Nanyang Polytechnic

IT1313

Operating Systems and Administration

Chapter 3

Computer Organization

Computer Organization

At the end of this chapter, you should be able to

- Describe the von Neumann Architecture of computers
- Describe the function of the various components
- Explain the two methods of determining I/O completion
- Explain what is Direct Memory Access

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.
- Although this course is NOT Computer Architecture, this chapter gives a short overview of areas covered.

Introduction

- 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

Introduction

- What is a program ?
  - File of instructions (Source file)
  - High level programming language
    - e.g. C, Java, Python
  - Low level programming language
    - e.g. assembly language (processor specific)
  - Machine language (processor specific)
- Computers execute machine language
- Programs must be compiled into machine language (EXE file)

Program Specification

Source

```
int a, b, c, d;
. . .
a = b + c;
d = a - 100;
```

translates

Assembly Language

```
; Code for a = b + c
    load      R3,b
    load      R4,c
    add       R3,R4
    store     R3,a

; Code for d = a - 100
    load      R4,=100
    subtract  R3,R4
    store     R3,d
```

Low-Level Language

High-Level Language

Machine Language

Assembly Language

```
; Code for a = b + c
    load      R3,b
    load      R4,c
    add       R3,R4
    store     R3,a

; Code for d = a - 100
    load      R4,=100
    subtract  R3,R4
    store     R3,d
```

Machine Language

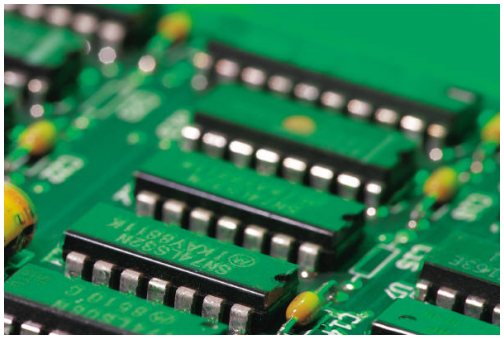
```
10111001001100…1
10111001010000…0
10100111001100…0
10111010001100…1
10111001010000…0
10100110001100…0
10111001101100…1
```

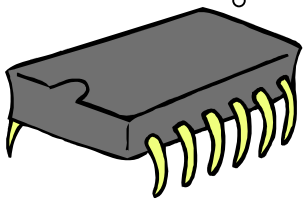
Stored Program Computers and Electronic Devices

