Chapter 4: Interrupts, Traps, and Exceptions

Discontinuities

Asynchronous: not part of intended activities

- Occurs unexpectedly with respect to other ongoing activities in the system

Synchronous: part of your intended activities

- Occurs at well-defined points of time aligned with the intended activities of the system.

Interrupts

An interrupt is the mechanism by which devices catch the attention of the processor.

- We will consider only discontinuities caused by external devices as interrupts

Exceptions

Programs may unintentionally perform certain illegal operations (for example divide by zero) or follow an execution path unintended in the program specification.

- Some languages such as Java can also intentionally generate an exception
- Exception = some condition that deviates from normal program execution

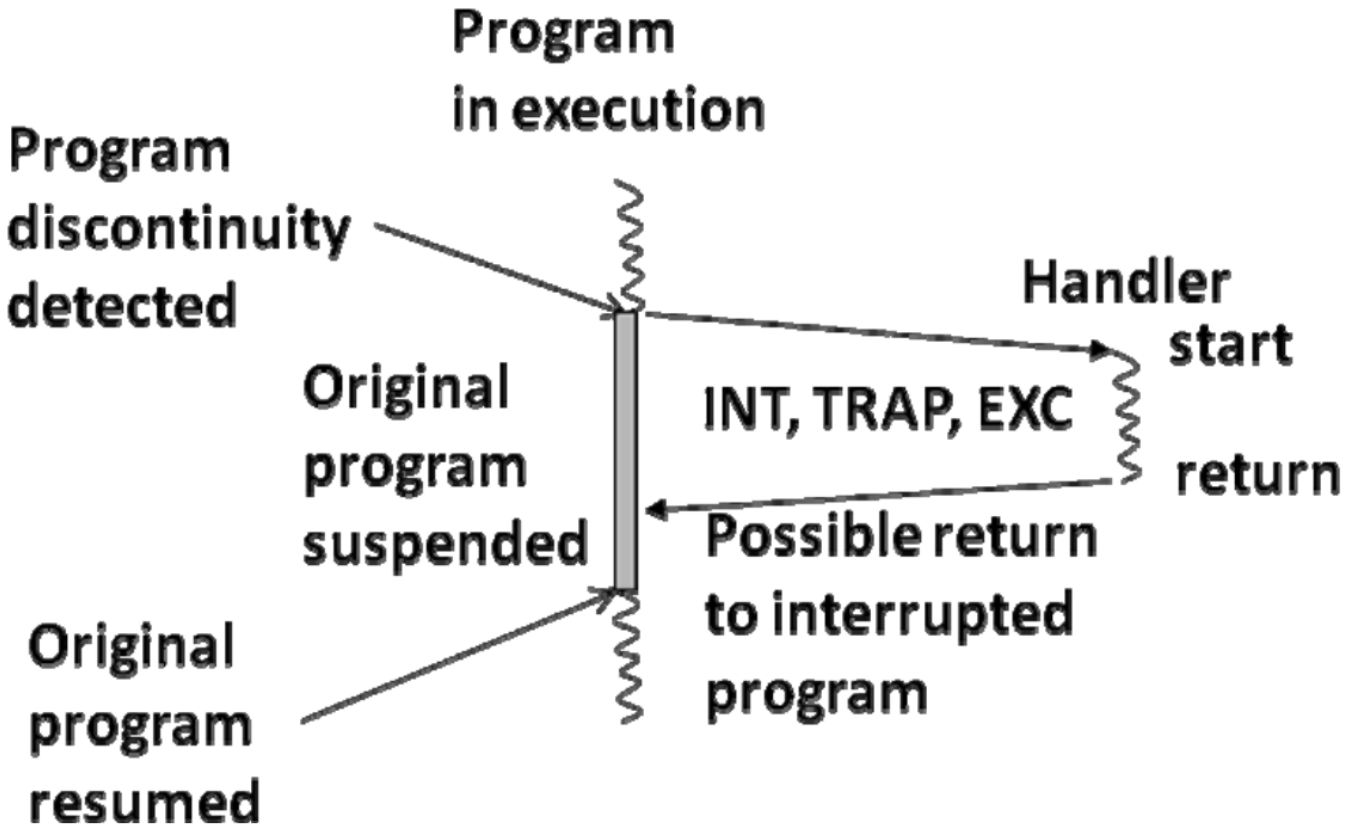
Traps

Programs often make system calls to read/write files or for other services from the system

