CPSC 232
Intro to Assembly Language Programming

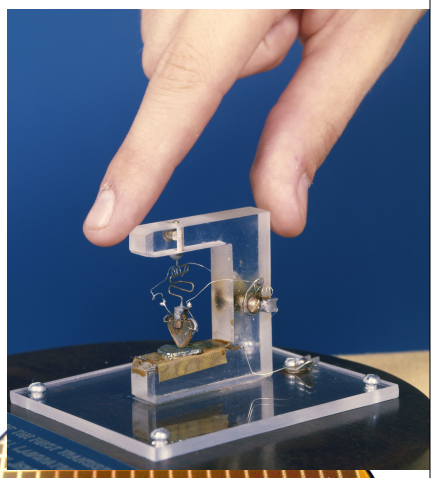
Hyde: Vol 2, Chpt 5 - Instruction Set Architectures

Topics

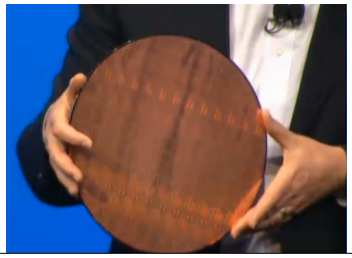
- Instruction Set Design
- Basic microcomputer design
- Instruction execution cycle
- Instruction Design Goals
- Addressing Modes
- Instruction Encoding
- IA-32 Processor Architecture
- IA-32 Memory Management
- Components of an IA-32 Microcomputer
- Input-Output System

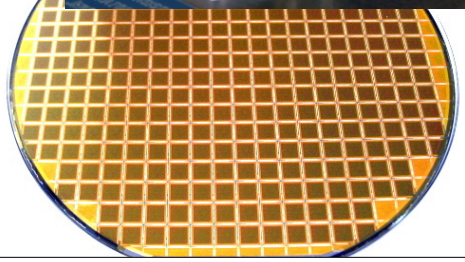
Instruction Set Architectures

- One of the most important design decisions (for CPU)
  - Instruction Set!
  - Impacts many other aspects of chip and system design/implementation



- ISA Design Factors:
  - **Die Real Estate**
    - Limited resource (space on chip die)
  - **Cost**
    - More transistors means more money
  - **Expandability**
    - How anticipate future processor needs?
  - **Legacy Support**
    - Once in ISA, difficult to remove! “Backwards compatibility”





3

3

Instruction Set Architectures

- ISA Design Factors (cont.):
  - **Complexity**
    - Too many features makes for complex ISA
    - Developers do not like complex systems!
  - **Power Consumption**
    - We are mobile, mobile device battery life becomes is important
  - **Heat Generation**
    - More transistors means more power consumption, tightly packed circuits, more heat generation

4

4

Transistor Count

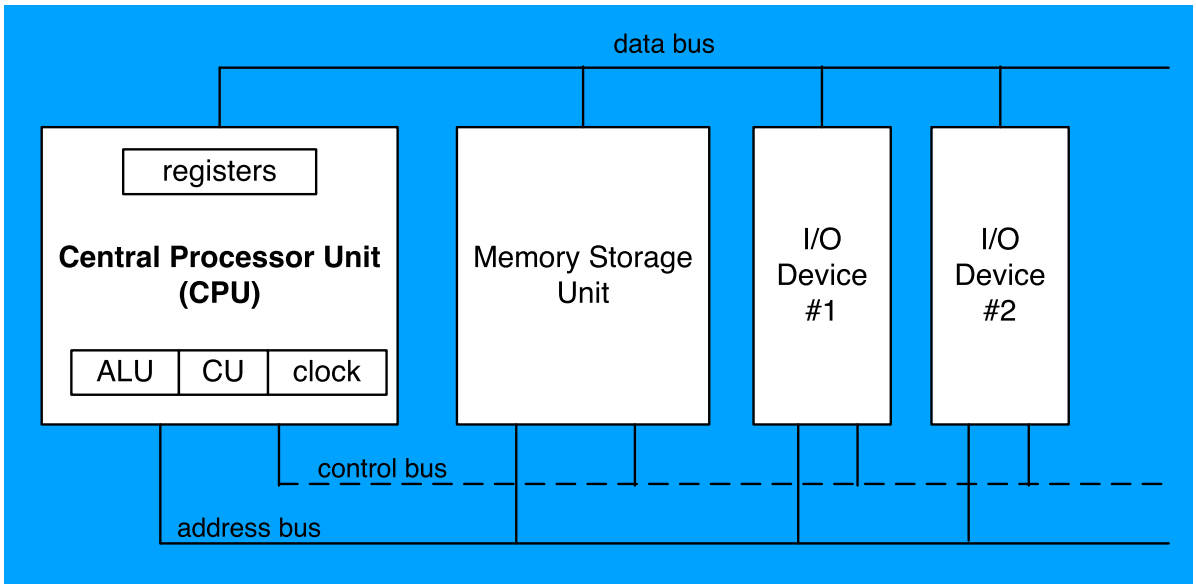
Processor	Year	# Transistors	Feature Size
8086	1978	30 K	3 μm
80186	1982	55 K	1.5 μm
80286	1982	134 K	1.5 μm
80386	1985	275 K	1 μm
80486	1989	1.18 M	800 nm
Pentium	1983	3.1 M	600 nm
Pentium 2	1997	7.5 M	250 nm
Pentium 4	2000	40 M	180 nm
i7 (quad-core)	2008	731 M	45 nm
i7 (6-core)	2010	1.17 B	32 nm
i7 (8-core)	2014	2.6 B	14 nm
Apple A11 (ARM)	2017	4.3 B	10 nm
Apple A14	2020	11.8 B	5 nm

nanometer: 1 billionth of meter

5

Basic Microcomputer Design

- Consider simple block diagram of a computer:
  - Clock synchronizes CPU operations
  - Control unit (CU) coordinates sequence of execution steps
  - ALU performs arithmetic and bitwise processing



