Introduction

CS 416: Operating Systems Design, Spring 2011

Department of Computer Science
Rutgers University

Rutgers Sakai: 01:198:416 Sp11

(https://sakai.rutgers.edu)

http://www.winlab.rutgers.edu/~chandrga/CS416_Spring2011.html

Logistics

Me: Gayathri Chandrasekaran
chandrga@cs.rutgers.edu

Office hour: Tuesdays, 3.30 – 4.30pm
Office: Hill 418

TA: Nishat Islam
nislam@cs.rutgers.edu

Recitation: Thursdays , 12:15pm-1:10pm, SEC 204.

Resources:

Rutgers Sakai: 01:198:416 Sp11 (https://sakai.rutgers.edu)

Course Overview

Goals:

Understanding of OS and the OS/architecture interface/interaction

Prerequisites:

113 & 211

What to expect:

We will cover core concepts and issues in lectures

In Recitations, you and your TA will practice paper & pencil problems and talk about the programming assignments

3 programming assignments

2-3 small paper & pencil assignments

1 Midterm and 1 Final

Warning

Do NOT ignore this warning!

The programming assignments will take a significant amount of time! Don't take this course if you are overloaded or do not have the needed background.

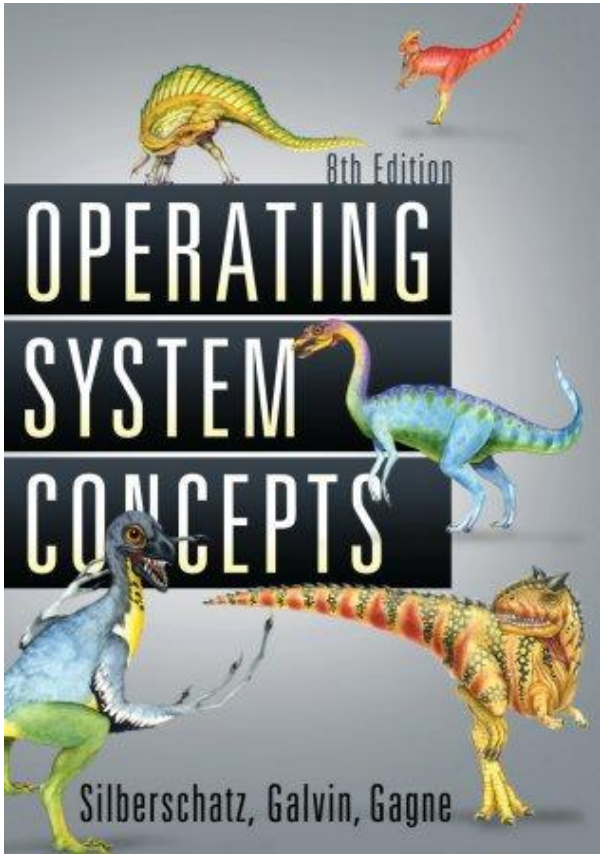
Assignments will help you get hands on and learn a lot.

You will have to work hard!

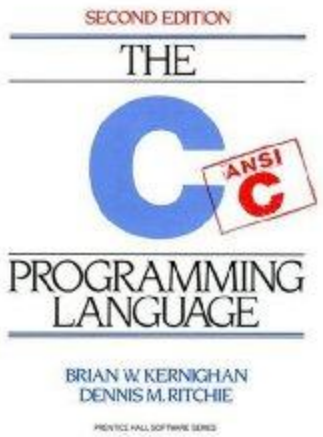
Textbook and Topics

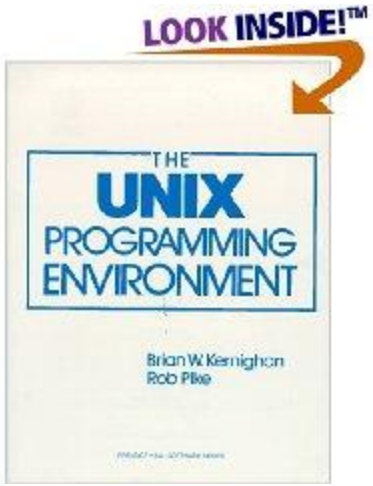
Silberschatz, Galvin, and Gagne. Operating System Concepts. John Wiley & Sons.

- Reading assignments based off of 8th edition
- Any of 6th, 7th, or 8th edition should be ok



Your favorite C book





Topics

Architecture Introduction

Processes and threads

Synchronization

Processor scheduling

Virtual memory

File systems

I/O systems

Security – (If time permits)

Grading

Rough guideline

* Assignment 1 -- 10%
* Assignment 2 -- 10%
* Assignment 3 -- 15%
* Homeworks -- 10%
* Midterm -- 20%
* Final Exam -- 35%

Grading should really be based on how much you have learned from the course

- Any concrete thing that you do towards convincing me of this may help
- For example, showing up at office hours and participating in class will likely help

Paper & Pencil Homeworks

2-3 paper & pencil homeworks

- Each will have some graded and some suggested but not graded problems
- Good practice is to do them all

Goals

- Understand concepts and issues
- Practice for tests

Homeworks are due by the end of the lecture on the due date

- Late hand-ins will not be accepted

Cheating


OMAHA WORLD-HERALD
KOTERBA
NO. YOU MAY NOT OUTSOURCE YOUR HOMEWORK TO ...

Programming Assignments

3 programming assignments (all in C)

- Shell & system call (Linux kernel 2.6)
- Partial threads package & multi-threaded server
- File system (Linux kernel 2.6)

Goals

- Improve design, implementation, and debugging skills
- Learn to read and understand existing code
- Learn the internals of an actual operating system

Programming assignments must be submitted via Sakai

- We will NOT accept any other form (in particular, NO EMAIL)
- Late hand-ins will not be accepted

Project Resources

Linux kernel hacking

stratus.rutgers.edu

(Use your remus credentials)

Non-kernel related assignment

Cereal cluster: http://cereal.rutgers.edu

Programming Tools

Emacs, vi-editor

Compiler: cc (gcc)

Project build: make (gmake)

Debugger: gdb

IDE: eclipse

For the Linux kernels, there are a number of on-line Web-based source code cross-referencing sites … choose your favorite

Try searching for “linux kernel cross reference” or “linux kernel browsing”

Today (and next lecture)

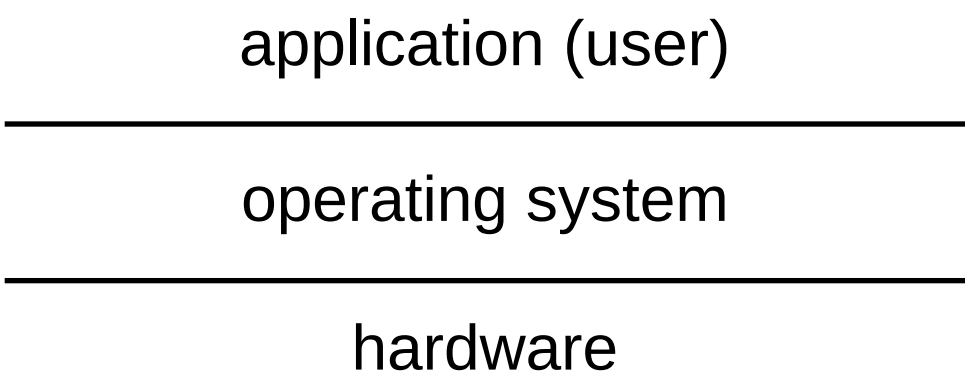
What is an Operating System?

Major OS components

Architectural refresher

SGG Chapter 1: 1.1 - 1.9, 1.13

What Is An Operating System?

