Essentials Of
Computer Organization

Douglas Comer

Computer Science Department
Purdue University

http://www.cs.purdue.edu/people/comer

I

Course Introduction
And Overview

Questions To Consider

Questions To Consider

- Most CS programs require an architecture course

Questions To Consider

- Most CS programs require an architecture course

- But is architecture useful?

Questions To Consider

- Most CS programs require an architecture course

- But is architecture useful?

   Is a knowledge of computer organization and the underlying hardware relevant for someone who intends to write software ?

- In other words:

   Why should you take this course seriously?

The Answers

The Answers

- The best programmers understand how programs affect the underlying hardware

    – Instructions

    – Memory accesses

    – I/O and timing

The Answers

- The best programmers understand how programs affect the underlying hardware

    – Instructions

    – Memory accesses

    – I/O and timing

- Knowledge of computer architecture is needed for later courses — it will help you understand topics like compilers and operating systems

The Answers

- The best programmers understand how programs affect the underlying hardware

  – Instructions

  – Memory accesses

  – I/O and timing

- Knowledge of computer architecture is needed for later courses — it will help you understand topics like compilers and operating systems

- Knowing about architecture can help you land and retain a good job

How This Course Helps

How This Course Helps

- Allows a programmer to write computer programs that are:

    - Faster

    - Smaller

    - Less prone to errors

How This Course Helps

- Allows a programmer to write computer programs that are:

    – Faster

    – Smaller

    – Less prone to errors

- Is key for programming embedded systems

    – Cell phones

    – Video games

    – MP3 players

    – Set-top boxes

The Bad News

The Bad News

- Hardware is ugly

    - Many details

    - Can be counter-intuitive

The Bad News

- Hardware is ugly

    – Many details

    – Can be counter-intuitive

- The subject is extensive

    – Cannot be mastered in one course

    – Requires background in electricity and electronics

    – Complexity is non-linear — small increases in size can produce large increases in hardware complexity

The Good News

The Good News

- It is possible to understand architectural components without knowing all low-level technical details

The Good News

- It is possible to understand architectural components without knowing all low-level technical details

- Programmers only need to know the essentials

    - Characteristics of major components

    - Role in overall system

    - Consequences for software

Goals Of The Course

- Explore basics of digital hardware

- Build simple circuits

- Learn about functional units in a computer

- Understand

    – How processors work

    – How memories are organized

    – How I/O operates

- Become better programmers

Organization Of The Course

Organization Of The Course

- Basics

    – A taste of digital logic

    – Data representations

Organization Of The Course

- Basics

  - A taste of digital logic

  - Data representations

- Processors

  - Types of processors

  - Instruction sets and operands

  - Assembly languages and programming

Organization Of The Course
(continued)

- Memory

  - Storage mechanisms

  - Physical and virtual memories and addressing

  - Caching

Organization Of The Course
(continued)

- Memory

    - Storage mechanisms

    - Physical and virtual memories and addressing

    - Caching

- Input/Output

    - Devices and interfaces

    - Buses and bus address spaces

    - Role of device drivers

Organization Of The Course
(continued)

- Advanced topics

  – Parallelism and parallel computers

  – Pipelining

  – Performance and performance assessment

  – Architectural hierarchy

What We Will Not Cover

- The course emphasizes breadth over depth

- Omissions

  - Most low-level details (e.g., discussion of electrical properties of resistance, voltage, current)

  - Quantitative analysis that engineers use to design hardware circuits

  - Design rules that specify how logic gates may be interconnected

  - VLSI chip design and tools (e.g., Verilog)

Terminology

- Three basic aspects of computer hardware

    – Architecture

    – Design

    – Implementation

Computer Architecture

- Refers to overall organization of computer system

- Analogous to a blueprint

- Specifies:

    - Functionality of major components

    - Interconnections among components

- Abstracts away details

Design

- Needed before a computer can be built

- Translates architecture into components

- Fills in details that the architectural specification omits

- Specifies items such as:

  - How components are grouped onto boards

  - How power is distributed to boards

- Note: many designs can satisfy a given architecture

Implementation

- All details necessary to build a system

- Includes the following:

    - Mechanical specifications of chassis and cases

    - Layout of components on boards

    - Power supplies and power distribution

    - Signal wiring and connectors

    - Part numbers to be used

Summary

- Understanding computer hardware is needed for programming excellence

- Course covers essentials of computer architecture

  - Digital logic

  - Processors, memory, I/O

  - Advanced topics such as parallelism and pipelining

- We will omit details and focus on concepts

- Labs form a key part of the course

Questions?

II

Fundamentals
Of
Digital Logic

Our Goals

- Understand

  - Fundamentals and basics

  - Concepts

  - How computers work at the lowest level

- Avoid whenever possible

  - Device physics

  - Engineering design rules

  - Implementation details

Electrical Terminology

- Voltage

    - Quantifiable property of electricity

    - Measure of potential force

    - Unit of measure: *volt*

- Current

    - Quantifiable property of electricity

    - Measure of electron flow along a path

    - Unit of measure: *ampere (amp)*

Analog For Electricity

- Voltage is analogous to water pressure

- Current is analogous to flowing water

- Can have

    - High pressure with little flow

    - Large flow with little pressure

Voltage

- Device used to measure called *voltmeter*

- Can only be measured as difference between two points

- To measure voltage

  – Assume one point represents zero volts (known as *ground*)

 – Express voltage of second point wrt ground

In Practice

- Typical digital circuit operates on five volts

- Two wires connect each chip to *power supply*

  - Ground (zero volts)

  - Power (five volts)

- Digital logic diagrams do not usually show power and ground connections

Transistor

- Basic building block of digital circuits

- Operates on electrical current

- Acts like a miniature switch — small input current controls flow of large current

- Three external connections

  – Emitter

  – Base (control)

  – Collector

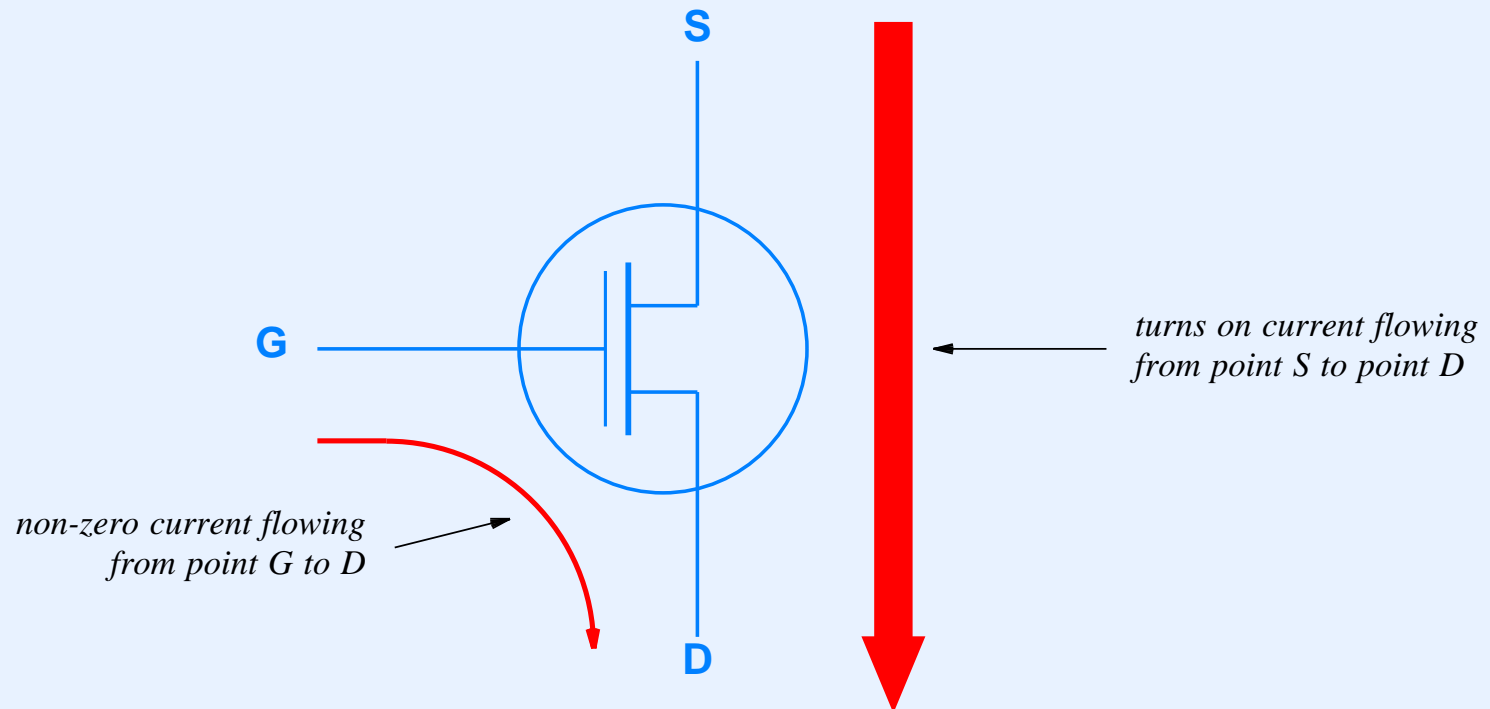- Current between base and emitter controls current between collector and emitter

Illustration Of A Transistor

large current flows
from point C to point E

small current flows
from point B to E
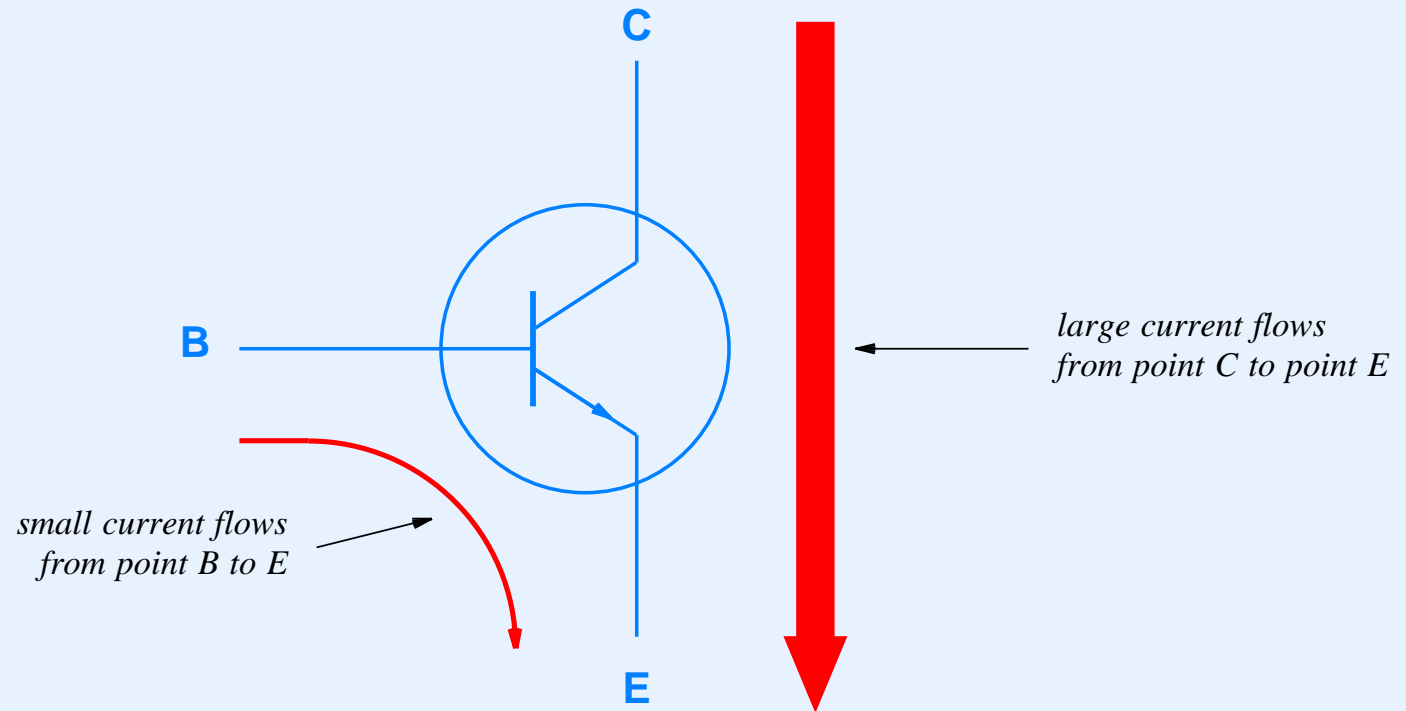
- Important concept: large current is proportional to small current (amplification)

Field-Effect Transistors

- Used on digital chips (CMOS)

- Configured to act as a switch

S

G

D

turns on current flowing from point S to point D

non-zero current flowing from point G to D

Boolean Logic

- Mathematical basis for digital circuits

- Three basic functions: *and, or,* and *not*

A	B	A and B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A or B
0	0	0
0	1	1
1	0	1
1	1	1

A	not A
0	1
1	0

Digital Logic

- Can implement Boolean functions with transistors

- Five volts represents Boolean *1*

- Zero volts represents Boolean *0*

Transistor Implementing Boolean Not
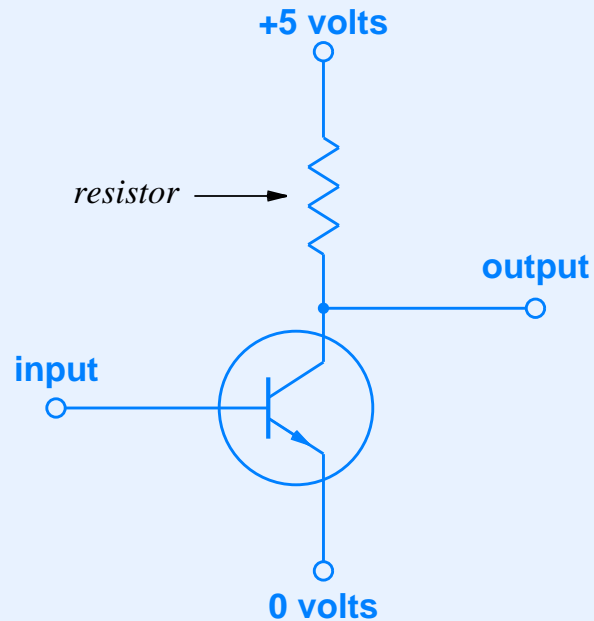
- When input is zero volts, output is five volts

