Windows Kernel Internals II
x86 overview
Traps, Interrupts, Exceptions

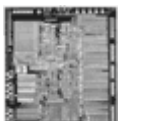
Eric Traut
Director
Windows Core Operating Systems Division
Microsoft Corporation

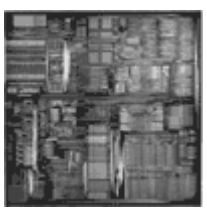
X86 Tutorial Part 1: The Basics

- Overview
  - x86 Family Tree
  - Notation & Data Types
  - Functional Unit & Register Overview
- Reading x86 assembly
- Calling Conventions
- x86 Instructions
  - Instruction Types
  - Instruction Encodings
- Floating Point Unit
- Vector Unit
- Atomic Instructions
- Processor Modes

X86 Family Tree





- 8080 (1974, 6K transistors, 2MHz)
- 8086/8088 (1978, 29K transistors, 5-10MHz)
- 80286 (1982, 134K transistors, 6-12.5MHz)
- 80386 (1985, 275K transistors, 16-33MHz)
- 80486 (1989, 1.2M transistors, 25-50MHz)
- Pentium (1993, 3.1M transistors, 60-66MHz)
- Pentium Pro (1995, 5.5M transistors, 90-200MHz)

X86 Family Tree (continued)

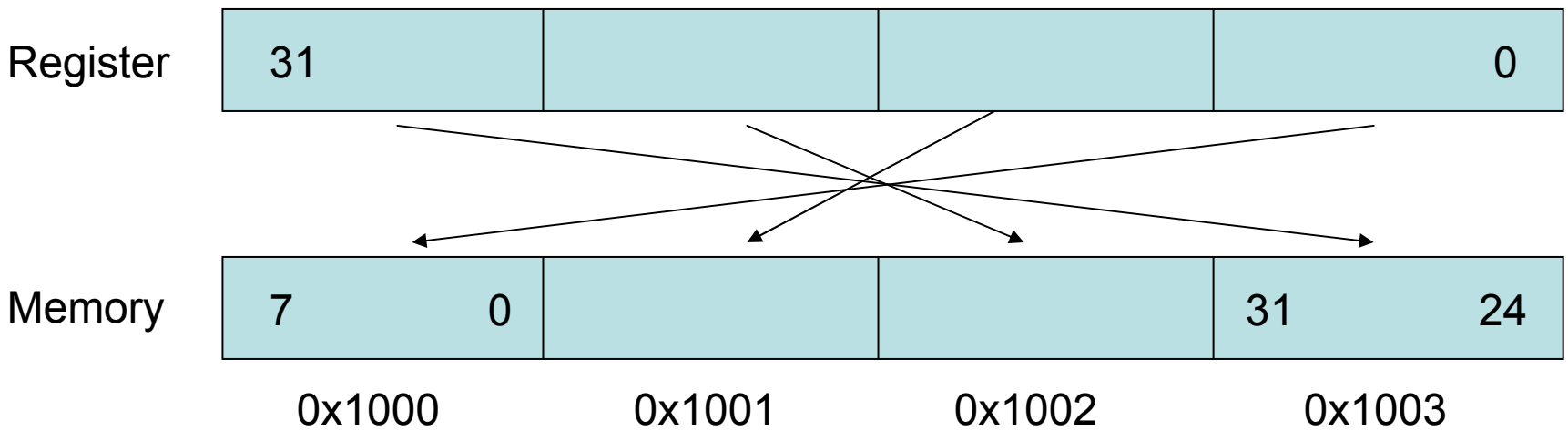
- Pentium II (1997, 3.1M transistors, 200-300MHz)
- Pentium III (1999, 9.5M transistors, 650MHz-1.2GHz)
- Pentium 4 (2000, 42M transistors, 1.3-3GHz)
- Prescott & x64

x86 Data Types

- Integers
  - 1 byte
  - 2 byte (word)
  - 4 byte (d-word)
  - 8 byte (q-word)
- Floating point
  - Single (32-bit)
  - Double (64-bit)
  - Extended (80-bit)
- MMX
  - 64-bit integer (1, 2, and 4-byte) vectors
- SSE
  - 128-bit single FP vectors
- SSE2
  - 128-bit integer (1, 2, 4 and 8-byte), single and double vectors

Bit and Byte Conventions

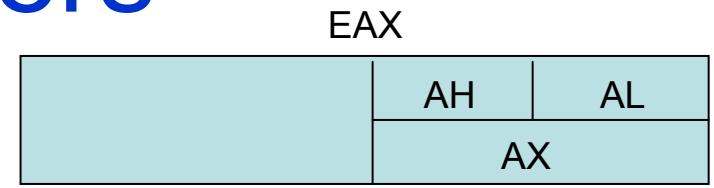
- All data stored in little endian format
- Bits counted from the least significant bit



X86 Notation (cont)

- Pointer notation XS:[Addr]
  - Far pointer versus near pointer
  - XS is a “segment” (has a base and a limit)
  - Addr is the effective address (offset into the segment)
  - Linear address = XS.base + Addr
- In Windows NT (Win32 subsystem), base is always 0 and limit is always 0xFFFFFFFF

32-bit General Purpose Registers



- Byte-addressable registers
  - EAX / AX / AH / AL : Implicit argument for many instruction forms
  - EDX / DX / DH / DL : Used in conj. with EAX for long operands
  - ECX / CX / CH / CL : Count register used for string ops
  - EBX / BX / BH / BL :
- Word-addressable registers
  - ESP / SP : Stack pointer
  - EBP / BP : Frame pointer
  - ESI / SI : Source pointer for string ops
  - EDI / DI : Dest. pointer for string ops

Segment Registers

- CS : code segment
- SS : stack segment
- DS : data segment
- ES : destination of string op
- FS : extra segment (thread local storage in Windows)
- GS : extra segment (generally unused)

Miscellaneous User Registers

- EIP / IP : “instruction pointer” (offset into code segment)
  - To get the linear address of the current instruction, add EIP to the base of the CS segment
- EFLAGS / FLAGS
  - Condition codes (ZF, SF, CF, OF, PF, AF)
  - TF : Generate trace exception (single step)
  - DF : Direction of string op
  - IF : Mask interrupts
  - Much more…

Instruction Mnemonics

- Most instructions take two operands
  - First operand is destination
  - Second operand is source
  - DEST ← DEST *op* SRC
- Brackets (generally) indicate a dereference

- Examples:

```
add     edx,ebx                 ; add ebx to edx
mov     ecx,[esp + 4] ; load a value from the stack
```

Instruction Mnemonics (cont)

- If segment is not explicit, default is used

```
add     dword fs:[eax],3     ; Override default segment DS
mov     ecx,ds:[esp]  ; Override default segment SS
```

- Size of operation is implicit except where not obvious

```
add     ax,3                 ; 16-bit operation
push    [eax]                ; Indeterminate size (assembler error)
```

Instruction Mnemonics (cont)

- Effective address can include base, index and scale (x1, x2, x4 or x8)

```
mov         eax,[esi + eax * 4]
```

- LEA can be used to calculate effective address (and do math cheaply)

```
lea eax,[eax + eax * 4]   ; multiply eax by 5
```

Calling Conventions

- No architecturally-specified or recommended calling conventions
- Several “standards”
  - C Call (_cdecl)
    - This Call
  - Standard Call (_stdcall)
  - Fast Call (_fastcall)

“C Call” Calling Conventions

- Standard convention for most C and C++ compilers
- Parameters pushed right to left
- Calling function pops the arguments
- “This Call” convention used for C++ instance methods

```
void CDeclFunction (char * p1, long p2, char p3) ;

CDeclFunction ( s_ptr, 2 , 'a' )

push    dword 0x61
push    dword 2
push    eax
call    _CDeclFunction
add     esp,0x0C
```

“Standard Call” Calling Conventions

- Used for Win32 calls
- Parameters passed right to left
- Called function pops arguments from stack

```
void StdCallFunction (char * p1, long p2, char p3) ;

StdCallFunction (s_ptr, 2 , ‘a');

push    dword 0x61
push    dword 2
push    eax
call    _StdCallFunction@9
```

