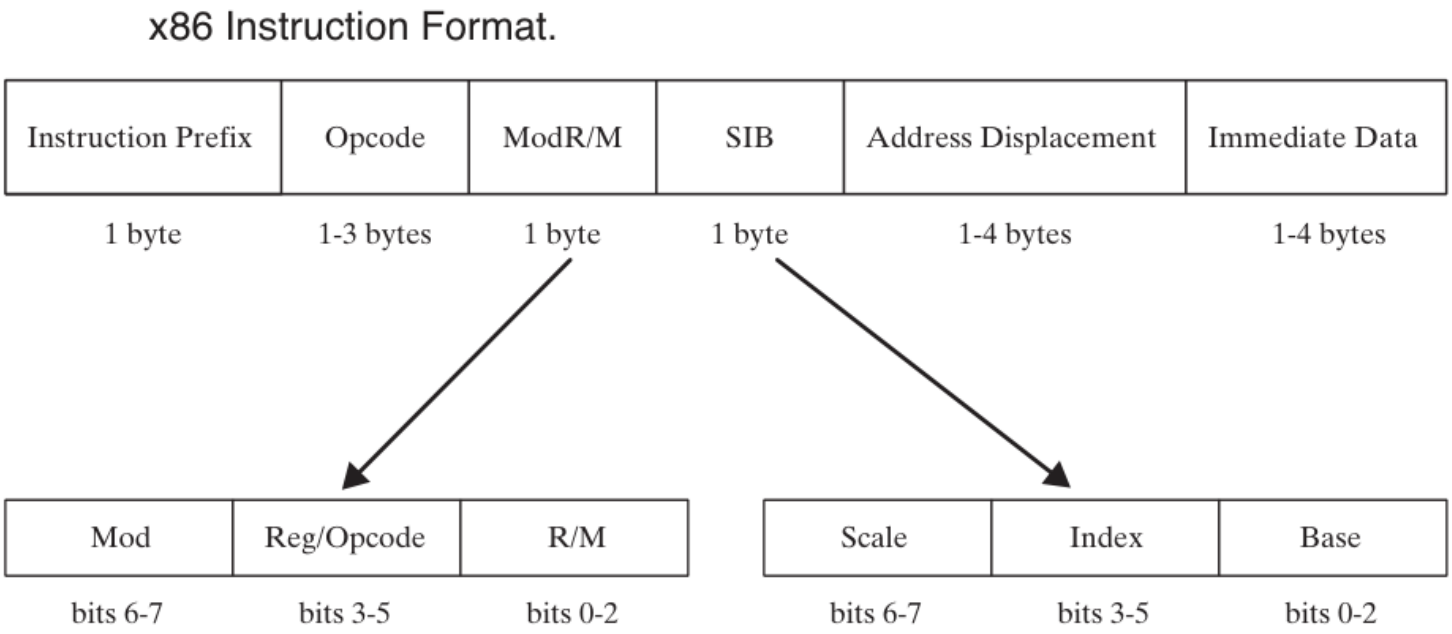
x86 Instruction Format

The passage you sent describes the general x86 machine instruction format and its components.

The general x86 machine instruction format contains the following fields:

- **Instruction prefix byte:** Overrides default operand sizes.
- **Opcode (operation code):** Identifies a specific variant of an instruction.
- **Mod R/M field:** Identifies the addressing mode and operands.
- **Scale index byte (SIB):** Used to calculate offsets of array indexes.
- **Address displacement field:** Holds an operand's offset, or it can be added to base and index registers in addressing modes such as base-displacement or base-index-displacement.
- **Immediate data field:** Holds constant operands.



Instruction prefix byte

The instruction prefix byte is a single byte that can be used to override the default operand sizes. For example, the REX prefix byte can be used to override the default operand size from 16 bits to 32 bits.

Opcode

The opcode is a single byte or multiple bytes that identifies a specific variant of an instruction. For example, the opcode for the ADD instruction is 01.

Mod R/M field

The Mod R/M field is a single byte that identifies the addressing mode and operands. The first two bits of the Mod R/M field indicate the addressing mode, and the remaining bits indicate the operands.

