ECE291

Lecture 2

All about memory

Lecture outline

- Addressing memory
- Data types
- The MOV instruction
- Addressing modes
- Instruction format and the mode byte

Terminology

- Bit: the fundamental, indivisible data unit in computers. Either a 0 or a 1.
- Nibble: 4 bits
- Byte: 8 bits
- Word: 2 bytes
- Double word: 4 bytes
- Quad word: 8 bytes

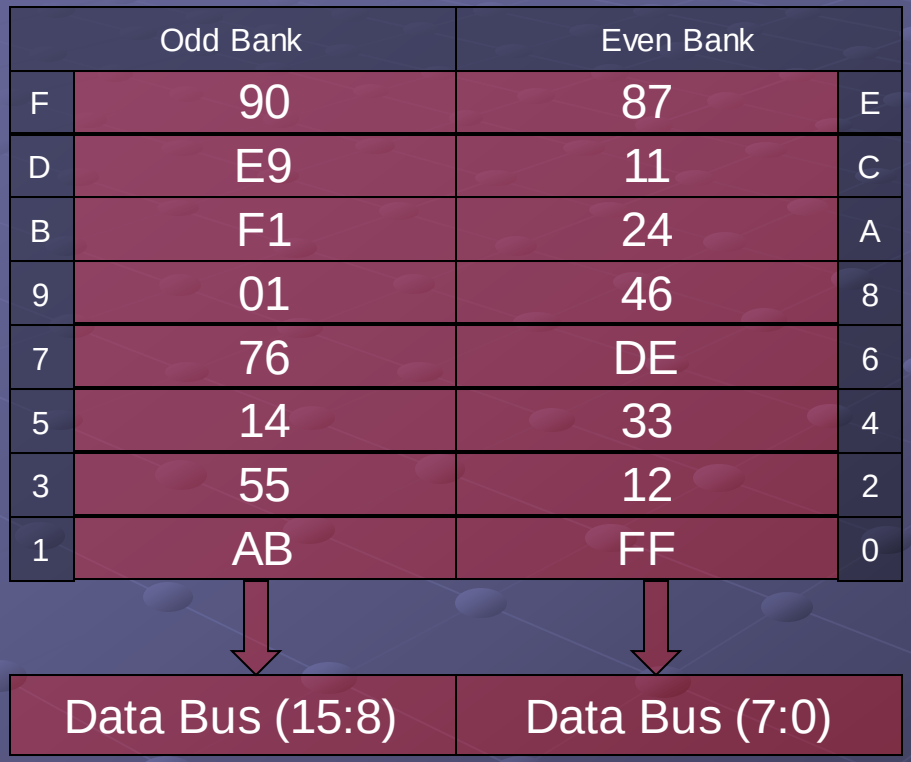
Memory

- The microprocessor can address a maximum of 2^n different memory locations, where n is the number of bits on the address bus
- Logical Memory
  - 80x86 exclusively supports byte addressable memory
  - Byte is the basic memory unit
  - When you specify address 24 in memory, you get the entire eight bits at memory location 24.
  - When the microprocessor addresses a 16-bit word of memory, two consecutive bytes are accessed.

Memory

- Physical memory
  - The physical memories of 80x86's differ in width
    - 8088 memory is 8 bits wide
    - 8086, 80286 memory is 16 bits wide
    - 80386 and above is 32 bits wide
  - For programming there is no difference in memory width because the logical memory is always 8-bits
  - Memory is organized in memory banks
    - A memory bank is an 8-bit wide section of memory
    - The 16-bit microprocessors contain two memory banks to form a 16-bit wide section of memory

