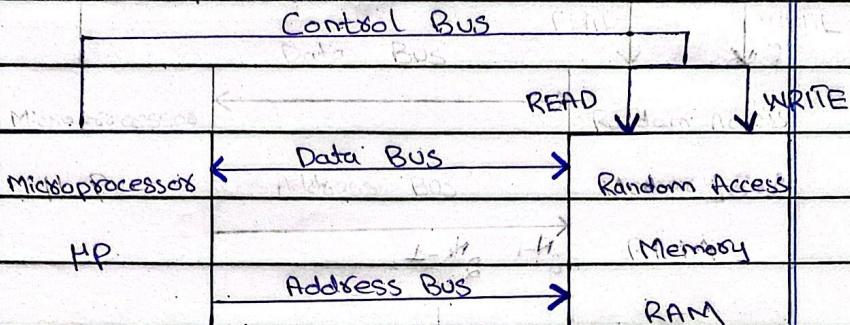


1.1

Computer System Design



Data Bus could be 8, 16, 32 or 64 bits. This bus is also bi-directional.

Address Bus could address 2^n , where n is number of address bits then 2^n gives total number of logical memory locations it can access.

1.2

Memory Operations

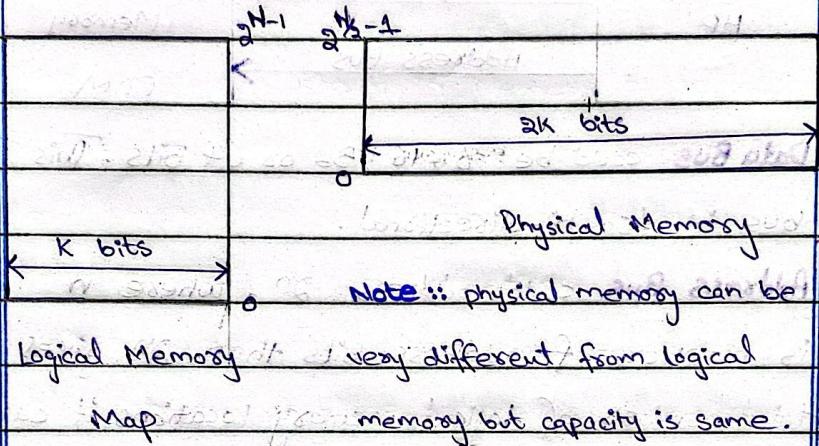
READ if you give high read control signal for a while with some address present in address bus then it will read data and place out on Data Bus towards processor.

WRITE if you give high on write control signal

than data present on Data Bus will be written on the location which address is in address Bus.

1.3

Logical Memory Maps



Address can be of N bits. Range [0 - N-1]

This address could refer to 2^N locations.

In each address you could write K bits of data. So Data Bus size will be [0 - K-1]

Note: This does not mean that physical address should be of K bits it could be 2^k bits, but in that case physical memory have half locations.

As programmes you need not to concern about that physical memory but only logical memory for N bits address only need K bits. otherwise you will read garbage or write onto other saved data.

2.1 Processing

Program It is a passive entity that contains the set of code required to perform a certain task.

Process is an active instance of the program which is loaded into memory and executed by processor.

How does CPU turns a

program into a Process?

Fetch \rightarrow Decode \rightarrow Execute

Load instruction
from memory

to CPU

Break down
to Machine

Code

Run the

Machine

Code

old up F₁D₁E₁ F₂D₂E₂ F₃D₃E₃ ...

New up F₁D₁E₁

F₂D₂E₂

F₃D₃E₃

Memory

Program
P₁

Here program is stored in Memory a₁ to a_m.

Suppose m instruction between addresses a₁ to a_m. The processor will fetch F, Decode D and execute E each instructions.

Modsen microprocessors use Pipeline where next fetch can be done as previous decodes.

in staircase manner.

2.2 Registers

Data Registers

Pointers/Index

Segments

Ax Accumulator IP Instruction CS code segment

Bx Base SI Source Index DS Data segment

Cx Counter DI Destination Index SS Stack segment

Dx Data SP Stack Pointer ES Extra segment

Flags Zero, Carry, sign and overflow.

Add E before each for extended 32 bits.



16 bit AX

EAX 32 bits

Note You can not access upper 16 bits of EAX as register alone.

SP can only perform last In first Out operations so we need another register to do random access that is BP (Base Pointer).

To do assay Manipulation of char array / string
we need **SI** and **DI**.

How they are used in processing?

Fetch	Execute									
IP Instruction Pointer	↓ operation code									
traces the line of code being executed and increases it by value N to fetch	<table border="1"> <tr> <td>Data Registered</td> <td>A</td> <td></td> </tr> <tr> <td>L</td> <td></td> <td>→</td> </tr> <tr> <td>U</td> <td></td> <td>Data Registered</td> </tr> </table>	Data Registered	A		L		→	U		Data Registered
Data Registered	A									
L		→								
U		Data Registered								

Note : Here N is operation output goes in diff code size . It can byte , word registers .
or Dword etc.

	→ CS	
.code	→ DS	{ code Memory block.
.data	→ SS	
.stack	→ ES	
.extra		What is complete addressing to memory?