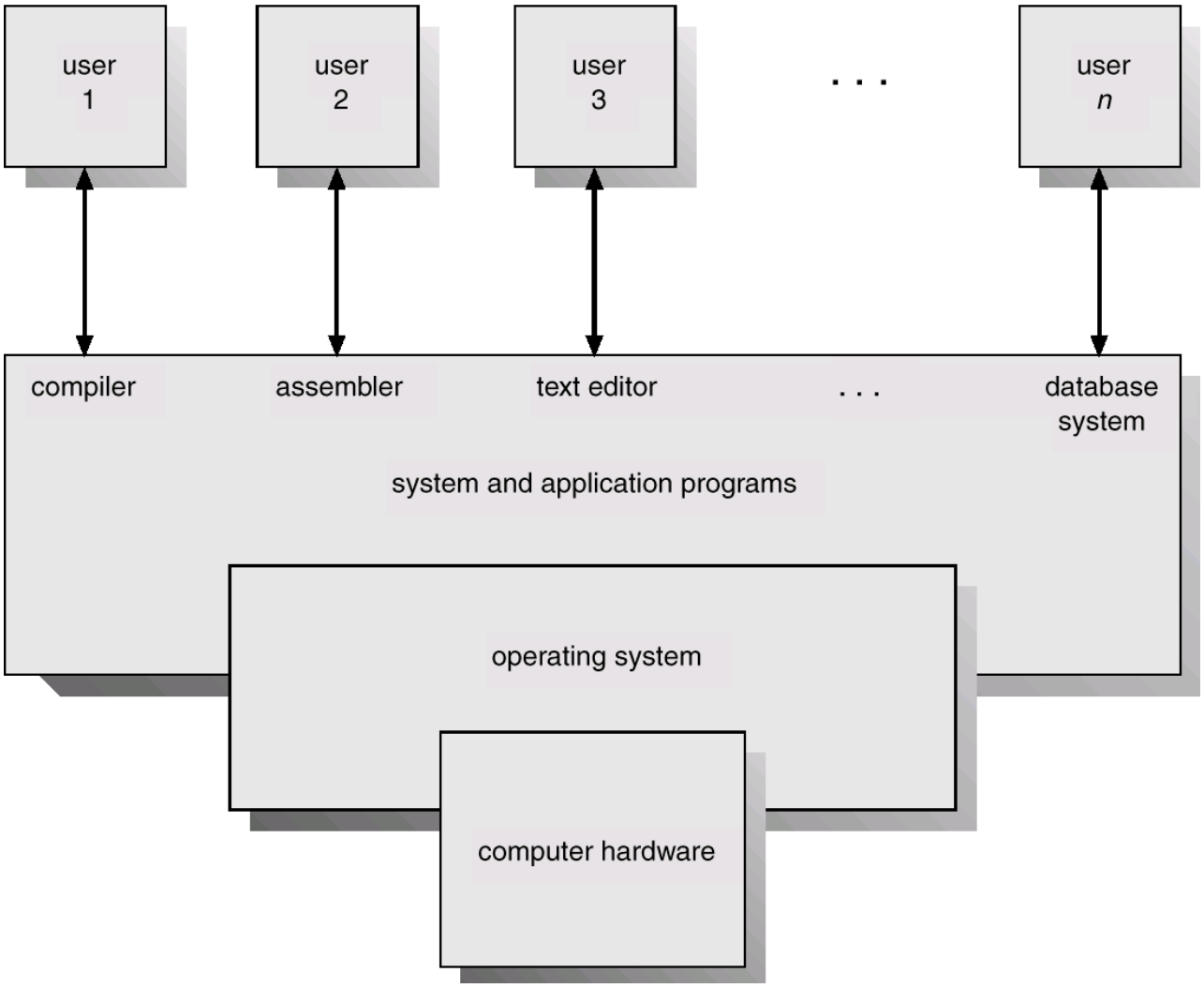


A software layer between the hardware and the application programs/users which provides a *virtual machine* interface: easy and safe

A *resource manager* that allows programs/users to share the hardware resources: fair and efficient

A set of utilities to simplify application development and execution

Abstract View of System Components



Why Do We Want An OS?

Benefits for application writers

- Easier to write programs
  - See high level abstractions instead of low-level hardware details
    - E.g. files instead of disk blocks
- Portability – Works for any underlying hardware

Benefits for users

- Easier to use computers
  - Can you imagine trying to use a computer without the OS?
- Safety
  - OS protects programs from each other
  - OS protects users from each other

Mechanism And Policy

application (user)

operating system: *mechanism+policy*

hardware

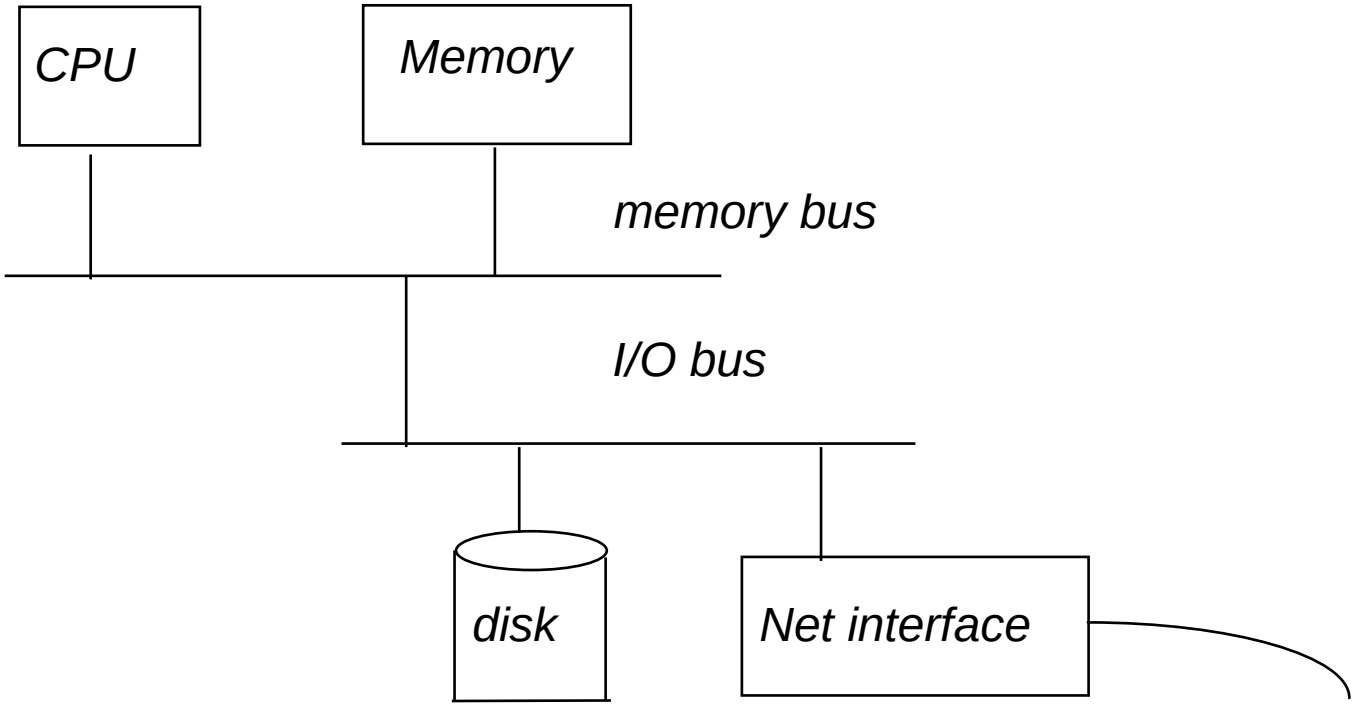
Mechanisms: data structures and operations that implement an abstraction (e.g. the buffer cache)

Policies: the procedures that guides the selection of a certain course of action from among alternatives (e.g. the replacement policy for the buffer cache)

Want to separate mechanisms and policies as much as possible

Different policies may be needed for different operating environments

Basic computer structure



Virtual Machine Abstractions

Processes: system abstraction – illusion of being the only job executing in the system

Threads: CPU abstraction – illusion of having a dedicated CPU

Virtual memory: memory abstraction – illusion of having an unlimited memory

File system: storage abstraction – illusion of structured, persistent storage system

Messaging: communication abstraction – illusion of reliable, ordered communication

Character and block devices: I/O abstraction – standardized I/F for devices

Major Issues In OS Design

Programming API: what should the VM look like?