6

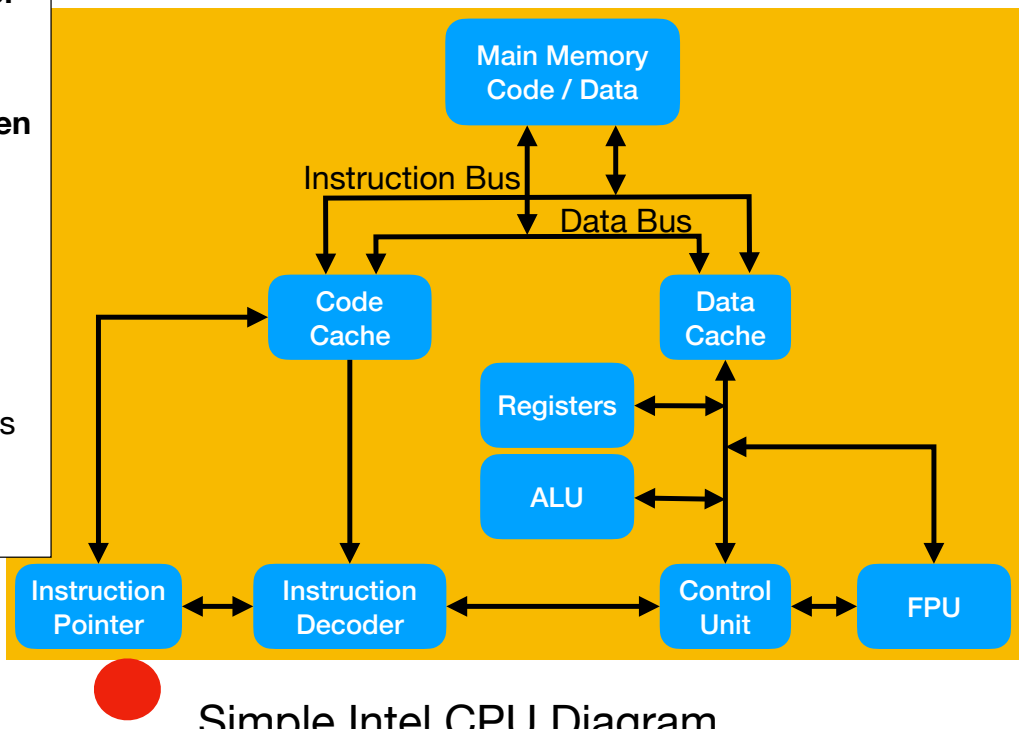
Instruction Execution Cycle

Consider Block Diagram of Intel CPU

Activities happen in cycles

Cycles:

- Fetch
- Decode
- Fetch operands
- Execute
- Store output



Simple Intel CPU Diagram

7

Instruction Decoding

- Recall that assembly is the human-readable form for machine instructions (machine code)
- Machine decodes machine instruction further to microcode or specific operations by the ALU, MMU, etc.
- Question: how does this decoding take place?
- What is the process?!
- Consider machine with 8 operations (opcodes):
  mov, add, sub, mul, div, and, or, xor
- Each takes two operands (2 register names, have 4 registers)
  eax, ebx, ecx, edx
- Example:
  add eax, ebx