- Trap lets the program to **fall into** the OS, which then decides what the user program wants
- Software interrupts = same as traps
- Internally generated conditions and are synchronously generated
- Some traps are intentional, others are not
- A “trap” is an internal condition that the currently running program has no way of dealing with on its own, and if anything, the system has to handle it

Type	Sync/Async	Source	Intentional?	Examples
Exception	Sync	Internal	Yes and No	Overflow, Divide by zero, Illegal memory address, Java exception mechanism
Trap	Sync	Internal	Yes and No	System call, Software interrupts, Page fault, Emulated instructions
Interrupt	Async	External	Yes	I/O device completion

Dealing With Discontinuities



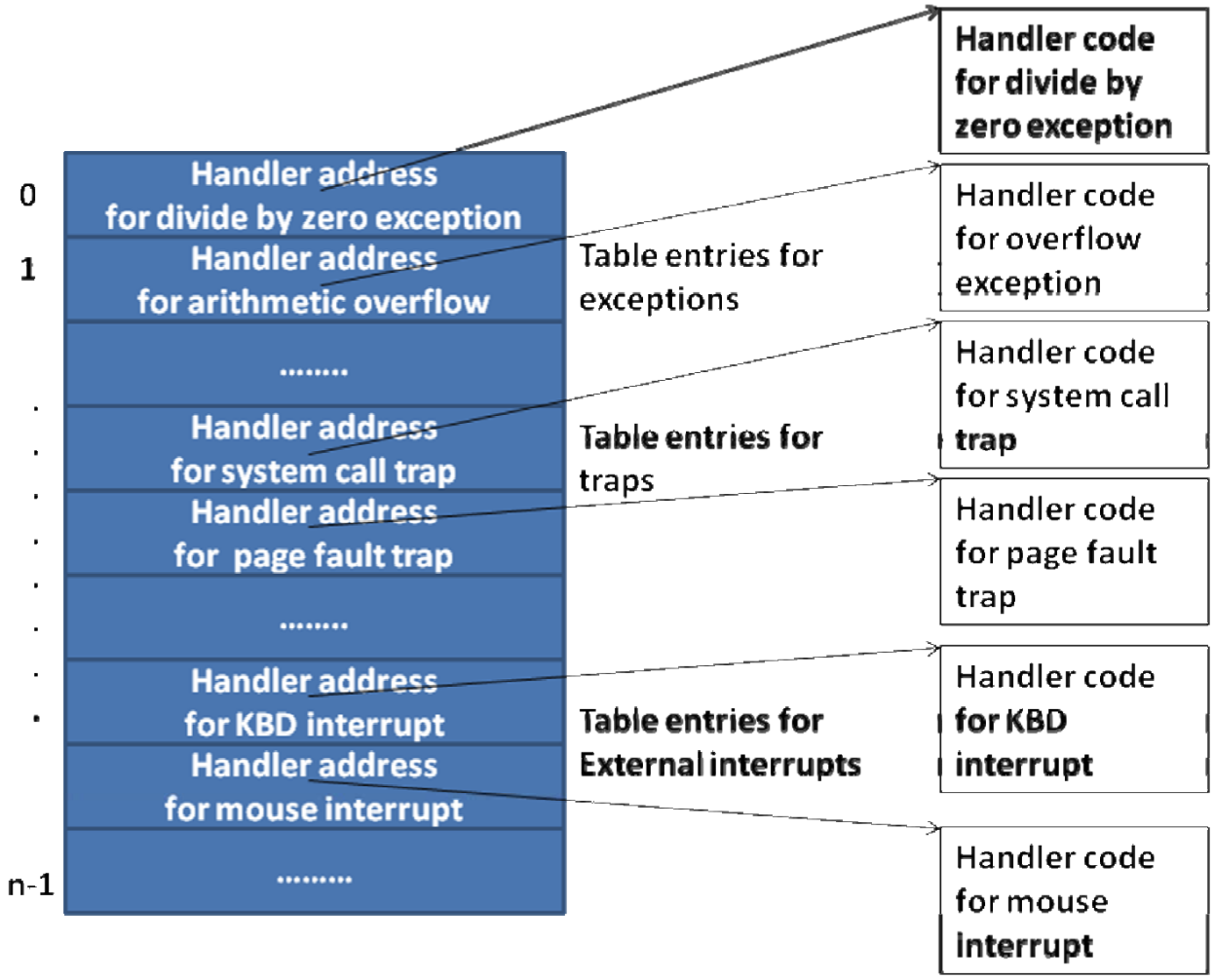
Four tricky things about discontinuities

