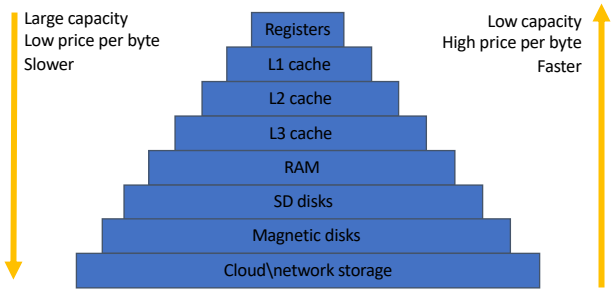
Registers
Instruction MOV and ADD

1

Registers

Registers are the top level of memory hierarchy



2

Registers

The number of registers in each CPU is very small

This number tends to increase over the time

Also, the registers size tends to increase over the time

Each register is identified by its name, not by an address.

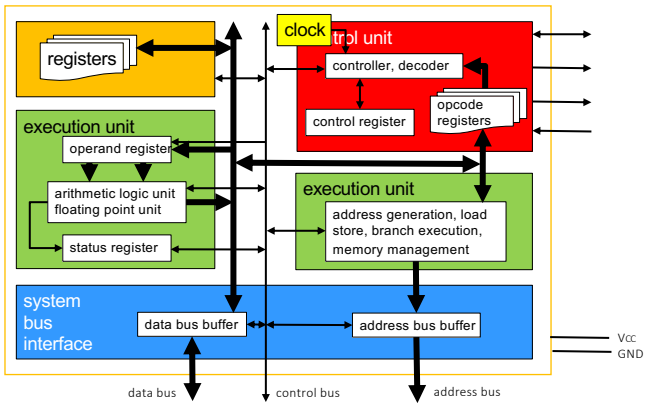
The name isn´t case sensitive.

Some **different registers** can share the same **physical space.**

In these cases, the changes in one register implies changes in the other, since they share the same physical space

Each new CPU that implements new registers must maintain the name and size of all registers prior to this CPU so that it can be compatible with the previous software.

3

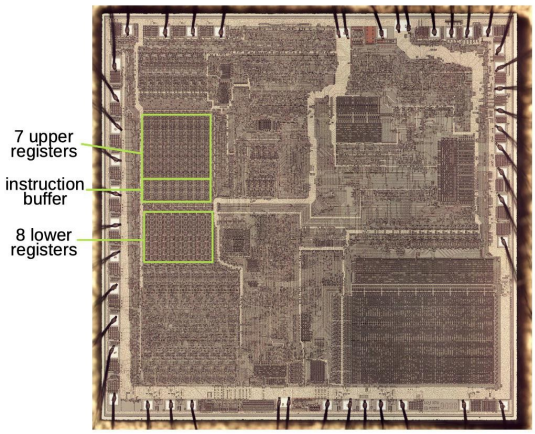


4

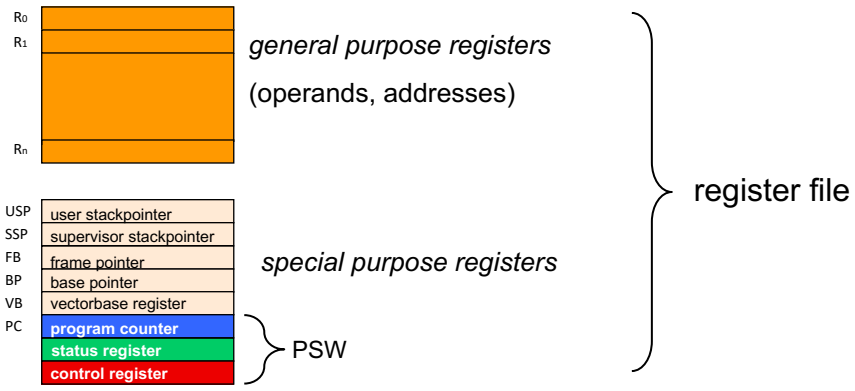
Registers

- Very fast memory with very low access time (< 1 ns)
- Direct selection of single registers via dedicated control lines
  - No address decoder / decoding necessary
- All registers are on-chip
  - No external access necessary with delays due to run-time, multiplexing, buffering etc.
- Can offer additional functions
  - Increment/decrement
  - Shift
  - Set to zero, hardwired to zero
- Several independent input/output port
  - Simultaneous writing and reading of several (different) registers possible
  - Today's superscalar processors are able to write 4 registers and read 8 registers in one clock cycle

