Georgia Tech Computer Science

CS2200
Systems and Networks
Spring 2024

Lecture 8: Interrupts

Alexandros (Alex) Daglis
School of Computer Science
Georgia Institute of Technology
adaglis@gatech.edu

Lecture slides adapted from Bill Leahy, Charles Lively of Georgia Tech

Interrupts, Traps and Exceptions

- Interrupts, traps and exceptions are discontinuities in program flow

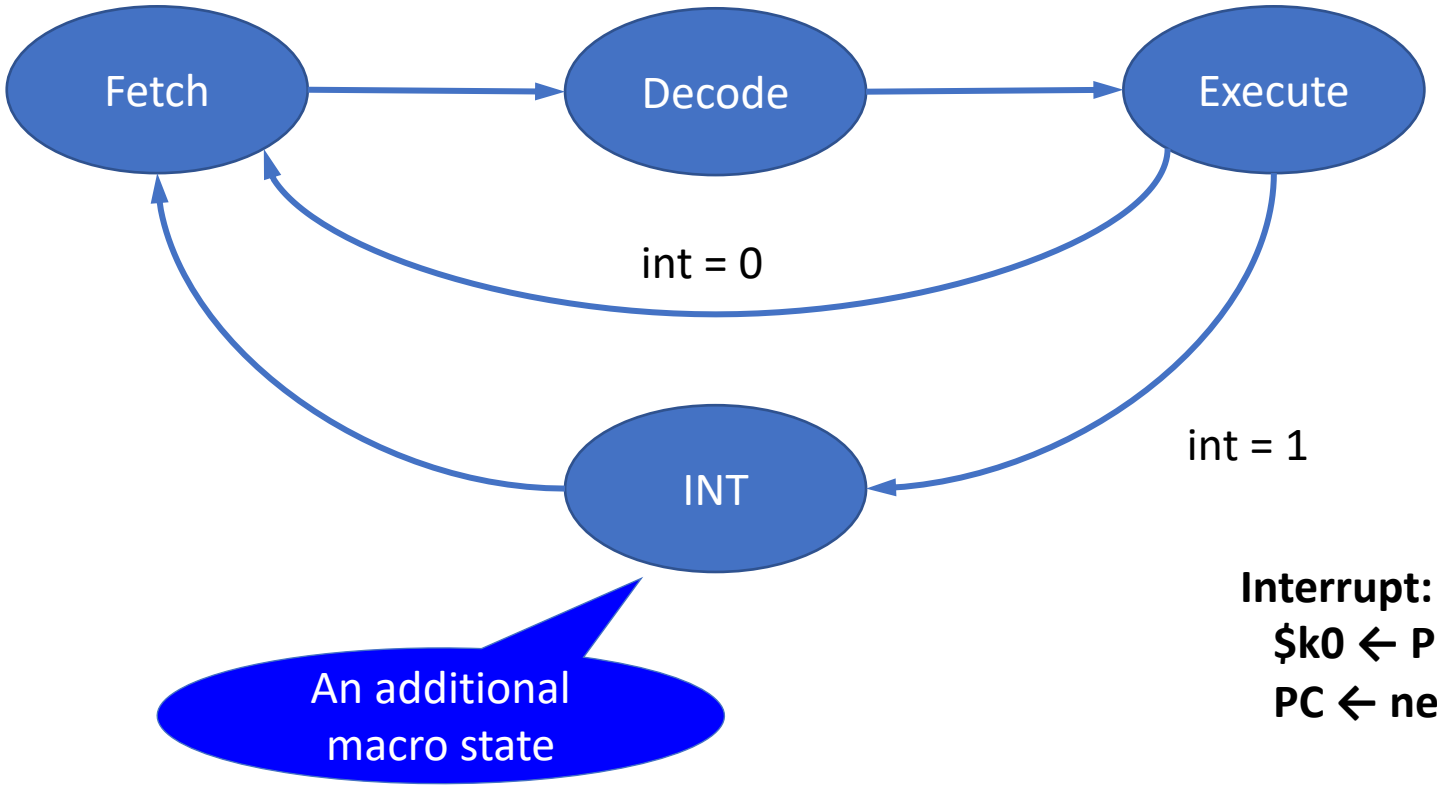
- Students asking a teacher questions in a classroom is a good analogy to the handling of discontinuities in program flow

Architectural enhancements to handle program discontinuities

- When should the processor handle an interrupt?
- How does the processor know there is an interrupt?
- How do we save the return address?
- How do we manufacture the handler address?
- How do we handle multiple cascaded interrupts?
- How do we return from the interrupt?

Modifications to FSM

Where should we take an interrupt?



Interrupt:
$k0 ← PC
PC ← new PC

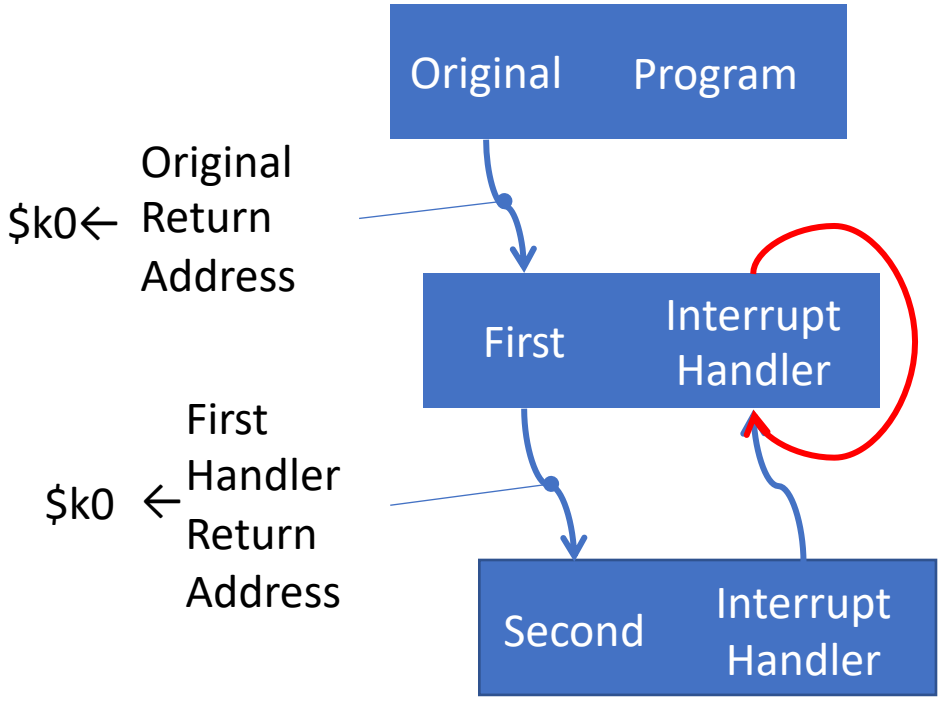
What needs to happen in software?

```
Handler:
  save processor registers;
  execute device code;
  restore processor registers;
  return to original program;
```

