

Last Time: Floating Point

- IEEE FP is all you need to care about
- Encoding:
 - sign bit + exponent + significand
 - Interpretation
- Lots of kinds of values:
 - NaN, infinities, normalized, denormalized, +/- zero
- Almost everything is approximate
 - HOW approximate often depends on you
- Floats are not reals

Today:

Machine-Level Programming 1

- **x86 Assembly Programmer's Execution Model**
- **Accessing Information**
 - Registers
 - Memory
- **Arithmetic operations**
- **Tools for manipulating and translating programs**
 - In Lab 2 you will become closely acquainted with these tools

Why Assembly?

- **It's ugly and you'll never write it most likely**
- **BUT**
 - key to understanding the compiler and the system interface
 - hence key to understanding performance issues
 - another piece of debugging evidence
 - some systems programs will need “embedded assembly”
 - » to say things you can't say in C or C++
- **Hence the big shift from writing assembly to reading**
 - These are very different skills
 - Generating good assembly is the compilers job
 - » lots of reasons: productivity, maintainability,

IA32 Processors

- **Totally dominate the desktop / server / workstation computer markets**
 - Transmeta Crusoe, IBM PowerPC, etc. are small players
- **Evolutionary Design**
 - Starting in 1978 with 8086
 - Added more features as time goes on
 - Still support old features, although seriously obsolete
- **Complex Instruction Set Computer (CISC)**
 - Many different instructions with many different formats
 - » But, only small subset encountered with Linux programs
 - » In fact, Intel tells compiler writers to generate RISC-like code
 - Hard to match performance of Reduced Instruction Set Computers (RISC)
 - But, Intel has done just that!

X86 Evolution: Programmer's View

- | Name | Date | Transistors |
|--|-------------|-------------|
| 8086 | 1978 | 29K |
| ▪ 16-bit processor. Basis for IBM PC & DOS | | |
| ▪ Limited to 1MB address space. DOS only gives you 640K | | |
| 80286 | 1982 | 134K |
| ▪ Added elaborate, but not very useful, segmented memory model | | |
| ▪ Basis for IBM PC-AT and Windows | | |
| 386 | 1985 | 275K |
| ▪ Extended to 32 bits. Added “flat addressing” | | |
| ▪ Capable of running modern Unix versions | | |
| ▪ Linux/gcc uses few instructions introduced post-386 | | |
| 486 | 1989 | 1.9M |
| Pentium | 1993 | 3.1M |
| ▪ Big change in microarchitecture | | |

X86 Evolution: Programmer's View

- | Name | Date | Transistors |
|--------------------|-------------|-------------|
| Pentium Pro | 1995 | 5.5M |

 - Big change in underlying microarchitecture
- | | | |
|-------------------|-------------|-------------|
| Pentium II | 1997 | 7.0M |
|-------------------|-------------|-------------|

 - Added conditional move instructions
- | | | |
|--------------------|-------------|-------------|
| Pentium III | 1999 | 8.2M |
|--------------------|-------------|-------------|

 - Added “streaming SIMD” instructions for operating on 128-bit vectors of 1, 2, or 4 byte integer or floating point data
- | | | |
|------------------|-------------|------------|
| Pentium 4 | 2001 | 42M |
|------------------|-------------|------------|

 - Another change in microarchitecture
 - Added 8-byte formats and 144 new instructions for streaming SIMD mode

x86 Clones: AMD

- **Historically**

- AMD has followed just behind Intel
- a little bit slower and a lot cheaper

- **More recently**

- recruited many top designers from the failed DEC
- exploited Intel's IA64, IXP12000, SA1100, distractions
- competition is now very close
 - » **AMD is actually winning in several sectors**
 - » **not however in profit**
- **developed their own 64 bit extension: Athlon 64**
 - » **permits larger virtual address space**
 - » **Some Linux distributions support this already**

New Species: IA64

- **Name Date Transistors**
- **Itanium 2000 10M**
 - A 64-bit architecture
 - Radically new instruction set designed for high performance
 - Can run existing IA32 programs
 - » On-board “x86 engine”
 - Joint project with HP (evolved from the PA-WW project @ HPL)
- **Itanium2 2002 22M**
 - Big performance boost
 - » essential since Merced (aka Itanium 1 was a dog)
 - » McKinley designed at HP facility for Intel process

