Computer Organisation
Estructura de Computadores

Grado de Ingeniería Informática

ETSINF

Unit 8: I/O Synchronisation

Goals

- To understand the need for I/O synchronisation and the existing synchronisation mechanisms
- To write programs that synchronise with peripherals by polling or responding to interrupts
- To learn how exceptions work in general and on MIPS-like processors
- To understand how the exceptions mechanism gives support to OS system functions
- To program exception handlers and system functions for the MIPS R2000

Contents

Bibliography

- D. Patterson, J. Hennessy
  - ✓ *Computer organization and design. The hardware/software interface.* 4th edition. 2009. Elsevier
    - Chapter 6
- W. Stallings
  - ✓ *Computer Organization and Architecture. Designing for Performance.* 7th edition. 2006. Prentice Hall
    - Chapter 7
- C. Hamacher, Z. Vranesic, S. Zaky
  - ✓ *Computer Organization.* 5th edition. 2001. McGraw-Hill
    - Chapter 4

I – Synchronization mechanisms

The need for synchronisation

CPU

Peripheral

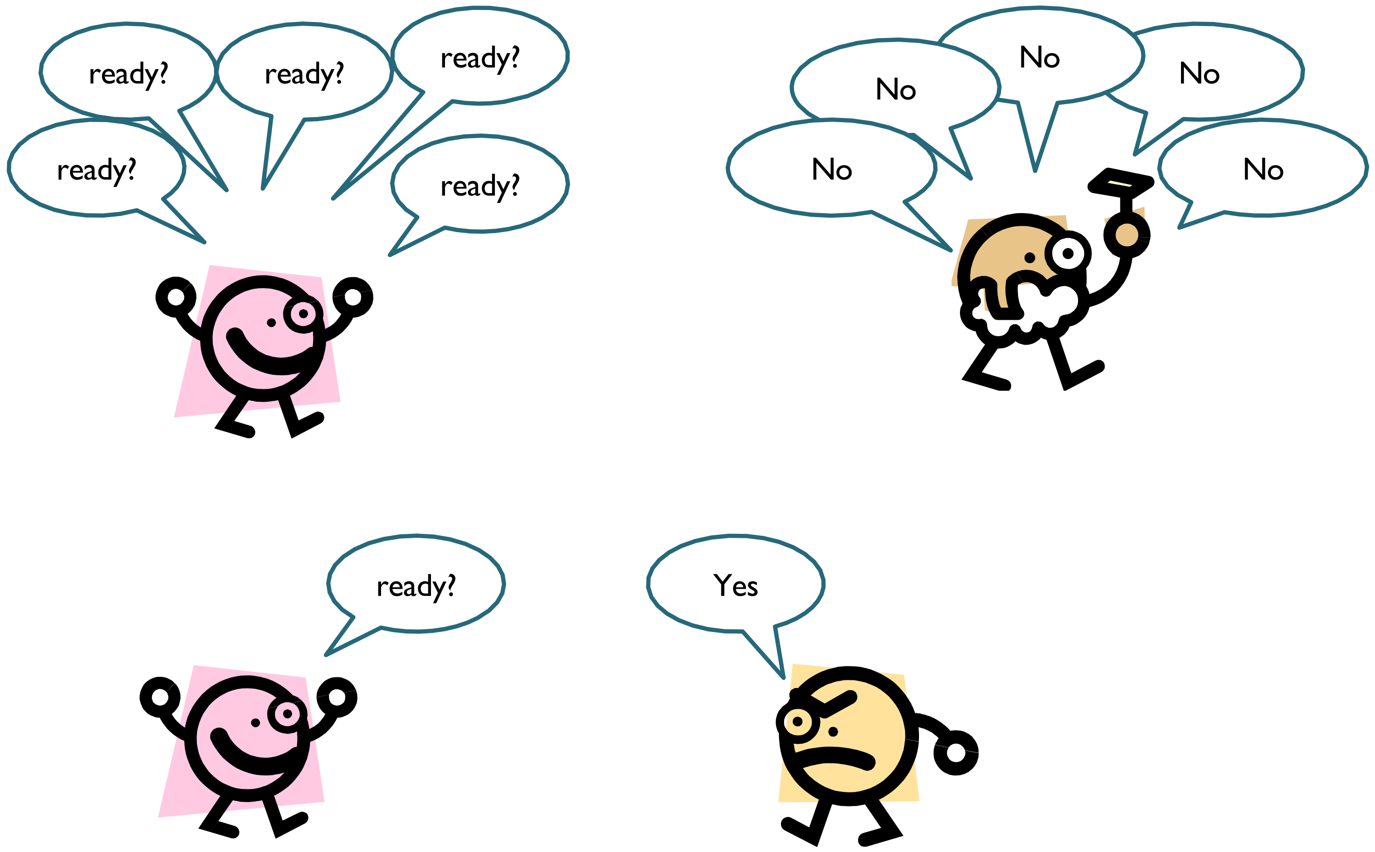
Synchronisation

- Concepts
  - ✓ Peripherals have their own working pace
    - A keyboard requires attention when a key is pressed
    - A printer needs its own time to process a character (or page…) before a new print command can be served
  - ✓ Synchronisation is the process whereby peripherals are used when it is the right time to use them – when the peripheral is ready
  - ✓ A peripheral is ready when it can communicate with the CPU
    - For an INPUT peripheral: when there is new data to transfer to the CPU
    - For an OUTPUT peripheral: when a previous operation has completed
    - Some peripherals are always ready, e.g., the 7-segment display
      - They don't require synchronisation → *DIRECT I/O devices*

Synchronisation schemes

- Direct I/O
  - ✓ Peripheral is always ready → no need for synchronisation
- Synchronized I/O
  - ✓ Synchronisation by Polling
    - The CPU has the initiative
    - A program checks the peripheral availability (*readiness*) continuously until peripheral is ready
      - The CPU is busy-waiting for the peripheral to be ready
  - ✓ Synchronisation by Interrupt
    - The peripheral has the initiative
    - When the peripheral becomes ready, it sends an interrupt signal to the CPU
      - Meanwhile, the CPU can do more productive work

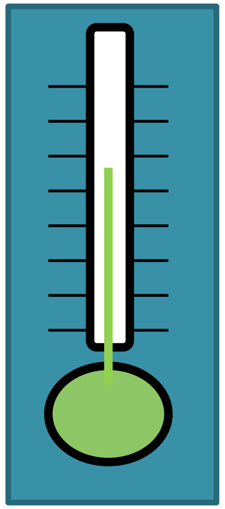
2 – Polling synchronisation



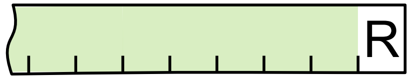
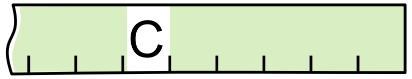
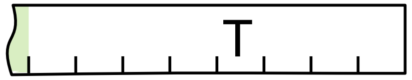
Interface elements for synchronisation

- Interface bits for the purpose of synchronisation
  - ✓ READY bit
    - Becomes active (e.g. R=1) when device is ready for a new operation
      - The peripheral writes this bit
    - Usually part of the Status register under the names *Ready*, *R*…
  - ✓ CANCEL bit
    - Setting this bit has the effect of clearing the Ready bit
    - Usually part of the Command register under names *Cancel*, *Clear*, *CL*…
  - ✓ Cancellation may be:
      - *Explicit*: CPU must assert the cancel bit to have the device reset the ready bit.
      - *Implicit or Automatic*: the adapter clears the ready bit whenever an interface register is accessed from the CPU.
        - NOTE: devices with automatic cancellation *do not* need a cancellation bit

Example 1: Thermometer



- Interface registers
  - ✓ Base Address BA= 0xFFFF0010
  - ✓ All unused bits = 0

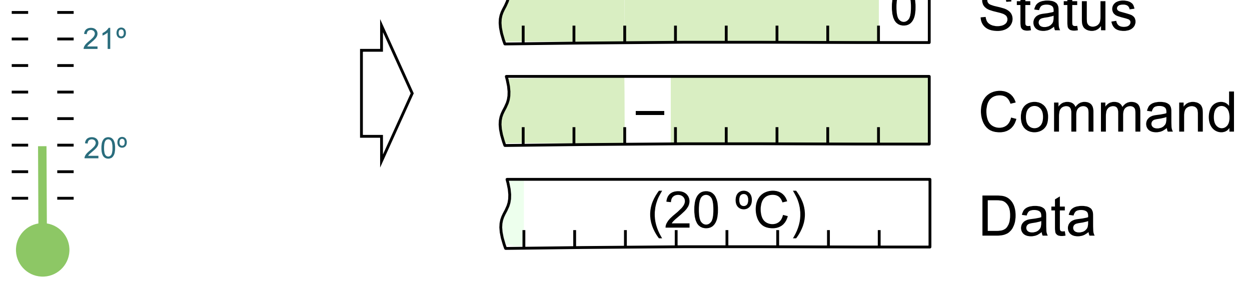
Name	Address	Access	Structure
Status	BA	Read	 R (bit 0): R=1 when temperature changes (Ready)
Command	BA+4	Write	 C (bit 5): when C is set to 1, R becomes 0 (Cancel) writing C=0 has no effect
Data	BA+8	Read	 T (bits 7…0): temperature

Example 1: Thermometer

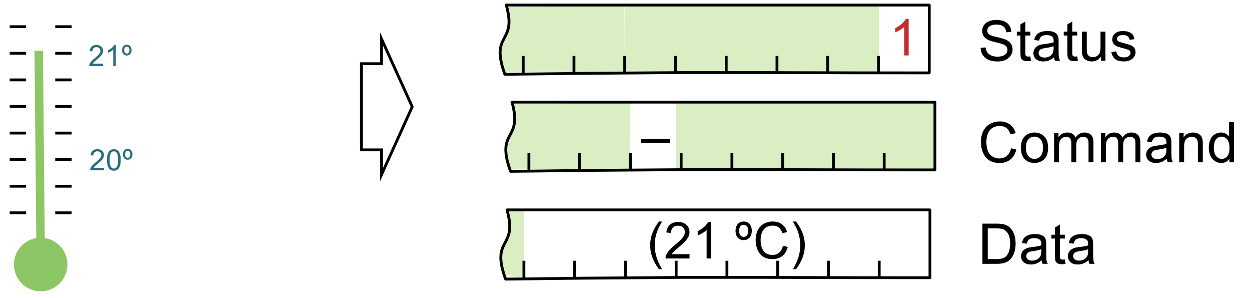
- Thermometer operation

Initial state:

Temperature = 20 ºC

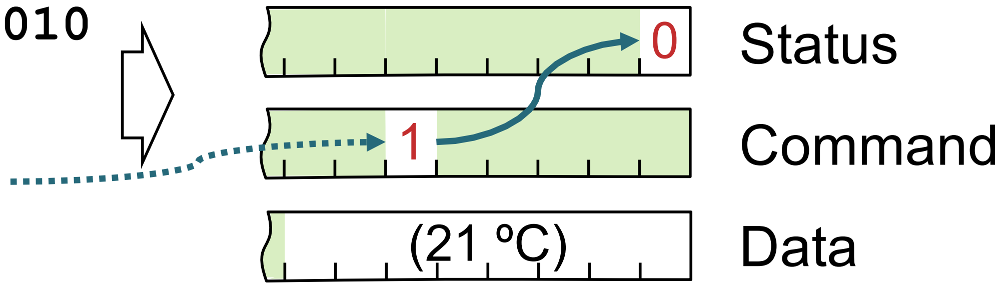


Peripheral ready: temperature changes to 21 ºC



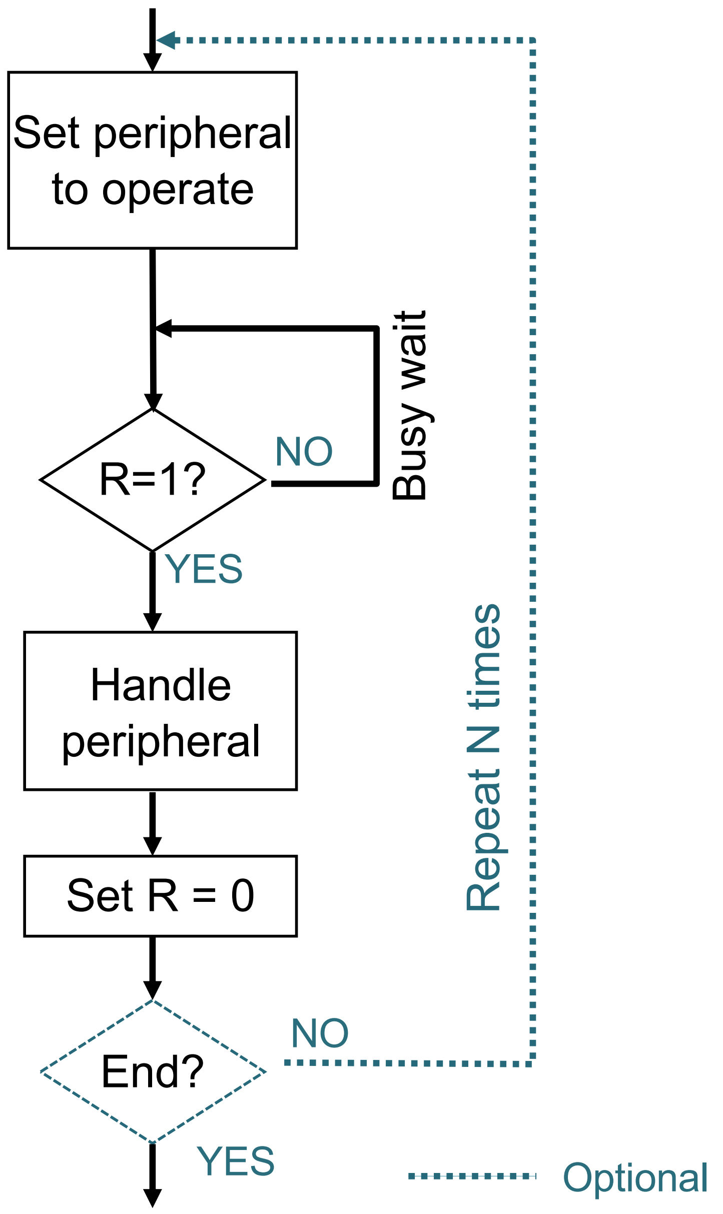
Cancellation command

```
la $t0,0xFFFF0010
li $t1,0x20
sb $t1,4($t0)
```

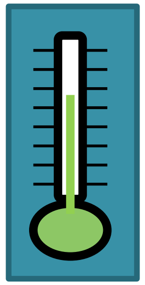


Programming polling synchronisation

- General scheme
  - ✓ First, the peripheral must be set up to operate
  - ✓ The program enters a loop polling for the device to be ready (R = 1)
  - ✓ When R=1 the peripheral must be handled (read/write its data)
  - ✓ A part of handling is cancelling the R bit (set R = 0) so that the peripheral can be used again
  - ✓ If necessary, the whole process is repeated



Example 1: Thermometer



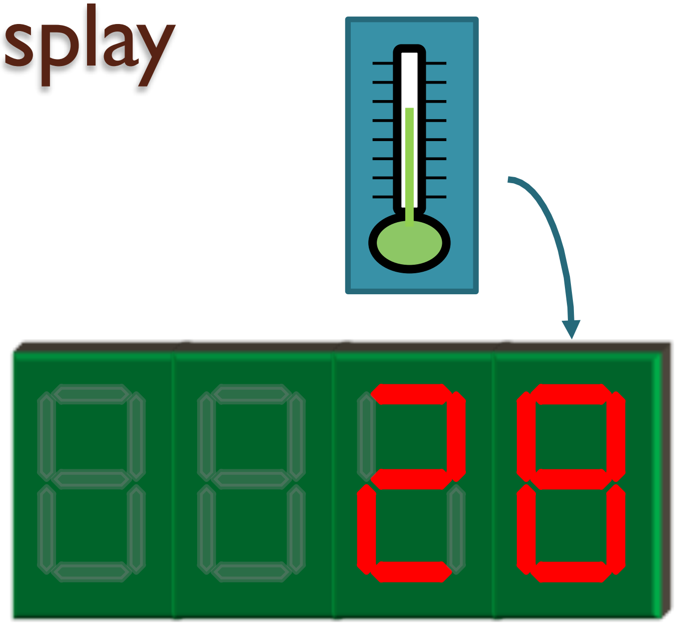
- Programming: read N temperatures

```
        .data   0x1000000
Temp:   .byte 0

        .text   0x00400000
start:  ............
        li $t3, N
loop:   ............
        ............
        jal ReadTemp
        ............
        addi $t3, $t3, -1
        bgtz $t3, loop
        ............

.end
```

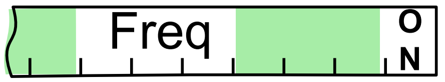
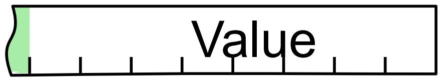
N times

```
ReadTemp:
        la $t0,0xFFFF0010      } Prepare base address

Poll:   lb $t1,0($t0)          }
        andi $t1,$t1,1         } Polling loop
        beqz $t1,Poll          }

        lb $t2,8($t0)          } Data transfer
        sb $t2, Temp           }

        li $t1,0x20            } Cancellation
        sb $t1,4($t0)          }

        jr $ra
```

Example 2: Thermometer + Display



- 4-digit display
  - Displays an 8-bit signed integer
  - Base address BA = 0xFFFF0020
  - Interface register

Name	Address	Access	Structure
Command	BA	Write	
Data	BA+4	Write	

Activates display and blinking:
- ON (bit 0): 0 off; 1 displays VALUE
- Freq (bits 6..4): blinking frequency in Hz
- Freq = 0: continuous display

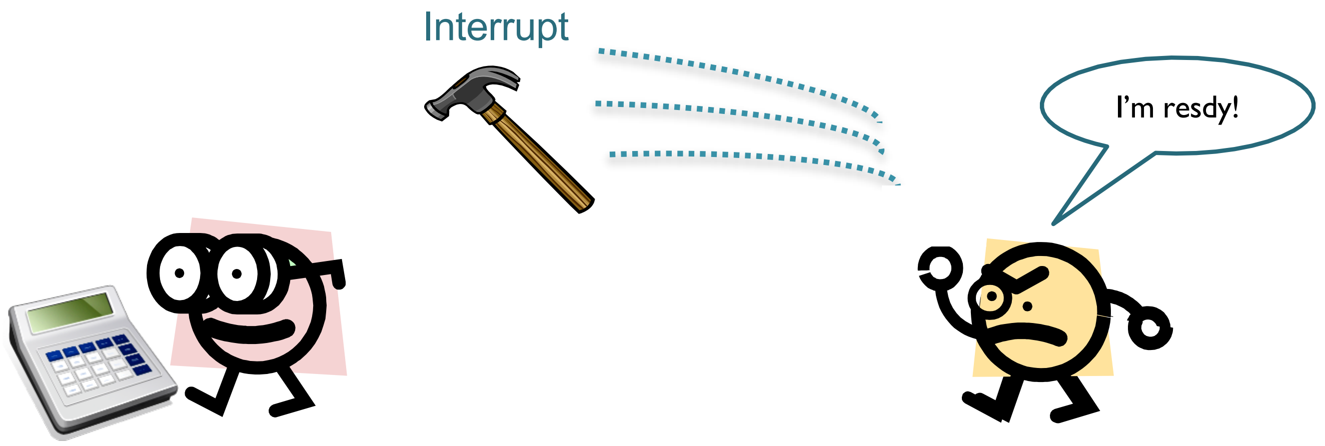
VALUE to display:
Signed 2'sC 8-bit integer (-127 .. +127)
Value is displayed when ON is set to 1

Example 2: Thermometer + Display

- Programming
  - ✓ Read temperature
  - ✓ Show on display
  - ✓ If temperature > 100 °C then blink at 2 Hz

