Basics of X86 architecture

Abhijit A M

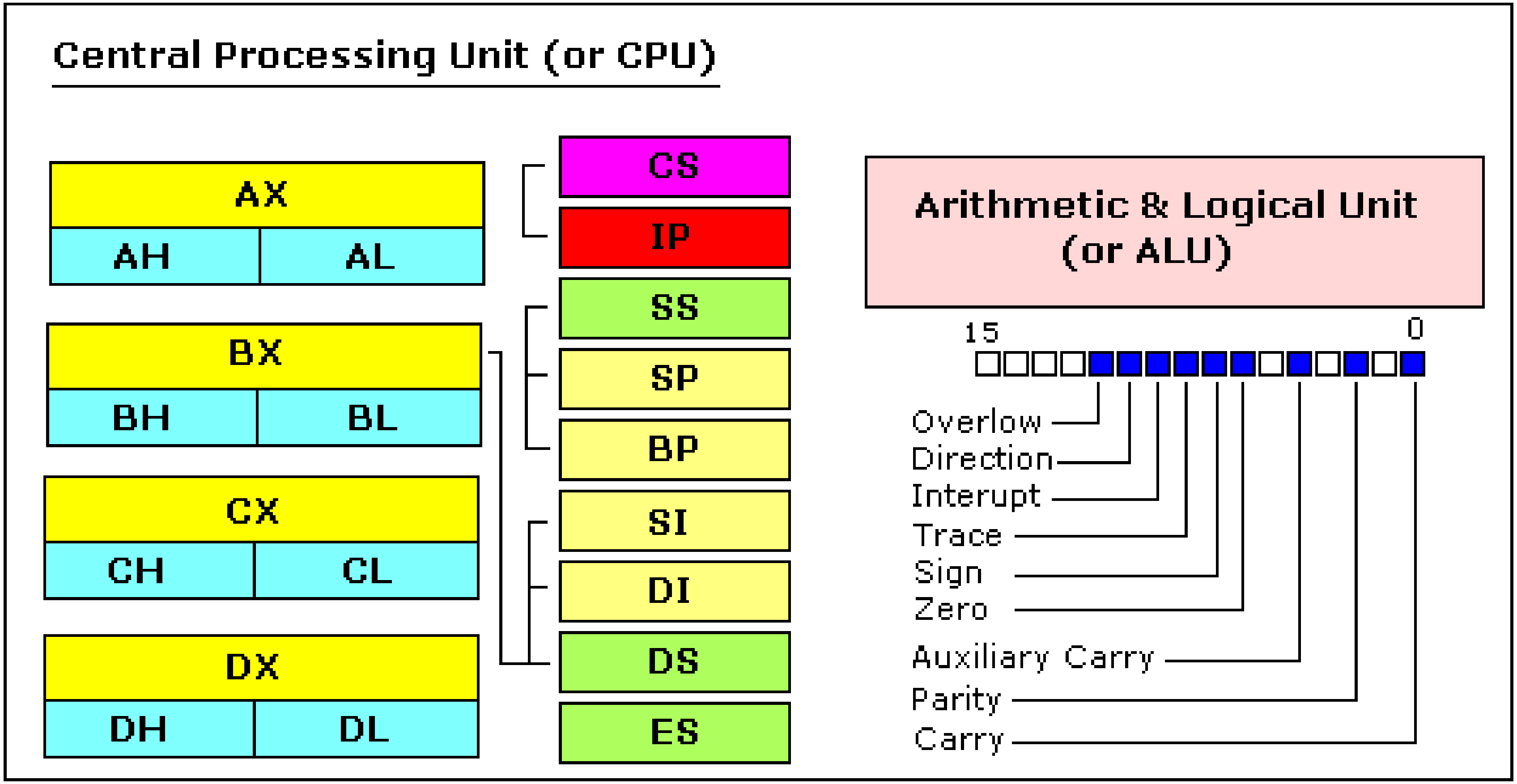
abhijit.comp@coep.ac.in

Credits: Notes by Prof. Sorav Bansal, https://www.felixcloutier.com/x86/, Intel Documentation

A processor typically has

- integer registers and their execution unit
- floating-point/vector registers and execution unit(s)
- memory management unit (MMU)
- multiprocessor/multicore: local interrupt controller (APIC)
- etc

8086: 16 bit CPU (precursor to x86 family)



https://jerry153fish.github.io/2016/01/01/8086-

8086 Registers

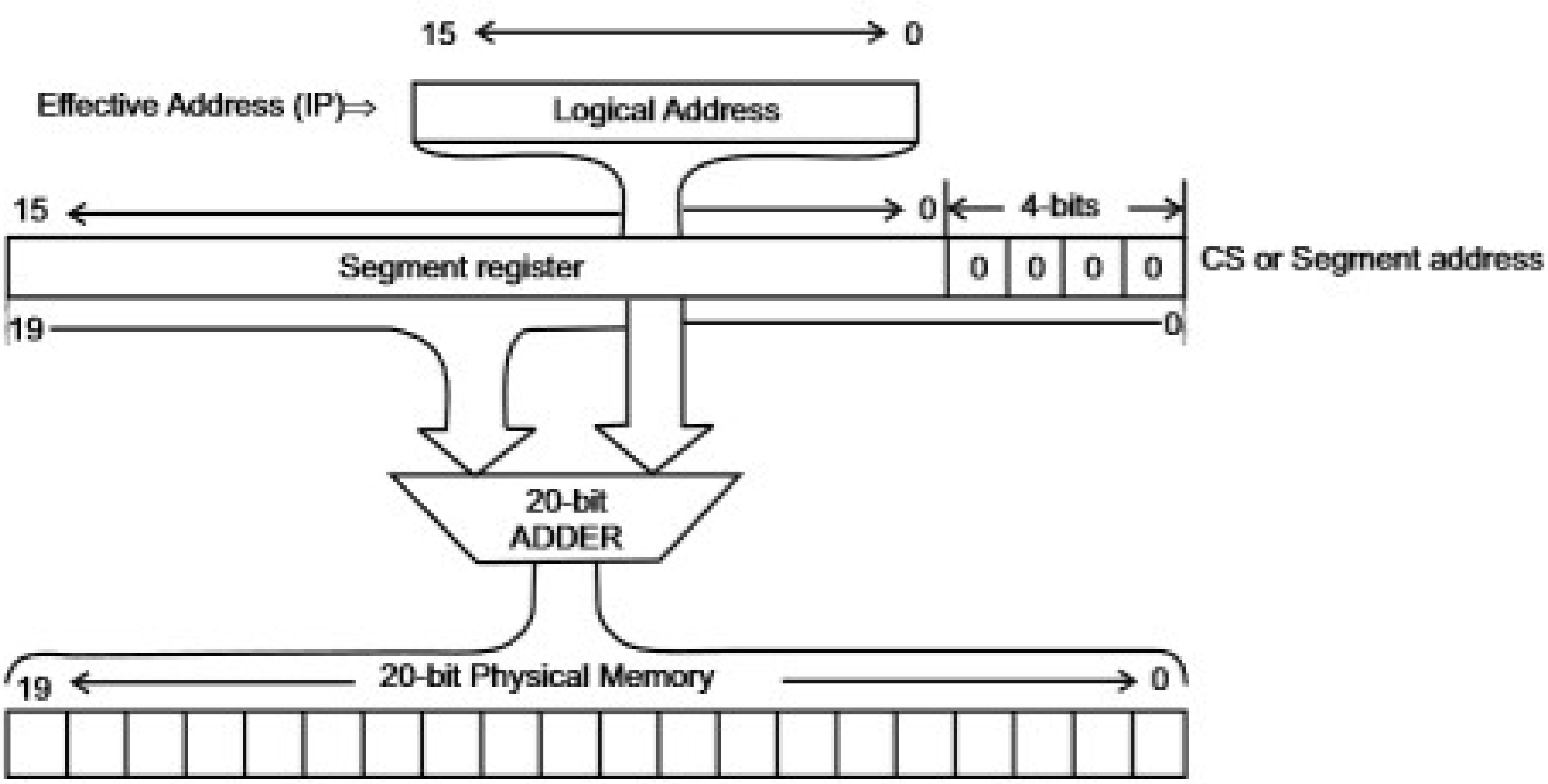
- 16 bit ought to be enough ! (?)
- General purpose registers
  - four 16-bit data registers: AX, BX, CX, DX
  - each in two 8-bit halves, e.g. AH and AL
- Registers for memory addressing
  - SP, BP, SI, DI : 16 bit
  - SP stack pointer, BP base pointer, SI Source Index, DI Destination Index
  - IP instruction Pointer
  - Addressible memory: 2^16 = 64 kb

