Machine-Level Representation

CSCI 2021: Machine Architecture and Organization

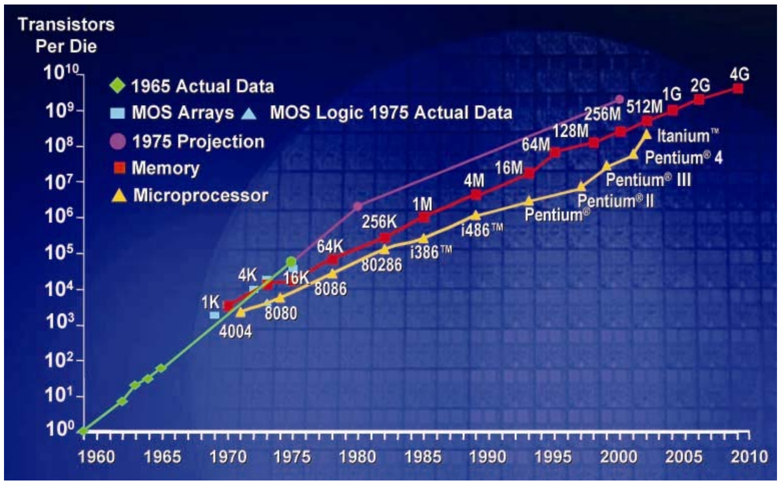
Antonia Zhai
Department Computer Science and Engineering
University of Minnesota
http://www.cs.umn.edu/~zhai

With Slides from Bryant and O'Hallaron

University of Minnesota

Detour: A short history of PC

Architecture design is driven by Moore's law



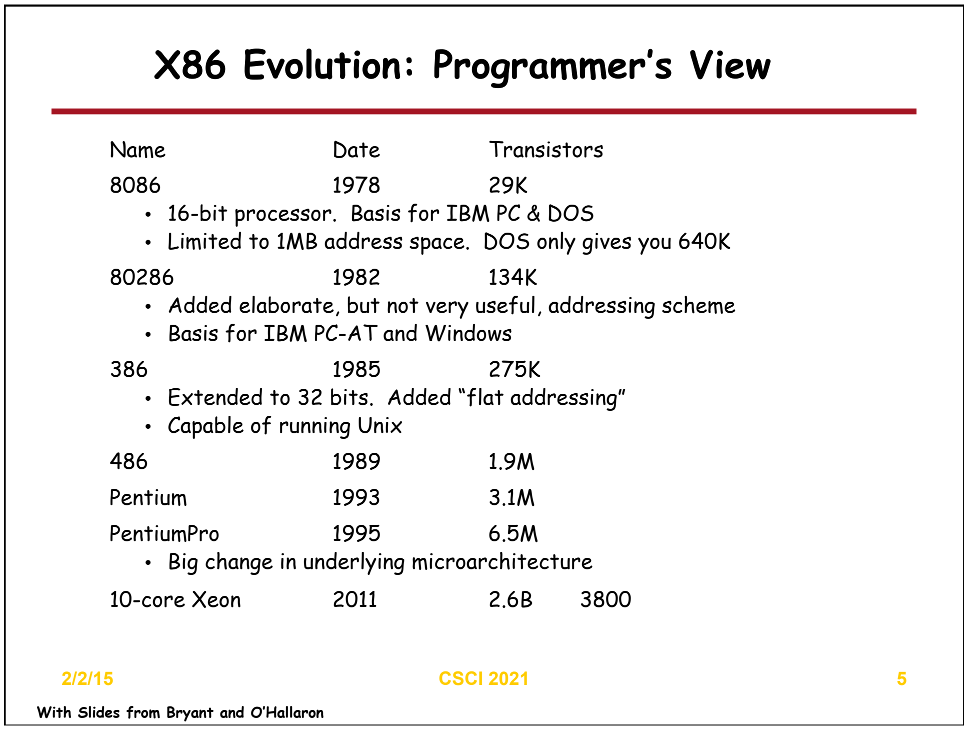
Intel Processors

Evolutionary Design

- Starting in 1978 with 8086
- Added more features as time goes on
- Still support old features, although obsolete

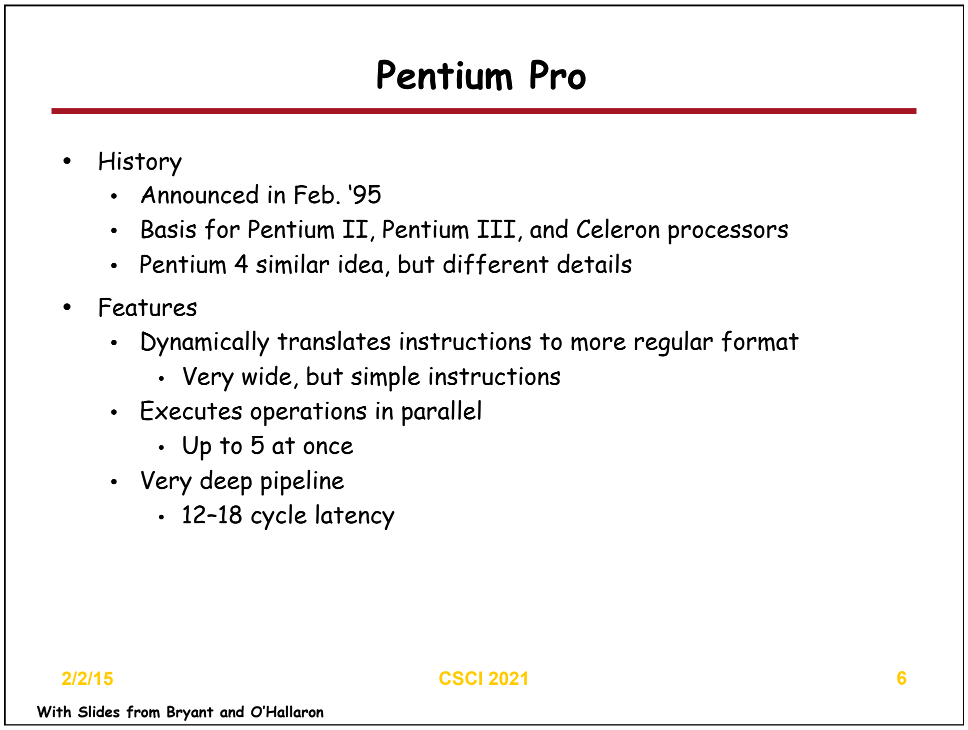
Complex Instruction Set Computer (CISC)

