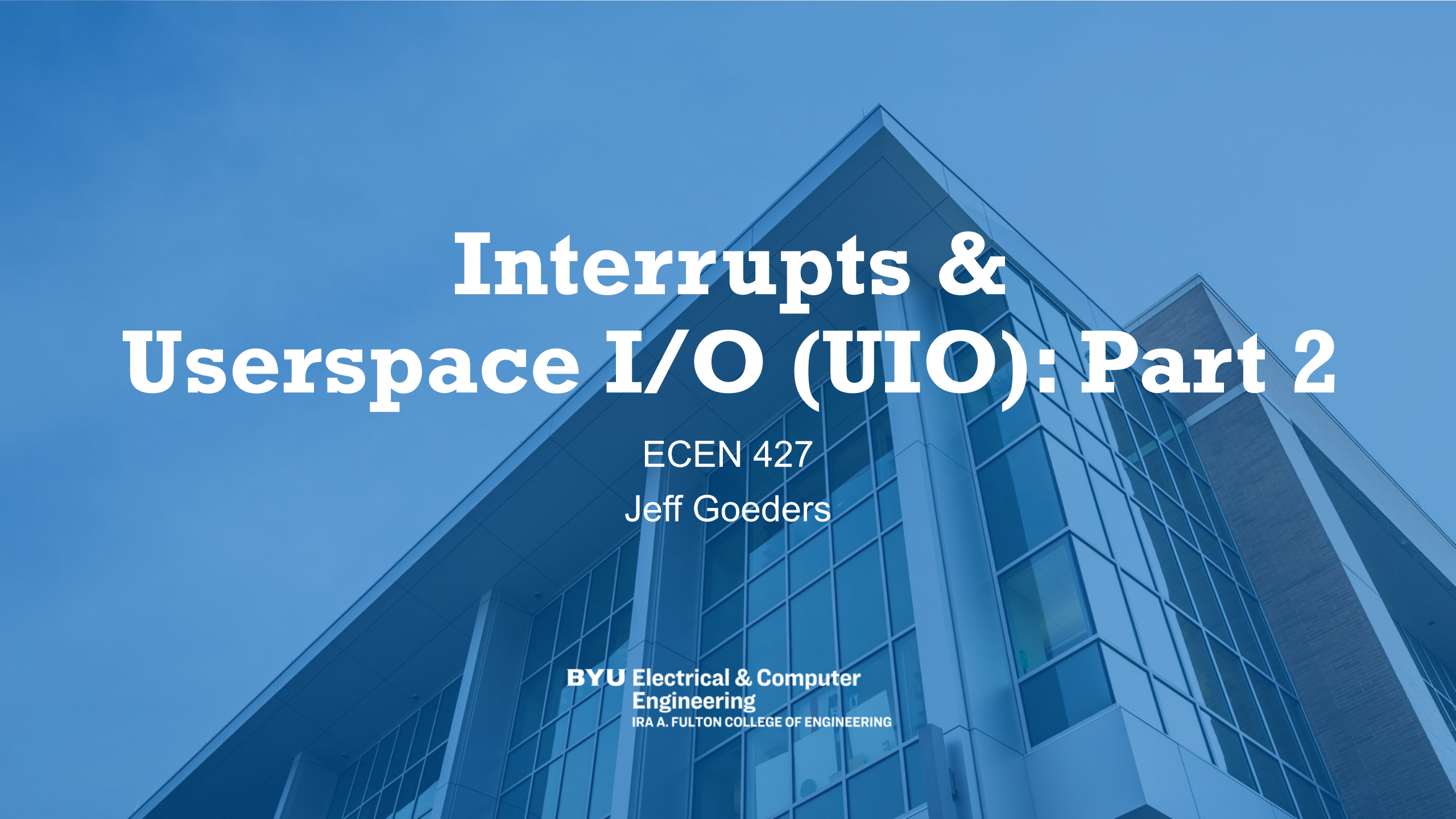


# Interrupts & Userspace I/O (UIO): Part 2

ECEN 427

Jeff Goeders

BYU Electrical & Computer Engineering
IRA A. FULTON COLLEGE OF ENGINEERING

# Learning Objectives



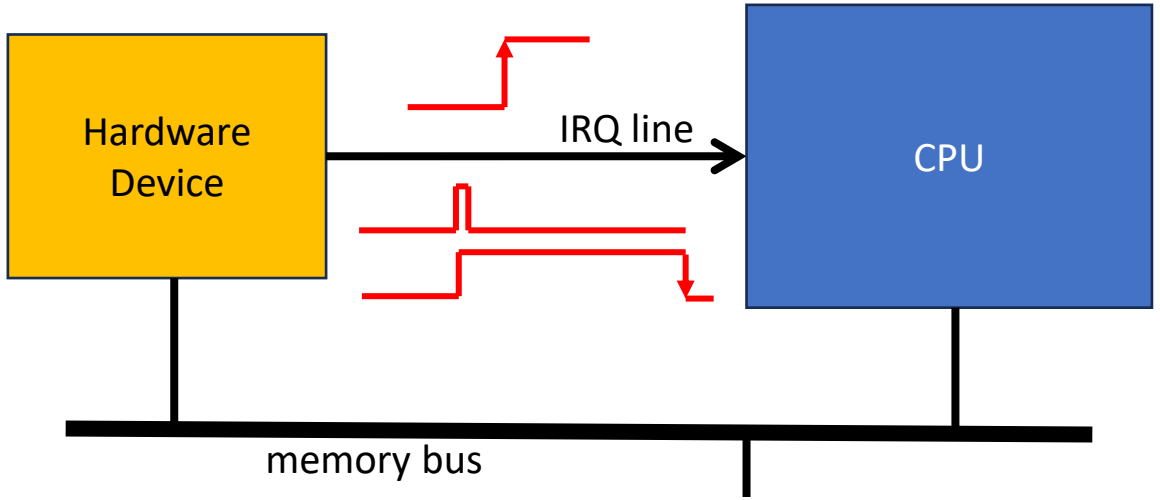
By the end of this lecture, you should be able to:

- Explain what an interrupt is and why interrupt-driven I/O is preferable to polling
- Describe the role of an interrupt controller and how the CPU identifies which device raised an interrupt
- Explain why interrupts cannot be handled in user space, and how UIO uses blocking read() and the process scheduler to approximate it
- Write user space code that enables, waits for, and re-enables UIO interrupts using write(), read(), and poll()

