

Interrupts

Peter Rounce
P.Rounce@cs.ucl.ac.uk

INTERRUPTS

An interrupt is a signal to the CPU from hardware external to the CPU that indicates that some event has occurred,
e.g. data has arrived at an input interface.

CPUs respond to an interrupt by switching from

- executing the instructions of the current program
- to executing instructions to handle the event that caused the interrupt.

The interrupt mechanism provides an efficient way to detect the occurrence of an event.

It is important to understand the implementation of interrupts
and when they should or should not be used.

Interrupts are a **key element** in the operation of most computer systems.

Why interrupts?

Polling is inefficient when data rates are low: - too many CPU cycles wasted executing instructions of polling loop.

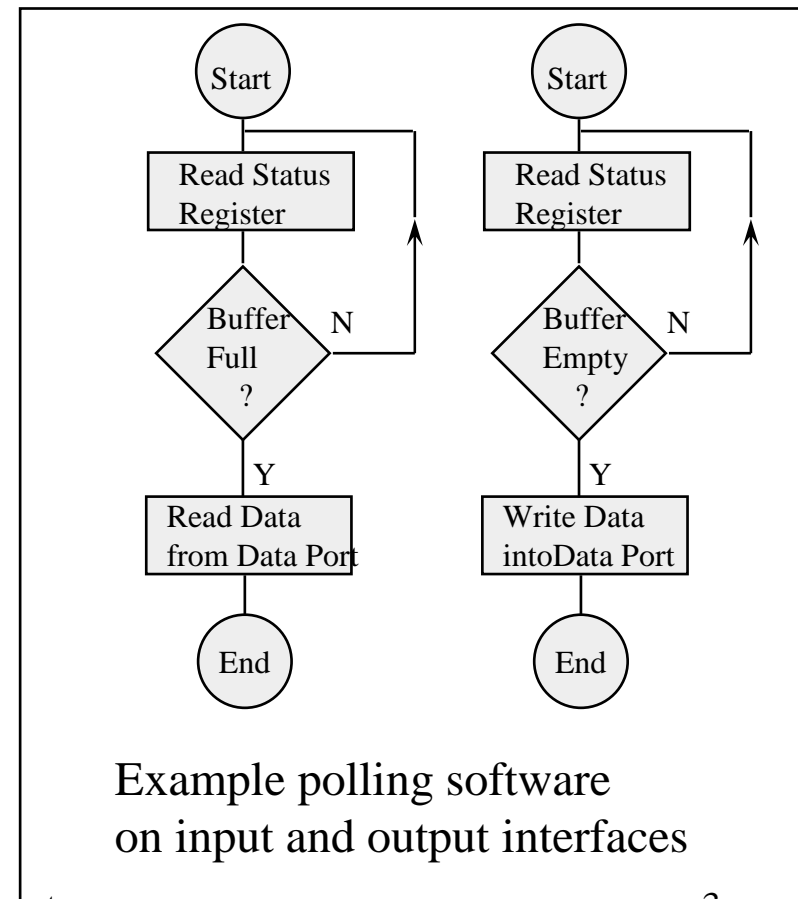
Polling is the wrong approach to handling events where the time of the occurrence of an event is uncertain or distant, i.e. 100s of clock cycles away.

The occurrence of the event should be the trigger for CPU action,

the CPU should not be checking for the event.

Examples of similar systems:

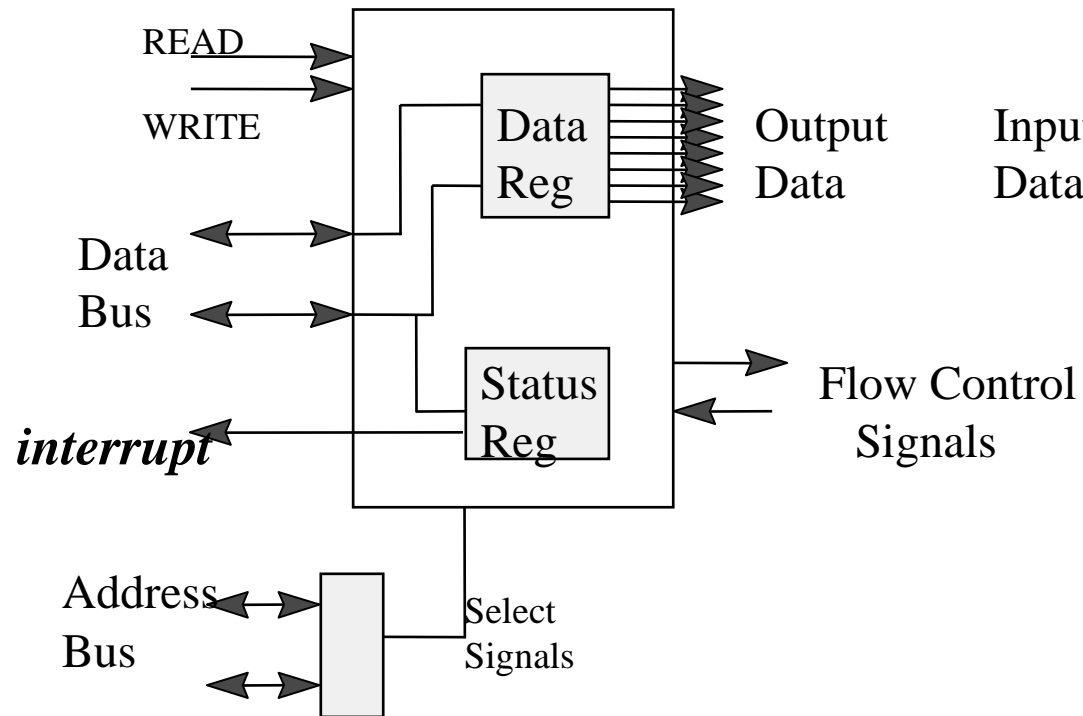
a knock on the door, the ringing of the telephone



Points to cover:-

- how is interrupt signal generated
- how are interrupts prioritised, if there are several at same time
- how does CPU identify event that generated interrupt
- how does CPU switch from to running software to deal with event
- how is it that program running at time of interrupt is not affected
- how does interrupt software exchange data with rest of system

8-Bit Parallel Output Interface

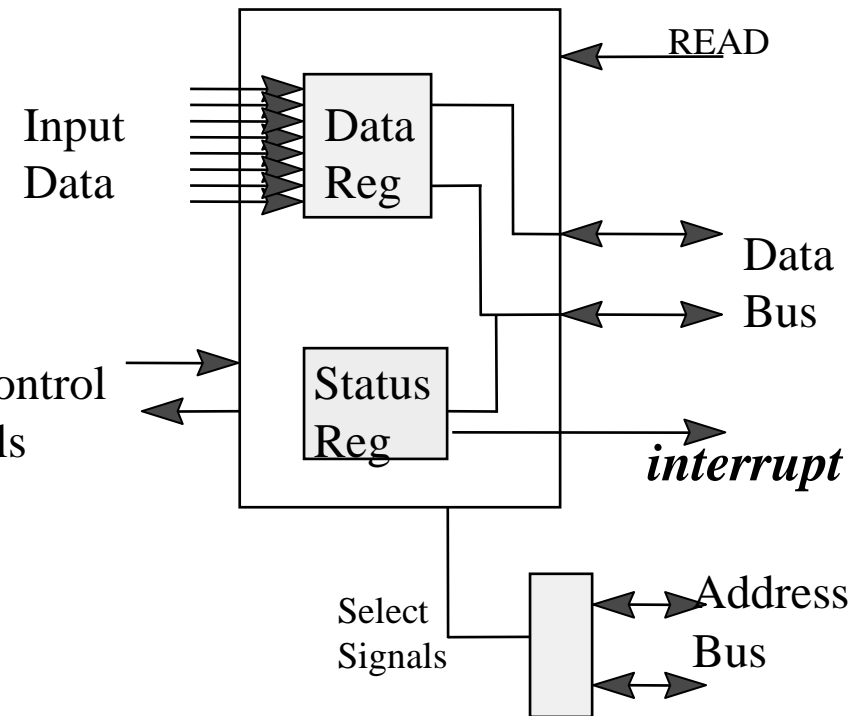


When Rx acknowledges receipt of data the TX interrupt request is activated

Writing new data into the Data Register or reading Status Register deactivates TX interrupt request