5

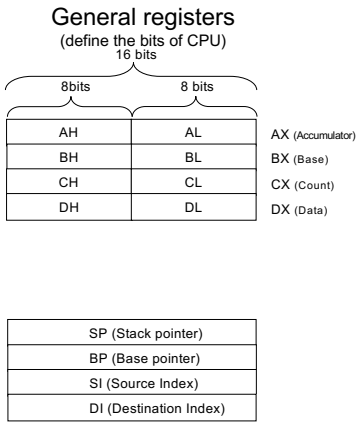


6



7

Registers 8086/286



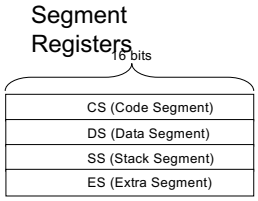
8

SP, BP, SI and DI

- **Stack Pointer (SP)** – The 16-bit SP register provides the offset value within the program stack.
- **Base Pointer (BP)** – The 16-bit BP register mainly helps in referencing the parameter variables passed to a subroutine.
- **Source Index (SI)** – It is used as source index for string operations.
- **Destination Index (DI)** – It is used as destination index for string operations.

9

Registers 8086/286



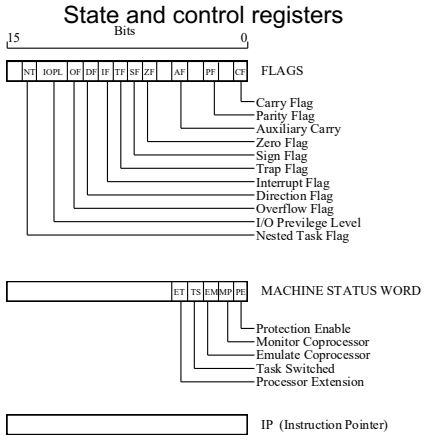
10

CS, DS and SS

- **Code Segment** − It contains all the instructions to be executed. A 16-bit Code Segment register or CS register stores the starting address of the code segment.
- **Data Segment** − It contains data, constants and work areas. A 16-bit Data Segment register or DS register stores the starting address of the data segment.
- **Stack Segment** − It contains data and return addresses of procedures or subroutines. It is implemented as a 'stack' data structure. The Stack Segment register or SS register stores the starting address of the stack.

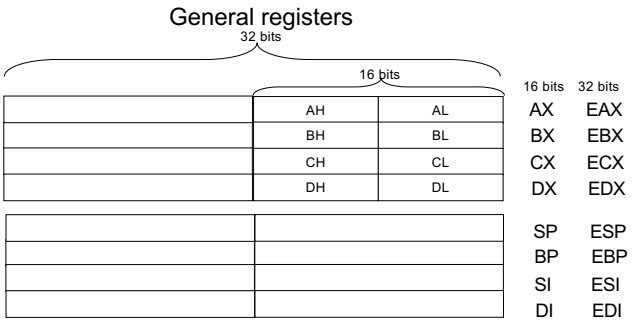
11

Registers 8086/286



12

386 registers



Find the value of AX, AH and AL in decimal if we put 04 03 02 01 H (hex) in EAX

13

Find the value of AX, AH and AL in decimal if we put 04 03 02 01 (hex) in EAX

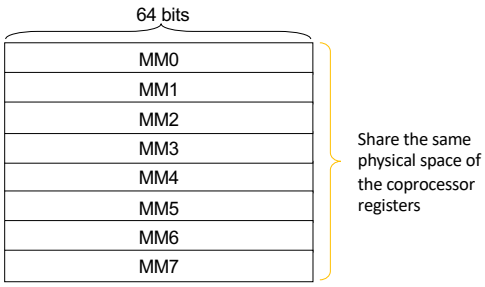
- EAX: **04 03 02 01**
- AX: 02 01
- AH: 02
- AL: 01

| 0000 0001 0010 0011 0100 0101 0110 0111 | ------> EAX
| 0100 0101 0110 0111 | ------> AX
| 0110 0111 | ------> AL
| 0100 0101 | ------> AH