# What is an interrupt?

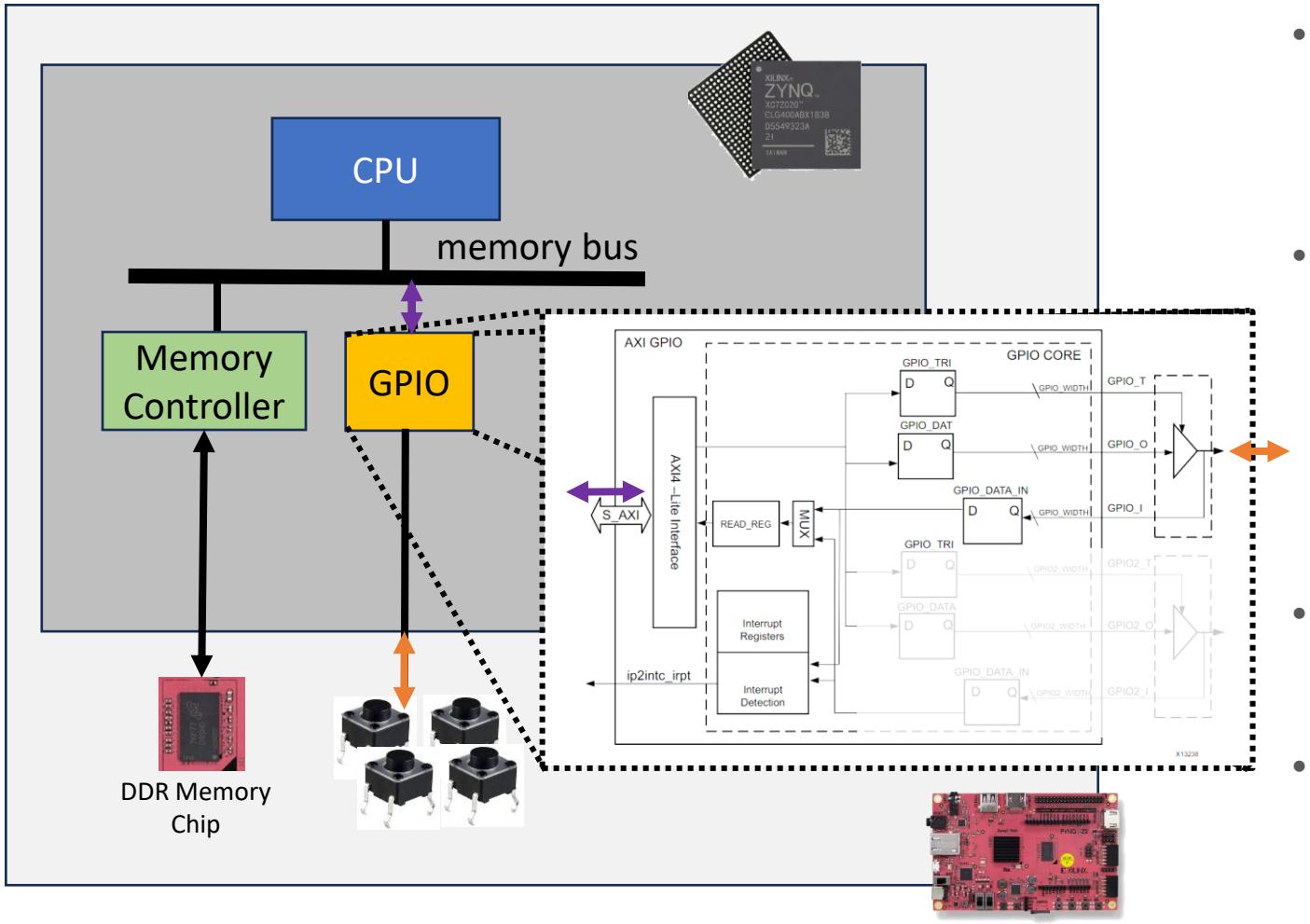


- An **interrupt** is a signal from a device to the CPU indicating that the device needs attention.
- There is a physical wire connecting the device to the CPU. This is an interrupt request line, or **IRQ line**.
- Typically, this device would be connected to the bus, so that the CPU could then interact with the device and get more information about why it sent an interrupt
- Some devices may send a short pulse.
- Others may keep interrupt high until CPU **acknowledges** the interrupt (via register write)



# What happens when the user pushes a button?






- Wire from button feeds into a single flip-flop of the GPIO_DATA_IN register (this register has one flip-flop for each input wire)
- When CPU performs **read** on offset 0x000 (GPIO_DATA), it is given the value of the GPIO_DATA_IN flip-flops.

| Address Space Offset[3] | Register Name | Access Type | Default Value | Description |
|---|---|---|---|---|
| 0x0000 | GPIO_DATA | R/W | 0x0 | Channel 1 AXI GPIO Data Register. |