1. Can happen anywhere during execution (even in the middle of exec)
2. Discontinuity is unplanned for and possibly unrelated to the current program. Hardware has to save program counter value before control goes to handler
3. Hardware has to determine the address of the handler to transfer control from the currently executing program to the handler
4. Handler has to discover how to resume normal program execution

Interrupt Vector Table

- Each discontinuity is given a unique number (also called a vector)
- OS sets up this table at boot time

- Processor uses IVT to look up a specific handler upon detecting a discontinuity



In the case of traps and exceptions, the hardware generates this vector internally. We introduce an exception/trap register (ETR), internal to the processor, for storing this vector

In summary,

1. The architecture may itself define a set of exceptions and specify the numbers (vector values) associated with them. These are usually due to runtime errors encountered during program execution (such as arithmetic overflow and divide by zero)
2. The operating system may define its own set of exceptions (software interrupts) and traps (system calls) and specify the numbers (vector values) associated with them
3. The operating system sets up the IVT at boot time with the addresses of the handlers for dealing with different kinds of exceptions, traps, and interrupts
4. During the normal course of execution, the hardware detects exceptions/traps and stashes the corresponding vector values in ETR
5. During normal course of execution, the hardware detects external interrupts and receives the vector value corresponding to the interrupting device

6. The hardware uses the vector value as an index into the IVT to retrieve the handler address to transfer control from the currently executing program

