Virtual Machines

Pr. Olivier Gruber

Full-time Professor

Université Joseph Fourier

Laboratoire d'Informatique de Grenoble

Olivier.Gruber@imag.fr

Acknowledgments

- Reference Book

  Virtual Machines
  Versatile Platforms for systems and processes

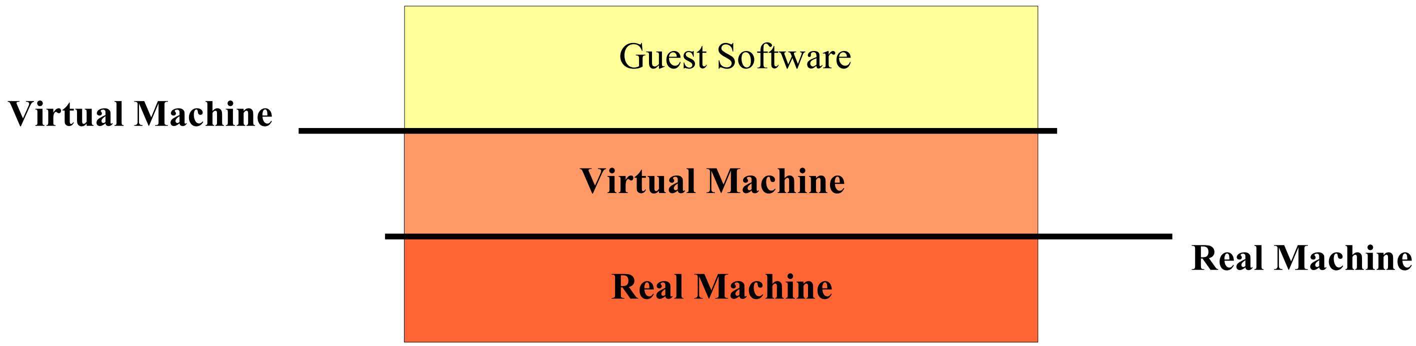
  James E. **Smith**, Ravi **Nair**

  Morgan Kaufmann

- Research Articles
  - Cited on various slides

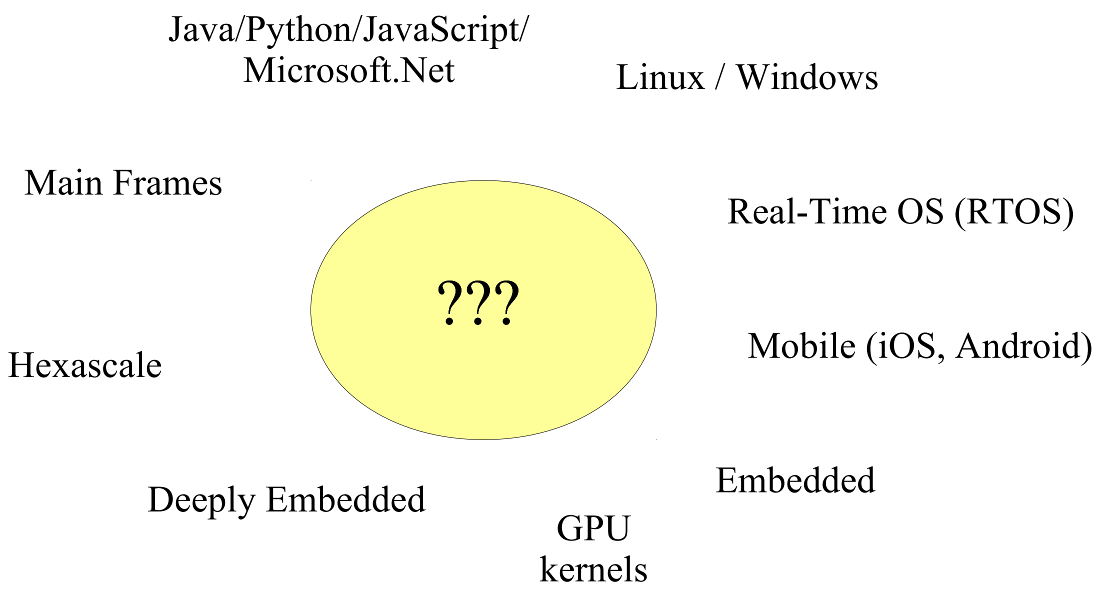
Virtual Machine Basics

- ***Virtual Machines*** versus ***Real Machines***
  - *A virtual machine defines a machine (interface)*
  - *A virtual machine is a machine (implementation)*



Virtual Machine Basics

- ***Virtual Machines*** versus ***Real Machines***
  - *So many virtual machines…*



Virtual Machine Basics

- ***Virtual Machines*** versus ***Real Machines***
  - *So many real machines…*

Java/Python/JavaScript/
Microsoft.Net

Linux / Windows

Main Frames

Real-Time OS (RTOS)

Smart Phones:
4 cores, 1.5GHz
2GB RAM or more
32GB storage or more

Hexascale

Mobile (iOS, Android)

DeepBlue:
1.5 M cores
1.5 M GB of memory
20K TeraFlops
8 Mega-watts
Under linux...

Deeply Embedded

Embedded

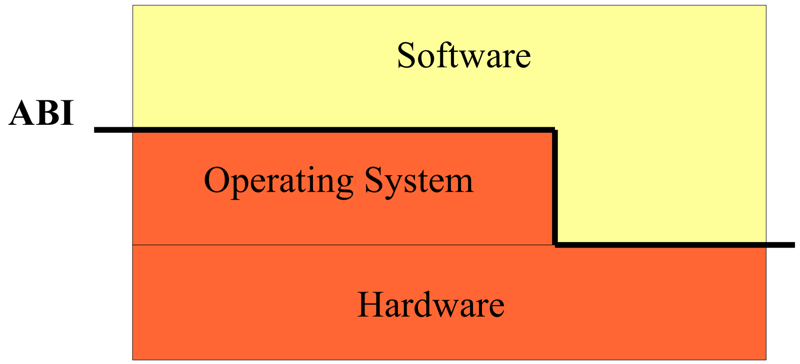
Historical perspective:
Sparc 1+ (1995)
32bit RISC
64MB RAM
12 MIPS @ 25MHz

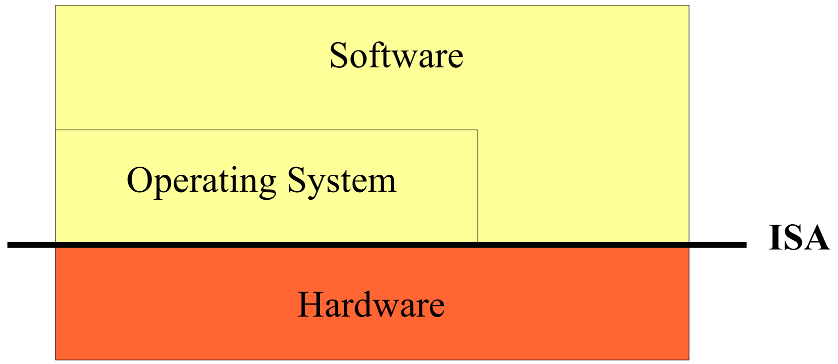
Kalray:
700 Gops, 230 GFLops
5 Watts
256 VILW processing cores
32 VILW resource mgt cores
28nm technology

Arduino:
Atmel 8bit RISC
128KB Flash, 8KB SRAM
4K EEPROM
16 MIPS @ 16MHz

Virtual Machine Basics

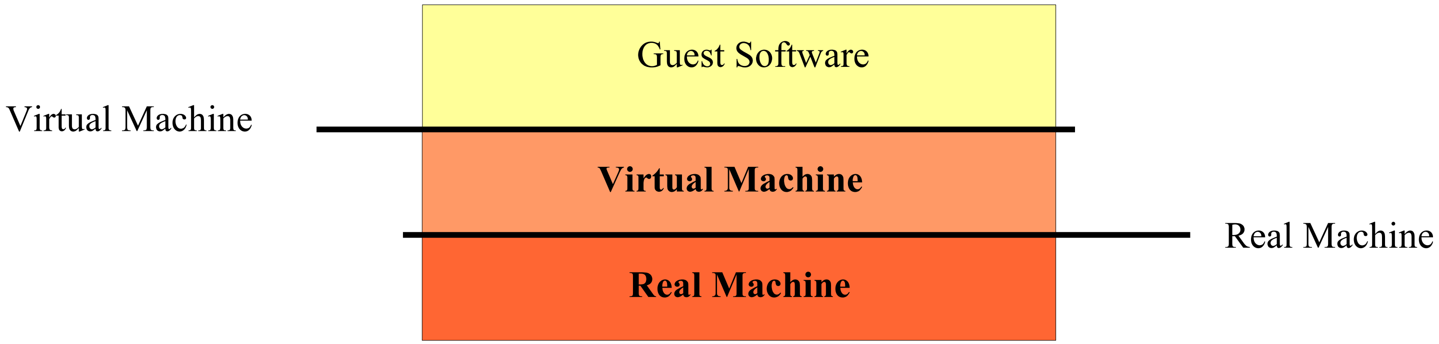
- *Instruction Set Architecture* (ISA)
  - Defines the instruction set
  - Defines other concepts such as page tables, traps, interrupts, etc.
- Application Binary Interface (ABI)
  - Defines core concepts above the ISA
  - Example:
    - Linux kernel system calls
    - Related to processes, threads, files, and devices





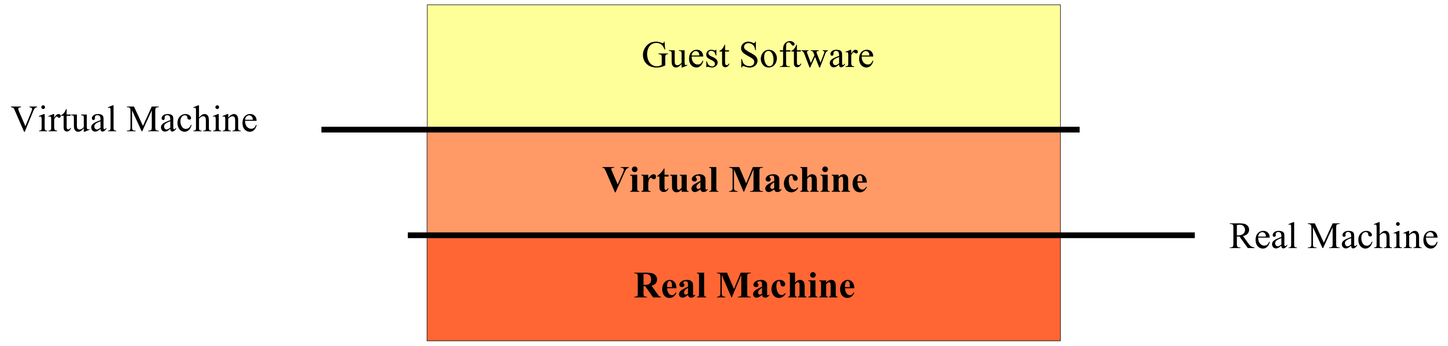
Virtual Machine Basics

- ***Virtual Machines*** versus ***Real Machines***
  - *Let's review what you should already know*
    - *Basics of hardware and operating systems*
    - *From the ground up...*
  - *But stepping back*
    - *Mostly from the perspective of why, merely discussing how...*
  - *Because not everything is Linux and Intel inside*
    - *Most certainly not the next generation of operating systems*



Virtual Machine Basics

- Today's Topics
  - Hardware basics and minimal virtual machine
  - Discussing the Palm-OS as an eye opener
  - Discussing Linux kernel, before it was a toolbox
  - Discussing thread-oriented and event-oriented programming
  - Discussing kernels vs distributions



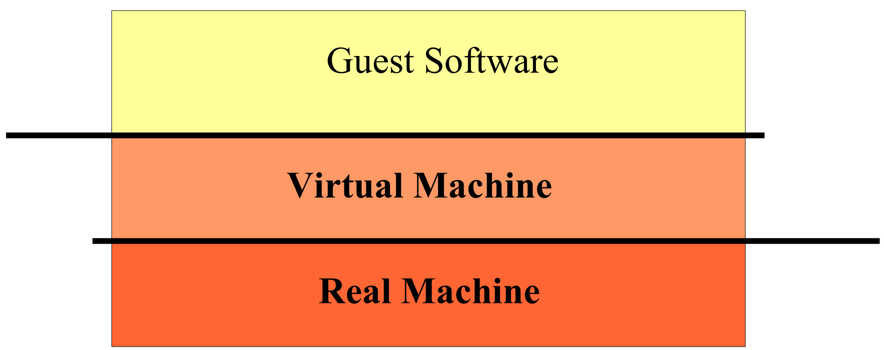
Virtual Machine Basics

- Back to basics...

