16m041 – CPU

User Manual

Contents

Operations

Register and memory access

MOV

Copys the value from the source register into the drain register. The operation is also used to copy values from the ***memory*** into a register or copy a value from a register into a given ***memory*** location. For reading from/ writing from the ***memory*** there are multiple ***addressing modes*** that can be used. The MOV (move) operation can be used with either 16-bit registers and 8-bit registers.

XCHG

The XCHG (exchange) operations switches the values of two registers regardless of the register size (either 8-bit or 16-bit).

AX = 1234h, BX = 5678h after XCHG AX,BX is performed AX = 5678h, BX = 1234h

LAHF

When performed the content of the ***Flagregister*** is copyed into the ***AH register***.

SAHF

Similar to ***LAHF*** the operations copys the content of the ***AH register*** into the ***flag register.***

XLAT

This operation translates a byte into the ***AL register***. It basicly acts like the ***MOV operation*** but uses a indexed and specific ***addessing mode***.

MOV AL,DS:[BX+AL]

POP

The POP operations copys a value from ***memory*** into the drain register. Oposed to the ***MOV operation*** the ***addressing mode*** is not changeable. The opperations accesses a fixed part of the ***memory*** (the ***Stack***) thus using the ***Stackpointer*** and ***Stacksegment*** to calculate the source address. After the value is fetched from the ***memory*** the ***Stackpointer*** is automaticly incremented.

POPF

This operations does the same as the ***POP*** operation exept the drain register is not changeable, the drain register is allways the ***Flagregister***.

PUSH

When performed the source register will be stored in the ***memory***. Similar to the POP operation and different to the ***MOV operation*** the addressing mode can not be changed and the address will

allways be calculated with the ***Stackpointer*** and the ***Stacksegment***. After the register value is stored in ***memory*** the ***Stackpointer*** is decremented automaticly.

PUSHF

This operation does the same as the ***PUSH*** operation but the source register can not be changed. When performed the ***Flagregister*** will be stored in ***memory***.

Arithmetic logic unit

All ALU (Arithmetic logic unit) operations have a source and drain part. A logic or arithmetic operation will be performed with the drain and source operants these can either be ***registers*** or a value from memory (using different ***addressing modes***). The result of the operation is allways stored in the ***AX register***.

Not all ALU operations are affected by ***flags*** but all ALU operations can set and clear ***flags*** depending on the result they yield.

ADD

Adds the drain and source parts.

ADD AX,BX | AX = 0012h, BX = 0209h → 021bh will be stored in AX

ADC

Similar to the ***ADD*** operation it adds the source and drain parts but also uses the ***Carry flag***. When the ***Carry flag*** is set 1 is added to the result if the ***Carry flag*** is not set nothing else will be set.

A*DC AX,BX | CF = 1, AX = 0210h, BX = 1222h → 1433h will be stored in AX*

AND

A logical AND will be perfomed.

AND AX,BX | AX = 1431h, BX = 0812h → 0010h will be stored in AX

DEC

The DEC does not have a source component it only takes in one parameter (drain). When performed it subtracts 1 from the drain component.

DEC CX | CX = 02F5h → 02F4h will be stored in AX

DIV

When performed the ***AX register*** will be divided by the source argument. When the CPU is in 8-bit mode the result will be stored in the ***AH register*** and the reminder is stored in the ***AL register*** otherwise when the CPU is in 16-bit mode the result of the division is stored in the ***AX register*** and the reminder in the ***DX register***.

DIV AX,12A3h | AX = 4214h → 0003h is stored in AX and 0A2B is stored in DX

DIV AX, A0h | AX = 00F4h → 01h is stored in AH amd 54h is stored in AL

INC

The INC operation is the same as the DEC operation exept for subtracting 1 from the drain component it adds 1.

INC AX | AX = 0129h → 012Ah will be stored in AX

XOR

An exclusive or will be performed with the drain and source components.

XOR AX,BX | AX = 1421h, BX = 5213h → 4632h will be stored in AX

SUB

When performed the source component will be subtracted from the drain component.

SUB AX, BX | AX = 5678h, BX = 1234h → 4444h will be stored in AX

MUL

This operation multiplys the drain component with the ***AX register***. If the CPU is in 8-bit mode the result will be stored reguraly in the ***AX register*** but when it is in 16-bit mode the upper half of the result will be stored in the ***AX register*** and the lower magnitude of the result is stored in the ***DX register***.

