Assembly Refs

Minimal x86 Opcode Reference

Based on our intel books chapter 3-6 + **MASM** :)

Data Transfer

Opcode	Description	Example
MOV	Move data (copy `src` to `dest`).	`mov eax, ebx`
CMOV	Conditional Move (e.g., `CMOVE` for "move if equal").	`cmove eax, ebx ; move if ZF=1`
XCHG	Exchange contents of two operands.	`xchg eax, ebx`
XADD	Exchange and Add.	`xadd [mem], eax`
BSWAP	Byte Swap (reverse byte order in a 32-bit register).	`bswap eax`
LEA	Load Effective Address (compute address of `src`, store in `dest`).	`lea eax, [ebx + 8]`
LDS/LES/LFS/LGS/LSS	Load Far Pointer (load `mem` into `reg` and segment register).	`lds esi, [my_ptr]`
MOVZX	Move with Zero-Extend (copy small `src` to large `dest`, fill high bits with 0).	`movzx eax, bl`
MOVSX	Move with Sign-Extend (copy small `src` to large `dest`, fill high bits with `src` sign bit).	`movsx eax, bl`
XLAT	Table Look-up Translation (replace `AL` with `[EBX + AL]`).	`xlat`

Stack Operations

Opcode	Description	Example
PUSH	Push onto stack (decrement stack pointer, store operand).	`push eax`
POP	Pop from stack (load operand, increment stack pointer).	`pop ebx`
PUSHA/PUSHAD	Push all general-purpose registers.	`pushad`
POPA/POPAD	Pop all general-purpose registers.	`popad`

PUSHF/PUSHFD	Push `EFLAGS` register onto stack.	`pushf`
POPF/POPFD	Pop `EFLAGS` register from stack.	`popf`
ENTER	Create a stack frame for a procedure.	`enter 8, 0`
LEAVE	Destroy a stack frame (reverses `ENTER`).	`leave`

Binary Arithmetic

Opcode	Description	Example
ADD	Add (unsigned/signed).	`add eax, 10`
ADC	Add with Carry (add operands + carry flag).	`adc eax, ebx`
SUB	Subtract (unsigned/signed).	`sub ecx, eax`
SBB	Subtract with Borrow (subtract operands - carry flag).	`sbb eax, 0`
INC	Increment by 1.	`inc ecx`
DEC	Decrement by 1.	`dec edx`
MUL	Unsigned Multiply (`(EDX:EAX) = EAX * src`).	`mul ebx`
IMUL	Signed Multiply.	`imul ebx`
DIV	Unsigned Divide (`EAX = (EDX:EAX) / src`).	`div ecx`
IDIV	Signed Divide.	`idiv ecx`
NEG	Negate (two's complement).	`neg eax`
CMP	Compare (sets flags by doing `dest - src` without storing result).	`cmp eax, 10`

Logical & Bitwise

Opcode	Description	Example
AND	Bitwise AND.	`and eax, 0xFF`
OR	Bitwise OR.	`or ebx, 1`
XOR	Bitwise Exclusive OR.	`xor eax, eax ; (zeros EAX)`
NOT	Bitwise NOT (one's complement).	`not ecx`
TEST	Test (sets flags by doing `dest & src` without storing result).	`test al, 0x80`
BT	Bit Test (copies bit from `src` to Carry Flag).	`bt eax, 10`

BTC	Bit Test and Complement (copies bit to CF, then flips it in `src`).	`btc eax, 10`
BTR	Bit Test and Reset (copies bit to CF, then clears it in `src`).	`btr eax, 10`
BTS	Bit Test and Set (copies bit to CF, then sets it in `src`).	`bts eax, 10`
BSF	Bit Scan Forward (find first '1' bit, scanning from LSB to MSB).	`bsf eax, edx`
BSR	Bit Scan Reverse (find first '1' bit, scanning from MSB to LSB).	`bsr eax, edx`

Shift & Rotate

Opcode	Description	Example
SHL/SAL	Shift Logical/Arithmetic Left.	`shl eax, 2`
SHR	Shift Logical Right (fill with 0).	`shr ebx, 1`
SAR	Shift Arithmetic Right (fill with sign bit).	`sar edx, 4`
SHLD	Shift Left Double Precision.	`shld eax, edx, 16`
SHRD	Shift Right Double Precision.	`shrd eax, edx, 16`
ROL	Rotate Left.	`rol al, 1`
ROR	Rotate Right.	`ror al, 1`
RCL	Rotate Left through Carry.	`rcl ebx, 1`
RCR	Rotate Right through Carry.	`rcr ebx, 1`