22/11/2011

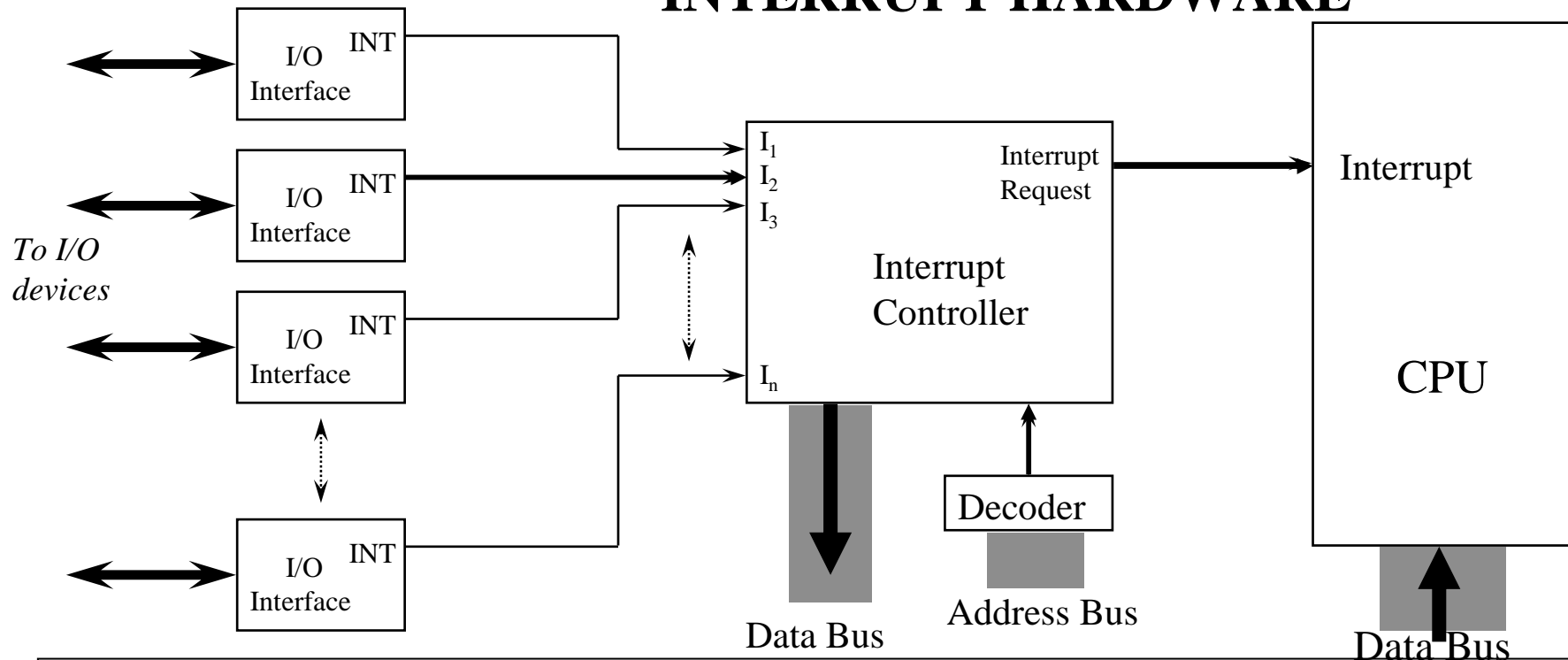
8-Bit Parallel Input Interface



Arrival of new data triggers activation of interrupt request

Reading data from Data Register deactivates interrupt request

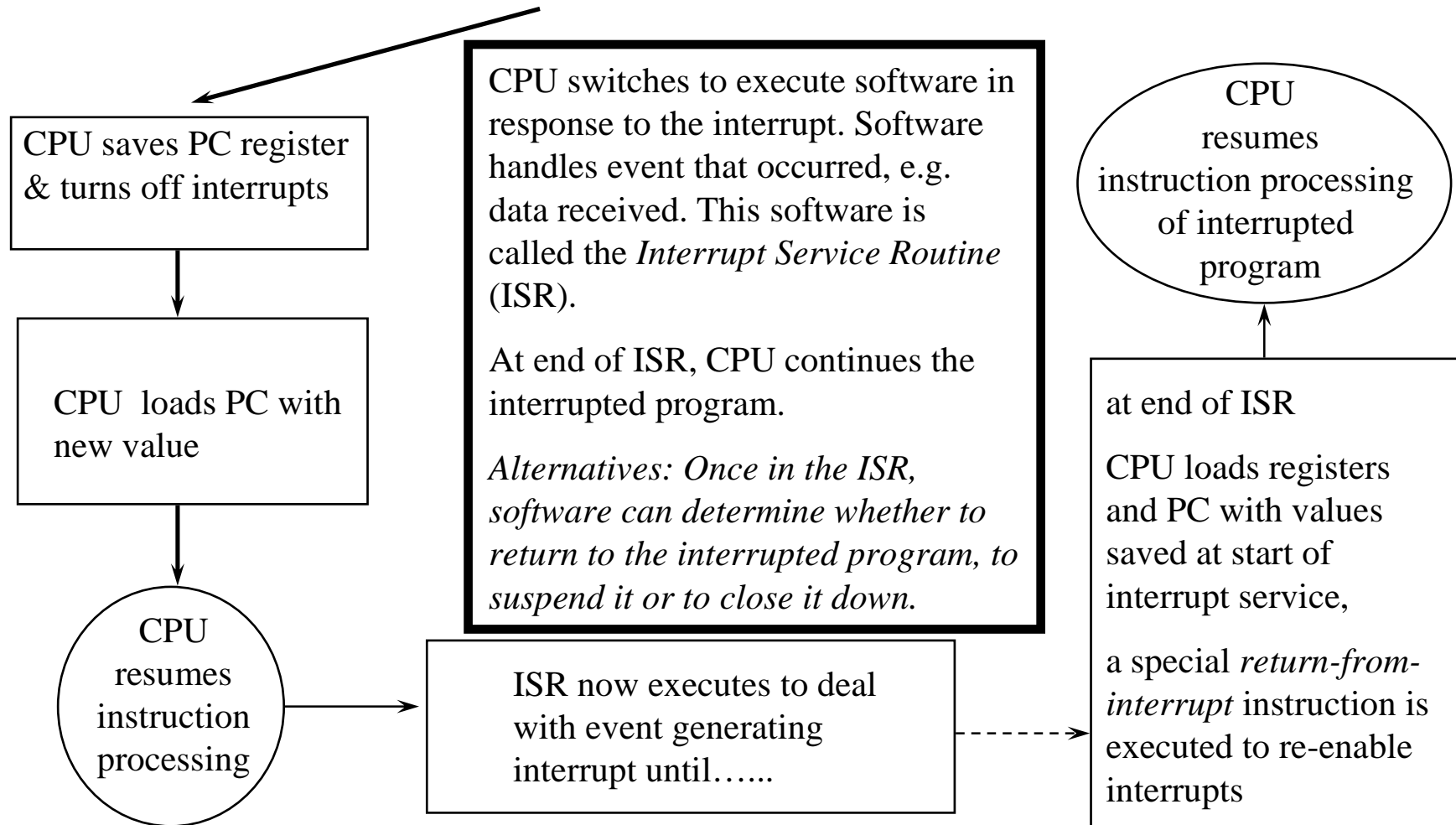
INTERRUPT HARDWARE



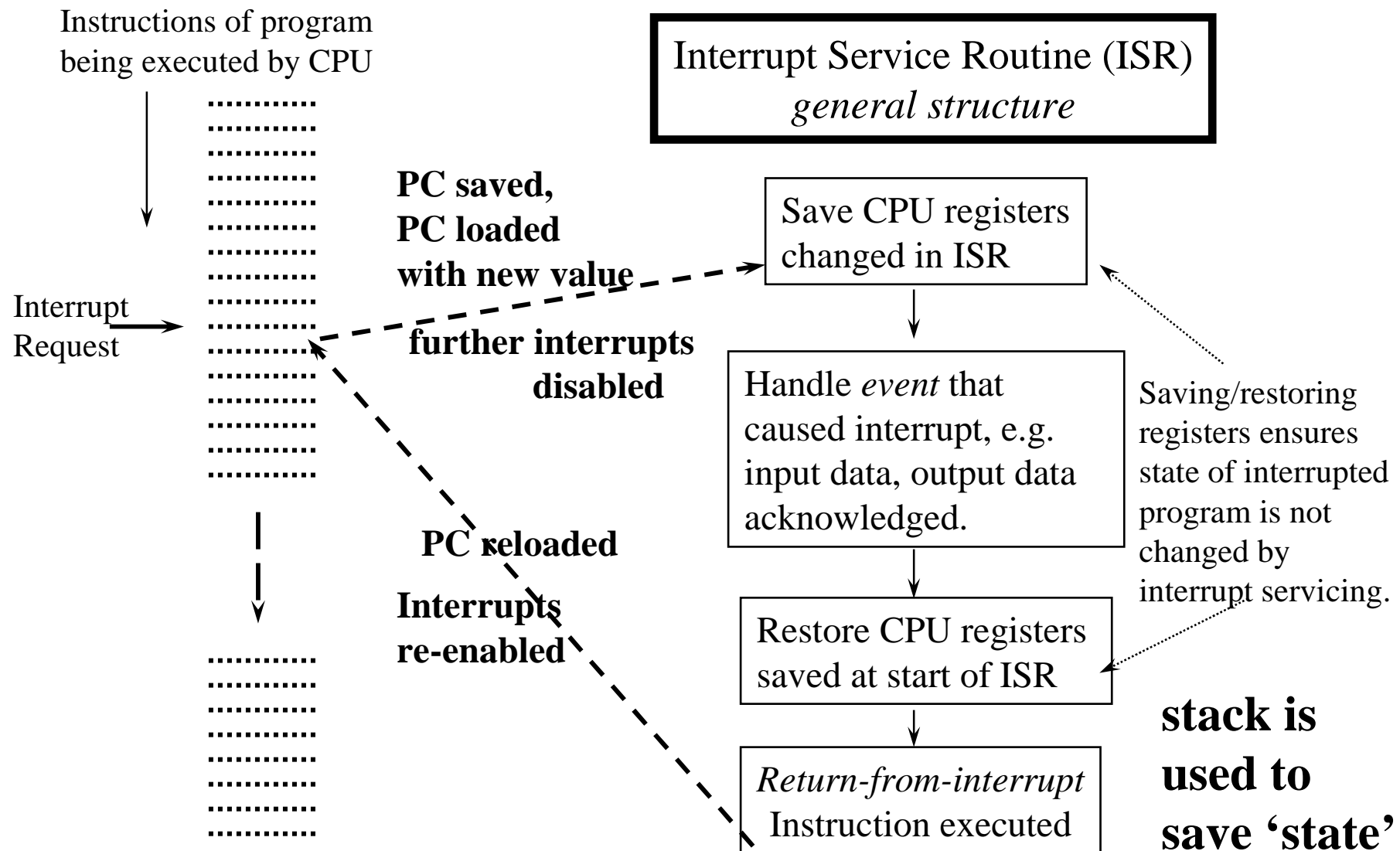
- 1) **I/O interface generates** an interrupt request signal on some Tx or Rx event.
 - 2) **Interrupt Controller** generates an interrupt request to CPU, when one or more inputs is active.
 - 3) CPU reads Interrupt Controller to get number (**vector**) identifying the highest priority active interrupt request (priority in by input number, or rotating priority allocation: the **vector** identifies the highest priority interrupt device requiring *service*).
- [This is why the controller is shown with connections to the system busses:to read vector]

CPU Activity in *servicing* an Interrupt Request

When Interrupt Request becomes *active*, CPU completes current instruction, and then executes the following operations automatically:-



Another way of viewing of Interrupt Servicing



Interrupt is very like a hardware generated function call!