| 0x0004 | GPIO_TRI | R/W | 0x0 | Channel 1 AXI GPIO 3-state Control Register. |
| 0x011C | GIER[1] | R/W | 0x0 | Global Interrupt Enable Register. |
| 0x0128 | IP IER[1] | R/W | 0x0 | IP Interrupt Enable Register (IP IER). |
| 0x0120 | IP ISR[1] | R/TOW[2] | 0x0 | IP Interrupt Status Register. |

- The *Interrupt Detection* circuitry is watching this GPIO_DATA_IN register to see when any bit changes.
- If configured to do so, the IRQ line (ip2intc_irpt) will be raised, and stay raised until acknowledged.

Note: As we talk about “registers”, and how software meets hardware, we should recognize that the term **register** can refer to both:

- Logical registers at a specific address. May be called *memory-mapped*, *architectural*, *software-visible*, or *programmer-visible* registers.
- Physical registers in the circuitry (*flip-flop*, or *storage element*)

# Why use interrupts?



- If we want to know when a user pressed a button, we could repeatedly check the GPIO_DATA register in a loop.
- This is called **polling**.

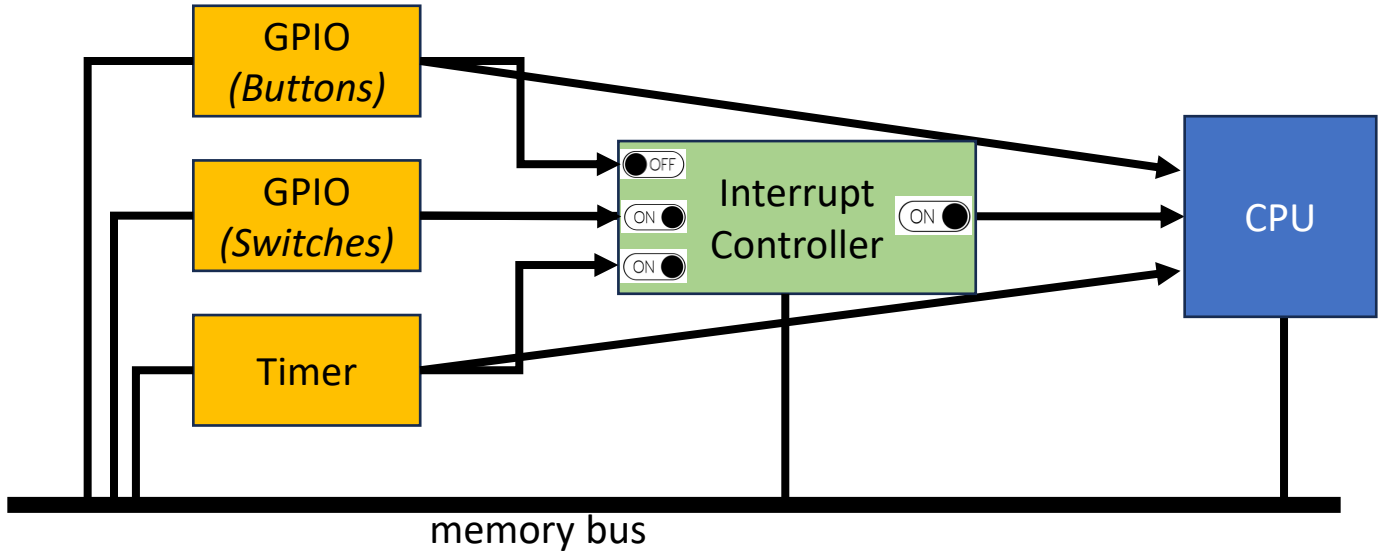
Example solns/userspace/apps/buttons_polling/main.c

