CS 2200
LAB 4

Announcements

- **Test 1** today!
- **Homework 2** is due September 12th at 11:59 PM (September 15 for section C)
- **Project 1** is due September 17th at 11:59 PM
  - Demos
    - Slots will open 9/22 and will take place the week of 9/25
    - Multiplier of your project grade (i.e. getting a 50% on the demo and 100% on the autograder is a 50% score)
    - Mandatory

Brief Overview: Microcontroller, Discontinuities, Interrupt Handlers

Microcontrollers

Finite State Machine

- The microcontroller reads the current microstate and sets the control signal flags so that data properly moves around on the bus from component to component
  - Load and drive signals
  - Selecting which register to write to or read from
  - Choosing which ALU function to use
  - etc
- In a non-pipelined processor (chapter 5), the Microcontroller is a FSM.

  Fetch → Decode → Execute → Fetch → Decode → …

  - How does it know which execute state to transition to?
  - How does it know when to branch?

Engineering Conditional Branches (in the FSM)

Finite state machines transition from state to state is **sometimes based on input.**

→ Use input to determine which microstate to transition to next.

- Read opcode to go from ifetch3 to exec1 (in reality, add1 or nand1 or beq1…)
- Read comparison register (Z register in the lc-2200) to make decision in branching from beq3 to either beq4 or ifetch1

Exercise: Reading Input from the Datapath

Take 2 minutes to think:

How can the microcontroller know when it needs to use the opcode or the comparison register to choose a next state?

Exercise: Reading Input from the Datapath

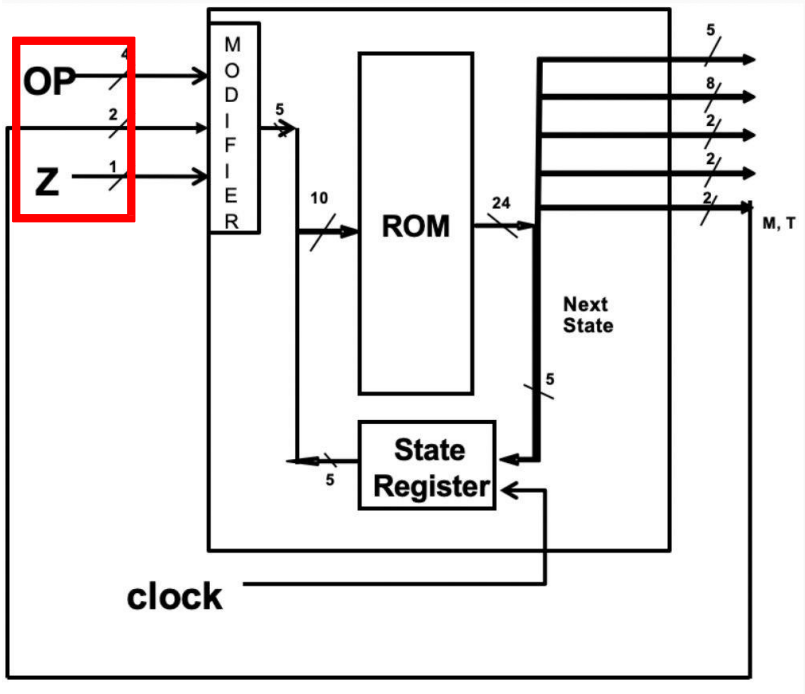
How can the microcontroller know when it needs to use the opcode or the comparison register to choose a next state?

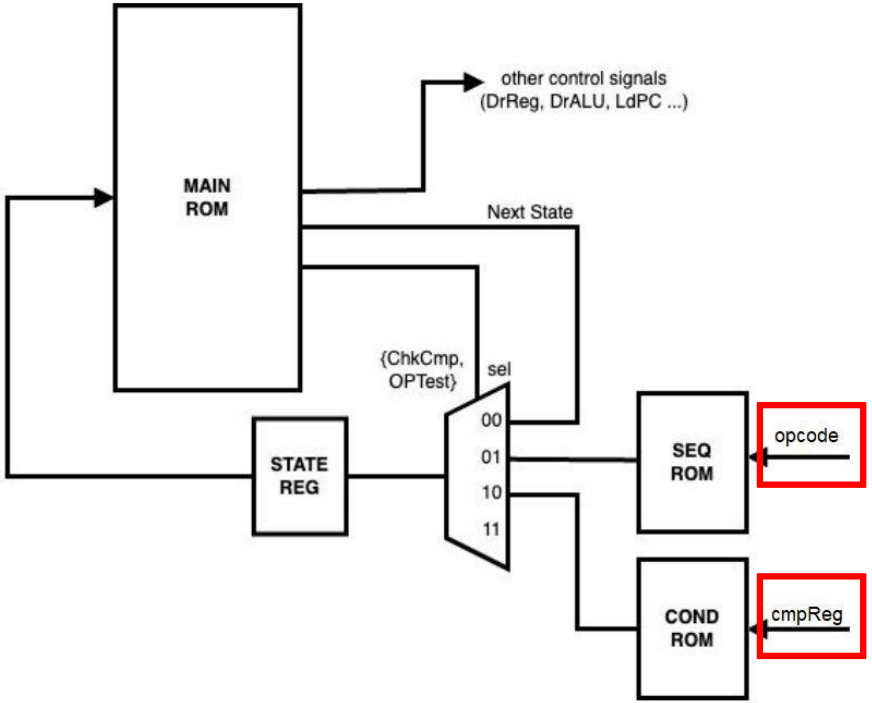
→ Have the FSM decide when it wants input using a “check opcode” and “check compreg” bit in the output (M and T in the lc-2200 and next-state mux select values 1 and 2 in the lc-2222)

- ifetch3 turns on the “check opcode” bit
- beq3 turns on the “check compreg” bit

What component actually reads the input?

1 “flat ROM” vs 3 simpler ROMS





Microcontroller Implementation - Flat ROM

Bits to represent the current state are longer since they include the opcode and comparison output.
This makes the ROM hardware bigger while including many unused states

Example:

- Ifetch1 <xxxx|x|00000> → ifetch2 <xxxx|x|00001> → ifetch3 <xxxx|x|00010>

- Ifetch3 → exec1 <xxxx|x|00011> → exec2 <xxxx|x|00100> → ……. → ifetch1
  - If opcode == 0000 (ADD): add1 <0000|x|00011> → add2 <0000|x|00100>
  - If opcode == 0101 (BEQ): beq1 <0101|x|00011] → beq2 <0101|x|00100>

```
MSB | --- 4 bit OP --- | --- 1 bit Z --- | --- 5 bit State --- | LSB
```

Microcontroller Implementation - Flat ROM

Bits to represent the current state are longer since they include the opcode and comparison output.
This makes the ROM hardware bigger while including many unused states

Example:

- Ifetch1 <0000|0|00000> → ifetch2 <0000|0|00001> → ifetch3 <0000|0|00010>

- Ifetch3 → exec1 <xxxx|0|00011> → exec2 <xxxx|0|00100> → ……. → ifetch1
  - If opcode == 0000 (ADD): add1 <0000|0|00011> → add2 <0000|0|00100>
  - If opcode == 0101 (BEQ): beq1 <0101|0|00011] → beq2 <0000|0|00110>
    (alternatives for beq2: 0000|0|00110 or with M bit set in beq1 you could use

```
MSB | --- 4 bit OP --- | --- 1 bit Z --- | --- 5 bit State --- | LSB
```

Microcontroller Implementation - 3 ROM

Smaller SEQ and COND rom hold the address in the main ROM of the relevant microstates.

