CS2507 Computer Architecture

Lecture 5

The Instruction Set (continued)

The Intel 8086 Processor

The 8086 processor is a 16-bit CPU. It has a 16-bit accumulator (A register) with a 16-bit PC, known now as the Instruction Pointer (IP).

When the 8086 was being designed, it was decided to increase the addressable main memory size to 1 megabyte (1M). Thus, the memory address length is 20 bits, and many of the extensions to the 8085 architecture arise from dealing with this increased address length.

One of the aims of the 8086 design was to maintain a level of backwards compatibility with the 8085 so that old software could easily be adapted to run on the new processor.

A far back as its 8080 development system, Intel had had a way of regarding memory that involved thinking about a program as being composed of four segments, each having a different purpose. These segments were called the Code Segment, the Data Segment, the Stack Segment and Extra Segment. The 8086 processor introduced architectural support for this logical separation, making a virtue out of the necessity of expanding the memory addresses.

Segments are supported by the inclusion of four 16-bit segment registers, the Code Segment Register (CS), the Data Segment Register (DS), the Stack Segment Register (SS) and the Extra Segment Register (ES). As these registers address a 1M address space, it is assumed that the four least significant bits (LSBs) of the segment address are zero. Thus, segments are aligned (i.e. begin at) a 16-byte boundary (an address evenly divisible by 16).

All other addresses produced in the processor are treated as 16-bit offsets within a particular segment. Thus, the maximum size of a memory segment is 64K. Segments are not mutually exclusive: they can overlap in any number of locations. Indeed, setting all segment registers to zero results in the previous 64K memory model.

Offsets are combined with segment register contents to generate effective addresses. Usually, the IP contains an offset within the code segment and the SP contains an offset within the stack segment. This may be specified at assembly level by writing CS:IP and SS:SP. Operand offsets usually act with respect to the DS or ES registers.

To form a 20-bit effective address, the four LSBs of the offset become bits 0-3 of the

effective address. Bits 4 - 19 of the effective address are constructed by adding bits 4-15 of the offset to bits 0-15 of the segment address.

For instance, suppose the CS register contains 302H and the IP contains 2A57H. Then, the code segment address is 3020H and the effective address is 5A77H.

Other registers are:

AX	16-bit register AX or 8-bit registers AH and AL	Accumulator
BX	16-bit register BX or 8-bit registers BH and BL	Base
CX	16-bit register CX or 8-bit registers CH and CL	Count
DX	16-bit register DX or 8-bit registers DH and DL	Data
SP	16-bit address register	Stack pointer
BP	16-bit address register	Base pointer
Flags	16-bit flags register	Contains 6 condition flags + 3 control flags

Instructions are variable-length and can occupy 1, 2 or 3, 4, 5 7 or 9 bytes.

Operand addressing modes available are implied, register, immediate, direct, register indirect, memory indirect, based and indexed. Indeed, although separate based and indexed addressing modes may be discussed as an abstraction, 80x86 addressing is best treated by saying that an operand address may be formed by adding a displacement to the contents of two other registers.

As in the case of the 8085, conditions are indicated by setting 1-bit flags to show the result of an ALU operation. In addition, there are control flags that can be set or reset in order to control CPU operation. The condition flags can be tested using conditional jump, call or return instructions. There are 6 condition flags: Overflow (OF), Sign (SF), Zero (ZF), Auxiliary Carry (AF), Parity (PF) and Carry (CF). There are 3 control flags: Trap (T), Interrupt (IF) and Direction (DF). The 9 flags are padded with 7 extra bits to form a 16-bit Flags register, so that SF, ZF, AF, PF and CF occupy bits 7, 6, 4, 2 and 0 of the Flags register, the same positions occupied by these flags in the 8085. The OF occupies bit 11 and the control flags DF, IF and TF are at bit positions 10, 9 and 8 of the Flags register. The Flags register is accessible via PUSHF, POPF, LAHF and SAHF instructions. In addition, the CF, IF and DF can be individually set or cleared..

The Overflow flag is set when a carry into the MSB does not result in a carry out, or when a carry out of the MSB occurs without a carry in. The flag indicates arithmetic overflow on signed arithmetic operations. The Sign flag is set when the result of an ALU operation is negative, according to the Two's Complement number representation. The Zero flag is set when the result of an operation is zero. The Auxiliary Carry flag is not explicitly accessible by conditional test instructions, but is used internally in the Binary Coded Decimal (BCD) correction instruction, Decimal Adjust Accumulator (DAA). The Parity flag is set if the A register contains an even number of 1 bits, a condition known as even parity. The Carry flag is set when a carry out of bit 7 occurs as a result of an arithmetic operation.

