Computer System Organization (CS2.201)

Lecture # 05-07

Instruction Set Architecture / Assembly Language Programming

Avinash Sharma

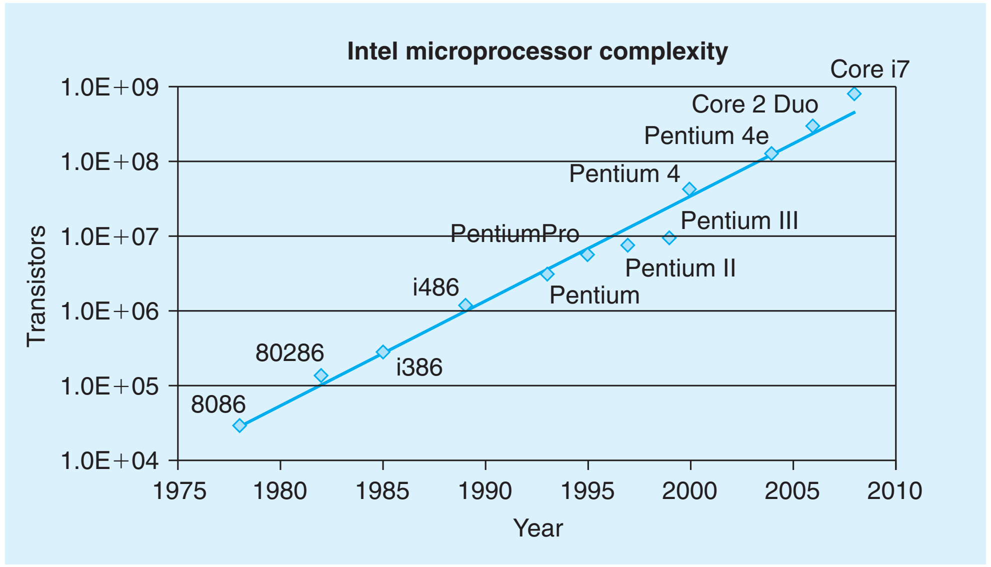
Center for Visual Information Technology (CVIT),

IIIT Hyderabad

Slide content acknowledgment: Dr. Suresh Purini & other public sources

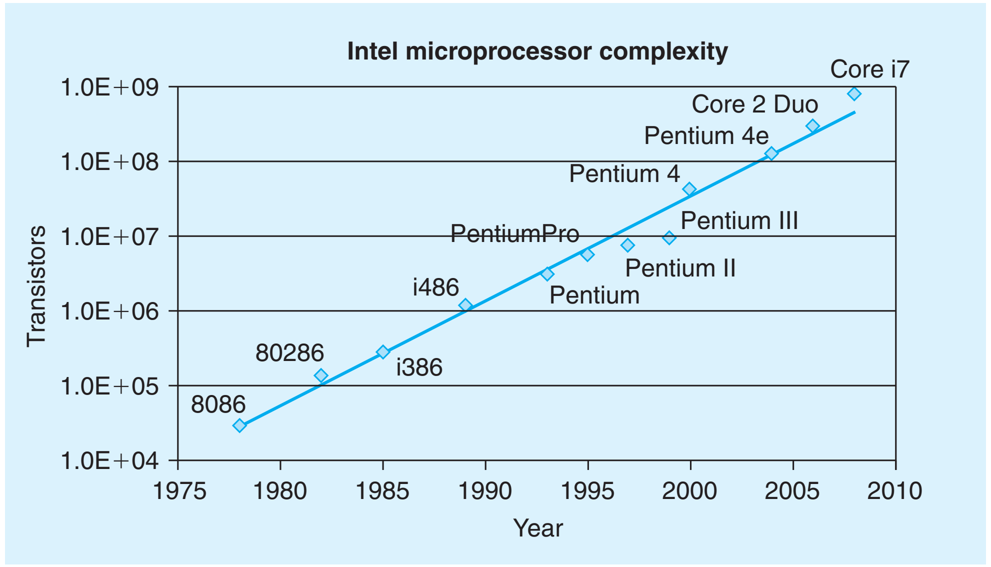
Intel Microprocessor Design History

- 8086: One of the first single-chip, 16-bit microprocessors.
- The 8088, a variant of the 8086 with a 20 bit address space, formed the heart of the original IBM personal computers.
- Architecturally, the machines were limited to a 655,360-byte address space as and the operating system reserved 393,216 bytes for its own use.



