Recall the Internal Architecture of the 8088

- General Register
- Operand Register
- ALU
- Flags
- Instruction Queue
- Address Generation & Bus Control
- Instruction Pointer
- Segment Register
- Bus Interface Unit (BIU)

(from Ch 4)
The programming model (software) summarizes all the information needed for programming the microprocessor.

**Example: 8088 Programming Model**

<table>
<thead>
<tr>
<th>Segment Registers</th>
<th>4 Data Registers</th>
<th>2 Pointer Registers</th>
<th>2 Index Registers</th>
<th>1 Flag Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS</td>
<td>AX</td>
<td>SP</td>
<td>BX</td>
<td>SP</td>
</tr>
<tr>
<td>DS</td>
<td>BX</td>
<td>BP</td>
<td>BX</td>
<td>BP</td>
</tr>
<tr>
<td>SS</td>
<td>CX</td>
<td>SI</td>
<td>DX</td>
<td>SI</td>
</tr>
<tr>
<td>ES</td>
<td>DX</td>
<td>DI</td>
<td>CS</td>
<td>DI</td>
</tr>
<tr>
<td></td>
<td>CS</td>
<td>IP</td>
<td></td>
<td>IP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Flags</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>AH</td>
<td>AL</td>
<td></td>
<td></td>
<td>SP</td>
<td></td>
</tr>
<tr>
<td>BH</td>
<td>BL</td>
<td></td>
<td></td>
<td>BP</td>
<td></td>
</tr>
<tr>
<td>CH</td>
<td>CL</td>
<td></td>
<td></td>
<td>SI</td>
<td></td>
</tr>
<tr>
<td>DH</td>
<td>DL</td>
<td></td>
<td></td>
<td>DI</td>
<td></td>
</tr>
</tbody>
</table>

- Accumulator
- Base
- Count
- Data
- Stack pointer
- Base pointer
- Source index
- Destination index
- Instruction pointer
- Code segment
- Data segment
- Stack segment
- Extra segment
- Flags
Programming Model for Intel 80386

8-bit Names

16-bit Names

32-bit Names

Name

Accumulator

Base index

Count

Data

Stack pointer

Base pointer

Destination index

Source index

Instruction pointer

Flags

Code

Data

Extra

Extra

Stack

: Only available in 80386 or higher
Program Visible and Invisible Registers

- The programming model of a microprocessor contains a concise description of the internal registers and their functions.

- Only the program visible registers (i.e. directly accessible by applications) are included in the programming model.

- All the registers in the Intel 8088 are program visible.

- The 80286, 80386, 80486 and Pentium also have program invisible registers (not accessed by normal software, but may be used by system software to control the microprocessor in protected mode)
Register in the 8088/8086 and 80286 may be grouped into 4 sets:

(i) **General registers** (AX, BX, CX, DX)
(ii) **Pointer and Index registers** (SP, BP, SI, DI, IP)
(iii) **Flag Register** (FLAGS)
(iv) **Segment Registers** (CS, DS, SS, ES)

The 80386, 80486 and Pentium microprocessors have 32-bit registers, and the 16-bit registers of the 8088 form a subset:
Software model of the 8088 microprocessor

The segment registers in MPU store the initial address information of the corresponding memory segments.

Q: 16-bit -> 20-bit

SR: status (flag) register
General Data Registers

(AX, BX, CX, DX)

- AX, BX, CX, DX are 16-bit registers.
- The 16-bit registers in this set may be split into two.

<table>
<thead>
<tr>
<th></th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>AX</td>
<td>AH</td>
<td>AL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BX</td>
<td>BH</td>
<td>BL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CX</td>
<td>CH</td>
<td>CL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DX</td>
<td>DH</td>
<td>DL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Other than being used as a general register,
- **AX** (accumulator) stores the result of many arithmetic and logic instructions.
- **BX** (Base) stores the base (offset) address data in memory and the base address of a table of data referenced by the translate (XLAT) instruction.
- **CX** stores the **count** for certain instructions (eg. Counter in the LOOP instruction, the shift count for shift instructions).
- **DX** holds the most significant part of the result of a 16-bit multiplication, the most significant part of a dividend before division, or I/O port number for a variable I/O instruction.
This set of registers usually store **offset addresses** of memory. **IP** usually stores the offset address of the next instruction in memory. **SP**, **BP**, **DI** and **SI** usually store the offset address of data in memory.

