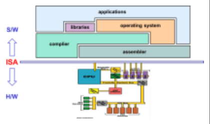
The HW/SW Interface

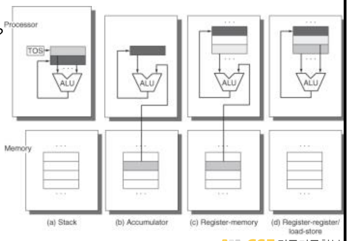
The x86 ISA: Machine-Level Programming Basics

```
0x08048394 <+0>:  push   %ebp
0x08048395 <+1>:  mov    %esp,%ebp
0x08048397 <+3>:  mov    0xc(%ebp),%eax
0x0804839a <+6>:  add    0x8(%ebp),%eax
0x0804839d <+9>:  pop    %ebp
0x0804839e <+10>: ret
0x8048394 <sum>:  0x55   0x89   0xe5 …
0x804839c <sum+8>:0x08   0x5d   0xc3
```

Instruction Set Architecture (ISA)

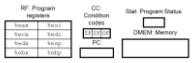
- ISA classification
  - RISC vs CISC
  - general-purpose
    - two or three operands?
    - # of memory operands?
    - classification
      - load-store
      - register-memory
      - memory-memory
  - stack
  - accumulator





Instruction Set Architecture (ISA)

- The ISA defines the **programmer-visible state** of a processor architecture



  - the ISA tells us what we can expect the processor to do when we execute instructions (but not *how* the processor does it)
  - memory addressing
    - interpretation of addresses
      - bytes, half words, words, double words, …?
    - byte ordering in memory
      - little-endian vs. big-endian
    - addressing modes
      - how to access objects in memory

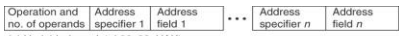
Instruction Set Architecture (ISA)

- The ISA defines the **programmer-visible state** of a processor architecture
  - type and size of operands
    - specifying types
      - byte, half word, word, float, double…
  - operations
    - arithmetic and logical
    - data transfer
    - control
    - system
    - floating point
    - string
    - multimedia

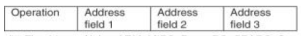
Rank	x86 operation	%
1	load	22
2	conditional branch	20
3	compare	16
4	store	12
5	add	8
6	and	6
7	sub	5
8	move reg-reg	4
9	call	1
10	return	1
	total	**96**

Instruction Set Architecture (ISA)

- The ISA defines the **programmer-visible state** of a processor architecture
  - instruction encoding
    - defines binary representation for each operation and operand
    - fixed size vs. variable size



(a) Variable (e.g., Intel 80x86, VAX)



(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)

x86 Machine-Level Programming Basics

- **History of Intel processors and architectures**
- C, assembly, machine code
- Assembly Basics: Registers, operands, move, push/pop
- Introduction to x86-64

Acknowledgement: slides based on the cs:app2e material

Intel x86 Processors

- Dominate laptop/desktop/server market
- Evolutionary design
  - Backwards compatible up until 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 Reduced Instruction Set Computers (RISC)
  - But, Intel has done just that!
    - In terms of speed. Less so for low power.

Intel x86 Evolution: Milestones

Name	*Date*	*Transistors*	*MHz*
8086	**1978**	**29K**	**5-10**

- First 16-bit processor. Basis for IBM PC & DOS
- 1MB address space

386	**1985**	**275K**	**16-33**

- First 32 bit processor , referred to as IA32
- Added "flat addressing"
- Capable of running Unix
- 32-bit Linux/gcc uses no instructions introduced in later models

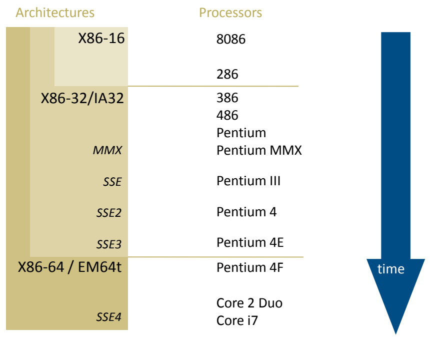
Pentium 4F	**2004**	**125M**	**2800-3800**

- First 64-bit processor, referred to as x86-64

Core i7	**2008**	**731M**	**2667-3333**

- multiple cores

Intel x86 Processors: Overview

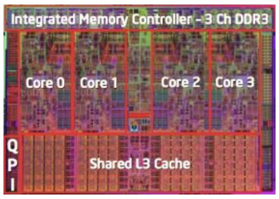


IA: often redefined as latest Intel architecture

Intel x86 Processors

- Machine Evolution

386	1985	0.3M
486	1989	1.9M
Pentium	1993	3.1M
Pentium/MMX	1997	4.5M
PentiumPro	1995	6.5M
Pentium III	1999	8.2M
Pentium 4	2001	42M
Core 2 Duo	2006	291M
Core i7 (4 cores)	2008	731M
Core i7 (6 cores)	2011	2'270M
Xeon E7 v2 (15 c)	2014	4'310M



