Interrupts

Chapter 20
S. Dandamudi
Outline

- What are interrupts?
- Types of interrupts
  - Software interrupts
  - Hardware interrupts
  - Exceptions
- Interrupt processing
  - Protected mode
  - Real mode
- Software interrupts
  - Keyboard services
    » int 21H DOS services
    » int 16H BIOS services
- Exceptions
  - Single-step example
- Hardware interrupts
  - Accessing I/O
  - Peripheral support chips
- Writing user ISRs
- Interrupt processing in PowerPC
- Interrupt processing in MIPS
What Are Interrupts?

• Interrupts alter a program’s flow of control
  ∗ Behavior is similar to a procedure call
    » Some significant differences between the two

• Interrupt causes transfer of control to an interrupt service routine (ISR)
  » ISR is also called a handler

• When the ISR is completed, the original program resumes execution

• Interrupts provide an efficient way to handle unanticipated events
## Interrupts versus Procedures

**Interrupts**
- Initiated by both *software* and *hardware*
- Can handle *anticipated* and *unanticipated* internal as well as external events
- ISRs or interrupt handlers are memory resident
- Use numbers to identify an interrupt service
- (E)FLAGS register is saved automatically

**Procedures**
- Can only be initiated by *software*
- Can handle *anticipated* events that are coded into the program
- Typically loaded along with the program
- Use meaningful names to indicate their function
- Do not save the (E)FLAGS register
A Taxonomy of Pentium Interrupts

- Pentium interrupts
  - Exceptions
    - Aborts
    - Faults
  - Software interrupts
    - Traps
  - Hardware interrupts
    - Maskable
    - Nonmaskable
Various Ways of Interacting with I/O Devices

Application program

DOS support

BIOS support

Input/Output devices
Protected Mode Interrupt Processing

• Up to 256 interrupts are supported (0 to 255)
  » Same number in both real and protected modes
  » Some significant differences between real and protected mode interrupt processing

• Interrupt number is used as an index into the Interrupt Descriptor Table (IDT)
  * This table stores the addresses of all ISRs
  * Each descriptor entry is 8 bytes long
    » Interrupt number is multiplied by 8 to get byte offset into IDT
  * IDT can be stored anywhere in memory
    » In contrast, real mode interrupt table has to start at address 0
Protected Mode Interrupt Processing (cont’d)

* Location of IDT is maintained by IDT register IDTR
  * IDTR is a 48-bit register
    » 32 bits for IDT base address
    » 16 bits for IDT limit value
      – IDT requires only 2048 (11 bits)
      – A system may have smaller number of descriptors
        ➔ Set the IDT limit to indicate the size in bytes
    » If a descriptor outside the limit is referenced
      – Processor enters shutdown mode

* Two special instructions to load *(lidt)* and store *(sidt)* IDT
  » Both take the address of a 6-byte memory as the operand
Interrupt Processing in Real Mode

• Uses an interrupt vector table that stores pointers to the associated interrupt handlers.
  * This table is located at base address zero.

• Each entry in this table consists of a CS:IP pointer to the associated ISRs
  * Each entry or vector requires four bytes:
    » Two bytes for specifying CS
    » Two bytes for the offset

• Up to 256 interrupts are supported (0 to 255).
### Interrupt Vector Table

<table>
<thead>
<tr>
<th>Memory address (in Hex)</th>
<th>CS high byte</th>
<th>CS low byte</th>
<th>IP high byte</th>
<th>IP low byte</th>
<th>CS</th>
<th>int type 255</th>
</tr>
</thead>
<tbody>
<tr>
<td>003FF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>003FE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>003FD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>003FC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>int type 2</td>
</tr>
<tr>
<td>0000A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00009</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>int type 1</td>
</tr>
<tr>
<td>00008</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00007</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>int type 0</td>
</tr>
<tr>
<td>00006</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00005</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00004</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00003</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00002</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Interrupt Number to Vector Translation

- Interrupt numbers range from 0 to 255
- Interrupt number acts as an index into the interrupt vector table
- Since each vector takes 4 bytes, interrupt number is multiplied by 4 to get the corresponding ISR pointer

**Example**
- For interrupt 2, the memory address is $2 \times 4 = 8H$
- The first two bytes at 8H are taken as the offset value
- The next two bytes (i.e., at address AH) are used as the CS value
A Typical ISR Structure

• Just like procedures, ISRs should end with a return statement to return control back

• The interrupt return (iret) is used for this purpose
  ;save the registers used in the ISR
  sti    ;enable further interrupts
  . . .
  ISR body
  . . .
  ;restore the saved registers
  iret   ;return to interrupted program
What Happens When An Interrupt Occurs?