Memory organization

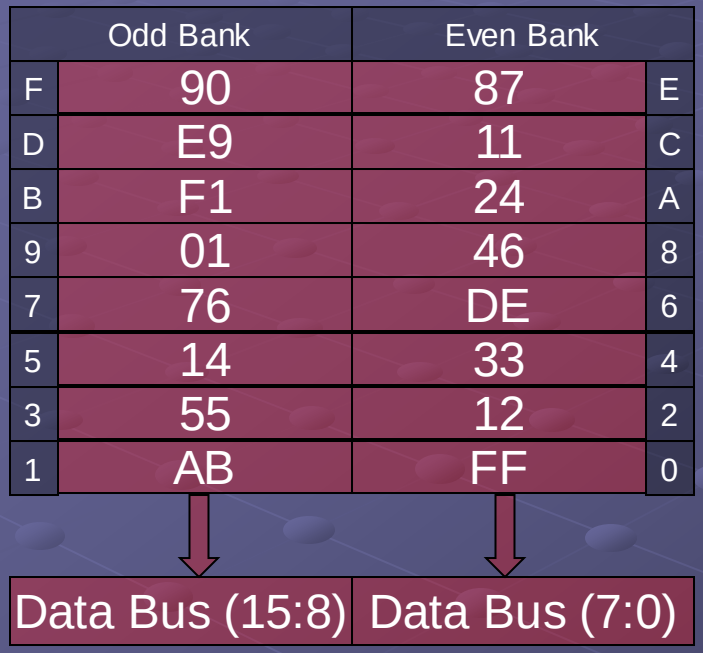


- Always byte addressable regardless of data size
- Read byte from 0
  - Ans = FF
- Read word from 0
  - Ans = ABFF
- Read double word from 0
  - Ans = 5512ABFF

Byte order

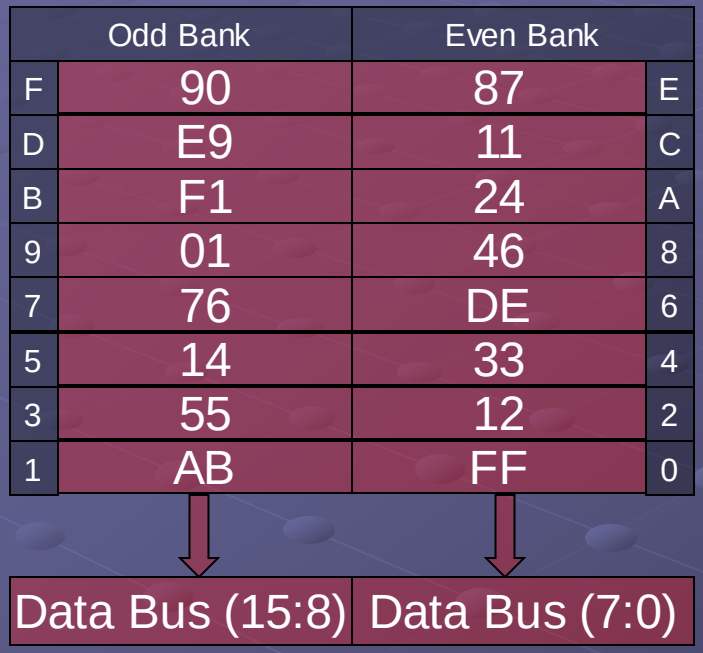
- Why was word[0] = ABFF and not FFAB?
- Depends on processor architecture
- X86 uses “Little Endian” notation
  - Requested address points to low order byte
  - Higher order bytes come from the next higher sequential addresses
- Many UNIX-based systems use “Big Endian” instead. Purely a function of processor architecture.

Byte alignment example



- What happens when we read word[0]?
- Data(15:8) <- AB
- Data(7:0) <- FF
- Great! High byte is where it's supposed to be!

Byte alignment example 2



- What happens when we read word[1]?
- Result *should* be 12AB
- Data(15:8) <- AB
- Data(7:0) <- 12
- Then we have to swap
- This is bad. Why?

Byte alignment

- Processor can read a row of memory at a time. In a 16-bit configuration, processor can read a whole word in one access.
- BUT, that word must all be on one row.
- Reading from word[1] means the processor first reads the upper half of the row containing address 1.
- Then it reads the lower half of the row containing address 2.
- But the bytes are in the wrong order so they have to be swapped.
- In the bad old days, you the programmer had to do this in your code. It's handled automatically now.
- But even so, you should avoid this situation. Why?

Byte alignment

- Steps for reading unaligned words
  - Read odd byte
  - Read even byte
  - Swap bytes for proper byte order
- Steps for reading aligned words
  - Read word from even address!
- Much faster for aligned words

