Microprocessor : 8086, 8088

What is a core?

chip, port no.

interrupt : প্রসেসর কাজের সময় আরেকটি কাজের request.

priority অনুযায়ী কাজ

segment
offset

flag : priority flag

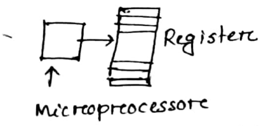
বই: 1. Assembly language programming & organization of IBM PC.

→ Ytha Yu
→ Charles Marcut

2. The Intel microprocessors —
Barry B Brey.

Assembly language? → completely highlevel না। কাছাকাছি। High level language.

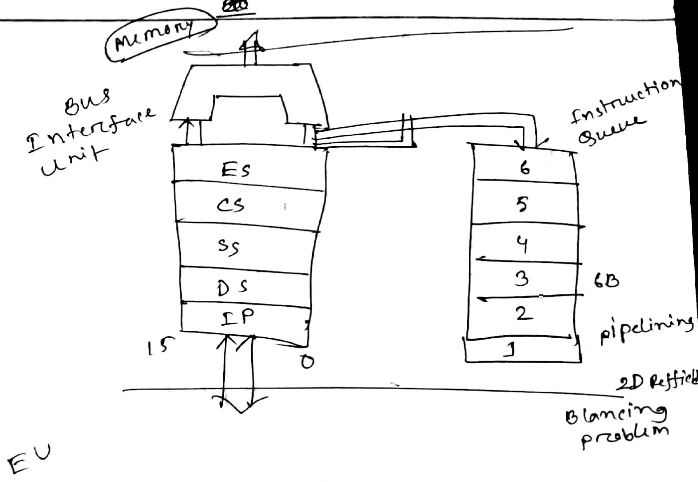
* Assembly তে design করলে cost কম পড়বে।



Microprocessore এর chip এর সাথে embaded থাকে registere & Kash. (দুইটাই memory)

* registere এ কাজ করব।
* Assemblere লাগে assembly language convert করাতে।
* computer registere এর help ছাড়া কিছু করাতে পারে না।
* Registere এর নাম AX, BX, CX, DX এগুলো প্রত্যেকটাই 16 bit
* প্রত্যেকটাকে আবার 8 bit high byte, 8 bit low byte এ ভাগ করতে পারি।

যেমন: AX কে ভাঙলে AH (8 bit high byte)
AL (8 bit low byte)



* Microprocessore এর কিছু generation আছে এগুলো কিছু architecture follow করে।

* 8086 microprocessore নিয়ে কথা বলব।
8088. microprocessore এর difference নিয়ে পরে কথা হবে। numbere দিয়ে express করা tough. তাই বিভিন্ন নাম দিয়ে শুরু করল। corei7 etc.

Bus Interface Unit

* Bus এর কাজ data transmit করা। তিনটি লাইন।
~~তিন ধরনের তথ্য transmit করে।~~
1. address bus (20 bit)
2. data ~~transmit করে~~ bus (16 bit)
3. control bus (16 bit)

e.g. address line এ address পাঠালে data line দিয়ে data আসবে।

* বিভিন্ন microprocessor এ বিভিন্ন size এর bus থাকে।
* bus এর size বাড়ানো হয় না কারণ তাহলে micro processor এর ~~...~~ complexity বেড়ে যায়।

* 2^{20} পরিমাণ memory microprocessor এর সাথে interface করতে পারবে। অর্থাৎ 2^{20} পরিমাণ RAM লাগাতে পারব

* Execution unit

segment register এগুলো:
- ES = Extra segment → extra দরকার হলে তখন এই segment এ
- CS = Code segment → code গুলো রাখা হয়
- DS = Data segment → সকল data রাখা হয়
- SS = Stack segment → stack লাগলে এই segment এ রাখা হয়

Suppose

12340H

code segment → [ ] → 20 bit দেওয়া 5টি Hexadecimal number আছে।

* CS এর ভেতর থাকে base টা

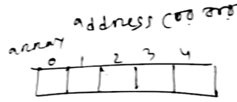
1234 H

segment : offset
1234H : 000H] এভাবে address লেখা হয়

4 ঘর বামে shift করে দিব।

12340H

address বের করতে



a[3]