Memory

Code [CS : IP] is complete address to location of memory block or instruction in code segment.

Data [DS : BX, DI, SI] Address of Data

Stack [SS : SP, BP] Address of stack

Extra [PS : BX, PI, SI] Address of destination data

for string operations.

Note :: Flag are used for branching and looping. These flags are set when some operation take place in Arithmetic Logic Unit and results are stored respective general purpose

Registers

To be very honest a computer is a dumb machine which could only perform three operations

LOAD

ADD

STORE

(LD)

Instruction set Intel 8086

Mnemonic \rightarrow Syntax/Type Example

MOV reg, reg or Reg Direct \rightarrow mov ax, bx

MOV reg, mem Immediate \rightarrow mov ax, 0x40

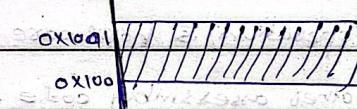
MOV mem, reg Direct \rightarrow mov 0x1320, ax

MOV reg, [addr]

MOV BX, WORD PTR [CX] \rightarrow 0x100

0x100

[DS:CX]



The whole shaded block will be moved to cx.

MOV BX, WORD PTR [CX+4] Reg Indirect + offset

MOVSX copy sign data low to high \rightarrow mov ecx, -16
mov sx ax, ecx

MOVZX copy unsigned data \rightarrow movzx ax, ecx

LAHF Load flags

Registers

SAHF Store flags

XCHG exchange used for swapping

INC	reg/mem	inc ax.
DEC	reg/mem	dec bx
Neg	reg/mem	neg cx
add	all	add ax, bx Ans in ax
Sub	all	sub ax, bx Ans in cx

i) Note: A high-level language like C could generate more than 1 types of code for same programs in Assembly but in assembly to machine there is one to one mapping.

ii) **Mov BX, WORD PTR [CX+4]**

Increases address inside cx by 4 and thus operation is hardware supported, not converted to other assembly code.

iii) Add Ax, [Bx] This will add the location pointed by bx to value in ax.

iv) **Mov Ax, Bx** where Bx is 0x0000 then zero flag will be set to 1.

v) **Clear Registers** **XOR Ax, Ax**
Ax will be zero and zero flag is set.

vi) **MUL** Multiplication where one register is set to AX as input and others can be of your choice and outputs in BX (lower bits) and DX (high bits).

IMUL is used for sign multiplication

vii) **DIV** In Division dividend will be placed in AX and BX (lower bits in AX and upper in DX).

Place divisor in CX. The output will be in AX and DX. ($AX \rightarrow \text{Quotient}$) ($DX \rightarrow \text{Remainder}$)

IDIV for signed division.

viii) **Division overflow** if a division does not produce a quotient that will not fit into the destination, a division overflow.

ix) **Compare** can be used for searching like `find()`.
lets assume find pattern in string array / memory.

`Mov AL, 0x21` // loading AL with search

`Mov CX, 0x100` // 256 count in CX

`Mov DI, Address_Data`

`Scasb` // compare AL to [DI]

`Cld` // clear Direction flag Auto inc

REPNE

REPNE Repeat until not equal or CX=0

D_i →

0x47
0x91
0x45

If first it will compare A1 value with value on address [D_i]. Then decrease 1 byte then compare. On index 2 found.

So value of CX will be 255 as result / Data found after 1 iteration.

⇒ CX is set to a number so that loop doesn't go infinite.

⇒ **CMPSB** can be used to compare two memory blocks of arrays in DS or ES with REP as above

⇒ **MOSW B/W/D** will copy all memory block from source to destination address. Note here no flag.

(CX) will be affected so use **REP**. (Non Arithmetic OP)

A stack pointer is used to deal Random Access Memory as stack. $\therefore \text{ESP} \rightarrow \text{RAM}$

Base of ESP comes from ESS. [ESS: ESP]

Complete Stack Address

Push A push operation decreases the stack pointer by appropriate amount according size of operation and copies a value into the location in the stack referred by stack pointer (ESP)

Pop does the opposite and return value on top of stack

ESP →	00000006	00000006	00000006
	000000AS	000000AS	000000AS
	ESP → 0000001	0000001	0000001

Push → Pop

Note:: Pop [EBX] here data from stack segment pointed by ESP will be copied to Data segment pointed by EBX. So it is very important to specify how many bytes to read.

Pop WORD PTR [EBX] \therefore Reads 16 bits

Pop DWORD PTR [EBX]

EBP can be used for random access of stack.

Mov Ax, WORD PTR [EBP - 4]

[E_{SS}: EBP] \hookrightarrow Part of Extended
stack segment
complete address

3.3

Sub Routines

A procedure is a named block of code defined with in PROC and ENPP directives.

CALL instruction loads function add to EIP before that push current add to stack pointed by ESP.

when return is executed ESP address is pop to EIP again for main/ to execute.

* This branching helps to reuse code without rewriting it.

Note :: All directive or mnemonic are hardware supported and they are not broken mechanically simplified they are executed ^{on} hardware level.