MinimalVirtual Machine ?

==

Minimal Operating System ?



Simple Real Machine ?

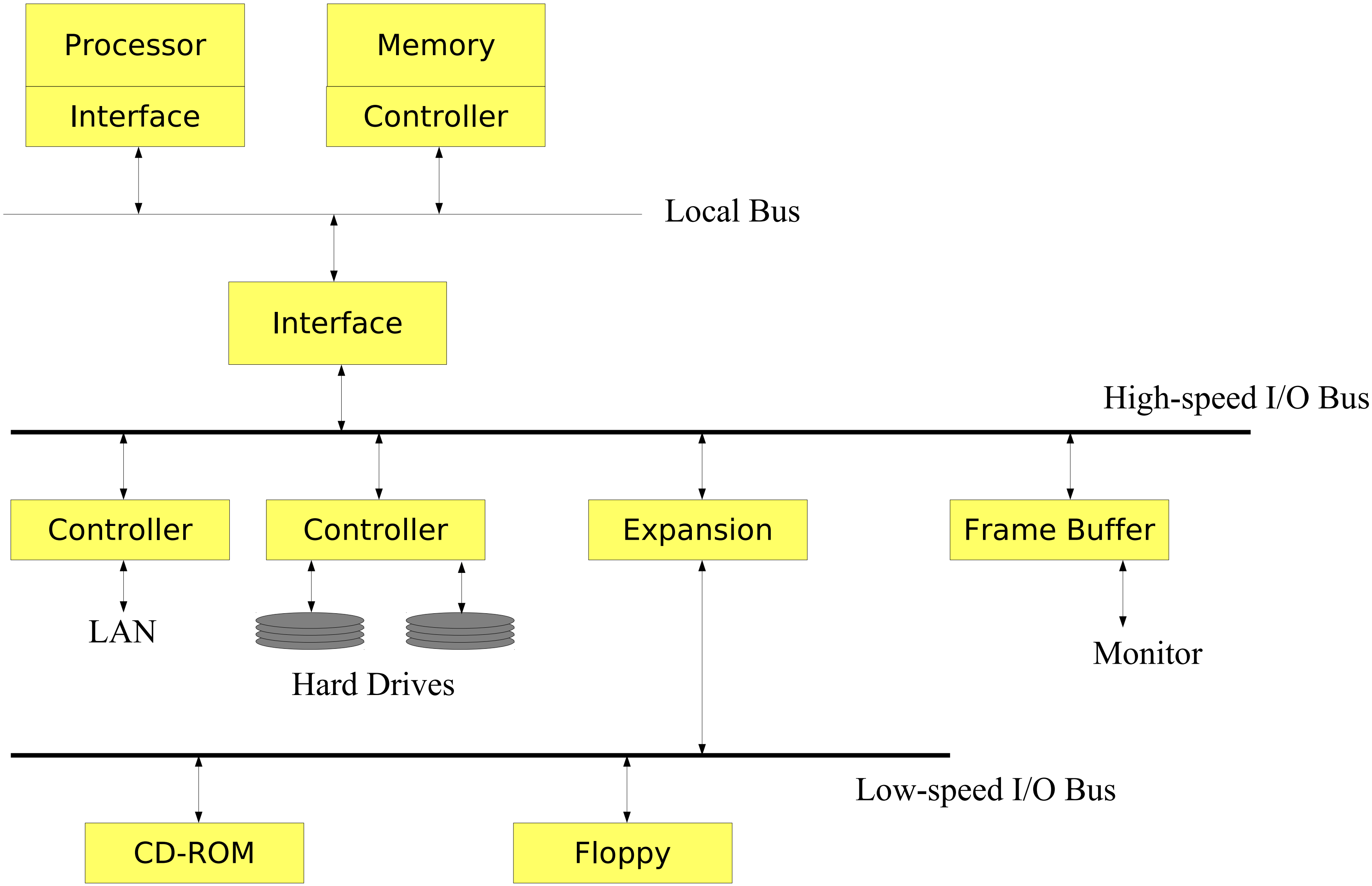
==

Single core
No MMU
Simple devices like serial lines

Computer Basics

- Hardware
  - Single-core CPU, with load/store interface
  - Memory controller and memory
  - Instructions: load/store, arithmetic, branches, etc.
  - Hardware registers
- Simple driver examples
  - UART, Mouse/Keyboard
  - Network Card with DMA
- Event-oriented programming
  - Halt/wakeup with polling
  - Interrupts with event-oriented scheduler
  - Separating top and bottom interrupt handlers

Hardware Architecture



Case Study: PalmOS

- Case study: PalmOS
  - Created in the late 90's
  - One of the first Personal Digital Assistant (PDA)
- Why discuss the PalmOS?
  - Remind you than not everything is Linux
  - Remind you than not everything is a general-purpose desktop
  - Illustrative of the embedded software philosophy
  - Open your mind...

Case Study: PalmOS

- Hardware target
  - Runs on a slow processor
  - Small amount of main memory (less than 1MB)
    - Persistent (battery-backed up)
    - No disk or no flash
- Essential use
  - Synchronized with a PC
  - Must run 3-4 weeks on single AA batteries
    - Still much better than any of today's smart phones...
    - Even though late 90's hardware had little or no power management
    - And today's smart phones have the latest in power management

Case Study: PalmOS

- Could PalmOS follow a traditional OS design?
  - Footprint is way too big
    - Linux kernel does not fit in 1MB
    - A tad better with embedded linux kernels, years later
  - Processes and threads are too costly
    - Virtual memory and hardware protection are unnecessary
  - File systems are inadequate
    - Can't have two copies of data and code (running image and files)
    - Can't pay the translation costs (from file to memory formats)
    - Does not provide fine-grain data replication

Case Study: PalmOS

- Central design points
  - Avoid dual storage of data (memory and file)
  - Avoid format translation (back and forth from files)
  - Keep it simple, help developers with replication
- Introduced the concept of the PalmOS database
  - A database is memory-resident collection of C structs
  - A database provides search and scan operations
    - Manages in-memory C structs
  - Applications get pointers back to C structs
    - Direct manipulations of the C structs
  - Direct manipulations of the C structs in database
    - Database packing is available to fight fragmentation
    - Invalidates pointers to database structs

Case Study: PalmOS

- Replication
  - Each database is replication-aware
    - Manages timestamp and dirty-bits
  - Device-level synchronization
    - PalmOS controls the two-way synchronization with the PC
- Conduits
  - Plugins for the replication engine
  - Translates PC data to Palm-suited data
- Example:
  - Adapting email to a small footprint device
    - Remove attachments or shorten long messages
  - Limit the number of messages per synchronization
    - Do not replicate all new emails
    - Could implement some LRU algorithm on emails to keep

Case Study: PalmOS

- Discussing the design
  - Applications are just event handlers
    - No threads and no process
    - Just a basic event loop for GUI and replication events
  - No saving of application state
    - There is no need, memory is persistent
    - Smaller application footprint (no save and load code)
    - Less overhead (no translation, no copy)
  - Instant-on property
    - No loading/saving of state
    - No loading of executable code
    - Just deliver *events* to the active application

From PalmOS to Unix…

Why would we need Unix/Linux?

What are the key concepts?

Unix – Linux

- A traditional view of an operating system
  - Goes all the way back to the 70s
  - Before it was a powerful toolbox
  - It was promoting a certain programming model (philosophy even)
- Kernel core concepts
  - Processes with the libC wrapping the ABI
  - Streams (files and pipes)
  - /dev and device drivers

Unix – Linux

- Design questions:
  - What is a process?
  - What is a stream?
  - Describe the programming model (philosophy even)
  - Compare it to the one promoted by DBMS
  - Discuss robustness
  - Discuss security/safety
- Does Unix/Linux Require hardware support?
  - Compared to the PalmOS for example

Unix – Linux

- Processes
  - Virtualize memory and memory isolation
  - Management entity - kill/clean – resource consumption – rights
  - Requires Page Tables or TLBs, Kernel/user mode, and timer interrupt
- Sweet spot
  - Use private files or communicate through pipes (safe programming)
  - Single threaded and blocking libC (easy programming)
- Opposite to DataBase Management Systems (DBMS)
  - Shared data, protected through transactions
  - Query capabilities

Unix – Linux

- Discuss main challenges and responsabilities
  - Fair/efficient scheduling
  - Fair/efficient resource allocation
  - Supporting DBMS implementations
  - Supporting High Performance Computing (HPC)
- More design questions
  - What is a device driver?
  - What is /dev for?
  - Why have file systems as a part of the Kernel?
  - Why is not the case for the Window Manager?

Unix – Linux

- Devices
  - Load/store interface
  - Interrupts
- Device drivers
  - Manages a device
  - Traditionally character and block devices
- The /dev directory
  - Special files, with major and minors
  - Single threaded and blocking libC (easy programming)
  - Opposite to DBMS model
- Boot process
  - BIOS, boot loader, kernel, init

Unix – Linux

- Threads vs Events
  - Two religious views, opposed since the beginning of time…
  - Today, both exist and both are in use
  - Often mixed by necessity
- Event-oriented programming
  - We already discussed it, it is the natural processing of a single core
- Introducing threads
  - Multiple threads per process
  - Requires synchronization
  - Blocking or non-blocking ABI
  - Meaningful both on single and multi-core hardware

Linux Kernel vs Distribution

- Two very different things
  - One kernel... (multiple versions of course)
  - Many distribution... with very different goals
  - Ubuntu, Debian, Raspbian, uc-linux, ...
- Google Android
  - Linux kernel
  - A few core libraries (webkit, gstreamer, etc.)
  - A Java-based distribution with its own software concepts
  - Components and services **distributed across processes**
- OSGi Platforms
  - Set-top boxes, ADSL boxes, factory-management gateways, etc.
  - An embedded operating system or not, possibly Linux
  - A world of services and modules
  - Remote management – install/uninstall/start/stop
  - But **everything runs in a single** Java Virtual Machine

Linux Kernel vs Distribution

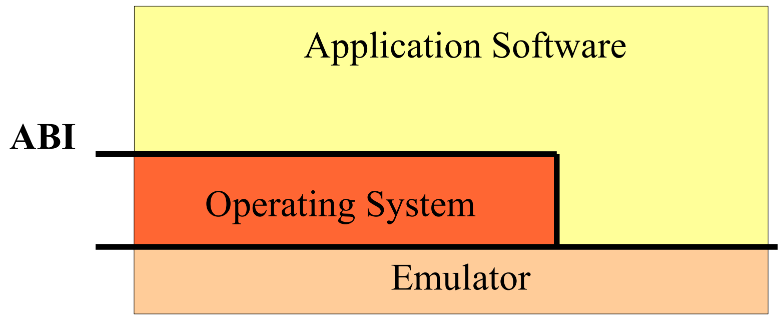
- OSGi vs Linux
  - Both virtual machines, both application platforms
  - Let's compare the platform concepts

✔	Memory Management	✔
✔	Threads and concurrency control	✔
✔	File system and sockets	✔
✔	Manage applications	✔
✔	Security	✔
🛇	Safety	✔

Not in Java but available in Android...
But let's discuss safety in Linux...

