COAL.

ASSEMBLY LANGUAGE (LAB)

→ Integer Constant

26 → decimal (26d)

26h → hexa-decimal

101010 b → binary.

→ Character Constant

'A' | "A"

//DATA TYPES//

EAX = 32 bit register.

(1) BYTE , SBYTE

↳ usigned int 0-256 ↳ signed int ; -128-127

(2) WORD

↳ 2 Byte signed 4 unsigned

(3) D WORD

↳ 32 Bit

(4) Q WORD

↳ 64 bit

(5) T BYTE

↳ 80 bit

* For string

greeling BYTE "GOOD MORNING", 0

↳ each byte for each char ↳ Terminator

→ For Array

list BYTE 20, 10

→ For Uninitialized Variable

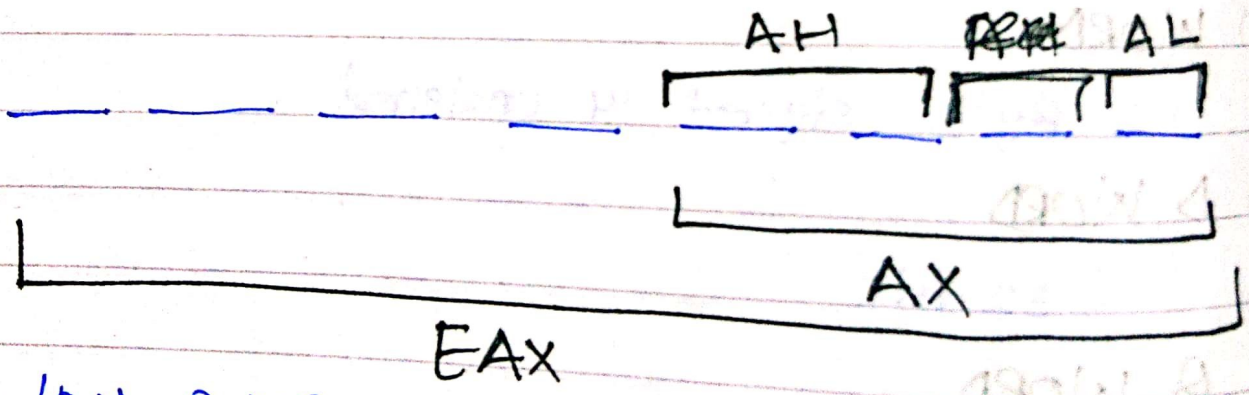
var SWORD ?

↳

Value will be assigned @ Run time

Registers

EAX = 32 bits ; each hexa = 4 bits ;



AL/AH = 8 bits

Ax = 16 bits

Array

→ var1 BYTE 20 DUP(0)

↳ 20 bytes reserved → initialized to 0

→ DUP(?) → All uninitialised

→ DUP("STACK")

// COAL — Sir Danish //

- Basic Micro-Processor design
  1- I/O devices)
  2- Memory Storage Unit
  3- CPU / ALU / CU / clock / registers
  4- buses.

- Control Bus
- data Bus
- Address bus

- ALU → arithmatic & bitwise op.
- clock → Synchronises CPU operations
- CU coordinates " "

INSTRUCTION EXECUTION CYCLE

- Any program that needs to be executed needs to be fetched into memory.
- Instructions loaded into Instruction Queue
- The operands are then fetched (operands = data values a + b
- ALU executes the instruction
- Output stored // learn diff b/w var & regis

→ Variables (Aspects of variables)
size, name, address, data
↳ mask over data

* • CODE VS • DATA
(instructions ↳ raw data

→ when an instruction is loaded into memory, segments are formed. Segments contain multiple addresses

→ data seg
→ code seg

segments:
→ data
→ code
→ stack
} segments are not contiguous in memory

↳ these segments, bee are stored in memory → base address
↳ Segments' base is assigned a base address within the segment the individual mem. locations are assigned offset

(Theory)

Status Registers → calculations etc (used in)
↳ eg. comparisons
x = var1 − var2
(var1 = var2) if x == 0 → zeroflag.
(var1 < var2) if x < 0 → signflag.
else var1 > .

MODES OF OPERATION

→ Protected mode → CPU, if working on segment P1, it cannot reach anyother memory segment.

→ System mode → CPU can access any segment of memory

//note: registers are global entity

Assembly language of
↳ X86 Irvine32
32 bit .Asm files.
MASM
visual studio

ASSEMBLY LANGUAGE FUNDAMENTALS (chap3)

```
main PROC
mov eax, 5        → operands
add eax, 6        → copy 5 to eax register
INVOKE ExitProcess, 0

main ENDP
```

move destination, source.

```
.data    // data segment → var global to prog.
   Sum Dword 0
.code    // instructions
```

MOV AH, AH → reg to reg.
MOV AH, 0AH → hexa value 0A, AH ↓ into

MOV AH, 'A'
↳ ASCII value (65) will be saved in AH

→ String literal → series of characters

Arithmatic Precedence

1st	( )	Parenthesis
2nd	− | +	Uniary (as sign)
3rd	× | ÷	
4th	MOD	
5th	Add | Sub	Binary

Reserve Words

eg. PROC cannot use as variable.

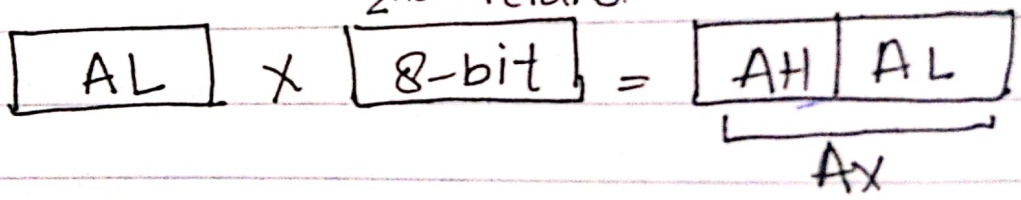
we can pick names for (identifiers) for
↳ Procedure, variable, labels

COAL (LAB)

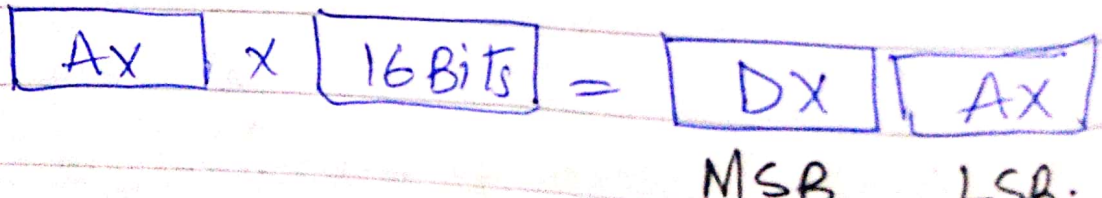
Types of Registers
- → General Purpose
  - → data reg
  - → Pointer reg.
  - → Index reg.
- → Control
- → Segment registers

EAX (Accumulator register)

→ For multiplication
  - ↳ the first operand is already present in the accumulator register ; the result is also stored in the same register



→ IF operand is larger than 8 bits; Data register is used.

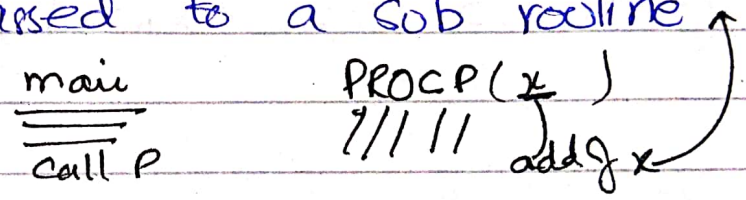


→ Pointer Registers

(1) Extended Instruction pointer
↳ add of next (executable) instruction

(2) Extended stack pointer
~~add of~~ offset value w.r.t.n stack

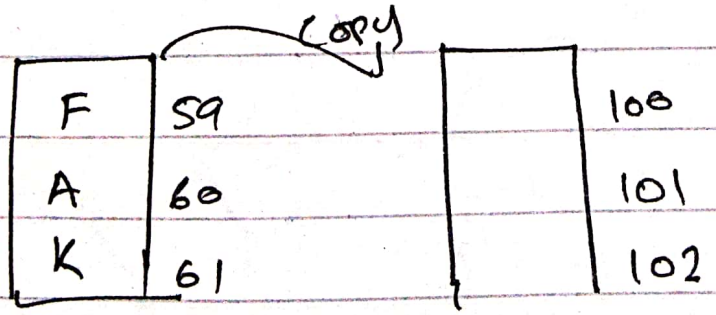
(3) Extended base pointer
helps in referencing the parameter variables passed to a sub routine



→ INDEX REGISTERS

- Extended Source Index (ESI)
  source index for string operations
- Extended Destination Index
  dest. index for string operation

eg.



ESI 59
EDI 100

//NOTE
→ CS, EIP & IP cannot be destination operands

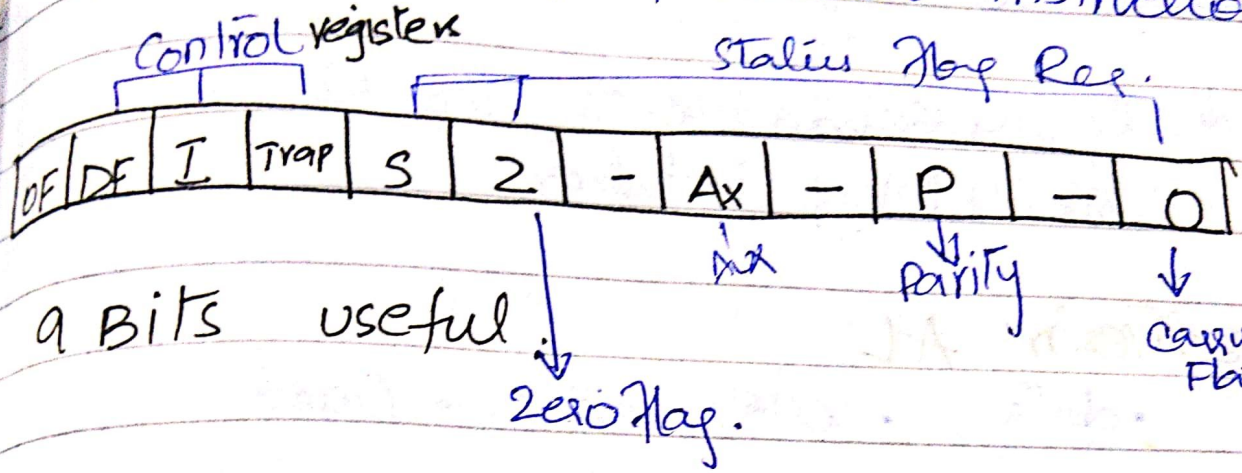
→ For Increment → INC, register
→ For decrement → DEC, register

E
→ MOV 2x Instruction
(MOV w zero extend)

MOV S x (MOV w sign extend)

FLAG REGISTER

↳ updates itself after each instruction



Assembly lang (theory)

Instruction → work to do 1 operation

mnemonics → short codes for operations
eg. MOV.

label ; a label give to some instruction so we can use it (goto) at any time in the programs.

eg. L1 : MOV Ax, Var1

↳ label.

target :
mov ax, bx.
Jmp target

looping

Directives → to make use of any resource outside the program.

↳ directly understandable to the microprocessor
@ time of linking (included)

directives in AL
.data , .code, .stack, . Dword

myword DWord 26 ; Dword Tells the assembler to reserve space in the program for a double word variable.

// Diff b/w directives & Instructions
Occupy address

• 386 → following program will be for a ...

• model → decides how many memory seg to be made for your program

• stack → 4096. (spare memory)
↳ creates a stack segment with 4096 bytes → 4kB

Read only memory

END P → end of procedure
END main → End of program
↳ entry point name of procedure.