Memory is byte-organised, with the Little-Endian convention employed for the storage of multiple-byte quantities. The restart address is FFFF0H and Intel literature "reserves" locations FFFF0H to FFFFFH (16 bytes) for a jump to the initial bootstrap loader. Memory locations 00000H to 003FFH are reserved for the interrupt vector, so 256 entries of 4 bytes each occupy 1K memory bytes. Note that full address quantities in this processor occupy 4 bytes, 2 each for the segment address and the offset.

Stack is located off-chip, in main memory and is accessed through SS: SP. The stack expands into memory locations having lower addresses than those already occupied. The SP is decremented before a byte is saved on the stack. Only 16-bit quantities are moved to and from the stack.

There is an allowance for a separate address space via Input and Output instructions.

The ALU directly provides for addition, subtraction, increment, decrement, multiplication and division of signed and unsigned 8-bit and 16-bit integers. Special instructions allow for adjustments to facilitate BCD and ASCII arithmetic. Comparisons are performed by internal subtraction with modification to allow for special cases. Logical AND, OR, NOT and XOR operations are provided, as are multiple-bit rotations and shifts of 8-, 9-, 16- and 17-bit quantities.

Special string instructions allow easy movement and comparison of strings as well as byte searches.

Machine control includes instructions to enable and disable interrupts, No-operation and Halt instructions. An Escape instruction allows the CPU to communicate a calculation to a numerical coprocessor. A Wait instruction puts the processor into a wait state until it receives a signal from a coprocessor or external device. A Lock instruction permits the processor to

share an external bus with several other processors.

Software interrupts allow 256 levels of specified Supervisor Call (SVC). In addition, there are two 1-byte software interrupts.

Instructions by Class

Data Movement Group

MOV ra, rb	Move (Copy) contents of rb to ra	r[a] <– r[b]
MOV memaddr, r	Move contents of r to memory	m<memaddr> <– r
MOV r, memaddr	Move contents of memory to register	r <– m<memaddr>
MOV r, data	Move immediate data to register	r8 <– data
MOV memaddr, data	Move immediate data to memory	m<memaddr> <– data
XCHG ra, rb	Exchange contents of ra and rb	ra <–> rb
XCHG memaddr, r	Exchange contents of mem & reg	m<memaddr> <–> r
XCHG r, memaddr	Exchange contents of mem & reg	r <–> m<memaddr>
XLAT	Translate AL from table	AL <– m<BX + AL>
LEA r16, memaddr	Load effective address	r16 <– offset(memaddr)
LDS r16, memaddr	Load Segment Register	
	r16[0..7] <– m<memaddr>	
	r16[8..15] <– m<memaddr + 1>	
	DS[0..7] <– m<memaddr + 2>	
	DS[8..15] <– m<memaddr + 3>	
LES r16, memaddr	Load Segment Register	
	r16[0..7] <– m<memaddr>	
	r16[8..15] <– m<memaddr + 1>	
	ES[0..7] <– m<memaddr + 2>	
	ES[8..15] <– m<memaddr + 3>	
LAHF	Load AH with Flags	AH <– Flags[0..7]

