WHAT IS AN OPERATING SYSTEM?

What is an OS

- Informally: Something that helps you use the available hardware
  - Orig.: **Expensive hardware, cheap people**
    Goal: maximize hardware utilization
  - Now: **cheap hardware, expensive people**
    Goal: make it easy for people to use.

- How to make it easier to use?
  Abstractions!

Operating System

- An OS is a standardized abstraction, a VM, that is implemented on the underlying machine.
  - Manages resources: processor, memory, I/O
  - Provides a set of services to system users
  - Consumes resources (complexity vs. cost)

Basic Elements

- What do you need in a computer?
  - One or more processing units (CPUs)
  - Memory
  - I/O modules
  - Timer(s)
  - Interrupt controller
  - System bus connecting them

Note, these are already abstractions of more complicated hardware systems.

- Question is: What do we need from these modules?

Basic Elements

- Processing unit
  - Some registers/buffers
    - Memory address resister (MAR)
      - Specifies the address for the next read or write
    - Memory buffer register (MBR)
      - Contains data written into memory or receives data read from memory
    - I/O address register
    - I/O buffer register

Most micro-controllers contain general purpose registers that take the role of the MAR and the MBR. On the CPU we have some registers but they result in a very small amount of memory (8-64 bits in size). Those registers are connected to main memory (0x0000-0xFFFF). Bits of this memory is mapped to flash and RAM and other bits of memory. For each chunk of memory we take a small piece where we store useful functions that allow processor to interact with main memory.

Basic Elements

- Main Memory
  - Volatile, referred to as real memory or primary memory
- I/O Modules
  - Secondary Memory Devices, communications equipment, terminals
- System bus
  - Communication among processors, main memory, and I/O modules

Computer Components

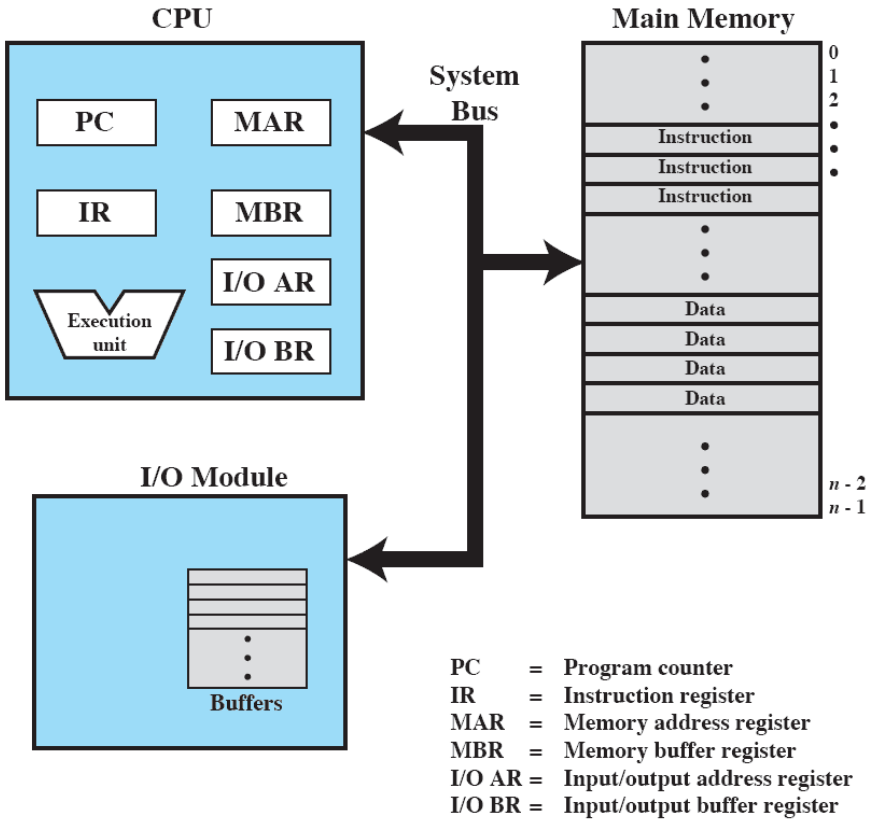


Figure 1.1 Computer Components: Top-Level View

In most computer architectures the program lives in RAM or micro-controllers. The program counter always contains the address of the next instruction for the program.

User-Visible Registers

- Enable programmer to minimize main memory references by optimizing register use
- Available to all programs – application programs and system programs
- Depend on computer architecture
- May be referenced by machine language
- Usually the compiler controls them
- Can be user controlled
  - **Register** keyword in C
  - **Volatile** keyword in C
  - **Clobber list**

You can use the voltile keyword or a cobbler list to keep the computer from optimizing certain registers.

User-Visible Registers

- Two categories of visible registers:

1. **Data registers:** store data (e.g., results from calculations; D0-D7 in M68k processors)
2. **Address register:** point to memory
   - Indexed addressing (Adding an index to a base value to get the effective address)
   - Segment pointer (When memory is divided into segments, memory is referenced by a segment and an offset)
   - Stack pointer (Points to top of stack for pop & push)

Control and Status Registers

- Invisible to the user (on most architectures)
- Used by processor to control operating of the computer
- Used by privileged OS routines to control the execution of programs

We want to hide control and status registers from the user but still have them around to show the status that the computer is in.

