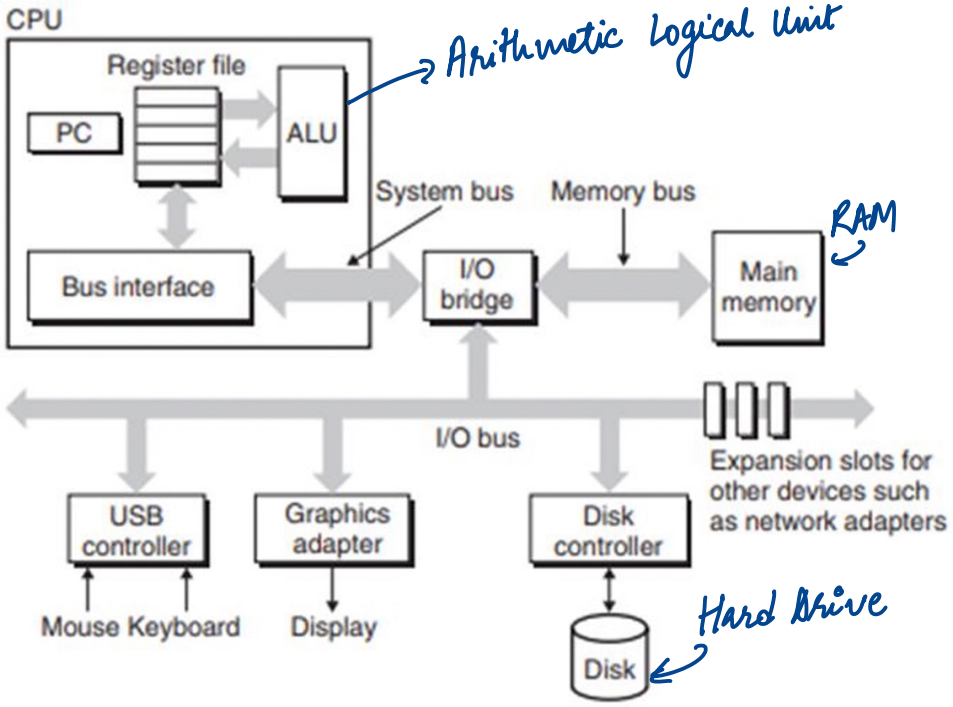
LECTURE 1

Hardware org. of a Computer

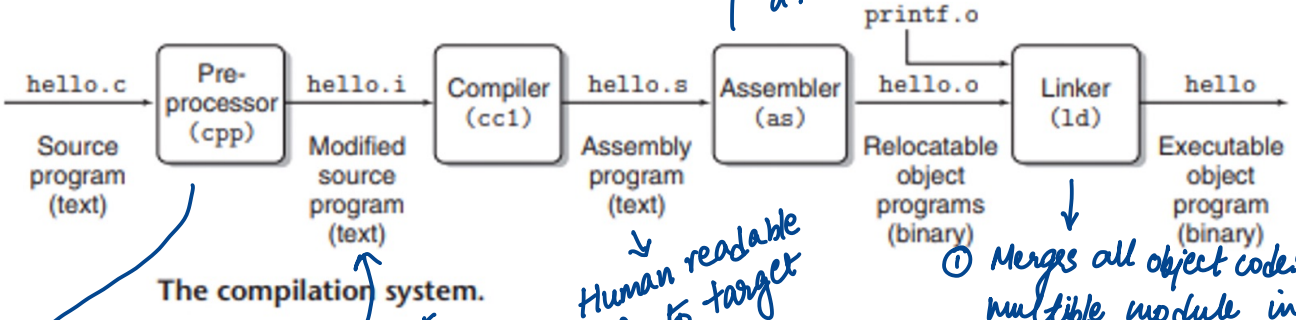


C Program Running on Computer / Compiler


```
#include <stdio.h>

int main()
{
    printf("hello, world\n");
}
```

Converts assembly code to binary / machine code a.k.a. OBJECT CODE



The compilation system.

what it does:
1. Removes Comments
2. Expands Macros (also #define values)
3. Expand <include> files
   ↳ like <stdio.h> etc.

results in

Human readable specific to target processor

① Merges all object codes from multiple module into a single one
② Library functions used are linked to our code

Static linking = copy function code to our code
dynamic linking = place name of library in binary file

https://medium.datadriveninvestor.com/compilation-process-db17c3b58e62 → link for above compilation material

VON NEUMANN ARCHITECTURE

TextBook 1; 1.3

- The most basic setup of computer designed first in 1946 forms the foundation of even current day machines.
- Characterized by:

HARDWARE

1. a CPU
2. main memory system
3. an I/O system

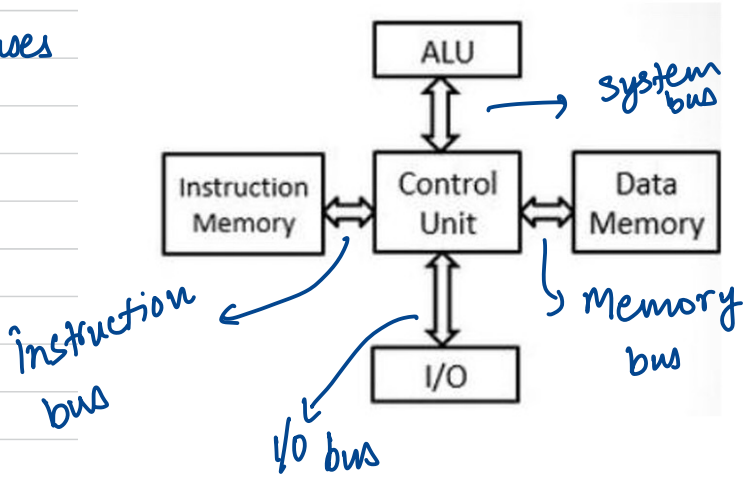
PROCESS

1. Data & instructions stored in single read-write memory
   i.e. No disk only RAM
2. Content of memory addressable by location. Data type not considered
3. Sequentially executed instructions (unless explicitly specified)
   i.e. No parallel processing
4. Single path (bus) b/w CPU & Main Memory
   - ↳ at a time either data flows from Memory to CPU or vice versa
   - ↳ called **von Neumann Bottleneck**
     - since throughput (or data transfer rate) is low.

→ To resolve the bottleneck, they came up with

HARVARD ARCHITECTURE

→ 2 separate Memory & buses
1. Instruction Memory
2. Data Memory



→ Now we see how a machine behaves when we give it an instruction.

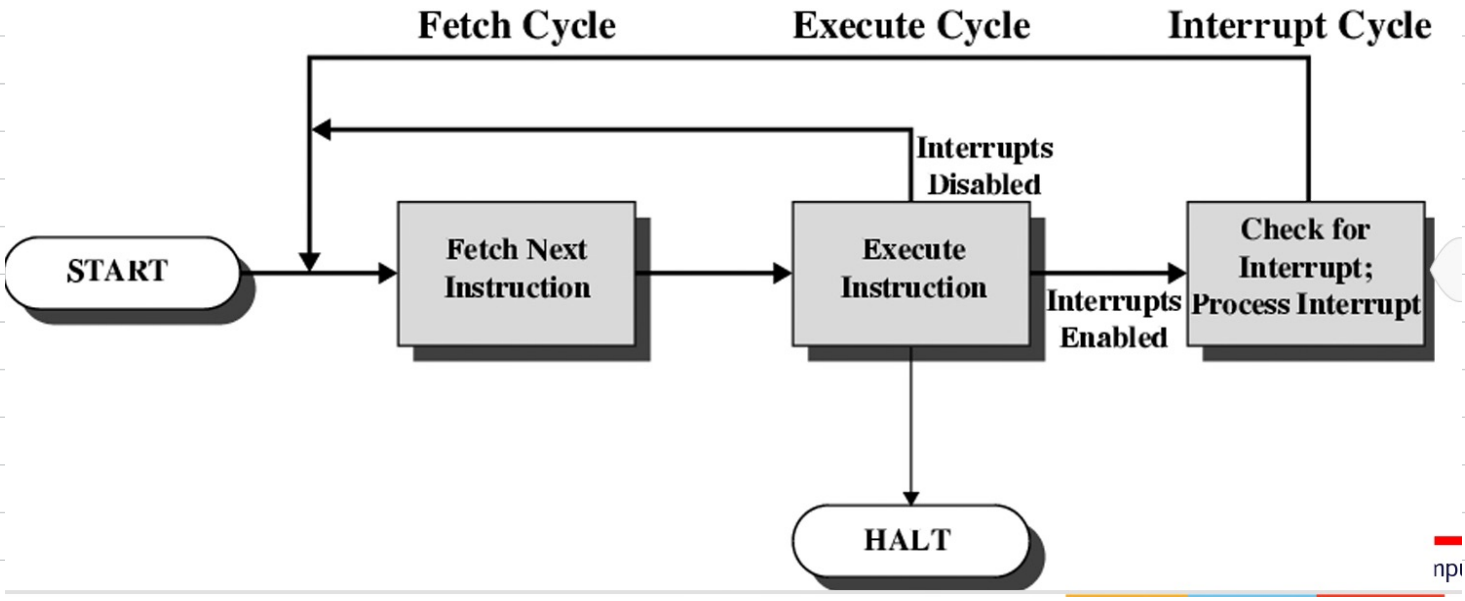
INSTRUCTION CYCLE

Textbook 1; 3.2

2 steps → FETCH
→ EXECUTE

at the end, check for INTERRUPT.