SAHF	Store AH into Flags	Flags[0..7] <– AH

Stack Operations

PUSH AX	SP <– SP - 1; m<SP> <– AH; SP <– SP - 1; m<SP> <– AL
PUSH BX	SP <– SP - 1; m<SP> <– BH; SP <– SP - 1; m<SP> <– BL
PUSH CX	SP <– SP - 1; m<SP> <– CH; SP <– SP - 1; m<SP> <– CL
PUSH DX	SP <– SP - 1; m<SP> <– DH; SP <– SP - 1; m<SP> <– DL
PUSH CS	SP <– SP - 1; m<SP> <– CS[8..15];
	SP <– SP - 1; m<SP> <– CS[0..7]
PUSH DS	SP <– SP - 1; m<SP> <– DS[8..15];
	SP <– SP - 1; m<SP> <– DS[0..7]
PUSH ES	SP <– SP - 1; m<SP> <– ES[8..15];
	SP <– SP - 1; m<SP> <– ES[0..7]
PUSH SS	SP <– SP - 1; m<SP> <– SS[8..15];
	SP <– SP - 1; m<SP> <– SS[0..7]
PUSH memaddr	SP <– SP - 1; m<SP> <– m<memaddr + 1>;
	SP <– SP - 1; m<SP> <– m<memaddr>
PUSHF	SP <– SP - 1; m<SP> <– Flags[8..15];
	SP <– SP - 1; m<SP> <– Flags[0..7]
POP AX	AL <– m<SP>; SP <– SP + 1; AH <– m<SP>; SP <– SP + 1
POP BX	AL <– m<SP>; SP <– SP + 1; AH <– m<SP>; SP <– SP + 1
POP CX	AL <– m<SP>; SP <– SP + 1; AH <– m<SP>; SP <– SP + 1
POP DX	AL <– m<SP>; SP <– SP + 1; AH <– m<SP>; SP <– SP + 1
POP CS	CS[0..7] <– m<SP>; SP <– SP + 1;
	CS[8..15] <– m<SP>; SP <– SP + 1;
POP DS	DS[0..7] <– m<SP>; SP <– SP + 1;
	DS[8..15] <– m<SP>; SP <– SP + 1;

POP ES	ES[0..7] <– m<SP>; SP <– SP + 1;
	ES[8..15] <– m<SP>; SP <– SP + 1;
POP SS	SS[0..7] <– m<SP>; SP <– SP + 1;
	SS[8..15] <– m<SP>; SP <– SP + 1;
POP memaddr	m<memaddr> <– m<SP>; SP <– SP + 1;
	m<memaddr + 1> <– m<SP>; SP <– SP + 1;
POPF	Flags[0..7] <– m<SP>; SP <– SP + 1;
	Flags[8..15] <– m<SP>; SP <– SP + 1;

Branch Group: Jump, Call, Return

JMP near_addr	PC[0..7] <– near_addr[0..7]
	PC[8..15] <– near_addr[8-.15]
JMP short	PC <– PC + short [0..7]
JMP r16	PC <– r16
JMP var16	PC[0..7] <– m<var16>
	PC[8..15] <– m<var16 + 1>
JMP far_addr	CS <– seg(far_addr);
	PC <– offset(far_addr)
JMP segr:r16	CS <– segr;
	PC <– r16
JMP var32	PC[0..7] <– m<addr32>
	PC[8..15] <– m<addr32 + 1>
	CS[0..7] <– m<addr32 + 2>
	CS[8..15] <– m<addr32 + 3>

JO short	**if** of **then** PC <– short
JNO short	**if** !of **then** PC <– short
JC/JB/JNAE short	**if** cf **then** PC <– PC + short[0..7]
JNC/JNB/JAE short	**if** !cf **then** PC <– PC + short[0..7]
JE/JZ short	**if** zf **then** PC <– PC + short[0..7]
JNE/JNZ short	**if** !zf **then** PC <– PC + short[0..7]
JBE/JNA short	**if** (cf OR zf) **then** PC <– PC + short[0..7]
JNBE/JA short	**if** !(cf OR zf) **then** PC <– PC + short[0..7]
JS short	**if** sf **then** PC <– PC + short[0..7]
JNS short	**if** !sf **then** PC <– PC + short[0..7]
JP/JPE short	**if** pf **then** PC <– PC + short[0..7]
JNP/JPO short	**if** !pf **then** PC <– PC + short[0..7]
JL/JNGE short	**if** (sf XOR of) **then** PC <– PC + short[0..7]
JNL/JGE short	**if** !(sf XOR of) **then** PC <– PC + short[0..7]
JLE/JNG short	**if** (zf + (sf XOR of)) **then** PC <– PC + short[0..7]
JNLE/JG short	**if** !(zf + (sf XOR of)) **then** PC <– PC + short[0..7]

LOOP short	CX <– CX - 1;
	if (CX != 0) **then** PC <– PC + short [0..7]
LOOPZ/LOOPE addr16	
	CX <– CX - 1;
	if ((CX != 0) AND (zf)) **then** PC <– PC + short [0..7]
LOOPNZ/LOOPNE addr16	
	CX <– CX - 1;
	if ((CX != 0) AND (!zf)) **then** PC <– PC + short [0..7]
JCXZ short	**if** (CX = 0) **then** PC <– PC + short [0..7]