Examples of Invis. Registers

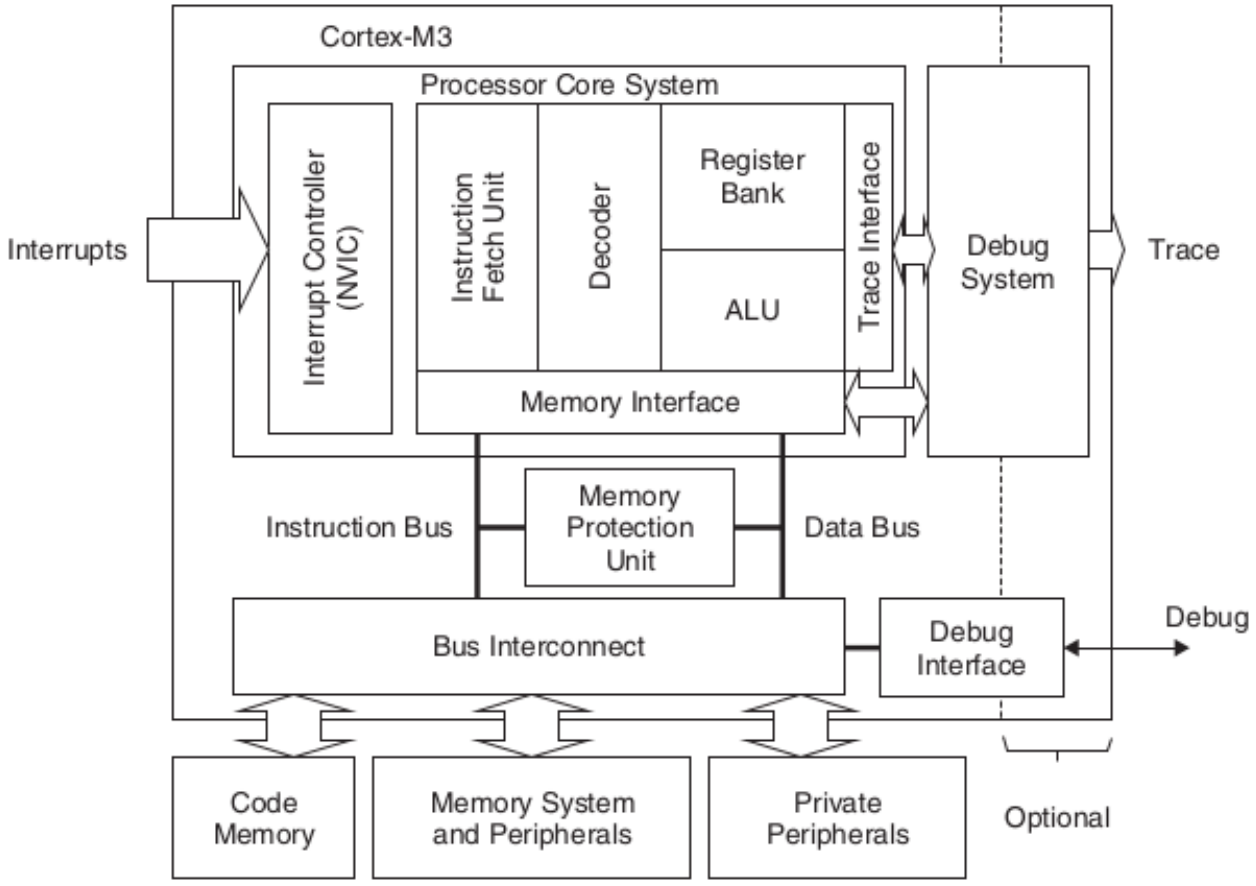
- Program counter (PC)
  - Contains the address of an instruction to be fetched
- Instruction register (IR)
  - Contains the instruction most recently fetched
- Program status word (PSW)
  - Contains status information and condition codes (e.g., Interrupts enabled, errors, supervisor mode bit)

Condition Codes or Flags

- Part of the PSW
- Bits set by processor hardware as a result of operations
  - Example: Positive, negative, zero, or overflow result of arithmetic operation
- Used for conditional branching
- Only implicitly accessible on most architectures
- Architecture specific ones include reset status (brownout, watchdog, power button).

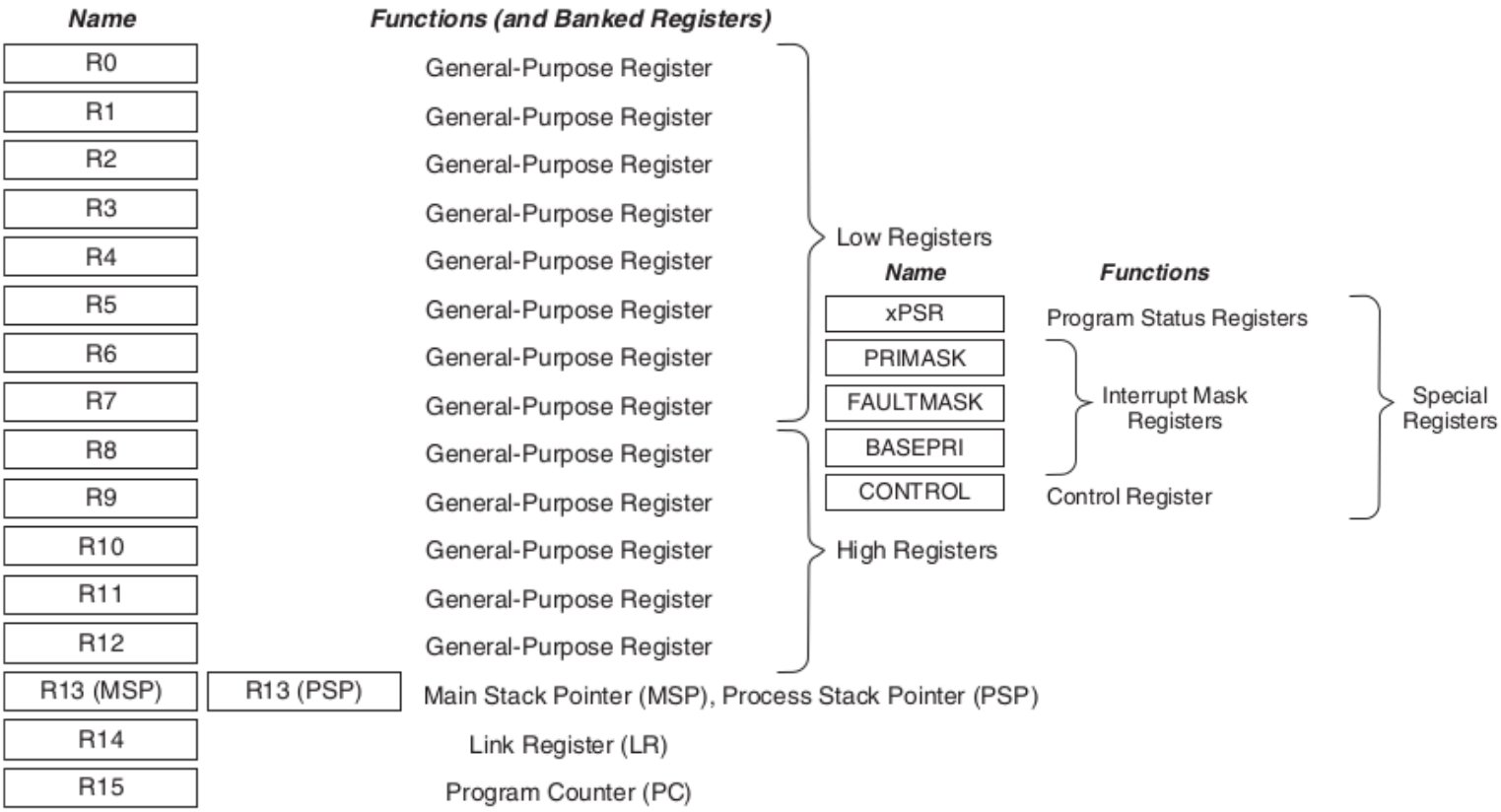
Brownout and watchdog are flags that talk about the powerstate of the system and allows for an interrupt based on it

Example: ARM Cortex-M3



Source: Yiu, The Definitive Guide to the Cortex-M3 (2007)

Example: ARM Cortex-M3



Source: Yiu, The Definitive Guide to the Cortex-M3 (2007)

Instruction Execution

- Simple version:
  1. Processor fetches instruction from memory
  2. Processor executes the instruction
  3. Goto 1
- Reality:
  - Pipeline
  - VLIW (very long instruction word; e.g. i860)
  - Superscalar architectures
    (e.g. P5, PowerPC 970)
  - … (see your computer architecture class)