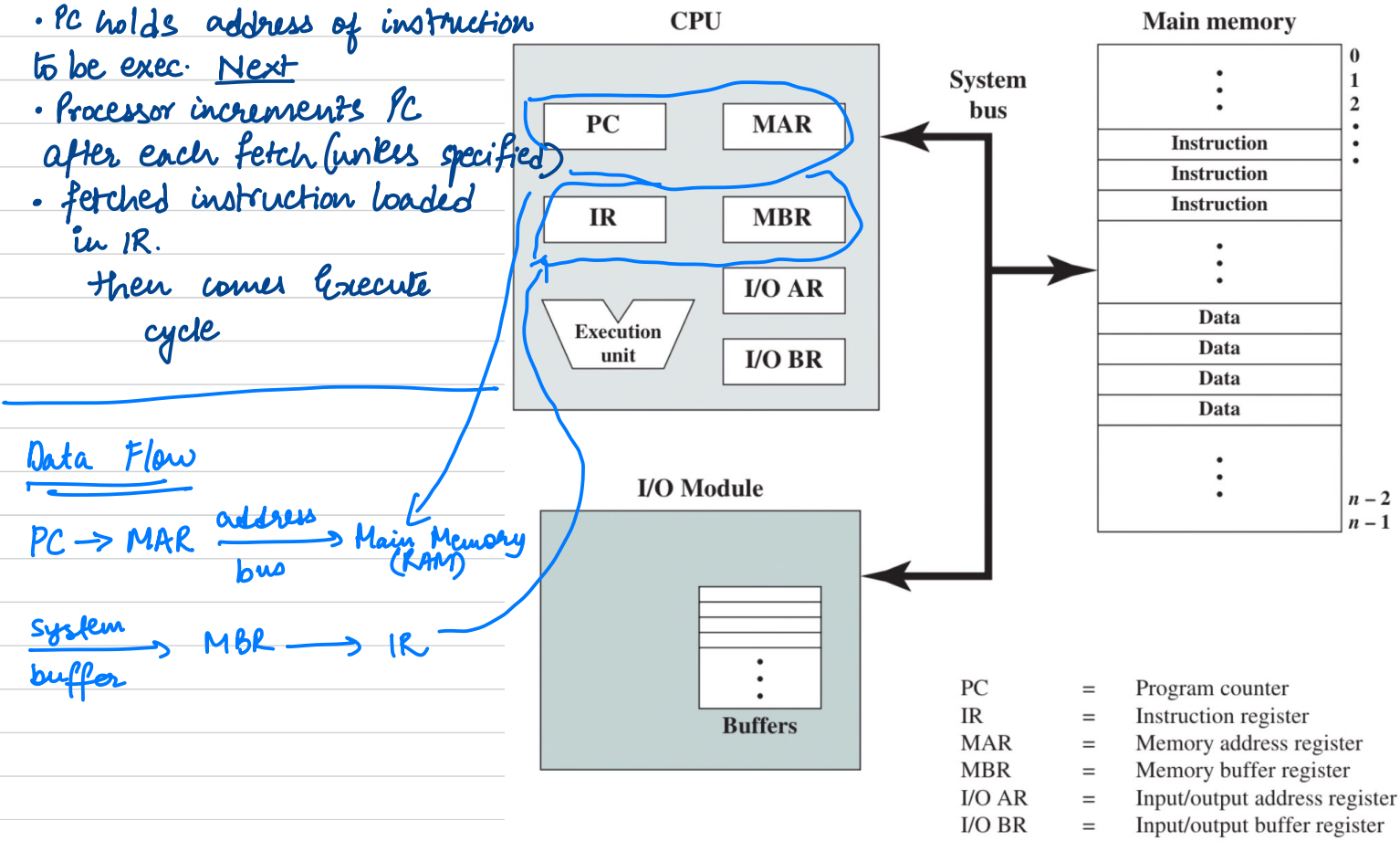
Instruction cycle diagram.



