Interrupts, Traps, Exceptions

Terminology

Term	Definition
Program Discontinuity	Any sort of interruption to a program that involves stopping execution on the processor to handle an external event
Synchronous Events	Happen at a well-defined point in time in sync with the processor
Asynchronous Events	Can happen at any time, usually in response to an external event (ex: clicking a link with your mouse)
Types of Discontinuities	Exceptions, Traps, Interrupts

Exceptions

- Response to illegal program behavior
- SEGFAULTS, divide by 0, illegal opcode

Traps

- Allow user program to access otherwise restricted resources managed by the OS
- System calls

Interrupts

- Produced by IO devices
- Mouse clicks, keyboard presses

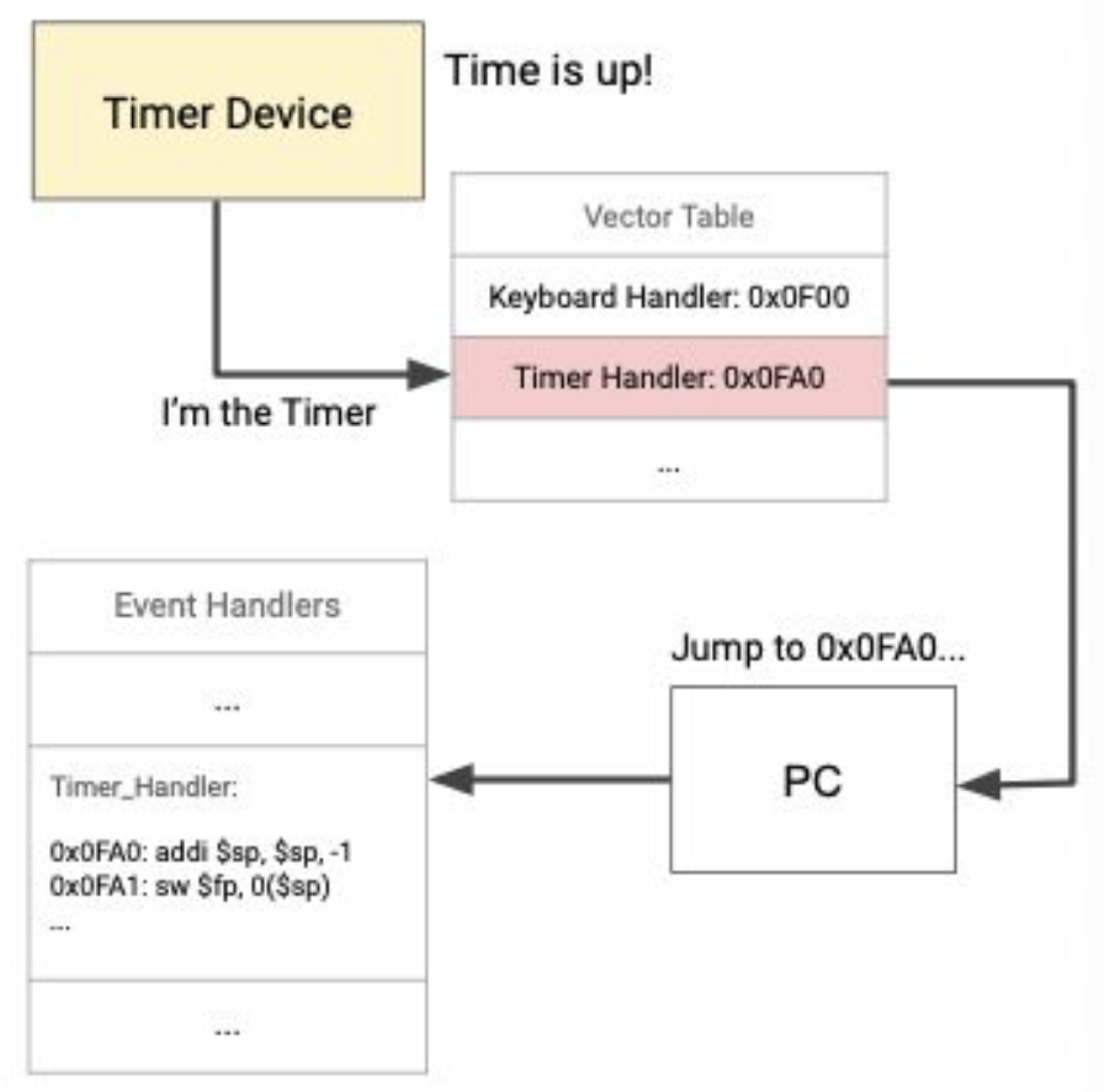
Handling Discontinuities

- OS needs to be able to handle each discontinuity
- Hardware needs to detect discontinuities, and the software needs to be able to handle them
- Jumping from user program to subroutine
  - Hardware detects a discontinuity
  - Hardware jumps to event handling routine
  - Subroutine handles the event
  - Subroutine returns to original program

The Hardware: Interrupt Devices and Steps

Interrupt Vector Table (IVT)

- How does the hardware know where to jump to?
    - Each device has its own device address
    - Exceptions and traps also have their own (but our main focus is interrupts)
- IVT stores 1-1 mapping between device ID and handler address for device interrupts
- OS is responsible for initializing the IVT



Program Discontinuities (FETCH, DECODE, EXECUTE)

- Synchronous events
    - Expected discontinuity
    - Detect exceptions and perform traps in the execute macrostate
- Asynchronous events
    - Unexpected

    - Interrupts can happen at any macrostate
    - How do we make sure our fetch-decode-execute is not interrupt?

The INT Macrostate

We need to check after after execute

- Instructions need to be atomic
- New macrostate: INT (to handle interrupts)