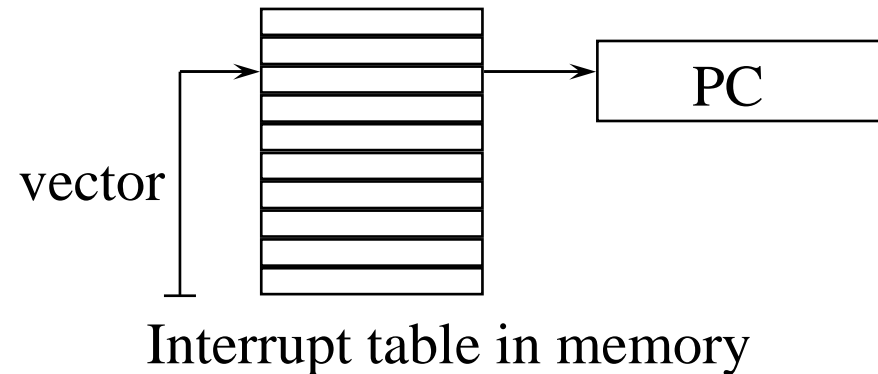
Where does new PC address come from?

Non-vectorized interrupts: new PC address is defined in CPU hardware, always go to same address:
find source of event by polling IO interfaces – slow!

Vectorized Interrupts: Vector read from controller is used to look up the address of software to handle interrupt, and placed in PC. This activity is done by the hardware.

This allows CPU to get quickly to the software to handle the event.

The interrupt table has to be set up by software .



MIPS uses Non-vectorized interrupt, so ISR is found by software.

Interrupt are very good for *low frequency* data.

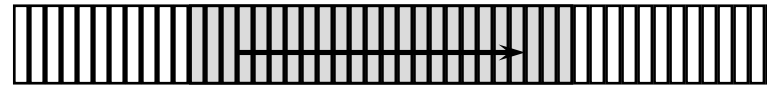
There is a large overhead in saving and restoring program state (registers & PC), which *limits* the maximum frequency of data that can be handled.

How does ISR handle interrupt event?

One way is to use circular buffers to queue data

Input Device:

ISR puts
input data on
to tail of
queue in
circular
buffer



data removed
by *consumer*
program
from head of
queue at
convenient
time

Output Device:

Data producer
program
puts output
data into tail
of queue.



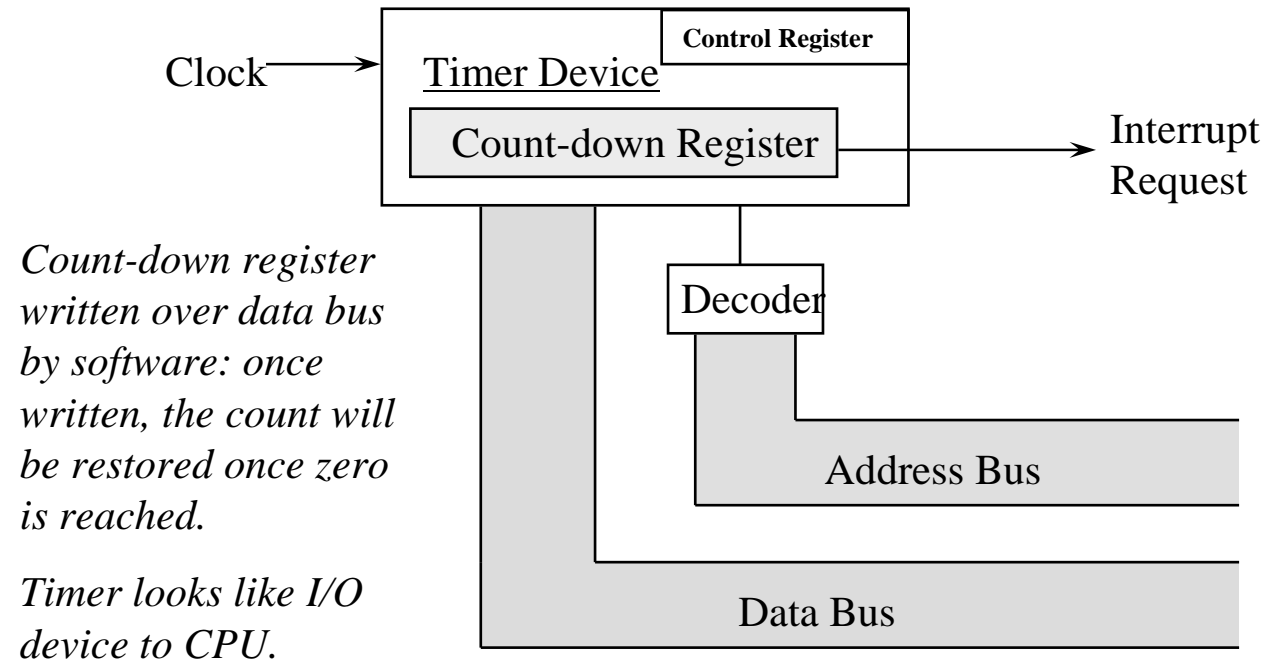
data removed
from head of
queue by ISR
and output