(0+3)
↳ offset

segment × 10 + offset

IP: Instruction pointer. (16 bit.)
পরবর্তীতে কোন instruction execute হবে তার address রাখে।

Instruction Queue:
এই মুহূর্তে যে instruction করছে তার পরে যে instruction execute করবে তার 6B reserve করে রাখে

Instruction queue কেন দরকার?

pipelining:

2B refilled:

2B বের করে execute করা হয়ে গেলে আবার 2B Load করবে।

* যেহেতু DS, CS, SS, 16 bit তাই সবসময় 16 bit ই Load করে রাখে instruction queue তে।

```
if ( ) {

}
else {

}
```

* Branching problem:

```
.MODE SMALL
.STACK 100H      → stack segment
.CODE            → code segment
MAIN PROC                          main () {

MAIN END                           return 0;
END MAIN                           }
```

* character, binary

1101 (Decimal)

1101 B (Binary)

1A0OH (Hexadecimal)

MOV Dest^n, source | Dest^n register অথবা source register হবে।

ADD Dest^n, source

A=2 where A variable

MOV AX, 2 (AX → register) → register AX এ 2 রাখবে

MOV A, AX → register থেকে নিয়ে A তে রাখবে।

MOV AX, 4

ADD A, AX — এভাবে দিলে A এর result রাখবে।

DB → B
DW → 2B
DD → 2W
DQ → 4W
DT → 10B

} variable declaration এ সময়সাপেক্ষে। এগুলো হলো Data type.

variable declaration:

```
Var1   DB   20H   [20H বাইটের]
Var1   DB   ?     [Blank variable]
Var1   DB   10H,20H,30H [Var1 একটি array হলো।]
```

```
SUB   D,S
ADD   D,S
MOV   D,S   [ডাটাগুলো আদানপ্রদান করবে]
XCHG  D,S   [ দুইটির value exchange করবে]
INC   D     [একটি বড় value বাড়িয়ে দিবে]
DEC   D     [কমিয়ে দিবে]
```

Interrupt

Somehow microprocessor কে interrupt দিলে আগের যেসব কোন ধরনের interrupt . সবগুলো function match করে। তারপর যে মেনুযায়ী কাজ করে।

1	$f_1()$
2	$f_2()$
3	$f_3()$

* Error msg দেওয়ার interrupt.

INT 21h [21h নাম্বার function টি নিয়ে আমরা কাজ করব]

INT 21h যখন দিবে তখন microprocessor input/output ডিভাইসগুলোর কাজ করে।

AH এ 1 রাখলে
single character input নিবে ব্যবহার AL এ

AH ← 2
single character output from DL and copy to AL.

AH ← 9
string output from DX
DX should contain the offset of string to be printed.

* প্রত্যেকটির ভিতরে offset segment থেকে data segment দিয়ে আলাদা করা যায় কি?

```
.MODEL SMALL/LARGE/MEDIEM   — code [illegible] segment লাগবে সেটার upon depend
.STACK 100H   → stack size এর মেমোরি
.CODE

MAIN PROC

MOV AH,1
INT 21h

MOV DL,AL
MOV AH,2
INT 21h

MAIN ENDP // main func এর শেষ
END MAIN ; total func. এর শেষ
```

* variable মানেই memory তে। memory মানেই data segment. সেজন্যই data segment design করতে হবে।

```
.MODEL SMALL
.STACK 100H
.DATA
MSG DB ; একটা array হিসেবে কাজ করবে।
'Please Enter a character $'
.CODE
MAIN PROC → MOV AX,@DATA
            MOV DS,AX     } অথবা MOV DS,@DATA হবে না  MOV এ দিয়ে [illegible]
LEA DX,MSG [Msg এর offset address টা লোড করে দিবে
            Load Effective Address]
MOV AH,9
INT 21h
```

```
MOV AH,1
INT 21h

MOV DL,AL
MOV AH,2
INT 21h      → MOV AH,4CH
               INT 21h

MAIN ENDP
END MAIN
```

google
Program segment Prefix 256 B

* program run এর শুরুতে একটুকু Load করে দেয়। program segment এর prefix DS, ES এ থাকে তাই কারণ Data segment থেকে read করা হয়।

