Lecture 30, Nov 24, 2022

Polling

- Repeatedly checks to see if a device is ready or if there has been an event, e.g. checking for a button input
- Example: system with 4 keys turning on LEDs, keys are at the lowest 4 bits of `0xFF200050`, LEDs at the lowest 4 bits of `0xFF200000`

```
_start:
    li s0, 0xff200050 # Load address of keys
    li s1, 0xff200000 # Load address of LEDs

# Repeatedly check the keys to see if they have been pressed
POLL:
    lw s2, 0(s0)
    beqz s2, POLL # If no keys are pressed, poll again
WAIT:
    lw s3, 0(s0)
    bnez s3, WAIT # Wait until keys are released
    li s3, 1 # If key 0 is pressed, then s3 is 0b0001
    bne s2, s3, CHECK_1
    li s4, 0 # Turn on 0 LEDs if key 0 is pressed
    j UPDATE_LED
CHECK_1:
    li s3, 2
    bne s2, s3, CHECK_2
    li s4, 1
    j UPDATE_LED
CHECK_2:
    li s3, 4
    bne s2, s3, CHECK_3
    li s4, 3
    j UPDATE_LED
CHECK_3:
    li s4, 7 # No need to do another check, this is the only scenario left
UPDATE_LED:
    s2 s4, 0(s1) # Actually update the LEDs
    j POLL
```

Interrupts

- Polling is inefficient, since the processor is constantly checking for events and so cannot do other work
- Instead we can use interrupts: the processor executes code normally, and when an event occurs the code execution is interrupted so the processor can handle the event
- The CPU first needs to be configured to accept interrupts
- Since interrupts can occur at any point during code execution, in order to return to execution after handling an interrupt, the CPU needs to save the state (this is done automatically)
  - Similar to calling subroutines, except all registers are saved
  - The CPU then jumps to an interrupt handler, and when the interrupt is done, it restores the registers and goes back to the old
- Unlike polling, interrupts can happen anytime (once set up), but is more difficult to do
  - Polling is good when the wait is short
  - Interrupts are better for medium to long wait events
- Examples of events handled by interrupts:

  - External devices such as UARTs, USBs, network adapters, etc
  - OS timer
  - Disk I/O
  - Debugging breakpoints
  - Program errors (e.g. misaligned memory access, divide by zero, segfaults)
    * These are also known as exceptions and always arise within the CPU
- Interrupts from external devices come from an IRQ (interrupt request) line when the interrupt is acknowledged, the CPU sends back a signal to the device via an IACK (interrupt acknowledge) line
  - When the external device receives the ack, it will de-assert the IRQ line
- Interrupt handling uses Control and Status Registers (CSRs), which are special registers that monitor system state
  - Cannot be repurposed and are not interchangeable
  - Can be read and written to, but need special instructions
  - `ustatus`, `uie` (interrupt enable), `utvec` (trap/interrupt handler base address), `uepc` (exception program counter)
  - CSRs that start with `u` are "user" registers
- System instructions:
  - `ebreak` - pause execution (breakpoint)
  - `uret` - return from interrupt handler
  - `csrrw` - read/write CSR
  - `csrrsi` - read and set bits in CSR (immediate)