory
- Memory reference can involve complex computation
  - Rb + S*Ri + D
  - Useful for arithmetic expressions, too
- Instructions can have varying lengths
  - IA32 instructions can range from 1 to 15 bytes

# Whose assembler?

**Intel/Microsoft Format**

```
lea   eax,[ecx+ecx*2]
sub   esp,8
cmp   dword ptr [ebp-8],0
mov   eax,dword ptr [eax*4+100h]
```

**GAS/Gnu Format**

```
leal (%ecx,%ecx,2),%eax
subl $8,%esp
cmpl $0,-8(%ebp)
movl $0x100(,%eax,4),%eax
```

- Intel/Microsoft Differs from GAS
  - Operands listed in opposite order
    `mov Dest, Src`  `movl Src, Dest`
  - Constants not preceded by '$', Denote hex with 'h' at end
    `100h`  `$0x100`
  - Operand size indicated by operands rather than operator suffix
    `sub`  `subl`
  - Addressing format shows effective address computation
    `[eax*4+100h]`  `$0x100(,%eax,4)`