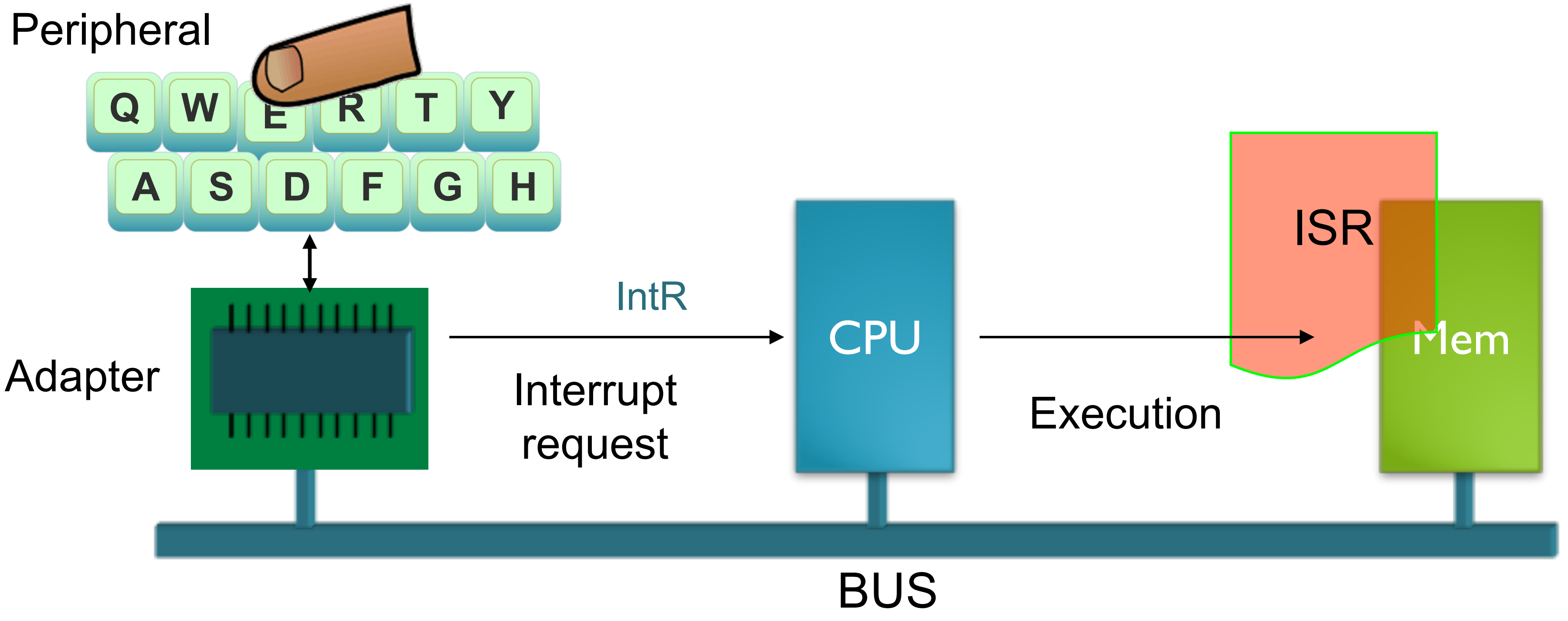
```
ReadTemp: la $t0,0xFFFF0010   # Therm's BA
Poll:     lb $t1,0($t0)       # Polling for
          andi $t1,$t1,1      #   R = 1
          beqz $t1,Poll
          lb $t2,8($t0)       # Read temp.
          li $t1,0x20         # Bit 5 = 1 (C)
          sb $t1,4($t0)       # Cancellation
          la $t0,0xFFFF0020   # Display's BA
          sb $t2, 4($t0)      # Value = temp
          li $t3,100          # If temp<=100…
          bgt $t2,$t3,blink
          li $t1, 0x01        # …then display
          sb $t1,0($t0)       # continuous
          jr $ra
blink:    li $t1,0x21         # …else
          sb $t1,0($t0)       # display
          jr $ra              # blinking
```

3 – Interrupt synchronisation



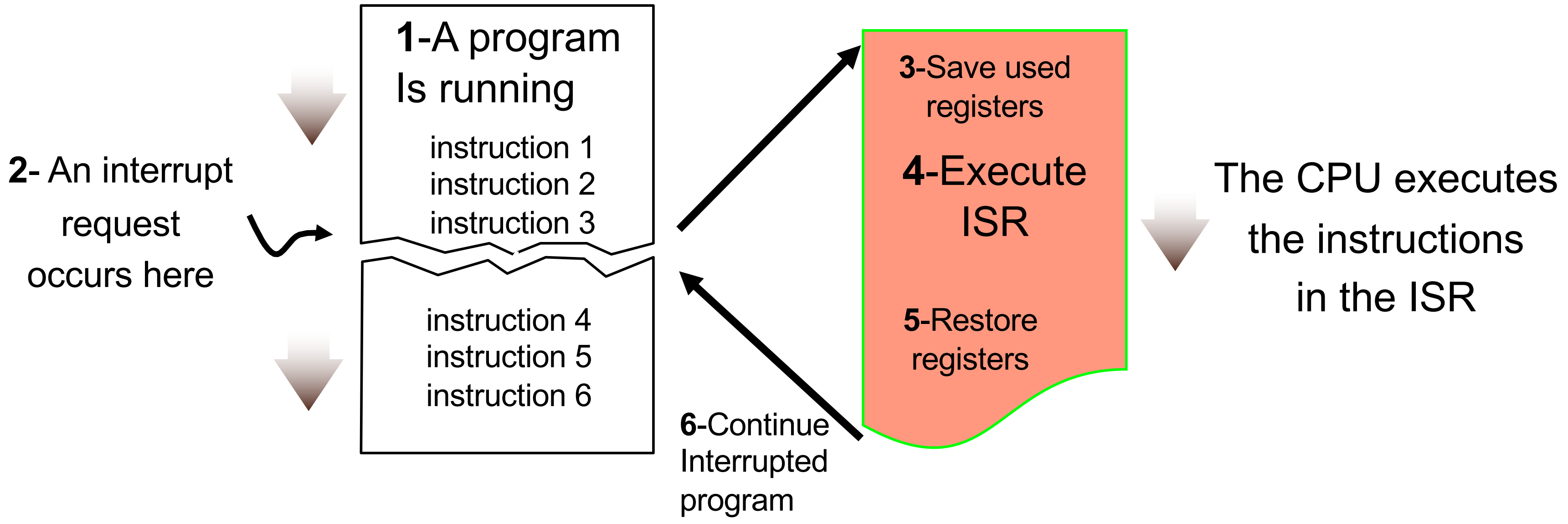
Interrupt synchronisation

- The interrupt request
  - When the peripheral is ready for a new operation, its adapter sends an electrical signal to the CPU (IntR: Interrupt Request)
  - The CPU then interrupts the running program and executes a particular piece of code called the Interupt Service Routine (ISR) or Interrupt Handler



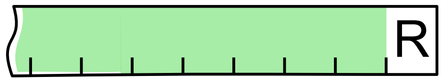
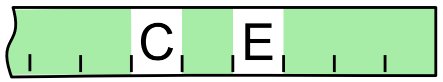
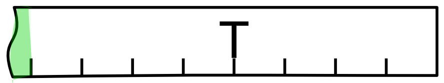
Interrupt synchronisation

- Control flow upon interrupts
  - ✓ The CPU may be running any process when an interrupt request arrives
  - ✓ That point in time is unpredictable, and the CPU needs to jump to the ISR in a way that allows it returning to the interrupted process at the exact point where the interrupt request occurred
  - ✓ Moreover, the ISR itself needs to use the same register file as the interrupted process

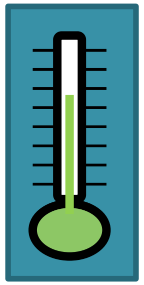


Example 3: Thermometer (with interrupt)

- Extension to the thermometer interface
  - ✓ Connected to IntR0

Name	Address	Access	Structure
Status	BA	Read	 R = 1 when temperature changes – This activates
IntR0			*iff* E=1 in Command register below
Command	BA	Write	 E (bit 3): Enable interrupt. Write 1 to enable interrupts and 0 to disable them C (bit 5): Clear interrupt. Write 1 to make R = 0, which in turn deactivates IntR0
Data	BA+4	Read	 T (bits 7…0): temperature

Example 3: Thermometer (with interrupt)



```
          .data   0x1000000
Temp:     .byte ?

          .text   0x00400000
start:    la $t0,0xFFFF0010
          li $t1,0x08
          sb $t1,0($t0)
          ............
          ............
          ............        # do other
          ............        # stuff
          lb $t2 Temp         # Use Temp
          ............
          ............

 .end
```

```
ISR_IntR0:
          la $t0,0xFFFF0010     Base address
          lb $t2,4($t0)         } Transfer
          sb $t2, Temp
          li $t1,0x20           } Cancel IntR0
          sb $t1,0($t0)
          return_from_ int
```

Compare with 13

Example 4: Thermometer + display

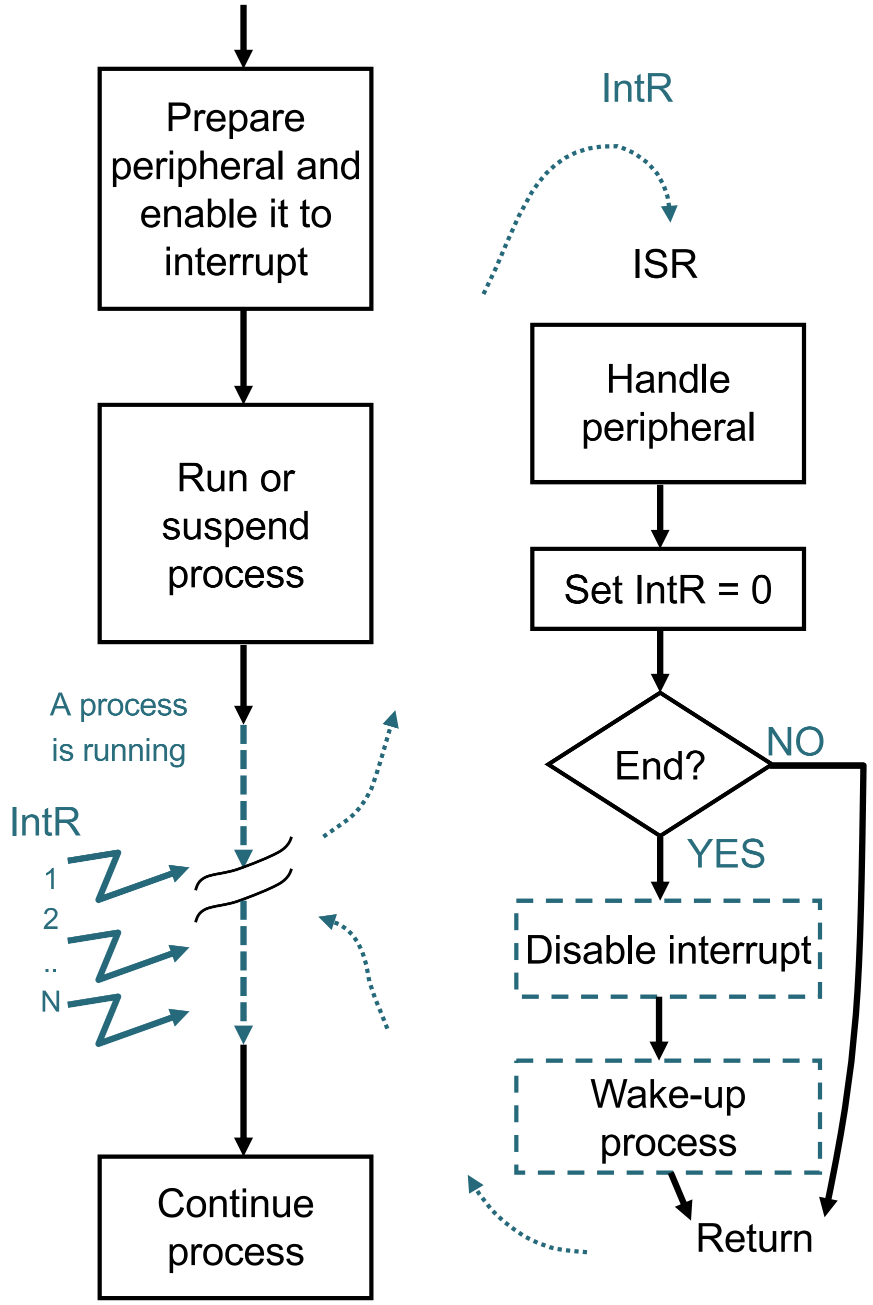
- ISR
  - ✓ Read temperature
  - ✓ Show temp in display
  - ✓ If temperature > 100 °C then blink at 2 Hz
  - ✓ Requires thermometer interrupts enabled

```
ISR_IntR0:
        la $t0,0xFFFF0010    # THERM. BA
        li $t1,0x28
        sb $t1,4($t0)        # Cancel IntR
        lb $t2,8($t0)        # Read temp.
        la $t0,0xFFFF0020    # DISPLAY BA
        sb $t2, 4($t0)       # Value = temp
        li $t3,100           # If temp>=100…
        bgt $t2,$t3,blink
        li $t1, 0x01         # …continuous
        sb $t1,0($t0)        # display
        return_from_int
blink:  li $t1,0x21          # else
        sb $t1,0($t0)        # blinking
        return_from_int      # display
```

Compare with 15

Interrupt synchronisation

- General scheme
  - ✓ The peripheral is set up to operate and enabled to synchronise using interrupt requests
  - ✓ The running program proceeds (or becomes suspended on multitask OS)
  - ✓ When an interrupt is requested (IntR), its associated ISR is executed
  - ✓ The request is handled in the ISR
  - ✓ The ISR MUST cancel the request (IntR=0)
  - ✓ The process may require several (*N*) IntR's to finish handling the device (until "*End*")
  - ✓ To stop receiving interrupts from the peripheral, its associated IntR must be disabled
  - ✓ After executing the ISR, the OS may need to wake-up the suspended process



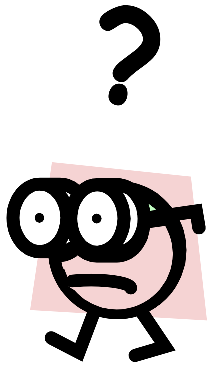
4 – Interrupt support

What CPU resources provide for interrupt support?

Can interrupts be silenced?

What about multiple interrupts?

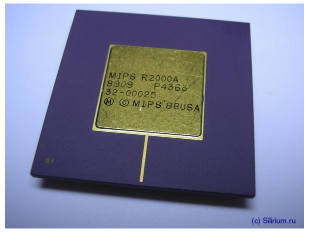
Where are ISR's located?



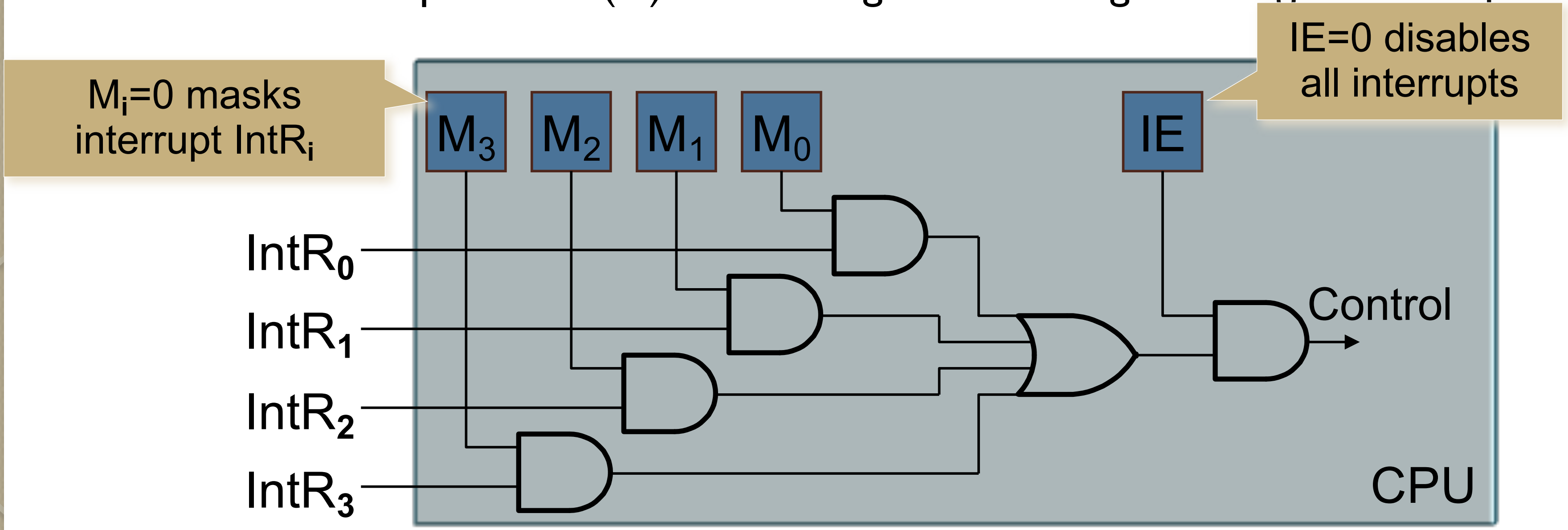
Interrupt support

- CPU support for interrupts
  - ✓ Support for interrupts is a part of a more general event handling mechanism in the processor: *exceptions and exception handling*
  - ✓ The control unit and instruction set must enable stopping and resuming program execution
  - ✓ Interrupt handling requires specific registers in the CPU
- I/O Interface support for interrupts
  - ✓ I/O adapters must include appropriate logic to support interrupt synchronisation
  - ✓ The I/O interface includes bits to control the interrupt mechanism (e.g., *Enable* and *Clear* bits)

CPU support for interrupts

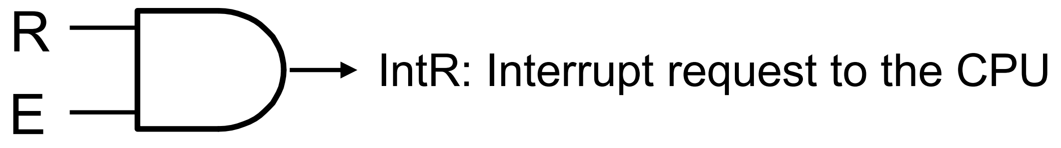


- Can CPUs support multiple interrupt sources?
  - ✓ Yes, CPUs use to have several interrupt request (IntR) inputs
  - ✓ Each interrupt has its own ISR attached
- Can interrupts be silenced?
  - ✓ Yes – the CPU can be configured to ignore one or more interrupts
    - Mask bits allow the CPU to ignore individual interrupt inputs
    - An Interrupt Enable (IE) bit allows global enabling/disabling of interrupts



Adapter support for interrupts

- ✓ The I/O interface includes bits for controlling if and how the peripheral will use interrupts
- ✓ E: Interrupt Enable bit
  - This bit is often part of the command or control register of the interface
  - Useful to enable (E=1) or disable (E=0) the sending of interrupt requests to the CPU
- ✓ R: Ready bit
  - This bit indicates readiness of the peripheral for a new operation
  - When interrupts are enabled in the interface (E=1), IntR is asserted when R=1

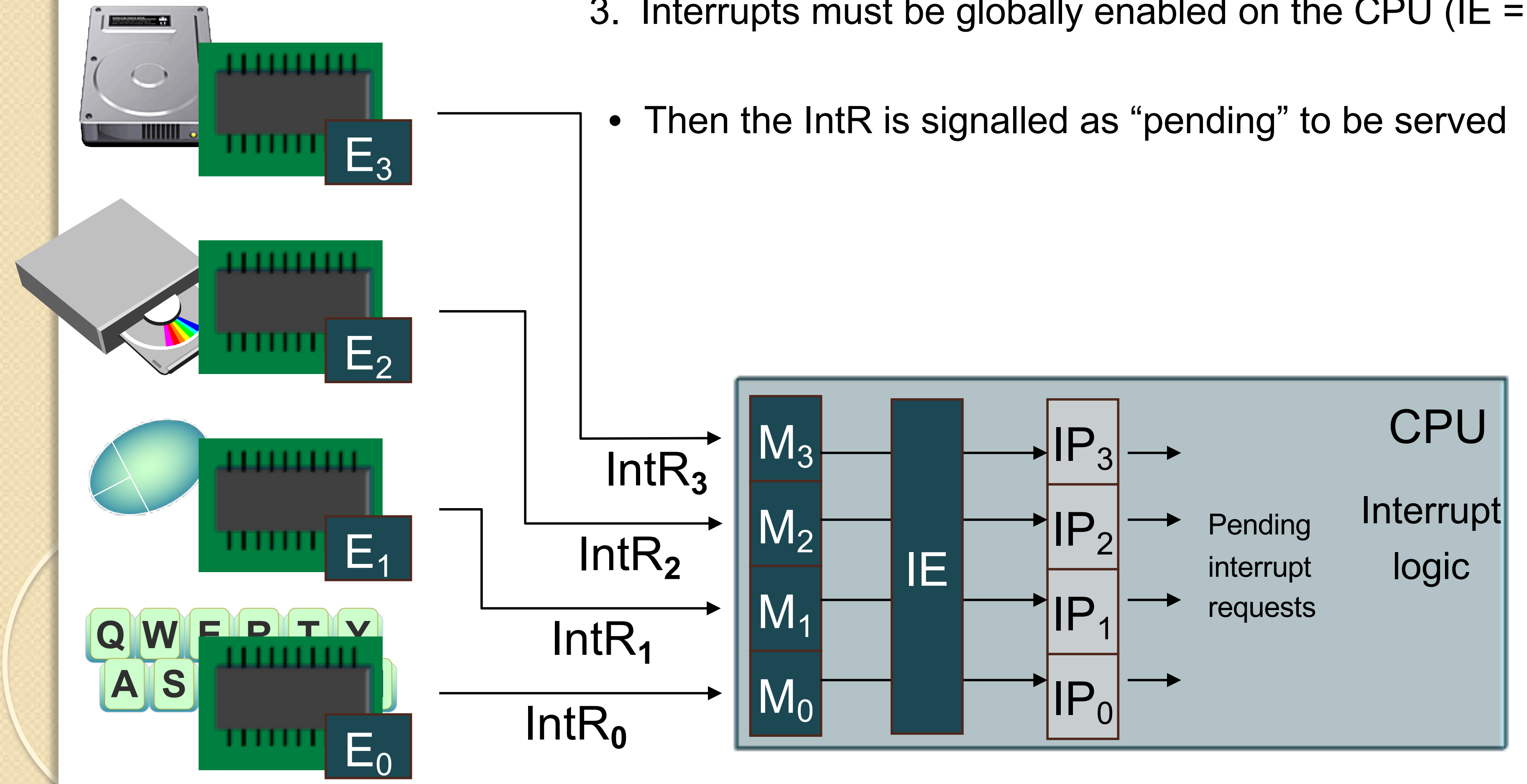


- ✓ C: Cancel
  - Setting C=1 makes R become 0, which cancels the interrupt request
  - Sometimes cancellation is automatic when an interface register is accessed (read or written)