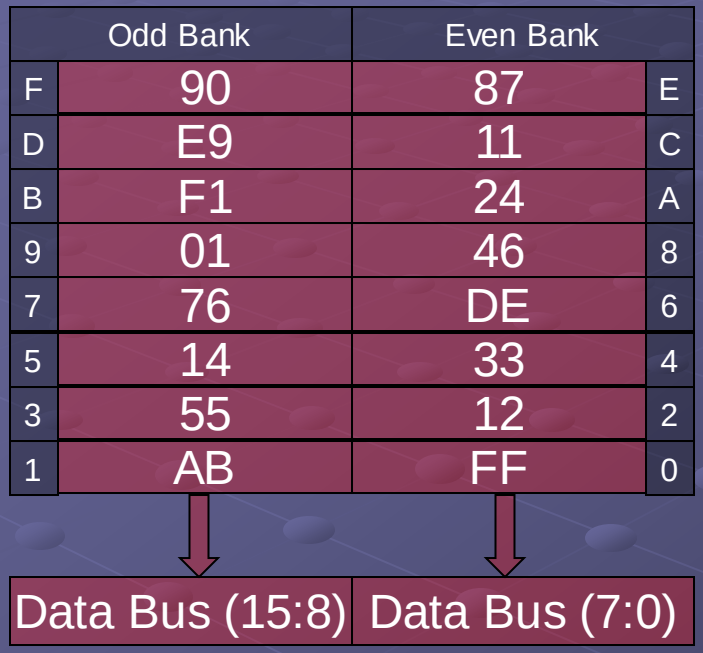
Data types

- Numbers
  - Bits, nibbles, bytes, words, double words, etc.
  - Unsigned/Signed
- Text
  - Letters and characters (7-bit ASCII standard)
    - E.g. ‘A’ = 65 = 41h
  - Extended ASCII (8-bit) allows for extra 128 graphical and symbolic characters
  - Strings: collection of characters
  - Documents: collection of strings

Data types

- Arrays of numbers or characters
- Floating point numbers (covered later)
- Images (bitmaps, gifs, tifs, jpgs)
- Video (MPEG, QuickTime, avi, asf)
- Audio (wave files, mp3's)
- Programs
  - Commands (MOV, JMP, AND, OR)
  - Collection of commands = subroutines
  - Collection of subroutines = programs

Data types



- To the computer, it's always 1's and 0's.
- What data is depends on how you the programmer uses it.
- Addresses 0 through F could be an array of 16 1-byte unsigned integers
- They could also be 4 double-word signed integers

Memory example

Address	Data	Interpretation	
60511h	12h	Word 1234h	3x1 integer array
60510h	34h		
6050Fh	34h	Word 3412h	
6050Eh	12h		
6050Dh	A5h	Word A55Ah	
6050Ch	5Ah		
55004h	FCh	JE .begin	
55003h	74h		
55002h	02h	.begin ADD AL, 2	
55001h	04h		
00000h		Bottom of address space	

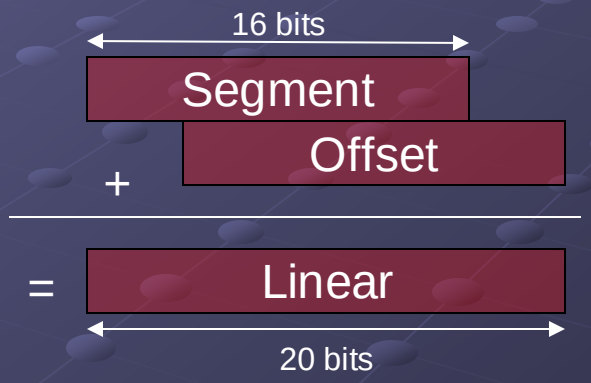
Address	Data	Interpretation
FFFFFh		Top of address space
75000h	55h	Byte
70009h	24h = '$'	String "ECE 291$"
70008h	31h = '1'	
70007h	39h = '9'	
70006h	32h = '2'	
70005h	20h = ' '	
70004h	45h = 'E'	
70003h	43h = 'C'	
70002h	45h = 'E'	

Real mode addressing

- 80286 and above operate in real or protected mode
- 8086, 8088, and 80186 only operate in real mode
- Real mode operation allows the microprocessor to address only the first 1MB of the memory space (even if on a Pentium computer)
- You are limited to a 20-bit address bus
- All real mode memory addresses consist of a *segment address* plus an *offset address*.
  - **The segment address** in one of the segment registers) defines the beginning address of any 64KB memory segment
  - **The offset address** selects a location within the 64KB memory segment

Real mode addressing

- Segment:offset combine to form a 20-bit linear or physical address.
- In real mode, each segment register (16-bits) is internally appended with a 0h on its rightmost end (i.e., the segment is shifted left by 4 bits).
- The segment and the offset are then added to form a 20-bit linear memory address.