Core i7 (45nm)

- Added Features
  - Instructions to support multimedia operations
    - Parallel operations on 1, 2, and 4-byte data, both integer & FP
  - Instructions to enable more efficient conditional operations

New Species: ia64, then IPF, then Itanium,…

Name	*Date*	*Transistors*
Itanium	**2001**	**10M**

- First shot at 64-bit architecture: first called IA64
- Radically new instruction set designed for high performance
- Can run existing IA32 programs
  - On-board "x86 engine"
- Joint project with Hewlett-Packard

Itanium 2	**2002**	**221M**

- Big performance boost

Itanium 2 Dual-Core	**2006**	**1.7B**

- Itanium has not taken off in marketplace
  - very fast (esp. FP), very hot, and very expensive
  - Lack of backward compatibility, no good compiler support, Pentium 4 got too good, overtaken by 64-bit x86 designs

x86 Clones: Advanced Micro Devices (AMD)

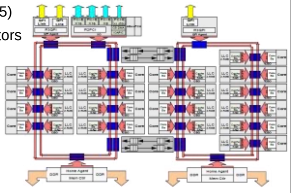
- Historically
  - AMD has followed just behind Intel
  - A little bit slower, a lot cheaper
- Then
  - Recruited top circuit designers from Digital Equipment Corp. and other downward trending companies
  - Built Opteron: tough competitor to Pentium 4
  - Developed x86-64, their own extension to 64 bits
- Recently
  - Intel much quicker with multi-core design
  - Intel currently far ahead in performance
  - Intel em64t backwards compatible to x86-64

Intel's 64-Bit

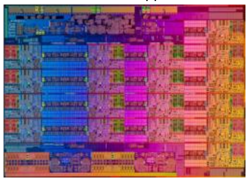
- Intel Attempted Radical Shift from IA32 to IA64
  - Totally different architecture (Itanium)
  - Executes IA32 code only as legacy
  - Application performance disappointing
- AMD Stepped in with Evolutionary Solution
  - x86-64 (now called "AMD64")
- Intel Felt Obligated to Focus on IA64
  - Hard to admit mistake or that AMD is better
- 2004: Intel Announces EM64T extension to IA32
  - Extended Memory 64-bit Technology
  - Almost identical to x86-64!

Intel Haswell

- Current leader (soon to be replace)
  - Xeon E7v3 (E7-8890v3) (May 2015)
    - 18-core chip: 5.5 billion transistors
    - 2.5/3.3GHz
    - 2 threads per core
    - no GPU on chip
  - virtualization support



source: The Platform



- 18 cores, 36 threads
- per core: 64KB L1, 256KB L2 cache
- 45MB shared L3 cache
- 165W TPD

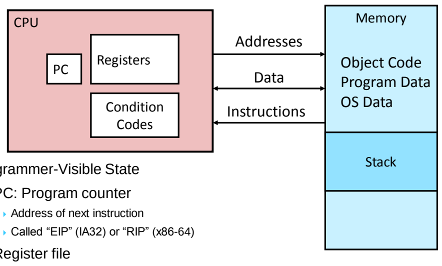
Our Coverage

- IA32
  - The traditional x86
- x86-64/EM64T
  - The emerging standard
- Presentation
  - ia32 in sections 3.1—3.7 of the textbook
  - x86-64 in section 3.13
  - lecture will cover both

x86 Machine-Level Programming Basics

- History of Intel processors and architectures
- **C, assembly, machine code**
- Assembly Basics: Registers, operands, move, push/pop
- Introduction to x86-64

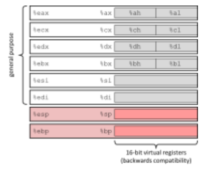
Assembly Programmer's View



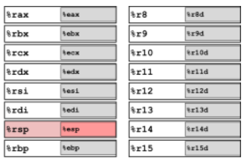
- Programmer-Visible State
  - PC: Program counter
    - Address of next instruction
    - Called "EIP" (IA32) or "RIP" (x86-64)
  - Register file
    - Heavily used program data
  - Condition codes
    - Store status information about most recent arithmetic operation
    - Used for conditional branching
- Memory
  - Byte addressable array
  - Code, user data, (some) OS data
  - Includes stack used to support procedures

x86 ISA

- x86 (32 bit):
  - register-memory architecture
    - 2 operands, max 1 memory
  - registers:
    - 6(8) general purpose registers, PC (EIP), condition codes, processor status
  - byte-addressed memory, little-endian
  - 1, 2, 4, 8 byte data types