14

News registers in Pentium MMX

(Not general registers. The CPU is a 32 bits CPU)

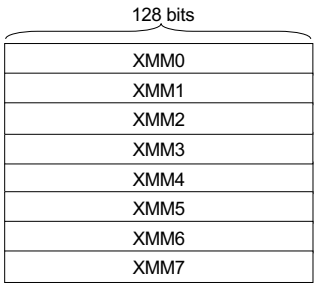


15

News registers in Pentium II

(Not general registers. The CPU is a 32 bits CPU)

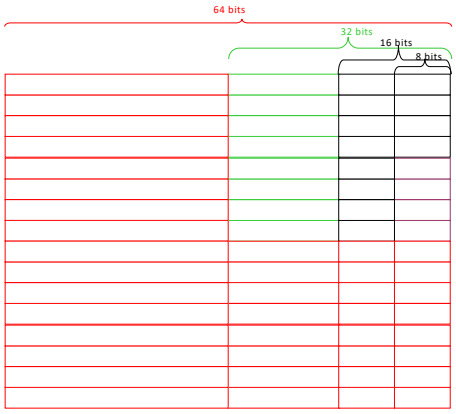
XMM registers



16

EM64T technology

General registers

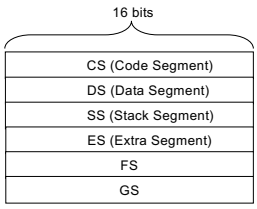


Bits			
8	16	32	64
AL,AH	AX	EAX	RAX
BL,BH	BX	EBX	RBX
CL,CH	CX	ECX	RCX
DL,DH	DX	EDX	RDX
SPL	SP	ESP	RSP
BPL	BP	EBP	RBP
SIL	SI	ESI	RSI
DIL	DI	EDI	RDI
R8L	R8W	R8D	R8
R9L	R9W	R9D	R9
R10L	R10W	R10D	R10
R11L	R11W	R11D	R11
R12L	R12W	R12D	R12
R13L	R13W	R13D	R13
R14L	R14W	R14D	R14
R15L	R15W	R15D	R15

17

EM64T technology

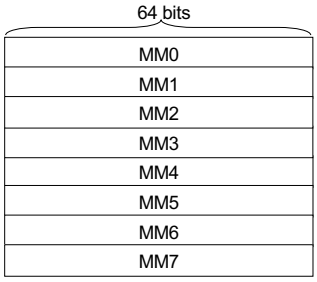
Segment registers
(all in the same)



18

EM64T technology

MMX registers
(all in the same)

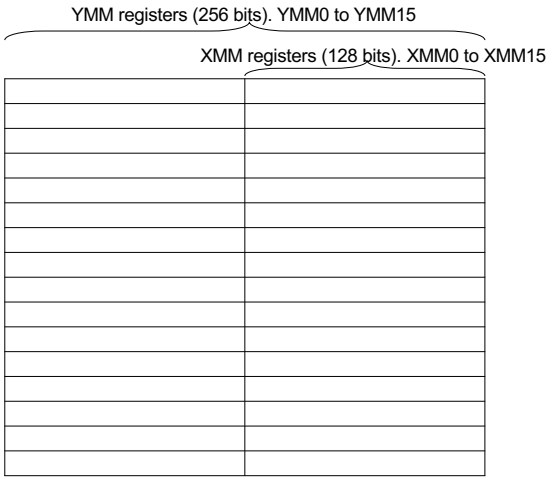


19

EM64T technology

20

AVX technology (Advanced Vector eXtension)



21

Register

- A register is a temporary storage area built into a CPU.
- A CPU register a small set of data holding places that built into a CPU.

