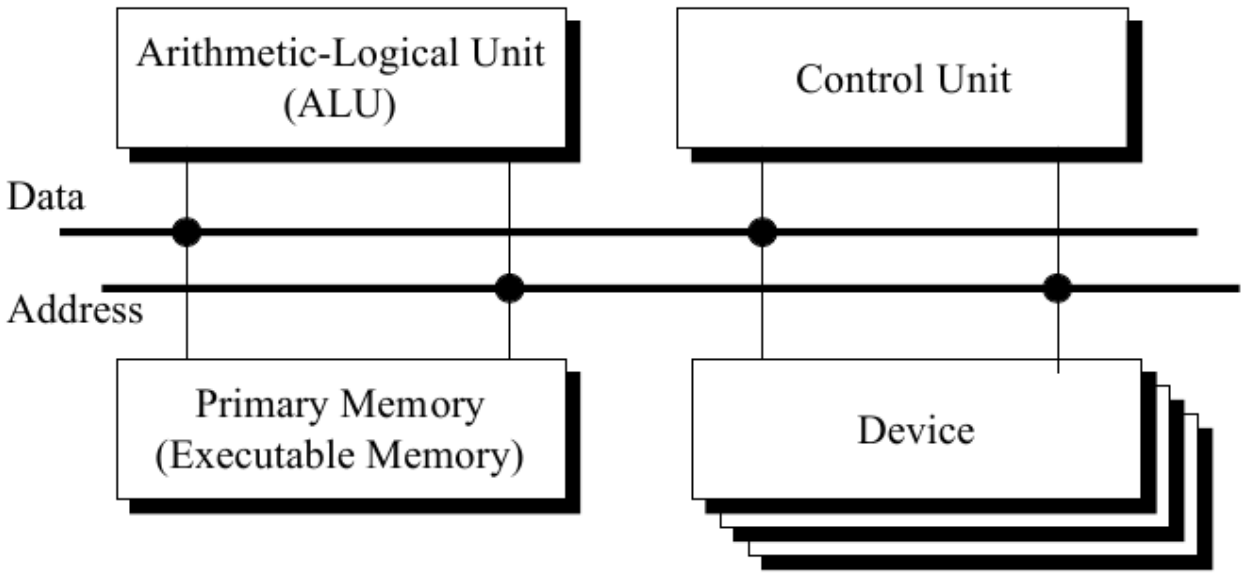
Review of Computer Organization

- von Neumann Architecture.
- Simple fetch-execute cycle.
- Interrupts and Interrupt Handlers

von Neumann Architecture



Bootstrap Loader

- A bootstrap loader begins loading the operating system and the control is eventually passed on to the operating system kernel.
- A bootstrap loader might alternatively load a more comprehensive loader that accomplishes loading the operating system kernel.
- A bootstrap loader typically resides in the ROM (Read Only Memory).

Bootstrap Loader (contd)

power-up sequence executes:

```
    load    PC, FIXED_LOC
```

a simple bootstrap loader:

```
FIXED_LOC:
        load    R1, =0
        load    R2, =LENGTH_OF_TARGET
        read    BOOT_DISK, BUFFER_ADDRESS
loop:   load    R3,  [BUFFER_ADDRESS, R1]
        store   R3, [FIXED_DEST, R1]
        incr    R1
        bleq    R1, R2, loop
        br      FIXED_DEST
```

Devices

- *block-oriented* versus *character-oriented* devices. E.g. Classify these devices: network interface, keyboard, CD drive, disk drive, floppy disk drive, mouse, tape drive, printer, sound card.
- The operating system tries to hide the details of the devices by using an interface common to all types of devices. The interface provides an abstract I/O paradigm. Typical operations include `open`, `close`, `read`, `write` and a general way of doing device specific operations (`ioctl` in Unix/Linux).

Device Controllers

- A device controller monitors the status of the device, provides commands for controlling the device and detects and possibly corrects some errors. This frees up the high level software (and the CPU).
- The interface between the device and its controller is conceptually transparent to the high-level software. Examples of such interfaces are SCSI (Small Computer System Interface), SATA (Serial Advanced Technology Attachment), SAS (Serial attached SCSI), Fibre Channel, USB, IDE (Integrated Drive Electronics), EIDE (Enhanced IDE) etc

Disk controller interfaces

- SATA. *Serial Advanced Technology Attachment*
  - Designed in 2003. Faster data transfer capability than the older EIDE interface, hot swapping capability, thinner cables and more reliable operation.
  - Speeds supported 1.5 GBit/s (150 MB/sec taking overhead into account), 3.0 GBit/s and 6.0 GBit/s. However actual transfer rates are currently limited by drive hardware.
- SCSI: *Small Computer System Interface* is another protocol used for connecting I/O devices (typically drives and tapes) to host systems. SCSI drives are usually faster, hot-swappable, more reliable and more expensive although SATA drives have similar performance now and as a result support for SCSI is diminishing, especially for desktop systems.

Device Drivers

- All the device management part of the operating system is encapsulated in device drivers.
- Device drivers for different drivers implement a similar interface so that higher level software has to know as little as possible about the device.
- The device driver along with the controller enables an application to run in parallel with the operation of the device. This CPU-I/O parallelism is used to improve CPU utilization although it isn't used to improve the performance of any one application (Why?). Another problem with independent I/O operation is the need to inform programs when I/O operation has completed (done via *interrupts*).

Interrupts and Traps

- *busy-wait* versus *interrupt* approach.
- *interrupt vector* and *interrupt handler*.
- *race condition* while the interrupt handler is in action.
- *trap* instruction and *trap handler*. Traps should be viewed as software interrupts.

Synchronous and Asynchronous I/O

To obtain the synchronous behavior of the read library call in a high level programming language, we must issue a low-level read call and block until the device has completed the read operation and the data is available in the buffer.

```
read(IO_PORT_TYPE io_port, char *buffer, int length) {
    /* starts the read on the device */
    start_low_level_read(device_parameters);
    /* wait until the operation completes */
    wait_for_device(...);
    /* copy the data into the memory buffer */
    copy_into_buffer(buffer);
    ...
    return;
}
```

An example of asynchronous I/O.

`xRead(FILE *fileID, char *buffer, int n, int *flag)`

Read up to *n* bytes into the address specified by `buffer`.

return value:

0 if it failed to start the read,

1 if it succeeded in starting the read.

The interrupt handler sets the flag to `TRUE` when the read operation completes. Here is a sketch of how to use `xRead(...)`.

How to use asynchronous I/O call?

```
// start a read operation
        flag = FALSE;
        if (!xRead(fileID, buffer, n, &flag)) {
        // could not start the read,
        // perform error processing.
        }
// continue to execute other stuff while
// I/O happens in parallel.
    ...
// Need to use the data, is it ready?
// Wait until flag gets set to TRUE.

        while (!flag);   // busy-wait

// Now we can use the data in the buffer.
```

Asynchronous I/O with the select system call

- Check out man page for `select()` system call for another way of performing asynchronous I/O. This system call lets the programmer check a number of I/O streams to see if one or more has changed status.
- This allows the implementation of a finite state machine, which is an alternative to multi-threading.