- x86_64 (64 bit):
  - same as above except
    - 16 general purpose registers
    - 16-byte data type



x86 ISA

- instruction format
  - operand size specifier
    - postfix to the operation
    - b (1 byte), w (2 bytes), l (4 bytes), q (8 bytes – x86_64 only)
  - two-operand format: first source is implicit destination

    a = a + b

    in x86 assembly:

    addl b, a

    - careful with non-commutative operations:

    subl a, b ≠ subl b, a

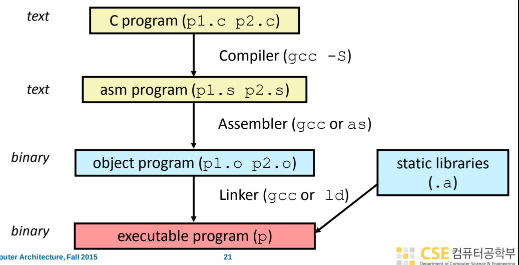
x86 ISA

- Operand specifiers

Type	Form	Operand value	Name
Immediate	`$Imm`	`Imm`	Immediate
Register	`Ea`	`R[Ea]`	Register
Memory	`Imm`	`M[Imm]`	Absolute
	`(Eb)`	`M[R[Eb]]`	Indirect
	`Imm(Eb)`	`M[R[Eb] + Imm]`	Base + displacement
	`(Eb, Ei)`	`M[R[Eb] + R[Ei]]`	Indexed
	`Imm(Eb, Ei)`	`M[R[Eb] + R[Ei] + Imm]`	Indexed
	`(, Ei, s)`	`M[R[Ei]*s]`	Scaled indexed
	`Imm(, Ei, s)`	`M[R[Ei]*s + Imm]`	Scaled indexed
	`(Eb, Ei, s)`	`M[R[Eb] + R[Ei]*s]`	Scaled indexed
	`Imm(Eb, Ei, s)`	`M[R[Eb] + R[Ei]*s + Imm]`	Scaled indexed

From C to Machine 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`**



Compiling To IA32 Assembly

C Code

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

`$ gcc -m32 -O -S code.c`

produces the file `code.s`:

Generated IA32 Assembly by gcc-4.4.6 on CentOS 6.3

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

→ compare output generated with `gcc -m32 -O0 -S code.c`

Compiling To IA32 Assembly – GCC Version

C Code

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

`$ gcc -m32 -O -S code.c`

produces the file `code.s`:

Generated IA32 Assembly by gcc-4.6.3 on Gentoo

```
sum:
.LFB0:
    .cfi_startproc
    movl    8(%esp), %eax
    addl    4(%esp), %eax
    ret
    .cfi_endproc
```

→ compare output generated with `gcc -m32 -O0 -S code.c`

Compiling To x86_64 Assembly

C Code

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

`$ gcc` **`-m64`** `-O -S code.c`

produces the file `code.s`:

Generated x86_64 assembly (identical for gcc-4.4.6/gcc-4.6.3)

```
sum:
.LFB0:
    .cfi_startproc
    leal    (%rsi,%rdi), %eax
    ret
    .cfi_endproc
```

→ compare output generated with `gcc -m64 -O0 -S code.c`

Assembly Characteristics: 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

Assembly Characteristics: Operations

- **Arithmetic/Logic Operations**
  Perform arithmetic/logic functions on registers or memory data
- **Memory Operations**
  Transfer data between memory and registers
  - Load data from memory into register
  - Store register data into memory
- **Control Transfer Operations**
  Transfer control
  - Unconditional jumps to/from procedures
  - Conditional branches
- **Special Operations**
  - processor control, loop operations, …

Object Code

IA32 Code for sum

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

- Total of 11 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*
    - Linking occurs when program begins execution

Machine Instruction Example

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

- C Code
  - Add two signed integers

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

Similar to expression:

```
x += y
```

More precisely:

```
int eax;
int *ebp;
eax += ebp[2]
```

- Assembly
  - Add two 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`**

```
0x0804809a:   03 45 08
```

- Object Code
  - 3-byte instruction
  - Stored at address **`0x0804809a`**

Disassembling Object Code

```
$ gcc -m32 -c -O code.c
$ objdump -d code.o
```

```
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
```

- Disassembler
  - 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 using gdb

- Within **`gdb`** Debugger

```
$ gcc -m32 -O -o p code.c
$ gdb p
```

```
(gdb) disassemble sum
Dump of assembler code for function sum:
   0x08048394 <+0>: push   %ebp
   0x08048395 <+1>: mov    %esp,%ebp
   0x08048397 <+3>: mov    0xc(%ebp),%eax
   0x0804839a <+6>: add    0x8(%ebp),%eax
   0x0804839d <+9>: pop    %ebp
   0x0804839e <+10>:ret
End of assembler dump.
(gdb) x/11xb sum
0x8048394 <sum>:     0x55   0x89   0xe5   0x8b   0x45   …
0x804839c <sum+8>:   0x08   0x5d   0xc3
(gdb)
```

What Can be Disassembled?

