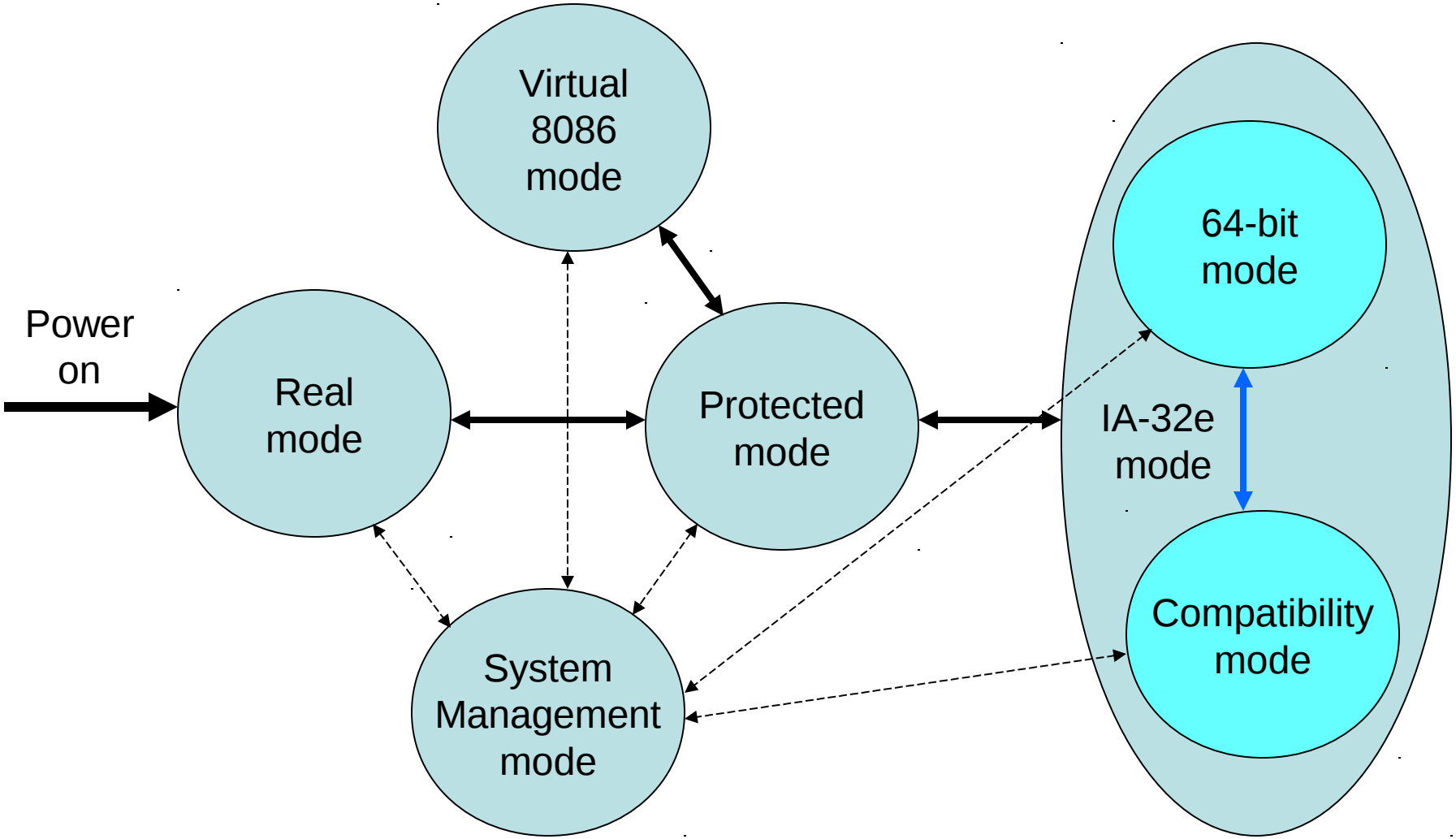
The various x86 ‘modes’

On understanding key differences among the processor’s several execution-architectures

The x86 operating modes



Why ‘mode’ matters

- Key differences among the x86 modes:
  - How memory is addressed and mapped
  - What instruction-set is available
  - Which registers are accessible
  - Which ‘exceptions’ may be generated
  - What data-structures are required
  - How task-switching can be accomplished
  - How interrupts will be processed