Run and look at top

# Interrupt Controllers



- A system likely has many devices that generate interrupts. However, a CPU has limited IRQ inputs.
- An **interrupt controller** aggregates multiple interrupt lines
  - When an interrupt is detected on an IRQ input, it raises its IRQ output
- An interrupt controller can typically:
  - Enable/disable individual inputs
  - Enable/disable its output
- How does the CPU know which device generated an input?
  - The interrupt controller is on the bus, so the CPU can talk to it
- The device that caused the interrupt is also on the bus, so the CPU can in turn talk to the originating device.



# Lab 2: M2

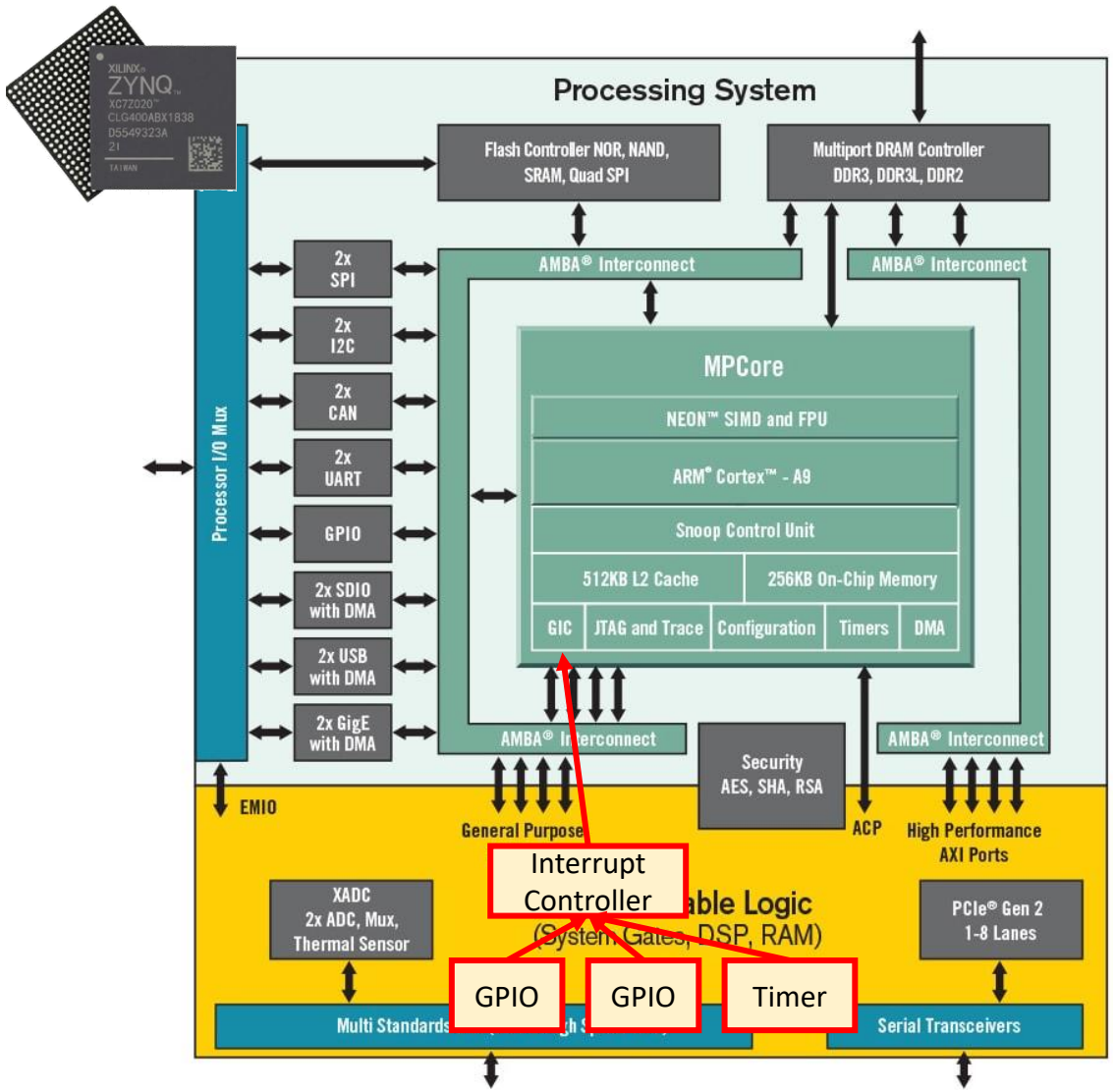


- In Lab2 M2, you will write a driver for an interrupt controller

- To keep things simple, this will again be a *user space driver*, interfacing with the UIO driver.