8086 address extension

- 8086 has 20-bit physical addresses ==> 1 MB AS
- the extra four bits usually come from a 16-bit "segment register":
  - CS - code segment, for fetches via IP
  - SS - stack segment, for load/store via SP and BP
  - DS - data segment, for load/store via other registers
  - ES - another data segment, destination for string operations
- %cs:%ip full address: %cs * 16 + %ip is actual address that goes on bus. virtual to physical translation is
  - pa = va + seg*16
- e.g. set CS = 4096 to execute starting at 65536

8086 address extension

- Extending ‘address space’ with the help of MMU
-



FLAGS register

- FLAGS - various condition codes: whether last arithmetic operation
  - overflowed
  - was positive/negative
  - was [not] zero
  - carry/borrow on add/subtract
  - etc.
- whether interrupts are enabled
- direction of data copy instructions
- Uses in JP, JN, J[N]Z, J[N]C, J[N]O ...

32 bit 80386

- boots in 16-bit mode, then on running particular instructions switches to 32-bit mode (protected mode)
  - For backward compatibility, INTEL continued this. The CPU starts in 16 bit mode, called real mode
  - If particular instructions are not executed (may be in boot loader) all processors will continue in real mode
- registers are 32 bits wide, called EAX rather than AX
  - EAX EBX ECX EDX
  - ESP EBP ESI EDI
- operands and addresses 32-bit in 32-bit mode,
  - e.g. ADD does 32-bit arithmetic
- Segment registers are 16 bit: CS, SS, DS, etc.

32 bit 80386

- Still possible to access 16 bit registers using AX or BX
  - Specifix coding of machine instructions to tell whether operands are 16 or 32 bit
  - prefixes 0x66/0x67 toggle between 16-bit and 32-bit operands/addresses respectively
  - in 32-bit mode, MOVW is expressed as 0x66 MOVW
  - the .code32 in bootasm.S tells assembler to generate 0x66 for e.g. MOVW
- 80386 also changed segments and added paged memory...

Summary of registers in 80386 (32 bit)

- **General registers**
  - 32 bits : EAX EBX ECX EDX
  - 16 bits : AX BX CX DX
  - 8 bits : AH AL BH BL CH CL DH DL

Summary of registers in 80386

- Expected usage of Segment Registers
  - CS: Holds the Code segment in which your program runs.
  - DS : Holds the Data segment that your program accesses.
  - ES,FS,GS : These are extra segment registers
  - SS : Holds the Stack segment your program uses.
- All 16 bit

Summary of registers in 80386

- For a typical register, the corresponding segment is used. Pairs of Indexes & pointers (Segment & Registers)
  - CS:EIP
    - Code Segment: Index Pointer
    - E.g. mov $32, %eax ==> The code of move instruction uses CS: EIP
  - SS:ESP
    - Stack Segment: ESP
    - E.g. push $32 ==> The $32 will be pushed on stack. Using SS: ESP address
  - SS:EBP
    - Stack Segment: EBP ==> mov (%ebp), %eax ==> for accessing (%ebp) SS: EBP address will be used
  - DS:ESI , ES: EDI .
    - ESI: Extended Source Index, EDI: Extended Destination Index
- The EFLAGS register for flags

X86 Assembly Code

- Syntax
  - Intel syntax: op dst, src (Intel manuals!)
  - AT&T (gcc/gas) syntax: op src, dst (xv6)
  - uses b, w, l suffix on instructions to specify size of operands
- Operands are registers, constant, memory via register, memory via constant

Examples of X86 instructions

AT&T syntax	"C"-ish equivalent	Operands
movl %eax, %edx	edx = eax;	register mode
movl $0x123, %edx	edx = 0x123;	immediate
movl 0x123, %edx	edx = *(int32_t*)0x123;	direct
movl (%ebx), %edx	edx = *(int32_t*)ebx;	indirect
movl 4(%ebx), %edx	edx = *(int32_t*) (ebx+4)	displaced

Instructions suffix/prefix

```
mov %eax, %ebx # 32 bit data

movw %ax, %bx  # move 16 bit data

mov %ax, %bx  # ax is 16 bit, so equivalent to movw

mov $123, 0x123  # Ambigious

movw $123, 0x123  # correct, move 16 bit data
```

Types of Instructions

- data movement
  - MOV, PUSH, POP, ...
- Arithmetic
  - TEST, SHL, ADD, AND, ...
- i/o
  - IN, OUT, ...
- Control
  - JMP, JZ, JNZ, CALL, RET
- String
  - REP MOVSB, ...
- System
  - IRET, INT

Interrupt handling

Privilege levels

- The x86 has 4 protection levels, numbered 0 (most privilege) to 3 (least privilege).
- In practice, most operating systems use only 2 levels: 0 and 3, which are then called kernel mode and user mode, respectively.
- The current privilege level with which the x86 executes instructions is stored in CPL field inside %cs register