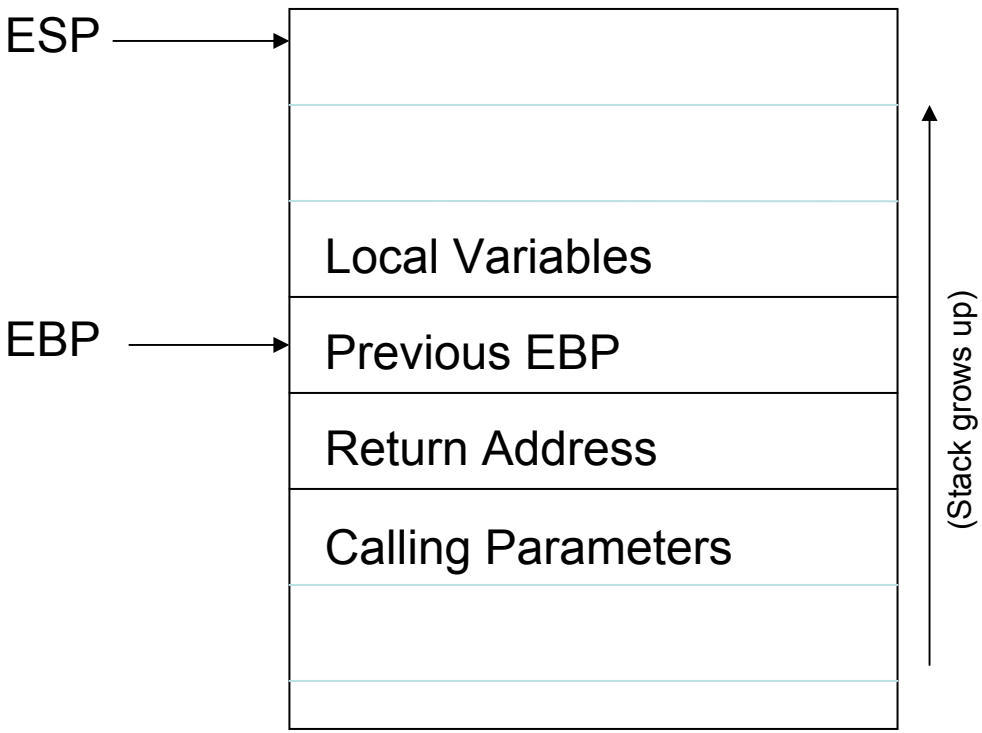
“Fast Call” Calling Conventions

- Pass parameters in registers where possible
- First DWORD parameters are passed in ECX and EDX, the rest are pushed onto the stack right to left
- Called function pops arguments from the stack

```
void FastCallFunction (char * p1, long p2, char p3) ;

FastCallFunction (s_ptr, 2 , ‘a');

push    dword 0x61
mov     edx,2
mov     ecx,eax
call    FastCallFunction@9
```

Stack Frames

- ESP points to top of stack
- EBP is often used to point to current stack frame



Creating a stack frame:

```
push   ebp
mov    ebp,esp
sub    esp,0x0C
```

Destroying a stack frame:

```
mov    esp,ebp
pop    ebp
```

String Operations

- Store, load, copy, compare, scan
- Work with REP, REPE, REPNE “prefixes”
  - Count is in ECX
  - ESI and/or EDI are incremented or decremented

- String store example:

```
cld                                     ; Move in forward direction
mov         ecx,30              ; Set up rep count
mov         edi,bufferPtr       ; Set up initial pointer
xor eax,eax             ; Clear eax
rep stosd                           ; Clear 120 bytes to zero
```

String Operations (cont)

- String copy example:

```
mov       ecx,30          ; Set up rep count
mov       esi,srcPtr              ; Set up source pointer
mov       edi,destPtr             ; Set up dest pointer
rep movsd           ; Copy 120 bytes
```

- String scan example:

```
mov       ecx,30          ; Set up rep count
mov       al,0x20         ; Look for a space
mov       esi,srcPtr              ; Set up source pointer
repne     scasb           ; Search for space
```

Control Flow Instructions

- Jump and call examples:

```
jmp *+32                ; Jump 32 bytes from here
call [eax]              ; Jump to value in eax
jnz          *+8                ; If ZF is clear, jump to *+8
```

- Comparisons with conditional jumps:

```
cmp          eax,ebx; Jump 32 bytes from here
jle   *+20              ; If eax <= ebx (signed), jump
jbe  *+20               ; If eax <= ebx (unsigned), jump
```

le = “less than or equal”, be = “below or equal”

X86 Floating Point

- Floating point stack
  - Operands are pushed on “stack”. Operations pop operands and push result back on.
  - Who thought this was a good idea?
  - No easy way to access specific registers – always relative to the “top of stack”
- Example:

```
fldpi                   ; Push pi onto the stack
faddp ST(3); Add the third value from the top
                        ; to ST(0), pop ST(0), and push
                        ; the result back onto the stack
```

X86 Vector Instructions

- MMX, 3DNow, SSE, SSE2
  - SIMD (single-instruction, multiple data)
  - Operate on “vectors” of integers or floating point data
  - Great for parallelizable algorithms
    - Graphics
    - Signal processing
- Streaming Extensions
  - Aid in data fetching
  - Programmer can specify data size and “stride”
  - Useful for data where normal assumptions about temporal locality do not hold

Atomic Instructions

- Special instructions for atomic actions
  - XCHG – swap (2 or 4 bytes)
  - CMPXCHG – compare and swap (2 or 4 bytes)
  - CMPXCHG8B – compare and swap (8 bytes)
  - ADD / OR / AND / XOR – read, modify, write
  - Add “LOCK” prefix for MP atomicity
- Example: Singly-linked list
  - On entry, edx points to list head, ebx points to new node

```
AddNodeToList :
    mov         eax,[edx]           ; Get current head
    mov         [ebx + fNext],eax   ; Set fNext of node
    lock cmpxchg ebx,[edx]          ; Set new head
    jne         AddNodeToList       ; Try again if it changed
```

Instruction Encodings

- Instructions are variable length (1 to 13 bytes)
- Primary opcode defined by first byte
- Immediate values are encoded within instruction stream
- Most operations are either 8-bit or X-bit where “X” is determined by the current mode of the processor (16-bit or 32-bit)
- Prefixes alter behavior of instruction
  - LOCK, REP, segment overrides, operand size, address size
- Accessing size opposite the current mode requires “operand size override prefix” (0x66)
- Example:

```
add     eax,0x1234    | 05 34120000
add     ax,0x1234     | 66 05 3412
```

ModRM Bytes

- Many instructions use ModRM byte to encode source/dest register and effective address

- Mode:
  - 00 : No immediate offset
  - 01 : Byte-sized offset
  - 10 : Dword-sized offset
  - 11 : Register operand (no memory access)

Processor Modes

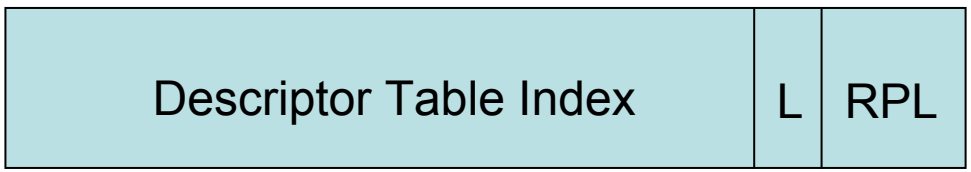
- Real Mode
  - Acts like 8086
  - Simple segment semantics
    - Segment length = 0xFFFF
    - Segment base = segment selector * 16
  - No protection
  - No paging
  - 16-bit addressing modes only (unless overridden)
  - 16-bit operands only (unless overridden)

Processor Modes

- Protected Mode
  - 16-bit versus 32-bit mode
    - controlled by size bit in CS segment descriptor
  - Rings 0 through 3
    - controlled by bottom two bits of CS selector
  - “virtual 8086” mode vs. native mode
    - controlled by v86 bit in EFLAGS register
    - acts like ring 3
  - Paging on versus paging off

Segments in Protected Mode

- Segment registers contain 16-bit selectors
  - Index into descriptor table (global or local, depending on L bit)



  - Each descriptor table entry is 8 bytes and specifies
    - Base
    - Limit
    - Segment type
    - Protection attributes

Segments (Windows Example)

- In kernel mode:
  - CS = 0x0008 (wide-open code segment)
  - DS = SS = 0x0010 (wide-open data segment)
- In user mode:
  - CS = 0x001B (wide-open code segment)
  - DS = SS = 0x0023 (wide-open data segment)
  - FS = 0x003B (small data segment)

Lord of the Rings

- Ring 3
  - User mode MMU semantics
  - Restricted access (can't execute privileged instructions)
- Ring 1 & 2
  - Privileged mode MMU semantics
  - Restricted access (can't execute privileged instructions)
- Ring 0
  - Full access to processor state