Basic Instruction Cycle

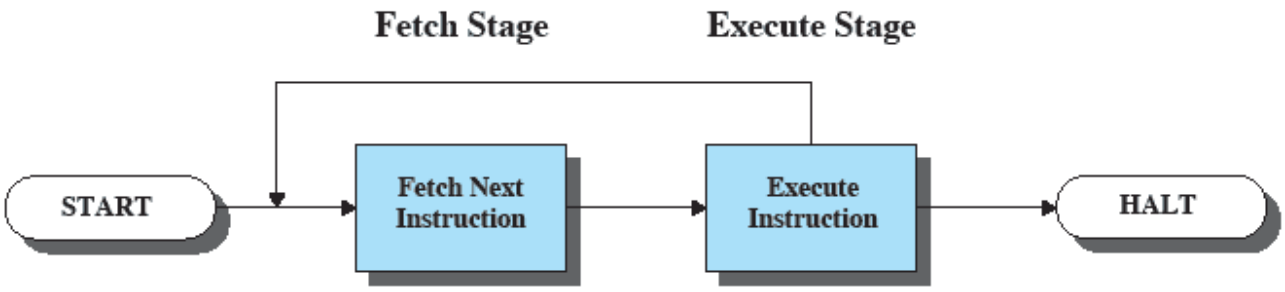


Figure 1.2 Basic Instruction Cycle

Instruction Fetch and Execute

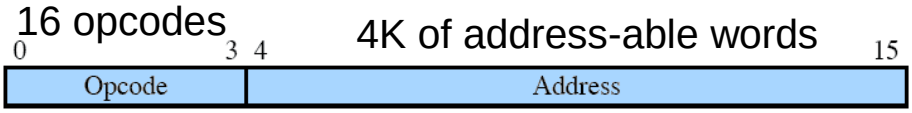
- The processor fetches the instruction from memory
- Program counter (PC) holds address of the instruction to be fetched next
- PC is incremented after each fetch

A thing to keep in mind is that the PC points to the next instruction after the fetch stage, this can fuck shit up if you use it incorrectly.

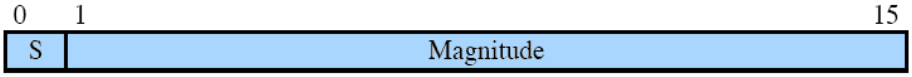
Instruction Register

- Fetched instruction loaded into instruction register (IR)
- Categories of actions dictated by the instruction
  - Processor-memory, processor-I/O, data processing, control (e.g., branching)
  - No clear boundaries; an instruction may include several of these actions

A Hypothetical Machine



(a) Instruction format



(b) Integer format

Program counter (PC) = Address of instruction
Instruction register (IR) = Instruction being executed
Accumulator (AC) = Temporary storage

(c) Internal CPU registers

0001 = Load AC from memory
0010 = Store AC to memory
0101 = Add to AC from memory

(d) Partial list of opcodes

Figure 1.3 Characteristics of a Hypothetical Machine

Operation 1 is a load operation into a accumunlator, operation 2 is a store that to memory, and operation 3 is a add to that from memory.

Example of Program Execution

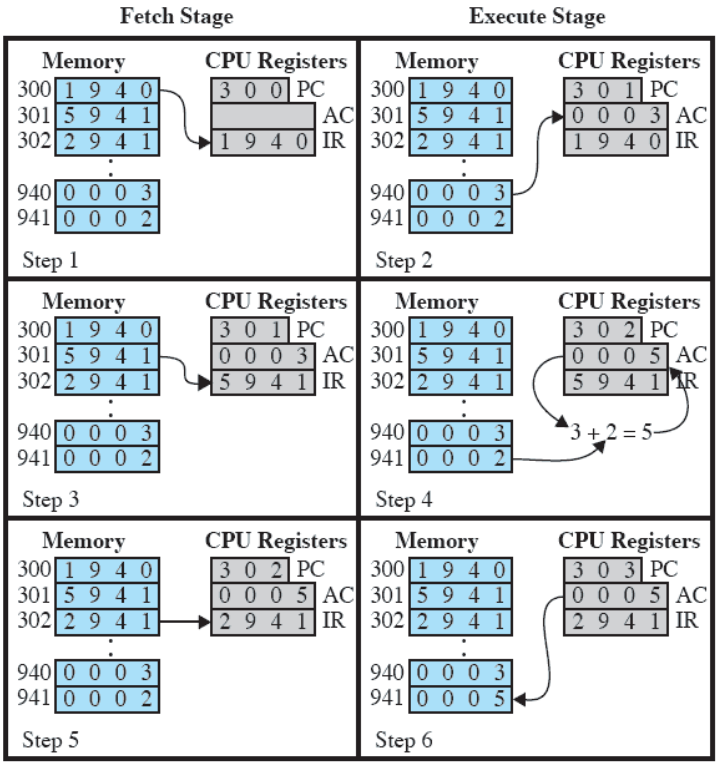


Figure 1.4 Example of Program Execution
(contents of memory and registers in hexadecimal)

In the initial condition we have the PC pointing to the next isntruction after the fetch (300). So the instruction that we have is number 1 (the load operation). During the execution we go to the address and get the data there and incrememnt the PC. In 301 we have the store command which we enact. Etc for the other command. We call this a two stage pipeline.

Interrupts

- Most I/O devices are slower than the processor
  - Processor must pause to wait for device
- Interrupts help to improve the processor utilization.
- The change the normal sequencing of the processor

Since most peripherals are slower than the computer so we use interrupts to make the program go at their stage, then return to the compture. To do this we need to change the execution cycle.

Classes of Interrupts

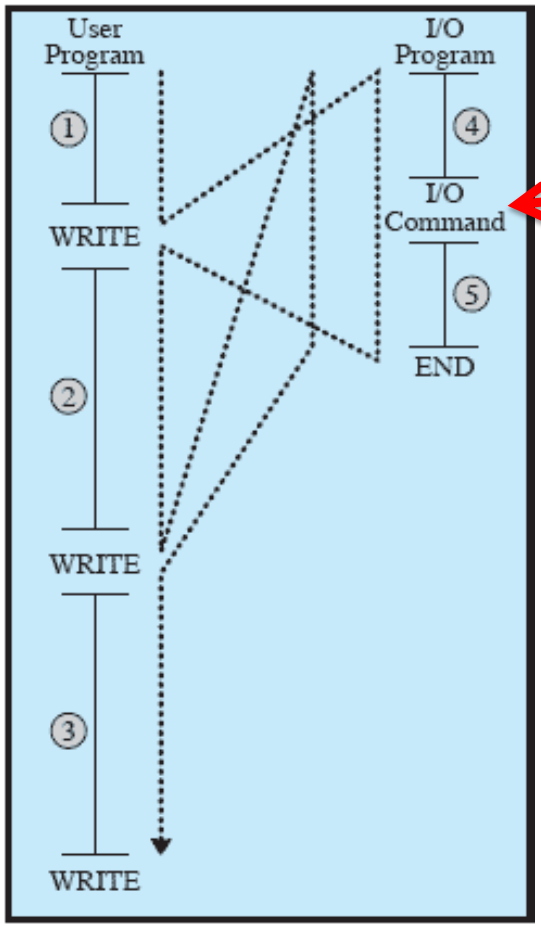
Table 1.1 Classes of Interrupts

Program	Generated by some condition that occurs as a result of an instruction execution, such as arithmetic overflow, division by zero, attempt to execute an illegal machine instruction, and reference outside a user's allowed memory space.
Timer	Generated by a timer within the processor. This allows the operating system to perform certain functions on a regular basis.
I/O	Generated by an I/O controller, to signal normal completion of an operation or to signal a variety of error conditions.
Hardware failure	Generated by a failure, such as power failure or memory parity error.