Scale index byte (SIB)

The scale index byte (SIB) is a single byte that is used to calculate offsets of array indexes. The SIB byte is used in conjunction with the base and index registers to calculate the effective address of an operand.

Address displacement field

The address displacement field is a single byte or multiple bytes that holds an operand's offset, or it can be added to base and index registers in addressing modes such as base-displacement or base-index-displacement.

Immediate data field

The immediate data field is a single byte or multiple bytes that holds constant operands. Immediate data operands are used in instructions such as MOV and ADD.

NB: Not all instructions contain all of the fields described above. For example, the ADD instruction only contains the opcode and Mod R/M field.

The x86 machine instruction format is a complex topic, but it is important to understand the basics of the format in order to write assembly language code.

assembly language code.

MOD FIELD

The MOD field values shown in the table below are used to specify the addressing mode for the operand in the Mod R/M field. The MOD field is two bits wide, and the possible values are:

MOD value	Addressing mode
00	Register direct
01	Register indirect with 8-bit displacement
10	Register indirect with 16-bit displacement
11	R/M field contains the operand address

In the context of x86 assembly language instruction encoding, these are the meanings of the Mod (Mode) values along with their corresponding Displacement (DISP) representations:

Mod 00: DISP = 0

This mode implies that the displacement (address offset) is 0. Disp-low and disp-high bytes are absent in the encoding, unless the R/M (Register/Memory) field equals 110, in which case the displacement is 32 bits long.

Mod 01: DISP = disp-low, sign-extended to 16 bits; disp-high is absent

In this mode, the displacement is included and consists of the disp-low byte sign-extended to 16 bits.
The disp-high byte is absent.

Mod 10: DISP = disp-high and disp-low are used

In this mode, both the disp-high and disp-low bytes are used to form the displacement.
The full 32-bit displacement is present.

Mod 11: R/M field contains a register number

In this mode, the R/M field does not indicate memory addressing but instead contains a register number.
No displacement is used in this case.

These modes are used to specify how memory operands are addressed and whether or not displacement values are included in the instruction encoding. The specifics of how these modes are used can vary depending on the particular instruction and addressing needs.

If mod field is 11:

If the MOD field is 11, then the R/M field contains the address of the operand. Otherwise, the R/M field contains a register code, and the operand address is calculated using the following formula:

```
operand_address = [register] + displacement
```

where [register] is the value of the register specified by the R/M field, and displacement is the displacement value specified in the instruction.

Here are some examples of how the MOD field is used to specify the addressing mode for the operand in the Mod R/M field:

Instruction	MOD field	R/M field	Operand address
MOV EAX, EAX	00	00	EAX register
MOV EAX, [EBX]	00	03	The memory location at the address in the EBX register
MOV EAX, [EBX + 10]	01	03	The memory location at the address in the EBX register plus 10 bytes
MOV EAX, [EBX + ESI * 4]	10	03	The memory location at the address in the EBX register plus the value of the ESI register multiplied by 4 bytes

Let's continue...

R/M	Effective Address
000	[BX + SI] + D16[a]
001	[BX + DI] + D16
010	[BP + SI] + D16
011	[BP + DI] + D16
100	[SI] + D16
101	[DI] + D16
110	[BP] + D16
111	[BX] + D16

[a]D16 indicates a 16-bit displacement.

I was using the table above, but it seems this is the new table:

R/M value	Effective Address
000	[BX] + D16
001	[BX + SI] + D16
010	[BP + SI] + D16
011	[BP + DI] + D16
100	[SI] + D16
101	[DI] + D16
110	[BP] + D16
111	[BX] + D16

The table on 16-bit R/M field values (for MOD = 10) shows the values of the R/M field for each mode. The R/M field is a single byte, and the first two bits of the R/M field indicate the addressing mode, and the remaining bits indicate the operands.

Here is an explanation of each row in the table:

The D16 in the table refers to a 16-bit displacement. The displacement is added to the base register or index register (or both) to calculate the effective address of the operand.

For example, the instruction **MOV EAX, [BX + 10]** has an R/M value of 000. This means that the operand address is calculated by adding the 16-bit displacement 10 to the BX register.