END P1 → In the whole main only P1 will work → rest procedures would be called

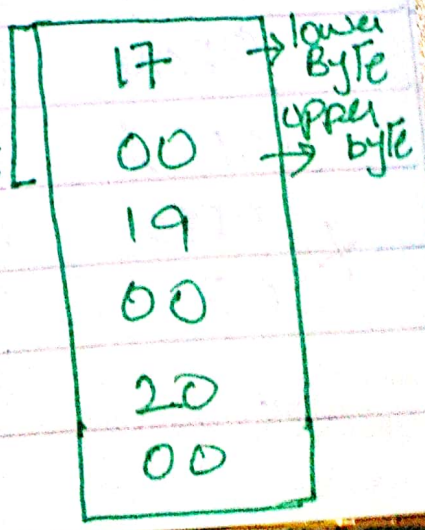
Defining data
// always under .data

X BYTE 17
↳ scalar

X mask over address

X BYTE 17,18,19
↳ array;

→ Any data > a byte is stored in chunks; lower bytes will come first
eg. X BYTE 17, 19, 20



// each element takes 2 bytes, where lower byte comes first.
// 17 = 0017 as a word

→ Since we only have one variable X → ~~name of the array~~ and it has 3 values 17, 18, 19 → here the other 2 values are unnamed → to reach them we use their offset

eg.

00	00	19	00	20	00
100	101	102	103	104	105
X					

To access 19 we use X+2

using Moliple initializers eg.

list 1 Byte 1, 2, 34
Byte 5, 6, 7, 8

To access 8 → we use list1+7.

list 2 Byte 10, 20, 30, 40
word 50, 60, 70, 80

0F01	10
0F02	20
0F03	30
0F04	40
0F05	50
0F06	00
0F07	60
0F08	00
0F09	70
0F0A	00
0F0B	80
0F0C	00

0F05 → element.

✓

X

8421 19 01

0919

To move 70 in Ax we write

Mov Ax, [list2 +8]

// list 2 +8 is basically 70
But since it is an array of word, it would return the whole word 0070
// It doesnt matter if we use list2+8 or list2+9 we would get 0070 → either addren would return the whole element

Arr word

135 1901h 2000h, 0F01h.
Str byte "COAL"

135	19 01	// EDS
136	01 19	↳ Data register
137	20 00	// if there ~~are~~ is only
138	00 20	one array and it
139	01 01	is un-named → we
13A	04 0F	will take the base
13B	'C'	addren of the data
13C	'O'	register
13D	A	
13E	L	

00 C

Defining Strings

→ greeting1 BYTE "GOOD MORNING",0

NULL Terminator

↳ else the string doesn't have an exit point NOT a part of the array.

* DUP operator (Duplicate)

var4 BYTE 10, 3 DUP(0), 20

↳ in memory

10	0	0	0	20

BYTE 4 DUP("STACK")

= "STACKSTACKSTACKSTACK"

BYTE 20 DUP(?)

↳ uninitialized

Map Vax Sword 17, 3 DUP(-17,19)

17	00	-17	00	-19	00	19	00	. . . . .

Var DWORD 3DUP(4 DUP('ABC'))

↳ size of data 36.

LITTLE ENDIAN ORDER.

↳ all data types larger than a byte store their individual bytes in reverse order

Var1 DWORD 12345678

78
56
34
12

→ .data vs .data?.

↳ data variables reserve bytes in the memory @ compiletime even if it is not used in the program

↳ any data variable under this is not initialized & doesn't take memory space UNTIL used in code.

.data ?

bigArray DWORD 5000 DUP(?) ↳ has to be initialized

↳ doesn't allocate memory until variable is used in code

SYMBOLIC CONSTANTS;

PI = 3.14267 . . . .

↳ not a variable; wherever you write PI, you get its value.

↳ not in memory → only in program.

↳ value cannot be changed @ runtime

CURRENT LOCATION COUNTER; ($)

V1 WORD 16h, 160h, 1600h.

D1 DWORD $

↓

Since this is the current location counter, it will assign 0C10907 to D1

0160

0C10901	16
0C10902	00
0C10903	60
0C10904	01
0C10905	00
0C10906	16
→ 0C10907	07
0C10908	09
0C10909	C1
0C1090A	00
0C1090B	

V1 WORD 160, 160h, 1600h.

LIST BYTE 10, 20, 30, 40

listsize BYTE = ($ - ~~VList~~)

$ = C10907

↳ This would give the offset (size of array) current location − start of array.

listSize = C10907 − C10901

= 6

// NOTE, listSize should be IMMEDIATLY After the list other wise current location would change accordingly

DATA TRANSFERS INSTRU (4.1)

// cannot use 2 memory operands.

eg Mov var1, var2 ✗
not possible

→ Addressing Modes
↳ diff ways of accessing memory

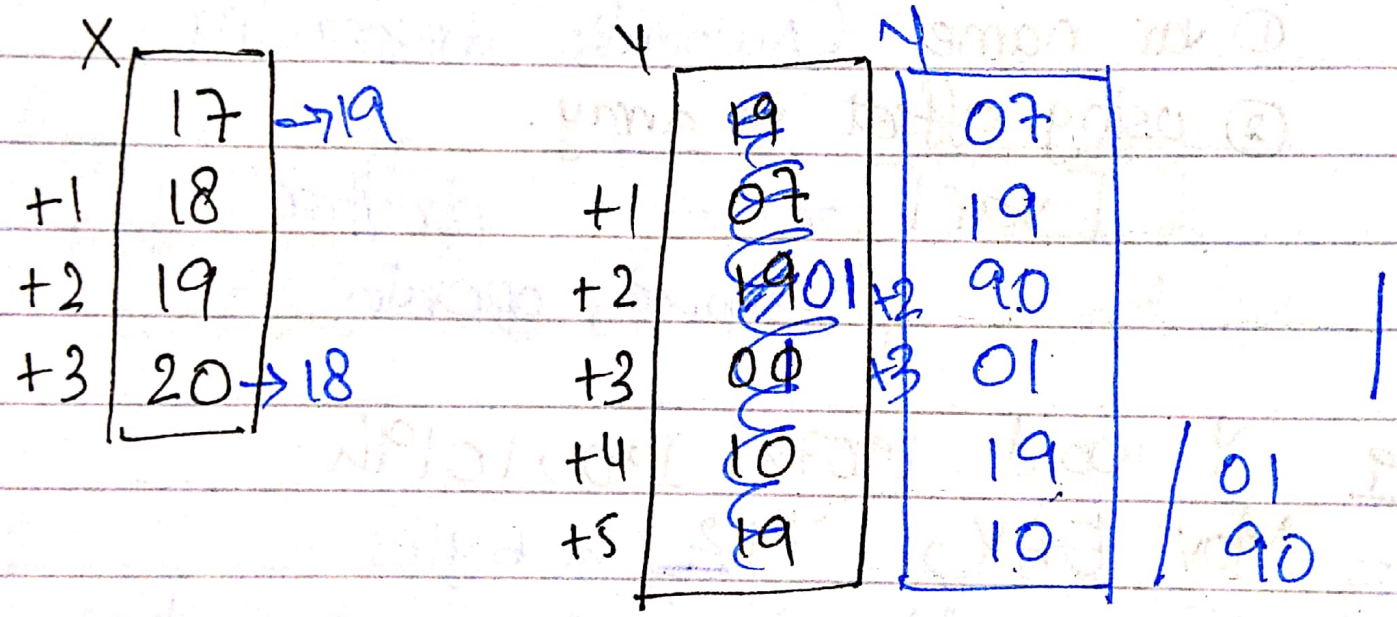
① var name (Automatic dreffrencing)
② using offset u array.
[var1] → manual dreffrencing
↳ dreffrencing operator

→ eg Y word 1907h, 190, 1019h.
Mov EAX, [Y+2] → Bytes
↳ this would give an error bec dest > source
↳ to solve this we have Movzx & MovSx
it is called data extension
Movzx → it would add zeros on the LHS.
MovSx → for signed numbers, it adds 1s to the LHS.
MSB
either (1/0)

→ XCHG; exchanges the values of two operands Atleast one register should be (Immediate operands arent allowed) Xchg mem, mem

Not allowed!

```
.data
    X  BYTE   17h, 18h, 19h, 20h
    Y  word   1907h, 0190h, 1019h.
```



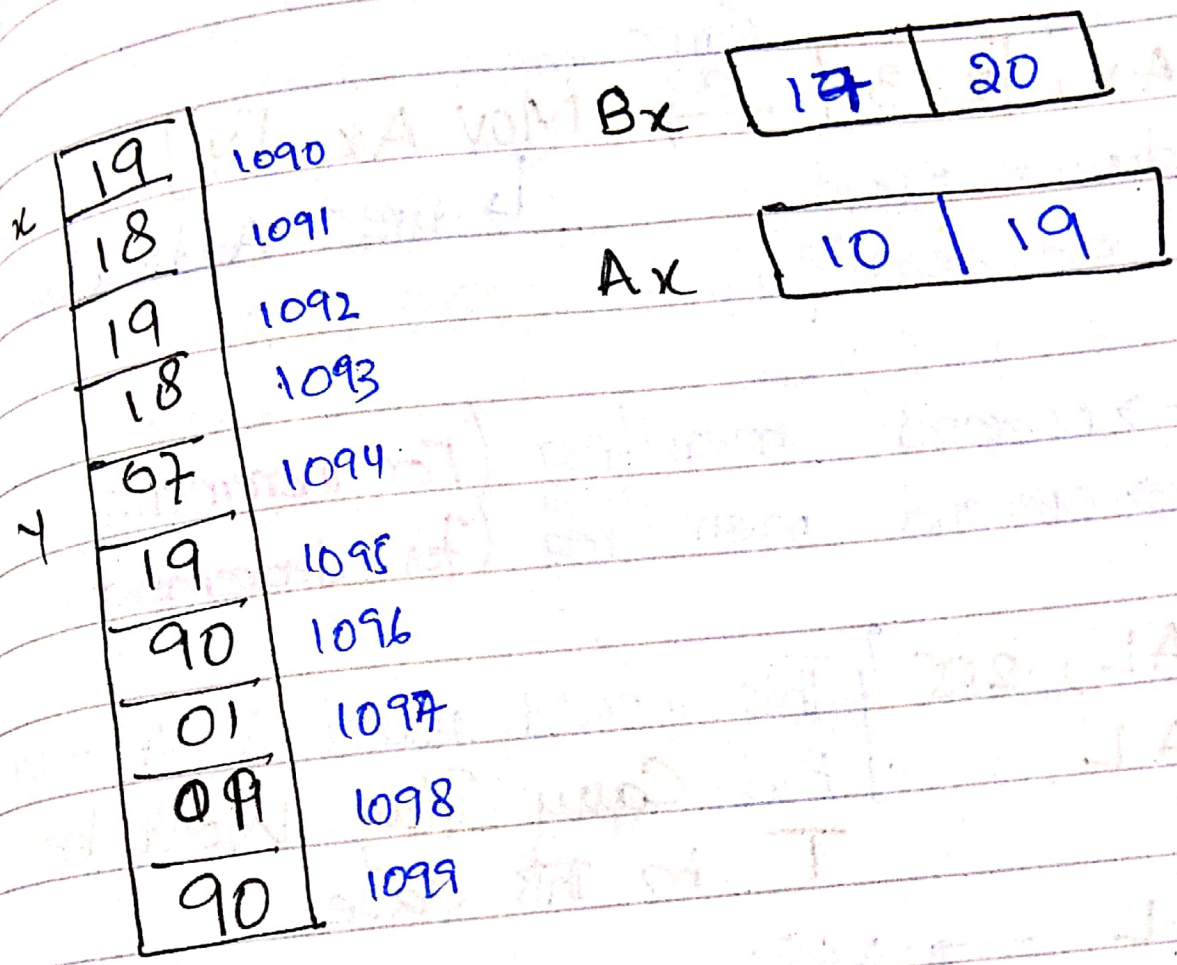

```
Mov BL,[x+1]      BL = 18
Mov BH,[x+2]      BH = 19
Mov Ax,[Y+2]      Ax = 1900  0190.
```

```
Xchg BL, x+3      BL=20   x+3=18
xchg AL, AH       AL=01   AH=90
xchg Ax [Y+4]     Ax=1019 ; Y+4=
```

xchg Bh , [x]

Bh = 17

x = 19



Q

X Byte 19, 20, 21
Y word 1701h.

