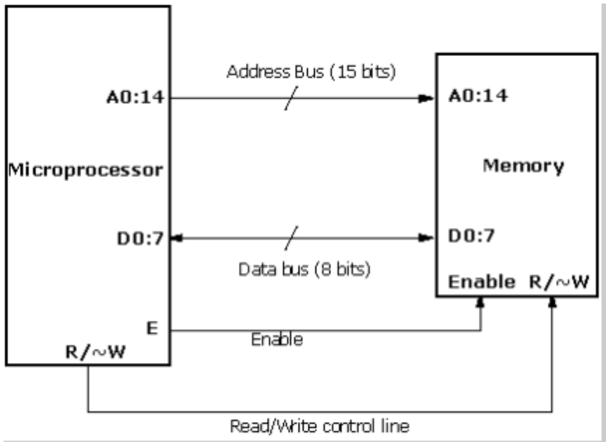
# C Programming and Assembly

August 27, 2022

# 1 Basic overview of $\mu$P

The main purpose of $\mu$P is to execute a program store in a memory.




**Address Bus:** logically address $2^N$ addresses where $N$ is the no. of address bits.
**Control Signal:** RD(Read), WR(Write)

## 1.1 Logical Memory Map

Consider $ADDR[0:N-1]$, we have $2^N$ reference location and $DATA[0:k-1]$, $k$ bits of information in each locations. e,g $0, 1, \dots 2^N - 1$ location and each with $k = 8$ bits of data

## 1.2 Instruction Cycle

**1. Fetch(F)**
**2. Decode(D)**
**3. Execute(E)**
Following cycle: $FI_1DI_1EI_1, FI_2DI_2EI_2 \dots FI_mDI_mEI_m$, where $I$ stands for Instructions
**For performing FDE what do we need ?**
For **FETCH:** fetching instructions from memory

- Need something to store(Instructions)
- Instructions Pointer: *always points to next instructions in the memory*

For **EXECUTE** we have following general purpose registers:
$a)\frac{E|\frac{AH(8bit)|AL(8bit)}{AX(16bit)}}{EAX(32bit)}$ $b)\frac{E|\frac{BH(8bit)|BL(8bit)}{BX(16bit)}}{EBX(32bit)}$ $c)\frac{E|\frac{CH(8bit)|CL(8bit)}{CX(16bit)}}{ECX(32bit)}$ $d)\frac{E|\frac{DH(8bit)|DL(8bit)}{DX(16bit)}}{EDX(32bit)}$
For array/string manipulation at the Hardware level:
1. Source Index(SI) Register: $\frac{E|SI(16bit)}{ESI(32bit)}$
2. Destination Index(DI) Register: $\frac{E|DI(16bit)}{EDI(32bit)}$

## 1.3 Partitioning of Memory

Segmentation of Code, Data, Function in memory

MEMORY $\boxed{\boxed{\text{CODE}}\ \boxed{\text{DATA}}\ \boxed{\text{STACK}}\ \boxed{\text{EXTRA SEGMENT}}}$

Segment register are associated with:

- CODE: $E|CS$
- DATA: $E|DS$
- STACK: $E|SS$
- EXTRA SEGMENT: $E|ES$

## 1.4 x86 Instructions

### 1.4.1 Data Transfer

1. $MOV[DEST][SRC]$

Register direct addressing: $MOV AX, BX$ $(AX \leftarrow BX)$
Immediate addressing: $MOV AX, 0 \times 40$ $(AX \leftarrow 0 \times 40)$
Direct addressing: $MOV\ 0 \times 1320, AX$ $(0 \times 1320 \leftarrow AX)$
Register indirect addressing: $MOV AX, [BX]$ $(AX \leftarrow [BX])$ content of BX is moved to AX
**WORD-16 bit & DWORD-32 bit**

### 1.4.2 ALU Instruction

$ADD\ AX, BX$ $(AX \leftarrow AX + BX)$
$SUB\ AX, BX$ $(AX \leftarrow AX - BX)$
$MUL\ [REG]$ : for multiplication it takes only one register the other is assume to be the prior register.