Pattern

Jacquard Loom

Variable Program





Stored Program Device

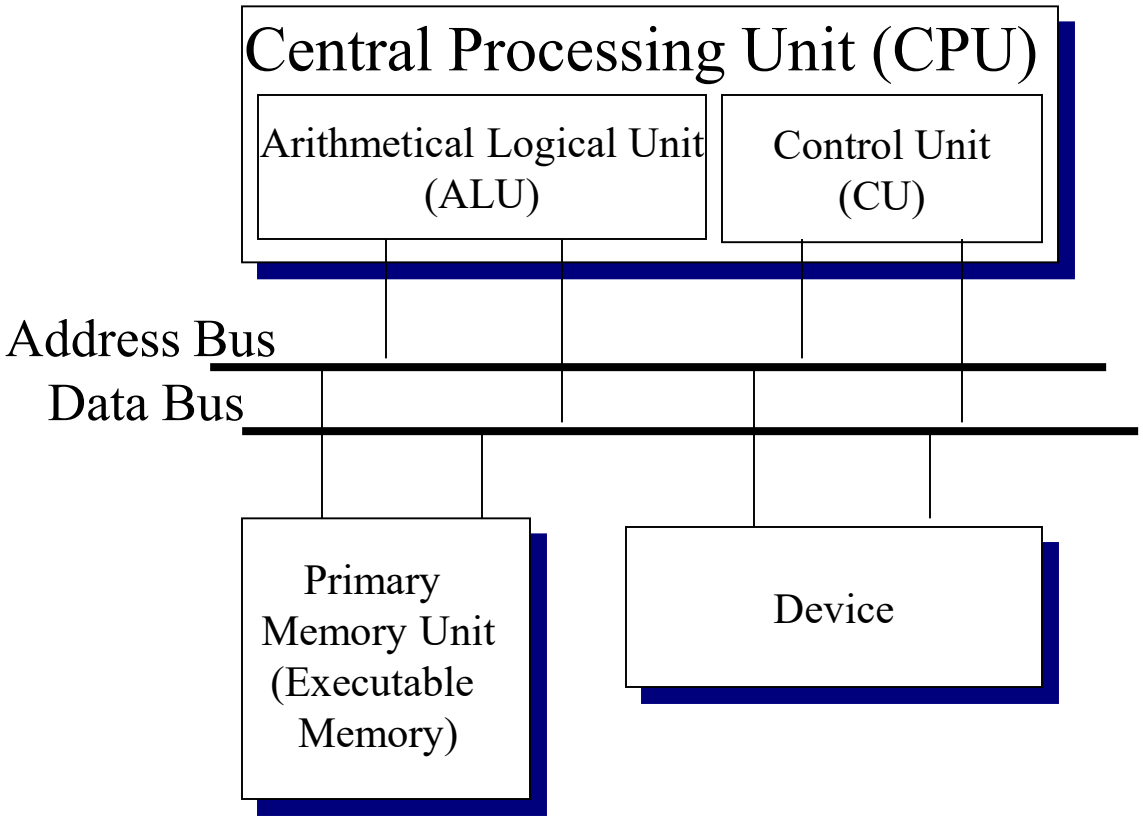
Fixed Electronic Device

The 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

The von Neumann Architecture

A von Neumann computer contains a CPU, which contains an ALU and control unit. 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 the CPU, primary memory and devices.