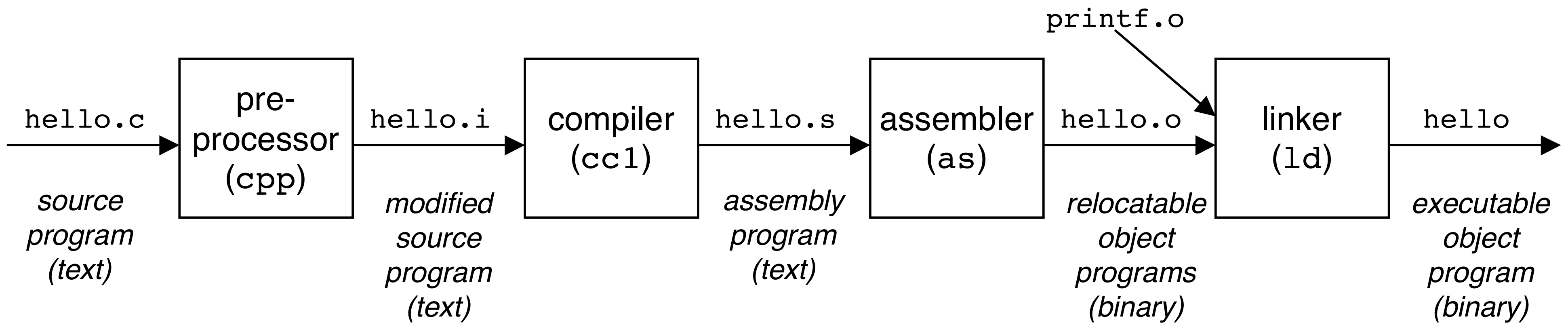
Intel Microprocessor Design History

- **80286:** More transistors.
- **i386:** Expanded to 32bits architecture. Added the flat addressing model used by Linux.
- **i486: I**ntegrated the floating-point unit onto the processor chip
- **Pentium:** Improved performance.
- **PentiumPro:** Introduced a radically new processor design, internally known as the *P6* microarchitecture. Added a class of "conditional move" instructions to the instruction set.



Typical Compilation System

```
gcc -o hello hello.c
```



Source: Bryant & O'Hallaron

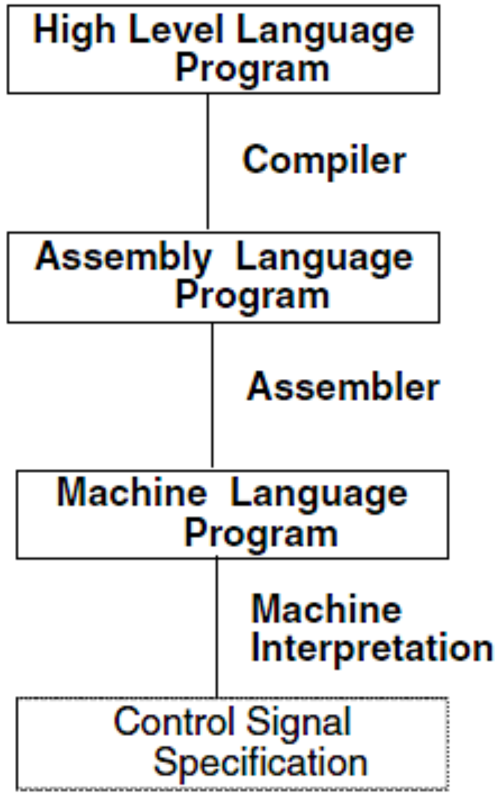
Programming Abstractions

We can program a microprocessor using

a) Instruction opcodes (also called Machine Code)
b) Assembly language
c) High level programming languages

- ❑ The level of abstraction increases from Top to Bottom.
- ❑ As the level of abstraction increases, ease of programmability also increases!
- ❑ Hmm, but we may lose the fine-grained control over the underlying hardware?