DS, ES এ Data segment এ restore করতে হবে।

Scanned by CamScanner

4 bit → 4004 (1971 সাল)
8 bit → 8008, 8080, 8085 → 6800 (motorola) → 6808 (motorola)
16 bit → 8086, 8088, 80186-286
32 bit → 80386, 486
64 bit → Pentium, PI→IV, Core 2DUO = Dual core, Xeon, Corei3 – i7

x86 architecture

powerpc 603, G4, G5 — motorola

Microprocessor এর register এর bit size অনুযায়ী data process করে। register এর সংখ্যাকে word size বলা হয়।

Instruction set:

Stack segment: Stack pointer / Base pointer
* আমরা সবসময় offset নিয়েই কাজ করব।

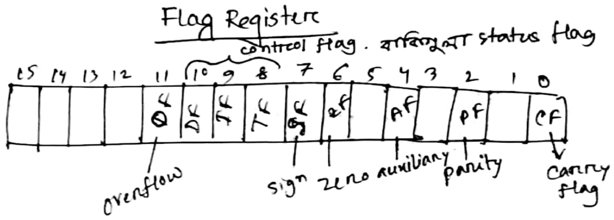
ES:DI }
DS:SI } 20 bit physical address তৈরি হয়।

* Segmentation এর benifit কি?
* 20 bit এর physical address কে 16 bit এর resister এ রাখতে পারতাম।

Assignment

assembly এর respect এ { Relocatable code / Absolute machine code
difference, Advantages, Disadvantages

Flag Register



CF ← If carry out from msb or borrow into msb

PF ← Even no. of 1 in low byte

AF ← If carry out from 3rd bit or borrow into 3rd bit

ZF ← If zero result

SF → If sign bit 1.

এই case গুলোতে এই flag গুলো 1 হবে। নইলে 0.

Control flag:
বিভিন্ন operation control এর জন্য control flag.

Direction flag:
Left to right or right to left.

~~AX = 1111 1110~~
~~BX = 1111~~

```
        F    F    F    E
AX =  1111 1111 1111 1110
BX =  1111 1110 1111 1111
        F    E    F    F
-------------------------
   (1) 1111 1111 0111 1101
AH  |   F    E  |  F    D  |  AL
```

MOV AX, FFFE

MOV BX, FEFF

CF = 1
PF = 0
AX = 1
ZF = 0
SF = 1

3D

সবগুলোই conditional jump instruction
unsigned representation.

Jump Instructions

CMP D,S

- > jump above ← JA → CF=0 , ZF=0
- < jump below ← JB → CF=0
- ≥ jump above equals ← JAE → CF=2
- ≤ jump below equals ← JBE → CF=1, ZF=2

JMP Destination Level

a≤10 ?

JA true হবে যদি ঠিক আগের operation এ CF=0, ZF=0 থাকে।

```
Sub Ax, 10
JA ~~~~
```

destination change হয়ে যাবে। এটা sub এর জন্য sub use না করে cmp use করা হয়

```
cmp Ax,10
JA ~~~~
```

বিয়োগ করবে। কিন্তু বিয়োগফল Ax এ store হবে না।

JA label ← এর goto এর মত।

Ex: JA myPos

myPos:

JMP Destination Level:

condition না থাকলে jump করে অন্য জায়গায় যাতে JMP Destination Level.

benifit of JMP:

conditional jump এর ক্ষেত্রে same segment এই থাকতে হবে। কিন্তু JMP তে same segment দরকার নাই.

a থেকে start করে 10 টা character print:

```
.CODE
mov DL, 65
PRINT:
mov AH,2
INT 21h
CMP DL,75
JA Exit
ADD DL,1
JMP PRINT

EXIT:
mov AH,4CH
INT 21h
MAIN ENDP
END MAIN
```