8

Opcode Decoding

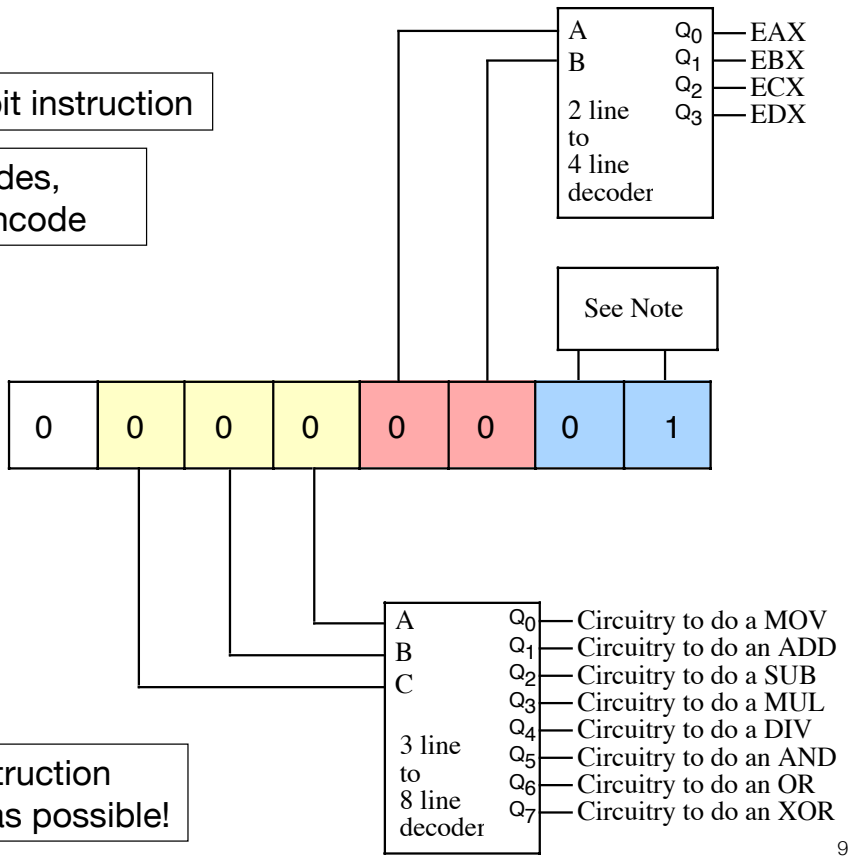
Pretend we have an 8-bit instruction

If have only 8 opcodes,
need only 3 bits to encode

4 registers,
need 2 bits

2 operands,
2 x 2 bits
4 bits total

Important: Want instruction decoding to be simple as possible!

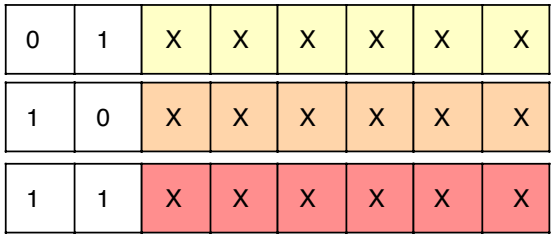


Variable Length Instructions

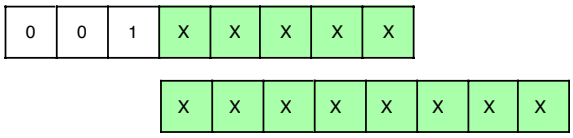
- Sometimes more efficient to not have all instructions be same length
- Need to have variable length instructions:
  - Support more operands
  - Support immediate values/constants
  - Encode more instructions
- Need a scheme to specify how many bits decoder needs to examine for instruction!

Opcode Decoding (Variable Instruction Size)

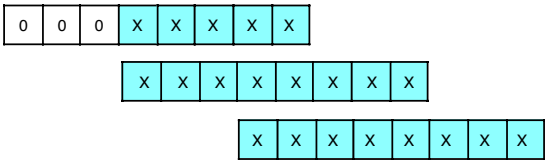
If the H.O. two bits of the first opcode byte are not both zero, then the whole opcode is one byte long and the remaining six bits let us encode 64 one-byte instructions. Since there are a total of three opcode bytes of these form, we can encode up to 192 different one-byte instructions.



If the H.O. three bits of our first opcode byte contain %001, then the opcode is two bytes long and the remaining 13 bits let us encode 8192 different instructions.



If the H.O. three bits of our first opcode byte contain all zeros, then the opcode is three bytes long and the remaining 21 bits let us encode two million (2^{21}) different instructions.



11

11

Variable Length Instructions

- Allow for more efficient space utilization
  - Individual instructions only consume as much space as needed
- Make decoding more complex
  - First determine size, get instruction bits, then decode

12

12

Grouping Instructions

- Similar instructions can be “grouped” together (based upon number of operands, operation performed, etc.)
  - add, sub
  - not
  - jmp
  - call
  - ret
- This can also support more efficient “instruction packing” (encoding assembly instruction in machine code)