- 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: 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, addressing scheme			
• Basis for IBM PC-AT and Windows			
386	1985	275K	
• Extended to 32 bits. Added "flat addressing"			
• Capable of running Unix			
486	1989	1.9M	
Pentium	1993	3.1M	
PentiumPro	1995	6.5M	
• Big change in underlying microarchitecture			
10-core Xeon	2011	2.6B	3800



Pentium Pro

- History
  - Announced in Feb. '95
  - Basis for Pentium II, Pentium III, and Celeron processors
  - Pentium 4 similar idea, but different details
- Features
  - Dynamically translates instructions to more regular format
    - Very wide, but simple instructions
  - Executes operations in parallel
    - Up to 5 at once
  - Very deep pipeline
    - 12–18 cycle latency

X86 Evolution: Clones

- Advanced Micro Devices (AMD)
  - Historically
    - AMD has followed just behind Intel
    - A little bit slower, a lot cheaper
  - Recently
    - Recruited top circuit designers from Digital Equipment Corp.
    - Exploited the fact that Intel distracted by IA64
    - Now are close competitors to Intel
  - Developing own extension to 64 bits

X86 Evolution: Clones

- Transmeta
  - Recent start-up
  - Radically different approach to implementation
    - Translates x86 code into "Very Long Instruction Word" (VLIW) code
    - High degree of parallelism
  - Shooting for low-power market

Road to IA64

Name	Date	Transistors
Itanium	2001	10M

- A 64-bit architecture
- Radically new instruction set designed for high performance
- Will be able to run existing IA32 programs
  - On-board "x86 engine"
- Joint project with Hewlett-Packard

Name	Date	Transistors
• Itanium 2	2002	221M

- Big performance boost

x86 Clones: Advanced Micro Devices (AMD)

- Historically
  - AMD has followed just behind Intel
  - A little bit slower, a lot cheaper
- Then
  - Recruited top circuit designers from Digital Equipment Corp. and other downward trending companies
  - Built Opteron: tough competitor to Pentium 4
  - Developed x86-64, their own extension to 64 bits

Intel's 64-Bit

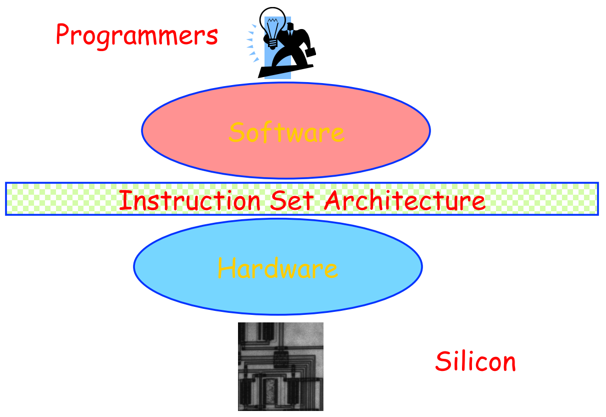
- Intel Attempted Radical Shift from IA32 to IA64
  - Totally different architecture (Itanium)
  - Executes IA32 code only as legacy
  - Performance disappointing
- AMD Stepped in with Evolutionary Solution
  - x86-64 (now called "AMD64")
- Intel Felt Obligated to Focus on IA64
  - Hard to admit mistake or that AMD is better
- 2004: Intel Announces EM64T extension to IA32
  - Extended Memory 64-bit Technology
  - Almost identical to x86-64!
- All but low-end x86 processors support x86-64
  - But, lots of code still runs in 32-bit mode

Definitions

- Architecture: (also instruction set architecture: ISA) The parts of a processor design that one needs to understand to write assembly code.
  - Examples: instruction set specification, registers.
- Microarchitecture: Implementation of the architecture.
  - Examples: cache sizes and core frequency.
- Example ISAs (Intel): x86, IA, IPF

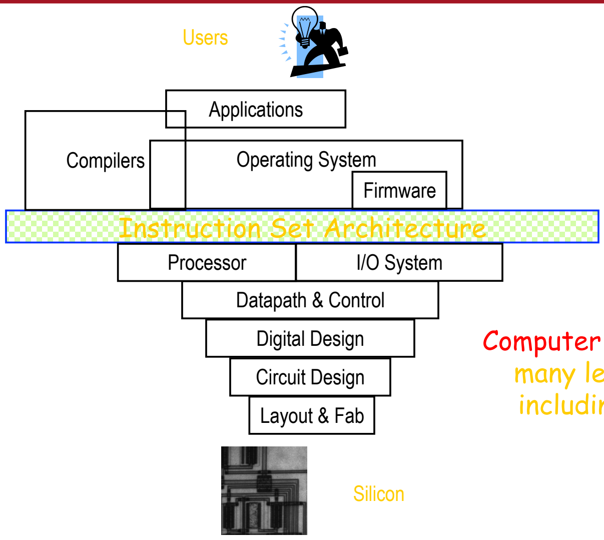
What is computer architecture?