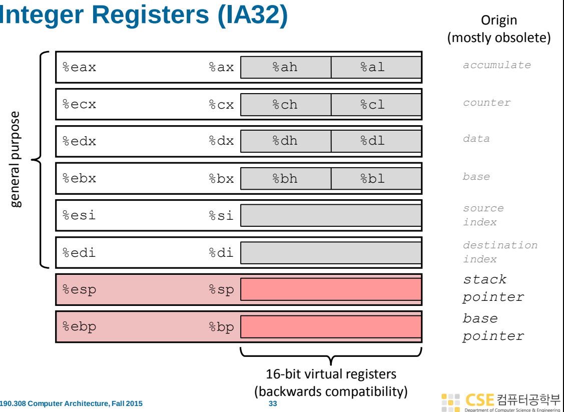
```
% objdump -d WINWORD.EXE

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 (not everything makes sense, of course)
- Disassembler examines bytes and reconstructs assembly source

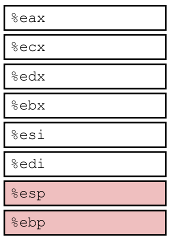
x86 Machine-Level Programming Basics

- History of Intel processors and architectures
- C, assembly, machine code
- **Assembly Basics: Registers, operands, move, push/pop**
- Introduction to x86-64

Integer Registers (IA32)

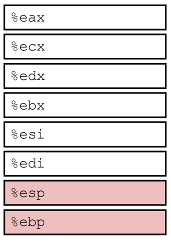


Moving Data: IA32



- Moving Data
  `movx` ***Source*, *Dest***
  with `x` in `{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 move instructions in typical code

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`	`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 transfer with a single instruction

Simple Memory Addressing Modes

- **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
```

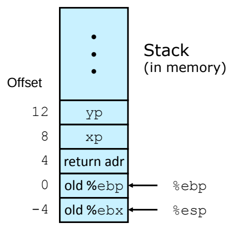
Using Simple Addressing Modes

```
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                         }
```