Resource management: how should resources be shared among multiple users?

Security: how to protect users from each other? How to protect programs from each other? How to project the OS from applications and users?

Communication: how can applications exchange information?

Concurrency: how do we deal with the concurrency that's inherent in OS'es?

Major Issues In OS Design

Performance: how to make it all run fast?

Reliability: how do we keep the OS from crashing?

Persistence: how can we make data last beyond program execution?

Accounting: how do we keep track of resource usage?

Distribution: how do we make it easier to use multiple computers in conjunction?

Scaling: how do we keep the OS efficient and reliable as the imposed load and so the number of computers grow?

Architectural Refresher

View of a computer from an Operating System's designer perspective

Operating system is a layer of software that creates a virtual machine

- OS also manages the resources of this machine but this mostly involves sharing policies so will be discussed later

These lectures will familiarize you with

- The underlying machine
- The extra hardware mechanisms needed for virtualization

Topics

The von Neumann architecture

CPU + memory

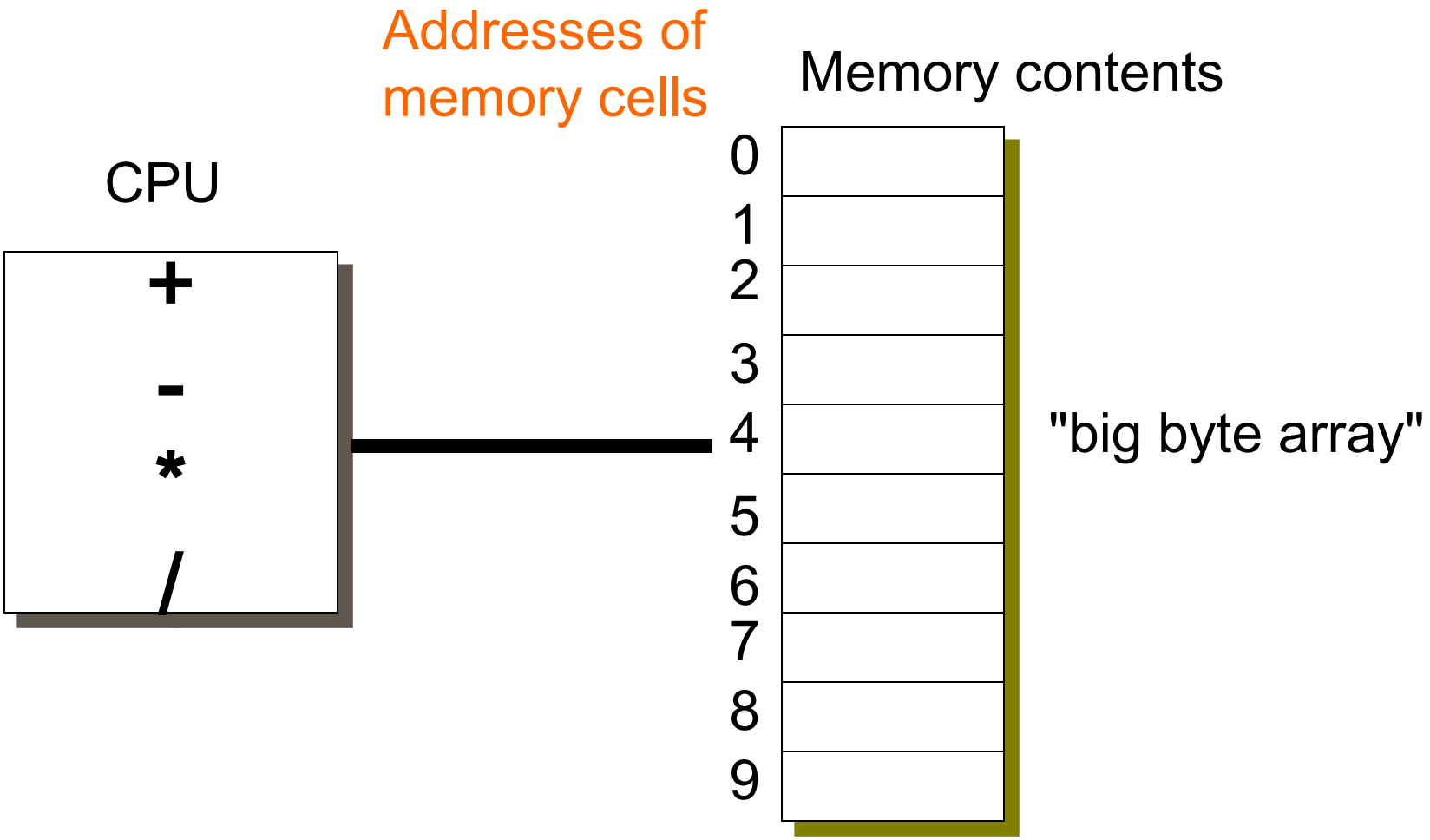
Hardware support for abstracting the basic machine

Modes, Exceptions, Traps and Interrupts

Input and Output

Network, storage and graphics

Conceptual Model



Operating System Perspective

A computer is a piece of hardware which runs the fetch-decode-execute loop

Next slides: walk through a very simple computer to illustrate

- Machine organization
  - What are the pieces and how they fit together
- The basic fetch-decode-execute loop
- How higher-level constructs are translated into machine instructions

At its core, the OS builds what looks like a more complex machine on top of this basic hardware

Fetch-Decode-Execute

Computer as a large, general purpose calculator

- want to program it for multiple functions

All von Neumann computers follow the same loop:

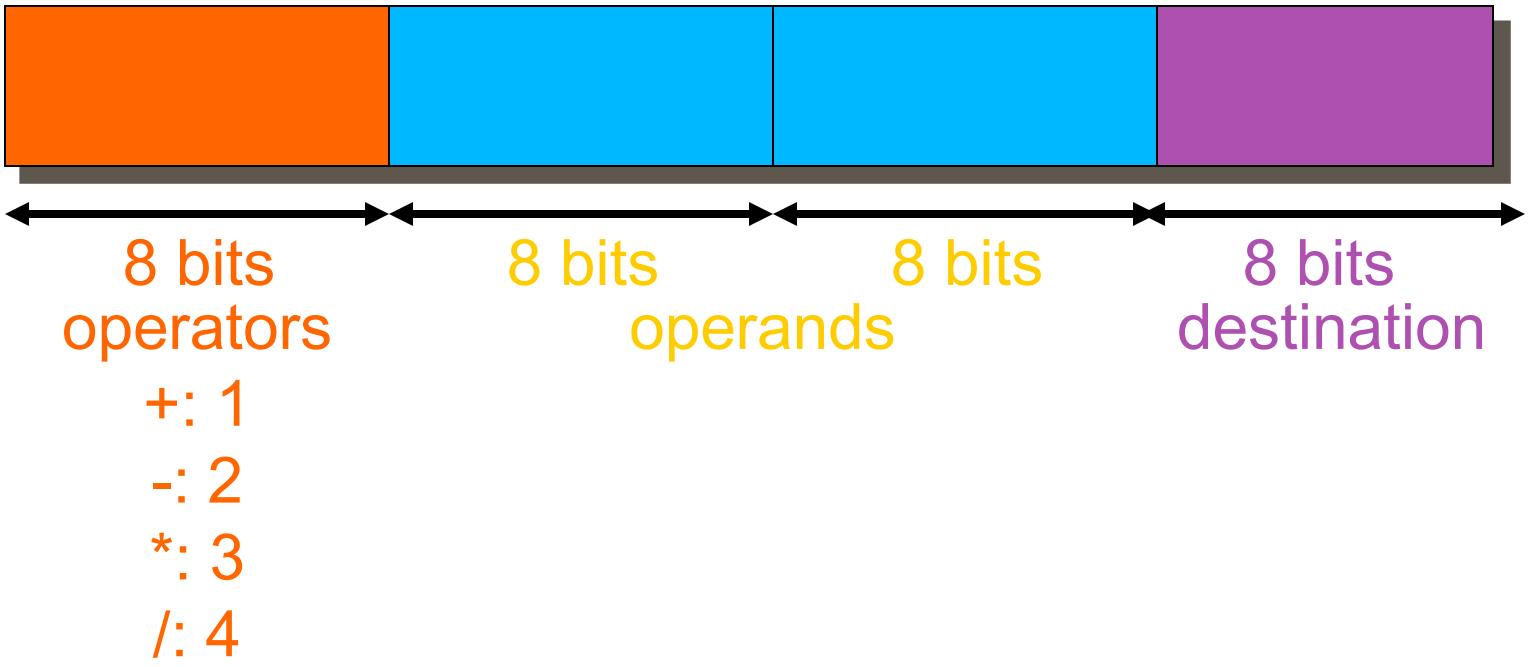
- Fetch the next instruction from memory
- Decode the instruction to figure out what to do
- Execute the instruction and store the result