4.1 C to Assembly (MSVC x86)

C **Assembly**

--- C:\Windows\system32\cmd.exe ---

```
#include <stdio.h>

int main() {
    printf("Hello World");
    return 0;
}
```

```
#include <stdio.h>

int main() {
    00311870 push    ebp
    00311871 mov     ebp,esp
    00311873 sub     esp,0C0h
    00311879 push    ebx
    0031187A push    esi
    0031187B push    edi
    0031187C mov     edi,ebp
    0031187E xor     ecx,ecx
    00311880 mov     eax,0CCCCCCCCCh
    00311885 rep stos  dword ptr es:[edi]
    00311887 mov     ecx,offset
    _43D8056B_CandASM01@cpp (031C008h)      call
    0031188C
    @_CheckForDebuggerJustMyCode@4 (031132Ah)      call

    printf("Hello World");
    00311891 push    offset string "Hello World" (0317B30h)
    00311896 call    _printf (03110D2h)
    0031189B add    esp,4

    return 0;
    0031189E xor     eax,eax
}

003118A0 pop    edi
003118A1 pop    esi
003118A2 pop    ebx
003118A3 add    esp,0C0h
003118A9 cmp     ebp,esp
003118AB call    __RTC_CheckEsp (031124Eh)
003118B0 mov     esp,ebp
003118B2 pop    ebp
003118B3 ret
```

--- No source file---

When you look at a simple "Hello, World!" program, you'll notice that it translates into a large amount of assembly code. Additionally, it's important to note that even after the main function, the code continues to run and includes a `ret` statement in `.asm`.

After the C program ends, the operating system typically manages any remaining cleanup tasks, deallocating resources used by the program and reclaiming memory. The specific operations shown in the assembly code snippet ensure that the function cleans up its own stack frame and returns control properly to the calling function or the operating system.

```
__asm {
    mov eax, a ; load a
    mov ebx, b ; load b
    xor ecx, ecx ; clear ecx
```

```
loop_start:
    test ebx, ebx ; check ebx
    jz loop_end ; if zero, end loop
    add ecx, eax ; accumulate result
    dec ebx ; decrement ebx
    jmp loop_start ; repeat loop
```

```
loop_end:
    mov result, ecx ; store result
}
```

```
std::cout << a << " * " << b << " = " << result <<
std::endl;
```

```
return 0;
}
```

```
004124E9 push    ebx
004124EA push    esi
004124EB push    edi
004124EC lea     edi,[ebp-2Ch]
004124EF mov     ecx,0Bh
004124F4 mov     eax,0CCCCCCCCCh
004124F9 rep stos dword ptr es:[edi]
004124FB mov     eax,dword ptr [_security_cookie
(041C004h)]
00412500 xor     eax,ebp
00412502 mov     dword ptr [ebp-4],eax
```

```
int a = 5; int b = 3;
00412505 mov     dword ptr [a],5
0041250C mov     dword ptr [b],3
int result = 0;
00412513 mov     dword ptr [result],0
```

```
__asm {
    mov eax, a; load a
0041251A mov     eax,dword ptr [a]
    mov ebx, b; load b
0041251D mov     ebx,dword ptr [b]
    xor ecx, ecx; clear ecx
00412520 xor     ecx,ecx
```

```
loop_start:
    test ebx, ebx; check ebx
00412522 test    ebx,ebx
    jz loop_end; if zero, end loop
00412524 je      loop_end (041252Bh)
    add ecx, eax; accumulate result
00412526 add     ecx,eax
    dec ebx; decrement ebx
00412528 dec     ebx
    jmp loop_start; repeat loop
00412529 jmp     main+42h (0412522h)
```

```
loop_end:
    mov result, ecx; store result
0041252B mov     dword ptr [result],ecx
}
```

```
std::cout << a << " * " << b << " = " << result << std::endl;
0041252E mov     esi,esp
00412530 push    offset std::endl<char,std::char_traits<char> > (041103Ch)
00412535 mov     edi,esp
00412537 mov     eax,dword ptr [result]
0041253A push    eax
0041253B push    offset string " = " (0419B30h)
00412540 mov     ebx,esp
00412542 mov     ecx,dword ptr [b]
00412545 push    ecx
00412546 push    offset string " *" (0419B34h)
0041254B mov     eax,esp
0041254D mov     edx,dword ptr [a]
00412550 push    edx
00412551 mov     ecx,dword ptr [_imp_std::cout
(041D0D8h)]
00412557 mov     dword ptr [ebp-0ECh],eax
0041255D call    dword ptr [_imp_std::basic_ostream<char,std::char_traits<char>>::operator<< (041D0A0h)]
```

```

00412563 mov     ecx,dword ptr [ebp-0ECh]
00412569 cmp     ecx,esp
0041256B call    __RTC_CheckEsp (04112A3h)
00412570 push   eax
00412571 call    std::operator<<<std::char_traits<char>
                  >(04111B8h)
00412576 add     esp,8
00412579 mov     ecx,ecx
0041257B call    dword ptr
                  [__imp_std::basic_ostream<char,std::char_traits<char>
                  >::operator<< (041D0A0h)]
00412581 cmp     ebx,esp
00412583 call    __RTC_CheckEsp (04112A3h)
00412588 push   eax
00412589 call    std::operator<<<std::char_traits<char>
                  >(04111B8h)
0041258E add     esp,8
00412591 mov     ecx,ecx
00412593 call    dword ptr
                  [__imp_std::basic_ostream<char,std::char_traits<char>
                  >::operator<< (041D0A0h)]
00412599 cmp     edi,esp
0041259B call    __RTC_CheckEsp (04112A3h)
004125A0 mov     ecx,ecx
004125A2 call    dword ptr
                  [__imp_std::basic_ostream<char,std::char_traits<char>
                  >::operator<< (041D0A4h)]
004125A8 cmp     esi,esp
004125AA call    __RTC_CheckEsp (04112A3h)
      return 0;
004125AF xor     eax,ecx
}
}

004125B1 pop    edi
004125B2 pop    esi
004125B3 pop    ebx
004125B4 mov     ecx,dword ptr [ebp-4]
004125B7 xor     ecx,ecx
004125B9 call    @__security_check_cookie@4
                  (041118Bh)
004125BE add     esp,0ECh
004125C4 cmp     ebp,esp
004125C6 call    __RTC_CheckEsp (04112A3h)
004125CB mov     esp,ebp
004125CD pop    ebp
004125CE ret

