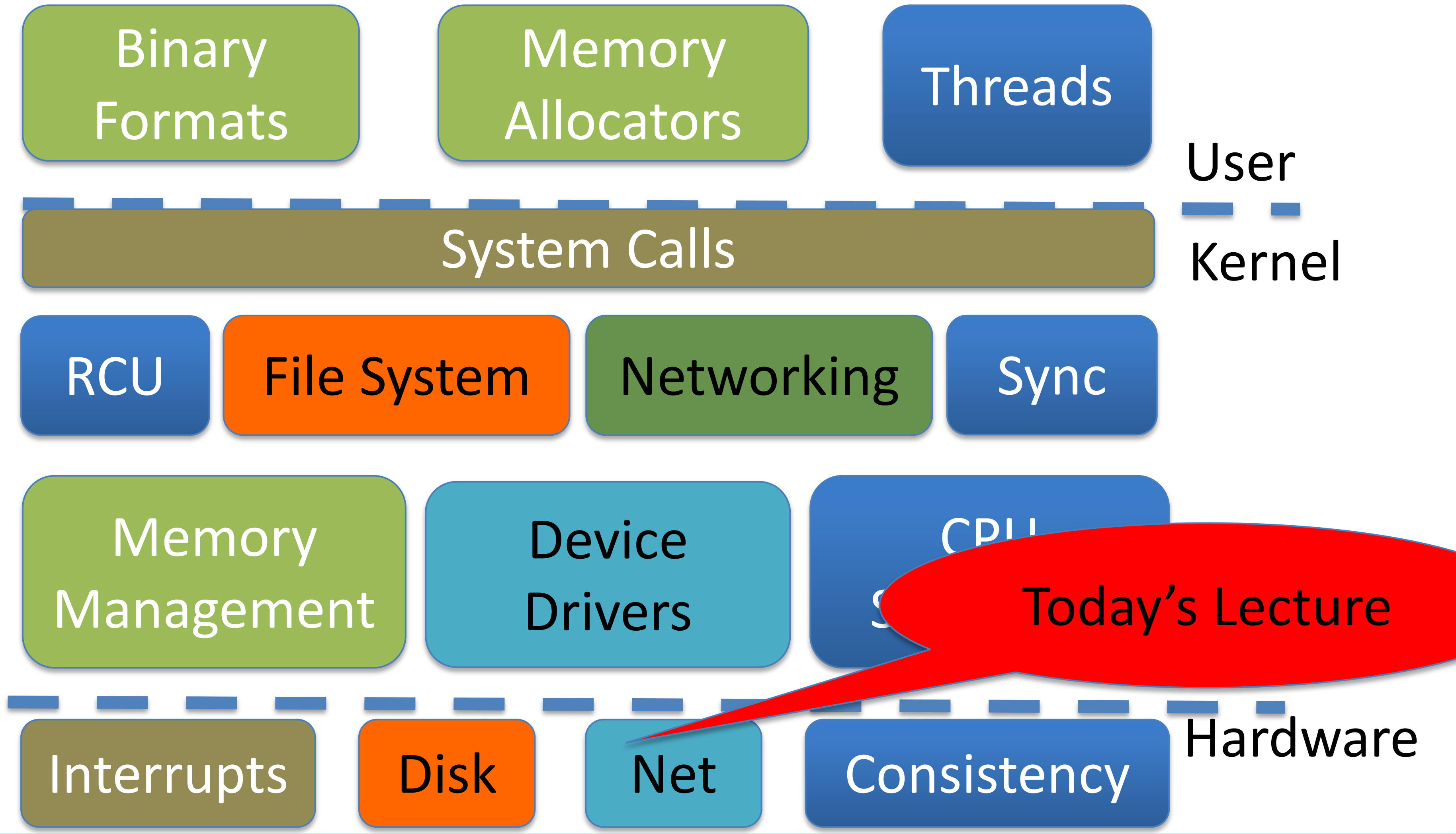
Device I/O Programming

Don Porter

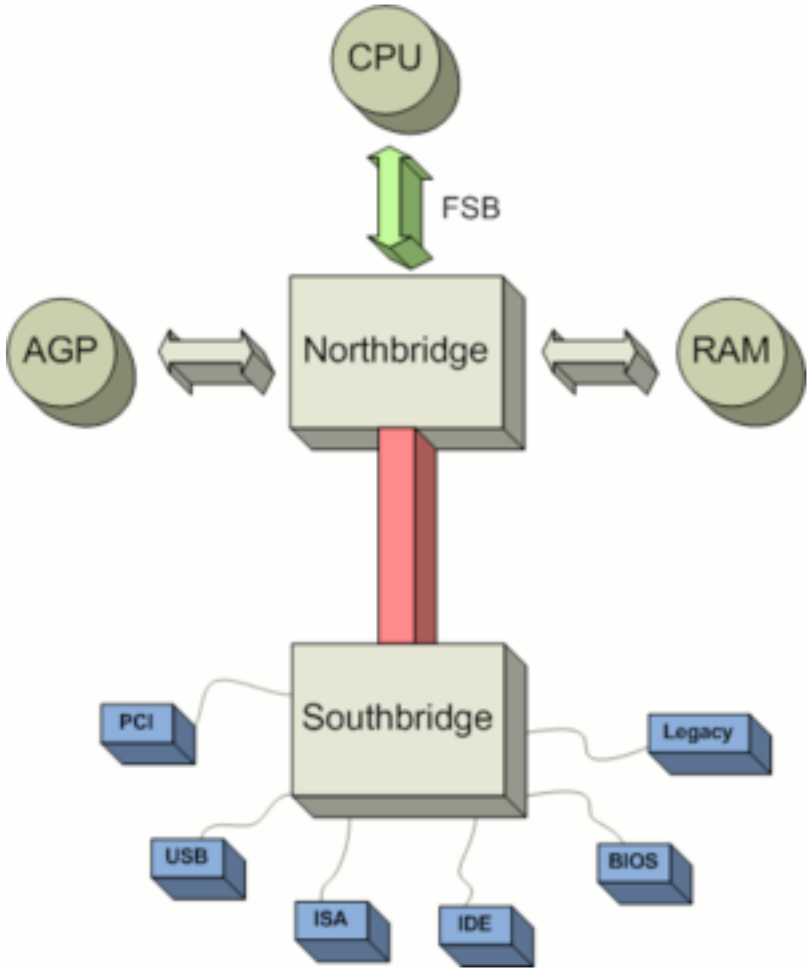
Logical Diagram



Overview

- Many artifacts of hardware evolution
  - Configurability isn't free
  - Bake-in some reasonable assumptions
  - Initially reasonable assumptions get stale
  - Find ways to work-around going forward
    - Keep backwards compatibility
- General issues and abstractions

PC Hardware Overview



- From wikipedia
- Replace AGP with PCIe
- Northbridge being absorbed into CPU on newer systems
- This topology is (mostly) abstracted from programmer

I/O Ports

- Initial x86 model: separate memory and I/O space
  - Memory uses virtual addresses
  - Devices accessed via ports
- A port is just an address (like memory)
  - Port 0x1000 is not the same as address 0x1000
  - Different instructions – inb, inw, outl, etc.

More on ports

- A port maps onto input pins/registers on a device
- Unlike memory, writing to a port has side-effects
  - “Launch” opcode to /dev/missiles
  - So can reading!
  - Memory can safely duplicate operations/cache results
- Idiosyncrasy: composition doesn’t necessarily work
  - outw 0x1010 <port> != outb 0x10 <port>

```
                          outb 0x10 <port+1>
```

Parallel port (+I/O ports)

(from Linux Device Drivers)