Modifications to FSM

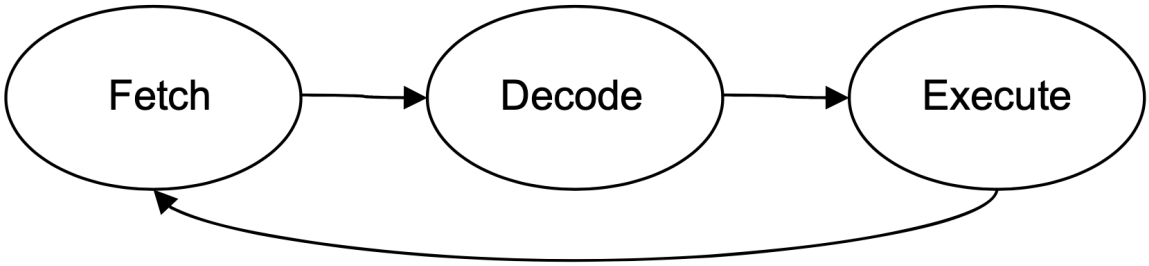
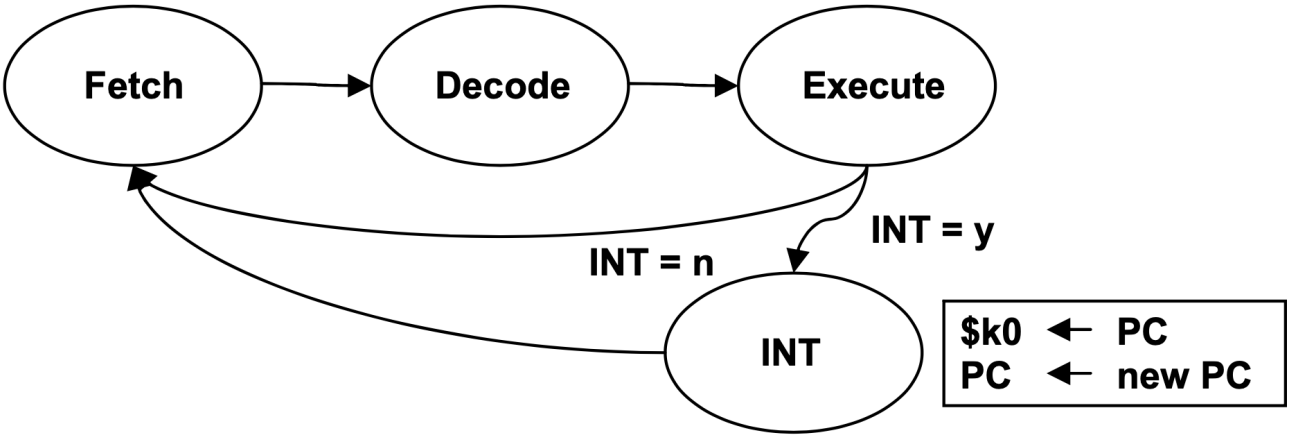


Figure 4.4-(a): Basic FSM of a processor



We introduce a new macro state **INT** that checks if there is a pending interrupt

- If INT=y, we transition to INT macrostate