MUL AX,BX | AX = 0012h, BX = 0004h → 0048h will be stored in AX

MUL AX,BX | AX = 0FFFh, BX = 01A6h → 001Ah will be stored in AX and 5E5Ah in DX

NEG

The drain operant's sign will be changed meaning when it is prositiv it will be turned negative and when it is negative it will be turned prositive. The operation only takes one argument, the source part is not used.

NEG AX | AX = 147Ah → EB86h will be stored in AX

NOT

This operation only takes one argument the source component is not used. With the drain part a logical not will be performed. The binary value will be inverted.

NOT AX | AX = 7A1Fh → 85E0h will be stored in AX

OR

When performed a logical or operation will be executed with the drain and source components.

OR AX,BX | AX = 1257h, BX = 731Ah → 735Fh will be stored in AX

ROL

Rotate the drain parameter left. The source component determains how far the drain component will be roteted.

ROL AX,000Ah | AX = 7A42h → 09E9h will be stored in AX

ROR

Rotate the drain parameter right. The source component determains how far the drain component will be roteted.

ROR AX,000Ah | AX = 7A42h → 909Eh will be stored in AX

SHL

A logical left shift will be performed on the drain parameter and the shift amount is determand by the source component. The new values entering from the right will allways be zero.

SHL AX,0002h | AX = 0785h → 1E14h will be stored in AX

SHR

Similar to the ***SHL operation*** but exept for a logical left shift a logical right shift will be performed.

SHR AX,0002h | AX = 0785h → 01E1h will be stored in AX

SAL

A arithmetic left shift will be performed on the drain parameter and the shift amount is determend by the source part. The new bits shifted in on the right will be have same value as the bit on the furthest right.

SAL AX, 0002h | AX = 0183h → 060Ch will be stored in AX

SAR

It is the same as the ***SAL operation*** exept for shifting to the left it will shift to the right.

SAL AX, 0002h | AX = 8183h → E060h will be stored in AX

SBB

When performed the source parameter will be subtracted from the drain component but contrary to the ***SUB operation*** it also uses the ***Carry flag***. If the ***Carry flag*** is set one is subtracted from the result and when it is not nothing further will be subtracted from the result.

SBB AX,0021h | CF = 1, AX = 1A78h →1A56h will be stored in AX

SBB AX,0021h | CF = 0, AX = 1A78h →1A57h will be stored in AX

RCL

This is similar to the ***ROL operation*** but it sets the Carryflag depending on the values shifted out.

RCR

This is similar to the ***ROR operation*** but it sets the Carryflag depending on the values shifted out.

CMP

A ***SUB operation*** will be perfomred but the result of the subtration will not be stored in ***AX register*** only the ***flags*** will be set. This operation is used for ***jump operations***.

CMP AX,BX → AX and BX will not be changed only flags will be set

AAA

ASCII adjust after addition used after an addition. This only performs on the ***AX register*** and is used on decimal-coded binary numbers and adjusts them so the value stays within ASCII space.

AAD

ASCII adjust after division used after an division. This only performs on the ***AX register*** and is used on decimal-coded binary numbers and adjusts them so the value stays within ASCII space.

AAM

ASCII adjust after multiplitation used after a multiplitation. This only performs on the ***AX register*** and is used on decimal-coded binary numbers and adjusts them so the value stays within ASCII space.

AAS

ASCII adjust after subtration used after a subtration. This only performs on the ***AX register*** and is used on decimal-coded binary numbers and adjusts them so the value stays within ASCII space.

DAA

Deciaml adjust after addition is used after an addition of two decimal-coded binary numbers. This allways perfomes the operation on the ***AX register***.

DAS

This performes a deciaml adjust after an subtraction. This is used with decimal-coded binary numbers. This only performs on the ***AX register***.

IDIV

This operation works the same as the DIV operation but differently to the DIV operation it uses the mathematical sign.

IMUL

This operation works the same as the ***MUL operation*** but differently to the ***MUL operation*** it uses the mathematical sign.

CBW

Converts the byte inside the ***AL register*** into a complete word in the ***AX register***. When the value of the ***AL regiseter*** is smaller then *80h* nothing happens else *FFh* will be stored in the ***AH register***.