Levels of Abstraction


High Level Language Program
Compiler
Assembly Language Program
Assembler
Machine Language Program
Machine Interpretation
Control Signal Specification

```
temp := v[k];
v[k] := v[k+1];
v[k+1] := temp;
```

```
lw   $15, 0($2)
lw   $16, 4($2)
sw   $16, 0($2)
sw   $15, 4($2)
```

*MIPS Assembly Language

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

*Microprocessor without Interlocked Pipeline Stages

Source: Prof. Cheung (ICL)

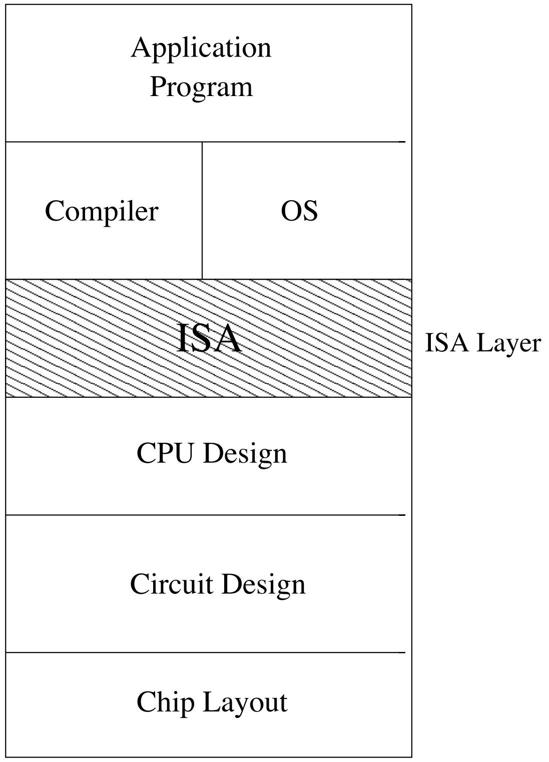
Instruction Set Architecture

Assembly Language View

- Processor state: registers, memory, etc.
- Instructions and how instructions are encoded

Layer of Abstraction

- Above: how to program machine, processor executes instructions sequentially
- Below: What needs to be built
  - Use variety of tricks to make it run faster, e.g., execute multiple instructions simultaneously
  - Safeguard to ensure that the overall behavior matches the sequential operation dictated by the ISA.



Instruction Set Architecture

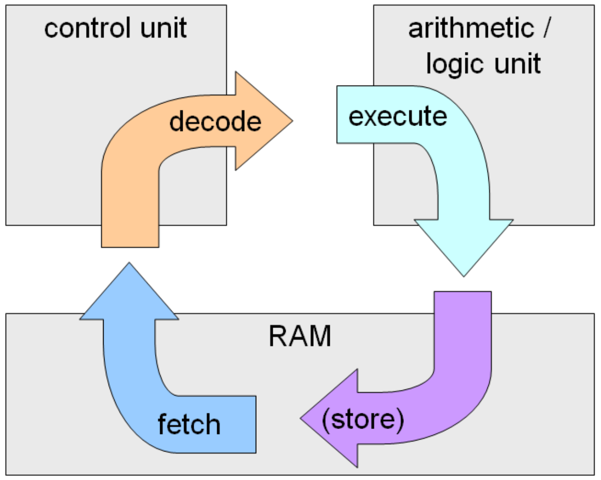
- x86-64, describe the behavior of a program as if each instruction is executed in sequence, with one instruction completing before the next one begins.
- The processor hardware is far more elaborate, executing many instructions concurrently, but they employ safeguards to ensure that the overall behavior matches the sequential operation dictated by the ISA.
- Machine-level program use a virtual address space, providing a memory model that appears to be a very large byte array.
- The actual implementation of the memory system involves a combination of multiple hardware memories and operating system software

Instruction Set Architecture

- There are various means of giving a semantics or meaning to a programming system.
- Probably the most sensible for an assembly (or machine) language is an operational semantics, also known as an *interpreter semantics*.
- That is, we explain the semantics of each possible operation in the language by explaining the effect that execution of the operation has on the machine state.