That's great, but…

- There are a couple of rubs.
- What happens when an interrupt handler takes an interrupt?

Handling cascaded interrupts



What needs to happen …

```
Handler:
    save processor registers
       (including $k0);
    execute device code;
    restore processor registers
       (including $k0);
    return to original program;
```

That's great, but…

- There are a couple of rubs.
- What happens when an interrupt handler takes an interrupt?
- OK. That's better. Save/restore $k0 in the handler.
- But one more little thing…
- What happens if the second interrupt hits before we save $k0?

What needs to happen …

```
Handler:
    save processor registers
       (including $k0);
    execute device code;
    restore processor registers
       (including $k0);
    return to original program;
```

What if an interrupt happens here?

No problem. We'll save $k0 first

How many instructions does it take to push a register on the stack?

What needs to happen …

```
Handler:
    save processor registers
       (including $k0);
    execute device code;
    restore processor registers
       (including $k0);
    return to original program;
```

Store $k0 first.

It takes
- Decrement $sp
- Store $k0,0($sp)

What if the interrupt happens after the Decrement!?!?

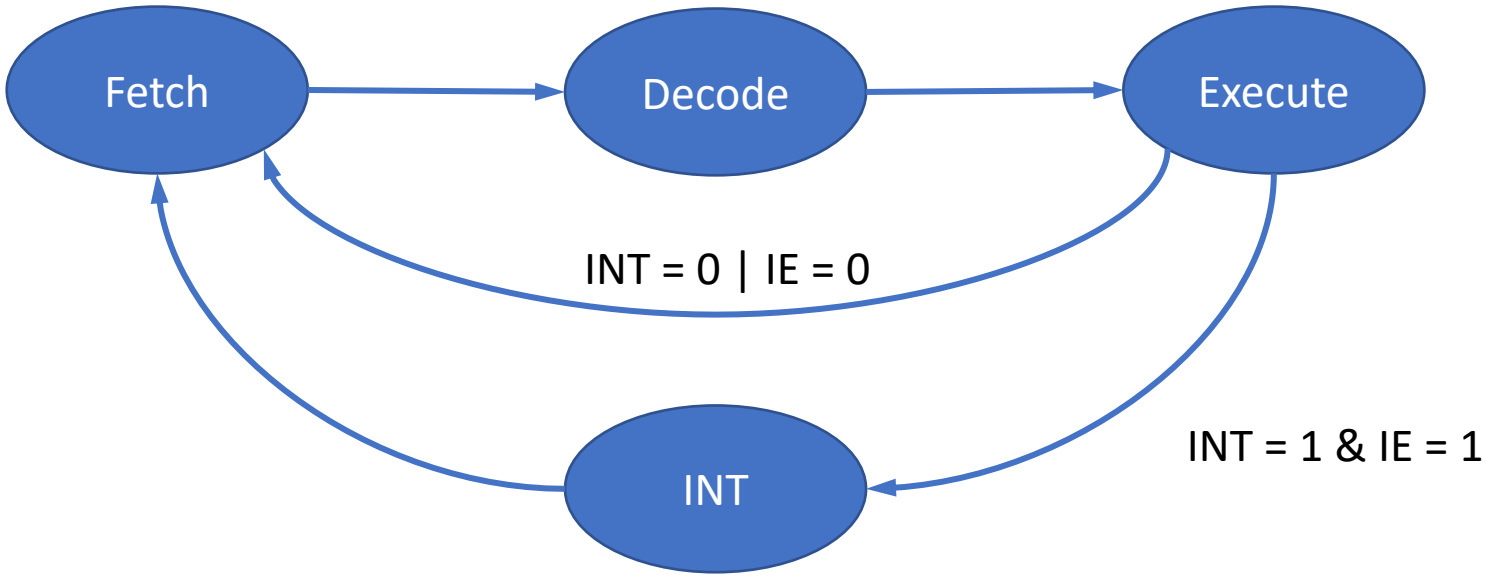
What are we lacking?

- We don't have a way to prevent an interrupt from happening between certain instructions
- In other words, we need for groups of machine instructions to behave **atomically** – i.e. as if they all were executed as a single instruction
- How could we do that?
- We could turn off interrupts between instructions?

The plan

- Create a new processor register, IE, that is 1 when interrupts are enabled
- For an interrupt to be recognized, i.e. for the microcode to advance to the INT macro state, an interrupt must be asserted **and** IE must be 1
- In the INT macro state, turn off IE before fetching the first instruction in the handler
- We need two more instructions: EI and DI to respectively set IE to 1 and 0.
- Use EI after pushing $k0 on the stack

Handling cascaded interrupts



Add 2 new instructions
Enable Ints (EI)
Disable Ints (DI)

Interrupt:
disable Ints
$k0 ← PC
PC ← new PC

Yay! This will work perfectly!

```
Handler:
    save $k0;
    enable interrupts
    save processor registers;
    execute device code;
    restore processor registers;
    disable interrupts;
    restore $k0;
    enable interrupts;
    return to original program;
```

Or does it?

