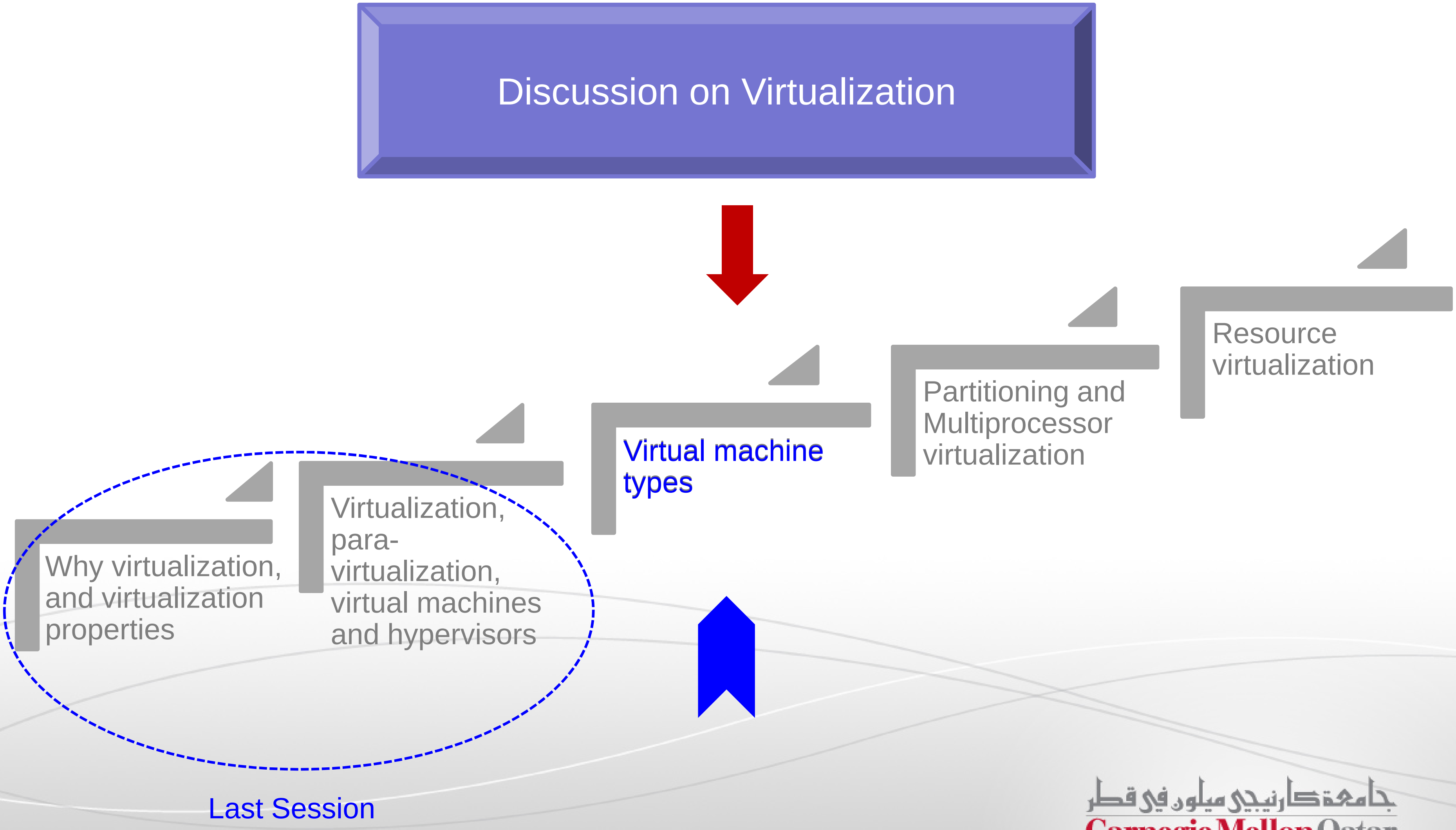
# Objectives



Discussion on Virtualization

Why virtualization, and virtualization properties

Virtualization, para-virtualization, virtual machines and hypervisors

Virtual machine types

Partitioning and Multiprocessor virtualization

Resource virtualization

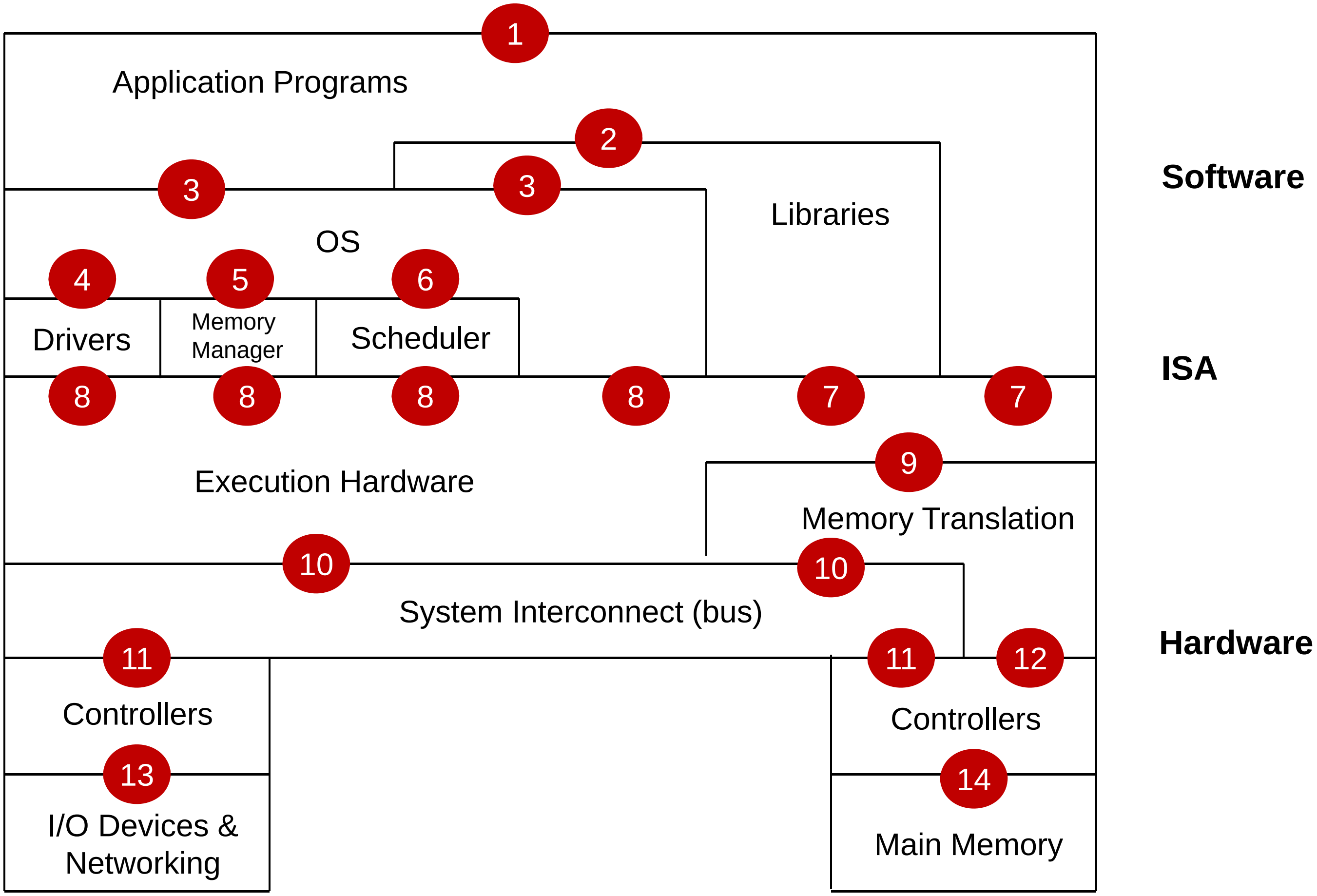
Last Session

# Background: Computer System Architectures

**Instruction Set Architecture (ISA): 7 & 8**

**Application Binary Interface (ABI): 3 & 7**

**Application Programming Interface (API): 2 & 7**



# Types of Virtual Machines

- As there is a process perspective and a system perspective of machines, there are also process-level and system-level VMs

- Virtual machines can be of two types:

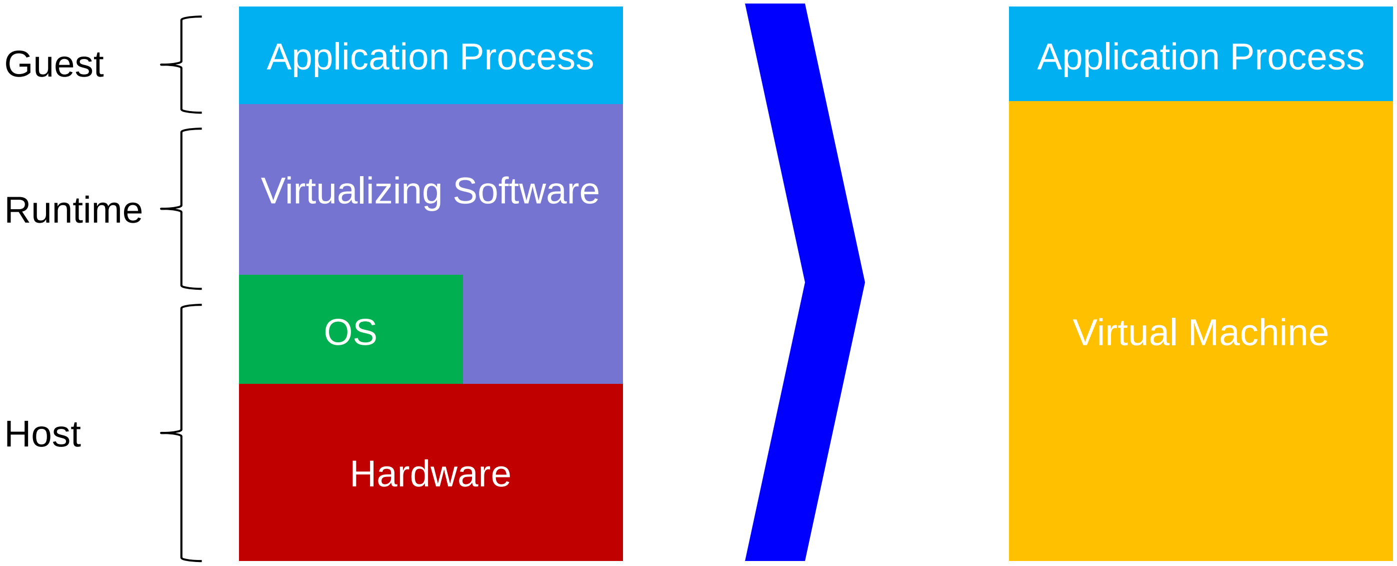
*1. Process VM*

- Capable of supporting an individual process

*2. System VM*

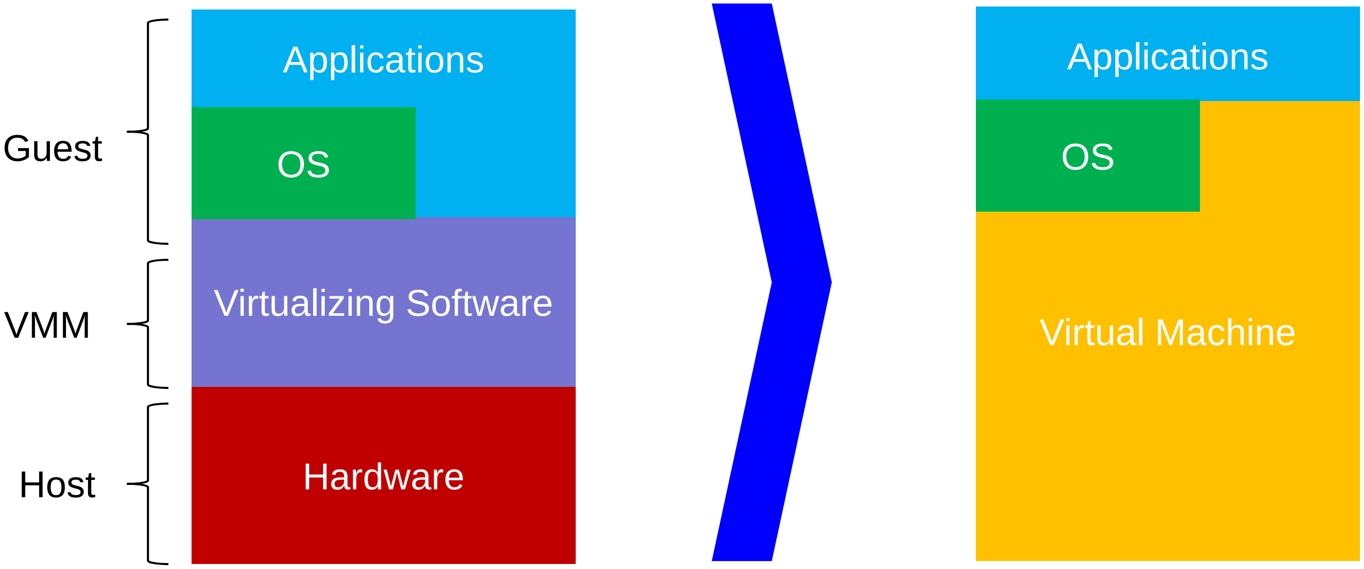
- Provides a complete system environment
- Supports an OS with potentially many types of processes

# Process Virtual Machine



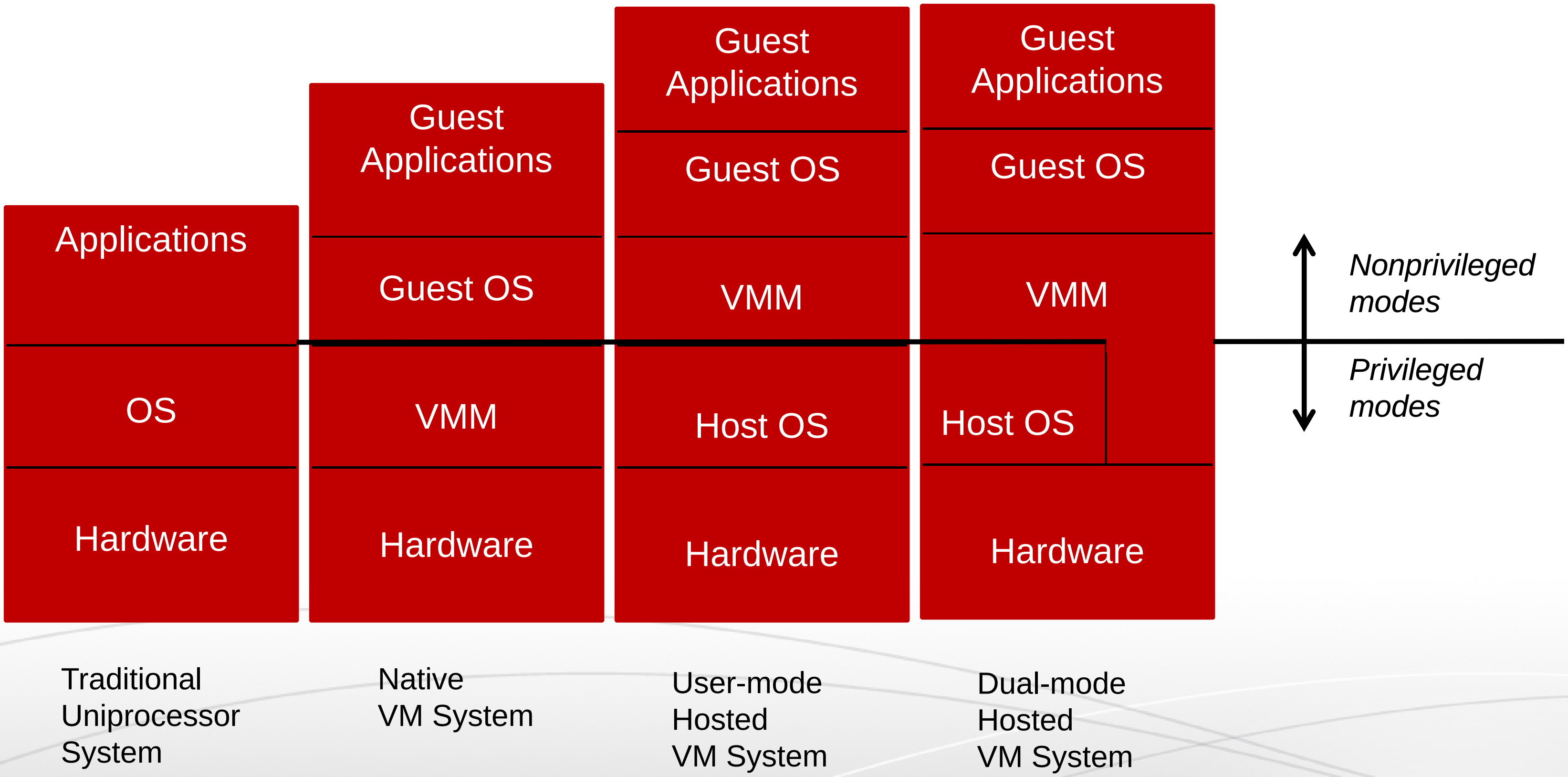
- ✓ Runtime is placed at the ABI interface
- ✓ Runtime emulates both user-level instructions and OS system calls

# System Virtual Machine



- ✓ VMM emulates the ISA used by one hardware platform to another, forming a system VM
- ✓ A system VM is capable of executing a system software environment developed for a different set of hardware