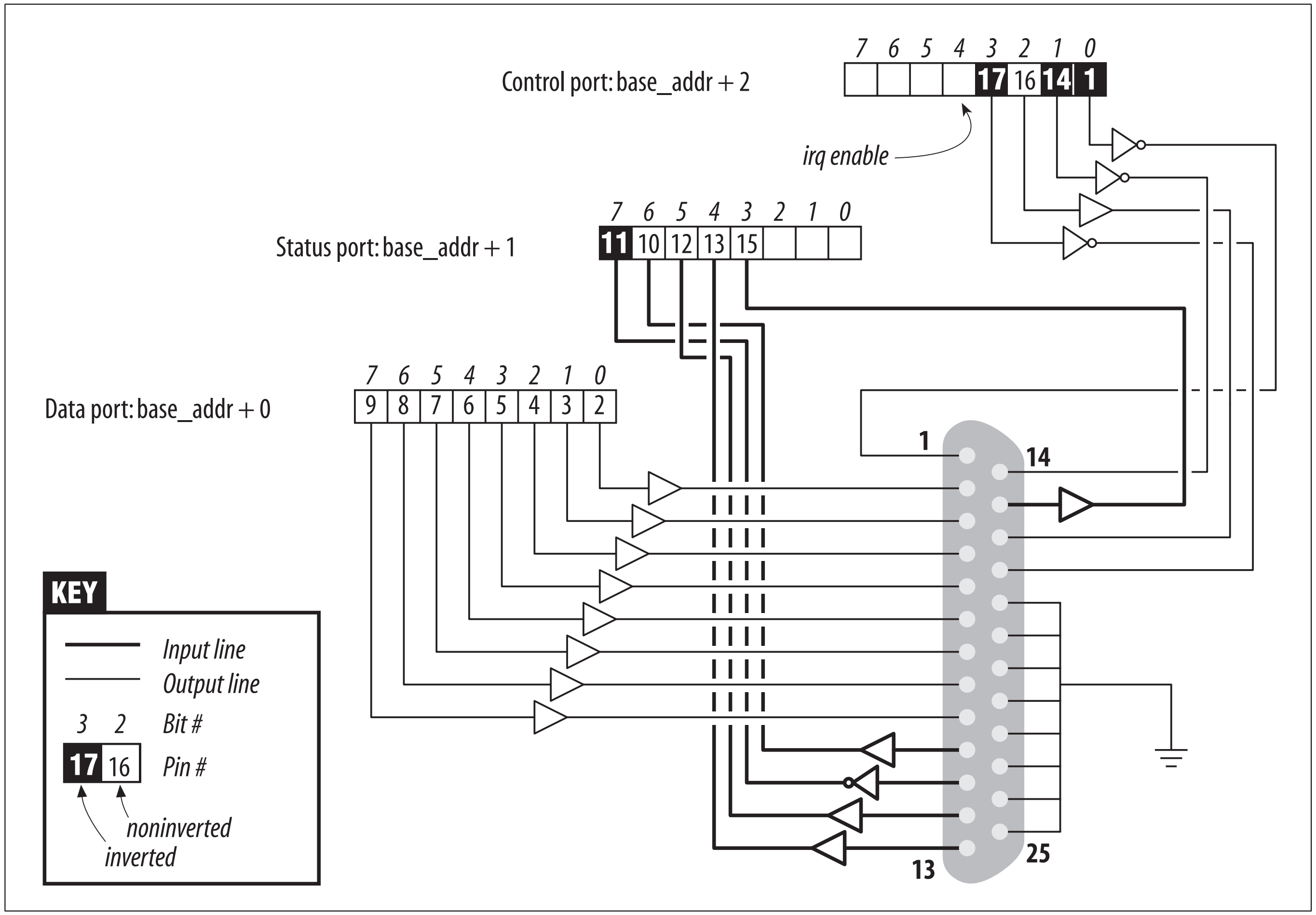


Figure 9-1. The pinout of the parallel port

Port permissions

- Can be set with IOPL flag in EFLAGS
- Or at finer granularity with a bitmap in task state segment
  – Recall: this is the “other” reason people care about the TSS

Buses

- Buses are the computer’s “plumbing” between major components
- There is a bus between RAM and CPUs
- There is often another bus between certain types of devices
  - For inter-operability, these buses tend to have standard specifications (e.g., PCI, ISA, AGP)
  - Any device that meets bus specification should work on a motherboard that supports the bus