Emulation

- ***Interpretation:***
  - *Interpretation* of individual guest instructions (fetch, decode and emulate)
  - Easy but slower
- ***Binary translation***
  - *Binary translation* of blocks of guest instructions to native instructions
  - More complex but faster (close to native performance)



Homework

- Read and understand the extra slides
  - They are necessary to understand the rest of the lectures
  - They are the absolute minimum for whoever has an interest in operating systems
- Design work
  - Design an emulator for the ISA defined on the next slide
  - Your emulator will use an interpretor technology
  - Your emulator will be a regular Linux or Windows process

Virtual Machine Basics

- System-On-Chip ISA

- Memory:
 non volatile memory (battery-backed up)
 can be flashed externally, while CPU is halted
- Reset on power up at address 0x1000.
- Registers:
 0-15 general-purpose regs, 32bits
 R15 stack pointer, but only by convention.
 R16 program counter
 R17 link (holds return address when branching)
- Instructions:
 mov reg, value
 mov reg,reg
 mov reg,[reg] and [reg],reg
 add/sub/mul/div reg,reg
 branch [reg] with link in R17
 condbr (conditional branch)
  reg<val
  reg==val
  reg>val
  reg!=val

- Kernel-User mode
 r18 status register for the CPU
  0x01 kernel mode if set
  0x02 physical or virtual memory if set
  0x04 interrupts disabled/enabled if set

 r19 page table @, one level, 4K pages
 r20 interrupt vector @, 32 entries
 r21 interrupt mask
- Hardware Devices
 One timer
  @ 0x0004 Read current time,
   With two reads of 32bits
  @ 0x0005 set timer
   With two writes of 32 bits
 One serial line...
  @ 0x0010 RX
  @ 0x0014 TX
  @ 0x0015 STATUS,
   0x01 available bytes on RX
   0x02 available space on TX

Extra Slides
On
Real Machines

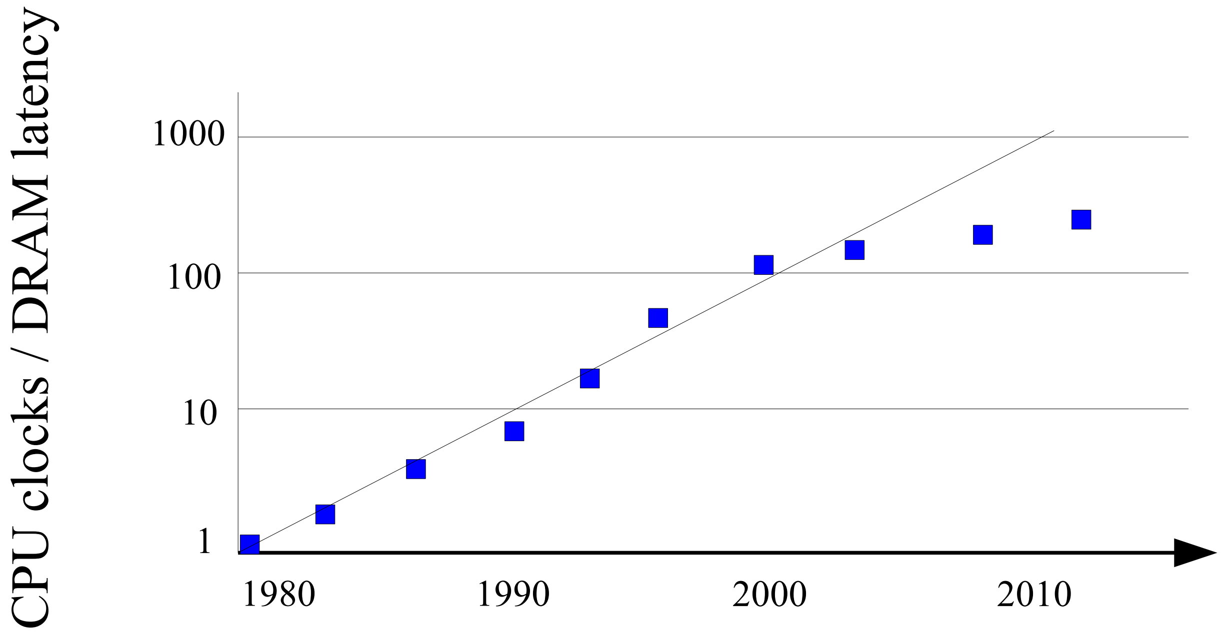
(homework for next week)

Quest for Performance...

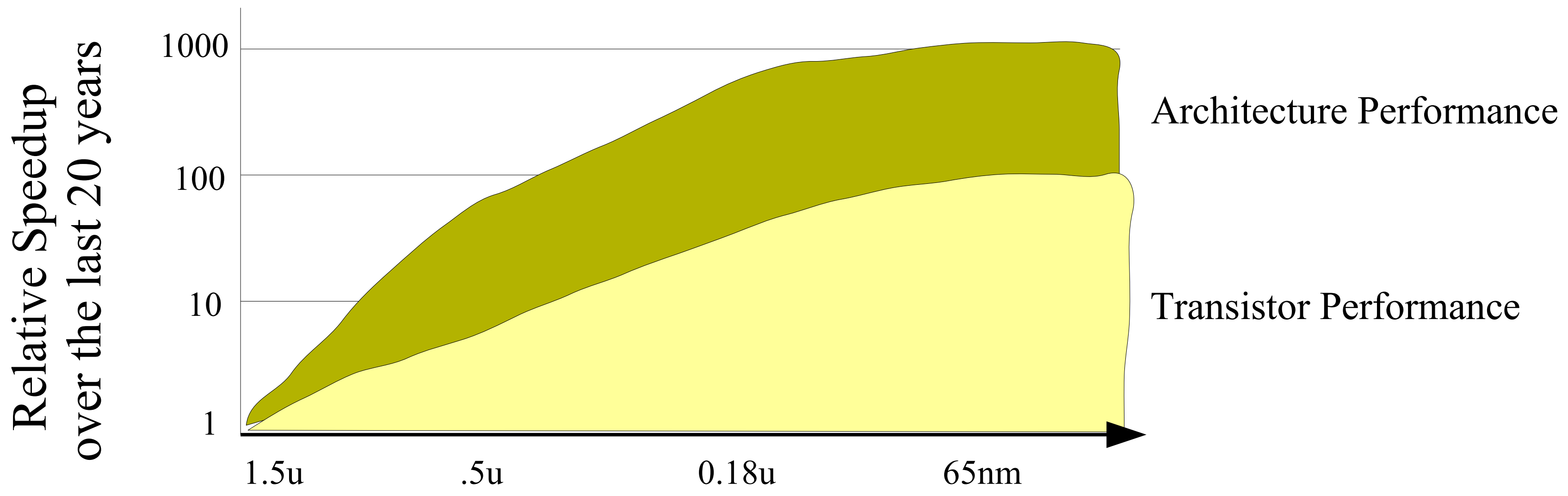
- Multiple facets
  - Transistor improvements
  - Architectural improvements
- Challenges
  - Memory latency
  - Power efficiency
  - Software obesity
  - Exploiting parallelism

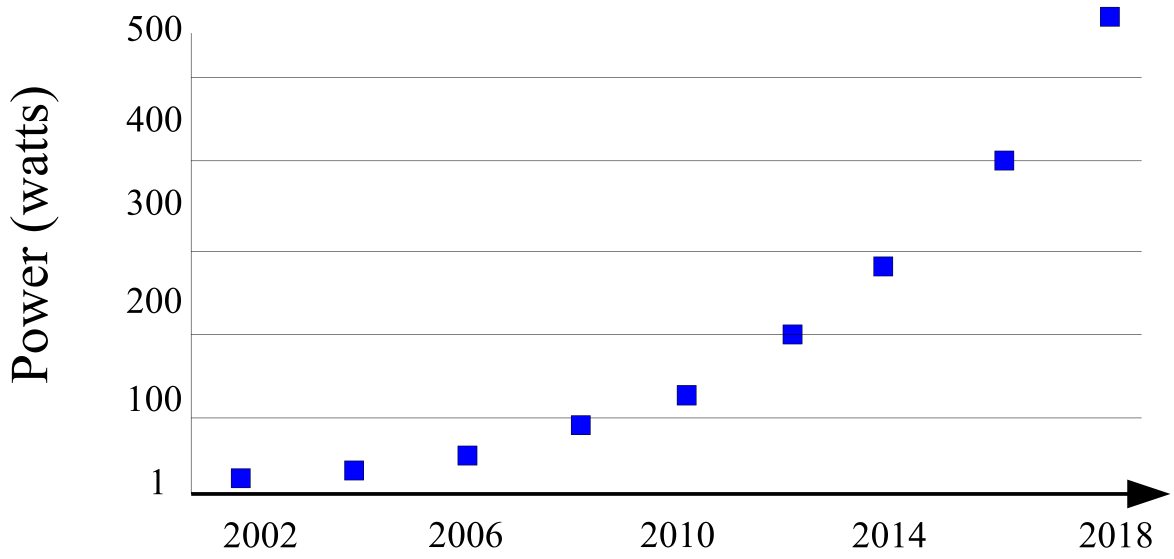
Memory Wall

- The challenge is memory latency…
  - Well over 100 of cycles
  - Flattening due to the flattening of CPU clock frequency



Discussing the Limits





Major Architectural Steps

- Memory Hierarchy
  - L1 and L2 caches
  - TLBs
- Internal core parallelism
  - Pipelining and Superscalar
  - Out-of-order execution (dependency driven execution)
- Speculative execution
  - Branch prediction and speculative execution
  - Complete speculative execution
- Hardware JIT compilation
  - Optimize, reorder, and parallelize
  - But maintain the outside contract (apparent sequential execution)

Major Architectural Steps

- Pros
  - Poorly written software execute fast
  - Peek performance is fantastic
  - Some of the gains can only be done in hardware at runtime
- Cons
  - Energy consumption and therefore heat production
  - Large surface on dies (limits registers, L1 cache size, etc.)
  - Not always effective (some programs are not faster)
  - Traps and interrupts are more expensive
  - Impacts the programming model
    - Requires fences with multiple cores
    - Cache flushes when writing device drivers

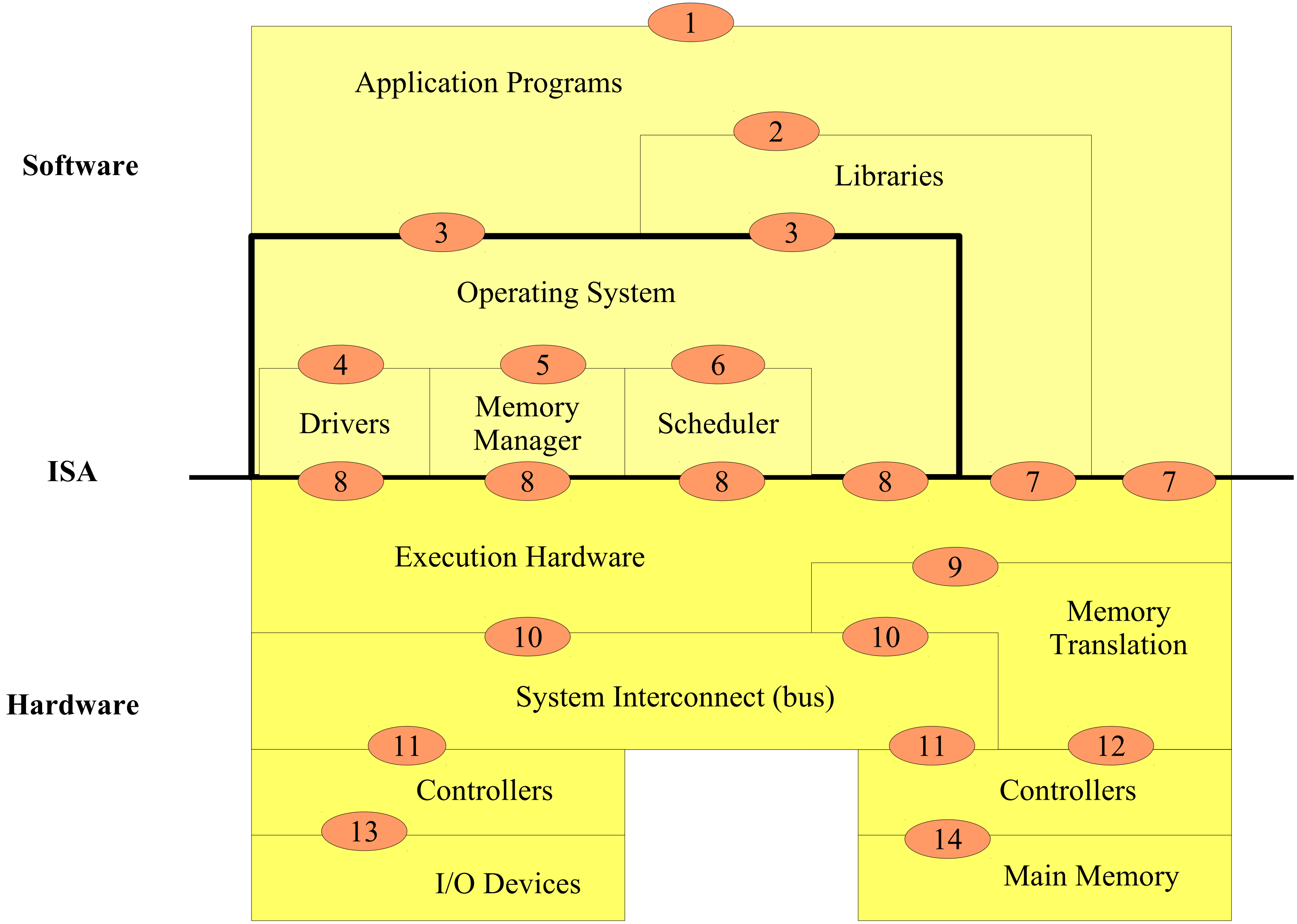
Major Architectural Steps

- Multiple cores
  - Performance through parallelism
- Success stories
  - GPUs for graphics
  - High-Performance Computing (HPC)
- Challenges
  - Who master parallel programming?
  - Not all applications have parallelism
  - Load-balancing versus execution locality
  - Race update conditions on shared data
  - Scalability of locking mechanisms
  - Energy consumption