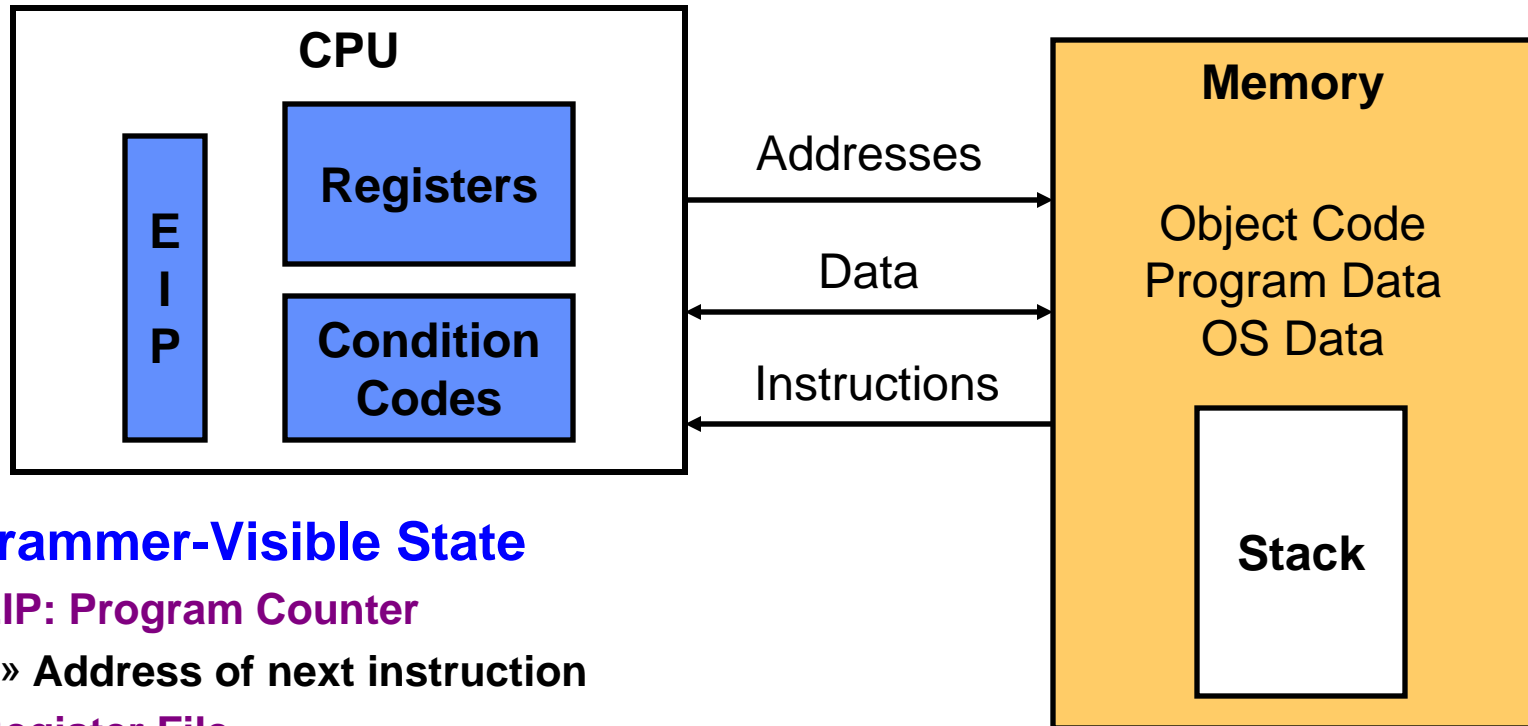
x86 Legacy

- **x86 started life as a 16-bit device**
 - hence word = 16 bits
 - Registers AX, BX, DX, etc.
- **x86 \geq 3 has 32-bit registers**
 - EAX, EBX, EDX, etc.
 - “E” means extended
- **Even on a Pentium 4, you can run in 8086 mode and use AX, BX, etc.**
 - In fact, these things boot up in 8086 mode and it's up to the Windows or Linux bootloader to make the jump into 32-bit mode
 - » In the Linux kernel see `arch/i386/boot/setup.S`
 - Makes life painful for low-level hackers...