# Native and Hosted VM Systems



Applications

OS

Hardware

Traditional Uniprocessor System

Guest Applications

Guest OS

VMM

Hardware

Native VM System

Guest Applications

Guest OS

VMM

Host OS

Hardware

User-mode Hosted VM System

Guest Applications

Guest OS

VMM

Host OS

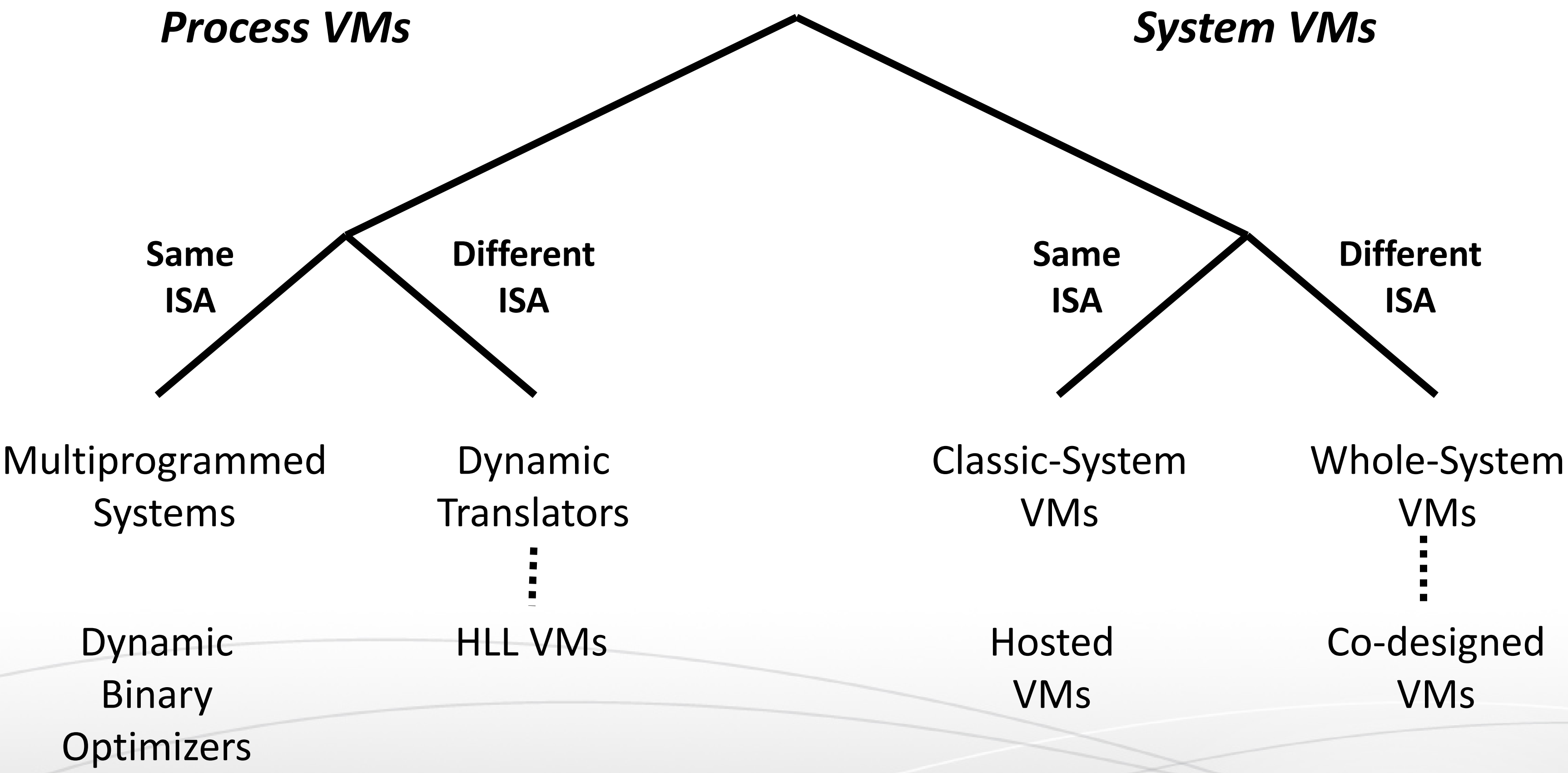
Hardware

Dual-mode Hosted VM System

*Nonprivileged modes*

*Privileged modes*

# A Taxonomy



*Process VMs*

*System VMs*

**Same ISA**

**Different ISA**

**Same ISA**

**Different ISA**

Multiprogrammed Systems

Dynamic Binary Optimizers

Dynamic Translators

HLL VMs

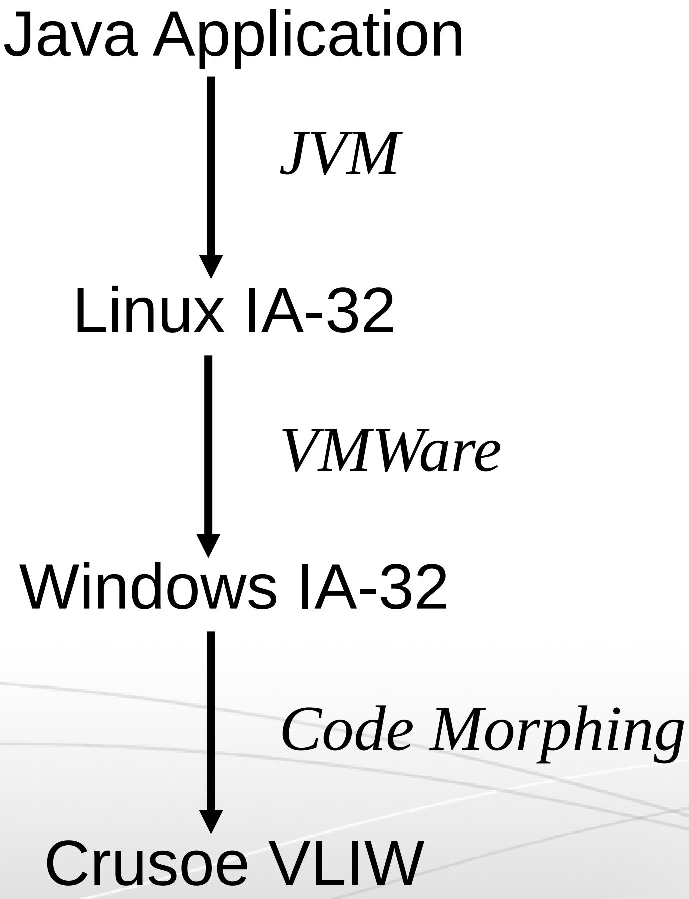
Classic-System VMs

Hosted VMs

Whole-System VMs

Co-designed VMs

# The Versatility of VMs



Java Application

*JVM*

Linux IA-32

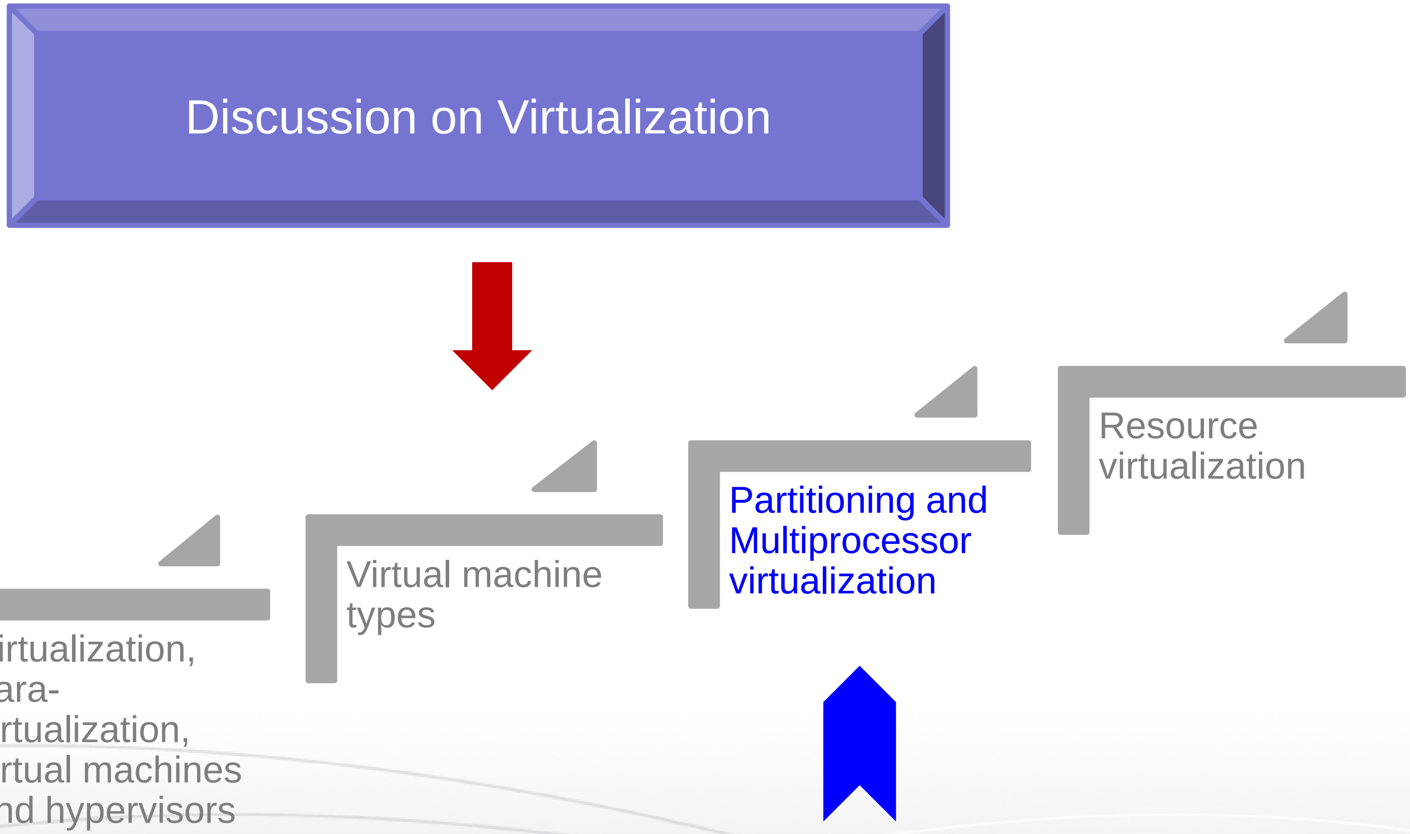
*VMWare*

Windows IA-32

*Code Morphing*

Crusoe VLIW

# Objectives



Discussion on Virtualization

Why virtualization, and virtualization properties

Virtualization, para-virtualization, virtual machines and hypervisors

Virtual machine types

Partitioning and Multiprocessor virtualization

Resource virtualization

# Multiprocessor Systems

- Multiprocessor systems might have 1000s of processors connected to TBs of memory and PBs of disk capacity

- Often there is a mismatch between the ideal number of processors an application needs and the actual number of physical processors available

- It is more often the case that applications cannot exploit more than a fraction of the processors available. The is mainly because of:

  - Limitations in the parallelism available in the programs

  - Limitations in the scalability of applications due to the overhead of communication between processors

# Partitioning

- The increasing availability of multiprocessor systems has led to the examination of techniques that can help *utilize* them more effectively

- Techniques have been developed in which the multiprocessor system can be partitioned into multiple partitions

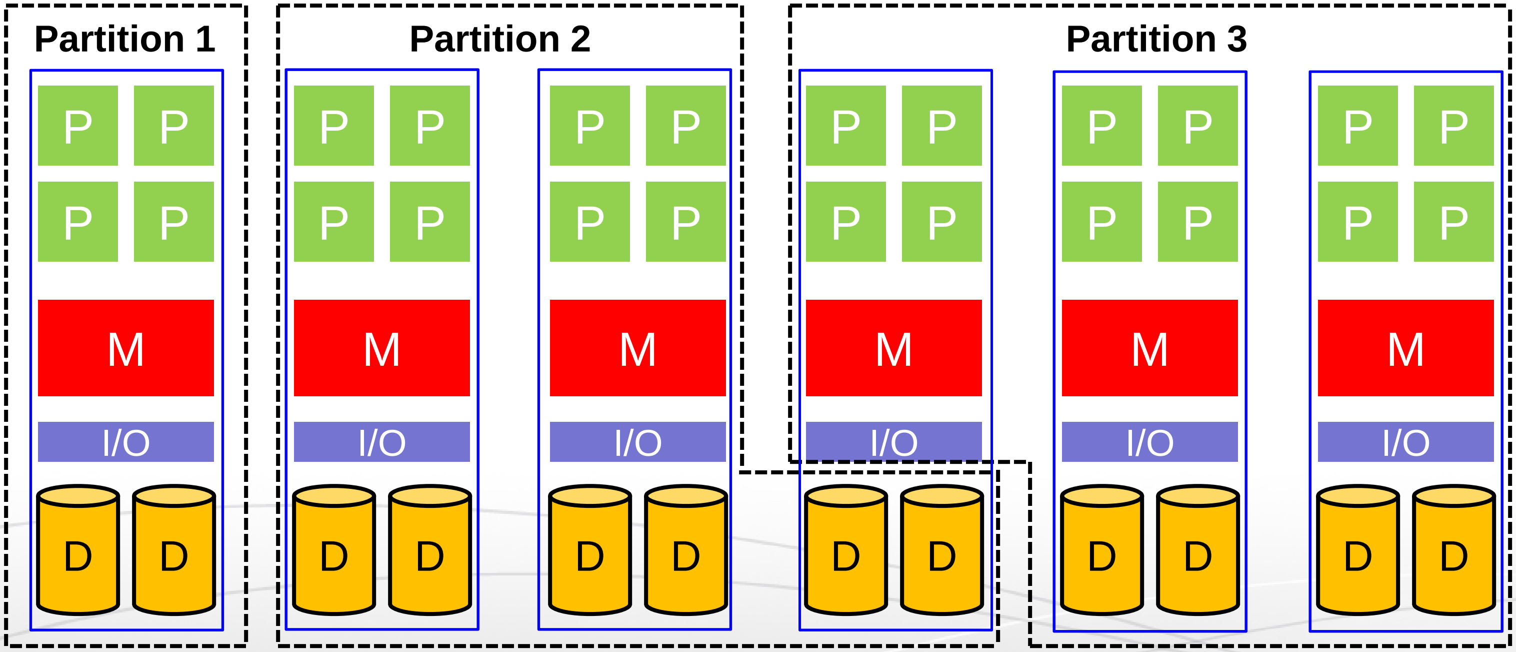
  - A partition is given a subset of the resources available on the system

- Hence, using partitioning, multiple applications can simultaneously exploit the available resources of the system

- Partitioning can be achieved:
  - Either in-space (referred to as physical partitioning)
  - Or in-time (referred to as logical partitioning)

# Physical Partitioning

- With physical partitioning, each partition is assigned resources that are physically distinct from the resources used by the other partitions



# Physical Partitioning

- Physical partitioning allows a partition to *own* its resources physically

- It is not permissible for two partitions to share the resources of a single system board

- Partitions are configured by a *central control unit* that receives commands from the console of the system admin and provisions hardware resources accordingly

- The number of partitions that can be supported in physically partitioned systems is limited to the number of available physical processors