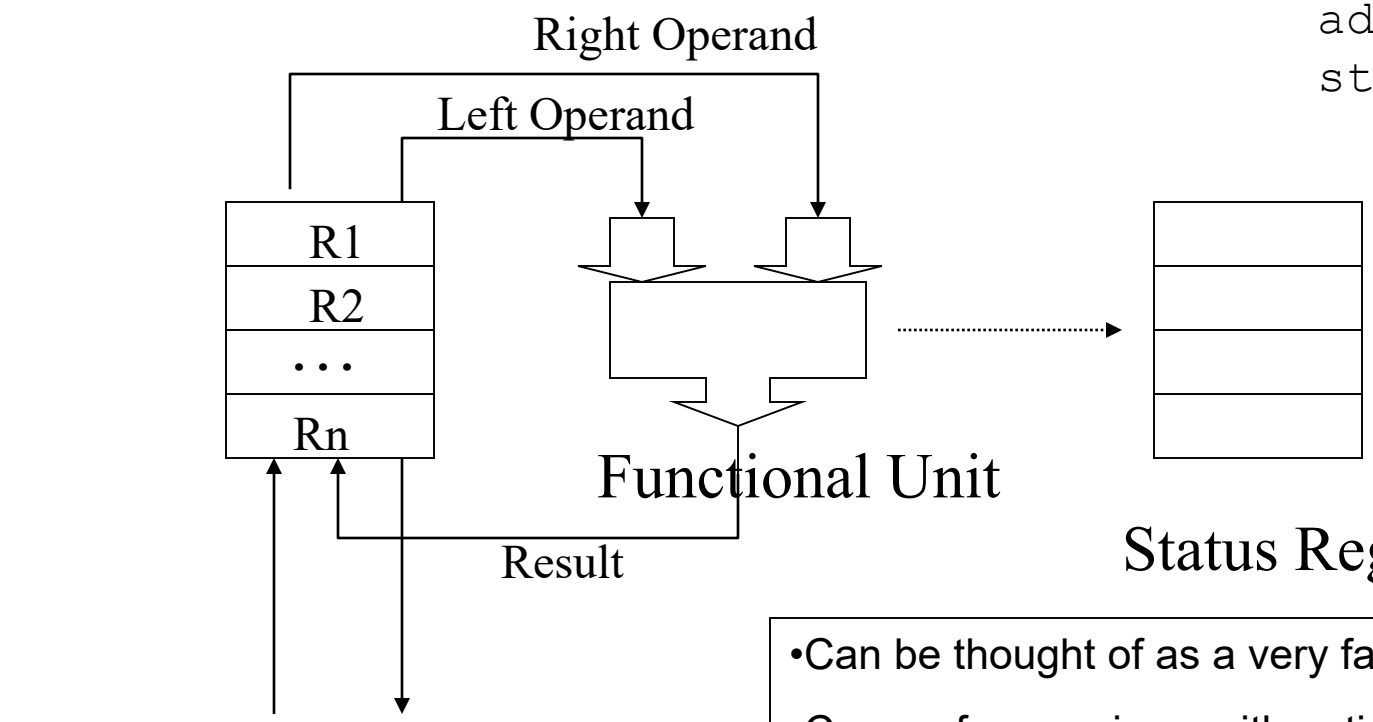


Central Processing Unit

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

The ALU

```
load    R3,b
load    R4,c
add     R3,R4
store   R3,a
```



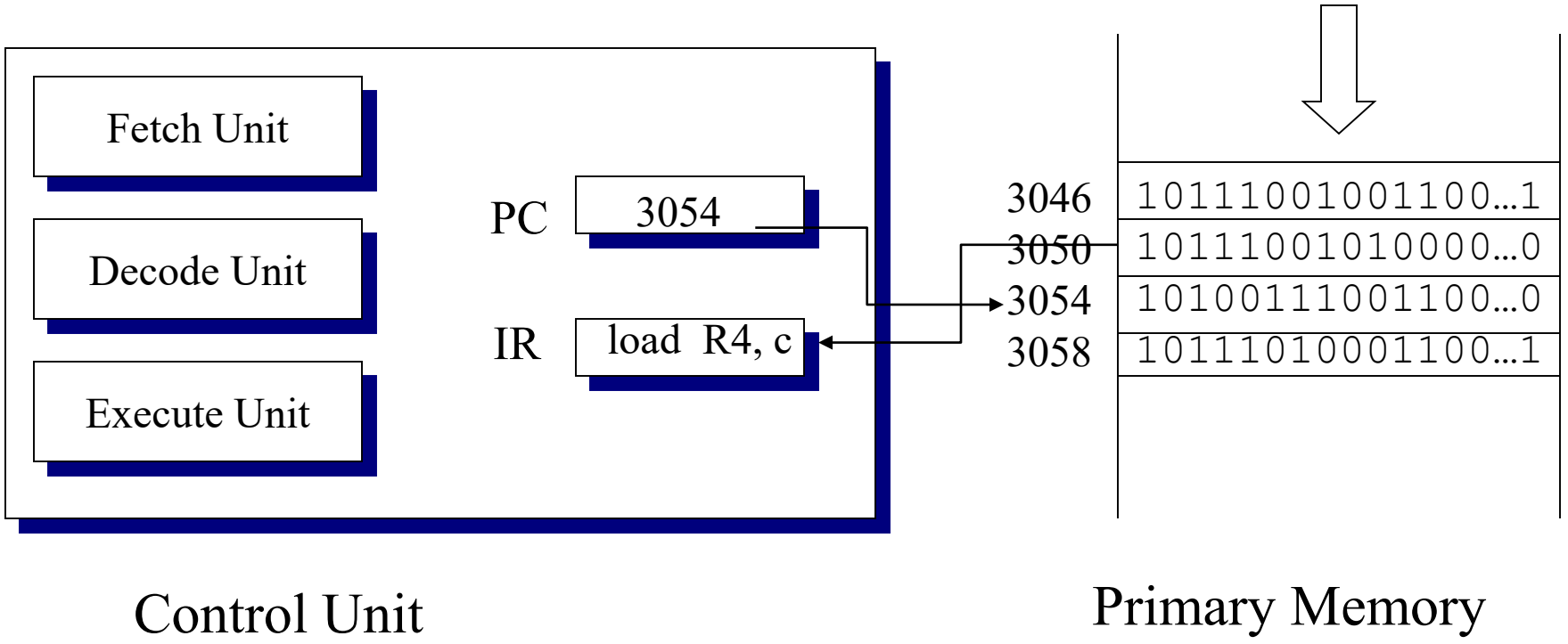
- Can be thought of as a very fast calculator.
- Can perform various arithmetic and logical operations.
- Typically has 32 to 64 registers (very fast memory).

Arithmetical-Logical Unit (ALU)

- Responsible for performing arithmetic and logical operations
- Comprises
  - Functional unit
    - Performs the operations
  - Registers (very fast memory):
    - Data, status registers
    - Loaded and saved to/from primary memory
    - 32 to 64 registers to hold 32-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

```
load   R3,b
load   R4,c
add    R3,R4
store  R3,a
```



Control Unit

- Causes a sequence of instructions stored in the memory to be retrieved and executed.
- Comprises
  - 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

Control Unit Operation

- When the computer is powered up, the control unit begins to execute the **fetch-execute** cycle until the computer is shut down.
  - 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