```

As you can observe, inline assembly is slightly modified to ensure compatibility, such as using dword ptr when converting int to 32-bit. The compiler does not alter its operations when generating assembly for the entire C program.

*However, it's important to note that this implementation of **MUL** is significantly slower than the original **MUL** in assembly because the latter is hardware-implemented, whereas this is a software approach.*

4.2 Pointer Arithmetic (MSVC x86)

```
#include <stdio.h>

int main() {
    char* a = 0;
    00651795 mov     dword ptr [a],0
    int* b = 0;
    0065179C mov     dword ptr [b],0
    char* a = 0;
```

```

int* b = 0;

a++;
b++;

return 0;
}

    a++;
006517A3 mov     eax,dword ptr [a]
006517A6 add     eax,1
006517A9 mov     dword ptr [a],eax
    b++;
006517AC mov     eax,dword ptr [b]
006517AF add     eax,4
006517B2 mov     dword ptr [b],eax

    return 0;
006517B5 xor     eax,eax
...

```

char occupies 1 byte in memory, so each subsequent char is placed immediately after the previous one. On the other hand, an int occupies 4 bytes, so each subsequent int is placed 4 bytes after the previous one in consecutive memory locations.

```

#include <stdio.h>

int main() {

const char* testString = "Hello, world!";
int len;

__asm {
    mov esi, testString
    xor ecx, ecx

    loop_start:
        cmp byte ptr[esi + ecx], 0
        je loop_end
        inc ecx
        jmp loop_start

    loop_end:
        mov len, ecx
}

printf("The length of the string is: %d\n", len);

return 0;
}

```

```

...
    __asm {
        mov esi, testString
008A189C mov     esi,dword ptr [testString]
        xor ecx, ecx
008A189F xor     ecx,ecx

        loop_start:
            cmp byte ptr[esi + ecx], 0
008A18A1 cmp     byte ptr [esi+ecx],0
            je loop_end
008A18A5 je     loop_end (08A18AAh)
            inc ecx
008A18A7 inc     ecx
            jmp loop_start
008A18A8 jmp     main+31h (08A18A1h)

        loop_end:
            mov len, ecx
008A18AA mov     dword ptr [len],ecx
    }
...

```

```

#include <stdio.h>

int main() {

const char* testString = "Hello, world!";
int length = 0;

while (testString[length] != '\0') {
    length++;
}

printf("The length of the string is: %d\n", length);

return 0;
}

```

```

...
    const char* testString = "Hello, world!";
00541895 mov     dword ptr [testString],offset string
"Hello, world!" (0547B30h)
    int length = 0;
0054189C mov     dword ptr [length],0

    while (testString[length] != '\0') {
005418A3 mov     eax,dword ptr [testString]
005418A6 add     eax,dword ptr [length]
005418A9 movsx   ecx,byte ptr [eax]
005418AC test    ecx,ecx
005418AE je     __$EncStackInitStart+3Fh (05418BBh)
            length++;
005418B0 mov     eax,dword ptr [length]
005418B3 add     eax,1
005418B6 mov     dword ptr [length],eax
        }

005418B9 jmp     __$EncStackInitStart+27h

```

```

printf("The length of the string is: %d\n", length);
005418BB mov     eax,dword ptr [length]
005418BE push    eax
005418BF push    offset string "The length of the string
is: %d@... (0547B40h)
005418C4 call    _printf (05410D2h)
005418C9 add    esp,8
...

```

It's notable that MSVC generates code that is often more logically correct than what an average person might write, even correcting inline assembly code where necessary.

Note: Don't use *length*; it is reserved word inline-assembly.

```
#include <stdio.h>

int main() {
    int a = 10, b = 3;
    int c, d;

    c = a / b;
    d = a % b;

    return 0;
}

...
int a = 10, b = 3;
007A1795 mov     dword ptr [a],0Ah
007A179C mov     dword ptr [b],3
int c, d;
c = a / b;
007A17A3 mov     eax,dword ptr [a]
007A17A6 cdq
007A17A7 idiv   dword ptr [b]
007A17AA mov     dword ptr [c],eax
d = a % b;
007A17AD mov     eax,dword ptr [a]
007A17B0 cdq
007A17B1 idiv   dword ptr [b]
007A17B4 mov     dword ptr [d],edx
...