- When input is five volts, output is zero volts
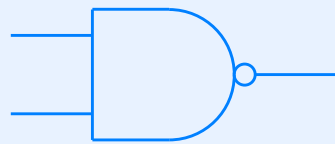
Logic Gate

- Hardware component

- Consists of integrated circuit

- Implements an individual Boolean function

- To reduce complexity, provide inverse of Boolean functions

  - Nand gate implements *not and*

  - Nor gate implements *not or*

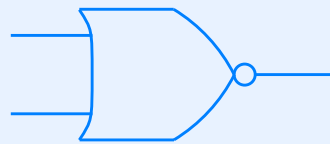 - Inverter implements *not*

Truth Tables For Nand and Nor Gates

A	B	A nand B
0	0	1
0	1	1
1	0	1
1	1	0

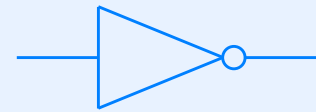| A	B	A nor B
0	0	1
0	1	0
1	0	0
1	1	0

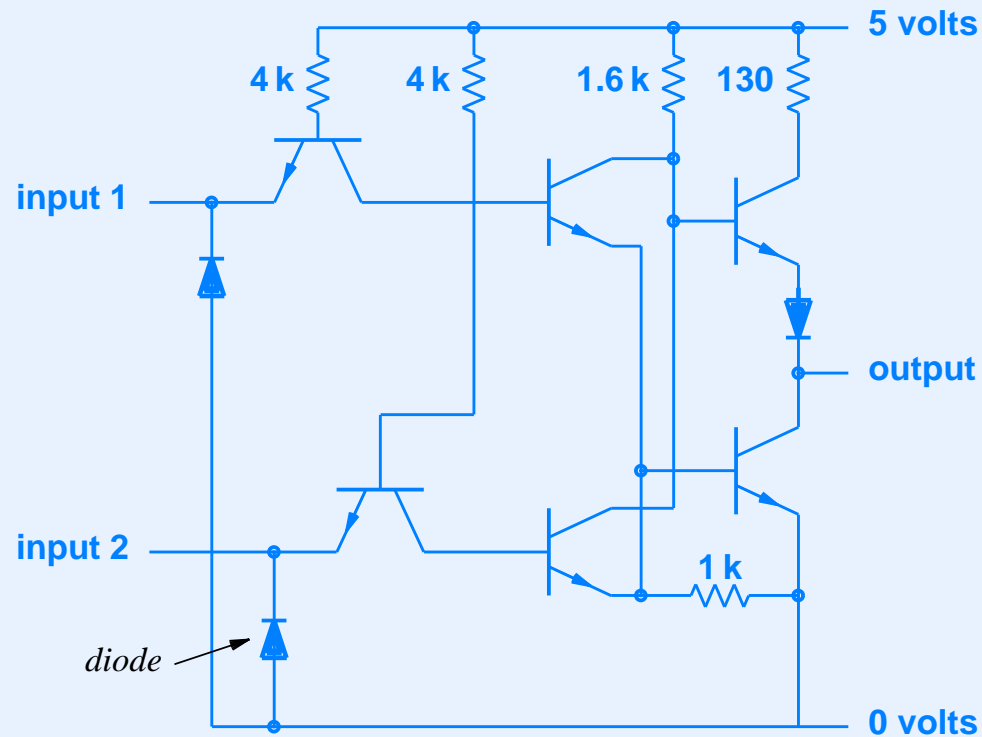# Symbols Used In Schematic Diagrams

nand gate　　　　　**nor gate**　　　　　**inverter**

Example Of Internal Gate Structure (Nor Gate)

- Solid dot indicates electrical connection

Technology For Logic Gates

- Most popular technology known as *Transistor-Transistor Logic* (*TTL*)

- Allows direct interconnection (a wire can connect output from one gate to input of another)

- Single output can connect to multiple inputs

  - Called *fanout*

  - Limited to a small number

Example Interconnection Of TTL Gates

- Two logic gates needed to form logical *and*

    – Output from nand gate connected to input of inverter

input from
power button

input from
disk

output
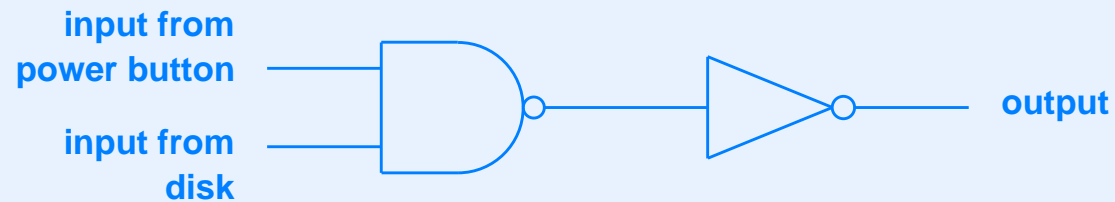
Consider The Following Circuit
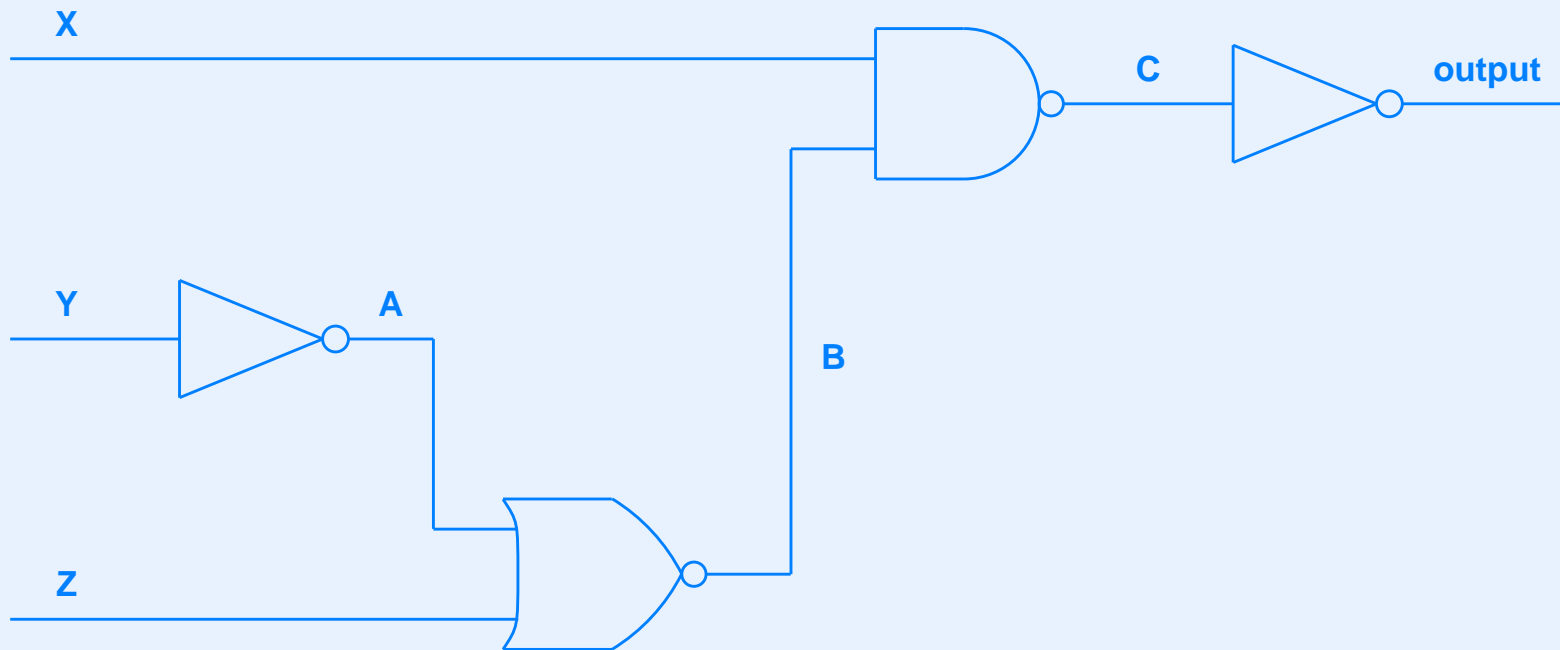
- Question: what does the circuit implement?

Two Ways To Describe Circuit

- Boolean expression

    – Often used when designing circuit

    – Can be transformed to equivalent version that takes fewer gates

- Truth table

    – Enumerates inputs and outputs

    – Often used when debugging a circuit

Describing A Circuit With Boolean Algebra
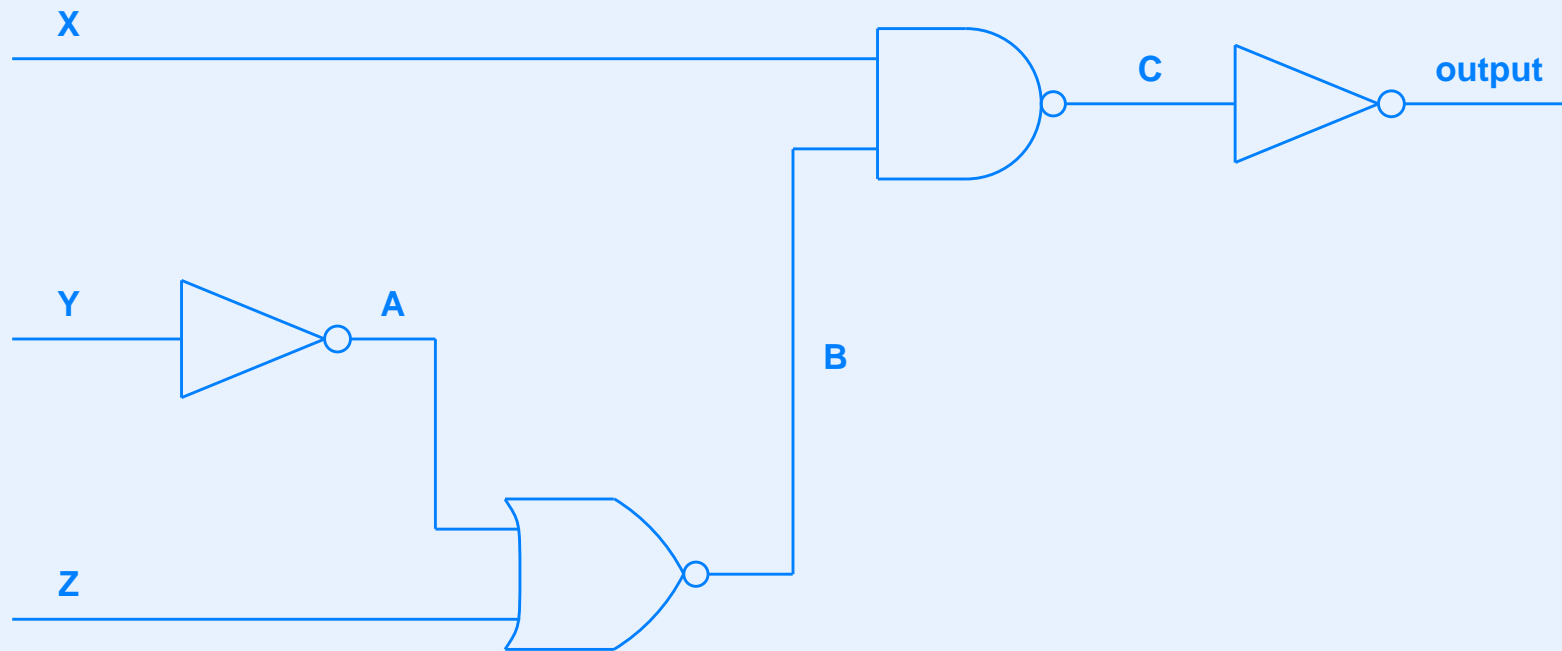
- Value at point *A* is *not Y*

- Value at *B* is:

$$Z \; nor \; (not \; Y)$$

Describing A Circuit With Boolean Algebra
(continued)