→ MOV AX, [X+3] Same as → MOV AX, [Y]
// ↳ direct offset addressing ↳ direct Addressing

* INC → operands mem/reg (For Increment)
* DEC → operand mem/reg (for decrement)

eg.

MOV AL, 255
INC AL

} This would result in AL=0000 But Carry flag won't be T in this case

DEC AL → 255.

eg.

X word 250, 1701, 1801

INC word PTr [X+2]
↳ need to tell compiler that H is byte/word etc bec we are using addrsn

1 1 1 1 (128 64 32 16 8 4 2 1)
+1 1 0 0 0 0 0 0 0 — 128.
1 0 0 0 0 0 0 0 1
0 1 1 1 1 1 1 1
255.

ADD, destination, source.
↳ operands → reg/mem
- mem, mem // not allowed
- size = size.

→ MOV AL, 255 // This would set the
ADD AL, 1 carry flag to 1,
CF=1, PF=1, AF=1
↳ AL = 0000 0000.

↳ OF=0 & SF=0 bec these flags are for signed operations

* SUB, destination, source.
→ MOV AL, −128 | would result in
SUB AL, 1 | AL = 1111 1111 (FF)
CF=0, ZF=0, PF=1
make sure AF=0, SF=1, OF=1

→ Carry Flag is only for Unsigned operati

$-5 + (5-4)$.

$-5 + 1$

-4.

---

* **Neg** (Negate operation)
  - ↳ reg / mem.
  - change from +ve → -ve & viseversa

```
Mov AL, -128
Neg AL.
```
would result in overflow.

→ Qs

$A = -B + (B-C)$

```
Mov Eax, B.
Sub Eax, C
```

X
```
Sub Eax, B
```

```
Neg B
Add eax, B.
Mov A, eax
```

IMPORTANT

Direct [1009FC1], VAR1, [VAR1]

Base OFFSET [var 6+12]

INDIRECT ADDRESSING [EBX], [ESI].

INDEXED ADDRESSING ARR[ESI], ARR[7]

(4.8) DATA RELATED OPERATORS & DIRECTIVES

- OFFSET - Operator
- PTR - operator
- TYPE - Operator
- LENGTHOF - Operator
- SIZEOF - operator
- LABEL - Directive

1) OFFSET

→ 10FOC1

19	
20	
21	←

Base Add 10FOC1
offset 2
offset Add 10FOC3

→ For address use 32 Bit destination

eg mov esi, OFFSET dval
↳ would move offset address of dval to esi (register)

2) PTR

↳ used to override the size of an operand.

eg myDouble Dword 12345678h // means we are accessing a subset (equal to word) of myDouble

mov, ax, myDouble
mov, ax, WORD PTR mydouble
↳ ax = ~~4856~~ 5678

mov

eg x BYTE 19, 20, 21, 22, 24, 25

.code

mov eax, Dword PTR [x+1] →

// eax = 24 22 21 20

Qs Why little endian? Importance

// search

3) TYPE

↳ Tells us the type of memory.

↳ no. of bytes occupied.

eg. x Byte 10.

Type x = 1 | size of single element in an array

4) LENGTHOF

↳ returns no. of elements in an array.

5) SIZEOF

↳ size of the whole array.

(no of elements * size of element)

Sizeof = lengthof * Type.

```
.data
    X   Dword   1717h, 1901FD01h.
    Y   Dword   ?

    mov  Y, offset [X+3]
    mov  EDX, [Y].
                ↳ doesnot dreference

Y = 1004.
EDX = 1004.

    mov  ESI , offset [x+3]
    mov  EBX, [ESI]
                ↳ ESI = 1004
```

↳ would move the value @ address 1004 to EBX.

-1001	17
+1 1002	17
+2 1003	0D
+3 1004	0D
1005	01
1006	FD
1007	01
1008	19
1009	04
100A	10
100B	00
100C	00

// hence proving registers can act like pointers, but variable cannot

// mov EBX, ESI

↳ would move VALUE of ESI [] → this would dreference

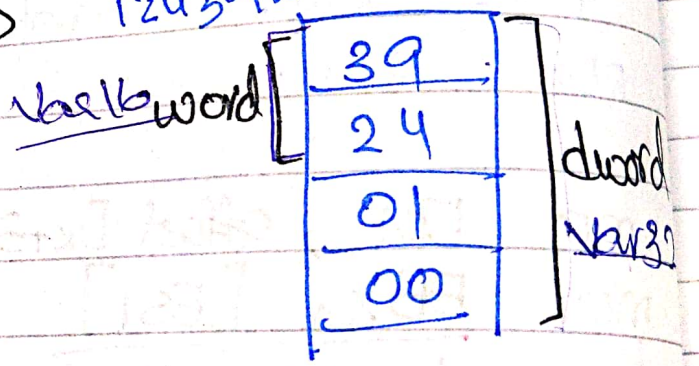
6) LABEL

eg
- ↳ assigns an alternate lable name
- ↳ Type to an existig storage location → doesnot allocate storage of its own

Qs 1

eg. .data
Val16 LABEL WORD
Val32 DWORD 12439h.

3) →



4) 4.4 INDIRECT ADDRESSING

```
.data
    val1 Byte 10h ,20h, 30h
.code
```

5)
```
    mov esi , OFFSET Val1
    mov al , [esi]
    inc esi
    mov al, [esi]
    inc esi
    mov al, [esi].
```

INC [Arr+2]

INC ESI → Address would inc
INC [ESI] → value would inc

INC [var1] , INC [var1]
↳ same thing ; would inc value.

INDEXED OPERANDS

```
.data
    array W word 1000h, 2000h, B000h.
.code
    mov esi,0
```

→ JMP AND LOOP OPERATIONS
↳ Conditional Jumps based on status
↳ Un-Conditional Jumps

★ JMP Instruction causes an unconditional transfer to a destination, identified by a code label

```
top:
    INC AX
    MOV Bx, AX
    jmp top
```

↳ offset of destination is given to pointer

★ LOOP Instruction
↳ loop according to ECX counter.

Step 1 CX (decrements)]
Step 2 IF (X > 0)] happen when reaches the Instruction of loop

4	1
3	2
2	3
1	4
0	

eg.

```
mov ax,0
mov ecx,5
L1: inc ax
loop L1
mov bx, ax
```

execute ↓

reads Loop ←

then dec ecx;

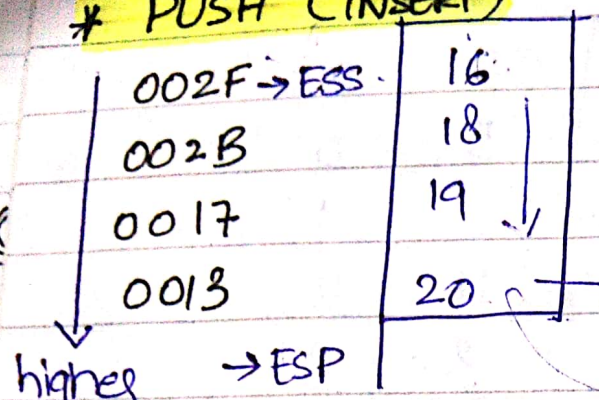
ecx	ax
5	0
5	1
4	2
3	3
2	4
1	5
0 — terminate	

loop L1

STACK OPERATIONS

↳ Last in first out.

* PUSH (INSERT)



- stack grows downwards
- ESS — base addres
- offset of most recent inset is given by ESP.

↳ Initially ESP = ESS

↳ then w/ every push ESP is decremented by 4.

↳ above example, the first element pushed was 16 so ESS points to that (base)

↳ 20 was the last element to be inserted so ESP points to that

// Memory segment → grows ↑

// stack Segment → grows ↓

STACK.

POP (TO GET)
↳ Every POP will Increment ESP by 4

→ when POP → the last element (20).
will be removed
↳ ESP will now point to addren of 19.

↳ INSTRUCTIONS
X PUSH reg / mem16 → allowed but don't use for now
PUSH reg / mem32 ✓
PUSH imm32 ✓
→ Push cannot be < than 16 bits

POP reg/mem16 } Imm not allowed
POP reg /mem16 }

• If operand is 16 bits, ESP is incremented / decremented by 2; if 32 bits by 4.

OF8 OFB 0IB
OF7 OFC 0IC
OF6 OFD 0ID
OF5 OFE
OFF
100

- PUSHAD

Pushed all 32-bit general purpose registers on the stack in the following order: EAX = CF

ECX = 12

EDX = 77

EBX = 99

ESP → value bef executing Pushad 100h

EBP = 70

ESI = 00

EDI = F9.

100	CF
~~104~~ OFC	12
OF8	77
OF4	99
OF0	100
OEC	70
	00
	F9

// You copied the values wrong

- POPAD

↳ Pops the same registers in the reverse order.

EDI — ESI — EBP ... etc

→ POPA & PUSHA → same ; but for 16 bits

5.2 DEFINING & USING PROCEDURES

↳ To define a subroutine

CALL → calling any Procedure

RET → where to return(?)

```
main PROC
:
:
End PROC(?)
```

```
P1 PROC                          EIP = 0017
eg: MOV Ax, 17h     0017
:   MOV Bx, 35h     0018
:   CALL P2         0019         EIP = 070C
:   MOV Cx, Ax      001A
:   RET             001B         stack = 001A
P1 ENDP
```

```
P2 PROC
:   MOV Cx, 11h     070C
:   ADD Ax, 2       070D
:   CALL P3         070E         EIP = 0986
:   ADD Cx, Dx      070F         stack = 070F
:   RET
P2 ENDP
```

```
P3 PROC
:   MOV Dx, 21h     0986
:   ADD Cx, Dx      0987
:   ADD Cx, Bx      0988
:   RET             0989 ; POP into EIP
P3 ENDP                          EIP = 070F
END P3
```

// procedure called @ end is completed first.
EIP → next instruction to be executed

//CALL will push the return address into stack and changes EIP to the next instruction to be executed (obv of the called proc)

// We can Pass Arguments through Registers since they are global entity

NOTE

END P1 → Tells this is the starting point of Program

if

END P2 → whole program starts from P2; P1 is useless

// Revision

```
L1:  Mov Cx ...
     :
     :
}.
```

local to the procedure; to make global use L1::

→ Call ⟶ JMP to another procedure and returns back

JMP → doesnt return

XOR → same Input
↳ 0 output

CONDITIONAL PROCESSING

AND INSTRUCTION

Performs a boolean AND operation b/w each pair of matching bits in two operands

AND destination , Source
↳ result stored in destination

eg.

```
MOV AL, 1010 1110b
MOV CL, 1101 1000b
AND AL,CL ;        AL= 1000 1000
```

↳ every time AND is performed
CF=0 ; OF=0 → 8 bits ka AND can never be 9 bits

OR INSTRUCTION

↳ same as And → Just diff operation
↳ in above eg.

$$AL = \frac{1111}{F}\ \frac{1110}{E}$$

XOR INSTRUCTION

AL = 0110 0110

// Carry & Overflow = 0 for all above

11111111
x→00000010
x→00000010

x-0010
1111
x-0010

eg X byte → value unknown

↳ to check value

ADD X, 1111 1111
OR X, 0000 0000

But these would change the value of 'X'

↳ to avoid this we use TEST

↳ Similar to AND, but this sets the flags and doesn't store in destination (X)

↳ eg to check the MSB of X

Test X, 1000 000

↳ In this case if ZF=0 → this means MSB was 1 if ZF=1 → MSB was 0.