Examples: Brownout; Watchdog

Program Flow of Control

No interrupts.



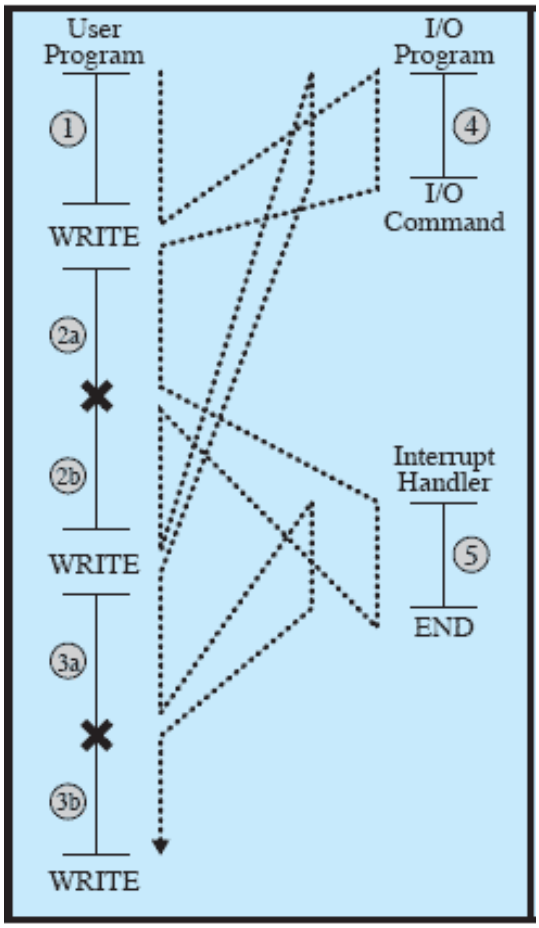
(a) No interrupts

Wait until completion Here.

In a program that doesnt have interrupts. We have a linear program that exectues, like the example before. When we need to interact with an IO device (say to read or write) so we need to do a subroutine call which involves spinning up some hardware (say a hard drive) so we are limited by the speed at which that call can be made (suuuper slow).

Program Flow of Control

Interrupt with short I/O wait.



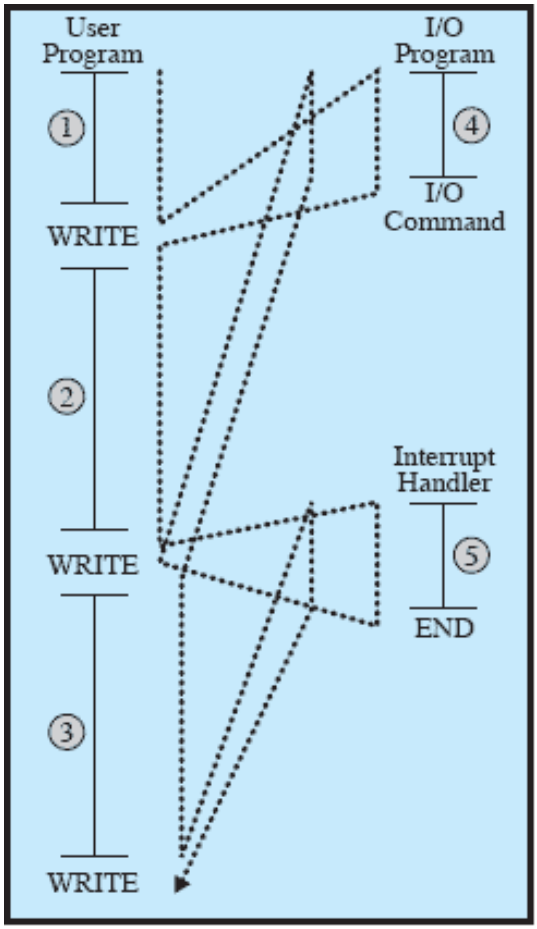
(b) Interrupts; short I/O wait

We isolate the waiting portion allowing us to go back to the main program while we wait for the IO. The write operation only executes the set up part and immediatly returns to the program and exectures the rest of the program. When the IO is done with its shit it interrupts the program to call it to finish the execution of that call now that the data needed is there. Then the program finishes doing its thing.

Program Flow of Control

Interrupt with long I/O wait.

Need to wait for first I/O here.



(c) Interrupts; long I/O wait

We can run into problems when we have multiple calls to the IO in a short period of time which means that we need to wait for the first call to finish before the second one can go. You can't really make a queue at the IO of commands, so the program grids to a halt.

Need to check the disk status before attempting to read or write.

Interrupt Stage

- Processor checks for interrupts
- If interrupt
  - Suspend execution of program (store snapshot!!)
  - Execute interrupt-handler routine
  - Resume execution of program
- This is a simplified version:
  - Nested interrupts?
  - Interrupt priorities?
  - Counting interrupts?

When an interrupt happens we need to store a snapshot of the program so that we can return to the same state once the interrupt handler is done.