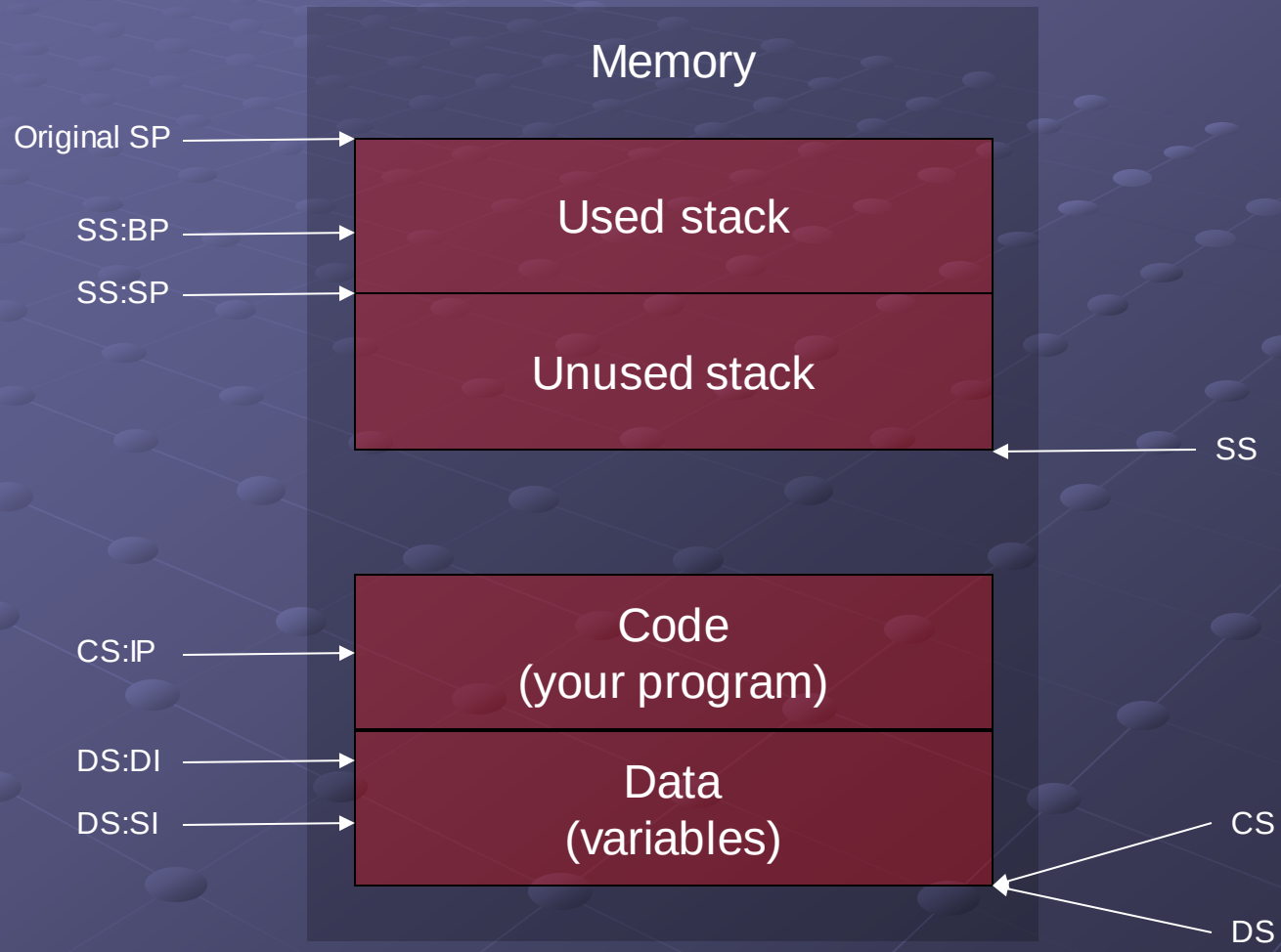
Examples

- Find the linear address for 2222h:3333h
  - 22220h + 03333h = 25553h
- Find the linear address for 2000h:5553h
  - 20000h + 05552h = 25553h
- Many different segment:offset address combinations map to the same linear address

Protected mode in brief

- In 80286's and above, addressing capabilities are extended by changing the function the CPU uses to convert a logical address to the linear address space.
  - PMode processors use an array to compute the physical address
  - Segment value is used as an index into this array (the segment descriptor table)
  - Contents of the selected array element provide the starting address for the segment
  - The CPU adds this value to the offset to obtain the physical address
- It's actually a bit more complicated.

Use of segments



The MOV instruction

- MOV moves data from one place to another
  - register to register
  - Register to memory
  - Memory to register
  - Immediate data to register
  - NEVER memory to memory
- Syntax: MOV destination, source
  - Destination and source can be many different things or combination of things.
  - What these things are determines the Addressing Mode.