Clocks (again, but different)

- CPU Clock Speed: What does it mean at electrical level?
  - New inputs raise current on some wires, lower on others
  - How long to propagate through all logic gates?
  - Clock speed sets a safe upper bound
    - Things like distance, wire size can affect propagation time
  - At end of a clock cycle read outputs reliably
    - May be in a transient state mid-cycle
- Not talking about timer device, which raises interrupts at wall clock time; talking about CPU GHz

Clock imbalance

- All processors have a clock
  - Including the chips on every device in your system
  - Network card, disk controller, usb controler, etc.
  - And bus controllers have a clock
- Think now about older devices on a newer CPU
  - Newer CPU has a much faster clock cycle
  - It takes the older device longer to reliably read input from a bus than it does for the CPU to write it

More clock imbalance

- Ex: a CPU might be able to write 4 different values into a device input register before the device has finished one clock cycle

- Driver writer needs to know this
  - Read from manuals
- Driver must calibrate device access frequency to device speed
  - Figure out both speeds, do math, add delays between ops
  - You will do this in lab 6! (outb 0x80 is handy!)

CISC silliness?

- Is there any good reason to use dedicated instructions and address space for devices?
- Why not treat device input and output registers as regions of physical memory?

Simplification

- Map devices onto regions of physical memory
  - Hardware basically redirects these accesses away from RAM at same location (if any), to devices
  - A bummer if you “lose” some RAM
- Win: Cast interface regions to a structure
  - Write updates to different areas using high-level languages
  - Still subject to timing, side-effect caveats

Optimizations

- How does the compiler (and CPU) know which regions have side-effects and other constraints?
  - It doesn’t: programmer must specify!

Optimizations (2)

- Recall: Common optimizations (compiler and CPU)
  - Out-of-order execution
  - Reorder writes
  - Cache values in registers
- When we write to a device, we want the write to really happen, now!
  - Do not keep it in a register, do not collect $200
- Note: both CPU and compiler optimizations must be disabled

volatile keyword

- A volatile variable cannot be cached in a register
  - Writes must go directly to memory
  - Reads must always come from memory/cache
- volatile code blocks cannot be reordered by the compiler
  - Must be executed precisely at this point in program
  - E.g., inline assembly
- __volatile__ means I really mean it!

Compiler barriers

- Inline assembly has a set of clobber registers
  - Hand-written assembly will clobber them
  - Compiler's job is to save values back to memory before inline asm; no caching anything in these registers
- "memory" says to flush all registers
  - Ensures that compiler generates code for all writes to memory before a given operation

CPU Barriers

- Advanced topic: Don’t need details
- Basic idea: In some cases, CPU can issue loads and stores out of program order (optimize perf)
  - Subject to many constraints on x86 in practice
- In some cases, a “fence” instruction is required to ensure that pending loads/stores happen before the CPU moves forward
  - Rarely needed except in device drivers and lock-free data structures

Configuration

- Where does all of this come from?
  - Who sets up port mapping and I/O memory mappings?
  - Who maps device interrupts onto IRQ lines?
- Generally, the BIOS
  - Sometimes constrained by device limitations
  - Older devices hard-coded IRQs
  - Older devices may only have a 16-bit chip
    - Can only access lower memory addresses

ISA memory hole

- Recall the "memory hole" from lab 2?
  - 640 KB – 1 MB
- Required by the old ISA bus standard for I/O mappings
  - No one in the 80s could fathom > 640 KB of RAM
  - Devices sometimes hard-coded assumptions that they would be in this range
  - Generally reserved on x86 systems (like JOS)
  - Strong incentive to save these addresses when possible