```

There is no assembly after compiling on declaration,
Additionally, the same code is compiled twice, as highlighted in red, because the compiling is **Not Intelligent**.

```
#include <stdio.h>

int main() {
    int a = 10, b = 3;
    int c, d;

    __asm {
        mov eax, dword ptr[a]
        cdq
        idiv dword ptr[b]
        mov dword ptr[c], eax
        mov dword ptr[d], edx
    }

    printf("Quotient :: %d \t Remainder :: %d", c, d);

    return 0;
}

...
cdq ; Convert doubleword in %eax to quadword in
;%edx:%eax
https://docs.oracle.open.solaris

In Intelligent Compiling, the compiler is well aware of the task being performed and optimizes the code to save computing power by reducing the number of operations executed. For instance, when training a neural network, this optimization can reduce millions of operations.

```

```
#include <stdio.h>

int main() {
    int a, b, c, d;
    int x = 10, y = 5;
00EB1795 mov     dword ptr [x],0Ah
00EB179C mov     dword ptr [y],5
...
int a, b, c, d;
int x = 10, y = 5;

```

```
int x = 10, y = 5;
```

```
a = x + y;  
b = x - y;  
c = x * y;  
d = x / y;
```

```
return 0;
```

```
}
```

a = x + y;	
00EB17A3	mov eax,dword ptr [x]
00EB17A6	add eax,dword ptr [y]
00EB17A9	mov dword ptr [a],eax
b = x - y;	
00EB17AC	mov eax,dword ptr [x]
00EB17AF	sub eax,dword ptr [y]
00EB17B2	mov dword ptr [b],eax
c = x * y;	
00EB17B5	mov eax,dword ptr [x]
00EB17B8	imul eax,dword ptr [y]
00EB17BC	mov dword ptr [c],eax
d = x / y;	
00EB17BF	mov eax,dword ptr [x]
00EB17C2	cdq
00EB17C3	idiv dword ptr [y]
00EB17C6	mov dword ptr [d],eax
...	

```
#include <stdio.h>
```

```
int main() {
```

```
    int a, b, c, d;
```

```
    int x = 10, y = 5;
```

```
    __asm {
```

In custom assembly, the repetitive instructions for all four arithmetic operations and optimizes them to a single **mov** instruction in custom assembly, significantly reducing the number of operations.

```
        mov     eax, dword ptr[x]
```

```
        add     eax, dword ptr[y]  
        mov     dword ptr[a], eax
```

```
        sub     eax, dword ptr[y]  
        mov     dword ptr[b], eax
```

```
        imul    eax, dword ptr[y]  
        mov     dword ptr[c], eax
```

```
        cdq  
        idiv    dword ptr[y]  
        mov     dword ptr[d], eax
```

```
}
```

```
    printf("Add :: %d Sub :: %d Mul :: %d Div :: %d",  
          a, b, c, d);
```

```
    return 0;
```

```
}
```

Using -O2 -O3 optimization can sometimes lead to larger code size and slower performance due to increased instruction cache misses and aggressive inlining.

```
#include <stdio.h>
```

```
int two() {  
    return 2;  
}
```

-O2 Flag gcc 14.1

```
two:  
    mov    eax, 2  
    ret
```

```
int main() {
    int t = two();
    return 0;
}

#include <stdio.h>

int two() {
    return 2;
}

int main() {
    int t = two();
    printf("%d",t);
    return 0;
}
```

main:

```
        xor    eax, eax
        ret
```

-O2 Flag gcc 14.1

two:

```
        mov    eax, 2
        ret
```

.LC0:

```
        .string "%d"
main:
        sub    rsp, 8
        mov    esi, 2
        mov    edi, OFFSET FLAT:.LC0
        xor    eax, eax
        call   printf
        xor    eax, eax
        add    rsp, 8
        ret
```

Source Code Preprocessed file Assembly file object file Executable file

Preprocessing

Preprocess handle directives like `#include`
`#define` etc. Also expands all macros and including
 contents into one file.

File Extension (.i) file

Command: `gcc -E source.c -o source.i`

Note:: The error of redefinition comes here.

Compiling / Assembling

Converts preprocessed file into assembly code.

File Extension (.s) file (.asm) Windows

Command: `gcc -S source.i -o source.s`

objecting

The assembler converts assembly code to
 machine code and generates an object file.

File Extension (.o)

Command: `gcc -c source.s -o source.o`

Note:: Static linked libraries are copied into final
 executable while dynamic link libraries are called
 while execution.