What if an interrupt occurs here?

Returning from the handler

- Returning involves jumping to the address in $k0 which can be accomplished with

```
jalr $k0, $zero
```

- But as we have just seen, an interrupt at precisely the wrong moment would destroy $k0 and cause a failure
- What do we need?
- All this needs to be atomic, too!

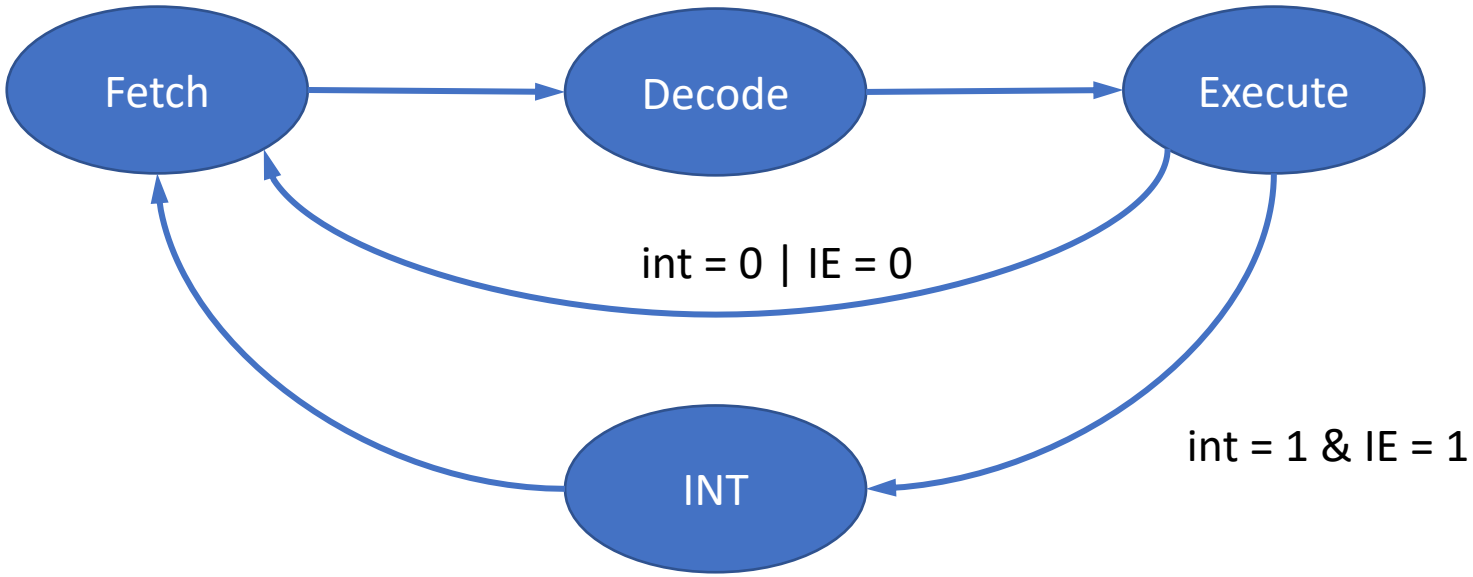
```
restore $k0;
enable interrupts;
return to original program;
```

Returning from the handler

- So we need another new instruction, RETI
- It atomically enables interrupts and sets the PC to return from the handler

- RETI:
  PC ← $k0
  IE ← 1

Handling cascaded interrupts



Add 3 new instructions
Enable Ints (EI)
Disable Ints (DI)
Return from interrupt (RETI)

Interrupt:
Disable Ints
$k0 ← PC
PC ← new PC

Summary of architectural enhancements to LC-2200 to handle interrupts (so far)

- Three new instructions to LC-2200:
  - Enable interrupts (EI)
  - Disable interrupts (DI)
  - Return from interrupt (RETI)

- Upon an interrupt, store the current PC implicitly into a special register $k0, disable interrupts, and set the PC to the address of the handler

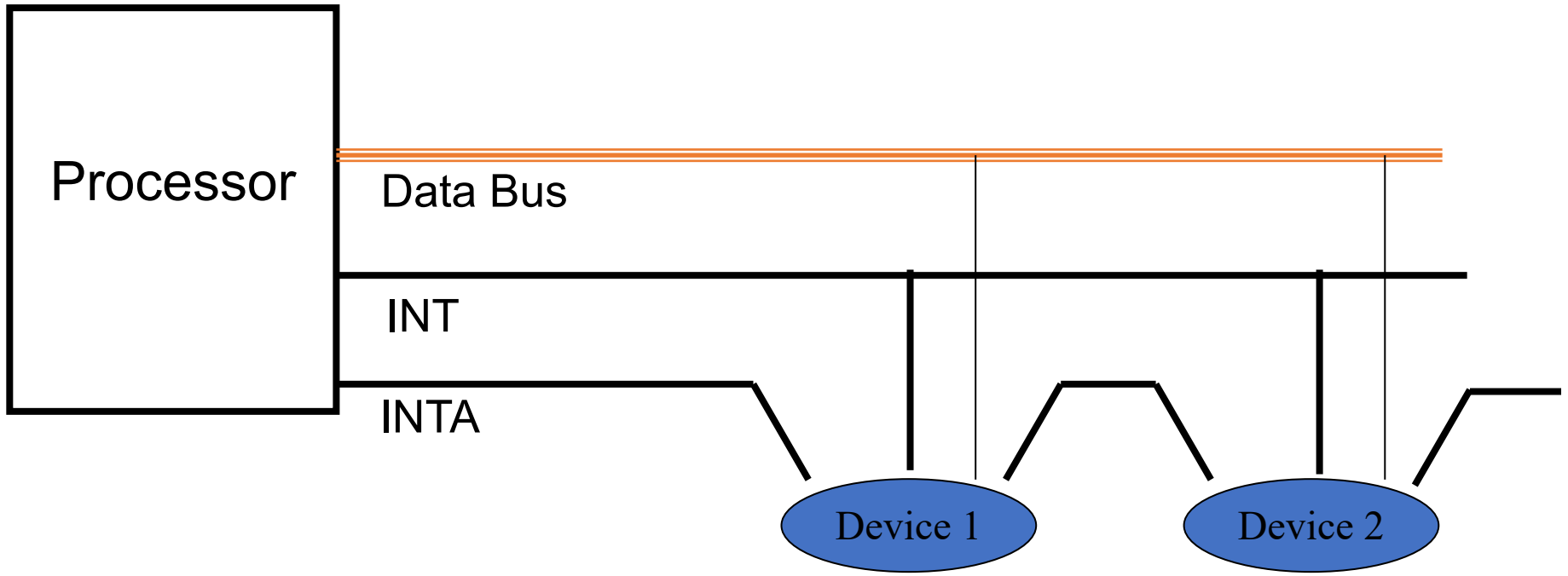
- Upon returning from an interrupt (RETI), store $k0 into the PC and enable interrupts.