Timer Device

CPU writes count into count-down register and starts counting by writing into a control register in the timer.

Register counted down by one on each clock rising edge.

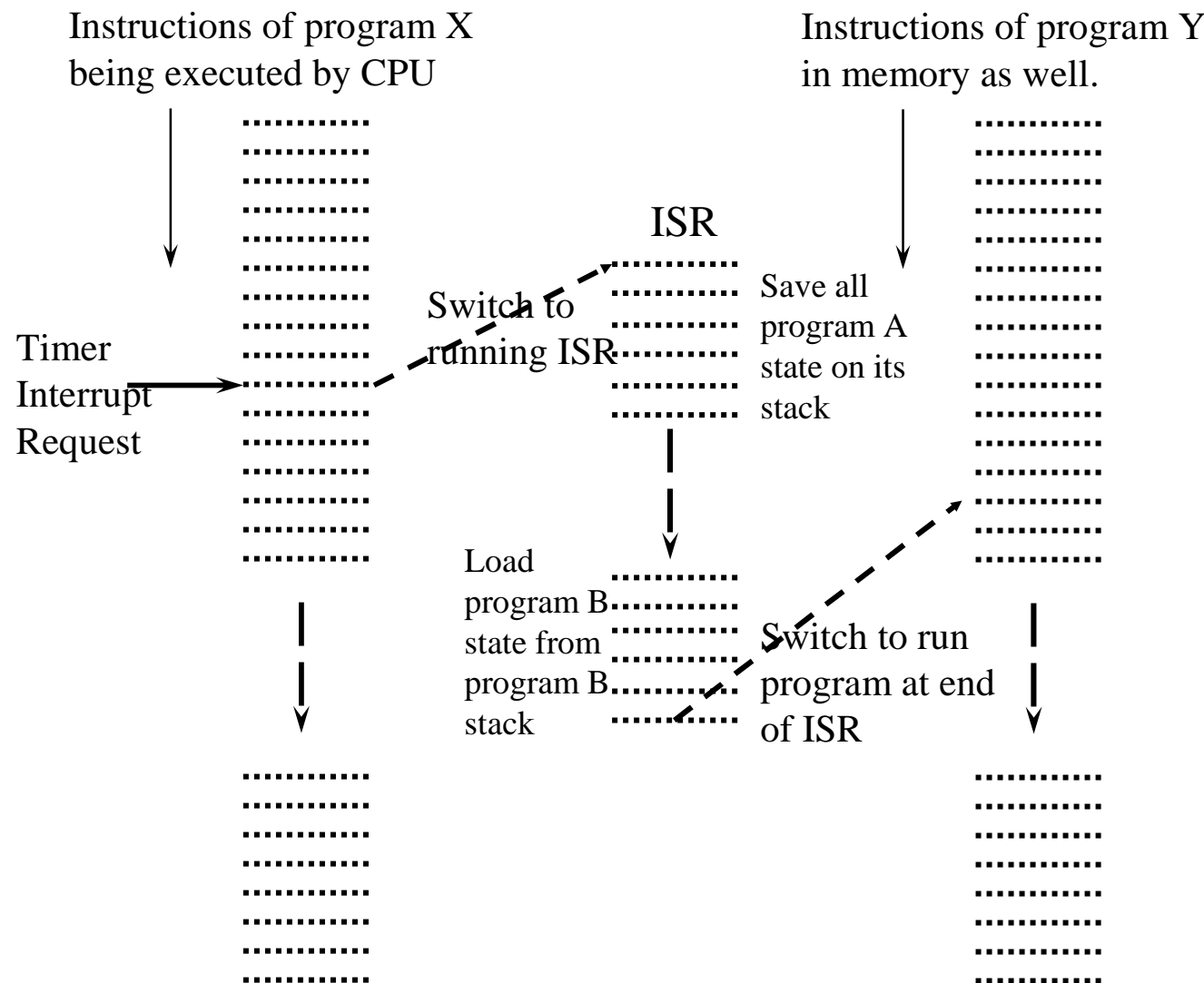
When count register reaches zero, interrupt generated and register reloaded.