Addressing modes

- ***Register*** – transfers a byte or word from the source register to the destination register

MOV BX, CX

- ***Immediate*** – transfers an immediate byte or word of data into the destination register or memory location

MOV AX, 3456h

- ***Direct*** – moves a byte or word between a memory location specified in the instruction and a register

MOV AL, [1234h]

(1234h is treated as a displacement within the data segment)

Addressing modes

- ***Register indirect*** (*base relative* or *indexed*) – transfers a byte or word of data between a register and the memory location addressed by an index (DI or SI) or base register (BP or BX)

MOV AX, [BX]

- ***Base plus index*** (*base relative indexed*) – transfers a byte or word of data between a register and the memory location addressed by a base register (BP or BX) *plus* index (DI or SI) register

MOV DX, [BX + DI]

Addressing modes

- ***Register relative*** – transfers a byte or word of data between a register and the memory location addressed by an index (DI or SI) or base register (BP or BX) plus a constant displacement

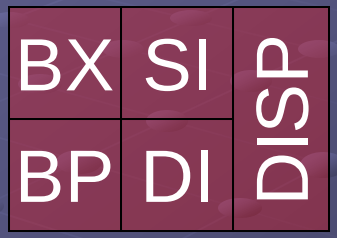
MOV AX, [BX + 1000h]

- ***Base relative plus index*** – transfers a byte or word of data between a register and the memory location addressed by a base register (BP or BX) plus an index register (DI or SI) *plus* a displacement

MOV DX, [BP + SI + 1000h]

Addressing modes

- If you learn one thing about addressing modes, it is that you can reference memory with:



- Any above register (BX, BP, SI, DI) or an immediate displacement (DISP)
- Sum of either column 1 register with either column 2 register (BX+SI, BX+DI, BP+SI, BP+DI) with immediate displacement (DISP)
- Never with the sum of registers in the same column (not BX+BP or SI+DI)

Machine language

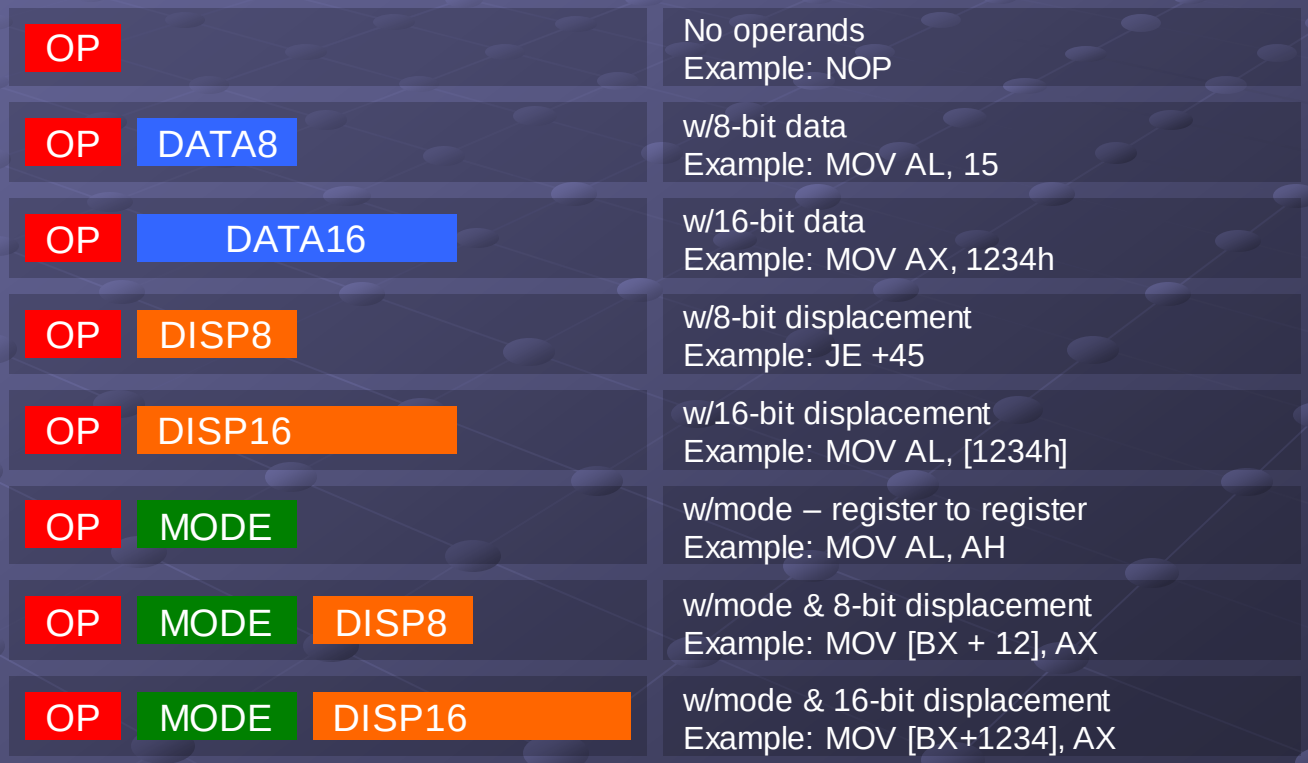
- Machine language is the native binary code that the CPU understands and uses as the instructions that control its operation.
- Interpretation of machine language allows debugging or modification at the machine language level.
- Machine language instructions are generally too complex to generate by hand. That's why we use NASM.
- 80x86 machine language instructions vary in length from 1 to as many as 13 bytes.
- There are over 20,000 variations of machine language instructions.