Assending

```
MOV AH,1
INT 21h
AL ← C1
mov BL, AL        ; AL এর value BL এ থাকবে
INT 21h
AL ← C2
mov AH,2
cmp AL,BL
JA Print          ; হলে (AL>BL)
mov DL,BL
Jump printit
Print:
mov DL,AL
printit:
INT 21h
```

ATMEGA 32 — 200/300tk

4C

Overflow(flag)

FFFF	1111	1111	1111	1111	→ -1
+0001	0000	0000	0000	0001	→ +1
	1 0000	0000	0000	0000	

Carry flag

→ unsigned হলে overflow হবে

```
     10          11          00 → 0
2's − 01         00 (2's)    01 → 1
      1           1          10 → -2
     10          01          11 → -1
     ↳ -2
```

1. Signed overflow
2. Unsigned overflow
3. Both
4. None

unsigned

```
  11 → 3
  01 → 1
 100 → 4     overflow
```

signed

```
  11 → -1
  01 → 1
  00 → 0     not overflow
```

- carry flag 1 হলে unsigned overflow
- signed integer এ যোগ বিয়োগ করার পর Limit cross করলে signed overflow.

overflow = 1 হলে signed.

MSB তে carry আসলো এবং carry বের হল এই দুইটি same না হলে
signed overflow | MSB-in ≠ MSB-out

<u>Signed overflow:</u>

7FFF → 0111 1111 1111 1111
7FFF → 0111 1111 1111 1111

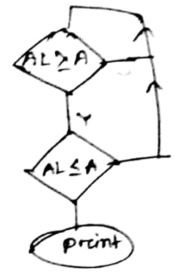
JG/JNLE JGE/JNL JL/JNGE JLE/JNG	signed jump Jc

CF = 1 এর উপর depend.

<u>4D</u>

```
'CODE
MOV AX, @DATA
MOV DS, AX
try-Again:
MOV, AH, 1
INT 21h ; scan করলাম

CMP AL, 'A'
JGE ChK-2 ; True হলে এটি execute হবে
JMP try-Again
```




```
ChK-2:
CMP AL, 'F'
JLE PRINT
JMP try-Again

PRINT: → mov BL, AL
mov AH,2
~~mov DL, Ah~~ mov DL, '1' ; যেহেতু A-F এ আছে এবং 1 এর পর something হবে 0-4
INT 21h
SUB BL, 'A' ; A থেকে A বিয়োগ = 0   B থেকে A = 1
mov AL, BL
INT 21h.
```

Scanned by CamScanner

LOOP

Lebel
—
—
Loop Lebel

mov cx, 10 / cx counter register. count করবে loop কতবার ঘুরবে।

8086 →
~~BIU~~ BIU
EU
~~PIN~~ Diagram

SA

AND destination, source } mask design এ
OR destination, source
XOR destination, source.
SHL destination, CL (shift left)
SHR destination, CL (shift right)
NOT destination ~~source~~

mask কেন বলা হয়?
mask বলতে কোন কিছুর filtering করা বুঝায়।
প্রয়োজনীয় জিনিস রাখব। অপ্রয়োজনীয়গুলো বাদ পড়ে যাবে।

1101
?

00 → 0
0(1) → 1
10 → 2
1(1) → 3
100 → 4
10(1) → 5
→ position এ 0 নাকি 1 তা check.

set করতে চাইলে OR. (set মানে 1 করা)

```
0000 1101
0000 1011      OR AL,02h
---------
 0    2
```

XOR →
mov AL,02h
AL এর value clear করতে বলল।
AND, XOR দিয়ে করতে পারি।

XOR AL, AL (কারন A, B এক হলেই XOR 0 হয়।

এটা faster কেন?
*carry নাই
*overflow নাই

AND

'a' → 41H → 01000001
20H → 00100000
'A' 61 → 01100001 (OR)

~~OR, 02h~~
OR AL, 20h

a → A = 41 → 61
b → B = 42 → 62

100 = 4
110 = 6

SHL:
<<
1 bit shift করলে
SHL destination, 1
*CL এ যতবারের তত bit করেই shift

AL | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 |
1 bit shift করব
AL | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |
Left most bit carry flag এ ঢুকে যাবে।
JC দিয়ে carry flag check করতে পারি।

SHR:
ODD even check করব।
একবার right shift করব JC দিয়ে cf check করব।

AL | | | | | |
3 2 1 0
4 3 2 1

NOT:
10111110
01000001

mov AL,0Fh
01h
or 1 করলে 01
দিলে most left
bit 1 হবে

1.