- Output is:

$$X \text{ and } (Z \text{ nor } (\text{not } Y))$$

Describing A Circuit With Boolean Algebra
(continued)
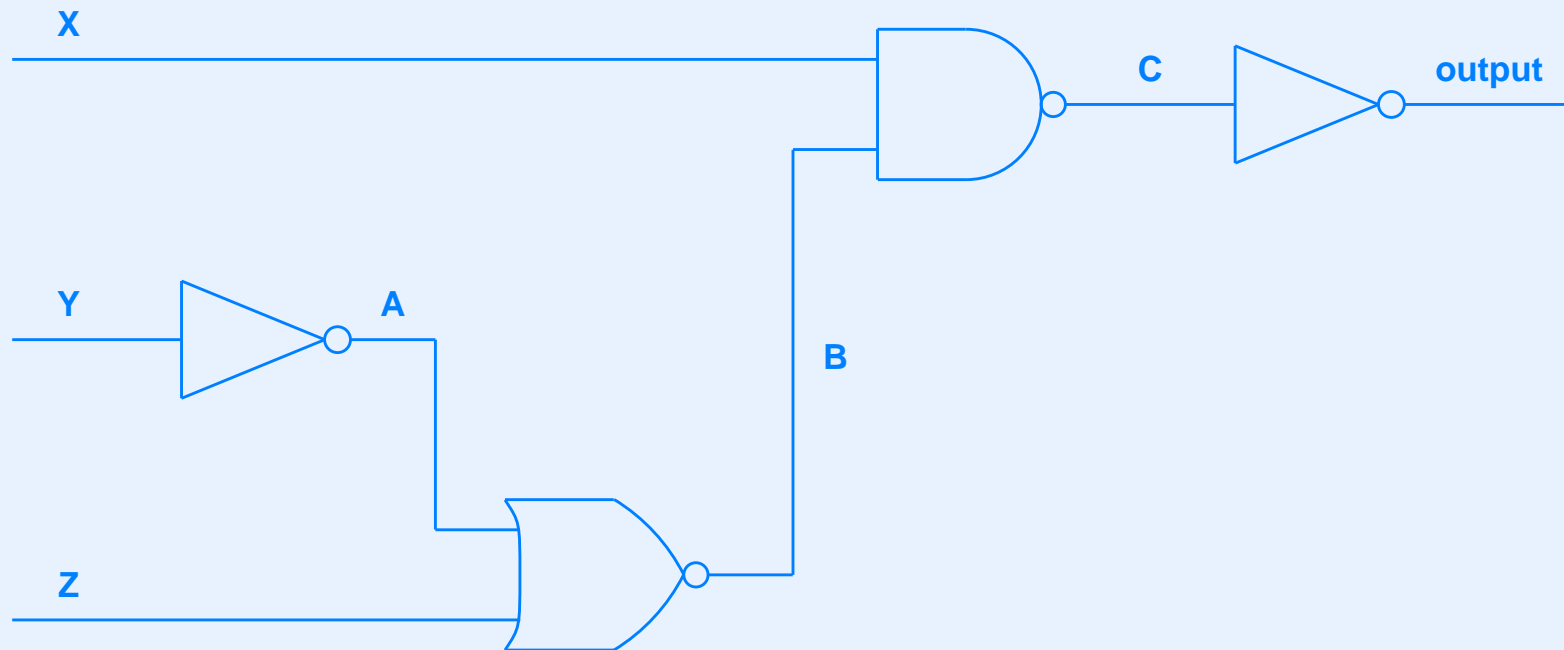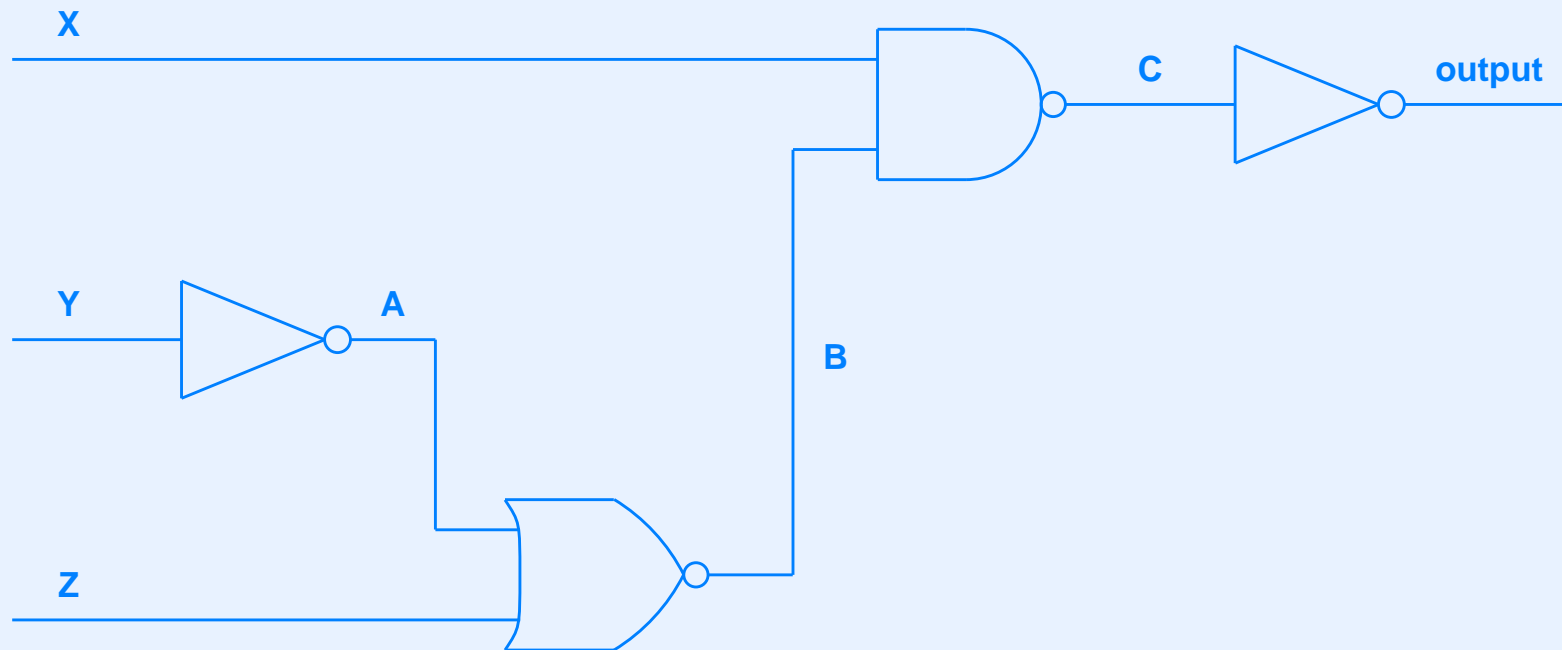

- Output is (alternative):

X and not (Z or (not Y))

Describing A Circuit With A Truth Table
(continued)

X	Y	Z	A	B	C	output
0	0	0	1	0	1	0
0	0	1	1	0	1	0
0	1	0	0	1	1	0
0	1	1	0	0	1	0
1	0	0	1	0	1	0
1	0	1	1	0	1	0
1	1	0	0	1	0	1
1	1	1	0	0	1	0

- Table lists all possible inputs and output for each

- Can also state values for intermediate points

Avoiding Nand / Nor Operations

- Circuits use nand and nor gates

- Sometimes easier for humans to use *and* and *or* operations

- Example circuit or truth table output can be described by Boolean expression:

$$X \text{ and } Y \text{ and } (not \ Z))$$

In Practice

- Only a few connections needed per gate

- Chip has many pins for external connections

- Result: can package multiple gates on each chip

Example Of Logic Gates

- 7400 family of chips

- Package is about one-half inch long

- Implement TTL logic

- Powered by five volts

- Contain multiple gates per chip

Examples Of Gates On 7400-Series Chips
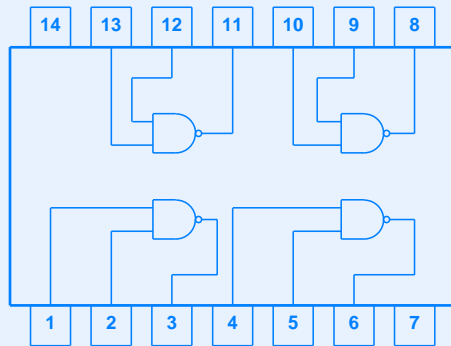
7400
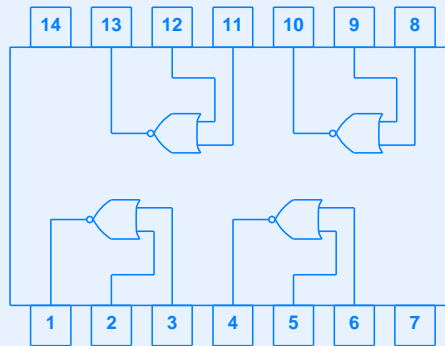
7402
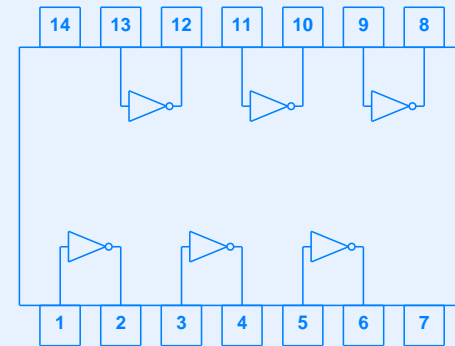
7404

- Pins 7 and 14 connect to ground and power

Logic Gates And Computers

Logic Gates And Computers

- Question: how can computers be constructed from simple logic gates?

Logic Gates And Computers

- Question: how can computers be constructed from simple logic gates?

- Answer: they cannot

Logic Gates And Computers

- Question: how can computers be constructed from simple logic gates?

- Answer: they cannot

- Additional functionality needed

    – Circuits that maintain state

    – Circuits that operate on a clock

Circuits That Maintain State

- More sophisticated than *combinatorial circuits*

- Output depends on history of previous input as well as values on input lines

Example Of Circuit That Maintains State

- Basic *flip-flop*

- Analogous to push-button power switch

- Each new 1 received as input causes output to reverse

  - First input pulse causes flip-flop to turn on

  - Second input pulse causes flip-flop to turn off

Output Of A Flip-Flop

in: 0 0 1 0 1 1 0 0 0 0 1 0 1 0 1 0

out: 0 0 1 1 0 0 0 0 0 0 1 1 0 0 1 1

time increases
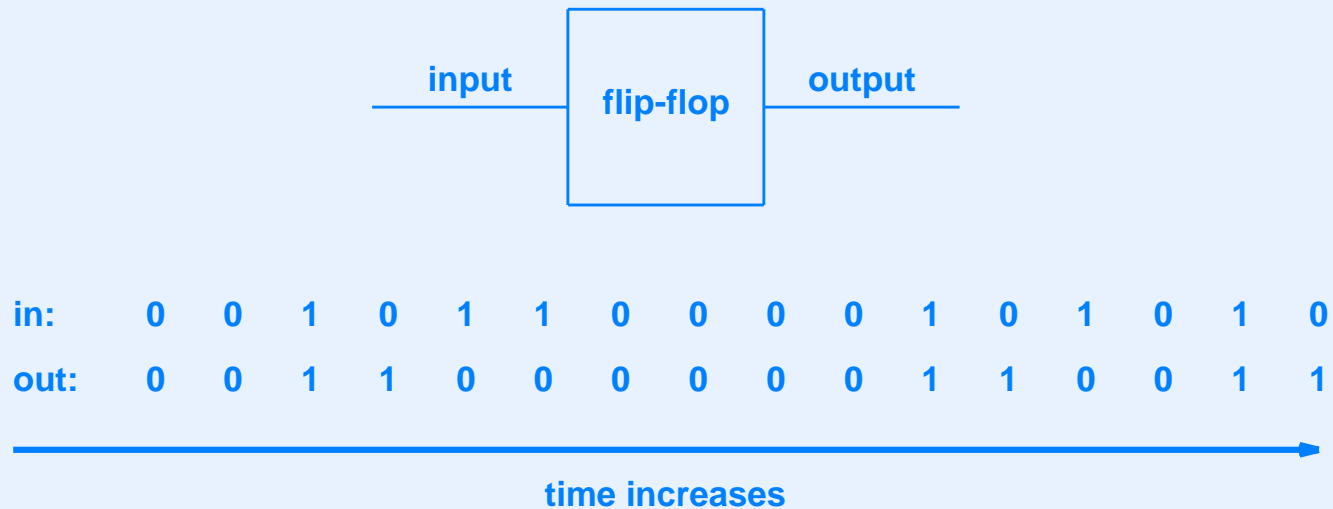
- Note: output only changes when input makes a transition from zero to one

Flip-Flop Action Plotted As Transition Diagram
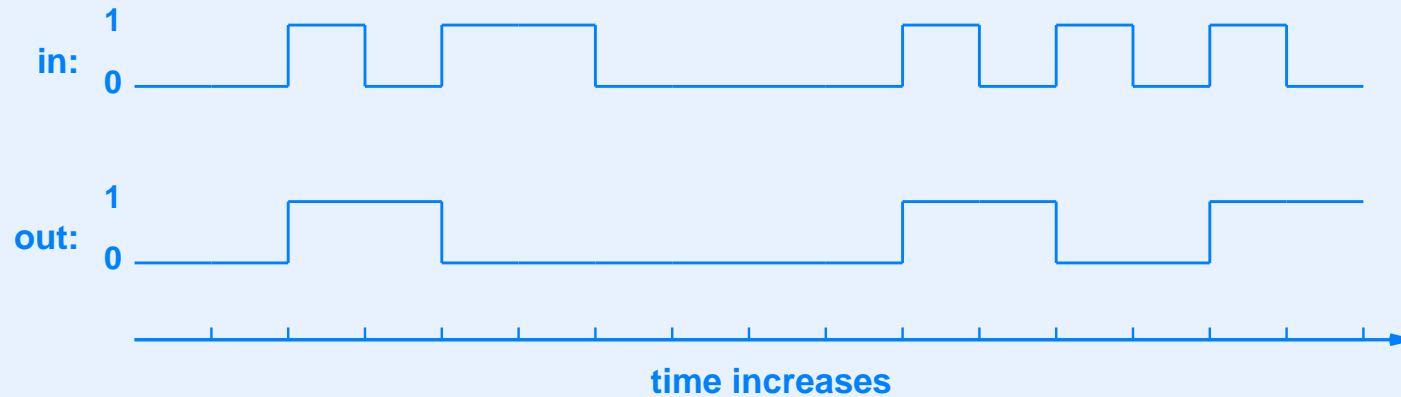
- Output changes on *leading edge* of input

- Also called *rising edge*

Binary Counter

- Counts input pulses

- Output is binary value

- Includes *reset line* to restart count at zero

- Example: 4-bit counter available as single integrated circuit

Illustration Of Counter

input	outputs	decimal
0	0 0 0	0
0	0 0 0	0
1	0 0 1	1
0	0 0 1	1
1	0 1 0	2
1	0 1 0	2
0	0 1 0	2
1	0 1 1	3
0	0 1 1	3
1	1 0 0	4
0	1 0 0	4
1	1 0 1	5