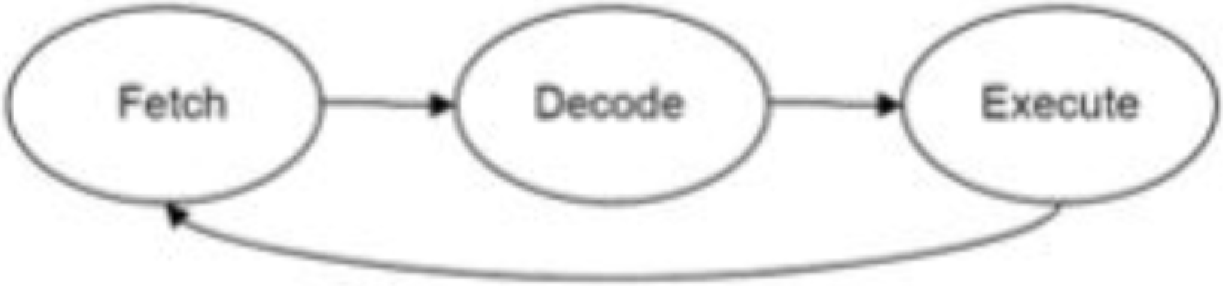


Figure 4.4-(a): Basic FSM of a processor

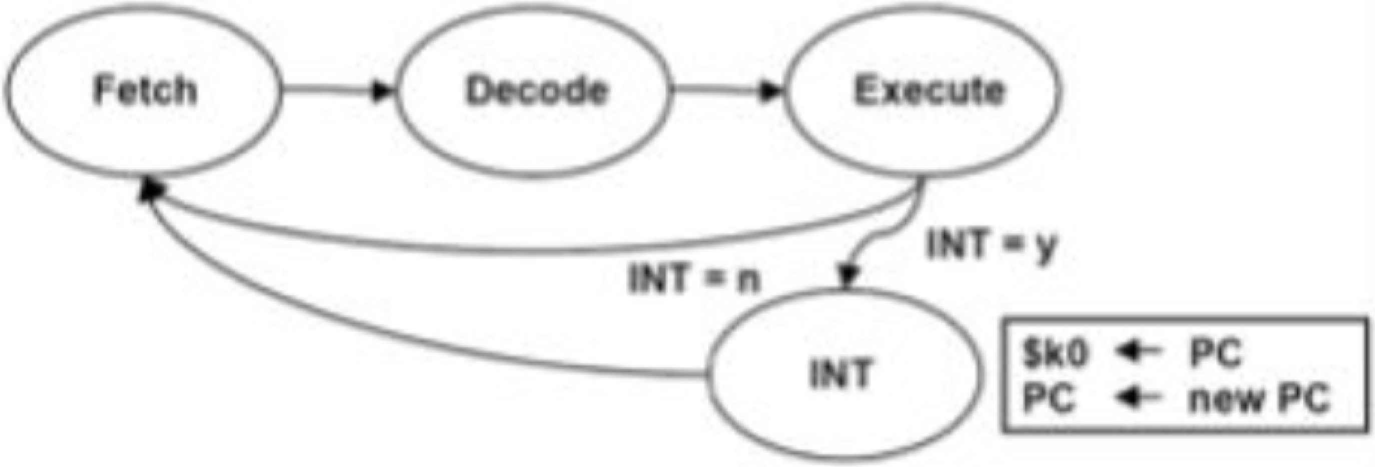


Figure 4.4-(b): Modified FSM for handling interrupts

What does the INT macrostate/hardware need to do?

- Save current PC to $k0
- Receive interrupting device address
- Look at handler address from IVT
- Continue FETCH

Interrupt Devices - Datapath Modifications

- INT line for the device to assert an interrupting signal
    - Remains floating/high impedance if no device is interrupting

- INTA line for the processor to acknowledge the interrupt
    - Devices are daisy chained on the INTA line
    - Devices pass through the INTA signal only if it’s not asserting INT
- Device Data Bus for bi-directional communication
    - Device asserts its ID to the data bus (NOT address bus) upon receiving INTA
- Device Address Bus for addressing the device by the processor