The path of an interrupt request

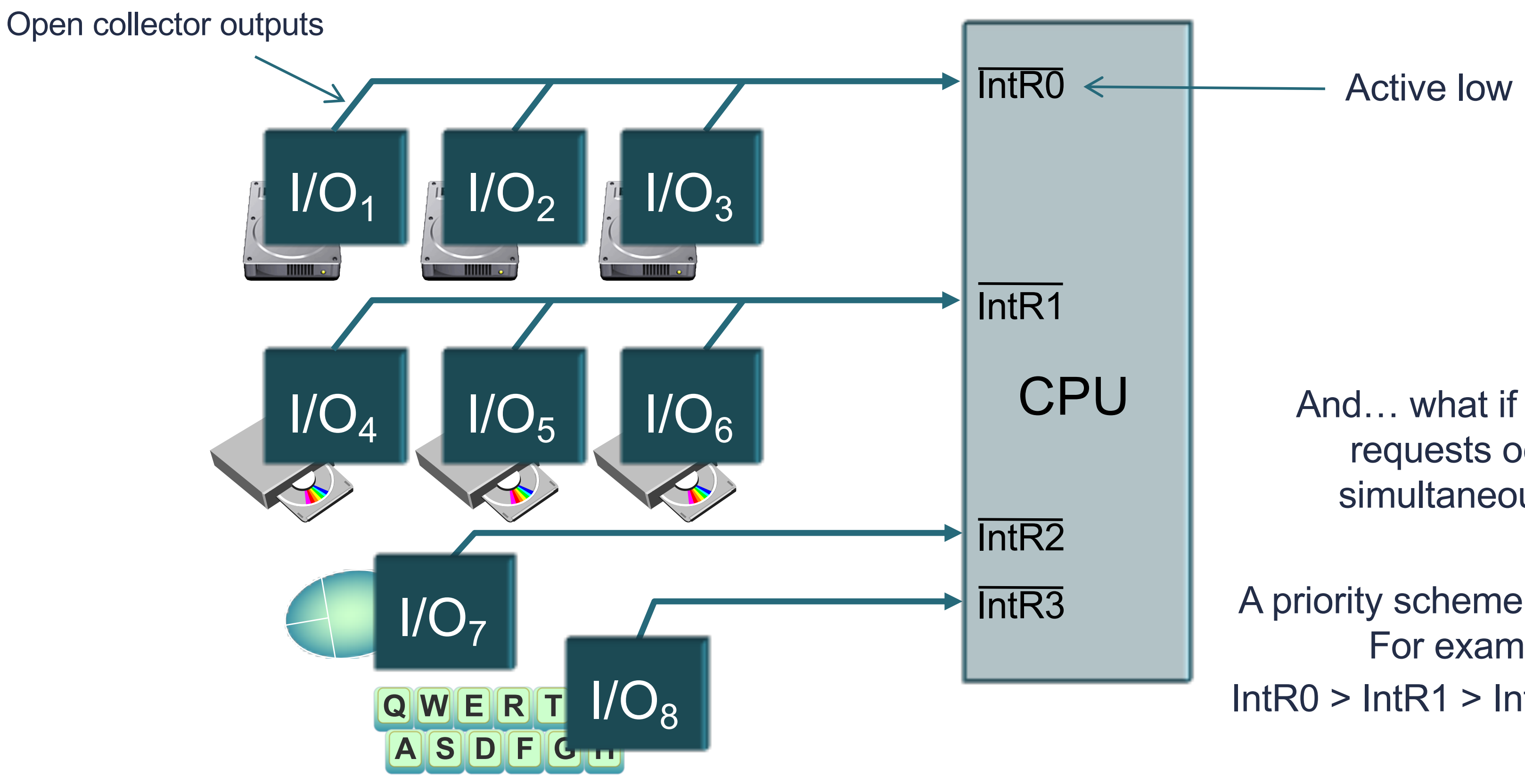
- Three conditions must hold for an IntR to interrupt the CPU:
  1. The IE_i bit must be asserted on its interface ($E_i = 1$)
  2. The corresponding IntR must be unmasked ($M_i = 1$)
  3. Interrupts must be globally enabled on the CPU ($IE = 1$)

  - Then the IntR is signalled as “pending” to be served



Connecting multiple interrupting devices

- How to connect more peripherals than IntR lines in the CPU?
- Connection schemes

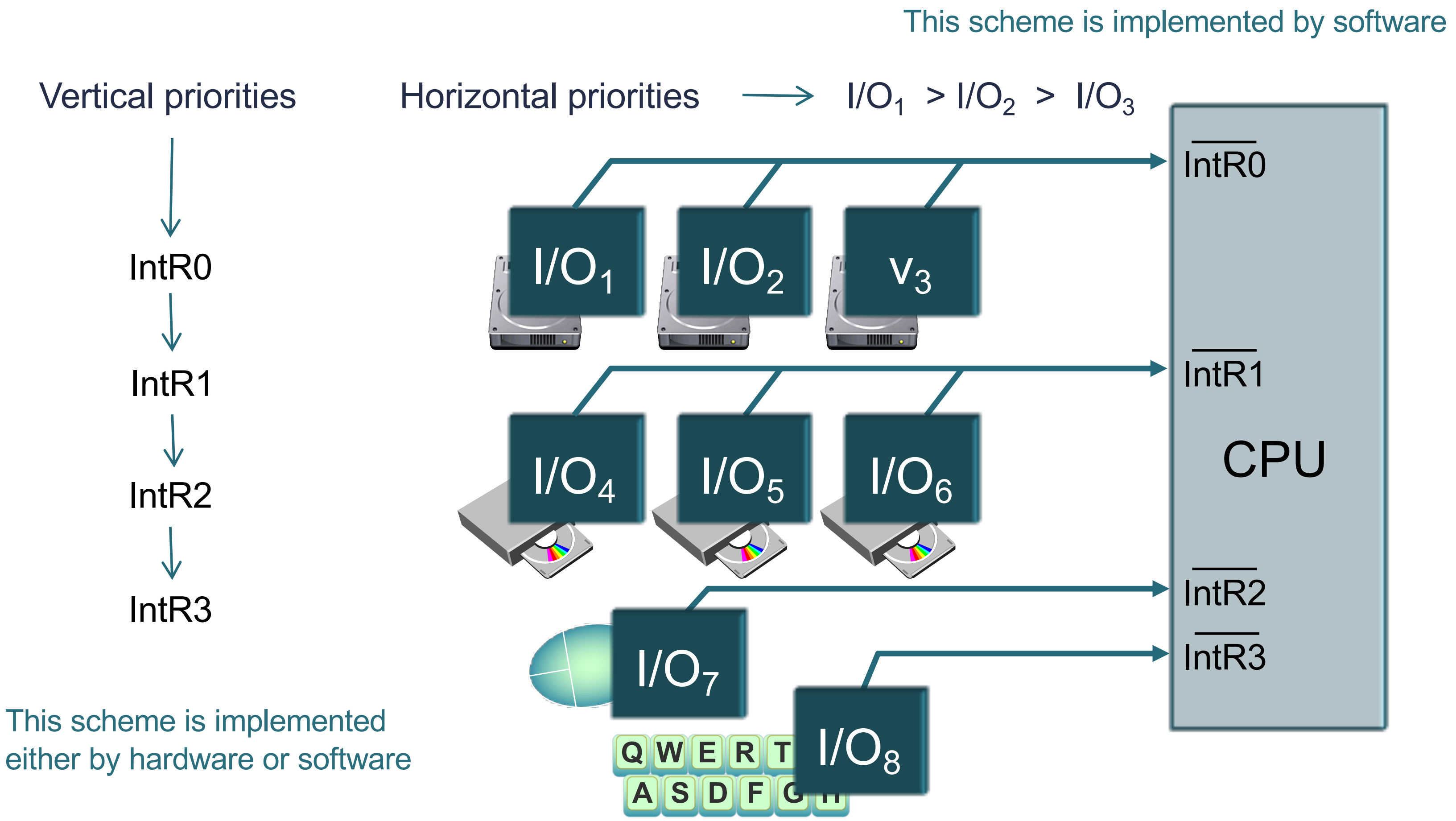


And… what if several requests occur simultaneously?

A priority scheme is needed. For example
IntR0 > IntR1 > IntR2 > IntR3

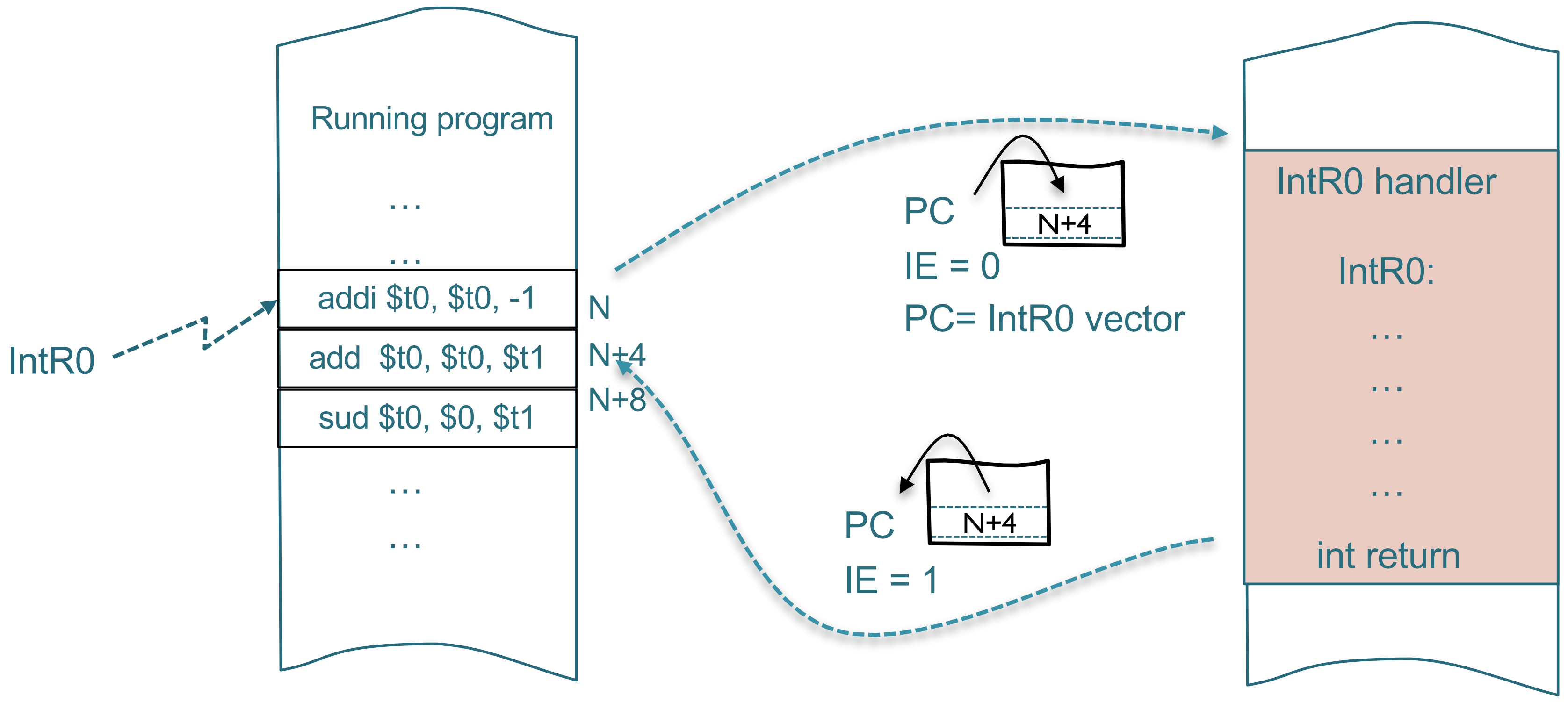
Interrupt priorities

- Two priority schemes apply:
    - Vertical priorities
    - Horizontal priorities



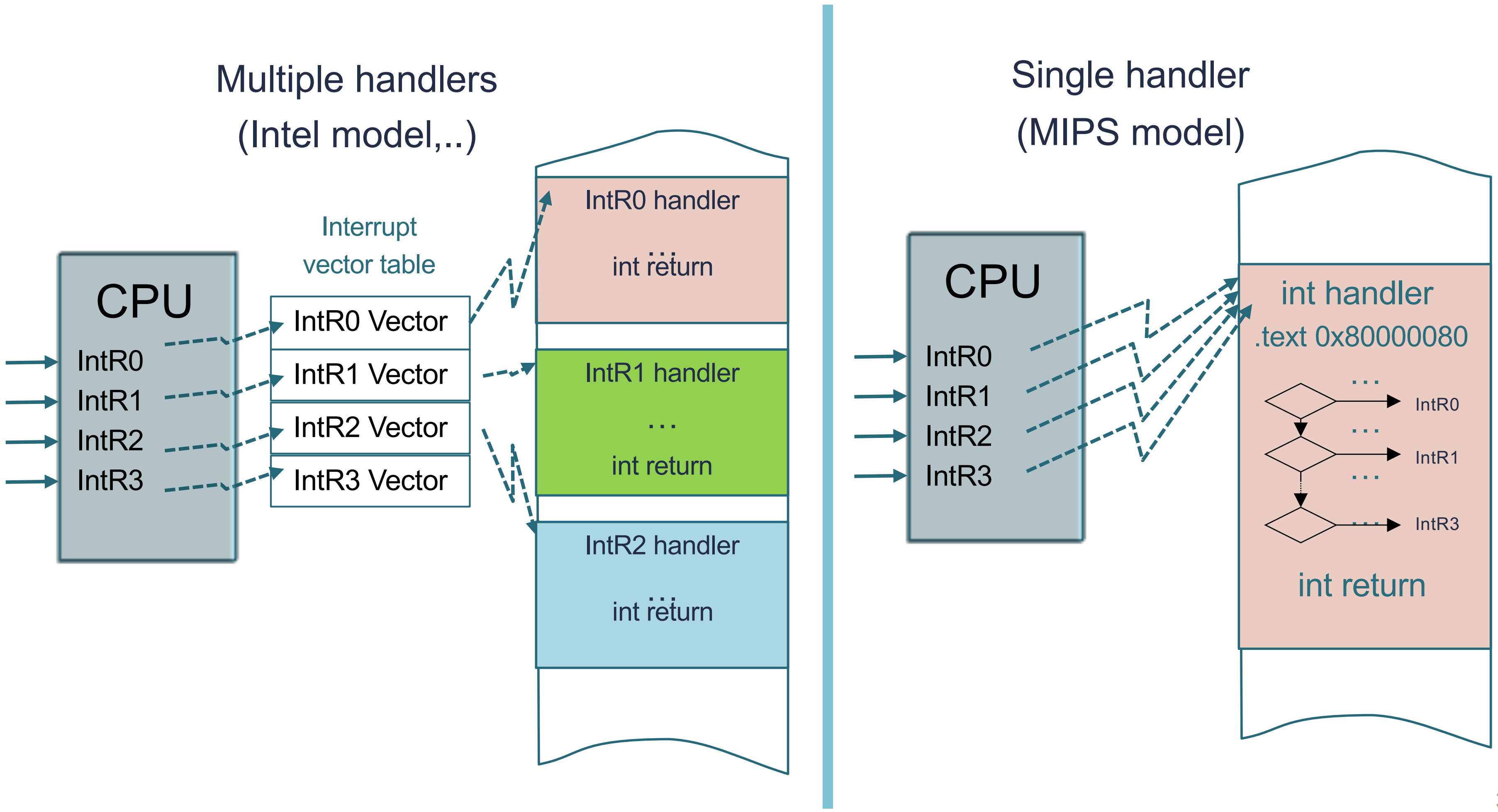
ISRs

- How does the CPU jump to the ISR?
  - ✓ The starting address of an ISR is called the INTERRUPT VECTOR
  - ✓ On each cycle, the CPU checks for pending interrupt requests
  - ✓ If there is any pending request, the ISR is called



ISRs

- Where are ISRs located?
  - ✓ Each interrupt request has a different ISR
  - ✓ There are two schemes:



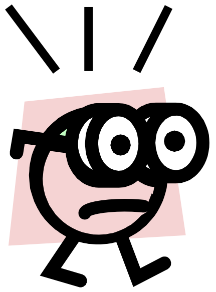
Synchronisation schemes: comparative

- Polling
  - ✓ Peripheral configuration
    - Interrupts disabled
  - ✓ Loop:
    - Read status
    - Iterate while not ready
  - ✓ Then handling
  - ✓ Then cancellation
    - explicit or automatic

- Interrupts
  - ✓ Peripheral configuration
    - Interrupts enabled
  - ✓ No loop
    - Asynchronous request
  - ✓ The handler is a separate piece of code, decoupled from synchronisation
  - ✓ Cancellation is a part of handling
    - explicit or automatic

5 – MIPS: Interrupts and exceptions

Handling "exceptional" events



$$\frac{1}{0} = ?$$

Exceptions

- *Exceptions* are events reflecting *special situations* which require execution of a specific *exception handler*
- Sources of exceptions
  - ✓ Interrupt requests from peripherals
  - ✓ Within the CPU: some ALU operations, the bus access logic, the TLB
  - ✓ Certain instructions
- Applications of exceptions
  - ✓ Peripheral synchronisation
  - ✓ Handling errors in running programs
  - ✓ Virtual memory support
  - ✓ Controlled execution of programs
  - ✓ Implementation of OS services

Exception types in MIPS R2000

EXCEPTIONS

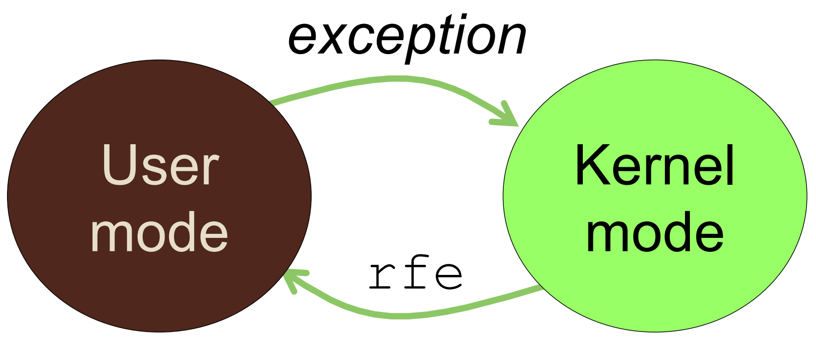
- 6 level-active external hardware interrupts (int_5*.. int_0*)
- 1 exception for handling OS calls (instruction **`syscall`**)
- 2 software interrupts (SW_1, SW_0)
- 3 exceptions for TLB handling
- 2 exceptions for use of forbidden or misaligned addresses
- 2 exceptions for bus errors – non existent address, parity error…
- 1 exception for arithmetic overflow
- 1 exception for illegal (reserved) instruction
- 1 exception for absent coprocessor
- 1 exception for handling breakpoints (instruction **`break`**)

MIPS R2000 Operating Modes

- The CPU may run in one of two operating modes:
  - ✓ User mode – intended for user programs
  - ✓ Kernel mode (AKA Supervisor mode) – intended for the OS, interrupt handlers and especially enabled programs
- Only in kernel mode, the CPU may:
  - ✓ Use some *privileged* registers
  - ✓ Execute some *privileged* instructions
  - ✓ Access the full range of memory addresses
    - User mode: 0x00000000 … 0x7FFF FFFF
    - Kernel mode: 0x00000000 … 0xFFFF FFFF

