Machine-Level Programming – Introduction

Today

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

Next time

- More of the same

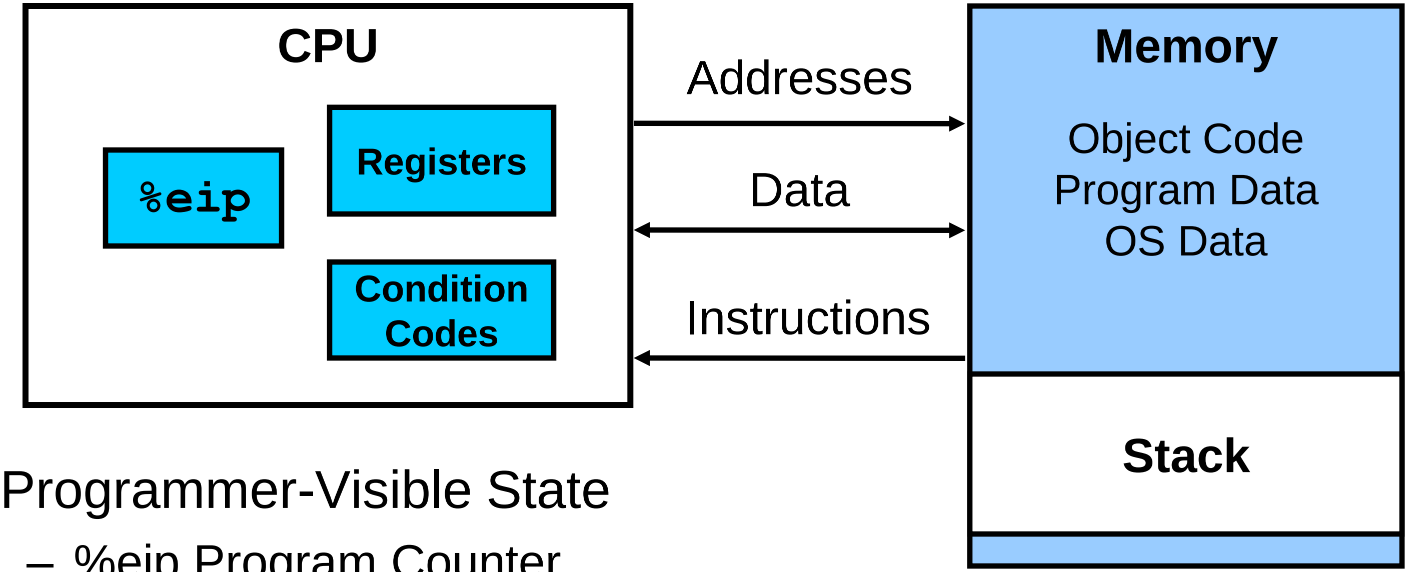
IA32 Processors

- Totally dominate computer market
- Evolutionary design
  - Backward compatible up to 8086 introduced in 1978
  - Added more features as time goes on
- 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 RISC (Reduced …)
    - But, Intel has done just that!
- X86 evolution clones: Advanced Micro Devices (AMD)
  - Historically followed just behind Intel
  - Then hired designers from DEC and others, built Opteron (competitor to Pentium 4), developed x86-64
  - Intel has been quicker w/ multi-core design

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	1995	6.5M	Added conditional move instructions; big change in underlying microarch (called P6 internally)
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	8B ints and floating point formats to vector instructions
Pentium 4E	2004	125M	Hyperthreading (able to run 2 programs simultaneously) and 64b extension
Core 2	2006	291M	P6-like, multicore, no hyperthreading
Core i7	2008	781M	Hyperthreading and multicore