# Physical Partitioning- Advantages

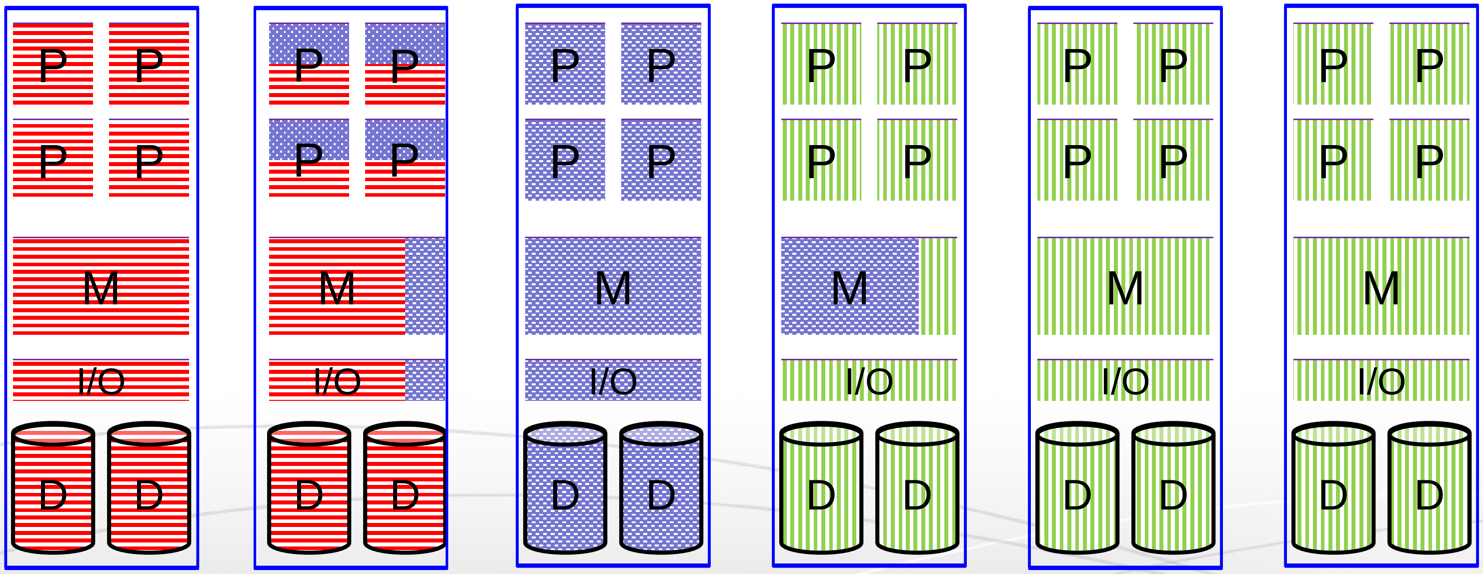
- Physical partitioning provides:
  - Failure Isolation: it ensures that in the event of a failure, only the part of the physical system that houses the failing partition will be affected
  - Better security isolation: Each partition is protected from the possibility of intentional or unintentional denial-of-service attacks by other partitions
  - Better ability to meet system-level objectives (these result from contracts between system owners and users of the system)
  - Easier management of resources: no need of sophisticated algorithms for scheduling and management of resources

# Physical Partitioning- Disadvantages

- While physical partitioning has a number of attractive features, it has some major disadvantages:
  - System utilization: Physical partitioning is probably not the ideal solution if system utilization is to be optimized
    - It is often the case that each of the physical partitions is underutilized
  - Load balancing: with physical partitioning, dynamic workload balancing becomes difficult to implement

# Logical Partitioning

- With logical partitioning, partitions share some of the physical resources, usually in a *time-multiplexed* manner



# Logical Partitioning

- With logical partitioning it is permissible for two partitions to share the resources of a single system board

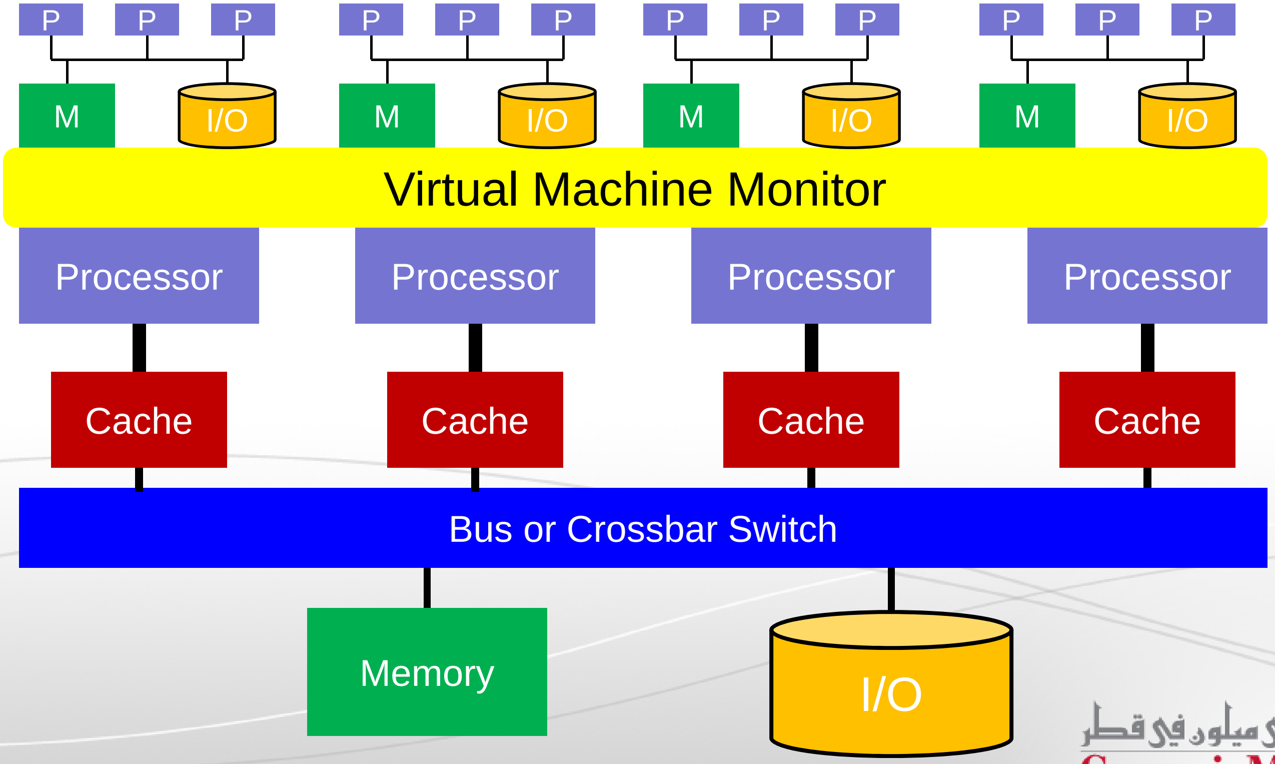
- Logical partitioning makes it possible to partition an ***n-way*** system into a system with more than ***n*** partitions, if so desired

- Logical partitioning is more flexible than physical partitioning but needs additional mechanisms to provide safe and efficient way of sharing resources

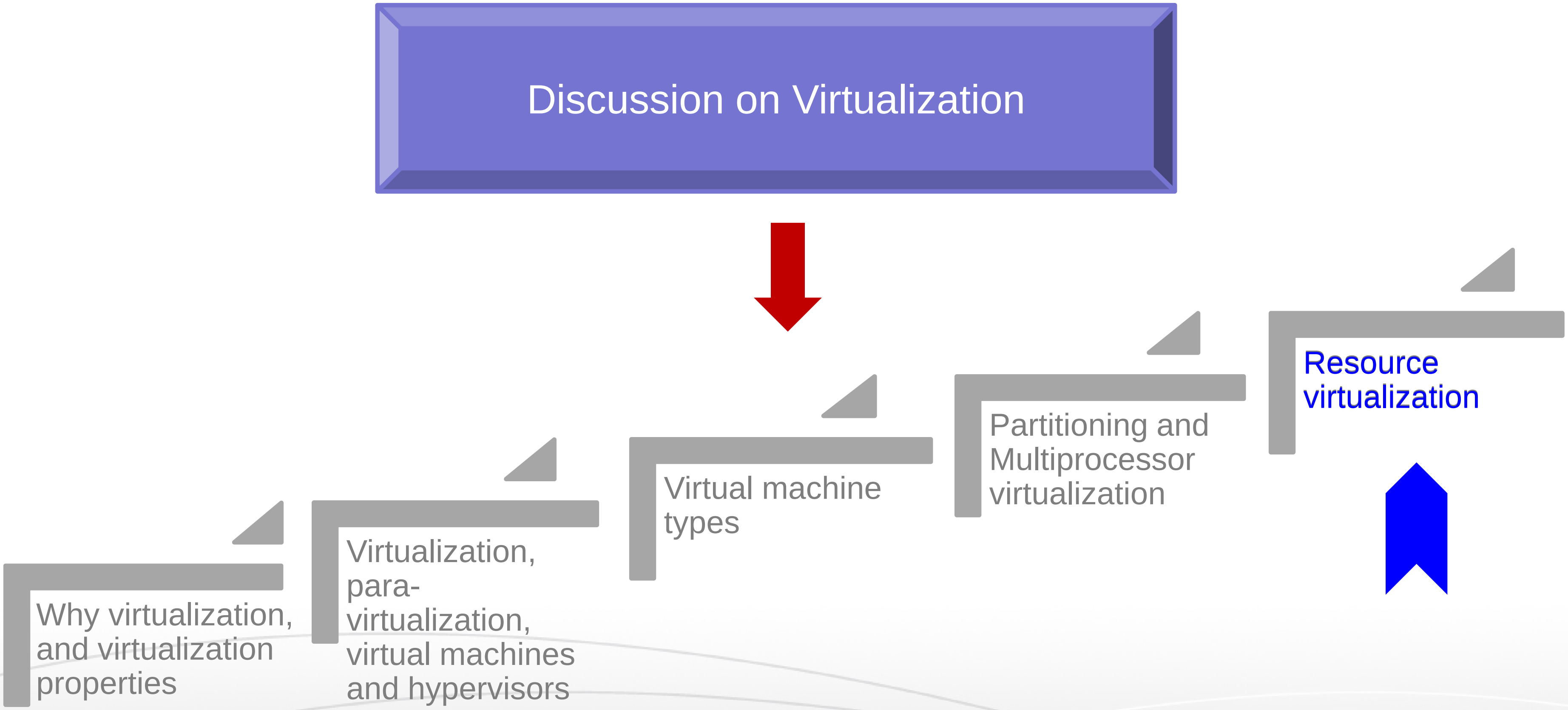
- Logical partitioning is usually done through a VMM or a hypervisor and provides what is referred to as *multiprocessor virtualization*

# Multiprocessor Virtualization

- A virtualized multiprocessor gives the appearance of a system that may or may not reflect the exact configuration of the underlying physical system