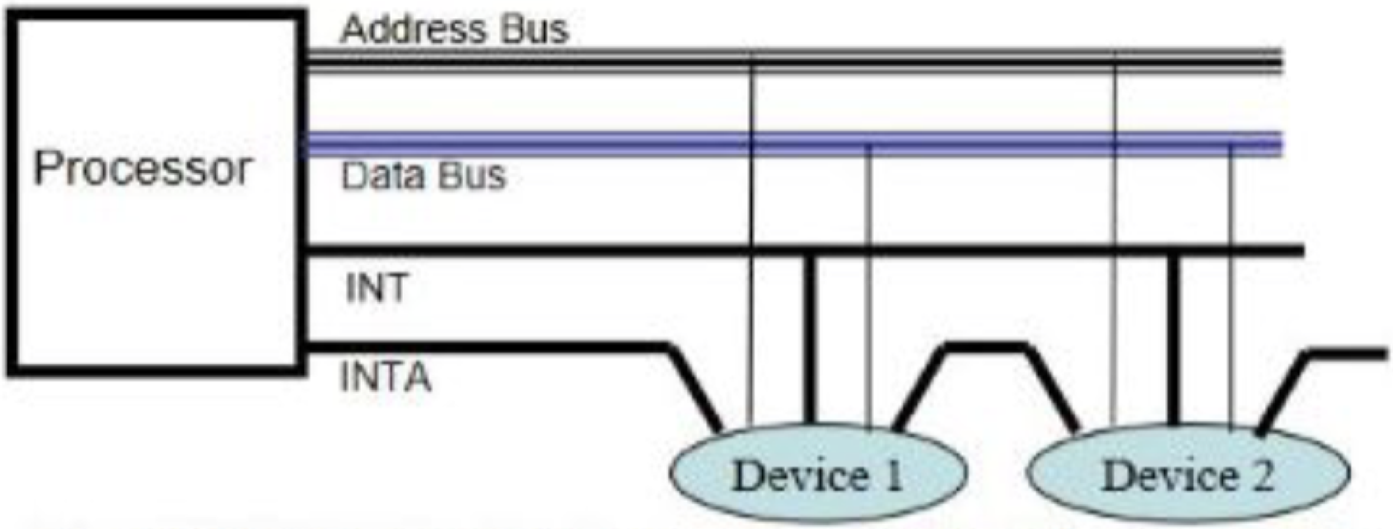
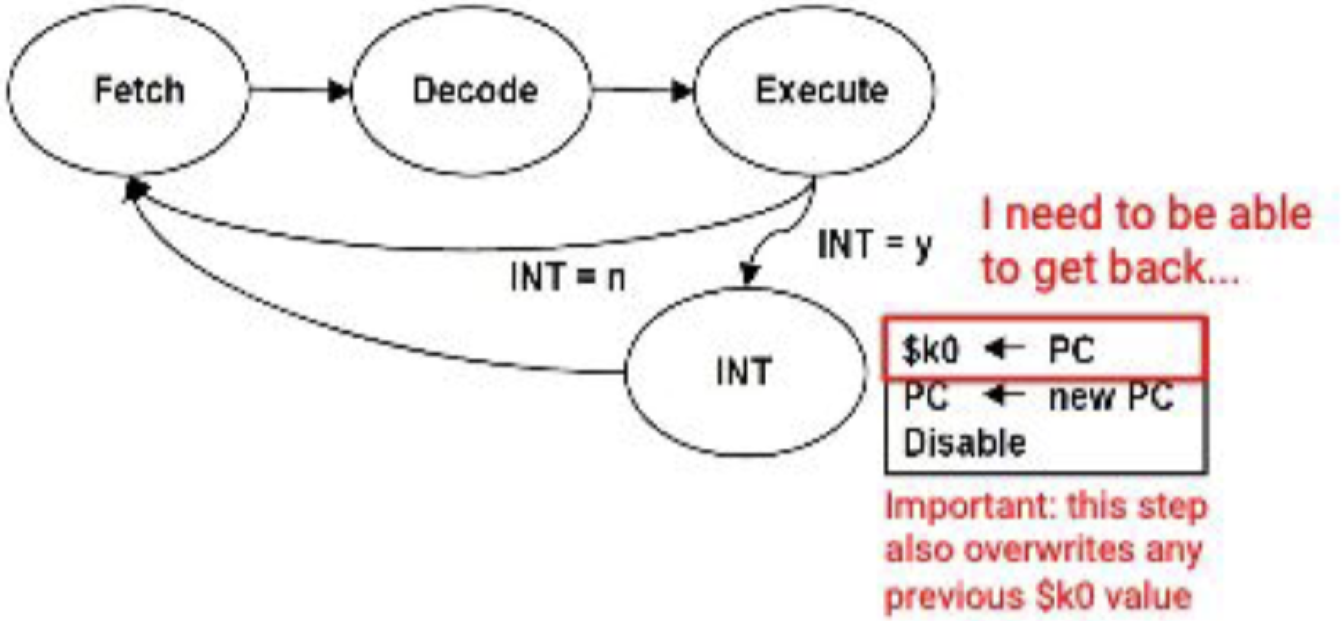