Listing 1: MUL example

```
MOV AX, 0x4500 ;AX <- 0x4500
MUL BX         ; AX <- AX*BX
```

$INC\ AL|AX|EAX$ $(AL \leftarrow AL + 1)$
$DEC\ AL|AX|EAX$ $(AL \leftarrow AL - 1)$
$CMP\ AX, BX$ $(AX - BX)$ the value of this is evaluated using FLAG Register
**Clearing Register** $XOR\ AX, BX$ $(AX \leftarrow 0)$

### 1.4.3 Stack Operation

STACK POINTER (ESP) - Last In First Out (LIFO) Operation
1. $PUSH\ AL|AX|EAX$ $[ESP \leftarrow ESP - 1/2/4]$
2. $POP\ AL|AX|EAX$ $[ESP \leftarrow ESP + 1/2/4]$
BASE POINTER (EBP) - Random access operation on stack

## 1.5 Call and Return Instructions

### 1.5.1 Subroutine (Function in C)

CALL: two way branching goes to subroutine then comes back to where it left.
$EIP \leftarrow EIP + N$ where, $N$ is the address of next instructions to be executed.

Listing 2: subroutine example

```
LOC_FUN:    ADD EBX, 0x0002;
            ADD EAX, 0x0003;
            SUB ECX, 0x0004;
            RET
MAIN:       XOR EBX, EBX;
            XOR EAX, EAX;
            XOR ECX, ECX;
            CALL LOC_FUN;
            XOR EBX, EBX;
            XOR EAX, EAX;
            XOR ECX, ECX;
            CALL LOC_FUN;
```

When CALL LOC_FUN is executed PUSH EIP is performed on stack and EIP $\leftarrow$ LOC_FUN, RET simply pops the top of stack into EIP.

## 1.6 Inline Assembly

Interoperation between C ans Assembly. Consider the following C code

Listing 3: Inline Example

```
void main() {
    int x=2;
    x = x + 2;
    printf("%d", x);
}
```

Say, we want to speed up $x = x + 2$ then we write the assembly version of those segment.

Listing 4: Inline Example

```
void main() {
    int x=2;
    //x = x + 2;
    __asm{
        MOV EAX, x      ; EAX <- x
        ADD EAX, 0x002  ; EAX <- EAX+2
        MOV x, EAX      ; x <- EAX
    }
    printf("%d", x);  // 4
}
```

### 1.6.1 Commonly used Data Types in C

- BYTE OF DATA: char
- WORD OF DATA: short int or int
- DWORD OF DATA: long int or int

Listing 5: repeatative addition

```
main() {
    // z=x*y
    short int x=2, y=3;
    int z=0;
    __asm{
        XOR EAX, EAX // EAX <- 0
        MOV ECX, y
MULT:   ADD EAX, x
        DEC ECX     // ECX <- ECX-1
        JNZ MULT // as long as ECX != 0 repeat adding
        MOV z, EAX // z=x*y

    }
}
```

Listing 6: Optimised vs unoptimised

```
int a,b,c,d;
int x=10,y=5;
a=x+y;
b=a-y;
c=b*y;
d=c/y
//---------- Un Optimised ---------//
mov x, 0x000a
mov y, 0x0005

mov eax,x
add eax,y
mov a,eax

mov ebx,a
sub ebx,y
mov b,ebx

mov eax,b
imul y
mov c,eax

mov ebx,c
idiv y
mov d,ebx

//------------ Optimised-------//
mov eax,x
add eax,y  // eax <- x+y
mov a,eax
sub ebx,y  // eax <- a-y
mov b,ebx
imul y     // eax <- b*y
mov c,eax
idiv y     // eax <- c/y
mov d,ebx
```

## 1.7 Pointer

Listing 7: pointer arithmetic

```
main() {
    char *pa=0; // mov pa,0x0000
    char *pb=0; // mov pb,0x0000
    /*
    mov eax,pa
    add eax,0x0001
    mov pa,eax
    */
    pa++;       // pa=pa+1
    /*
    mov eax,pb
    add eax,0x0004
    mov pb,eax
    */
    pb++;       // pb=pb+4
}
```

Listing 8: character pointer

```
main () {
    char *s = "This is all nonsense";
    int i=0;
    for(i;s[i]!='\0';i++){}
}
//-------------asm optimised----------//
int i=0;
int cnt=0;
__asm {
    mov ecx,0
    mov ebx,s
cm :cmp byte ptr [ebx],0x00
    jz done
    inc ecx
    in ebx
    jmp cm
done: mov cnt, ecx
}
```