ISA is a contract between the software and the hardware



With Slides from Bryant and O'Hallaron

What do we study in computer Architecture?



Computer architecture: Cover many levels of abstractions including ISA, organization and hardware

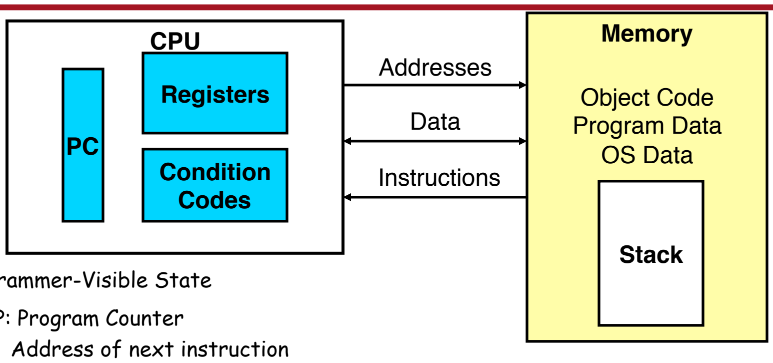
With Slides from Bryant and O'Hallaron

This Course

- IA32
  - The traditional x86
- x86-64
  - The new standard
- Book
  - Sections 3.1—3.12: IA32
  - Section 3.13: x86-64
- This class
  - Mostly IA32

Assembly Language Details

Assembly Programmer's View



Programmer-Visible State

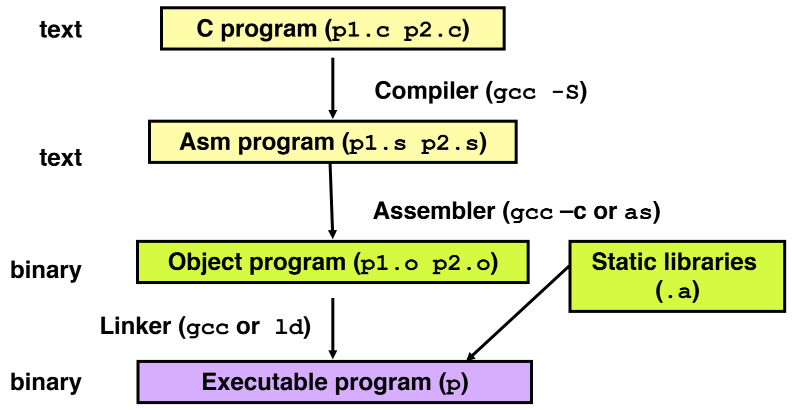
- EIP: Program Counter
  - Address of next instruction
- Register File
  - Heavily used program data
- Condition Codes
  - Store status information about most recent arithmetic operation

Memory: Byte addressable array

- Code, user data, (some) OS data
- Includes stack

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 (`-o`) `p`



C to Assembly

C Code

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

Generated IA32 **Assembly**

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

Obtain with command

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

Produces file `code.s`

Some compilers use instruction "`leave`"

Assembly

Minimal 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
  - 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 for sum

```
0x000000 <sum>:
     0x55
     0x89
     0xe5
     0x8b
     0x45
     0x0c
     0x03
     0x45
     0x08
     0x5d
     0xc3
```

Assembler

```
code.o:     file format elf32-i386

Disassembly of section .text:

00000000 <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:   5d              pop    %ebp
   a:   c3              ret
```

ode

erent files

- Total of 11 by
- Each instruc
  2, or 3 bytes

execution

Machine Instruction Example

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

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

Similar to expression
`x += y`

```
6:   03 45 08
```

- C Code: Add two signed integers
- Assembly: Add 2 4-byte integers
  - Signed/unsigned? Same instruction
  - 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 `0x6`

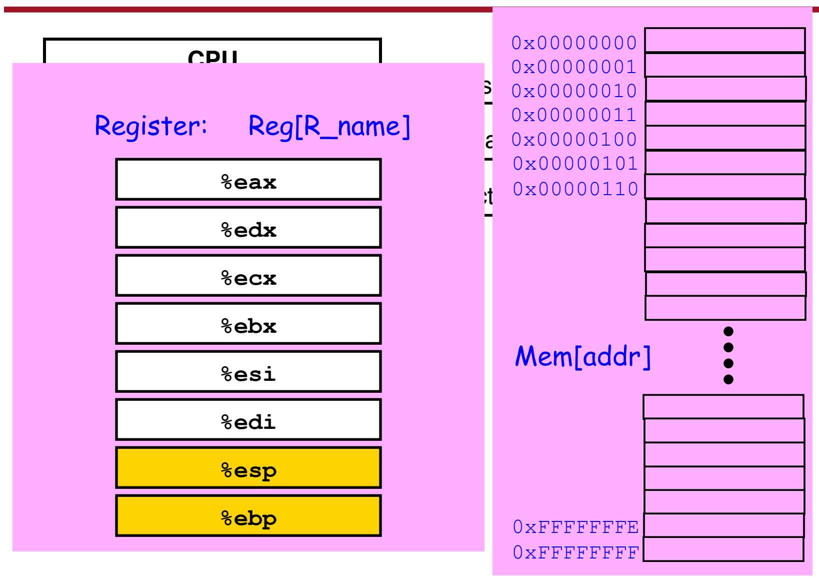
Disassemble

```
% objdump -d /soft/gcc-4.0.0/debian/bin/gcc
/soft/gcc-4.0.0/debian/bin/gcc:     file format elf32-i386
Disassembly of section .init:
08048e44 <_init>:
 8048e57:       e8 88 04 00 00          call   80492e4 <call_gmon_start>
 8048e5c:       e8 df 04 00 00          call   8049340 <frame_dummy>
 8048e61:       e8 5a e5 00 00          call   80573c0
<__do_global_ctors_aux>
 8048e66:       5b                      pop    %ebx
 8048e67:       c9                      leave
 8048e68:       c3                      ret
Disassembly of section .plt:

08048e6c <nl_langinfo@plt-0x10>:
 8048e6c:       ff 35 e0 d5 05 08       pushl  0x805d5e0
```