Mode transitions

- The processor starts up in ‘real mode’
- Mode-transitions normally happen under program control (except for transitions to System Management Mode)
- Details of programming a mode-change depend on which modes are involved
- Some mode-transfers aren’t possible (and some mode-changes aren’t documented)

Classroom/Lab machines

- The workstations in our Kudlick classroom and Fifth-Floor CS Labs use slightly older Pentium processors that lack the circuitry for supporting the IA-32e mode
- But the new Core-2 Duo machines, which we can access remotely, do implement all of the ‘modes’ we previously depicted

Enabling ‘protected-mode’

- Protected-mode was first introduced in the 80286 processor (used in the IBM-PC/AT)
- Intel added some special system registers to support protected-mode, and to control the transition from the power-on ‘real-mode’
  - Global Descriptor Table register (GDTR)
  - Interrupt Descriptor Table register (IDTR)
  - Local Descriptor Table register (LDTR)
  - Task Register (TR)
  - Machine Status Word (MSW)

Machine Status Word

Legend:

PE (Protected-Mode Enabled): 0=no, 1=yes
MP (Math-coprocessor Present): 0=no, 1=yes
EM (Emulate Math-coprocessor): 0=no, 1=yes
TS (Task has been Switched): 0=no, 1=yes

This register's defined bits would initially be zeros at system startup, but they could be modified under program control by executing the special LMSW instruction (Load Machine Status Word), or could be inspected by executing the SMSW instruction (Store Machine Status Word)

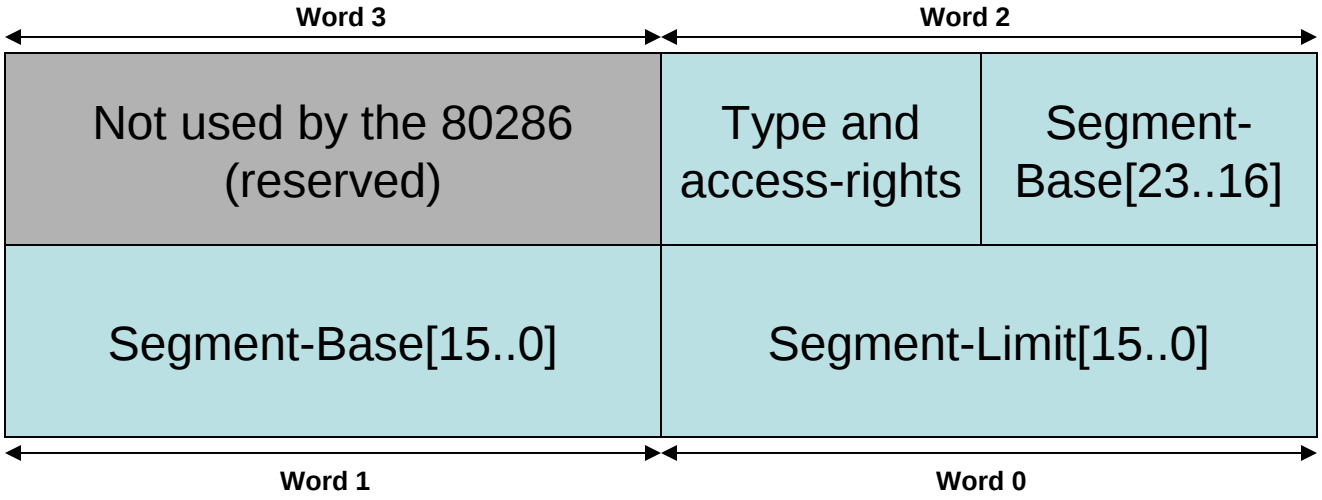
Coding the transition

- So here's the code-fragment for switching to 'protected-mode' from 'real-mode':

```
# This code illustrates an 80286 processor's transition from real-mode
# to protected-mode – but note that interrupts must not be allowed here

        cli                     # clear IF-bit in FLAGS register

        smsw    %ax             # get current settings from MSW
        or      $1, %ax         # set the PE-bit's image to 1
        lmsw    %ax             # load the new setting into MSW

# OK, the processor is now executing in 'protected-mode'
```