CWD

Similar to ***CBW*** but it converts a word from the ***AX register*** into a double world stored in DX:AX. If the ***AX register*** is smaller then *8000h* nothing will be performed else *FFFFh* will be stored in the ***DX register***.

TEST

This is the same as the logical ***AND operation*** but it does not store the result of the operation in the AX register it is only used to set and clear the ***zeroflag***.

CPU controll and flag manipulation

These operations are used to set and clear flags as well as changing the CPU controll flow. These operations can not read or write to memory or change the values of the ***registers*** exept for the

flagregsiter but it only changes single bits of the ***Flagregister*** supposed to multiple bits/ a whole byte.

CLC

This operation clears the ***carryflag***.

CLD

This operation clears the ***directionflag***.

CLI

This operation clears the ***interruptflag***.

CMC

This operation inverts the ***carryflag***. If the ***carryflag*** is set it will be cleared and if it is not set it will be set.

STC

This operation sets the ***carryflag***.

STD

This operation sets the ***directionflag***.

STI

This operation sets the ***interruptflag***.

LOCK

This operation is a prefix. While the next operation is executed interrupts are disabled.

WAIT

When executed the CPU stops executing and returns as soon as the CPU recieves a test signal on the ***test-pin***.

HLT

When executed the CPU goes to sleep and stops executing and when a reset or an interrupt accoure

the CPU returns to executing.

NOP

This is an empty clycle. The CPU does nothing when executing this operation.

Pointer loading and calculation

These operations load a far pointer.

LDS

Load pointer using the ***DS register***

LDS SI,1234h,5678h → SI = 1234h, DS = 5678h

LEA

Load the effective address to register

LES

Load pointer using the ***ES register***

LES SI,1234h,5678h → SI = 1234h, ES = 5678h

String loading and manipulation

These operations are designed to manipulate and move strings.

LODSB

Load the value stored at DS:SI into the ***AL register***

LODSW

Load the value stored at DS:SI into the ***AL register*** then increment the SI register and store the value from the new address in the ***AH register***.

MOVSB

Load the value stored at DS:SI and store it in the memory location ES:DI. This operation does not affect any registers.

MOVSW

Load the value stored at DS:SI and store it in the memory location ES:DI then increment the ***SI register*** and the ***DI register*** after that load the new value from DS:SI and then store it at ES:DI. This operation does not affect any registers.

STOSB

Stores the value from in the AL register in the memory location ES:DI.

STOSW

Stores the value from in the AL register in the memory location ES:DI and then DI gets incremented and the value from the ***AH register*** get stored at the new location ES:DI.

SCASB

This compares (by subtraction) the value from the ***AL register*** and the value stored at ES:DI. This operation does not affect any ***registers*** and only affects the ***Flagregister*** like the ***CMP operation***.

SCASW

Similar to the ***SCASB operation*** but it compares the whole ***AX register*** and not just one half. The string part is loaded from ES:DI and ES:(DI+*0001h*).

CMPSB

The same as the ***SCASB operation*** exept it uses DS:SI.

CMPSW

The same as the ***SCASB operation*** exept it uses DS:SI and DS:(SI+*0001h*).

Un-/Conditioan jumps and function calls

JCXZ

Set the ***instructionpointer*** to a specific value if CX is zero (conditional jump).

JE

Set the ***instructionpointer*** to a specific value if the ***zeroflag*** is set (conditional jump).

JG

Set the ***instructionpointer*** to a specific value if the ***carryflag*** is set and ***zeroflag*** is not set (conditional jump).

JGE

Set the ***instructionpointer*** to a specific value if the ***carryflag*** is set or the ***zeroflag*** is set(conditional jump).

JL

Set the ***instructionpointer*** to a specific value if the ***carryflag*** is not set and the ***zeroflag*** is not set (conditional jump).

JLE

JMP

Set the ***instructionpointer*** to a specific value (unconditional jump).

JNZ

Set the ***instructionpointer*** to a specific value if the ***zeroflag*** is not set (conditional jump).

JO

Set the ***instructionpointer*** to a specific value if the ***overflowflag*** is set (conditional jump).

JP

Set the ***instructionpointer*** to a specific value if the ***parityflag*** is set (conditional jump).

JPE

Set the ***instructionpointer*** to a specific value if the parity is even (conditional jump).