(a)

time increases

(b)
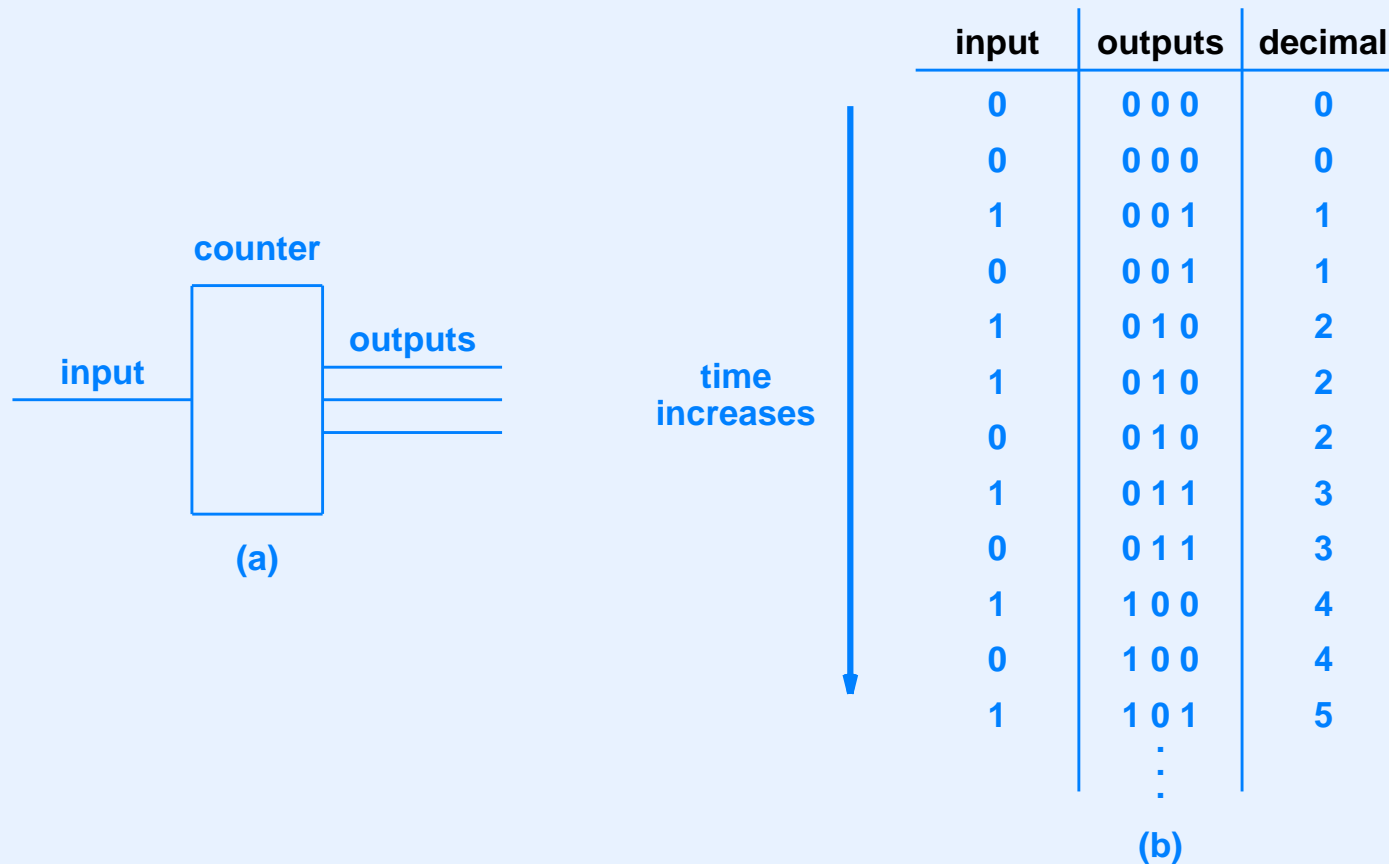
- Part (a) shows the schematic of a counter chip

- Part (b) shows the output as the input changes

Clock

- Electronic circuit that pulses regularly

- Measured in cycles per second (Hz)

- Digital output of clock is sequence of 0 1 0 1 ...

- Permits active circuits

Demultiplexor

- Takes binary number as input

- Uses input to select one output

- Technical distinction

  - *decoder* simply selects one output

  - *demultiplexor* feeds a special input to the selected output

- In practice: engineers often use the term "demux" for either, and blur the distinction

Illustration Of Demultiplexor

- Binary value on inputs determines which output is active
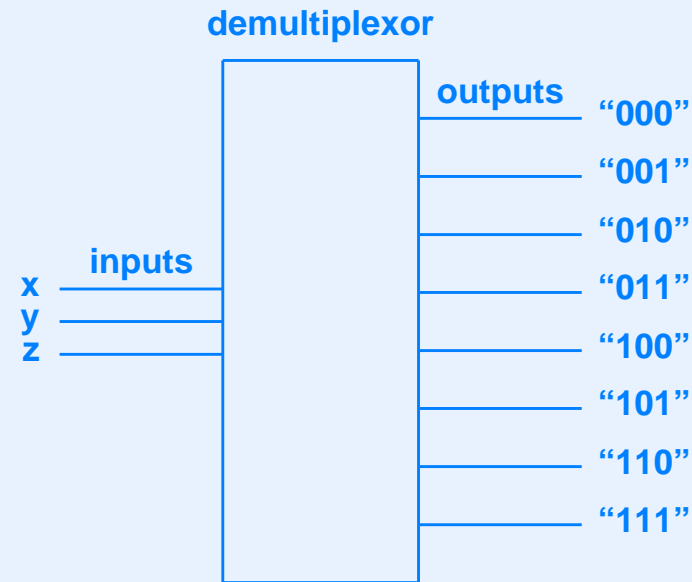
Example: Execute A Sequence Of Steps

- Desired sequence

  - Test the battery

  - Power on and test the memory

  - Start the disk spinning

  - Power up the display

  - Read boot sector from disk into memory

  - Start the CPU

Circuit To Execute A Sequence

demultiplexor

counter

clock

not used

test battery

test memory

start disk

power CRT

read boot blk

start CPU

not used
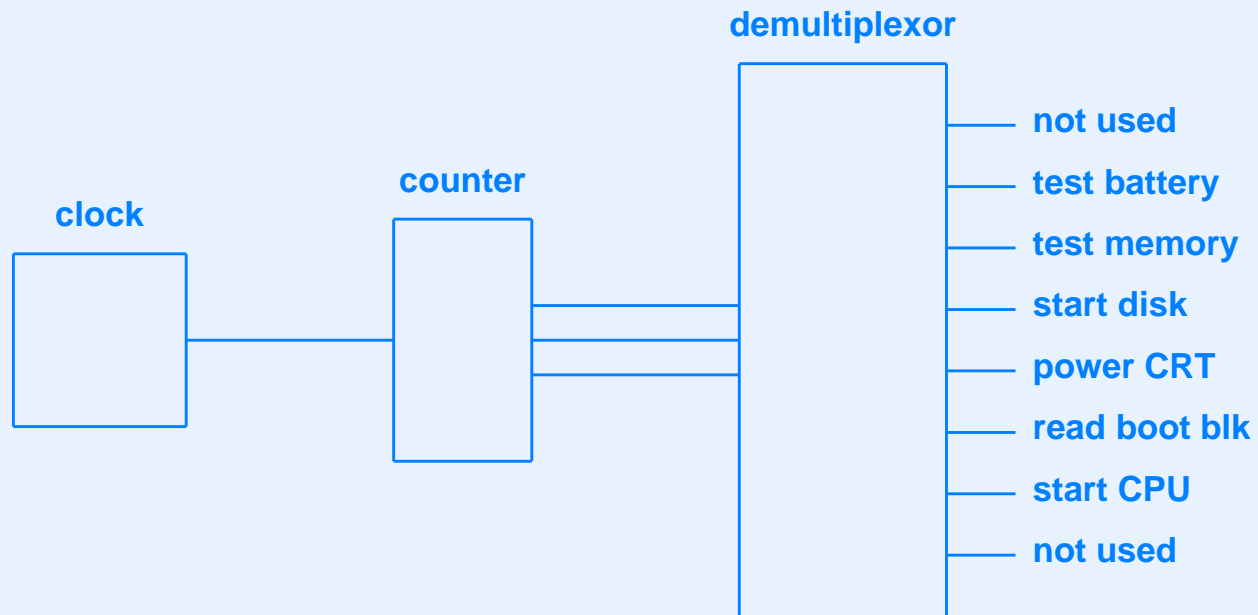
Feedback

- Output of circuit used as an input

- Allows more control

- Example: stop sequence when output *F* becomes active

- Boolean algebra

CLOCK *and* (*not* F)

Illustration Of Feedback For Termination
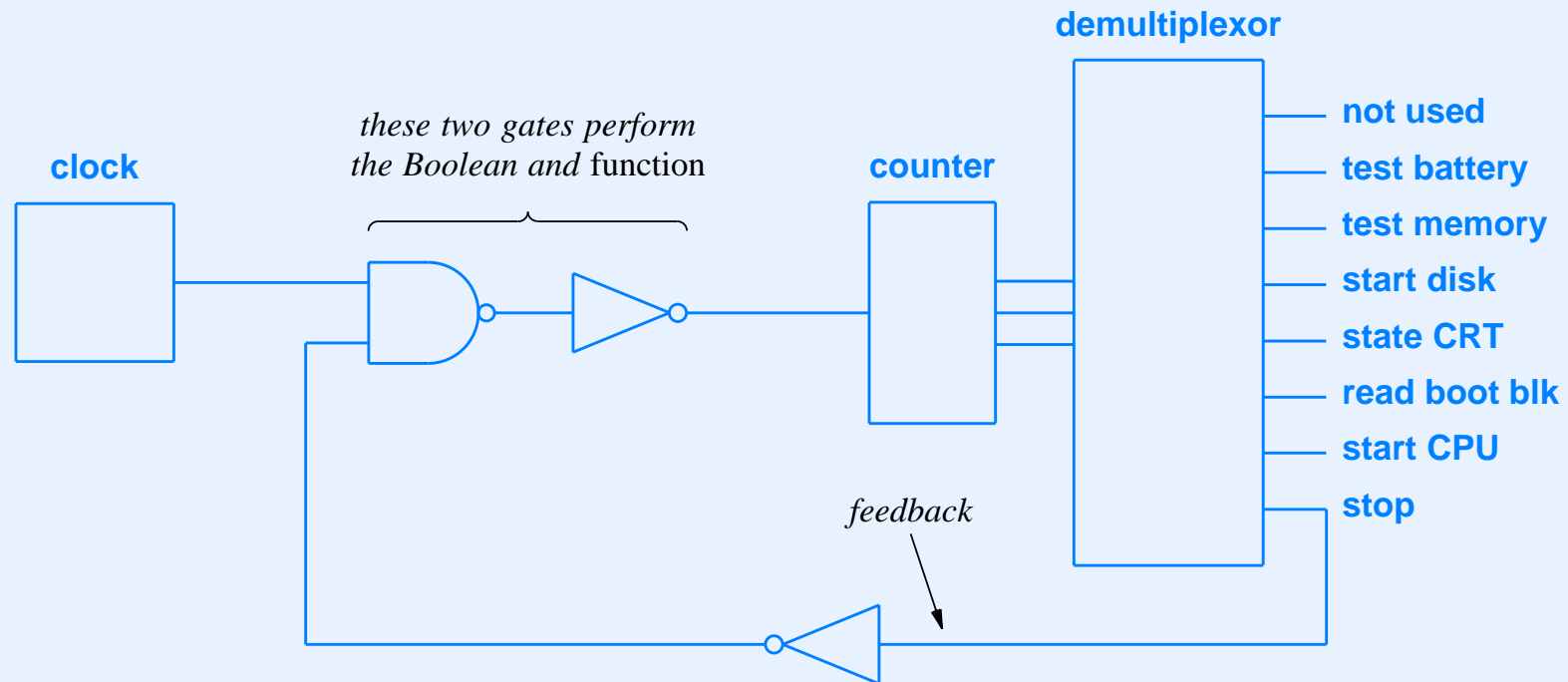
- Note additional input needed to restart sequence

Spare Gates

- Note: because chip contains multiple gates, some gates may be unused

- May be able to substitute spare gates in place of additional chip

- Example uses spare nand gate as inverter by connecting one input to five volts:

$$1 \; nand \; x \quad = \quad not \; x$$

Practical Engineering Concerns

- Power consumption (wiring must carry sufficient power)

- Heat dissipation (chips must be kept cool)

- Timing (gates take time to settle after input changes)

- Clock synchronization (clock signal must reach all chips simultaneously)

Illustration Of Clock Skew
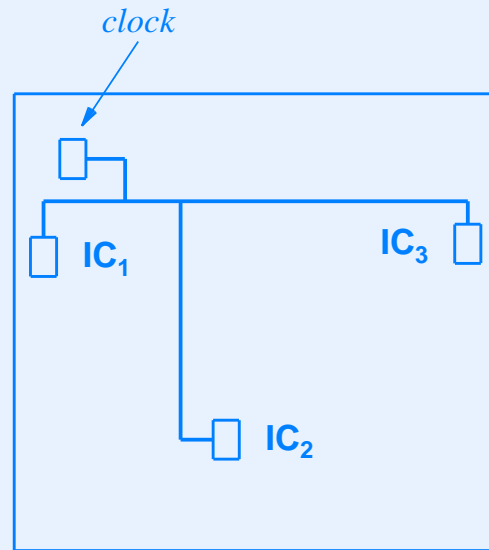
- Length of wire determines time required for signal to propagate

Classification Of Technologies

Name	Example Use
Small Scale Integration (SSI)	The most basic logic such as Boolean gates
Medium Scale Integration (MSI)	Intermediate logic such as counters
Large Scale Integration (LSI)	More complex logic such as embedded processors
Very Large Scale Integration (VLSI)	The most complex processors (i.e., CPUs)

Levels Of Abstraction

Abstraction	Implemented With
Computer	Circuit board(s)
Circuit board	Components such as processor and memory
Processor	VLSI chip
VLSI chip	Many gates
Gate	Many transistors
Transistor	Semiconductor implemented in silicon

Reconfigurable Logic

- Alternative to standard gates

- Allows chip to be configured multiple times

- Can create

  - Various gates

  - Interconnections

- Most popular form: *Field Programmable Gate Array* (*FPGA*)

Summary

- Computer systems are constructed of digital logic circuits

- Fundamental building block is called a *gate*

- Digital circuit can be described by

  - Boolean algebra (most useful when designing)

  - Truth table (most useful when debugging)

- Clock allows active circuit to perform sequence of operations

- Feedback allows output to control processing

- Practical engineering concerns include

  - Power consumption and heat dissipation

  - Clock skew and synchronization

Questions?