Hardware details for handling external interrupts

- What we have presented thus far is what is required for interrupts, traps and exceptions

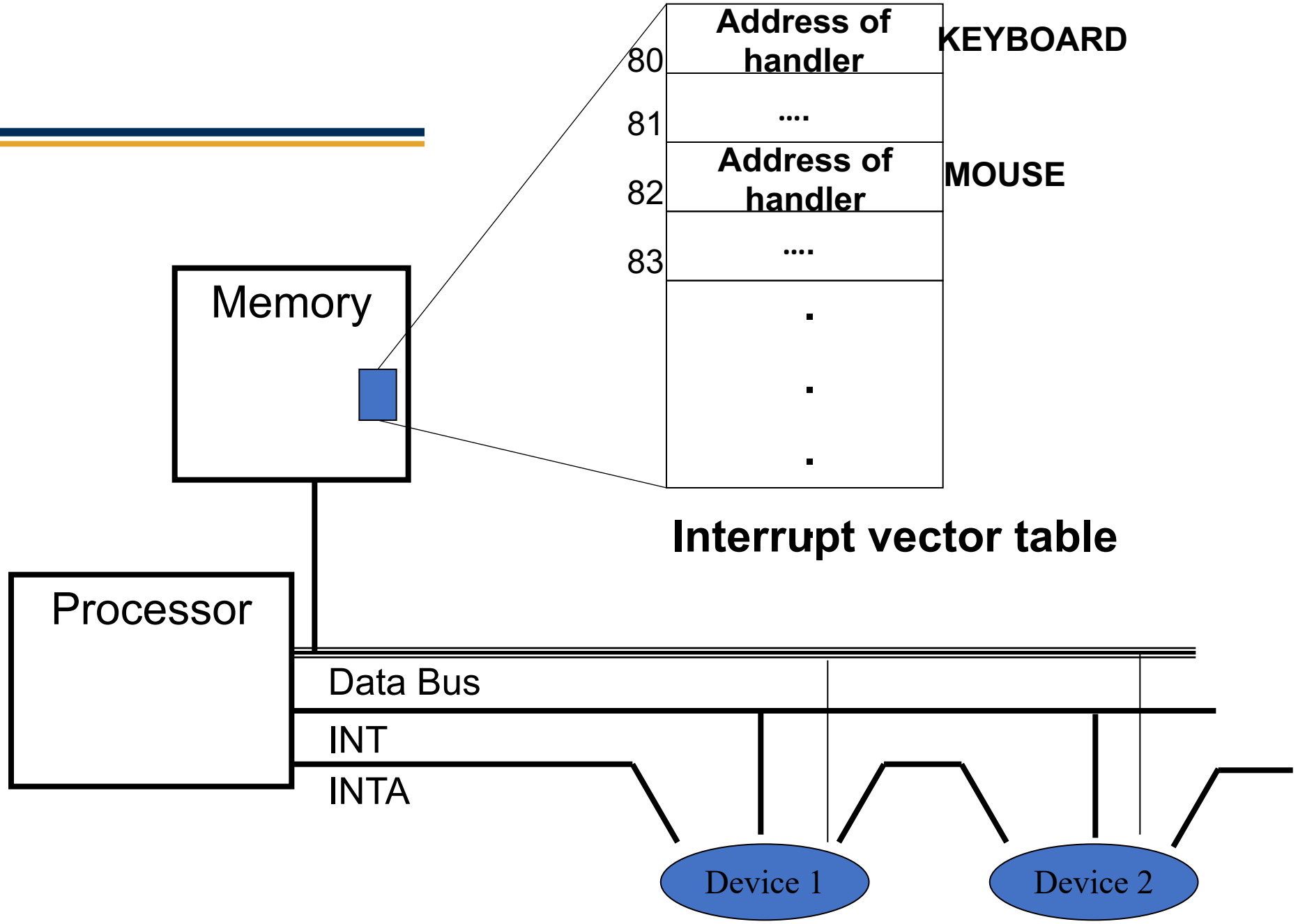
- What do we need specifically for external interrupts?
  - How does the processor know an external interrupt occurred?