JPO

Set the ***instructionpointer*** to a specific value if the parity is odd (conditional jump).

JS

Set the ***instructionpointer*** to a specific value if the ***signflag*** is set (conditional jump).

JZ

Set the ***instructionpointer*** to a specific value if the ***zeroflag*** is set (conditional jump).

JNO

Set the ***instructionpointer*** to a specific value if the ***oveflowset*** is not set (conditional jump).

JNP

Set the ***instructionpointer*** to a specific value if the ***parityflag*** is not set (conditional jump).

JA

The same as the ***JG operation***.

JAE

The same as the ***JGE operation***.

JB

The same as the ***JG operation***.

JBE

The same as the ***JGE operation***.

JNA

The same as the ***JL operation***.

JNAE

Set the ***instructionpointer*** to a specific value if the ***carryflag*** is not set or the ***zeroflag*** is set (conditional jump).

JNB

The same as the ***JL operation***.

JNBE

The same as the ***JNAE operation***.

JNE

Set the ***instructionpointer*** to a specific value if the ***zeroflag*** is not set (conditional jump).

JNG

Set the ***instructionpointer*** to a specific value if the ***carryflag*** is not set (conditional jump).

JNGE

The same as the ***JNAEoperation***.

JNL

The same as the ***JGE operation***.

JNLE

The same as the ***JGE operation***.

JNS

Set the ***instructionpointer*** to a specific value if the ***signflag*** is not set (conditional jump).

CALL

Set the ***instructionpointer*** to a specific value and stores the register in ***Stack***. Used for function calls it stores the return address in ***memory***.

RET

Fetches the return address set in the ***CALL operation*** and also restores the ***register*** values to the state from before the ***CALL operation***.

Conditional loops

LOOP

When executed a jump will be performed if the ***CX register*** is not zero. When executed the CX register will be decremented automaticly.

LOOPE

When executed a jump will be performed if the ***CX register*** is not zero and the ***zeroflag*** is set. When executed the CX register will be decremented automaticly.

LOOPNE

When executed a jump will be performed if the ***CX register*** is not zero and the ***zeroflag*** is not set. When executed the CX register will be decremented automaticly.

LOOPNZ

When executed a jump will be performed if the ***CX register*** is zero and the ***zeroflag*** is not set. When executed the CX register will be decremented automaticly.

LOOPZ

When executed a jump will be performed if the ***CX register*** is zero and the ***zeroflag*** is not set. When executed the CX register will be decremented automaticly.

Device in-/output

OUT

Outputs a ***byte register*** to a external device and the port is specified by the ***DX register***.

IN

Reads an value from an external device into a ***byte register*** and the port is specified by the ***DX register***.

ESC

Signals an external device that there are operations avalible for the external device.

INT

Triggers an ***interrupt***.

INTO

Triggers an ***interrupt*** if the ***overflowflag*** is set.

IRET

Returns from an ***interrupt***, simular to the ***RET operation***.

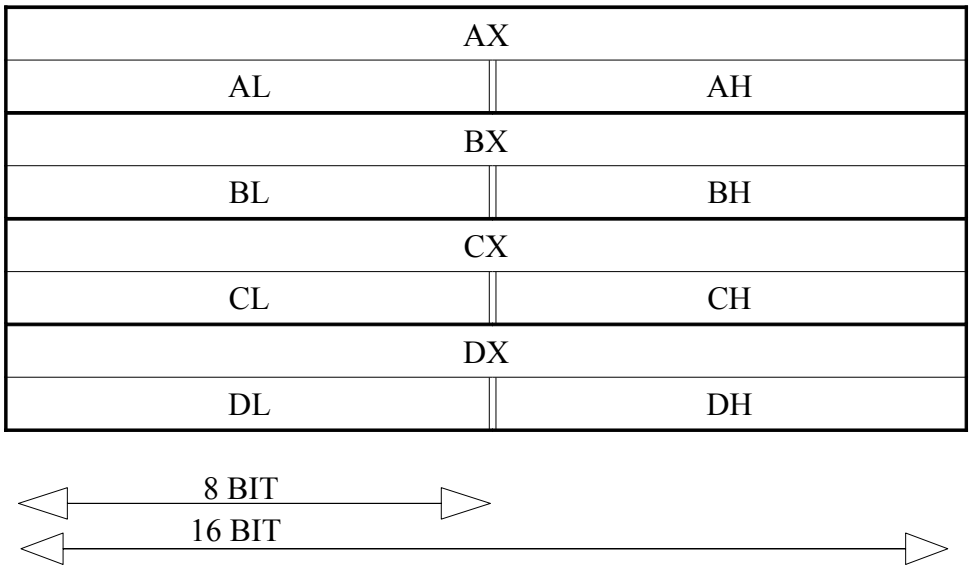
CPU architecture

Registers

The CPU has 14 registers and they are all can hold a value of 16-bit.