Instruction Set Architecture

- The most fundamental abstraction for the machine semantics for the x86 is the ***fetch-decode-execute*** cycle.
- The machine repeats the following steps :
  1. fetch the next instruction from memory (the PC tells you which is next);
  2. decode the instruction (in the control unit);
  3. execute the instruction, updating the state appropriately;
  4. go to step 1.



This is also called the von Neumann architecture.

Assembly Code Representation

- Very close to machine code.
- More readable textual format, as compared to the binary format of machine code.
- Understand assembly code and how it relates to the original C code is a key step in understanding how computers execute programs.
- Enable visibility to key process states like *registers, program counter,* etc.

Instruction Format

- IA32 instructions can range in length from 1 to 15 bytes.
- The instruction encoding is designed so that commonly used instructions and those with fewer operands require a smaller number of bytes as compare to less common ones or ones with more operands.
- The instruction format is designed in such a way that from a given starting position, there is a unique decoding of the bytes into machine instructions.

Assembly Code Example

C Code (code.c)

```c
int accum = 0;

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

Parts of Assembler Output

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

gcc -O1 -S code.c

Parts of binary object file

```
00000000 <sum>:
Offset   Bytes
    0:   55
    1:   89 e5
    3:   8b 45 0c
    6:   03 45 08
    9:   01 05 00 00 00 00
    f:   5d
   10:   c3
```

Disassembler Output