Assembly programmer's view

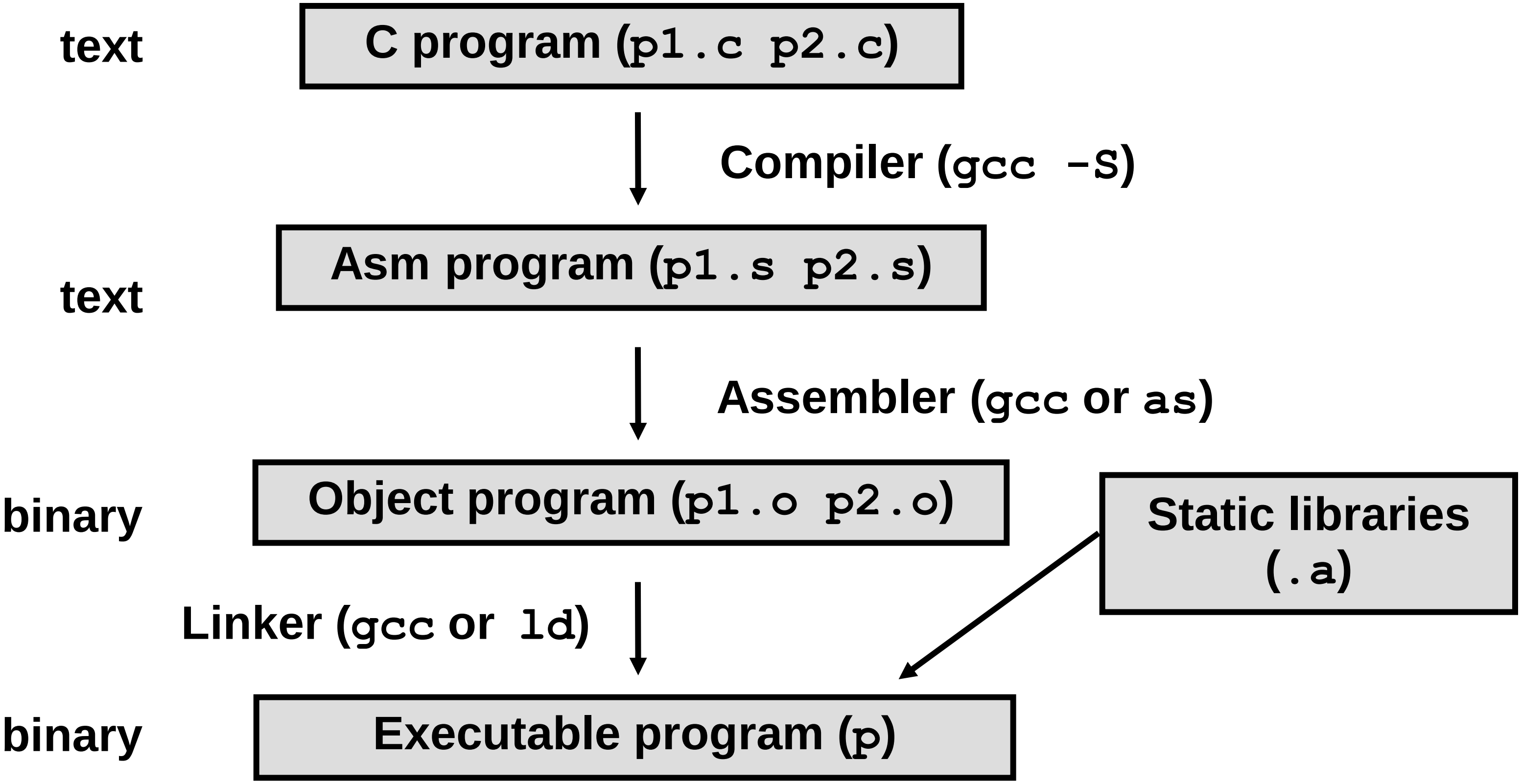


- Programmer-Visible State
  - %eip Program Counter (%rip in x86-64)
    - Address of next instruction
  - Register file (8x32bit)
    - Heavily used program data
  - Condition codes
    - Store status information about most recent arithmetic operation
    - Used for conditional branching
  - Floating 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 -O1 p1.c p2.c -o p`
  - Use level 1 optimizations (-O1); put resulting binary in file p



text — **C program (`p1.c p2.c`)**

Compiler (`gcc -S`)

text — **Asm program (`p1.s p2.s`)**

Assembler (`gcc` or `as`)

binary — **Object program (`p1.o p2.o`)**

Static libraries (`.a`)