System Management Mode

- Used for power management and fixing bugs in hardware
- Similar to real mode, but with wide-open segments
- SMIs are generated by chipset (e.g. in response to I/O port accesses)
- SMIs (system management interrupts)
  - Processor state completely saved
  - RSM instruction completely restores it

X86-64 Modes

- Long mode
  - 64-bit mode
    - Segmentation goes away (mostly)
    - New general-purpose registers (RAX, etc. + R8-R15)
    - New XMM registers (XMM8-XMM15)
    - RFLAGS and RIP become 64-bit
  - Compatibility mode
    - Allows existing 32-bit and 16-bit applications to run
    - Doesn’t support v86 mode
    - Doesn’t support real mode
    - Assumes 64-bit OS
- Legacy mode
  - Runs as traditional IA32 processor

Additional Information

- Intel Architecture: Software Developer's Manual Vol. 1-3
  - http://www.intel.com/design/pentium/manuals/
- AMD x86-64 Architecture: Programmer's Manual Vol 1-5
  - http://www.amd.com/us-en/Processors/DevelopWithAMD/
- Intel Secrets
  - http://www.x86.org/secrets/

X86 Tutorial Part 2: Advanced Topics

- CPUID
- Timing
- Control Registers
- Debugging Capabilities
- Performance Counters
- I/O
- Exceptions
- Interrupts & APIC
- Paging

CPUID

- Gets information about processor
- Requested type passed in EAX
- Information returned in EAX, EDX, ECX, EBX
- Examples of information
  - Family, stepping
  - Manufacturer (“GenuineIntel”, “AuthenticAMD”, “CyrixInstead”, “ConnectixCPU”)
  - Feature bits (MMX, APIC, PAE, SSE, SSE2)
  - Cache & TLB information (line size, capacity, associativity, etc.)

Timing

- Time Stamp Counter (TSC)
  - Counts at cycle rate of processor
  - 64-bit counter
  - RDTSC instruction reads value into EDX | EAX
  - Readable from within ring 3 (or not)
- Issues with TSC
  - May be reset at any time (e.g. hibernation, synchronization after fault tolerant mirroring)
  - May differ between processors in MP system
  - Counts at rate of processor which can change (e.g. SpeedStep) or stop (in low-power states)

Control Registers

- CR0: specific bits control processor behavior
  - PE = Protected mode enabled?
  - PG = Paging enabled?
  - WP = Write protection applies to ring 0-2?
  - TS = FPU & SSE state is inaccessible
- CR2: page fault linear address
  - Address whose attempted access resulted in a page fault exception
  - Used by page fault handler

Control Registers (cont.)

- CR3: page table base address
  - Physical address points to top level of page tables
- CR4: miscellaneous flags
  - VME = virtual machine extensions enabled? (used only with v86 mode)
  - TSD = can ring-3 code access TSC?
  - PAE = physical address extension enabled?
  - PGE = global pages enabled?
  - PCE = can ring-3 code access perf counters?

Debugging Capabilities

- DR0 – DR7
  - Four Hardware-based breakpoints
    - Execution breakpoints
    - Memory breakpoints
    - I/O breakpoints
  - Can enable/disable independently for user and privileged modes
- Single stepping (TF bit in EFlags)
- INT 3 (0xCC) – single-byte trap
- ICEBPT (0xF1) – trap into ICE (INT 1)

Performance Counters

- Allow software to count events within the processor
- Two to four performance counters depending on processor model
- Examples:
  - Measure data cache miss rate per cycle
  - Measure TLB misses due to instruction cache fetches

Device I/O

- Two ways to “talk to” I/O devices
  - Memory-mapped I/O (MMIO)
    - Accessed with normal memory instructions
  - Programmed I/O (PIO)
    - Accessed with IN & OUT instructions
    - I/O address space is 16-bit
    - I/O addresses traditionally called “I/O ports”
- Legacy versus PnP
  - Newer devices can be programmed to use specified I/O ranges to avoid conflicting with other devices
  - Older devices had hard-coded ranges or dip switches

Interrupt Descriptor Table

- Interrupt Descriptor Table Register (IDTR)
  - Points to 256-entry table (8 bytes per entry)
  - Entries correspond to interrupt / exception vectors
  - Vectors 0x00 through 0x1F are reserved for processor exceptions
  - Vectors 0x20 through 0xFF can be used for external interrupts or software interrupts (INT instructions)
- Each descriptor contains
  - Descriptor privilege level (DPL) - typically ring 0
  - CS and EIP of handler
  - “Gate” type:
    - Trap gates keep interrupts enabled
    - Interrupt gates disable interrupts
    - Task gates swap most of the processor state

Task State Segment

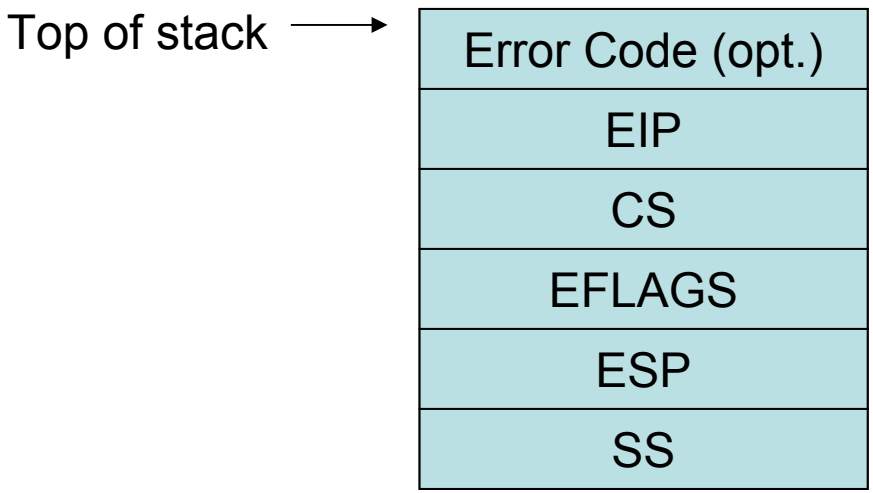
- Task Register (TR)
  - Points to TSS
  - Data structure that contains, among other things:
    - SS & ESP for ring 0 (for trap or interrupt gates)
    - Space to save entire task state (for task gates)
  - I/O protection bitmap
    - Found at end of TSS
    - Contains 64K bits, one per address in I/O address space
    - Enables trapping on specific I/O ports

Exception Processing

- When an exception or interrupt occurs
  - IDT descriptor is accessed
  - If transition to “inner ring” (e.g. ring 3 to ring 0)
    - New SS & ESP are loaded from TSS
  - On destination stack
    - Push SS & ESP (if stack switch occurred)
    - Push EFLAGS
    - Push CS & EIP at time of exception (or next EIP)
  - For some exceptions, push error code
  - Load new CS & EIP
  - Modify EFLAGS
    - Disable v86 (if necessary)
    - Disable single stepping (TF)
    - Disable IF (if interrupt gate)

Exception Handlers

- Pop error code (if necessary)
- Handle exception (e.g. map page or respond to INT instruction)
- Execute IRET to return to excepting instruction



Kernel Calls

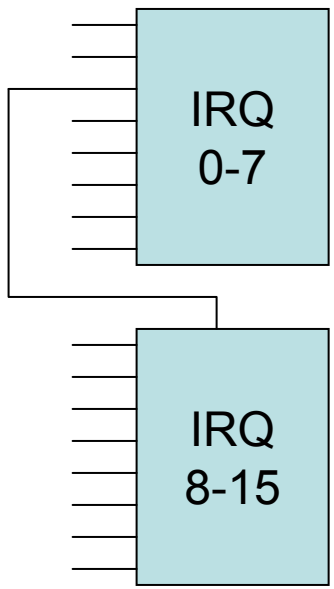
- INT instructions
  - NT traditionally used INT 0x2E
  - ~2500 cycles on Pentium 4 (round trip)
- SYSENTER / SYSEXIT
  - Skip some of the ring transition steps
  - Assumes wide-open segments
  - Assumes ring 3 / ring 0
  - Assumes trap gates
  - Doesn’t push anything on destination stack
  - ~500 cycles on Pentium 4 (round trip)

Interrupts

- Maskable interrupts
  - Can be masked with IF = 0 (in EFLAGS)
  - Generated by external 8259 PIC or internal APIC
  - External devices & inter-processor interrupts (IPIs)
- Non-maskable interrupt (NMI)
- System management interrupts (SMIs)