- Anything that can be interpreted as executable code
- Disassembler examines bytes and reconstructs assembly source

Machine Model



Data Access

Register:

- Just name the register: Ex. %eax
- For the rest of this class, Reg[R] ➔ the value in the register R

Memory:

- Normal (R) Mem[Reg[R]]
  - Register R specifies memory address

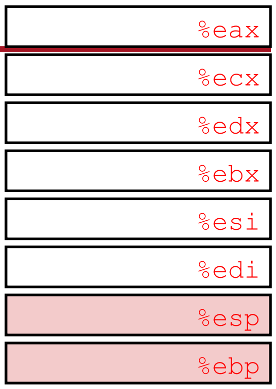
```
movl (%ecx),%eax
```

- Displacement D(R) Mem[Reg[R]+D]
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

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

With Slides from Bryant and O'Hallaron

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"

With Slides from Bryant and O'Hallaron

`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

An Example

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

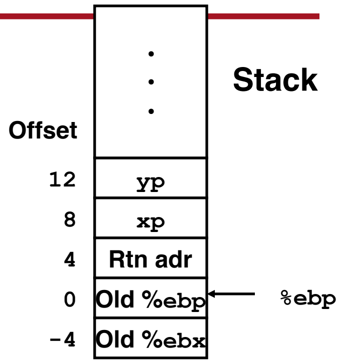
```
swap:
   pushl %ebp
   movl  %esp,%ebp           Set
   pushl %ebx                Up

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

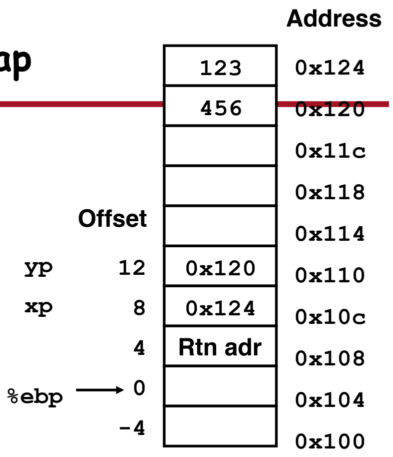


Register	Variable
%ecx	yp
%edx	xp
%eax	t1
%ebx	t0

```
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
```

With Slides from Bryant and O'Hallaron

Understanding Swap



%eax	
%edx	
%ecx	
%ebx	
%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
```

With Slides from Bryant and O'Hallaron

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 $\longrightarrow$	0		0x104
	-4		0x100

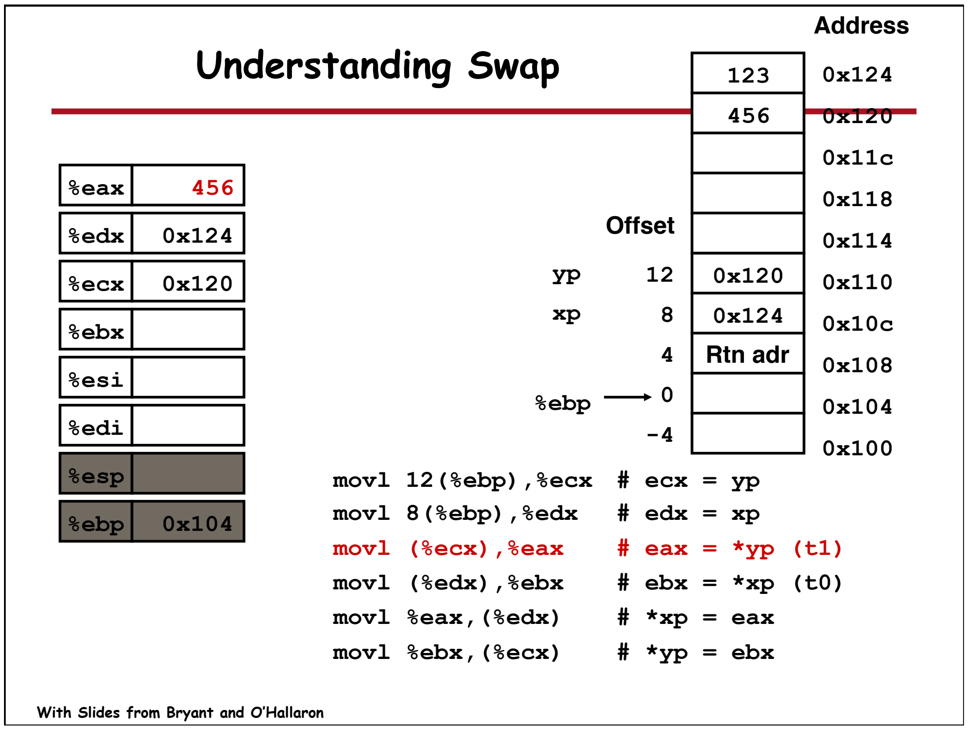
```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```

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 $\longrightarrow$	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
```