Wiring for external interrupts

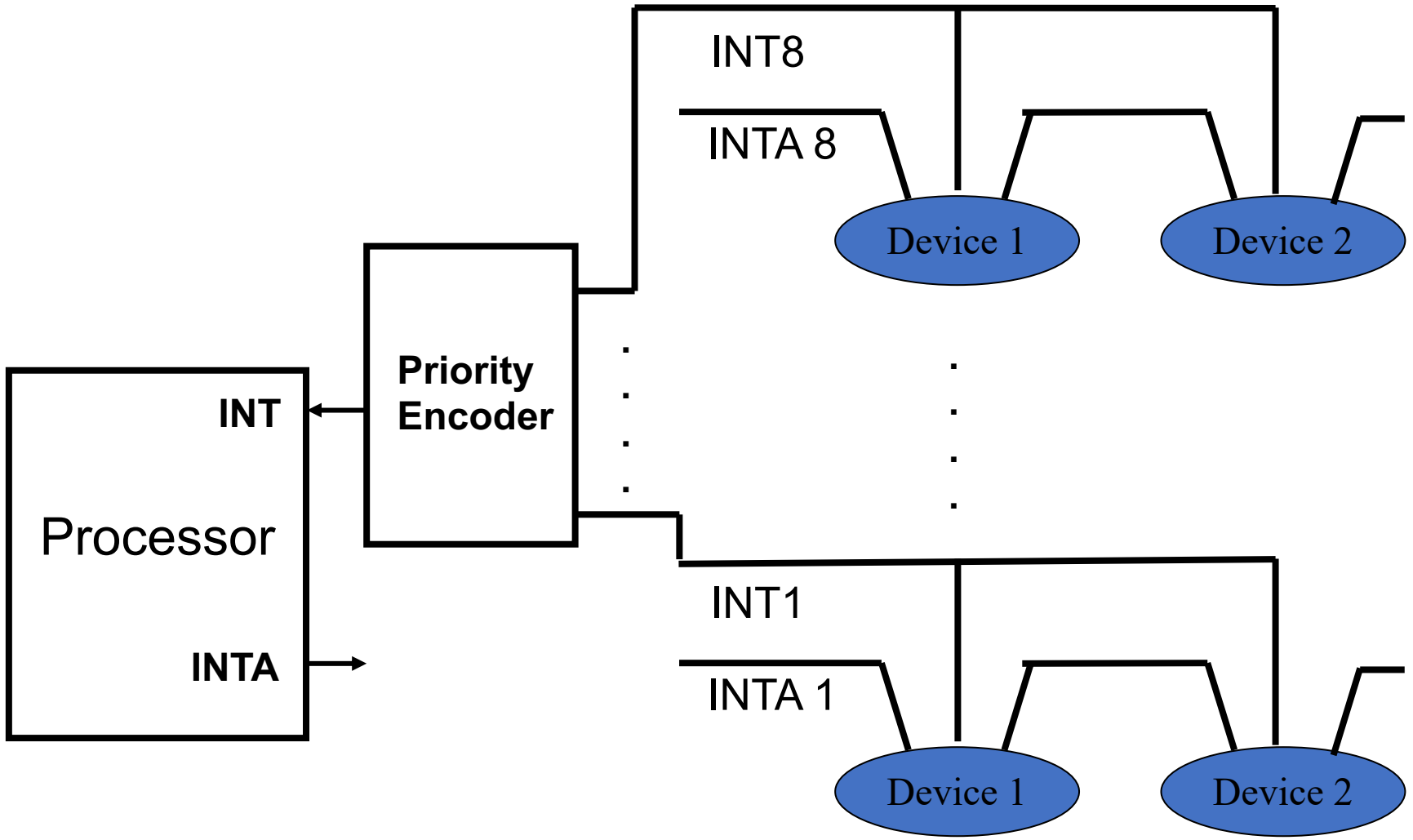


What happens at an interrupt?

- Device asserts the INT bus (it's wired so multiple devices can do this simultaneously)
- At the completion of the current instruction, CPU sees INT signal (IE = 1 & INT = 1) and microcode cycles into the INT macro state
- Microcode raises the INTA signal line
- Devices pass-through the INTA signal if they are not interrupting; otherwise the first interrupting device asserts its ID on the data bus
- Microcode reads the data bus and uses the ID as an index to determine which entry in the IVT to use to set the PC


Address of handler
KEYBOARD
80
81
....
Address of handler
MOUSE
82
83
....
Memory
Interrupt vector table
Processor
Data Bus
INT
INTA
Device 1
Device 2

Multiple interrupt priority levels



Priority Encoder (PE)

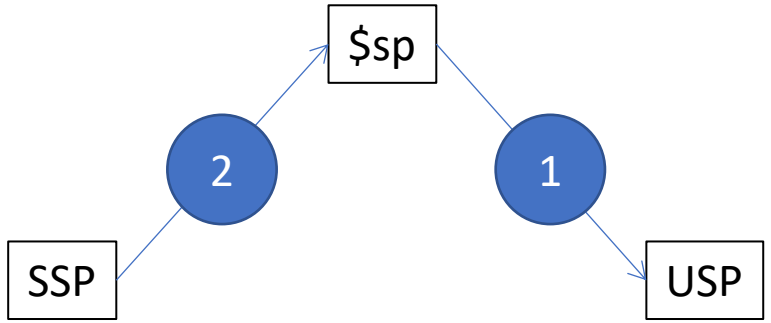
- A priority encoder takes 2^n inputs and produces a 1-bit INT output and an n-bit ID output.
- If any of the input lines is high, the PE asserts the INT output
- The PE asserts the encoded value of the first high input line onto the ID output
  - E.g. if input 5 and 7 are high on a 3-bit PE, then it asserts INT and ID=101
  - If only input 7 is high, then it asserts INT and ID=111

Where to save/restore CPU registers in the interrupt handler

- The user stack?
- Bad idea. The user doesn't even have to set $sp if he doesn't feel like it. Bad practice, but real possibility.
- Where, then?
- How about we let the OS have a system stack that we know is handled properly?