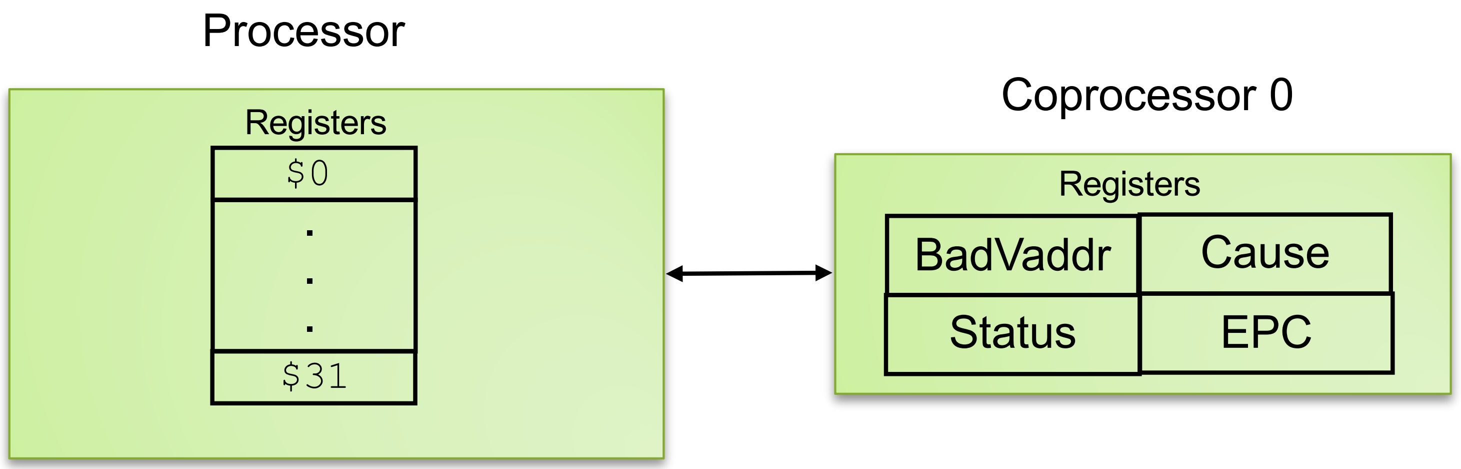
The mode change

- From User to Kernel
  - ✓ Is the processor's reaction to an exception occurrence
    - Mode is changed to *kernel*
    - Interrupts are automatically disabled
    - The CPU starts to execute the exception handler
- From Kernel to User
  - ✓ Occurs when the CPU executes the privileged instruction **`rfe`** (*restore from exception*)
    - As a privileged instruction, **`rfe`** can only be executed while in *kernel* mode



CP0: the exception coprocessor

- Coprocessor 0 (CP0) implements the support to modes
  - ✓ CP0 can only be accessed while in Kernel mode
  - ✓ CP0 contains the registers needed for handling exceptions: cause of the exception, PC value at the time of the exception, and other relevant details



CP0 registers for exception handling

Number	Name	Description
`$8`	$BadVaddr	Virtual address that caused page miss (when applicable)
`$12`	$Status	Interrupt masks, running mode and global interrupt enable
`$13`	$Cause	Cause of exception and pending exceptions
`$14`	$EPC	PC address of the instruction affected by the exception

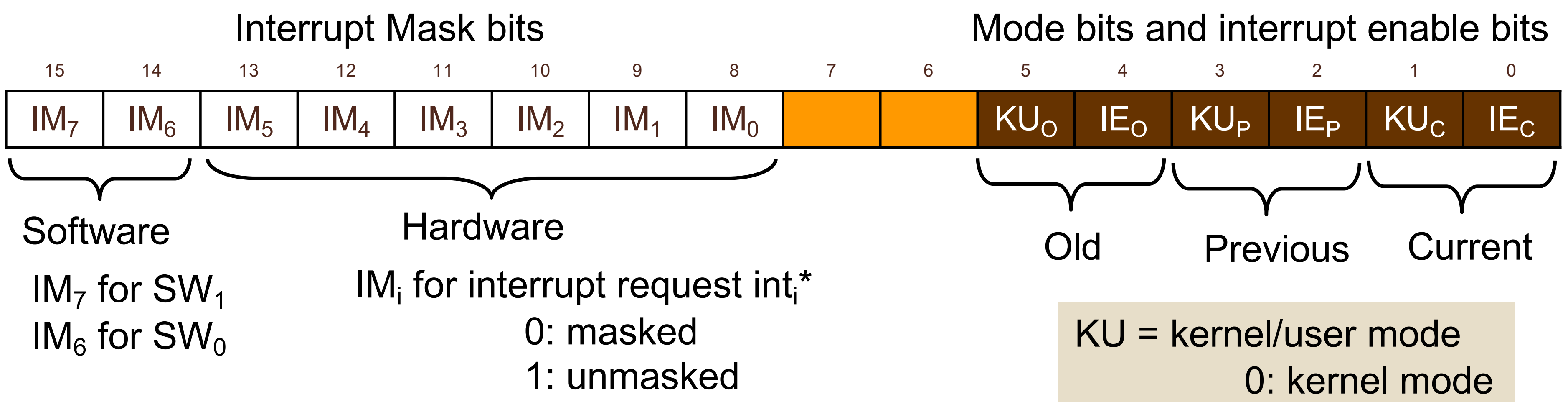
Instructions for accessing CP0 registers

Move from CP0: `mfc0` $R_{general}$, R_{CP0} $R_{general} \leftarrow R_{CP0}$

Move to CP0: `mtc0` $R_{general}$, R_{CP0} $R_{general} \rightarrow R_{CP0}$

Example: Reading the Cause register into `$t0`: `mfc0 $t0,$Cause`

Status Register (SR) structure



IM_7 for SW_1
IM_6 for SW_0

IM_i for interrupt request int_i*
0: masked
1: unmasked

KU = kernel/user mode
0: kernel mode
1: user mode

IE = Interrupt Enable
0: disabled
1: enabled

Cause Register (CR) structure

Pending interrupts (1 = pending)

Exception code

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IP_5	IP_4	IP_3	IP_2	IP_1	IP_0	SW_1	SW_0			EC_3	EC_2	EC_1	EC_0		

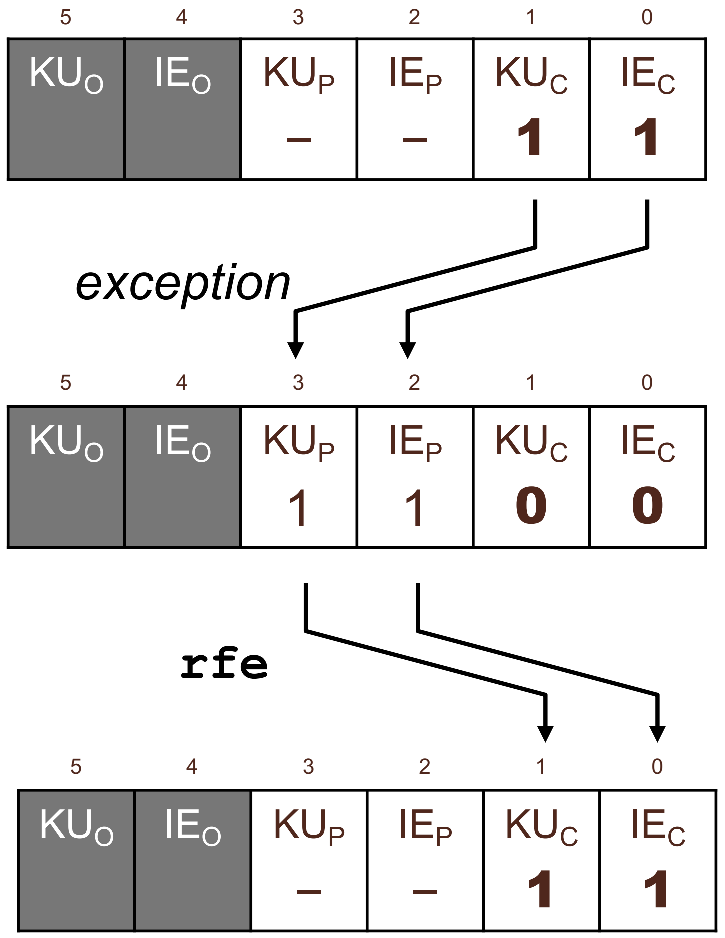
$EC_{3..0}$	Name	Exception cause
0	INT	Interrupt request (hardware or software interrupts)
1	TLBPF	(TLB) Attempt to write on privileged page
2	TLBML	(TLB) Attempt to fetch instruction from invalid page
3	TLBMS	(TLB) Attempt to read data from invalid page
4	ADDRL	Address error during read (data or code)
5	ADDRS	Address error during write
6	IBUS	Bus error on instruction memory
7	DBUS	Bus error on data memory
8	SYSCALL	System call (execution of instruction syscall)
9	BKPT	Execution of break instruction (breakpoint)
10	RI	Attempt to execute a reserved instruction
11	CU	Coprocessor unavailable
12	OVF	Arithmetic overflow

Exception Program Counter register (EPC)

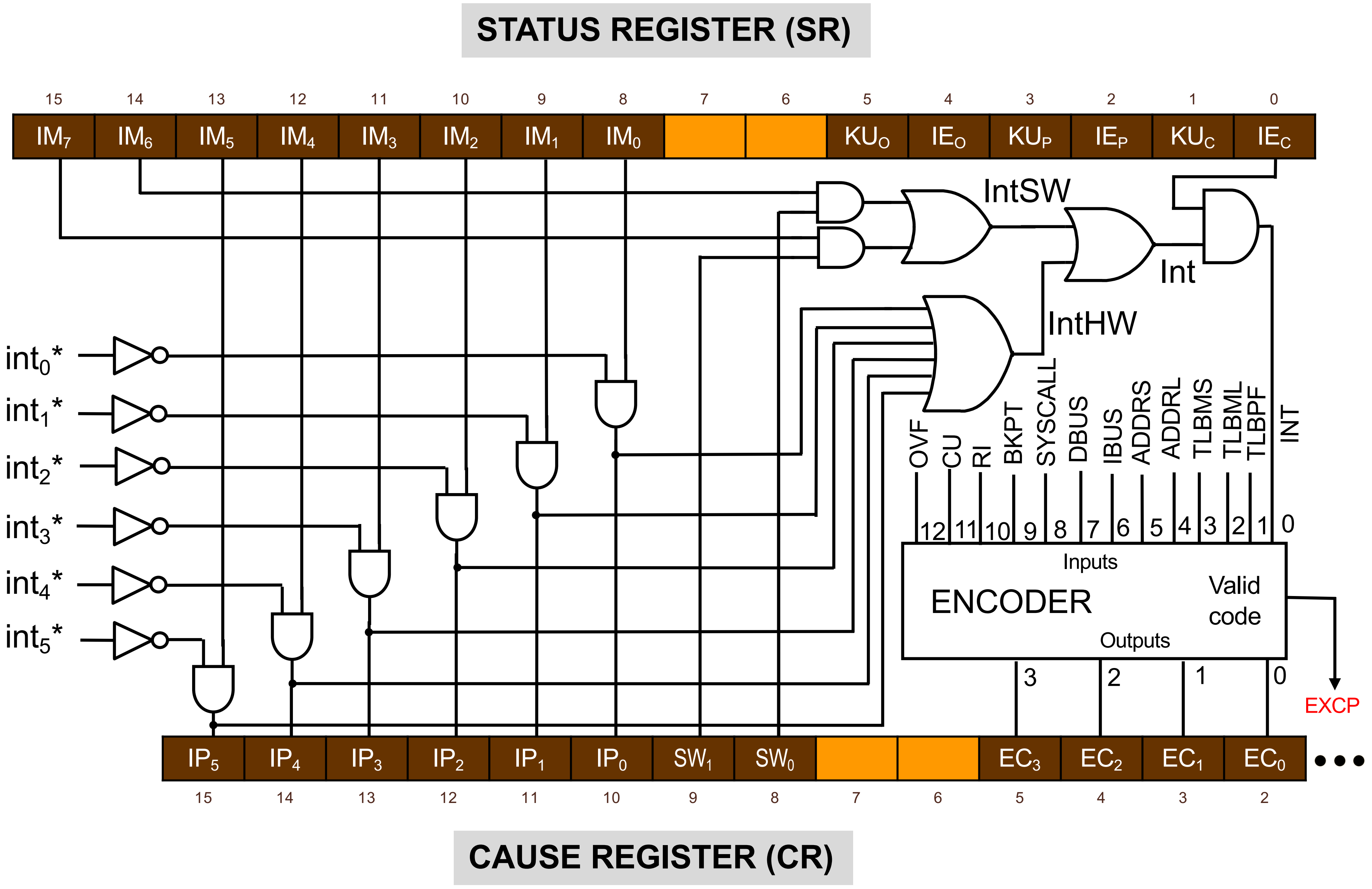
- The EPC contains the PC value when the exception occurred
  - ✓ The instruction pointed to by EPC has not completed its execution
- Depending on the type of exception, EPC points to…
  - ✓ Page faults: the instruction whose fetch failed, or the load/store instruction that originated the failed access
  - ✓ Arithmetic overflow, bus errors, etc.: the offending instruction
  - ✓ Interrupts: the instruction that was about to be executed when the interrupt request was detected

The mode change in CP0

- A user program is running
  - Interrupts are enabled
  - CPU runs in User mode
- An exception occurs
  - *Current* values assigned to *previous*
  - Current KU and IE are set to 0
  - EPC = PC value
  - Cause and BadVaddr registers are updated
  - PC = start address of exception handler
- Handler execution
  - Kernel mode, interrupts disabled
- Restore (rfe):
  - Back to the initial state

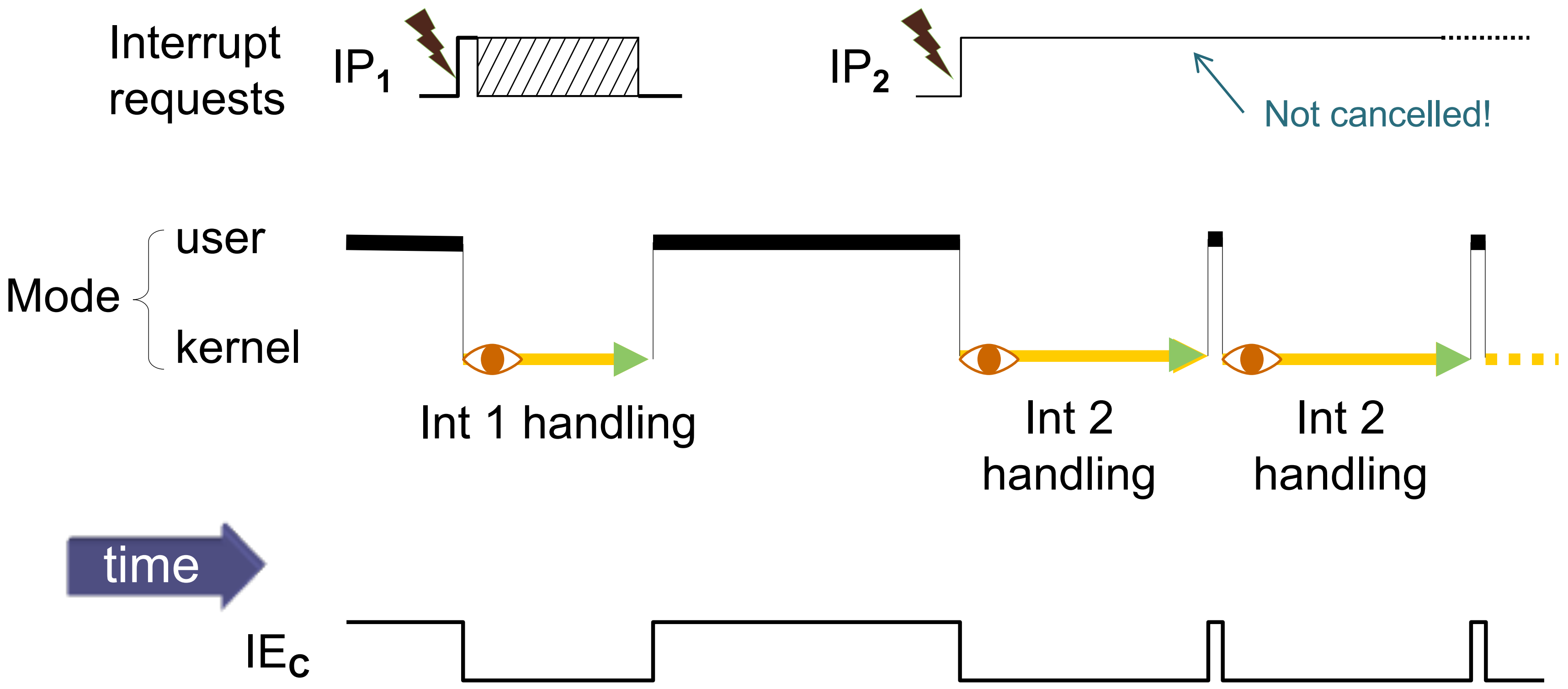


Hardware for exception control



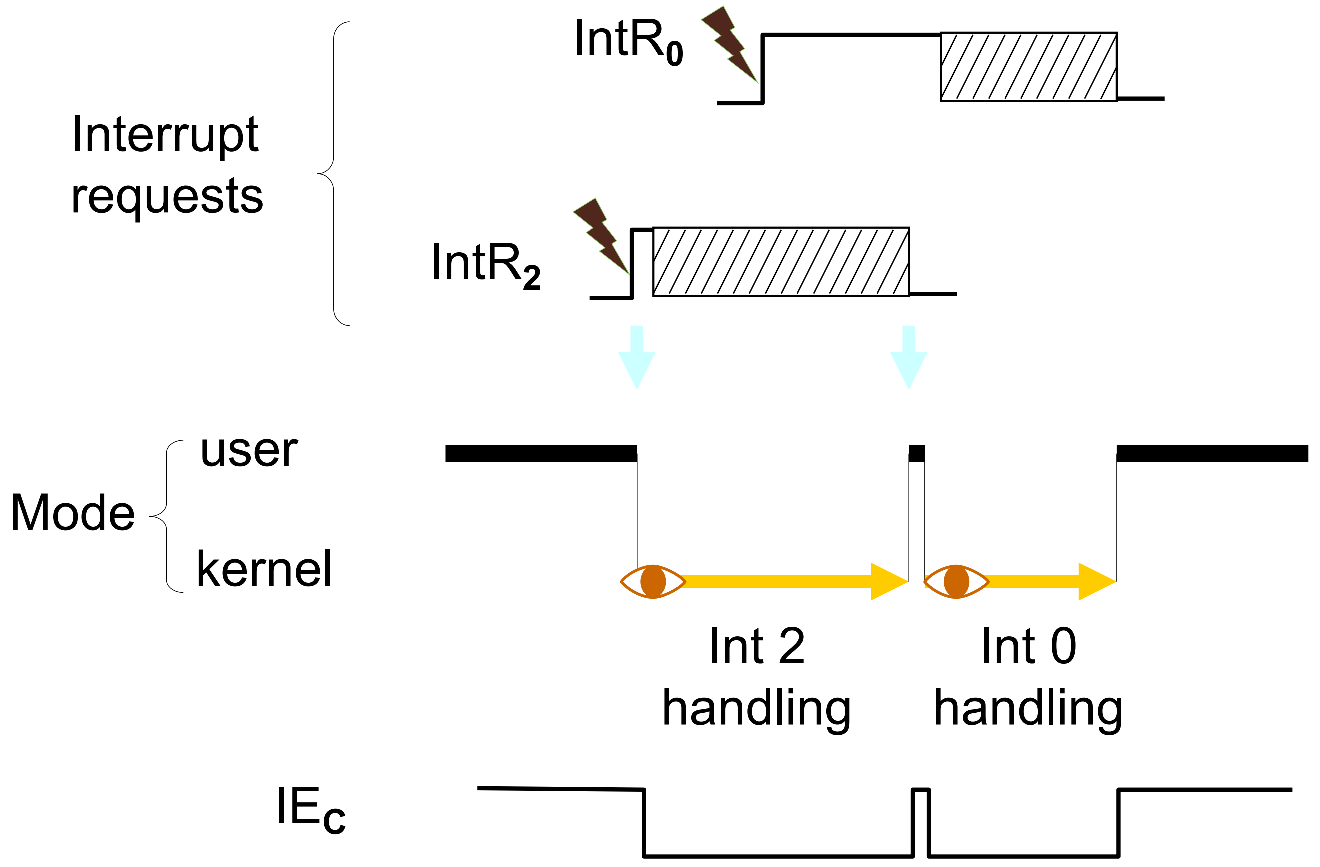
Interrupt handling

- Example chronogram 1
  - ✓ The peripheral's interrupt request must be cancelled as part of the interrupt handling
  - ✓ In this example, Int1 is handled correctly; Int2 is not