NOT INSTRUCTION

↳ only on bits

MOV AL, 10h
NOT AL ; AL = 1110 1111

↳ doesn't have any change on flag

// 'NOT' can be used to encrypt data
for eg A = 41h ; if we NOT A
it turns to A=BE ;

↳ A = 0100 0001

A' = 1011 1110 .

↳ change this @ destination .

CMP INSTRUCTION (dest – source)

we use this to compare integers .

This performs an implied subtraction of a source operand from dest

// ONLY FLAGS ARE AFFECTED

if

CF = 1, ZF = 0 } source >>

CF = 0, ZF = 1 } destination = source

SF = CF } dest >>

when signed operands

6.3 CONDITIONAL JMPS

Based on ↳ ecx
↳ signed
↳ unsigned
↳ flag

eg.

```
cmp eax, 0
Jz  L1        ; Jump if zeroflag = 1
              ↳ In this case ZF=1 when eax is also 0.
:
L1:
```

eg2

```
add dl, 10110000b
Jnz L2        ; Jump when ZF != 0
              ↳ if dl = 4Fh then this would mean result is 0 hence this loop would not work.
:
L2
```

eg& Jcond — destination JMPS based on flags

- Jz
- Jnz
- Jc
- Jnc

JUMPS BASED ON EQUALITY

- JE → Jump if equal (left op = right op)
- JNE → " " not equal
- JCXZ Jump if CX = 0
- JECXZ " " ECX = 0
- JRCXZ Jump if RCX = 0.

```
CMP BL, AL
JNE L1
```

} For JUMP inst ; it responds to the instruction immediatly above it

Ex1

```
XOR ECX, ECX
JECXZ L2
```

} since xor same bits then ans is defo zero

// JE → Indirectly checks zero flag //

Ex2

```
mov bx, 1234
sub bx, 1234
Jne L5    ; Check if ZF=0
Je L1     ; check if ZF=1
```

↳ JE == JZ

↳ UNSIGNED COMPARISION

JA	Jump it above (left op > right op)
JNBE	Jump it not below or equal
JAE	Jump it above or equal (dest >= source)
JNB	Jump it not below
JB	Jump it below (dest < source)
JNAE	Jump it not above or equal
JBE	Jump it below or equal (dest <= source)
JNA	Jump it not above

Example

```
if ( A > B || B < C && B > D )
    A ++;
else
    B++;
    C++;
    D++;
```

↳ write code in asm.

```
mov  eax, A
mov  ebx, B
mov  ecx, C
mov  edx, D
```

```
Cmp  B , D
JA  L1
L1 :
    Cmp  B, C
    JA  L2

L2:
    inc  A
```

X

```
Cmp A , B
JA  L2
```

```
cmp B, D
 JA  L1
L1:
   cmp  B, C
    JA  L2
L2:
   Inc A
cmp  A, B
   JA  L2
   Jmp CondElse
CondElse
   Inc  B
   Inc  C
   Inc  D
```

SIGNED COMPARISON

↳ Jumps based on signed

JG → Jump if greater

JNLE → Jump if not less or equal

6.4 CONDITIONAL LOOP INSTRUCTION

step 1 : decrement ecx
step 2 : check if ecx > 0

↳ LOOPE / LOOPZ

i) decrements ecx
(ii) if ecx > 0 AND if zero flag = 1
Then JMP dest.

eg:

```
MOV  CX, 10
L1 : SUB CX, 1
     LOOPZ L1
```

} loop will not work; since zero flag will not be set.

↳ LOOPNZ / LOOPNE

↳ checks if ecx > 0 AND ZF = 0
then it will loop / JMP to destination

Q: Take user input of character
↳ output if the entered is a vowel, consonant OR non char

Chapter 7: INTEGER ARITHMETIC

↳ In order to Encrypt – Decrypt data → must make sure you dont loose any data

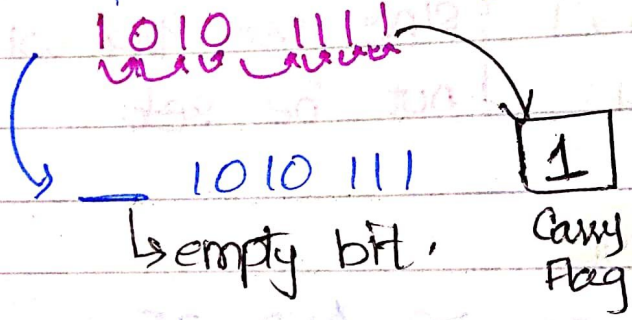
↳ SHIFT / ROTATE INSTRUCTIONS :

↳ to shift any number of bits / data

* Count → No. of shifts to perform
* Direction → Right or left?

↳ shift & rotate changes OF/CF

eg Shift 1 place to right



// to overcome this empty MSB. we have :

→ Logical shifting

↳ stores zero @ the empty bit place

→ Arithmatic shifting

↳ stores the same bit that was present before the shift

↳ In this ex. MSB remains 1.

SHL INSTRUCTION

Operand types

performs a logical left shift.
→ MSB → carry flag
→ LSB → stored zero

SHL	reg, imm	→ max 8 bits
SHL	mem, imm	no concept of same size of operands.
SHL	reg, CL	
SHL	mem, CL	

eg.

```
MOV AL, 0F0h
SHL AL, 5
```

0000 1111 0000
1111 0000 0000
↳ will eventually; after 5 shifts turns to zero

NOTE SHL reg, reg
↓
destination, no_of_iterations

SHR Instruction

↳ same as SHL → just direction changed
highest (MSB) → replaced w zero
LSB → copied @ CF

eg SHR 1111 0000, 2

#1 0111 1000 ; CF=0
#2 0011 1100 ; CF=0

carry flag holds most recently kicked out bit HELL w previous shift

00 10
01 10

↳ SAL (shift Arithmatic Left)

eg ↳ works the same as the SHL

↳ so irl SAR is the only Arithmatic shift

↳ SAR (shift Arithmatic Right)

eg MOV AL, 1001 1000b

Qs SAR AL, 3.

3)

#1 1100 1100 ; CF=0

#2 1110 0110 ; CF=1

#3 1111 0011 ; CF=0

↳ this is an Arithmatic shift to Right where:

4) Arithmatic

CF = LSB

MSB = last MSB (retains the sign Bit)

↓ shifts

5)

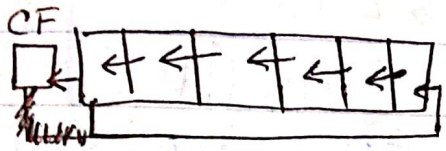
ROTATE INSTRUCTIONS

↳ **ROL** Rotate left

↳ Bit wise rotation towards left

↳ not losing any bit of data since after one rotation MSB = LSB.

↳ copy of LSB is present in carry flag



↳ eg MOV AX, 1234, 4321

↳ do the work w rol inst

```
ROL AL,4
ROL AH,4
Rol AX,8
```

```
        3   4
R1 ↳ 0011  0100
R2 ↳ 0110  1000
R3 ↳ 1101  0000
R4 ↳ 1010  0001
     0100  0011
```

After 4 Rol of AL

AL = 4, 3 (0100 0011)

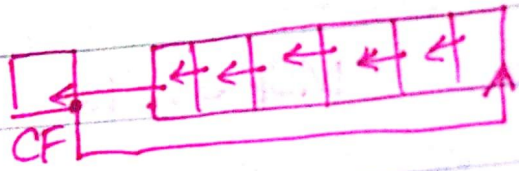
ROR INSTRUCTION

↳ works in the same way as rol but does it in opp direction

↳ RCL (rotate carry left)

eg ↳ shifts each bit to the left, copies carry flag to the LSB and THEN copies MSB to the CF

NO! @ the same time

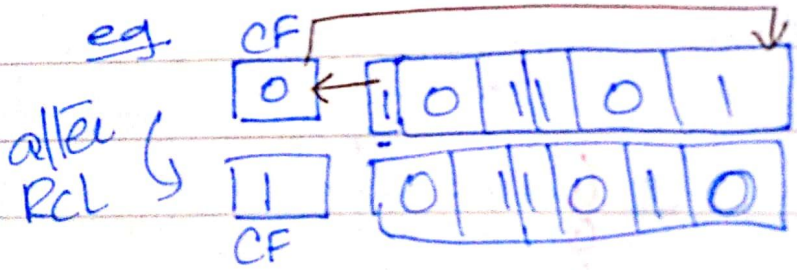


Qs

Prev value of CF → LSB
MSB → new value of carryFlag } Atomic operation

3) ↳ Simultaneously copies CF to LSB & puts MSB into CF

eg



4)

↳ RCR

5) Does the same shit as rcl only in opposite direction (towards right)

* SHLD / SHRD → source always a register

↳ reg16, reg16, CL/imm8
mem16, reg16, CL/imm8
reg32, reg32, CL/imm8
mem32, reg32, CL/imm8

↳ SHLD (shift left double)

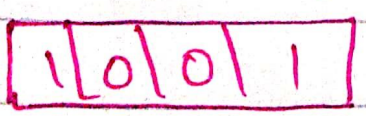
↳ shifts towards the left & MSB keeps on Ammending the carry flag.
The empty bits @ MSB of destination are filled by MSB of source.

eg

```
MOV AX, 1234
MOV CX, 9797
SHLD CX, AX, 4.
```

1001 0111 1001 0111
0111 1001 0111 0001 → CF = 1
7 97 1

D , S , Count



, 2

after shift →

0 1 1 1 0 1 1 1

Q4 ↳SHRD (right double)

↳ works the same as shld only destinations M.S.B are filled by the LSB of source

Q5 ENCRYPTION & DECRYPTION

Encrypt	to decrypt
3 ROL Ax,5	NOT Ax
Add Ax,3	dec Ax
ROL Ax,7	ror Ax,7
Inc Ax	Sub Ax,3
NOT Ax	ROR Ax,5

4

5

8421

Cx = 1010 0000 1010 0000

Q)

```
Mov Ax, 7907h
Mov Cx, A0A0h
SHRD Ax, Cx, 7
```

1st) 0111 1001 0000 0111
CF 2nd) 0 011 1100 1000 0011 → CF=1

3rd) 0 001 1110 0100 0001 CF=1

4th) 0 000 1111 0010 0000 CF=1

5th) 0 000 0111 1001 0000 CF=0

6th) 1 000 0011 1100 1000 CF=0

7th) 0 100 0001 1110 0100 CF=0

0 100 0000 1111 0010 CF=0

⇒ 4 0 F 2

MULTIPLICATION & DIVISION INSTRUCTIONS

↳ unsigned → MUL / DIV
↳ signed → IMUL / IDIV

↳ IDIV & IMUL keeps in mind the value of the MSB (signed bit)

// MUL takes one operand

Multiplicand	Multiplier	Product
AL	reg/Imm8	AX
AX	reg/Imm16	~~EAX~~ DX, AX
EAX	reg/Imm32	EDX, EAX

↳ upper 32 = EDX
~~Lower 16~~ = Eax
lower 32

// After Multiplication, CF/OF = 0
if upper half of the result is zero;
1 otherwise
↳ meaning result is exceeding size of operand

SIGNED MULTIPLICATION ⇒ IMUL

↳ 3 - types of IMul.

- IMul A → 1 operand
- IMul A, B → 2 operand
- IMUL A, B, C → 3 operand

* 1 operand → Same as Mul, A.