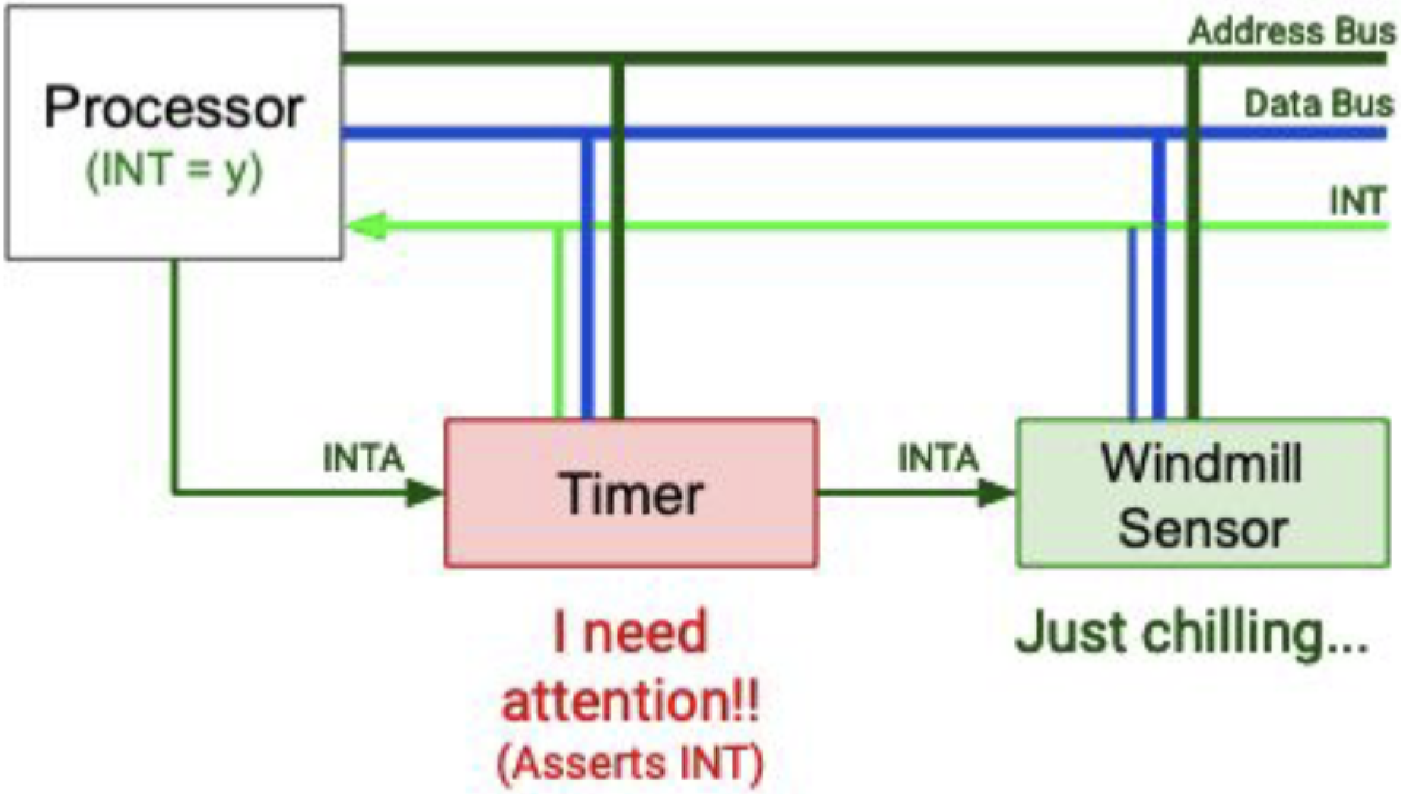