Machine language

- Real mode uses 16-bit instructions
  - This means that instructions use 16-bit offset addresses and 16-bit registers
  - This does *not* mean the instructions are 16-bits in length
- Protected mode can use 16 or 32 bit instructions
  - The 32-bit instruction mode assumes all offset addresses are 32 bits as well as all registers

Instruction format

Opcode	Mode	Displacement	Data/Immediate



Addressing modes

- Each instruction can only access memory once
  - MOV [VAR1], [VAR2] is invalid!
  - MOV AX, [VAR2] followed by MOV [VAR1], AX is okay
- For 2-operand instructions, size of operands must match
  - Compare 8-bit numbers to 8-bit numbers
  - Compare 16-bit numbers to 16-bit numbers
  - CMP AH, AX is invalid!
- Mode byte encodes which registers the instruction uses
- When writing instructions, if data sizes aren't obvious from the context, you must explicitly state the size.
  - MOV BYTE [BX], 12h
  - MOV [BX], WORD 12h

Addressing mode examples

Instruction	Comment	Addressing Mode	Memory Contents
MOV AX, BX	Move to AX the 16-bit value in BX	Register	89 D8 OP MODE
MOV AX, DI	Move to AX the 16-bit value in DI	Register	89 F8 OP MODE
MOV AH, AL	Move to AL the 8-bit value in AX	Register	88 C4 OP MODE
MOV AH, 12h	Move to AH the 8-bit value 12H	Immediate	B4 12 OP DATA8
MOV AX, 1234h	Move to AX the value 1234h	Immediate	B8 34 OP DATA16
MOV AX, CONST	Move to AX the constant defined as CONST	Immediate	B8 lsb msb OP DATA16
MOV AX, X	Move to AX the address or offset of the variable X	Immediate	B8 lsb msb OP DATA16
MOV AX, [1234h]	Move to AX the value at memory location 1234h	Direct	A1 34 12 OP DISP16
MOV AX, [X]	Move to AX the value in memory location DS:X	Direct	A1 lsb msb OP DISP16

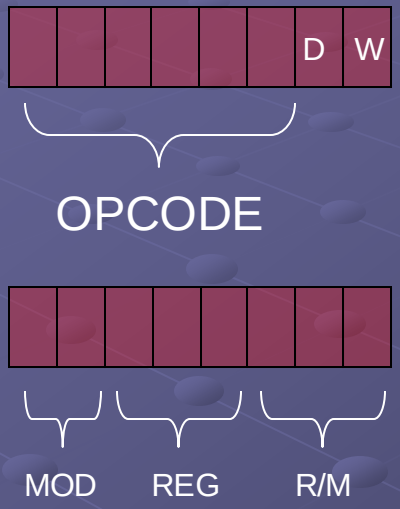
Addressing mode examples

Instruction	Comment	Addressing Mode	Memory Contents
MOV [X], AX	Move to the memory location pointed to by DS:X the value in AX	Direct	A3 lsb msb OP DATA16
MOV AX, [DI]	Move to AX the 16-bit value pointed to by DS:DI	Indexed	8B 05 OP MODE
MOV [DI], AX	Move to address DS:DI the 16-bit value in AX	Indexed	89 05 OP MODE
MOV AX, [BX]	Move to AX the 16-bit value pointed to by DS:BX	Register Indirect	8B 07 OP MODE
MOV [BX], AX	Move to the memory address DS:BX the 16-bit value stored in AX	Register Indirect	89 07 OP MODE
MOV [BP], AX	Move to memory address **SS:BP** the 16-bit value in AX	Register Indirect	89 46 OP MODE
MOV AX, TAB[BX]	Move to AX the value in memory at DS:BX + TAB	Register Relative	8B 87 lsb msb OP MODE DISP16
MOV TAB[BX], AX	Move value in AX to memory address DS:BX + TAB	Register Relative	89 87 lsb msb OP MODE DISP16
MOV AX, [BX + DI]	Move to AX the value in memory at DS:BX + DI	Base Plus Index	8B 01 OP MODE