• Push flags register onto the stack
• Clear interrupt enable and trap flags
  * This disables further interrupts
  * Use sti to enable interrupts
• Push CS and IP registers onto the stack
• Load CS with the 16-bit data at memory address
  interrupt-type * 4 + 2
• Load IP with the 16-bit data at memory address
  interrupt-type * 4
Interrupt Enable Flag Instructions

• Interrupt enable flag controls whether the processor should be interrupted or not

• Clearing this flag disables all further interrupts until it is set
  * Use `cli` (clear interrupt) instruction for this purpose
  * It is cleared as part interrupt processing

• Unless there is special reason to block further interrupts, enable interrupts in your ISR
  * Use `sti` (set interrupt) instruction for this purpose
Returning From An ISR

• As in procedures, the last instruction in an ISR should be \textit{iret}

• The actions taken on \textit{iret} are:
  * pop the 16-bit value on top of the stack into IP register
  * pop the 16-bit value on top of the stack into CS register
  * pop the 16-bit value on top of the stack into the flags register

• As in procedures, make sure that your ISR does not leave any data on the stack
  * Match your push and pop operations within the ISR
Software Interrupts

- Initiated by executing an interrupt instruction
  \[\text{int } \text{interrupt-type}\]
  \text{interrupt-type} \text{ is an integer in the range 0 to 255}
- Each interrupt type can be parameterized to provide several services.
- For example, DOS interrupt service int 21H provides more than 80 different services
  * AH register is used to identify the required service under int 21H.
Example DOS Service: Keyboard

- DOS provides several interrupt services to interact with the keyboard
- AH register should be loaded with the desired function under `int 21H`
- Seven functions are provided by DOS to read a character or get the status of the keyboard
  » See Section 12.5.2 for details
- We look at one function to read a string of characters from the keyboard.
A DOS Keyboard Function

- Function 0AH --- Buffered Keyboard Input
  
  **Inputs:** $AH = 0AH$  
  $DS:DX = \text{pointer to the input buffer}$  
  (first byte should be buffer size)

  **Returns:** character string in the input buffer

- Input string is terminated by CR
- Input string starts at the third byte of the buffer
- Second byte gives the actual number of characters read (excluding the CR)
### Input Buffer Details

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>( l-1 )</th>
<th>( l )</th>
<th>( l+1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( l )</td>
<td>( m )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Input buffer for character string**

\[ l = \text{maximum number of characters (given as input to the function)} \]

\[ m = \text{actual number of characters in the buffer excluding CR (returned by the function)} \]
A Keyboard Example

- **GetStr** procedure to read a string from the keyboard (see io.mac)
- Expects buffer pointer in AX and buffer length in CX
- Uses DOScall macro:

```assembly
DOScall MACRO fun_num
    mov AH, fun_num
    int 21H
ENDM
```

```plaintext
Proc_GetStr ()
Save registers used in proc.
if (CX < 2) then CX := 2
if (CX > 81) then CX := 81
Use function 0AH to read input string into temp. buffer str_buffer
Copy input string from str_buffer to user buffer and append NULL
Restore registers
```
BIOS Keyboard Services

• BIOS provides keyboard services under int 16H
• We focus on three functions provided by int 16H
  * Function 00H --- To read a character
  * Function 01H --- To check keyboard buffer
  * Function 02H --- To check keyboard status
• As with DOS functions, AH is used to identify the required service
• DOS services are flexible in that the keyboard input can be redirected (BIOS does not allow it)
BIOS Character Read Function

• Function 00H --- Read a char. from the keyboard

  **Inputs:** AH = 00H

  **Returns:** if AL is not zero

  AL = ASCII code of the key
  AH = Scan code of the key

  if AL is zero

  AH = Scan code of the extended key

• If keyboard buffer is empty, this function waits for a key to be entered
BIOS Keyboard Buffer Check Function

• Function 01H --- Check keyboard buffer
  
  **Inputs:** AH = 01H
  
  **Returns:** ZF = 1 if keyboard buffer is empty
  ZF = 0 if not empty
  
  ASCII and Scan codes are placed in AL and AH as in Function 00H

• The character is **not** removed from the keyboard buffer
## BIOS Keyboard Status Check Function

- **Function 02H --- Check keyboard status**
  - **Inputs:** \( AH = 02H \)
  - **Returns:**
    - \( AL = \) status of shift and toggle keys
- **Bit assignment is shown on the right**