Linking

Linker take one or more objects and combine them into single executable. It resolves references between objects and include libraries.

file Type (None) Unix (.exe) Windows,

Command `gcc source.c -o executable`

Note:: DLL or Dynamic linked Libraries and static are to be specified as command line argument while executing file

`gcc main -L /path/to/lib -l libname -o main.o`

∴ Before linking the definition of any function used doesn't even matter only declaration is required in corresponding .h file (Not found linker error)

CALL ① Push Return address to stack → Control transfer

② Load **EIP** Function Address from main to **fn**

RET ① Pop Return address of main to **EIP**

This means implicitly the stack changes for program to execute properly. (**ESP** changes)

5.2 C/C++ to Assembly with Stack

```
int main()
{
    int arr[] = { 2, 3, 7, 6, 7 };
    int length = sizeof(arr) / sizeof(arr[0]);

    printf("Starting the assembly function...\n");
    int result = add_func(length, arr);
    printf("Result: %d\n", result);

    return 0;
}
```

Generate assembly for C code
using **GCC** no Optimization

```
add_func:
    push rbp
    mov rbp, rsp
    mov DWORD PTR [rbp-20], edi
    mov QWORD PTR [rbp-32], rsi
    mov DWORD PTR [rbp-4], 0
    mov DWORD PTR [rbp-8], 0
    jmp .L2

.L3:
    mov eax, DWORD PTR [rbp-8]
    cdqe
    lea rdx, [0+rax*4]
    mov rax, QWORD PTR [rbp-32]
    add rax, rdx
    mov eax, DWORD PTR [rax]
    add DWORD PTR [rbp-4], eax
    add DWORD PTR [rbp-8], 1

.L2:
    mov eax, DWORD PTR [rbp-8]
    cmp eax, DWORD PTR [rbp-20]
    jl .L3
    mov eax, DWORD PTR [rbp-4]
    pop rbp
    ret
```

```
add_func ENDP
END
```

To optimize your C code for accumulating an integer array, consider custom optimizations that suit your specific problem. While GCC may generate large code with many operations, you can tailor the optimizations yourself to use minimal memory and cpu.

Simplicity is key in AI tasks since they already demand significant computation and memory. Keeping things straightforward helps manage these challenges

Custom Assembly in **add.asm**

```
.model flat, STDCALL
.code
add_func PROC C
    ; Save the base pointer
    push ebp
    mov ebp, esp

    ; Load the length of the array into ECX
    mov ecx, [ebp+8]

    ; Load the pointer to the array into ESI
    mov esi, [ebp+12]

    ; Initialize EAX to 0
    xor eax, eax

loop_label:
    ; Add the current element to EAX
    add eax, [esi]

    ; Increment the pointer to the next element
    add esi, 4

    ; Loop until ECX is 0
    loop loop_label

    ; Restore the base pointer
    pop ebp
    ret

add_func ENDP
END
```

In assembly language, parameters are accessed using the stack pointer relative to **EBP** with a positive offset [**EBP + N**], while local variables are accessed with a negative offset [**EBP - N**]. Integer values are returned through the **EAX** register, and global variables are accessed directly by their memory address.

Custom Printf

```
#include <stdio.h>
```

Include **irvine32.inc**

```

#endif _cplusplus
extern "C" {
#endif

// Declare the assembly function
extern int add_func(int length, int* arr);

// Declare the assembly function
extern int printf_func(const char* String, ...);

#ifndef _cplusplus
}
#endif

int main() {

    int arr[] = { 2, 3, 7, 6, 7, -5, 6, 9 };
    int length = sizeof(arr) / sizeof(arr[0]);

    int result = add_func(length, arr);
    printf_func("%d Printed from Custom Printf \n"
Result: %d\n", 1, result);

    return 0;
}

.code
printf_func PROC C
    ; Save the base pointer
    push ebp
    mov ebp, esp

    ; Load the format string into ESI
    add ebp, 8
    mov esi, [ebp]

print_loop:
    ; Load the byte at ESI into AL
    mov al, [esi]

    ; Check for null terminator
    cmp al, 0
    je print_done

    ; Check for '%'
    cmp al, '%'
    je check_next_char

    ; sys_write for one character
    call WriteChar

    inc esi      ; move to the next character
    jmp print_loop ; repeat the loop

check_next_char:
    inc esi      ; move to the next character
    mov al, [esi]
    cmp al, 'd'
    je print_int

    ; unknown format specifier, ignore it
    jmp print_loop

print_int:
    ; load the integer from the stack
    add ebp, 4
    mov eax, [ebp]
    call WriteInt

    ; move to the next character
    inc esi
    jmp print_loop

print_done:
    ; Restore the base pointer
    pop ebp
    ret

printf_func ENDP
END

```

main02.cpp

```
#include <iostream>

#ifndef _cplusplus
extern "C" {
#endif
```

```
Include Irvine32.inc

.code
printcpp PROC C
    push ebp
```

printcpp.asm

```

extern "C" void printcpp(void* thisPtr);
}

#endif __cplusplus
#endif

class MyClass {
private:
    int number;      // First data member
    char text[20];   // Second data member (char array)

public:
    // Constructor to initialize data members
    MyClass(int num, const char* txt) : number(num) {
        strcpy_s(text, txt); // Copy string to char array
    }

    // Member function to print the values of data
    // members using 'this' pointer
    void printData() const {
        std::cout << "this-> " << this << "\n";
        std::cout << &number << "\t" << number << "\n";
        std::cout << &text << "\t" << text << "\n";
    }

    // Member function to call the assembly routine
    void callAssembly() {
        printcpp(this);
    }
};

int main() {
    MyClass obj(42, "Hello, World!");

    std::cout << "\nCpp :: \n";
    obj.printData();

    std::cout << "\nAssembly :: \n";
    obj.callAssembly();

    return 0;
}