# Zynq-7000 Interrupts






- The Zynq-7000 has a built-in interrupt controller, the *Global Interrupt Controller (GIC)*
  - This is managed by the Linux kernel
- To give you experience using an interrupt controller, we have added one to the FPGA fabric.
- This feeds into the GIC (cascading interrupt controllers are possible on complex systems)

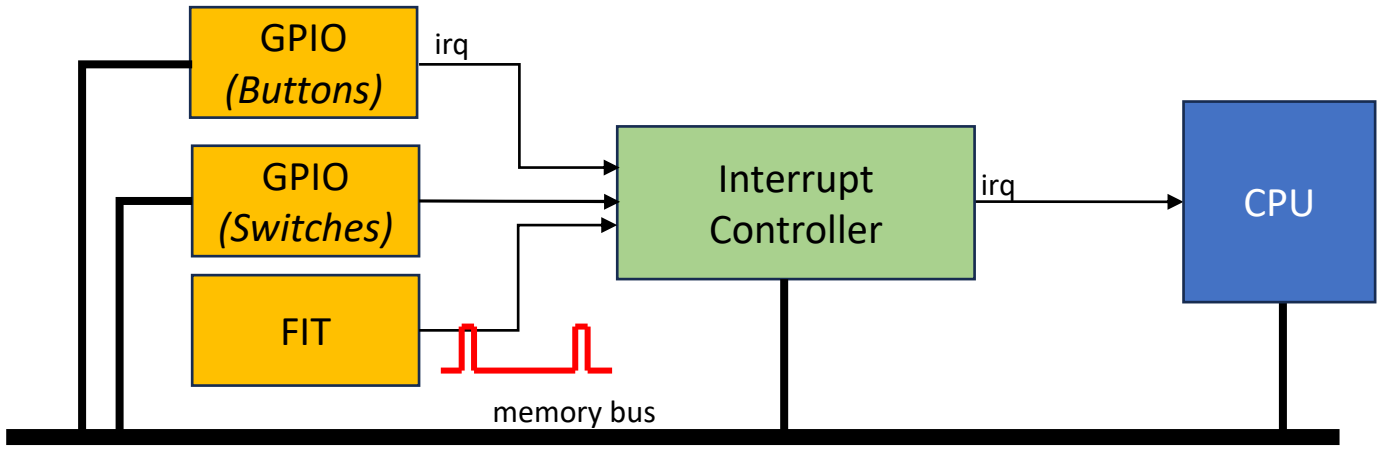
# Interrupts & UIO

# Our Hardware System



This portion of the hardware system is applicable to Lab 2

- (There's also another GPIO block for the LEDs, but not shown here)



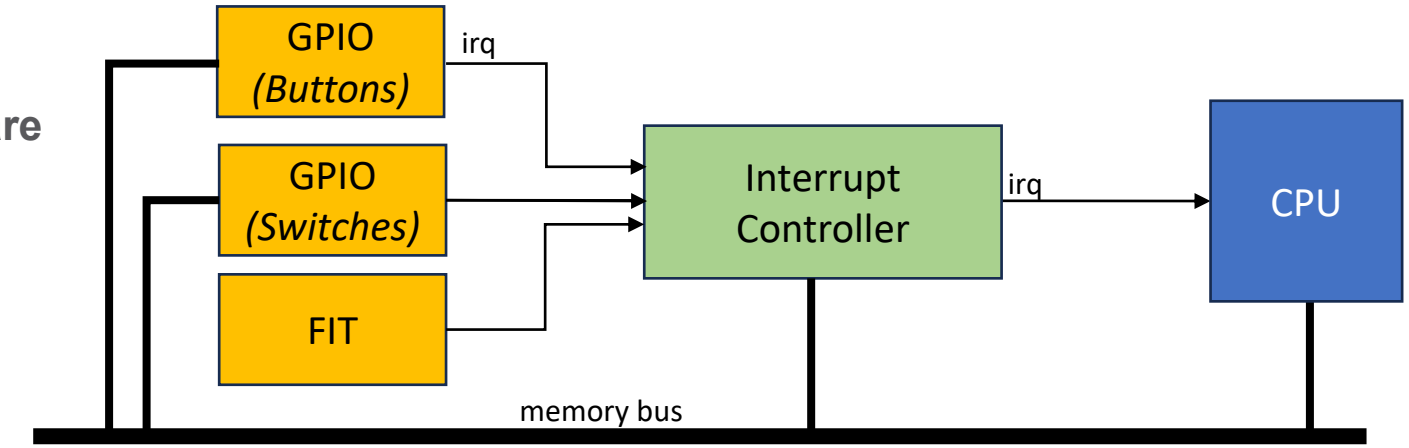

The FIT is a Fixed Interval Timer

- Generates an interrupt pulse at a pre-determined interval (30Hz / 16.67ms)
- Not on the CPU bus. No way to change period at runtime. It's baked into the hardware.