Example:

- During ifetch3, the OPtest bit is enabled, setting the next-state mux control lines to 01
- SEQ ROM is used for the next state after fetch
- The opcode is 0010 (ADDI) and addi1 is located at line 13 in the main ROM
- The SEQ ROM maps the input 0010 to the output 13 and this goes into the state register
- State 13 in the main ROM says to put the contents of RegY in A.

Everything works!

Discontinuities: Interrupts, Exceptions, Traps

Program Discontinuities

Sometimes we must temporarily halt execution to perform some necessary user task. eg: keyboard input

- Analogy: A teacher stopping lecture to answer a student's question.

Need to be able to seamlessly exit and re-enter a program without affecting its flow.

- Analogy: Teacher resumes lecture after answering the question.

Exercise: Which is which?

Identify the following as interrupts, traps, or exceptions

System call

Segfault

int x = 10/0;

memory allocation

(stack overflow the error)

Force quit program

Comparison

	Interrupts	Traps	Exceptions
Occurrence	Usually due to I/O devices	Allows user to access protected OS resources	Illegal/exceptional behavior in program
Async/Sync	Asynchronous	Synchronous	Synchronous
Intentional?	Yes	Yes and No	Yes and No
Source	External	Internal	Internal

Synchronous - Occurs at well-defined points in time and in sync with CPU

Asynchronous - Occurs at random time intervals based on user/device inputs, out-of-sync with CPU

Exercise: Which is which?

Identify the following as interrupts, traps, or exceptions

System call

Segfault

int x = 10/0;

memory allocation

(stack overflow, the error)

Force quit program

Exercise: Which is which?

Identify the following as interrupts, traps, or exceptions

<interrupt>

System call
<trap>

Segfault
<exception>

<interrupt>

memory allocation
<trap>

(stack overflow, the error)
<exception>

int x = 10/0;
<exception>

Force quit program
<interrupt>

Handling Interrupts

Exercise: Handling Discontinuity

OS should be able to handle any discontinuity

Where does the responsibility fall?

Microarchitecture

ISA

Detecting discontinuities

Exercise: Handling Discontinuity

OS should be able to handle any discontinuity

Where does the responsibility fall?

Microarchitecture

- Detecting discontinuities

ISA

Detecting discontinuities

Exercise: Handling Discontinuity

OS should be able to handle any discontinuity

Where does the responsibility fall?

Microarchitecture

- Detecting discontinuities

ISA

Jumping to event handling subroutine

Exercise: Handling Discontinuity

OS should be able to handle any discontinuity

Where does the responsibility fall?

Microarchitecture

- Detecting discontinuities
- Jumping to event handling subroutine

ISA

Jumping to event handling subroutine

Exercise: Handling Discontinuity

OS should be able to handle any discontinuity

Where does the responsibility fall?

Microarchitecture

- Detecting discontinuities
- Jumping to event handling subroutine

ISA

Executing subroutine

Exercise: Handling Discontinuity

OS should be able to handle any discontinuity

Where does the responsibility fall?

Microarchitecture

- Detecting discontinuities
- Jumping to event handling subroutine

ISA

- Executing subroutine

Executing subroutine

Exercise: Handling Discontinuity

OS should be able to handle any discontinuity

Where does the responsibility fall?

Microarchitecture

- Detecting discontinuities
- Jumping to event handling subroutine

ISA

- Executing subroutine

Return back to original program

Exercise: Handling Discontinuity

OS should be able to handle any discontinuity

Where does the responsibility fall?

Microarchitecture

- Detecting discontinuities
- Jumping to event handling subroutine

ISA

- Executing subroutine
- Return back to original program

Return back to original program

How to INT

- After each execute macrostate, it checks for pending interrupts
  - Interrupt (INT macrostate):
    - disable interrupts
    - save the PC to $k0
    - save the current mode onto the stack
    - load a new PC value to jump to the interrupt handling code
    - continue to fetch state
  - No Interrupt:
    - continue to fetch state

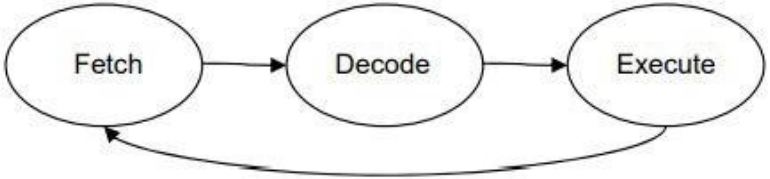


Figure 4.4-(a): Basic FSM of a processor

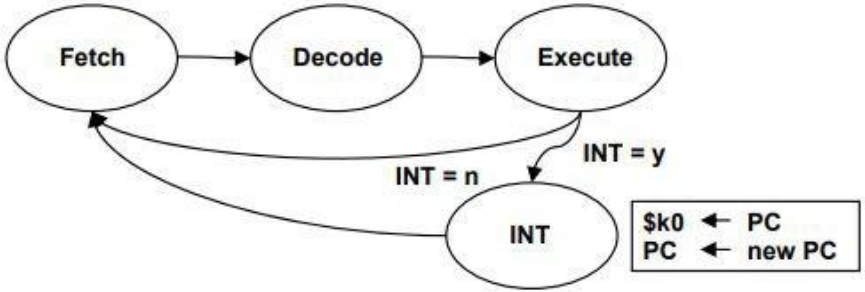
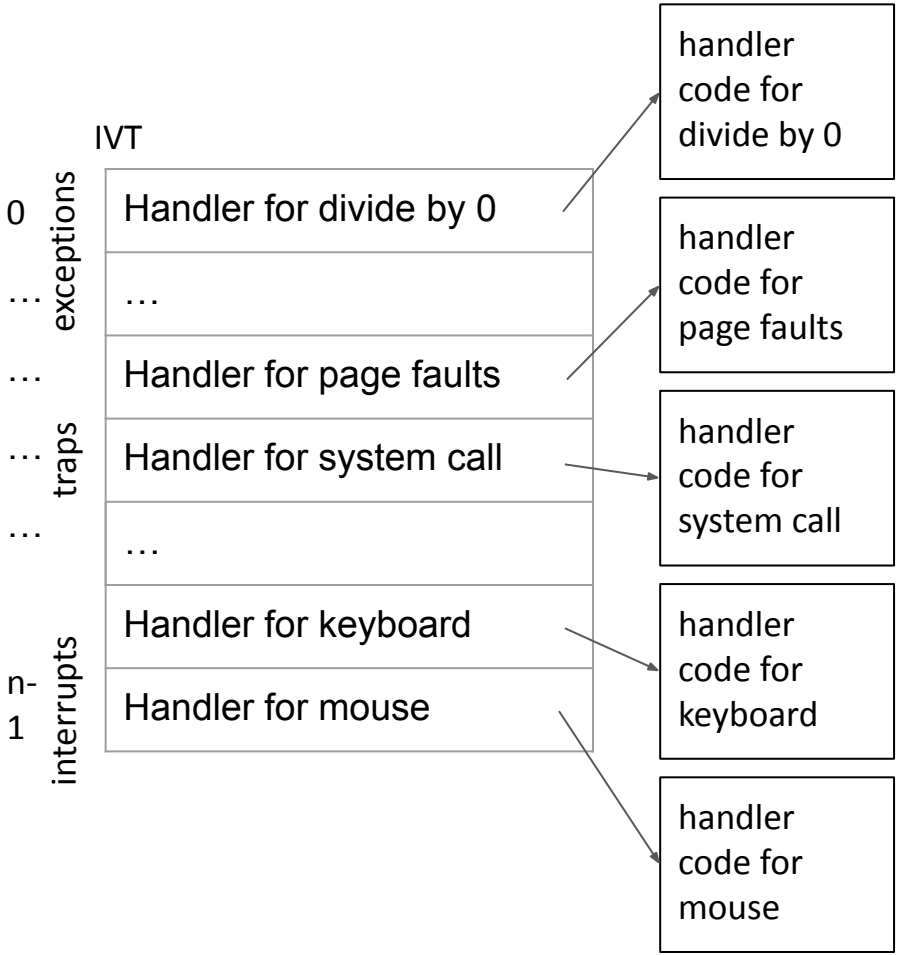