- If INT = n, we resume to Fetch instruction

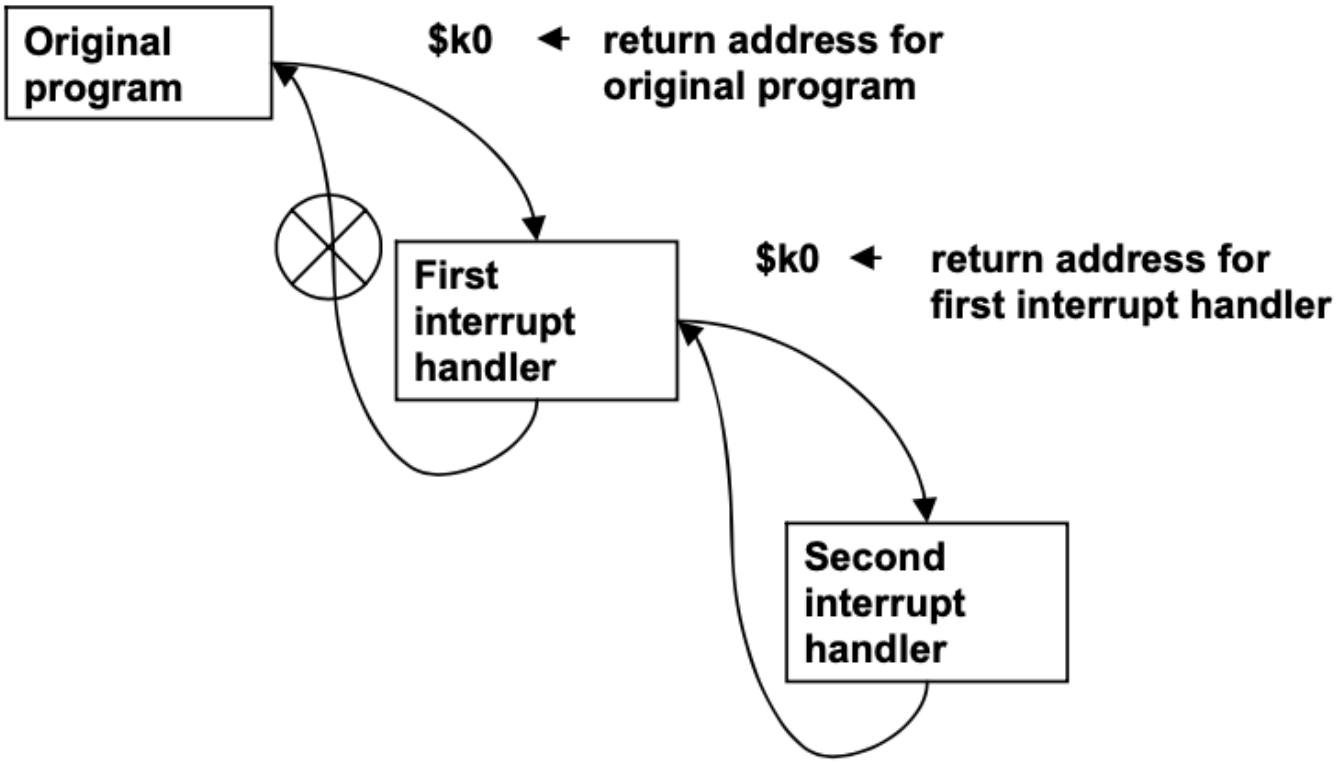
We don't want to check for interrupt after each macrostate since the datapath of a processor includes several internal registers that are not visible at the level of the instruction set architecture of the processor