* 2 operand

eg IMul Dx, Cx (Dx Multiplied by Cx)

↳ used when we know my result is not supposed to exceed 16 bits

↳ Any result > than sizeof (operand) will loose data.

↳ any bits > than 16 will be truncated

eg DX, -8
CX, -8
] IMul DX, CX
↳ DX = DX * CX

D 7
ME 5

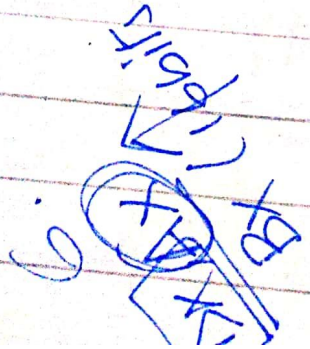
DIV

* 3 Operands IMUL
↳ IMUL BX, CX, 8

then BX = CX * 8.

↳ In 3 operand Imul, we provide the destination ourselves.

↳ TASK → When does Carry Flag change w Imul



D+1

DIV INSTRUCTION

Div reg/mem → DIVISOR

↳ DIVIDENT is implicitly taken

↳ if divisor is 8 x bits ; divident will be 2x bits

2 Results; quotient & remainder

		RESULTS	
DIVIDENT	DIVISOR	QUOTIENT	REMAINDER
AX	mem/reg8	AL	AH
DX, AX	mem/reg16	AX	DX
EDX, EAX	mem/reg32	EAX	EDX

eg
```
mov ax, 0083h   ; dividend
mov bl, 2       ; divisor
div bl          ; AL=41h
                  AH=01h
```

SIGNED DIVISION

↳ signed operands

↳ Sign extension (MOVSX) necessary

eg
```
MOV DX, 0
MOV AX, -1
MOV BX, 1
IDIV BX
```

↳ to preserve the sign

DX AX / BX

MOVSX DX, AX

Note: MOVSX only converts MSB's into signed bit

INSTRUCTIONS FOR SIGNED INSTRUCTIONS

1. CBW (convert byte to word)
   ↳ CBW; MOVSX AX, AL
   Extends MSB of AL into AX
2. CWD (convert word to double)
   ↳ CWD; MOVSX DX, AX
   Extends MSB of AX to DX
3. CDQ (convert double to Qword)
   ↳ CDQ; MOVSX EDX, EAX
   Extends MSB of eax into EDX

7.4 EXTENDED ADDITION & SUBTRACTION

↳ ADC (Add with carry)

this Instrc adds both a source operand and the contents of the carry flag to the destination

ADC dest, source
; dest = dest + source + CF
@ ↓ LSB

CF → Purani wali Inst ki carry

WJ11010101

SBB Subtract _____ borrow

SBB dest, source
; dest = (dest - source) - CF

TASK: MID → Qs 2(a) even odd Number.
↳ array of 1000 elements (random Pos)
↳ ~~calc~~ move odd into 2nd array

→ MEMORY SEGMANTS

- stack → local Data → every function gets its own stack frame
  - → Ret Addesses
  - → Arguments
  - → Register values
- Data
- Code

- Every called procedure gets its own stack frame

No of stack frames = No of called Procedures + 1 (of the main func)

- Arguments → values passed to func @ runtime

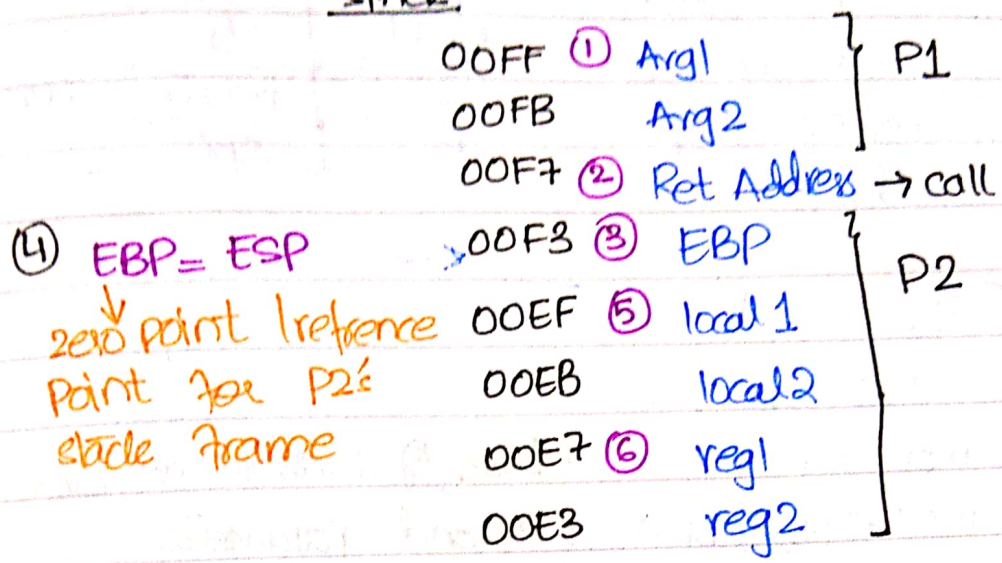
- → EBP used as zero point

HOW STACK FRAME IS CREATED?

1. Passed arguments are pushed on stack
2. The Subroutines is called, causing the subroutine return addresses to be pushed on the stack
3. As the subrou begins to execute; EBP is pushed on the stack → EBP = Zero Point = ESP
4. local variables
5. used Registers

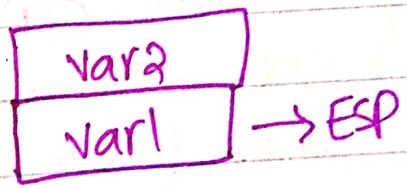
ADVANCED PROCEDURES

STACK



To use Registe stack for Passing Parameters

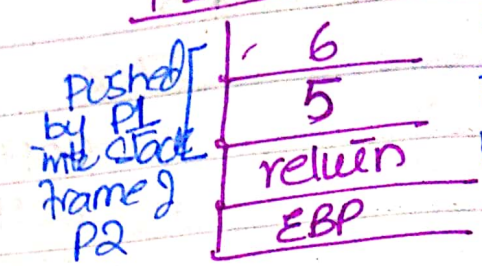
```
.data
    var1
.code
    Push offset var1      Pass by ref
    Call P2.
```




```
.data          → Pass by val
    var1
    var2
.code
    Push var1
    Push var2
    Call P2
```

P2 :
Push EBP
mov ebp, esp
mov eax, [ebp+
add eax,

P2 stack.



P2:
Push ebp
* mov ebp, esp ; base of stack frame // zero point? ✓
mov eax, [ebp+12] ; second Parameter
mov eax, [ebp+8] ; first Parameter
Pop ebp

// To clean stack after calling P2
bring esp back @ start of stack frame P2
call P2
add ebp, 8
↳ ebp Points @ start of stack frame;
next line over write data

* we moved esp into ebp to make ebp the refrence / zero point so we can easily access Arguments by going up the stack (add 4) or local variables by going down the stack (sub 4)

Eg

```
main PROC
    Call Example1
Exit
Main ENP

Example1 PROC
    Push6
    Push5
    call AddTwo
    ret
Example1 ENPP
```

FF12	ret		
FF0D	EBP	main	
FF09	ret		Example1
FF05	EBP		
FF01	6		
FE0B	5	Add Two	
FE09	ret		
EE05	EBP		

// Every time a function is called; EBP → Push
ur EBP == ESP

→ CALLING CONVENTIONS [C, STD]

The C calling convention solves the problem of cleaning up the runtime stack in a simple way

↳ calling convention reclaims the space occupied by the Arguments bagi POP Ebp ur extra work → manually

↳ hence the add esp, & → size of Arguments is done @ back end

2. STD calling Convention
↳ returns an integer value to the calling procedure (P1)
↳ an integer parameter to the RET instruction, which in turn adds integer to ESP after returning to the calling procedure
↳ The integer = size of Arguments

(calling) P1 → adds int to ESP
(called) P2 → returns int

// IMP
// Popping out EBP makes sure we donnot loose the zero reference point of the prev call

eg.

	FF09	ret
EBP=ESP=FF05	FF05	EBP
	FF01	6
	FE0D	5
	FE09	ret
EBP = FE05	FE05	EBP → FF05

↳ when popped EBP again FF05

SAVING & RESTORING REGISTERS

↳ if registers are pushed into stack REGISTERS SHOULD BE POPED OUT FIRST & IS A MUST

then POP out ebp then return address & the clean the Arguments

// LOCAL VARIABLES

```
MySub PROC
  push ebp
  mov ebp, esp.
  sub esp, 8        // saving space for local variables

  mov Dword Ptr [ebp-4], 10     ; x
  mov Dword ptr [ebp -8], 80    ; y
  mov esp, ebp                  ; remove locals from stack
  pop ebp
  ret
```

ret address
EBP → EBP
10(x) → EBP-4
20(y) EBP-8

ENTER & LEAVE INSTRUCTIONS

↳ pushes EBP on the stack
↳ sets EBP to base of stack frame
↳ reserves space for local variables.

ENTER numbytes, nestlvl

Entee does all what we do manually.

Proves that two/more Procedures can have same nesting level.

P1 NL (1)
call P2 (2)
call P3
call P4 (2)

// Leave (no arguments)
↳ reverts whatever is done by Entee

LOCAL DIRECTIVE

P1 PROC
Local x : Byte , Y[10] : word.
↳ way to make local variables
↳ can only be used w/in P1
↳ if used, Local be used Jost after the P1 Proc line.

TASK // lea Instr wali slide using Local directive

Lea → load effective Address
// used to get address of anything @ runtime
lea esi, [ebp-30]

RECURSION

↳ for base case cmp w any data variable ; eg if 4 recursions
→ need to cmp ecx with 0
→ more memory consumption.

Calculating a Factorial

:
:
:
!

INVOKE DIRECTIVE

↳ only available in 32-bit
using INVOKE is reduced to a single line in which the arguments are listed in reverse order (assuming STDCALL is in effect)

eg
```
Push TYPE array
Push length of array
Push OFFSET
```

ADDR → Same As ~~Addres~~ offset

// By reference

→ INVOKE Sub2, ADDR byteVal

sends offset of byteVal to sub2

// By value

→ INVOKE Sub1, byteVal, wordVal

sends value of val in wordval to sub1

* // can also send Constants

PROC DIRECTIVE

↳ to define function Parameters

// P1 PROC, x: BYTE, W: PTR word

↳ this is a signature of this procedure

// when invoking this; first passing will be a value of byte-size

// the second passing is an addres.

PROTOTYPE

* To define a subroutine / procedure prototype

P1 Proto x: Byte, W PTR byte

TASK 11 Prev task using USES

USES OPERATOR

```
ArraySum Proc USES esi ecx
    mov eax,0
L1:
    ⋮
    LOOPL1
ret
ArraySum EndP
```

⟹ same as

```
ArraySum Proc
Push esi
Push ecx
mov eax,0
L1:
    ⋮
    LOOPL1
POP ecx
POP esi
ArraySum.
```

Uses is an operator which automatically Pushes & pops registers in a procedure.

// Argument size mismatch

→ making sure that the argument sent is the same size as the parameter that recieves it

→ Never send constant value to a pointer parameter

In 32-bit mode, if we use parameters of a diff size; it prevents ESP from being aligned on a double word boundary

↳ here a page fault may occur & runtime Performance may be degraded. // Memory is not Propely aligned.