- A register may hold an instruction, a storage address, or any kind of data (such as a bit sequence or individual characters).
- Some instructions specify registers as part of the instruction.
  An instruction can specify that the contents of two defined registers be added together and then placed in a specified register.

- A register must be large enough to hold an instruction.
  A 64-bit computer, a register must be 64 bits in length.

22

Instruction MOV

MOV is the must used instruction in assembly

The MOV format is:

MOV *destination, source*

This instruction move the contends of the *source* to the *destination*

Source and *destination* must have the same size

Destination can be a register or memory reference (variable or address)

Source can be a register, a memory reference or a value

Simultaneous isn´t possible use memory reference as *destination* and *source*

23

Instruction MOV

examples

MOV AH,AL; put the contends of AL in AH (8 bits registers)

MOV AX,BX; put the contends of BX in AX (16 bits registers)

MOV ECX,EDX; put the contends of EDX in ECX (32 bits registers)

MOV EAX,BX; invalid instruction. Parameters with different size

MOV AL,10; put the decimal 10 in AL

MOV AL,10B; put the binary 00000010 in AL

MOV AL,2FH; put the hex 2F in AL

MOV AL,0A2H; put the hex A2 in AL. (if the first hex digit is a letter must be preceded by 0.

MOV AL,'A'; put the ASCII 'A' (decimal 65), in AL

MOV AL,256; 256 as 9 bits (100000000B). This instruction is valid but it only put the value 00000000B in AL. All bits in the left of the 7 bit are ignored

24

Instruction MOV
examples

If you define the variables `int x,y; //32 bits integer`

MOV x,0; put 0 in variable x;

MOV x,EAX; put the contends of EAX in variable x

MOV x,AX; invalid instruction. Parameters with different size

MOV EBX,x; put the contends of variable x in EBX

MOV AL,x; invalid instruction. Parameters with different size

MOV x,y; invalid instruction. The two parameters can´t be simultaneous memory reference

25

Instruction MOV
examples

If EDI as an address value (32 bits)

MOV [EDI],AL; put the contends of AL in memory at address EDI

Address	Contends (HEX)
EDI-2	??
EDI-1	??
EDI	AL
EDI+1	??
EDI+2	??
EDI+3	??
EDI+4	??
EDI+5	??

SI = Source Index

DI = Destination Index

26

ESI and EDI

- ESI and EDI are general purpose registers.
- If a variable is to have register storage class, it is often stored in either ESI or EDI.
- A few instructions use ESI and EDI as pointers to source and destination addresses when copying a block of data.

27

Instruction MOV

examples

If EDI as an address value (32 bits)

MOV [EDI],AX; put the contends of AX in memory at address EDI

Address	Contends (HEX)	
EDI-2	??	
EDI-1	??	
EDI	AL	AX
EDI+1	AH	
EDI+2	??	
EDI+3	??	
EDI+4	??	
EDI+5	??	

28

Instruction MOV
examples

If EDI as an address value (32 bits)

MOV [EDI],EAX; put the contends of EAX in memory at address EDI

Address	Contends (HEX)
EDI-2	??
EDI-1	??
EDI	AL
EDI+1	AH
EDI+2	XX
EDI+3	XX
EDI+4	??
EDI+5	??

EAX

29

Instruction MOV
examples

If EDI as an address value (32 bits)

MOV [EDI],1; Error if the compiler doesn't know the size of the pointer [EDI]

MOV BYTE PTR [EDI],1; Always work. BYTE PTR define [EDI] as a pointer to 1 byte

Address	Contends (HEX)
EDI-2	??
EDI-1	??
EDI	1
EDI+1	??
EDI+2	??
EDI+3	??
EDI+4	??
EDI+5	??

1 represented with 1 byte

30

Instruction MOV
examples

If EDI as an address value (32 bits)

MOV WORD PTR [EDI],1; WORD PTR define [EDI] as a pointer to 2 bytes

Address	Contends (HEX)
EDI-2	??
EDI-1	??
EDI	1
EDI+1	0
EDI+2	??
EDI+3	??
EDI+4	??
EDI+5	??

