CS2507 Computer Architecture

Lecture 3

The Instruction Set

The 8085 Processor

The 8085 processor is an 8-bit CPU. It has an 8-bit accumulator (ACC or A) with a 16-bit PC. Thus the maximum addressable main memory size is 64K. Other registers are:

BC	16-bit register B or 8-bit registers B and C	General purpose
DE	16-bit register D or 8-bit registers D and E	General purpose
HL	16-bit register H or 8-bit registers H and L	Data pointer
SP	16-bit address register	Stack pointer
Flags	8-bit flags register	Contains 5 condition flags

Instructions are variable-length and can occupy 1, 2 or 3 bytes.

Operand addressing modes available are implied, register, immediate, direct and register indirect (using the BC, DE and HL register pairs as 16-bit pointers to memory).

Conditions are indicated by setting 1-bit flags to show the result of an ALU operation. The flags can be tested using conditional jump, call or return instructions. There are 5 condition flags: Sign (s), Zero (z), Auxiliary Carry (ac), Parity (P) and Carry (cy). These flags are padded with 3 extra bits to form an 8-bit Flags register, so that s, z, ac, p and cy occupy bits 7, 6, 4, 2 and 0 of the Flags register. The Flags register is not accessible as an entity in its own right, but forms the low byte of the 16-bit Program Status Word (PSW). The high byte of the PSW is obtained from the A register. The PSW is accessible only via PUSH and POP stack 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.

Stack is located off-chip, in main memory and is accessed through the 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 and subtraction (including increment and decrement) of signed and unsigned 8-bit integers, although a few 16-bit operations are permitted. Comparisons are performed by internal subtraction with modification to allow for special cases. Logical AND, OR, NOT and XOR operations are provided, as are single-bit rotations of 8- and 9-bit quantities.

Machine control includes instructions to enable and disable interrupts, set and read an interrupt mask as well as the standard No-operation and Halt instructions.

Software interrupts allow for 8 levels of Supervisor Call (SVC).

Instructions by Class

Data Movement Group

MOV ra, rb	Move (Copy) contents of rb to ra	r8[a] <– r8[b]
MOV M, r	Move contents of r to memory	m<HL> <– r8
MOV r8, M	Move contents of memory to register	r8 <– m<HL>
MVI r8, imm8	Move immediate data to register	r8 <– imm8
MVI M, imm8	Move immediate data to memory	m<HL> <– imm8
LXI B, imm16	Move immed data to register pair BC	B <– imm16[8..15]; C <– imm16[0..7]
LXI D, imm16	Move immed data to register pair DE	D <– imm16[8..15]; E <– imm16[0..7]
LXI H, imm16	Move immed data to register pair HL	H <– imm16[8..15]; L <– imm16[0..7]
STAX B	Store indirect	m<BC> <– A

STAX D	Store indirect	m<DE> <– A
LDAX B	Load indirect	A <– m<BC>
LDAX D	Load indirect	A <– m<DE>
STA addr	Store A direct	m<addr> <– A
LDA addr	Load A direct	A <– m<addr>
SHLD addr	Store H & L direct	m<addr> <– L;
		m<addr + 1> <– H
LHLD addr	Load H & L direct	L <– m<addr>;
		H <– m<addr + 1>
XCHG	Exchange contents of DE and HL	D <–> H; E <–> L

Stack Operations

PUSH B	SP <– SP - 1; m<SP> <– B; SP <– SP - 1; m<SP> <– C
PUSH D	SP <– SP - 1; m<SP> <– D; SP <– SP - 1; m<SP> <– E
PUSH H	SP <– SP - 1; m<SP> <– H; SP <– SP - 1; m<SP> <– L
PUSH PSW	SP <– SP - 1; m<SP> <– Flags; SP <– SP - 1; m<SP> <– A
POP B	C <– m<SP>; SP <– SP + 1; B <– m<SP>; SP <– SP + 1
POP D	C <– m<SP>; SP <– SP + 1; B <– m<SP>; SP <– SP + 1
POP H	C <– m<SP>; SP <– SP + 1; B <– m<SP>; SP <– SP + 1
POP PSW	C <– m<SP>; SP <– SP + 1; B <– m<SP>; SP <– SP + 1
XTHL	L <–> m<SP>; H <–> m<SP+ 1>
SPHL	SP <– HL
LXI SP, imm16	SP <– imm16
INX SP	SP <– SP + 1
DCX SP	SP <– SP - 1