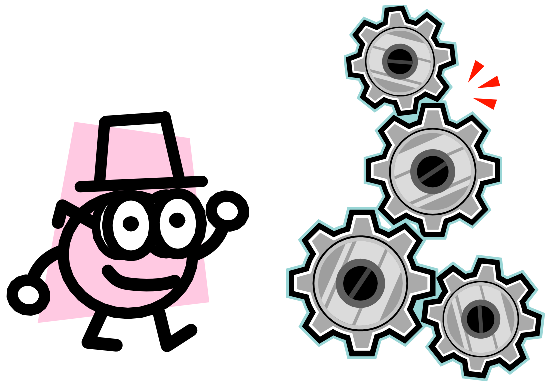


Interrupts while executing an ISR

- Example chronogram 2:
  - ✓ An int0 request arrives while servicing int2



6 – MIPS: Exception Handler Design



From exception to handler: automatic actions

C0 registers upon exception:

Status (\$12): KU_C , IE_C → KU_P , IE_P

0 , 0 → KU_C , IE_C

Cause (\$13): Cause of exception → $EC_{0..3}$ (IP_n, if interrupt)

EPC (\$14): PC of affected instruction → \$EPC

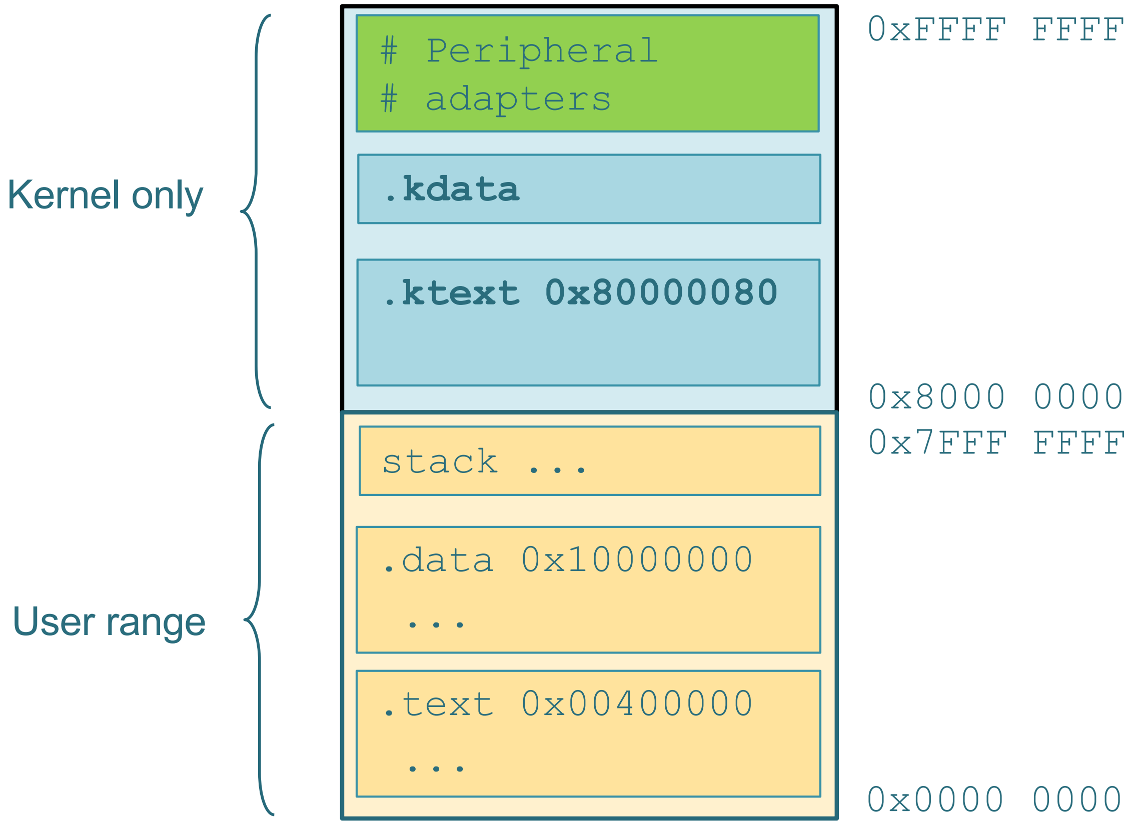
And then

Starting address of exception handler → PC

i.e.

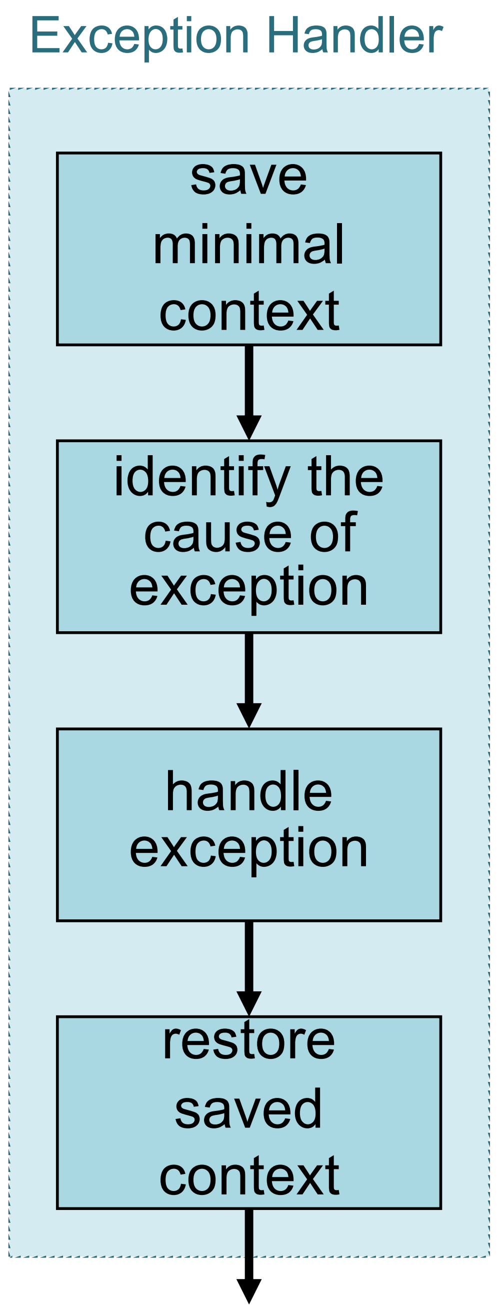
0x8000 0080 → PC

MIPS R2000 memory layout



Exception handler design

- Context management
  - ✓ The handler must be executed without affecting the running program
- Identifying and handling the cause
  - ✓ All exceptions involve the execution of the same handler
- Atomicity
  - ✓ Execution of the handler cannot be interrupted
  - ✓ The handler's code must not cause new exceptions – or it must be prepared for that
  - ✓ Interrupts are disabled during the execution of the handler



Handler's variables

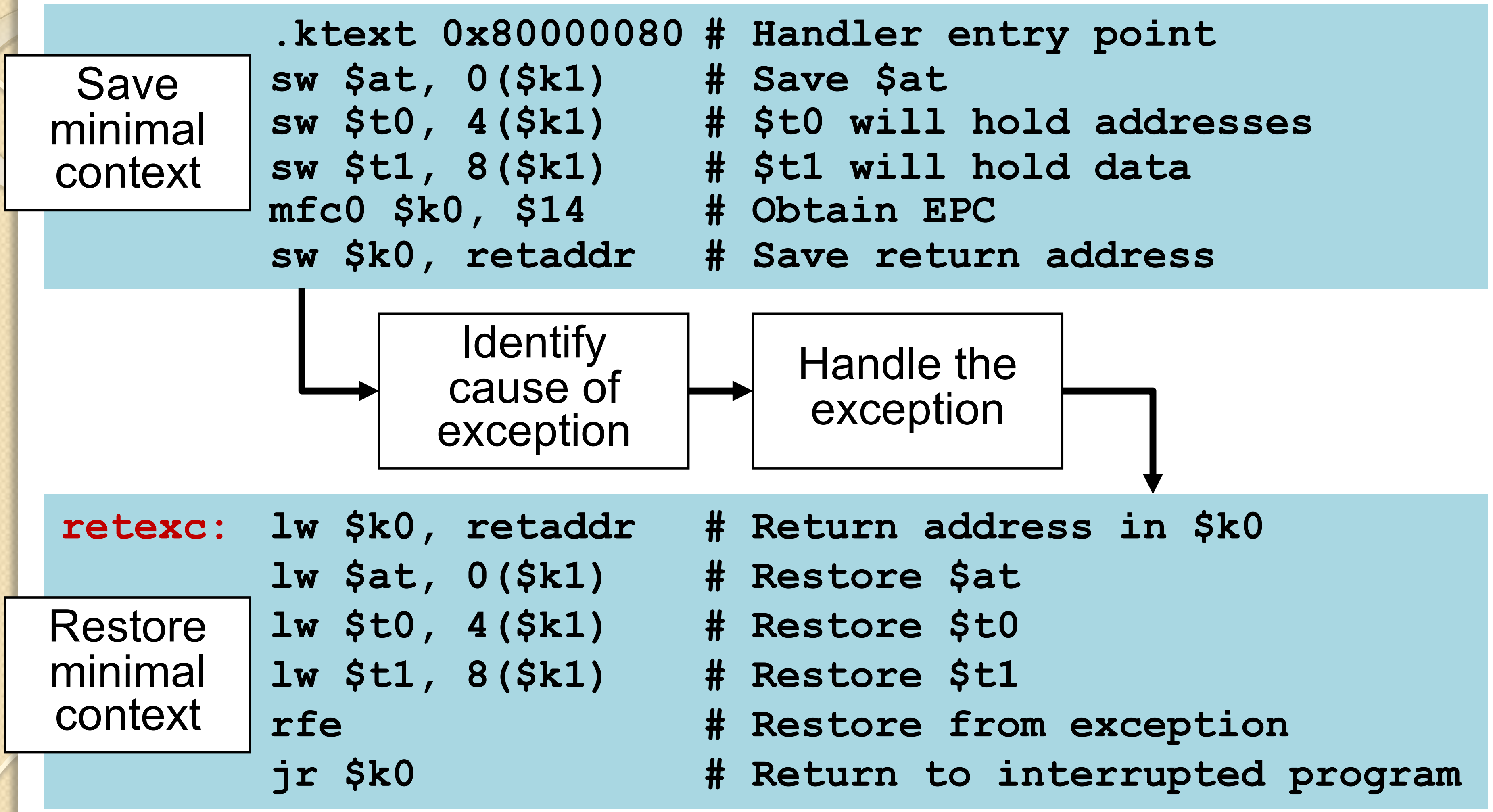
- Needs
  - ✓ Enough space to temporarily save the minimal context:
    - Registers used by the handler $t0, $t1, and $at (used by the assembler)
    - Return address
  - ✓ Specific variables needed for each particular handling routines
- Dedicated registers
  - ✓ Registers $k0 and $k1 are reserved for the OS and they are NOT a part of the programs' context
    - $k0 is used as a temporary register by the handler and will contain the return address
    - $k1 is used to hold the base address of the minimal context

Handler's variables

- Needs
  - ✓ Enough space to temporarily save the minimal context:
    - Registers used by the handler $t0, $t1, and $at (used by the assembler)
    - Return address
  - ✓ Specific variables needed for each particular handling routines
- Dedicated registers
  - ✓ Registers $k0 and $k1 are reserved for the OS and they are NOT a part of the programs' context
    - $k0 is used as a temporary register by the handler and will contain the return address
    - $k1 is used to hold the base address of the minimal context

```
            .kdata
## space for the minimal context
savereg:    .word 0,0,0  # for saving $t0,$t1, and $at
retaddr:    .word 0      # for saving the return address
```

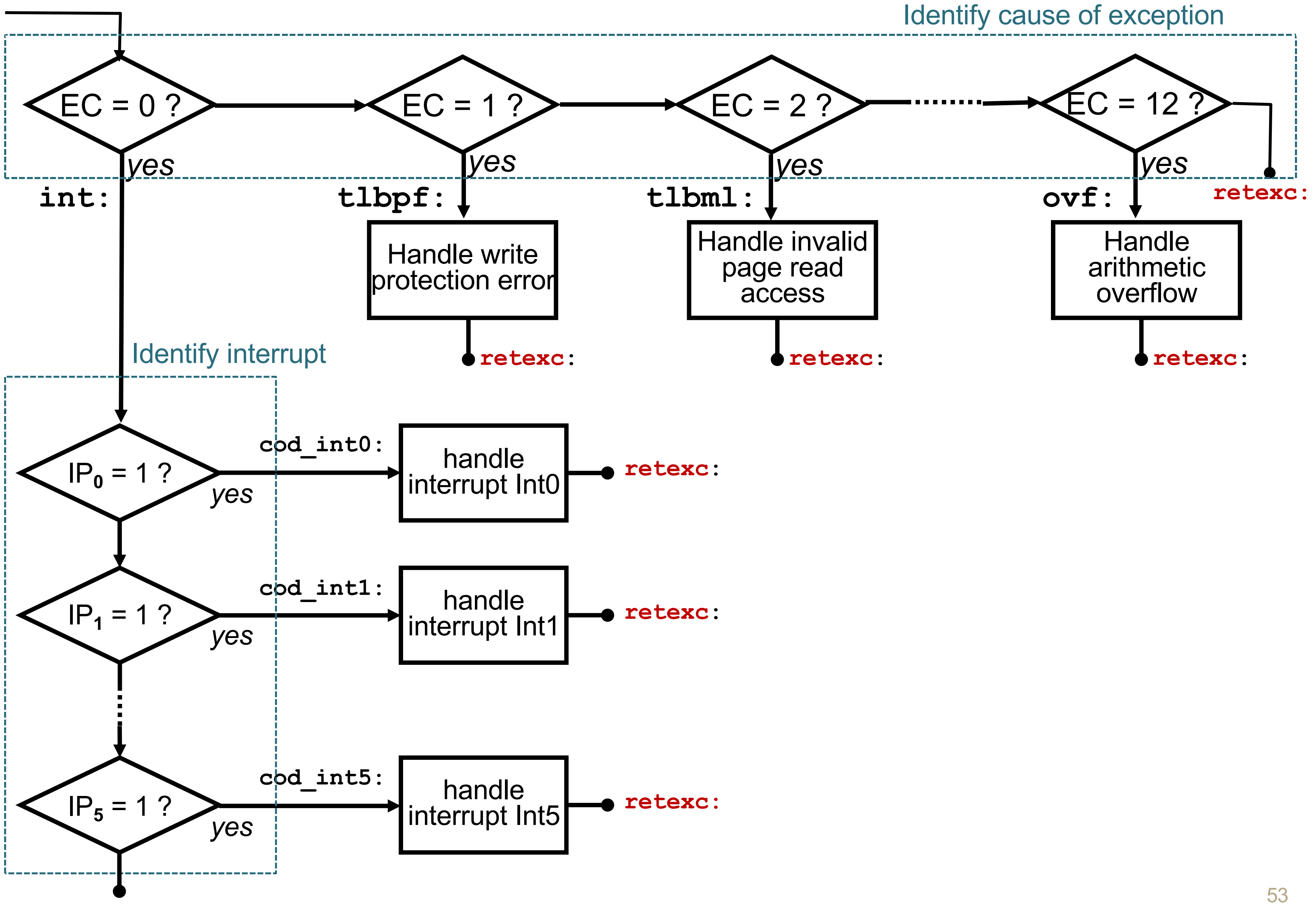
Saving and restoring the minimal context



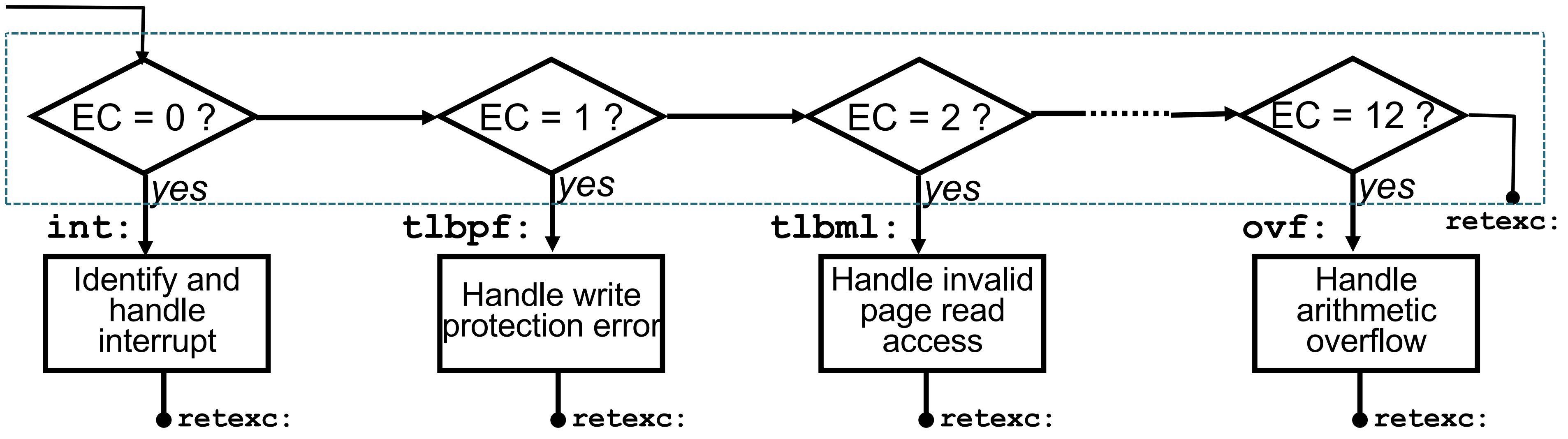
```
         .ktext 0x80000080 # Handler entry point
         sw $at, 0($k1)    # Save $at
         sw $t0, 4($k1)    # $t0 will hold addresses
         sw $t1, 8($k1)    # $t1 will hold data
         mfc0 $k0, $14     # Obtain EPC
         sw $k0, retaddr   # Save return address
```

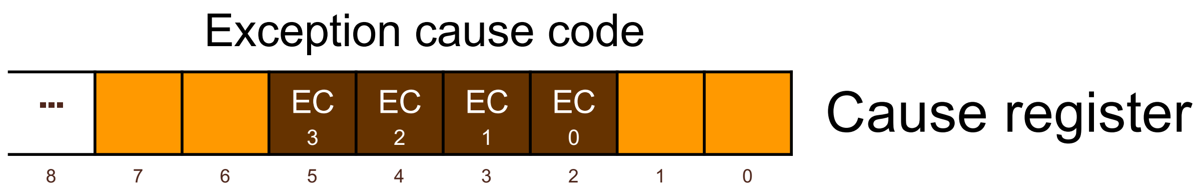
```
retexc:  lw $k0, retaddr   # Return address in $k0
         lw $at, 0($k1)    # Restore $at
         lw $t0, 4($k1)    # Restore $t0
         lw $t1, 8($k1)    # Restore $t1
         rfe               # Restore from exception
         jr $k0            # Return to interrupted program
```