8259 Interrupt Controller

- Two 8259's wired in "cascade mode"
- Each 8259 has:
  - 8 interrupt request lines (IRQs)
  - Vector base
  - Highest-priority interrupt
- IMR (mask register)
- IRR (request register)
- ISR (service register)
- Command to EOI (end of interrupt)

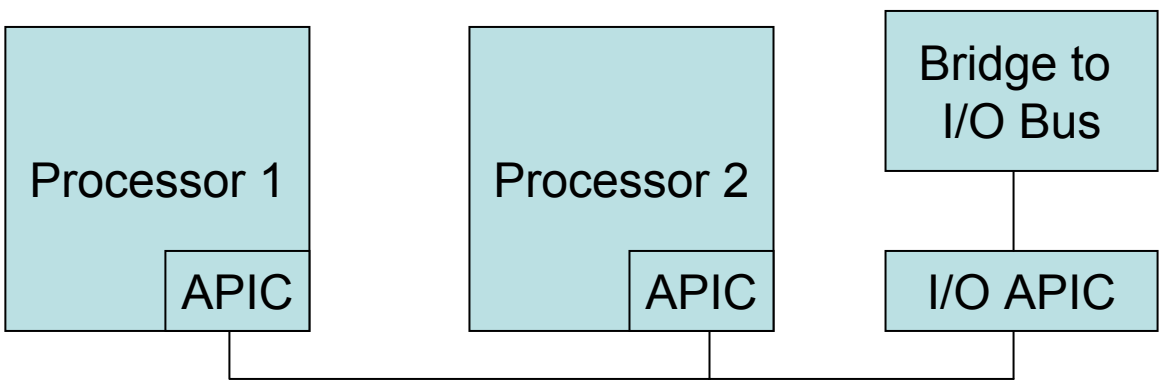


Standard Interrupt Sequence

- Device pulls IRQ high, sets bit in IRR
- PIC determines this IRQ is highest-priority
- PIC notifies processor that interrupt is pending
- If interrupts are unmasked, processor ack's the interrupt and asks for the vector
- PIC returns vector and sets bit in ISR
- Kernel runs the interrupt service routine which tells the device interrupt is handled
- Device lowers IRQ
- HAL sends EOI to PIC
- PIC clears the bit in the ISR

Advanced Programmable Interrupt Controller

- Integrated into the processor
- Works in conjunction with external “I/O APIC”
- Supports IPIs (inter-processor interrupts)
- Supports 224 IRQs (256 minus 32 reserved exception vectors)
- Supports a high-precision timer interrupt
- Memory-mapped registers

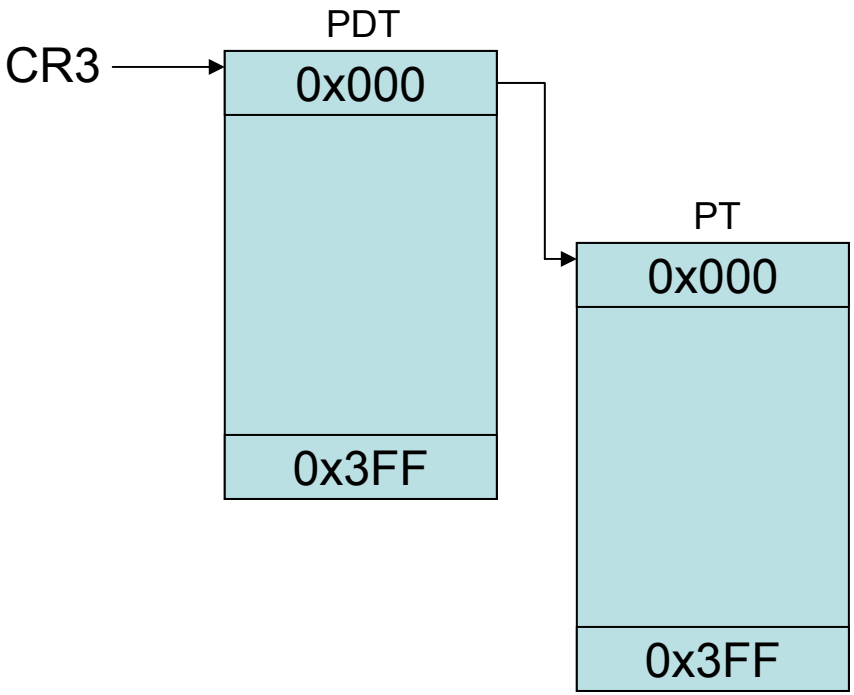


Paging

- Address translation
  - Virtual (logical) addresses
  - Physical addresses
- Translation defined by page tables
- Translations cached by translation look-aside buffer (TLB)
- Granularity of translation is a “page” (4K)

Two-level Page Tables

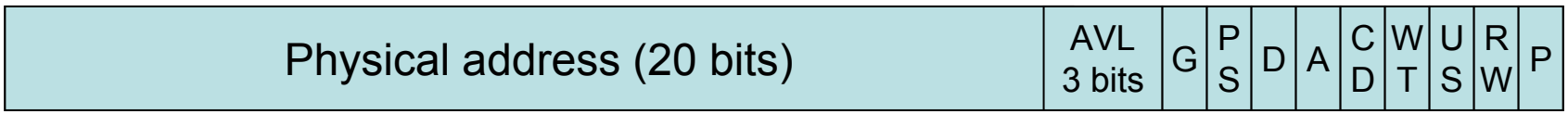
- 32-bit virtual address
  - First 10 bits used to index into page directory table
  - Next 10 bits used to index into page table



PDT index (10 bits)	PT index (10 bits)	Page offset (12 bits)

PDEs and PTEs

- Entries within PDT and PT have almost identical format



G = Global page (mapped into every address space)

PS = Page size (only used in PDE)

D = Dirty (only used in PTE)

A = Accessed

CD = Cache disabled

WT = Cache write-through (versus copy back)

US = 1 indicates user-accessible (versus supervisor only)

RW = 1 indicates writable (versus read-only)

P = 1 indicates present

Managing the TLB

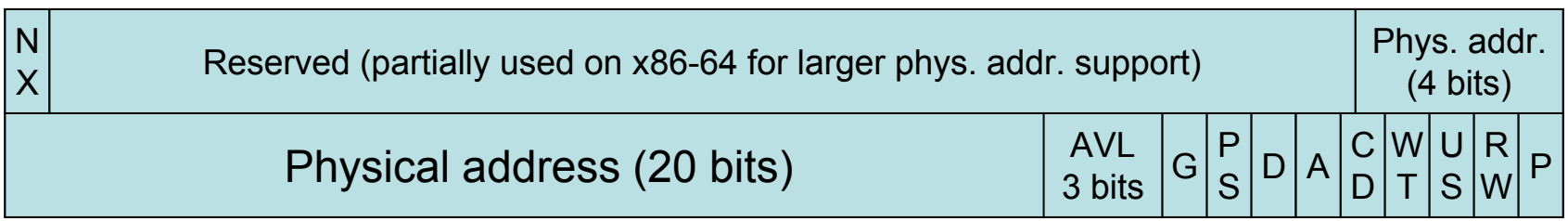
- For “present” pages, translation can be cached in on-chip TLB
- When PTE is modified, TLB must be flushed
  - TLBs are not “tagged” with address space ID
- Three methods to flush TLB
  - INVLPG invalidates a single page
  - Reloading CR3 invalidates all non-global pages
  - Modifying CR4.PGE invalidates all pages
- Software TLB “shoot down”
  - Requires IPIs to tell other processors to flush their TLBs

Large Pages

- If PS (page size) bit is set within PDE
  - PDE doesn't point to a PT
  - PDE refers directly to a "large page" (4MB normal, 2MB PAE)
- Reduce pressure on TLB
- Used in Windows for kernel code