CALL nearproc_addr	SP <– SP - 1; m<SP> <– PC[8..15];
	SP <– SP - 1; m<SP> <– PC[0..7];
	PC <- nearproc_addr
CALL r16	SP <– SP - 1; m<SP> <– PC[8..15];
	SP <– SP - 1; m<SP> <– PC[0..7];
	PC <- r16
CALL addr16	SP <– SP - 1; m<SP> <– PC[8..15];
	SP <– SP - 1; m<SP> <– PC[0..7];
	PC <- m<addr16>
CALL farproc_addr	SP <– SP - 1; m<SP> <– CS[8..15];
	SP <– SP - 1; m<SP> <– CS[0..7];
	CS <- seg(farproc_addr);
	SP <– SP - 1; m<SP> <– PC[8..15];
	SP <– SP - 1; m<SP> <– PC[0..7];
	PC <- offset(farproc_addr)
CALL segr:r16	SP <– SP - 1; m<SP> <– CS[8..15];
	SP <– SP - 1; m<SP> <– CS[0..7];
	CS <- segr;
	SP <– SP - 1; m<SP> <– PC[8..15];
	SP <– SP - 1; m<SP> <– PC[0..7];
	PC <- r16

CALL addr32	SP <– SP - 1; m<SP> <– CS[8..15];
	SP <– SP - 1; m<SP> <– CS[0..7];
	CS <- seg(addr32);
	SP <– SP - 1; m<SP> <– PC[8..15];
	SP <– SP - 1; m<SP> <– PC[0..7];
	PC <- offset(addr32)
RET(N)	PC[0..7]<– m<SP>; SP <– SP + 1;
	PC[8..15] <– m<SP>; SP <– SP + 1
RET(N) imm16	PC[0..7]<– m<SP>; SP <– SP + 1;
	PC[8..15] <– m<SP>; SP <– SP + 1;
	SP <– SP + imm16
RET(F)	PC[0..7]<– m<SP>; SP <– SP + 1;
	PC[8..15] <– m<SP>; SP <– SP + 1;
	CS[0..7]<– m<SP>; SP <– SP + 1;
	CS[8..15] <– m<SP>; SP <– SP + 1
RET(F) imm16	PC[0..7]<– m<SP>; SP <– SP + 1;
	PC[8..15] <– m<SP>; SP <– SP + 1;
	CS[0..7]<– m<SP>; SP <– SP + 1;
	CS[8..15] <– m<SP>; SP <– SP + 1
	SP <– SP + imm16

Arithmetic Group

INC r8	r8 <– r8 + 1
INC r16	r16<– r16+ 1
INC addr	m< addr > <– m< addr > + 1

DEC r8	r8 <– r8 - 1
DEC r16	r16<– r16- 1
DEC addr	m< addr > <– m< addr > - 1
ADD ra, rb	ra <– ra + rb
ADC ra, rb	ra <– ra + rb + CF
ADD ra, addr	ra <– ra + m< addr >
ADC ra, addr	ra <– ra + m< addr > + CF
ADD addr, ra	m< addr > <– m< addr > + ra
ADC addr, ra	m< addr > <– m< addr > + ra + CF
ADD ra, immed	ra <– ra + immed
ADC ra, immed	ra <– ra + immed + CF
ADD addr, immed	m< addr > <– m< addr > + immed
ADC addr, immed	m< addr > <– m< addr > + immed + CF
SUB ra, rb	ra <– ra - rb
SBB ra, rb	ra <– ra - rb - CF
SUB ra, addr	ra <– ra - m< addr >
SBB ra, addr	ra <– ra - m< addr > - CF
SUB addr, ra	m< addr > <– m< addr > - ra
SBB addr, ra	m< addr > <– m< addr > - ra - CF
SUB ra, immed	ra <– ra - immed
SBB ra, immed	ra <– ra - immed - CF
SUB addr, immed	m< addr > <– m< addr > - immed
SBB addr, immed	m< addr > <– m< addr > - immed - CF
CMP ra, rb	ra - rb (Flags affected; no operation result saved)
CMP ra, addr	ra - m< addr > (Flags affected; no operation result saved)
CMP ra, immed	ra - immed (Flags affected; no operation result saved)