“Y86” Instruction Encoding

This is a “pretend” machine

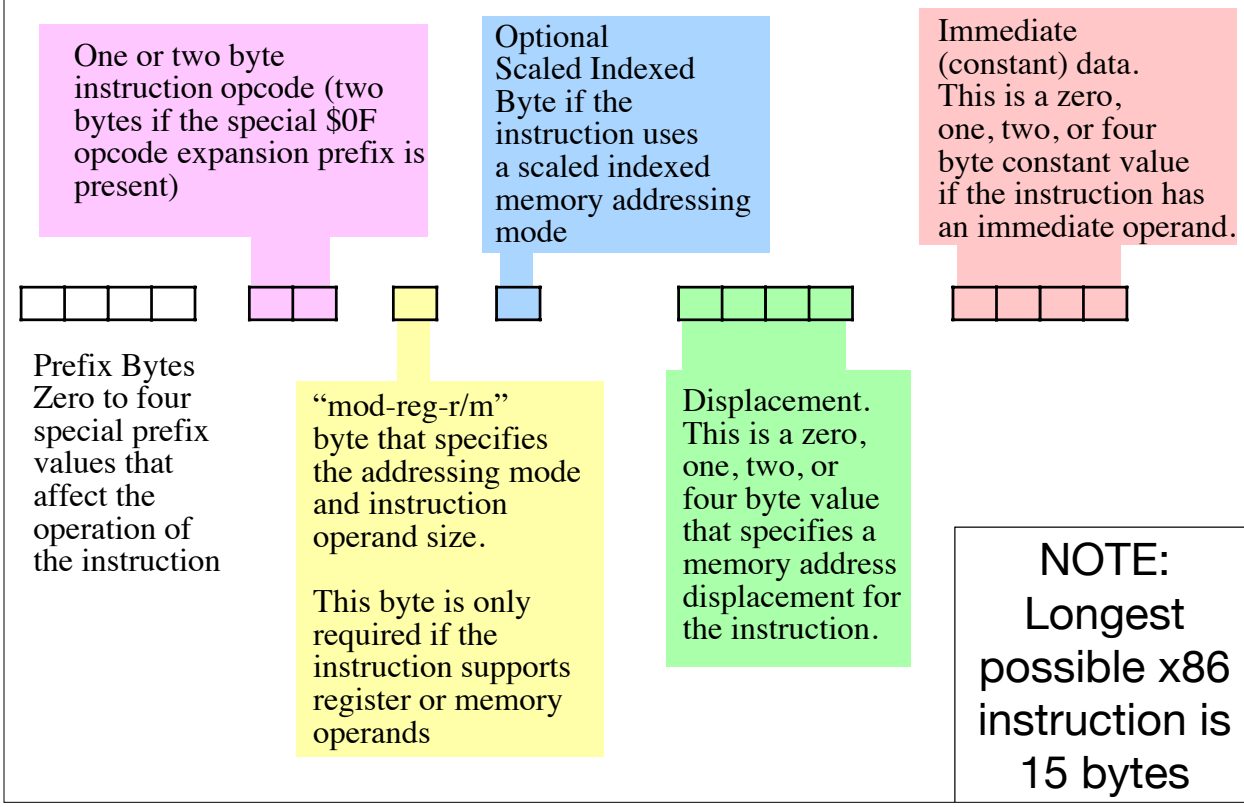
opcode destination source

i i i	r r	mmm
000 = special	00 = AX	0 0 0 = AX
001 = or	01 = BX	0 0 1 = BX
010 = and	10 = CX	0 1 0 = CX
011 = cmp	11 = DX	0 1 1 = DX
100 = sub		1 0 0 = [BX]
101 = add		1 0 1 = [xxxx+BX]
110 = mov(mem/reg/const, reg)		1 1 0 = [xxxx]
111 = mov(reg, mem)		1 1 1 = constant

This 16-bit field is present only if the instruction is a jump instruction or an operand is a memory addressing mode of the form [xxxx+bx], [xxxxx], or a constant.

x86 Instruction Encoding

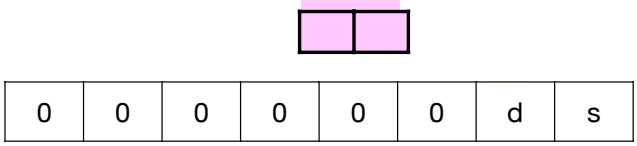
We will look at each of these fields over next slides



NOTE: Longest possible x86 instruction is 15 bytes

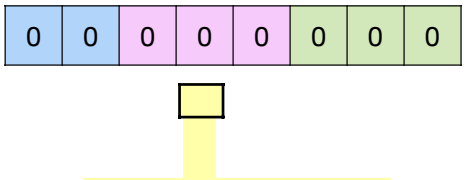
Opcodes

One or two byte instruction opcode (two bytes if the special $0F opcode expansion prefix is present)



- Add opcode:
  d=0: add from register to memory
  d=1: add from memory to register
  s=0: add eight-bit operands
  s=1: add 16-bit or 32-bit operands

MOD-REG-R/M Field



- MOD: Specifies if R/M provides register, memory (2 bits)
- REG: Specifies register address (3 bits)
- R/M: Specifies memory, indirect, or register address (3 bits)

17

REG Decoding

REG Value	Register if data size is eight bits	Register if data size is 16-bits	Register if data size is 32 bits
%000	al	ax	eax
%001	cl	cx	ecx
%010	dl	dx	edx
%011	bl	bx	ebx
%100	ah	sp	esp
%101	ch	bp	ebp
%110	dh	si	esi
%111	bh	di	edi

18

MOD Decoding

MOD	Meaning
%00	Register indirect addressing mode or SIB with no displacement (when R/M=%100) or Displacement only addressing mode (when R/M=%101).
%01	One-byte signed displacement follows addressing mode byte(s).
%10	Four-byte signed displacement follows addressing mode byte(s).
%11	Register addressing mode.

19

MOD-R/M Decoding

MOD	R/M	Addressing Mode
%00	%000	[eax]
%01	%000	[eax+$disp_8$]
%10	%000	[eax+$disp_{32}$]
%11	%000	register (al/ax/eax)[a]
%00	%001	[ecx]
%01	%001	[ecx+$disp_8$]
%10	%001	[ecx+$disp_{32}$]
%11	%001	register (cl/cx/ecx)
%00	%010	[edx]
%01	%010	[edx+$disp_8$]
%10	%010	[edx+$disp_{32}$]
%11	%010	register (dl/dx/edx)
%00	%011	[ebx]
%01	%011	[ebx+$disp_8$]
%10	%011	[ebx+$disp_{32}$]
%11	%011	register (bl/bx/ebx)