<table>
<thead>
<tr>
<th>Bit#</th>
<th>Key assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Right SHIFT down</td>
</tr>
<tr>
<td>1</td>
<td>Left SHIFT down</td>
</tr>
<tr>
<td>2</td>
<td>CONTROL down</td>
</tr>
<tr>
<td>3</td>
<td>ALT down</td>
</tr>
<tr>
<td>4</td>
<td>SCROLL LOCK down</td>
</tr>
<tr>
<td>5</td>
<td>NUMBER LOCK down</td>
</tr>
<tr>
<td>6</td>
<td>CAPS LOCK down</td>
</tr>
<tr>
<td>7</td>
<td>INS LOCK down</td>
</tr>
</tbody>
</table>
A BIOS Keyboard Example

- BIOS, being a lower-level service, provides more flexibility
- FUNNYSTR.ASM reads a character string from the keyboard and displays it along with its length
  * The input string can be terminated either by pressing both SHIFT keys simultaneously, or by entering 80 characters, whichever occurs first.
- We use BIOS function 02H to detect the first termination condition.
Display and Printer Support

• Both DOS and BIOS provide support for Printer and Display screen

• An example DOS int 21H character display function

  Function 02H --- Display a char. to screen

  **Inputs:**  AH = 02H
              DL = ASCII code of the character
to be displayed

  **Returns:** nothing

• See `proc_nwln` procedure for usage
Exceptions

• Three types of exceptions
  ∗ Depending on the way they are reported
  ∗ Whether or not the interrupted instruction is restarted
    » Faults
    » Traps
    » Aborts

• Faults and traps are reported at instruction boundaries

• Aborts report severe errors
  ∗ Hardware errors
  ∗ Inconsistent values in system tables
Faults and Traps

• Faults
  » Instruction boundary before the instruction during which the exception was detected
  » Restarts the instruction
  » Divide error (detected during `div/idiv` instruction)
  » Segment-not-found fault

• Traps
  » Instruction boundary immediately after the instruction during which the exception was detected
  » No instruction restart
  » Overflow exception (interrupt 4) is a trap
  » User defined interrupts are also examples of traps
Dedicated Interrupts

• Several Pentium predefined interrupts --- called dedicated interrupts

• These include the first five interrupts:

<table>
<thead>
<tr>
<th>interrupt type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Divide error</td>
</tr>
<tr>
<td>1</td>
<td>Single-step</td>
</tr>
<tr>
<td>2</td>
<td>Nonmaskable interrupt (MNI)</td>
</tr>
<tr>
<td>3</td>
<td>Breakpoint</td>
</tr>
<tr>
<td>4</td>
<td>Overflow</td>
</tr>
</tbody>
</table>
Dedicated Interrupts (cont’d)

• Divide Error Interrupt
  * CPU generates a type 0 interrupt whenever the div/idiiv instructions result in a quotient that is larger than the destination specified

• Single-Step Interrupt
  * Useful in debugging
  * To single step, Trap Flag (TF) should be set
  * CPU automatically generates a type 1 interrupt after executing each instruction if TF is set
  * Type 1 ISR can be used to present the system state to the user
Dedicated Interrupts (cont’d)

• Breakpoint Interrupt
  * Useful in debugging
  * CPU generates a type 3 interrupt
  * Generated by executing a special single-byte version of int 3 instruction (opcode CCH)

• Overflow Interrupt
  * Two ways of generating this type 4 interrupt
    » int 4 (unconditionally generates a type 4 interrupt)
    » into (interrupt is generated only if the overflow flag is set)
  * We do not normally use into as we can use jo/jno conditional jumps to take care of overflow
A Single-Step Interrupt Example

- Objectives:
  * To demonstrate how ISRs can be defined and installed (i.e., user defined ISRs)
  * How trap flag can be manipulated
    » There are no instruction to set/clear the trap flag unlike the interrupt enable flag `sti/cli`

- We write our own type 1 ISR that displays the contents of AX and BX registers after each instruction has been executed
Two Services of int 21H

- **Function 35H --- Get interrupt vector**
  
  **Inputs:** $AH = 35H$
  $AL = \text{interrupt type number}$
  
  **Returns:** $ES:BX = \text{address of the specified ISR}$

- **Function 25H --- Set interrupt vector**
  
  **Inputs:** $AH = 25H$
  $AL = \text{interrupt type number}$
  $DS:DX = \text{address of the ISR}$
  
  **Returns:** nothing
Hardware Interrupts

- Software interrupts are synchronous events
  - Caused by executing the `int` instruction
- Hardware interrupts are of hardware origin and asynchronous in nature
  - Typically caused by applying an electrical signal to the processor chip
- Hardware interrupts can be
  - Maskable
  - Non-maskable
    - Causes a `type 2` interrupt
How Are Hardware Interrupts Triggered?