Generalpurpose Registers

The first four registers (*AX/BX/CX/DX*) are general purpose registers all of them have a specific use and they are also split into two 8-bit registers. That means it is possible to address the lower and higher byte of a general purpose register seperatly.



AX Register

This register's purpose it that is used as an accumulator that means when ever an ***ALU operation*** is executed the result will automaticly be stored in the AX register if the operation yields thus the AX register is allways the drain component of an ALU operation. The AL/AH registers are also used in the String operations (due to the *ASCII format* beeing 7 bits big/ the size of a "*char*" is 8 bits). AX can also form a 32-bit value with the ***DX register*** with the upper two bytes beeing stored in AX and the lower two byte beeing stored in the ***DX register***. The AX register is also used for sending data to outside devices.

BX Register

This register is used for addressing and can be used for everything else usually used in combination with the ***AX register*** for ***ALU operations***.

CX Register

The CX register is used for counting and for the ***Loop operations***. For convinience the CPU has operations dedicated to incrementing and decrementing the CX register whitch are faster than using the ***INC*** and ***DEC operations***. It is also used in some operations for addressing espacially when dealing with an indexed ***address mode***.

DX Register

The DX register is used as the lower two byte of the DX: ***AX register*** when performing 32 bit operations. The register also specifies the port/ address when the ***IN*** and ***OUT operations*** are performed.

Pointer Registers

Registers five to eight are pointers (*SP/BP/SI/DI*), these pointers can only be addressed as a whole (they can not be split into two one byte registers). They are used for addressing and some operations use these registers by default or expect these registers to be used to calculate the drain/source address.

SP
BP
SI
DI

SP Register

This register is allways used when the operations ***PUSH*** and ***POP*** are performed. The register's main purpose is to address the ***Stack*** and to keep track of the current stack position. The stackpointer should not be used for ***ALU operations*** but it can be used.

BP Register

The Basepointer (BP) is used to address the ***Stack*** but contrary to the ***Stackpointer*** (SP register) it is not used with the ***PUSH*** and ***POP*** operations rather it is used to access arguments for function calls (***CALL operation***).

SI Register

The Sourceindex (SI Register) is used as a pointer to a source location or source tabele in ***memory***. Its main use is for the ***String operations***. This pointer is mainly used to create addresses with the ***DS register***.

DI Register

The Drainindex (DI Register) is a poiner used for calculating drain locations or drain tables in ***memory***. This register's main purpose is to form addresses for the ***String operations*** and forms an address with the ***ES registe***r.

Segment Registers

The next four are segment registers (*CS/DS/SS/ES*), they are two byte registers and the single bytes can not be addressed seperatly. The segment registers all point to a specific point in ***memory*** and are used to section off the ***RAM***. These registers are not meant to be used in ***ALU operations*** but can be.

CS
DS

SS
ES

CS Register

This register marks the begin of the ***Codesegment*** in ***memory***.

DS Register

This register marks the begin of the ***Datasegment*** in ***memory***.

SS Register

This register marks the begin of the ***Stacksegment*** in ***memory***.

ES Register

This register marks the begin of the ***Extrasegment*** in ***memory***.

Special Purpose Registers

These registers have a special purpose and can not be used for other things thus they can not be used in ***ALU operations*** or "normal" ***Move operations.***

Instructionpointer

The Instructionpointer allways points at a location in memory where the next operation is stored. It is used in the FETCH cycle. The register can only be manipulated by un-/conditional jumps and ***function calls***.

Flagregister

The Flagregister does not consits of a normal register but of multiple single bits that make up the register. These bits are set when ever an ALU operation is performed the single bits can also be set by the ***CPU controll and flag manipulation operations***. These bits controll the internal flow of the CPU and also change the behavior of the conditional ***jump operations***.