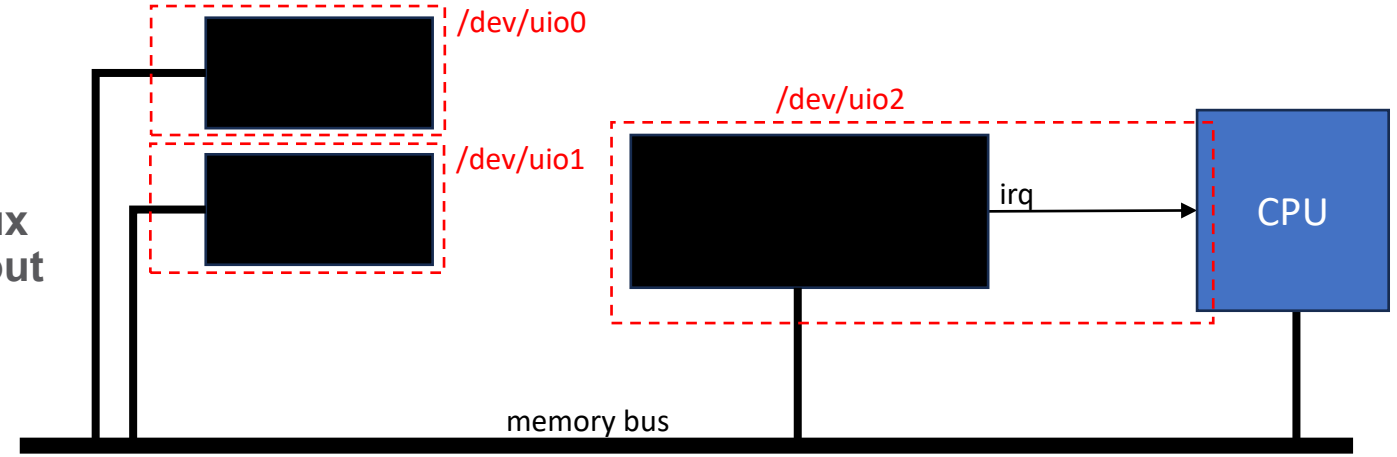
# What Linux Knows About Our Hardware



**Our hardware system**




**What Linux knows about**



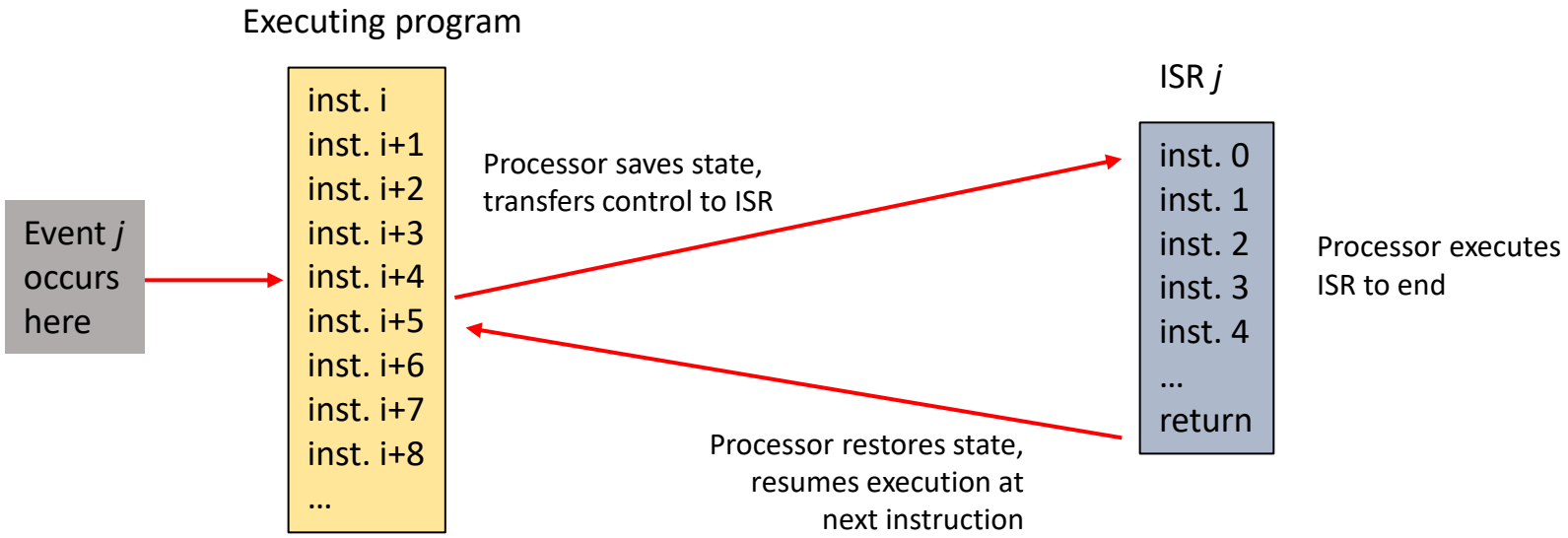

- Linux knows nothing about our hardware.
  - Each device is a black box, managed by the UIO driver.
  - It doesn't know that the interrupt controller is an interrupt controller
  - It doesn't even know about the FIT
- Each device is just "something on the bus"
  - UIO allows **mmap()**
- The interrupt controller is unique, it has an interrupt line in the CPU
  - UIO provides some more syscalls to help manage how Linux responds to an interrupt from this device.
- Linux doesn't know about the interrupt lines between the devices we manage.

What tells Linux about these devices? The **device tree** (we will learn about later in the course)
https://byu-cpe.github.io/ecen427/documentation/platform-device-tree/

# What happens when a CPU is interrupted?



- The processor responds by:
  - Saving the state of the code that was running
  - Executing an **interrupt service routine**
  - Resuming original execution




- **Problem:** There is no mechanism to have ISR code in user space.
  - Interrupts must be handled by kernel code

If we can't have interrupt routines in user space, does that mean we are forced to do polling?

No!

With the help of the UIO driver, and the process scheduler, we can do better…

# Process States



- In the first lecture we talked briefly about processes, and how the operating system virtualizes the CPU in order to run multiple processes.
- The OS scheduler is responsible for selecting which process will run in the next **time slice**.
- Sometimes processes "block" and need to wait for an OS operation to complete.
  - Example: process calls `read()` to get data from a file on disk. This can take several milliseconds.
  - Rather than blocking all execution, the process goes to sleep, and the OS can wake it back up when the data is ready.