Figure 4.4-(b): Modified FSM for handling interrupts

IVT and Handler

How does it find the interrupt handling code?

- Each discontinuity has a unique number called a vector
- Interrupt Vector Table (IVT) uses the unique number to map it to the associated handler code
  - IVT is a fixed-size table of handler addresses
  - Usually starts at the bottom (0x0) in memory



$k0

- Reserved register for saving the current PC before going to handle the interrupt
  - You want to be able to continue from where you left off
- You can think of it as similar to $ra but for interrupts

Handling multiple interrupts

Exercise:
See any problems here?

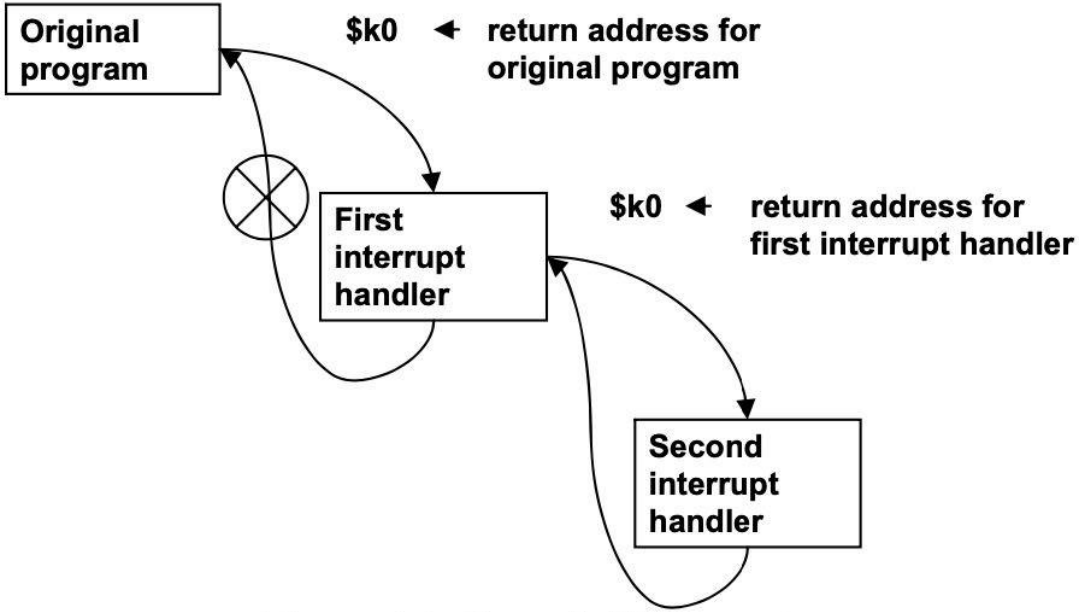


Figure 4.6: Cascaded interrupts


new $k0
old $k0

Let’s save $k0!

Discussion:

What are some possible techniques to make sure we don’t overwrite the old $k0?

Interrupt Handler Steps

Similar steps to function calling convention

- **Save $k0 to stack**
- Save ALL registers onto the stack
- Execute particular handler code
- Restore ALL registers from the stack
- **Restore $k0 from stack**
- Return back to the original program (with JALR)

Interrupt Handler Steps

What if a new interrupt happens while saving or restoring $k0?

- **Save $k0 to stack**
- Save ALL registers onto the stack
- Execute particular handler code
- Restore ALL registers from the stack
- **Restore $k0 from stack**
- Return back to the original program (with JALR)

Think about the race condition…

Exercise: Race Conditions

What are some ways to prevent race conditions when accessing $k0?

Exercise: Race Conditions

What are some ways to prevent race conditions when accessing $k0?

→ Solution 1: Disabling/Enabling Interrupts

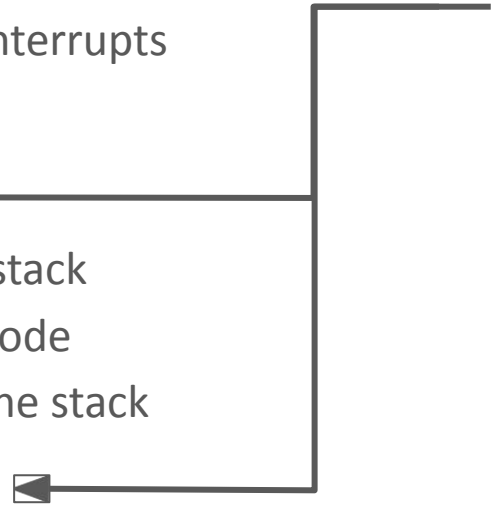
→ Solution 2: Atomic Instructions

Interrupt Handler Steps

What if a new interrupt happens while saving or restoring $k0?

Solution 1: Disabling/Enabling Interrupts

- **Save $k0 to stack**
- **Enable interrupt (EI)**
- Save ALL registers onto the stack
- Execute particular handler code
- Restore ALL registers from the stack
- **Disable interrupt (DI)**
- **Restore $k0 from stack**
- Return back to the original program (with JALR)



New instructions:

- EI (Enable Interrupt)
- DI (Disable Interrupt)

Interrupt Handler Steps

- Save $k0 to stack
- **Enable interrupt (EI)**
- Save ALL registers onto the stack
- Execute particular handler code
- Restore ALL registers from the stack
- **Disable interrupt (DI)**
- Restore $k0 from stack
- Return back to the original program (with JALR)

A

B

C

Exercise:

We need to re-enable interrupts before returning! Where should we do that?

Interrupt Handler Steps

- Save $k0 to stack
- **Enable interrupt (EI)**
- Save ALL registers onto the stack
- Execute particular handler code
- Restore ALL registers from the stack
- **Disable interrupt (DI)**
- Restore $k0 from stack
- Return back to the original program (with JALR)

Exercise:

We need to re-enable interrupts before returning! Where should we do that?

A - EI right after DI is pointless

B - If new interrupt here, we'll lose $k0

C - Will not execute; we have left the interrupt handler and gone back to the original program

Since none of these work, can we use solution #2?