Control unit
↳ Control bus

Microprogram controlled

Hardware controlled

RISC & CISC architecture

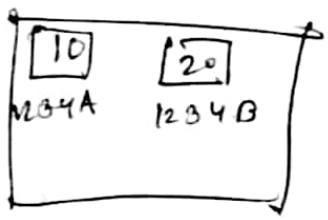
RISC : Reduced Instruction set Computer
CISC : Complex Instruction set Computer

basic difference

ISA → Instruction set Architecture.

mov AX, [1234] → memory operand

ISA



2 No way

RISC
LOAD AX, [1234A]
LOAD BX, [1234B]
ADD AX, BX
STORE [1234A], AX

* single cycle

MOV

H.W ? easy.

1 No. way

CISC ADDITION 1234A, 1234B

H.W ? tough

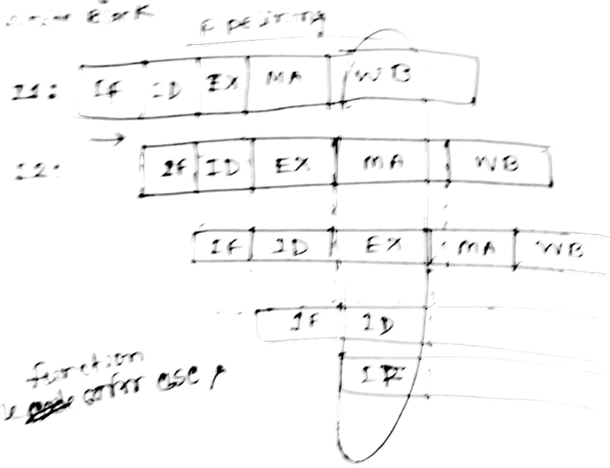
* multiple cycle

* Basic form instruction are common.

IF: Instruction Fetch
ID: Decode
EX: Execute
MA: Memory Access
WB: Write Back

Pipelining
→ RAM operation

Pipelining



available function বেশি ISC।

Power consumption:
তাই সাধারণত বেশি হবেই

RAM এর সাথে connection বেশি থাকে হবে?
RISC এ code ইন্সট্রাকশন বেশি হয়। So, connection বেশি
হবে RISC এর সাথে।

RISC এ register বেশি। So খরচ বেশি।

RISC vs CISC

Time = ICPT → Clock per time (s)

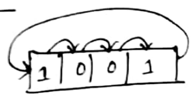
- No. of Ins./program
- cycle/Inst
- clk period/cycle

CT Flag register এর আগে করবো।

6D

ROR → Rotate Right
ROL → Rotate Left
RCR →
RCL →

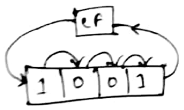
ROR:



ROR AX,CL

একটা করে swift হবে। আবার ঘুরে আসবে। একটার কপি CF এ চলে যাবে
SHR দিলে চারটা বের করে ফেললে সবগুলা 0 দিয়ে replace হয়ে যাবে।
কিন্তু ROR এ তা হবে না।

RCR:



CF এ যা আছে তা বামে আসবে। ডানের টা CF এ চলে যাবে।

AX → 16 bit নিতে পারবে। 16 bit input দিতে চাচ্ছি।

AL = এ 1 আছে মানে 1 এর ascii আছে।

AL = 31h বা 00110001

DX = 10110 . . . . একটা Input

```
AL  = 00110001
AND   00000001
      --------
      00000001
```

```
     DL = 00000000
     AL = 00000001
(OR)    --------
          00000001
```

```
XOR DL,DL   ; DL 0 করে দিলাম
MOV AH,1
INT 21H         → mov cx,8
AND AL,01h
SHL DL,1
OR  DL,AL
```