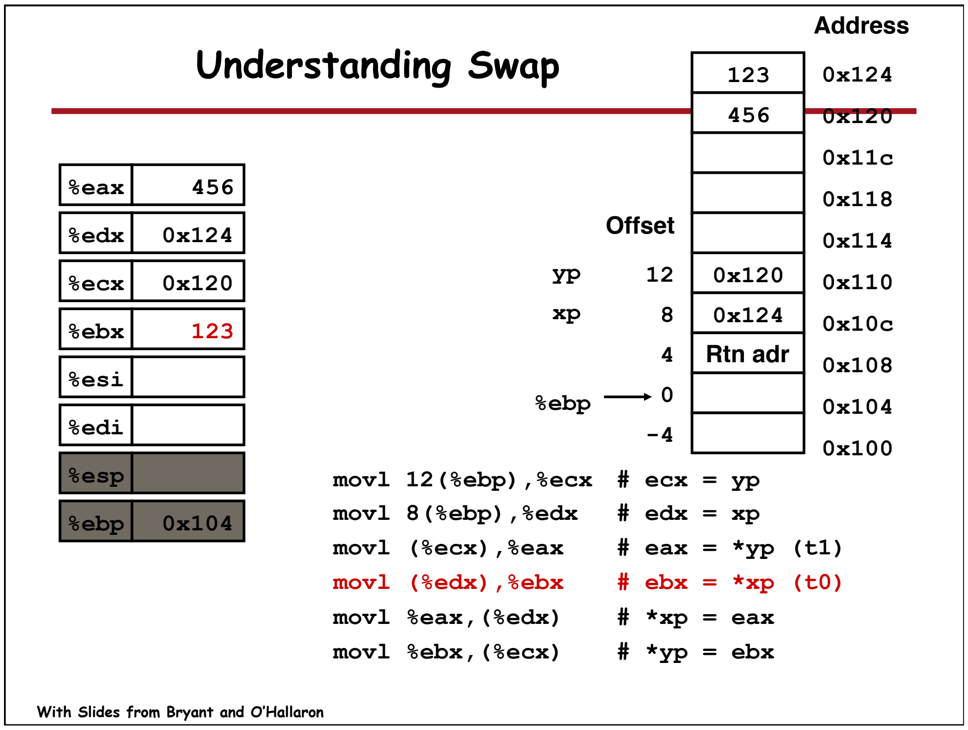
Understanding Swap

%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
```

With Slides from Bryant and O'Hallaron



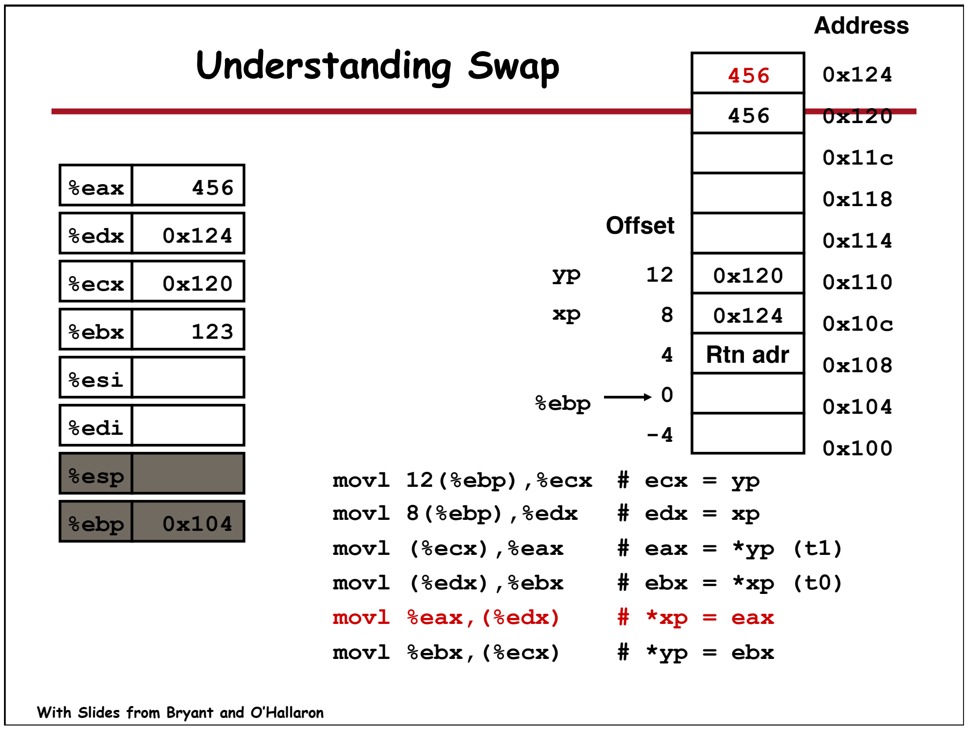
Understanding Swap

%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
```