(RETI) Interrupt Handler

- New instruction: RETI
  - Solution # 2
  - Instructions are atomic - because the microcode can't be interrupted, RETI either executes fully or does not execute at all
  - It can do all of the following without getting interrupted:
    - PC ← $k0, Enable Interrupt, Restore Mode (simultaneously!)

New instructions:

- EI (Enable Interrupt)
- DI (Disable Interrupt)
- RETI

Interrupt Handler (finalized) Steps

- **Save $k0 to stack**
- **Enable interrupt (EI)**
- Save ALL registers onto the stack
- Execute particular handler code
- Restore ALL registers from the stack
- **Disable interrupt (DI)**
- **Return to where it interrupted from (RETI)**

New instructions:

- EI (Enable Interrupt)
- DI (Disable Interrupt)
- RETI

Finalized INT + Interrupt Handler Steps

INT macrostate:

- Save the PC to $k0
- Load a new PC value to jump to the interrupt handling code
- Save the mode onto the system stack
- Switch the mode to kernel if not already in kernel mode
- **Disable interrupts (DI)**

RETI:

- Load PC from $k0
- Restore mode from the system stack
- **Enable interrupts (EI)**

Exercise: Jumping To The Right Handler

Take 2 minutes to think:

How can the microcontroller know which device is interrupting and where the handler code address lives in the IVT?

Exercise: Jumping To The Right Handler

How can the microcontroller know which device is interrupting and where the handler code address lives in the IVT?

→ Have the device assert its IVT index onto the bus when it is acknowledged!

- The Fetch/Decode/Execute loop will check for interrupts and transfer to the INT macrostate if an interrupt is detected
- If an interrupt exists, the microcode will acknowledge the first interrupting device in the chain and tell it to assert its index onto the bus

Interrupt Datapath

1. Device asserts INT to convey interrupt pending
2. Processor checks INT bus.
   a. If interrupts are enabled, processor will enter INT macrostate and assert INTA (interrupt acknowledge) onto datapath
3. INTA passes through devices (with daisy chaining) until it reaches one that is interrupting
4. The processor will look up that vector in the IVT and assert the handler's address onto the address bus.
5. The PC will change to the address of the handler.
6. Interrupt Handler will now take over.

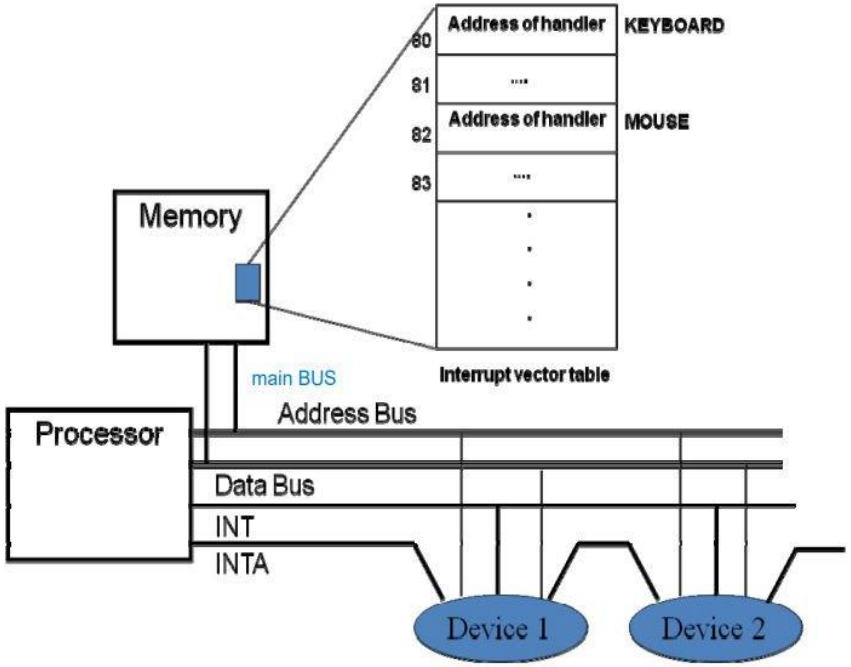


Figure 4.12: Processor-device interaction to receive the interrupt vector

INT macrostate in datapath

- $k0 ← PC
- Asserts INTA
- Receive interrupt vector from device
- handler address = IVT[interrupt vector]
- PC ← handler address
- disable interrupt

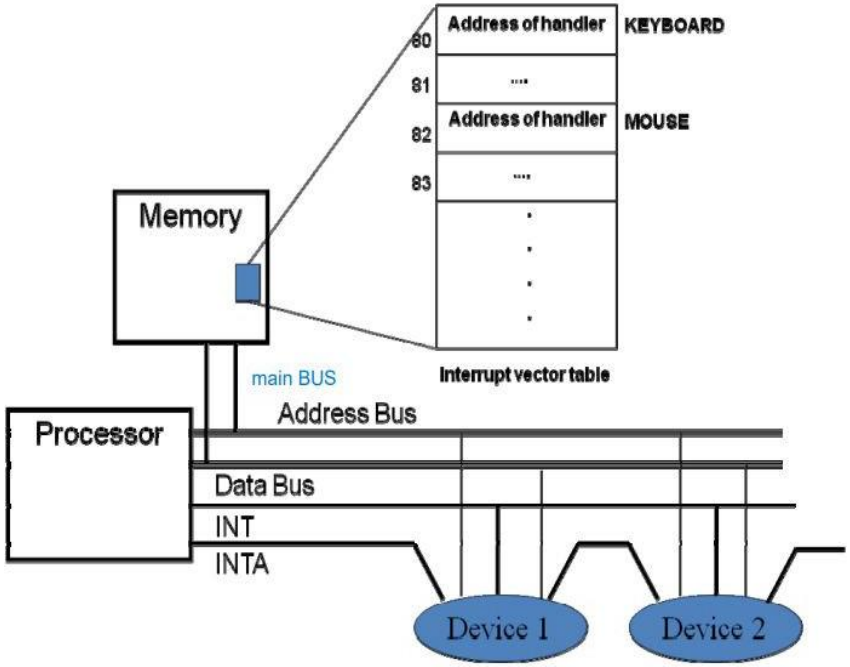


Figure 4.12: Processor-device interaction to receive the interrupt vector

Exercise: Interrupt Datapath

If Device 1 and Device 2 assert the INT bus at the same time, which interrupt will be handled first and why?

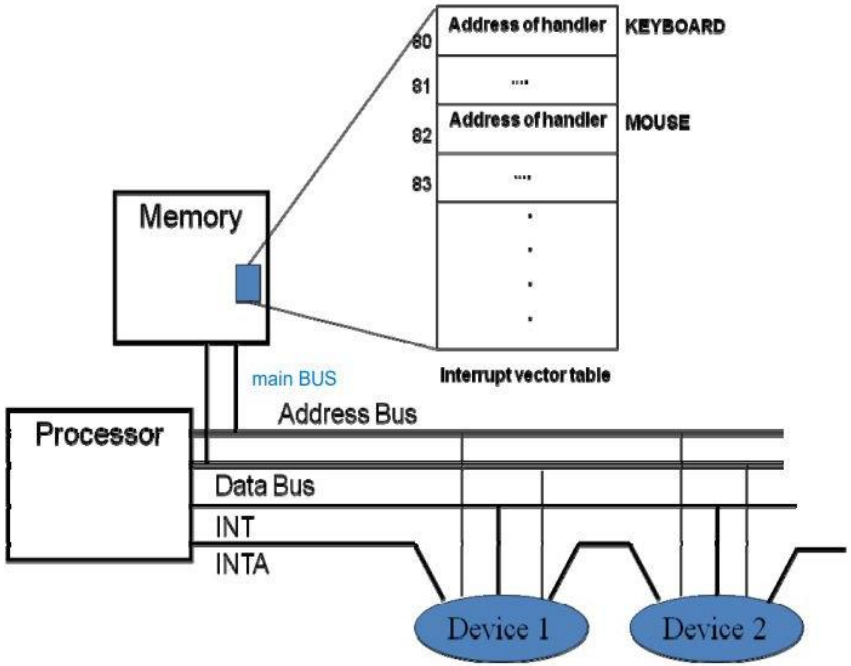


Figure 4.12: Processor-device interaction to receive the interrupt vector

Exercise: Interrupt Datapath

If Device 1 and Device 2 assert the INT bus at the same time, which interrupt will be handled first and why?