Linker (`gcc` or `ld`)

binary — **Executable program (`p`)**

Compiling into assembly

C code

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

Obtain with command

```
gcc –O1 -S code.c
```

Produces file `code.s`

Generated assembly

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

Some compilers or optimization levels use `leave`

Assembly characteristics

- gcc default target architecture: I386 (flat addressing)
- Minimal data types
  - “Integer” data of 1 (byte), 2 (word), 4 (long) or 8 (quad) 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

Obtain with command

```
gcc –O1 -c code.c
```

Produces file `code.o`

Embedded within, the 11-byte sequence for `sum`

Code for `sum`

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

- Assembler
  - Translates `.s` into `.o`
  - Binary encoding of each instruction
  - Nearly-complete image of exec code
  - Missing linkages between code in different files

Getting the byte representation

Object

```
0x0 <sum>:     0x55 0x89 0xe5 0x8b 0x45 0x0c 0x03 0x45
0x8 <sum+8>:   0x08  0x5d  0xc3
```

- Within gdb debugger
  - Once you know the length of sum using the disassembler
  - E**x**amine the 24 bytes starting at `sum`

```
% gdb code.o
(gdb) x/11xb sum
```

Machine instruction example

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

- C Code
  - Add two signed integers

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

Similar to C expression

`x += y`

- 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`

```
0x80483d6: 03 45 08
```

- Object code
  - 3-byte instruction
  - Stored at address `0x80483d6`

And now the executable

- To generate executable requires linker
  - Resolves references bet/ files (One object file must contain main)
  - Combines with static run-time libraries (e.g., `printf`)
  - Some libraries are *dynamically linked* (i.e. at execution)

Obtain with command

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

C code

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

Disassembling object code

- Disassembler
  - `objdump -d prog`
  - 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

Disassembled

```
080483d0 <sum>:
 80483d0:       55                      push    %ebp
 80483d1:       89 e5                   mov     %esp,%ebp
 80483d3:       8b 45 0c                mov     0xc(%ebp),%eax
 80483d6:       03 45 08                add     0x8(%ebp),%eax
 80483d9:       5d                      pop     %ebp
 80483da:       c3                      ret
```