```
00000000 <sum>:
push   %ebp
mov    %esp,%ebp
mov    0xc(%ebp),%eax
add    0x8(%ebp),%eax
add    %eax,0x0
pop    %ebp
ret
```

Processor State

- The program counter (PC , %eip in IA32) indicates the address in memory of the next instruction to be executed.
- The integer register file contains eight named locations storing 32-bit values. These registers can hold:
  - Addresses (corresponding to C pointers) or integer data.
  - Critical parts of the program state
  - Temporary data, such as the local variables of a procedure, and the value to be returned by a function.

Processor State

- The condition code registers hold status information about the most recently executed arithmetic or logical instruction.
- These are used to implement conditional changes in the control or data flow, such as is required to implement *if* and *while* statements.
- A set of floating-point registers store floating-point data.

Data Formats

- Intel uses the term “word” to refer to a 16-bit data type.
- Hence, 32- bit quantities as “double words” and 64-bit quantities as “quad words.”

C declaration	Intel data type	Assembly code suffix	Size (bytes)
`char`	Byte	`b`	1
`short`	Word	`w`	2
`int`	Double word	`l`	4
`long int`	Double word	`l`	4
`long long int`	—	—	4
`char *`	Double word	`l`	4
`float`	Single precision	`s`	4
`double`	Double precision	`l`	8
`long double`	Extended precision	`t`	10/12

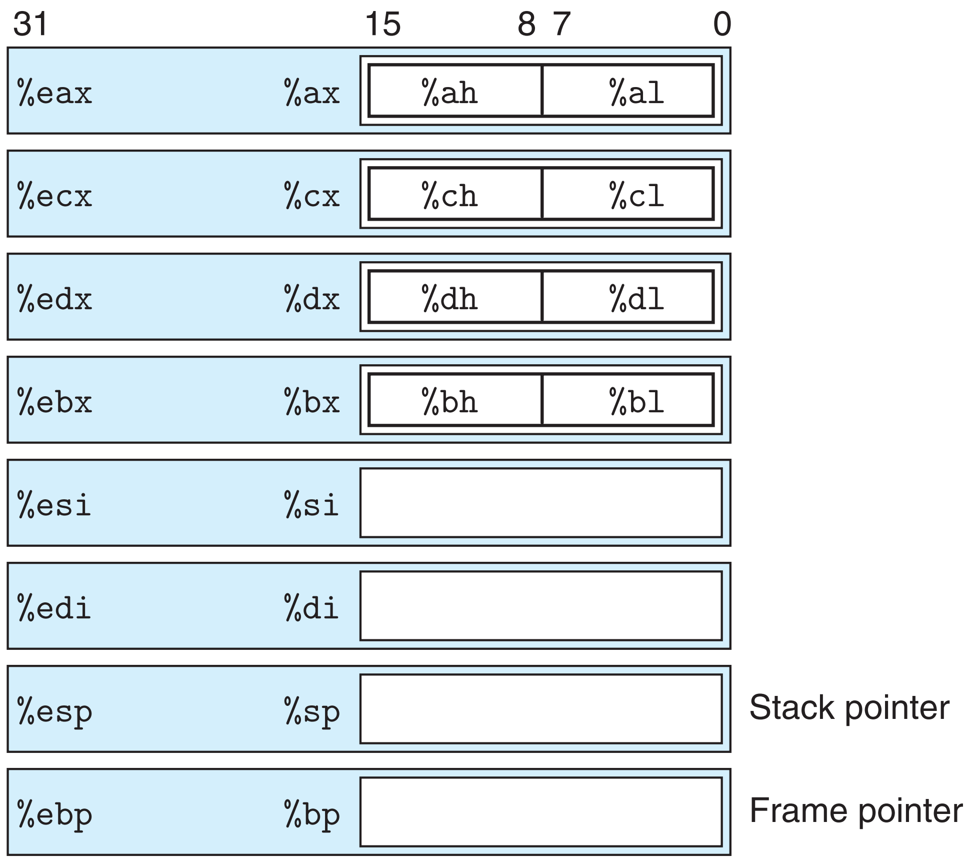
Sizes of C data types in IA32.

Data Formats

- IA32 does not support long long data type (recently introduced in C) in hardware. Instead, the compiler must generate sequences of instructions that operate on 32 bits data.
- Most assembly-code instructions generated by gcc have a single-character suffix denoting the size of the operand e.g., movb (move byte), movw (move word), and movl (move double word).
- Note that the assembly code uses the suffix ‘l’ to denote both a 4-byte integer as well as an 8-byte double-precision floating-point number.
