Communicating between CPU and I/O Device

In Lab 3, we had explicit instructions for input/output.

```plaintext
IN RA
OUT RB
```

Problem: What if there is more than one device?

Solution: Create a number of "ports"; each IN and OUT instruction would indicate the port, as well as the data to transfer

- This is what is done in Intel machines

Memory Mapped I/O

Another Approach:

- Reserve some "special" memory locations
- You can use instructions like lw and sw to transfer data to and from these special memory locations
- But, each of these "special" memory locations is somehow connected to an I/O device

Memory Mapped I/O: Output to Device

When you write to this "special memory location", the data you write is sent to the I/O Device
Memory Mapped I/O: Output to Device

Problem: Usually, the I/O Device can not accept data as fast as the CPU can send it (for example, a disk is pretty slow compared to a CPU)

Solution: Reserve another “Special Memory Location”. Call it “Ready”. The I/O device can set this memory location when it is ready to accept data. The CPU can clear this memory location when it sends data.

Code would do something like:

```plaintext
send data
set ready to 0
wait until ready = 1 ; I/O Device will set ready when it is ready for a new piece of data
send data
set ready to 0
wait until ready = 1
send data
set ready to 0
etc...
```

This technique is called “Polling”. After sending a piece of data, the CPU has to keep checking READY until it goes to 1.

A waste of the CPU computing power!
- Maybe we could be doing something else while we are waiting
- Switch to a different task, serve a different user, etc.

What can we do?
Interrupt-Driven I/O:

Here's a new approach:

- CPU will send a piece of data
- CPU will then go off and do something else
- When the I/O device is ready for a new piece of data, it will “interrupt” the CPU
- CPU will “satisfy the request” (i.e., send another piece of data)
- CPU will go back to what it was doing before it was interrupted

More Technical Details:

Any CPU has a number of input wires called “Interrupt Lines”
- The number of these lines vary from processor to processor

Each Interrupt Line is driven by an I/O Device

When one of these lines is toggled, the processor finishes the current instruction, and branches to a predetermined “interrupt service routine”.

The interrupt service routine is an assembly language routine which services the interrupt (i.e., sends a new piece of data)

When the interrupt service routine is done, the processor goes back to what it was doing before it was interrupted.

Example:

```
LW $t0, 0($s0)
ADD $t1, $t2, $t3
SUB $t2, $t1, $t4
SW $t2, 4($s1)
ADD $t1, $zero, $t2
e tc.
```

```
ADD $t4, $zero, $s5
LW $t5, 0($s6)
```

```
SW $t4, 0($s7)
```

```
```
INTERUPT
SERVICE
ROUTINE
```

MAIN
PROGRAM
Other Uses of Interrupts:

Power Supply Monitor:
- If power is about to die, we would like to interrupt the computer

Timer Chips:
- Often we want to do something at fixed intervals (i.e., update a clock).
- We could have a timer device that interrupts the processor every so often (typically, every 1/60th of a second)

Network Connections:
- When a packet arrives on a network, may want to interrupt the processor so it can deal with it.

Issues:

Say we have several I/O devices. When we get an interrupt, how do we know which device wants attention?
- If every device has its own interrupt wire, then it is easy
- But may not be feasible. Most processors have some mechanism to allow I/O devices to identify themselves after interrupting

What if we have an interrupt interrupting an interrupt service routine?
- Maybe we want to assign priorities to different types of interrupts
- E.g., if the power is about to die, that is pretty important.

If we have a pipelined design, we need to squash everything in the pipeline whenever we get an interrupt

Exceptions:

Exceptions are interrupts that originate within the CPU:
- Divide by zero
- Page Fault
- Illegal Instruction

In some processors, you use exceptions to call O/S routines
- Often called a “trap”