III

Data And Program Representation

Digital Logic

- Built on two-valued logic system

- Can be interpreted as

    - *Five volts* and *zero volts*

    - *High* and *low*

    - *True* and *false*

Data Representation

- Builds on digital logic

- Applies familiar abstractions

- Interprets sets of Boolean values as

    – Numbers

    – Characters

    – Addresses

Binary Digit (Bit)

- Direct representation of digital logic values

- Assigned mathematical interpretation

    – *0* and *1*

- Multiple bits used to represent complex data item

Byte

- Set of multiple bits

- Size depends on computer

- Examples of byte sizes

  - CDC: 6-bit byte

  - BBN: 10-bit byte

  - IBM: 8-bit byte

- On many computers, smallest addressable unit of storage

- Note: following most modern computers, we will assume an 8-bit byte

Byte Size And Values

- Number of bits per byte determines range of values that can be stored

- Byte of k bits can store 2^k values

- Examples

    - Six-bit byte can store 64 possible values

    - Eight-bit byte can store 256 possible values

Binary Representation

000	**010**	**100**	**110**
001	**011**	**101**	**111**

- All possible combinations of three bits

Meaning Of Bits

- Bits themselves have no intrinsic meaning

- Byte merely stores string of 0's and 1's

- All interpretation determined by use

Example Of Interpretation

- Assume three bits used for status of peripheral devices

  – First bit has the value 1 if a disk is connected

  – Second bit has the value 1 if a printer is connected

  – Third bit has the value 1 if a keyboard is connected

Arithmetic Values

- Combination of bits interpreted as an integer

- Positional representation uses base 2

- Note: interpretation must specify order of bits

Illustration Of Positional Interpretation

$$2^5 = 32 \qquad 2^4 = 16 \qquad 2^3 = 8 \qquad 2^2 = 4 \qquad 2^1 = 2 \qquad 2^0 = 1$$
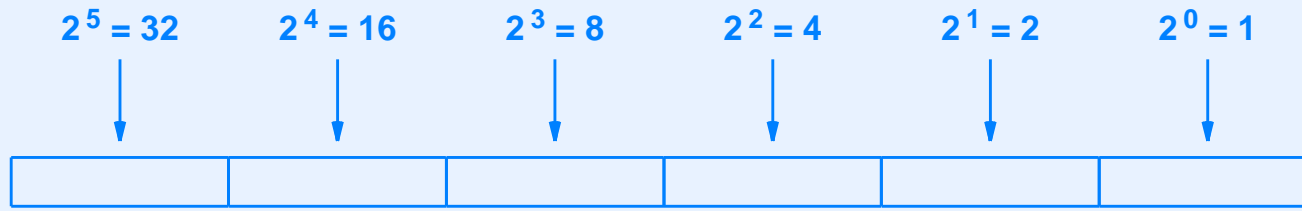
- Example:

$$0 \; 1 \; 0 \; 1 \; 0 \; 1$$

is interpreted as:

$$0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \; = \; 21$$

The Range Of Values

A set of k bits can be interpreted to represent a binary integer. When conventional positional notation is used, the values that can be represented with k bits range from 0 through $2^k - 1$.

Hexadecimal Notation

- Convenient way to represent binary data

- Uses base 16

- Each hex digit encodes four bits

Hexadecimal Digits

Hex Digit	Binary Value	Decimal Equivalent
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

Hexadecimal Constants

- Supported in some programming languages

- Typical syntax: constant begins with *0x*

- Example:

$$0xDEC90949$$

Character Sets

- Symbols for upper and lower case letters, digits, and punctuation marks

- Set of symbols defined by computer system

- Each symbol assigned unique bit pattern

- Typically, character set size determined by byte size

Example Character Encodings

- EBCDIC

- ASCII

- Unicode

EBCDIC

- Extended Binary Coded Decimal Interchange Code

- Defined by IBM

- Popular in 1960s

- Still used on IBM mainframe computers

- Example encoding: lower case letter *a* assigned binary value

10000001

ASCII

- American Standard Code for Information Interchange

- Vendor independent: defined by American National Standards Institute (ANSI)

- Adopted by PC manufacturers

- Specifies 128 characters

- Example encoding: lower case letter a assigned binary value

$$01100001$$

Full ASCII Character Set

00	nul	01	soh	02	stx	03	etx	04	eot	05	enq	06	ack	07	bel
08	bs	09	ht	0A	lf	0B	vt	0C	np	0D	cr	0E	so	0F	si
10	dle	11	dc1	12	dc2	13	dc3	14	dc4	15	nak	16	syn	17	etb
18	can	19	em	1A	sub	1B	esc	1C	fs	1D	gs	1e	rs	1F	us
20	sp	21	!	22	"	23	#	24	$	25	%	26	&	27	'
28	(	29	)	2A	*	2B	+	2C	,	2D	–	2E	.	2F	/
30	0	31	1	32	2	33	3	34	4	35	5	36	6	37	7
38	8	39	9	3A	:	3B	;	3C	<	3D	=	3E	>	3F	?
40	@	41	A	42	B	43	C	44	D	45	E	46	F	47	G
48	H	49	I	4A	J	4B	K	4C	L	4D	M	4E	N	4F	O
50	P	51	Q	52	R	53	S	54	T	55	U	56	V	57	W
58	X	59	Y	5A	Z	5B	[	5C	\	5D	]	5E	^	5F	_
60	`	61	a	62	b	63	c	64	d	65	e	66	f	67	g
68	h	69	i	6A	j	6B	k	6C	l	6D	m	6E	n	6F	o
70	p	71	q	72	r	73	s	74	t	75	u	76	v	77	w
78	x	79	y	7A	z	7B	{	7C			7D	}	7E	~	7F	del

Unicode

- Each character is 16 bits long

- Can represent larger set of characters

- Motivation: accommodate languages such as Chinese

Integer Representation In Binary

- Each binary integer represented in k bits

- Computers have used k = 8, 16, 32, 60, and 64

- Many computers support multiple integer sizes (e.g., 16 and 32 bit integers)

- 2^k possible bit combinations exist for k bits

- Positional interpretation produces *unsigned integers*

Unsigned Integers

- Straightforward positional interpretation

- Each successive bit represents next power of 2

- No provision for negative values

- Arithmetic operations can produce *overflow* or *underflow* (result cannot be represented in k bits)

- Handled with *wraparound* and *carry bit*

Illustration Of Overflow

```
        1  0  0
  +     1  1  0
  _____
  1  0  1  0
```

overflow   *result*
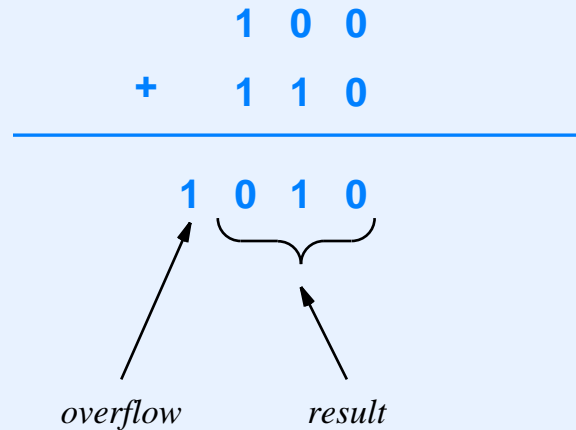
- Values wrap around address space

- Hardware records overflow in separate carry indicator

Signed Values

- Needed by most programs

- Several representations possible

- Each has been used in at least one computer

- Some bit patterns used for negative values (typically half)

- Tradeoff: unsigned representation cannot store negative values, but can store integers that are twice as large as a signed representation

Example Signed Integer Representations

- Sign magnitude

- One's complement

- Two's complement

- Note: each has interesting quirks

Sign Magnitude Representation

- Familiar to humans

- First bit represents sign

- Successive bits represent absolute value of integer

- Interesting quirk: can create negative zero

One's Complement Representation

- Positive number uses positional representation

- Negative number formed by inverting all bits of positive value

- Example of 4-bit one's complement

  - 0 0 1 0 represents 2

  - 1 1 0 1 represents −2

- Interesting quirk: two representations for zero (all 0's and all 1's)

Two's Complement Representation

- Positive number uses positional representation

- Negative number formed by subtracting 1 from positive value and inverting all bits of result

- Example of 4-bit two's complement

    - 0 0 1 0 represents 2

    - 1 1 1 0 represents −2

    - High-order bit is set if number is negative

- Interesting quirk: one more negative values than positive values

Example Of Values In Unsigned And Two's Complement Representations

Binary Value	Unsigned Equivalent	Two's Complement Equivalent
1111	15	−1
1110	14	−2
1101	13	−3
1100	12	−4
1011	11	−5
1010	10	−6
1001	9	−7
1000	8	−8
0111	7	7
0110	6	6
0101	5	5
0100	4	4
0011	3	3
0010	2	2
0001	1	1
0000	0	0

Implementation Of Unsigned
And Two's Complement

A computer can use a single piece of hardware to provide unsigned or two's complement integer arithmetic; software running on the computer can choose an interpretation for each integer.

- Example (k = 4)

    – Adding 1 to binary 1 0 0 1 produces 1 0 1 0

    – Unsigned interpretation goes from 9 to 10

    – Two's complement interpretation goes from –7 to –6

Sign Extension

- Needed when computer has multiple sizes of integers

- Works for unsigned and two's complement representations

- Extends high-order bit (known as *sign bit*)

Explanation Of Sign Extension

- Assume computer

    – Supports 32-bit and 64-bit integers

    – Uses two's complement representation

- When 32-bit integer assigned to 64-bit integer, correct numeric value requires upper sixteen bits to be filled with zeroes for positive number or ones for negative number

- In essence, sign bit from shorter integer must be *extended* to fill high-order bits of larger integer

Example Of Sign Extension During Assignment

- The 8-bit version of integer –3 is:

  1 1 1 1 1 1 0 1

- The 16-bit version of integer –3 is:

  1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1

- During assignment to a larger integer, hardware copies all bits of smaller integer and then replicates the high-order (sign) bit in remaining bits

Example Of Sign Extension During Shift

- Right shift of a negative value should produce a negative value

- Example

  – Shifting –4 one bit should produce –2 (divide by 2)

  – Using sixteen-bit representation, –4 is:

    1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0

- After right shift of one bit, value is –2:

    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0

- Solution: replicate high-order bit during right shift

Summary Of Sign Extension

Sign extension: in two's complement arithmetic, when an integer Q composed of K bits is copied to an integer of more than K bits, the additional high-order bits are made equal to the top bit of Q. Extending the sign bit means the numeric value remains the same.

A Consequence For Programmers

Because two's complement hardware performs sign extension, copying an unsigned integer to a larger unsigned integer changes the value; to prevent such errors from occurring, a programmer or a compiler must add code to mask off the extended sign bits.

Numbering Bits And Bytes

- Need to choose order for

    - Storage in physical memory system

    - Transmission over serial medium (e.g., a data network)

- Bit order

    - Handled by hardware

    - Usually hidden from programmer

- Byte order

    - Affects multi-byte data items such as integers

    - Visible and important to programmer

Possible Byte Order

- Least significant byte of integer in lowest memory location

    – Known as *little endian*

- Most significant byte of integer in lowest memory location

    – Known as *big endian*

- Other orderings

    – Digital Equipment Corporation once used an ordering with sixteen-bit words in big endian order and bytes within the words in little endian order.