Transfer of Control via Interrupts

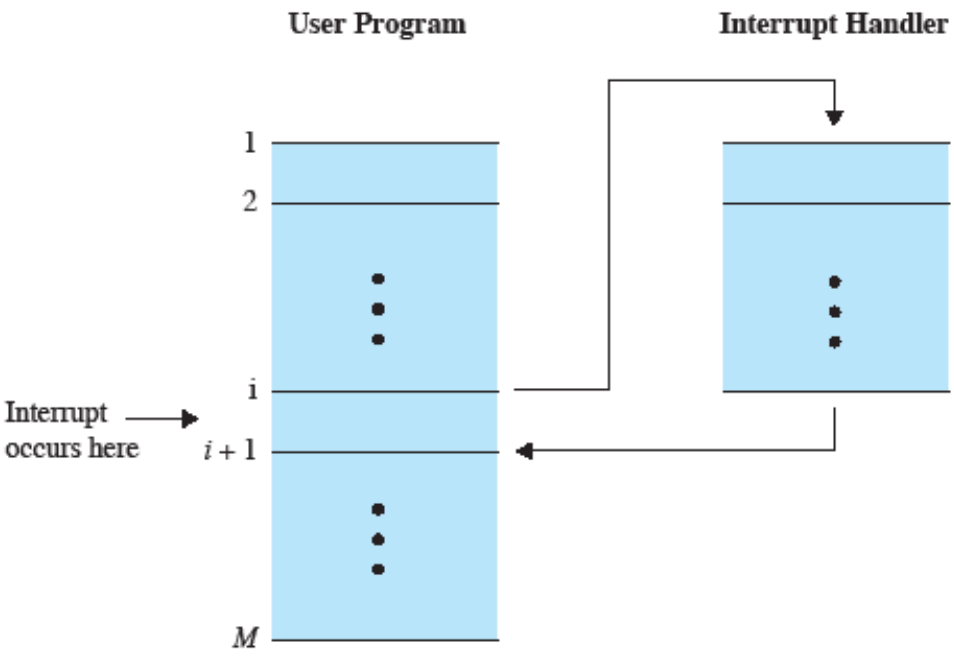


Figure 1.6 Transfer of Control via Interrupts

The interrupt handler should save all the state of the program and then return that state once its done dealing with its shit. It also needs to reset the program status word and the pc value. The program status word contains the flags from the operation and a whole bunch of shit it didnt get since he talks to quickly. Basically this guy is really important to restore.

Simple Interrupt Processing

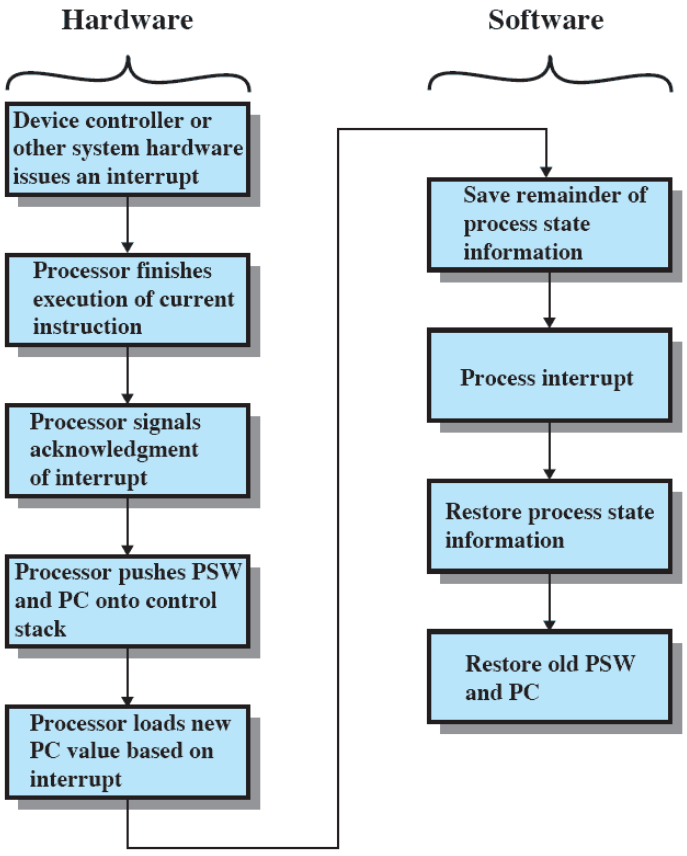


Figure 1.10 Simple Interrupt Processing

Discuss:
Lean ISR vs heavy ISR.

Project:
- Exception stack frame
- RTI instruction

… ARM TRM

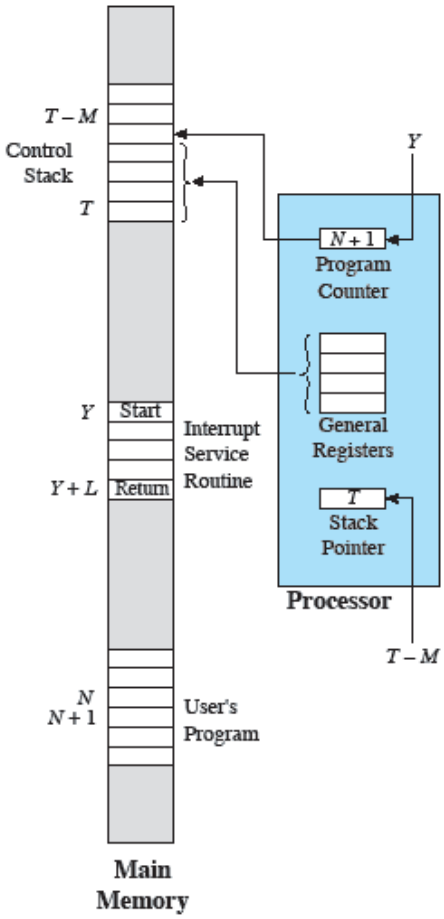
The first thing that happens, before the interrupt handler, is pushing the PSW onto the stack (remember stack pushes downward in memory). Next we push on the program counter, counts as the state of the program since this is just a pointer to it. Now we can enact the interrupt service routine. A interrupt vector is a table numbering all of th interrupts within which are the starting addresses of each of the interrupt service routines that service it. To start the interrupt handler make the PC equal to that address. We cannot start that yet though since there is still a ton of data unsaved (local variables and such). The hardware used in this course does some of this for us in the exception stack register (not fully sure on the name). Once this is pushed onto the stack we can safely start the interrupt handler.

Once the interrupt service routine we write all of the general registers used by the program off of the execution stack in reverse order. Return from interrupt (RTI) on ARM restores the PC and PSW for us.

If you have any stateful libraries (things that need to know what happened to the past) you need to store their state onto the stack so that they dont lose their place in the interrupt. Basically if you use the memory, you need to store it (store all data registers).

An Interrupt Occurs

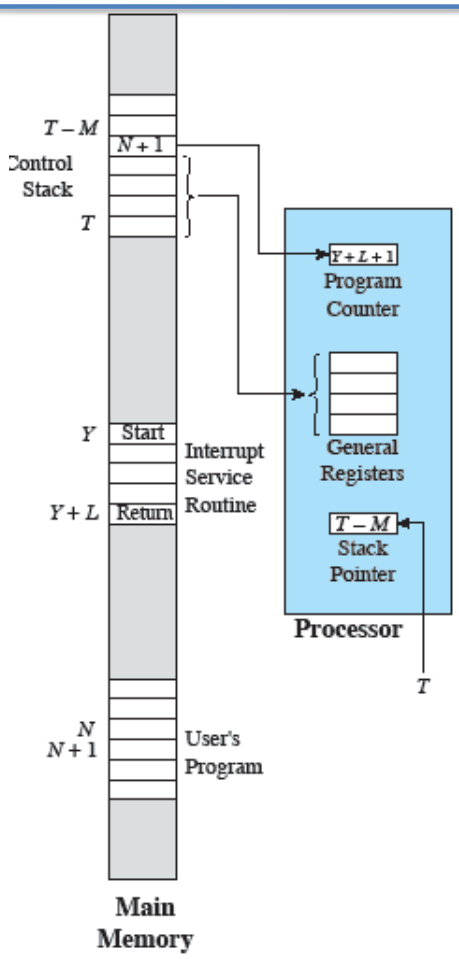
Storing a snapshot.