# Objectives



Discussion on Virtualization

Why virtualization, and virtualization properties

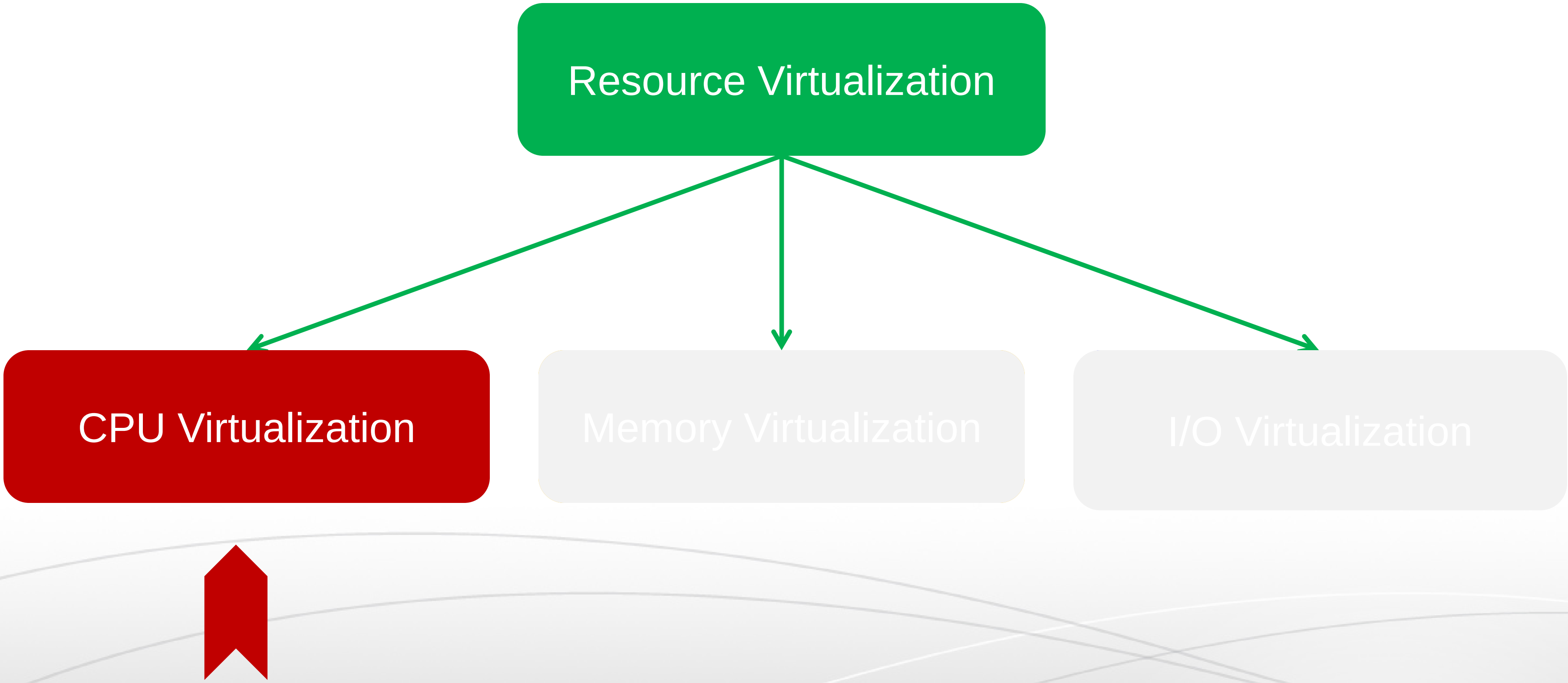
Virtualization, para-virtualization, virtual machines and hypervisors

Virtual machine types

Partitioning and Multiprocessor virtualization

Resource virtualization

# Resource Virtualization



Resource Virtualization

CPU Virtualization

Memory Virtualization

I/O Virtualization

# CPU Virtualization

- Interpretation and Binary Translation
- Virtualizable ISAs

# CPU Virtualization

- Interpretation and Binary Translation
- Virtualizable ISAs

# Instruction Set Architecture

- Typically, the architecture of a processor defines:

  1. A set of storage resources (e.g., registers and memory)
  2. A set of instructions that manipulate data held in storage resources

- The definition of the storage resources and the instructions that manipulate data are documented in what is referred to as Instruction Set Architecture (ISA)

- Two parts in the ISA are important in the definition of VMs:

  1. User ISA: visible to user programs
  2. System ISA: visible to supervisor software (e.g., OS)

# Ways to Virtualize CPUs

- The key to virtualize a CPU lies in the execution of the guest instructions, including both system-level and user-level instructions

- Virtualizing a CPU can be achieved in one of two ways:

1. Emulation: the only processor virtualization mechanism available when the ISA of the guest is different from the ISA of the host

2. Direct native execution: possible only if the ISA of the host is identical to the ISA of the guest

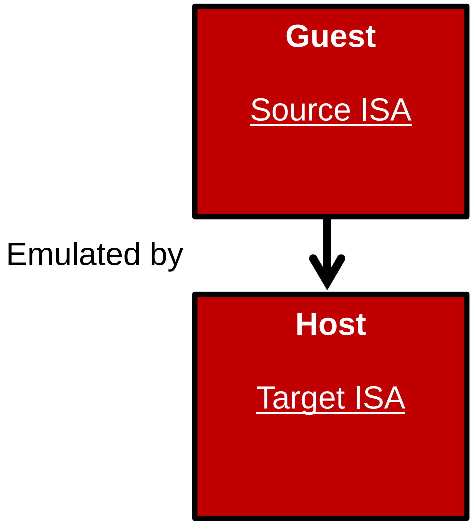
# Emulation

- Emulation is the process of implementing the interface and functionality of one system (or subsystem) on a system (or subsystem) having different interface and functionality

- In other words, emulation allows a machine implementing one ISA (the target), to reproduce the behavior of a software compiled for another ISA (the source)

- Emulation can be carried out using:

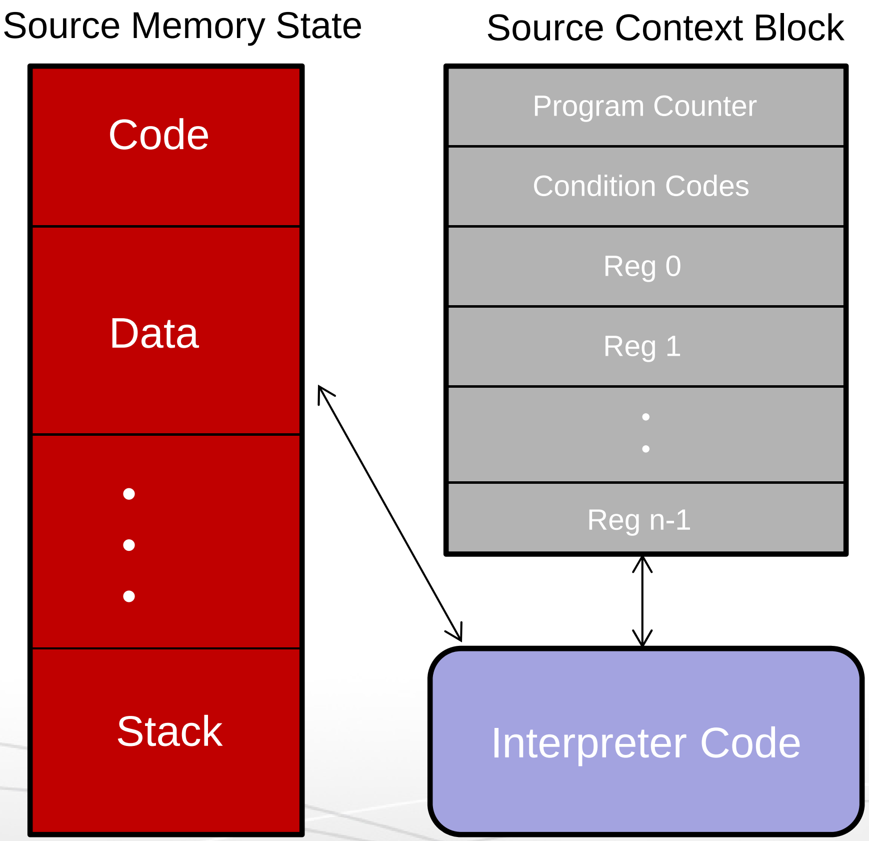
  1. Interpretation
  2. Binary translation



# Basic Interpretation

- Interpretation involves a 4-step cycle (all in software):

1. Fetching a source instruction
2. Analyzing it
3. Performing the required operation
4. Then fetching the next source instruction

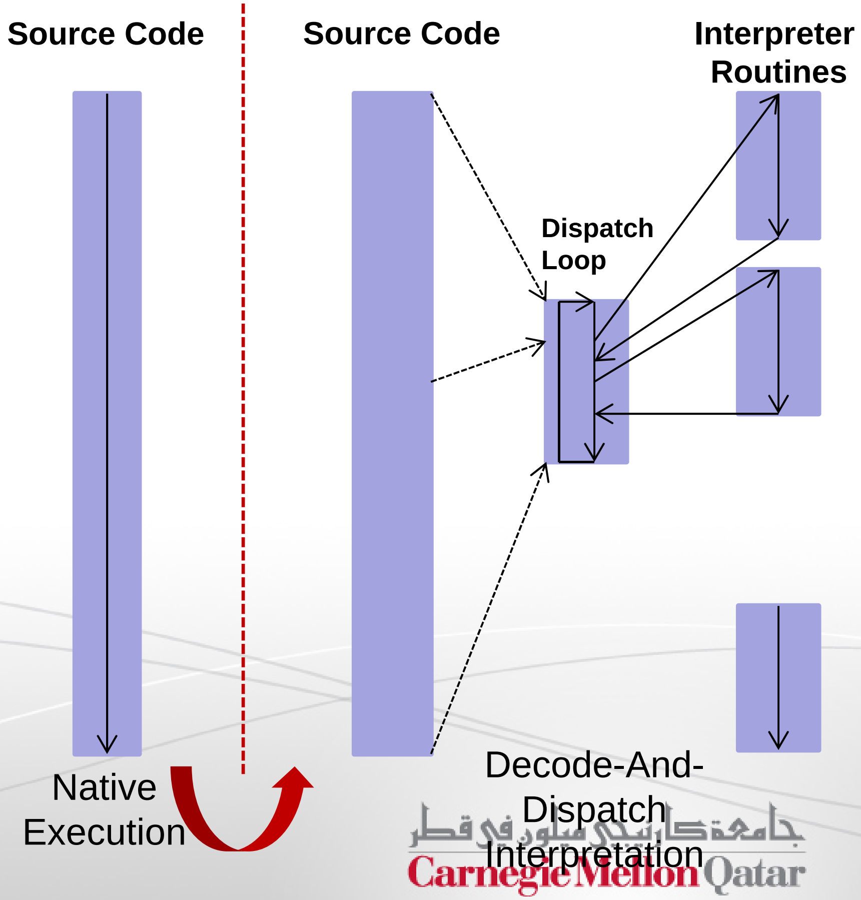


# Decode-And-Dispatch

- A simple interpreter, referred to as decode-and-dispatch, operates by stepping through the source program (instruction by instruction) reading and modifying the source state

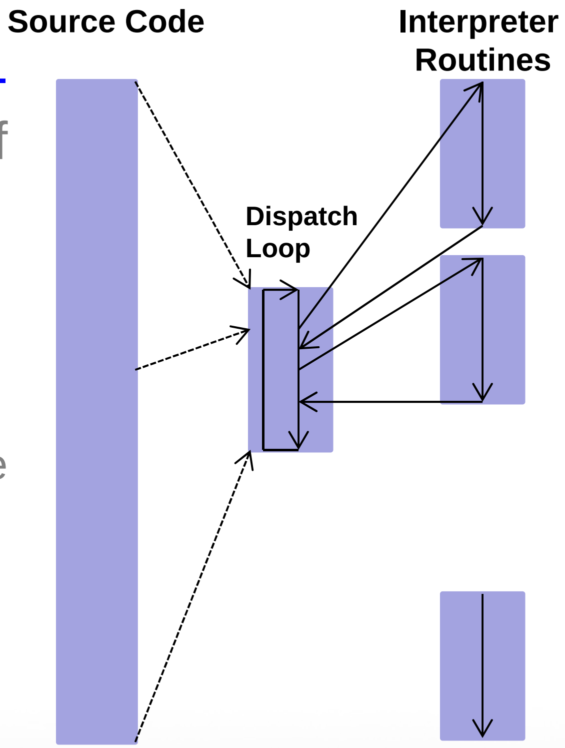
- Decode-and-dispatch is structured around a central loop that decodes an instruction and then dispatches it to an interpretation routine