- Non-maskable interrupt is triggered by applying an electrical signal to the MNI pin of Pentium
  * Processor always responds to this signal
  * Cannot be disabled under program control
- Maskable interrupt is triggered by applying an electrical signal to the INTR (INTerrupt Request) pin of Pentium
  * Pentium recognizes this interrupt only if IF (interrupt enable flag) is set
  * Interrupts can be masked or disabled by clearing IF
How Does the CPU Know the Interrupt Type?

- Interrupt invocation process is common to all interrupts
  * Whether originated in software or hardware
- For hardware interrupts, CPU initiates an interrupt acknowledge sequence
  * CPU sends out interrupt acknowledge (INTA) signal
  * In response, interrupting device places interrupt type number on the data bus
  * CPU uses this number to invoke the ISR that should service the device (as in software interrupts)
How can More Than One Device Interrupt?

- Processor has only one INTR pin to receive interrupt signal
- Typical system has more than one device that can interrupt --- keyboard, hard disk, floppy, etc.
- Use a special chip to prioritize the interrupts and forward only one interrupt to the CPU
- 8259 Programmable Interrupt Controller chip performs this function (more details later)
Direct Control of I/O Devices

- Two ways of mapping I/O ports:
  * Memory-mapped I/O (e.g., MIPS)
    » I/O port is treated as a memory address (I/O port is mapped to a location in memory address space (MAS))
    » Accessing an I/O port (read/write) is similar to accessing a memory location (all memory access instructions can be used)
  * Isolated I/O (e.g., Pentium)
    » I/O address space is separate from the memory address space
      – leaves the complete MAS for memory
    » Separate I/O instructions and I/O signals are needed
    » Can’t use memory access instructions
    » Can also use memory-mapped I/O and use all memory access instructions
Pentium I/O Address Space

- Pentium provides 64 KB of I/O address space
  - Can be used for 8-, 16-, and 32-bit I/O ports
- Combination cannot exceed the total I/O space
  - 64K 8-bit I/O ports
    » Used for 8-bit devices, which transfer 8-bit data
    » Can be located anywhere in the I/O space
  - 32K 16-bit I/O ports (used for 16-bit devices)
    » 16-bit ports should be aligned to an even address
  - 16K 32-bit I/O ports (used for 32-bit devices)
    » Should be aligned to addresses that are multiples of four
    » Pentium supports unaligned ports, but with performance penalty
  - A combination of these for a total of 64 KB
Pentium I/O Instructions

• Pentium provides two types of I/O instructions:
  * Register I/O instructions
    » Used to transfer data between a register (accumulator) and an I/O port
    » in — to read from an I/O port
    » out — to write to an I/O port
  * Block I/O instructions
    » Used to transfer a block of data between memory and an I/O port
    » ins — to read from an I/O port
    » outs — to write to an I/O port
Register I/O Instructions

• Can take one of two forms depending on whether a port is directly addressable or not
  – A port is said to be directly addressable if it is within the first 256 ports (so that one byte can be used specify it)

• To read from an I/O port
  \[ \text{in } \text{accumulator, port8} \quad -- \quad \text{direct addressing format} \]
  \[ \text{port8 is 8-bit port number} \]

  \[ \text{in } \text{accumulator, DX} \quad -- \quad \text{indirect addressing format} \]
  \[ \text{port number should be loaded into DX} \]
  \[ \text{accumulator can be AL, AX, or EAX (depending on I/O port)} \]

• To write to an I/O port
  \[ \text{out port8, accumulator} \quad -- \quad \text{direct addressing format} \]
  \[ \text{out DX, accumulator} \quad -- \quad \text{indirect addressing format} \]
Block I/O Instructions

- Similar to string instructions
- **ins** and **outs** do not take any operands
- I/O port address should be in DX
  - *No direct addressing format is allowed*
- **ins** instruction to read from an I/O port
  » ES:(E)DI should point to memory buffer
- **outs** instruction to write to an I/O port
  » DS:(E)SI should point to memory buffer
- **rep** prefix can be used for block transfer of data as in the string instructions
8259 Programmable Interrupt Controller