FETCH CYCLE

- PC holds address of instruction to be exec. Next
- Processor increments PC after each fetch (unless specified)
- fetched instruction loaded in IR.

then comes Execute cycle



Data Flow

PC → MAR → (address bus) → Main Memory (RAM)

System buffer → MBR → IR

Figure 3.2 Computer Components: Top-Level View

EXECUTE CYCLE

- Processor interprets instruction in IR & executes.
- Type of instructions :-
  1. Processor - Memory : Data from memory to Processor or vice versa
  2. Processor - I/O : Data from input (Keystroke, mouse click) to processor OR from processor to output (display screen, buzz on game controller)
  3. Data Processing : processor performs arithmetic / logical ops. on data.
  4. Control : the instruction that can modify the address of next-to-be-fetched instruction. i.e. Think of GOTO statement in C

- So instruction can be a combo of above.

involved in exchange b/w processor & memory / I/O

INSTRUCTION CYCLE STATE DIAGRAM.

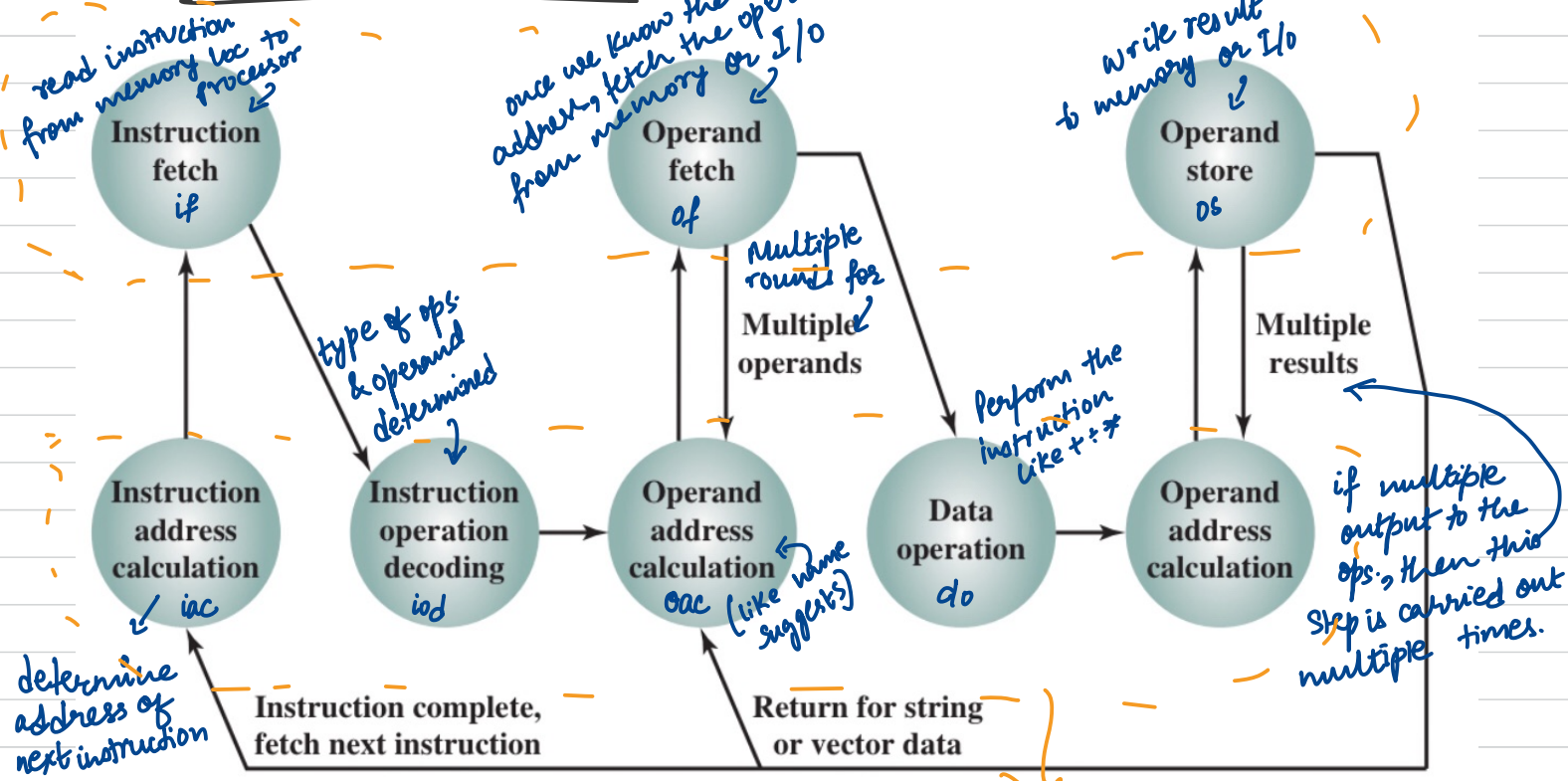


Figure 3.6 Instruction Cycle State Diagram

- For any given Instruction cycle, a state can be null OR a state can be visited multiple times
  (combo of different instruction types.)