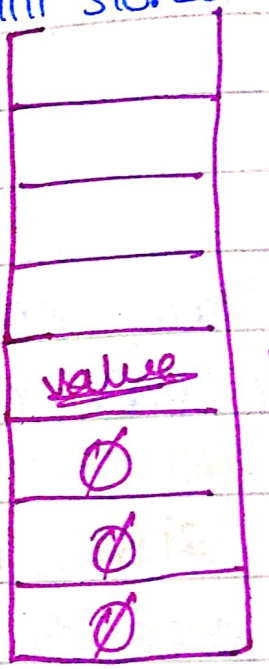
↳ Same size Arg = Aligned Val

Note.
If a 64-bit Argument is sent → it is stored in 2 hissay (parts) High level & Low level.

NON DOUBLE LOCAL VARIABLES

if for example we use a x:byte as a local variable, it it still stored on 4 bytes on the stack

// Every local variable data is given the size of your architecture
// if byte word .... etc
~~Any~~ Each are reserved a memory of 4 bytes



Note: These Null bytes are not a part of x:byte → just present here for memory alignment. // Dont confuse w/ casting

Quize Next Week → Chap 6,7 & 8.

CHAPTER 9
STRING & ARRAYS

- string primitive Instructions

↳ MOVSB, MOVSW, MOVSD

↳ these instructions copy data from the memory location pointed to by ESI to the memory location pointed to by EDI

```
.data
    source  Dword
    target  Dword ?
.code
    move  esi, OFFSET source
    mov   edi, offset target
    movsd
```

↳ will copy contents of a [esi] to [edi] → of size 4 bytes bec [sd]

↳ will also inc 4 int esi & edi since command is movsd

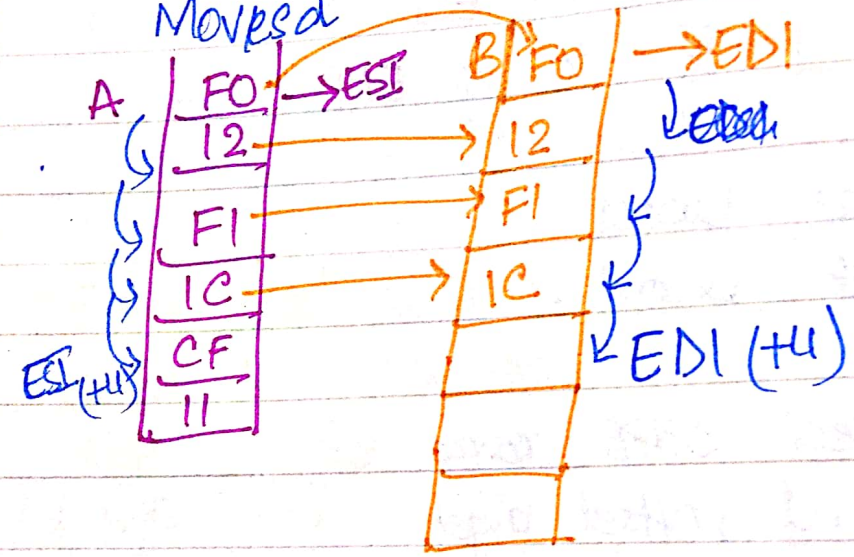
DF // Direction flag → 0 → forward
→ 1 → backwards

eg

A word F2F0 1CF1 11CFh
B word 3 DUP(?)

mov esi, offset A
mov edi, offset B

Movsd



// To copy 1000 word elements you might use movsw a 1000 times in a loop

REPEAT INSTRUCTION

X word ... 1000 → esi
Y word .... 1000 → edi

```
CX ← 1000
L1 :
    movsw
loop L1
```

OR

```
CX ← 1000
rep movsw
```

Both work the same way only rep instruction only works on string primitives and rep decrements cx as well.

// keep in mind that when processing word array use movsw only if you use movsb → then MSB will be saved as LSB & vice versa due to little Endian.

COMPARE INSTRUCTION

→ CMPSB, CMPSW, SMPSD.

↳ Compares a memory operand pointed to by esi with memory operand pointed to by EDI

esi →	12
	99
	00
	10

edi	76
	12
	00
	02

cmpsb → means 12, 76.

cmpsw → 9912, 1276.

cmpsd → 10009912, 02001276.

↳ compares a whole doubleword from esI to a doubleword from EDI

REPE → repeat if cx != 0 & zeroflag is ON

Example Comparing 2 strings

source M A R T I N _ _ _
↑ esi

Dest M A R T I N E Z
↑ edi

will compare till reaches the first space bec of repe.
repe will not repeat when zero flag is off that is when 'E' & space is not equal

↳ SCASB, SCASW, SCASD.

↳ scasb compares a value in al w address by edi
↳ scasw → AX by EDI
↳ scasd → EAX by EDI

// Program to Search substring in String

```
.data
    alpha "ABCDEFGH", 0

    mov edi, offset alpha ; EDI points to string
    mov al, 'F'           ; search F
    mov ecx length of alpha ;
    cld                   ; direction Foward
    repne scasb           ; repeat while ! equal
    jnz quit              ; quit if letter not found
    dec edi               ; found: edi since
                          will be pointing to G
                          dec edi will point
                          it to F again
```

CISC → Complex Instruction set
RISC → reduced

STOSB, STOSW, STOSD // STORING.

→ Stores the contents of Accumulator to the memory by EDI

↳ Source = AL/AX/EAX

dest = EDI // Increments EDI

```
.data
Count = 100
```

// check from slides

LODSB, LODSW, LODSD. // LOADING

loads a byte/word/DW from memory at ESI to AL/AX or EAX

Source = ESI // INC ESI

Dest = AL/AX/EAX

// Good code of Array Mul check from slides

TWO DIMENSIONAL ARRAY

- Two methods of arranging rows & columns are: Row-Major
  Column Major

Array =

10	20	30	40	50
60	70	80	90	A0
B0	C0	D0	E0	F0

Row Major

10 20 30 40 50 60 70 80 ....

Column Major

10 60 B0 20 70 C0 30 80 D0

// No. & type of elements Must be same in each row

Base-Index Operands

[base + index]

// Pointer needed for each row [pointing]
esI used to access elements/ columns in each row

Arr [ebx] [esI]

Declaration

Table B BYTE 10, 20, 30, 40, 50

RowSize = ($ - Table B)

Ex Sum of column 1 in eax. Array[5][3]

```
mov ebx, offset Array.
mov ecx, 3
L1:
    add al, [ebx + column Index]
    add ebx, row size
Loop l
```

↓ Accessing element in Array

MACHINE LANGUAGE

INSTRUCTION FORMATS

Instruction Prefix	Opcode	Mod/M	SIB	Add Displacem	Imm data
1	1-3	1	1	1-4	1-4

- Instructions are stored in little endian order, so the prefix byte is located @ instructions starting address.

- Every Instruction has an opcode, remaining fields are optional

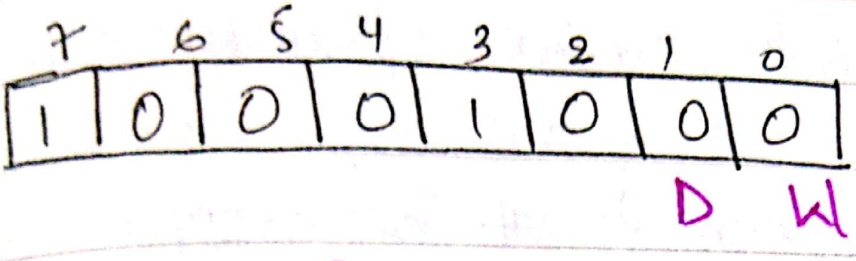
mnemonic → mask over opcode

eg opcode of ADD = 00

- Most Instruction are of 2-3 Bytes.

OPCODE

eg opcode for mov = 88 → not necessarily hence



D bit — Direction

// comming from reg D=0
going to reg
from I reg } = D=1

W bit → Width

if operands are == 8bits ; W=0

if operands are > 8bits ; W=1

// op code 88 when D=0 & size>8.

MOV Ax, var1

INSTRUCTION FORMAT

1. Instruction prefix byte
   for extended operands > 16 bits

2. OP code byte
   necessary numeric # for every Instruction

OPCODE BYTE — Instruction Code (bit # 2-7), Direction (bit #1), Size (Bit0)

Direction (Dbit)
- if data Comming FROM Reg = 0
- if data Comming TO Reg = 1

SIZE (W bit)
- if operands of 8 bits 0
- if operands > 8 bits 1

eg.

MOV. Cx, Var1 MOV=88.

↳ 10001 0 1 1
D W

3. MOD R/M Byte

MOD (6-7 bits) Reg# (3-5) Mem# (0-2)

MOD – Addressing mode #?.

		Binary
①	MOV Ax, Bx → register mode	11
②	MOV Var 1, Bx → Mem w/o displacement	00
③	MOV [ESI+16h], → mem w 8bit "	01
④	MOV [ESI+1670h], mem w/ 16bit "	10

Register # (source ka if register mode)

each reg has its own; eg. aL — 000
cL — 001
dL — 010
bL — 011

Memory # (calculated from Table) → but if reg mod then of dest

eg. MOV Ax, Bx.

11	011	000
mode	Reg# source	Reg# dest

eg. ADD Ax, Cx

0000 0001 opcode byte
11 001 000 RM byte.

INSTRUCTION ENCODING. (ADDRESS DISPLACEMENT)

eg MOV [ESI + 17CFh] x DL

DW

1000 1000 10 010 100

opcode Mod R/M

append offset displacement

88 94 DF17

Displacement in form of little endian

if same instruction but offset 17h. (8bit)

MOV [ESI + 17h] x DL

1000 1000 01 010 100 offset

1000 1000 01 010 100 17

8 8 9 4 17

8. MOV [EDI+1709F070], EDX

1000 1001, 10 010 101, 70 F0 09 17,
opcode, MOD R/M, Offset 32. (Add displacement)

89 95 70 F0 09 17

→ SINGLE BYTE INSTRUCTION

eg

CBW	98	already Pre-defined/cakt
LODSB	AC	
INC DX	42.	

* register increments are optimized for code size 4

↳ SINGLE IMMEDIATE DATA

→ when the only operand of an instruction is an immediate data then usea machine code → OP code followed by data

eg Push 57F0.
Ret 8

eg Ret 8.
C2 0800
for Ret

Always Appended.

↳ SINGLE OPERAND (Register) Instructions

eg ⇒ PUSH DX

↳ In this case ADD reg# to OP code by

P

Push DX
= 52 + 2
= 54

If EDX, Instruction Prefix scn.

*

↳ Register Immediate Instructions

Immediate operands are appended to instructions in little Endian order.

//

→ The encoding format of a MOV imm that moves an immediate value to a register

B8+ rw dw

↳ where opcode by value is B8+rw, indicating that a register is ADDED to B8

↳ dw is the immediate word operand

ADD CX, 190FH.

byte

0000 0000 + 001, 0F19,

reg added — operand Appended

↳ SINGLE OPERAND (MEMORY)

Assembly

INC BYTE PTR [BX][SI]

IDIV WORD PTR [DI] + 1A2Bh. ⟶ ASSEMBLY LANGUAGE

opcode	mod-reg-r/m	Machine language
F7	10 111 101	F7 BD 2B 1A

opcode extended bits for IDIV (No reg)

// REG bits of R/M byte holds OP-code extension in these instructions

CISC : Reg – Mem
RISC : at max and min 4 byte encodig]?
↳ alligns data → MIPS?

x ——— x ——— x

chapter 13 : High Level Language Inteface

↳ where convinience 4 developement is more Imp than time a size of code → HLL is used.

- model Directive

↳ • model Memory Model, Calling Convent (STC, C), Stack Distance

Memory Models:

Explains code 4 data segments in code

- Tiny → Code, data, stack @ same place
- Small → 1 code 4 1 data segment
  if ECS == EDS → same segment
- Medin → mul code, 1 data segment
- Compact → 1 code, mul data segment
- large → mul code, mul data segment
- Hoge → " 4 greater size

- Flat → default (protected mode)

CISC : Reg – Mem
RISC : at max and min 4 byte encodig ?
↳ alligns data → MIPS ?

x ——— x ——— x

chapter 13 : High Level Language Inteface

↳ where convinience & developemeit is more Imp than time or size of code → HLL is used.