20

MOD-R/M Decoding

MOD	R/M	Addressing Mode
%00	%100	SIB Mode
%01	%100	SIB + $disp_8$ Mode
%10	%100	SIB + $disp_{32}$ Mode
%11	%100	register (ah/sp/esp)
%00	%101	Displacement Only Mode (32-bit displacement)
%01	%101	[ebp+$disp_8$]
%10	%101	[ebp+$disp_{32}$]
%11	%101	register (ch/bp/ebp)
%00	%110	[esi]
%01	%110	[esi+$disp_8$]
%10	%110	[esi+$disp_{32}$]
%11	%110	register (dh/si/esi)
%00	%111	[edi]
%01	%111	[edi+$disp_8$]
%10	%111	[edi+$disp_{32}$]
%11	%111	register (bh/di/edi)

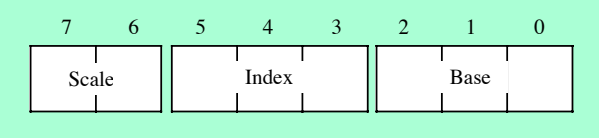
Instruction Prefix Bytes

- Up to four bytes (four groups):
  - Group 1: Lock & repeat
    - LOCK = 0xF0
    - REPNE/REPNZ = 0xF2
    - REPE/REPZ = 0xF3
  - Group 2: Segment override (Specify segment for memory access)
    - CS = 0x2E
    - DS = 0x3E
    - ES = 0x26
    - FS = 0x64
    - GS = 0x65
    - SS = 0x36
  - Group 2: Branch hints (jump with condition code)
    - Predict branch not taken (0x2E)
    - Predict branch taken (0x3E)
  - Group 3: Operand size prefix
    - Specify non-default operand size for 16 or 32 bit operands (0x66)
  - Group 4: Address size prefix
    - Specify non-default address size for 16 or 32 bit address (0x67)

SIB Mode

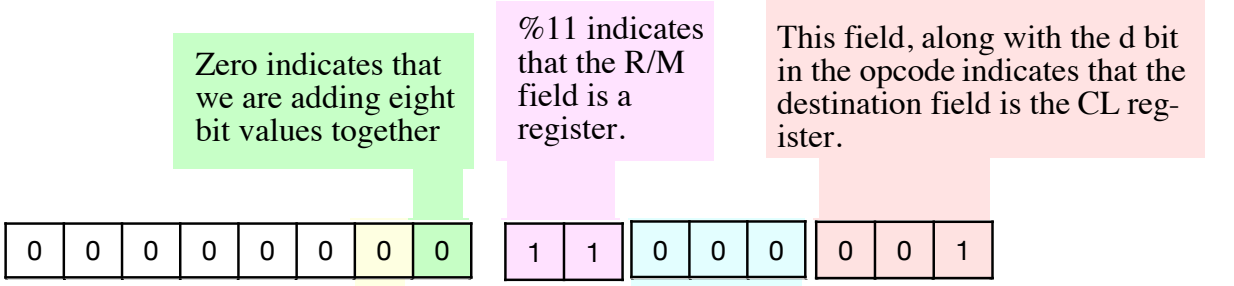
Scale Value	Index*Scale Value
%00	Index*1
%01	Index*2
%10	Index*4
%11	Index*8

Index	Register
%000	EAX
%001	ECX
%010	EDX
%011	EBX
%100	Illegal
%101	EBP
%110	ESI
%111	EDI



Base	Register
%000	EAX
%001	ECX
%010	EDX
%011	EBX
%100	ESP
%101	Displacement-only if MOD = %00, EBP if MOD = %01 or %10
%110	ESI
%111	EDI

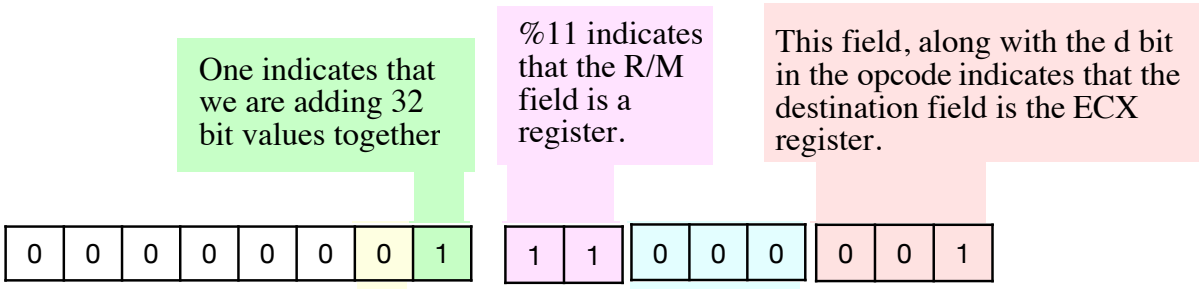
Back to that Y86 Add Instruction!



ADD(al, cl) = $00, $C1

Note: this is MASM syntax!
NASM would be:
`add cl, al`

Back to that Add Instruction!



One indicates that we are adding 32 bit values together

%11 indicates that the R/M field is a register.

This field, along with the d bit in the opcode indicates that the destination field is the ECX register.

%000000 indicates that this is an ADD instruction.

Zero indicates that we are adding the REG field to the R/M field.

This field, along with the d bit in the opcode, indicates that the source field is the EAX register.