- 'oac' appears twice ∵ instruction can have a read, a write, or both.

INTERRUPT CYCLE

- So far the processor has carried out linear execution of instructions. Now, we see what happens when it faces an interruption.
- Almost all machine components can generate an INTERRUPT.

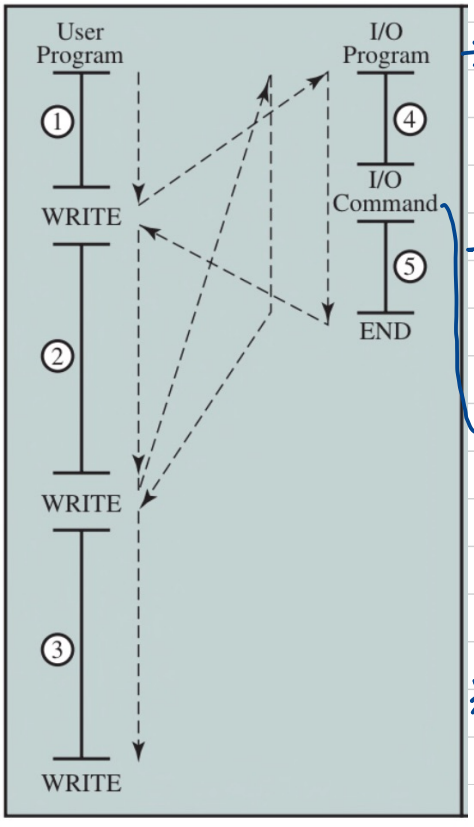
Classes of Interrupt :-

1. Program : mostly occurs in the course of an instruction execution.
   ex : division by zero, arithmetic overflow, reference outside a user's allowed memory
2. Timer : Generated by a timer within processor.
   allows OS to perform regularly scheduled tasks.
3. I/O : Generated by an I/O controller
   ex : signal a normal completion of ops, request service from processor
4. Hardware failure : Generated by say power failure or check bit error in RAM (also called Memory parity error)

★ Interrupts are created to ↑ efficiency

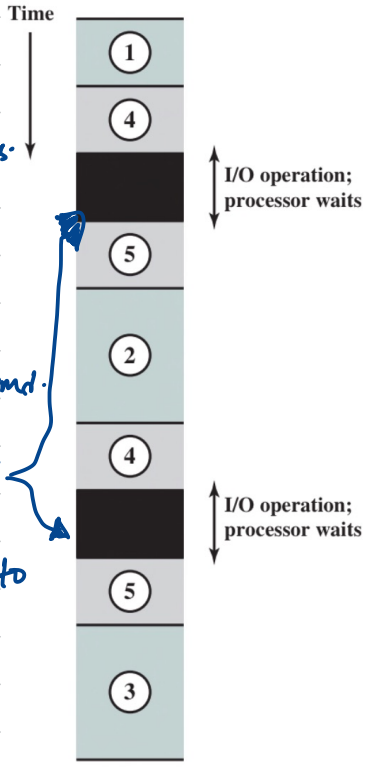
How?? A processor is usually faster than any external component. Say processor has to transfer data to printer using Instruction cycle. After each write, processor must pause for printer to catch up. Precious processing time is wasted.
Interrupt is designed to allow processor to move on to next Inst. (Instruction) while printer is trudging along