```

```

    mov ebp, esp
    xor eax,eax
    add ebp, 8
    mov eax, [ebp]
    call WriteHex

    mov al,9
    call WriteChar
    xor eax,eax

    mov eax, [ebp]
    mov ebx, [eax]
    mov eax, ebx
    call WriteInt

    mov al,10
    call WriteChar
    xor eax,eax

    mov eax, [ebp]
    add eax, 4
    call WriteHex

    mov al,9
    call WriteChar
    xor eax,eax

    mov edx, [ebp]
    add edx, 4
    call WriteString

    pop ebp
    ret

```

```

printcpp ENDP
END

```

Console Output

```

Cpp :: 
this-> 00BAF974
00BAF974      42
00BAF978      Hello, World!

Assembly :: 
00BAF974      +42
00BAF978      Hello, World!

```

In C++, the **this** pointer points to the start of the memory block where the class's member variables are stored contiguously. You can use this pointer to iterate through the memory and access all class members sequentially.

Note that the **this** pointer is implicitly passed to assembly code in the EAX register by C++.

push ebp

Save current stack frame, allowing

function ~~to~~ to use ebp for parameters

pop ebp

or local variables. Then atleast restore

it for smooth transition back to

main or calling function.

RET N

Perform task of RET pop address

main and put in EIP. In RET N

it adds immediate value to ESP.

Hence clean up stack upto N Bytes.

SUB ESP, 0x40

This will move ESP pointer to 64

bytes behind which can be used

for storing local variable.

Note: This sub ESP could result in overriding memory
of other system and could result in system crash.

Avoid this as much as you can as I have not got it
my programmes as examples.

6.1 Recursion & Assembly

C

```
unsigned long long factorial_recursive(int n) {
    if (n == 0 || n == 1) {
        return 1; // Base case
    }
    else {
        return n * factorial_recursive(n - 1); // Recursive call
    }
}
```

Recursive Factorial of 63 = 1585267068834414592

Time taken: 5455058 nanoseconds

Assembly

```
factorial_recursive:
    push rbp
    mov rbp, rsp
    push rbx
    sub rsp, 24
    mov DWORD PTR [rbp-20], edi
    cmp DWORD PTR [rbp-20], 0
    je .L2
    cmp DWORD PTR [rbp-20], 1
    jne .L3

.L2:
    mov eax, 1
    jmp .L4

.L3:
    mov eax, DWORD PTR [rbp-20]
    movsx rbx, eax
    mov eax, DWORD PTR [rbp-20]
    sub eax, 1
    mov edi, eax
    call factorial_recursive
    imul rax, rbx

.L4:
    mov rbx, QWORD PTR [rbp-8]
    leave
    ret
```

```
unsigned long long factorial_iterative(int n) {
    unsigned long long result = 1;
    for (int i = 2; i <= n; i++) {
        result *= i;
    }
    return result;
}
```

Iterative Factorial of 63 = 1585267068834414592

Time taken: 2978189 nanoseconds

```
factorial_iterative:
    push rbp
    mov rbp, rsp
    mov DWORD PTR [rbp-20], edi
    mov QWORD PTR [rbp-8], 1
    mov DWORD PTR [rbp-12], 2
    jmp .L2

.L3:
    mov eax, DWORD PTR [rbp-12]
    cdqe
    mov rdx, QWORD PTR [rbp-8]
    imul rax, rdx
    mov QWORD PTR [rbp-8], rax
    add DWORD PTR [rbp-12], 1

.L2:
    mov eax, DWORD PTR [rbp-12]
    cmp eax, DWORD PTR [rbp-20]
    jle .L3
    mov rax, QWORD PTR [rbp-8]
    pop rbp
    ret
```

1. Function Call Overhead

- Pushing registers onto the stack
- Saving the current instruction pointer
- Jumping to the function
- Returning from the function
- Popping registers from the stack

2. Stack Overflow Risk

Deep recursion can lead to stack overflow errors, causing the program to crash.

These are the reasons why Recursion can be inefficient when computing factorials.

6.2 C & Assembly Subroutines

At assembly level there are some hardware accelerators which will run in linear time as they are implemented in hardware itself. If we use these in assembly then we can boost our computational power and reduce time.

```
%%writefile str_len.asm ; returns length of given pointer to string
```

```
section .text  
global str_len
```

```
str_len:
```

```
    mov edi, [esp+4] ; Load pointer to string  
  
    mov ecx, -1      ; Set loop counter to -1  
    xor al, al       ; Clear al register  
    repne scasb     ; Scan string for null terminator  
    mov eax, ecx     ; Move negative length to eax  
  
    neg eax         ; Convert to positive length  
    dec eax         ; Adjust for null terminator  
    ret             ; Return length in eax
```