(a) Interrupt occurs after instruction at location N

Returning from an Interrupt

Restoring from a snapshot.



(b) Return from interrupt

When an interrupt occurs, store a snapshot by saving the necessary registers. When you return from it make sure you pop in reverse order

Multiple Interrupts?

- Multiple I/O, timers, ADC, PWM settings, RS232, CAN, EEPROM, I2C, etc.
- Approach 1:
  - Disable interrupts in ISR
  - No priority information
  - Not useful for time critical elements
- Approach 2:
  - Priorities for ISR
  - How to assign priorities
  - finite number of priority levels

When you have multiple interrupts the interrupt vector already prioritizes interrupts so when multiple interrupts occur you can make sure that the top one is executed or you can have nested interrupts that each defer things as they go to the next highest in the list. I think

Simple Interrupt Processing

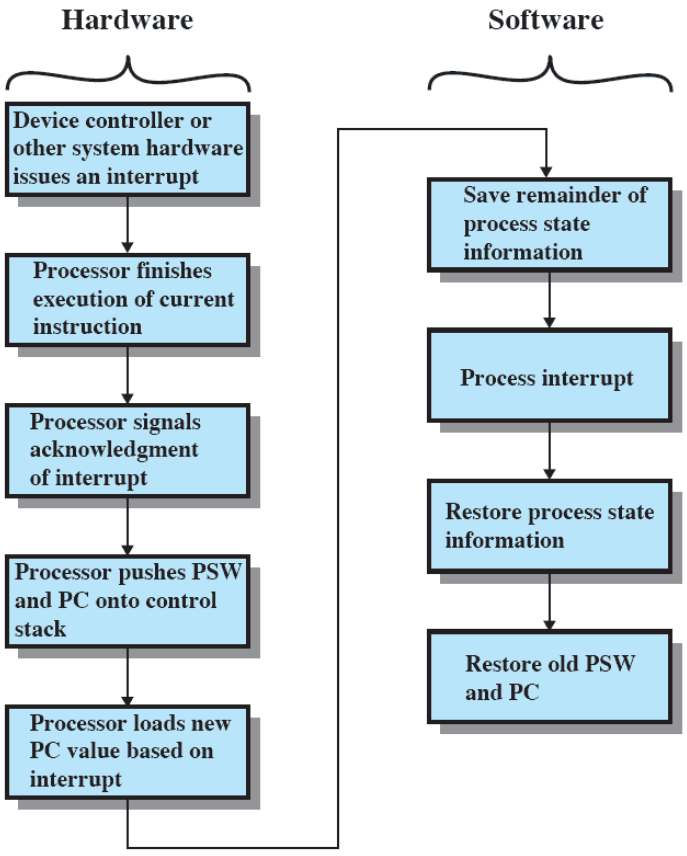


Figure 1.10 Simple Interrupt Processing

Discuss:
Lean ISR vs heavy ISR.

Project:
- Exception stack frame
- RTI instruction

… ARM TRM

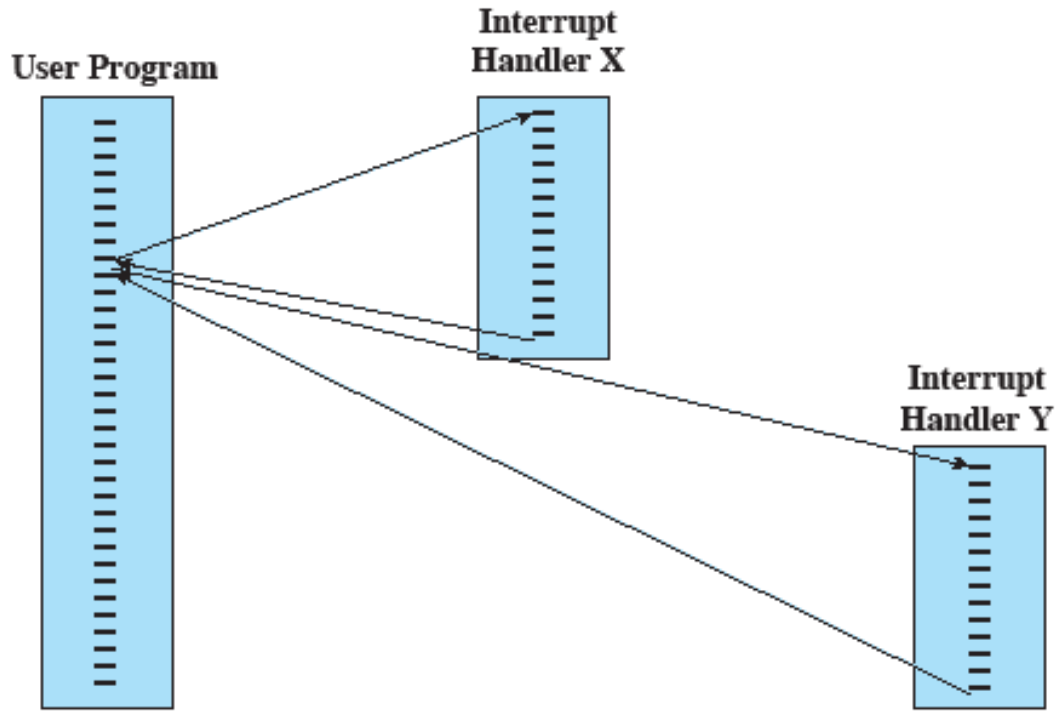
This page is very important to know for the exam. Hardware support ends after the PC is moved to point at the interrupt handler. The software needs to handle the saving and restoring of data and PC/PSW when its done.

```
void UART_ISR() {
    prologue //save all data and necissary registers (push R0 ... R15)
    ...
    compiled c compiled
    ...
    epilogue //reload all data and registers (pop R15 ... R0)
    RTI
}
```

Modern computer architectures don't really care about assembly and assume you use c programming. This means that the hardware will do alot of the saving of registers for us.