Program flow without Interrupts



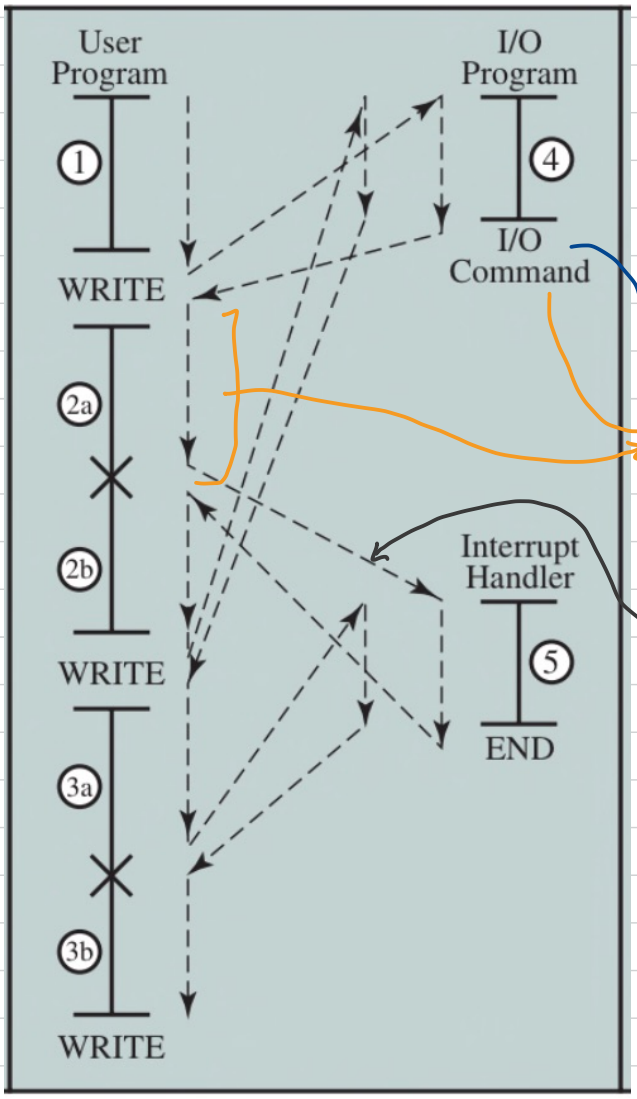
(a) No interrupts

→ ① ② ③ doesn't involve I/O only processor
→ WRITE calls are to an I/O prog. (a sys. utility)
→ I/O has 3 sections:
#1. ④ prepare for the actual I/O op.
Ex: copying data to special buffer
prep. parameters for device cmd.
#2. The actual I/O cmd.
w/o interrupts, the processor waits for this to be finished.
Processor may continue to run test to check if I/O cmd. is finished.
#3. ⑤ Complete the ops.
Ex: Setting a flag indicating Success/failure



(a) Without interrupts

Program with Interrupts

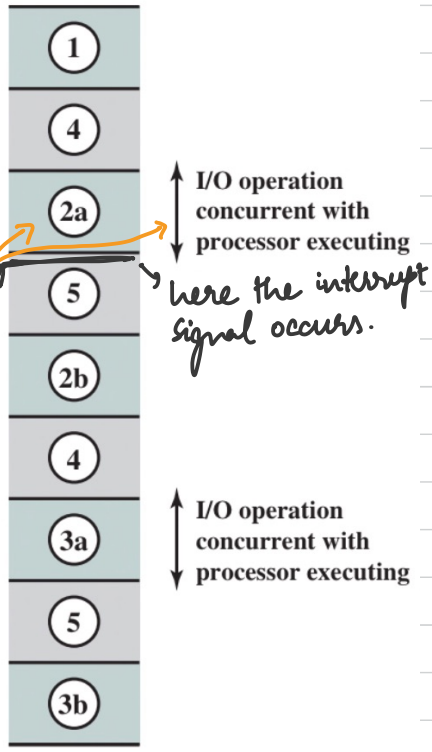


(b) Interrupts; short I/O wait

Now in this case control of program goes back to processor while I/O cmd. is being carried out. (#3. part in previous section)

These are happening concurrently

→ Once I/O controller is done, it sends an interrupt signal for processor to resume (5) and then (2b). This repeats again with the next WRITE



(b) With interrupts

→ Now let's see how the interrupt happens and what interrupt handler does.