Understanding Swap

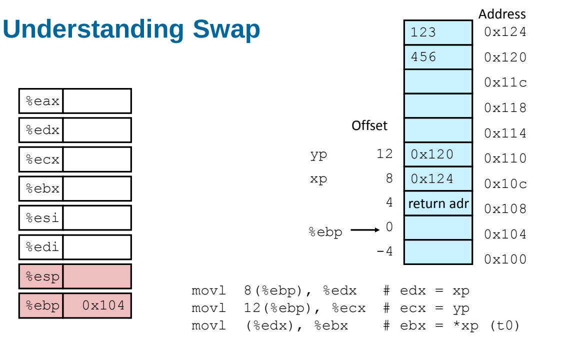
```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



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

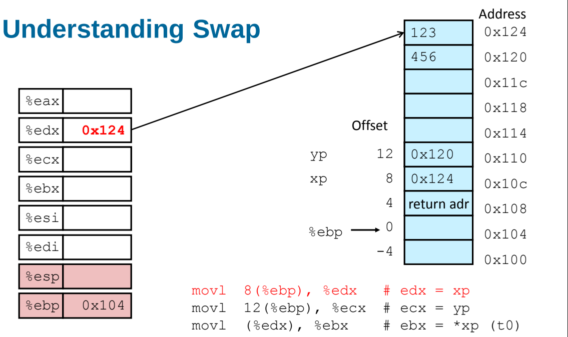
```
movl  8(%ebp), %edx     # edx = xp
movl  12(%ebp), %ecx    # ecx = yp
movl  (%edx), %ebx      # ebx = *xp (t0)
movl  (%ecx), %eax      # eax = *yp (t1)
movl  %eax, (%edx)      # *xp = t1
movl  %ebx, (%ecx)      # *yp = t0
```

Understanding Swap



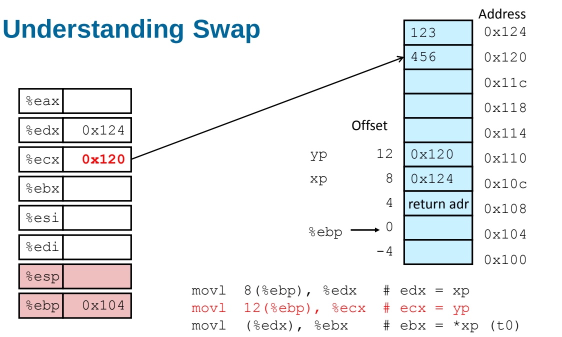
```
movl  8(%ebp), %edx     # edx = xp
movl  12(%ebp), %ecx    # ecx = yp
movl  (%edx), %ebx      # ebx = *xp (t0)
movl  (%ecx), %eax      # eax = *yp (t1)
movl  %eax, (%edx)      # *xp = t1
movl  %ebx, (%ecx)      # *yp = t0
```

Understanding Swap



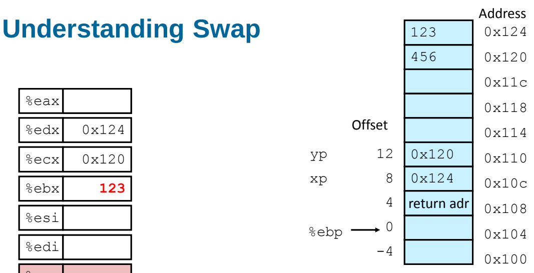
```
movl  8(%ebp), %edx     # edx = xp
movl  12(%ebp), %ecx    # ecx = yp
movl  (%edx), %ebx      # ebx = *xp (t0)
movl  (%ecx), %eax      # eax = *yp (t1)
movl  %eax, (%edx)      # *xp = t1
movl  %ebx, (%ecx)      # *yp = t0
```

Understanding Swap



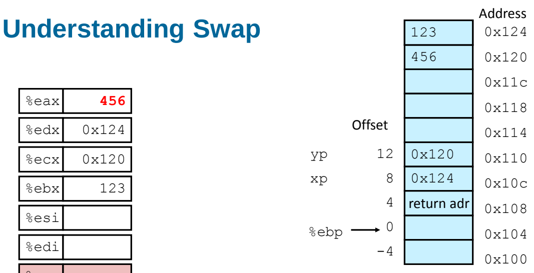
```
movl  8(%ebp), %edx     # edx = xp
movl  12(%ebp), %ecx    # ecx = yp
movl  (%edx), %ebx      # ebx = *xp (t0)
movl  (%ecx), %eax      # eax = *yp (t1)
movl  %eax, (%edx)      # *xp = t1
movl  %ebx, (%ecx)      # *yp = t0
```

Understanding Swap



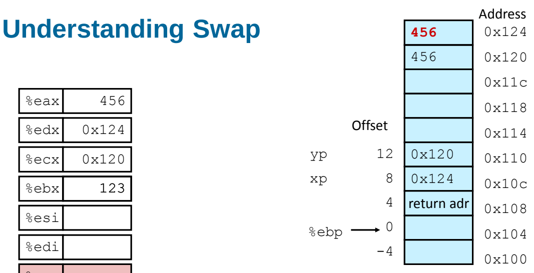
```
movl  8(%ebp), %edx    # edx = xp
movl  12(%ebp), %ecx   # ecx = yp
movl  (%edx), %ebx     # ebx = *xp (t0)
movl  (%ecx), %eax     # eax = *yp (t1)
movl  %eax, (%edx)     # *xp = t1
movl  %ebx, (%ecx)     # *yp = t0
```

Understanding Swap



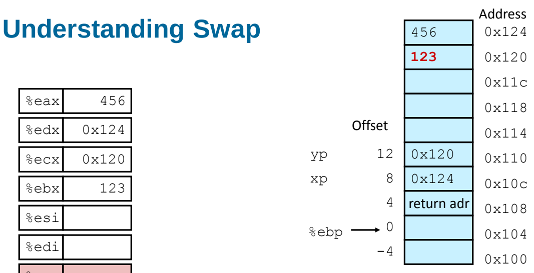
```
movl  8(%ebp), %edx    # edx = xp
movl  12(%ebp), %ecx   # ecx = yp
movl  (%edx), %ebx     # ebx = *xp (t0)
movl  (%ecx), %eax     # eax = *yp (t1)
movl  %eax, (%edx)     # *xp = t1
movl  %ebx, (%ecx)     # *yp = t0
```

Understanding Swap



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

Understanding Swap



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

Complete Memory Addressing Modes

- 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

Discussed later in detail

Sign/Zero Expansion

- **MOV** with operands of different sizes

```
void main(void)
{
  char c = -1;
  int  i = c;

  printf("%d\n", i);
}
```

vs

```
void main(void)
{
  unsigned char c = -1;
  int  i = c;

  printf("%d\n", i);
}
```

- expected result?

Sign/Zero Expansion

- **MOV** with operands of different sizes