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]
```

ATT Format

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

- Intel/Microsoft Differs from ATT
  - 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)`

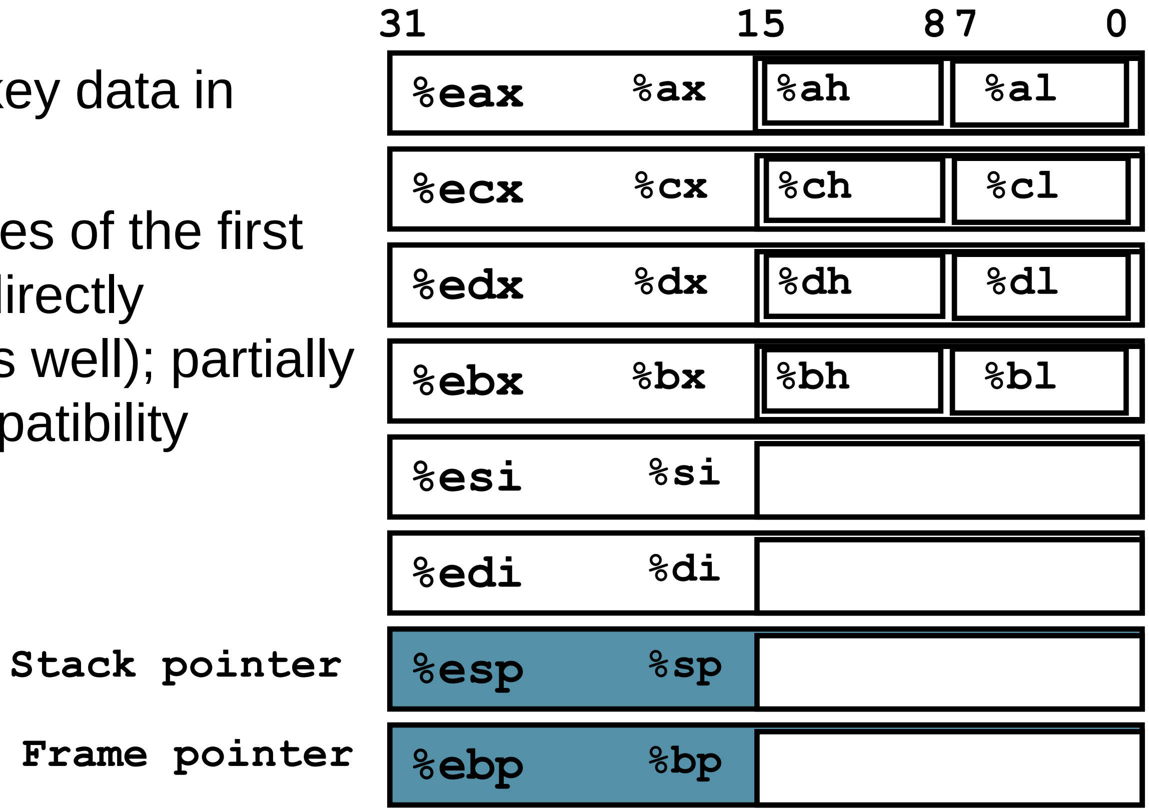
Data formats

- “word” – For Intel, 16b data type due to its origins
  - 32b – double word
  - 64b – quad words
- The overloading of “l” in GAS causes no problems since 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

Accessing information

- 8 32bit registers
- Six of them mostly for general purpose
- Last two point to key data in a process stack
- Two low-order bytes of the first 4 can be access directly (low-order 16bit as well); partially for backward compatibility



	31	15	8 7	0
	%eax	%ax	%ah	%al
	%ecx	%cx	%ch	%cl
	%edx	%dx	%dh	%dl
	%ebx	%bx	%bh	%bl
	%esi	%si		
	%edi	%di		
Stack pointer	%esp	%sp		
Frame pointer	%ebp	%bp		

Operand specifiers

- Most instructions have 1 or 2 operands
  - Source: constant or read from register or memory
  - Destination: register or memory
  - Types:
    - Immediate – constant, denoted with a “$” in front (e.g. $-57, $0x1F)
    - Register – either 8 or 16 or 32bit registers
    - Memory – location given by an effective address
- Operand forms – last is the most general
  - s, scale factor, must be 1, 2, 4 or 8
  - Other memory forms are cases of it
    - Absolute - M[Imm]; Based + displacement: M[Imm + R[E_b]]

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]	Absolute, Indirect, Based + displacement, Indexed, Scale indexed

Practice problem

Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Register	Value
%eax	0x100
%ecx	0x1
%edx	0x3

Operand	Form	Value
%eax	R[%eax]	0x100
0x104	M[0x104]	0xAB
$0x108	0x108	0x108
(%eax)	M[R[%eax]]	0xFF
4(%eax)	M[4 + R[%eax]]	0xAB
9(%eax,%edx)	M[9 + R[%eax] + R[%edx]]	0x11
260(%ecx,%edx)	M[260 + R[%ecx] + R[%edx]]	0x13
0xFC(,%ecx,4)	M[0xFC + R[%ecx]*4]	0xFF
(%eax,%edx,4)	M[R[%eax]+ R[%edx]*4]	0x11

Moving data

- Among the most common instructions

Instruction	Effect	Description
`mov{l,w,b}` S,D	D ← S	Move double word, word or byte
`movs{bw,bl,wl}` S,D	D ← SignExtend(S)	Move sign-extended byte to word, to double-word and word to double-word
`movz{bw,bl,wl}` S,D	D ← ZeroExtend(S)	Move zero-extended byte to word, to double-word and word to double-word
`pushl` S	R[%esp] ← R[%esp] – 4; M[R[%esp]] ← S	Push S onto the stack

`popl` S	D ← M[R[%esp]]
	R[%esp] ← R[%esp] + 4;
		Pop S from the stack

- e.g.