Timer generates interrupts at regular time intervals.

Instead of re-starting interrupted program after timer interrupt, operating system may suspend this program and switch to run another program. Thus, each program is executed for a time slice at regular intervals: this is how an operating system runs several programs at once.

Switching processes on interrupt in an operating system environment, e.g. linux.



On any interrupt, but particularly on a timer interrupt, the operating can store the 'state' of the interrupted program, i.e. the register contents, on the program stack. It can then resume the execution of a previously suspended program or even start the execution of a new program (usually called a process). In this way, the operating system can seem to be executing several processes at a time by running allowing one process to execute on the CPU for short period (a 'time slice') before suspending the process and allowing another process to run.

The ISRs are considered part of the operating system.

A process can be suspended for a long while while awaiting some I/O activity to complete.

Exceptions

These are very similar to interrupts in many ways.

Exceptions are caused by the current instruction being executed.

Interrupts are always triggered outside the CPU and are caused by events that are being awaited e.g. an I/O transfer completing, a timer count elapsing.

Exceptions are either triggered inside the CPU or is caused by an error condition on a transfer to memory or I/O interface. The last is signalled via BUS ERROR input to the CPU.

Internal CPU exceptions are often error conditions, e.g. divide-by-zero exception, an address error (perhaps an odd address where an even one is expected)

An exception is processed in an exactly similar way to an Interrupt except that the vector used to look up the vector table is generated by the CPU.

Exceptions are generally triggered by software actions rather than hardware events.


System Call Exception – the exception that is an NOT an error

Used in a computer with a protection system to allow a user program to call a particular service in the operating system to perform a “privileged” operation.

Example services:

open a file for reading or writing
reading or writing data to an open file
write data to the screen
allocate some space on the heap.

*All these are activities for which
could corrupt the system, breach
security*



User Code:- Set up data on stack for service required
Set identifier for service
Execute system call instruction – “sc”, “trap”
system call exception generated;
“exception service routine” for a system call executed
When service completes the user program will continue as
if a method/function/subroutine call had been made

MIPS Exceptions

address error exception on load
address error exception on load
bus error on instruction fetch
bus error on data load or store
system call exception
breakpoint exception
reserved instruction exception
arithmetic overflow exception

MIPS Operation

On interrupt/exception:

Hardware operations

- wait for completion of current instruction
- save pc to epc register (exception program counter register)
- set type of event in *cause* register
- load pc with address 80000080
- restart execution

Interrupt/Exception Service Routine resides at address 0x80000080

The ISR must discover the cause of the event and then execute software to deal with the event.

It is the ISR responsibility to save the state (register values) of the interrupted program if this program is to be continued later.

If interrupt occurred, then must discover what device generated it.