Identify cause of exception: flow diagram



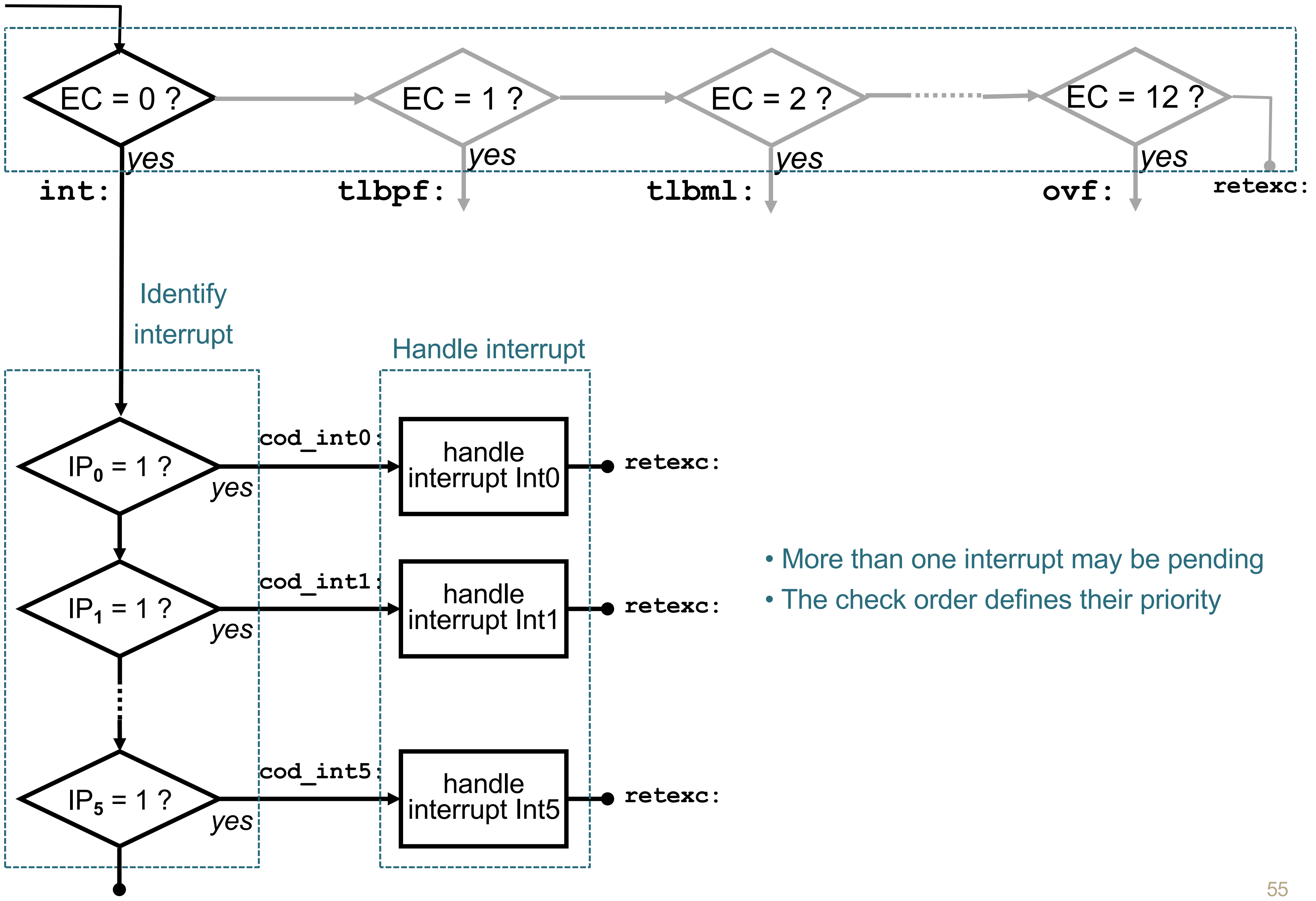
Identify cause of exception


EC = 0 ?
EC = 1 ?
EC = 2 ?
EC = 12 ?
yes
yes
yes
yes
int:
tlbpf:
tlbml:
ovf:
retexc:
Identify and handle interrupt
Handle write protection error
Handle invalid page read access
Handle arithmetic overflow
retexc:
retexc:
retexc:
retexc:



```
mfc0 $k0, $13              # Read Cause register
andi $t0, $k0, 0x003c      # Mask out cause code (*4)
beq $t0, $zero, int        # Compare with 0 → jump to int:
li $t1, 4                  # Code 1*4
beq $t0, $t1, tlbfp        #   compare and jump if equal
li $t1, 8                  # Code 2*4
beq $t0, $t1, tlbml        #   compare and jump if equal
...
li $t1, 0x30               # Code 12*4
beq $t0, $t1, ovf          #   compare and jump if equal
b retexc
```

Identify an interrupt



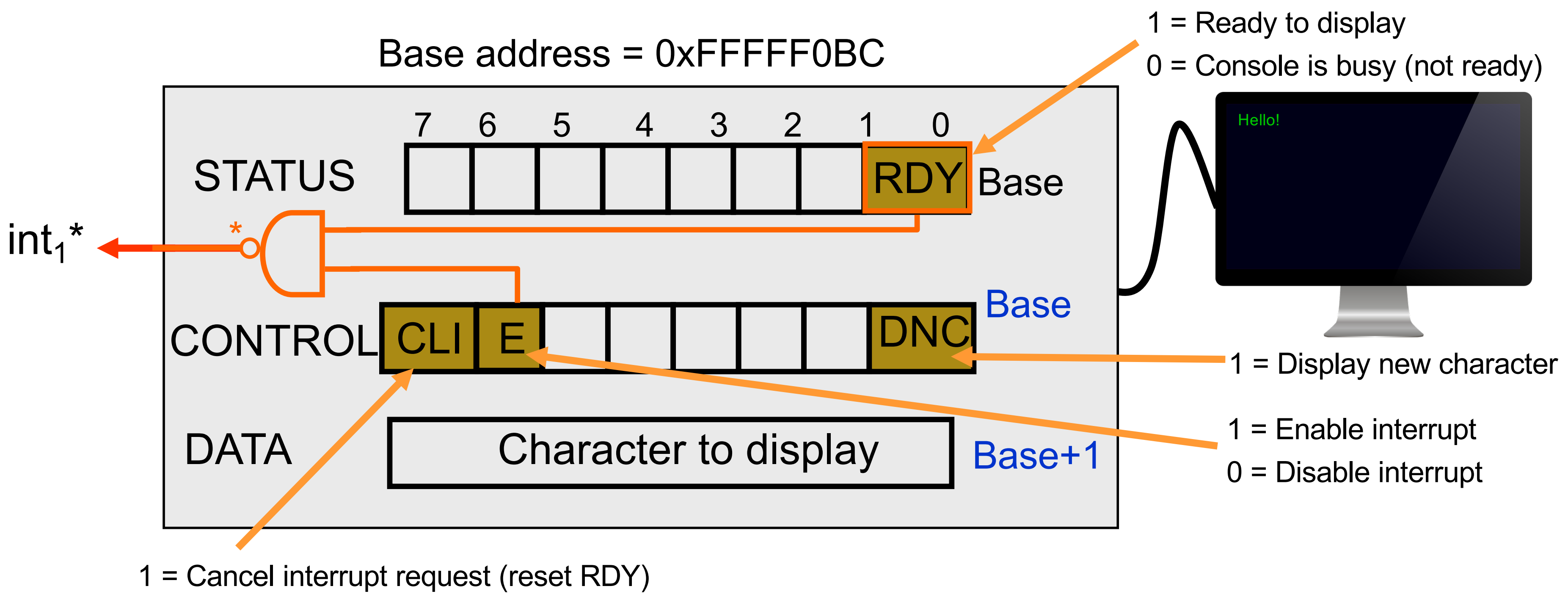
- More than one interrupt may be pending
- The check order defines their priority

Interrupt identification code

```
int:
        andi $t0, $k0, 0x0400      # check IP0
        bne  $t0, $zero, cod_int0 #
        andi $t0, $k0, 0x0800      # check IP1
        bne  $t0, $zero, cod_int1 #
        ...
        andi $t0, $k0, 0x8000      # check IP5
        bne  $t0, $zero, cod_int5 #
        b retexc                   # Spurious interrupt

cod_int0:  ### code for handling interrupt int0*
        ...
        b retexc ### end of handler for int0*

cod_int1:  ### code for handling interrupt int1*
        ...
        b retexc ### end of handler for int1*

cod_int5:    ### code for handling interrupt int5*
        ...
        b retexc ### end of handler for int5*
```

IP_5	IP_4	IP_3	IP_2	IP_1	IP_0	SW_1	SW_0			EC_3	EC_2	EC_1	EC_0		
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

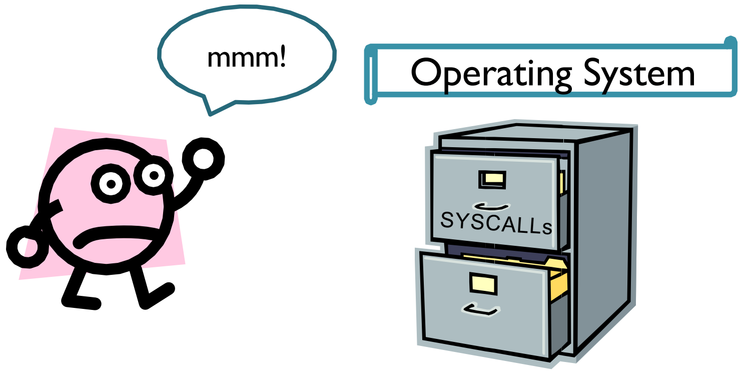
Interrupt handling – Example: a console



```
cod_int1:
   ## Handler for console interrupts

   li   $t0,0xFFFFF0BC  # Printer's base address
   lb   $t1,next_char   # Read char to print from a memory variable
   sb   $t1,1($t0)      # Write the char's code to the DATA reg.
   li   $t1,0xC1        # Set bits 0, 6 and 7 to 1 (CLI,E,DNC)
   sb   $t1,0($t0)      # Write 11000001 to the CONTROL reg.
   b retexc             # End of handler
```

7– MIPS: I/O by means of OS system calls

- The link between user and system code
- The 'syscall' instruction
- Implementing OS functions

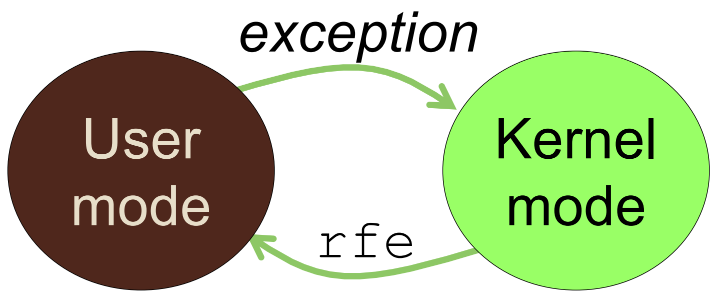


Exceptions and the OS

- The exception mechanism supports the needs of OSs:
  - ✓ The hardware handles kernel and user modes in a safe manner
  - ✓ Exceptions are an efficient event-handling mechanism
  - ✓ Exceptions facilitate process handling (active, suspended, etc.)
  - ✓ Exceptions simplify context changes between concurrent processes

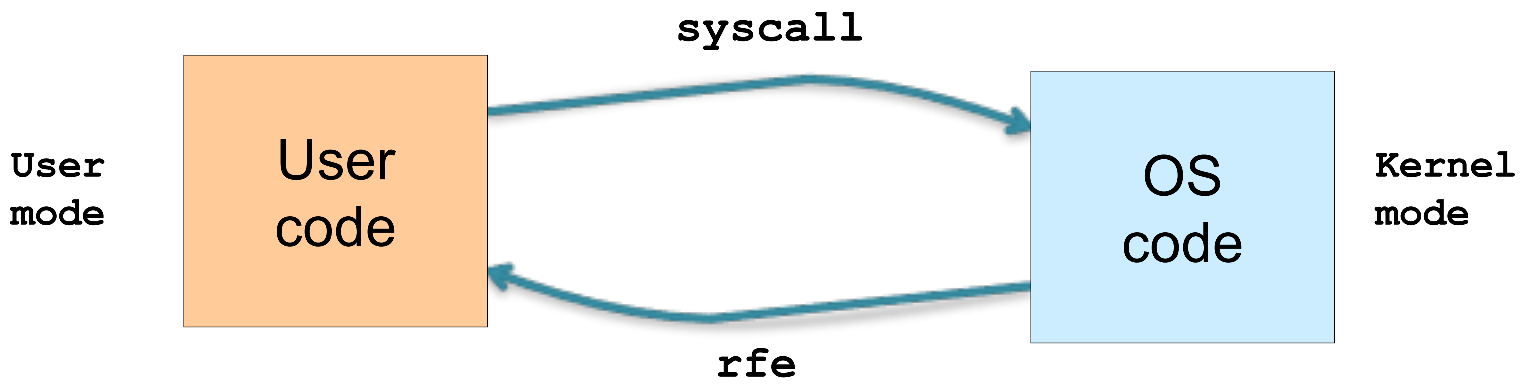
Exceptions and the OS

- Organization of an OS
  - The OS is composed by a number of code fragments specialized in handling all computer resources (processor, memory, and the I/O system).
  - User programs run in user mode.
  - A set of predefined events cause the execution of the appropriate OS handlers in kernel mode.
- The OS handles the I/O
  - Interrupts cause the change to kernel mode and the execution of the appropriate OS handler.
  - I/O interfaces are mapped to a privileged memory area, only accessible in kernel mode.



System calls: link between user programs and OS

- System calls are the link between user and OS code
  - User programs invoke OS functions by means of syscall instructions
  - A syscall is a particular type of exception
- Independence between user programs and the OS
  - User programs can run under different OS versions and hardware configurations.



Calling OS functions

- In order to call an OS function, programs must first prepare the parameters (if any) and then issue syscall
- Upon return from the system call, the program finds the result (if any) in the designated register
- Parameters and results
  - Elementary types (within word size) – passed/returned in registers
  - Composite types (larger) – dereferenced by register-size pointers
- For each OS, a convention exists specifying the available functions, their parameters and results

Implemented system functions in PCSpim

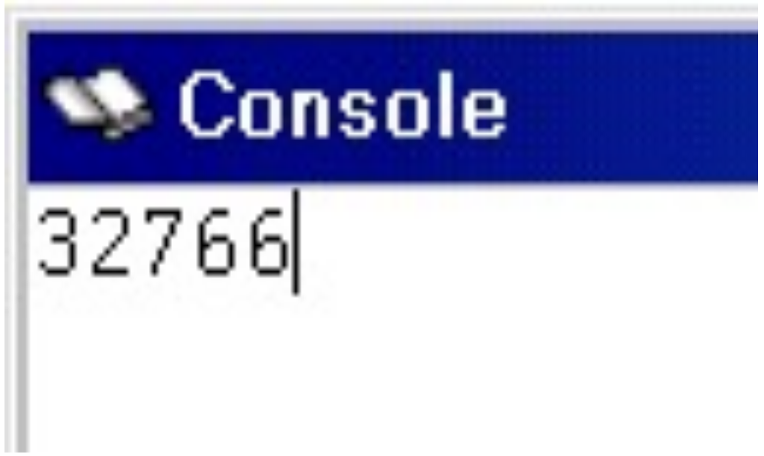
Function	Index ($v0)	Arguments	Result
`print_int`	1	`$a0` = integer	
`print_float`	2	`$f12` = floating point	
`print_double`	3	`$f12` = double precision	
`print_string`	4	`$a0` = pointer to string	
`read_int`	5		Integer (in `$v0`)
`read_float`	6		Floating point (in `$f0`)
`read_double`	7		Double precision (in `$f0`)
`read_string`	8	`$a0` = pointer to string `$a1` = string length	
`print_char`	11	`$a0` = character	
`read_char`	12		Character (in `$v0`)

Syscall example in PCSPim

User program

```
li $v0,1              # Index 1: print_int (to console)
li $a0,0x7ffe         # Value to print
syscall               # Calling the function
```

Effect

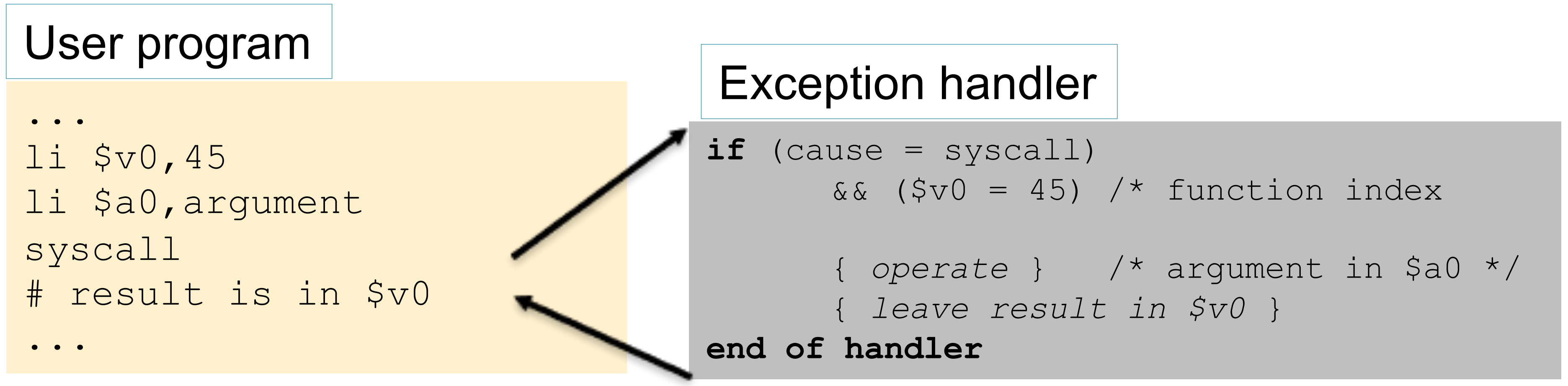


```
32766
```

$7FFE_{16} = 32766_{10}$

Syscall implementation: MIPS R2000 case

- The SYSCALL instruction
  - ✓ Triggers an exception with code number 8 in CR (Cause Register).
  - ✓ Conventionally, $v0 contains the index of the requested function
    - Example: function 45, requests one argument in $a0, returns result in $v0.

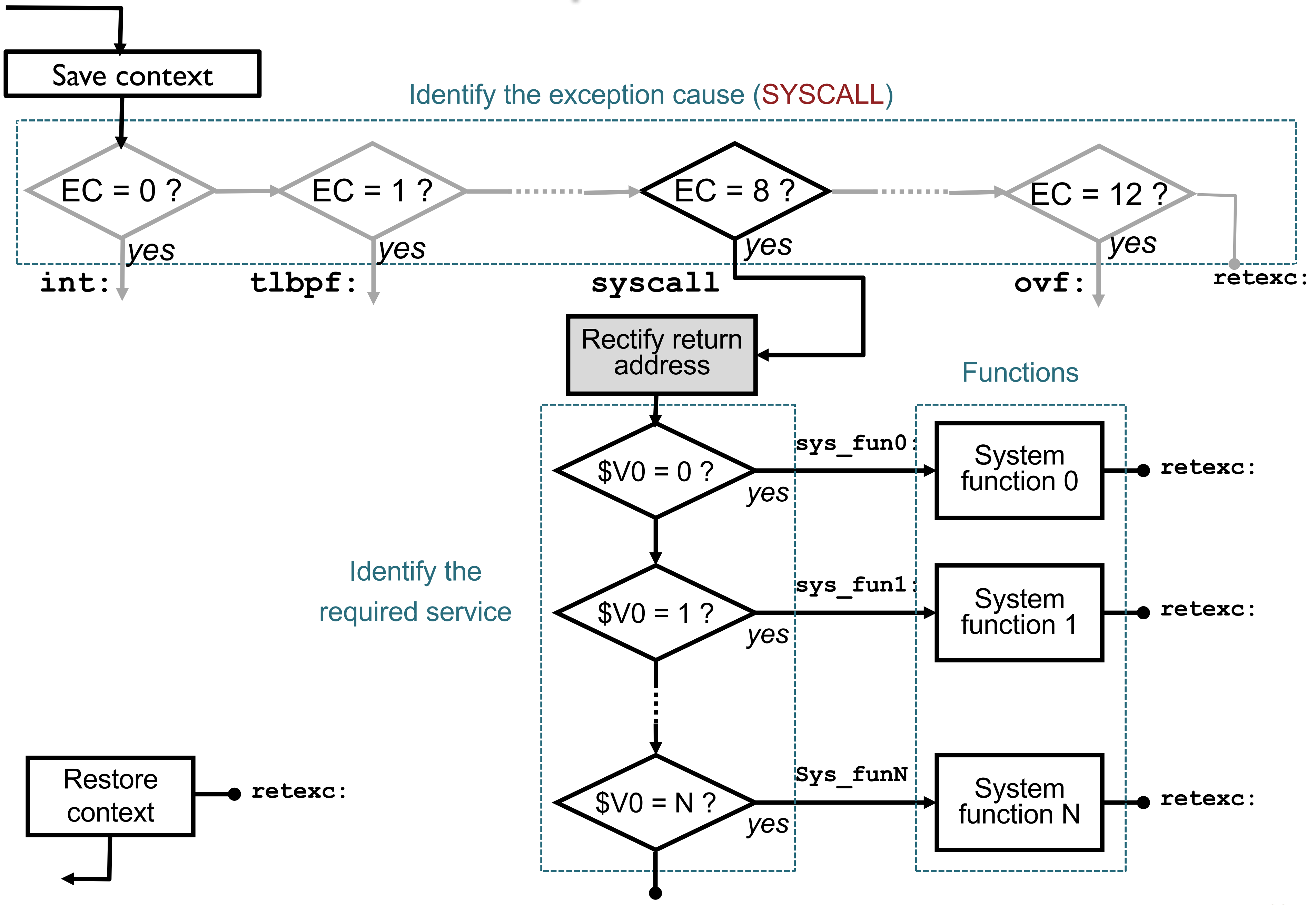

User program

```
...
li $v0,45
li $a0,argument
syscall
# result is in $v0
...
```

Exception handler

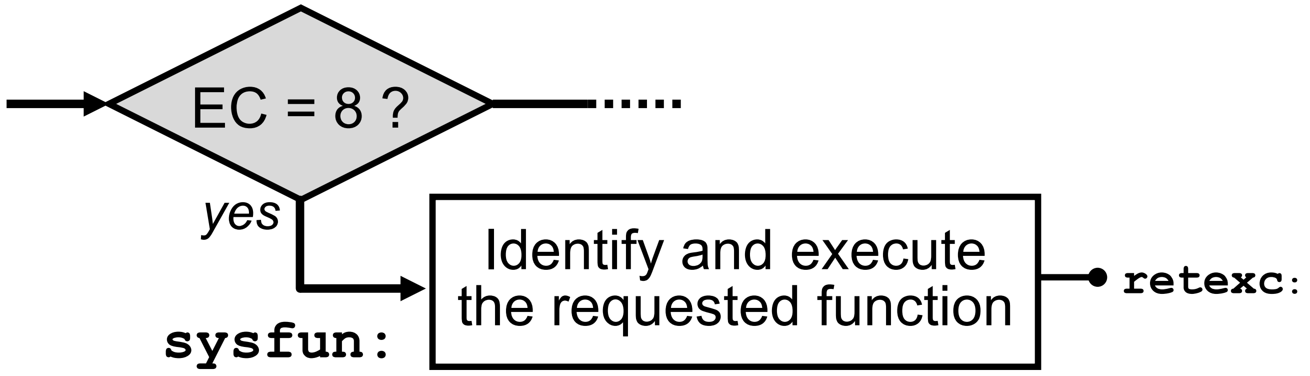
```
if (cause = syscall)
      && ($v0 = 45) /* function index

      { operate }   /* argument in $a0 */
      { leave result in $v0 }
end of handler
```