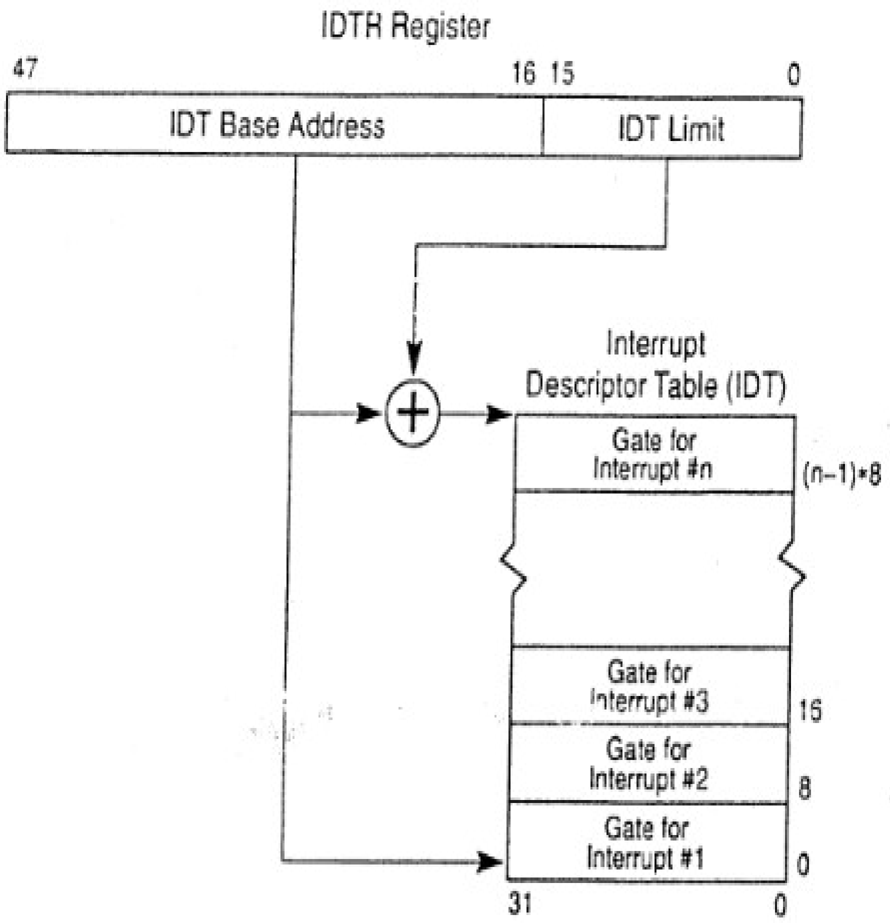
Privilege levels

- Changes automatically on
  - “int” instruction
  - hardware interrupt
  - exeception
- Changes back on
  - iret
- “int” 10 --> makes 10th hardware interrupt. S/w interrupt can be used to create hardware interrupt’
- Xv6 uses “int 64” for actual system calls

Interrupt Descriptor Table (IDT)

- IDT is an in memory table. IDT defines intertupt handlers
- Has 256 entries
  - each giving the %cs and %eip to be used when handling the corresponding interrupt.
- Interrupts 0-31 are defined for software exceptions, like divide errors or attempts to access invalid memory addresses.
  - Xv6 maps the 32 hardware interrupts to the range 32-63 and uses interrupt 64 as the system call interrupt

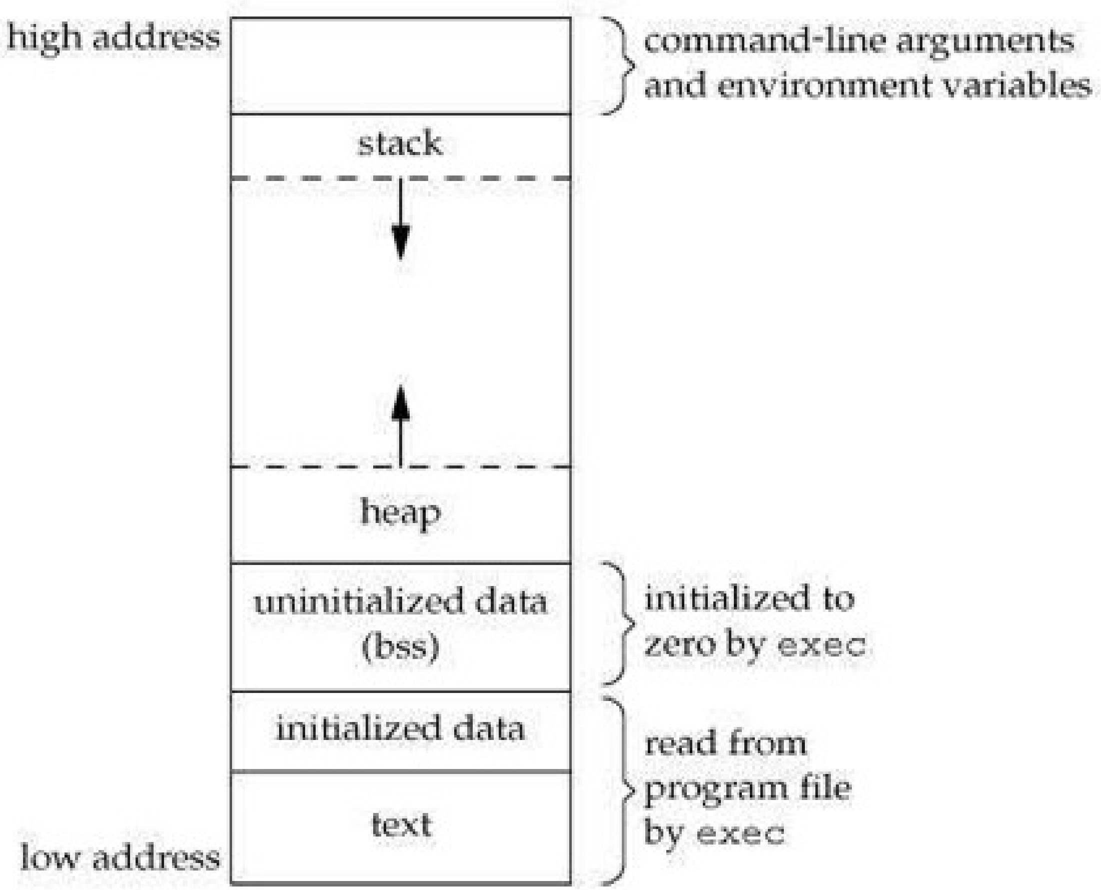
IDTR and IDT



- **An entry in IDT is called a "gate"**
- **IDTR is a CPU register, IDT is in memory table**

Memory Management in x86

Memory Layout of a C Program


high address
low address
command-line arguments and environment variables
stack
heap
uninitialized data (bss)
initialized data
text
initialized to zero by exec
read from program file by exec

text, data, bss are also present in the ELF file
stack and heap are not present in ELF file
WHY?

Text: object code of the program

Data: Global variables + (local/global) Static variables

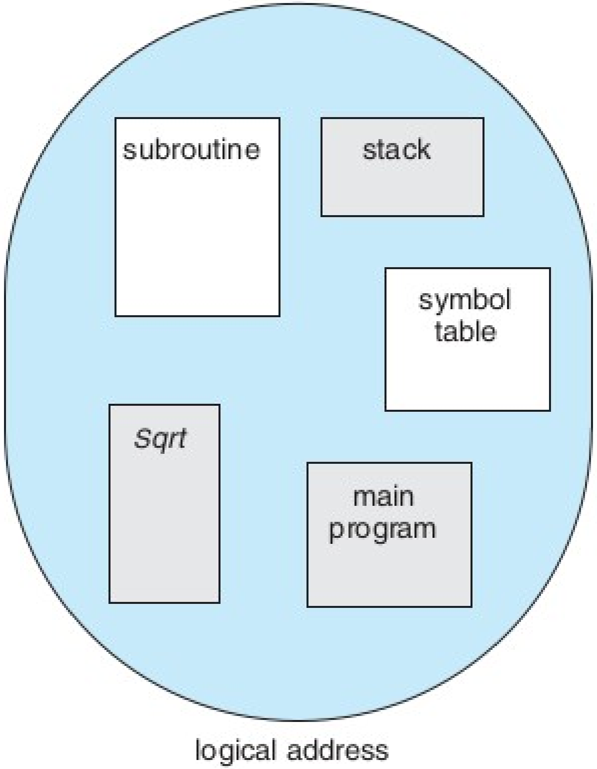
BSS: Uninitialized data, globals that were not initialized

Stack: Region for allocating local variables, function parameters, return values

Heap: Region for use by malloc(), free()

Arguments, Environment variables: Initialized by kernel (during exec)

Why the Segment:Offset Pairs In x86 ?