Instructions are simple. Examples:

- Increment the value of a memory cell by 1
- Add the contents of memory cells X and Y and store in Z
- Multiply contents of memory cells A and B and store in B

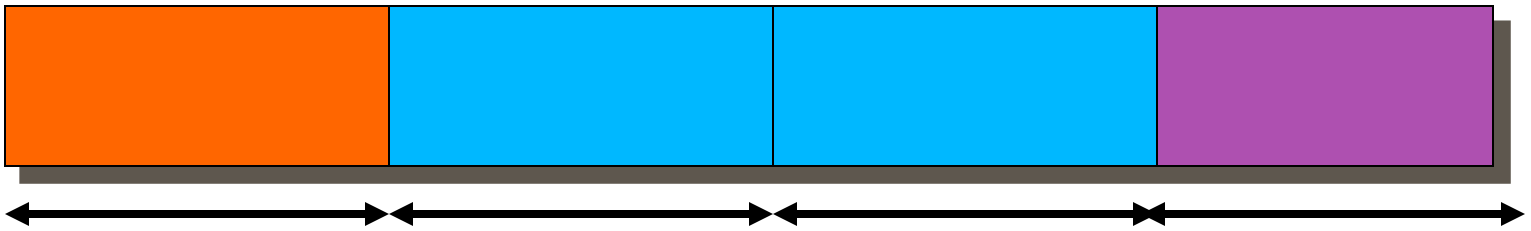
Instruction Encoding

How to represent instructions as numbers?



Example Encoding

Add cell 28 to cell 63 and place result in cell 100:



8 bits operator

+: 1

-: 2

*: 3

/: 4

8 bits source operands

Cell 28

8 bits source operands

Cell 63

8 bits destination

Cell 100

Instruction as a number in:

Decimal: 1:28:63:100

Binary: 00000001:00011100:00111111:01100100

Hexadecimal: 01:1C:3F:64

Example Encoding (cont)

How many instructions can this encoding have?

8 bits, 2^8 combinations = 256 instructions

How much memory can this example instruction set support?

Assume each memory cell is a byte (8 bits) wide

Assume operands and destination come from the same memory

8 bits per source/dest = 2^8 combinations = 256 bytes

How many bytes did we use per instruction?

4 bytes per instruction

How could we get more memory without changing the encoding?

Why is this simple encoding not realistic?

The Program Counter

Where is the “next instruction” held in the machine?

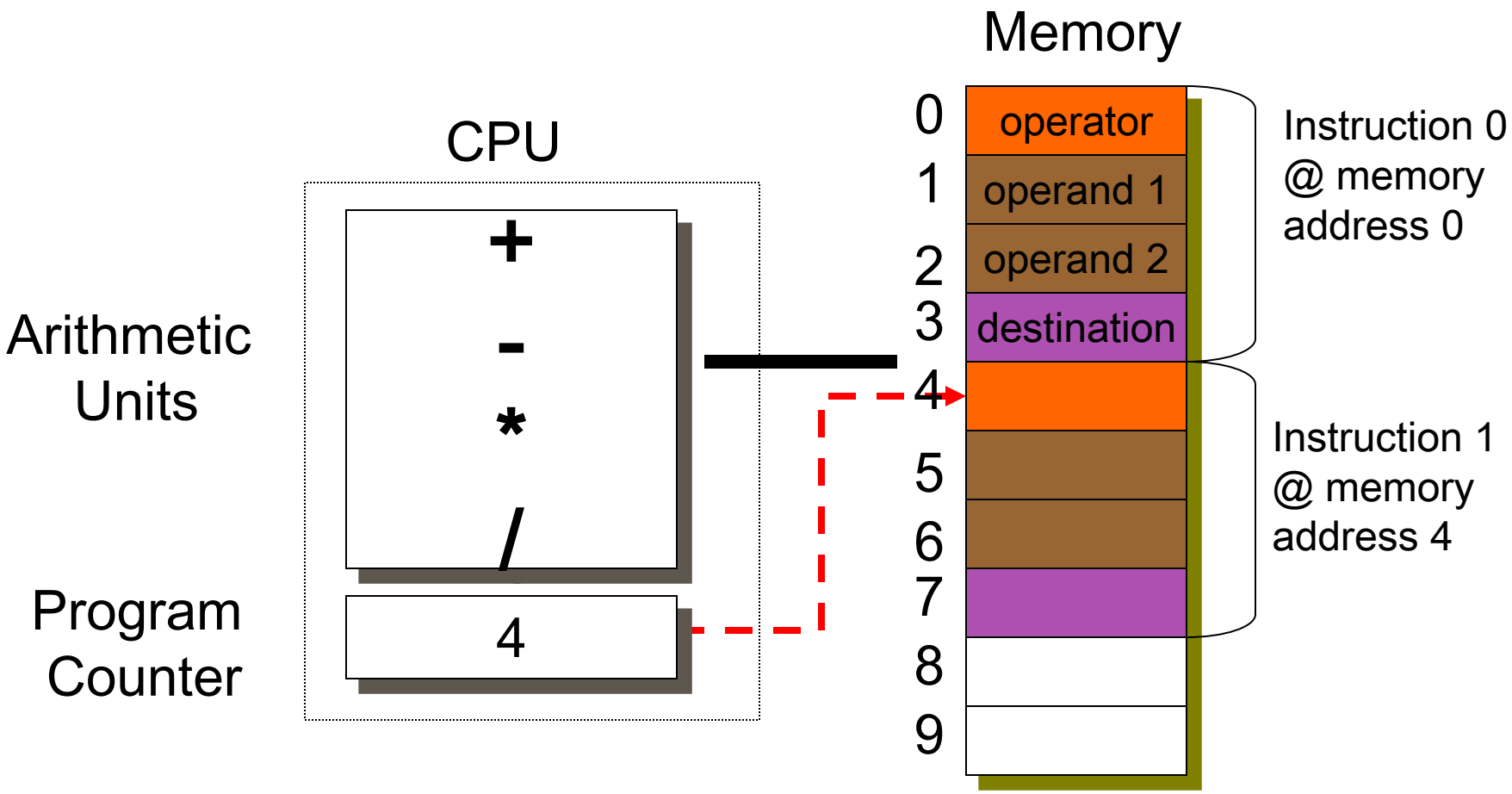
In a special memory cell in the CPU called the “program counter" (the PC)

- Special purpose memory in the CPU and devices are called registers

Naive fetch cycle: Increment the PC by the instruction length (4) after each execute

- Assumes all instructions are the same length

Conceptual Model



Memory Indirection

How do we access array elements efficiently if all we can do is name a cell?