The **SCASB** instruction in x86 assembly compares the byte in AL with the byte pointed to by EDI (or ESI) and adjusts the index based on the Direction Flag. With **REPNE**, **SCASB** scans the string, continuing until a match is found or ECX reaches zero.

```
%%writefile str_cmp.asm ; Compare both string where difference found returns address
```

```
section .text  
global str_cmp
```

```
str_cmp:
```

```
    mov edi, [esp+4] ; Load pointer to string 1  
    mov esi, [esp+8] ; Load pointer to string 2  
  
    CLD             ; Clear direction flags for string ops  
  
    mov ecx, -1      ; Initialize loop counter to -1  
  
    repz cmpsb      ; Compare bytes in both strings until ECX is 0  
  
    jne found_difference ; If not equal, jump to difference handling  
  
    xor eax, eax     ; If equal, return 0 in EAX  
    ret
```

```
Found_difference:
```

```
; Return index of first mismatch  
    mov eax, ecx      ; Move loop counter to EAX  
    neg eax          ; Convert to positive index  
    dec eax          ; Adjust for 0-based indexing  
    ret
```

The **repz** prefix in x86 assembly repeats a string operation (like **cmpsb** or **scasb**) while the zero flag (ZF) is set and ECX is non-zero. The **cmpsb** instruction compares bytes at ESI and EDI, adjusting these registers based on the Direction Flag (DF).

```
%%writefile mem_set.asm ; Initialize all memory location to Null
```

```
section .text  
global mem_set
```

mem_set:

```
CLD          ; clear direction flags  
  
mov edi, [esp+4] ; Load array pointer into EDI  
mov ecx, [esp+8] ; Load array length into ECX  
  
xor al, al      ; Set AL to 0 (null value)  
  
rep stosb      ; Fill array with null values  
ret
```

The **stosb** instruction stores the value in AL into the memory location pointed to by EDI (or RDI), then adjusts EDI based on the Direction Flag (DF). The rep prefix repeats a string operation (like **stosb**) for a count specified by ECX, continuing until ECX reaches zero.

```
%%writefile mem_cpy.asm           ; copy contents from one memory to other  
section .text  
global mem_cpy
```

```
mem_cpy:  
CLD          ; clear direction flags  
  
mov edi, [esp+4] ; Load string pointers into EDI and ESI  
mov esi, [esp+8]  
mov ecx, [esp+12] ; N bytes to copy  
  
rep movsb      ; Compare bytes in both strings  
ret
```

The **movsb** instruction in x86 assembly moves a byte from the memory location pointed to by ESI (or RSI in 64-bit) to the memory location pointed to by EDI (or RDI in 64-bit). After the byte is moved, ESI and EDI are either incremented or decremented based on the state of the direction flag (**DF**).

C Declarations

```
// Declare the str_len function, which calculates the length of a string  
extern "C" int str_len(const char *str); // Returns the length of the input string  
  
// Declare the str_cmp function, which compares two strings  
extern "C" int str_cmp(const char *str1, const char *str2); // Returns 0 if equal, non-zero if not equal  
  
// Declare the mem_set function, which sets a block of memory to a specific value  
extern "C" void mem_set(const char *destination, int N); // Sets N bytes of memory at destination to 0  
  
// Declare the mem_cpy function, which copies a block of memory from one location to another  
extern "C" void mem_cpy(const char *destination, const char *source, int N); // Copies N bytes from source to destination
```

6.3 Local Variables in Subroutines

In continuous running applications, functions should clear their local variables before exiting to prevent security vulnerabilities. This practice helps avoid data leaks and ensures sensitive information isn't left in memory. Consistently clearing memory enhances both security and stability of the application.

Login system vulnerability: User credentials remain in memory after login function exits, leaving sensitive data exposed if not properly cleared.

Freeing dynamic memory ensures sensitive info is erased, protecting against unauthorized access or memory dumping.

The screenshot shows the Microsoft Visual Studio IDE interface. The top menu bar includes 'File', 'Edit', 'View', 'Project', 'Build', 'Tools', 'Help', and 'Solution Explorer'. The 'Solution Explorer' window on the right lists 'Git Changes'. The main workspace has tabs for 'Disassembly', 'add.asm', 'memfree.asm', and 'main02.c'. The 'memfree.asm' tab is active, displaying assembly code:

```
1 .model flat, STDCALL
2
3 .code
4
5 mem_free PROC C
6
7     CLD          ; clear direction flags
8
9     mov edi, [esp+4] ; Load array pointer into EDI
10    mov ecx, [esp+8] ; Load array length into ECX
11
12    xor al, al    ; Set AL to 0 (null value)
13
14    rep stosb    ; Fill array with null values
15
16
17    ret         ; Return
18
```

The status bar at the bottom shows 'ret <1ms elapsed'.

The 'Registers' window on the right shows register values:

Register	Value
EAX	003BFA00
EBX	0042F000
ECX	00000000
EDX	00000000
ESI	003BFA00
EDI	003BFB58
EIP	008E509D
ESP	003BF9F4
EBP	003BFB6C
EFL	00000024

The 'Memory' window below shows the memory dump starting at address 0x003BFA04. The 'Call Stack' tab is also visible at the bottom left.

If you prefer not to use dynamic memory, you can use memfree (previously known as memset method) to clear the memory space. This helps improve security by ensuring sensitive data is erased.