```
movl $0x4050, %eax         Immediate to register
movw %bp, %sp              Register to register
movb (%edi, %ecx), %ah     Memory to register
```

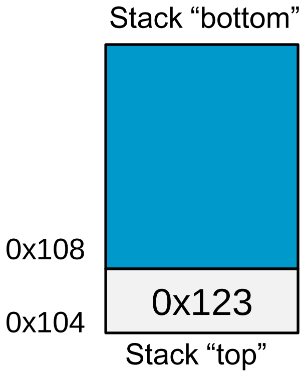
Moving data

- Note the differences between `movb, movsbl and movzbl`

```
   Assume %dh = CD, %eax = 98765432
   movb %dh,%al                 %eax = 987654CD
  movsbl %dh,%eax               %eax = FFFFFFCD
  movzbl %dh,%eax               %eax = 000000CD
```

- Last two work with the stack

```
%eax = 0x123, %esp = 0x108
pushl %ebp
   subl $4, %esp
   movl %ebp, (%esp)
```



- Since stack is part of program mem, you can really access any part of it using standard memory addressing

`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;`**

IA32 restriction – cannot move between two memory locations with one instruction

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

```
xp at %ebp+8, yp at %ebp+12

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

Stack set up

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

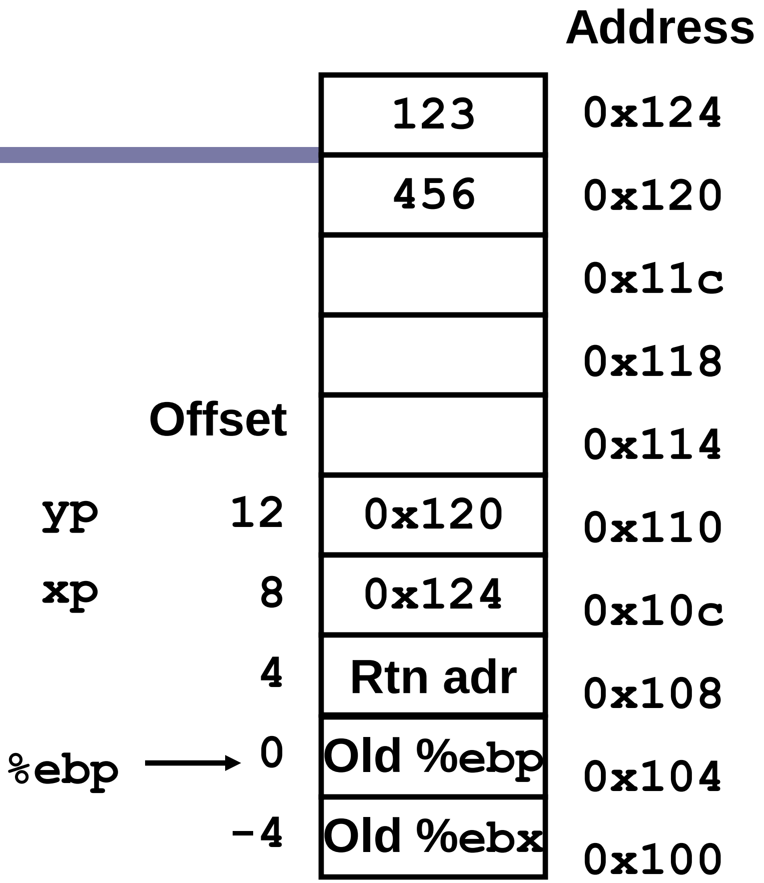
Body

```
    popl %ebx
    leave
    ret
```