⟶ Device 1 will be handled first because of daisy chaining!

Recall: What is daisy chaining?

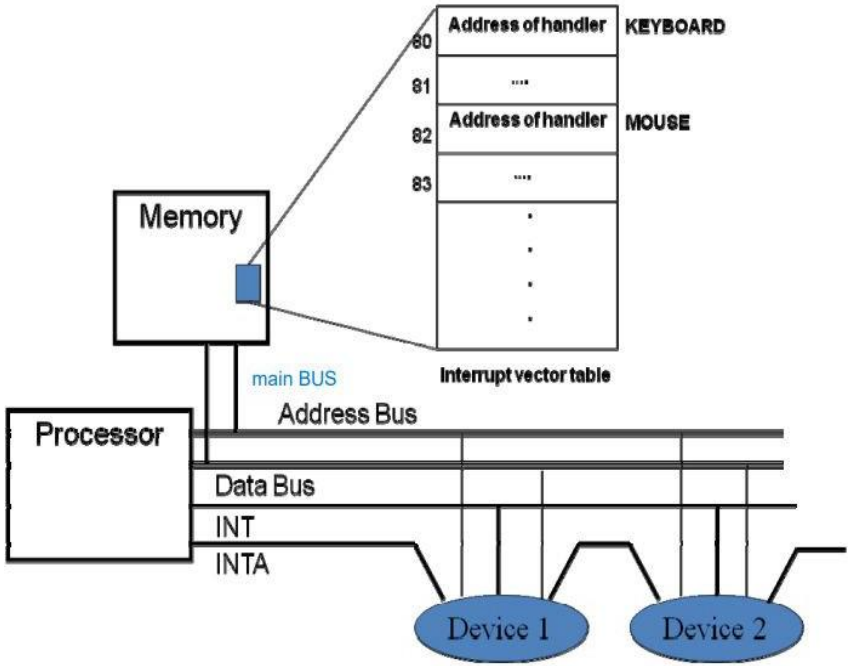


Figure 4.12: Processor-device interaction to receive the interrupt vector

Exercise: Processor Modes

What mode are we in prior to the interrupt handler?

Exercise: Processor Modes

What mode are we in prior to the interrupt handler?

User Mode

What mode are we in during the interrupt handler?

Exercise: Processor Modes

What mode are we in prior to the interrupt handler?

User Mode

What mode are we in during the interrupt handler?

Kernel Mode

What informs us of which mode we are in?

Switch happens in the INT macrostate!

Exercise: Processor Modes

What mode are we in prior to the interrupt handler?

User Mode

What mode are we in during the interrupt handler?

Kernel Mode

What informs us of which mode we are in?

Privileged mode bit in the processor

Why do we need to switch modes?

Exercise: Processor Modes

What mode are we in prior to the interrupt handler?

User Mode

What mode are we in during the interrupt handler?

Kernel Mode

What informs us of which mode we are in?

Privileged mode bit in the processor

Why do we need to switch modes?

For security - Only system level programs should have access

What would happen if a user program tries to use the interrupt handler?

Exercise: Processor Modes

What mode are we in prior to the interrupt handler?

User Mode

What mode are we in during the interrupt handler?

Kernel Mode

What informs us of which mode we are in?

Privileged mode bit in the processor

Why do we need to switch modes?

For security - Only system level programs should have access

What would happen if a user program tries to use the interrupt handler?

Memory protection exception

Exercise: Tracing Stack Mode - What happens to the mode in cascaded interrupts?

For each numbered position in the following diagram, write the mode that the processor is currently in:

1. In the original program
2. In the first interrupt handler
3. In the second interrupt handler
4. After returning back to the first interrupt handler
5. After returning back to the original program

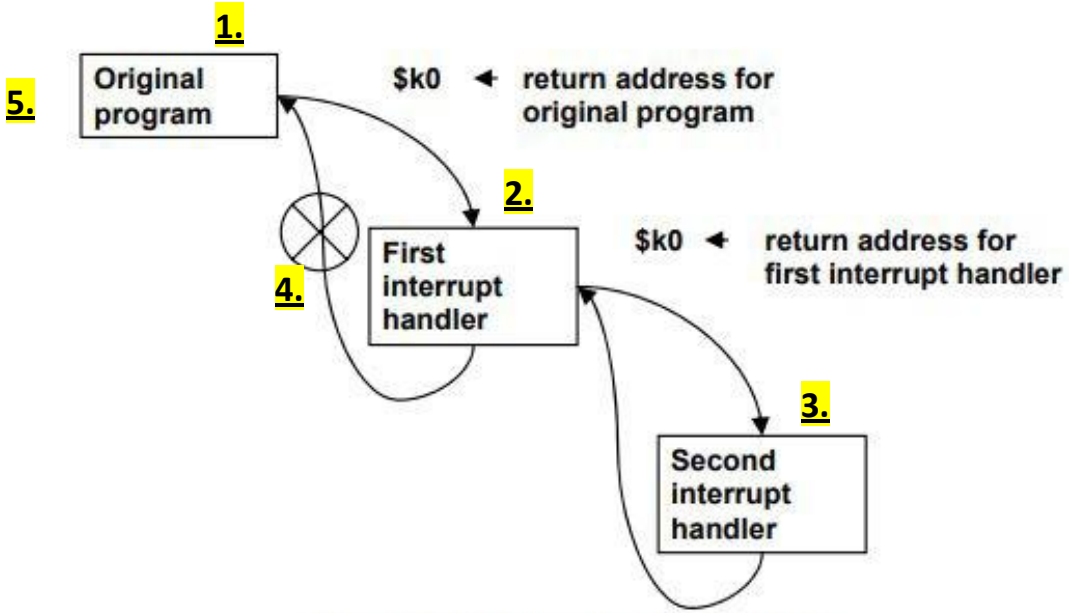


Figure 4.6: Cascaded interrupts

Exercise: Tracing Stack Mode - What happens to the mode in cascaded interrupts?