Modify the operand to allow for fetching an operand "through" a memory location

> E.g.: LOAD [5], 2 means fetch the contents of the cell whose address is in cell 5 and put it into cell 2
>
> So if cell 5 had the number 100, we would place the contents of cell 100 into cell 2

This is called indirection

> Fetch the contents of the cell “pointed to” by the cell in the opcode

Steal an operand bit to signify if an indirection is desired

Conditionals and Looping

Instructions that modify the Program Counter

Branch Instructions

If the content of this cell is [zero, non zero, etc.], set the PC to this location

jump is an unconditional branch

Example: While Loop

```
while (counter > 0) {
    sum = sum + Y[counter];
    counter--;
};
```

Variables to memory cells:

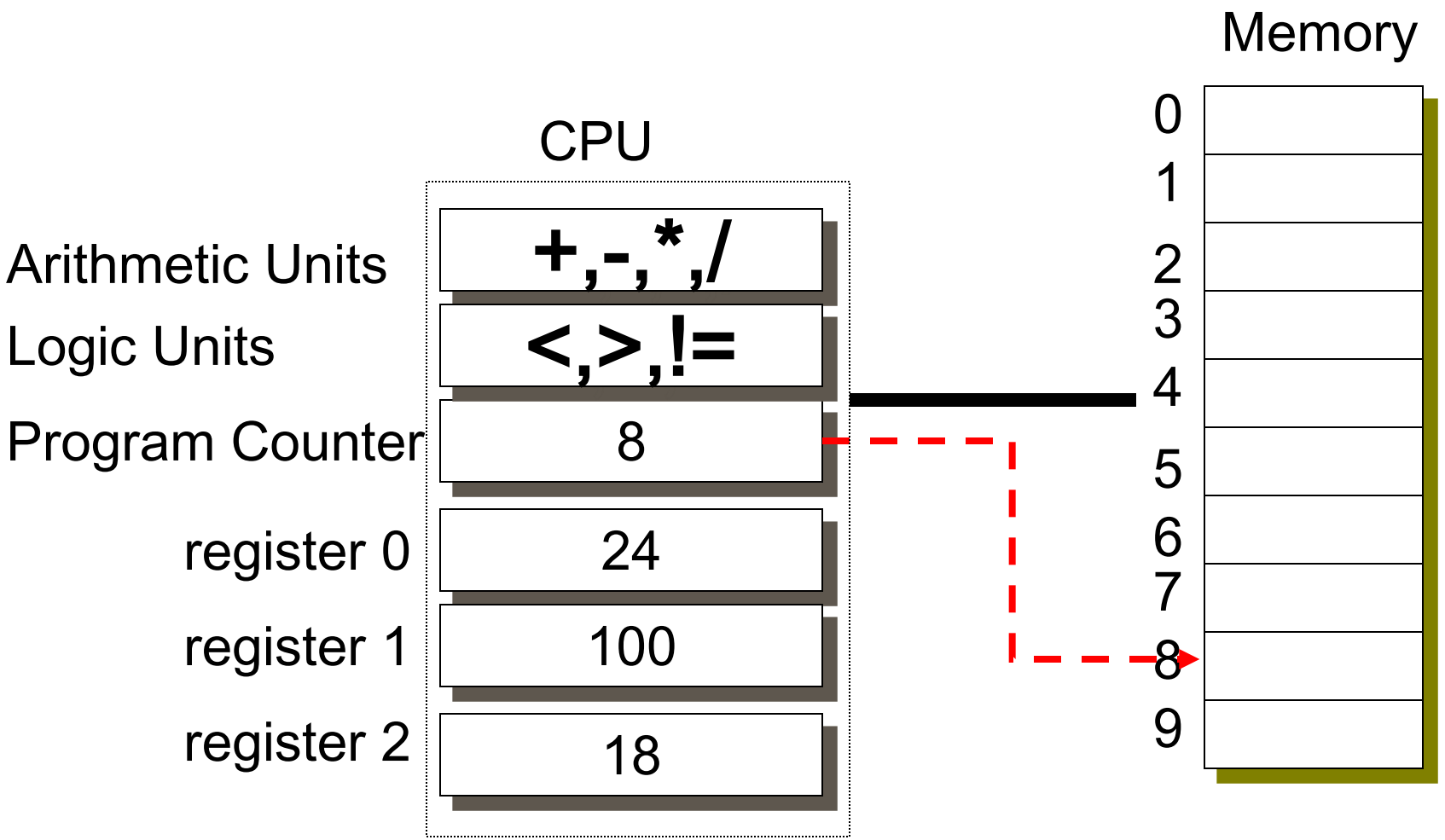
```
counter is cell 1
sum is cell 2
index is cell 3
Y[0]= cell 4, Y[1]=cell 5…
```

Memory cell address	Assembler label	Assembler "mnemonic"	English
100	LOOP:	BNZ 1,END	// branch to address of END // if cell 1 is not 0.

104		ADD 2,[3],2	// Add cell 2 and the value
			// of the cell pointed to by
			// cell 3 then place the
			// result in cell 2

108		DEC 3	// decrement cell 3 by 1
112		DEC 1	// decrement cell 1 by 1
116		JUMP LOOP	// start executing from the // address of LOOP
120	END:	<next code block>	

Register Machine Model



Registers (cont)

Most CPUs have 16-32 “general purpose” registers

- All look the “same”: combination of operators, operands and destinations possible

Operands and destination can be in:

- Registers only (Sparc, PowerPC, Mips, Alpha)
- Registers & 1 memory operand (x86 and clones)
- Any combination of registers and memory (Vax)

Only memory operations possible in "register-only" machines are load from and store to memory

Operations 100-1000 times faster when operands are in registers compared to when they are in memory

Save instruction space too

- Only address 16-32 registers, not GB of memory

Typical Instructions

Add the contents of register 2 and register 3 and place result in register 5

ADD r2,r3,r5

Add 100 to the PC if register 2 is not zero

Relative branch

BNZ r2,100

Load the contents of memory location whose address is in register 5 into register 6

LDI r5,r6

Abstracting the Machine

Bare hardware provides a computation device

How to share this expensive piece of equipment between multiple users?*

Sign up during certain hours? Give program to an operator?

Software to give the illusion of having it all to yourself while actually sharing it with others (time-sharing)!

This software is the Operating System

Need hardware support to "virtualize" machine

* Software that makes it easy for 1 expensive user to use multiple devices!!

Architecture Features for the OS

Next we'll look at the mechanisms the hardware designers add to allow OS designers to abstract the basic machine in software

- Processor modes
- Exceptions
- Traps
- Interrupts

These require modifications to the basic fetch-decode-execute cycle in hardware

Processor Modes

OS code is stored in memory … von Neumann model, remember?

- What if a user program modifies OS code or data?