- It uses a *switch statement* to call a number of routines that emulate individual instructions



# Decode-And-Dispatch- Drawbacks

- The central dispatch loop of a decode-and-dispatch interpreter contains a number of branch instructions
  - Indirect branch for the switch statement
  - A branch to the interpreter routine
  - A second register indirect branch to return from the interpreter routine
  - And a branch that terminates the loop
- These branches tend to degrade performance



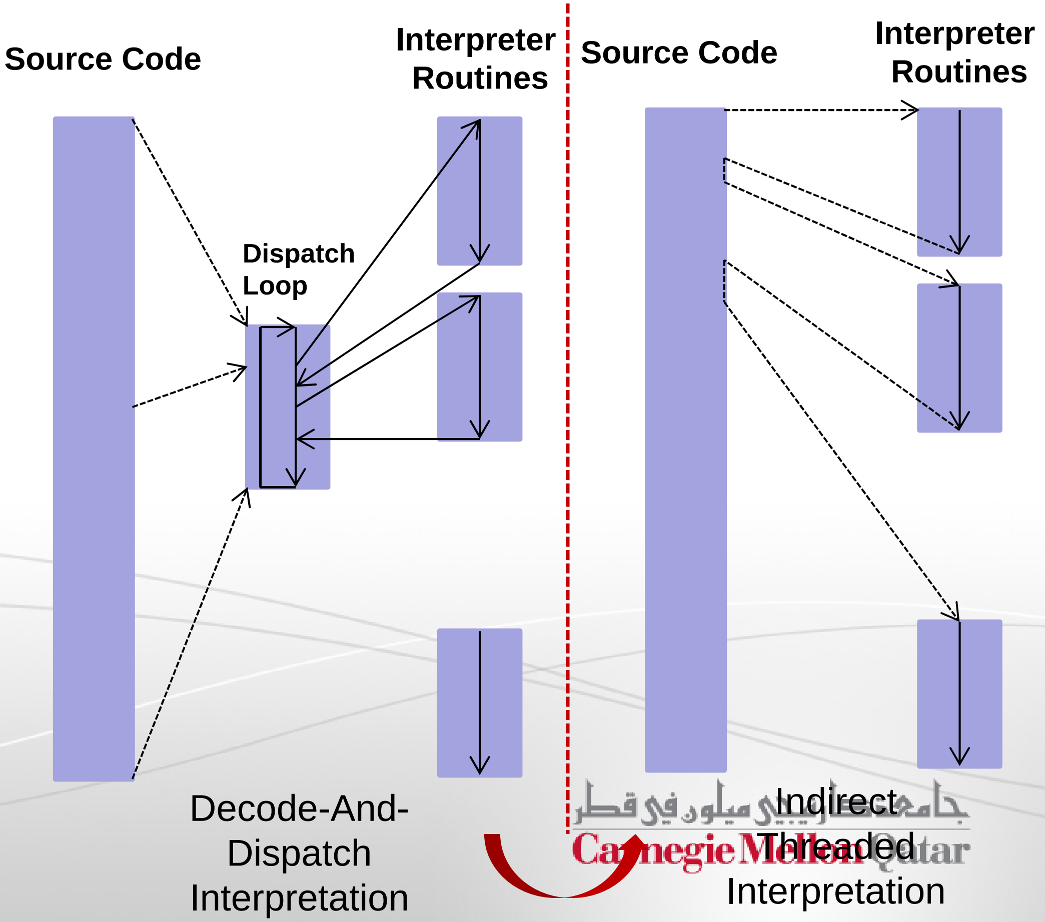
Decode-And-Dispatch Interpretation

# Indirect Threaded Interpretation

- To avoid some of the branches, a portion of the dispatch code can be appended (threaded) to the end of each of the interpreter routines

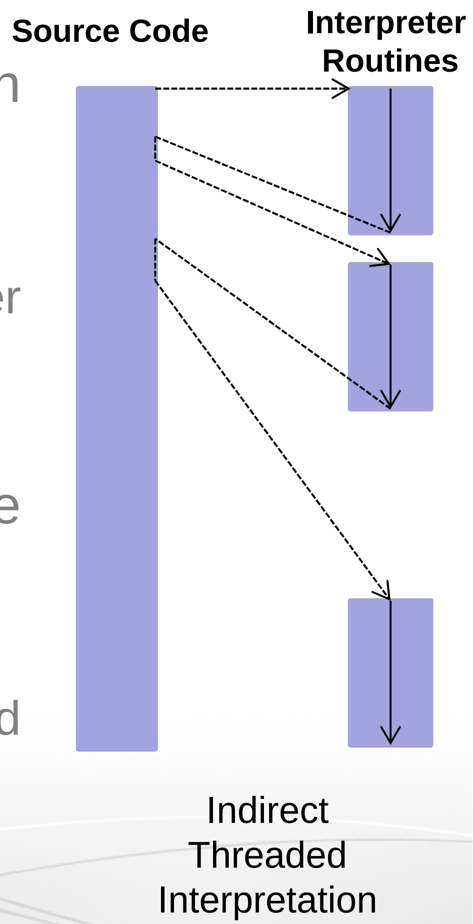
- To locate interpreter routines, a dispatch table and a jump instruction can be used when stepping through the source program

- This scheme is referred to as indirect threaded interpretation



# Indirect Threaded Interpretation- Drawbacks

- The dispatch table causes an overhead when looked up:
  - It requires a memory access and a register indirect branch
- An interpreter routine is invoked every time the same instruction is encountered
  - Thus, the process of examining the instruction and extracting its various fields is always repeated



Indirect Threaded Interpretation

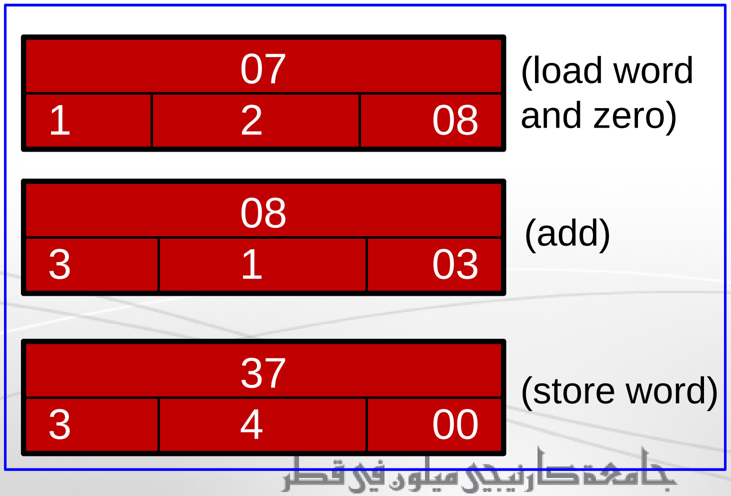
# Predecoding (1)

- It would be more efficient to perform a repeated operation *only once*
- We can save away the extracted information of an instruction in an intermediate form
- The intermediate form can then be simply reused whenever an instruction is re-encountered for emulation
- However, a Target Program Counter (TPC) will be needed to step through the intermediate code

PowerPC source code

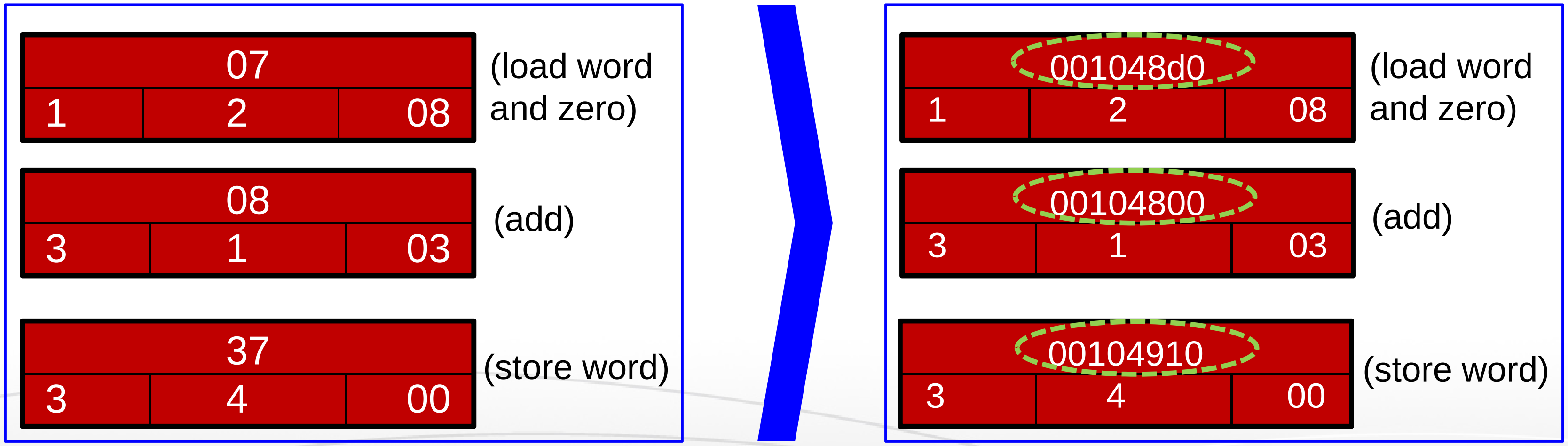
```
Lwz r1, 8(r2)    //load word and zero
Add r3, r3, r1   //r3 = r3 +r1
Stw  r3, 0(r4)   //store word
```

PowerPC program in
*predecoded intermediate form*



# Predecoding (2)

- To avoid a memory lookup whenever the dispatch table is accessed, the opcode in the intermediate form can be replaced with the address of the interpreter routine



07
1 2 08
(load word and zero)

08
3 1 03
(add)

37
3 4 00
(store word)