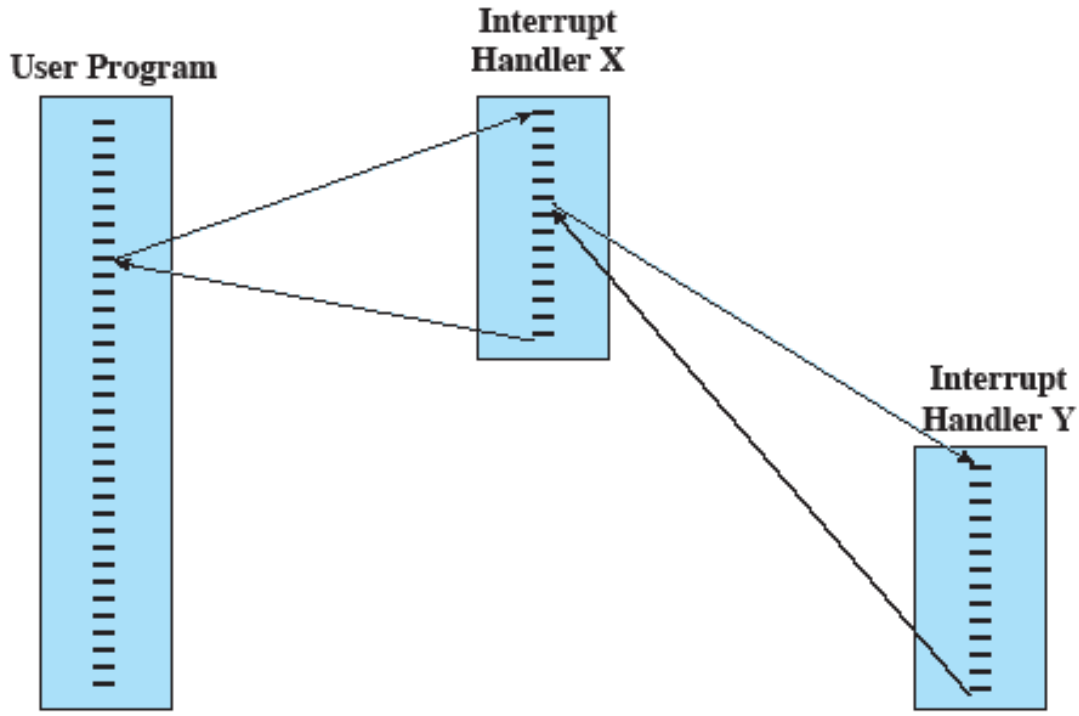
1. hardware pushes R0..R4, R13, R14 (called exception stack frame)
   (a) R15 = PC
   (b) R14 = Link Register
   (c) R13 = Stack Pointer
2. writes a special value in the link register to 0xffff
3. execute interrupt service routine (the steps below are baked onto the board)
   (a) prologue of ISR (pushing of registers) is usually a c function, described in standard for ARM
   (b) store R4-R12
   (c) exectute handler
   (d) store the result of the funtion in R0
   (e) pop R4-R12
   (f) return to code (BX LR)

Sequential Interrupt Processing



(a) Sequential interrupt processing

Nested Interrupt Processing



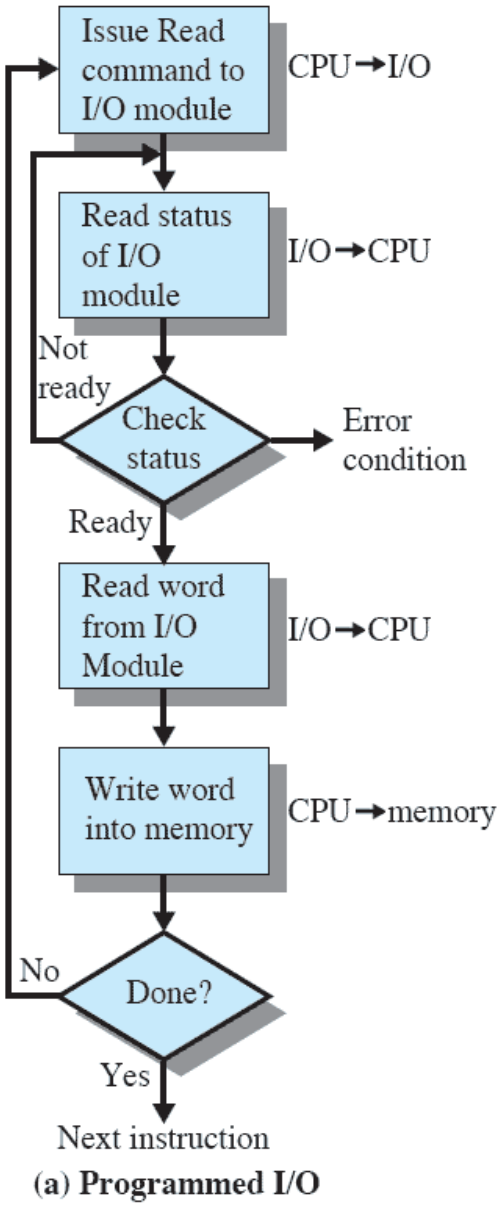
(b) Nested interrupt processing

Multiprogramming

- Long I/O requests still leave the processor idle → run multiple programs
- Processor has more than one program to execute
- The sequence the programs are executed depend on their relative priority and whether they are waiting for I/O
- After an interrupt handler completes, control may not return to the program that was executing at the time of the interrupt

Programmed I/O

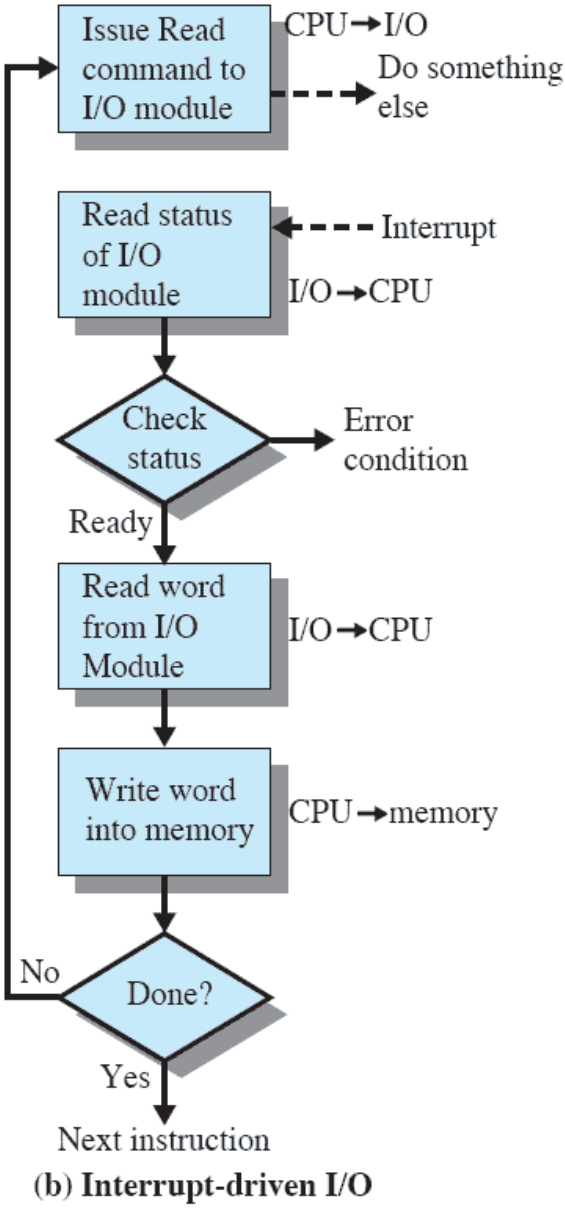
- I/O module performs the action, not the processor
- Sets the appropriate bits in the I/O status register
- No interrupts occur
- Processor checks status until operation is complete
- Called busy waiting