Real Machines

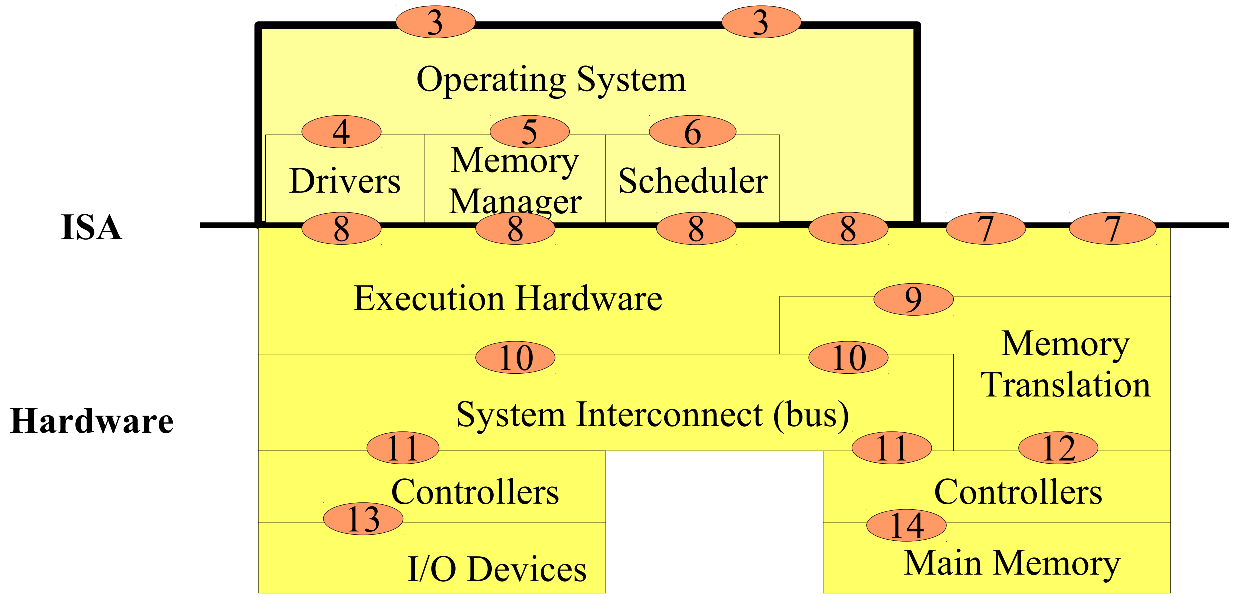
- Computer System Basics
  - Outline of the major components of computer systems
    - Their interfaces
    - The resources managed through those interfaces
  - We will look at
    - Primary hardware components
      - Processor, memory and I/O
      - Instruction Set Architecture (ISA)
    - Organization of a traditional operating system
      - Emphasis on managing system resources
      - Such as the processor, memory, or I/O devices
    - Discussing microkernels
      - Architecture, design, acceptance and performance

Computer System Architecture



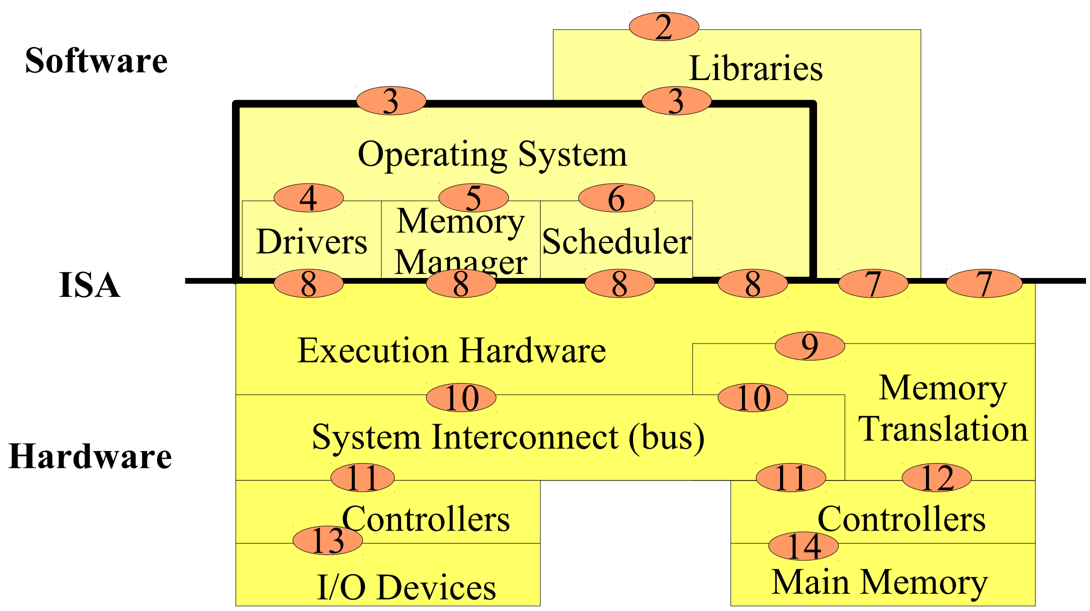
Instruction Set Architecture

- Instruction Set Architecture has two parts
  - User-level ISA (7)
    - Aspects that are visible to non-priviledged code
  - System-level ISA (8)
    - Aspects that are visible to priviledged code
    - Of course, the system-level ISA includes the user-level ISA



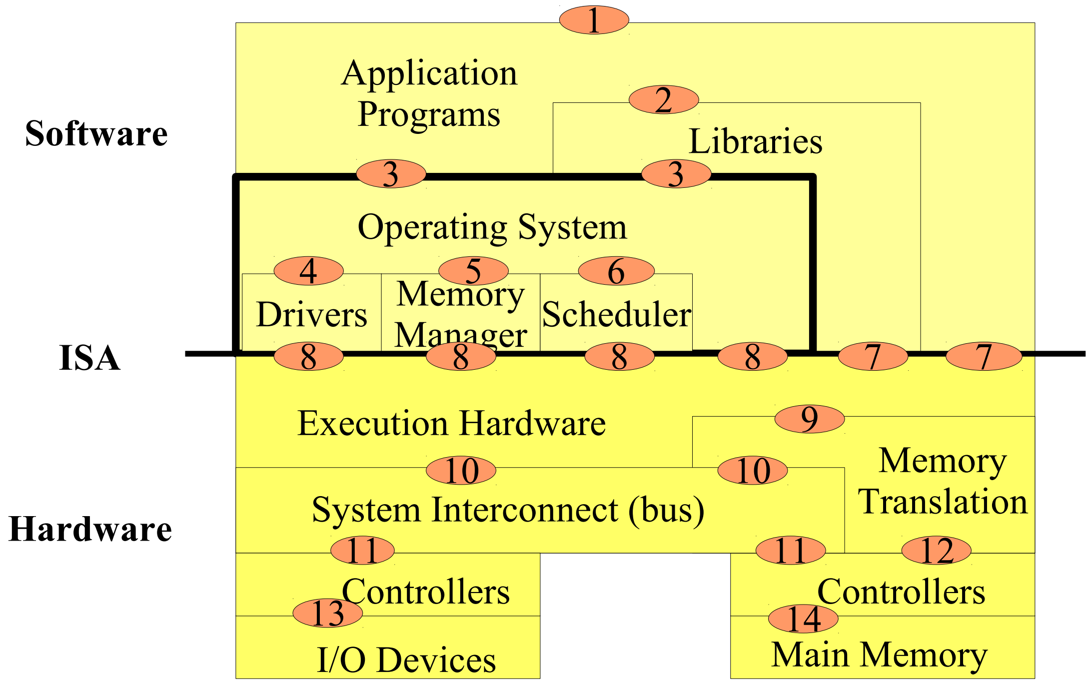
Application Interface

- Application Binary Interface (**ABI**)
  - User-level ISA 7
    - Aspects that are visible to non-priviledged code
  - System-call interface 3
    - Provide indirect access to shared resources
    - System calls use a trap mechanism to priviledged code in the OS
    - Each operating system specifies how parameters are passed

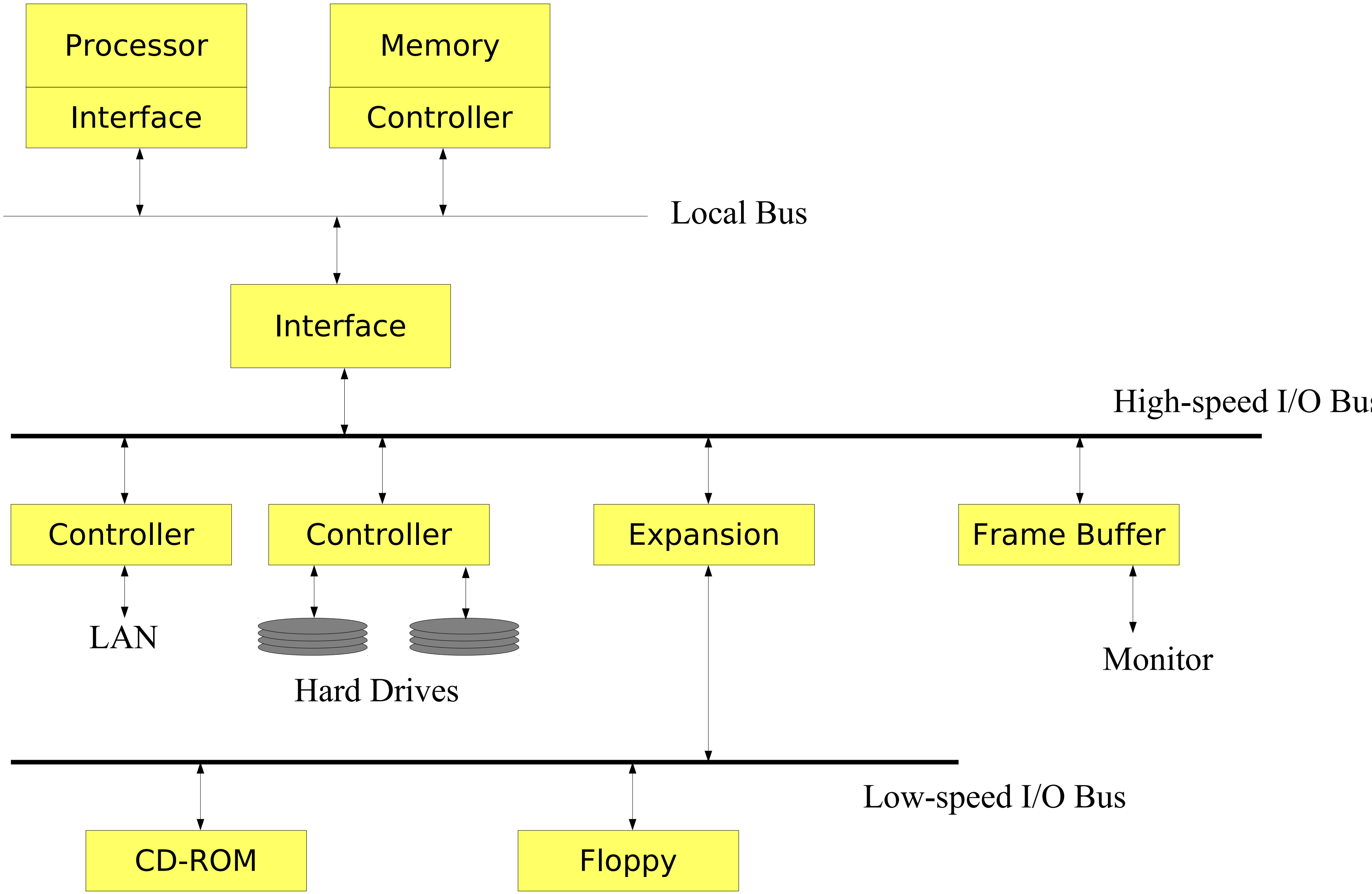


Application Interface

- Application Programming Interface (**API**) 2
  - Usually defined with respect to a *High-Level Language* (**HLL**)
  - A key element is the definition of standard libraries
  - Such libraries are defined at source-code level