ADD(eax, ecx) = $01, $C1

Note: this is MASM syntax!
NASM would be:

```
add ecx, eax
```


IA-32 Processor Architecture

- Modes of operation
- Basic execution environment
- Floating-point unit
- Intel Microprocessor history

Modes of Operation

- Protected mode
  - native mode (Windows, Linux)
  - We will work & play here
- Real-address mode
  - native MS-DOS
- System management mode
  - power management, system security, diagnostics
- Virtual-8086 mode
  - Hybrid of Protected each program has its own 8086 computer

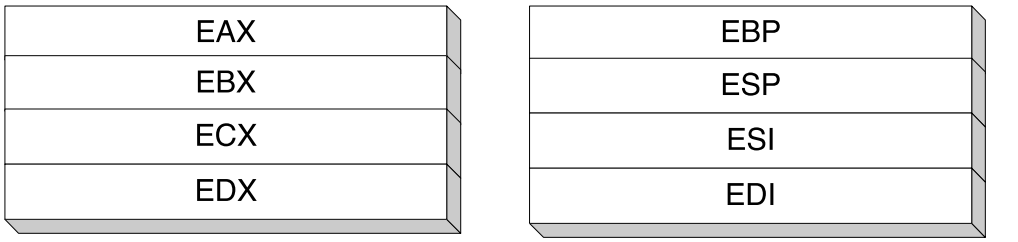
Basic Execution Environment

- General-purpose registers
- Index and base registers
- Specialized register uses
- Status flags
- Floating-point, MMX, XMM registers
- Addressable memory

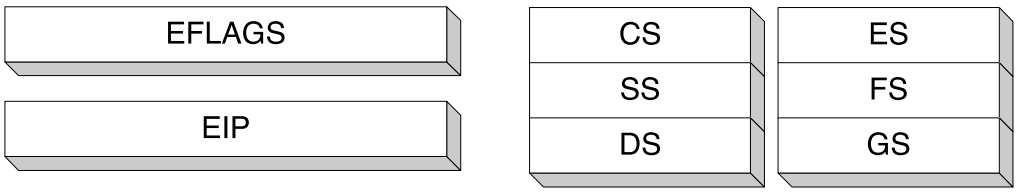
General-Purpose Registers

Named storage locations inside the CPU, optimized for speed.

32-bit General-Purpose Registers

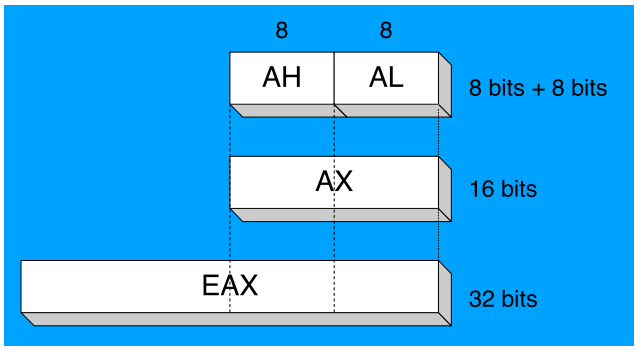


16-bit Segment Registers



Accessing Parts of Registers

- Use 8-bit name, 16-bit name, or 32-bit name
- Applies to EAX, EBX, ECX, and EDX



32-bit	16-bit	8-bit (high)	8-bit (low)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

Index and Base Registers

- Some registers have only a 16-bit name for their lower half:
- (there is no bh, bl, sh, sl. . .)

32-bit	16-bit
ESI	SI
EDI	DI
EBP	BP
ESP	SP

Some Specialized Register Uses (1 of 2)

- General-Purpose:
  - EAX – accumulator
  - ECX – loop counter
  - ESP – stack pointer
  - ESI, EDI – source/destination index registers (used for arrays, etc.)
  - EBP – extended frame pointer (stack base pointer)

- Segment Registers:
  - CS – code segment
  - DS – data segment
  - SS – stack segment
  - ES, FS, GS - additional segments

Some Specialized Register Uses (2 of 2)

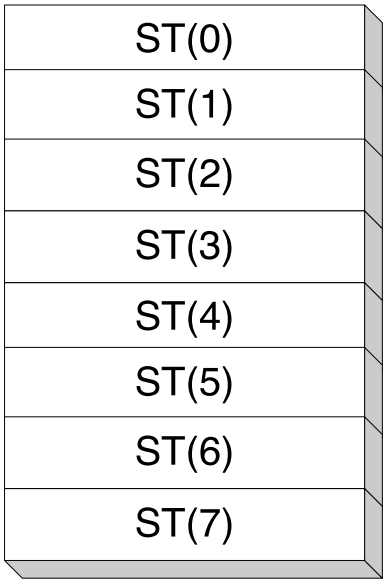
- EIP – instruction pointer
- EFLAGS
  - Status and control flags
  - Each flag is a single binary bit
  - Captures results of certain math/logic operations

EFlags: Status Flags

- Carry
  - Unsigned arithmetic out of range
- Overflow
  - Signed arithmetic out of range
- Sign
  - Result is negative
- Zero
  - Result is zero
- Auxiliary Carry
  - Carry from bit 3 to bit 4 (low nibble to high nibble)
- Parity
  - Sum of “1” bits is an even number