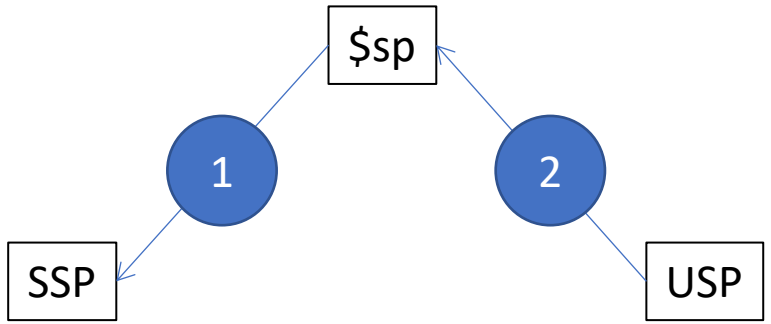
Stack for saving/restoring

- Hardware has no guarantee for stack behavior by user program (register/conventions)
- Equip processor with 2 saved stack pointers (User/System)
- On interrupt, save *user* stack pointer from $sp and restore the *system* stack pointer to $sp
- We'll need two more registers, USP and SSP

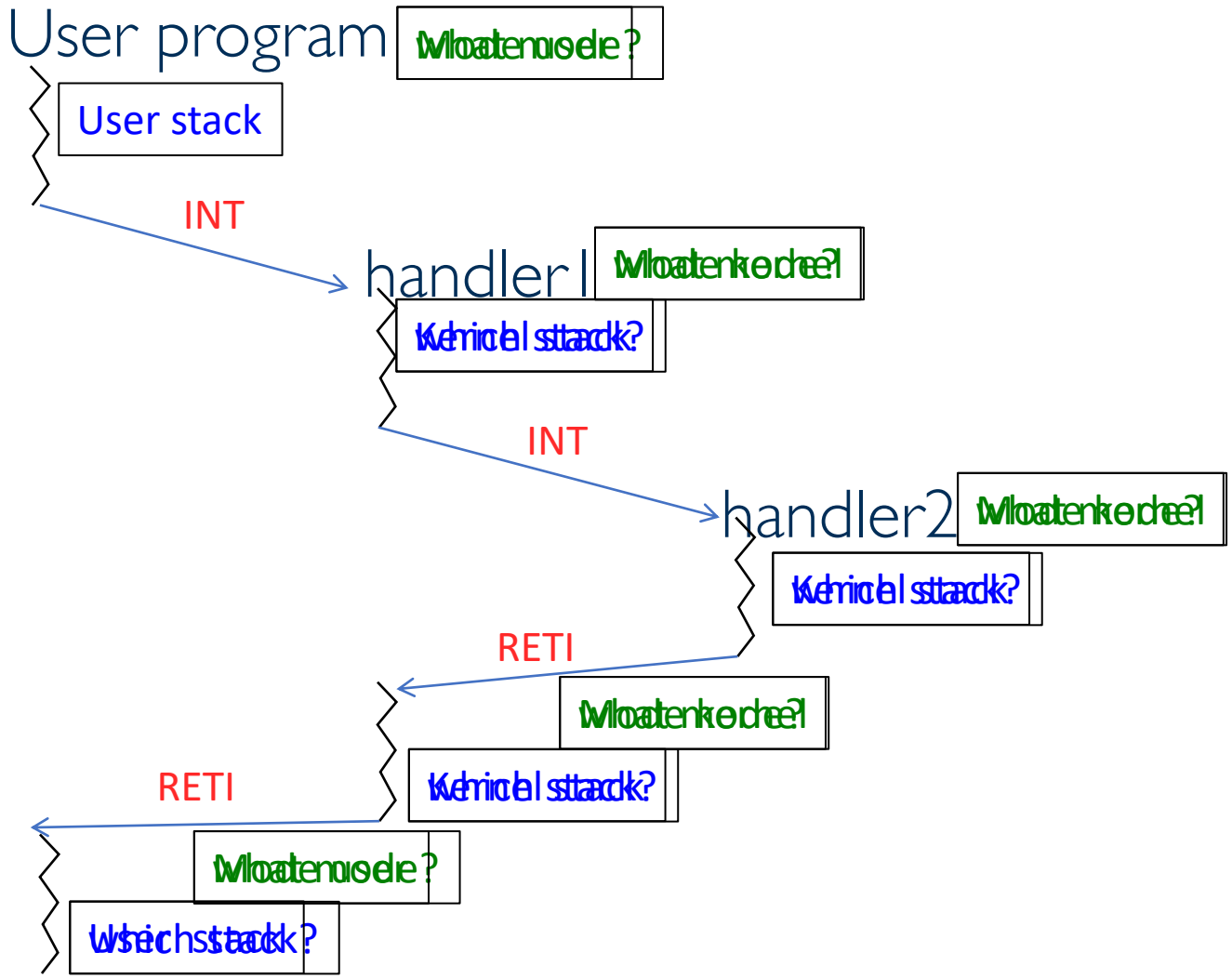


Stack for saving/restoring

- Use system stack for saving all necessary information
- Upon completion of interrupt restore registers, etc.
- Then restore user stack pointer by reversing earlier swap
- Keep a user/kernel mode flag to record whether we're using the user or kernel stack



Stacks and modes during interrupts



Summary of interrupt actions

INT macro state:

$k0 ← PC
Assert INTA to acknowledge interrupt
Receive IV (interrupt vector) from the device on the data bus
PC ← Mem[IV]
if user mode,
    USP ← $sp; $sp ← SSP
Push mode on stack
mode ← kernel
Disable interrupts

RETI instruction:

PC ← $k0
Pop mode from system stack
if user mode,
    SSP ← $sp; $sp ← USP
Enable interrupts