- This causes no ambiguity, since floating point involves an entirely different set of instructions and registers.

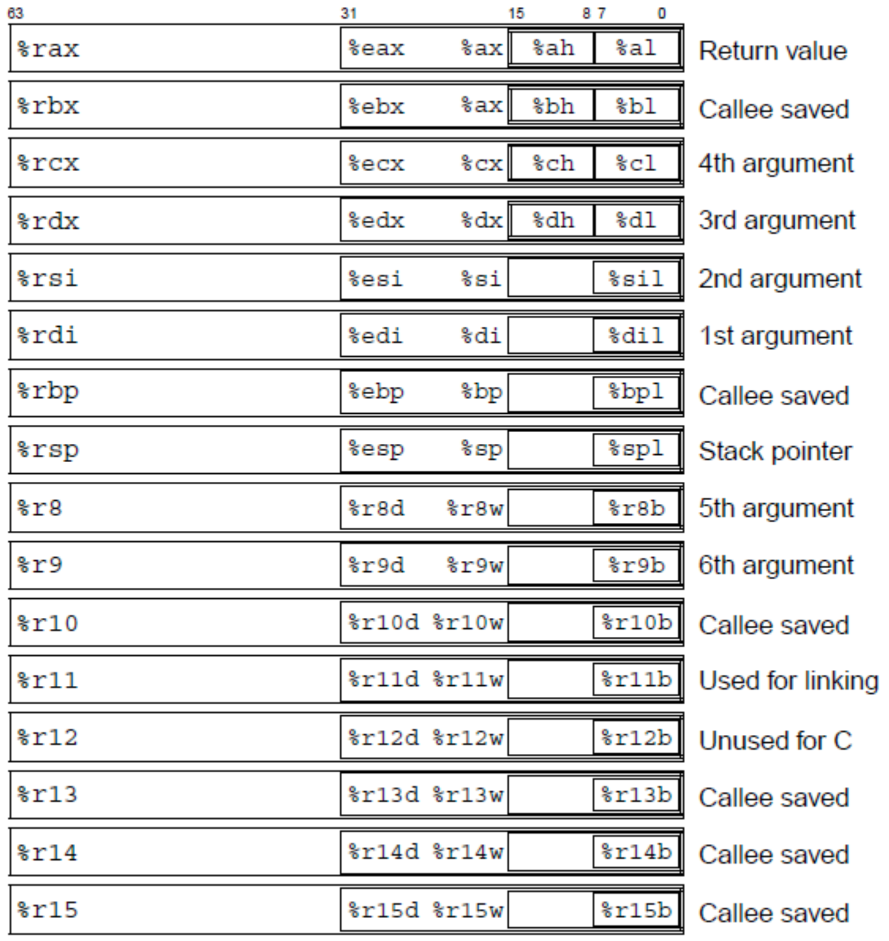
Accessing Information: IA32 Integer Registers

- 8 registers used to store integer/pointers
  - First six registers can be considered general-purpose registers, except some instructions that uses specific registers as source and/or destination.
  - Last two registers, contain pointers to important places in the program stack
  - The low-order 2 bytes of the first four registers can be independently read or written by the byte operation instructions.
  - Similarly, the low-order 16 bits of each register can be read or written by word operation instructions.



Accessing Information: x86-64 Integer Registers

- 16 registers to store integer/pointers
  - First six registers can be considered general-purpose registers, except some instructions that uses specific registers as source and/or destination.
  - Last two registers, contain pointers to important places in the program stack
  - The low-order 2 bytes of the first four registers can be independently read or written by the byte operation instructions.
  - Similarly, the low-order 16 bits of each register can be read or written by word operation instructions.



63	31		15	8 7	0
%rax	%eax	%ax	%ah	%al	Return value
%rbx	%ebx	%ax	%bh	%bl	Callee saved
%rcx	%ecx	%cx	%ch	%cl	4th argument
%rdx	%edx	%dx	%dh	%dl	3rd argument
%rsi	%esi	%si		%sil	2nd argument
%rdi	%edi	%di		%dil	1st argument
%rbp	%ebp	%bp		%bpl	Callee saved
%rsp	%esp	%sp		%spl	Stack pointer
%r8	%r8d	%r8w		%r8b	5th argument
%r9	%r9d	%r9w		%r9b	6th argument
%r10	%r10d	%r10w		%r10b	Callee saved
%r11	%r11d	%r11w		%r11b	Used for linking
%r12	%r12d	%r12w		%r12b	Unused for C
%r13	%r13d	%r13w		%r13b	Callee saved
%r14	%r14d	%r14w		%r14b	Callee saved
%r15	%r15d	%r15w		%r15b	Callee saved

Operand Specifiers