Figure 4.10: Datapath enhancements for handling interrupts

Interrupt Steps

1. Timer asserts INT signal
2. Processor saves current PC to $k0
3. Processor acknowledges with INTA
4. Device drives the data bus with its ID
5. Processor stores device ID to MAR (lookup via IVT)
6. Processor sets new PC from MEM[ID]
7. Processor disables interrupts


Fetch
Decode
Execute
INT = n
INT = y
INT
I need to be able to get back...
$k0 ← PC
PC ← new PC
Disable
Important: this step also overwrites any previous $k0 value



What if Windmill Sensor also needs attention?

- Because the timer device is interrupting, it won't pass the INTA signal to the Windmill Sensor device
- The Windmill Sensor will keep asserting the INT line until it is handled by processor

The Software: Interrupt Handlers

What does the subroutine have to do when interrupted?

- Provide handler code

A couple of questions to ask

- Where is the PC value of the original function, and how do we make sure we can access it after the interrupt has been handled?
- What if there's another interrupt during our interrupt?
- What happens to all our processor registers when we jump to a different program entirely?

List of things the handler needs to do

1. Enable and disable the interrupts at the correct time
2. Execute handler code
3. Save and restore registers to and from the stack
4. Return to the original program safely

New Instruction - RETI

We need to turn on interrupts again after we completed handling it. How can we ensure that we return back to the program at the right PC value AND that the program can be interrupted again?

RETI (Return-from-interrupt/ return and enable interrupts)

- Enable interrupts
- PC <= $k0

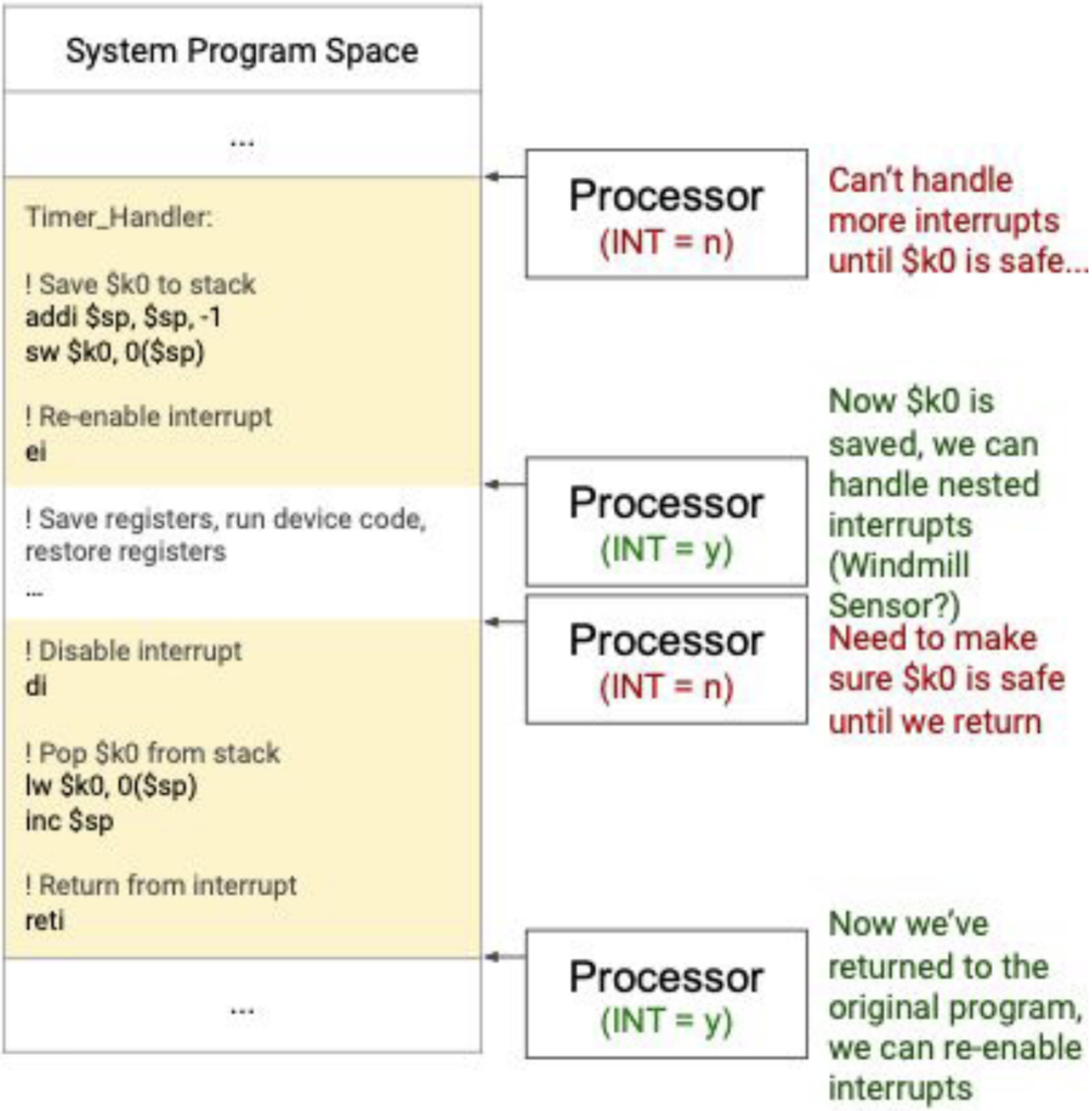
Also, we don't want to save registers on the user stack since we might overwrite/lose some important information! Thus, we have our own stack