gcc

- **gcc targets a ton of architectures**
 - Very general compiler design
 - Individual gcc ports tend to not be top-notch
- **Modern machines are difficult targets for compilers**
 - Very dynamic and speculative
 - Branch predictors, lots of caches, etc.
- **gcc basically targets a 386**
 - Performance falls short of code generated by Intel and Microsoft compilers
 - Doesn't make good use of MMX, SSE, SSE2

Assembly Programmer's View



• Programmer-Visible State

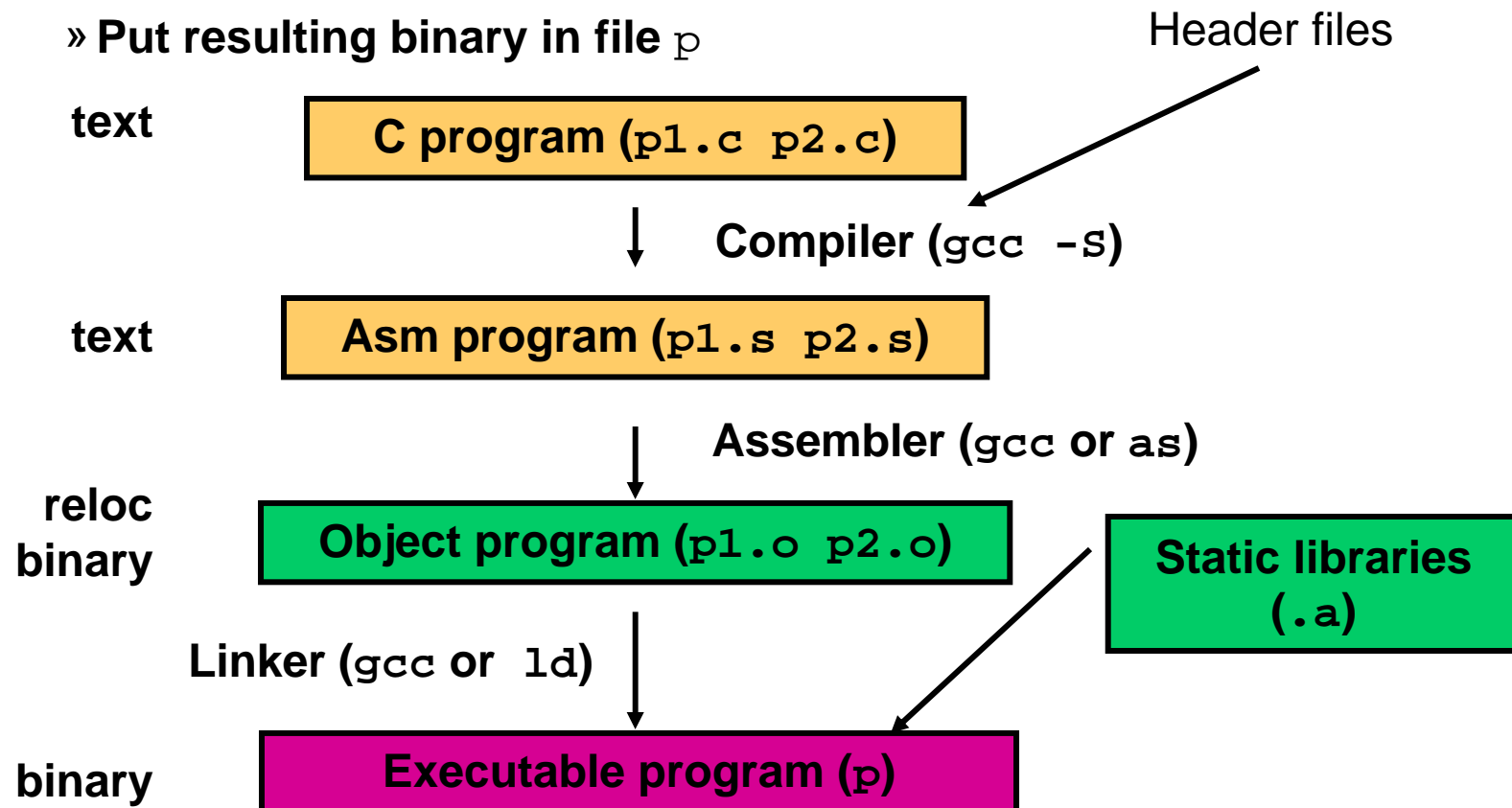
- **EIP: Program Counter**
 - » Address of next instruction
- **Register File**
 - » Heavily used program data
- **Condition Codes**
 - » Store status information about most recent arithmetic operation
 - » Used for conditional branching