Flags

Bit		Flagname
0	CF	Carryflag
1	-	-
2	PF	Parityflag
3	-	-
4	AF	AL-Carryflag
5	-	-
6	ZF	Zeroflag
7	SF	Signflag

8	TF	Trapflag
9	IF	Interruptflag
10	DF	Directionflag
11	OF	Overflowflag
12	-	-
13	-	-
14	-	-
15	-	-

- The Carryflag gets set when the result of the last ALU operation is bigger than two bytes or the result of a subtraction is smaller than zero
- The Parityflag gets set when the parity of the result of the last ALU operation is even
- The AL Carryflag gets set when the result of an 8-bit operation is bigger than 8-bit or the result of a subtration is smaller than zero
- The Zeroflag gets set when the result of the last ALU operation is equal to zero
- The Signflag gets set when the result of the last ALU operation is negative
- When Trapflag is set the CPU is in single step mode and pauses after every operation it can also be set by performing an illegal operation
- If the Interruptflag is set the CPU ignores all incomming interrupts it is not set automaticly
- The Directionflag determains the direction strings are processed when it is set the direction from the highes to lowes address and when it is not set strings are processed in the other directio
- The Overflowflag is set when an overflow accoured in the last ALU operation an overflow accours when the resulting sign of an addition is wrong

Memory Addressing

The CPU uses 16-bit registers and has a 16-bit bus but the address-bus is 20-bit big. To calculate a 20-bit address from 16-bit values two registers are used to generate an address one register acts as the offset and the other as the segment.

The segment registers can be used for the segment part of the address while a wider varirety can be used as the offset. When calculating the address the offset is shifted to the left 4 times and than added to the segment.

DS:SI *DS = 1292h, SI = 1123h*

SHL SI, 0004h → 21230h

DS:SI = *DS + 21230h = 224C2h*

The offset can be a register but does not have to be one it can also be an immidiate value (a constand passed as an arrgument).

When addressing the memeory there are two big differences the "normal" memory (Datasegment/ Extrasegment), the Stack and the "Codespace" (Codesegment). While the "normal" memory can be

addressed in many different way and by multiple registers, the Stack can only be addressed using the SP and the BP in the same way the Codespace can only be addressed using the IP (Instructionpointer) and the Codesegment.

Memory Segmentation (example)

Segment	Addres	Value
ES	*7FFFFh*	[...]
	[...]	[...]
	70000h	[...]
SS	*5FFFFh*	[...]
	[...]	[...]
	50000h	[...]
CS	*3FFFFh*	[...]
	[...]	[...]
	30000h	[...]
DS	*2FFFFh*	[...]
	[...]	[...]
	20000h	[...]

It is possible to overlap the segments this happens when the difference between the Segmentregisters is not big enough and this makes it possible to access segments with operations that are not supposed to access this part (segment) of the memory but this method is not adviced since it can lead to curruptin of data and ultimatly the current programm.

Addressing Modes

These are ultimatly the different methods that are used to access the memory and the possible combinations of registers and segments and offsets.

Immplied

The data value/data adress is implicitly associated with the instruction

**PUSHF / POPF*

Register

References the data in a register or a register pair

**MOV AX, BX*

Immediate

the data is provided in the instruction

**MOV AX, #0A7Fh*

Direct

The instructio operand specifies the address where the data is located

**MOV AX, 01A5Fh*

Register Indirect

The instructio specifies a register containing an address, where the data is located. This addressing mode works with SI, DI, BX, BP registers

**MOV AX, DS:SI*

Based

An one byte or two byte operand is added to the contents of a base register (BX or BP), the resulting value is a pointer to a location where the data is stored

**MOV AX,DS:[BX+A7h]*

Indexed

An one byte or two byte operand is added to the contents of a index register (SI or DI), the resulting value is a pointer to a location where the data is stored

**MOV AX,DS:[SI+14h]*

Based Indexed

The content of a base register (BX or BP) is added to the contents of an index register (DI or SI) the resulting value is a pointer to a location of the data

**MOV AX,DS:[BP+SI]*

Based Indexed with displacement

An one byte or two byte operand is added to the of a base register (BX or BP) and the contents of an index register (DI or SI) the resulting value is a pointer to a location of the data