Control Flow

Opcode	Description	Example
JMP	Unconditional Jump.	`jmp my_label`
Jcc	Conditional Jump (e.g., `JE`, `JNE`, `JG`, `JL`).	`je is_equal`
SETcc	Conditional Set (sets byte to 0 or 1 based on flags).	`sete al ; Set AL if equal (ZF=1)`
CALL	Call procedure (push return address, jump).	`call my_function`
RET	Return from procedure (pop address, jump).	`ret`

JCXZ/JECXZ	Jump if `CX` / `ECX` is zero.	`jecxz is_zero`
LOOP	Loop (decrement `ECX`, jump if not zero).	`loop my_loop`
LOOPE/LOOPZ	Loop while Equal/Zero (decrement `ECX`, jump if not zero AND `ZF=1`).	`loope my_loop`
LOOPNE/LOOPNZ	Loop while Not Equal/Not Zero (decrement `ECX`, jump if not zero AND `ZF=0`).	`loopne my_loop`
BOUND	Check array bounds against a register value.	`bound eax, [my_array]`

String Operations

(Operate on `[ESI]` and/or `[EDI]`, update pointers based on `DF`)

Opcode	Description	Example
MOVS	Move String (`[ESI]` to `[EDI]`).	`movsb` (byte), `movsw`, `movsd`
LODS	Load String (`[ESI]` to `AL/AX/EAX`).	`lodsb` (byte), `lodsw`, `lodsd`
STOS	Store String (`AL/AX/EAX` to `[EDI]`).	`stosb` (byte), `stosw`, `stosd`
CMPS	Compare String (`[ESI]` with `[EDI]`).	`cmpsb` (byte), `cmpsw`, `cmpsd`
SCAS	Scan String (compare `AL/AX/EAX` with `[EDI]`).	`scasb` (byte), `scasw`, `scasd`
INS	Input from Port to String (read from `DX` port to `[EDI]`).	`insb` (byte), `insw`, `insd`
OUTS	Output String to Port (write `[ESI]` to `DX` port).	`outsb` (byte), `outsw`, `outsd`
REP	Repeat prefix (repeats string op `ECX` times).	`rep movsb`
REPE/REPZ	Repeat while Equal/Zero.	`repe cmpsb`
REPNE/REPNZ	Repeat while Not Equal/Not Zero.	`repne scasb`

BCD & ASCII Arithmetic

Opcode	Description	Example
DAA	Decimal Adjust after Addition (corrects `AL` after `ADD` on BCD).	`daa`
DAS	Decimal Adjust after Subtraction (corrects `AL` after `SUB` on BCD).	`das`

AAA	ASCII Adjust after Addition.	`aaa`
AAS	ASCII Adjust after Subtraction.	`aas`
AAM	ASCII Adjust after Multiplication.	`aam`
AAD	ASCII Adjust before Division.	`aad`

Flag (EFLAGS) Control

Opcode	Description	Example
CLC	Clear Carry Flag (`CF=0`).	`clc`
STC	Set Carry Flag (`CF=1`).	`stc`
CMC	Complement Carry Flag (flips `CF`).	`cmc`
CLI	Clear Interrupt Flag (disable maskable interrupts).	`cli`
STI	Set Interrupt Flag (enable maskable interrupts).	`sti`
CLD	Clear Direction Flag (`DF=0` , string ops increment).	`cld`
STD	Set Direction Flag (`DF=1` , string ops decrement).	`std`
LAHF	Load `AH` from Flags (copy `SF,ZF,AF,PF,CF` to `AH`).	`lahf`
SAHF	Store `AH` into Flags (copy `AH` into `SF,ZF,AF,PF,CF`).	`sahf`

I/O & Interrupts

Opcode	Description	Example
IN	Input from port (read from port to `AL/AX/EAX`).	`in al, 0x60`
OUT	Output to port (write `AL/AX/EAX` to port).	`out 0x61, al`
INT	Software Interrupt (call interrupt vector `n`).	`int 0x80`
INT3	Breakpoint Interrupt (1-byte `INT 3`).	`int3`
INTO	Interrupt on Overflow (call `INT 4` if `OF=1`).	`into`
IRET/IRETD	Return from Interrupt.	`iret`

Processor Control

Opcode	Description	Example
NOP	No Operation (takes 1 cycle, does nothing).	`nop`
HLT	Halt (stops CPU execution until an interrupt occurs).	`hlt`

WAIT	Wait (halts for coprocessor, $\overline{TEST}$ pin).	`wait`
ESC	Escape (passes instruction to coprocessor).	`esc`
LOCK	Lock Prefix (asserts $\overline{LOCK}$ pin for atomic operation).	`lock add [mem], eax`

Misc

Writing Single Character

```
; --- Print the character 'A' ---
    MOV AH, 02h
    MOV DL, 'A'
    INT 21h
```

Reading Single Character

```
; --- Read a key from the user ---
    MOV AH, 01h
    INT 21h

    ; The character is now in AL. You can move it:
    ; MOV BH, AL
```