- Most instructions have one or more operands, used as source and destination references
- Three types of operand exists
  - Constant (Immediate)
  - Register
  - Memory

Type	Form	Operand value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	$\mathtt{E}_a$	$R[\mathtt{E}_a]$	Register
Memory	Imm	$M[Imm]$	Absolute
Memory	$(\mathtt{E}_a)$	$M[R[\mathtt{E}_a]]$	Indirect
Memory	$Imm(\mathtt{E}_b)$	$M[Imm + R[\mathtt{E}_b]]$	Base + displacement
Memory	$(\mathtt{E}_b, \mathtt{E}_i)$	$M[R[\mathtt{E}_b] + R[\mathtt{E}_i]]$	Indexed
Memory	$Imm(\mathtt{E}_b, \mathtt{E}_i)$	$M[Imm + R[\mathtt{E}_b] + R[\mathtt{E}_i]]$	Indexed
Memory	$(, \mathtt{E}_i, s)$	$M[R[\mathtt{E}_i] \cdot s]$	Scaled indexed
Memory	$Imm(, \mathtt{E}_i, s)$	$M[Imm + R[\mathtt{E}_i] \cdot s]$	Scaled indexed
Memory	$(\mathtt{E}_b, \mathtt{E}_i, s)$	$M[R[\mathtt{E}_b] + R[\mathtt{E}_i] \cdot s]$	Scaled indexed
Memory	$Imm(\mathtt{E}_b, \mathtt{E}_i, s)$	$M[Imm + R[\mathtt{E}_b] + R[\mathtt{E}_i] \cdot s]$	Scaled indexed

Operand Specifiers

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

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

Operand	Value
%eax	
0x104	
$0x108	
(%eax)	
4(%eax)	
9(%eax,%edx)	
260(%ecx,%edx)	
0xFC(,%ecx,4)	
(%eax,%edx,4)	

Data Movement Instructions

- Generally used for copy data from one location to another.
- IA32/x86-64 imposes the restriction that a move instruction cannot have both operands refer to memory locations.
- Copying a value from one memory location to another requires two instructions
    - First to load the source value into a register,
    - Second to write this register value to the destination.

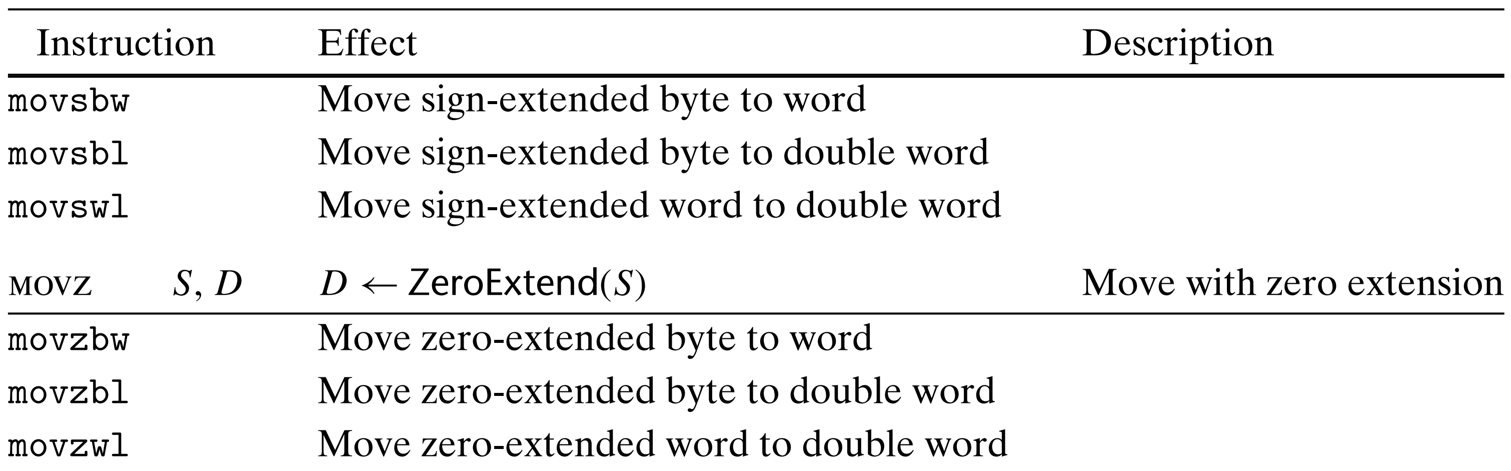
Data Movement Instructions

Instruction		Effect	Description
MOV	S, D	$D \leftarrow S$	Move
`movb`		Move byte	
`movw`		Move word	
`movl`		Move double word	
MOVS	S, D	$D \leftarrow \text{SignExtend}(S)$	Move with sign extension

Data Movement Instructions