**MOV AX,DS:[BP+SI+#F8h]*

* note that the example operations could be illegal operations, these are just to show the principle on how the different addressing modes work

Assembly

TEXT

List of all valid Operations

MOV (AX/BX/CX/DX/SP/BP/SI/DI),(AX/BX/CX/DX/SP/BP/SI/DI)

MOV (AL/AH/BL/BH/CL/CH/DL/DH), (AL/AH/BL/BH/CL/CH/DL/DH)

MOV (AX/BX/CX/DX),(CS/DS/SS/ES)

MOV (CS/DS/SS/ES),(AX/BX/CX/DX)

MOV (AX/BX/CX/DX/SP/BP/SI/DI), #0000h

MOV (AL/AH/BL/BH/CL/CH/DL/DH), #00h

MOV (AX/BX/CX/DX/SP/BP/SI/DI), $0000h

MOV (AL/AH/BL/BH/CL/CH/DL/DH), $0000h

MOV $0000h, (AX/BX/CX/DX/SP/BP/SI/DI)

MOV $0000h, (AL/AH/BL/BH/CL/CH/DL/DH)

MOV AL/AH/BL/BH/CL/CH/DL/DH/AX/BX/CX/DX, ES:BX/DS:BX/DS:SI/ES:DI/SS:BP

MOV ES:BX/DS:BX/DS:SI/ES:DI/SS:BP, AL/AH/BL/BH/CL/CH/DL/DH/AX/BX/CX/DX

MOV (AX/BX/CX/DX/SP/BP/SI/DI), ES:BX+#00h/DS:BX+#00h/SS:BP+#00h

MOV DS:BX+#00h/DS:BX+#00h/SS:BP+#00h, (AX/BX/CX/DX/SP/BP/SI/DI)

MOV (AX/BX/CX/DX), DS:SI+#00h/ES:DI+#00h

MOV DS:SI+#00h/ES:DI+#00h, (AX/BX/CX/DX)

MOV (AX/BX/CX/DX), SS:BP+SI/SS:BP+DI/DS:BX+SI/DS:BX+DI/ES:BX+SI/ES:BX+DI

MOV SS:BP+SI/SS:BP+DI/DS:BX+SI/ES:BX+DI/ES:BX+SI/ES:BX+DI, (AX/BX/CX/DX)

MOV (AX/BX/CX/DX),
SS:BP+SI+#00h/SS:BP+DI+#00h/DS:BX+SI+#00h/DS:BX+DI+#00h/ES:BX+SI+#00h/ES:BX+DI+#00h

MOV SS:BP+SI+#00h/ SS:BP+DI+#00h/ DS:BX+SI+#00h/ ES:BX+DI+#00h/ ES:BX+SI+#00h/ ES:BX+DI+#00h, (AX/BX/CX/DX)

XCHG (AX/BX/CX/DX/SP/BP/SI/DI), (AX/BX/CX/DX/SP/BP/SI/DI)

LAHF

SAHF

XLAT (MOV AL,DS:[BX+AL])

POP (AX/BX/CX/DX/SP/BP/SI/DI)

PUSH (AX/BX/CX/DX/SP/BP/SI/DI)

POPF

PUSHF

AAA

AAD

AAM

AAS

DAA

DAS

CBW

CWD

NEG AX

NOT AX

INC AX

DEC AX

ADD AX,(AX/BX/CX/BX/SP/BP/SI/DI)

ADD AL,(AL/AH/BL/BH/CL/CH/DL/DH)

ADD AX,#0000/$0000/ES:BX/DS:BX/DS:SI/ES:DI/SS:BP

ADC AX,(AX/BX/CX/BX/SP/BP/SI/DI)

ADC AL,(AL/AH/BL/BH/CL/CH/DL/DH)

ADC AX,#0000/$0000/ES:BX/DS:BX/DS:SI/ES:DI/SS:BP

AND AX,(AX/BX/CX/BX/SP/BP/SI/DI)

AND AL,(AL/AH/BL/BH/CL/CH/DL/DH)

AND AX,#0000/$0000/ES:BX/DS:BX/DS:SI/ES:DI/SS:BP

DIV AX,AX,(AX/BX/CX/BX/SP/BP/SI/DI)

DIV AL,AL,(AL/AH/BL/BH/CL/CH/DL/DH)