Finish

Understanding swap

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



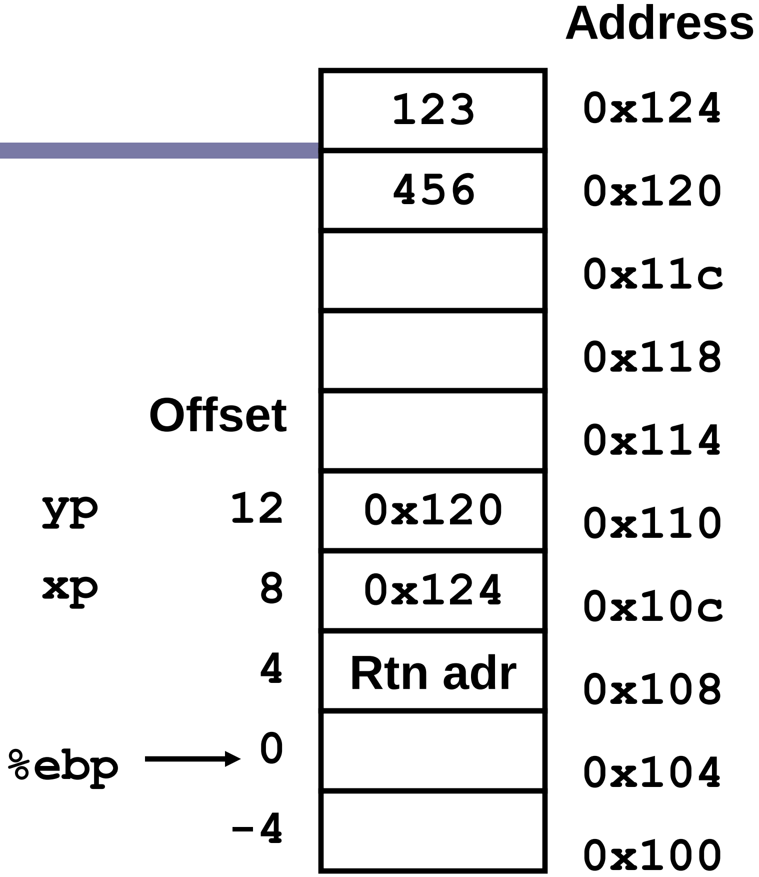
	Offset		Address
		123	0x124
		456	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp →	0	Old %ebp	0x104
	-4	Old %ebx	0x100

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

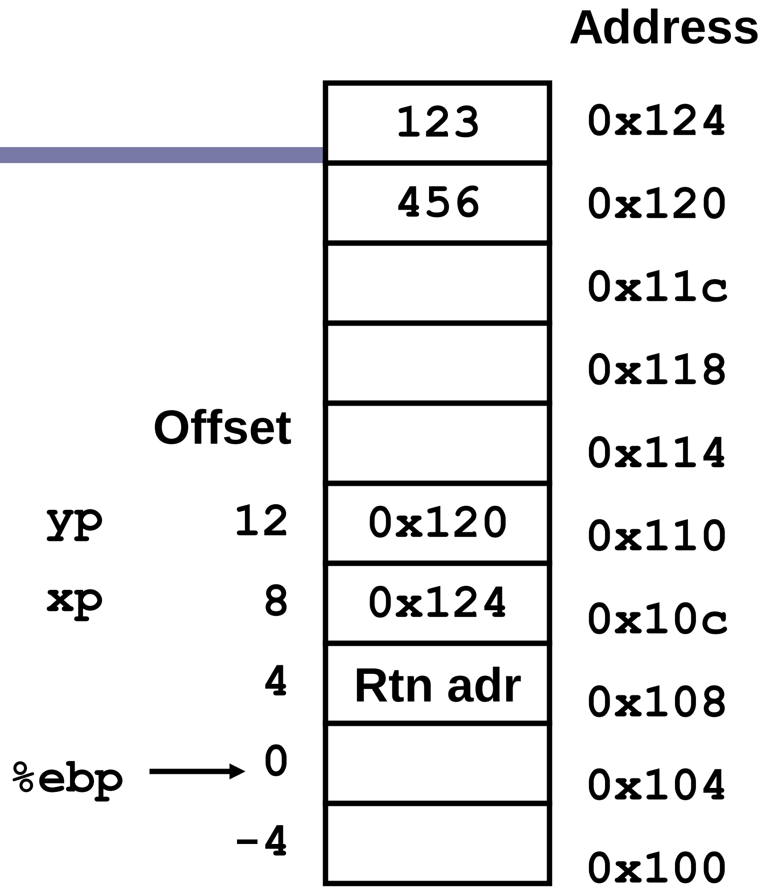
%eax	
%edx	
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