String I/O

Input

```
.DATA
    myBuffer DB 31, ?, 31 DUP(?)

.CODE
    ; --- Read a string from the user ---
    MOV AH, 0Ah
    LEA DX, myBuffer    ; Point DX to the start of the buffer
    INT 21h
```

Output

```
.DATA
    myMessage DB 'Hello, world!', 0Ah, 0Dh, '$'
    ; 0Ah = Line Feed (new line)
    ; 0Dh = Carriage Return
    ; $   = String terminator
```

```
.CODE
MAIN PROC
    ; --- Set up DS ---
    MOV AX, @DATA
    MOV DS, AX

    ; --- Print the string ---
    MOV AH, 09h
    LEA DX, myMessage   ; Load Effective Address of myMessage into DX
    INT 21h

    ; --- Exit ---
    MOV AX, 4C00h
    INT 21h
MAIN ENDP
```

Template

A full code structure. It's better to learn a full code structure I guess...

```
; ------------------------------------------------------------------------
-----
; A basic template for a 16-bit MASM/TASM assembly program (.COM or .EXE)
; This template is for an .EXE file which has separate segments.
; ------------------------------------------------------------------------
-----

; --- Model Directive ---
; Defines the memory model.
; SMALL: 1 code segment (64K), 1 data segment (64K). Good for most simple
programs.
.MODEL SMALL
.STACK 100h     ; Define the stack size (256 bytes)

; --- Data Segment ---
; All initialized and uninitialized variables go here.
.DATA
    ; Example initialized variable
    Message   DB  'Hello, world!', 0Dh, 0Ah, '$'  ; String terminated with
$

    ; Example uninitialized variable
    UserInput DB  80 DUP(?)   ; A buffer to store 80 bytes

; --- Code Segment ---
; All executable instructions go here.
.CODE
; --- Main Procedure ---
; This is the main entry point for the program.
```

```
Main PROC
    ; --- Boilerplate: Set up Data Segment (DS) ---
    ; In an .EXE program, DS must be manually set to point to the .DATA
segment.
    mov ax, @DATA       ; Get the address of the .DATA segment
    mov ds, ax          ; Set the Data Segment (DS) register

    ; --- Your Code Goes Here ---

    ; Example: Call a custom procedure
    ; We assume AX and CX might have important values before this call
    mov ax, 1234h
    mov cx, 5678h
    call MyProcedure
    ; After MyProcedure returns, AX and CX will still have
    ; 1234h and 5678h because the procedure saved them.

    ; Example: Print the 'Message' string using DOS interrupt 21h,
function 09h
    lea dx, Message     ; Load Effective Address of Message into DX
    mov ah, 09h         ; Set DOS function 09h (print string)
    int 21h             ; Call DOS interrupt

    ; --- Program Exit ---
    ; This is the standard way to exit a DOS program and return to the
command line.
    mov ax, 4C00h       ; Set DOS function 4C00h (Exit with return code 0)
    int 21h             ; Call DOS interrupt

Main ENDP

; --- Other Procedures ---
; It's good practice to define other procedures outside of Main.

; MyProcedure: A simple example procedure
; Description: This procedure demonstrates the NEAR type, the USES
;              directive, and how to preserve register values.
; Input: None
; Output: None
;
; PROC [type]:
;   - NEAR (default for .MODEL SMALL): The procedure is in the same
;     code segment. 'call' and 'ret' will be 'near' (push/pop IP only).
;   - FAR: The procedure is in a different code segment. 'call' and
;     'ret' will be 'far' (push/pop CS and IP).
;
; USES [reg1] [reg2] ...:
;   - This is a high-level MASM directive that automatically generates
;     PUSH instructions for the listed registers at the start of
;     the procedure and corresponding POP instructions just before
```

```
;     the 'ret'.
;   - This is critical for preserving the values of registers that
;     the main program (the "caller") might be using.
;
MyProcedure PROC NEAR USES ax cx
    ; The 'USES ax cx' directive automatically generates:
    ;   push ax
    ;   push cx
    ; ...at the beginning of the procedure.

    ; Procedure code would go here
    ; We are free to use AX and CX without worrying about
    ; messing up their values for the caller.
    mov cx, 10          ; Example: use CX for a loop counter
    mov ax, 0           ; Example: use AX as an accumulator

MyLoop:
    ; ... do something 10 times ...
    add ax, cx
    dec cx
    jnz MyLoop

    ; The assembler will automatically insert:
    ;   pop cx
    ;   pop ax
    ; ...right before the 'ret' instruction, restoring the original
values.

    ret                 ; Return from procedure (pops return address from
stack)
MyProcedure ENDP

; --- End of Program ---
; The 'END Main' directive tells the assembler where the program execution
should start.
END Main
```