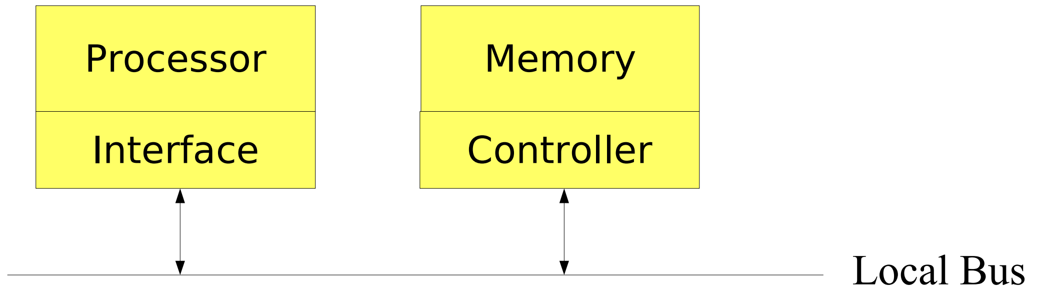


Hardware Architecture



Processor

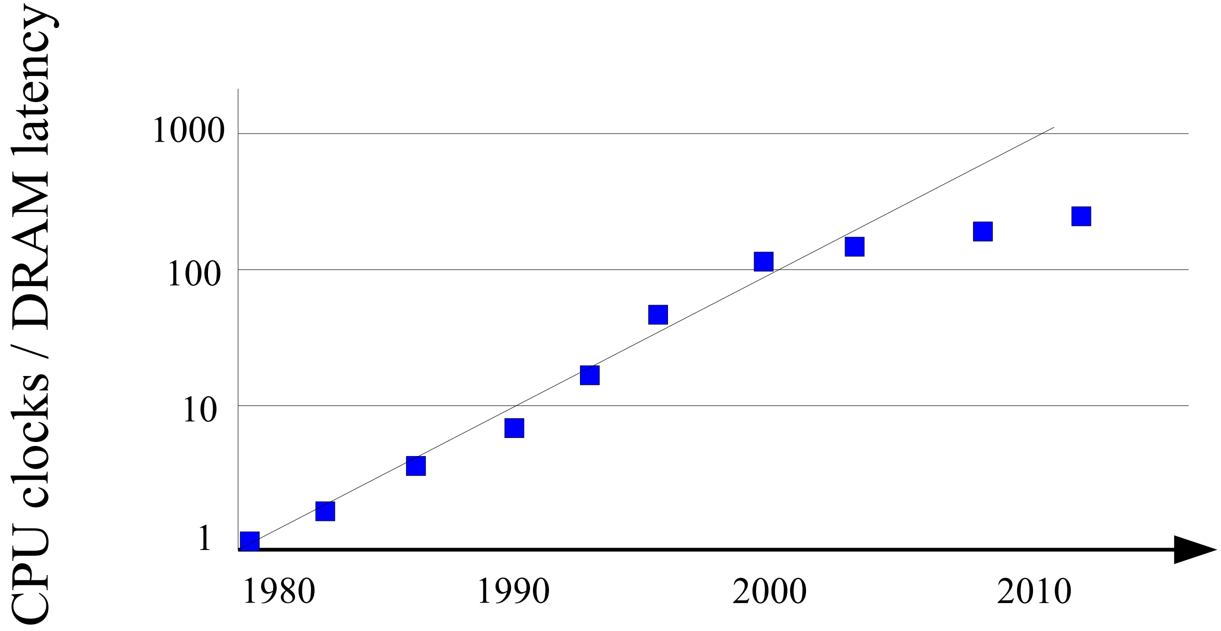
- Basic processing
  - Fetch, decode, and issue instructions
  - You could write a simple interpreter...
- CISC[1] or RISC[2]
  - Complex instructions versus simpler instructions
  - RISC = "*Relegate Interesting Stuff to Compilers*"
  - Sometimes CISC outside, but RISC inside...



1. Complex Instruction Set Computers (CISC)
2. Reduced Instruction Set Computers (RISC)

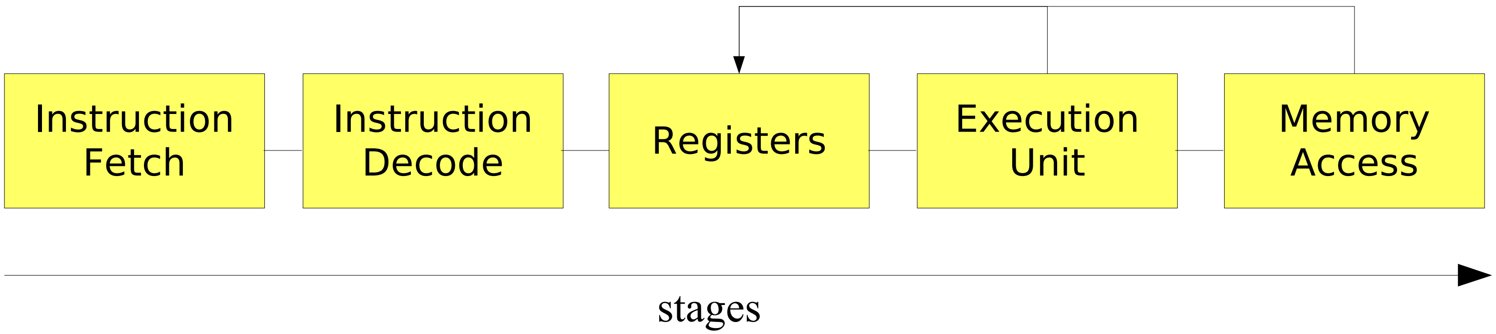
Processor

- The challenge is the memory barrier...
  - Well over 100 of cycles
  - Flattening due to the flattening of CPU clock frequency



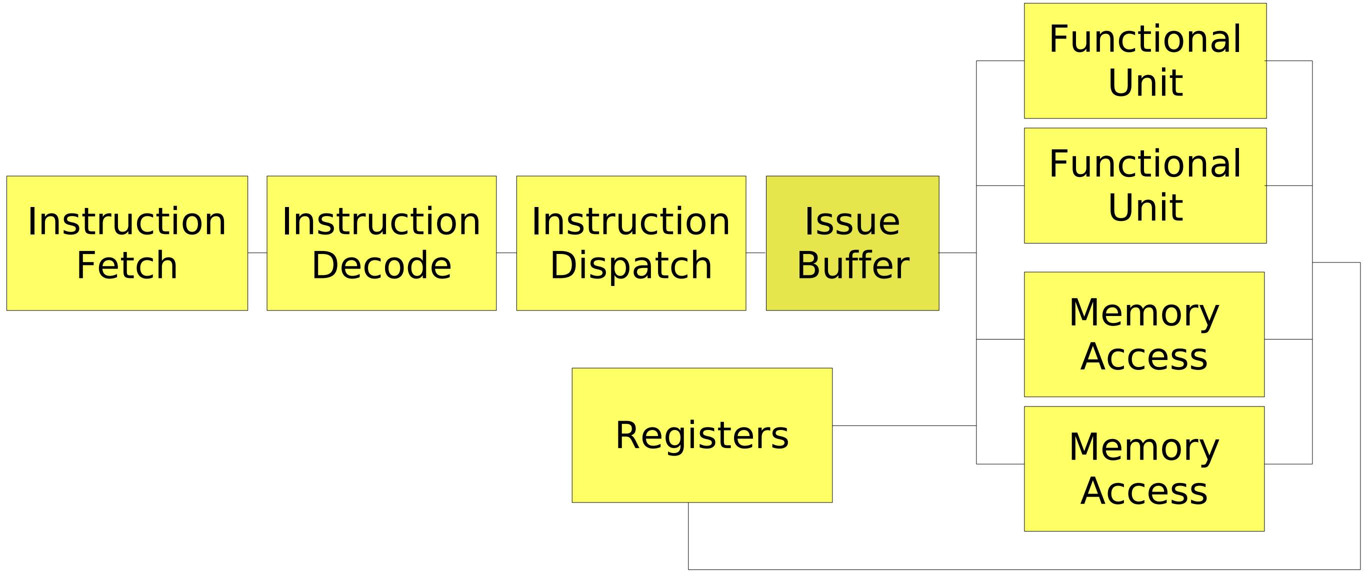
Processor

- Different types of processors
  - In-order pipeline
  - Superscalar
  - Very Long Instruction Word (VLIW)
- In-order pipeline
  - Multiple instructions may be in the pipeline at the same time
  - Only one instruction is in each stage at any given time
  - Stalls happen when instructions must wait for their operands



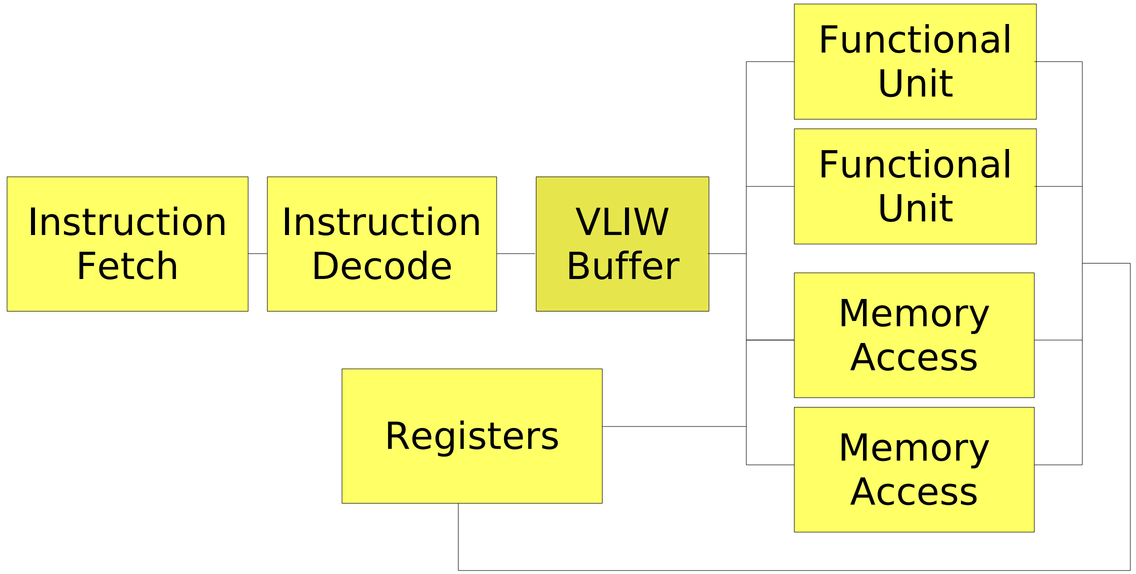
Superscalar Processors

- High-performance processors
  - Introduce automatic instruction-level parallelism
    - Several instructions can be fetched and decoded in the same clock cycle
    - Decoded instructions are dispatch into instruction issue buffer
      - Begin execution when their input operands are ready
      - Without regard to the original program sequence
  - Properties
    - Peak instruction throughput is higher
    - Hopefully reduces stalls