▪ 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 -O p1.c p2.c -o p`
 - » Use optimizations (`-O`)
 - » Put resulting binary in file `p`



A Note on File Extensions

- **GCC has some standards**
 - you can name things in a non-standard way
 - DON'T – it's bad style
- **C Conventions**
 - .c are source files → text
 - .i are preprocessed files → text
 - .s are assembly files → text
 - .o are relocate-able object files → binary
 - .a are archive files – e.g. static libraries → binary
 - Files with no extension are often executable → binary
- **Getting the output you want**
 - gcc foo.c → a.out executable
 - gcc -o foo foo.c → foo executable
 - man gcc is your friend – there are about a zillion options

A Note on Preprocessing

- **Compiler or cpp → .i file**
 - **What really happens?**
 - » bring in any of the #include files
 - » expand macros
 - » provide line #'s
- **Why?**
 - get everything you need in one place
 - macros make up for poor language
 - » No concept of software modules or ADTs
 - » Const variables weak vs. literal constants
 - » Historically unreliable support for inline functions

Compiling Into Assembly

• C Code

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

Obtain with command

```
gcc -O -S code.c
```

Produces file `code.s`

`-O` optimizes lightly

Generated Assembly

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

labels, opcodes, operands

note opcode suffix `l` = long = 32b

`%xxx` → x86 register names
more on this soon

Assembly Characteristics

- **Minimalist Data Types**

- “Integer” data of 1, 2, or 4 bytes
 - » Data values
 - » Addresses (untyped pointers)
- Floating point data of 4, 8, or 10 bytes
- No aggregate types such as arrays or structures
 - » It’s all just bits

- **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 (call / return)
 - » Conditional branches
- Special-purpose: I/O, processor control, OS support, etc.

Object Code

Code for sum

0x401040 <sum>:

0x55

0x89

0xe5

0x8b

0x45

0x0c

0x03

0x45

0x08

0x89

0xec

0x5d

0xc3

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

• Assembler

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

• Linker

- Resolves references between files
- Combines with static run-time libraries
 - » E.g., code for `malloc`, `printf`
- Some libraries are *dynamically linked* (e.g. `.dll`'s in Windows)
 - » Linking occurs at program startup time

Machine Instruction Example

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

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

Similar to
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

- **Type this:**

```
objdump -d p.o
```

- **Output:**

```
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
```

- Shows you what the OS + hardware sees
- Analyzes bit pattern of series of instructions
- Produces approximate rendition of assembly code
- Can be run on either a .out (linked executable) or .o file

More GAS, Disassembly, .o etc.

- **Difference between GAS and objdump**

- **Objdump shows the actual instructions as well as the assembly version**

- **Naming convention differences**

- » **GAS uses l, w, b suffixes (long, word, byte)**

- typically you'll find opcodes for all 3 – e.g. MOVL, MOVW, MOVB

- l also used to mean double float – ambiguity resolved by use of different regs

- » **objdump doesn't use the suffixes (why?)**

- **Disassembly of linked vs. unlinked files**

- **unlinked**

- » **you'll still see symbolic references between files and globals**

- **linked**

- » **they just become addresses since all references are resolved**

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**

`gdb p`

`disassemble sum`

- Disassemble procedure

`x/13b sum` (`x/13xb sum` → 13hex bytes)

- Examine the 13 bytes starting at `sum`

What Can be Disassembled?

```
% objdump -d WINWORD.EXE
```

```
WINWORD.EXE:      file format pei-i386
```

```
No symbols in "WINWORD.EXE".
```

```
Disassembly of section .text:
```

```
30001000 <.text>:
```

```
30001000:  55                push    %ebp
30001001:  8b ec            mov     %esp,%ebp
30001003:  6a ff            push    $0xffffffff
30001005:  68 90 10 00 30   push   $0x30001090
3000100a:  68 91 dc 4c 30   push   $0x304cdc91
```

- Anything that can be interpreted as executable code
- Disassembler examines bytes and reconstructs assembly source
 - » Impossible in theory (reduces to halting problem)
 - » Easy in practice unless it's a virus you're disassembling

Moving Data

- **Moving Data**

`movl Source, Dest`: Move 4-byte (“long”) word

- Accounts for 31% of all instructions in sample
 - » why?