The table also shows that the R/M value 111 is the same as the R/M value 000. This is because the R/M field is only 3 bits wide, so there are only 8 possible values. The R/M value 111 is used to indicate that the base register is BX.

The 16-bit R/M field table is a useful reference for understanding how addressing modes are used in assembly language instructions.

SINGLE BYTE INSTRUCTIONS

The simplest type of instruction is one with either no operand or an implied operand. Such instructions require only the opcode field, the value of which is predetermined by the processor's instruction set.

Table below lists a few common single-byte instructions. It might appear that the INC DX instruction slipped into the table by mistake, but the designers of the instruction set decided to supply unique opcodes for certain commonly used instructions. As a consequence, register increments are optimized for code size and execution speed.

Instruction	Opcode
AAA	37
AAS	3F
CBW	98
LODSB	AC
XLAT	D7
INC DX	42

The table of single-byte instructions that you provided lists the following instructions:

- **AAA:** ASCII adjust for addition
- **AAS:** ASCII adjust for subtraction
- **CBW:** Convert byte to word
- **LODSB:** Load byte from string
- **XLAT:** Translate
- **INC DX:** Increment DX

The **AAA and AAS instructions** are used to adjust the carry flag after performing addition or subtraction operations on ASCII characters.

The **CBW instruction** converts a byte to a word by filling the upper 8 bits of the word with the sign bit of the byte.

The **LODSB instruction** loads a byte from the current position in the string pointer (SI).

The **XLAT instruction** translates a byte using the ASCII translation table in memory.

The **INC DX instruction** increments the DX register by one.

The **INC DX instruction** is included in the table of single-byte instructions because it is a commonly used instruction that is optimized for code size and execution speed.

The designers of the instruction set decided to supply unique opcodes for certain commonly used instructions, such as INC DX, in order to improve the performance of programs that use these instructions frequently.

Here is an example of how the INC DX instruction can be used:

```
687 MOV DX, 10
688 INC DX
```

This code will increment the DX register by one, so the value of DX will be 11 after the code is executed.

Single-byte instructions are the simplest type of instruction, but they can be used to perform a variety of operations. By understanding how to use single-byte instructions, you can write efficient and effective assembly language code.

MOV IMMEDIATE TO REGISTER

The passage you sent describes how to encode MOV instructions that move immediate values to registers in x86 assembly language.

The encoding format for a MOV instruction that moves an immediate word into a register is:

```
B8 + rw dw
```

where:

B8 is the opcode for moving an immediate value to a register.
rw is the register code for the destination register.
dw is the immediate word operand, low byte first.

To encode a MOV immediate instruction, you must first determine the register code for the destination register. The register codes are listed in the following table:

Register	Code
AX/AL	0
CX/CL	1
DX/DL	2
BX/BL	3
SP/AH	4
BP/CH	5
SI/DH	6
DI/BH	7

Once you have determined the register code, you can encode the MOV immediate instruction as follows:

- Add the register code to B8 to get the opcode.
- Append the immediate operand to the opcode, low byte first.

For example, the following instruction moves the immediate value 1 to the AX register:

MOV AX, 1

The register code for AX is 0, so the opcode for this instruction is B8 + 0 = B8.

The immediate operand is 1, so the machine code for this instruction is B8 01 00.

The machine instruction is B8 01 00 (in hexadecimal).

Encoding Steps:

The opcode for moving an immediate value to a 16-bit register is B8.
The register number for AX is 0, so 0 is added to B8.
The immediate operand (0001) is appended to the instruction in little-endian order (01, 00).

Another example is the following instruction, which moves the immediate value 1234h to the BX register:

MOV BX, 1234h

The register code for BX is 3, so the opcode for this instruction is B8 + 3 = BB.

The immediate operand bytes are 34 12, so the machine code for this instruction is BB 34 12.

The machine instruction is BB 34 12.

Encoding Steps:

The opcode for moving an immediate value to a 16-bit register is B8.
The register number for BX is 3, so add 3 to B8, producing opcode BB.
The immediate operand bytes are 34 12.

You can practice encoding MOV immediate instructions by hand to improve your skills.