Introduce modes of operation

- Instructions can be executed in user mode or system mode

A special register holds which mode the CPU is in

Certain instructions can only be executed when in system mode

Likewise, certain memory location can only be written when in system mode

- Only OS code is executed in system mode
  - Only OS can modify its memory
- The mode register can only be modified in system mode

Simple Protection Scheme

All addresses < 100 are reserved for operating system use

Mode register provided

zero = CPU is executing the OS (in system mode)

one = CPU is executing in user mode

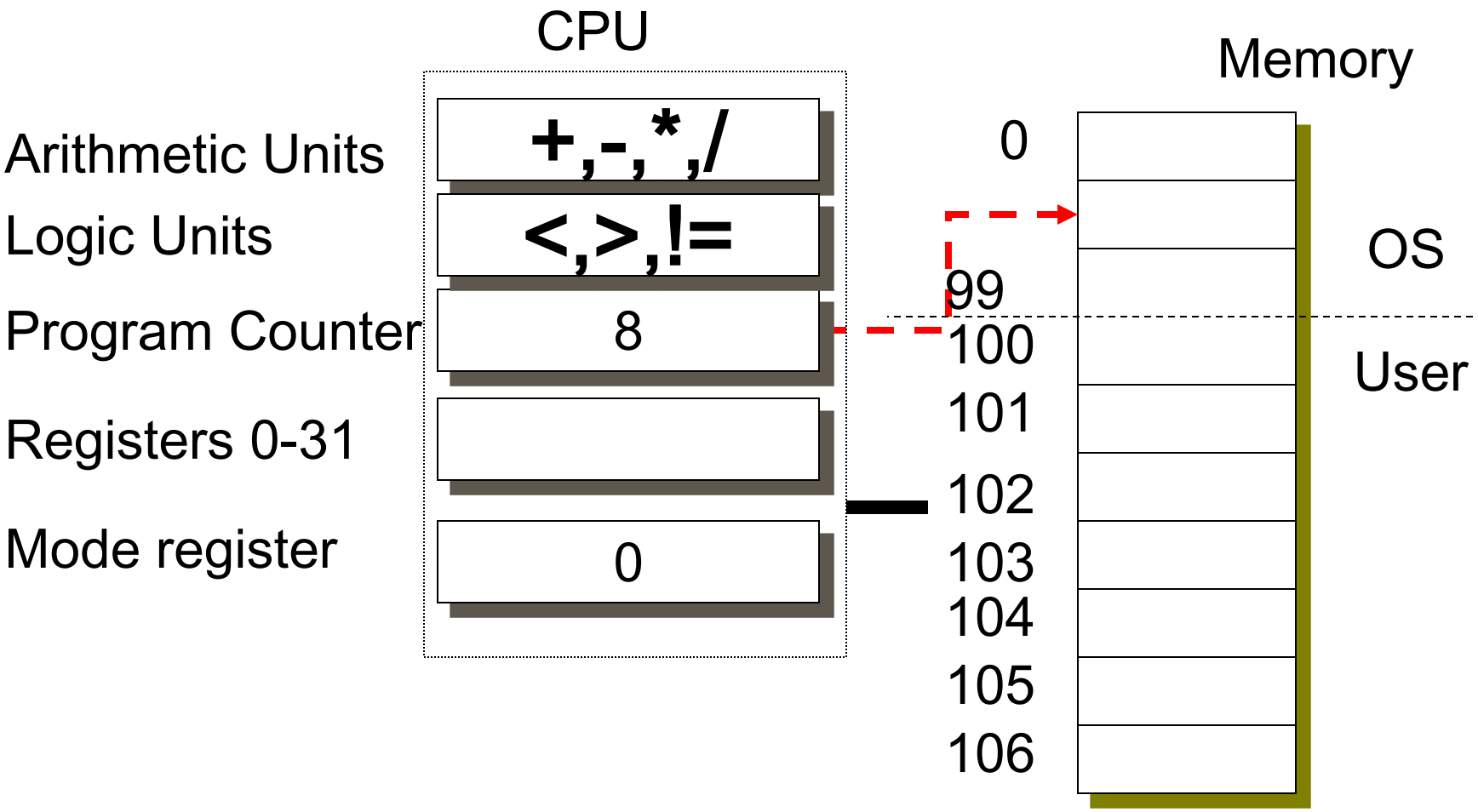
Hardware does this check:

On every fetch, if the mode bit is 1 and the address is less than 100, then do not execute the instruction

When accessing operands, if the mode bit is 1 and the operand address is less than 100, do not execute the instruction

Mode register can only be set if mode is 0

Simple Protection Model



Fetch-decode-execute Revised

```
Fetch:
        if (( the PC < 100) &&  ( the mode register == 1)) then
                Error! User tried to access the OS
        else
                fetch the instruction at the PC
Decode:
        if (( destination register == mode) && ( the mode register == 1)) then
                Error! User tried to set the mode register
        < more decoding >
Execute:
        if (( an operand < 100) && ( the mode register == 1) then
                error! User tried to access the OS
        else
                execute the instruction
```

Exceptions

What happens when a user program tries to access memory holding the operating system code or data?

Answer: exceptions

An exception occurs when the CPU encounters an instruction which cannot be executed

Modify fetch-decode-execute loop to jump to a known location in the OS when an exception happens

Different errors jump to different places in the OS (are "vectored" in OS speak)

Fetch-decode-execute with Exceptions

Fetch:

if ((the PC < 100) && (the mode bit == 1)) then

set the PC = 60

set the mode = 0

fetch the instruction at the PC

60 is the well known entry point for a memory violation

Decode:

if ((destination register == mode) && (the mode register == 1)) then

set the PC = 64

set the mode = 0

goto fetch

< more decoding >

64 is the well known entry point for a mode register violation

Execute:

< check the operands for a violation>

Access Violations

Notice both instruction fetch from memory and data access must be checked

Execute phase must check both operands

Execute phase must check again when performing an indirect load

This is a very primitive memory protection scheme. We'll cover more complex *virtual memory* mechanisms and policies later in the course

Recovering from Exceptions

The OS can figure out what caused the exception from the entry point

But how can it figure out where in the user program the problem was?

Solution: add another register, the PC’

When an exception occurs, save the current PC to PC’ before loading the PC with a new value

OS can examine the PC' and perform some recovery action

Stop user program and print an error message: error at address PC'

Run a debugger

Fetch-decode-execute with Exceptions & Recovery

Fetch:

if ((the PC < 100) && (the mode bit == 1)) then

set the PC' = PC

set the PC = 60

set the mode = 0

Decode:

if ((destination register == mode) && (the mode register == 1)) then

set the PC' = PC

set the PC = 64

set the mode = 0

goto fetch

< more decoding >

Execute:

…

Traps

Now we know what happens when a user program illegally tries to access OS code or data

How does a user program legitimately access OS services?

Solution: Trap instruction

A trap is a special instruction that forces the PC to a known address and sets the mode into system mode

Unlike exceptions, traps carry some arguments to the OS

Foundation of the system call

Fetch-decode-execute with traps

```
Fetch:
        if (( the PC < 100) && ( the mode bit == 1)) then
                < memory exception>
Decode:
        if (the instruction is a trap) then
                set the PC' = PC
                set the PC = 68
                set the mode = 0
                goto fetch
        if (( destination register == mode) && ( the mode bit == 1)) then
                 < mode exeception >
Execute:
        …
```

Traps

How does the OS know which service the user program wants to invoke on a trap?

User program passes the OS a number that encodes which OS service is desired

This example machine could include the trap ID in the instruction itself:

Most real CPUs have a convention for passing the trap code in a set of registers

E.g. the user program sets register 0 with the trap code, then executes the trap instruction

Returning from a Trap

How to "get back" to user mode and the user's code after a trap?

- Set the mode register = 0 then set the PC?
  - But after the mode bit is set to user, exception!
- Set the PC, then set the mode bit?
  - Jump to "user-land", then in kernel mode

Most machines have a "return from exception" instruction

- A single hardware instruction:
  - Swaps the PC and the PC'
  - Sets the mode bit to user mode
- Traps and exceptions use the same mechanism (RTE)

Interrupts

How can we force a the CPU back into system mode if the user program is off computing something?