8 times যেতে 8bit input নেওয়া

Hexadecimal:

0-9 → AL = 00110001
A-F

```
AL = 00110001
     00001111
-------------
AL = 00000001
```

```
      3    9
AL = 0011 1001
     0000 1111
--------------
AL = 0000 1001
            9
```

```
     DL = 0000 0000
     AL = 0000 1001
(OR) ---------------
     DL = 0000 1001
```

9192

```
(4 bit shift) DL = 0001 0000
              AL = 0000 0001
              ---------------
              DL = 1001 0001
                     9    1
```

FOR 0-9

```
XOR DL,DL
mov AH,1
INT 21h
if (0-9)
AND AL,0Fh
SHL DL,4
OR DL,AL
```

FOR A-F

0ABACh

```
            4    1  h
AL  =     0100 0001
SUB       0011 0111
            3    7  h
-------------------
AL  =     0000 1010
```

65 - 41h
55 → 37h
10

else ~~AND AL, 3~~
SUB AL, 37h

7C

Stack Operation

← SP → Stack pointer

16 bit operation:
- PUSH source
- PUSHF → Flag register
- POP destination
- POPF → Flag register reload করে

Function:

~~[illegible]~~

```
NAME PROC

RET
NAME ENDP
```

```
MAIN PROC

MAIN ENDP
```

* myfunc নামের func.

```
Myfunc PROC
```
→ parameter passing

```
RET
Myfunc PROC
```

(**) stack এর concept দিয়ে allocate হবে deallocate হবে।

PUSHF ← CALL Myfunc.

Self: * calling [illegible] address [illegible] হবে।

Multiplication

MUL source (→ Reg)

IMUL source

↗ signed operation এর জন্য Integer multiplication

AX = Source × AL

DX : AX = Source × AX

(DX ↙ upper half; AX ↳ lower half)

AX = 1111 . . . 1 B

BX → .

* Pipeline
* Multiplication, AND, OR, XOR, SHL, SHR, ROR, ROL, NOT
* Stack
* Overflow
* RISC, SISC
* Masking
* Hex

—0—

8(C)

$100000 \times s\% \times C_s \times t_c + 100000 \times (100-s)\% \times C_e \times t_e$

Decimal input:

0–9

134 এরকম number নিতে চাচ্ছি।

(BX ↓, AX ↓)

0 × 10 + 1 = 1

1 × 10 + 3 = 13

13 × 10 + 4 = 134

Input: AH ← 1,
INT
AH ← 0

```
AND  AX,000Fh
MOV  DX,AX ;1 (Input)
MOV  AX,10
MUL  BX ;Result in AX
ADD  AX,DX
MOV  BX,AX
```

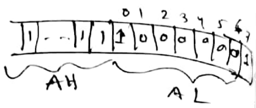
DIV divisor
IDIV divisor

divisor) AX (AL → ভাগফল AL এ থাকবে

(→ 16 bit AX এর value থাকবে)

AH → AH এ remainder থাকবে

Remainder's sign = dividend's sign



~~CWD~~
~~CBW~~ AL এর 0 তে sign bit আছে। So AH এ সবগুলো 1 হয়ে যাবে।

divisor) DX:AX (AX

DX

CWD = word-double word
CBW = Byte-word

BL → 2) -5 (← AL

```
mov AL, -5
CBW
mov BL, 2
IDIV BL
```

ADDRESSING MODE

Register indirect:

int *P

Register contains the address of some memory. যার register pointer হিসেবে কাজ করবে।

MOV AX, [Register]

DS:SI

SI, DI → offset address

Myarray DW 10, 20, 30, 40 / A, B, C, D

* Load effective address দিয়ে এইটার address পাবো। p = &a এর মতো;

```
MOV CX, 4
MOV AH, 2
LEA SI, Myarray
PRINT:
MOV DL, [SI]   ; 1st value print হবে
INT 21h
ADD SI, 2
loop print
```

Based / Indexed Addressing mode

[offset + displacement] or

offset [displacement]