Addressing mode examples

Instruction	Comment	Addressing Mode	Memory Contents
MOV [BX + DI], AX	Move to the memory location pointed to by DS:X the value in AX	Base Plus Index	89 01 OP MODE
MOV AX, [BX + DI + 1234h]	Move word in memory location DS:BX + DI + 1234h to AX register	Base Rel Plus Index	8B 81 34 12 OP MODE DISP16
MOV word [BX + DI + 1234h], 5678h	Move immediate value 5678h to memory location BX + DI + 1234h	Base Rel Plus Index	C7 81 34 12 78 56

Instruction components



- The OPCODE byte contains the opcode, as well as direction (D) and data size (W) bits.
- The MODE byte only exists in instructions that use register addressing modes.
- The MODE byte encodes the target and source for 2-operand instructions.
- The target and source are specified in the REG and R/M fields.

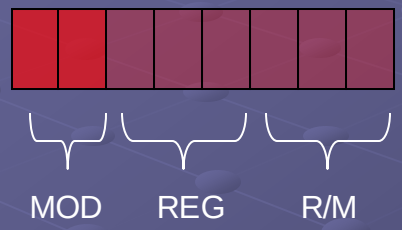
Instruction components

- OPCODE (one or two bytes) selects the operation (e.g., addition, subtraction, move) performed by the microprocessor

- D – direction of data flow
  - D = 0 Data flow REG -> R/M
  - D = 1 Data flow R/M -> REG
- W – data size
  - W = 0 data is byte sized
  - W = 1 data is word sized or double word sized depending on Real vs. Prot. Mode

Instruction components

- MOD field specifies the addressing mode for the selected instruction and whether a displacement is present with the specified addressing mode



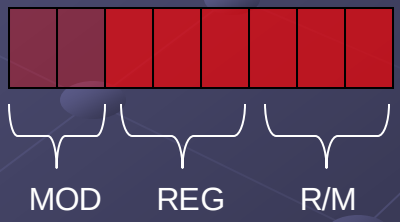
MOD	FUNCTION
00	No displacement
01	8-bit sign-extended displacement
10	16-bit displacement
11	R/M is a register (register addressing mode)

- If the MOD field contains a 00, 01, or 10, the R/M field selects one of the memory-addressing modes. For example:
  - MOV AL, [DI] no displacement (mode 00)
  - MOV AL, [DI + 2] 8-bit displacement (mode 01)

Instruction components

Register assignments for the REG and R/M fields

Code	W=0 (Byte)	W=1 (Word)	W=1 (DWord)
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



Register assignment example

Consider the 2-byte machine language instruction 8BECh. Assume 16-bit instruction mode.

Binary representation: 1000 1011 1110 1100, from which we get:

OPCODE:	100010	MOV
D	1	data goes from R/M to REG
W	1	data size is 16-bits or one word
MOD	11	R/M field indicates a register
REG	101	Register BP
R/M	100	Register SP

Therefore the instruction is MOV BP, SP

Code	W=0	W=1	W=1
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

R/M weirdness

- If the MOD field contains 00, 01, or 10, the R/M fields takes on a new meaning
- Examples:
  - If MOD = 00 and R/M = 101 the addressing mode is [DI]
  - If MOD = 01 or 10 and R/M = 101 the addressing mode is [DI+33h] or [DI+2222h] where 33h and 2222h are arbitrary displacement values

R/M Code	Function
000	DS:BX+SI
001	DS:BX+DI
010	SS:BP+SI
011	SS:BP+DI
100	DS:SI
101	DS:DI
110	SS:BP
111	DS:BX

MOD	FUNCTION
00	No displacement
01	8-bit sign-extended displacement
10	16-bit displacement
11	R/M is a register (register addressing mode)

Example

Consider machine language instruction 8A15h

Binary representation: 1000 1010 0001 0101, from which we get:

OPCODE:	100010	MOV
D	1	data goes from R/M to REG
W	0	data byte sized
MOD	00	R/M field indicates a mem addr mode
REG	010	Register DL
R/M	101	DS:DI