```
movl $0x4050,%eax          Immediate--Register, 4 bytes
movw  %bp,%sp              Register--Register,  2 bytes
movb (%edi,%ecx),%ah       Memory--Register,    1 byte
movb $-17,(%esp)           Immediate--Memory,   1 byte
movl %eax,-12(%ebp)        Register--Memory,    4 bytes
```

Data Movement Instructions



Instruction		Effect	Description
`movsbw`		Move sign-extended byte to word	
`movsbl`		Move sign-extended byte to double word	
`movswl`		Move sign-extended word to double word	
MOVZ	S, D	$D \leftarrow$ ZeroExtend(S)	Move with zero extension
`movzbw`		Move zero-extended byte to word	
`movzbl`		Move zero-extended byte to double word	
`movzwl`		Move zero-extended word to double word	

- In case of x86-64, we have additional set of instruction for quad word too, i.e., movsbq, movswq and movslq.
- However, for unsigned copying, only movzbq, movzwq exist as movl already set the high order 4 bytes to zero while copying 4 byte to 8 byte register.

Data Movement Instructions

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

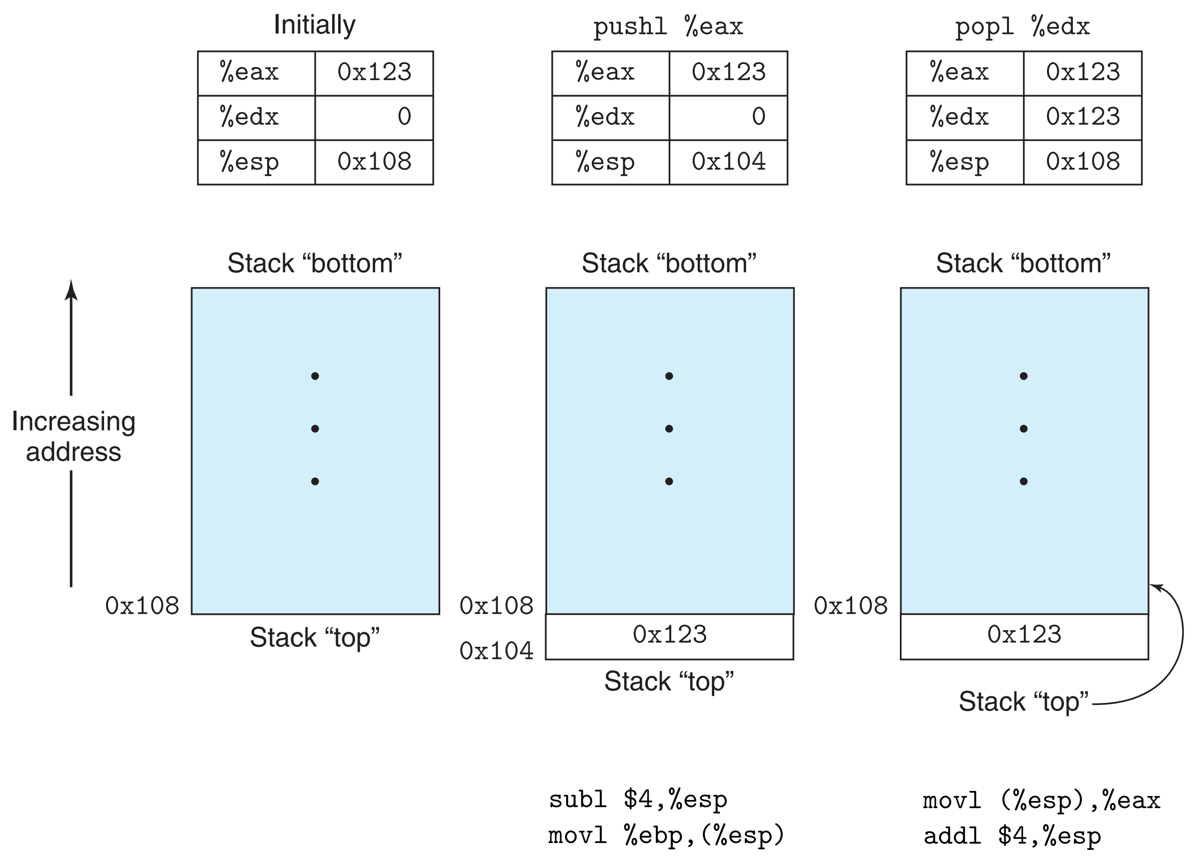
IA 32 Example

Data Movement Instructions

Instruction		Effect	Description
`pushl`	S	$\mathsf{R}[\texttt{\%esp}] \leftarrow \mathsf{R}[\texttt{\%esp}] - 4;$ $\mathsf{M}[\mathsf{R}[\texttt{\%esp}]] \leftarrow S$	Push double word
`popl`	D	$D \leftarrow \mathsf{M}[\mathsf{R}[\texttt{\%esp}]];$ $\mathsf{R}[\texttt{\%esp}] \leftarrow \mathsf{R}[\texttt{\%esp}] + 4$	Pop double word

Data Movement on Stack: Push & Pop

IA 32 Example



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

```
movl (%esp),%eax
addl $4,%esp
```

Data Movement Instructions Example

(a) C code

```c
int exchange(int *xp, int y)
{
    int x = *xp;

    *xp = y;
    return x;
}
```

(b) Assembly code

```
  xp at %ebp+8, y at %ebp+12
    movl    8(%ebp), %edx     Get xp
   By copying to %eax below, x becomes the return value
    movl    (%edx), %eax      Get x at xp
    movl    12(%ebp), %ecx    Get y
    movl    %ecx, (%edx)      Store y at xp
```