	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

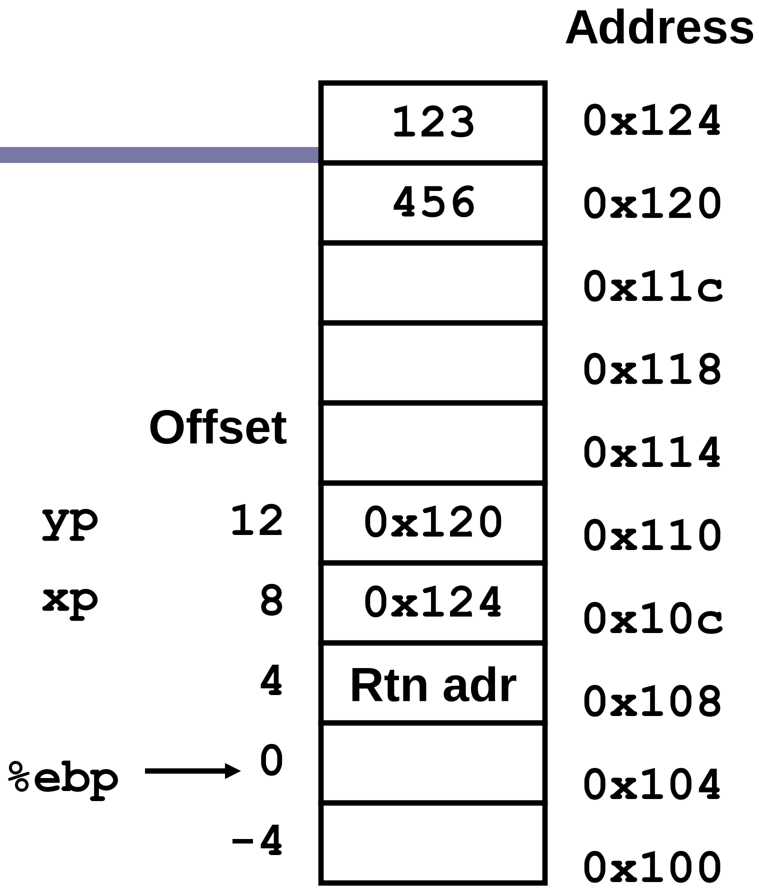
%eax	456
%edx	0x124
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