VLIW Processors

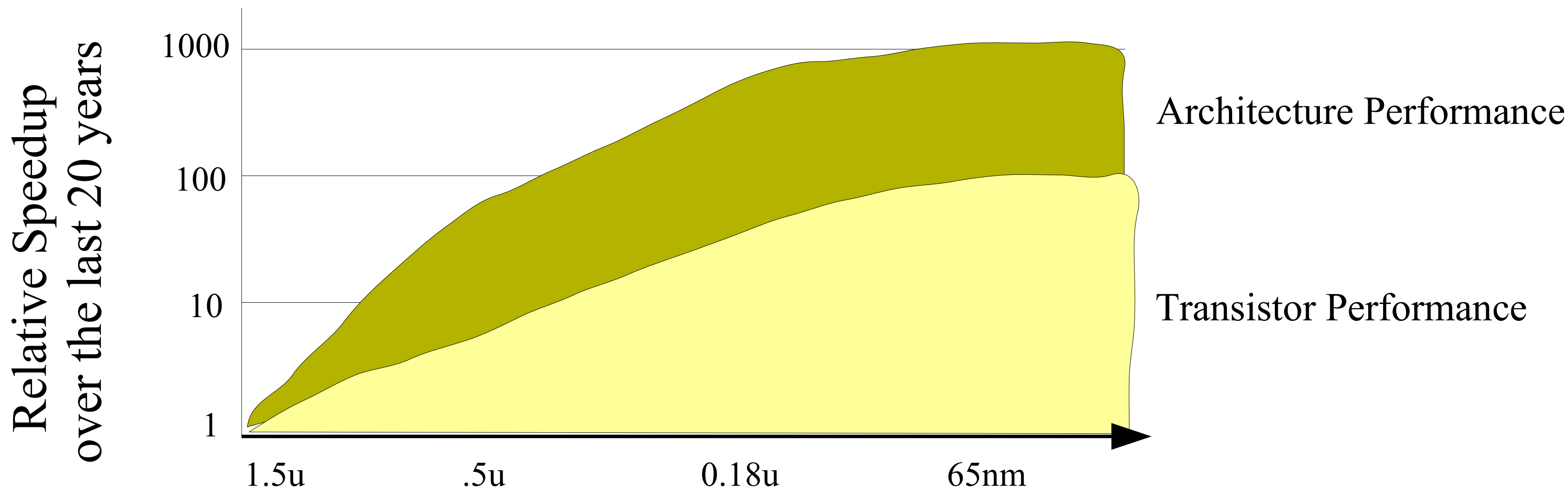
- Compilers must produce VLIW
  - Combine parallel intructions into a very long instruction word (VLIW)
  - VLIW are executed in sequence
- VLIW parallelism
  - A VLIW can be fetched and decoded in the same clock cycle
  - The instructions of the VLIW proceed in parallel
  - Begin execution when their input operands are ready

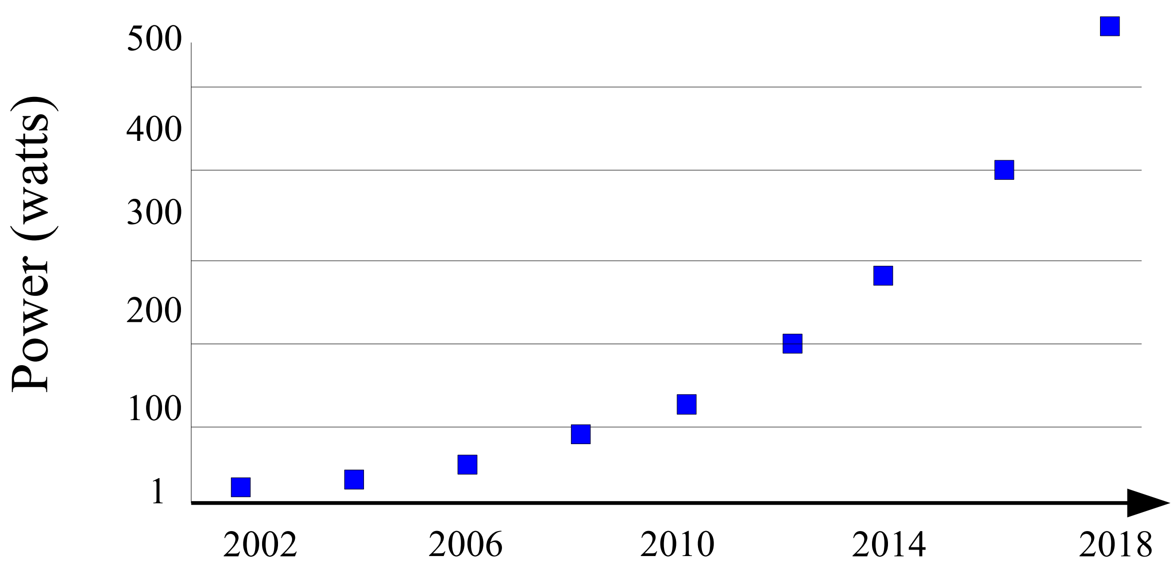


Hyper-threaded Processors

- Still the memory barrier…
  - As processors are going faster, the memory barrier is increasing
  - Can the hardware switch threads when staling?
- Operating system scheduling
  - More often well over thousands of instructions
  - Incompatible with the few-hundred-instruction-long stalls
- Faking cores
  - The OS sees multiple cores, but they are virtual cores
  - The hardware has everything it needs to context switch

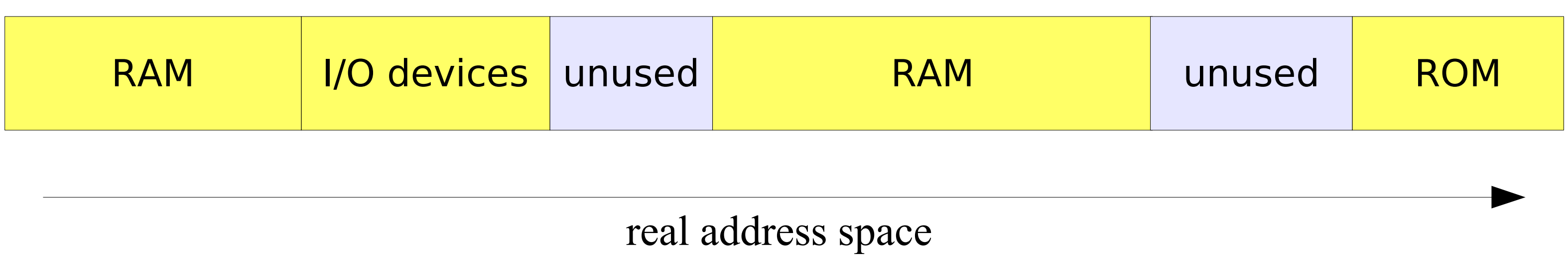
Discussing the Limits





Memory

- Memory System
  - A combination of main memory and cache memories
    - Cache memories are generally hidden from software (hardware managed)
    - Memory access is at least per byte, but it may be a word (16 or 32 bits)
    - Often, memory access is per *line* of 32 to 128 bytes
  - Composite main memory
    - The address space may be composed of different types of memory
      - RAM, ROM, I/O memory, others
      - Each may have its own instruction sets for reading or writing
    - Usually divided in pages (like 4KB pages)
      - Pages may have different access privileges (read, write, execute)

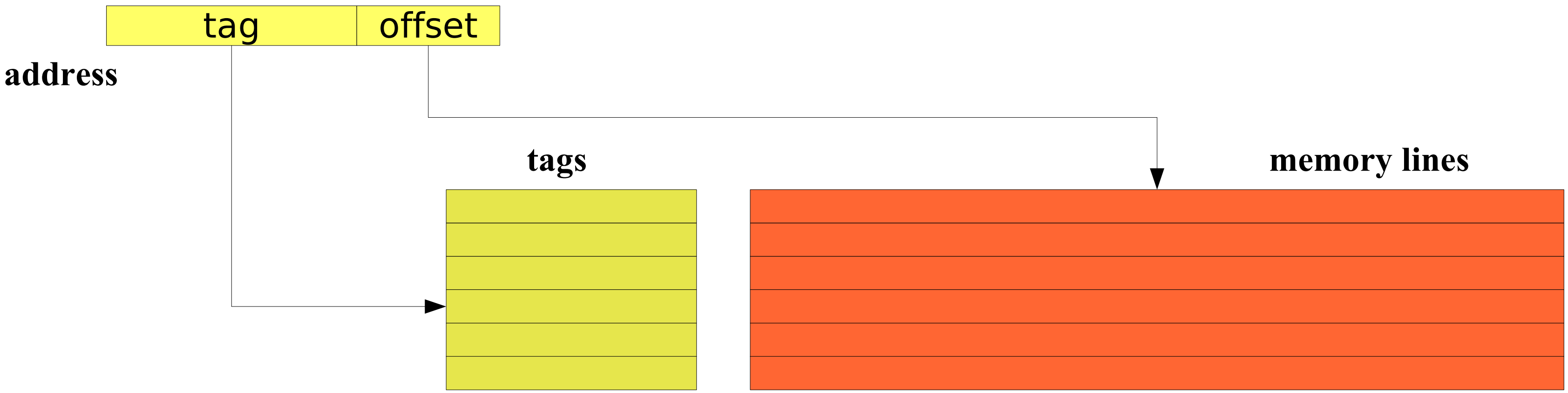


Memory

- Cache Memories
  - Hiding high memory latencies
    - Many tens or hundred clock cycles (for in-memory pages)
  - Works on the principle of *locality*
    - *Temporal locality* (what has been used recently is likely to be used again)
    - *Spatial locality* (what is close to what is being used is likely to be used)
  - In 65nm technology
    - 10MB on-die cache (L1)
    - As much as 40% of total die area

Memory

- Cache Memories
  - Caches memory lines (called *cache lines*)
    - A *cache hit* finds the addressed data in the cache
    - A *cache miss* does not and loads a memory line
    - A replacement algorithm must be in place to free cache lines



Input/Output Systems

- Architecture
  - Consist of a number of buses
    - That connect the processor and memory to I/O devices
    - Such buses are often standardized (PCI or AGP)
    - Devices often use a controller to connect to such buses
  - A bus is a conduit for device commands and for data transfers

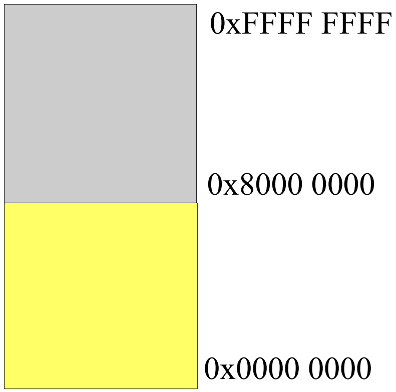
- Different Designs
  - Programmed I/O
    - Processor issues a request and polls for its completion
  - Interrupt-driven I/O
    - Processor issues a request and is interrupted when completed
    - Processor controls any data transfer from the controller to memory
  - DMA I/O
    - Processor issues a request and is interrupted when completed
      - Controllers have *Direct Memory Access* (DMA)
    - Could use special processors called *I/O Processors* (IOPs)

User-level ISA

- Instruction Set Architecture
  - *Storage resources*, e.g. memory and registers
  - An *instruction set*
- Register Architecture
  - General-purpose registers
    - Used to hold any operands to instructions
  - Typed registers
    - Such as floating-point registers
  - Special-purpose registers,
    - Program Counter (PC), status registers or stack registers

User-level ISA

- Memory Architecture
  - Defines through an address space
    - Usually 32bit addresses
    - Could be 64bit on newer processors
    - Usually divided between user and kernel
  - Flat or segmented address space
    - Flat address space
      - Addresses in load/store instructions represent virtual addresses
      - The MIPS 32-bit ISA has a flat address space, from 0x00 to 0x7FFF FFFF
    - Segmented
      - Addresses in load/store instructions are relative to segments
      - The Intel IA-32 and PowerPC have both a segmented address space