For each numbered position in the following diagram, write the mode that the processor is currently in:

1. In the original program
2. In the first interrupt handler
3. In the second interrupt handler
4. After returning back to the first interrupt handler
5. After returning back to the original program

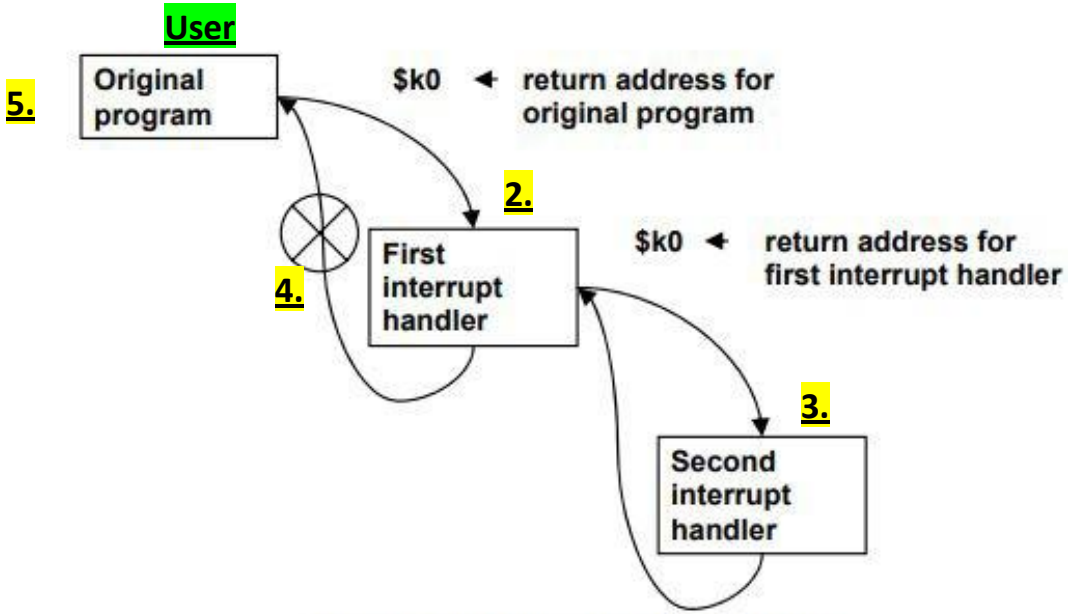


Figure 4.6: Cascaded interrupts

Exercise: Tracing Stack Mode - What happens to the mode in cascaded interrupts?

For each numbered position in the following diagram, write the mode that the processor is currently in:

1. In the original program
2. In the first interrupt handler
3. In the second interrupt handler
4. After returning back to the first interrupt handler
5. After returning back to the original program

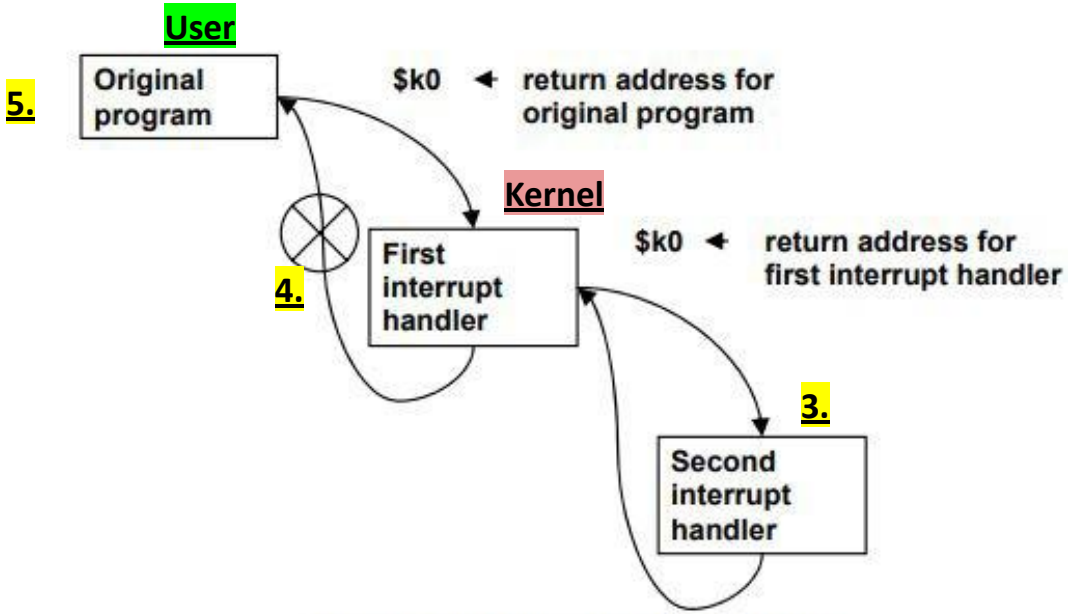


Figure 4.6: Cascaded interrupts

Exercise: Tracing Stack Mode - What happens to the mode in cascaded interrupts?

For each numbered position in the following diagram, write the mode that the processor is currently in:

1. In the original program
2. In the first interrupt handler
3. In the second interrupt handler
4. After returning back to the first interrupt handler
5. After returning back to the original program

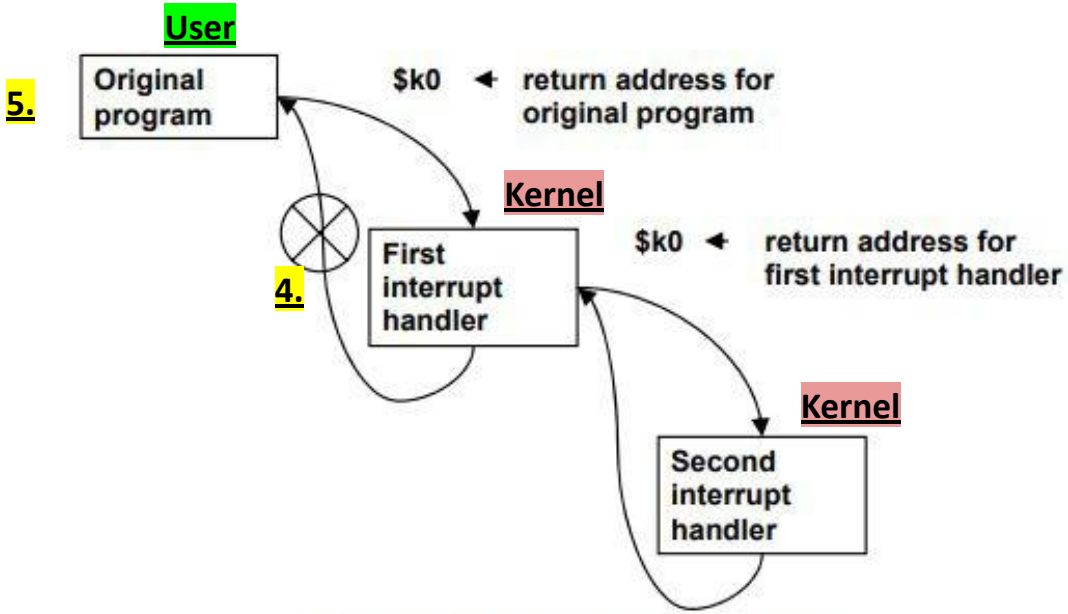


Figure 4.6: Cascaded interrupts

Exercise: Tracing Stack Mode - What happens to the mode in cascaded interrupts?