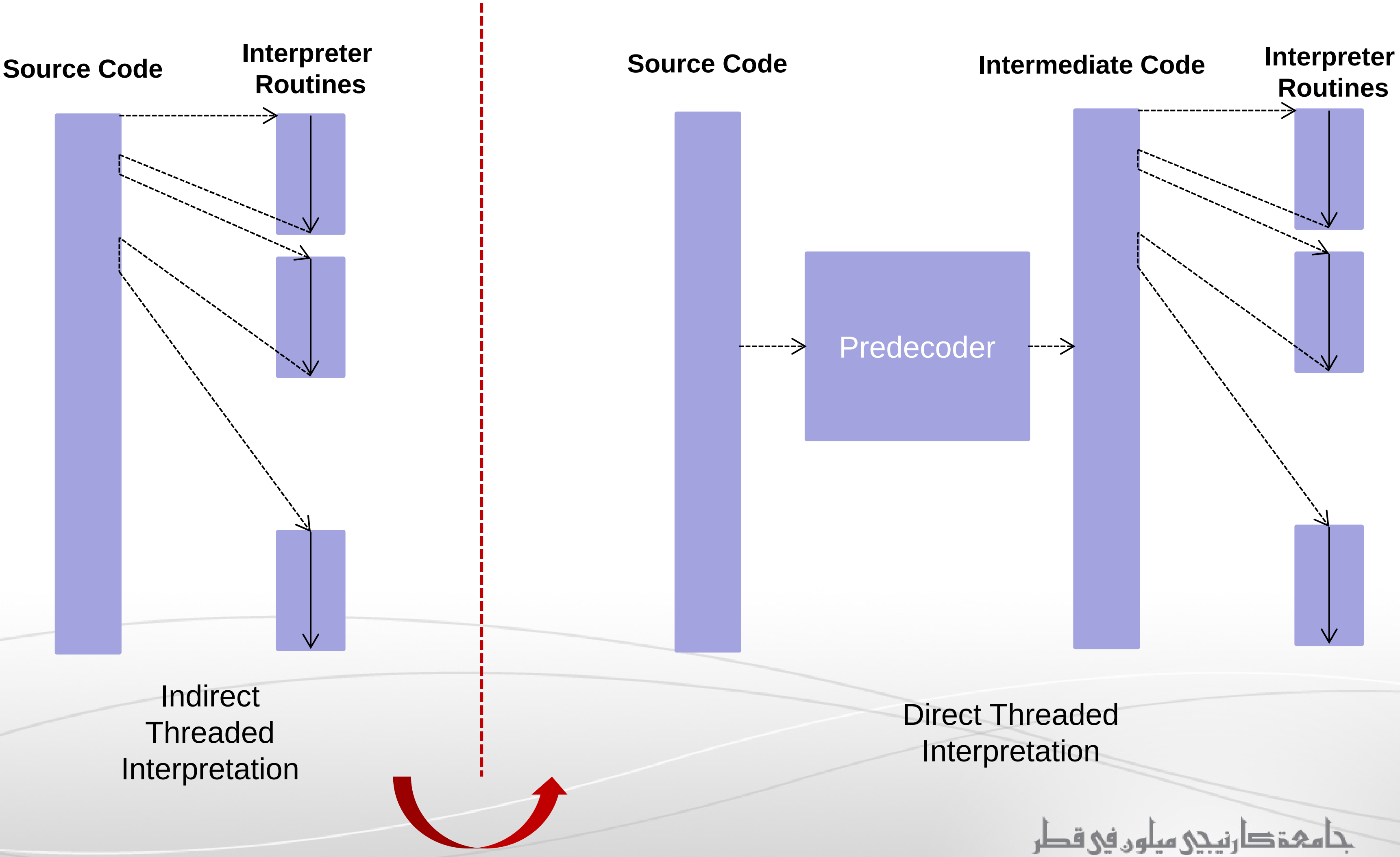
001048d0
1 2 08
(load word and zero)

00104800
3 1 03
(add)

00104910
3 4 00
(store word)

- This leads to a scheme referred to as direct threaded interpretation

# Direct Threaded Interpretation



جامعة كارنيجي ميلون في قطر
Carnegie Mellon Qatar

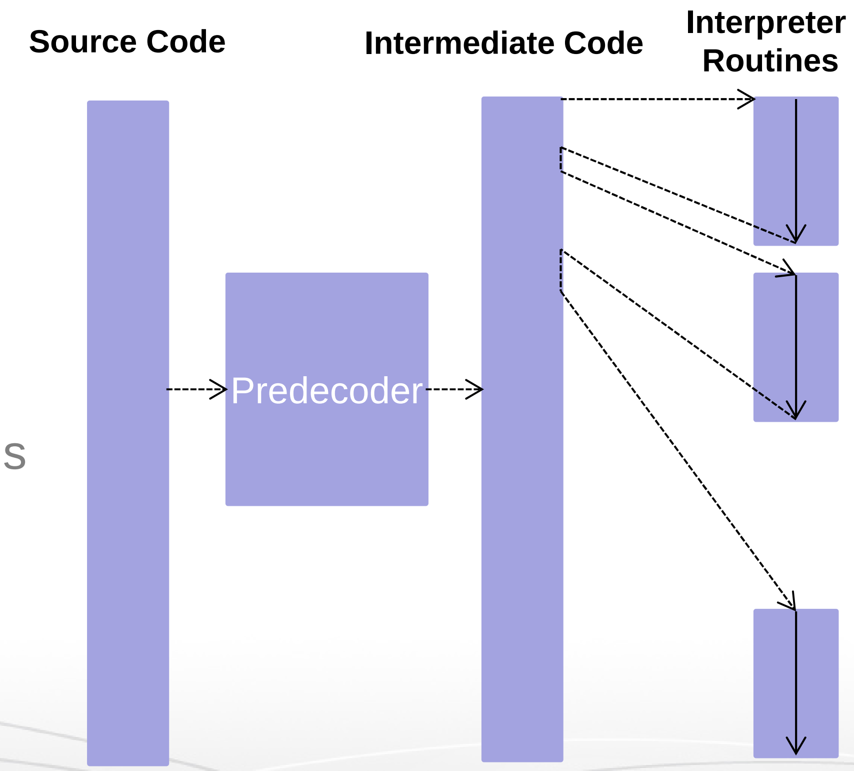
# Direct Threaded Interpretation-Drawbacks

- Direct threaded interpretation still suffers from major drawbacks:

1. It limits portability because the intermediate form is dependent on the exact locations of the interpreter routines

2. The size of predecoded memory image is proportional to the original source memory image

3. All source instructions of the same type are emulated with the same interpretation routine



جامعة كارنيجي ميلون في قطر
Carnegie Mellon Qatar

# Binary Translation

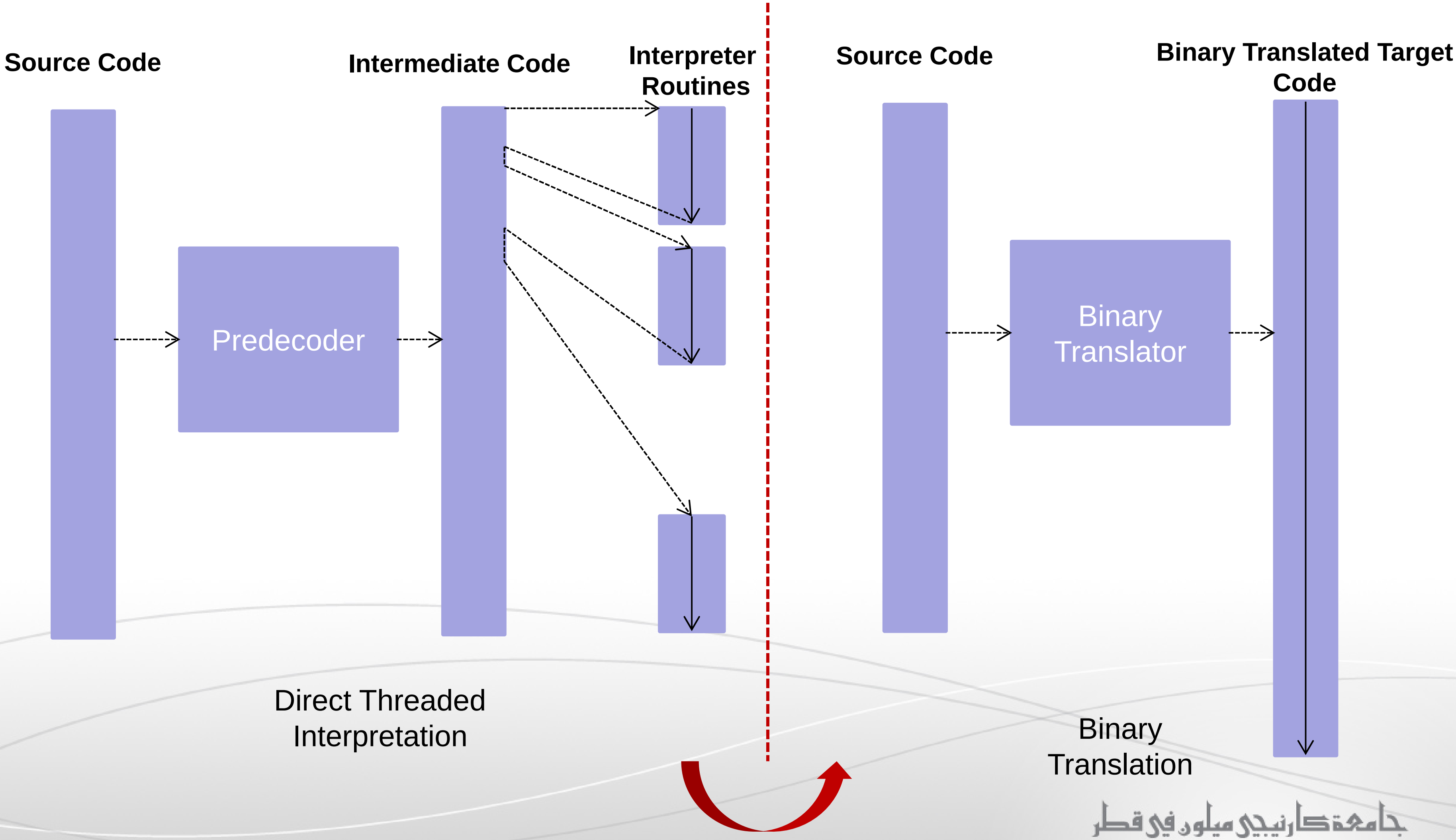
- Performance can be significantly enhanced by mapping each individual source binary instruction to its own customized target code

- This process of converting the *source binary program* into a *target binary program* is referred to as binary translation

- Binary translation attempts to amortize the fetch and analysis costs by:

  1. Translating a block of source instructions to a block of target instructions
  2. Caching the translated code for repeated use

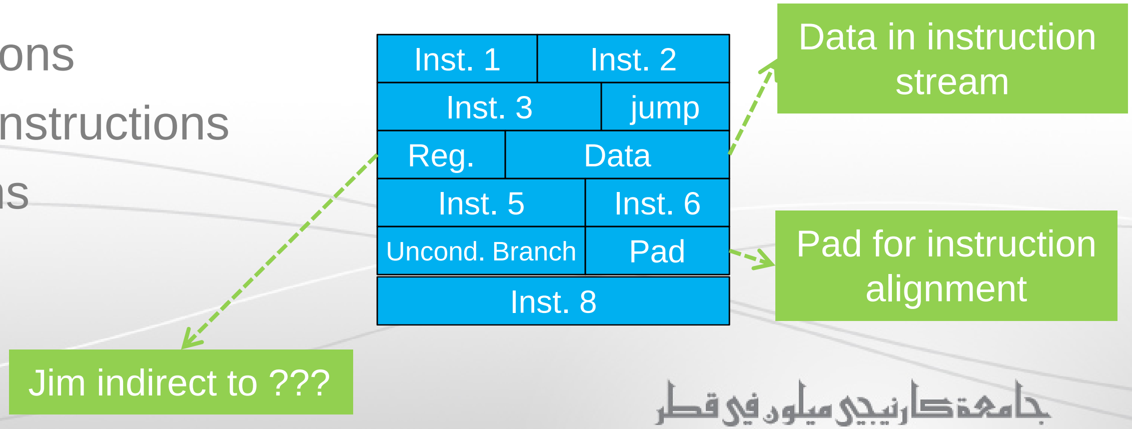
# Binary Translation



جامعة كارنيجي ميلون في قطر
Carnegie Mellon Qatar

# Static Binary Translation

- It is possible to binary translate a program in its entirety before executing the program
- This approach is referred to as static binary translation
- However, in real code using conventional ISAs, especially CISC ISAs, such a static approach can cause problems due to:
    - Variable-length instructions
    - Data interspersed with instructions
    - Pads to align instructions
    - Register indirect jumps