**SP**, **BP**, **DI** and **SI** may also be used for general purposes.

**SP** (**Stack Pointer**) is used in the **PUSH** and **POP** instructions for operations on a LIFO (Last-In, First-Out) stack.

**BP** (**Base Pointer**) is often used in addressing an array of data in the stack memory.

**DI** (**Destination Index**) usually stores the indirect destination address of data from an instruction.

**SI** (**Source Index**) is used when indirectly addressing source data in certain string instructions.
# 8086 Flag Register Format

<table>
<thead>
<tr>
<th>BIT</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>OF</td>
<td>DF</td>
<td>IF</td>
<td>TF</td>
<td>SF</td>
<td>ZF</td>
<td>U</td>
<td>AF</td>
<td>U</td>
<td>PF</td>
<td>U</td>
<td>CF</td>
<td></td>
</tr>
</tbody>
</table>

- **U** = UNDEFINED
- **CARRY FLAG** - Set up carry out of MSB
- **Parity Flag** – Set if result has even parity
- ** Auxiliary Carry Flag** for BCD
- **Zero Flag** – Set if result = 0
- **Sign Flag** = MSB of result
- **Single Step Trap Flag**
- **Interrupt Enable Flag**
- **String Direction Flag**
- **Overflow Flag**
9 of the 16 bits in FLAGS can be set to one when certain events occur (i.e., they flag the occurrence of an event.) The other 7 bits are unused.

All bits are set to zero on power up.

Conditional jump instructions test the O, S, Z, P and C flags.

Instructions (pushF, popF, LAHF, SAHF) are available for transferring the contents of the FLAGS register to/from the stack or to/from the AH register. Other instructions are available for manipulating certain flag bits (e.g., STI, CLI, STD, CLD, STC, CLC, CMC).

For setting the flag T, see example 11-1 in Brey’s.
Flags in the FLAGS register are either condition or control flags.

**Condition flag** consist of:
- **C** (carry flag) - set to 1 when the result of an addition has a carry out of the most significant byte. Other instructions can also affect C (e.g. subtraction)
- **P** (parity flag) - set to 1 if the low order byte of the result contains an even number of ones; otherwise it is set to zero.
- **A** (auxiliary carry flag) - set to one if there is a carry out of bit 3 during an addition or a borrow by bit 3 during subtraction.
- **Z** (zero flag) - set to 1 if the result is zero; Z is otherwise zero.
- **S** (sign flag) - equal to the most significant bit of the result (i.e. set to 1 if the result is negative)
- **O** (overflow flag) - set if a result is out of range (e.g. when adding two positive numbers and the result appears negative)
Control Flags

Three bits (D, I, T) in the flags register control the operation of the microprocessor under the following circumstances:

- **D (direction flag)** - in certain string manipulation instructions, D determines whether the string is processed from the lowest address (D=0) or the highest address (D=1).
  - D=0: auto-increment
  - D=1: auto-decrement

- **I (interrupt flag)** - determines whether a maskable interrupt is recognized by the microprocessor. If I=1, a maskable interrupt is possible, otherwise the interrupt is ignored.

- **T (trap flag)** - if T=1, a trap (e.g., for single stepping through a program) is executed after every instruction.
Example of effect of an addition on the FLAGS register

(1) If the following addition is carried out,

\[
\begin{array}{c}
0010 \ 0011 \ 0100 \ 0101 \\
+ \ 0011 \ 0010 \ 0001 \ 1001 \\
\hline
0101 \ 0101 \ 0101 \ 1110
\end{array}
\]

the FLAGS register’s condition flags are set as follows:
- S (sign) = 0, Z (zero) = 0, P (parity) = 0, C (carry) = 0,
- A (aux carry) = 0, O (overflow) = 0