আগেরটার মতন, ~~Addressing~~ mode টি pointer এর মত কাজ করে। এটা array র মত কাজ করে।

~~Myarray [AX]~~

```
MOV AX,0
mov R,Myarray [AX]
ADD AX, 2
```

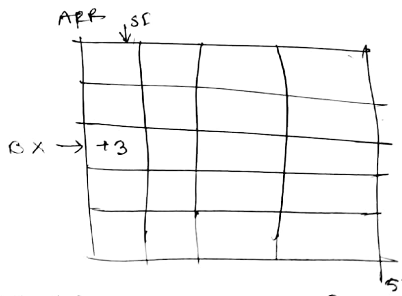
ARR [R] [C]

* Base এবং register একই সাথে ব্যবহার হবে।

DW মানে word type

2A

ARR [BX] [SI]




```
MOV CX, 3
XOR SI, SI
MOV BX, 16
ADD ARR [BX] [SI], CX
ADD SI, 2
```

→ CX এ 3 রাখছি 3 যোগ করানোর জন্য। CX দিয়ে Loop চালাব না। তাহলে তো CX change হয়ে যাবে। তাই Jump দিয়ে loop control করব।

Scanned by CamScanner

```
MOV AX,STUDENT
MOV BH,ROLL
```

LABEL

```
STUDENT LABEL WORD
ROLL   DB 153001
MARKS  DB 30
```

MOV AX,BL ? কারণ AX=16, BL=8 bit

PTR

```
MOV BYTE PTR [BX],10
    WORD
```

BYTE pointer মানে byte type এ value দিবে

```
MSG DB 'RUET $'
LEA MSG
```


XLAT

Translate এর জন্য কাজে লাগে। XLAT এর কাজ আগে যা শিখেছি তা দিয়েই করা সম্ভব। তবে এই instruction ও লাগে মাঝেমাঝে।

BX ← table index
AL ← Byte value to be converted
XLAT (BX, AL এর value এর উপর depend করে কাজ করে)

CODE

B	C	D	E	F	A

CODE একটি array. BX এ array র index রাখব AL এ value দিয়ে

```
LEA BX,CODE
MOV AL,01h
XLAT
```

CODE (65h, 66h, 68h)

	Y	Z	A	B	

01h CODE এর মধ্যে খোঁজা হবে। তার পর যে value পাব সেটা দিয়ে replace করে দিব।

MSG

D	C	F

এই table কে CODE table এর value দিয়ে replace করব

~~LEA MSG~~

```
LEA SI,MSG
MOV AL,[SI]   (D কে AL এ রাখলাম। MSG)
XLAT
MOV DL,AL
MOV AH,2      } BAD
INT 21h
INC SI
```

DUP → Duplicate করার জন্য

CODE DB 65 DUP (' '), YZAB . . .

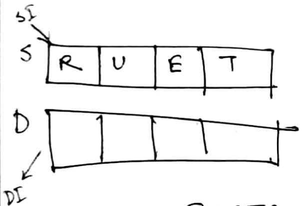
String

* MOVSB → Byte — mov এর মত কাজ করে। MOV করার কাজ করে।
* MOVSW → word

ES:DI ← DS:SI

ES:DI → Destination array

DS:SI → Source Array



S এর value D তে mov করবো।

```
mov AX, @DATA
mov ES, AX
CLD
LEA SI, S
LEA DI, D
MOVSB
MOVSB
MOVSB
MOVSB
```

[DATA segment এর Address ES এ চলে আসল]

DF

0 → SI / DI (বাড়তে থাকে) → CLD

1 ← SI/DI (কমতে থাকে) → STD

LODSB/W	AL/AX	DS:SI
STOSB/W	ES:DI	AL/AX

* LODSB বললে AL এ যাবে LODSW বললে AX এ যাবে।

ES:DI [A | B | C | D]   [C | D] DS:SI

* ES:DI যেখানে point করে আছে সেটা এবং DS:SI যেটা point করে আছে সেটার সাথে compare হবে।