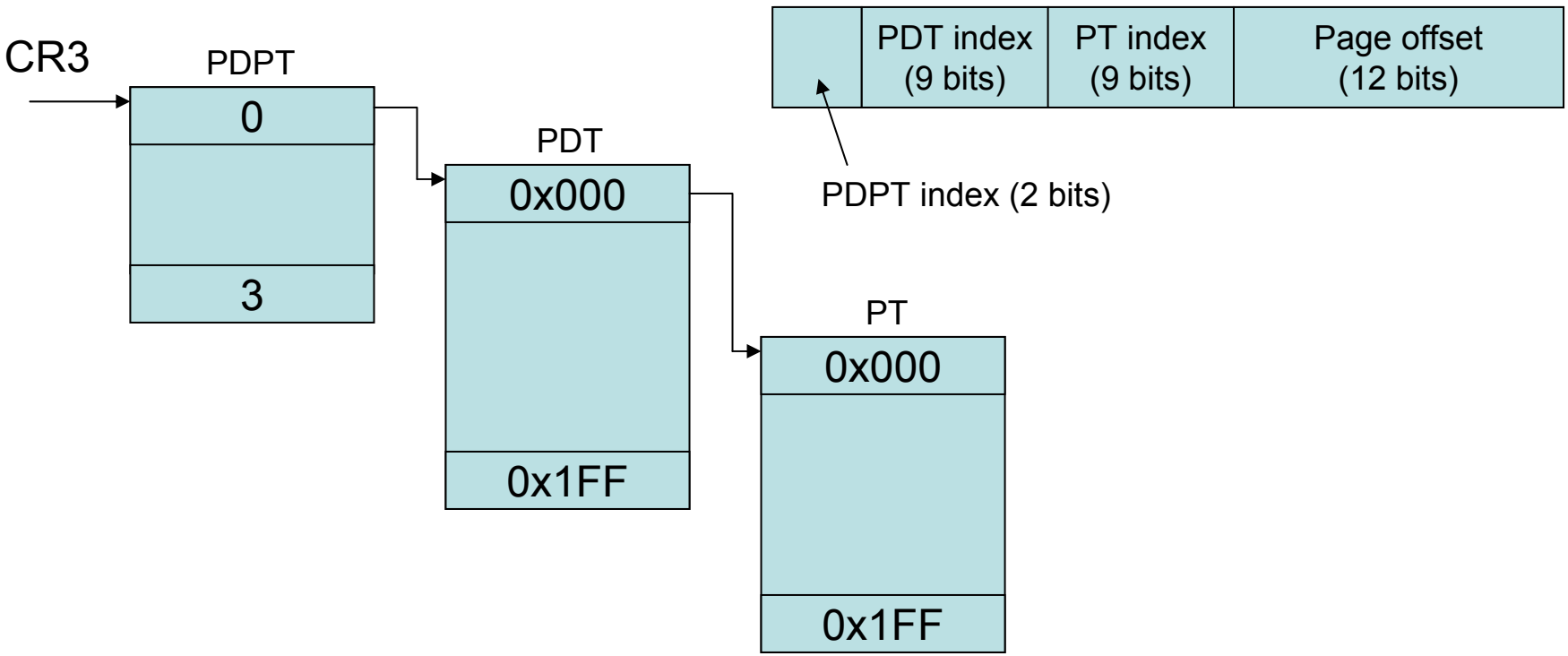
Physical Address Extension (PAE)

- Allows access to more than 4GB of physical memory
- Requires larger PTEs (to accommodate more than 20 bits of physical address)
- PTEs expanded to 8 bytes
- PDTs and PTs only hold half as many entries (512), so twice as much memory is needed for page tables
- An additional third level is needed
  - PDPT (page descriptor pointer table)
- Support for NX (no execute) bit on some processors



PAE (cont.)

- 32-bit virtual address
  - First 2 bits used to index into PDPT
  - Next 9 bits used to index into PDT
  - Next 9 bits used to index into PT



x86-64 Paging

- Uses four-level page table structure
- 52-bit physical address space
- 48-bit virtual address space

Sign-ext. bit 47 (16 bits)	PML4 index (9 bits)	PDPT index (9 bits)	PDT index (9 bits)	PT index (9 bits)	Page offset (12 bits)

Windows Kernel Internals II
Traps, Interrupts, Exceptions

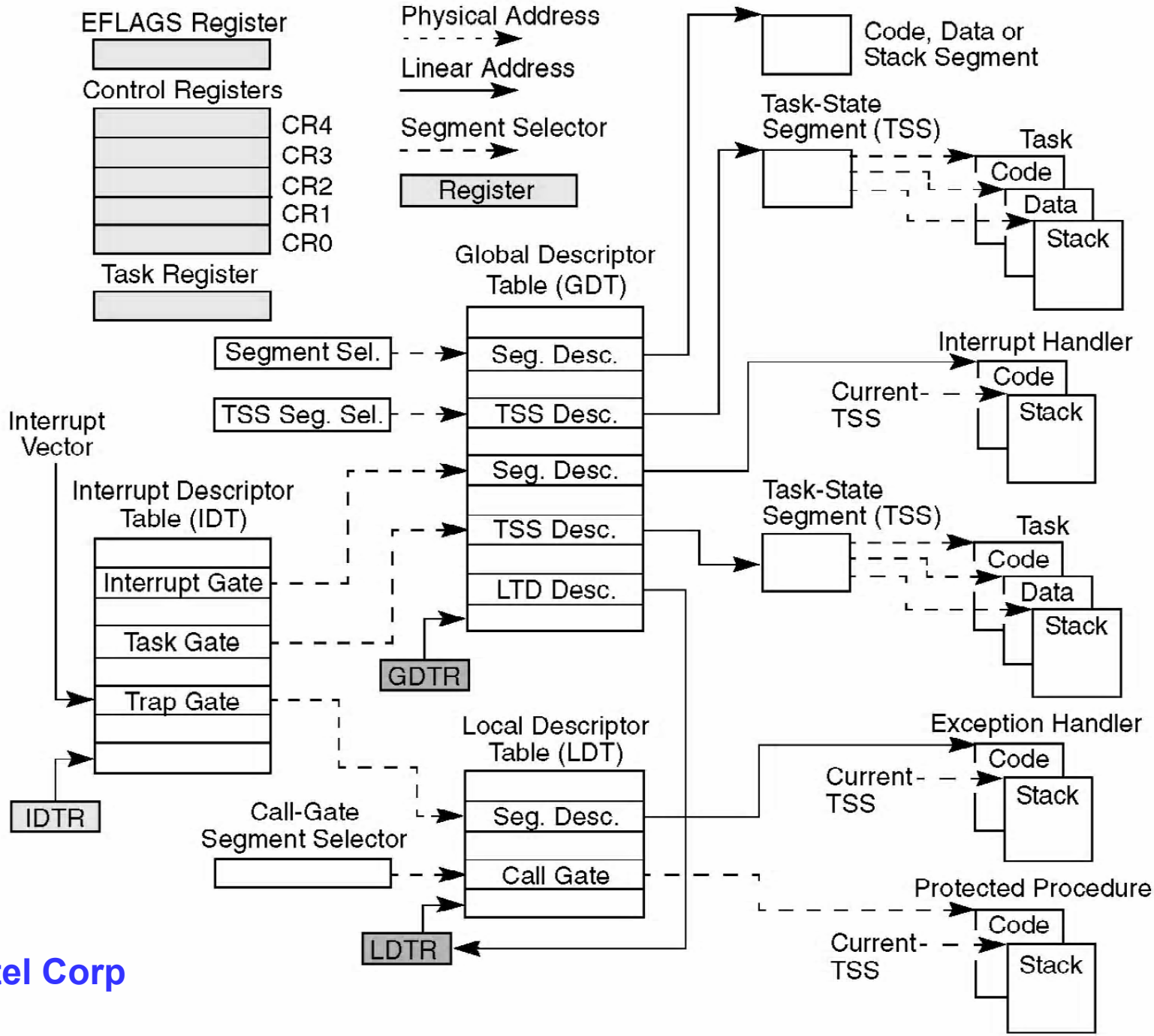
Dave Probert, Ph.D.
Advanced Operating Systems Group
Windows Core Operating Systems Division
Microsoft Corporation

What makes an OS interesting!