New hotness: PCI

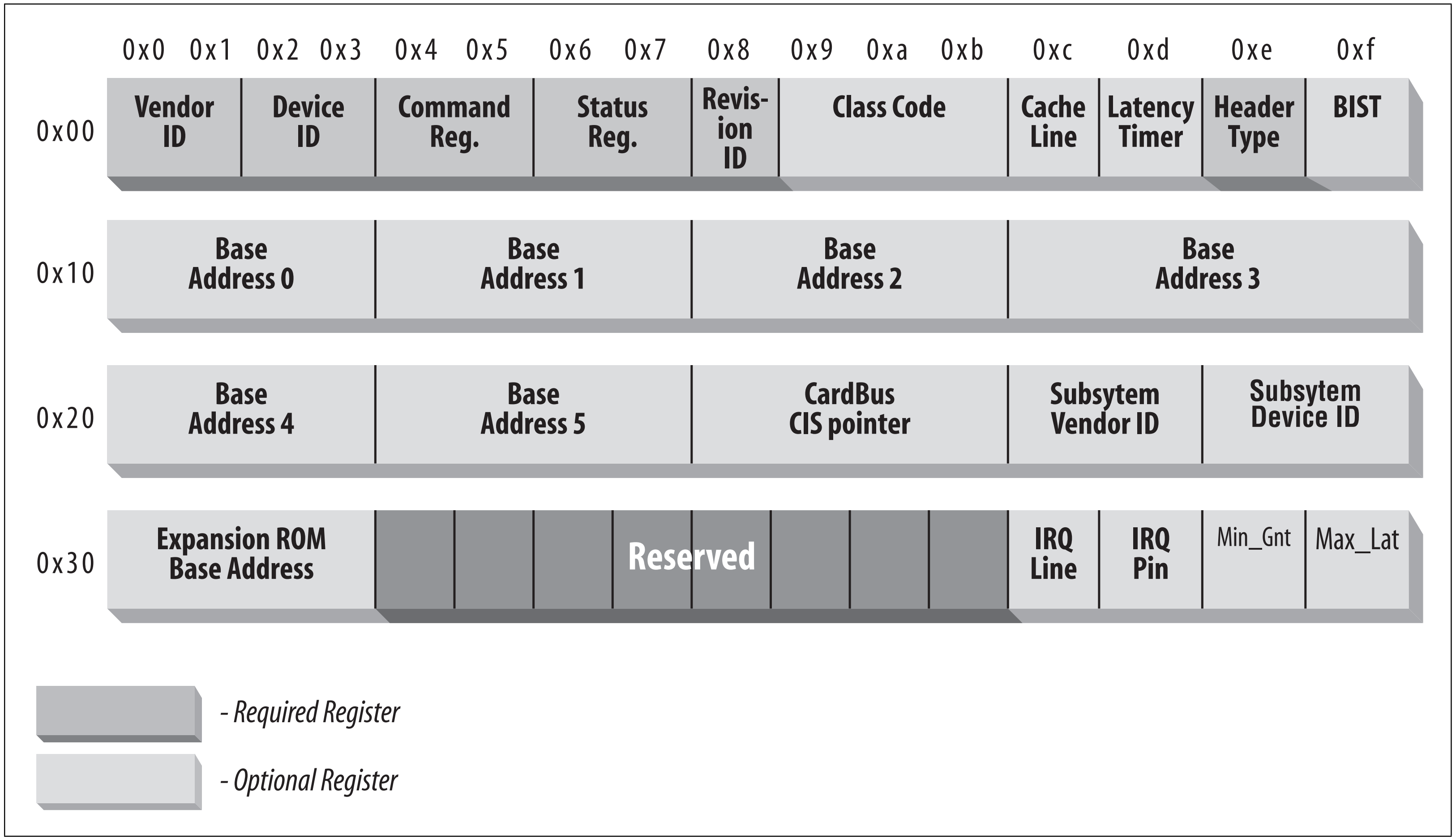
- Hard-coding things is bad
  - Willing to pay for flexibility in mapping devices to IRQs and memory regions
- Guessing what device you have is bad
  - On some devices, you had to do something to create an interrupt, and see what fired on the CPU to figure out what IRQ you had
  - Need a standard interface to query configurations

More flexibility

- PCI addressing (both memory and I/O ports) are dynamically configured
  - Generally by the BIOS
  - But could be remapped by the kernel
- Configuration space
  - 256 bytes per device (4k per device in PCIe)
  - Standard layout per device, including unique ID
  - Big win: standard way to figure out my hardware, what to load, etc.