- Note: only big and little endian storage are popular

Illustration Of Big And Little Endian Byte Order

0	1	2	3
0x00	0x00	0x00	0x01

Big Endian

3	2	1	0
0x00	0x00	0x00	0x01

Little Endian

- Note: difference is especially important when transferring data between computers for which the byte ordering differs

Floating Point

- Fundamental idea: follow standard scientific representation

- Store two basic items

- Example: Avogadro's number

$$6.022 \times 10^{23}$$

# Floating Point Representation

- Use base 2 instead of base 10

- Keep two conceptual items

    – Exponent that specifies the order of magnitude in a base

    – Mantissa that specifies most significant part of value

Optimizing Floating Point

- Value is normalized

- Leading bit is implicit

- Exponent is biased to allow negative values

- Normalization eliminates leading zeroes

- No need to store leading bit (0 is special case)

Example Floating Point Representation: IEEE Standard 754

- Specifies single-precision and double-precision representations

- Widely adopted by computer architects

31	23	0
S	exponent	mantissa (bits 0 – 22)

63	52	0
S	exponent	mantissa (bits 0 - 51)
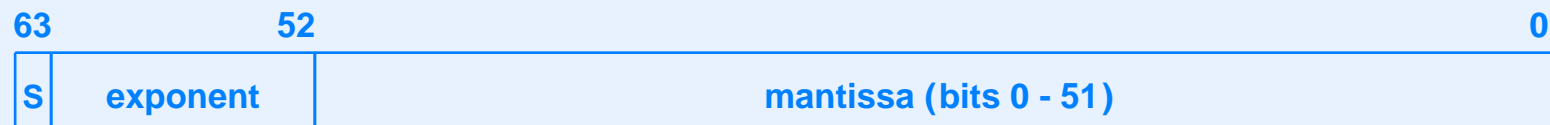
Special Values In IEEE Floating Point

- Zero

- Positive infinity

- Negative infinity

- Note: infinity values handle cases such as the result of dividing by zero

Range Of Values In IEEE Floating Point

- Single precision range is:

$$2^{-126} \text{ to } 2^{127}$$

- Decimal equivalent is approximately:

$$10^{-38} \text{ to } 10^{38}$$

- Double precision range is:

$$10^{-308} \text{ to } 10^{308}$$

Data Aggregates

- Typically arranged in contiguous memory

- Example: three integers

0	1	2	3	4	5
integer #1		integer #2		integer #3	

- More details later in the course

Summary

- Basic output from digital logic is a bit

- Bits grouped into sets to represent

  – Integers

  – Characters

  – Floating point values

- Integers can be represented as

  – Sign magnitude

  – One's complement

  – Two's complement

Summary

- One piece of hardware can be used for both

    - Two's complement arithmetic

    - Unsigned arithmetic

- Bytes of integer can be numbered in

    - Big-endian order

    - Little-endian order

- Organizations such as ANSI and IEEE define standards for data representation

Questions?

IV

Processors

Terminology

- The terms *processor* and *computational engine* refer broadly to any mechanism that drives computation

- Wide variety of sizes and complexity

- Processor is key element in all computational systems

Von Neumann Architecture

- Characteristic of most modern processors

- Reference to mathematician John Von Neumann who was one of the computer architecture pioneers

- Fundamental concept is a *stored program*

- Three basic components interact to form a computational system

  - Processor

  - Memory

  - I/O facilities

Illustration Of Von Neumann Architecture

computer

processor	**memory**

input/output facilities
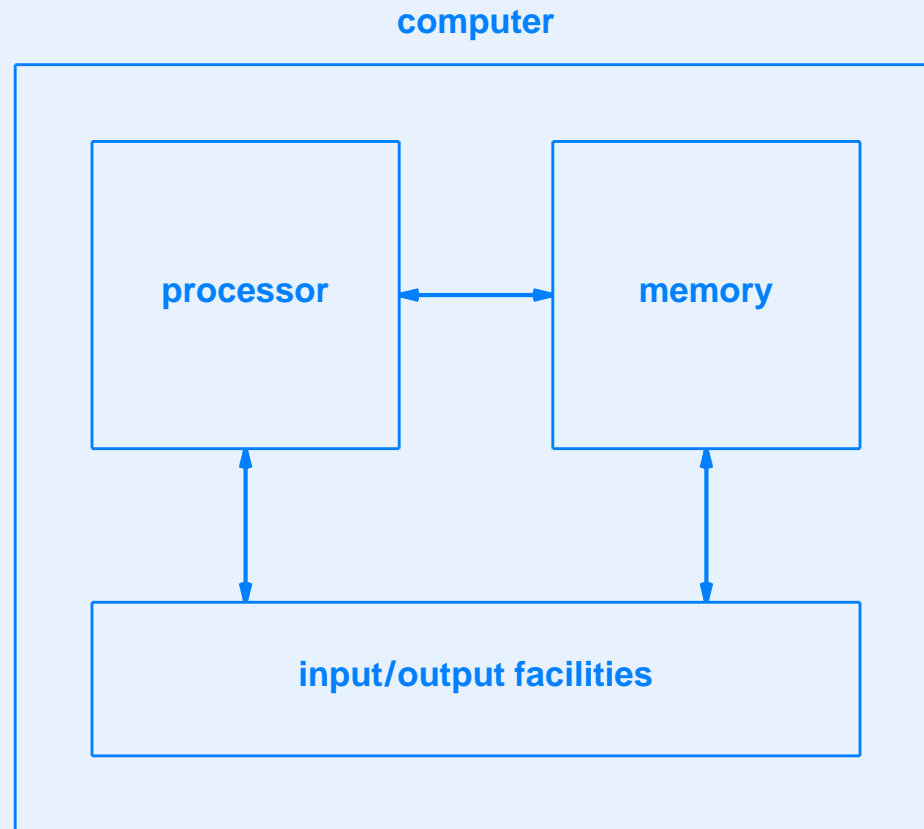
Processor

- Digital device

- Performs computation involving multiple steps

- Wide variety of capabilities

- Mechanisms available

  - Fixed logic

  - Selectable logic

  - Parameterized logic

  - Programmable logic

Fixed Logic Processor

- Least powerful

- Performs a single operation

- Functionality hardwired (cannot be changed)

- Example: math coprocessor that computes *sine*

Selectable Logic Processor

- Slightly more powerful than fixed logic

- Can perform more than one function

- Exact function specified each time processor invoked

- Example: math coprocessor that computes *sine* or *cosine*

Parameterized Logic Processor

- Accepts set of parameters that control computation

- Parameters set for each invocation

- Example

  – Compute hash function, $h(x)$, that multiplies argument x by a constant p and computes the remainder modulo constant q

 – Parameters specify constants p and q used in computation

Programmable Logic Processor

- Greatest flexibility

- Function to compute can be changed

- Sequence of steps can be specified for each invocation

- Example: conventional CPU

Hierarchical Structure And Computational Engines

- Most computer architecture follows a hierarchical approach

- Subparts of a large, central processor are sophisticated enough to meet our definition of processor

- Some engineers use term *computational engine* for subpiece that is less powerful than main processor
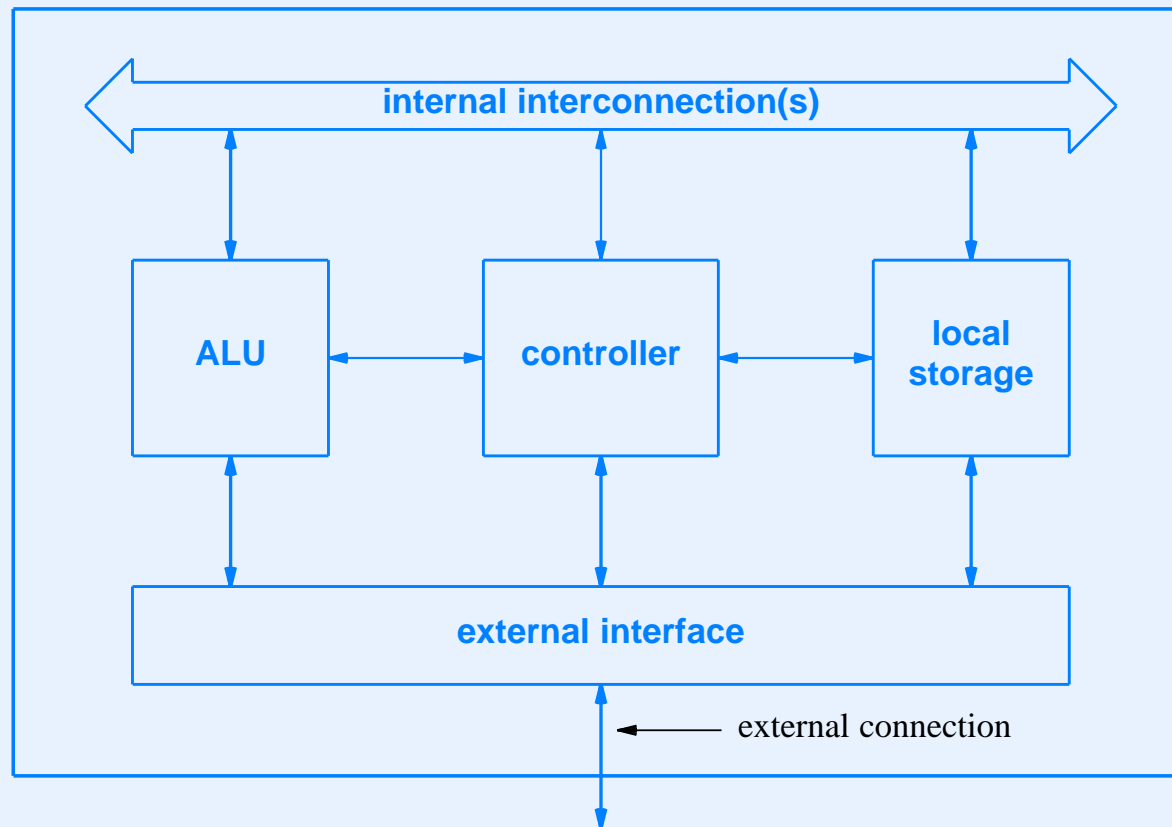
Illustration Of Processor Hierarchy
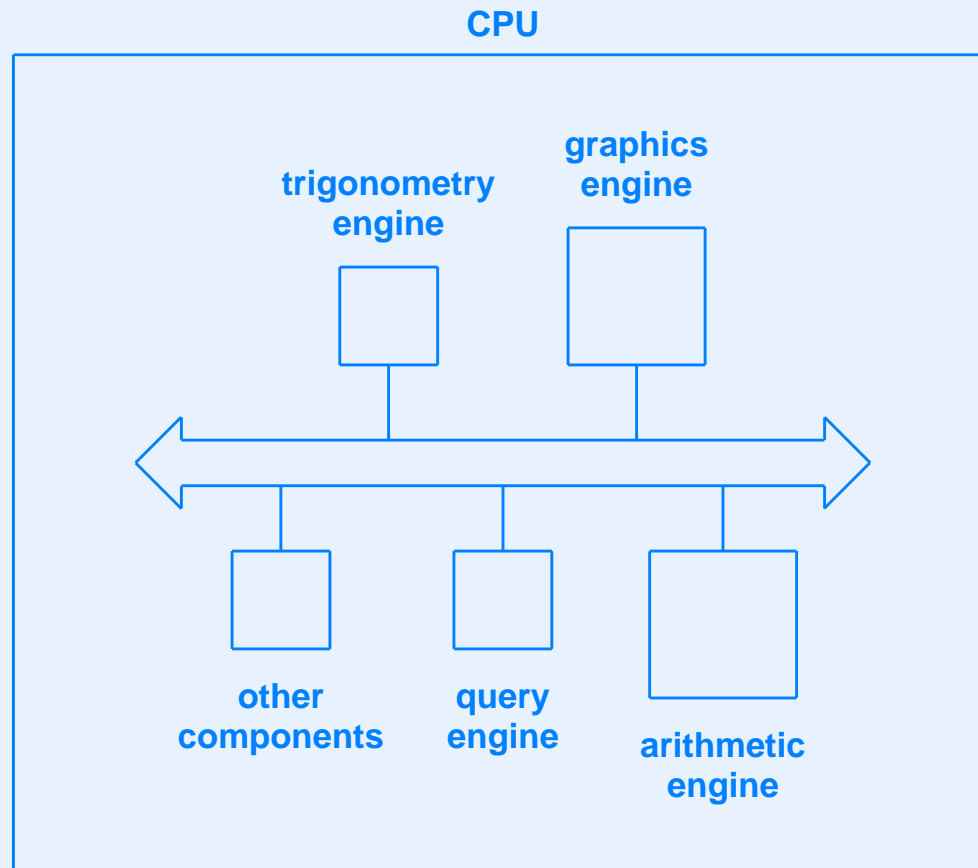
Major Components Of A Conventional Processor

- Controller

- Computational engine (ALU)

- Internal interconnection(s)

- External interface

- Local data storage

Illustration Of A Conventional Processor

Parts Of A Conventional Processor

- Controller

    – Overall responsibility for execution

    – Moves through sequence of steps

    – Coordinates other units

- Computational engine

    – Operates as directed by controller

    – Typically provides arithmetic and Boolean operations

    – Performs one operation at a time

Parts Of A Conventional Processor
(continued)

- Internal interconnections

    – Allow transfer of values among units of the processor

    – Sometimes called *data path*

- External interface

    – Handles communication between processor and rest of computer system

    – Provides connections to external memory as well as external I/O devices

Parts Of A Conventional Processor
(continued)

- Local data storage

    – Holds data values for operations

    – Values must be inserted (e.g., loaded from memory) before the operation can be performed

    – Typically implemented with *registers*

Arithmetic Logic Unit
(ALU)

- Main computational engine in conventional processor

- Complex unit that can perform variety of tasks

- Typical ALU operations

    – Arithmetic (integer add, subtract, multiply, divide)

    – Shift (left, right, circular)

    – Boolean (*and, or, not, exclusive or*)

Processor Categories And Roles

- Many possible roles for individual processors in

    - Coprocessors

    - Microcontrollers

    - Microsequencers

    - Embedded system processors

    - General-purpose processors

Coprocessor

- Operates in conjunction with and under the control of another processor

- Usually

    - Special-purpose processor

    - Performs a single task

    - Operates at high speed

- Example: floating point accelerator

Microcontroller

- Programmable device

- Dedicated to control of a physical system

- Example: run automobile engine or grocery store door

Example Steps A Microcontroller Performs (Automatic Door)

```
do forever {
        wait for the sensor to be tripped;
        turn on power to the door motor;
        wait for a signal that indicates the
                door is open;
        wait for the sensor to reset;
        delay ten seconds;
        turn off power to the door motor;
}
```

Microsequencer

- Similar to microcontroller

- Controls coprocessors and other engines within a large processor

- Example: move operands to floating point unit; invoke an operation; move result back to memory

Embedded System Processor

- Runs sophisticated electronic device

- Usually more powerful than microcontroller

- Example: control DVD player, including commands received from a remote control as well as from the front panel

General-Purpose Processor

- Most powerful type of processor

- Completely programmable

- Full functionality

- Example: CPU in a personal computer

Processor Implementation

- Originally: discrete logic

- Later: single circuit board

- Now

    – Single chip

    – Part of a chip

Definition Of Programmable Device

To a computer architect, a processor is classified as programmable if at some level of detail, the processor is separate from the program it runs. To a user, it may appear that the program and processor are integrated, and it may not be possible to change the program without replacing the processor.