(2) If instead the following were performed,

\[
\begin{array}{c}
0101 \ 0100 \ 0011 \ 1001 \\
+ \ 0100 \ 0101 \ 0110 \ 1010 \\
\hline
1001 \ 1001 \ 1010 \ 0011
\end{array}
\]

the FLAGS register’s condition flags would be set as follows:
- S (sign) = 1, Z (zero) = 0, P (parity) = 1, C (carry) = 0,
- A (aux carry) = 1, O (overflow) = 1
## Segment Registers

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td><strong>CS</strong></td>
<td>CODE SEGMENT Register</td>
</tr>
<tr>
<td><strong>DS</strong></td>
<td>DATA SEGMENT Register</td>
</tr>
<tr>
<td><strong>ES</strong></td>
<td>EXTRA SEGMENT Register</td>
</tr>
<tr>
<td><strong>SS</strong></td>
<td>STACK SEGMENT Register</td>
</tr>
</tbody>
</table>

- **Segment registers** are 16-bit registers used in conjunction with the index and pointer registers to generate the physical 20-bit address.

- **Code segment** (CS) is the section of memory used to store the program instructions and procedures. CS stores the starting address of the program code. In the 8086 (and 80286) the code segment is limited to 64Kbytes in length (in 80386 the maximum length is 4Gbytes).

- **Data segment** (DS) contains data used by the program. DS stores the starting address of the data segment.

- **Extra segment** is another data segment which is used by some string instructions.

- **Stack segment** (SS) is the section in memory called stack which is used for storage of register contents and data.
Segmentation

- A segment in the 8088 system is a continuous section of memory of up to 64Kbytes in length.

- Since the segment address is shifted by 4 bits to form the 20-bit address, the start address of a segment can only occur at 16 data byte intervals. Valid start address 00000h, 00010h, 00020h, …

- Increasing the segment register value by 1 will increase the physical starting address by 16.

- The 64-Kbyte block defined by the start address of a segment may overlap with other segments or be completely in its own separate area of memory. -- Don’t try it unless you really know what you are doing.

- Segments allow packages of information (e.g. a data table, or a subroutine) to be kept separately - it is not necessary to fill all 64K of the segment and the programmer can make the segment of arbitrary size (up to 64K byte, in 16 byte increments).
Segments and Offsets

- The 20-bit address in memory is generated by adding a 16-bit offset to a 16-bit segment address. The segment address defines the start of a 64K-byte memory block within the 1M-byte address space, and the offset defines the exact memory location within that 64K-byte block.

- A 20-bit address is formed by shifting the 16-bit segment address by 4 bits and adding to the 16-bit offset.

```
Bit 15 11 7 3 0 Shift 4 bits and append 0
Segment

Bit 15 11 7 3 0 Offset

+ 20-bit physical address
```
Advantages of the segment + offset method include:

1. Program code can easily be reallocated in memory (useful for multi-tasking)
2. Most operations can be performed by changing only the 16 bit offset. The offset can be stored in 16-bit registers (20-bit registers are not necessary), allowing for easier interface to 8- and 16-bit wide memory.
The microprocessor has a set of rules that define the segment register and offset register combination used by certain addressing modes. However, the default can be overridden by using the segment override prefix:

- MOV CL, [BP] (move a byte from location SS:[BP] to CL)
- MOV CL, DS:[BP]

### Offset register

<table>
<thead>
<tr>
<th>Offset register</th>
<th>Default Segment register</th>
<th>Override Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP</td>
<td>CS</td>
<td>Never</td>
</tr>
<tr>
<td>SP</td>
<td>SS</td>
<td>Never</td>
</tr>
<tr>
<td>BP</td>
<td>SS</td>
<td>DS, ES or CS</td>
</tr>
<tr>
<td>SI, DI (not include strings)</td>
<td>DS</td>
<td>ES, SS or CS</td>
</tr>
<tr>
<td>DI for string instructions</td>
<td>ES</td>
<td>Never</td>
</tr>
</tbody>
</table>
One way that four 64-Kbyte segments might be positioned within the 1-Mbyte address space of an 8086