PCI Configuration Layout

From device driver book



	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xa	0xb	0xc	0xd	0xe	0xf
0x00	Vendor ID		Device ID		Command Reg.		Status Reg.		Revis-ion ID	Class Code			Cache Line	Latency Timer	Header Type	BIST
0x10	Base Address 0				Base Address 1				Base Address 2				Base Address 3			
0x20	Base Address 4				Base Address 5				CardBus CIS pointer				Subsystem Vendor ID		Subsytem Device ID	
0x30	Expansion ROM Base Address				Reserved								IRQ Line	IRQ Pin	Min_Gnt	Max_Lat

▨ *- Required Register*

□ *- Optional Register*

Figure 12-2. The standardized PCI configuration registers

PCI Overview

- Most desktop systems have 2+ PCI buses
  - Joined by a bridge device
  - Forms a tree structure (bridges have children)

PCI Layout

From Linux Device Drivers

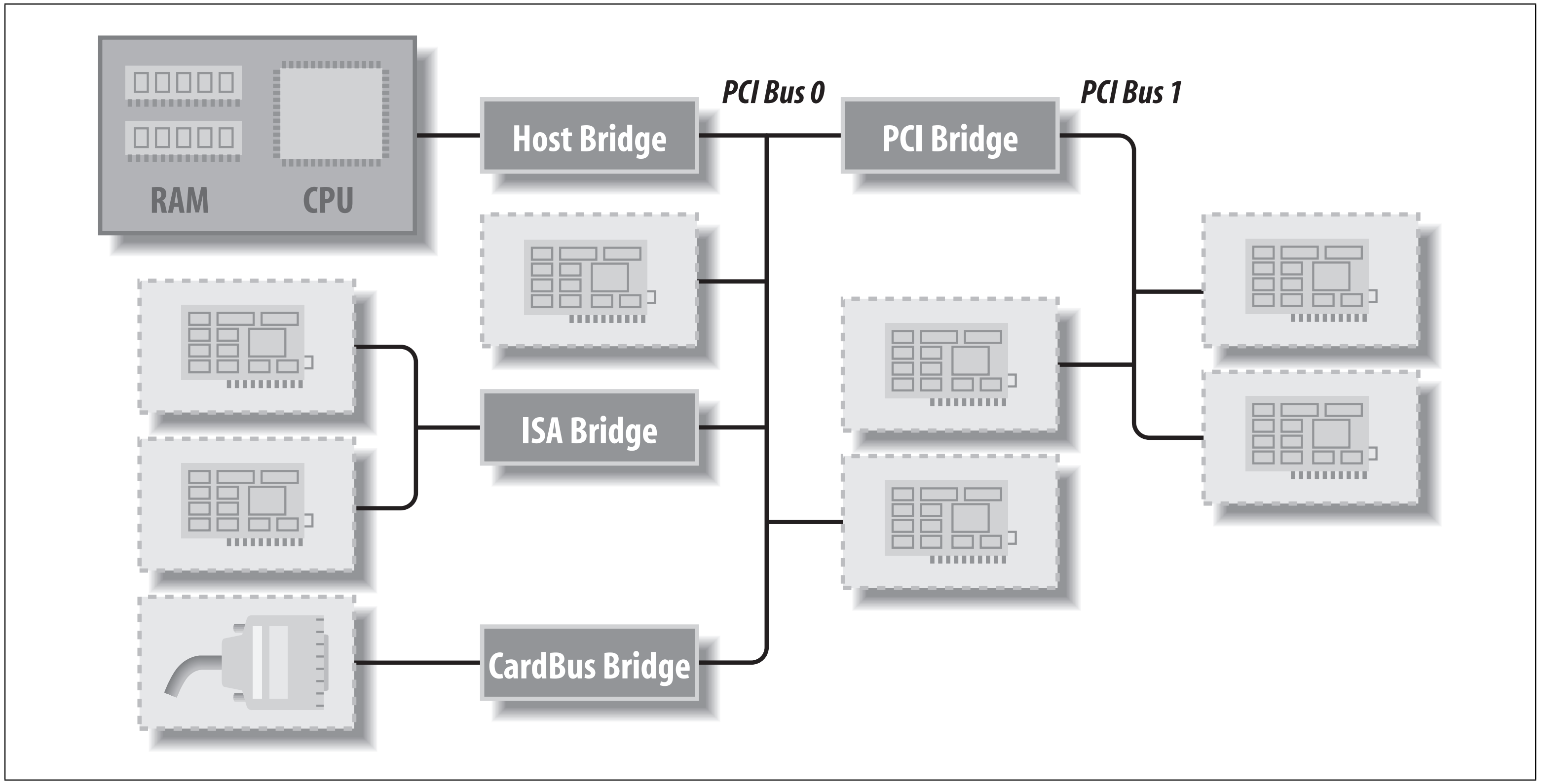


Figure 12-1. Layout of a typical PCI system

PCI Addressing

- Each peripheral listed by:
  - Bus Number (up to 256 per domain or host)
    - A large system can have multiple domains
  - Device Number (32 per bus)
  - Function Number (8 per device)
    - Function, as in type of device, not a subroutine
    - E.g., Video capture card may have one audio function and one video function
- Devices addressed by a 16 bit number

PCI Interrupts

- Each PCI slot has 4 interrupt pins
- Device does not worry about how those are mapped to IRQ lines on the CPU
  - An APIC or other intermediate chip does this mapping
- Bonus: flexibility!
  - Sharing limited IRQ lines is a hassle. Why?
    - Trap handler must demultiplex interrupts
  - Being able to "load balance" the IRQs is useful

Direct Memory Access (DMA)

- Simple memory read/write model bounces all I/O through the CPU
  - Fine for small data, totally awful for huge data
- Idea: just write where you want data to go (or come from) to device
  - Let device do bulk data transfers into memory without CPU intervention
  - Interrupt CPU on I/O completion (asynchronous)

DMA Buffers

- DMA buffers must be physically contiguous
- Devices do not go through page tables
