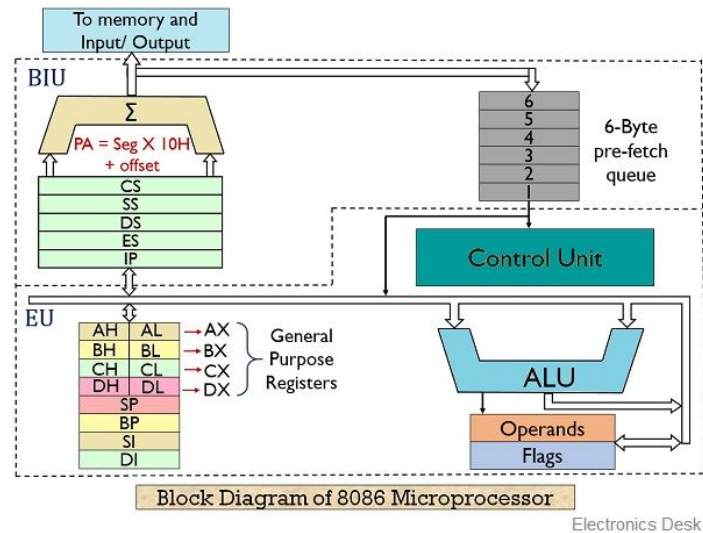


## 1

EE2003

## Intel 8086



## Execution Unit

- Flag register used to store some result about an operation. Example result of a comparison or overflow bit etc. Used by the ALU
- E(xtended)SP: Stack pointer
- EA/B/C/D: General registers. Do whatever you want
- ESI/ EDI: For string instructions. strlen() or memcpy()

The Extended registers **extend** the A/B/C/DX registers to 32 bit

## Bus Interface Unit

**CS, SS, DS** store the starting address of code, stack and data segments. The IP(instruction pointer) or the A,B,C,D registers are used to store the offset from the start address. Formula is given by:

$$\text{Address} = \text{Segment} \ll 4 + \text{offset}$$

BP and SP point to SS, IP points to CS, A/B/C/D point to DS (or ES, prof did not remember)

## x86 Ops

```

1  MOV EAX, EBX      ; EAX <- EBX, register direct
2  MOV EAX, [ECX]    ; Load from memory, register indirect
3  MOV [ECX], EAX    ; store to memory
4  MOV EAX, [EBX - N] ; register indirect with offset, requires the EDS as well
5  MOV EAX, 0x1602   ; immediate mode, no address calculation required
6  MOV EAX, EBX - ECX ; THIS IS WRONG
7
8
9  ; Load Effective Addr:
10 ; Essentially you can do math in the right operand.
11 ; Used in compilers for things like
12 ; int *p = &point[i].ycoord;
13 LEA EAX, [EBP+8*EAX+4]
14
15 ; ALU Ops
16 ADD EAX, EBX      ; EAX <- EAX + EBX
17 SUB EAX, DWORD PTR[EBX]
18 SUB EAX, WORD PTR[EBX]
19 SUB EAX, BYTE PTR[EBX]
20
21 ; Logical Ops
22 AND EAX, DWORD PTR [EBP + 4]

```

1. Write assembly to do the following:

$$EAX = x * y + a - b$$

$$EBX = (x \oplus y) | (a \wedge b)$$

```

1  ; first part
2  MOV EAX, X
3  MUL y
4  MOV EBX, a
5  SUB EBX, b
6  ADD EAX, EBX
7
8  ;second part
9  MOV EBX, x
10 XOR EBX, y
11 MOV ECX, a
12 AND ECX, b
13 OR EBX, ECX

```

2. Write a program to evaluate  $z = x * y$  using repeated addition

```
1 XOR EAX, EAX
2 MOV EBX, y
3 CMP EBX, 0
4 JZ done
5 add:
6 ADD EAX, x
7 DEC EBX
8 JNZ add
9
10 done:
11 NOP
```

3. Write a program to calculate the string length of a constant string

```
1 ; assuming the data is little endian
2 XOR EAX, EAX
3 loop:
4 CMP [EBX], 0
5 JZ done
6 INC EBX
7 INC EAX
8 JMP loop
9
10 done:
11 NOP
```

4. Write a program to swap two integers x and y

```
1 Swap(int *pX, int *pY){
2     __asm{
3         MOV EAX, pX
4         MOV EBX, pY
5
6         PUSH DWORD PTR [EAX]
7         PUSH DWORD PTR [EBX]
8         POP DWORD PTR [EAX]
9         POP DWORD PTR [EBX]
10    }
11 }
```

### 5. Look at these swaps and realize the differences

```

1 Swap(int *pX, int *pY){
2     __asm{
3         ; this thing is like gae
4         ; the most useless thing ever
5         PUSH pX
6         PUSH pY
7         POP pX
8         POP pY
9     }
10 }
11
12 Swap2(int x, int y){
13     __asm{
14         ; this thing is like gae
15         ; the most useless thing ever
16         PUSH x
17         PUSH y
18         POP x
19         POP y
20     }
21 }

```

## Compiling a C program

### Optimization

You can optimize your \*.c code by using the `\O1` flag if using MSVC. It removes dead code and can optimize to reduce instructions. However one should be very careful and not trust the compiler to do their job perfectly all the time.

```

1 int __stdcall Fn(int x, int y){
2     /*
3     * Prologue
4     * PUSH EBP          ; store the previous base
5     * MOV EBP, ESP      ; set the top as the new base
6     * SUB ESP, 0x40 ; leave some space out for local variables(?)
7     */
8     // MOV [EBP-4], 0
9     int z = 0;
10    z = x+y;
11    /* MOV EAX, 8[EBP]
12    * MOV EBX, 12[EBP]
13    * ADD EAX, EBX
14    */
15    MOV [EBP-4], EAX
16    return z;

```

```

17      /* Epilogue
18      *  ADD ESP, 0x40
19      *  POP EBP
20      *  RET 8          ; doing __stdcall forces compiler to do the
21      *                  ; stack cleanup
22      */
23  }
24  int main(){
25      int z=0;
26      z = Fn(2,4);
27  }

```

### Optimization

`__stdcall` forces the callee to do the stack cleanup. Normal behaviour is `RET 0` in the callee function and `ADD ESP, N` in the caller function. Directly doing `RET N` is faster (by one clock cycle) but you cannot do variable arguments.

### Pushing parameters

The parameters passed to a function are pushed from right to left. For a call like:

$$f(a, b, c \dots z)$$

the stack will look like:

a	b	c	...	z
ESP				EBP

### Optimization

`__fastcall` is a calling convention (specific to MSVC) that asks the compiler to use registers to store parameters (wherever possible). This will speed up memory access.