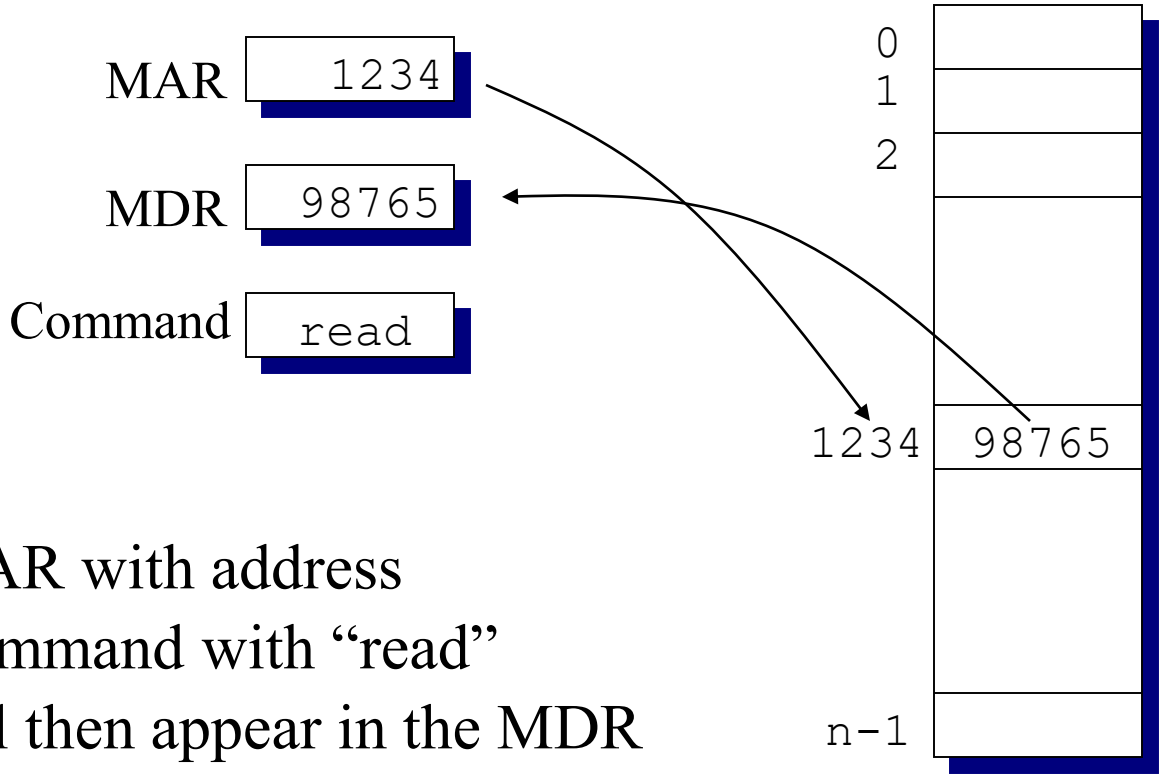
- Fetch phase: Instruction retrieved from memory
- Execute phase: ALU op, memory data reference, I/O, etc.

```
PC = <machine start address>;
IR = memory[PC];
haltFlag = CLEAR;
while(haltFlag not SET) {
    execute(IR);
    PC = PC + sizeof(INSTRUCT);
    IR = memory[PC]; // fetch phase
};
```

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 or 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).

Primary Memory Unit



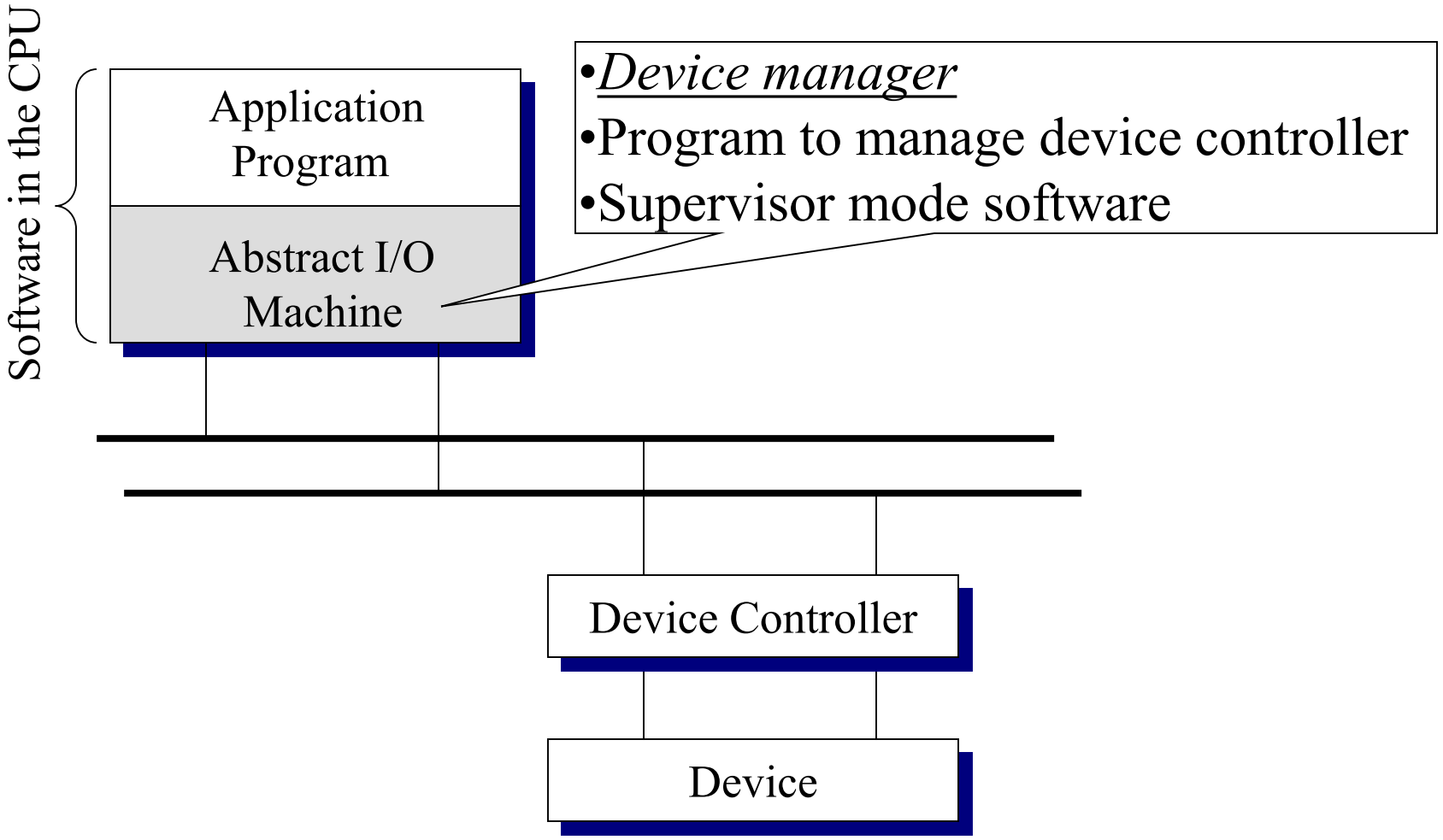
Read Op:

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

Input/Output 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

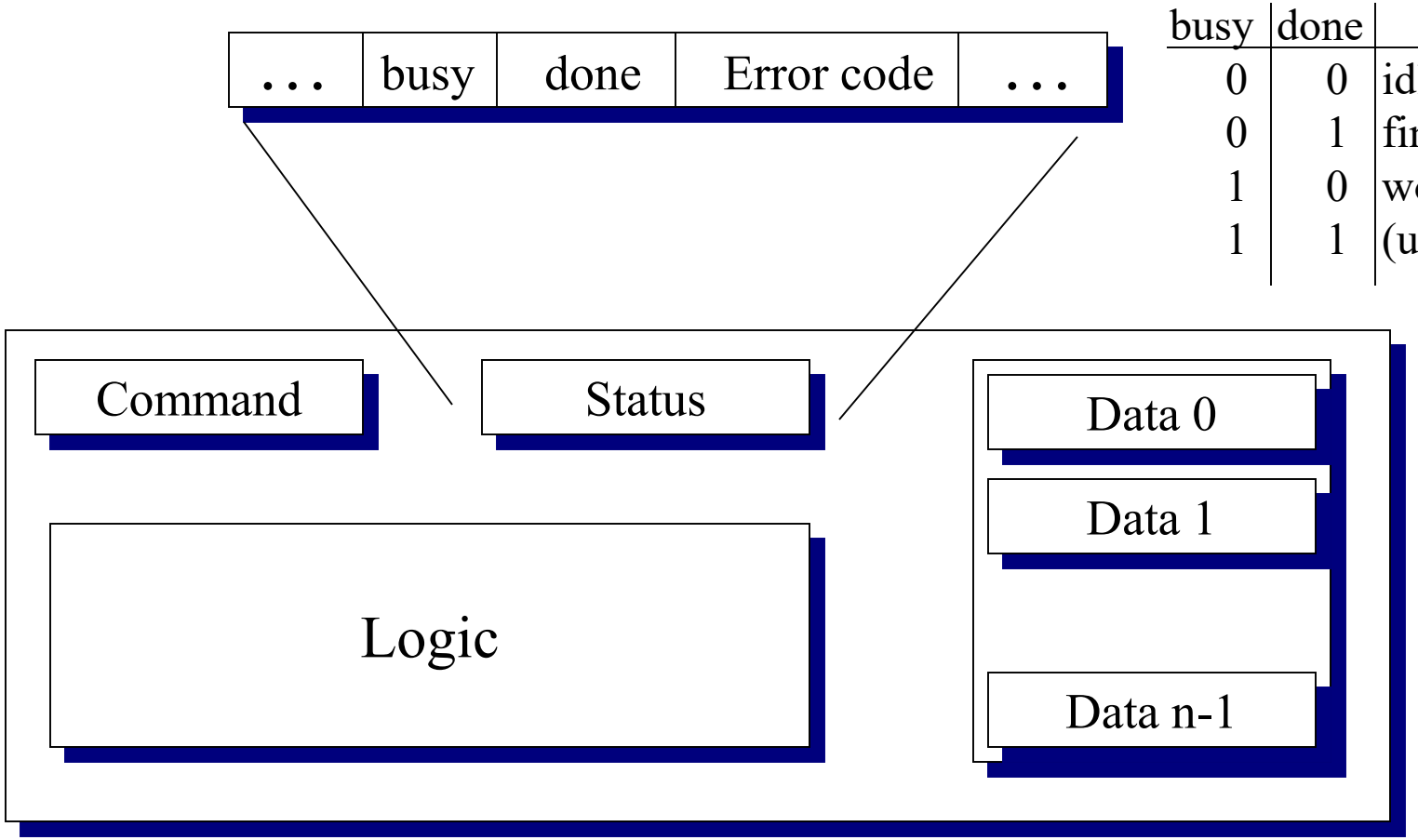
The Device-Controller-Software Relationship