(a) Programmed I/O

Interrupt-Driven I/O

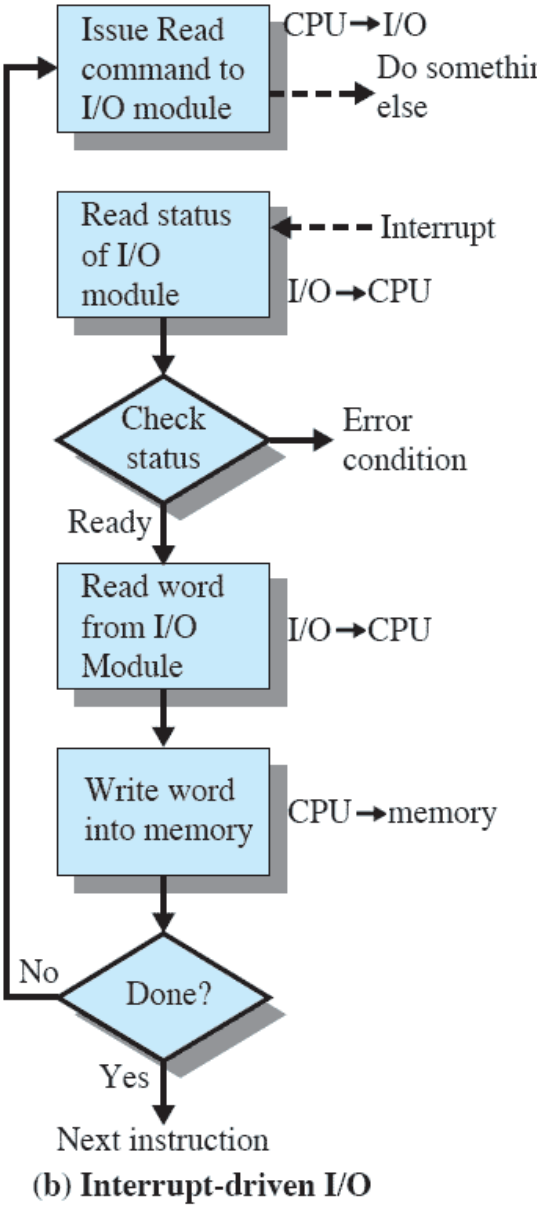
- Processor is interrupted when I/O module ready to exchange data
- Processor saves context o program executing and begins executing interrupt- handler



(b) Interrupt-driven I/O

Interrupt-Driven I/O

- No needless waiting
- Consumes a lot of processor time compared to DMA, because every word read or written passes through the processor

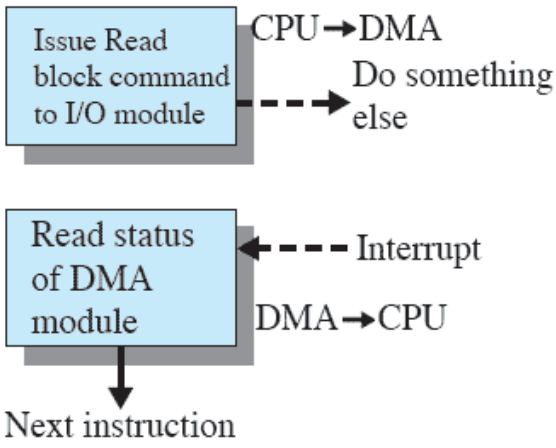


(b) Interrupt-driven I/O

1. Start IO read command
2. Return to program
3. program is interrupted when the IO read is done and ready to be used

Direct Memory Access

- Transfers a block of data directly to or from memory
- An interrupt is sent when the transfer is complete
- More efficient
- Not always available (e.g., external peripherals)



(c) Direct memory access

Memory Hierarchy

- Major constraints in memory
  - Amount
  - Speed
  - Expense
- Memory must be able to “keep up” with processor
- Memory cost must be reasonable

Memory Hierarchy

- Traversing the hierarchy
  - Decreasing cost per bit
  - Increasing capacity
  - Increasing access time
  - Decreasing frequency of access

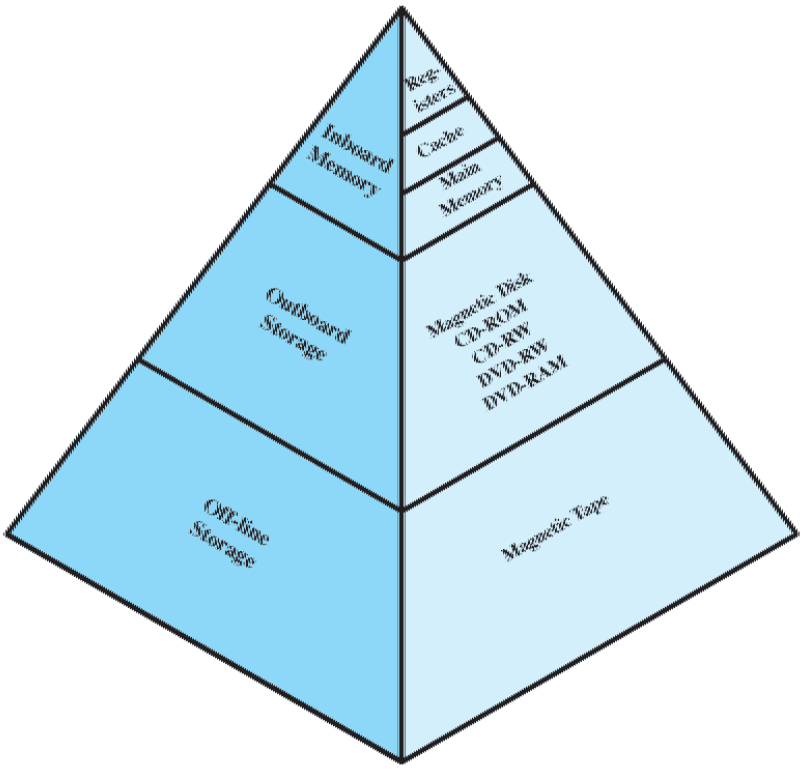


Figure 1.14 The Memory Hierarchy

Principle of Locality

- Memory references by program tend to cluster
- Objective:
  - Organize data s.t. accesses on each level are substantially less than on the next level of the memory hierarchy
  - Can be applied to multiple levels
    - Register file
    - Processor cache(s) & core-coupled memory
    - SRAM
    - Disk (more on that later)

We want to organize the access that we need to make to be as little as possible (use fast memory whenever possible) by abusing the fact that memory acesses by the user ten to cluster, so when we access slooooooow memory we want to return a lot of data so that we dont have to access it as often.

Cache Memory

- Invisible to OS
- Interacts with other memory management HW
- Processor accesses memory at least once per instruction cycle
- Processor execution is limited by memory cycle time
  - Hint: Figure out what “flash wait states” are for your project MCU
- Exploit the principle of locality with small, fast memory