- Some buses (SBus) can use virtual addresses; most (PCI) use physical (avoid page translation overheads)

Ring buffers

- Many devices pre-allocate a "ring" of buffers
  - Think network card
- Device writes into ring; CPU reads behind
- If ring is well-sized to the load:
  - No dynamic buffer allocation
  - No stalls
- Trade-off between device stalls (or dropped packets) and memory overheads

IOMMU

- It is a pain to allocate physically contiguous regions
- Idea: “virtual addresses” for devices
  - We can take random physical pages and make them look contiguous to the device
  - Called “Bus address” for clarity
- New to the x86 (called VT-d)
  - Until very recently, x86 kernels just suffered

A note on memory protection

- If I can write to a network card's control register and tell it where to write the next packet
  - What if I give it an address used for something else?
    - Like another process's address space
  - Nothing stops this
- DMA privilege effectively equals privilege to write to any address in physical memory!

Why does x86 now care about IOMMUs?

- Virtualization! (VT-d)
- Scenario: system with 4 NICs, 4 VMs
- Without IOMMU: Hypervisor must mediate all network traffic
- With IOMMU: Each VM can have a different virtual bus address space
  - Looks like a single NIC; can only issue DMAs for its own memory (not other VM's memory)
  - No Hypervisor mediation needed!

VT-d Limitations

- IOMMU device restrictions are all-or-nothing
  - Can’t share a network card
  - Although some devices may fix this too
- VT-d is only for devices on the PCI-Express bus
  - Usually just graphics and high-end network cards
  - Legacy PCI devices are behind a bridge
    - All-or-nothing for an entire bridge
  - Similarly, no per-disk access control
    - All-or-nothing for disk controller (which multiplexes disks)

Summary

- How to access devices: ports or memory
- Issues with CPU optimizations, timing delays, etc.
- Overview of PCI bus
- Overview of DMA and protection issues
  - IOMMU and use for virtualization