- model Directive

↳ • model Memory Model, Calling Convent (STC, C), Stack Distance

Memory Models:

Explains code & data segments in code

- Tiny → Code, data, stack @ same place
- Small → 1 code & 1 data segment
  if ECS = EDS → same segment
- Medium → mul code, 1 data segment
- Compact → 1 code, mul data segment
- large → mul code, mul data segment
- Huge → " & greater size
- Flat → default (protected mode)

• STACK DISTANCE

near stack: Physically stack is present in data segment (EDS == ESS)

far stack: A diff / separate segment for stack

// Near Call → calling & called proc are on same segment

// far Call → calling & called proc are on different segments

↳ default depends on our model.
↓
memory

// EES, EFS → used to check extra segments

THE __asm Directive // Inline assembly.

To write assembly in C++

```
__asm
{
   code block.
   //data to be made in HLL part.
}
```

OR

__asm statement

This has its own features & limitations

Limitations

→ no data definition directives
→ cannot use anything other than PTR
→ cannot refrence segments w name (eg .data)
→ Cannot make assumptions about registers since they are global to the system.

: In slides.

// operating System Locks resources till one is using it.

Homework:

- File Encryption Example
- SIB

MIPS ARCHITECTURE & Assembly of MIPS.

RISC :
- MIPS
- Load store

ISC
↳ Instruction set architecture
everything a machine language programmer needs to know in order to program a computer

RISC → faster → ; Simple instructions; addressing v easy ; Pipelining

PIPELINING

eg ✓ MOV Ax, 12
✓ MOV Bx, 13
MOV Cx, Ax
✓ MOV Dx, 12
ADD Dx, Cx

Pipelining allows to execute INDEPENDENT Instruction w/o being dependent on anything else. (Increase Processor Speed/time)

This is called Parallel processing.

IN Software (wise) This is called Parallelism
↳ INSTRUCTION LEVEL PARALLELISM.

↳ DATA LEVEL PARALLELISM

```
Mov Val, 17
Mov Val[2], 19
Mov Ax, Val[4]
Mov Dx, Ax
Mov Val[4], Ax
```

✓ If You execute these Instructions Parallelly then we can access the memory @ the same time → data level Parallelism

We can also execute 2 Instructions Simultaneously hence → Instruc L. Parallel

RISC → every Instr 4 bytes → easier.

// Homework

mobile type devices w/ CISC? CISC / RISC → state of practices (practical systems in market)

↳ TASK LEVEL Parallelism

different types of tasks ~~to~~ running simultaneously // processor switches v easily & v fast

eg vlc, google ~~acts~~ together

// NOTE : CISC → mushkil bec of complexity

// has a lil pipelining

// Better for memory (shayad ?) ✓ yes

//ILP doesn't only mean Instruction executing @ the same time ; also means to not be dependent on any Instruction for another (!waiting)

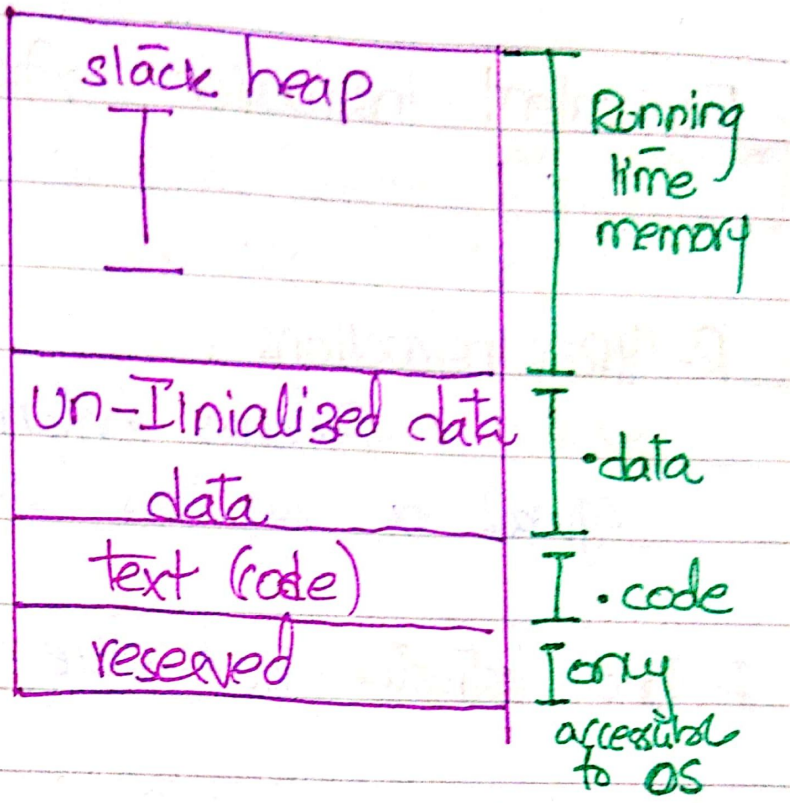
LOAD-store
↳ ALU not allowed @ on memory

↳ check Memory data types
eg Word in CISC → Halfwords in RISC

↳ 32 4-byte Registers
- all accessible → except for zero register
- zero-register always has 0

↳ to Access Register
R1 | $1 → Register 1
R0 → R31 → 32 registers present in the register file (log 32 registers)
- check register usage table

MEMORY LAYOUT



Homework

// Addressing Modes in MIPS (CISC)

Instruction Set
- 21 arithmetic
- 8 logical
- 8 bit manipulation
- 12 comparision
- 25 branch/jump
- 15 load Instructions
- 10 store Instructions

(check Table)

LOAD & STORES.

LB → load byte
LH → load halfword
SB → store byte
SH → store halfword

eg

LB $, memory

SB $, memory

Equivalent Instruction → from Slides

R-Type Instructions : Performs Arithmetic & logical on registers (shifts)

* I-Type Instruction : load-store & Immediate

J-Type Instruction : used for Jumps & branchy

// Register Operands take less than 1 cycle to execute

// Declaration

.data
x : .byte 0x12 [x: .byte 1,2,3 → Array]

str : .asci "Hello world \n"

null character

* SHI, SLO → Results of mul & div ; depends on size of results

Registers

$s0 = R[16]

name of register ↓ (name reg file no.

16th register

MISCELLANEOUS REGISTERS

$PC	Program Counter [every 4 byte step = 1 offset]
$status or $psw	status Register (flags)
$cause	Exception cause register
$hi $lo	Imul Idiv results storage

HOMEWORK

#3 Flags in MIPS

IN DIVIDE : Lo = Result (Quotient)

Hi = Remainder

ARITHMATIC & LOGICAL

add S1, S2, S3 S1 = S2 + S3

OR S1, S2, S3 S1 = S2 OR S3

DATA TRANSFER

Lw $S1, 100($S2) ⇒ $S1 = Memory[$S2 + 100]

Sw $S1, 100($S2) ⇒ Memory[$S2+100] = $S1

Branch - alter program flow

beq $S1, S2, 25

if($S1 == $S2)

PC = PC + 4 + 4*25

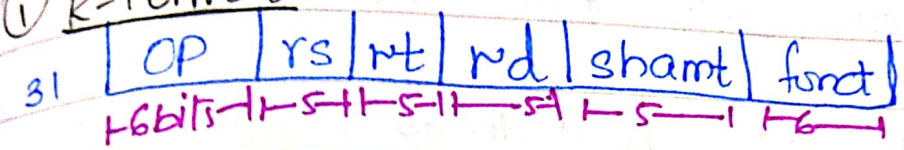
should use label here

MIPS INSTRUCTIONS

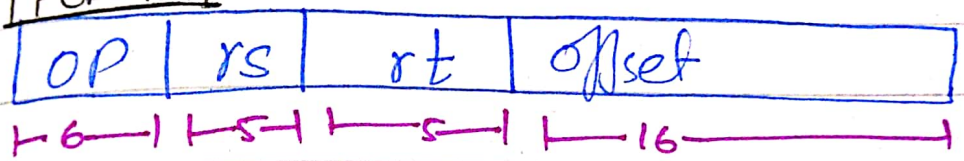
similarities in formats ease

Different formats for different purposes.

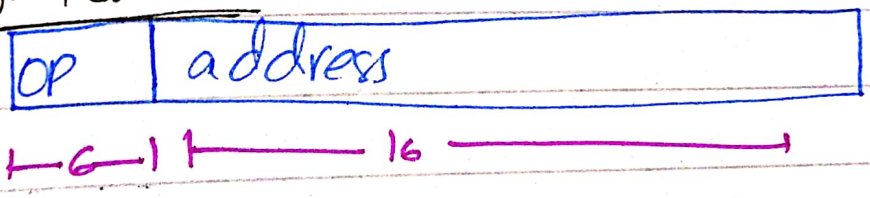
① R-Format



② I FORMAT



③ J- Format



rs — Source Reg #1

rt — Source Reg #2

rd — Destination Register

shamt - Shift amount — ?

funct — function # ? provided

eg

add $S0, $S1, $S2 (registers 16, 17, 18)

op	rs	rt	rd	shift	funct
0	17	18	16	0	32
000000	10001	10010	10000	0000	100000

↳ machine code generated

// command ka pata funct # se chalta hai

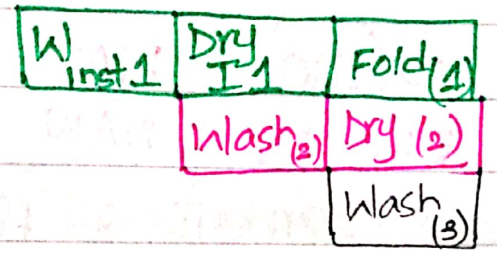
5 STAGES OF EXECUTING INSTRUCTIONS (MIPS)

1) Fetch
2) Decode
3) ALU / effective Addren
4) loading Memory (Reading)
5) Inseting result back

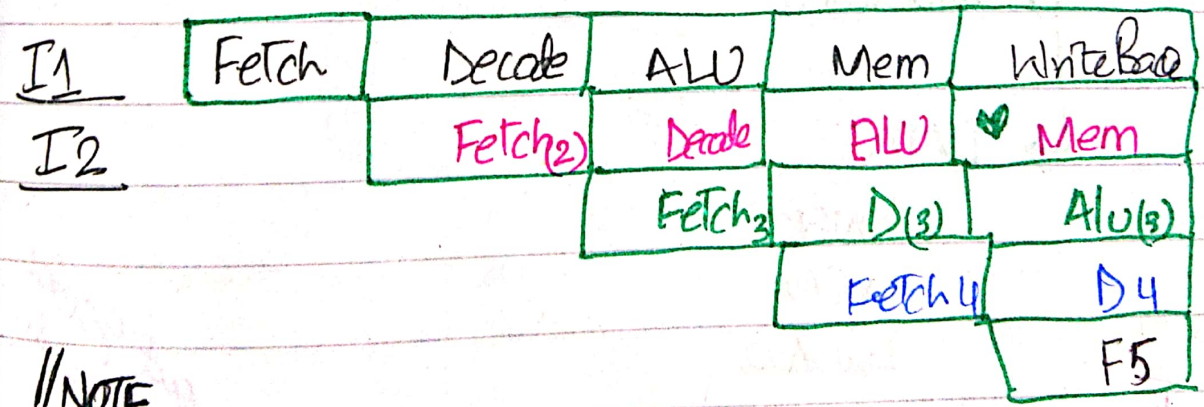
// ALU's take onli 4 stages (No read memory)
// load / store is expensive in respect of steps (5 steps)