Floating-Point, MMX, XMM Registers

- Eight 80-bit floating-point data registers
  - ST(0), ST(1), . . . , ST(7)
  - Arranged in a stack
  - Used for all floating-point arithmetic
- Eight 64-bit MMX registers for single-instruction multiple-data (SIMD) operations
  - Designed to support graphics calculations
  - Not used so much anymore
  - Dirty secret: MMX regs are aliases for mantissa portion of FP registers (ST0-ST7)
- Eight 128-bit XMM registers
  - Used in graphics calculations
  - Used in Intel SSE (supercedes MMX)
  - Streaming SIMD Extensions
  - Truly separate registers



A Word on CISC and RISC

- CISC – complex instruction set
  - large instruction set (large # of instructions)
  - high-level operations (more powerful instructions)
    - instruction does more
    - often takes more time for instruction to complete execution
  - often requires microcode interpreter
  - examples: Intel 80x86 family
- RISC – reduced instruction set
  - small instruction set (usually fewer instructions)
  - simple, atomic instructions
    - instruction does less
    - often complete execution in one time cycle
  - directly executed by hardware
  - examples:
    - ARM (Advanced RISC Machines)
    - DEC Alpha (now Compaq, now HP)

IA-32 Memory Management

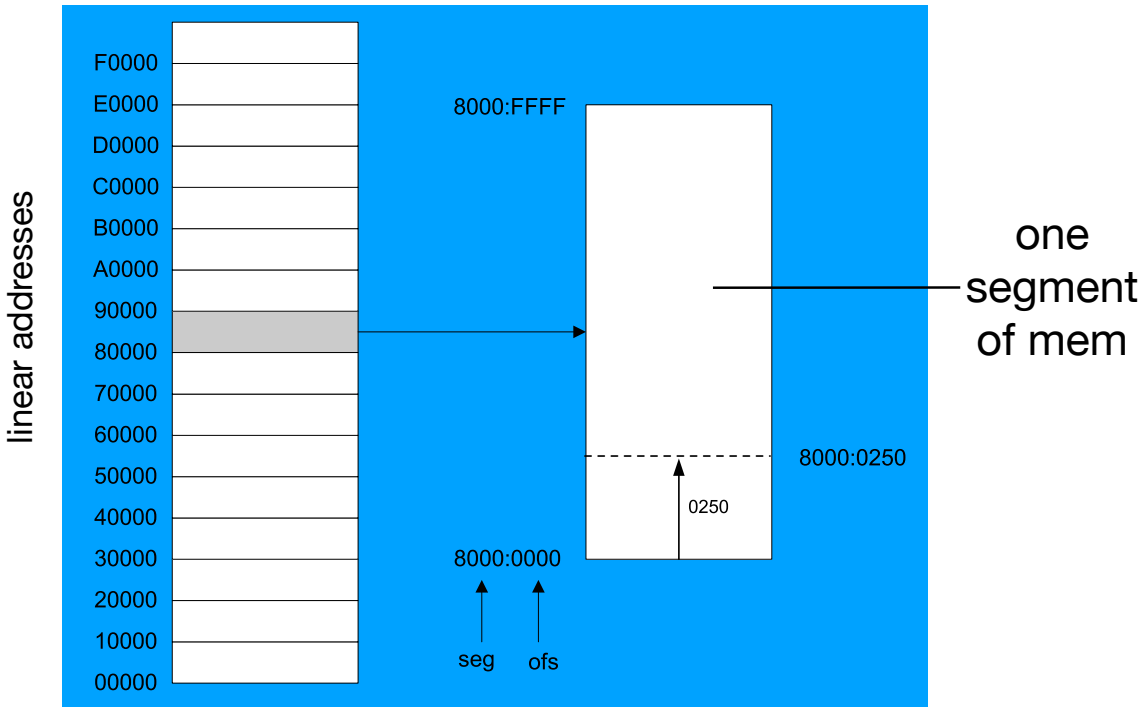
- Real-address mode
  - 20-bit address
  - 1 MB RAM space
  - Calculating linear addresses in real-address mode
- Protected mode
  - Introduced by Intel 80286
  - 32-bit address (386)
    - 4 GB RAM space (386)
  - Introduced segments (code, stack, data)
  - Introduced virtual memory & paging
  - (remember, we work & play here)
- Multi-segment model
- Paging

Real-Address mode

- 1 MB RAM maximum addressable
  - 20-bit address yields 1MB RAM
  - 80286 has 16-bit registers, 20-bit address, how?!
    - Split address into “selector” & offset (2 16-bit values)
- Application programs can access any area of memory
- Single tasking
- Supported by MS-DOS operating system
- BIOS usually runs in real mode, too

Segmented Address

- Segmented memory addressing: absolute (linear) address is a combination of a 16-bit selector value added to a 16-bit offset



39

39

Calculating Linear Addresses

- 20-bit address: selector value + offset
  - Selector stored in segment register (16-bit)
  - Offset stored in any general purpose register (16-bit)

- Given a selector value, multiply it by 16 (add a hexadecimal zero), and add it to the offset — This gives us our 20-bit address

- Example: convert 08F1:0100 to a linear address

Adjusted selector value:	0 8 F 1 0
Add the offset:	0 0 1 0 0
Linear address:	0 9 0 1 0

40

Your turn . . .

What linear address corresponds to the segment/offset address 028F:0030?

028F0 + 0030 = 02920

(remember, add hex zero to selector value)
Always use hexadecimal notation for addresses.