Inst. 1 | Inst. 2
Inst. 3 | jump
Reg. | Data
Inst. 5 | Inst. 6
Uncond. Branch | Pad
Inst. 8

Data in instruction stream

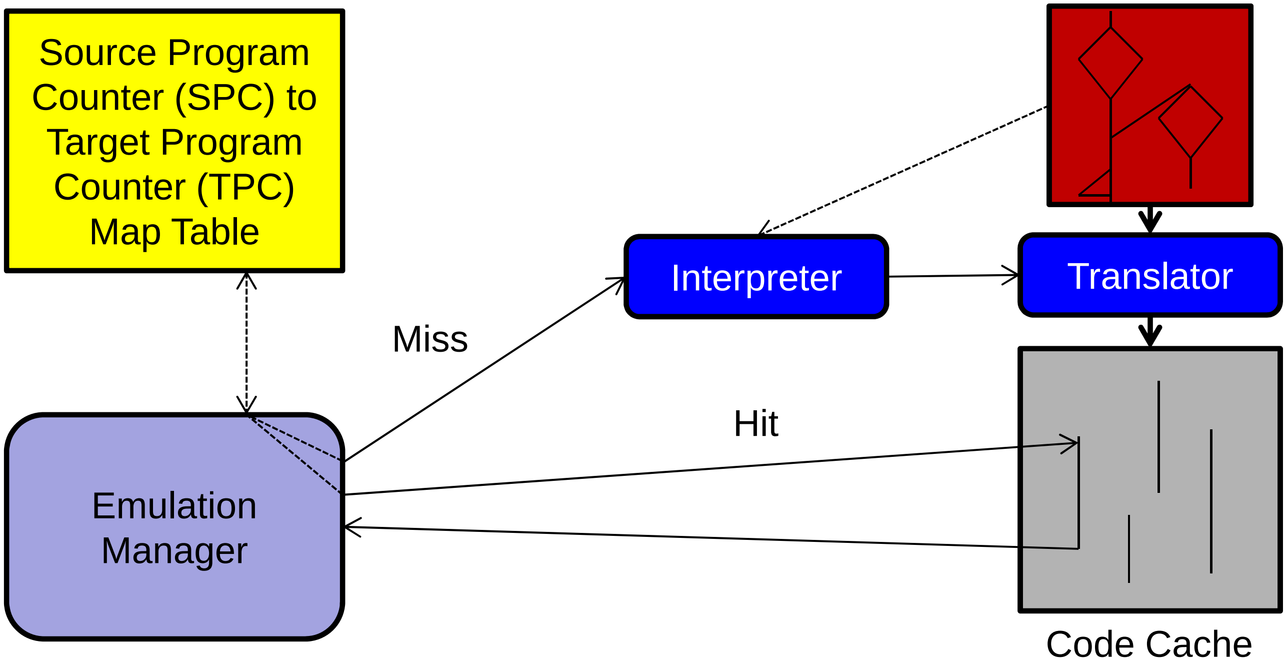
Pad for instruction alignment

Jim indirect to ???

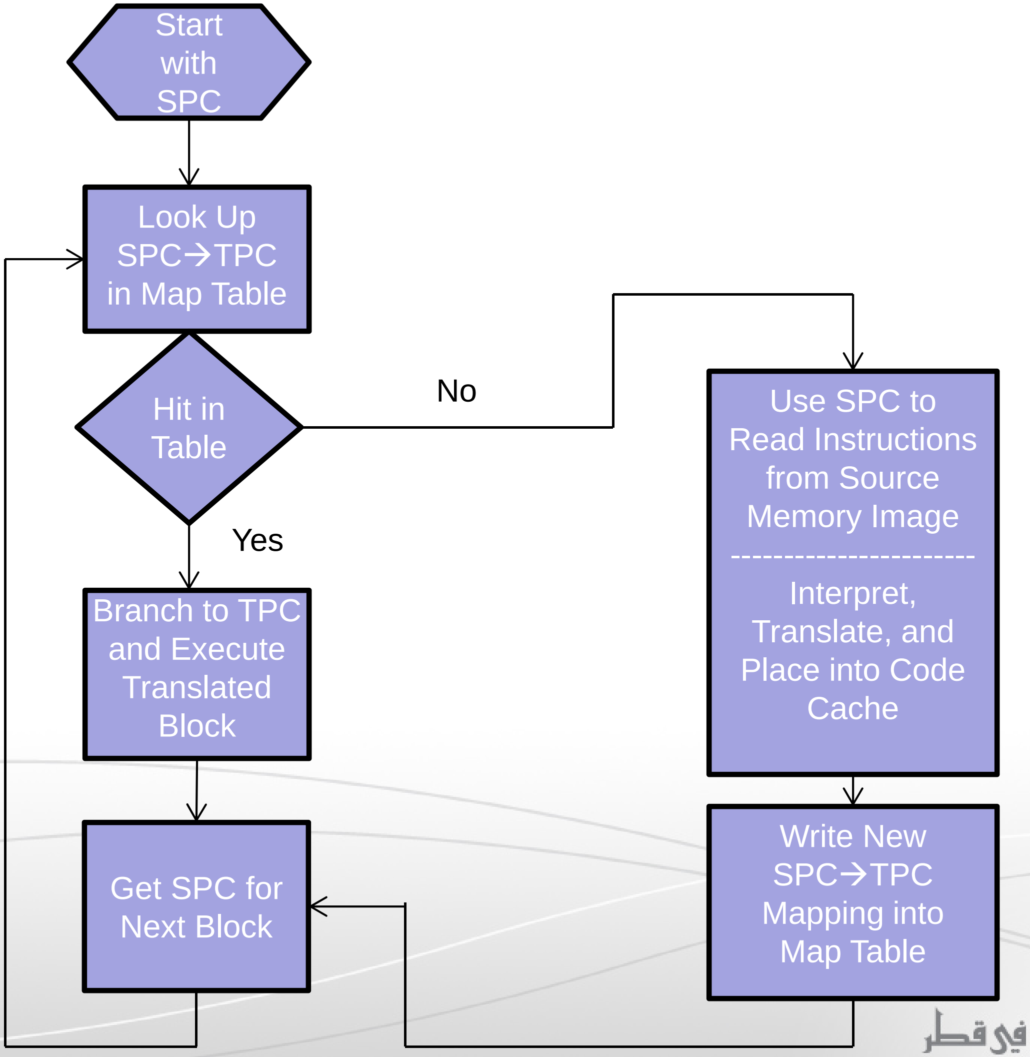
# *Dynamic* Binary Translation

- A general solution is to translate the binary while the program is operating on actual input data (i.e., *dynamically*) and interpret new sections of code *incrementally* as the program reaches them

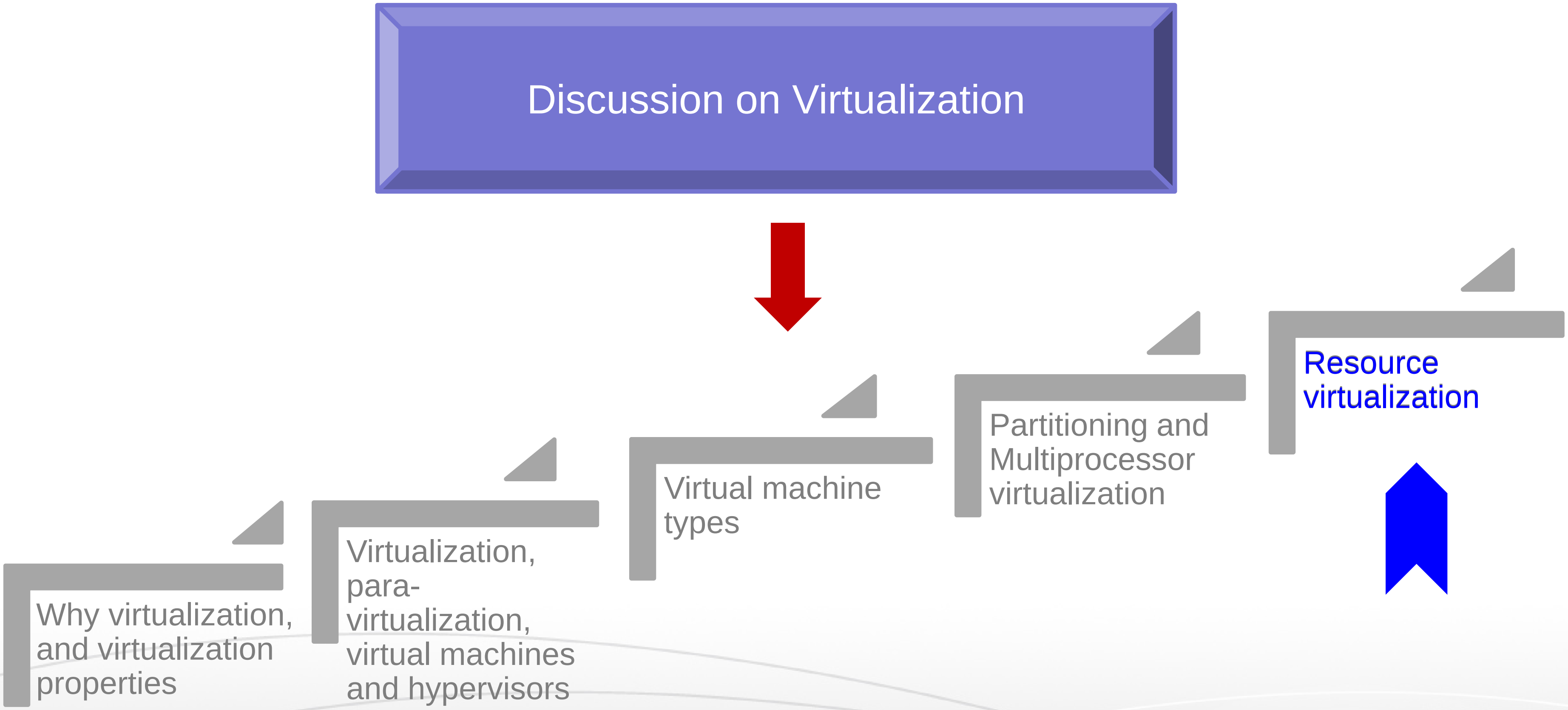
- This scheme is referred to as dynamic binary translation


Source Program Counter (SPC) to Target Program Counter (TPC) Map Table
Interpreter
Translator
Miss
Hit
Emulation Manager
Code Cache

# *Dynamic* Binary Translation


Start with SPC
Look Up SPC→TPC in Map Table
Hit in Table
No
Yes
Branch to TPC and Execute Translated Block
Get SPC for Next Block
Use SPC to Read Instructions from Source Memory Image
-----------------------
Interpret, Translate, and Place into Code Cache
Write New SPC→TPC Mapping into Map Table

# Next Class



Discussion on Virtualization

Why virtualization, and virtualization properties

Virtualization, para-virtualization, virtual machines and hypervisors

Virtual machine types

Partitioning and Multiprocessor virtualization

Resource virtualization