CMP addr, ra	m< addr > - rb (Flags affected; no operation result saved)
CMP addr, immed	m< addr > - immed (Flags affected; no operation result saved)
NOT ra	ra <– NOT(ra)
NOT addr	m<addr> <– NOT(m< addr >)
NEG ra	ra <– NOT(ra) + 1
NEG addr	m<addr> <– NOT(m< addr >) + 1
AAA	**if** (AL[0..3] > 9) **OR** (af = 1) **then**
	{AL <– AL + 6; AH <– AH + 1; CF <– AF <– 1};
	else
	{CF <– AF <– 0};
	AL <– AL **AND** 0FH
DAA	**if** (AL [0..3] > 9) **OR** (af = 1) **then** {AL <– AL + 6; af <– 1};
	if (AL [4..8] > 9) **OR** (cf = 1) **then** {AL <– AL + 60H; cf <– 1}
AAS	**if** (AL[0..3] > 9) **OR** (af = 1) **then**
	{AL <– AL - 6; AH <– AH - 1; CF <– AF <– 1}
	else
	{CF <– AF <– 0};
	AL <– AL **AND** 0FH
DAS	**if** (AL [0..3] > 9) **OR** (af = 1) **then** {AL <– AL - 6; af <– 1};
	if (AL [4..8] > 9) **OR** (cf = 1) **then** {AL <– AL - 60H; cf <– 1}
MUL r8	AX <– AL * r8
MUL m8	AX <– AL * m8
MUL r16	DX:AX <– AX * r16
MUL m16	DX:AX <– AX * m16
IMUL r8	AX <– AL * r8 ;Signed
IMUL m8	AX <– AL * m8 ;Signed

IMUL r16	DX:AX <– AX * r16	;Signed
IMUL m16	DX:AX <– AX * m16	;Signed
AAM	AH <– AL / 10; AL <– AL % 10	
AAM imm	AH <– AL / imm; AL <– AL % imm	
DIV r8	AL <– AX / r8; AH <– AX % r8	
DIV m8	AL <– AX / m8; AH <– AX % m8	
DIV r16	AX <– DX:AX / r16; DX <– DX:AX % r16	
DIV m16	AX <– DX:AX / m16; DX <– DX:AX % m16	
IDIV r8	AL <– AX / r8; AH <– AX % r8	;Signed
IDIV m8	AL <– AX / m8; AH <– AX % m8	;Signed
IDIV r16	AX <– DX:AX / r16; DX <– DX:AX % r16	;Signed
IDIV m16	AX <– DX:AX / m16; DX <– DX:AX % m16	;Signed
AAD	AL <– AL + AH * 10; AH <– 0	
AAD imm	AL <– AL + AH * imm; AH <– 0	
CBW	AH[0..7] <– AL[7]	
CWD	DX[0..15] <– AX[15]	

Logical Group

AND ra, rb	ra <– ra AND rb	
AND addr, rb	m< addr > <– m< addr > AND rb	
AND ra, addr	ra <– ra AND m< addr >	
AND ra, immed	ra <– ra AND immed	
AND addr, immed	ra <– m< addr > AND immed	
TEST ra, rb	ra AND rb	;Set flags only
TEST addr, rb	m< addr > AND rb	; – no assignment
TEST ra, immed	ra AND immed	
TEST addr, immed	m< addr > AND immed	

OR ra, rb	ra <– ra OR rb
OR addr, rb	m< addr > <– m< addr > OR rb
OR ra, addr	ra <– ra OR m< addr >
OR ra, immed	ra <– ra OR immed
OR addr, immed	ra <– m< addr > OR immed
XOR ra, rb	ra <– ra XOR rb
XOR addr, rb	m< addr > <– m< addr > XOR rb
XOR ra, addr	ra <– ra XOR m< addr >
XOR ra, immed	ra <– ra XOR immed
XOR addr, immed	ra <– m< addr > XOR immed
SHL/SAL ra, 1	cf <– ra[msb]; ra[msb] <– ra[msb - 1]; …
	…; ra[lsb + 1] <– ra[lsb]; ra[lsb]<– 0
SHL/SAL m, 1	cf <– m[msb]; m[msb] <– m[msb - 1]; …
	…; m[lsb + 1] <– m[lsb]; m[lsb]<– 0
SHL/SAL ra, CL	**repeat**
	cf <– ra[msb]; ra[msb] <– ra[msb - 1]; …
	…; ra[lsb + 1] <– ra[lsb]; ra[lsb]<– 0
	CL **times**;
SHL/SAL m, CL	**repeat**
	cf <– m[msb]; m[msb] <– m[msb - 1]; …
	…; m[lsb + 1] <– m[lsb]; m[lsb]<– 0
	CL **times**;
SHR ra, 1	cf <– ra[lsb]; ra[lsb] <– ra[lsb + 1]; …
	…; ra[msb - 1] <– ra[msb]; ra[msb] <– 0;
SHR m, 1	cf <– m[lsb]; m[lsb] <– m[lsb + 1]; …
	…; m[msb - 1] <– m[msb]; m[msb] <– 0;