- 8259 can service up to eight hardware devices
  * Interrupts are received on IRQ0 through IRQ7
- 8259 can be programmed to assign priorities in several ways
  * Fixed priority scheme is used in the PC
    » IRQ0 has the highest priority and IRQ7 lowest
- 8259 has two registers
  * Interrupt Command Register (ICR)
    » Used to program 8259
  * Interrupt Mask Register (IMR)
8259 PIC (cont’d)
## 8259 PIC (cont’d)

- Mapping in a single 8259 PIC systems

<table>
<thead>
<tr>
<th>IRQ#</th>
<th>Interrupt type</th>
<th>Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>08H</td>
<td>System timer</td>
</tr>
<tr>
<td>1</td>
<td>09H</td>
<td>Keyboard</td>
</tr>
<tr>
<td>2</td>
<td>0AH</td>
<td>reserved (2nd 8259)</td>
</tr>
<tr>
<td>3</td>
<td>0BH</td>
<td>Serial port (COM1)</td>
</tr>
<tr>
<td>4</td>
<td>0CH</td>
<td>Serial port (COM2)</td>
</tr>
<tr>
<td>5</td>
<td>0DH</td>
<td>Hard disk</td>
</tr>
<tr>
<td>6</td>
<td>0EH</td>
<td>Floppy disk</td>
</tr>
<tr>
<td>7</td>
<td>0FH</td>
<td>Printer (LPT1)</td>
</tr>
</tbody>
</table>
8259 PIC (cont’d)

• Interrupt Mask Register (IMR) is an 8-bit register
  * Used to enable or disable individual interrupts on lines IRQ0 through IRQ7
    » Bit 0 is associated with IRQ0, bit 1 to IRQ1, . . .
    » A bit value of 0 enables the corresponding interrupt (1 disables)

• Processor recognizes external interrupts only when the IF is set

• Port addresses:
  * ICR: 20H
  * IMR: 21H
**Example:** Disable all 8259 interrupts except the system timer

```
mov    AL, 0FEH
out    21H, AL
```

- 8259 needs to know when an ISR is done (so that it can forward other pending interrupt requests)
  - End-of-interrupt (EOI) is signaled to 8259 by writing 20H into ICR
```
mov    AL, 20H
out    20H, AL
```
  - This code fragment should be used before `iret`
8255 Programmable Peripheral Interface Chip

- Provides three 8-bit registers (PA, PB, PC) that can be used to interface with I/O devices
- These three ports are configured as follows:
  - PA -- Input port
  - PB -- Output port
  - PC -- Input port
- 8255 also has a command register
- 8255 port address mapping
  - PA --- 60H
  - PB --- 61H
  - PC --- 62H
  - Command register --- 63H
Keyboard Interface

• PA and PB7 are used for keyboard interface
  * PA0 – PA6 = key scan code
  * PA7 = 0 if a key is depressed
  * PA7 = 1 if a key is released

• Keyboard provides the scan code on PA and waits for an acknowledgement
  * Scan code read acknowledge signal is provided by momentarily setting and clearing PB7
    » Normal state of PB7 is 0

• Keyboard generates IRQ1
  » IRQ1 generates a type 9 interrupt
Interrupt Processing in PowerPC

• Pentium uses the stack to store the return address
• PowerPC and MIPS use registers
  * A characteristic of RISC processors
• Maintains system state information in machine state register (MSR)
  * POW: Power management enable
    » POW = 0 – Normal mode
    » POW = 1 – Reduced power mode
  * EE: Interrupt enable bit
    » Similar to IF in Pentium
Interrupt Processing in PowerPC (cont’d)

* PR: Privilege level
  » PR = 0 – Executes both user- and supervisor-level instructions
  » PR = 1 – Executes only user-level instructions

* SE: Single-step enable bit
  » SE = 0 – Normal execution
  » SE = 1 – Single-step

* IP: Exception prefix bit
  » Indicates whether an exception vector is prepended with Fs or 0s

* IR: Instruction translation bit

* DR: Data address translation bit

* LE: Endianness (1 = little-endian, 0 = big-endian)
Interrupt Processing in PowerPC (cont’d)

PowerPC exception classification

- PowerPC exceptions
  - Asynchronous
    - Maskable
    - Nonmaskable
  - Synchronous
    - Precise
    - Imprecise
Interrupt Processing in PowerPC (cont’d)