Your turn . . .

What segment addresses correspond to the linear address 28F30h?

Strange side-effect: Many different segmented addresses can produce the linear address 28F30h.

For example:
28F0:0030, 28F3:0000, 28B0:0430, . . .

386 Protected Mode (1 of 2)

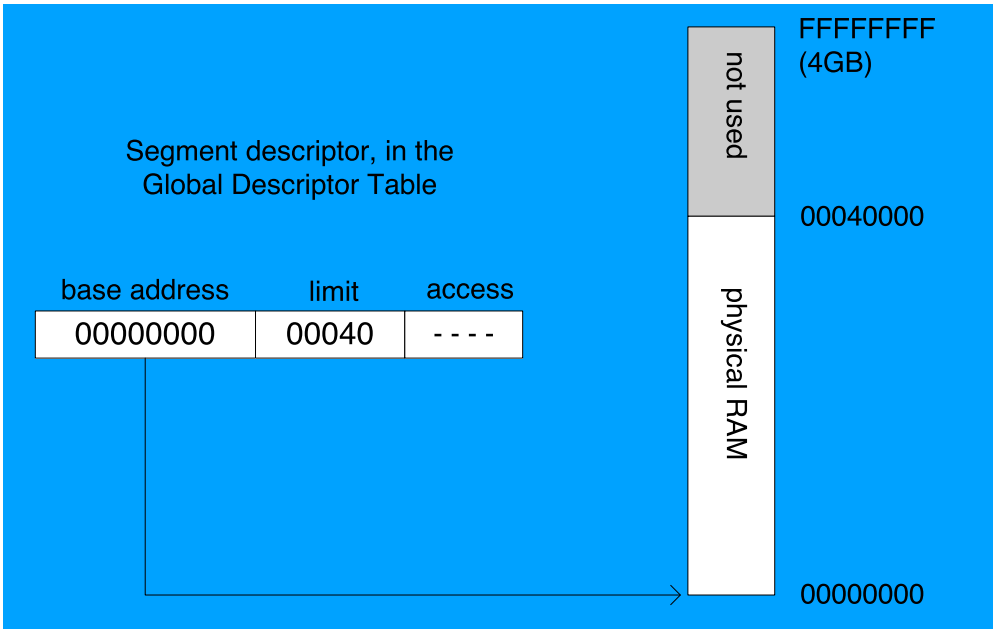
- 4 GB addressable RAM
  - 32-bit address
  - (00000000 to FFFFFFFF)
- Each program assigned a memory partition which is protected from other programs
- Designed for multitasking
- Supported by Linux & MS-Windows

Protected mode (2 of 2)

- Segment descriptor tables
- Program structure
  - code, data, and stack areas
  - CS, DS, SS segment descriptors
  - global descriptor table (GDT)
    - Gives us ability to have relocatable code
    - Look up segment & get real base offset within memory
- MASM Programs use the Microsoft flat memory model

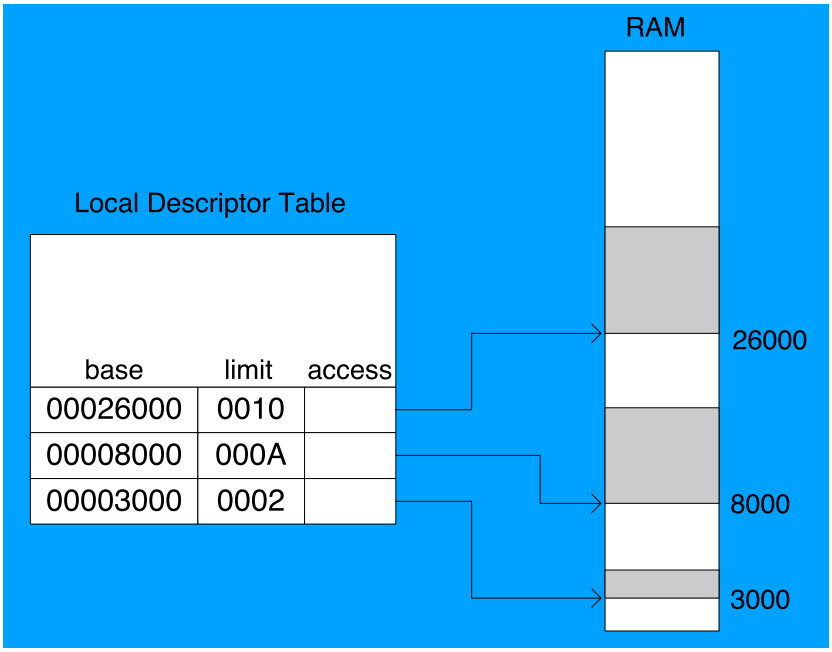
Flat Segment Model

- Single global descriptor table (GDT).
- All segments mapped to entire 32-bit address space



Multi-Segment Model

- Each program has a local descriptor table (LDT)
  - Holds descriptor for each segment used by the program



Paging

- Supported directly by the CPU
- Divides each segment into 4096-byte blocks called pages
- Sum of all programs can be larger than physical memory
- Part of running program is in memory, part is on disk
- Virtual memory manager (VMM) – OS utility that manages the loading and unloading of pages
- Page fault – issued by CPU when a page must be loaded from disk