Segmentation
Compiler’s view of the program

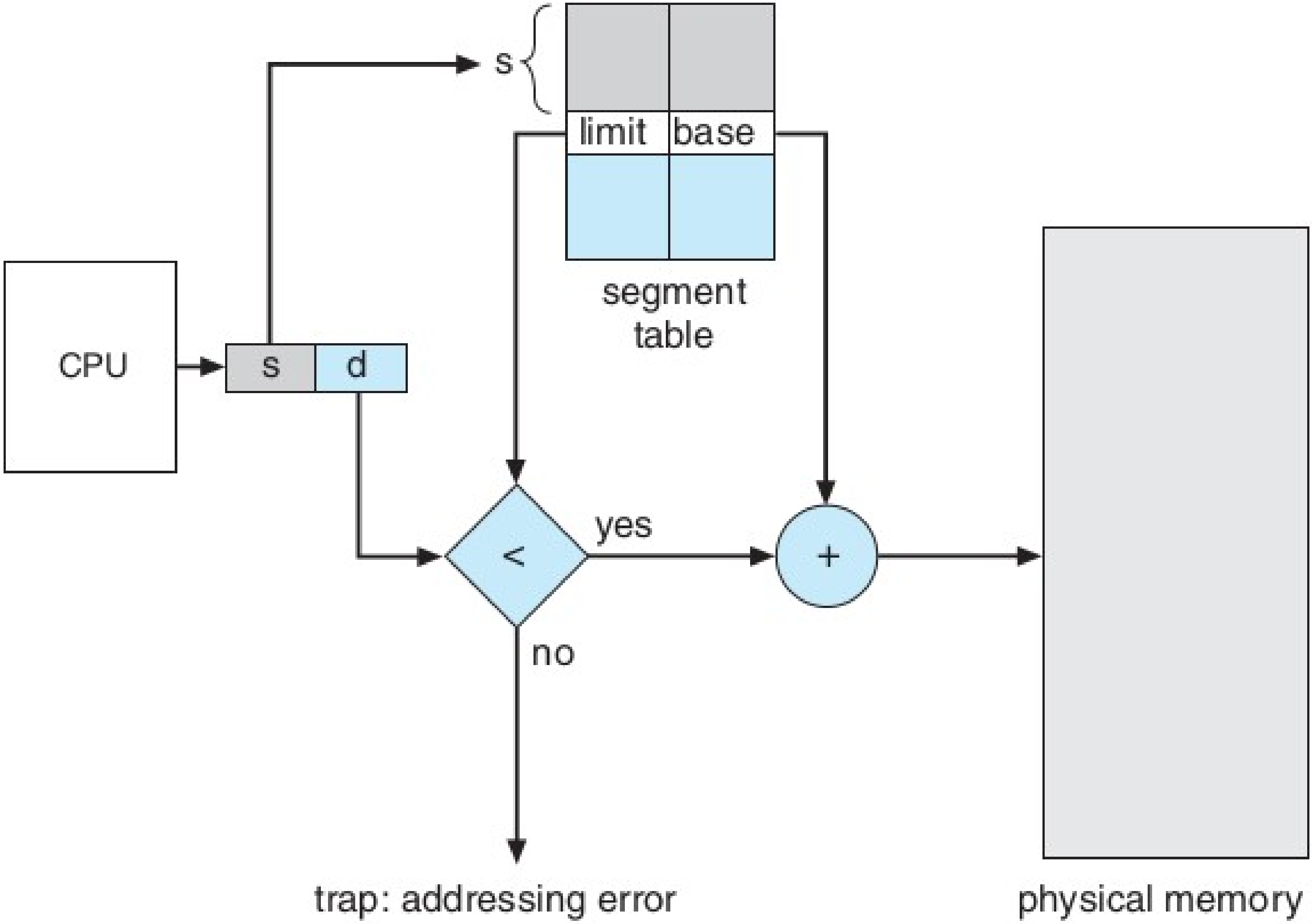
Compiler generates object code **assuming** that different memory regions corresponding to the program are different “segments” in memory

Segment: a continuous chunk

Segment register gives the location of the chunk, index/offset register gives the offset

E.g. in CS:IP CS gives the location of the segment, IP gives the index in it

Segmentation
Address Translation, general idea (not x86)