With Slides from Bryant and O'Hallaron



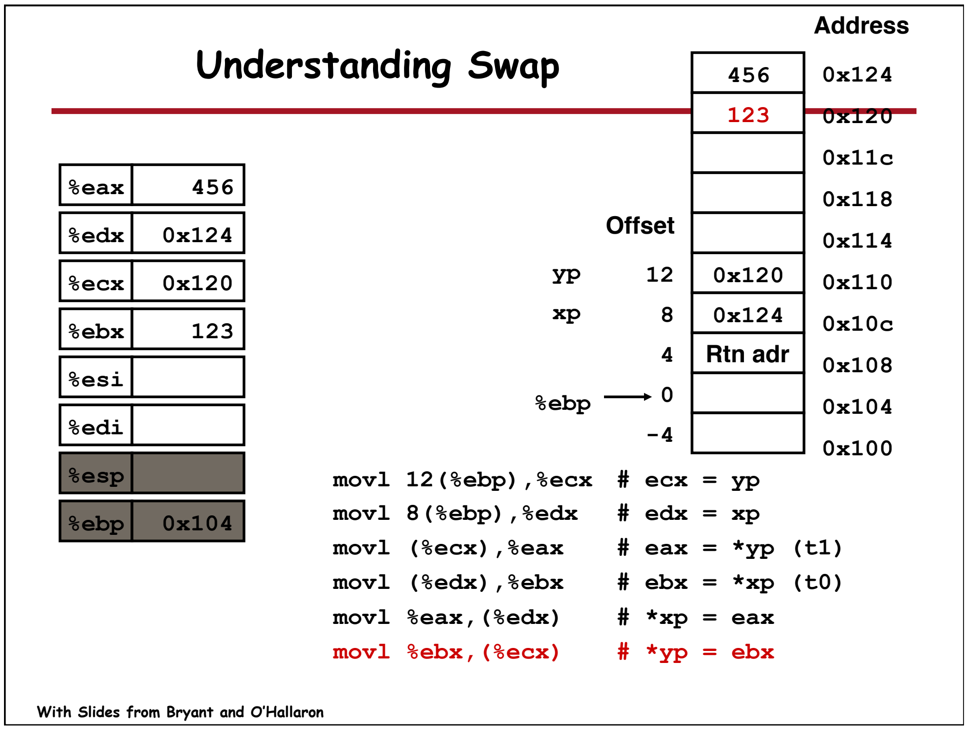
Understanding Swap

%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
```

With Slides from Bryant and O'Hallaron



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
```

With Slides from Bryant and O'Hallaron

More on 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

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]]

With Slides from Bryant and O'Hallaron

Address Computation Examples

`%edx`	`0xf000`
`%ecx`	`0x100`

Expression	Computation	Address
`0x8(%edx)`		
`(%edx,%ecx)`		
`(%edx,%ecx,4)`		
`0x80(,%edx,2)`		

Address Computation Instruction

- `leal` *Src*,*Dest*
  - *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, or 8.

With Slides from Bryant and O'Hallaron

Some Arithmetic Operations

- **Two Operand 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!**
- **No distinction between signed and unsigned int (why?)**