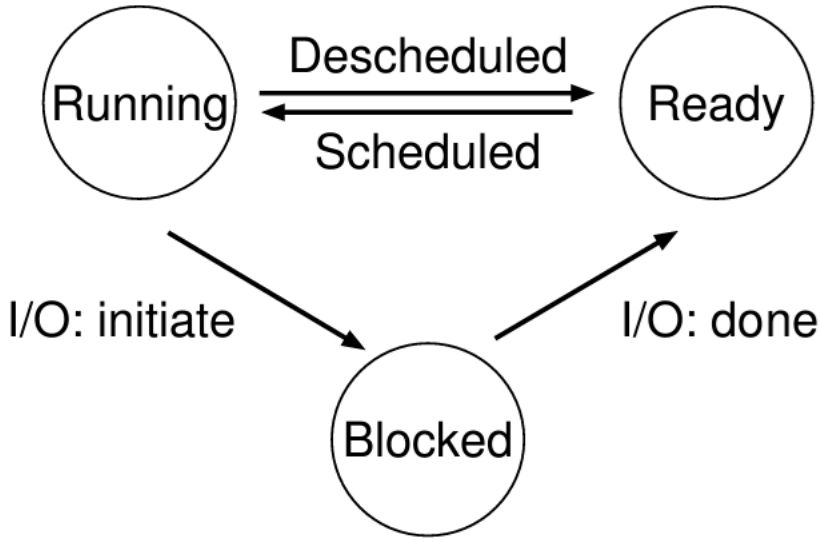


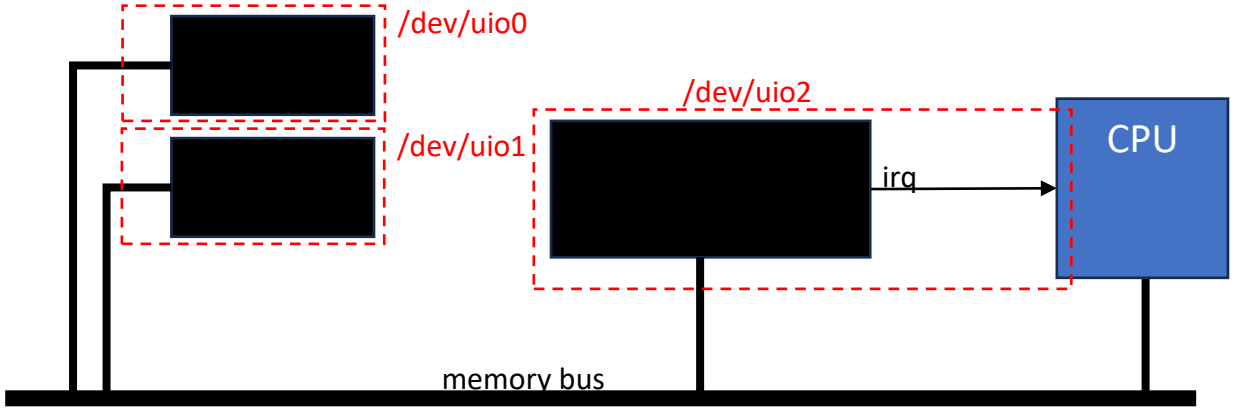

Figure 4.2: **Process: State Transitions**

# UIO read() & poll()



If we use the **read()** syscall on a UIO device, our process will block until an interrupt occurs

```
int fd = open("/dev/uio2", O_RDWR);   // UIO device file
while (1) {
  read(fd, …);                        // BLOCKS here until interrupt. Process goes to sleep
  handle_interrupt();                 // ...process wakes up, and can service interrupt
}
```




This can be thought of as handling interrupts in user space, but remember that it's **not a true interrupt handler routine**.

- When an interrupt arrives to the CPU from this irq line, it will call the ISR function in the UIO driver. This will in turn wake up your process and mark it as ready to run (you still need to wait for the scheduler to run your process)
- You can use the **poll()** syscall to determine if an interrupt has occurred without blocking and going to sleep.

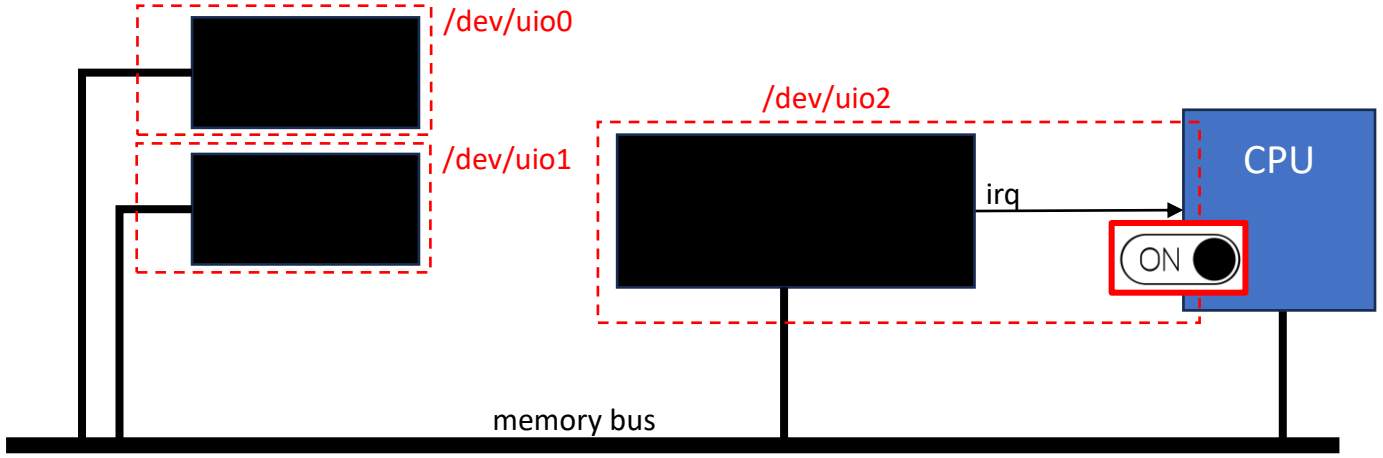
# UIO Interrupt Handling vs Kernel Interrupt Handling



| | In-kernel driver | UIO + blocking read() |
|---|---|---|
| *Where your code runs* | Inside the kernel | Normal program in user space |
| *How it's notified* | CPU jumps directly to your function | Your thread sleeps in read(), kernel wakes it |
| *Response Time from Interrupt* | Fast — a few microseconds | Slower — maybe 10–30 microseconds, because waking a sleeping process takes time |
| *What you're allowed to do* | Restricted to certain kernel operations, but can access certain things that would not be available to user space (e.g. DMA) | Anything a user space program can do |

# UIO write()






- There is a ON/OFF switch within the UIO driver that will determine whether an interrupt from this device will be serviced. If not enabled, the interrupt will be ignored.
- You must use the **write()** syscall and write a **1** (4 byte size) to the UIO device to turn this on.
- Every time the device generates an interrupt, the UIO driver will turn this back OFF.

# UIO Example



```
int fd = open("/dev/uio2", O_RDWR);   // UIO device file

write(fd, …);                         // Write a 4-byte 1 to turn on interrupt handling

while (1) {
  read(fd, …);                        // BLOCKS here until interrupt. Process goes to sleep
  handle_interrupt();                 // ...process wakes up, and can service interrupt
  write(fd, …);                       // Turn interrupt handling back on
}
```

How do you write() a 1 value of 4 bytes?

https://man7.org/linux/man-pages/man2/write.2.html