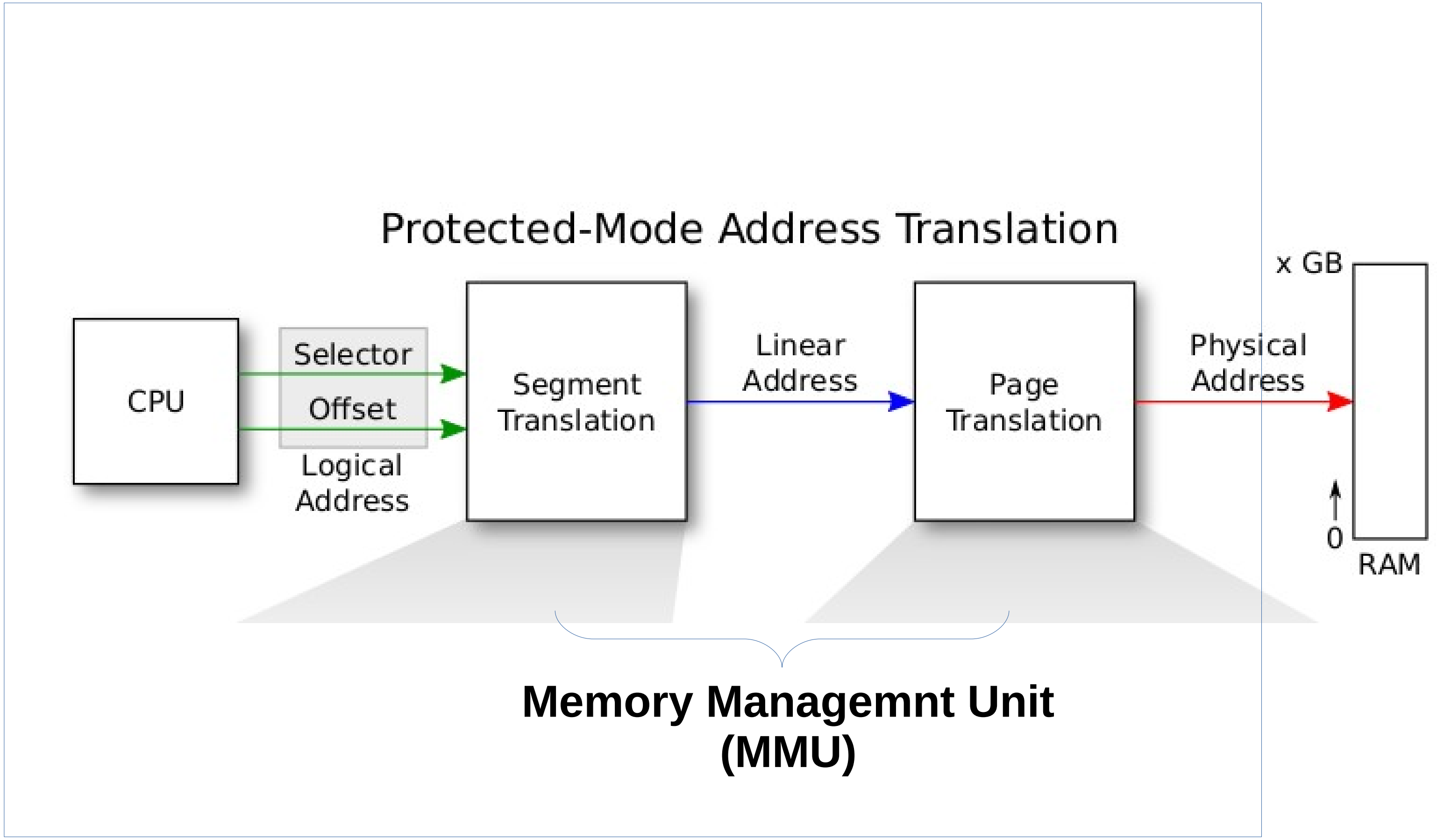
Real mode and protected mode

- Beware: note the same as user mode and kernel mode!
- Backward compatibility was desired by Intel
  - 80286 was 32 bit, 8086 was 16 bit
  - Binary encoding for 80286 was different, registers were 32 bit, etc; also more speed, different memory management hardware, etc.
  - But still they wanted their customers to keep running earlier object code on new processor
  - So they ensured that the processor boots up as if it was 16 bit 8086/8088. REAL MODE!
  - On running particular machine instruction sequence, the CPU will change to 32 bit . PROTECTED MODE!

More on real mode

- addresses in real mode always correspond to real locations in memory.
  - Memory address calculation done by MMU in real mode: segment*16+offset
- no support for memory protection, multitasking, or code privilege levels (user/kernel mode!)
- May use the BIOS routines or OS routines
- DOS like OS were written when PCs were in the era of 8088/real-mode.
  - Single tasking systems

X86 address : protected mode address translation



Memory Managemnt Unit (MMU)

Processor Chip

X86 segmentation

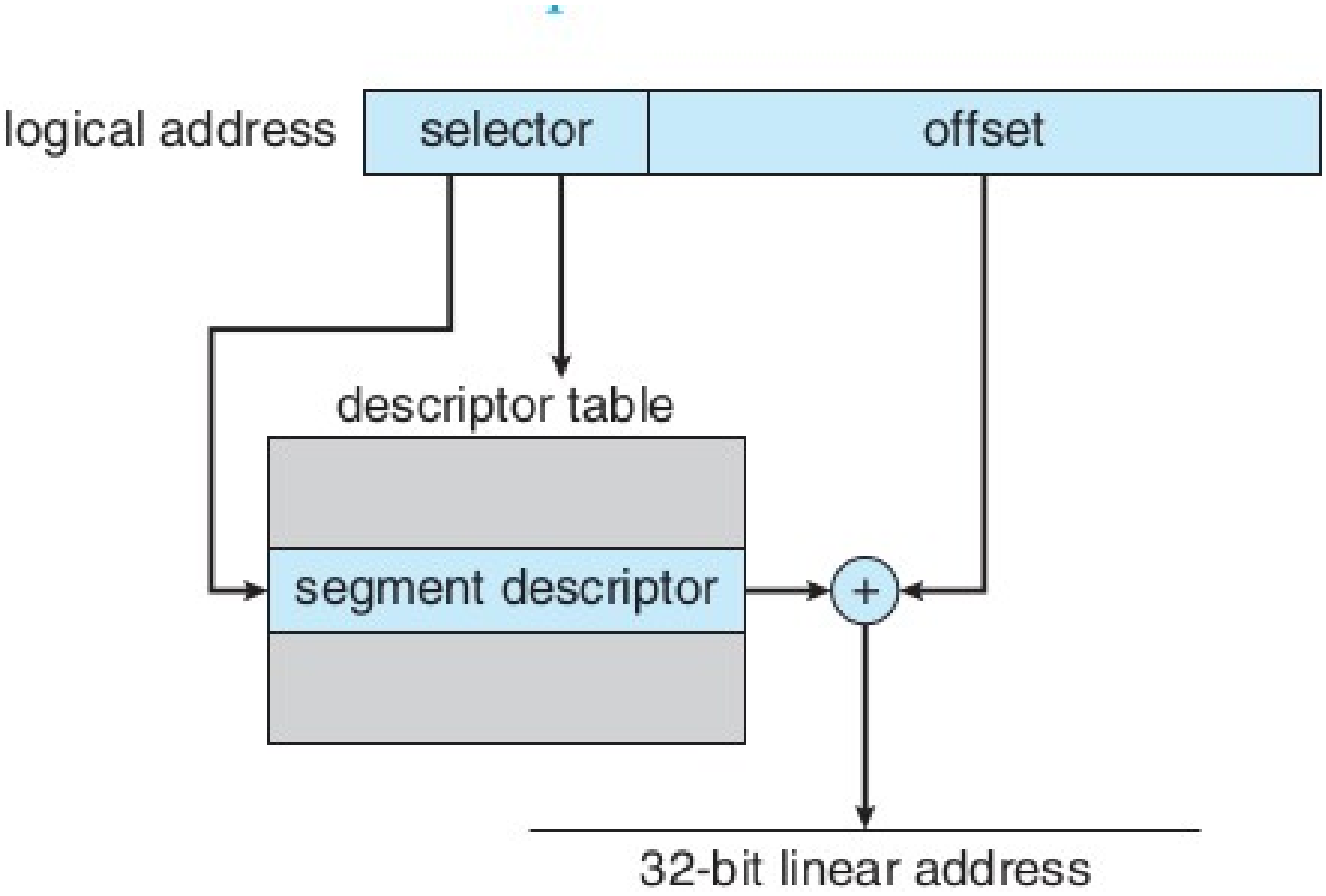


Figure 8.22 IA-32 segmentation.

