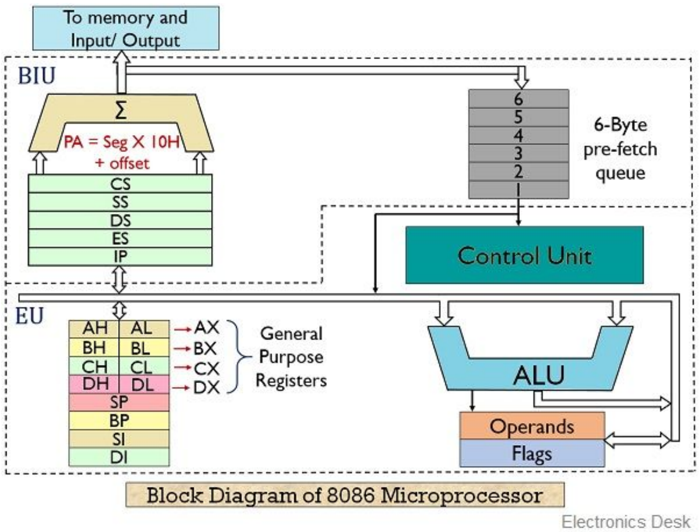
Name: Krutarth Patel

1

EE2003

Intel 8086



Execution Unit

- Flag register used to store some result about an operation. Example result of a comparision 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:

$$Address = Segment << 4 + offset$$

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

```
MOV EAX, EBX       ; EAX <- EBX, register direct
MOV EAX, [ECX]     ; Load from memory, register indirect
MOV [ECX], EAX     ; store to memory
MOV EAX, [EBX - N] ; register indirect with offset, requires the EDS as well
MOV EAX, 0x1602    ; immediate mode, no address calculation required
MOV EAX, EBX - ECX ; THIS IS WRONG


; Load Effective Addr:
; Essentially you can do math in the right operand.
; Used in compilers for things like
; int *p = &point[i].ycoord;
LEA EAX, [EBP+8*EAX+4]

; ALU Ops
ADD EAX, EBX        ; EAX <- EAX + EBX
SUB EAX, DWORD PTR[EBX]
SUB EAX, WORD PTR[EBX]
SUB EAX, BYTE PTR[EBX]

; Logical Ops
AND EAX, DWORD PTR [EBP + 4]
```

1. Write assembly to do the following:
$EAX = x * y + a - b$
$EBX = (x \oplus y)|(a \wedge b)$

```
; first part
MOV EAX, X
MUL y
MOV EBX, a
SUB EBX, b
ADD EAX, EBX

;second part
MOV EBX, x
XOR EBX, y
MOV ECX, a
AND ECX, b
OR EBX, ECX
```

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

```
XOR EAX, EAX
MOV EBX, y
CMP EBX, 0
JZ done
add:
ADD EAX, x
DEC EBX
JNZ add

done:
NOP
```

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

```
; assuming the data is little endian
XOR EAX, EAX
loop:
CMP [EBX], 0
JZ done
INC EBX
INC EAX
JMP loop

done:
NOP
```

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

```
Swap(int *pX, int *pY){
        __asm{
                MOV EAX, pX
                MOV EBX, pY

                PUSH DWORD PTR [EAX]
                PUSH DWORD PTR [EBX]
                POP DWORD PTR [EAX]
                POP DWORD PTR [EBX]
        }
}
```

5. Look at these swaps and realize the differences

```
Swap(int *pX, int *pY){
        __asm{
                ; this thing is like gae
                ; the most useless thing ever
                PUSH pX
                PUSH pY
                POP pX
                POP pY
        }
}

Swap2(int x, int y){
        __asm{
                ; this thing is like gae
                ; the most useless thing ever
                PUSH x
                PUSH y
                POP x
                POP y
        }
}

Swap3(int *pX, int *pY){
        __asm{
                ; this thing is not gae
                MOV EAX, pX
                MOV EBX, pY
                PUSH DWORD PTR [EAX]
                PUSH DWORD PTR [EBX]
                POP DWORD PTR [EAX]
                POP DWORD PTR [EBX]
        }
}
```

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 intstructions. However one should be very careful and not trust the compiler to do their job perfectly all the time.

```
int __stdcall Fn(int x, int y){
        /*
        * Prologue
        * PUSH EBP          ; store the previous base
```

```
        * MOV EBP, ESP        ; set the top as the new base
        * SUB ESP, 0x40 ; leave some space out for local variables(?)
        */
        // MOV [EBP-4], 0
        int z = 0;
        z = x+y;
        /* MOV EAX, 8[EBP]
        *  MOV EBX, 12[EBP]
        *  ADD EAX, EBX
        */
        MOV [EBP-4], EAX
        return z;
        /* Epilogue
        *  ADD ESP, 0x40
        *  POP EBP
        *  RET 8        ; doing __stdcall forces compiler to do the
        *                ; stack cleanup
        */
}
int main(){
        int z=0;
        z = Fn(2,4);
}
```

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:

$$f(a, b, c...z)$$

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.

Recursion

```c
int fact(int n)
{
	int ans = 1;
	if(0 != n){
		ans = n*fact(n-1);
	}
	return ans;
}

int main()
{
	int ans = 0;
	ans = fact(3);
}
```

This gives the assembly output as:

```
	; Prologue
	PUSH EBP
	MOV EBP, ESP
	PUSH ECX

	MOV EAX, 1
	MOV EBX, [EBP+8]
	MOV [EBP-4], EAX
	CMP EBX, 0
	JZ DONE
	DEC EBX
	PUSH EBX
	CALL fact
	ADD ESP, 4
	MUL [EBP+8]

DONE:
	; Epilogue
	ADD ESP, 4
	POP EBP
	RET 0
```

The stack will look like(NOTE it grows downwards):

ans-main
3
ret-addr-main
EBP-main
ans-fact1
2
ret-addr1(0xC200)
EBP-fact1
ans-fact2
1
ret-addr2(0xC200)
EBP-fact2
ans-fact3
0
ret-addr3(0xC200)