Fetch-Execute Cycle

- Basis for programmable processors

- Allows processor to move through program steps automatically

- Implemented by processor hardware

- Note:

At some level, every programmable processor implements a fetch-execute cycle.

Fetch-Execute Algorithm

Repeat forever {

    Fetch: access the next step of the program from the location in which the program has been stored.

    Execute: Perform the step of the program.

}

- Note: we will discuss in more detail later

Clock Rate And Instruction Rate

- Clock rate

    - Rate at which gates are clocked

    - Provides a measure of the underlying hardware speed

- Instruction rate

    - Measures the number of instructions a processor can *execute* per unit time

    - Varies because some instructions take more time than others

Clock Rate And Instruction Rate
(continued)

The fetch-execute cycle does not proceed at a fixed rate because the time required to execute a given instruction depends on the operation being performed. An operation such as multiplication requires more time than an operation such as addition.

Stopping A Processor

- Processor runs fetch-execute indefinitely

- Software must plan next step

- When last step of application program finishes

  – Embedded system: processor enters a loop testing for a change in inputs

  – General purpose system: operating system executes an infinite loop

- Note: a few processors provide a way to stop the fetch-execute cycle until I/O activity occurs

Starting A Processor

- Processor hardware includes a reset line that stops fetch-execute during power-down

- During power-up, logic holds the reset until the processor and memory are initialized

- Power-up steps known as *bootstrap*

Summary

- Processor performs a computation involving multiple steps

- Many types of processors

    – Coprocessor

    – Microcontroller

    – Microsequencer

    – Embedded system processor

    – General-purpose processor

- Arithmetic Logic Unit (ALU) performs basic arithmetic and Boolean operations

Summary
(continued)

- Hardware in programmable processor runs fetch-execute cycle

- Most modern processors consist of single integrated circuit

Questions?

V

Processor Types
And
Instruction Sets

What Instructions Should A Processor Offer?

What Instructions Should A Processor Offer?

- Minimum set is sufficient, but inconvenient

What Instructions Should A Processor Offer?

- Minimum set is sufficient, but inconvenient

- Extremely large set is convenient, but inefficient

What Instructions Should
A Processor Offer?

- Minimum set is sufficient, but inconvenient

- Extremely large set is convenient, but inefficient

- Architect must consider additional factors

  - Physical size of processor

  - Expected use

  - Power consumption

What Instructions Should A Processor Offer?

- Minimum set is sufficient, but inconvenient

- Extremely large set is convenient, but inefficient

- Architect must consider additional factors

    – Physical size of processor

    – Expected use

    – Power consumption

The Point About Instruction Sets

The set of operations a processor provides represents a tradeoff among the cost of the hardware, the convenience for a programmer, and engineering considerations such as power consumption.

Representation Details

- Architect must choose

  - Set of instructions

  - Exact representation hardware uses for each instruction (*instruction format*)

  - Precise meaning when instruction executed

- The definition of an *instruction set* includes all details

Parts Of An Instruction

- *Opcode* specifies instruction to be performed

- *Operands* specify data values on which to operate

- *Result location* specifies where result will be placed

Instruction Format

- Instruction represented as binary string

- Typically

    – Opcode at beginning of instruction

    – Operands follow opcode

opcode	operand 1	operand 2	. . .

Instruction Length

- Fixed-length

  - Every instruction is same size

  - Hardware is less complex

  - Hardware can run faster

- Variable-length

  - Some instructions shorter than others

  - Appeals to programmers

  - More efficient use of memory

The Point About Fixed-Length Instructions

When a fixed-length instruction set is employed, some instructions contain extra fields that the hardware ignores. The unused fields should be viewed as part of a hardware optimization, not as an indication of a poor design.

General-Purpose Registers

- High-speed storage device

- Usually part of the processor (on chip)

- Each register small size (typically, each register can accommodate an integer)

- Basic operations are *fetch* and *store*

- Numbered from 0 through $N-1$

- Many processors require operands for arithmetic operations to be placed in general-purpose registers

Floating Point Registers

- Usually separate from general-purpose registers

- Each holds one floating-point value

- Many processors require operands for floating point operations to be placed in floating point registers

Example Of Programming With Registers

- Task

    – Add the contents of variables X and Y

    – Place the result in variable Z

Example Of Programming With Registers

- Task

  - Add the contents of variables X and Y

  - Place the result in variable Z

- Example steps

  - Load a copy of X into register 3

  - Load a copy of Y into register 4

  - Add the value in register 3 to the value in register 4, and direct the result to register 5

  - Store a copy of the value in register 5 in Z

- Note: the above assumes registers 3, 4, and 5 are available for use

Terminology

- *Register spilling*

    – Refers to placing current contents of registers in memory for later recall

    – Occurs when registers needed for other computation

- *Register allocation*

    – Choose which values to keep in registers at any time

    – Programmer or compiler decides

Double Precision

- Refers to value that is twice as large as usual

- Most hardware does not have dedicated registers for double precision computation

- Approach taken: programmer can use a contiguous pair of registers to hold a double precision value

- Example: load a double precision value into registers 3 and 4

Register Banks

- Registers partitioned into disjoint sets called *banks*

- Additional hardware detail

- Optimizes performance

- Complicates programming
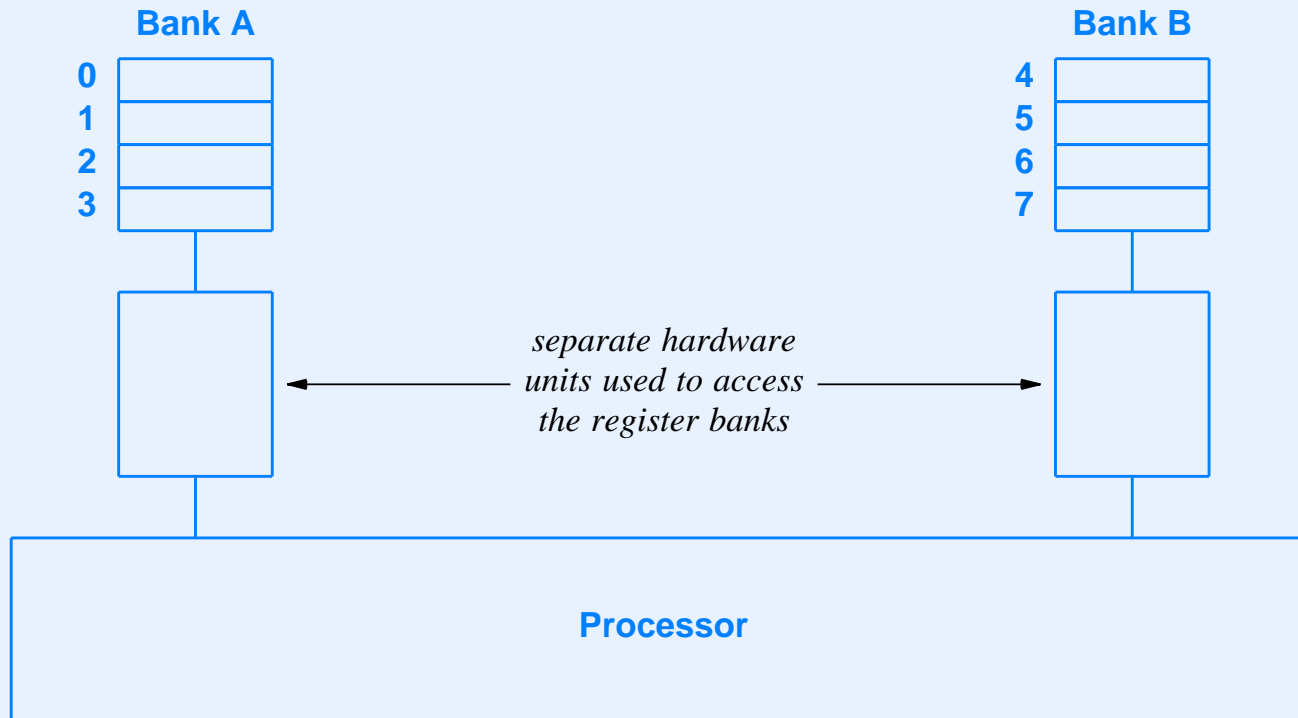
Register Banks

- Registers partitioned into disjoint sets called *banks*

- Additional hardware detail

- Optimizes performance

- Complicates programming

Typical Register Bank Scheme

- Registers divided into two banks

- ALU instruction that takes two operands must have one operand from each bank

- Programmer must enforce separation into banks

- Having two operands from the same bank causes a run-time error

Why Register Banks Are Used

- Parallel hardware facilities for each bank

- Allows both banks to be accessed simultaneously

Consequence For Programmers

- Operands must be assigned to banks

- Even trivial programs cause problems

- Example

$$R \leftarrow X + Y$$
$$S \leftarrow Z - X$$
$$T \leftarrow Y + Z$$

Register Conflicts

- Occur when operands specify same register bank

- May be reported by compiler / assembler

- Programmer must rewrite code or insert extra instruction to copy an operand value to the opposite register bank

Two Types Of Instruction Sets

- CISC: Complex Instruction Set Computer

- RISC: Reduced Instruction Set Computer

CISC Instruction Set

- Many instructions (often hundreds)

- Given instruction can require arbitrary time to compute

- Examples of complex CISC instructions

    – Move graphical item on bitmapped display

    – Copy or clear a region of memory

    – Perform a floating point computation

RISC Instruction Set

- Few instructions (typically 32 or 64)

- Each instruction executes in one clock cycle

- Example: MIPS instruction set

- Omits complex instructions

    - No floating-point instructions

    - No graphics instructions

Summary Of Instruction Sets

A processor is classified as CISC if the instruction set contains instructions that perform complex computations that can require long times; a processor is classified as RISC if it contains a small number of instructions that can each execute in one clock cycle.

Execution Pipeline

- Important part of processor design

- Optimizes performance

- Permits processor to complete more instructions per unit time

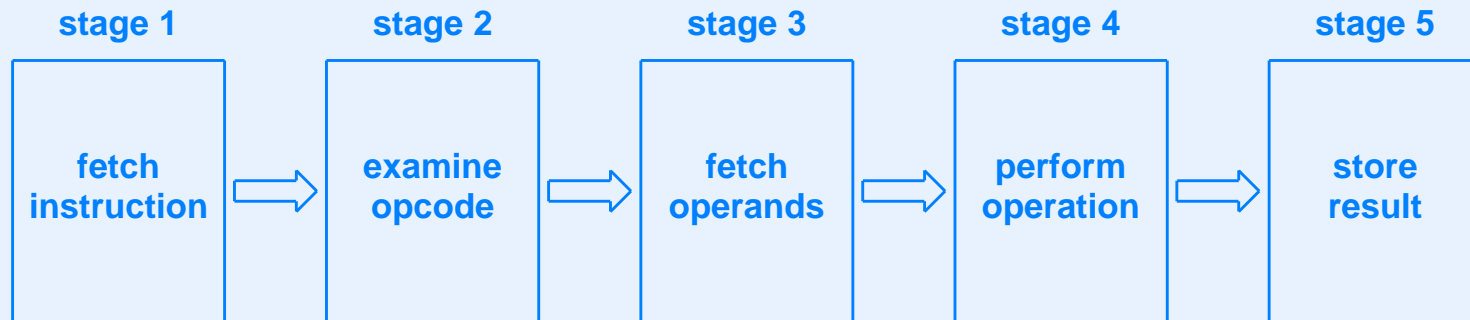- Typically used with RISC instruction set

Basic Steps In A Fetch-Execute Cycle

- Fetch the next instruction

- Examine the opcode to determine how many operands are needed

- Fetch each of the operands (e.g., extract values from registers)

- Perform the operation specified by the opcode

- Store the result in the location specified (e.g., a register)

To Optimize Instruction Cycle

- Build separate hardware block for each step

- Arrange to pass instruction through sequence of hardware blocks

- Allows step K of one instruction to execute while step K−1 of next instruction executes

Illustration Of Execution Pipeline

stage 1	stage 2	stage 3	stage 4	stage 5
fetch instruction	examine opcode	fetch operands	perform operation	store result

- Example pipeline has five stages

- All stages can operate at a given time

Pipeline Speed

- All stages operate in parallel

- Given stage can start to process a new instruction as soon as current instruction finishes

- Effect: N-stage pipeline can operate on N instructions simultaneously

Illustration Of Instructions In A Pipeline

clock	stage 1	stage 2	stage 3	stage 4	stage 5
1	inst. 1	-	-	-	-
2	inst. 2	inst. 1	-	-	-
3	inst. 3	inst. 2	inst. 1	-	-
4	inst. 4	inst. 3	inst. 2	inst. 1	-
5	inst. 5	inst. 4	inst. 3	inst. 2	inst. 1
6	inst. 6	inst. 5	inst. 4	inst. 3	inst. 2
7	inst. 7	inst. 6	inst. 5	inst. 4	inst. 3
8	inst. 8	inst. 7	inst. 6	inst. 5	inst. 4

Time

RISC Processors And Pipelines

Although a RISC processor cannot perform all steps of the fetch-execute cycle in a single clock cycle, an instruction pipeline with parallel hardware provides approximately equivalent performance: once the pipeline is full, one instruction completes on every clock cycle.