DIV AX,AX,#0000/$0000/ES:BX/DS:BX/DS:SI/ES:DI/SS:BP

XOR AX,AX,(AX/BX/CX/BX/SP/BP/SI/DI)

XOR AL,AL,(AL/AH/BL/BH/CL/CH/DL/DH)

XOR AX,AX,#0000/$0000/ES:BX/DS:BX/DS:SI/ES:DI/SS:BP

SUB AX,AX,(AX/BX/CX/BX/SP/BP/SI/DI)

SUB AL,AL,(AL/AH/BL/BH/CL/CH/DL/DH)

SUB AX,AX,#0000/$0000/ES:BX/DS:BX/DS:SI/ES:DI/SS:BP

MUL AX,AX,(AX/BX/CX/BX/SP/BP/SI/DI)

MUL AL,AL,(AL/AH/BL/BH/CL/CH/DL/DH)

MUL AX,AX,#0000/$0000/ES:BX/DS:BX/DS:SI/ES:DI/SS:BP

OR AX,AX,(AX/BX/CX/BX/SP/BP/SI/DI)

OR AL,AL,(AL/AH/BL/BH/CL/CH/DL/DH)

OR AX,AX,#0000/$0000/ES:BX/DS:BX/DS:SI/ES:DI/SS:BP

ROL AX,AX,(AX/BX/CX/BX/SP/BP/SI/DI)

ROL AL,AL,(AL/AH/BL/BH/CL/CH/DL/DH)

ROL AX,AX,#0000/$0000/ES:BX/DS:BX/DS:SI/ES:DI/SS:BP

ROR AX,AX,(AX/BX/CX/BX/SP/BP/SI/DI)

ROR AL,AL,(AL/AH/BL/BH/CL/CH/DL/DH)

ROR AX,AX,#0000/$0000/ES:BX/DS:BX/DS:SI/ES:DI/SS:BP
SHL AX,AX,(AX/BX/CX/BX/SP/BP/SI/DI)
SHL AL,AL,(AL/AH/BL/BH/CL/CH/DL/DH)
SHL AX,AX,#0000/$0000/ES:BX/DS:BX/DS:SI/ES:DI/SS:BP
SHR AX,AX,(AX/BX/CX/BX/SP/BP/SI/DI)
SHR AL,AL,(AL/AH/BL/BH/CL/CH/DL/DH)
SHR AX,AX,#0000/$0000/ES:BX/DS:BX/DS:SI/ES:DI/SS:BP
SAL AX,AX,(AX/BX/CX/BX/SP/BP/SI/DI)
SAL AL,AL,(AL/AH/BL/BH/CL/CH/DL/DH)
SAL AX,AX,#0000/$0000/ES:BX/DS:BX/DS:SI/ES:DI/SS:BP
SAR AX,AX,(AX/BX/CX/BX/SP/BP/SI/DI)
SAR AL,AL,(AL/AH/BL/BH/CL/CH/DL/DH)
SAR AX,AX,#0000/$0000/ES:BX/DS:BX/DS:SI/ES:DI/SS:BP
SBB AX,AX,(AX/BX/CX/BX/SP/BP/SI/DI)
SBB AL,AL,(AL/AH/BL/BH/CL/CH/DL/DH)
SBB AX,AX,#0000/$0000/ES:BX/DS:BX/DS:SI/ES:DI/SS:BP
RCL AX,AX,(AX/BX/CX/BX/SP/BP/SI/DI)
RCL AL,AL,(AL/AH/BL/BH/CL/CH/DL/DH)
RCL AX,AX,#0000/$0000/ES:BX/DS:BX/DS:SI/ES:DI/SS:BP
RCR AX,AX,(AX/BX/CX/BX/SP/BP/SI/DI)
RCR AL,AL,(AL/AH/BL/BH/CL/CH/DL/DH)
RCR AX,AX,#0000/$0000/ES:BX/DS:BX/DS:SI/ES:DI/SS:BP
CMP AX,AX,(AX/BX/CX/BX/SP/BP/SI/DI)
CMP AL,AL,(AL/AH/BL/BH/CL/CH/DL/DH)
CMP AX,AX,#0000/$0000/ES:BX/DS:BX/DS:SI/ES:DI/SS:BP

CLC
CLD
CLI
CMC
STC
STD
STI

LOCK

WAIT

HLT

NOP