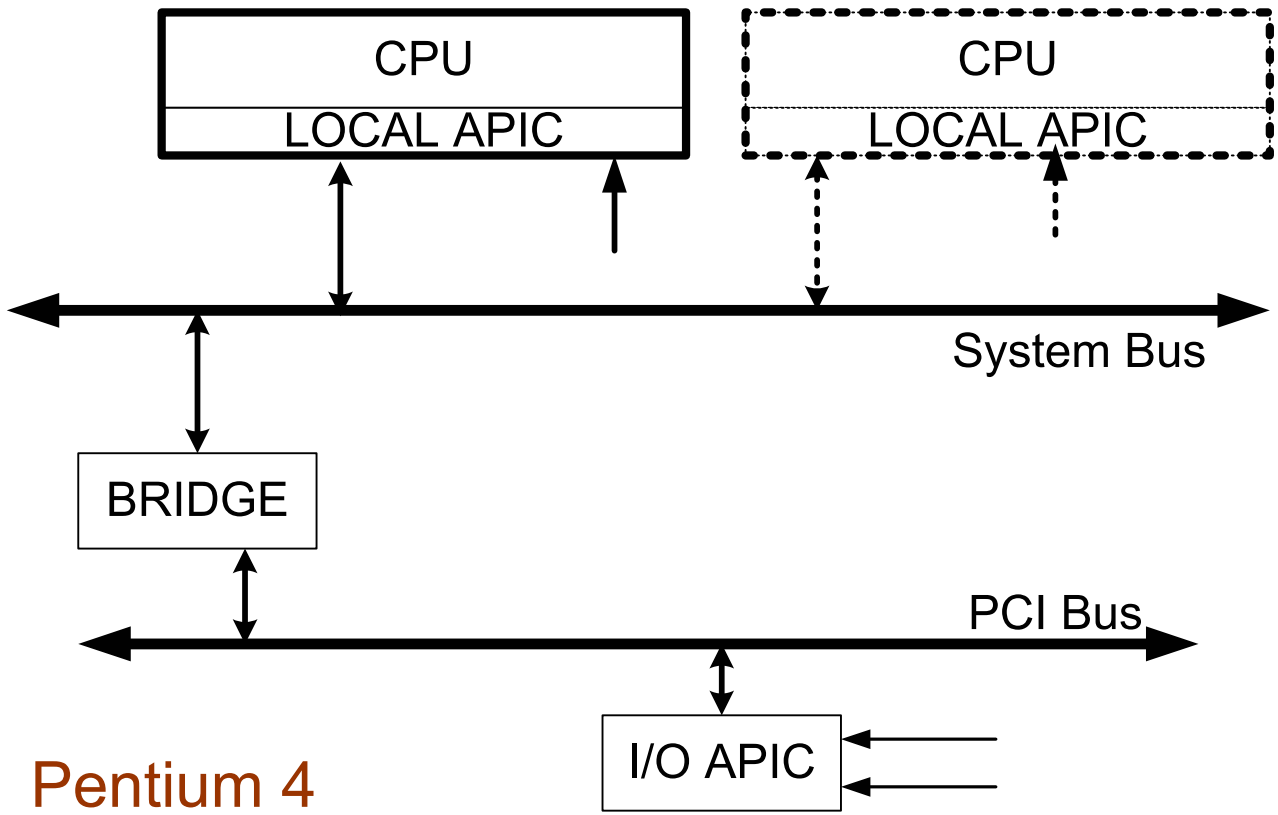
Fundamental abstractions in modern CPUs:

- normal processor execution
- virtual address protection/mapping
- interruptions to the above

Traps, hardware interrupts (devices, timers), exceptions, faults, machine checks, software interrupts


EFLAGS Register
Control Registers
CR4
CR3
CR2
CR1
CR0
Task Register
Physical Address
Linear Address
Segment Selector
Register
Code, Data or Stack Segment
Task-State Segment (TSS)
Task
Code
Data
Stack
Global Descriptor Table (GDT)
Segment Sel.
Seg. Desc.
TSS Seg. Sel.
TSS Desc.
Seg. Desc.
TSS Desc.
LTD Desc.
Interrupt Handler
Code
Current-TSS
Stack
Interrupt Vector
Interrupt Descriptor Table (IDT)
Interrupt Gate
Task Gate
Trap Gate
GDTR
Task-State Segment (TSS)
Task
Code
Data
Stack
Local Descriptor Table (LDT)
Exception Handler
Code
Current-TSS
Stack
IDTR
Call-Gate Segment Selector
Seg. Desc.
Call Gate
LDTR
Protected Procedure
Code
Current-TSS
Stack

Intel's System Architecture



Pentium 4

The local APIC

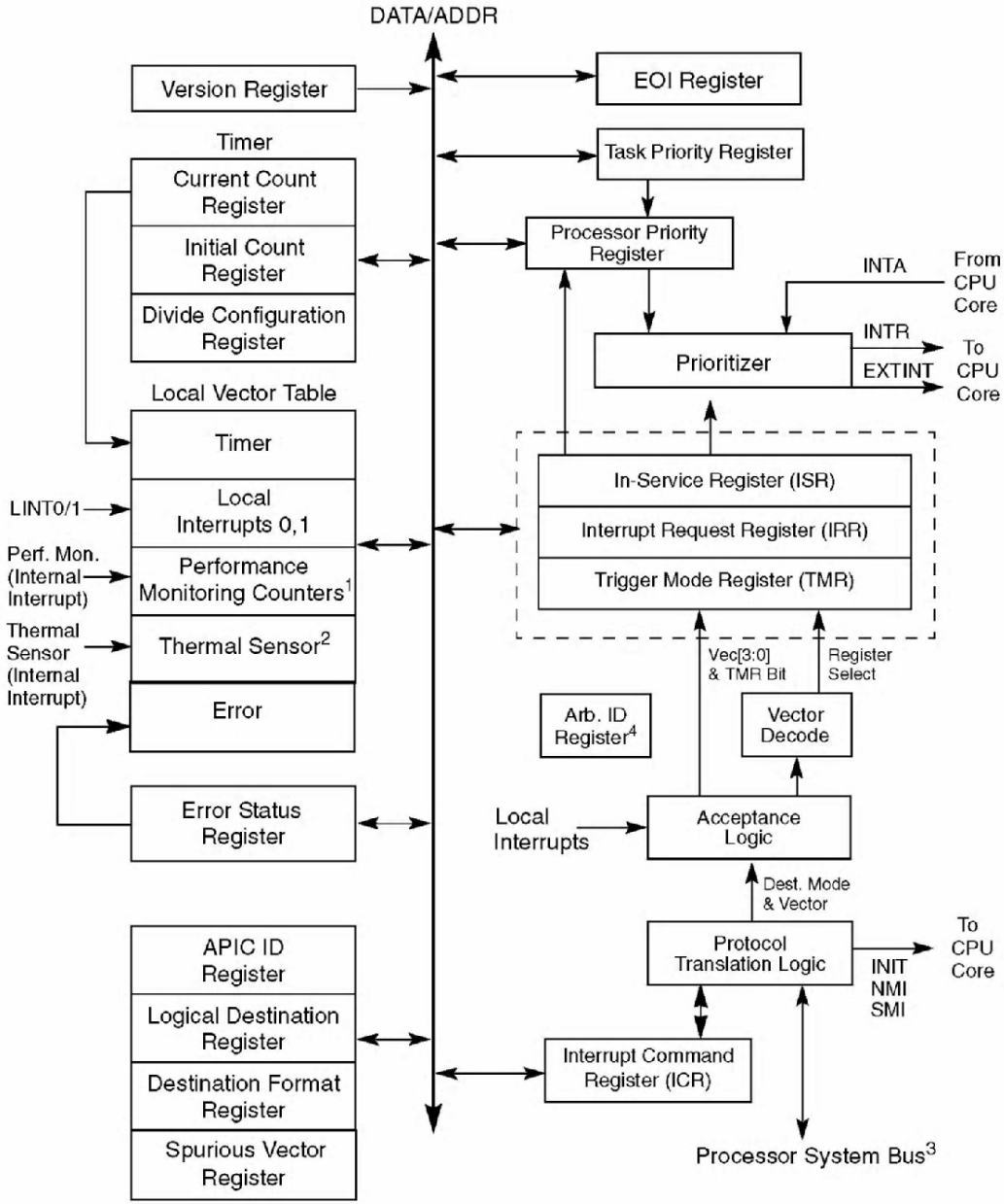
APIC: Advanced Programmable Interrupt Controller)

Local APIC built into modern Pentium processors

Receives interrupts from:

- processor interrupt pins
- external interrupt sources
  - hardwired devices
  - timers (including internal timer)
  - Perf monitors
  - Thermal monitors
  - Internal errors
- and/OR an external I/O APIC