A working interrupt handler

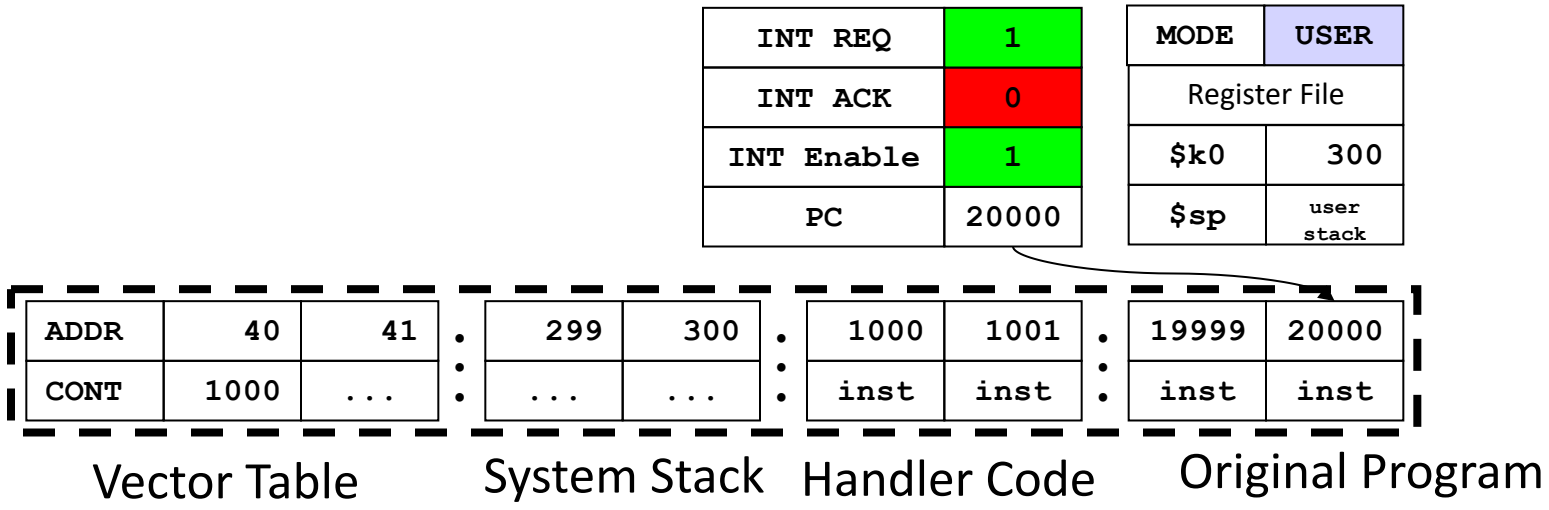
```
Handler:
  // handler starts with interrupts disabled
  push $k0 onto system stack;
  enable interrupts;
  save processor registers to system stack;
  execute device code;
  restore processor registers from system stack;
  disable interrupts;
  pop $k0 from system stack;
  // handler ends with interrupts disabled
  return to original program using RETI;
```

Architecture enhancements to LC-2200 for interrupts

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

Putting it all together

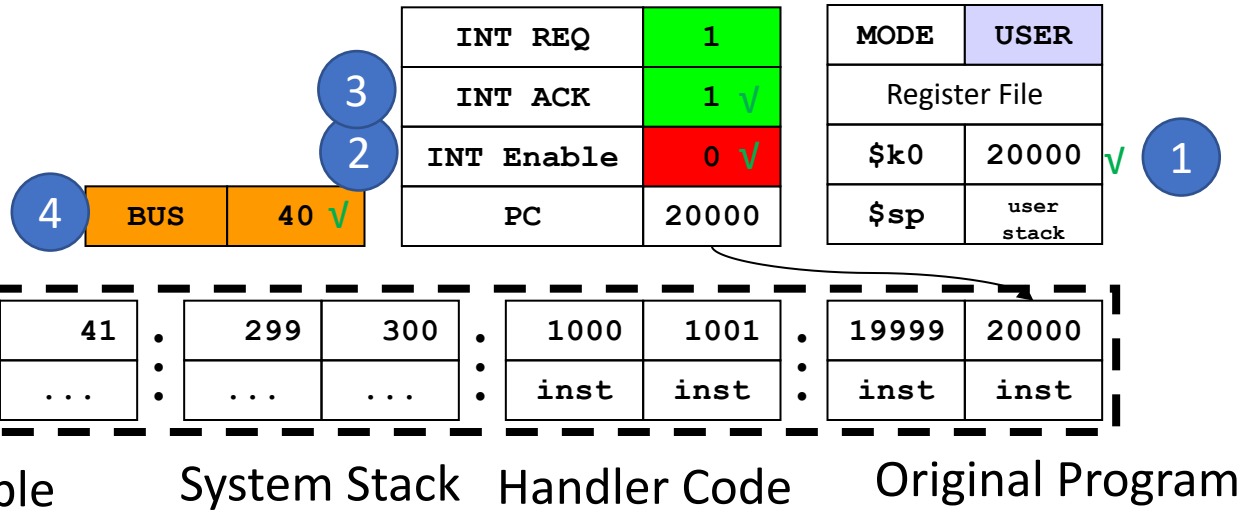
Executing instruction at 19999. The PC has already been incremented. Device signals interrupt in middle of instruction. $sp points to user stack



Putting it all together

Interrupt has been noticed.