Therefore the instruction is MOV DL, [DI]

MOD	FUNCTION
00	No displacement
01	8-bit sign-extended displacement
10	16-bit displacement
11	R/M is a register (register addressing mode)

R/M Code	Function
000	DS:BX+SI
001	DS:BX+DI
010	SS:BP+SI
011	SS:BP+DI
100	DS:SI
101	DS:DI
110	SS:BP
111	DS:BX

Code	W=0	W=1	W=1
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

Direct addressing mode

- Direct Addressing mode (for 16-bit instructions) occurs whenever memory is referenced by only a displacement

  MOV [1000h], DL moves the contents of DL into data segment memory location 1000h

  MOV [NUMB], DL moves the contents of DL into symbolic data segment memory location NUMB

MOV [1000h], DL

Whenever the instruction has only a displacement:

MOD is always 00
R/M is always 110

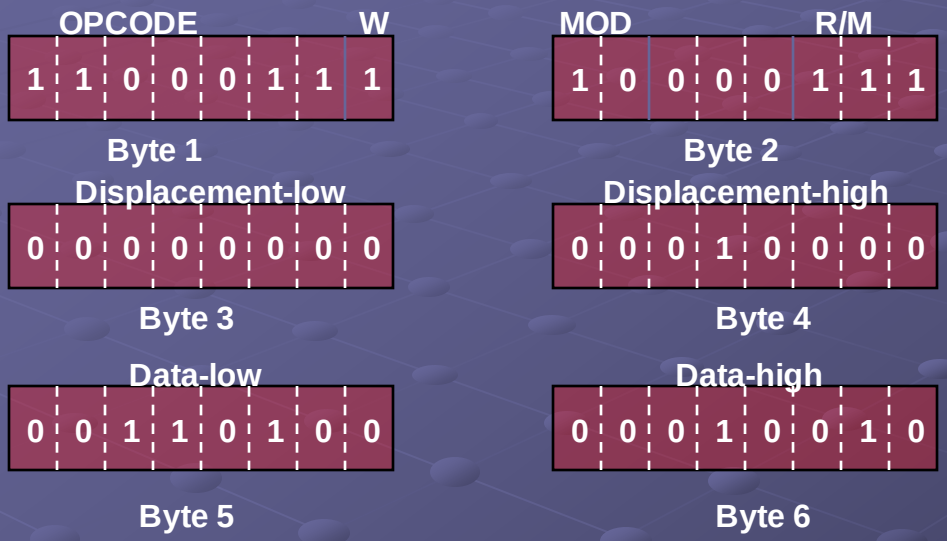
But wait…

- Doesn't this cause a problem?
  - What about mov dl, [bp]?
  - No displacement, Mode 00
  - [BP] addressing mode, R/M 110
- This actually assembles as mov dl, [bp+0]
  - 8-bit displacement, Mode 01
  - [BP] addressing mode, R/M 110
- Note that this makes [BP] move instructions at least three bytes long

R/M Code	Function
000	DS:BX+SI
001	DS:BX+DI
010	SS:BP+SI
011	SS:BP+DI
100	DS:SI
101	DS:DI
110	SS:BP
111	DS:BX

Immediate Instruction

Consider an instruction: MOV WORD[BX + 1000h], 1234h



R/M Code	Function
000	DS:BX+SI
001	DS:BX+DI
010	SS:BP+SI
011	SS:BP+DI
100	DS:SI
101	DS:DI
110	SS:BP
111	DS:BX

Moves 1234h into the word-sized memory location addressed by the sum of 1000h, BX, and DS x 10h

WORD directive indicates to the assembler that the instruction uses a word-sized memory pointer (if the instruction moves a byte of immediate data, then BYTE directive is used).

The above directives are only needed when it is not clear if the operation is a byte or a word, e.g.,

MOV [BX], AL clear - a byte is moved

MOV [BX], 1 not clear, can be byte, word, or double word-sized

MOV BYTE [BX], 1 or

MOV [BX], BYTE 1

Segment MOV Instructions

- The contents of a segment register are moved by MOV, PUSH, POP
- Segment registers are selected by appropriate setting of register bits (REG field)

Code	Segment Register
000	ES
001	CS
010	SS
011	DS
100	FS
101	GS

Note: MOV CS, ?? and POP CS are not allowed

Example: MOV BX, CS

REG is 001 => selects CS
R/M is 011 => selects BX

Note that the OPCODE for this instruction is different from the prior MOV instructions

Assignments

- HW0 due Friday at 11:59