X86 paging

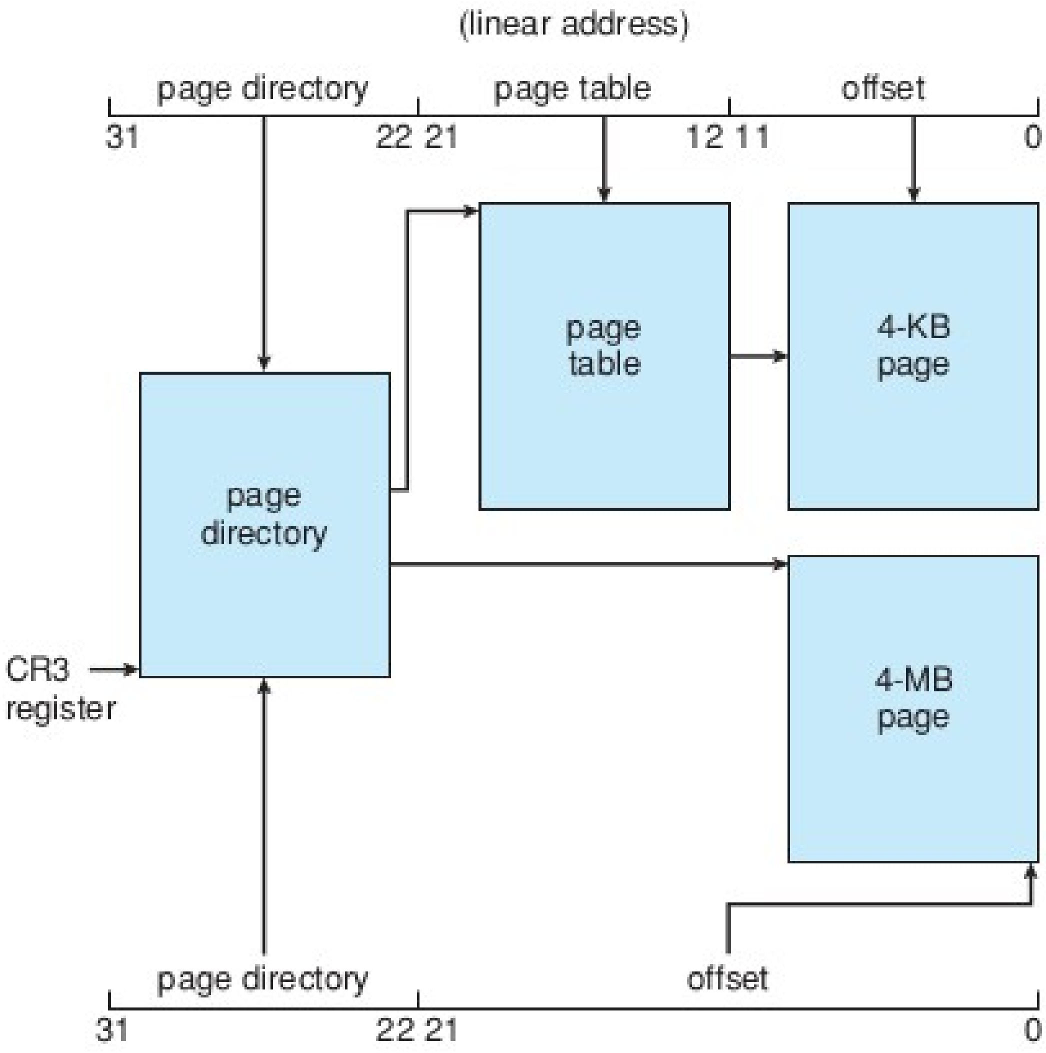
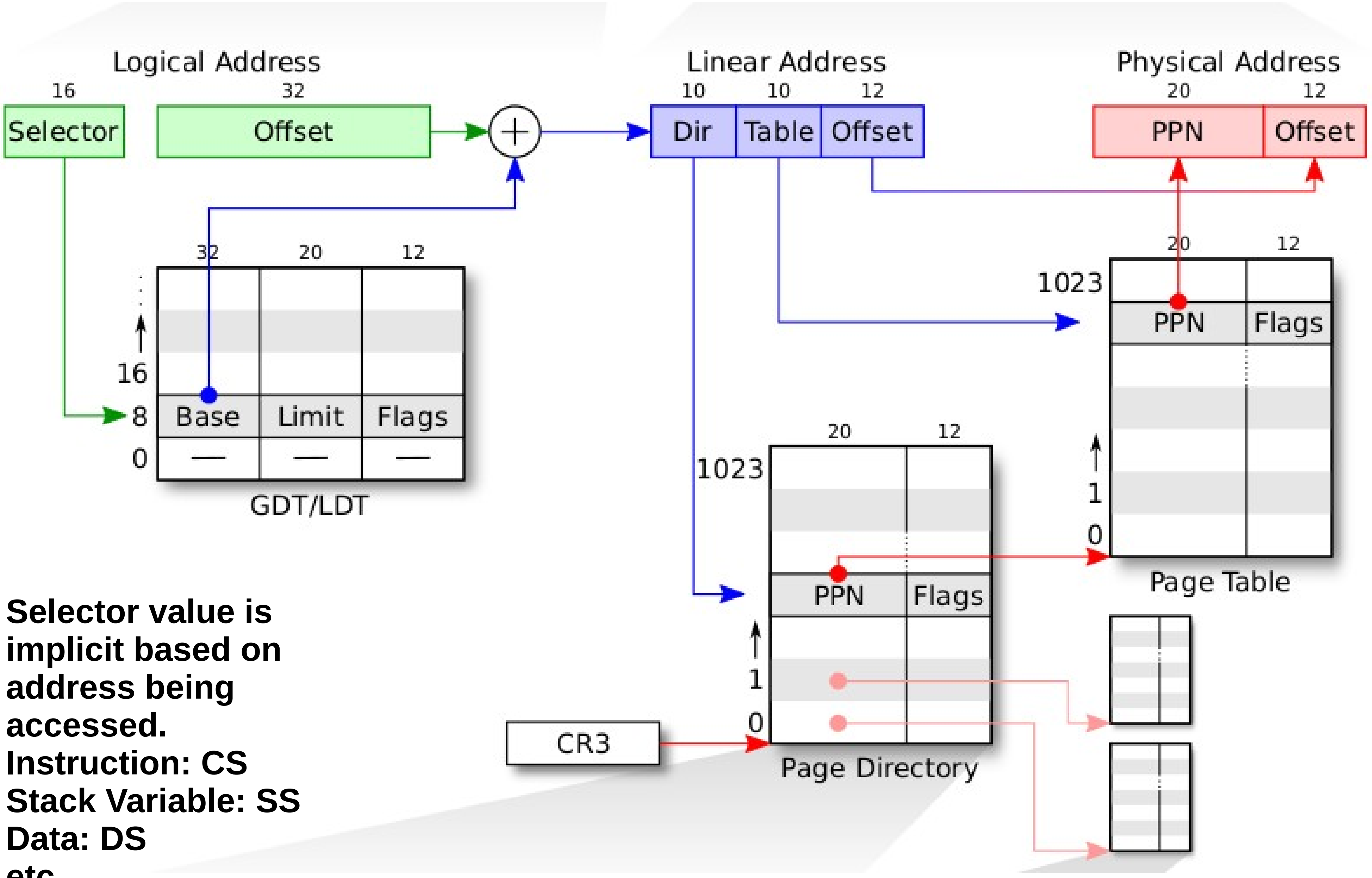