• Asynchronous
  * Maskable
  * Nonmaskable
    » As in Pentium

• Synchronous
  * Precise
    » Associated with an instruction that caused it
  * Imprecise
    » No such association
    » Used for floating-point instructions
Interrupt Processing in PowerPC (cont’d)

- Interrupt processing (similar to Pentium’s)
  * Saves machine state information
    » Uses save/restore register SRR1 register
  * Disables further interrupts
  * Saves the return address
    » Uses save/restore register SRR0 register
  * Loads the interrupt handler address
    » Each exception has a fixed offset value
    » Offsets are used directly (unlike Pentium)
    » Offsets are separated by 256 bytes
    » 01000 – 02FFH: reserved for implementation-specific exceptions
## Interrupt Processing in PowerPC (Cont’d)

<table>
<thead>
<tr>
<th>Exception type</th>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>System reset</td>
<td>00100</td>
<td>System reset exception (similar to Reset in the Pentium)</td>
</tr>
<tr>
<td>Machine check</td>
<td>00200</td>
<td>An external interrupt caused by bus parity error, access to invalid physical address, etc.</td>
</tr>
<tr>
<td>DSI</td>
<td>00300</td>
<td>Invalid data memory access exception (e.g., memory protection violation, address cannot be translated)</td>
</tr>
<tr>
<td>ISI</td>
<td>00400</td>
<td>Invalid instruction memory access exception (e.g., instruction fetch from a no-execute segment, instruction fetch violates memory protection)</td>
</tr>
<tr>
<td>External interrupt</td>
<td>00500</td>
<td>External interrupt as in Pentium</td>
</tr>
<tr>
<td>Alignment</td>
<td>00600</td>
<td>Alignment exception (e.g., operand is not aligned)</td>
</tr>
<tr>
<td>Program</td>
<td>00700</td>
<td>Program exception (e.g., illegal instruction, privileged instruction execution is attempted in user-level)</td>
</tr>
<tr>
<td>System call</td>
<td>00C00</td>
<td>System call exception occurs when a system call (SC) instruction is executed (similar to the int instruction of the Pentium)</td>
</tr>
<tr>
<td>Trace</td>
<td>00D00</td>
<td>Trace exception (for debugging purposes)</td>
</tr>
</tbody>
</table>
Interrupt Processing in PowerPC (Cont’d)

- If IP = 1, leading prefixes are Fs
- Example:
  - IP = 1: 00000500H
  - IP = 0: FFF00500H
- Return from exceptions
  - Similar to Pentium’s
  - Uses `rfi` instruction
  - Copies SRR1 into MSR
  - Transfers control to the instruction at the address in SRR0
Interrupt Processing in MIPS

• Does not use vectored interrupts

• On interrupts
  * Enters the kernel mode
  * Disables further interrupts
  * Transfers control to a handler located at a fixed address
    » Handler saves the context
      – Program counter
      – Current operating mode (user or supervisor)
      – Status of interrupts (enables or disabled)
  * Stores return address
    » Uses a register: EPC (exception program counter) register
Interrupt Processing in MIPS (cont’d)

• Registers for interrupt processing are not on the main processor
  * Located in coprocessor 0 (CP0)
    » Register 14 is used as EPC
    » EPC can be copied to main processor using \texttt{mfc0} (move from coprocessor 0)

• Interrupt type is provided by the \textit{Cause} register
  * Register 13 of CP0 is used as the \textit{Cause} register
    » Also contains information on pending interrupts
      – 8 bits are used for this purpose
        ➔ Two are used for software interrupts
### Interrupt Processing in MIPS (cont’d)

#### Table 20.6 Some example MIPS exceptions

<table>
<thead>
<tr>
<th>Type number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>External interrupt</td>
</tr>
<tr>
<td>8</td>
<td>Syscall exception</td>
</tr>
<tr>
<td>9</td>
<td>Breakpoint exception</td>
</tr>
<tr>
<td>12</td>
<td>Arithmetic overflow exception</td>
</tr>
<tr>
<td>13</td>
<td>Trace exception</td>
</tr>
</tbody>
</table>
Interrupt Processing in MIPS (cont’d)

• Cause and EPC registers of CP0 are loaded into k0 and k1 registers
  
  \begin{verbatim}
  mfc0 $k0,$13  # copy Cause register into $k0
  mfc0 $k1,$14  # copy EPC register into $k1
  \end{verbatim}

• One difference

  * Return address = address of the interrupted instruction
    » We need to add 4 to get to the instruction following the interrupted instruction
      \begin{verbatim}
      addiu $k1,$k1,4
      rfe
      jr $k1
      \end{verbatim}