Structure of the exception handler



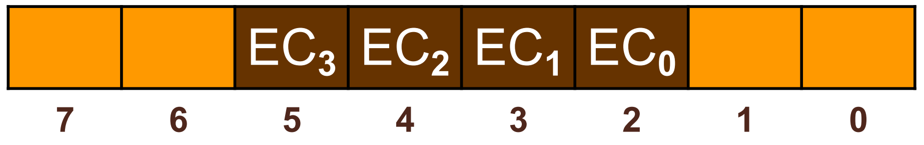
Identifying syscall

- The syscall exception has a corresponding exception code 8



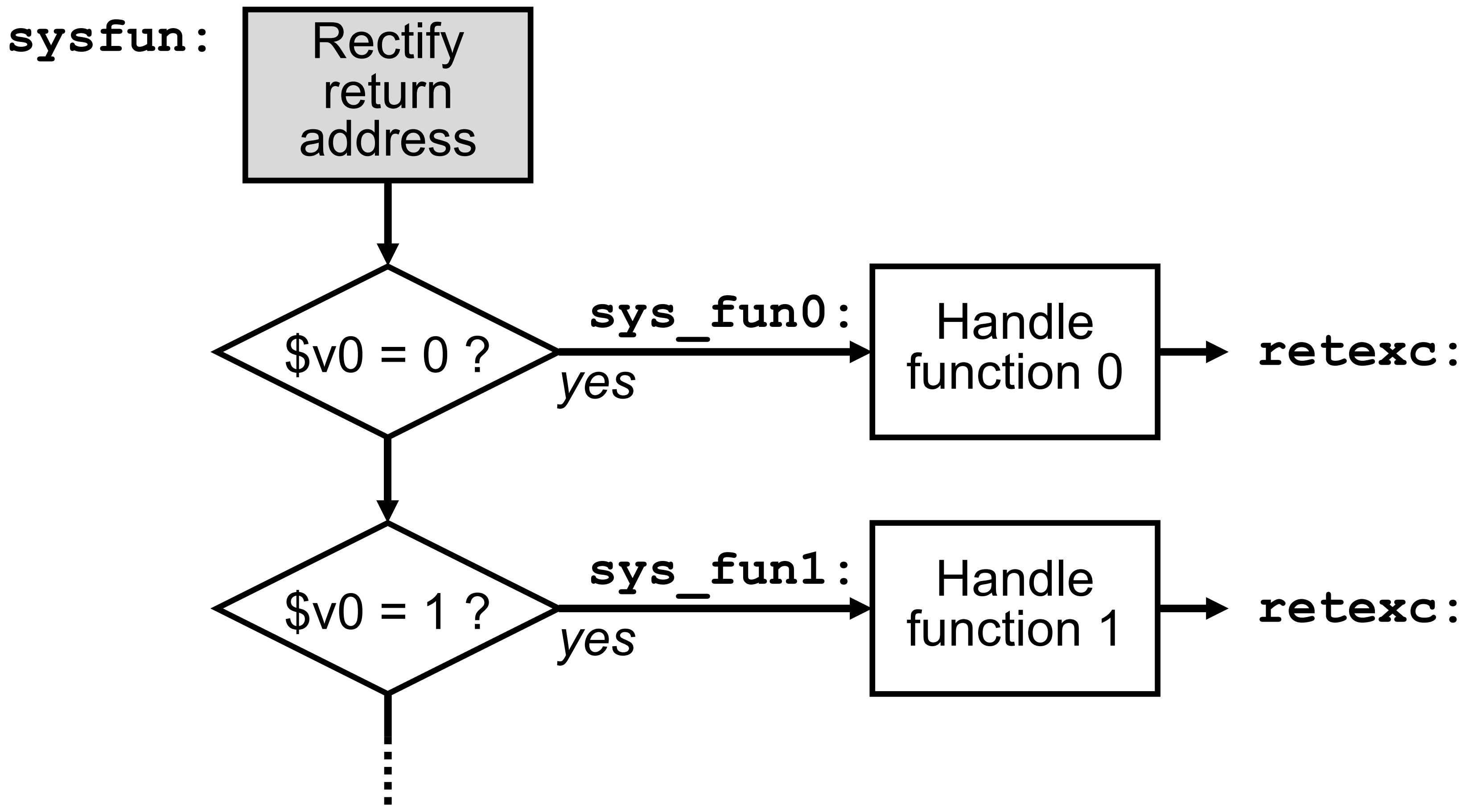
```
mfc0 $k0, $13                # Read Cause register
andi $t0, $k0, 0x003c        # Isolate code*4
beq $t0, $zero, int          # Code 0 (int)
li $t1, 4                    # Code 1*4 (tlbfp)
beq $t0, $t1, tlbfp
...
li $t1, 0x20                 # Code 8*4 (syscall)
beq $t0, $t1, sysfun
...
```

Exception code in $Cause register



Identify the requested function

- The return address must be rectified (next slide)
- $v0 holds the index of service requested



Rectifying the return address: why and how

- Upon an exception occurrence, EPC points to the non-completed instruction
- Cases:
  - ✓ Interrupt: the handler must return to EPC (next instruction)
  - ✓ Virtual memory exceptions: EPC points to the offending instruction. We normally want to retry the same instruction after the OS has loaded the missing page
  - ✓ Syscall: EPC points to the syscall instruction, but we want to return to the next one (EPC+4).

Now in assembly…

```
Sysfun: ### syscall handling: Cause has been identified as EC=8
           # rectify return address
           lw $t0,retaddr
           addi $t0,$t0,4                # retaddr = retaddr + 4
           sw $t0, retaddr

           # select function to execute based on index ($v0)
           beq $v0, $0, sys_fun0
           li $t0, 1
           beq $v0, $t0, sys_fun1
           li $t0, 2
           beq $v0, $t0, sys_fun2
           ...
           ...
Sys_fun0: ### Handler for function with index 0
           ...
           b retexc

Sys_fun1: ### Handler for function with index 1
           ...
           b retexc
```

Implementing system functions

Case 1: Returning system variables

Case 2: Accessing a direct I/O peripheral

Case 3: Access interrupting device (no suspension)

Waiting for I/O – Handling process states

Case 4: Synchronisation with suspension

Case 1. Returning system variables

- Specification
  - ✓ P_Model: Returns a code that identifies the processor model
  - ✓ Sys_Ver: Returns the OS version
- Comments
  - ✓ For a given system, these are constants in the kernel memory area: the handler will just copy their values back to the caller
  - ✓ No need to access any peripherals

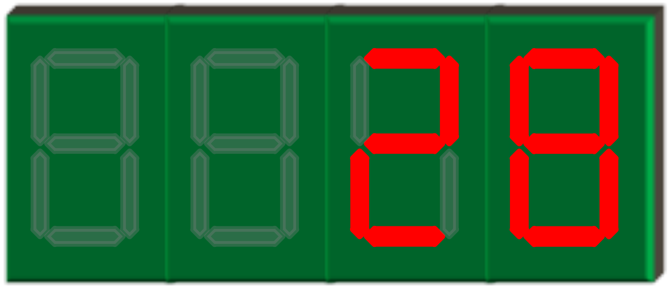
Function	Index	Arguments	Result
`P_Model`	`$v0` = 9991	–	`$v0` = processor model code
`Sys_Ver`	`$v0` = 9992	–	`$v0` = OS version code

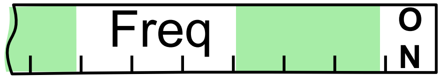
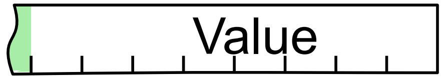
Case 1. Handler implementation

```
Sysfun:   lw $t0, retaddr
          addi $t0,$t0,4   # retaddr = retaddr + 4
          sw $t0, retaddr

# Branch depending on index in $v0
          ...
          li $t0, 9991             # $v0 = 9991?
          beq $t0, $v0, P_model    # Branch to P_Model
          li $t0, 9992             # $v0 = 9992?
          beq $t0, $v0, Sys_ver    # Branch to Sys_Ver
          ...

P_Model:  lw $v0, proc_model_code
          b retexc

Sys_Ver:  lw $v0, OS_version_code
          b retexc
```

Case 2. Accessing a direct I/O peripheral

- ✓ Access to peripherals can only be granted by syscalls, since I/O interfaces reside in kernel memory
- ✓ Example: access to the 4-digit display with base address 0xFFFF0020.



Name	Address	Access	Structure
Command	BA	Write	 *Activates display and blinking:* • ON (bit 0): 0 off; 1 displays VALUE • Freq (bits 6..4): blinking frequency in Hz • Freq = 0: continuous display
Data	BA+4	Write	 *VALUE to display:* Signed 2'sC 8-bit integer (-128 .. +127) Value is displayed when ON is set to 1

Case 2. Accessing a direct I/O peripheral

- Display functions:
  - ✓ Display_Show: Shows a value in the display
  - ✓ Display_Blink: Defines a blinking frequency
  - ✓ Display_Clear: Switches display off

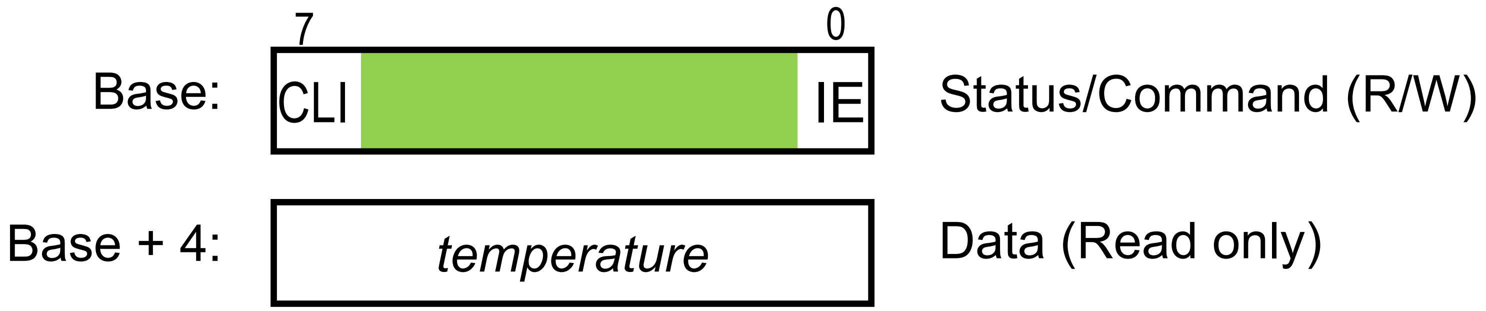
Function	Index	Arguments	Result
`Display_Show`	`$v0 = 9980`	`$a0 = value`	`'value' is displayed`
`Display_Blink`	`$v0 = 9981`	`$a0 = Frequency (0 = continuous)`	`Defines the blinking frequency (effective at next Display_Show)`
`Display_Clear`	`$v0 = 9982`	`-----`	`Turns display off`

Case 2. Implementation of the display functions

```
         .kdata
Blink:   .byte  0  # Blinking frequency, aligned to bit 4
         ...

Display_Show:   # Display value held in $a0, at frequency given by Blink
          la $t0,0xFFFF0020   # Base address of display
          sb $a0, 4($t0)      # Write value to Data register
          lb $t1, Blink       # Use freq. given by Blink
          ori $t1, $t1, 1     # Set the ON bit (turn on)
          sb $t1, 0($t0)      # Write to Command register
          b retexc

Display_Blink:   # Set blink at frequency given in $a0
         sll $t0, $a0, 4      # Shift to match position of Freq field
         sb $t0, Blink        # Store in Blink for future use
         b retexc

Display_Clear:   # Switch display off
          la $t0, 0xFFFF0020  # Base address of display
          sb $zero, 0($t0)    # Reset the ON bit (turn off)
          b retexc
```

Case 3. Access interrupting device – **no wait**

- Example device: a temperature sensor (thermometer)
  - ✓ Upon a temperature change, the thermometer causes an interrupt
  - ✓ The interrupt handler updates a private kernel variable
  - ✓ Thermometer interface adapter:
    - Base address: **`0xFFFFB900`**; interrupts via int4*



Function	Index	Arguments	Result
Get_Temp	$v0 = 9975	—	$v0 = temperature

Case 3. Implementation

```
                .kdata
                ...
temperature:    .byte 0                 # OS private variable
                ...

                .ktext
                ...
```

```
                # Handler section for int4*
int4:           la $t0,0xFFFFB900       # Base address
                lb $t1,4($t0)           # Get temp from thermometer
                sb $t1,temperature      # Update temperature
                li $t1,0x81             # Value for CLI=1, IE=1
                sb $t1,0($t0)           # Cancel interrupt
                b retexc
                ...
```

```
                # Handler section for Get_Temp
Get_Temp:       lb $v0,temperature      # Read last updated temp.
                b retexc
                ...
```

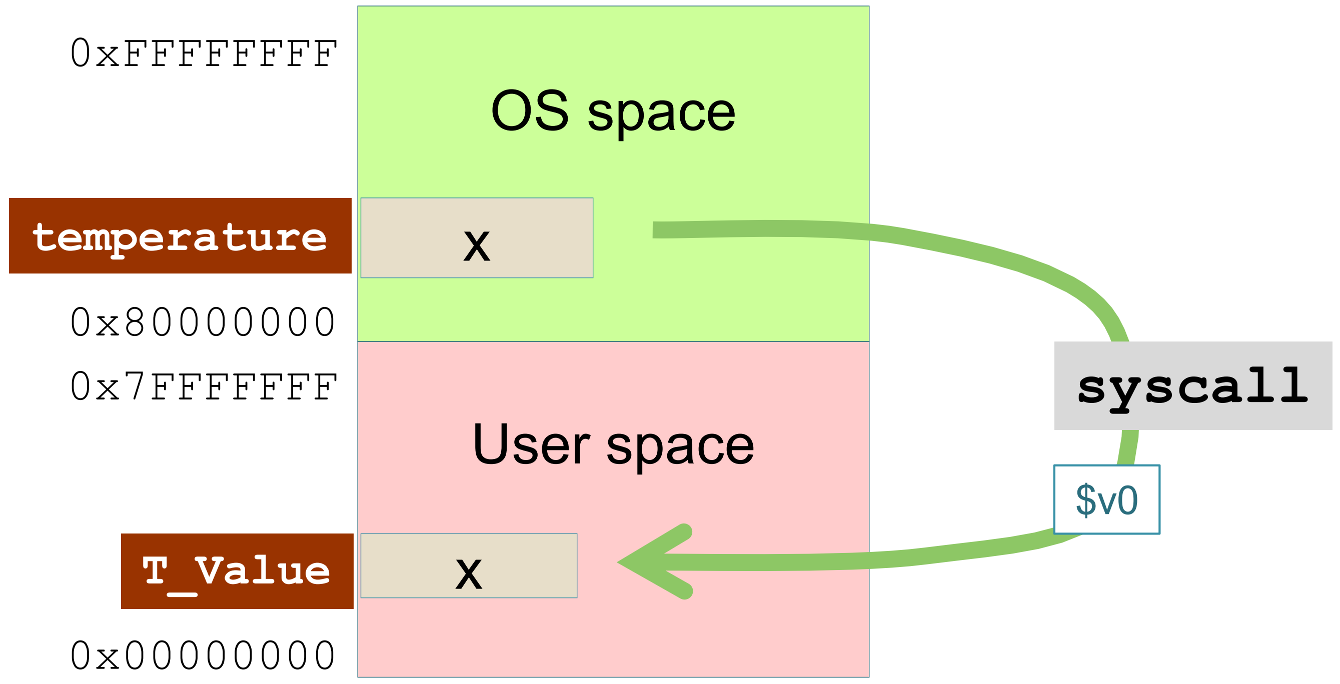
Case 3. Use from a user program perspective

- Obtain temperature using Get_Temp, save it to own variable T_Value and print it using print_int

```
         .data
T_Value: .byte 0           # User variable for temperature

         .text
         # Obtain temperature
         li $v0, 9975      # Index for Get_Temp
         syscall           # Temperature returned in $v0
         sb $v0, T_Value   # Save temperature to T_Value

         # Print temperature to console
         move $a0, $v0     # Pass argument in $a0
         li $v0, 1         # Index 1 for print_int
         syscall           # Print temperature to console
```

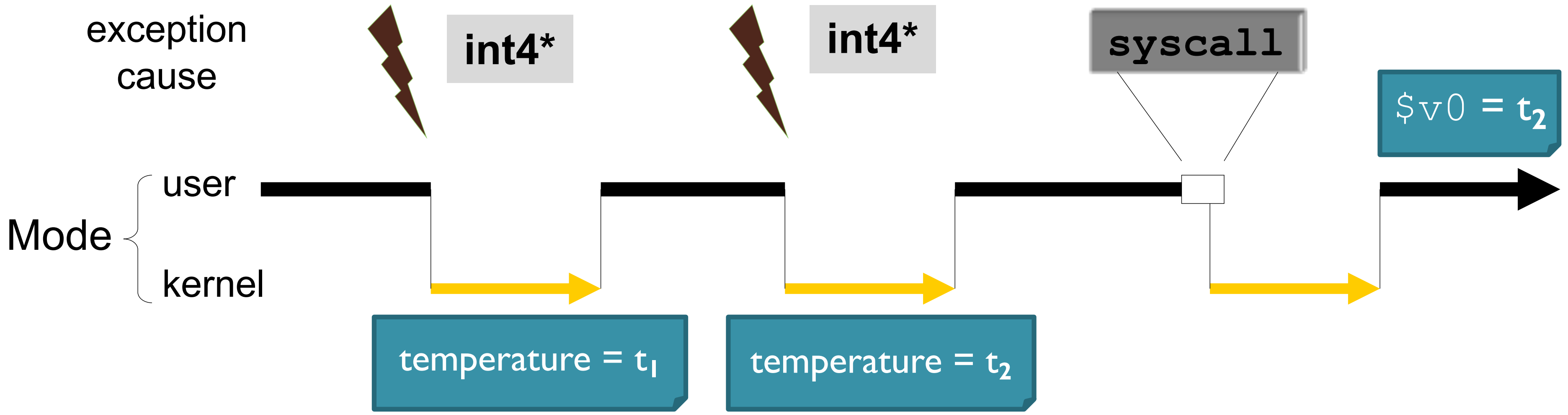
Data flow between kernel and user spaces

- ✓ The value of the temperature variable is copied from the kernel to the user space by means of Get_Temp – similar to P_Model and Sys_Ver
- ✓ The variable is kept updated by the int4 handler.
  - The initialization code must enable int4 (and give an initial value to the *temperature* variable)