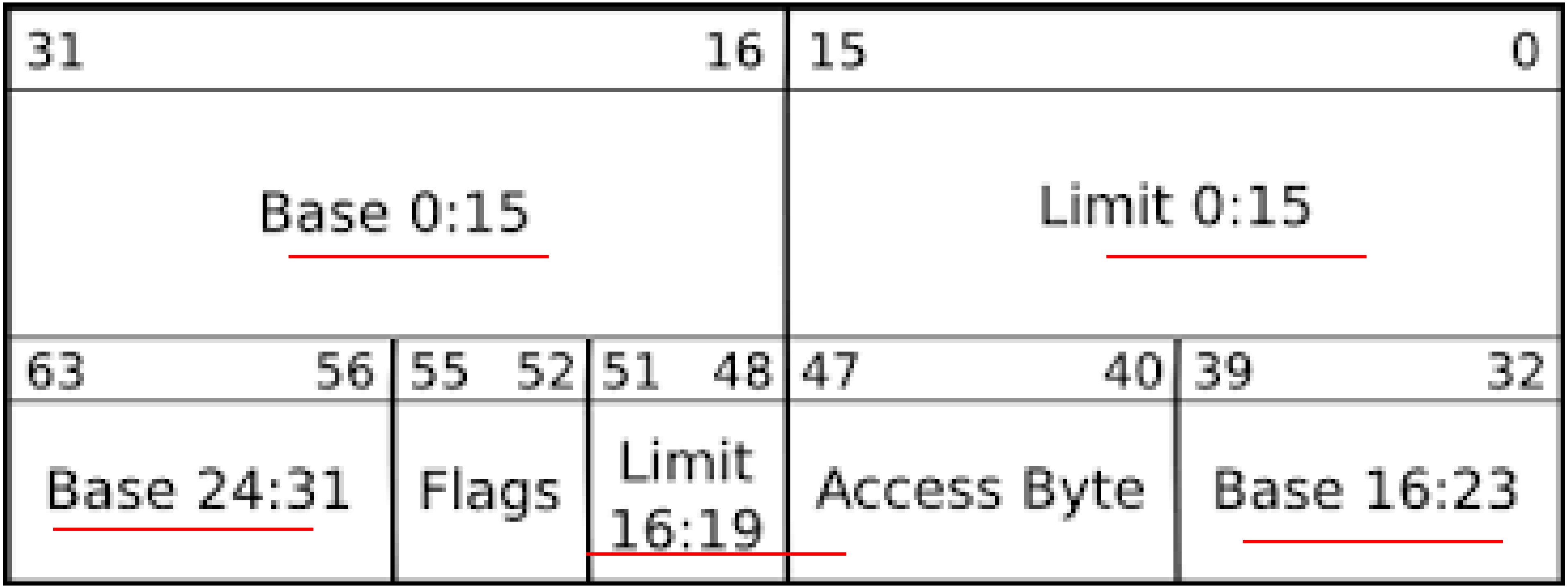
Figure 8.23 Paging in the IA-32 architecture.

Segmentation + Paging



Selector value is implicit based on address being accessed.
Instruction: CS
Stack Variable: SS
Data: DS
etc.

GDT Entry

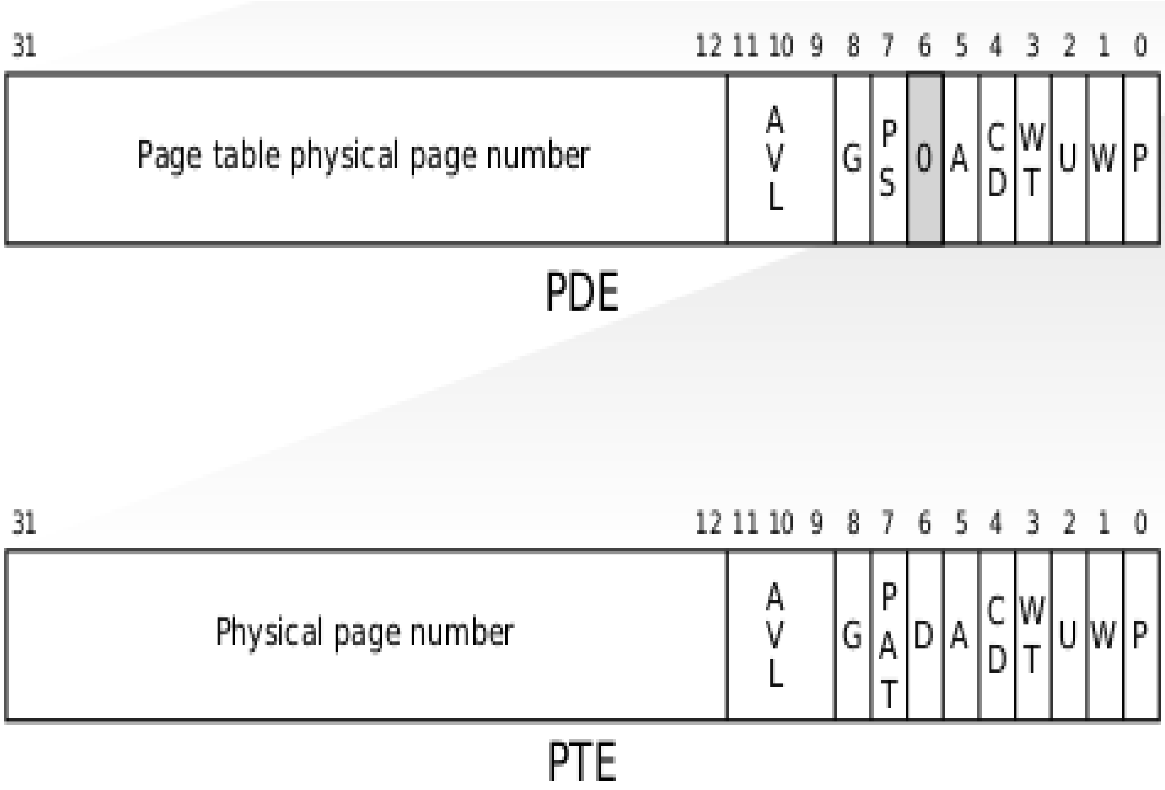


31 16	15 0
Base 0:15	Limit 0:15

63 56	55 52	51 48	47 40	39 32
Base 24:31	Flags	Limit 16:19	Access Byte	Base 16:23

This is an in Memory entry. The diagram shows the format.

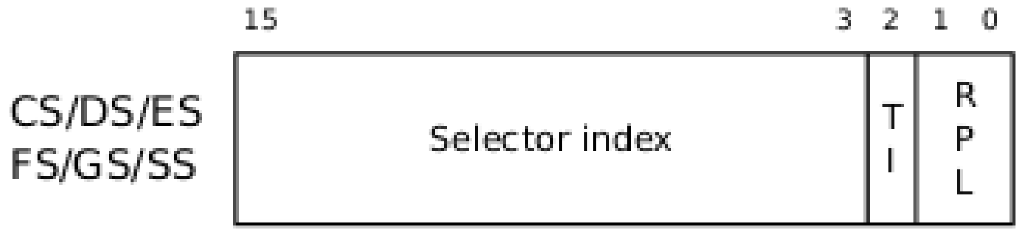
Page Directory Entry (PDE)
Page Table Entry (PTE)



P	Present
W	Writable
U	User
WT	1=Write-through, 0=Write-back
CD	Cache disabled
A	Accessed
D	Dirty
PS	Page size (0=4KB, 1=4MB)
PAT	Page table attribute index
G	Global page
AVL	Available for system use

These are in memory entries. The diagrams show the format of each entry.