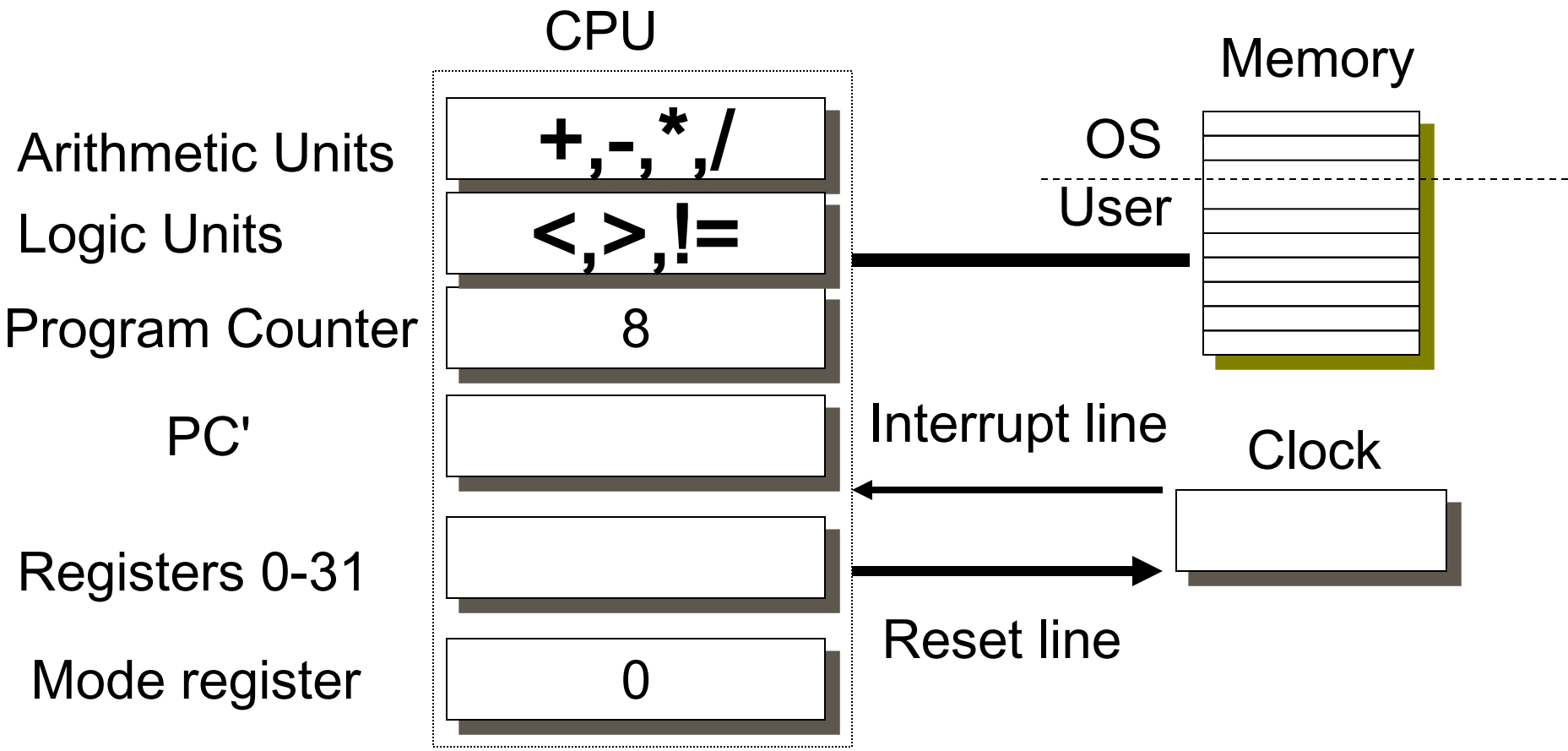
Solution: Interrupts

An interrupt is an external event that causes the CPU to jump to a known address

Link an interrupt to a periodic clock

Modify fetch-decode-execute loop to check an external line set periodically by the clock

Simple Interrupt Model



The Clock

The clock starts counting to 10 milliseconds

The clock sets the interrupt line "high" (e.g. sets it logic 1, maybe +5 volts)

When the CPU toggles the reset line, the clock sets the interrupt line low and starts count to 10 milliseconds again

Fetch-decode-execute with Interrupts

```
Fetch:
        if (the clock interrupt line == 1) then
                set the PC' = PC
                set the PC = 72
                set the mode = 0
                goto fetch
        if (( the PC < 100) && ( the mode bit == 1)) then
                < memory exception >
        fetch next instruction
Decode:
        if (the instruction is a trap) then
                < trap exception >
        if (( destination register == mode) && ( the mode bit == 1)) then
                 < mode exeception >
        <more decoding>
Execute:  …
```

Entry Points

What are the "entry points" for our little example machine?

60: memory access violation

64: mode register violation

68: User-initiated trap

72: Clock interrupt

Each entry point is typically a jump to some code block in the OS

All real OS'es have a set of entry points for exceptions, traps and interrupts

Sometimes they are combined and software has to figure out what happened.

Saving and Restoring Context

Recall the processor state:

PC, PC', R0-R31, mode register

When an entry to the OS happens, we want to start executing the correct routine then return to the user program such that it can continue executing normally

Can't just start using the registers in the OS!

Solution: save/restore the user context

Use the OS memory to save all the CPU state

Before returning to user, reload all the registers and then execute a return from exception instruction

Input and Output

How can humans get at the data?

How to load programs?

What happens if I turn the machine off?

Can I send the data to another machine?

Solution: add devices to perform these tasks:

- Keyboards, mice, graphics
- Disk drives
- Network cards

A Simple I/O device

Network card has 2 registers:

a store into the "transmit" register sends the byte over the wire.

Transmit often is written as TX (E.g. TX register)

a load from the "receive" register reads the last byte which was read from the wire

Receive is often written as RX

How does the CPU access these registers?