Branch Group: Jump, Call, Return

JMP addr	PC <– addr
JC addr	**if** cy **then** PC <– addr
JNC addr	**if** !cy **then** PC <– addr
JZ addr	**if** z **then** PC <– addr
JNZ addr	**if** !z **then** PC <– addr
JP addr	**if** !s **then** PC <– addr
JM addr	**if** s **then** PC <– addr
JPE addr	**if** p **then** PC <– addr
JPO addr	**if** !p **then** PC <– addr
PCHL	PC <– HL
CALL addr	SP <– SP - 1; m<SP> <– PC[8..15];
	SP <– SP - 1; m<SP> <– PC[0..7];
	PC <- addr
CC	**if** cy **then** {SP <– SP - 1; m<SP> <– PC[8..15];
	SP <– SP - 1; m<SP> <– PC[0..7];
	PC <- addr}
CNC	**if** !cy **then** {SP <– SP - 1; m<SP> <– PC[8..15];
	SP <– SP - 1; m<SP> <– PC[0..7];
	PC <- addr}
CZ	**if** z **then** {SP <– SP - 1; m<SP> <– PC[8..15];
	SP <– SP - 1; m<SP> <– PC[0..7];
	PC <- addr}
CNZ	**if** !z **then** {SP <– SP - 1; m<SP> <– PC[8..15];
	SP <– SP - 1; m<SP> <– PC[0..7];
	PC <- addr}

CP	**if** !s **then** {SP <– SP - 1; m<SP> <– PC[8..15];
	SP <– SP - 1; m<SP> <– PC[0..7];
	PC <- addr}
CM	**if** s **then** {SP <– SP - 1; m<SP> <– PC[8..15];
	SP <– SP - 1; m<SP> <– PC[0..7];
	PC <- addr}
CPE	**if** p **then** {SP <– SP - 1; m<SP> <– PC[8..15];
	SP <– SP - 1; m<SP> <– PC[0..7];
	PC <- addr}
CPO	**if** !p **then** {SP <– SP - 1; m<SP> <– PC[8..15];
	SP <– SP - 1; m<SP> <– PC[0..7];
	PC <- addr}
RET	PC[0..7]<– m<SP>; SP <– SP + 1;
	PC[8..15] <– m<SP>; SP <– SP + 1
RC	**if** cy **then** {PC[0..7]<– m<SP>; SP <– SP + 1;
	PC[8..15] <– m<SP>; SP <– SP + 1}
RNC	**if** !cy **then** {PC[0..7]<– m<SP>; SP <– SP + 1;
	PC[8..15] <– m<SP>; SP <– SP + 1}
RZ	**if** z **then** {PC[0..7]<– m<SP>; SP <– SP + 1;
	PC[8..15] <– m<SP>; SP <– SP + 1}
RNZ	**if** !z **then** {PC[0..7]<– m<SP>; SP <– SP + 1;
	PC[8..15] <– m<SP>; SP <– SP + 1}
RP	**if** !s **then** {PC[0..7]<– m<SP>; SP <– SP + 1;
	PC[8..15] <– m<SP>; SP <– SP + 1}
RM	**if** s **then** {PC[0..7]<– m<SP>; SP <– SP + 1;
	PC[8..15] <– m<SP>; SP <– SP + 1}

RPE	**if** p **then** {PC[0..7]<– m<SP>; SP <– SP + 1;
	PC[8..15] <– m<SP>; SP <– SP + 1}
RPO	**if** !p **then** {PC[0..7]<– m<SP>; SP <– SP + 1;
	PC[8..15] <– m<SP>; SP <– SP + 1}