- Each gets its own register (USP for users, SSP for system)
- On INT, switch to the system stack and save the previous execution mode on the stack
- During RETI, pop this saved mode off the stack and restore it to the processor

Handling Interrupts – the whole shebang

1. Save $k0 on the stack
2. Enable interrupts (EI)
3. Save ALL processor registers on the stack
4. Execute handler code
5. Restore ALL processor registers from the stack
6. Disable interrupts (DI)
7. Restore the value of $k0 from the stack

8. Return to the original program (with RETI)



```
Timer_Handler:

! Save $k0 to stack
addi $sp, $sp, -1
sw $k0, 0($sp)

! Re-enable interrupt
ei

! Save registers, run device code,
restore registers
...

! Disable interrupt
di

! Pop $k0 from stack
lw $k0, 0($sp)
inc $sp

! Return from interrupt
reti
```

Can't handle more interrupts until $k0 is safe...

Now $k0 is saved, we can handle nested interrupts (Windmill Sensor?)

Need to make sure $k0 is safe until we return

Now we've returned to the original program, we can re-enable interrupts

Performance Metrics

Metrics Intro

- Need to quantify improvements in various aspects of our system to optimize it
    - How can we measure how well our datapath is doing, and how can we use that information as a benchmark for when we make our datapath better?
- Space metrics = how much memory a program uses
- Time metrics = how much time a program has to execute

Space-Time Relationship

- Very large program where only a few lines are actually executed (think conditionals)
    - Bad space
    - Good time
- For loop with not many lines of code that runs a million times
    - Good space
    - Bad time

Definitions

- Execution time - cycles per instruction

The total time a program takes to execute

- Each Instruction takes a certain number of cycles to execute - this is the Cycles Per Instruction (CPI)
- With n instructions the Execution Time is

$$Execution\ time\ = \left(\sum_{j=1}^{n} CPI_j \cdot Clock\ Cycle\ Time\right)$$

This is a difficult calculation - imagine if you have millions of instructions!