PIPELINING

→ Traditional example of washing, Drying & Folding clothes



In Example of Mul Instruction in Computer



// NOTE
if Instruction 1 = load = 5 steps && Ins 2 = ALU = 4 steps then would be WB Too ; And 2 Instruction will be using WB @ the same time = STRUCTURAL HAZARD

eg.
1. LW $1, 100($7)
2. Add $11, $7, $10
3. Sob $2, $1, $3

DATA HAZARD

⇒ Instruction 3 is trying to read smth that isn't already written by 1.
3 is reading RAW read After write

⇒ Blw 2 & 1 there is no hazard since both are ONLY Reading

4. LW $7, 50($30) WAR w(2)
5. Add $2, $4, $7 WAW w(3)

// Never keep dependent in pipeline together

HAZARDS

- structural
- DATA
  - WAR
  - RAW
  - WAW

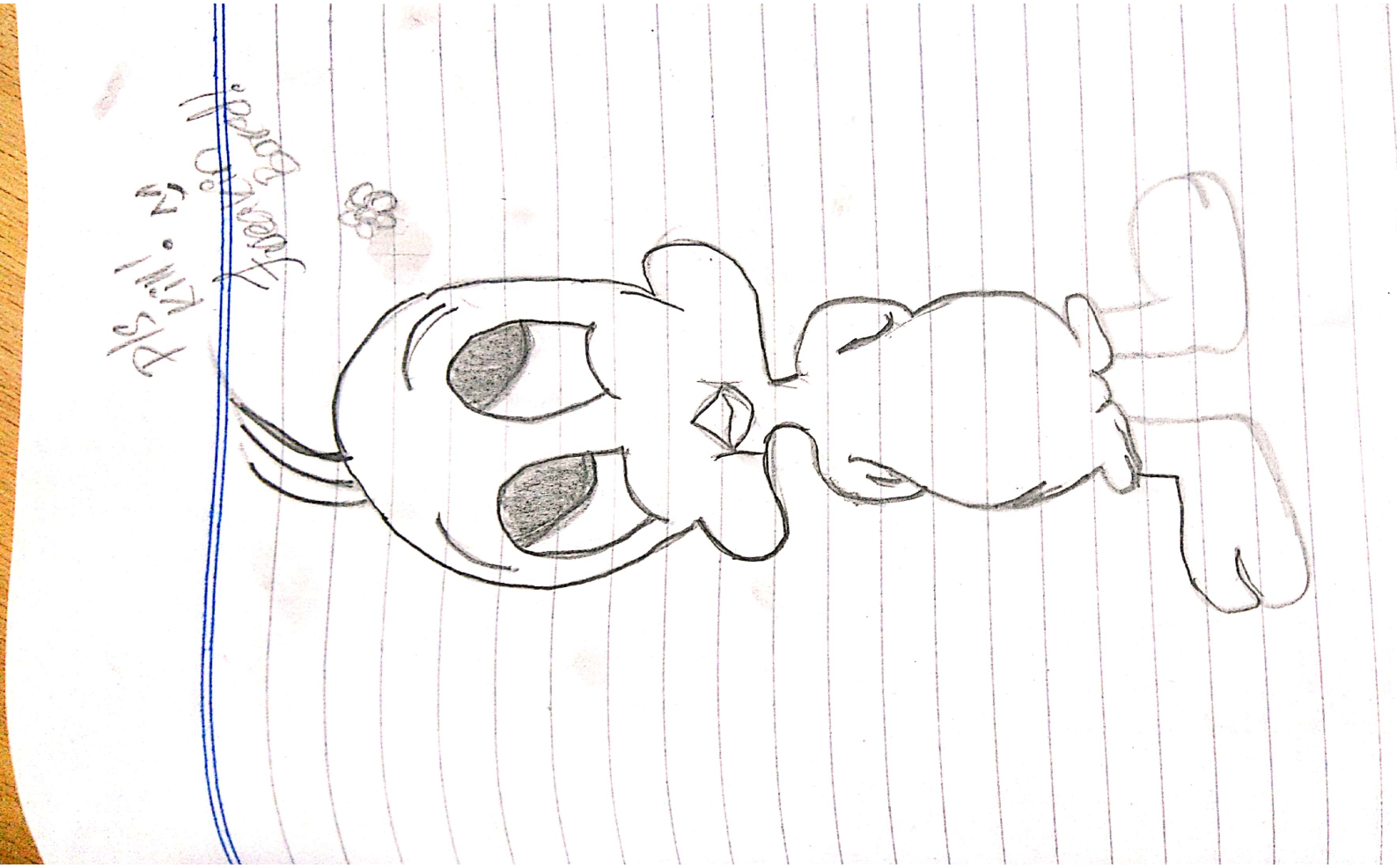
// one of the solutions ⇒ changing order (out of order execution)
↳ for structural hazards use a stall

eg. ~~ALOAD~~

LOAD	F	D	ALU	Mem	WB	
ALU		F	D	ALU	—	WB

↓ Stall.

// CISC is hard to handle w/ Pipeline



CACHE.

↳ Principle of locality ; most programs donnot use every Part of Program (?)

↳ block minimum bytes(?) of data that can be present Inot present on the cache'

↳ Mapping :
data from memory ⟶ cache' is called mapping

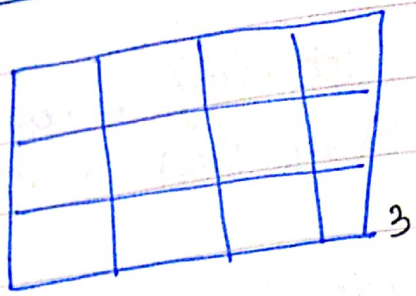
↳ Replacement Algorithm :
Algorithms to replace a block of memory in cache'
- least recently used (LRU)
- First in First out
- least frequently used (LFU)
- Random

Cache' organization
↳ physical address → actual addren @ memory
↳ logical addren → OS assigned addren also known as virtual addren
(due to mullipe Rams)

Memory location that we talk about are logical addrenes

In a cache'

// m blocks, called lines



12 bytes

MAPPING Functions

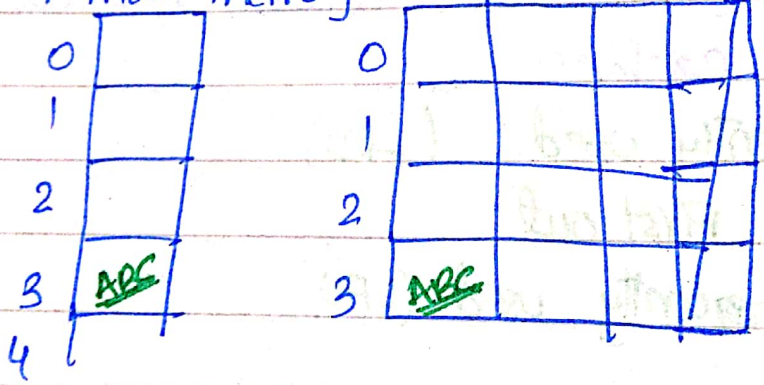
1. Direct Mapping:

$i = j \bmod m$

// maps each block of main memory into only one possible cache line.

eg

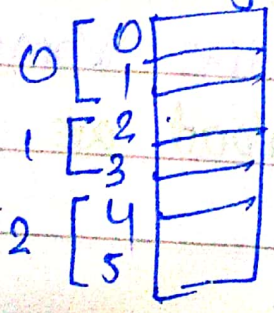
if main memory byte.



If data @ 3rd line in main memory then in cache' ⇒ 3 mod 4 (4lines)

hence @ 3

If memory = d.word. then each block = 4?2



Then j = 0 | 1 | 2

$i = j \bmod m$

i = cache line #

j = main memory BLOCK #

m = number of lines in the cache

* Advantage ⇒ Easier to find!.
* Disadvantage → Same cache' line is reused again and again even if all cache' is empty
  → Replacement Algo also on that same line

Word Addressable Memory:

every 4 byte = 1 block in main memory → logical addresses are given to every 4 bytes and b a single byte.

NOTE // Rem : J is Memory BLOCK number not the byte memory

2. Associative Mapping
↳ Maps each main Memory block into ANY line of the cache

Advantage : No fragments of cache (empty places hone k bawajood replacing)

Disadvantage :
Finding data is very hard; can also mean sequential slowdip

3. Set Associative Mapping → Most efficient
In this case cache consists of a number of sets, each of which consists of a number of lines, the relationships are

"set" Collection of lines

$m = v \times k$
$i = j \bmod v$

i = cache set #
j = main memory block #
v = # of sets
m = number of lines in cache.
k = # of lines in each set

// Returns a set # → and data is randomly mapped onto any line of that set

→ lesser Fragmentation than Direct
→ lesser Distortion of data

Replacement:
If whole set is filled → it uses replacement Algos to get rid of any block. for replacement

"INTERUPTS & RELATED CONCEPTS"

→ Interupts → exception
↳ if user exception then HARDWARE Interupts

Hardware Interupts → highest priority (1)

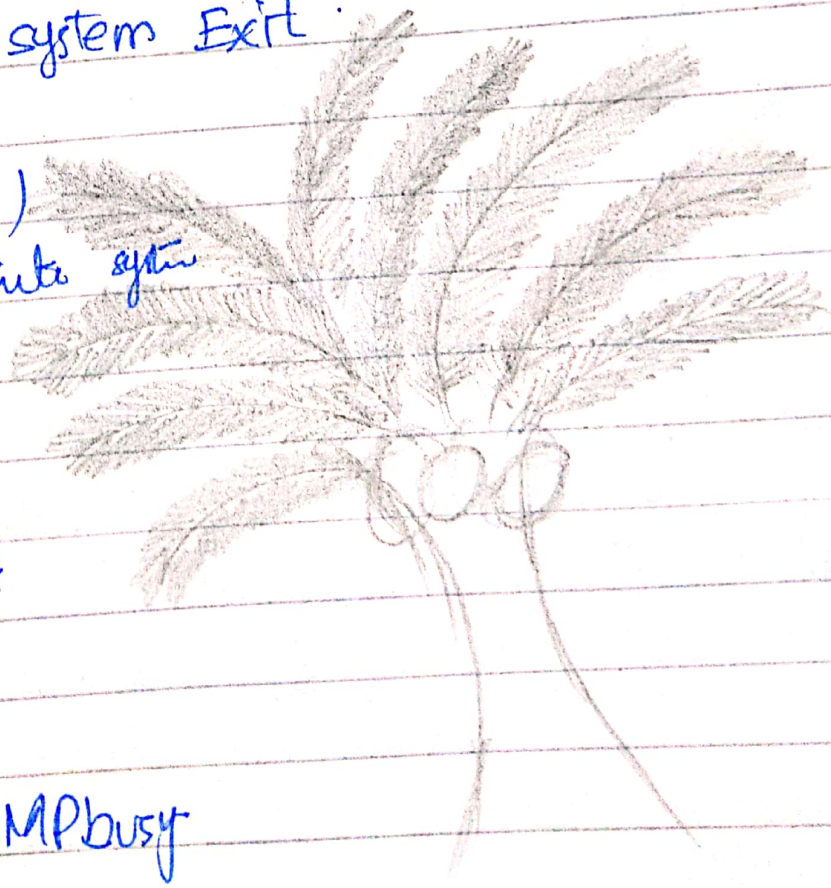
→ Software ⟹ system Exit.

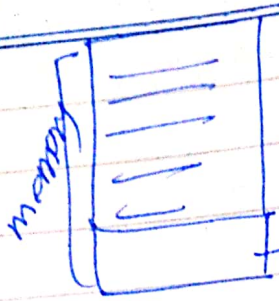
→ system (exceptions)
eg program into system memory

→ control B = 0
↳ No Interupt

Control Bus = 1
↳ Interupt

→ if no responce → MP busy





PCB → process control Block