Case 3. Example of evolution of variables

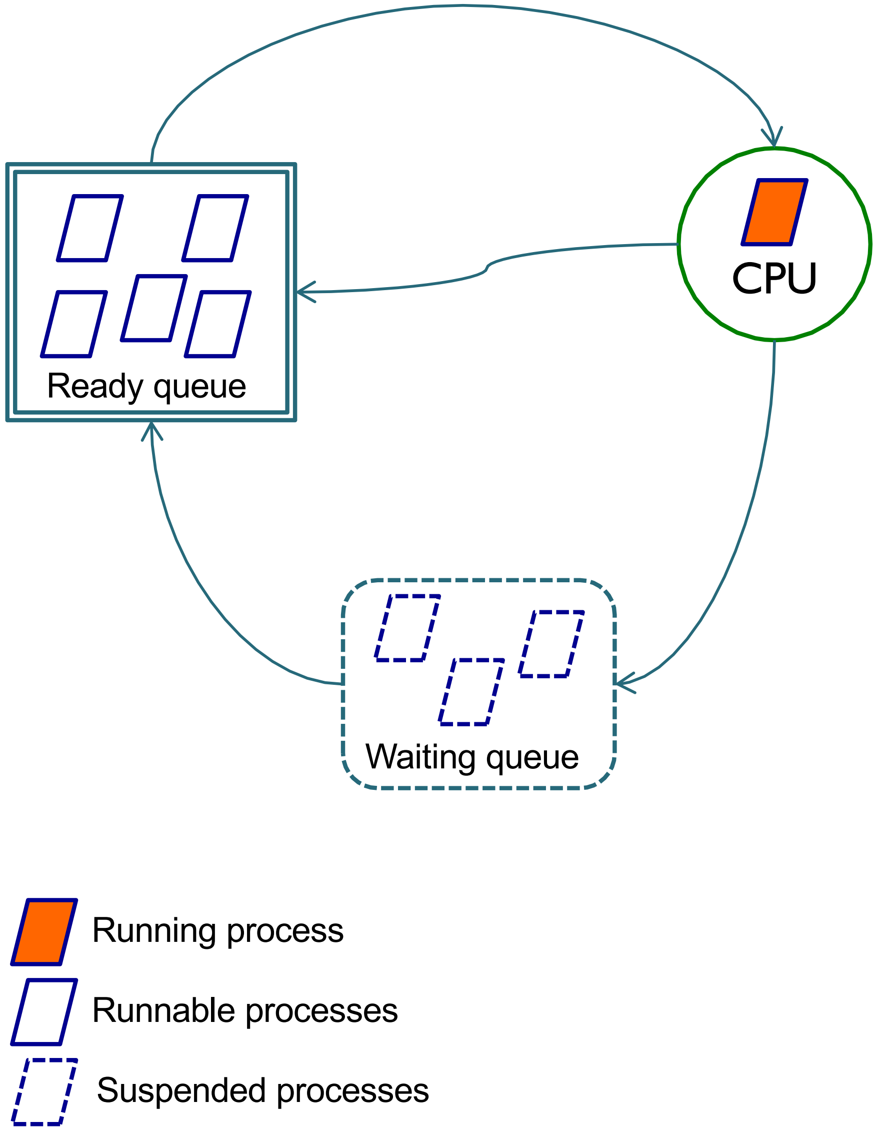
- The thermometer causes interrupts that keep the kernel's temperature variable updated
- When calling Get_Temp the user program obtains the newest stored temperature value
  - ✓ Example chronogram: the thermometer causes two interrupts to communicate values t_1 and t_2 before the program calls Get_Temp
    - Note t_1 is never read by the user



Waiting for I/O – Blocking system functions

- Some system functions may make the calling process wait for a peripheral to be ready (Case 4)
  - ✓ The calling process cannot make any progress until function completes
  - ✓ E.g. wait for a temperature change, a keyboard input, an incoming message…
- Using exceptions, the OS can:
  - ✓ Stop a running process and set it to a *suspended* state
  - ✓ Resume execution of a suspended process when the I/O completes, i.e. set it to a *runnable* or *ready* state
  - ✓ Among all ready processes select the *running* process (*process switching*)
- The OS scheduler handles process states

A Simple Model



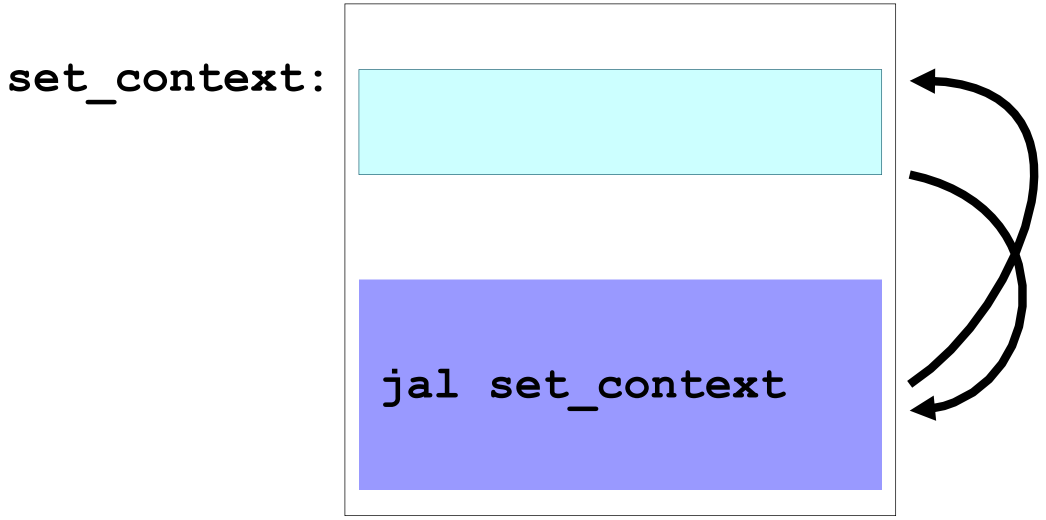
- A single process (per CPU) is running, and it continues to run until an exception occurs
- Depending on the exception, the running process may move to either:
  - The Ready queue, if it continues to be runnable (e.g., end of quantum under round-robin scheduling)
  - The Waiting queue, if it needs to be suspended to wait for a peripheral (a temperature change, a disk transfer…)
  - Ultimately, the process could also be terminated and destroyed due to normal termination or a fatal error (not shown here)
- Upon an interrupt, if a suspended process becomes runnable again, then it is moved back to the Ready queue
- Finally, the dispatcher moves runnable processes from the ready queue to the CPU

The context switch

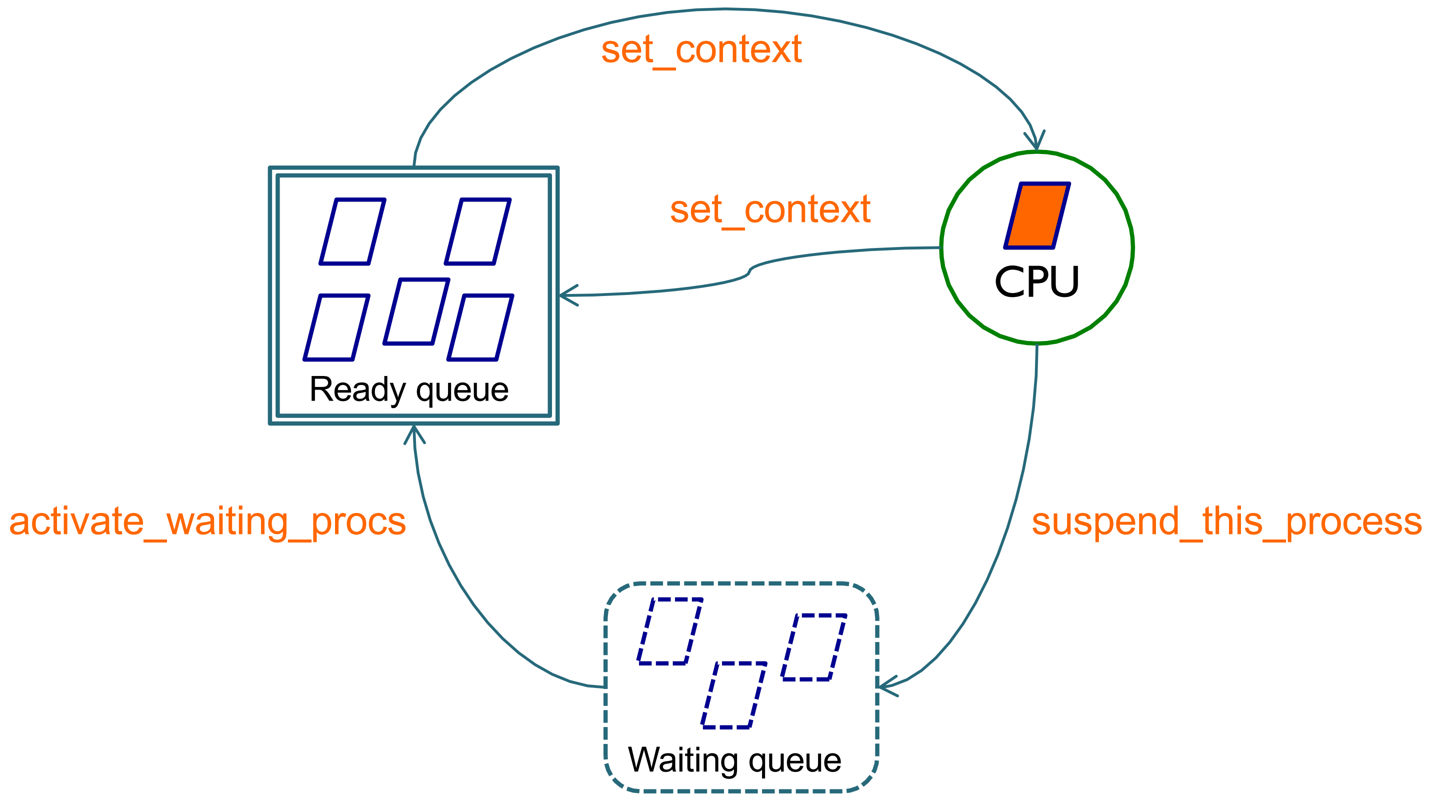
- Switching the context means changing the currently *running* process with another *runnable* process
- The context of the running process must be preserved, so that the process can be later resumed. It includes at least:
  - ✓ the set of general-purpose registers, including the minimal context (**`$at`**, **`$t0`** and **`$t1`**)
  - ✓ the current Program Counter of the running process (based on EPC)
  - ✓ This information is held in the PCB (Process Control Block)
- A context switch from a running process P to a runnable process Q implies:
  - transfer the context of P to P’s PCB
  - transfer Q’s PCB to the corresponding registers

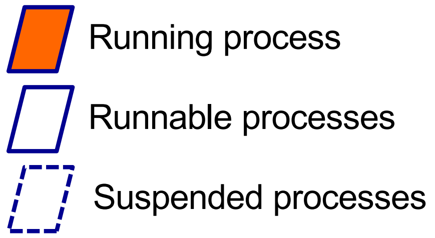
Process scheduling

- Scheduling is the process by which the OS selects the running process
- Process scheduling relies on three primitives:
  - ✓ set_context – for context switch
  - ✓ suspend_this_process – for suspending the running process
  - ✓ activate_waiting_procs – for resuming suspended processes
- We assume these primitives are available as parameterless functions



Simple Model Revisited

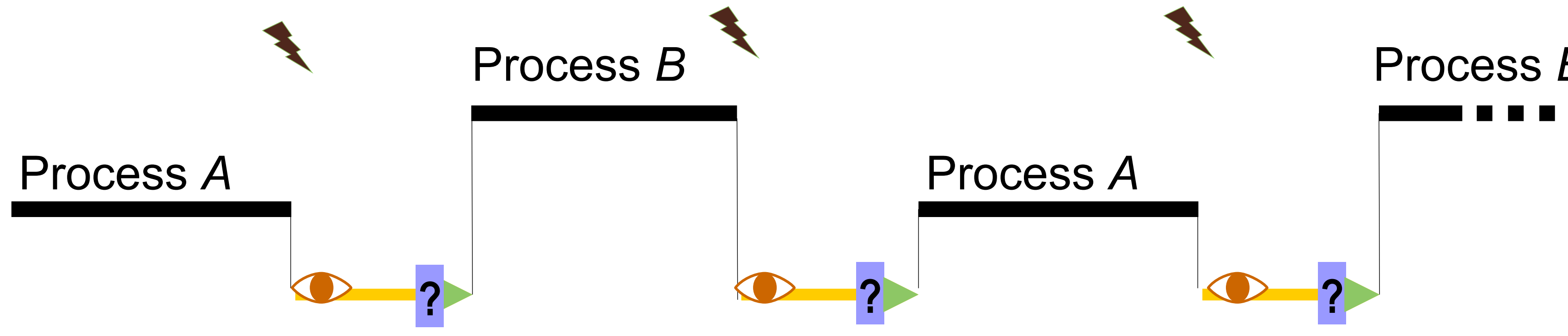




Context switch

- set_context
  - ✓ Selects the next running process after execution of exception handler
    - No need for a context switch when the selected process is the currently running process
    - When the selected running process changes, set_context makes the context switch (registers, return address, etc.) and updates the ready queue, if needed
  - ✓ A side effect of set_context is the assignment of **`retaddr`**
    - It is set to the PC of the selected running process

When to change the context?



= clock interrupt

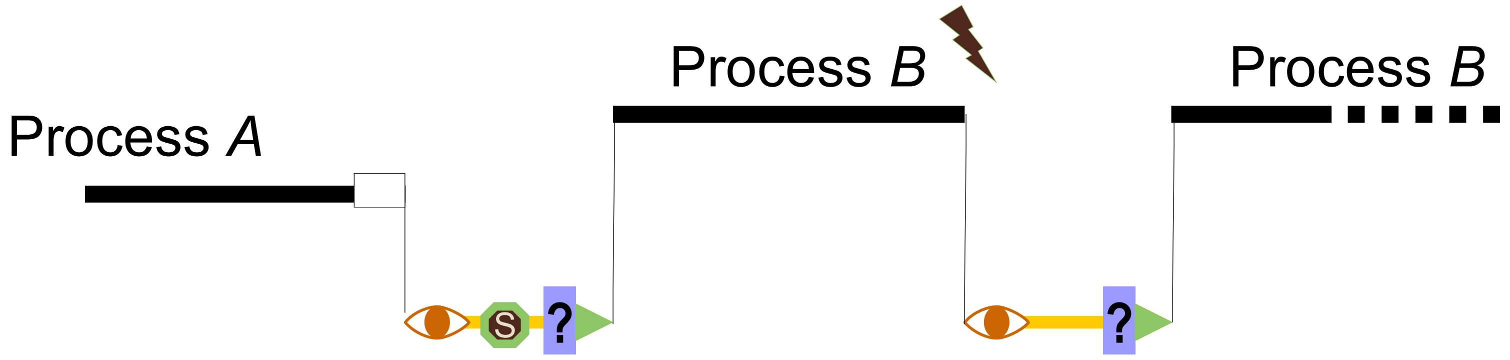
? = `jal` *set_context*

- In the return section of the exception handler

```
retexc:
          jal set_context     # Possible context switch

          lw $k0, retaddr     # return address in $k0
          lw $at, 0($k1)      # }
          lw $t0, 4($k1)      # } restore minimal context
          lw $t1, 8($k1)      # }
          rfe                 # back to user mode
          jr $k0              # resume selected process
```

Suspending a process

- suspend_this_process
  - ✓ Changes state of the running process to suspended and saves its PCB
  - ✓ Example:
    - function syscall F: makes process A wait until peripheral P is ready
    - Handling of F suspends process A
    - A will not be selected by set_context for execution until it becomes runnable again (P becomes ready)



☐ = **`syscall F`**

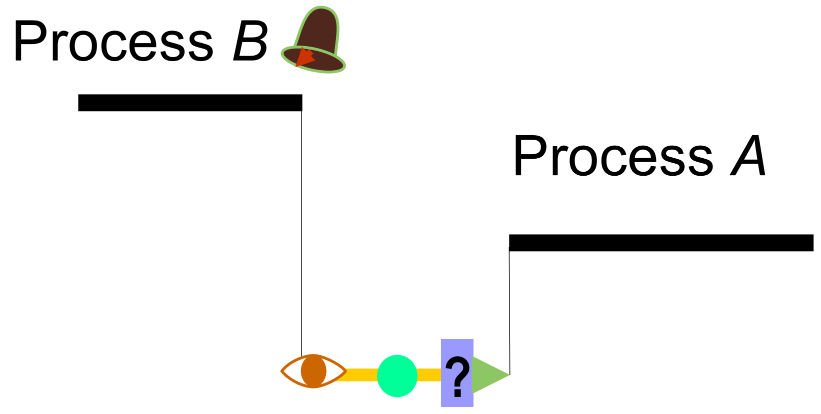
🅢 = **`jal`** *suspend_this_process*

⚡ = clock interrupt

? = **`jal`** *set_context*

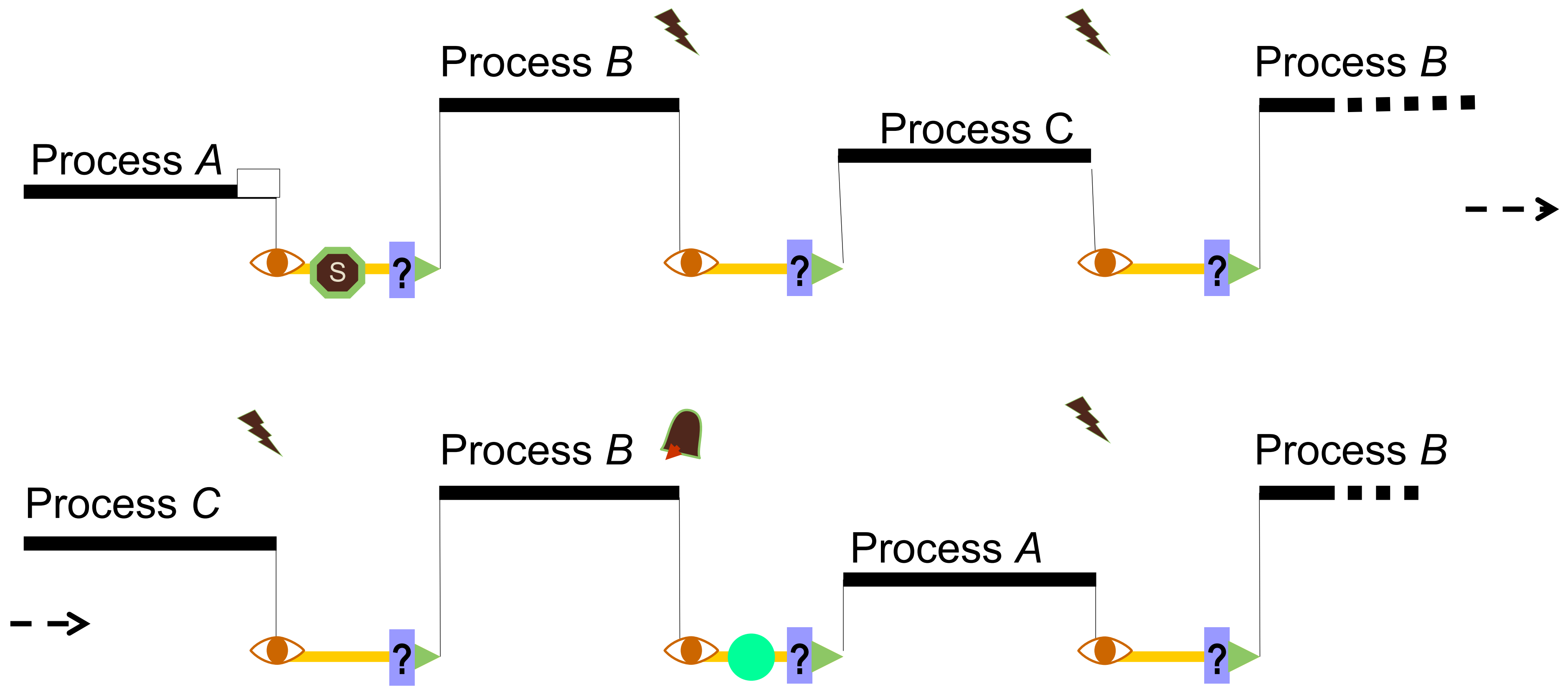
Activating a process

- activate_waiting_procs
  - ✓ Changes the state of suspended processes to runnable and inserts their PCBs in the ready queue
  - ✓ Continuing from the previous example…
    - P's interrupt handler reactivates process A
    - Process A becomes runnable and eligible by the scheduler



= *P's interrupt*   ● = `jal` *activate_waiting_procs*   ? = `jal` *set_context*

Example



☐ = **`syscall`**

● = **`jal`** *suspend_this_process*

= *P's* interrupt

● = **`jal`** *activate_waiting_procs*

? = **`jal`** *set_context*

Case 4. Input from peripheral – **with wait**

- Peripheral:
  - Thermometer that causes int4* upon temperature changes
- Function ‘get_temp_wait’ :
  - Allows a program to wait until thermometer provides a new temperature. Returns temperature in **`$v0`**
  - Must execute ‘suspend_this_process’ to suspend the calling program and insert it in the waiting queue for the thermometer
- Interrupt:
  - A subprogram ‘update_temps’ modifies $v0 in the context of all processes that are waiting for the thermometer (0, 1 or more)
  - A call to activate_waiting_procs moves all potential waiters from the *Waiting* to the *Ready* queue

Case 4. Implementation of get_temp_wait

- Only needs to change the process state: running → suspended
  - ✓ The calling process is suspended waiting for a new temperature

```
                .kdata
temperature:    .byte 0

                .ktext
                ...
                # System function
get_temp_wait:
                jal suspend_this_process
                b retexc
                ...
```

- ✓ In 'retexc' the scheduler will set a new context, because the calling process is not runnable now

Case 4. Handling the interrupt

- The interrupt handler must update the temperature in the PCBs of all processes that are waiting for a new temperature
- When the handler completes, all waiting processes become eligible for execution

```
      .ktext

      # Handler for int4*
int4: li $t0,0xFFFFB9000          # Base address
      lb $t1,4($t0)               # Read temperature
      sb $t1,temperature          # Update temperature variable

      jal update_temps
      jal activate_waiting_procs

      li $t1,0x81                 # Value for CLI=1; IE=1
      sb $t1,0($t0)               # Clear interrupt
      b retexc
```

End of Unit 8