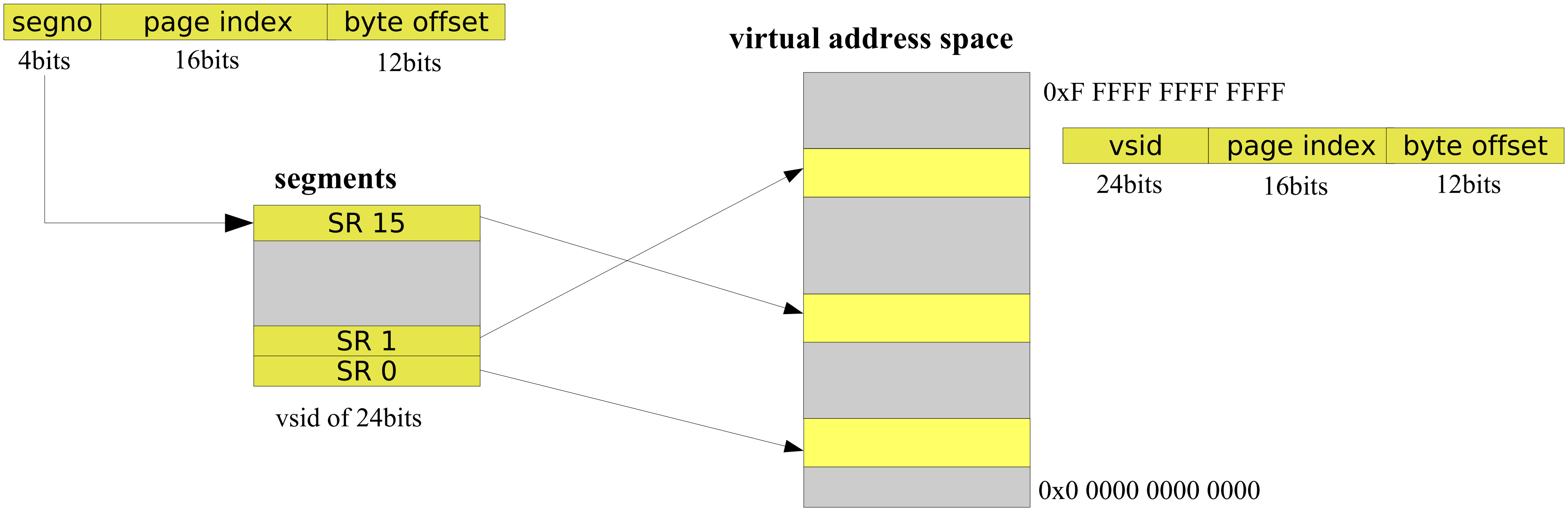


User-level ISA

- Memory Architecture
  - The Intel IA-32 memory
    - Virtual addresses are 32bit addresses
    - Supports up to 64K segments, each segment is up to 4 GB
    - Provide only 6 segment registers
      - Hence, at any point in time, only 6 segments are accessible
      - Each load/store instruction specifies a segment and 32bit offsets
        - The offset can be an immediate value or the addition of an immediate value and the content of a general-purpose register
  - Could be used as a flat address space
    - By setting all segment registers to the same base address
    - Done by both Unix and Windows

User-level ISA

- Memory Architecture
  - The PowerPC three types of addresses
    - Effective addresses (32-bit address space)
      - Divided into 16 segments of 256MB (28 bit)
      - Top 4 bits index the segment register (SR0 to SR15)
      - Notice that pointer arithmetics may change the segment index
    - Virtual addresses (52-bit address space)
      - But real addresses are 32-bit addresses



User-level ISA

- User-level Instruction Set
  - A mean of transforming data held in registers and memory
  - Instructions are grouped according to what they manipulate

Memory Instructions	**Integer Instructions**	**Floating-point Instructions**	**Branch Instructions**
load byte load word store byte load double load float ...	add compare logical exclusive OR rotate left with carry to-byte or to-long ...	add float add double convert to integer compare double compare float ...	relative branch absolute branch branch if-negative jump to subroutine return ...

User-level ISA

- Memory Instructions
  - Load from memory to a register, store a register to memory
  - User-level addresses called *virtual*, *logical* or *effective addresses*
- Integer Instructions
  - Such as arithmetic, logical and shift operations
  - In CISC[1] ISAs
    - Addressing mode may be a mix of registers and offsets
    - Arithmetic instructions may involve registers and memory locations
  - In RISC[2] ISAs
    - A simpler instruction format
    - Addressing mode may still be a mix of registers and offsets
    - Arithmetic instructions are only on registers

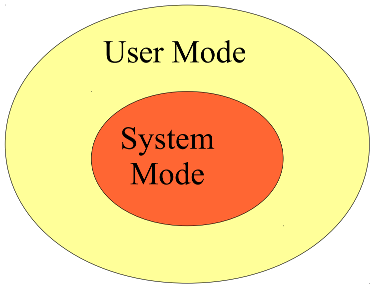
1. Complex Instruction Set Computers (CISC)
2. Reduced Instruction Set Computers (RISC)

User-level ISA

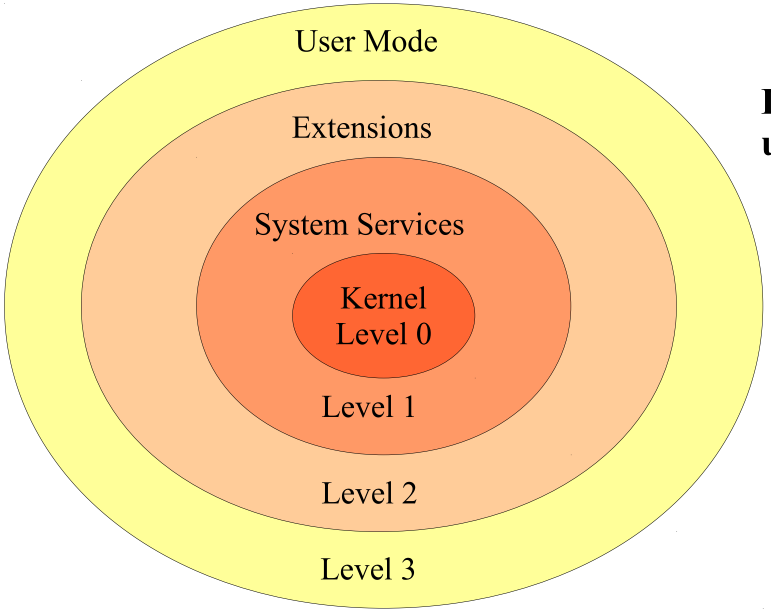
- Floating-point Instructions
  - Usually refers to floating-point registers
    - The Intel IA-32 uses a stack for floating-point registers
    - Other architectures may use directly accessible typed registers
- Branch Instructions
  - Branch instructions change the flow of control
    - Accomplished by changing the *Program-Counter* register
    - Changes where the next instruction is fetched
    - Greatly impacts the pipeline effectiveness
  - Different branch instructions
    - Branch instructions may be conditional or indirect (using a register)
    - Branch-and-link, a jump to a subroutine that also saves a return address

System-level ISA

- Resource Management
  - User-level ISA
    - Mostly about getting user tasks done
  - System-level ISA
    - Mostly about management of system resources
      - Process, memory and I/O management
    - Require priviledges
      - *User mode* versus *system mode (also called kernel or priviledged)*



Most OS relies on two levels only



Intel IA-32 supports up to four levels

System-level ISA

- System-level Registers
  - Most ISAs include special registers
    - To assist with hardware resource management
  - System clock register
    - Records the number of clock ticks elapsed since last reset
  - Trap and interrupt registers
    - Records information about the occurrence of traps and interrupts
    - *Mask register* inhibits or allows traps and interrupts
  - Translation Table Pointers
    - Support virtual address spaces
    - Maps memory pages or segments to real memory

System-level ISA

- Processor Management
  - Requires minimal support
    - A *system-return instruction*
      - Jumps to user code
      - Switches to user-level mode
    - Interval timer
      - Getting back the control after some elapsed time
      - Uses an interrupt to switch back to system mode
  - Traps and interrupts
    - Need specific support (mix of hardware and software)

System-level ISA

- Processor Management
  - Traps and interrupts
    - A trap occurs as a side effect of the execution of an instruction
      - Corresponds to *exception conditions* such as arithmetic overflows, page faults or violations of memory-access priviledges...
      - The ISA specifies traps on a per instruction basis
    - Interrupts are caused by the occurance of external events
      - Interrupts are not related to the execution of specific instructions
      - Examples are I/O interrupts or timer interrupts
    - Traps and interrupts may be masked
  - Trap-like Instructions
    - Some instructions are designed to act as explicit or conditional traps
      - The most important example is the system-call instruction
      - Details about system calls are part of the *Application Binary Interface*

System-level ISA

- Trap Handling
  - Processor goes into a *precise state* with respect to the trapping instruction
    - All instructions prior to the trapping instruction are completed and make all their specified register and memory modifications
    - Depending on the ISA, the instruction causing the exception either completes (e.g. an overflow exception) or does not cause any change of state (e.g. page faults)
    - None of the instructions following the trapping instructions modify the registers or memory in any way (this is important when having instruction-level parallelism, either pipeline or superscalar)
  - The program counter is saved
    - In an ISA-specific location (either register or memory).
    - Some or all of the registers may be saved by the hardware implementation
    - On RISC processors, this is left to the trap- or interrupt-handling software
  - The processor is placed in system mode

System-level ISA

- Trap Handling
  - Control is transferred to a memory location that is specified in the ISA
    - This code may complete the save of the *precise state* of the processor
      - E.g. saves registers if the hardware didn't do it
    - This code may transfer execution to a user-level handler
      - Like in the case of arithmetic overflow
  - Upon trap-handing completion
    - Restore the saved *precise state*
    - Jumps back to the location that trapped
      - For most traps, the trapped instruction is re-executed
      - Otherwise, the trapped instruction just completes and the execution proceeds with the next instruction in sequence

System-level ISA

- Interrupt Handling
  - Interrupts are treated in a manner similar to traps
    - The *precise state* of the processor must be produced
  - Some liberty
    - Because it is caused by an external event, there is some liberty in deciding when to treat an interrupt, making the saving of a precise state simpler
  - Interrupts may be disabled
    - Some interrupts are not maskable such as *power-failure* or *high-temperature* interrupts

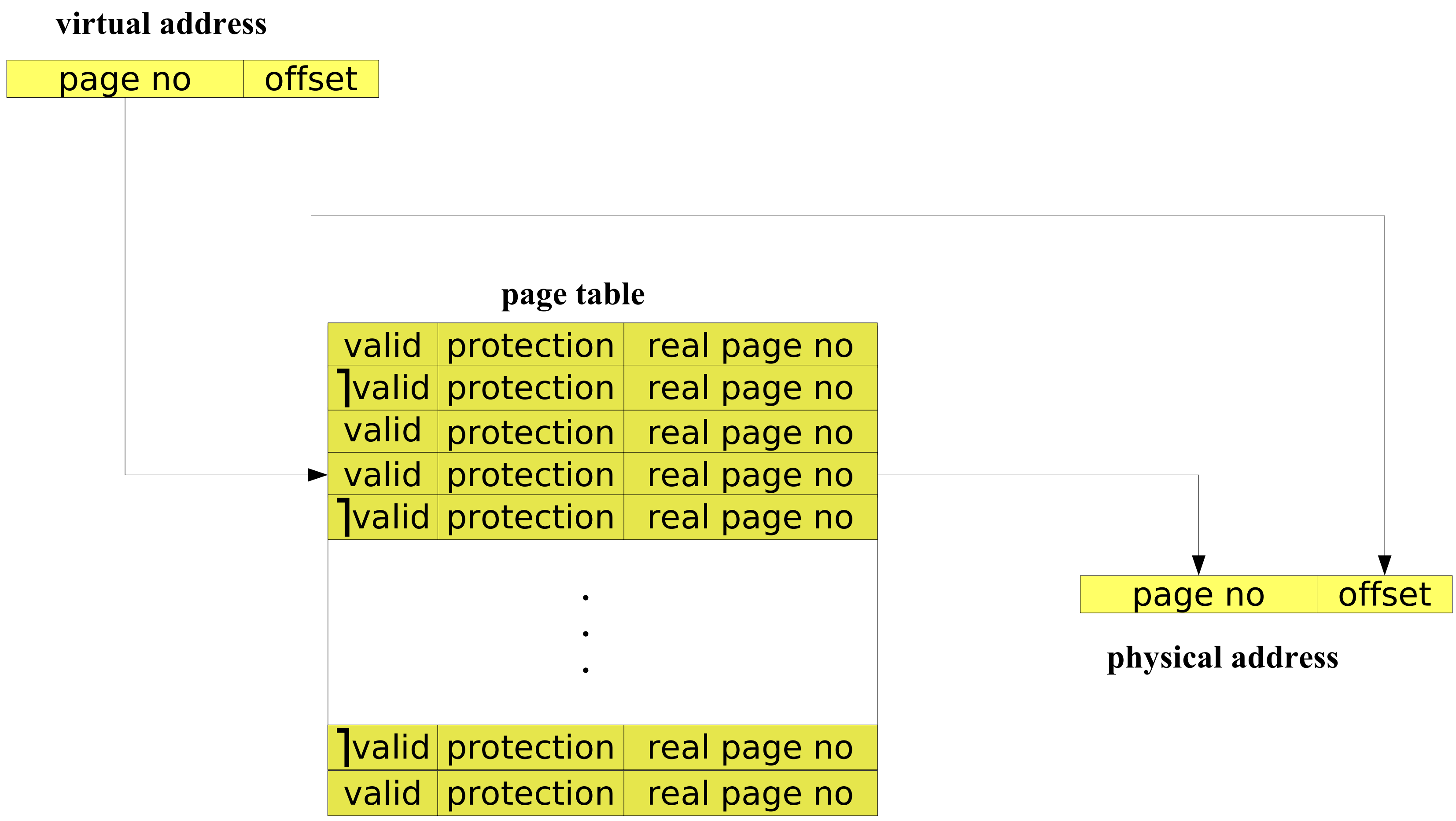
System-level ISA

- Memory Management
  - Goals
    - Provide virtual memory larger than physical memory
    - Share physical memory amongst processes
    - Isolate processes
    - Provide fine-grain access protection (read/write/execute)
  - Main concepts
    - Page Tables
      - Supports virtual-to-physical memory mapping
    - Translation Look-aside Buffer
      - Small associative cache to speed-up address translation