Transfer of control during Interrupts

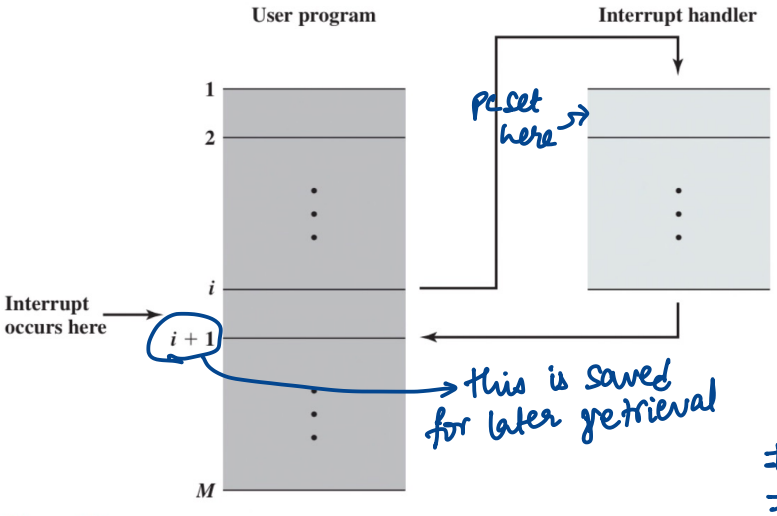


Figure 3.8 Transfer of Control via Interrupts

→ The Operating System / processor handles the whole interruption process

→ An interrupt cycle is added to inst. cycle (next page for new state diagram)

→ Now processor actively checks if any interrupt signal is present
if not, next operand is fetched & it carries on.
else if interrupt signal is detected
#1. Suspend current prog. exec.
#2. Save its context i.e. next inst. address & any other processor parameter
#3. sets prog. counter to starting address of interrupt handler