With Slides from Bryant and O'Hallaron

Some Arithmetic Operations

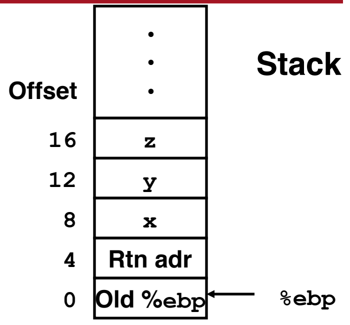
- One Operand Instructions

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

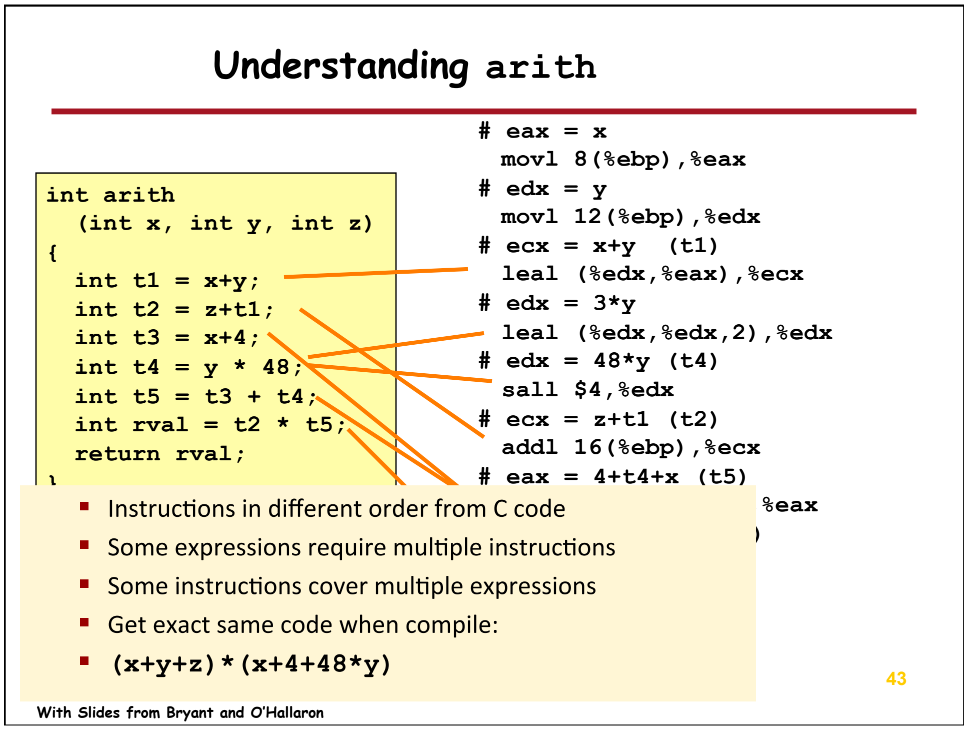
- See book for more instructions

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`

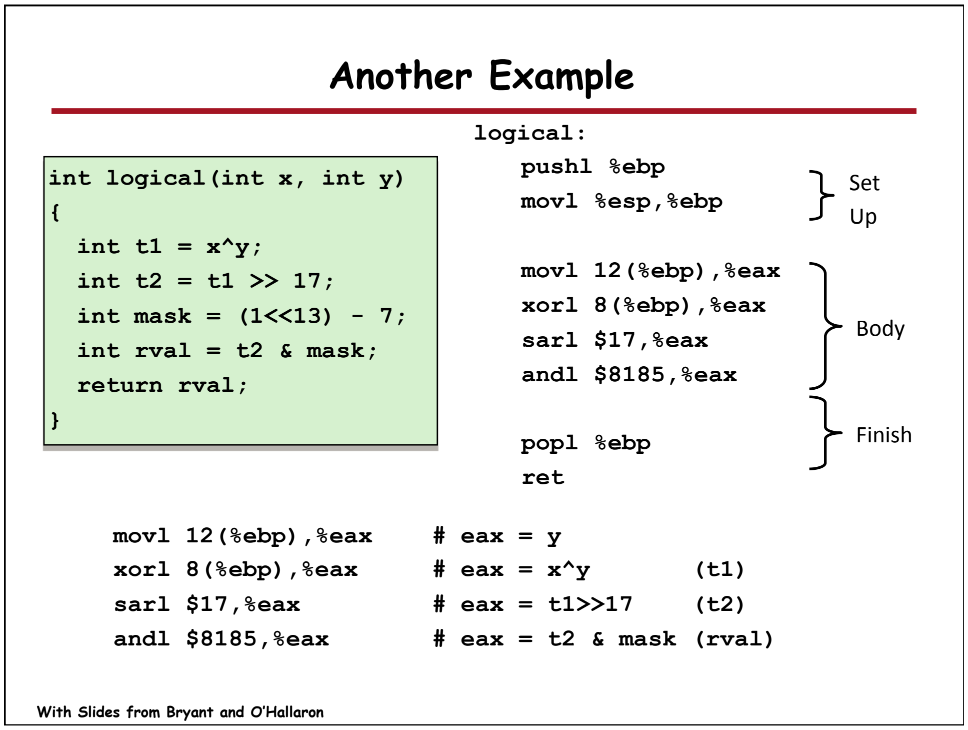
```
int arith
  (int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
```

```
# 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)
                          %eax
                         )
```

- 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)`

43

With Slides from Bryant and O'Hallaron



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;
}
```

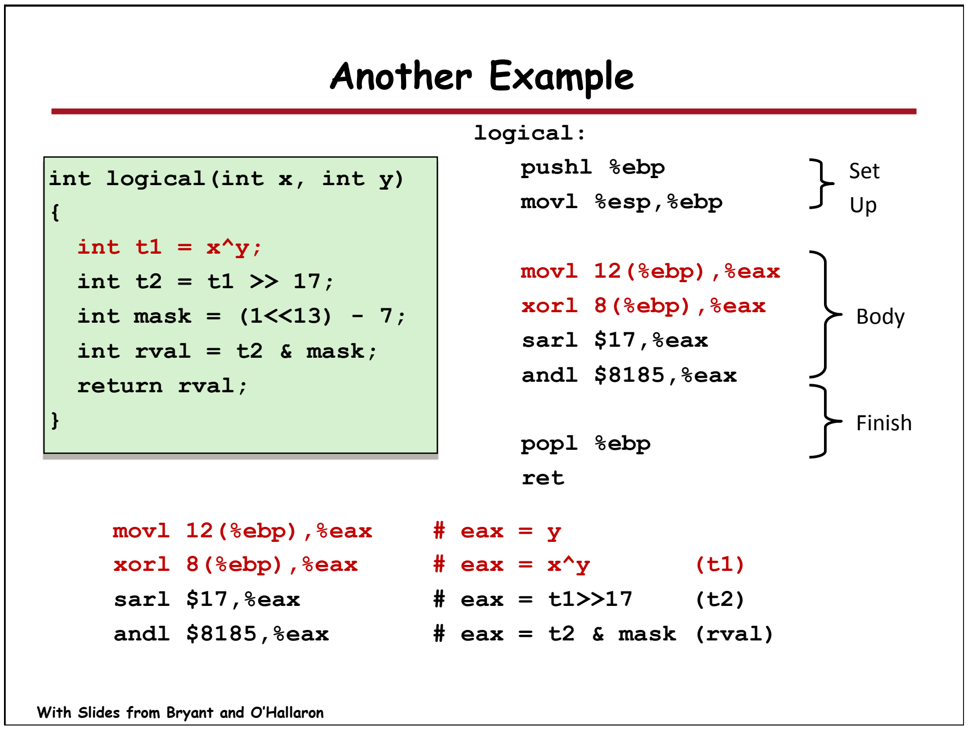
```
logical:
    pushl %ebp             } Set
    movl %esp,%ebp         } Up

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

    popl %ebp              } Finish
    ret                    }
```

```
movl 12(%ebp),%eax    # eax = y
xorl 8(%ebp),%eax     # eax = x^y       (t1)
sarl $17,%eax         # eax = t1>>17    (t2)
andl $8185,%eax       # eax = t2 & mask (rval)
```

With Slides from Bryant and O'Hallaron



Another Example

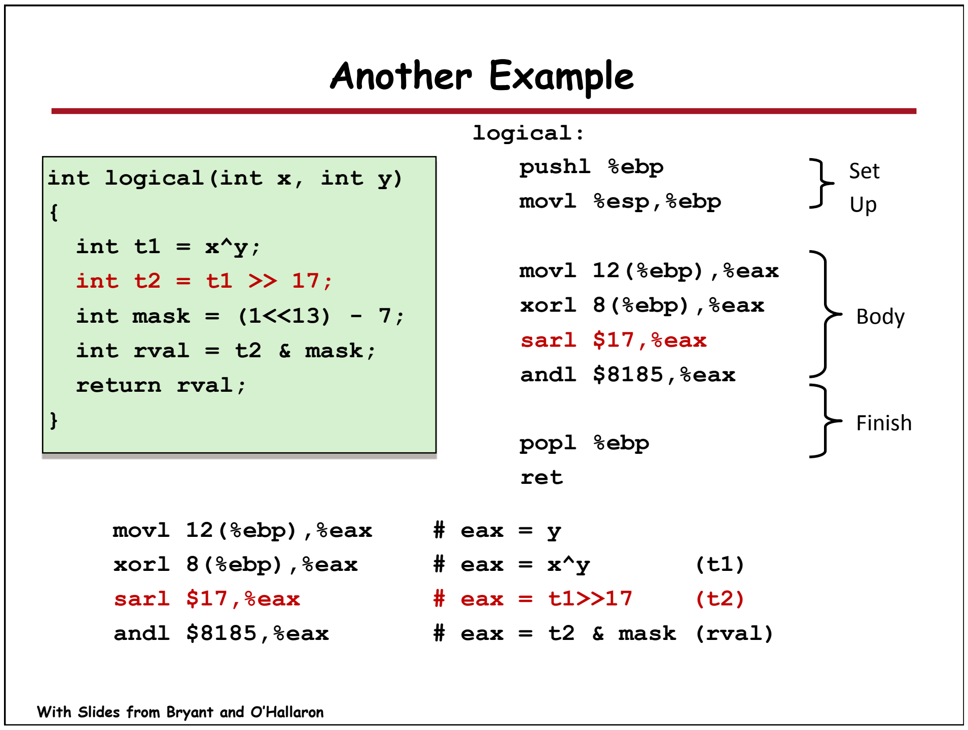
```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