For each numbered position in the following diagram, write the mode that the processor is currently in:

1. In the original program
2. In the first interrupt handler
3. In the second interrupt handler
4. After returning back to the first interrupt handler
5. After returning back to the original program

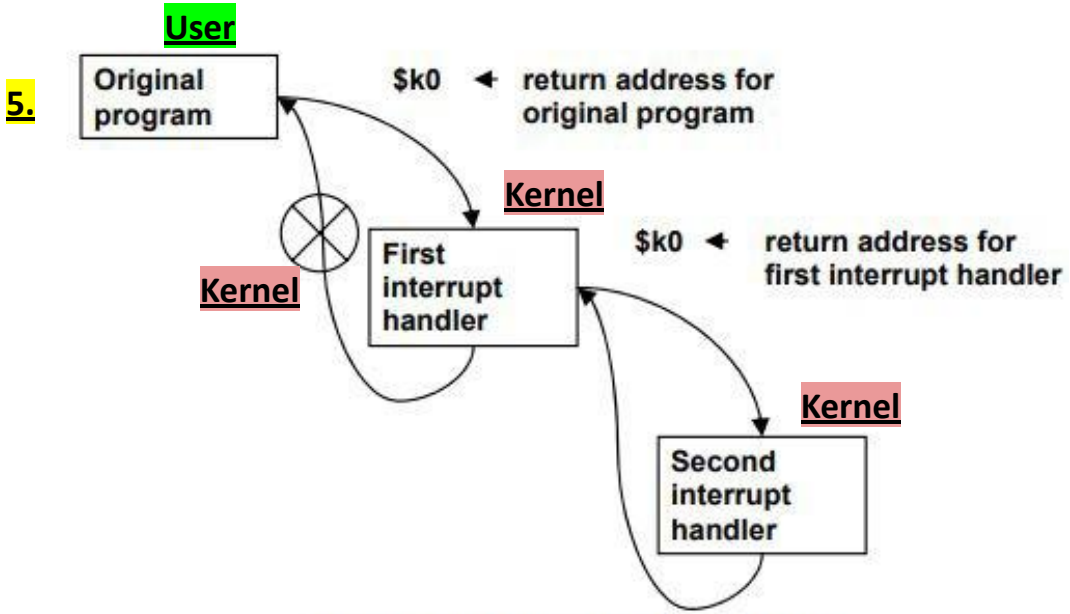


Figure 4.6: Cascaded interrupts

Exercise: Tracing Stack Mode - What happens to the mode in cascaded interrupts?

For each numbered position in the following diagram, write the mode that the processor is currently in:

1. In the original program
2. In the first interrupt handler
3. In the second interrupt handler
4. After returning back to the first interrupt handler
5. After returning back to the original program

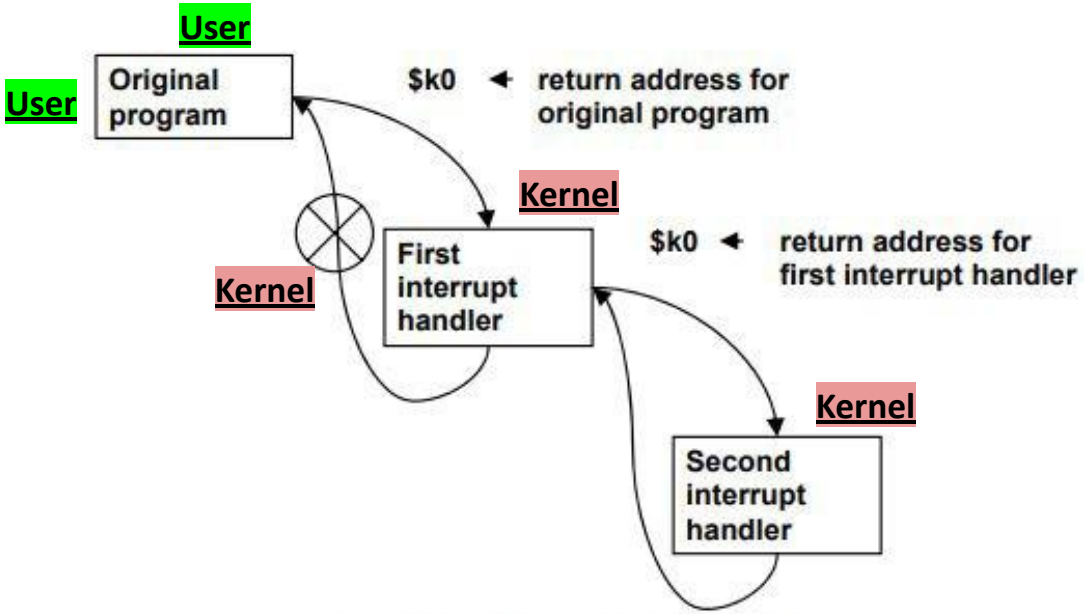


Figure 4.6: Cascaded interrupts

Exercise: Tracing Stack Mode - What happens to the mode in cascaded interrupts?

How do we make sure we set the mode back to user after the last handler?

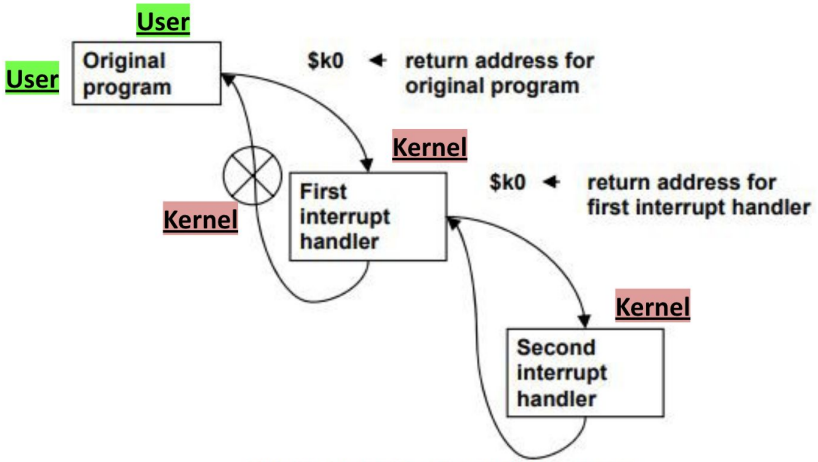


Figure 4.6: Cascaded interrupts

Exercise: Tracing Stack Mode - What happens to the mode in cascaded interrupts?

How do we make sure we set the mode back to user after the last handler?

We store the current mode onto the stack and make sure to restore it in RETI!

Which stack do we save it on and why?

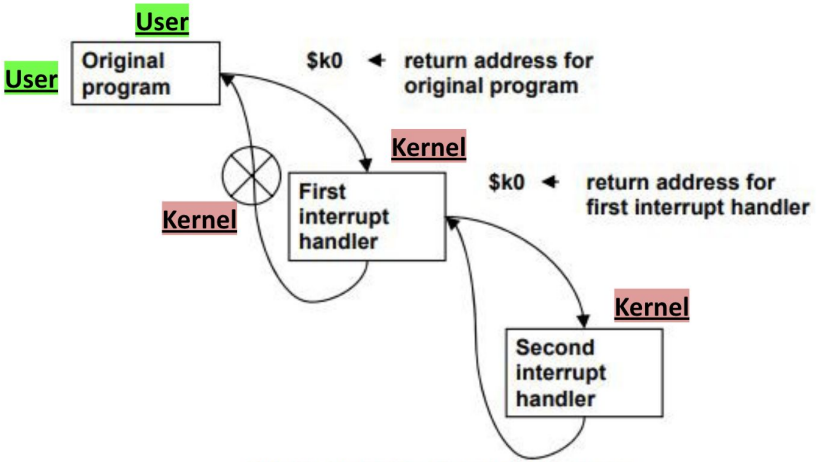


Figure 4.6: Cascaded interrupts

Exercise: Tracing Stack Mode - What happens to the mode in cascaded interrupts?