- Approximate using Average CPI

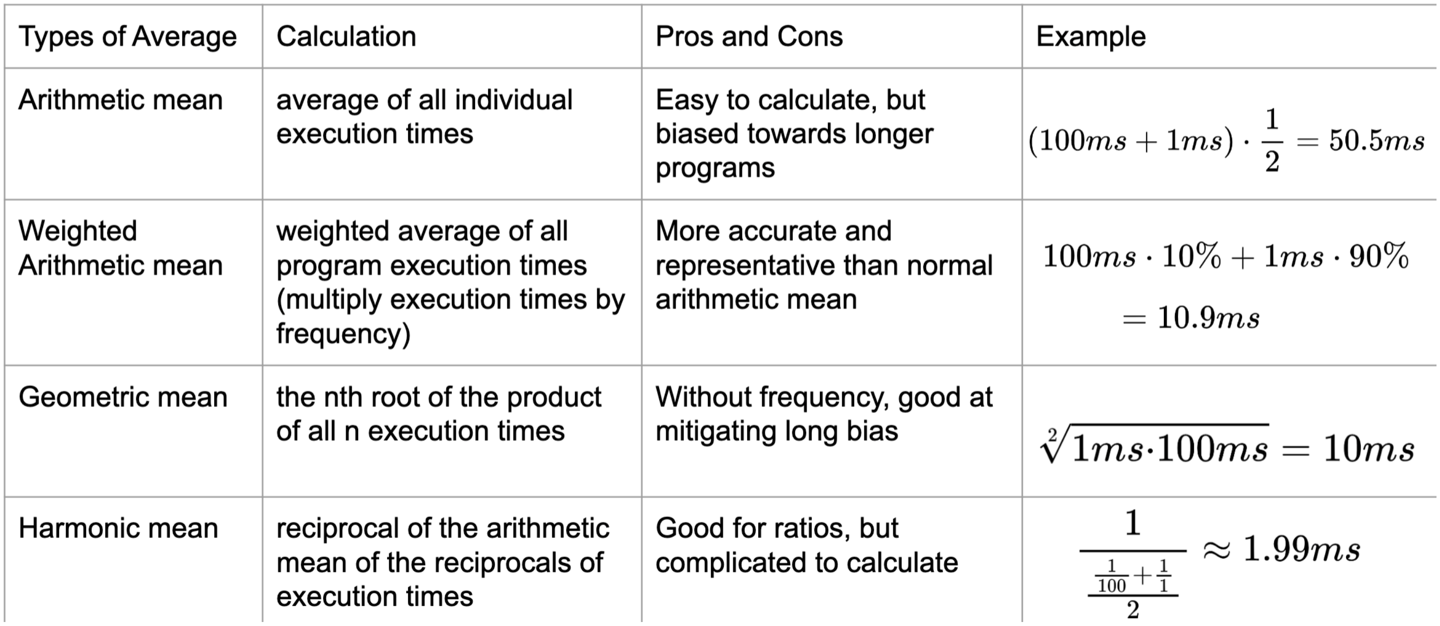
$$Execution\ time = n \cdot CPI_{avg} \cdot clock\ cycle\ time$$

- Static Instruction Frequency

Ways to measure Instruction

- Static instruction frequency - number of times the instruction appears in compiled code
- Dynamic instruction frequency - the number of time the instruction is executed

- Dynamic Instruction Frequency



Types of Average	Calculation	Pros and Cons	Example
Arithmetic mean	average of all individual execution times	Easy to calculate, but biased towards longer programs	$(100ms + 1ms) \cdot \frac{1}{2} = 50.5ms$
Weighted Arithmetic mean	weighted average of all program execution times (multiply execution times by frequency)	More accurate and representative than normal arithmetic mean	$100ms \cdot 10\% + 1ms \cdot 90\%$ $= 10.9ms$
Geometric mean	the nth root of the product of all n execution times	Without frequency, good at mitigating long bias	$\sqrt[2]{1ms \cdot 100ms} = 10ms$
Harmonic mean	reciprocal of the arithmetic mean of the reciprocals of execution times	Good for ratios, but complicated to calculate	$\frac{1}{\frac{\frac{1}{100}+\frac{1}{1}}{2}} \approx 1.99ms$

- How to speedup?

Ways to speedup

- Decrease Clock Cycle
- Reorganize Datapath to reduce CPI
- Reduce number of executed instructions

Speedup formula lets us quantify & compare these changes

- We can compare a program across two processors, or before and after an improvement to the same processor (these are the same formula, different contexts)

$$Speedup_{AoverB} = \frac{Execution\ time\ on\ Processor\ B}{Execution\ time\ on\ Processor\ A}$$

$$Speedup_{improved} = \frac{Execution\ time\ Before\ Improvement}{Execution\ time\ After\ Improvement}$$

- Improvement in Execution Time

$$Improvement = \frac{old\ time - new\ time}{old\ time}$$

- Amdahl's Law

Measures how much a change improved our execution time, but considers how much of the total time was actually affected by the improvement

$$Time_{after} = Time_{unaffected} + Time_{affected}/x$$