

# Interrupts

Eric McCreath

# Interrupts

"Interrupt - An event that alters (or interrupts) the normal fetch-decode-execute cycle of execution in the system."

Glossary of Computer Organization and Architecture Ed. 2,  
Null and Lobur.

When an interrupt occurs the currently executing program is suspended and the program counter jumps to the interrupt handler (normally at a fixed location in memory). Once the interrupt handler completes the suspended program is restarted.

The interrupt handler must return the CPU in exactly the same way it finds it. Hence the interrupted program is 'unaware' that the interrupt has every taken place.

# Interrupts

Interrupts can be caused by:

- hardware events that electrically signal the CPU. (e.g. a key pressed, DMA transfer completing, network packet received, timer going off, a signal from another CPU), these are asynchronous events.
  - software initiated events including:
    - events that are used to capture problems in the execution of instructions (divide by zero, executing privileged instructions, reading/writing memory that out of the address range of the program executing)
    - or events to call operating system functions (provides a secure way of moving from an unprivileged to privileged mode).
- The CPU generally handles interrupts all in the same way (this is independent to how they are caused).

# Interrupt Handler

- The interrupt handler routine is a function for handling the interrupt event.
- When an interrupt occurs a table, at a known location in memory, is used to determine which interrupt handler to start executing.
- The interrupt handler routine must be written very carefully such that the state of the CPU is returned exactly how it was found. This will involve storing and restoring the register values that are used by the interrupt handler (generally done with the stack).

# Interrupt Handler

- The interrupt latency is the time between when an interrupt happens and the code in the interrupt handling routine starts executing.
- If interrupts are occurring more frequently than the handler can service them, then requests will be lost (interrupt storm).

# Interrupt Handler

- Often the interrupt handler is the very minimum code for dealing with the interrupt, anything that can be left for later is left for later.
- Linux, and other operating systems, divide the interrupt handling routine into 2 halves, the top-half which actually handles the interrupt and the bottom half which later completes the work associated with the event that initiated the interrupt. The top-half would schedule the bottom-half, also because two halves are working on the same data structures they need to be carefully synchronized.

# Masking of Interrupts

- Generally an interrupt handler can be interrupt by another interrupt.
- Some interrupts need to be handled more quickly than others. Hence there is generally a hierarchy associated with interrupts such that interrupts of lower or equal importance do not interrupt the code of an interrupt handler (the handler first completes, the mask is removed this enables the pending interrupt).
- rPeANUt has an interrupt mask bit which is set when an interrupt occurs.
  - If you want the interrupt handler to be interruptable then you need to clean this bit at the beginning of the handler.
  - If you don't want this handler to be interruptable then you should clean the mask bit before the handler returns.

When an interrupt occurs the current PC is pushed onto the stack and the PC is set to the address associated with that interrupt.

Any registers used by the interrupt service routine must be saved and restored. The interrupt event also sets the interrupt mask high which should be cleared before the interrupt service routine finishes. The standard 'return' instruction is used to return from interrupts.

- The interrupts and their addresses are given below:

interrupt	address	description
memory fault	0x0000	This happens when memory is accessed that is not addressable.
IO device	0x0001	This happens when interrupts are enabled on the terminal device and a key is hit.
trap	0x0002	This interrupt happens when the trap instruction is executed.
timer	0x0003	When the timer is enabled and every 1000 clock cycles.





The IO Device interrupt is disabled by default. If you wish to enable it set bit 0 of the IO device's control register to 1 (this is memory mapped so it is a matter of write to address 0xFFF2):

```
store ONE 0xFFF2
```

If the interrupt bit is set then when the a key is hit an interrupt is generated. This

- pushes the current PC onto the stack,
- sets the interrupt mask, and
- jumps to address 0x0001.

Because all the address for the interrupts are next to each other the handler would normally jump to the code that handles it.

```
0x0001 : jump iodevhandler
```



The handler needs to:

- store any registers it uses on the stack handle the interrupt,
  - handle the interrupt,
  - restore the registers,
  - reset the interrupt mask (at some point during the handler),
- and
- return from the interrupt.

The below handler echos key presses:


```
iodevhandler : push R0
                load 0xFFFF0 R0
                store R0 0xFFFF0
                pop R0
                reset IM
                return
```

# Example

```
memfault:halt
iodev: jump serviceio
trap : jump servicetrap
```

```
serviceio: push R0
           push R1
           load 0xFFFF0 R0
           store R0 0xFFFF0
           pop R1
           pop R0
           reset IM
           return
servicetrap: push R0
            push R1
            push R2
            load #trapstr R0
            load #1 R1
            jump stbool
```

iodev interrupt just  
echos the key  
pressed




```
stloop:    store R2 0xFFFF0
           add R0 R1 R0
stbool:    load R0 R2
           jumpnz R2 stloop
           pop R2
           pop R1
           pop R0
           reset IM
           return
```

```
trapstr: block #"trap"
```

```
0x0100: load #1 R0
        trap
        store R0 0xFFFF2
loop:   load #1 R0
        add R0 R1 R1
        jump loop
```

The trap  
interrupt  
prints 'trap'  
to the terminal



# Exercises

- Get the timer interrupt to regularly print a "\*" to the terminal. Have the main program just spin in a tight loop.