How do we make sure we set the mode back to user after the last handler?

We store the current mode onto the stack and make sure to restore it in RETI!

Which stack do we save it on and why?

We use the system stack for convenience and safety reasons - we do not want user-level programs to be able to change their mode maliciously.

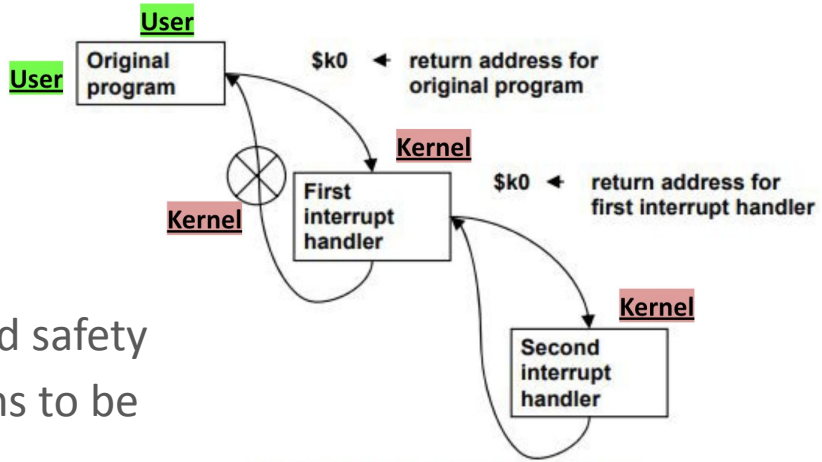


Figure 4.6: Cascaded interrupts

Project 1 Q&A

Today’s Number

4813

Go jackets

Thanks for Attending :)

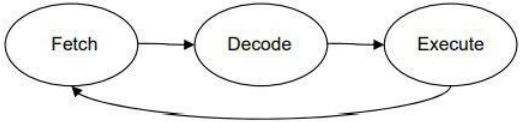


Figure 4.4-(a): Basic FSM of a processor

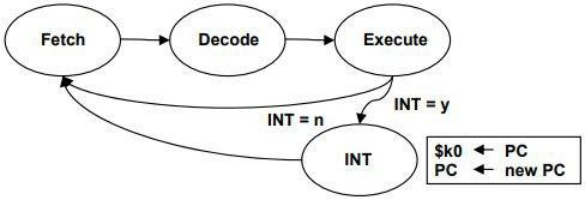


Figure 4.4-(b): Modified FSM for handling interrupts

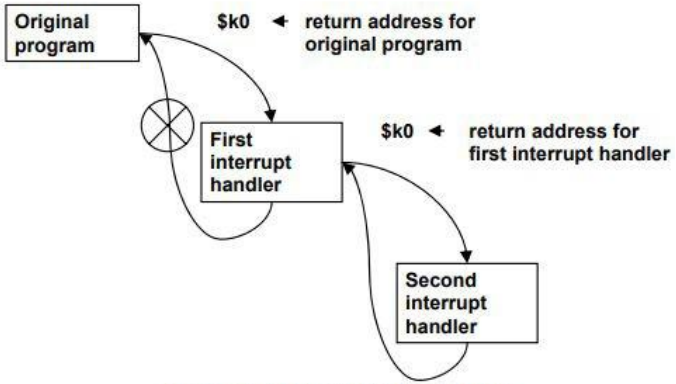
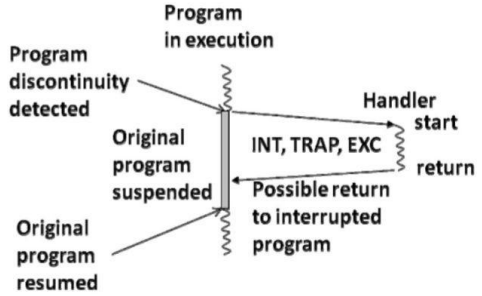


Figure 4.6: Cascaded interrupts



	Interrupts	Traps	Exceptions
Occurrence	Usually due to I/O devices	Allows user to access protected OS resources	Illegal/exceptional behavior in program
Async/Sync	Asynchronous	Synchronous	Synchronous
Intentional?	Yes	Yes and No	Yes and No
Source	External	Internal	Internal

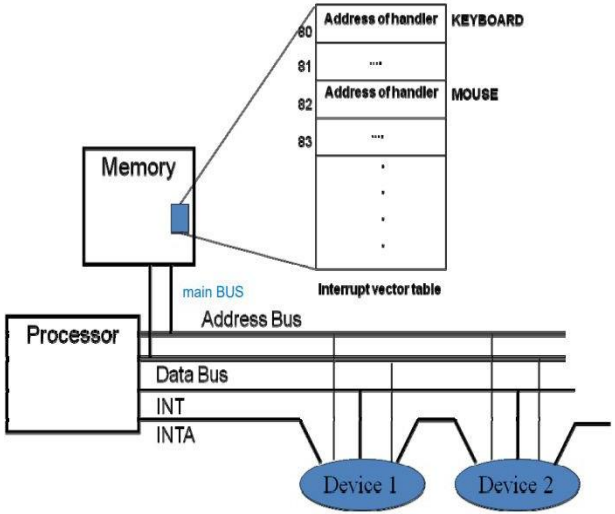


Figure 4.12: Processor-device interaction to receive the interrupt vector

Name	Notation	Units	Comment
Memory footprint	-	Bytes	Total space occupied by the program in memory
Execution time	$(\sum CPI_j) * \text{clock cycle time}$, where $1 \leq j \leq n$	Seconds	Running time of the program that executes *n* instructions
Arithmetic mean	$(E_1+E_2+\ldots+E_p)/p$	Seconds	Average of execution times of constituent *p* benchmark programs
Weighted Arithmetic mean	$(f_1*E_1+f_2*E_2+\ldots+f_p*E_p)$	Seconds	Weighted average of execution times of constituent *p* benchmark programs
Geometric mean	p^{th} root $(E_1*E_2*\ldots*E_p)$	Seconds	p^{th} root of the product of execution times of *p* programs that constitute the benchmark
Harmonic mean	$1/(((1/E_1) +(1/E_2) +\ldots+(1/E_p))/p)$	Seconds	Arithmetic mean of the reciprocals of the execution times of the constituent *p* benchmark programs
Static instruction frequency		%	Occurrence of instruction *i* in compiled code
Dynamic instruction frequency		%	Occurrence of instruction *i* in executed code
Speedup (M_A over M_B)	E_B/E_A	Number	Speedup of Machine A over B
Speedup (improvement)	E_{Before}/E_{After}	Number	Speedup After improvement
Improvement in Exec time	$(E_{old}-E_{new})/E_{old}$	Number	New Vs. old
Amdahl's law	$Time_{after} = Time_{unaffected} + Time_{affected}/x$	Seconds	*x* is amount of improvement