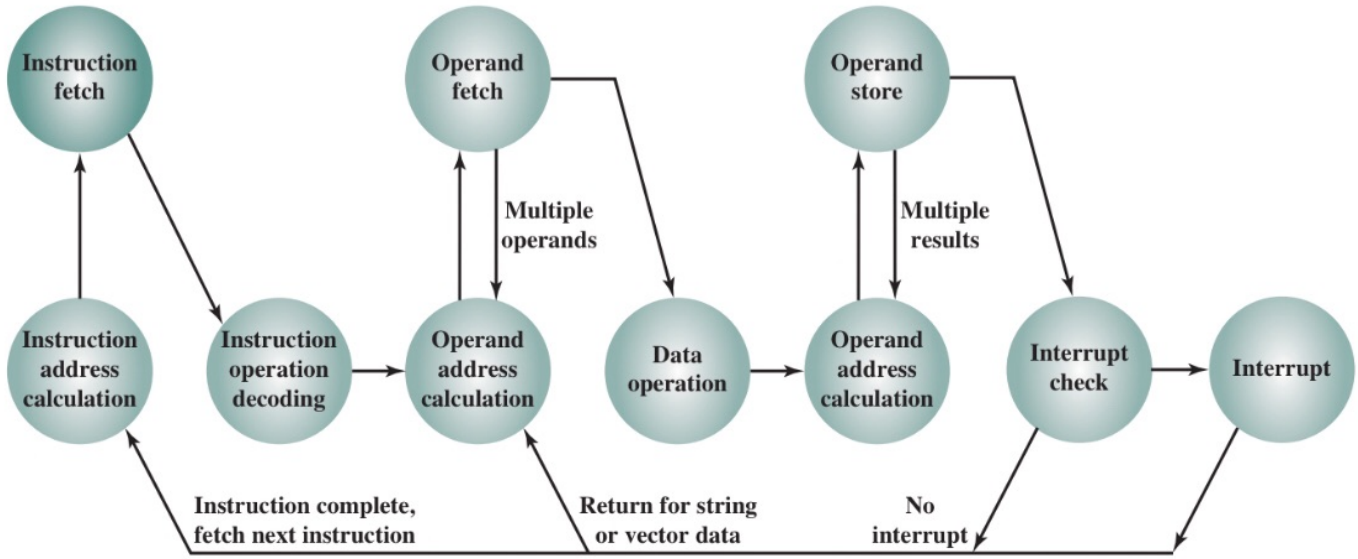


Figure 3.12 Instruction Cycle State Diagram, with Interrupts

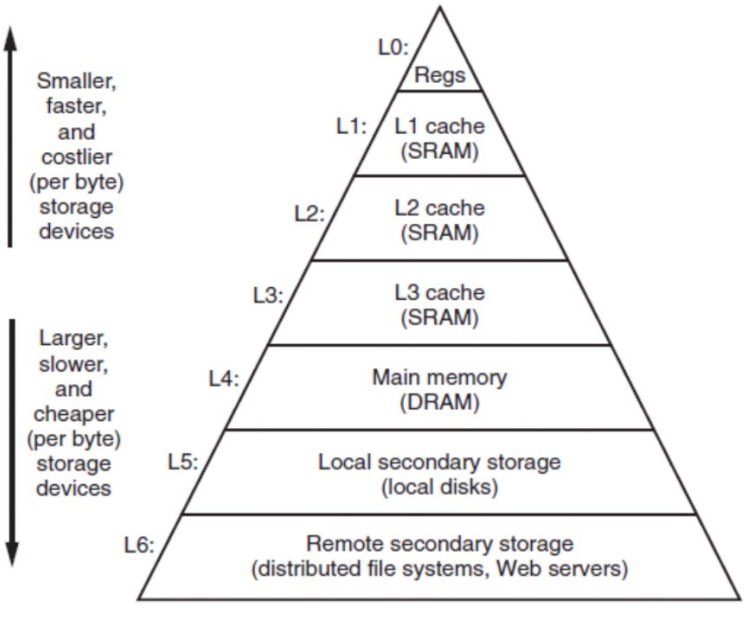
→ Now, (i) Fetch cycle begins for instructions in Interrupt handler.
- determines the type of interrupt
- knows what to do.

(ii) Execute cycle runs to process interrupt

(iii) Control goes back to processor, where it retrieves the saved content & resumes processing.

* For multiple interrupts, they are put in a queue. Processor services them by various scheduling algo (TO BE COVERED LATER)

Memory hierarchy



An example of a memory hierarchy.

- Faster access time, greater cost per bit
- Greater capacity, smaller cost per bit
- Greater capacity, slower access time

- Hence smaller & faster memories serve as registers, buffers within the processors & located closer to the CPU as compared to bigger memory / disks.

Operating Systems — Class slides

→ Bunch of software / program that acts as a go-between the machine & user

- Kernel – The one prog. that's always running on the machine
  everything else is either Sys prog. (part of OS) or app. programs.
- bootstrap prog – loaded at power up from ROM / EPROM
  - a.k.a firmware
  - initializes the whole machine
  - loads OS Kernel & starts its exec.

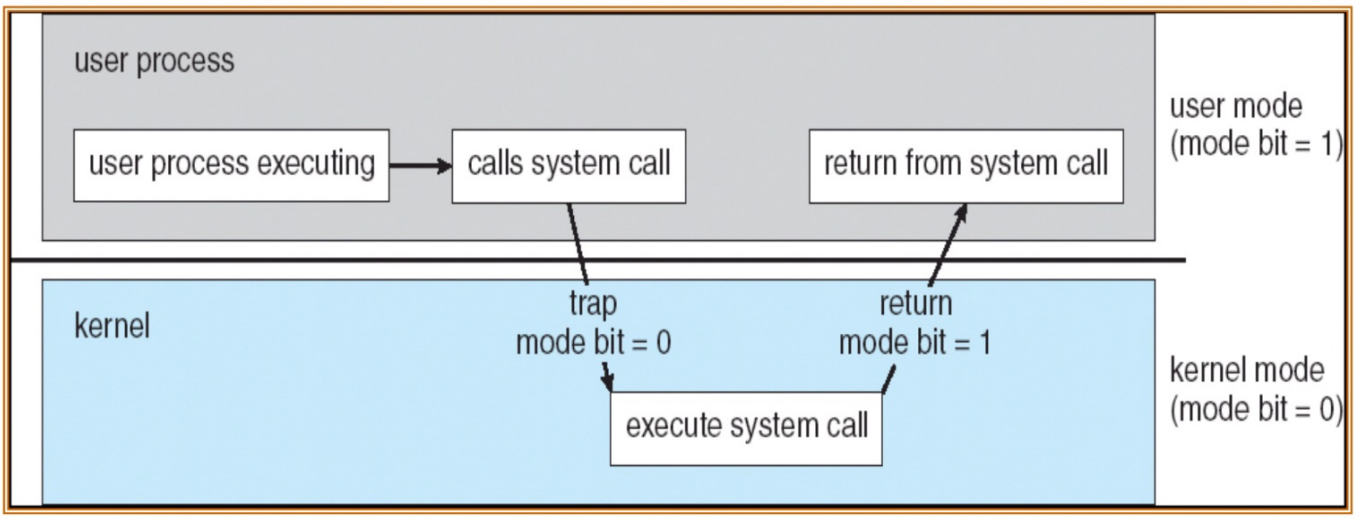
- Dual mode ops.

User mode

→ mode bit = 1
→ user prog. exec. in this mode
→ Some memory areas are protected from user access
→ certain instructions can't be exec. in this mode

Kernel mode

→ mode bit = 0
→ Sys. calls usually exec. in this mode
→ protected memory area is accessible
→ privileged instructions can be exec.



— END —