Introduction

- Design and implementation of OS is closely dependent on the function of the computer.
- Understanding of computer organization is critical to the appreciation of OS design.
- Modern computers based on concept of a stored program which determines how a computer function.
- Inspired by the Jacquard Loom, which was used to weave patterns into cloth.
- Different programs perform different sequences of operation.
- Ability to solve many types of problems.
- What is a program?
    - File of instructions (Source file)
    - High level programming (e.g. C, Python, Java)
    - Low level programming (e.g. Assembly - processor specific)
    - Machine language (processor specific)
- Computers execute machine language.
- Programs must be complied into machine language (exe file).

Program Specification

- High-level language translates to low-level language.
- Low-level language is compiled into machine language.

Von Neumann Architecture

- Forms the basis for almost all modern computer systems. Most other specialized systems evolved from this architecture.
- Has a fixed set of electronic parts, which can be manipulated to perform various tasks determined by a variable program.
- Consists of the following parts:
    - A Central Processing Unit (CPU)
    - A primary memory unit
    - A collection of I/O Devices
    - Buses to interconnect the components
- How it is interconnected:
    - CPU
        - Arithmetical Logical Unit (ALU)
        - Control Unit (CU)
    - Primary Memory Unit (Executable Memory)
    - Device
    - Address Bus
    - Data Bus
- The control unit decodes stored instructions and the ALU executes them. The primary (executable) memory is used to store the program and data that are operated on by the CPU. The devices are used for input, output, communications and storage. The bus interconnections and the CPU, primary memory and devices.

Central Processing Unit

- The CPU is the brain of the computer
- Made up of:
    - Arithmetical Logical Unit (ALU)
    - Control Unit (CU)

Arithmetical Logical Unit (ALU)

- Can be thought of as a very fast calculator.
- Can perform various arithmetic and logical operations.
- Typically has a 32 to 64 registers (very fast memory).
- Responsible for performing arithmetic and logical operations
- Comprises of:
    - Functional unit
        - Performs the operations
        - Includes:
            - Left and Right Operand
    - Registers (Very fast memory)
        - Data, status registers
        - Loaded and saved to/from primary memory
        - 32 to 64 registers to hold 32-bit/ 64-bit data
- Computations are accomplished by:
    - Loading binary values into registers.
    - Performing operations on the registers using the function unit.
    - Storing the result back into a general register.
    - Saving the register contents back to memory.

Control Unit

- Causes a sequence of instructions stored into the memory to be retrieved and executed.
- Comprises of:
    - Fetch Unit
        - Fetches an instruction from memory.
    - Decode Unit
        - Decode an instruction.
    - Execute Unit
        - Signal ALU to execute instruction.
    - Instruction Register (IR)
        - Contains a copy of the current instruction.
    - Program Counter register (PC)
        - Contains the memory address of the next instruction the unit is to load.
- Works based on fetch-execute cycle.
- When the computer is powered up, the control unit begins to execute the [[L3 - Computer Organization#^fe0ba7|Fetch-Execute Cycle]] until the computer is shut down.

Fetch-Execute Cycle

^fe0ba7

- Fetch phase
    - Instruction retrieved from memory at location specified by Program Counter (PC)
    - Loaded into Instruction Register (IR)
    - PC is incremented
- Execute phase
    - ALU operation
    - Cause memory data reference, I/O operation

Control Unit Operation

Fetch phase

- Instruction retrieved from memory.

Execute phase

- ALU op, memory data reference, I/O, etc.

Primary Memory Unit

- Stores both programs and data while they are being operated on by the CPU.
- Interface between CPU and memory consists of 3 registers:
    - Memory address register (MAR)
        - Stores address of data to be read from or written to.
    - Memory data register (MDR)
        - Stores data that is read of to be written.
    - Command register (CMD)
        - Stores the command to be executed.
- Stores programs and data in binary format.
- Often referred to as random access memory (RAM).