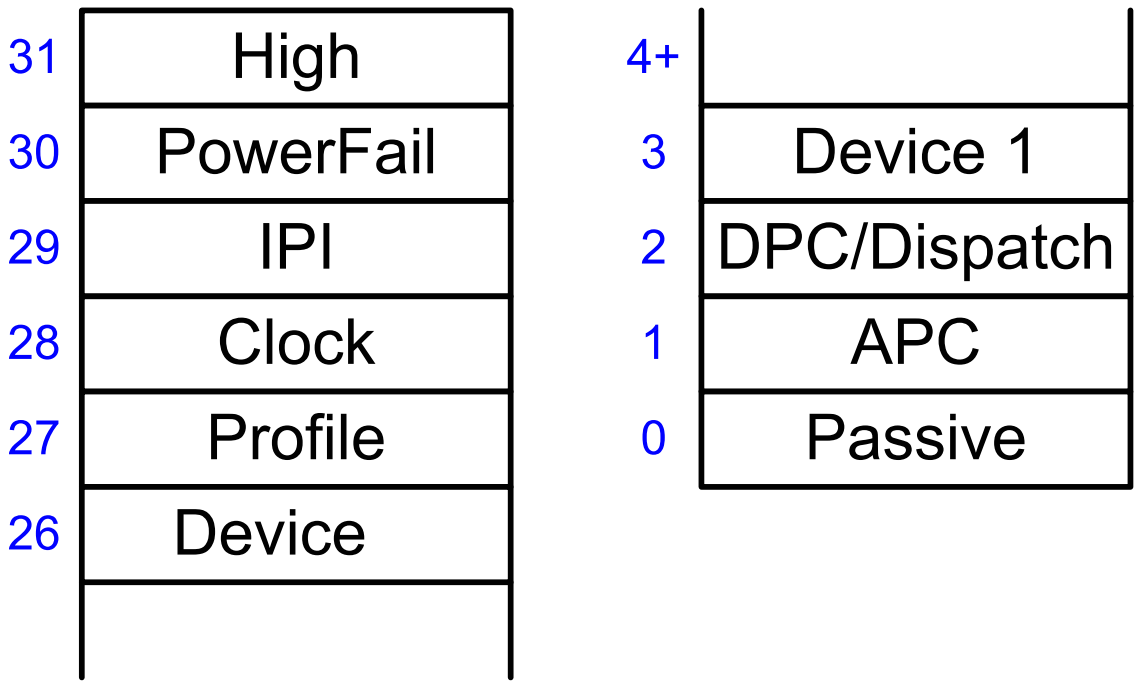
Sends IPIs in MP systems



1. Introduced in P6 family processors.
2. Introduced in the Pentium 4 and Intel Xeon processors.
3. Three-wire APIC bus in P6 family and Pentium processors.
4. Not implemented in Pentium 4 and Intel Xeon processors.

Figure 8-4. Local APIC Structure

NT Interrupt levels



Software Interrupt Delivery

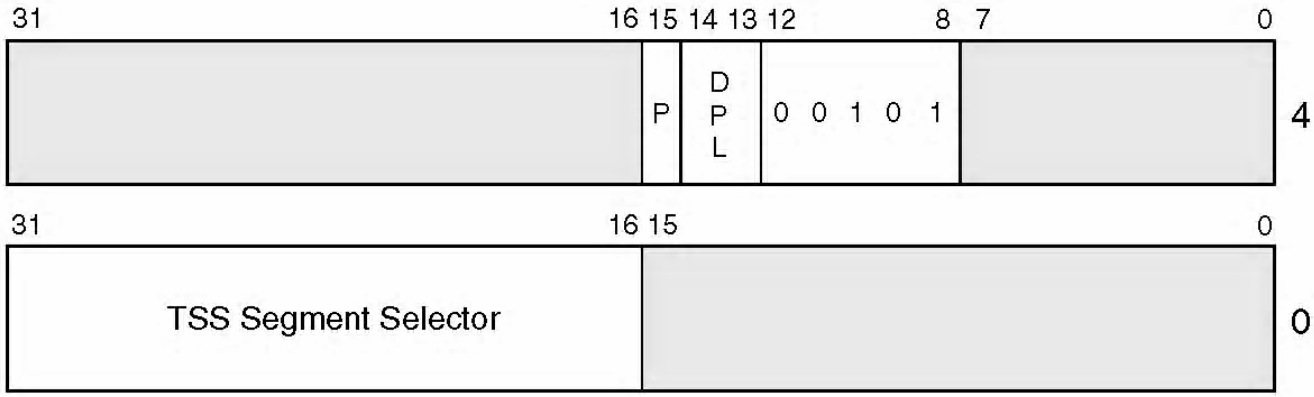
Software interrupts delivered by writing ICR in APIC

```
xor     ecx, ecx
mov     cl, _HalpIRQLtoTPR[eax]     ; get IDTEntry for IRQL
or      ecx, (DELIVER_FIXED OR ICR_SELF)
mov     dword ptr APIC[LU_INT_CMD_LOW], ecx

_HalpIRQLtoTPR  label   byte
            db      ZERO_VECTOR         ; IRQL 0
            db      APC_VECTOR          ; IRQL 1
            db      DPC_VECTOR          ; IRQL 2


#define APC_VECTOR              0x3D    // IRQL 01 APC
#define DPC_VECTOR              0x41    // IRQL 02 DPC
```

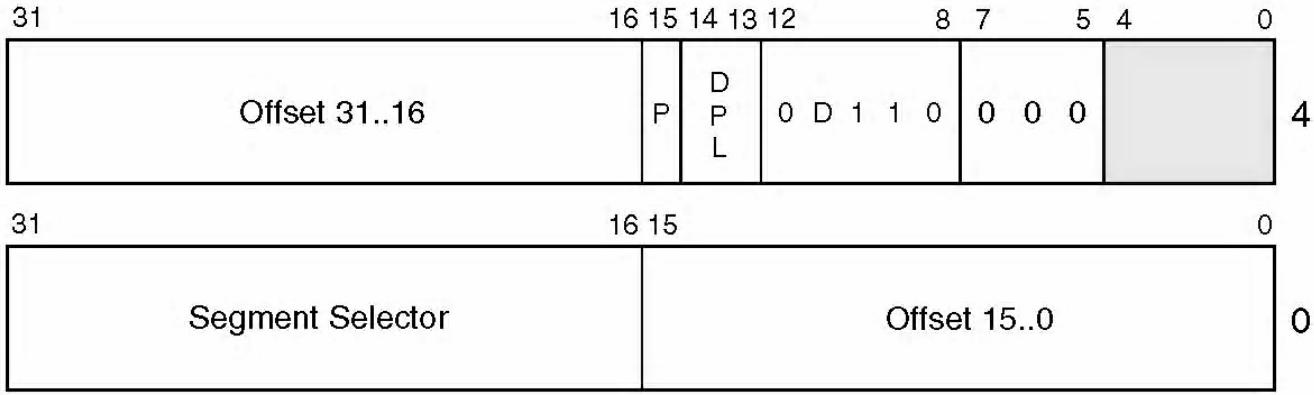
IDT table

```
_IDT            label byte

IDTEntry _KiTrap00  ; 0: Divide Error
IDTEntry _KiTrap01  ; 1: DEBUG TRAP
IDTEntry _KiTrap02  ; 2: NMI/NPX Error
IDTEntry _KiTrap03  ; 3: Breakpoint
IDTEntry _KiTrap04  ; 4: INTO
IDTEntry _KiTrap05  ; 5: PrintScreen
IDTEntry _KiTrap06  ; 6: Invalid Opcode
IDTEntry _KiTrap07  ; 7: no NPX
IDTEntry _KiTrap08  ; 8: DoubleFault
IDTEntry _KiTrap09  ; 9: NPX SegOvrn
...
IDTEntry _KiTrap0A  ; A: Invalid TSS
IDTEntry _KiTrap0B  ; B: no Segment
IDTEntry _KiTrap0C  ; C: Stack Fault
IDTEntry _KiTrap0D  ; D: GenProt
IDTEntry _KiTrap0E  ; E: Page Fault
IDTEntry _KiTrap0F  ; F: Reserved
IDTEntry _KiTrap10  ;10: 486 coproc
IDTEntry _KiTrap11  ;11: 486 align
IDTEntry _KiTrap0F  ;12: Reserved
IDTEntry _KiTrap0F  ;13: XMMI
IDTEntry _KiTrap0F  ;14: Reserved
```

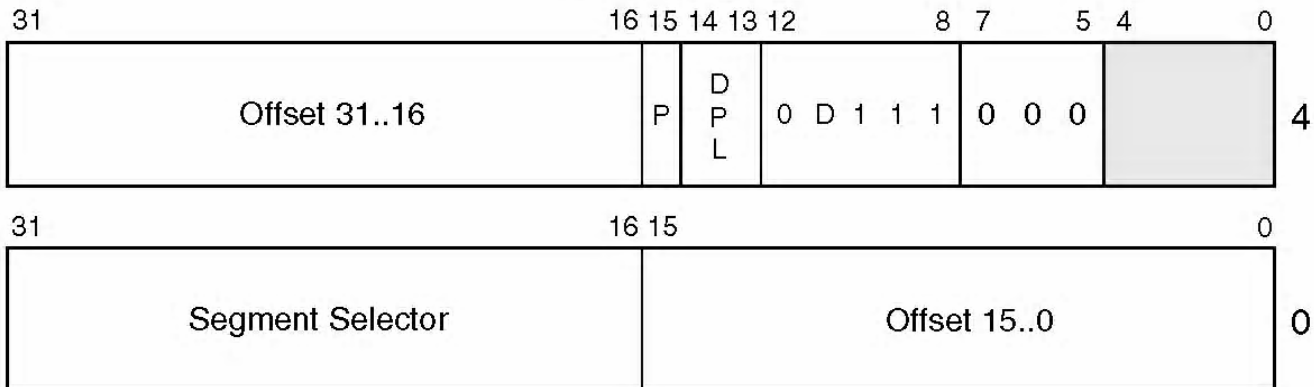
Task Gate



Interrupt Gate



Trap Gate



Intel Corp

Entry of Interrupt Descriptor Table (KIDTENTRY)

```
typedef struct _KIDTENTRY {
   USHORT Offset;
   USHORT Selector;
   USHORT Access;
   USHORT ExtendedOffset;
} KIDTENTRY;
```