```
void main(void)
{
  char c = -1;
  int  i = c;

  printf("%d\n", i);
}
```

VS

```
void main(void)
{
  unsigned char c = -1;
  int  i = c;

  printf("%d\n", i);
}
```

- disassembly

```
…
movb   $0xff, -0x5(%rbp)
movsbl -0x5(%rbp), %eax
…
```

```
…
movb   $0xff, -0x5(%rbp)
movzbl -0x5(%rbp), %eax
…
```

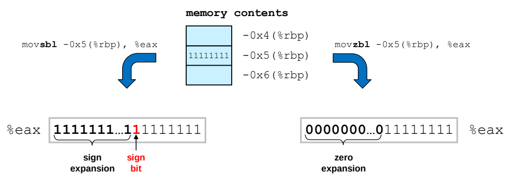
Sign/Zero Expansion

- **MOV** with operands of different sizes

```
void main(void)
{
  char c = -1;
  int  i = c;

  printf("%d\n", i);
}
```

VS

```
void main(void)
{
  unsigned char c = -1;
  int  i = c;

  printf("%d\n", i);
}
```



Sign/Zero Expansion

- **MOV** with operands of different sizes
  - `movsbl`: move **s**ign-extand **b**yte to double word(**l**)
  - `movzbl`: move **z**ero-expand **b**yte to double word(**l**)

Instruction		Effect	Description
MOV	**S, D**	**D ← S**	**Move (equal-sized operands)**
`movb`		move byte	
`movw`		move word (16-bit)	
`movl`		move double word (32-bit)	
`movq`		move quad word (64-bit)	
MOVS	**S, D**	**D ← SignExtend(S)**	**Move with sign extension**
`movsb[w,l,q]`		move sign-extended byte to word, double word, quad word	
`movsw[l,q]`		move sign-extended word to double word, quad word	
`movslq`		move sign-extended double word to quad word	
MOVZ	**S, D**	**D ← ZeroExtend(S)**	**Move with zero extension**
`movzb[w,l,q]`		move zero-extended byte to word, double word, quad word	
`movzw[l,q]`		move zero-extended word to double word, quad word	
`movzlq`		move zero-extended double word to quad word	

  - `movs/z`: always size(source operand) < size(destination operand)

Stack Operations: PUSH & POP

- `%esp`/`%rsp` point to a memory region call the *stack*
- the stack holds temporary data associated with currently active functions
- `push/pop` are used to store/retrieve temporary data onto/from the stack

Instruction		Effect	Description
STACK			**Stack operations**
`pushl`	`S`	`R[%esp] ← R[%esp] – 4;` `M[R[%esp]] ← S`	push double word onto stack
`popl`	`D`	`D ← M[R[%esp]];` `R[%esp] ← R[%esp] + 4`	pop double word from stack

  - `pushl s` has same effect as "`subl $4, %esp; movl s, (%esp)`"
  - `popl d` has same effect as "`movl (%esp), d; addl $4, %esp`"
  - size modifiers apply: `push[b,w,l,q], pop[b,w,l,q]`