- **Operand Types**

- **Immediate: Constant integer data**
 - » Like C constant, but prefixed with ‘\$’
 - » E.g., \$0x400, \$-533
 - » Encoded with 1, 2, or 4 bytes
- **Register: One of 8 integer registers**
 - » But `%esp` and `%ebp` reserved for special use
 - » Others have special uses for particular instructions
 - RISC and compiler types now say “GRRRroan”
- **Memory: 4 consecutive bytes of memory**
 - » Various “addressing modes”

x86 Int Registers

<code>%eax</code>
<code>%edx</code>
<code>%ecx</code>
<code>%ebx</code>
<code>%esi</code>
<code>%edi</code>
<code>%esp</code>
<code>%ebp</code>

Integer Register File Stuff

- **Low order words**

- legacy from 16-bit days when macho programmers wrote in asm
 - » names live on
- `%ax, %cx, %dx, %bx, %si, %di, %sp, %bp`

- **For the low order word**

- available in the a,c,d, and b regs
 - » can access the high or low order byte
 - e.g. for `%ax ::= %ah : %al`

- **Conventions**

- `%eax` gets the return value from a function call
- `%ebp` points to the base of the call parameters
 - » parameters indexed off this “frame pointer”
- `%esp` is the stack pointer
- for special 64-bit arithmetic
 - » `%edx : %eax` are viewed as a single 64 bit register pair

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

Assembler Gotcha

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 hexadecimal 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)

Simple Addressing Modes

- **Immediate \$10**

- Source value is a constant

```
movl $10,eax
```

- **Indirect (R) Mem[Reg[R]]**

- Register R specifies memory address

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

- **Displacement D(R) Mem[Reg[R]+D]**

- Register R specifies start of memory region
- Constant displacement D specifies offset

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

Using Simple Addressing Modes

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

illustrates the calling conventions

swap:

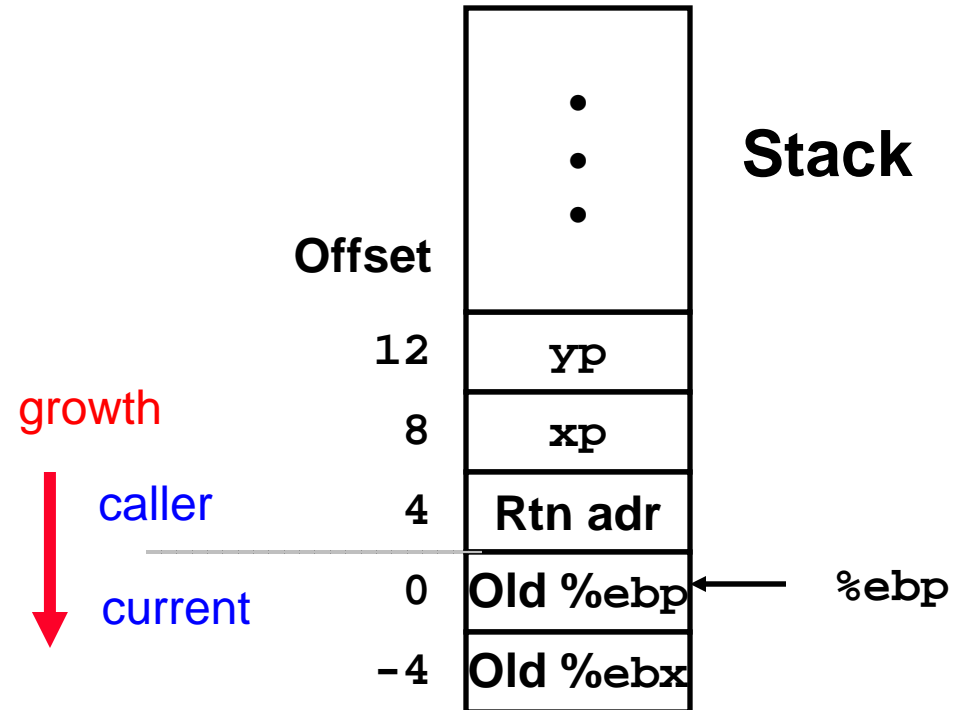
```
    pushl %ebp          }
    movl  %esp,%ebp    } Set Up
    pushl %ebx

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



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

		Address	
		123	0x124
		456	0x120
			0x11c
			0x118
	Offset		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

		Address	
		123	0x124
		456	0x120
			0x11c
			0x118
	Offset		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

		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

		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

		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

		Address	
		456	0x124
		123	0x120
			0x11c
			0x118
	Offset		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