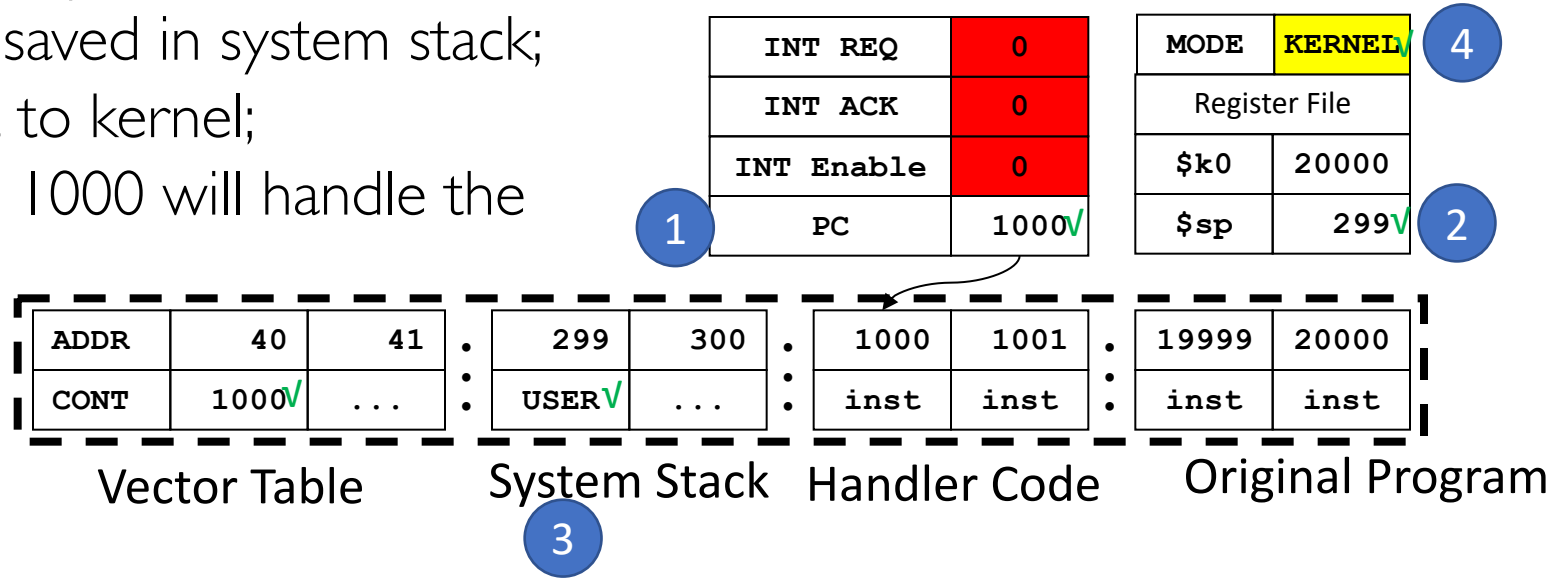
1. $k0 gets PC.
2. Interrupts are disabled.
3. Interrupt is acknowledged.
4. Device puts vector on bus.



Putting it all together

1. Handler address is put into PC
2. $sp now points to system stack;
3. Current mode is saved in system stack;
4. New mode is set to kernel;

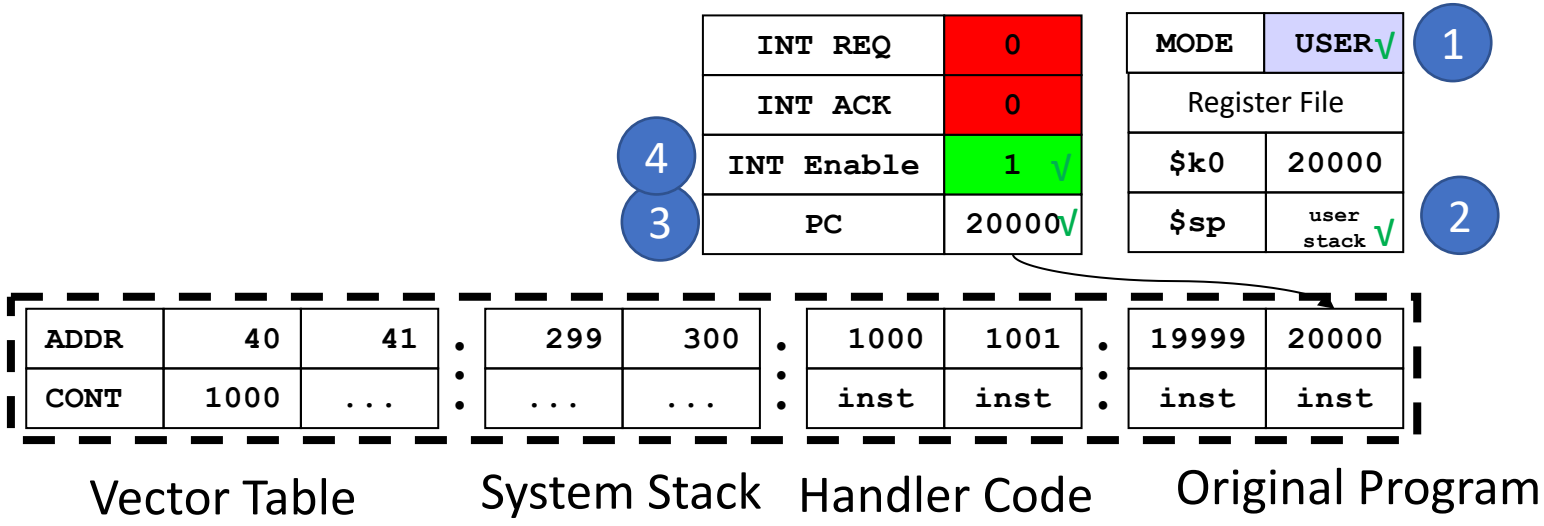
Interrupt code at 1000 will handle the interrupt. 1



Putting it all together

Handler completes.

1. RETI instruction restores mode from system stack;
   since returning to user program in this example, sets Mode to User;
2. $sp now points to user stack;
3. copies $k0 into PC;
4. re-enables interrupts



Summary

- Interrupts help a processor communicate with the outside world.
- An interrupt is a specific instance of program discontinuity.
- Processor/Bus enhancements included
  - Three new instructions
  - User stack and system stack pointers
  - Mode bit
  - INT macro state
  - Control lines called INT and INTA

Summary

- Software mechanism needed to handle interrupts; traps and exceptions are similar.
- Discussed how to write a generic interrupt handler that can handle nested interrupts.
- Intentionally simplified. Interrupt mechanisms in modern processors are considerably more complex. For example, modern processors categorize interrupts into two groups: *maskable* and *non-maskable*.
  - maskable: Interrupts that can be temporarily turned off
  - Non-maskable: Interrupts that cannot be turned off

Summary

- Presented simple treatment of the interrupt handler code to understand what needs to be done in the processor architecture to deal with interrupts. The handler would typically do a lot more than save processor registers.

- LC-2200 designates a register $k0 for saving PC in the INT macro state. In modern processors, there is no need for this since the hardware automatically saves the PC on the system stack.