Instruction	Description	Example
`MOV`	Moves source operand to destination operand	

`MOVZX`	ZX translates to Zero eXtend
	A source operand for this instruction has to have length less than destination's length. That is, MOVZX RAX, BX is correct, but MOVZX EAX, EBX is not. When moving source to destination, all destination's bits which didn't overlap with source's bits, get replaced with zeros, even when they contained zeros only.
		`MOV   EBX, 0x12345678`
		`MOV   AX,  0x6666`
		`MOVZX EBX, AX            ; EBX = 0x00006666`

`MOVSX`	SX is Sign Extend.
	It operates the same way as MOVZX, but it chooses to fill spare bits with 1's or 0's depending on most significant bit of source. If source's MSB is 1, then it fills destination with 1's, otherwise with 0's.
		`MOV EBX, 0x12345678`
		`MOV BX, 0x989A           ; 0x98 byte starts with 1, because`
		`                           0x98 is 10011000 in binary`
		`MOVSX EBX, BX            ; EBX = 0xFFFF989A, F is 1111`

`LEA`	Load Effective Address
	This instruction is a bit special. Its second operand(source) is always bound with brackets. But in fact, it doesn't dereference memory at source operand. Instead, it only takes the source address or value.
		`LEA RAX, QWORD PTR [RBX]       ; RAX=RBX`
		`LEA RAX, QWORD PTR [0xBADF00D] ; RAX=0xBADF00D`

`ADD`	It adds operands and stores result in destination operand	
`SUB`	Substract source operand from destination and store result in destination operand	

`MUL`	Takes only 1 operand. Performs unsigned multiplication on operand and number given in AL, AX, EAX, RAX, depending on operand's size, and stores result in AX, DX:AX, EDX:EAX, RDX:RAX respectively. If result can be fit only in one register, DX, EDX, or RDX stores zero.
	
	I hope this remainder can give you simplified overview:
	`EDX:EAX = EAX * operand`
		`MOV AX, 16`
		`MOV DL, 32`
		`MUL DL                   ; AX = 512`

`DIV`	Takes only 1 operand. Works similarly as MUL: take dividend from AX, DX:AX, EDX:EAX, RDX:RAX(depending on operand's size), divide it by operand(divisor), store result in AL, AX, EAX,	

	RAX, and put remainder in AH, DX, EDX, RDX, respectively. Tip for this one is: `EAX = EDX:EAX / operand, and store remainder in EDX`	

XOR	XOR's both operands and stores result in destination. XORing two bits works this way: if both bits are equal, result of XOR is 0, if bits are not equal, result is 1	`MOV EAX, 0x34          ; EAX = 0011 0100`
		`MOV EBX, 0x25          ; EBX = 0010 0101`
		`XOR EAX, EBX           ; EAX = 0001 0001`

AND	Performs bitwise AND operation on both operands and stores the result in destination operand.	
TEST	Does the same as AND, but doesn't store operation result to destination. The only thing it does is changing flags in EFLAGS register. Normally this instruction represents condition in the program's code, and it TESTs it.	
OR	Performs bitwise OR operation on both operands and stores the result.	

SHR/SHL and SAR/SAL	Bit shift instructions. SAR and SAL are arithmetic shifts to right and left, SHR and SHL are logical shifts(I don't know what H stands for here). These instructions are often used to perform division(right shift) or multiplication(left shift) by 2. 1st operand specifies data to be shifted, 2nd operand takes number of bits to shift. With every shift turn depending on shift direction, MSB/LSB is shifted to carry flag(CF), other bits are shifted as well. Difference between arithmetic and logical shifts is what number is left as result. Arithmetic shift "saves" the sign(minus or plus) of number, while logical shifts don't care about it. This difference persists only if shift-to-right is involved, otherwise shifts are interchangeable.
	
	You could use arithmetic shifts to perform signed multiplication or division, otherwise use logical shift. Note this is not perfect instruction to perform arithmetical operations, because it rounds result and makes it look bad.
		`MOV AH, 0xFA     ; AH = 11111010(250 if unsigned, -6 if signed)`
		`SHR AH, 2        ; AH = 00111110(62 if unsigned, the same if signed)`
		
		But if you used arithmetic shift instead of logical, you would get different result:
		
		`MOV AH, 0xFA`
		`SAR AH, 2        ; AH = 11111110(254 if unsigned, -2 if signed)`

JMP	Go-to equivalent in assembly language.	It's equivalent to: `MOV EIP, operand`

`J(cc)`	Conditional JMP. This condition triggers if (cc) is 1, there is a lot of them, you can find the whole list in Intel Manual	
`NOP`	This is the instruction which does nothing at all. Hence its name: No Operation	

`CALL`	Call the function and change execution flow	`It's equivalent to:`
		`      PUSH EIP`
		`      MOV EIP, operand(function address)`

`RET`	Return from function.
	This instruction can optionally take one argument. If it is specified, before exiting from function, argument's value will be added to ESP register, this is done in order to free up stack space taken by function arguments
		`It’s equivalent to:`
		`      ADD ESP, operand    ; if operand was specified`
		`      POP EIP`

`PUSH`	Push operand to stack	
`POP`	Pop a stack item pointed at by ESP to the operand	

`LEAVE`	Instruction to leave current stack frame	`It's equivalent to:`
		`      MOV ESP, EBP`
		`      POP EBP`