SHR ra, CL **repeat**

cf <– ra[lsb]; ra[lsb] <– ra[lsb + 1]; …

…; ra[msb - 1] <– ra[msb]; ra[msb] <– 0;

CL **times**;

SHR m, CL **repeat**

cf <– ra[lsb]; m[lsb] <– m[lsb + 1]; …

…; m[msb - 1] <– m[msb]; m[msb] <– 0;

CL **times**;

SAR ra, 1 cf <– ra[lsb]; ra[lsb] <– ra[lsb + 1]; …

…; ra[msb - 1] <– ra[msb];

SAR m, 1 cf <– m[lsb]; m[lsb] <– m[lsb + 1]; …

…; m[msb - 1] <– m[msb];

SAR ra, CL **repeat**

cf <– ra[lsb]; ra[lsb] <– ra[lsb + 1]; …

…; ra[msb - 1] <– ra[msb];

CL **times**;

SAR m, CL **repeat**

cf <– ra[lsb]; m[lsb] <– m[lsb + 1]; …

…; m[msb - 1] <– m[msb];

CL **times**;

RCL ra, 1 tmp <– cf; cf <– ra[msb]; ra[msb] <– ra[msb - 1]; …

…; ra[lsb + 1] <– ra[lsb]; ra[lsb]<– tmp

RCL m, 1 tmp <– cf; cf <– m[msb]; m[msb] <– m[msb - 1]; …

…; m[lsb + 1] <– m[lsb]; m[lsb]<– tmp

RCL ra, CL **repeat**

tmp <– cf; cf <– ra[msb]; ra[msb] <– ra[msb - 1]; …

	…; ra[lsb + 1] <– ra[lsb]; ra[lsb]<– tmp
	CL **times**;
RCL m, CL	**repeat**
	tmp <– cf; cf <– m[msb]; m[msb] <– m[msb - 1]; …
	…; m[lsb + 1] <– m[lsb]; m[lsb]<– cf
	CL **times**;
RCR ra, 1	cf <– ra[lsb]; ra[lsb] <– ra[lsb + 1]; …
	…; ra[msb - 1] <– ra[msb]; ra[msb] <– cf;
RCR m, 1	cf <– m[lsb]; m[lsb] <– m[lsb + 1]; …
	…; m[msb - 1] <– m[msb]; m[msb] <– cf;
RCR ra, CL	**repeat**
	cf <– ra[lsb]; ra[lsb] <– ra[lsb + 1]; …
	…; ra[msb - 1] <– ra[msb]; ra[msb] <– cf;
	CL **times**;
RCR m, CL	**repeat**
	cf <– ra[lsb]; m[lsb] <– m[lsb + 1]; …
	…; m[msb - 1] <– m[msb]; m[msb] <– cf;
	CL **times**;
ROL ra, 1	cf <– ra[msb]; ra[msb] <– ra[msb - 1]; …
	…; ra[lsb + 1] <– ra[lsb]; ra[lsb]<– cf
ROL m, 1	cf <– m[msb]; m[msb] <– m[msb - 1]; …
	…; m[lsb + 1] <– m[lsb]; m[lsb]<– cf
ROL ra, CL	**repeat**
	cf <– ra[msb]; ra[msb] <– ra[msb - 1]; …
	…; ra[lsb + 1] <– ra[lsb]; ra[lsb]<– cf

	CL **times**;
ROL m, CL	**repeat**
	cf <– m[msb]; m[msb] <– m[msb - 1]; …
	…; m[lsb + 1] <– m[lsb]; m[lsb]<– cf
	CL **times**;
ROR ra, 1	cf <– ra[lsb]; ra[lsb] <– ra[lsb + 1]; …
	…; ra[msb - 1] <– ra[msb]; ra[msb] <– cf;
ROR m, 1	cf <– m[lsb]; m[lsb] <– m[lsb + 1]; …
	…; m[msb - 1] <– m[msb]; m[msb] <– cf;
ROR ra, CL	**repeat**
	cf <– ra[lsb]; ra[lsb] <– ra[lsb + 1]; …
	…; ra[msb - 1] <– ra[msb]; ra[msb] <– cf;
	CL **times**;
ROR m, CL	**repeat**
	cf <– ra[lsb]; m[lsb] <– m[lsb + 1]; …
	…; m[msb - 1] <– m[msb]; m[msb] <– cf;
	CL **times**;