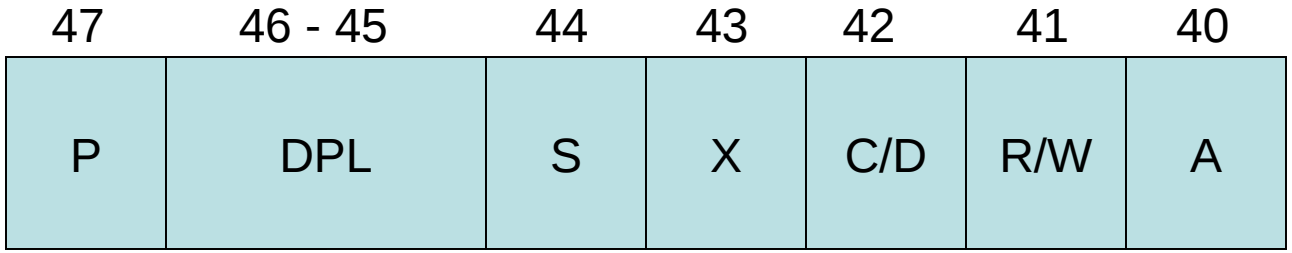
Descriptor tables

- Memory-addressing in protected-mode is based upon some special data-structures called ‘Segment Descriptors’ that define a memory-region and assign its properties



The quadword descriptor data-structure (64-bits)

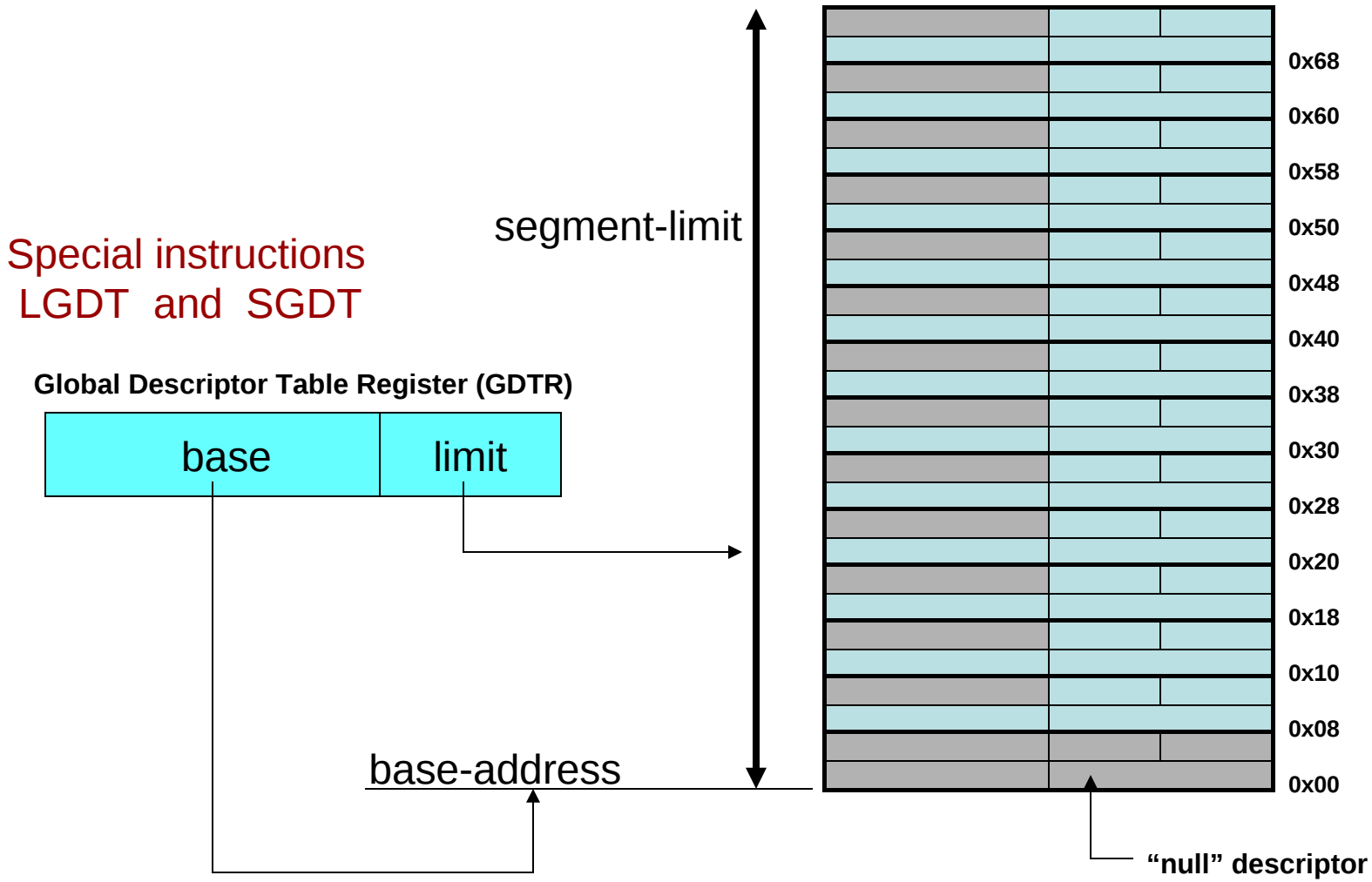
Type and access-rights



Legend:

P = Present (1=yes, 0=no)
DPL = Descriptor Privilege Level (00=supervisor, 11=user)
S = System-segment (0=yes, 1=no)
X = eXecutable: 1=yes (i.e., code-segment), 0=no (i.e., data-segment)
C/D = Conforming code (1=yes, 0=no) when X-bit equals 1
expands-Down (1=yes, 0=no) when X-bit equals 0
R/W = Readable (1=yes, 0=no) when X-bit equals 1
Writable (1=yes, 0=no) when X-bit equals 0
A = segment has been Accessed by the CPU (1=yes, 0=no)

Global Descriptor Table

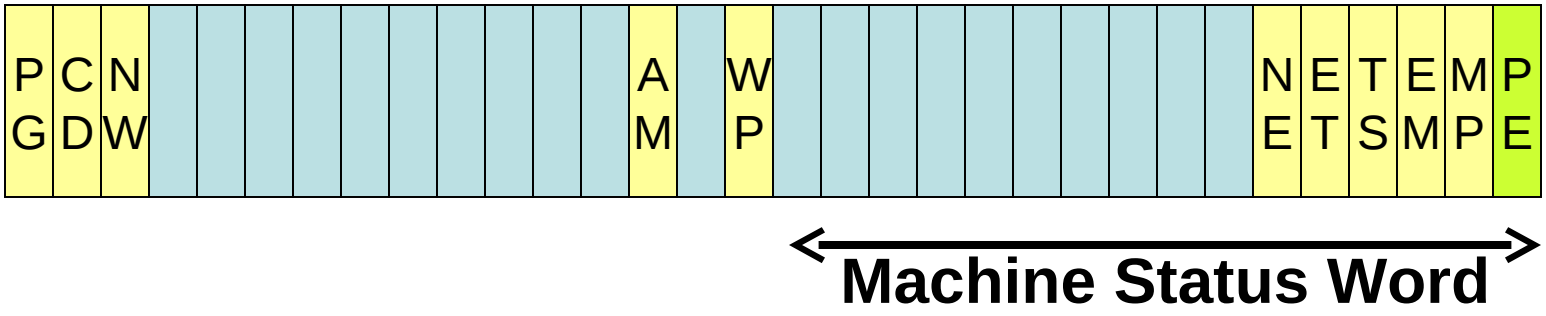


Backward compatibility

- Intel’s 80386 (and later) processors are ‘backwardly compatible’ with the 80286, but they implement some extensions to support 32-bit registers and addresses (and some early forms of ‘virtualization’)
- Register MSW is enlarged and renamed, (and its ‘stickly PE-bit’ design-flaw is corrected in the renamed mechanism)

Control Register 0

- Register CR0 is the 32-bit version of the old MSW register (Machine Status Word)
- It contains the PE-bit (Protection Enabled)
  - when PE=0 the CPU is in real-mode
  - when PE=1 the CPU is in protected-mode