Read Operation:

1. Load MAR with address
2. Load Command with "read"
3. Data will then appear in the MDR

Write Operation:

1. CPU puts data into MDR
2. CPU puts location into MAR
3. Load Command with "write"

Input/Output (I/O) Devices

- Each device operation is controlled by a device controller.
- Device controller connects device to the computer's address and data bus.
- Provides an interface which the OS (Device manager) can use to manipulate device.
- Interfaces varies among controllers.
- OS provides abstraction to hide differences from programmer.

The Device-Controller-Software Relationship

- Abstract I/O Machine
    - Device manager
    - Program to manage device controller
    - Supervisor mode software

Device Controller Interface

- Device may need constant attention/monitoring during operation.
- Device controller does this with mainly hardware algorithms.
- Software interface (device driver) provided by controller allow OS to operate and synchronize controller allows OS to operate and synchronize its behavior with the device operation.
- Device controller include the following as part of the interface:
    - Data registers
    - Command registers
    - Status flags with includes done, busy and error code

CPU-I/O Overlap

- A CPU can only run one application at a time as it is time-multiplex shared
- Multiprogramming
- A program is running on the CPU while another is running on the device.

Determining When I/O is complete

- When the CPU initiates I/O, we need the device to 'notify' the CPU when the I/O is done.
- Two ways to do this:
    - [[L3 - Computer Organization#^a21394|Polling]]
    - [[L3 - Computer Organization#^dc8d29|Interrupt]]

CPU/Device Operation

- Performing a Write Operation (polling).

```
while(deviceNo.busy || deviceNo.done) <waiting>;
deviceNo.data[0] = <value to write>
deviceNo.command = WRITE;
while(deviceNo.busy) <waiting>;
deviceNo.done = TRUE
```

- Devices much slower than CPU.
- CPU waits while device operates.
- Would like to multiplex CPU to a different process while I/O is in process.
- This is possible using the interrupt method.

Polling

^a21394

- Simplest way is for CPU to keep polling the device to see state of the I/O.
- Device implements the status of the device as a flag.
- If the I/O is not done, the CPU executes a **busy-wait** command to wait for the I/O to end, but the CPU is effectively waiting and doing nothing.
- Waste precious processor cycles.
- Software:

```
…
// Start the device
…
While((busy == 1) || (done == 1))
wait();
// Device I/O complete
…
done = 0;
```

- Hardware:

```
…
while((busy == 0) && (done == 1))
wait();
// Do the I/O operation
busy = 1;
…
```

Interrupt

^dc8d29

- A more advanced but more complicated way is to have the CPU implement an interrupt request flag.
- When device IO is done, the device sets the interrupt request flag to signal the end of IO.
- The CPU, on its fetch cycle, would detect the flag and proceed to execute a set of routines to service the IO.
- CPU incorporates an "interrupt pending" flag.
- When `device.busy` -> FALSE, interrupt pending flag is set.
- Hardware "tells" OS that the **interrupt** occurred.
- **Interrupt handler** part of the OS makes process ready to run.

```
//Fetch-Execute Cycle with an Interrupt
while (haltFlag not set during execution) {
IR = memory[PC];
PC = PC + 1;
execute(IR);
if (InterruptRequest) {
```

```
    /* Interrupt the current process */
    /* Save the current PC in address 0 */
        memory[0] = PC;
    /* Branch indirect through address 1 */
        PC = memory[1];
    }
}
```

Direct Memory Access (DMA)

- In conventional design, the CPU transfers data between the controller data registers and the primary memory.
- This means that the CPU is involved in all operations on the memory.
- Problem is that most operations require memory access.
- In I/O operations, when the data is to be copied to memory is large, the CPU can get very busy just copying data.
- It is more efficient to implement direct memory access (DMA)
- DMA memory controllers are able to read/write data from/to memory without CPU intervention.
- The DMA controller is like a mini CPU, which is able to perform the tasks that the CPU would otherwise have to perform.
- CPU can start a DMA block transfer and then perform other work in parallel with the DMA operation. This can significantly increase the machine's I/O performance.
    - How does it knows when to start a DMA block transfer?