1 represented with 2 bytes

31

Instruction MOV
examples

If EDI as an address value (32 bits)

MOV DWORD PTR [EDI],1; DWORD PTR define [EDI] as a pointer to 4 bytes

Address	Contends (HEX)
EDI-2	??
EDI-1	??
EDI	1
EDI+1	0
EDI+2	0
EDI+3	0
EDI+4	??
EDI+5	??

1 represented with 4 bytes

32

Instruction MOV
examples

If EDI as an address value (32 bits)

MOV [EDI+1],AL; put the contends of AL in memory at address EDI+1

Address	Contends (HEX)
EDI-2	??
EDI-1	??
EDI	??
EDI+1	AL
EDI+2	??
EDI+3	??
EDI+4	??
EDI+5	??

33

Instruction MOV
examples

If EDI as an address value and EBX an index value (4)

MOV [EDI+EBX],AL; put the contends of AL in memory at address EDI+EBX or in this case EDI+4

Address	Contends (HEX)
EDI-2	??
EDI-1	??
EDI	??
EDI+1	??
EDI+2	??
EDI+3	??
EDI+4	AL
EDI+5	??

34

Instruction MOV
examples

Its possible multiply the index by 2, 4 or 8

If EDI as an address value and EBX an index value (2)

MOV [EDI+EBX*2],AL; put the contends of AL in memory at address EDI+EBX*2 or in this case EDI+2*2=EDI+4

Address	Contends (HEX)
EDI-2	??
EDI-1	??
EDI	??
EDI+1	??
EDI+2	??
EDI+3	??
EDI+4	AL
EDI+5	??

35

Instruction MOV
examples

If you define the variable `char lista[100]; //array of 100 chars`

MOV lista[0],AL; put the contends of AL in memory at address lista + 0

Address	Contends (HEX)
lista+0	AL
lista+1	??
lista+2	??
lista+3	??
lista+4	??
lista+5	??
lista+6	??
lista+7	??

36

Instruction MOV
examples

If you define the variable `char lista[100]; //array of 100 chars`

MOV lista[0],AX; Invalid instruction. Different parameter size

37

Instruction MOV
examples

If you define the variable `char lista[100]; //array of 100 chars`

MOV lista[EDI],AL; put the contends of AL in memory at address lista + EDI.
If EDI is 3 means lista+3

Address	Contends (HEX)
lista+0	??
lista+1	??
lista+2	??
lista+3	AL
lista+4	??
lista+5	??
lista+6	??
lista+7	??

Index EDI, in this case, can be multiply by 2, 4 or 8

38

Instruction MOV
examples

If you define the variable `int lista[100]; //array of 100 integers`

MOV EAX,1

MOV lista[0],EAX; put the contends of EAX in memory at address lista + 0

Address	Contends (HEX)
lista+0	1
lista+1	0
lista+2	0
lista+3	0
lista+4	??
lista+5	??
lista+6	??
lista+7	??

39

Instruction MOV
examples

There is a big difference between lista[n] used in high level languages and lista[n] used in assembly

In high level languages lista[n] identify the n element in the lista.
The memory position is lista+n*(size of element) = lista+n*4 for int elements

lista[1]=1;
Write the value 1 in the address lista+1*4

Address	Contends (HEX)
lista+0	??
lista+1	??
lista+2	??
lista+3	??
lista+4	1
lista+5	0
lista+6	0
lista+7	0

In assembly lista[n] identify the address at position lista+n

MOV lista[1],1;
Write the value 1 in the address lista+1

Address	Contends (HEX)
lista+0	??
lista+1	1
lista+2	0
lista+3	0
lista+4	0
lista+5	??
lista+6	??
lista+7	??

40

Instruction ADD

The ADD format is:

ADD *destination, source* (destination=destination+source)

ADD sums the *destination* with *source* and put the result in the *destination*

The ADD instruction affect the FLAGS

The ADD use all the rules presented for mov instruction

41

Move the contends of variable y to variable x

```
MOV EAX, y;     put the contends of variable y in EAX
MOV x, EAX;     put the contends of EAX in variable x
```

42