REPE — SCAN B/W

REPEZ — CMPSB/W

ES:DI [A | B | C | D] → targetstring   AL/AX

প্রথমবার AL এর সাথে 1st ঘরের compare হবে। match করলে ZF = 1

বিয়োগ করে A - A = 0

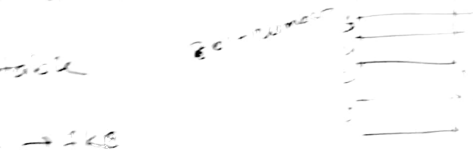
ZF = 1

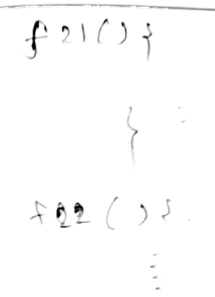
CMPSB/W:

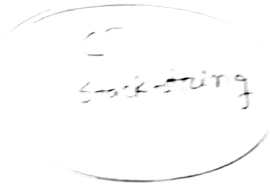
SCAN একটা একটার করে যে কাজটা করে CMPSB/W সেটা দুটো একটা array এর জন্য সম্পাদন করে।

REPE (repeat while equal)
REPNZ (while not zero)

Assembly ka
Hardware related







Timing of cost

For read/write cycle.

Co-processor

max → Multiprocessor
min

8086 এর সাথে coprocessor নিয়ে কাজ করতে পারি।
8086 max mode এ ব্যবহৃত।
mode check করতে pin আছে। pin দেখলেই বুঝব কোন mode এ আছে।

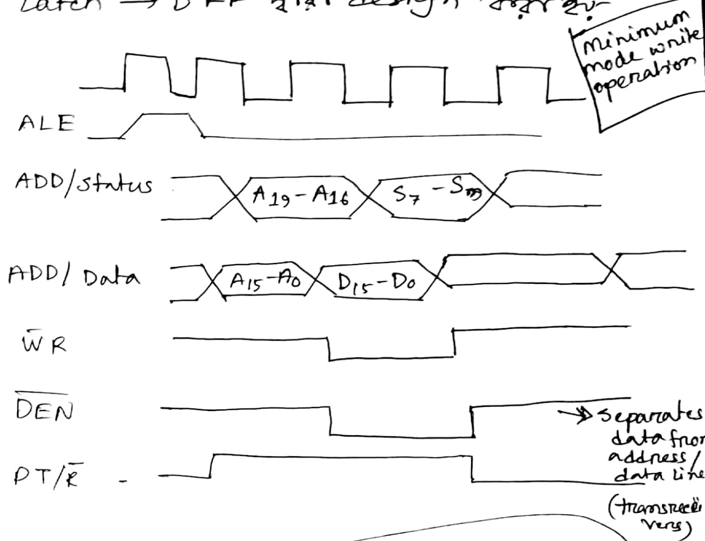
8087 → mathcoprocessor.
mathematical কাজগুলো 8086 কে সাহায্য করে।

8086 এর block diagram

Latch →
Transreceiver →
memory →
I/O →

Assignment on —
Latch, Transreceiver, Clockgenerator কে কোন microprocessor এর সাথে ~~কাজ~~ কাজ করে এবং তাদের প্রয়োজনীয় কাজ কি?

Latch → DFF দ্বারা design করা হয়



Maximum mode এর operation. read এর আর write cycle এর আর Self-study

max mode
↳ Co proc
↳ Bus controller

DMA operation:

Direct memory Access.

Any device থেকে memory data আদান প্রদানের সময় microprocessor কে disturb করতে না।

1. Microprocessor কে 1st এ জানানো হয় যে DMA operation করব।
2. Micro.p: acknowledgement দিবে
3. control চলে DMA controller কে দিয়ে দিবে।

বাকি কাজ DMA controller করে।

5 Ram এর performance ভাল নাকি D Ram এর ভাল? D Ram এর দাম কত। কোন কোনটা? quality কি একই থাকে?

Interrupt:

IACK, INTR

Co-processor → block diagram

RISC

CISC

Direct redio ram ✗ প্রয়োজন নাই

Micro controllers এর use নিয়ে একটা ধারণা হবে।