In short, INT macrostate

1. Saves PC into $k0
2. Takes PC Value of the handler address from the device, loads it into PC, and goes to FETCH

Since an interrupt can happen at any time, the handler has no way of knowing which registers are being used. Thus, it has to save **ALL** visible registers even if the program only uses R2 and R3.

Handling Cascaded Interrupts



By the time we get to the second interrupt handler, we already lost the return address to the original program.

Therefore, while we are in the INT state, we need to hand over control to the handler and disable interrupts

What if we do this?

```
Handler:
    /* Interrupts are disabled when we enter */
        save $k0
        enable interrupts
        save processor registers
        execute device code
        restore processor registers
        disable interrupts
        restore k0
        return to original program
```

Okay, so let's say another interrupt occurs at the instruction `restore $k0`. When will our original program be resumed? **Never**

It should be noted, however, that it might not always be prudent to entertain a second interrupt while servicing the current one

We have a choice between

- ignore the interrupt for a while (e.g. OS is handling a higher priority interrupt)
- attend to the interrupt immediately

Returning from the handler

To return to the original program, we have to do something like

```
enable interrupts
J      $k0
```

What if an interrupt occurs between `enable interrupt` and `J $k0`? We introduce a new instruction that then does both

`Return from Interrupt (RETI)`

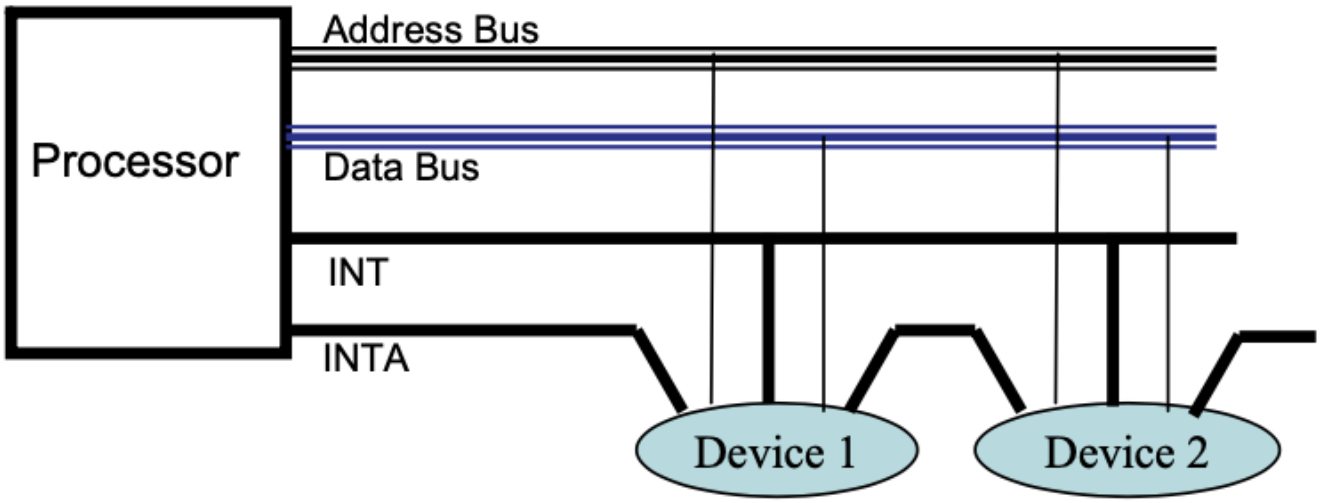
- Load PC from $k0
- Enable interrupts

This instruction is atomic, so no interrupt can occur before this instruction is fully executed

Complete Interrupt Handler

```
Handler:
        /* The interrupts are disabled when we enter */
        save $k0;
        enable interrupts;
        save processor registers;
        execute device code;
        restore processor registers;
        disable interrupts;
        restore $k0;
        return from interrupt;
        /* interrupts will be enabled by return from interrupt */
```