Device Controller Interface

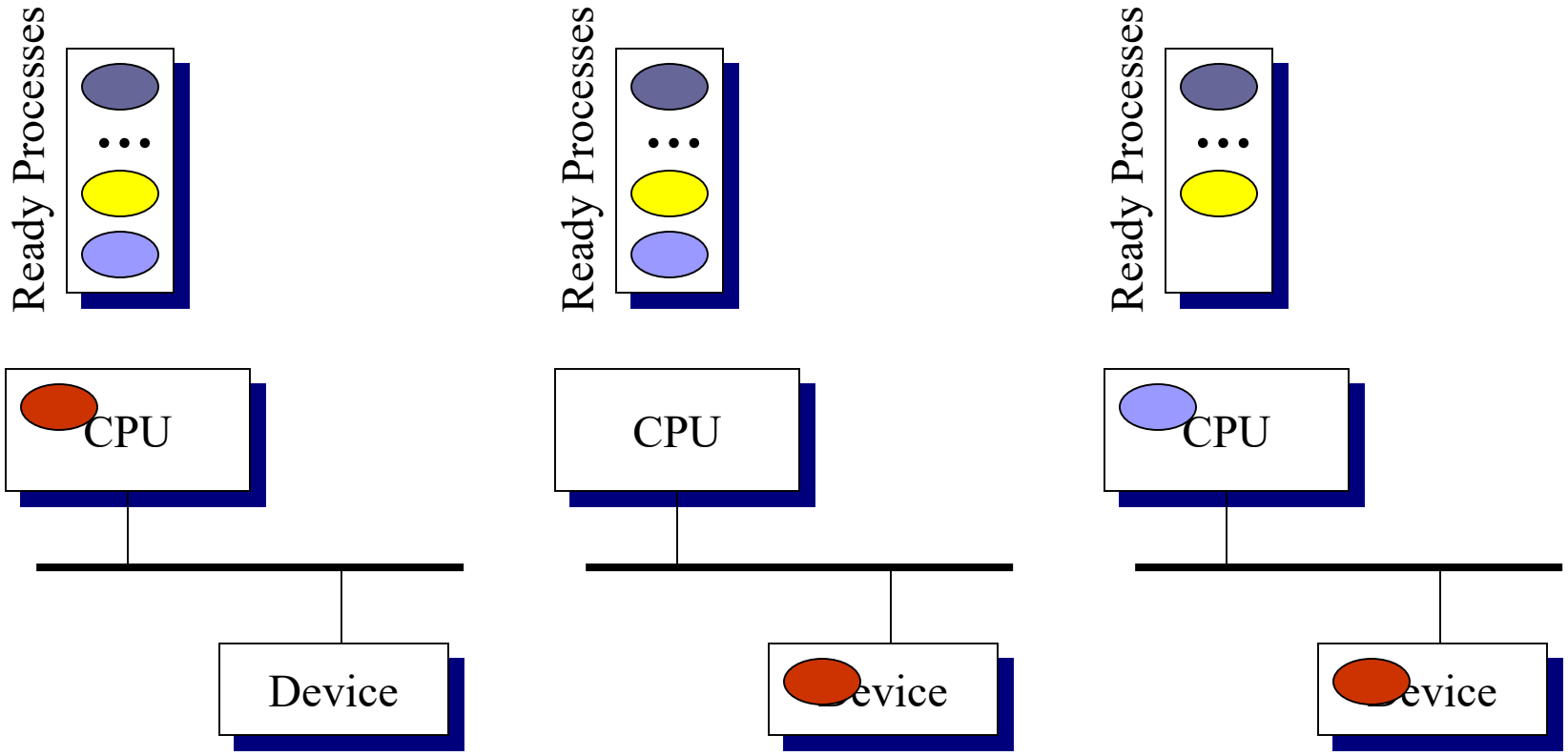
- Device may need constant attention/monitoring during operation
- Device controller does this with mainly hardware algorithms
- Software interface (device driver) provided by controller allows OS to operate and synchronize its behavior with the device operation
- Device controller include the following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

Device Controller Interface



busy	done	
0	0	idle
0	1	finished
1	0	working
1	1	(undefined)

CPU-I/O Overlap



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
  - **Polling**
  - **Interrupt**

CPU/Device operation

- Performing a Write Operation (polling)