System-level ISA

- Page Tables
  - Supports virtual-to-physical memory mapping
    - One such page table per process
    - Requires a replacement strategy (often Least-Recently-Used)
  - Per virtual page
    - Valid bit (the page is mapped in memory or not)
    - Protection bits (read, write and execute privileges)
    - The page address in physical memory

System-level ISA



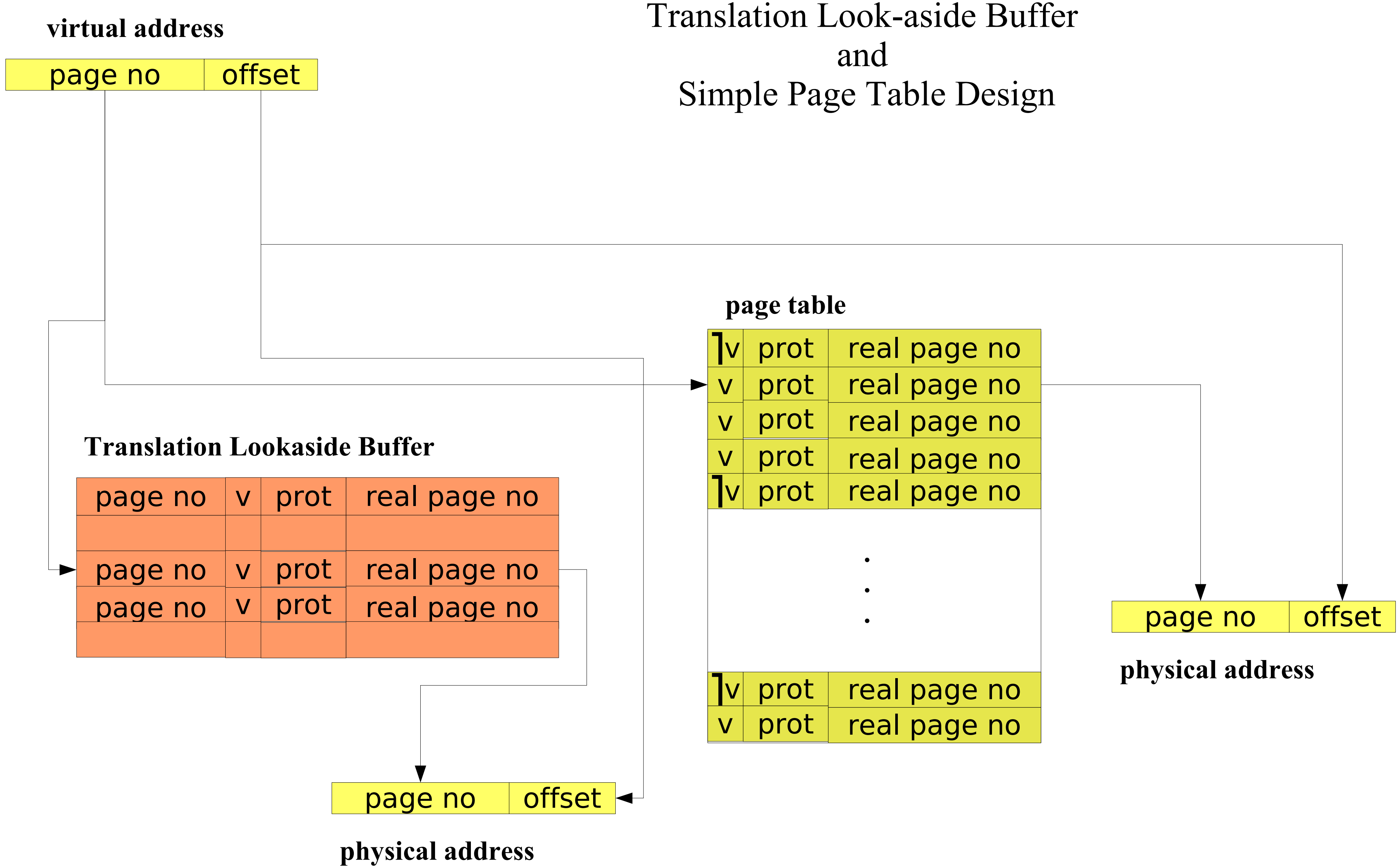
Simple Page Table Design

System-level ISA

- Memory Management
  - Translation Look-aside Buffer
    - Small associative cache to speed-up address translation
  - In most architectures
    - A lookup is done in parallel with a cache access
    - Hence, TLBs incur no specific performance penalty
  - TLB caches page protections
    - So that accesses can be checked at the TLB level if there is a hit

System-level ISA

Translation Look-aside Buffer
and
Simple Page Table Design



System-level ISA

- Memory Management
  - Mix of hardware and software, the frontier depends on the ISA
- Architected Page Table
  - Page table defined in the ISA
    - TLB is in hardware, mostly transparent but for a purge instruction
  - A page table miss is a trap
    - The information about the page fault is defined in the ISA
    - Page table format is defined in the ISA
- Architected TLB
  - TLB defined in the ISA
    - Special instructions to read or write TLB entries, a TLB miss is a trap
  - Page table is done in software
    - Without design constraints, the hardware is unaware of the page table
    - Opens the possibility for inverted page tables for large address spaces

System-level ISA

- Memory Management

	Architected TLB	Architected Page Table
TLB entry format	Defined in ISA	Left to hardware design
TLB configuration	Defined in ISA	Left to hardware design
Page table entry format	Left to OS implementation	Defined in ISA
Page table configuration	Left to OS implementation	Defined in ISA
Miss in TLB	Causes TLB fault to OS	Hardware accesses page table
Miss in page table	Detected and handled by OS	Causes page fault
New entry in TLB	Made by OS	Made by hardware
New entry in page table	Made by OS	Made by OS

System-level ISA

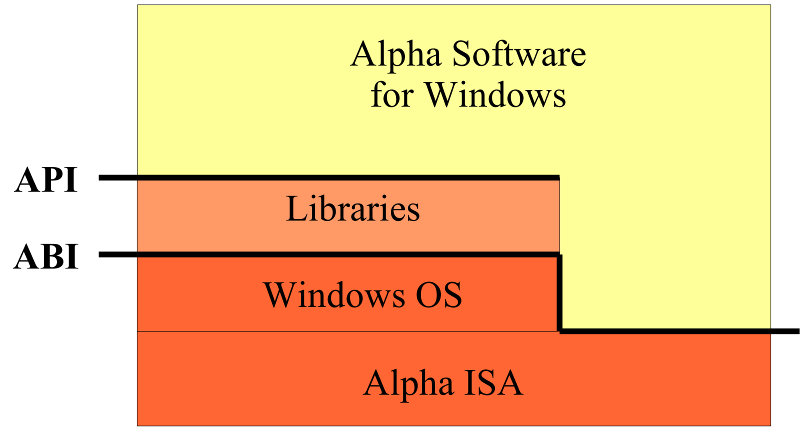
- Examples
  - Architected Hierarchical page tables
    - A page table is organized as a hierarchy
    - The hardware knows how to walk through the hierarchy
    - The software knows how to add/remove/change an entry
  - Inverted page tables
    - A page table is organized as a hash table
    - The hardware knows only how to use the TLB
    - Hashtable lookups and inserts are achieved in software

System-level ISA

- Input/Output Management
  - Some ISA have specific I/O instructions
    - The instructions look like load and store instructions
      - The address identifies the device
      - The value is either data or command
    - Examples: IBM System/360 or the Intel IA-32
  - Some ISA have memory-mapped I/O
    - Use regular load and store instructions
      - Not on real memory however, within a special address range
        - The address identifies the device or a special port of a device
        - One device may be mapped at several memory location
      - The read/written value is either data or command
  - Interrupts are a part of most I/O architectures
    - A way for getting the attention of the operating system
    - Indicate an external event or the completion of a request

Operating System Interface

- The foundation
  - A process is the foundation of this virtualization
    - A virtual address space, with one or more threads
    - System calls, a way to request a service from the OS
    - Signals for handling traps and interrupts
  - ABI versus API
    - Usually, applications do not use the binary interface directly
    - Applications use libraries offering a higher-level programming interface



Digital Alpha Machine running Windows OS

Operating System Interface

- Process
  - A virtualized memory space
    - Provides two illusions
      - Owning the entire memory
      - A potentially larger amount of memory
    - Mapping to real memory through a page table
  - Process switching
    - Steps
      - Require to change the page table pointer
      - Flush the TLB, and L1,L2 caches
    - Overhead
      - Memory barrier hit
        - TLB is empty, so page table lookup will occur
        - The content of L1,L2 caches are irrelevant and must be flushed
      - Potential disk barrier hit
        - Pages in memory may not be the one needed

Operating System Interface

- Threads
  - A virtualized execution flow
    - Reified through a Thread Control Block (TCB)
      - A program counter, user-level processor registers
      - A stack pointer for push and popping stack frames
    - Needs a stack
      - A contiguous memory segment for the stack
      - Using memory protection to grow the stack when necessary
  - Thread switching
    - Threads are interrupted through the timer interrupt
      - The scheduler is the interrupt handler for the timer
        - The scheduler finishes the save of the thread context
        - It chooses what thread should be next to run
        - Restore the context of that thread, jump back to user-level code
        - Invalid TLB if switching between processes
    - Overhead
      - L1,L2 caches and TLB most likely irrelevant
      - Pipeline stalls (new working set and new locality)

Operating System Interface

- Signals
  - Signals virtualize ISA traps and interrupts up to user-level code
    - Timer interrupt or overflow trap
    - Memory violation traps (protection violation or non-valid address)
  - Signal handlers
    - Default handlers are provided
    - Applications may redefine them with user-level handlers
    - Through the sigvec() system call
  - Signals may be masked
    - Through the sigblock() or sigsetmask() system call
    - Some signals cannot be masked (SIGSTOP and SIGKILL)
  - Signal occurrences
    - Either because of real traps or interrupts
    - Could be software generated through the kill() system call

Operating System Interface

- System Calls
  - A trap in kernel mode
  - Carries different service requests
    - Either through values in ISA-specified registers
    - Or through data structures in memory
    - This is all operating system specific
  - Different system calls
    - For process management
    - For memory management
    - For Input/Output operations

```
#include <syscall.h>
extern int syscall(int,...);

int file_close(int fileDescriptor) {

  return syscall(SYS_close, fileDescriptor);
}
```

Operating System Interface

- System Calls
  - Process management system calls
    - Create or terminate processes
      - Examples such as Linux fork(), exec() or exit() system calls
    - Other system calls
      - Synchronization ones such as wait(), sleep() or wakeup()
      - Others such as setpriority() or getrusage()
  - Memory management system calls
    - Use the malloc/free API, internally uses the sbrk() system call
    - Manipulate memory protection through mprotect() system call
    - Shared mapped segments through shmget() system call

Operating System Interface

- System Calls
  - Input/Output system calls
    - Applications do not directly use I/O instructions or I/O memory
      - I/O memory is not mapped in application processes
      - I/O instructions are privileged
    - Make device-independent system calls
      - Like open(), read(), write(), and close()
  - Two Device Categories
    - Character devices
      - Direct communication with application code
      - A character at a time
    - Block devices
      - Larger granularity of interactions
      - Data transfers happen through memory buffers

Operating System Interface

- Device drivers
  - Execute privileged code
  - Implements device-independent system calls in a device-dependent way
  - Directly using I/O instructions or load and store instructions in I/O memory
  - Responsible of both issuing commands and handling interrupts