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

    movl 12(%ebp),%eax       ⎫
    xorl 8(%ebp),%eax        ⎬ Body
    sarl $17,%eax            ⎪
    andl $8185,%eax          ⎭
                             ⎫
    popl %ebp                ⎬ Finish
    ret
```

```
movl 12(%ebp),%eax    # eax = y
xorl 8(%ebp),%eax     # eax = x^y        (t1)
sarl $17,%eax         # eax = t1>>17     (t2)
andl $8185,%eax       # eax = t2 & mask  (rval)
```

With Slides from Bryant and O'Hallaron



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 12(%ebp),%eax       ⎫
    xorl 8(%ebp),%eax        ⎬ Body
    sarl $17,%eax            ⎪
    andl $8185,%eax          ⎭
                             ⎫
    popl %ebp                ⎬ Finish
    ret
```

```
movl 12(%ebp),%eax    # eax = y
xorl 8(%ebp),%eax     # eax = x^y        (t1)
sarl $17,%eax         # eax = t1>>17     (t2)
andl $8185,%eax       # eax = t2 & mask  (rval)
```

With Slides from Bryant and O'Hallaron

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

    popl  %ebp              } Finish
    ret                     }
```

```
movl 12(%ebp),%eax     # eax = y
xorl 8(%ebp),%eax      # eax = x^y        (t1)
sarl $17,%eax          # eax = t1>>17     (t2)
andl $8185,%eax        # eax = t2 & mask  (rval)
```

X86-64 Assembly

Data Representations: IA32 + x86-64

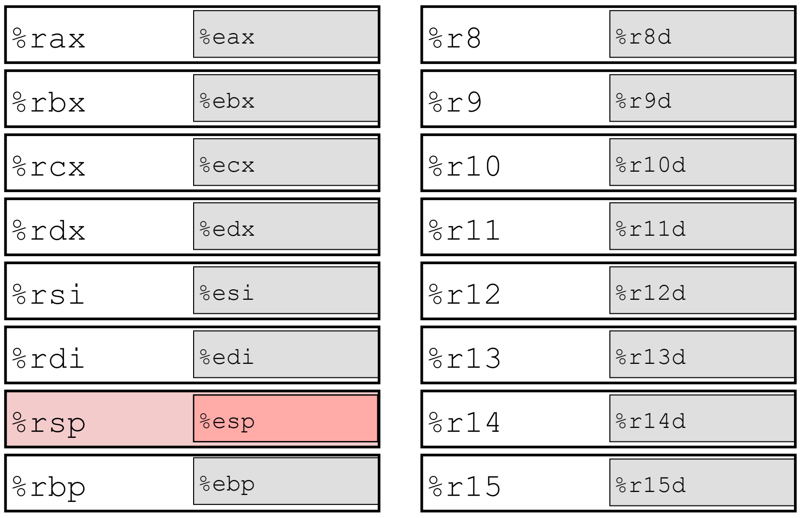
- Sizes of C Objects (in Bytes)

C Data Type	Generic 32-bit	Intel IA32	x86-64
unsigned	4	4	4
int	4	4	4
long int	4	4	8
char	1	1	1
short	2	2	2
float	4	4	4
double	8	8	8
long double	8	10/12	16
char *	4	4	8

*Or any other pointer

With Slides from Bryant and O'Hallaron

x86-64 Integer Registers



- Extend existing registers. Add 8 new ones.
- Make `%ebp`/`%rbp` general purpose

With Slides from Bryant and O'Hallaron

Instructions

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

- New instructions:
  - `movl` → `movq`
  - `addl` → `addq`
  - `sall` → `salq`
  - etc.

- 32-bit instructions that generate 32-bit results
  - Set higher order bits of destination register to 0
  - Example: `addl`

With Slides from Bryant and O'Hallaron

32-bit code for swap

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

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

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

   popl   %ebx
   popl   %ebp                     Finish
   ret
```

With Slides from Bryant and O'Hallaron

64-bit code for swap

```
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)


  ret
```

Set Up

Body

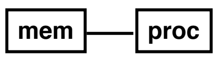
Finish

- Operands 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

Summary

Machine Models

C



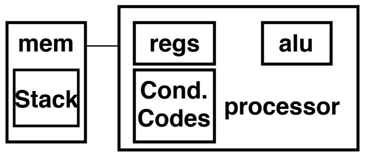
Data

1) `char`
2) `int`, `float`
3) `double`
4) `struct`, array
5) pointer

Control

1) loops
2) conditionals
3) `switch`
4) Proc. call
5) Proc. return

Assembly



1) byte
2) 2-byte word
3) 4-byte long word
4) contiguous byte allocation
5) address of initial byte

3) branch/jump
4) `call`
5) `ret`