<table>
<thead>
<tr>
<th>PHYSICAL ADDRESS</th>
<th>MEMORY</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFFFFH</td>
<td>HIGHEST ADDRESS</td>
</tr>
<tr>
<td>7FFFFH</td>
<td>TOP OF EXTRA SEGMENT</td>
</tr>
<tr>
<td>70000H</td>
<td>EXTRA SEGMENT BASE</td>
</tr>
<tr>
<td>5FFFFH</td>
<td>ES = 7000H</td>
</tr>
<tr>
<td>50000H</td>
<td>TOP OF STACK SEGMENT</td>
</tr>
<tr>
<td>4489FH</td>
<td>STACK SEGMENT BASE</td>
</tr>
<tr>
<td>348A0H</td>
<td>SS = 5000H</td>
</tr>
<tr>
<td>2FFFFH</td>
<td>TOP OF CODE SEGMENT</td>
</tr>
<tr>
<td>20000H</td>
<td>CODE SEGMENT BASE</td>
</tr>
<tr>
<td></td>
<td>CS = 348AH</td>
</tr>
<tr>
<td></td>
<td>TOP OF DATA SEGMENT</td>
</tr>
<tr>
<td></td>
<td>BOTTOM OF DATA SEGMENT</td>
</tr>
</tbody>
</table>
Addition of IP to CS to produce the physical address of the code type

Diagram

Assume
CS=348AH
IP=4214H

Computation

(a)

(b)

4489FH  TOP OF CODE SEGMENT
38AB4H  CODE BYTE
348A0H  START OF CODE SEGMENT
CS=348AH  IP=4214H

PHYSICAL ADDRESS

MEMORY

HARDWIRED ZERO

3 4 8 A 0

4 2 1 4

3 8 A B 4
Addition of SS and SP to produce the physical address of the top of the stack

Assume
SS=5000H
SP=FFE0H

Computation

| SS   | 50000000 00000000 |
| SP   | FFFE0000 |

Adding SS and SP produces the physical address of the top of the stack.

Diagram

- PHYSICAL ADDRESS
- MEMORY
- 5FFFFH (TOP OF STACK SEGMENT)
- 5FFE0H (TOP OF STACK)
- SP=FFE0H
- 50000H (START OF STACK SEGMENT)
- SS=5000H

HARDWIRED ZERO
Addition of data segment register and effective address to produce the physical address of the data byte

Assume
DS=2000H

Computation

\[ \text{EA: effective address} \]
Number in specific microprocessors

Number with several bytes may be stored in 2 ways:

- **Low Order Byte First (LOBF)** architecture (also called *little endian*) - the least significant byte is put in the first (lowest) memory address. Examples of microprocessors which use this convention include Intel 8086/8088

- **High Order Byte First (HOBF)** architecture (also called *big endian*) - the most significant byte is put first. Example include the Motorola 680X0 microprocessors
Number in specific microprocessors

Big-endian CPU

<table>
<thead>
<tr>
<th>Reg</th>
<th>B_n-1</th>
<th>...</th>
<th>B2</th>
<th>B1</th>
<th>B0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSB</td>
<td>n-1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Memory

- B_{n-1} (MSB)
- B1
- B0 (LSB)

Operand

Big-endian (high-order-byte-first) architecture

Little-endian CPU

<table>
<thead>
<tr>
<th>Reg</th>
<th>B_{n-1}</th>
<th>...</th>
<th>B2</th>
<th>B1</th>
<th>B0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSB</td>
<td>n-1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Memory

- B0 (MSB)
- B1
- B2
- B_{n-1} (LSB)

Operand

Little-endian (low-order-byte-first) architecture
FAQ

- The overflow flag will be set if the resultant number is too big (e.g. when adding two positive number) or too small (e.g. when adding two negative numbers) to present.

Example:

```assembly
mov al,0a0h
mov ah,0b0h
add al,ah
```

The result is ax=0b050h with C=1, Z=0, S=0, O=1, P=1, A=0.