Once you have encoded an instruction, you can check your work by inspecting the code generated by MASM in a **source listing file.**

PUSH CX:

The machine instruction is 51.

Encoding Steps:

The opcode for PUSH with a 16-bit register operand is 50.
The register number for CX is 1, so add 1 to 50, producing opcode 51.

REGISTER MODE INSTRUCTIONS

If you want to understand this subtopic better, check the last line of this subtopic, go to that reference.

The passage you sent describes how to encode MOV instructions that use register operands in x86 assembly language.

The encoding format for a MOV instruction that moves a value from one register to another is

```
89/r reg
```

where:

- **89** is the opcode for moving a value from one register to another.
- **/r** indicates that a Mod R/M byte follows the opcode.
- **reg** is the register code for the destination register.

The Mod R/M byte contains a 3-bit identifier for each register operand.

The Mod R/M byte is made up of three fields (mod, reg, and r/m). A Mod R/M value of D8, for example, contains the following fields:

mod	reg	r/m
11	011	000

The **Mod field is 11**, indicating that the r/m field contains a

register number.

The **reg field is 011**, indicating that the source operand is the BX register.

The **r/m field is 000**, indicating that the destination operand is the AX register.

Therefore, the machine code **89 D8** is for the instruction **MOV AX, BX,** which moves the value in the BX register to the AX register.

The register codes are listed in the following table:

R/M	Register	R/M	Register
000	AX or AL	100	SP or AH
001	CX or CL	101	BP or CH
010	DX or DL	110	SI or DH
011	BX or BL	111	DI or BH

For example, the following instruction moves the value in the BX register to the AX register:

MOV AX, BX

The register code for BX is 3 and the register code for AX is 0, so the machine code for this instruction is 89 D8.

Another example is the following instruction, which moves the value in the CL register to the AL register:

MOV AL, CL

The register code for CL is 1 and the register code for AL is 0, so the machine code for this instruction is 89 C8.

Sample MOV Instruction Encodings, Register Operands.

Instruction	Opcode	mod	reg	r/m
mov ax,dx	8B	11	000	010
mov al,dl	8A	11	000	010
mov cx,dx	8B	11	001	010
mov cl,dl	8A	11	001	010

You can practice encoding MOV instructions that use register operands by hand to improve your skills. Once you have encoded an instruction, you can check your work by inspecting the code generated by MASM in a source listing file.

==========

To calculate the value 89 D8 from the R/M fields, you need to follow these steps:

Identify the addressing mode. The Mod field is 11, which indicates that the addressing mode is register direct.

Identify the source operand. The reg field is 011, which indicates that the source operand is the BX register.

Identify the destination operand. The r/m field is 000, which indicates that the destination operand is the AX register.

Encode the opcode. The opcode for the MOV instruction is 89.

Encode the Mod R/M byte. The Mod R/M byte is encoded as follows:

- **Mod field: 11 (register direct)**
- **reg field: 011 (BX register)**
- **r/m field: 000 (AX register)**

Therefore, the machine code for the instruction MOV AX, BX is 89 D8.

Here is a table that summarizes the encoding of the MOV instruction:

Opcode	Mod R/M byte	Instruction
89	00 /r	MOV register, register
89	01 /r	MOV register, memory
89	10 /r	MOV memory, register
89	11 /r	MOV memory, memory

The reg field and r/m field of the Mod R/M byte identify the source and destination operands, respectively. The Mod field indicates the addressing mode for the operand.

So, this only applies to mov instruction?

Yes, the encoding scheme you described only applies to the MOV instruction. Other instructions may have different encoding schemes.

For example, the ADD instruction has the following encoding scheme:

Opcode	Mod R/M byte	Instruction
01	00 /r	ADD register, register
01	01 /r	ADD register, memory
01	10 /r	ADD memory, register
01	11 /r	ADD memory, memory

The ADD instruction is similar to the MOV instruction, but it adds the source operand to the destination operand instead of moving the source operand to the destination operand.

Different instructions have different encoding schemes because they

perform different operations. The encoding scheme for each instruction is specified in the **Intel instruction set reference.**