Datapath Details For Interrupts

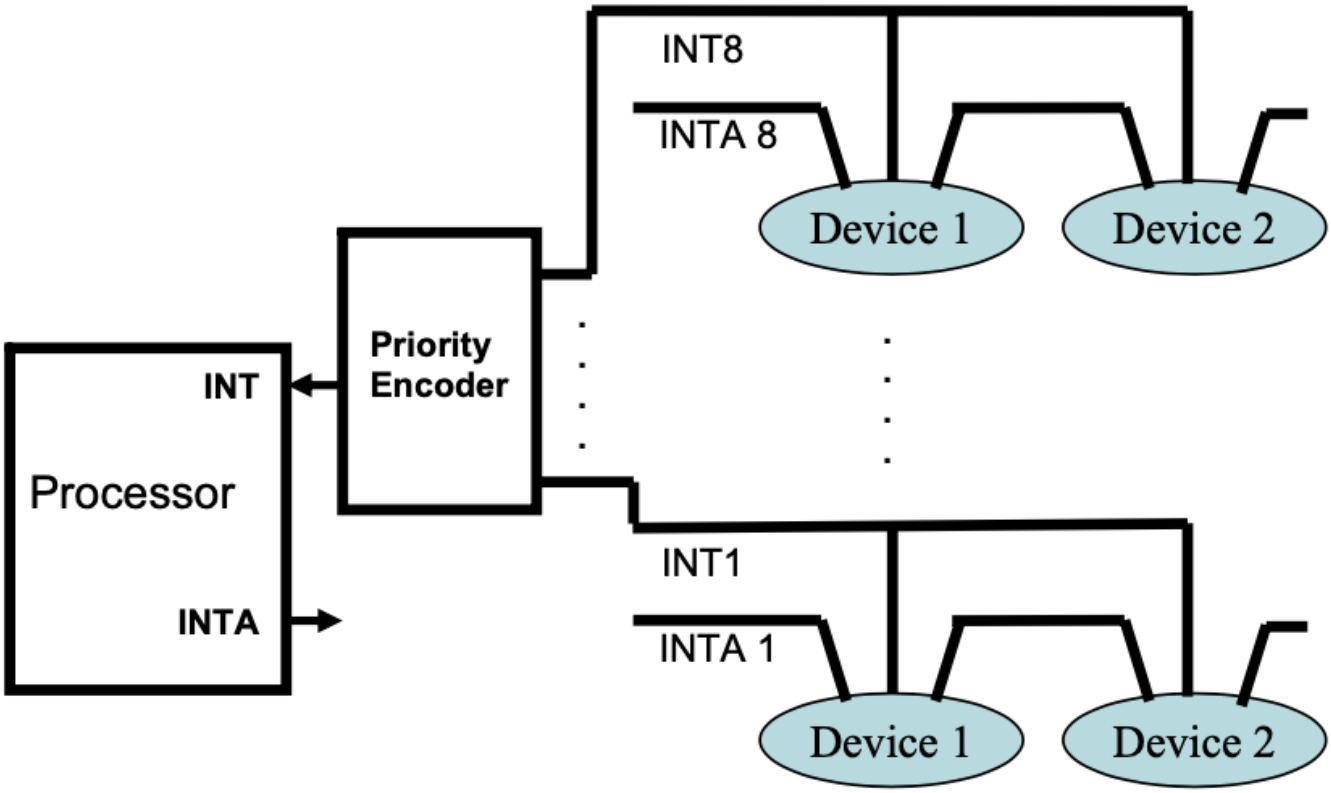


Upon an interrupt, the processor asserts the INTA line (happens in the INT macro state)

- Only one device should get this acknowledgement
- INTA daisy chains all the devices
- Daisy chain is simple but suffers from latency

Each pair of INT and INTA lines corresponds to a priority level

- Device priority is linked to the speed of the device
- The higher the speed of the device the greater the chance of data loss, and hence greater is the need to give prompt attention to that device