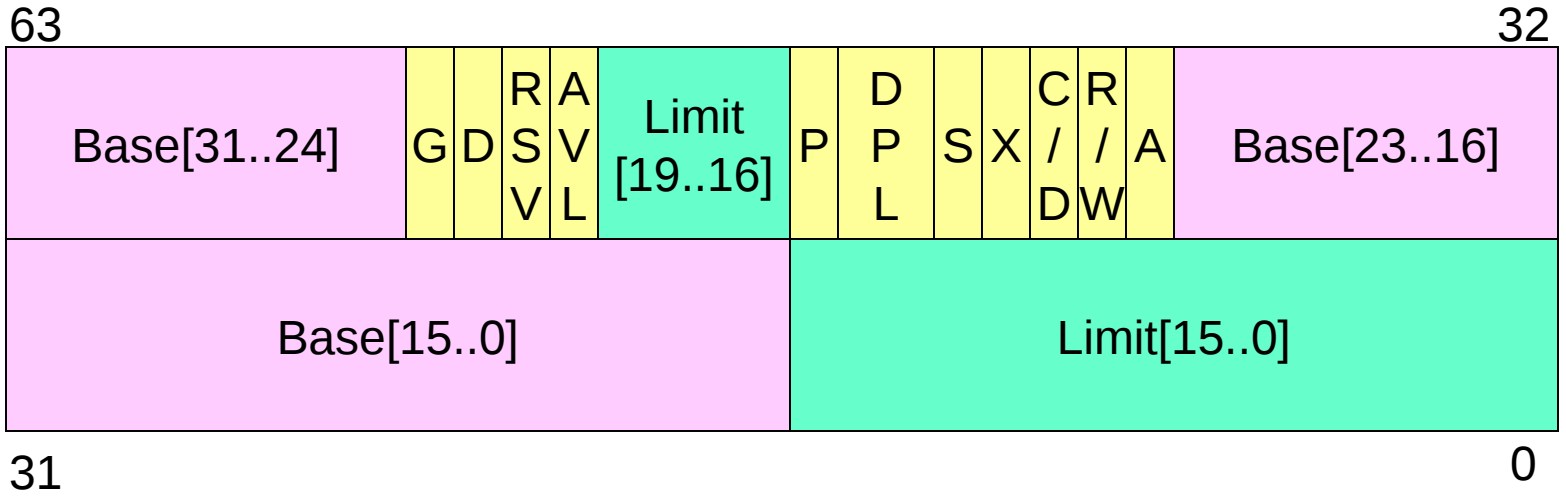
Enter or leave protected-mode

- Here are code-fragments for entering, and for leaving, protected-mode on 32-bit CPU

```
# entering protected-mode (with interrupts turned off)
        mov     %cr0, %eax      # get current Machine Status
        bts     $0, %eax        # set image of bit #0 (the PE-bit)
        mov     %eax, %cr0      # enter protected-mode
```

```
# leaving protected-mode (with interrupts turned off)
        mov     %cr0, %eax      # get current Machine Status
        btr     $0, %eax        # reset image of bit #0 (the PE-bit)
        mov     %eax, %cr0      # leave protected mode
```

Enhanced Descriptor-Format



Legend:

G = Granularity: (0=byte-granularity, 1=page-granularity)
D = Default operand and address size: (0=16-bits, 1=32-bits)
RSV = Reserved bit (but recently used for 64-bit technology)
AVL = Available (this bit can be used by programmers for any purpose)

Several instances of this basic **'segment-descriptor'** data-structure will occur in the **Global Descriptor Table** (and maybe also in some **Local Descriptor Tables**)

Initializing register GDTR

- Setting up your Global Descriptor Table might be accomplished as in this fragment

```
# We shall assume that this segment resides at memory-address 0x10000
            …
            lgdt      regGDT                        # load system register GDTR
            …
#-------------------------------------------------------------------------------------------
theGDT: .quad     0x0000000000000000      # the required ‘null’ descriptor
            .quad     0x008F92000000FFFF      # 4GB writable data-segment
            .quad     0x00009A010000FFFF      # 64KB readable code-segment
            .equ      limGDT, (. – theGDT) – 1  # segment-limit for this GDT
#-------------------------------------------------------------------------------------------
regGDT: .word     limGDT, theGDT, 0x0001  # image for register GDTR
#-------------------------------------------------------------------------------------------
```

Examining register CR0

- We could modify our ‘eflags.s’ program so it would display the current value in CR0
- Just remember that CR0 is the renamed Machine Status Word register, then use the (unprivileged) ‘smsw’ instruction – but use an ‘l’ suffix (for longword) and a 32-bit register-operand (e.g., %eax), like this:

```
smswl    %eax    # store current  value from register CR0 into EAX
```

In-class exercise #1

- Make a copy of the ‘eflags.s’ demo (from our class website), and name it ‘cr0.s’
- Replace the two instructions that it uses to get the value of EFLAGS with just a single instruction that will get the value of CR0
- Change the message-string appropriately
- Then assemble, link, and execute your modified demo-program

In-class exercise #2

- Can you write a program that will show the contents of the system-register GDTR?
- You need to use the SGDT-instruction
- But it will require a memory-operand big enough to hold this register's 48-bits
- And you will need to adjust the counter in your program-loop for 48 binary-digits, as well as your message-string's format