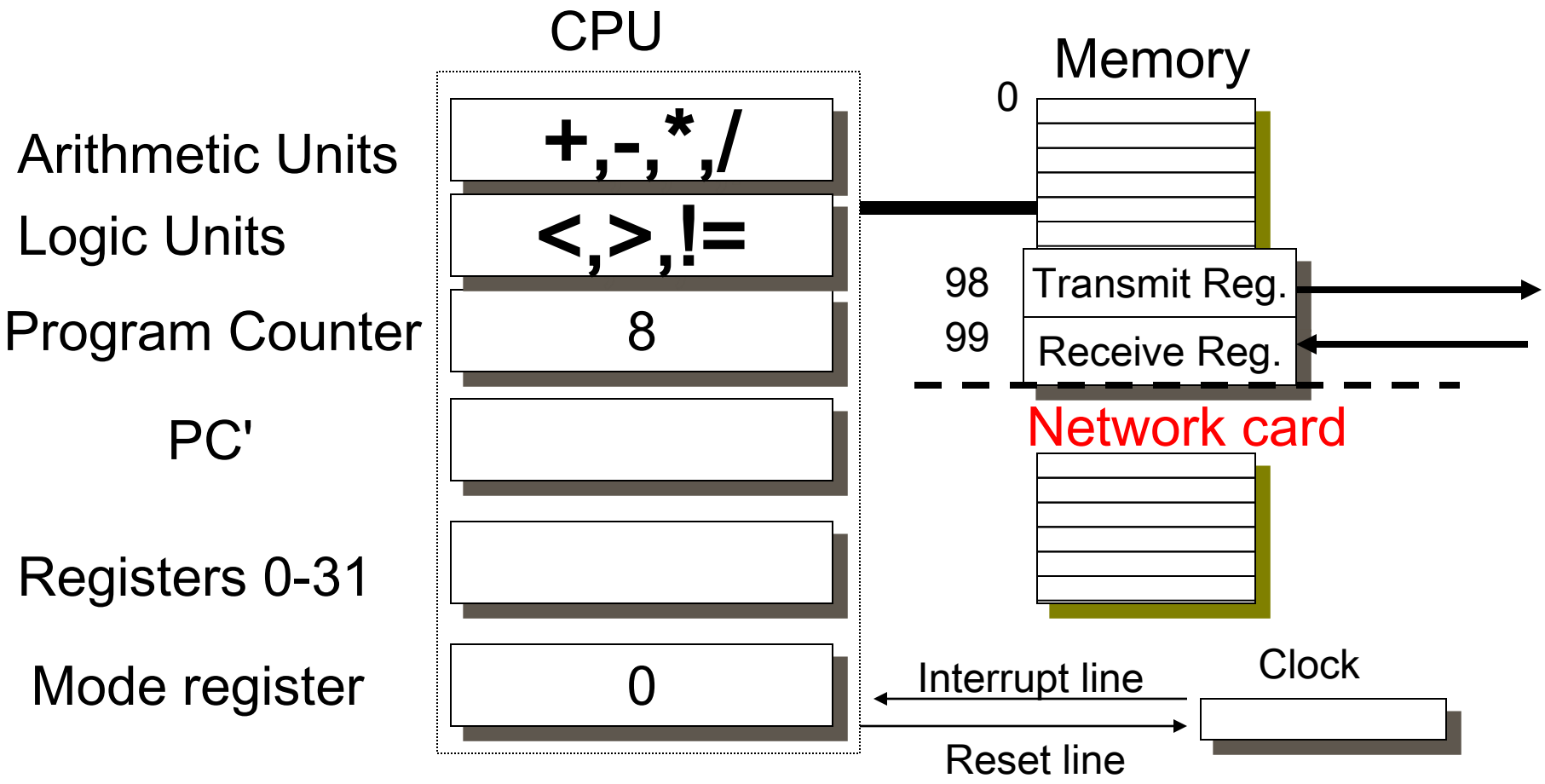
Solution: map them into the memory space

An instruction that access memory cell 98 really accesses the transmit register instead of memory

An instruction that accesses memory cell 99 really accesses the receive register

These registers are said to be memory-mapped

Basic Network I/O



Why Memory-Mapped Registers

"Stealing" memory space for device registers has 2 functions:

- Allows protected access --- only the OS can access the device.
  - User programs must trap into the OS to access I/O devices because of the normal protection mechanisms in the processor
  - Why do we want to prevent direct access to devices by user programs?
- OS can control devices and move data to/from devices using regular load and store instructions
  - No changes to the instruction set are required
  - This is called programmed I/O

Status Registers

How does the OS know if a new byte has arrived?

How does the OS know when the last byte has been transmitted? (so it can send another one)

Solution: status registers

A status register holds the state of the last I/O operation

Our network card has 1 status register

> To transmit, the OS writes a byte into the TX register and sets bit 0 of the status register to 1. When the card has successfully transmitted the byte, it sets bit 0 of the status register back to 0.
>
> When the card receives a byte, it puts the byte in the RX register and sets bit 1 of the status register to 1. After the OS reads this data, it sets bit 1 of the status register back to 0.

Polled I/O

To Transmit:

```
While (status register bit 0 == 1);        // wait for card to be ready
TX register = data;
Status reg = status reg | 0x1;             // tell card to TX (set bit 0 to 1)
```

Naïve Receive:

```
While (status register bit 1 != 1);        // wait for data to arrive
Data = RX register;
Status reg = status reg & 0x01;            // tell card got data (clear bit 1)
```

Cant' stall OS waiting to receive!

Solution: poll after the clock ticks

```
If (status register bit 1 == 1)
    Data = RX register
Status reg = status reg & 0x01;
```

Interrupt driven I/O

Polling can waste many CPU cycles

On transmit, CPU slows to the speed of the device

Can't block on receive, so tie polling to clock, but wasted work if no RX data

Solution: use interrupts

When network has data to receive, signal an interrupt

When data is done transmitting, signal an interrupt.

Polling vs. Interrupts

Why poll at all?

Interrupts have high overhead:

- Stop processor
- Figure out what caused interrupt
- Save user state
- Process request

Key factor is frequency of I/O vs. interrupt overhead

Direct Memory Access(DMA)

Problem with programmed I/O: CPU must load/store all the data into device registers.

The data is probably in memory anyway!

Solution: more hardware to allow the device to read and write memory just like the CPU

- Base + bound or base + count registers in the device
- Set base + count register
- Set the start transmit register
- I/O device reads memory from base
- Interrupts when done

PIO vs. DMA

Overhead less for PIO than DMA

- PIO is a check against the status register, then send or receive
- DMA must set up the base, count, check status, take an interrupt

DMA is more efficient at moving data

- PIO ties up the CPU for the entire length of the transfer

Size of the transfer becomes the key factor in when to use PIO vs. DMA

Example of PIO vs. DMA

Given:

A load costs 100 CPU "cycles“ (time units)

A store costs 50 cycles

To process an interrupt costs 2000 instructions, each an average each of 2 cycles

To send a packet via PIO costs 1 load + 1 store per byte

To send via DMA costs 4 loads + interrupt

Find the packet size where transmitting via DMA costs less CPU cycles than PIO

Example PIO vs. DMA

Find the number of bytes were PIO==DMA (cutoff point)

cycles per load: L

cycles per store: S

byte in the packet: C

Express simple equation for CPU cycles in terms of cost per byte:

# of cycles for PIO = L + S*B

# of cycles for DMA = setup + interrupt

# of cycles for DMA = 4L + 4000

Set PIO cycles equal to DMA cycles and solve for bytes:

L+S*B = 4L+4000

100+50B = 4(100)+4000

B = 86 bytes (cutoff point)

When the packet size is >86 bytes, DMA costs less cycles than PIO.

Typical I/O devices

Disk drives:

Present the CPU a linear array of fixed-sized blocks that are persistent across power cycles

Network cards:

Allow the CPU to send and receive discrete units of data (packets) across a wire, fiber or radio

Packet sizes 64-8000 bytes are typical

Graphics adapters:

Present the CPU with a memory that is turned into pixels on a screen

Recap: the I/O design space

Polling vs. interrupts

How does the device notify the processor an event happened?

- Polling: Device is passive, CPU must read/write a register
- Interrupt: device signals CPU via an interrupt

Programmed I/O vs. DMA

How the device sends and receives data

- Programmed I/O: CPU must use load/store into the device
- DMA: Device reads and writes memory

Practical: How to boot?

How does a machine start running the operating system in the first place?

- The process of starting the OS is called booting

Sequence of hardware + software event form the boot protocol

Boot protocol in modern machines is a 3-stage process

- CPU starts executing from a fixed address
- Firmware loads the boot loader
- Boot loader loads the OS

Boot Protocol

(1) CPU is hard-wired to start executing from a known address in memory

E.g., on x86 this address is 0xFFFF0 (hexadecimal)

This memory address is typically mapped to read-only memory (ROM)

(2) ROM contains the “boot” code

This kind of read-only software is called firmware

On x86, the starting address corresponds to the BIOS (basic input-output system) boot entry point

This "firmware" code contains only enough enough code to read 1 block from the disk drive. This block is loaded and then executed. This program is the boot loader.

Boot Protocol (cont)

(3) The boot loader can then load the rest of the operating system from disk. Note that this point the OS still is not running

- The boot loader can know about multiple operating systems
- The boot loader can know about multiple versions of the OS

Why Have A Boot Protocol?

Why not just store the OS into ROM?

Separate the OS from the hardware

- Multiple OSes or different versions of the OS

Want to boot from different devices

- E.g. security via a network boot

OS is pretty big (4-8MB). Rather not have it as firmware

Next

Processes and threads