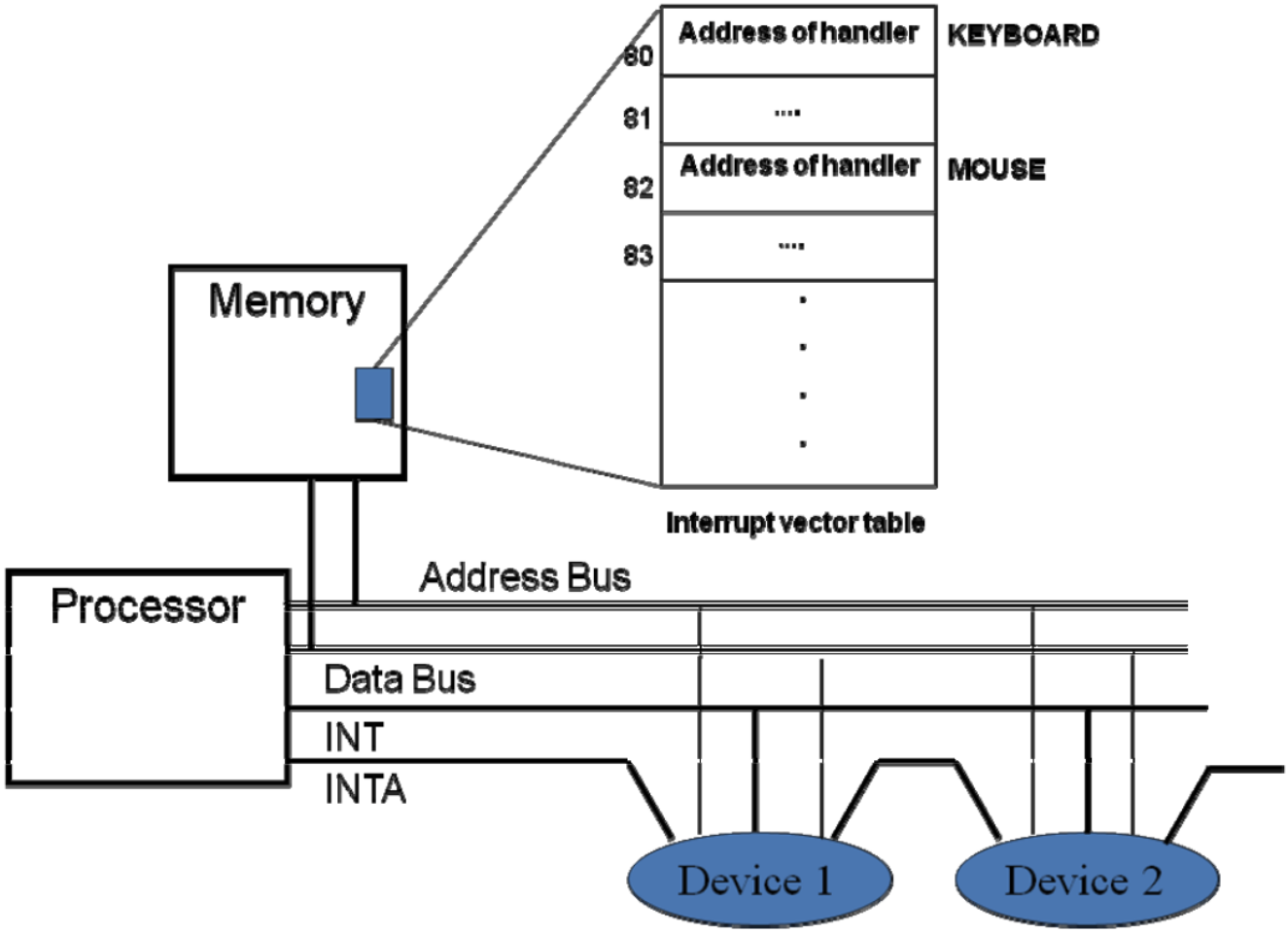


In modern processors, the burden of determining which device needs attention is shifted to an external hardware unit called **interrupt controller**

- Takes care of fielding the interrupts
- OS chains the interrupt service routines for a given priority level in a LinkedList
- Servicing an interrupt walks through LinkedList to determine the first device that has an interrupt pending

Upon receiving the INTA signal from the processor, the device puts its vector on the data bus

- Processor uses vector as the index to lookup in the IVT
- Processor retrieves handler address which loads into PC



Stack for Saving/Restoring

How does the handler know which part of the memory is to be used as a stack?

Solution: dedicate a user stack and system stack

- Upon entering the INT macro state, the FSM performs stack switching
- Need another stack pointer
- Introducing a mode bit in the processor to toggle between user and kernel mode
  - Don't want regular users executing these privileged instructions

```
INT macro state:
        $k0 PC;

        ACK INT by asserting INTA;
        Receive interrupt vector from device on the data bus;
        Retrieve address of the handler from the interrupt vector table;
        PC handler address retrieved from the vector table;
        Save current mode on the system stack;
```

```
        mode = kernel; /* noop if the mode is already kernel */
        Disable interrupts;

RETI:
        Load PC from $k0;
        /* since the handler executes RETI, we are in the kernel mode */

        Restore mode from the system stack; /* return to previous mode */
        Enable interrupts;
```

Summary

To deal with program discontinuities, we added the following architectural/hardware enhancements to LC-2200:

1. An interrupt vector table (IVT), to be initialized by the operating system with handler addresses.
2. An exception/trap register (ETR) that contains the vector for internally generated exceptions and traps.
3. A Hardware mechanism for receiving the vector for an externally generated interrupt.
4. User/kernel mode and associated mode bit in the processor.
5. User/system stack corresponding to the mode bit.
6. A hardware mechanism for storing the current PC implicitly into a special register $k0, upon an interrupt, and for retrieving the handler address from the IVT using the vector (either internally generated or received from the external device).
7. Three new instructions to LC-2200: Enable interrupts, Disable interrupts, Return from interrupt