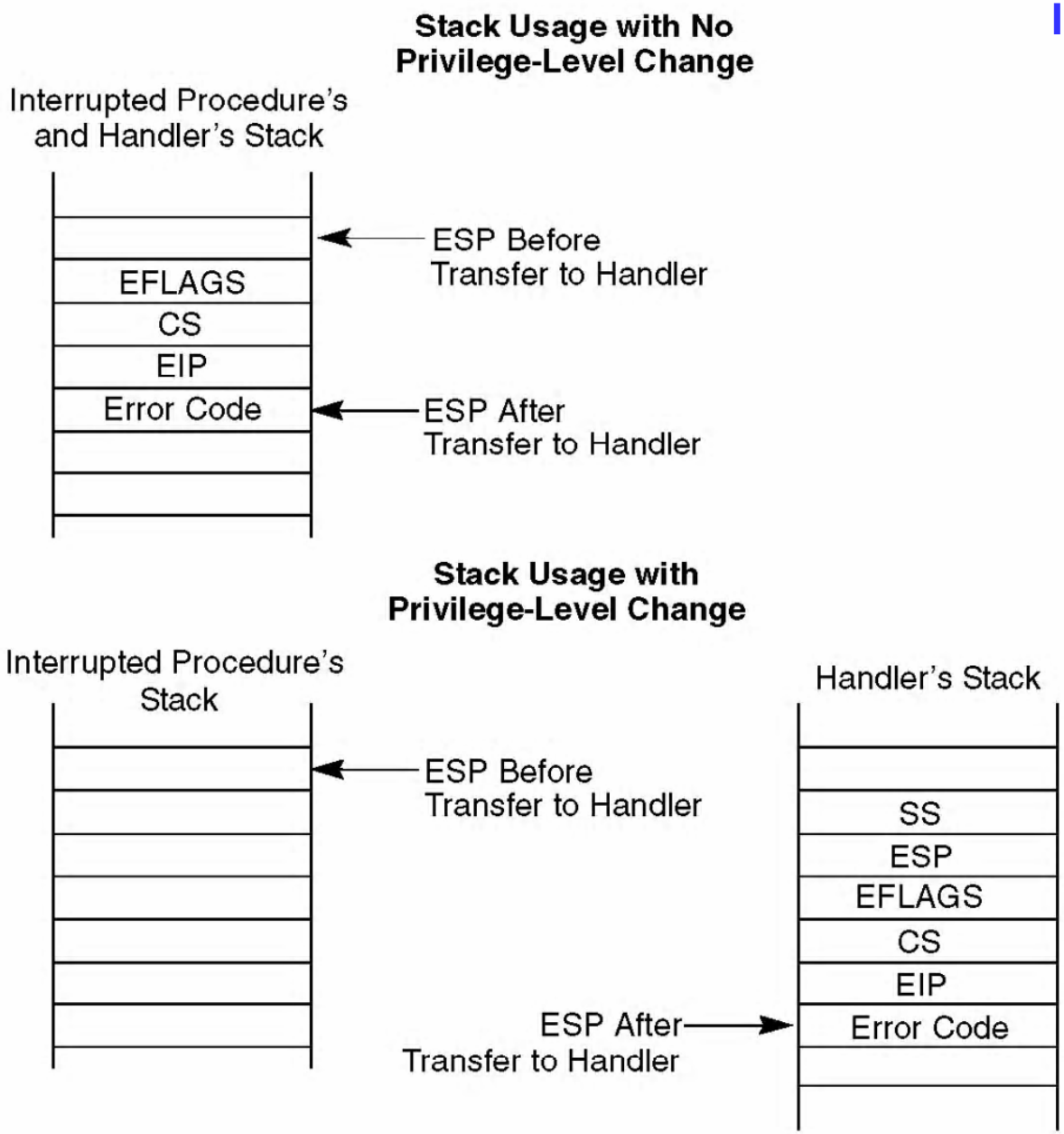


Figure 5-4. Stack Usage on Transfers to Interrupt and Exception-Handling Routines

_KiTrapxx - trap entry points

Entry points are for internally generated exceptions not external interrupts, or user software interrupts

On entry the stack looks like:

[ss]
[esp]
eflags
cs
eip
ss:sp-> [error]

CPU saves previous SS:ESP, eflags, and CS:EIP on the new stack if there was a privilige transition

Some exceptions save an error code, others do not

ENTER_TRAP

Macro Description: Build frame and set registers needed by trap or exception.

Save:

Non-volatile regs,
FS,
ExceptionList,
PreviousMode,
Volatile regs
Seg Regs from V86 mode
DS, ES, GS

Don't Save:

Floating point state

Set:

Direction,
DS, ES

Don't Set:

PreviousMode,
ExceptionList

Intel exception lexicon

Faults - correctable, faulting instuction re-executed

Traps - correctable, trapping instruction generally skipped

Aborts - unrecoverable, cause

CommonDispatchException()

CommonDispatchException (

ExceptCode - Exception code to put into exception record

ExceptAddress - Instruction at which HW exception

NumParms, Parameters 1, 2, 3

)

Allocates exception record on stack

Sets up exception record using specified parameters

Sets up arguments and calls _KiDispatchException()

KiDispatchException()

KiDispatchException (
    IN PEXCEPTION_RECORD ExceptionRecord,
    IN PKEXCEPTION_FRAME ExceptionFrame,
    IN PKTRAP_FRAME TrapFrame,
    IN KPROCESSOR_MODE PreviousMode,
    IN BOOLEAN FirstChance
)

Move machine state from trap and exception frames to a context frame

Select method of handling the exception based on previous mode

Kernel-mode: try KD, try RtlDispatchException(), otherwise bugcheck

User-mode: try DebugPort, else copy exception to user stack, set
    TrapFrame->Eip = (ULONG)KeUserExceptionDispatcher

and return

PspLookupKernelUserEntryPoints()

```
// Lookup the user mode "trampoline" code for exception dispatching
PspLookupSystemDllEntryPoint
        ("KiUserExceptionDispatcher“, &KeUserExceptionDispatcher)
// Lookup the user mode "trampoline" code for APC dispatching
PspLookupSystemDllEntryPoint
                ("KiUserApcDispatcher", &KeUserApcDispatcher)
// Lookup the user mode "trampoline" code for callback dispatching
PspLookupSystemDllEntryPoint
            ("KiUserCallbackDispatcher", &KeUserCallbackDispatcher)
// Lookup the user mode "trampoline" code for callback dispatching
PspLookupSystemDllEntryPoint ("KiRaiseUserExceptionDispatcher",
                                &KeRaiseUserExceptionDispatcher)
```

KeUserExceptionDispatcher

ntdll:KiUserExceptionDispatcher()

// Entered on return from kernel mode to dispatch user mode exception
// If a frame based handler handles the exception
// then the execution is continued
// else last chance processing is performed

basically this just wraps **RtlDispatchException()**

RtlDispatchException()

RtlDispatchException(ExceptionRecord, ContextRecord)

```
// attempts to dispatch an exception to a call frame based handler
// searches backwards through the stack based call frames
// search begins with the frame specified in the context record
// search ends when handler found, stack is invalid, or end of call chain

for (RegistrationPointer = RtlpGetRegistrationHead();
     RegistrationPointer != EXCEPTION_CHAIN_END;
     RegistrationPointer = RegistrationPointer->Next)
{
        check for valid record (#if ntos:  check DPC stack too)
        switch RtlpExecuteHandlerForException()
        case ExceptionContinueExecution: return TRUE
        case ExceptionContinueSearch: continue
        case ExceptionNestedException: …
        default:  return FALSE
}
```

Discussion