Arithmetic Group

INR r8	r8 <– r8 + 1
DCR r8	r8 <– r8 - 1
INR M	m<HL> <– m<HL> + 1
DCR M	m<HL> <– m<HL> - 1
INX B	BC <– BC + 1
INX D	DE <– DE + 1
INX H	HL <– HL + 1
DCX B	BC <– BC - 1
DCX D	DE <– DE - 1
DCX H	HL <– HL - 1
ADD r8	A <– A + r8
ADC r8	A <– A + r8 + C
ADD M	A <– A + m<HL>
ADC M	A <– A + m<HL> + C
ADI imm8	A <– A + imm8
ACI imm8	A <– A + imm8 + C
DAD B	HL <– HL + BC
DAD D	HL <– HL + DE
DAD H	HL <– HL + HL
DAD SP	HL <– HL + SP
SUB r8	A <– A - r8

SBB r8	A <– A - r8 - C
SUB M	A <– A - m<HL>
SBB M	A <– A - m<HL> - C
SUI imm8	A <– A - imm8
SBI imm8	A <– A - imm8 - C

Logical Group

ANA r8	A <– A AND r8
XRA r8	A <– A XOR r8
ORA r8	A <– A OR r8
CMP r8	A - r8 (Flags affected; no operation result saved)
ANI imm8	A <– A AND imm8
XRI imm8	A <– A XOR imm8
ORI imm8	A <– A OR imm8
CPI imm8	A - imm8 (Flags affected; no operation result saved)
RLC	cy <– A7; A7 <– A6; A6 <– A5; A5 <– A4; A4 <– A3; A3 <– A2; A2 <– A1; A1 <– A0; A0 <– cy
RRC	cy <– A0; A0 <– A1; A1 <– A2; A2 <– A3; A3 <– A4; A4 <– A5; A5 <– A6; A6 <– A7; A7 <– cy;
RAL	tmp <–cy; cy <– A7; A7 <– A6; A6 <– A5; A5 <– A4; A4 <– A3; A3 <– A2; A2 <– A1; A1 <– A0; A0 <– tmp
RAR	tmp <– cy ; cy<– A0; A0 <– A1; A1 <– A2 ; A2 <– A3; A3 <– A4; A4 <– A5; A5 <– A6; A6 <– A7; A7 <– tmp;

Special ALU Group

CMA	A <– NOT A
STC	cy <– 1
CMC	cy <– NOT cy
DAA	**if** (A[0..3] > 9) **OR** (ac = 1) **then** {A <– A + 6};
	if (A[4..8] > 9) **OR** (cy = 1) **then** {A <– A + 60H}

Machine Control Group

EI	Enable interrupts **after** execution of **next** instruction
DI	Disable interrupts **after** execution of **this** instruction
NOP	Do nothing
HLT	Halt the instruction cycle.

Mask & Serial I/O Group

RIM	Read Interrupt Mask
	Following execution of RIM, the A register contains:
	A0, A1, A2: RST5.5, RST6.5, RST7.5 masks; 1 = masked
	A3: Interrupt Enable Flag; 1 = enabled
	A4, A5, A6: Pending Interrupts; 1 = Pending
	A7: Serial Input Data bit
SIM	Set Interrupt Mask
	Before execution of SIM, the A register must be set as follows:
	A0, A1, A2: RST5.5, RST6.5, RST7.5; 0 = enabled; 1 = masked
	A3: Mask Set Enable; 0 = Ignore A0 - A2; 1 = Apply A0-A2.
	A4: if A4 == 1, reset RST7.5 latch to OFF.
	A5: ignore
	A6: if A6 == 1, output A7 to Serial Output Data latch
	A7; ignore if A6 == 0

Restart (Software Interrupt) Group

RST code	SP <– SP - 1; m<SP> <– PC[8..15];
	SP <– SP - 1; m<SP> <– PC[0..7];
	PC <- code * 8

Note: The RST *code* operand is limited to a value between 0 and 7. Therefore, the address of the next instruction executed is one of the values 0, 8, 10H, 18H, 20H, 28H, 30H, 38H.

Power-on or RESET sets the PC to 0.

In the 8085, a signal on one of the inputs TRAP, RST5.5, RST6.5 or RST 7.5 causes execution of an internal RST instruction with the next executed instruction coming from addresses 24H, 2CH, 34H or 3CH respectively.