	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

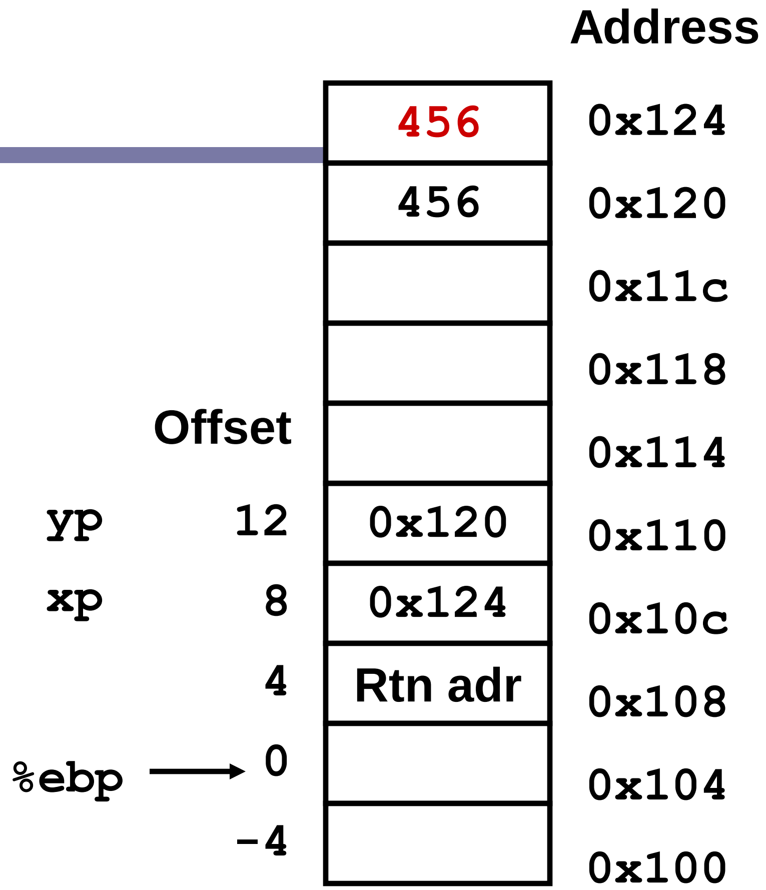


	Offset		Address
		123	0x124
		456	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp →	0		0x104
	-4		0x100

```
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
```

A second example

```
void decode1(int *xp, int *yp, int *zp);
```

xp at %ebp+8, yp at %ebp+12, zp at %ebp+16

```
movl 8(%ebp),%edi     Get xp
movl 12(%ebp),%edx    Get yp
movl 16(%ebp),%ecx    Get zp
movl (%edx),%ebx      Get y
movl (%ecx),%esi      Get z
movl (%edi),%eax      Get x
movl %eax,(%edx)      Store x at yp
movl %ebx,(%ecx)      Store y at zp
movl %esi,(%edi)      Store z at xp
```

```
void decode(int *xp,
            int *yp,
            int *zp)
{
  int tx = *xp;
  int ty = *yp;
  int tz = *zp;

  *yp = tx;
  *zp = ty;
  *xp = tz;
}
```

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

Some arithmetic operations

- One operand instructions

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

Some arithmetic operations

- Two operand instructions

Instruction	Effect	Description
`addl` S,D	D ← D + S	Add
`subl` S,D	D ← D – S	Substract
`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

- Shifts

Instruction	Effect	Description
`sall` k,D	D ← D << k	Left shift
`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

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
```

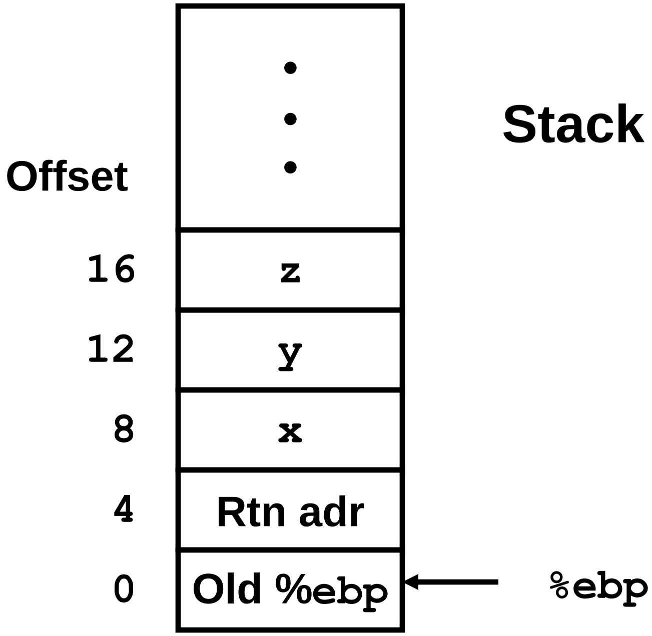
Body

```
    leave
    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)
```

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

    leave                         Finish
     ret
```

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

Next time …

- Breaking with the sequence … control
  - Condition codes
  - Conditional branches
  - Loops
  - Switch