Machine Control Group

CLC	cf <– 0
STC	cf <– 1
CMC	cf <– NOT cf
CLD	df <– 0
STD	df <– 1
CLI	if <– 0
	Disable interrupts **after** execution of **this** instruction
STI	if <– 1
	Enable interrupts **after** execution of **next** instruction
NOP	Do nothing
HLT	Halt the instruction cycle.
WAIT	; for coprocessor operations
ESC	; for coprocessor operations
LOCK	; for multiple processor operations

Software Interrupt Group

INT code	SP <– SP - 1; m<SP> <– Flags[8..15];
	SP <– SP - 1; m<SP> <– Flags[0..7];
	SP <– SP - 1; m<SP> <– CS[8..15];
	SP <– SP - 1; m<SP> <– CS[0..7];
	SP <– SP - 1; m<SP> <– PC[8..15];
	SP <– SP - 1; m<SP> <– PC[0..7];
	PC[0..7] <- m< code*4 >
	PC[8..15] <– m< code*4 + 1 >
	CS[0..7] <– m< code*4 + 2 >
	CS[8..15] <– m< code*4 + 3 >

INT3 SP <– SP - 1; m<SP> <– Flags[8..15];

SP <– SP - 1; m<SP> <– Flags[0..7];

SP <– SP - 1; m<SP> <– CS[8..15];

SP <– SP - 1; m<SP> <– CS[0..7];

SP <– SP - 1; m<SP> <– PC[8..15];

SP <– SP - 1; m<SP> <– PC[0..7];

PC[0..7] <- m< 3*4 >

PC[8..15] <– m< 3*4 + 1 >

CS[0..7] <– m< 3*4 + 2 >

CS[8..15] <– m< 3*4 + 3 >

INTO **if** (of) **then {**

SP <– SP - 1; m<SP> <– Flags[8..15];

SP <– SP - 1; m<SP> <– Flags[0..7];

SP <– SP - 1; m<SP> <– CS[8..15];

SP <– SP - 1; m<SP> <– CS[0..7];

SP <– SP - 1; m<SP> <– PC[8..15];

SP <– SP - 1; m<SP> <– PC[0..7];

PC[0..7] <- m< 4*4 >

PC[8..15] <– m< 4*4 + 1 >

CS[0..7] <– m< 4*4 + 2 >

CS[8..15] <– m< 4*4 + 3 > }

IRET PC[0..7]<– m<SP>; SP <– SP + 1;

PC[8..15] <– m<SP>; SP <– SP + 1;

CS[0..7]<– m<SP>; SP <– SP + 1;

CS[8..15] <– m<SP>; SP <– SP + 1

Flags[0..7]<– m<SP>; SP <– SP + 1;

Flags[8..15] <– m<SP>; SP <– SP + 1

Note: The INT *code* operand is limited to a value between 0 and 255. Therefore, the interrupt vector extends from 0 to 3FFH.

Power-on or RESET sets the PC to FFFF0H.

I/O Group

IN AL, port8	AL <– io<port8>
IN AX, port8	AX <– io<port8>
IN AL, DX	AL <– io<DX>
IN AX, DX	AX <– io<DX>
OUT port8, AL	io<port8><– AL
OUT port8, AX	io<port8> <– AX
OUT DX, AL	io<DX> <– AL
OUT DX, AX	io<DX> <– AX

String Instructions

```
MOVSB       m<ES:DI> <– m<DS:SI>;
            if (df == 0) {
              DI <– DI + 1;
              SI <– SI + 1 }
            else {
              DI <– DI - 1;
              SI <– SI - 1 }
```

MOVSW

```
m<ES:DI> <– m<DS:SI>;
if (df == 0) {
  DI <– DI + 1;
  SI <– SI + 1 }
else {
  DI <– DI - 1;
  SI <– SI - 1 }
m<ES:DI> <– m<DS:SI>;
if (df == 0) {
  DI <– DI + 1;
  SI <– SI + 1 }
else {
  DI <– DI - 1;
  SI <– SI - 1 }
```

CMPS

SCAS

LODS

STOS

REP