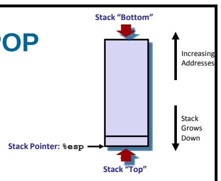
Stack Operations: PUSH & POP

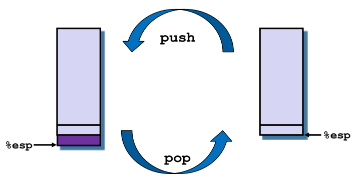
- `%esp`/`%rsp` point to a memory region call the *stack*
- the stack holds temporary data associated with currently active functions
- `push/pop` are used to store/retrieve temporary data onto/from the stack

Instruction		Effect	Description
STACK			**Stack operations**
`pushl`	`S`	`R[%esp] ← R[%esp] – 4;` `M[R[%esp]] ← S`	push double word onto stack
`popl`	`D`	`D ← M[R[%esp]];` `R[%esp] ← R[%esp] + 4`	pop double word from stack

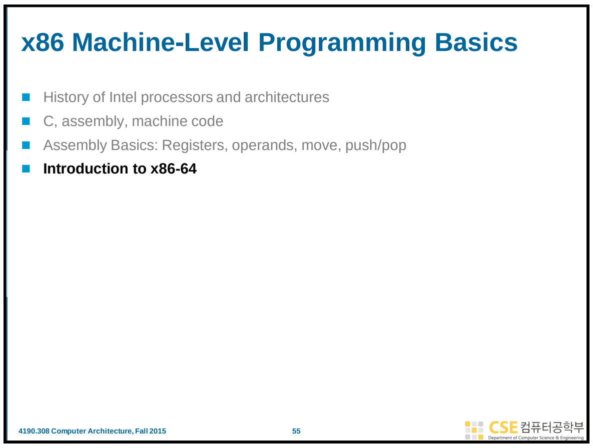
  - `pushl s` has same effect as "`subl $4, %esp; movl s, (%esp)`"
  - `popl d` has same effect as "`movl (%esp), d; addl $4, %esp`"
  - size modifiers apply: `push[b,w,l,q], pop[b,w,l,q]`

Stack Operations: PUSH & POP



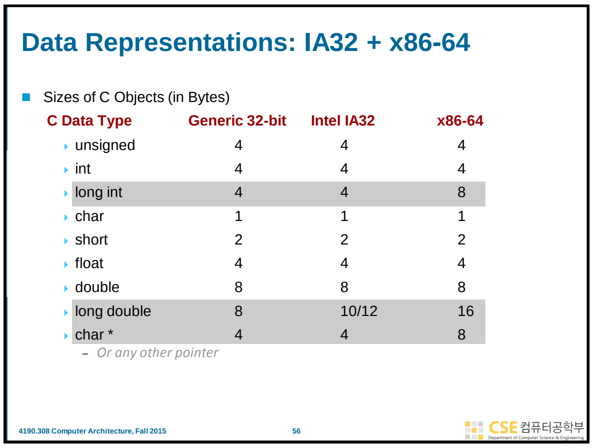


- maintaining stack disciplince: typically
  - `push/pop` occur in pairs
  - and in reverse order: `push a, push b, push c` → `pop c, pop b, pop a`



x86 Machine-Level Programming Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move, push/pop
- **Introduction to x86-64**

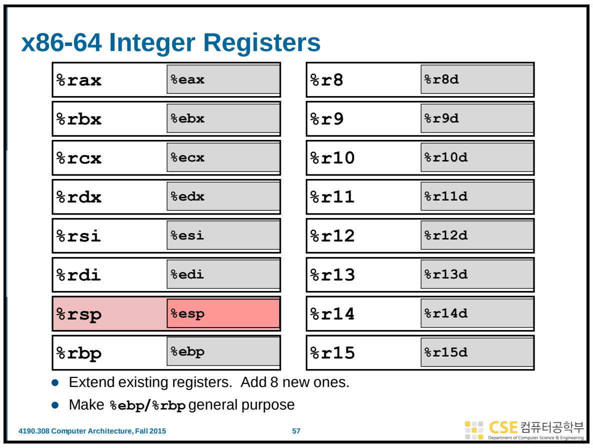


Data Representations: IA32 + x86-64

- 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

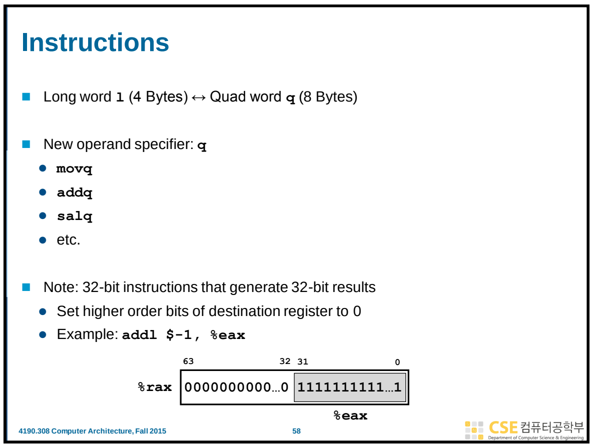
– *Or any other pointer*



x86-64 Integer Registers

%rax	**%eax**	**%r8**	**%r8d**
%rbx	**%ebx**	**%r9**	**%r9d**
%rcx	**%ecx**	**%r10**	**%r10d**
%rdx	**%edx**	**%r11**	**%r11d**
%rsi	**%esi**	**%r12**	**%r12d**
%rdi	**%edi**	**%r13**	**%r13d**
%rsp	**%esp**	**%r14**	**%r14d**
%rbp	**%ebp**	**%r15**	**%r15d**

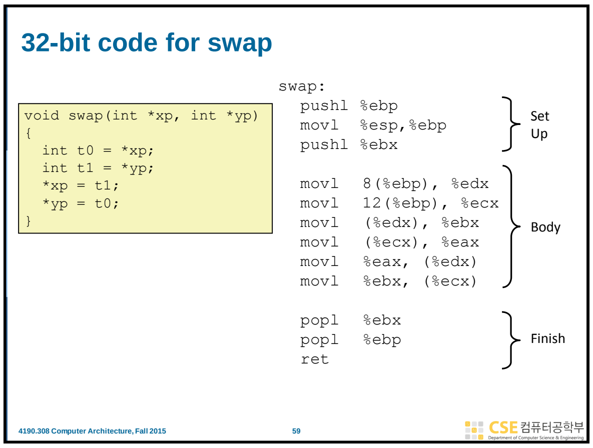
- Extend existing registers. Add 8 new ones.
- Make **%ebp/%rbp** general purpose



Instructions

- Long word **l** (4 Bytes) ↔ Quad word **q** (8 Bytes)
- New operand specifier: **q**
  - **movq**
  - **addq**
  - **salq**
  - etc.
- Note: 32-bit instructions that generate 32-bit results
  - Set higher order bits of destination register to 0
  - Example: **addl $-1, %eax**

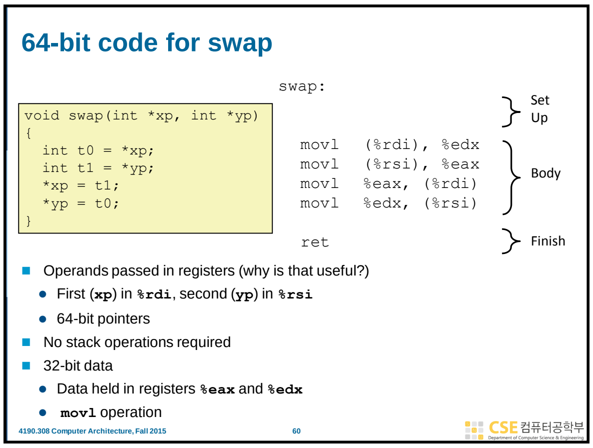
```
         63          32  31           0
%rax   0000000000…0  1111111111…1
                        %eax
```



32-bit code for swap

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