```

Indexed Addressing Modes

- **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 8 integer registers
- **Ri:** Index register: Any, except for `%esp`
 - » **Unlikely you’d use `%ebp`, either**
- **S:** Scale: 1, 2, 4, or 8

- **RISC machines don’t support modes like this**

More Indexing Mode Cases

- **Special Cases**

(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

<code>%edx</code>	<code>0xf000</code>
<code>%ecx</code>	<code>0x100</code>

Expression	Computation	Address
<code>0x8(%edx)</code>	<code>0xf000 + 0x8</code>	<code>0xf008</code>
<code>(%edx,%ecx)</code>	<code>0xf000 + 0x100</code>	<code>0xf100</code>
<code>(%edx,%ecx,4)</code>	<code>0xf000 + 4*0x100</code>	<code>0xf400</code>
<code>0x80(,%edx,2)</code>	<code>2*0xf000 + 0x80</code>	<code>0x1e080</code>

Address Computation Instruction

- **leal *Src, Dest***

- “load effective address”
- *Src* is address mode expression
- Set *Dest* to address denoted by expression

- **Uses**

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

Some Arithmetic Operations

Format

Computation

- **Two Operand Instructions**

`addl Src, Dest`

$Dest = Dest + Src$

`subl Src, Dest`

$Dest = Dest - Src$

`imull Src, Dest`

$Dest = Dest * Src$

`sall Src, Dest`

$Dest = Dest \ll Src$

Also called `shll`

`sarl Src, Dest`

$Dest = Dest \gg Src$

Arithmetic

`shrl Src, Dest`

$Dest = Dest \gg Src$

Logical

`xorl Src, Dest`

$Dest = Dest \wedge Src$

`andl Src, Dest`

$Dest = Dest \& Src$

`orl Src, Dest`

$Dest = Dest | Src$

More Arithmetic

- One Operand Instructions

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
```

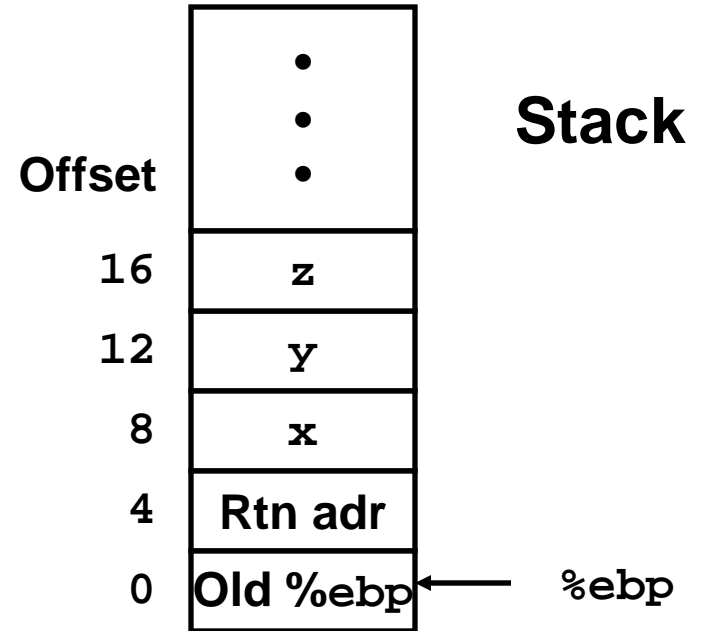
} Body

```
    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
    movl %esp,%ebp
```

} Set Up

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

} Body

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

} Finish

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

```
    eax = x
    eax = x^y      (t1)
    eax = t1>>17  (t2)
    eax = t2 & 8185
```

CISC Properties

- **Instruction can reference different operand types**
 - Immediate, register, memory
- **Arithmetic operations can read/write memory**
 - As opposed to RISC arithmetic ops that only access registers
- **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

Putting This Stuff Into Practice

- **Lab 2 comes out on Thursday**
 - gdb practice
 - disassembly practice
 - should be “fun”