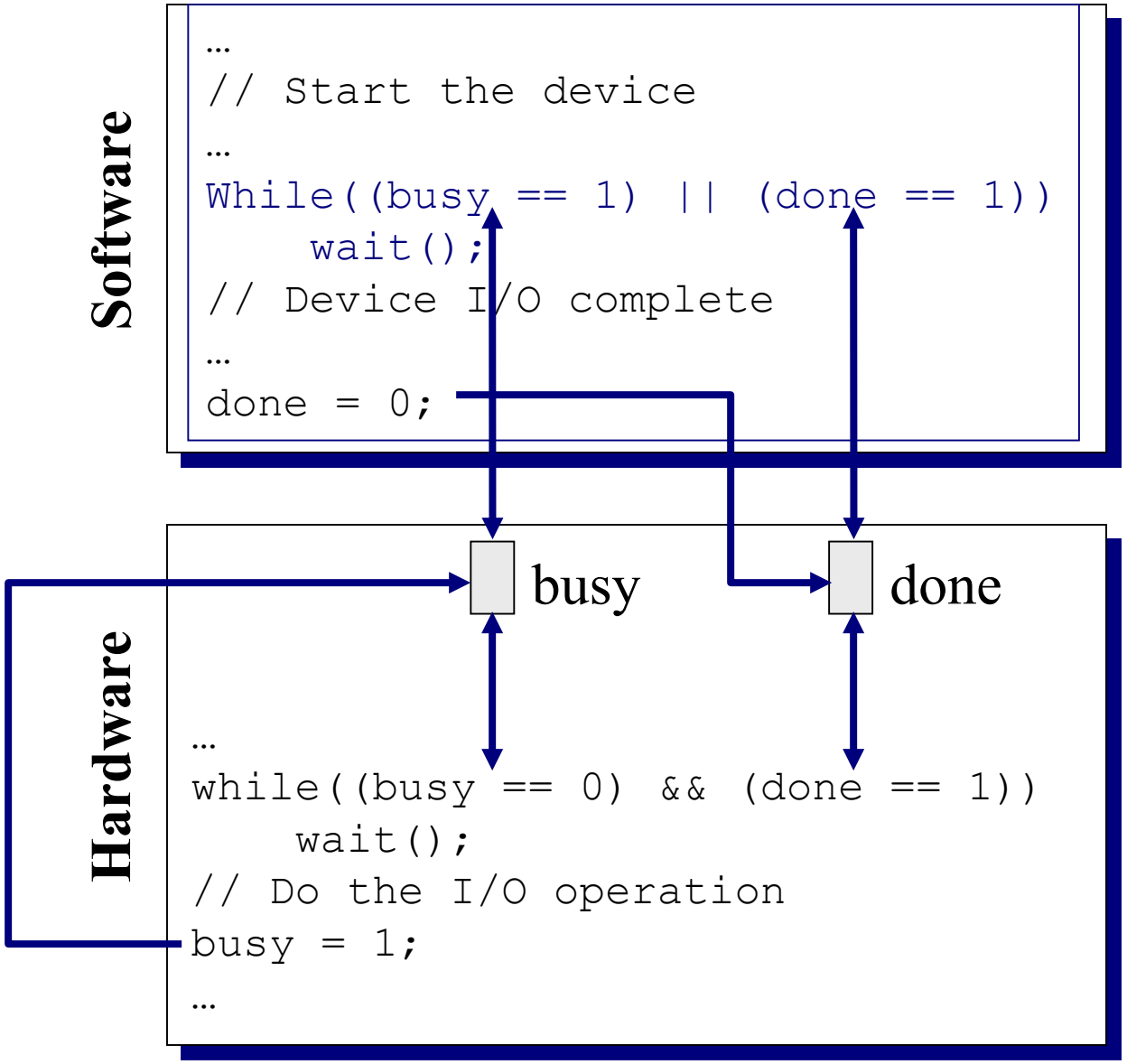
```
while(deviceNo.busy || deviceNo.done) <waiting>;
deviceNo.data[0] = <value to write>
deviceNo.command = WRITE;
while(deviceNo.busy) <waiting>;
deviceNo.done = TRUE;
```

- 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

Determining When I/O is Complete

- Polling
  - 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

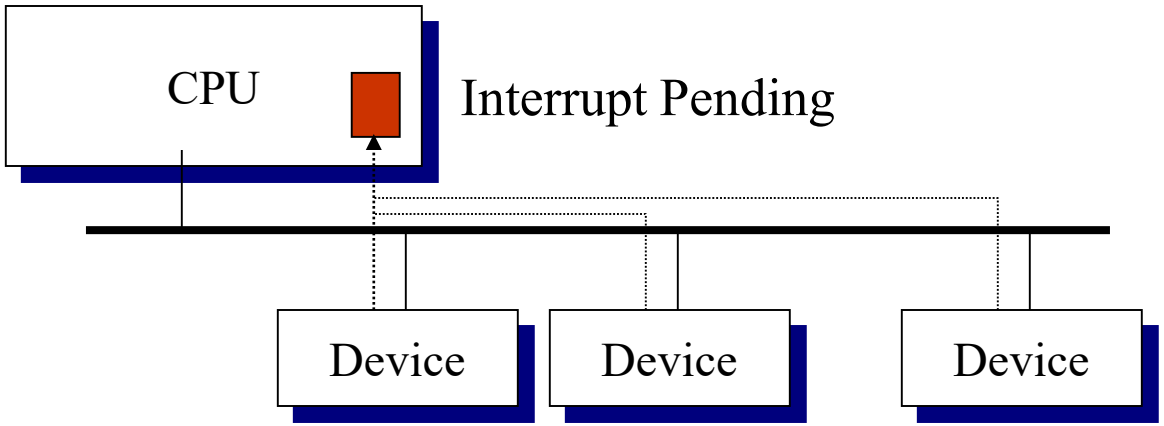
Polling I/O


Software
…
// Start the device
…
While((busy == 1) || (done == 1))
wait();
// Device I/O complete
…
done = 0;
busy
done
Hardware
…
while((busy == 0) && (done == 1))
wait();
// Do the I/O operation
busy = 1;
…

Determining When I/O is Complete

- Interrupt
  - 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.