```
swap:
   pushl %ebp
   movl  %esp,%ebp          } Set 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
```

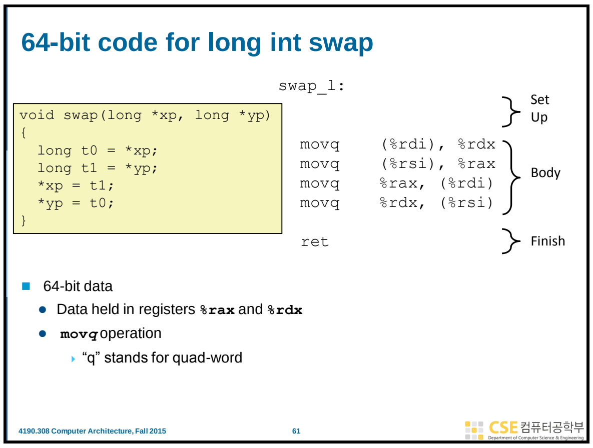


64-bit code for swap

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

```
swap:
                            } Set Up
   movl  (%rdi), %edx
   movl  (%rsi), %eax       } Body
   movl  %eax, (%rdi)
   movl  %edx, (%rsi)

   ret                      } Finish
```

- Operands passed in registers (why is that 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

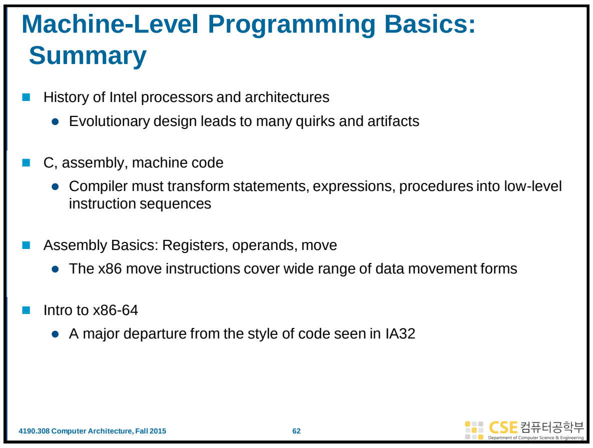


64-bit code for long int swap

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

```
swap_l:
                                   } Set Up
  movq    (%rdi), %rdx   ⎫
  movq    (%rsi), %rax   ⎬ Body
  movq    %rax, (%rdi)   ⎪
  movq    %rdx, (%rsi)   ⎭

  ret                    } Finish
```

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



Machine-Level Programming Basics: Summary

- History of Intel processors and architectures
  - Evolutionary design leads to many quirks and artifacts
- C, assembly, machine code
  - Compiler must transform statements, expressions, procedures into low-level instruction sequences
- Assembly Basics: Registers, operands, move
  - The x86 move instructions cover wide range of data movement forms
- Intro to x86-64
  - A major departure from the style of code seen in IA32