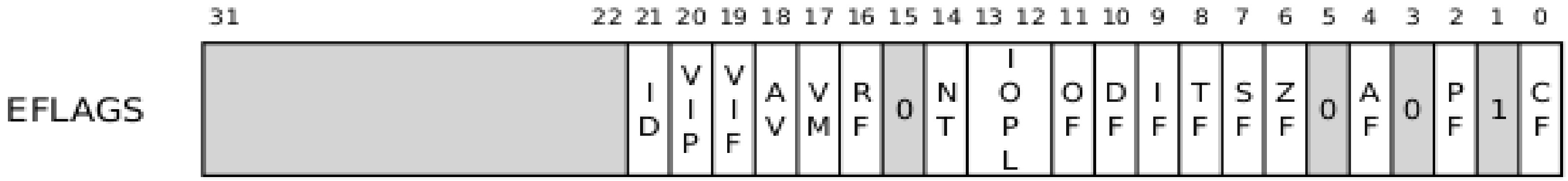
Segment selector



	15–3	2	1–0
CS/DS/ES FS/GS/SS	Selector index	TI	RPL

TI	Table index (0=GDT, 1=LDT)
RPL	Requester privilege level

EFLAGS register



	31–22	21	20	19	18	17	16	15	14	13 12	11	10	9	8	7	6	5	4	3	2	1	0
EFLAGS		ID	VIP	VIF	AV	VM	RF	0	NT	IOPL	OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

Status flags

CF	Carry flag
PF	Parity flag
AF	Auxiliary carry flag
ZF	Zero flag
SF	Sign flag
OF	Overflow flag

Control flags

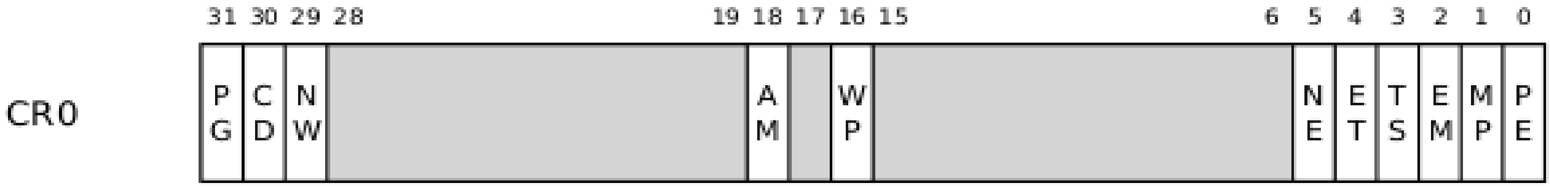
DF	Direction flag

System flags

TF	Trap flag
IF	Interrupt enable flag
IOPL	I/O privilege level
NT	Nested task

RF	Resume flag
VM	Virtual-8086 mode
AC	Alignment check
VIF	Virtual intr. flag
VIP	Virtual intr. pending
ID	ID flag

CR0



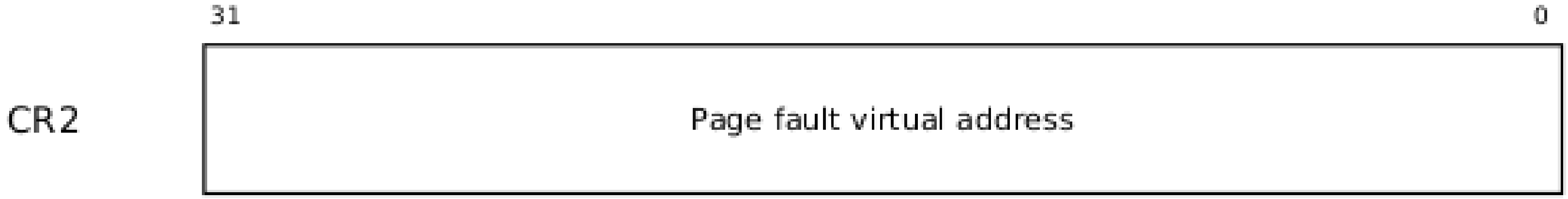
PE	Protection enabled	ET	Extension type	NW	Not write-through
MP	Monitor coprocessor	NE	Numeric error	CD	Cache disable
EM	Emulation	WP	Write protect	PG	Paging
TS	Task switched	AM	Alignment mask		

PG: Paging enabled or not

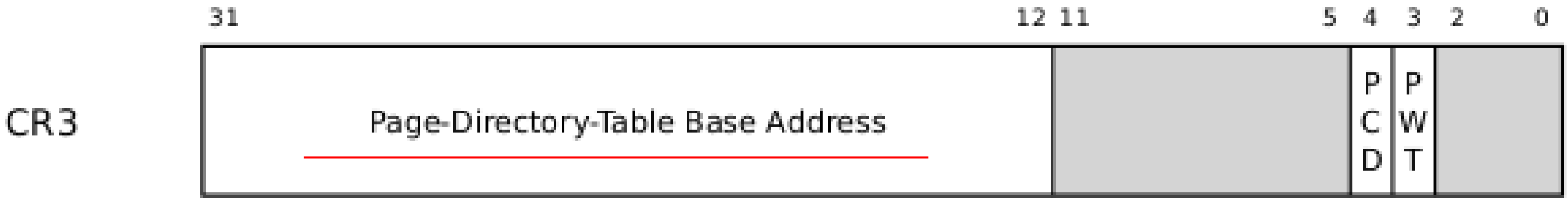
WP: Write protecion on/off

PE: Protection Enabled --> protected mode.

CR2

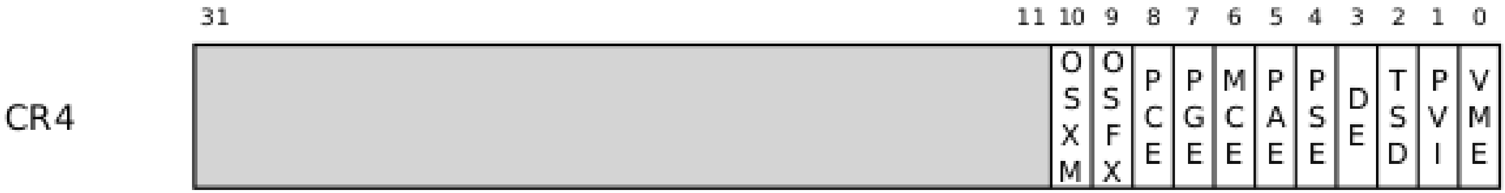


CR3



PWT	Page-level writes transparent	PCT	Page-level cache disable

CR4



VME	Virtual-8086 mode extensions	MCE	Machine check enable
PVI	Protected-mode virtual interrupts	PGE	Page-global enable
TSD	Time stamp disable	PCE	Performance counter enable
DE	Debugging extensions	OSFXSR	OS FXSAVE/FXRSTOR support
PSE	Page size extensions	OSXMM-EXCPT	OS unmasked exception support
PAE	Physical-address extension		