Determining When I/O is Complete – Interrupt Flag



- 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];
   }
}
```

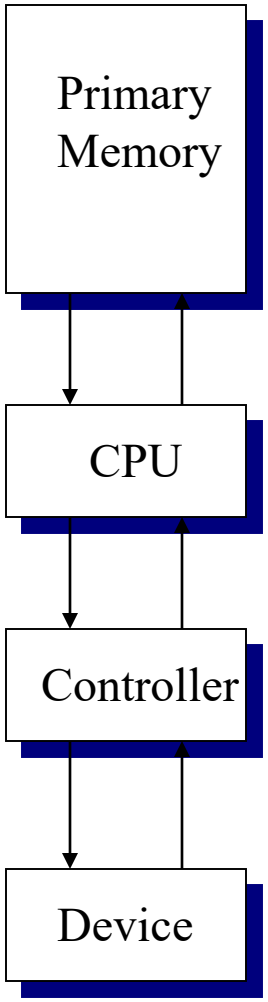
Direct Memory Access

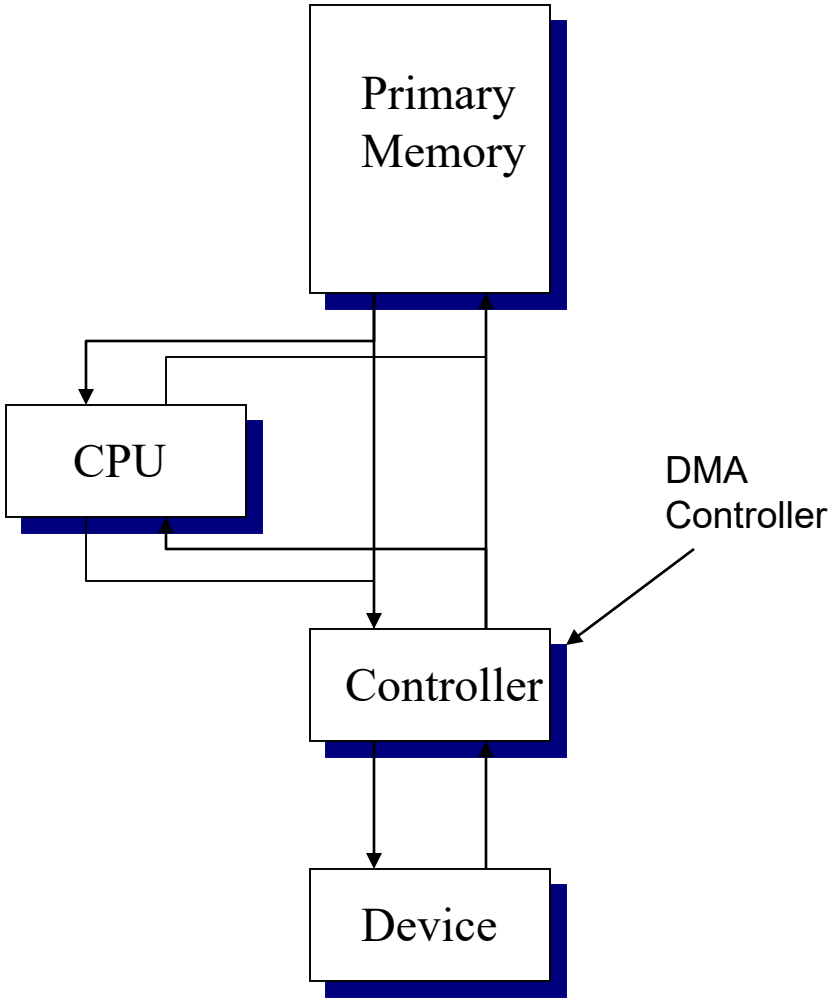
- In conventional design, the CPU transfers data between the controller data registers and the primary memory.
- This means that the CPU is involved in *all* operations on the memory.
- Problem is that most operations require memory access.
- In I/O operations, when the data to be copied to memory is large, the CPU can get very busy just copying data.

Direct Memory Access

- 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.

Direct Memory Access





Conclusion

- The modern computer is based on the von Neumann architecture which enables a fixed set of electronic parts to perform variable tasks.
- While interfaces between the parts are mainly hardware oriented, software interfaces exist between the hardware and the OS.