Using A Pipeline

- Pipeline is *transparent* to programmer

- Disadvantage: programmer who does not understand pipeline can produce inefficient code

- Reason: hardware automatically *stalls* pipeline if items are not available

Example Of Instruction Stalls

- Assume

  - Need to perform addition and subtraction operations

  - Operands and results in registers *A* through *E*

  - Code is:

    Instruction K: C ← add A B
    Instruction K+1:   D ← subtract E C

- Second instruction stalls to wait for operand *C*

Effect Of Stall On Pipeline

clock	stage 1	stage 2	stage 3	stage 4	stage 5
1	inst. K	inst. K-1	inst. K-2	inst. K-3	inst. K-4
2	inst. K+1	inst. K	inst. K-1	inst. K-2	inst. K-3
3	inst. K+2	inst. K+1	inst. K	inst. K-1	inst. K-2
4	inst. K+3	inst. K+2	(inst. K+1)	inst. K	inst. K-1
5	-	-	(inst. K+1)	-	inst. K
6	-	-	inst. K+1	-	-
7	inst. K+4	inst. K+3	inst. K+2	inst. K+1	-
8	inst. K+5	inst. K+4	inst. K+3	inst. K+2	inst. K+1

Time

- *Bubble* passes through pipeline

Actions That Cause A Pipeline Stall

- Access external storage

- Invoke a coprocessor

- Branch to a new location

- Call a subroutine

Achieving Maximum Speed

- Program must be written to accommodate instruction pipeline

- To minimize stalls

    – Avoid introducing unnecessary branches

    – Delay references to result register(s)

Example Of Avoiding Stalls

C ← add A B	C ← add A B
D ← subtract E C	F ← add G H
F ← add G H	M ← add K L
J ← subtract I F	D ← subtract E C
M ← add K L	J ← subtract I F
P ← subtract M N	P ← subtract M N
(a)	**(b)**

- Stalls eliminated by rearranging (a) to (b)

A Note About Pipelines

Although hardware that uses an instruction pipeline will not run at full speed unless programs are written to accommodate the pipeline, a programmer can choose to ignore pipelining and assume the hardware will automatically increase speed whenever possible.

No-Op Instructions

- Have no effect on

    – Registers

    – Memory

    – Program counter

    – Computation

- Can be inserted to avoid instruction stalls

- Often used by a compiler

Use Of No-Op

- Example

  Instruction K: C $\leftarrow$ add A B

  Instruction L+1: no-op

  Instruction K+2: D $\leftarrow$ subtract E C

- No-op allows time for result from register *C* to be fetched for *subtract* operation

Forwarding

- Hardware optimization to avoid a stall

- Allows ALU to reference result in next instruction

- Example

  Instruction K: C ← add A B

  Instruction K+1: D ← subtract E C

- Forwarding hardware passes result of *add* operation directly to ALU for the next instruction

Types Of Operations

- Operations usually classified into groups

- An example categorization

  - Arithmetic instructions (integer arithmetic)

  - Logical instructions (also called Boolean)

  - Data access and transfer instructions

  - Conditional and unconditional branch instructions

  - Floating point instructions

  - Processor control instructions

Program Counter

- Hardware register

- Used during fetch-execute cycle

- Gives address of next instruction to execute

- Also known as *instruction pointer*

Fetch-Execute Algorithm Details

Assign the program counter an initial program address.
Repeat forever {

   Fetch: access the next step of the program from the
   location given by the program counter.

   Set an internal address register, A, to the address
   beyond the instruction that was just fetched.

   Execute: Perform the step of the program.

   Copy the contents of address register A to the
   program counter.

}

Branches And Fetch Execute

- Absolute branch

    – Typically named *jump*

    – Operand is an address

    – Assigns operand value to internal register *A*

- Relative branch

    – Typically named *br*

    – Operand is a signed value

    – Adds operand to internal register *A*

Subroutine Call

- Jump subroutine (*jsr* instruction)

  – Similar to a *jump*

  – Saves value of internal register *A*

  – Replaces *A* with operand address

- Return from subroutine (*ret* instruction)

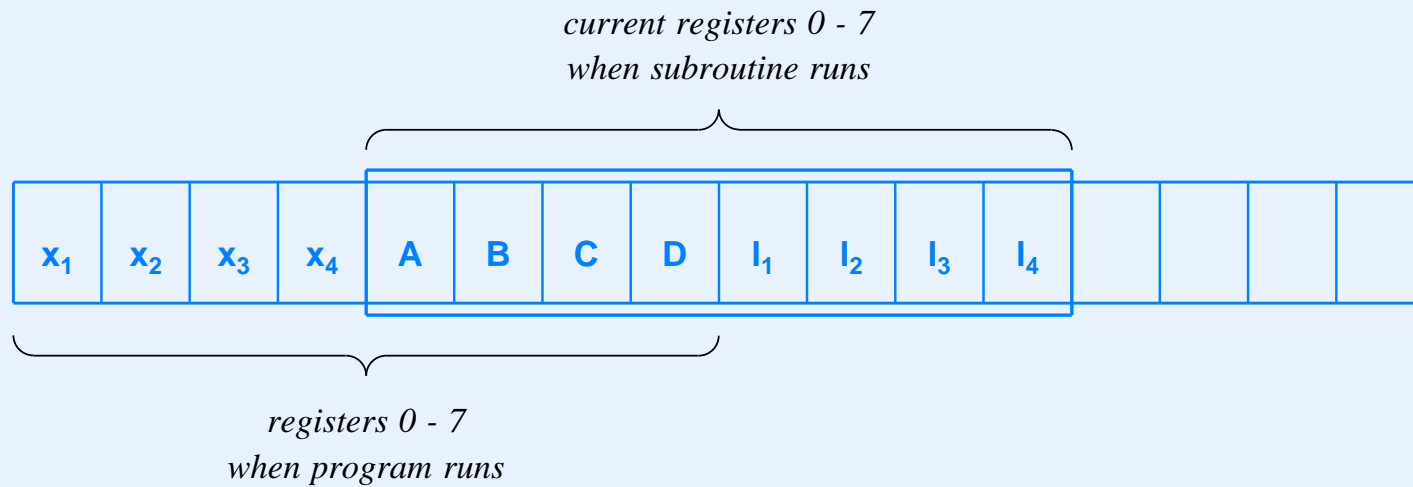 – Retrieves value saved during *jsr*

  – Replaces *A* with saved value

Passing Arguments

- Multiple methods have been used

- Examples

    – Store arguments in memory

    – Store arguments in special-purpose hardware registers

    – Store arguments in general-purpose registers

- Many techniques also used to return result from *function*

Register Window

- Hardware optimization for argument passing

- Processor contains many general-purpose registers

- Only a small subset of registers visible at any time

- Caller places arguments in reserved registers

- During procedure call, register window moves to hide old registers and expose new registers

Illustration Of Register Windows

current registers 0 - 7
when subroutine runs

| x_1 | x_2 | x_3 | x_4 | A | B | C | D | I_1 | I_2 | I_3 | I_4 | | | | |

registers 0 - 7
when program runs

Example Instruction Set

- Known as *MIPS* instruction set

- Early RISC design

- Minimalistic

MIPS Instruction Set (Part 1)

Instruction	Meaning
Arithmetic	
add	integer addition
subtract	integer subtraction
add immediate	integer addition (register + constant)
add unsigned	unsigned integer addition
subtract unsigned	unsigned integer subtraction
add immediate unsigned	unsigned addition with a constant
move from coprocessor	access coprocessor register
multiply	integer multiplication
multiply unsigned	unsigned integer multiplication
divide	integer division
divide unsigned	unsigned integer division
move from Hi	access high-order register
move from Lo	access low-order register
Logical (Boolean)	
and	logical *and* (two registers)
or	logical *or* (two registers)
and immediate	*and* of register and constant
or immediate	*or* of register and constant
shift left logical	Shift register left N bits
shift right logical	Shift register right N bits

MIPS Instruction Set (Part 2)

Instruction	Meaning
Data Transfer	
load word	load register from memory
store word	store register into memory
load upper immediate	place constant in upper sixteen bits of register
move from coproc. register	obtain a value from a coprocessor
Conditional Branch	
branch equal	branch if two registers equal
branch not equal	branch if two registers unequal
set on less than	compare two registers
set less than immediate	compare register and constant
set less than unsigned	compare unsigned registers
set less than immediate	compare unsigned register and constant
Unconditional Branch	
jump	go to target address
jump register	go to address in register
jump and link	procedure call

MIPS Floating Point Instructions

Instruction	Meaning

Arithmetic

FP add	floating point addition
FP subtract	floating point subtraction
FP multiply	floating point multiplication
FP divide	floating point division
FP add double	double-precision addition
FP subtract double	double-precision subtraction
FP multiply double	double-precision multiplication
FP divide double	double-precision division

Data Transfer

load word coprocessor	load value into FP register
store word coprocessor	store FP register to memory

Conditional Branch

branch FP true	branch if FP condition is true
branch FP false	branch if FP condition is false
FP compare single	compare two FP registers
FP compare double	compare two double precision values

Aesthetic Aspects Of Instruction Sets

- Elegance

  - Balanced

  - No frivolous or useless instructions

- Orthogonality

  - No unnecessary duplication

  - No overlap among instructions

Principle Of Orthogonality

The principle of orthogonality specifies that each instruction should perform a unique task without duplicating or overlapping the functionality of other instructions.

Condition Codes

- Extra hardware bits (not part of general-purpose registers)

- Set by ALU on each instruction

- Indicate

  – Overflow

  – Underflow

  – Other exceptions

- Tested in *conditional branch* instruction

Example Of Condition Code

```
cmp     r4, r5      # compare regs. 4 & 5, and set condition code

be      lab1        # branch to lab1 if cond. code specifies equal

mov     r3, 0       # place a zero in register 3

lab1:   …program continues at this point
```

- Above code places a zero in register 3 if register 4 is not equal to register 5

Questions?

VI

Operand Addressing
And
Instruction Representation

Number Of Operands Per Instruction

- Four basic architectural types

  - 0-address

  - 1-address

  - 2-address

  - 3-address

0-Address Architecture

- No explicit operands in the instruction

- Program

  - Pushes operands onto stack in memory

  - Executes instruction

- Instruction execution

  - Removes top N items from stack

  - Leaves result on top of stack

Illustration Of 0-Address Instructions

- Example: add 7 to variable *X* in memory

  push X
  push 7
  add
  pop X

- *Add* instruction removes two arguments from stack and leaves result on stack

1-Address Architecture

- Analogous to a calculator

- One explicit operand per instruction

- Processor has special register known as *accumulator*

  - Holds second argment for each instruction

  - Used to store result of instruction

Illustration Of 1-Address Instructions

- Example: add 7 to variable *X* in memory

<div align="center">

load X

add 7

store X

</div>

- *Load* places value in accumulator from memory

- *Store* places accumulator value in memory

- *Add* increases value in accumulator

2-Address Architecture

- Two explicit operands per instruction

- Result overwrites one of the operands

- Operands known as *source* and *destination*

- Works well for instructions such as memory copy

Illustration Of 2-Address Instructions

- Example: add 7 to variable *X* in memory

<p style="text-align:center; color:blue">add 7, X</p>

- Computes $X \leftarrow X + 7$

3-Address Architecture

- Three explicit operands per instruction

- Operands specify *source*, *destination*, and *result*

Illustration Of 3-Address Instructions

- Example: add variable *Y* to variable *X* and place result in variable *Z*

add X, Y, Z

Operand Types

- Source operand can specify

  - A signed constant

  - An unsigned constant

  - The contents of a register

  - A value in memory

- Destination operand can specify

  - A single register

  - A pair of contiguous registers

  - A memory location

Operand Types

- Each operand has a *type*

- An operand that specifies a constant is known as *immediate* value

- Memory references are usually much more expensive than immediate values or register accesses

Von Neumann Bottleneck

On a computer that follows the Von Neumann architecture, the time spent performing memory accesses can limit the overall performance. Architects use the term Von Neumann bottleneck *to characterize the situation, and avoid the bottleneck with techniques such as restricting most operands to registers.*

Operand Encoding

- Implicit type encoding

  - For given opcode, the type of each operand is fixed

  - More opcodes required

  - Example: opcode is *add_signed_immediate_to_register*

- Explicit type encoding

  - Operand specifies type and value

  - Fewer opcodes required

  - Example: opcode is *add*, operands specify *register* and *immediate*

Example Of Implicit Encoding

Opcode	Operands	Meaning
Add register	R1 R2	R1 ← R1 + R2
Add immediate signed	R1 I	R1 ← R1 + I
Add immediate unsigned	R1 UI	R1 ← R1 + UI
Add memory	R1 M	R1 ← R1 + memory[M]

Examples Of Explicit Encoding

- *Add* operation with registers 1 and 2 as operands

opcode	operand 1		operand 2	
add	register	1	register	2

- *Add* operation with register and signed immediate value of −93 as operands

opcode	operand 1		operand 2	
add	register	1	signed integer	−93
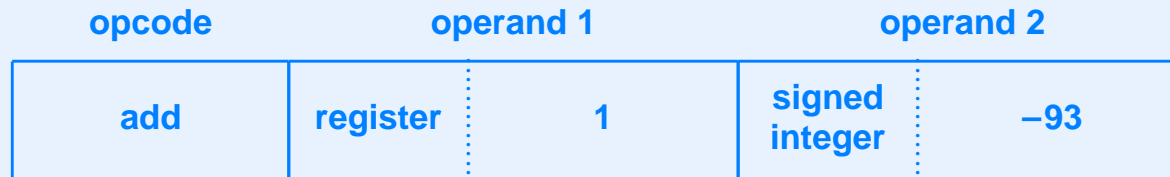
Operands That Combine Multiple Types

- Operand contains multiple items

- Processor computes operand value from individual items

- Typical computation: sum

- Example

  - A *register-offset* operand specifies a register and an immediate value

  - Processor adds immediate value to contents of register and uses result as operand

Illustration Of Register-Offset

opcode	operand 1			operand 2		
add	register-offset	2	−17	register-offset	4	76

- First operand consists of value in register 2 minus 17

- Second operand consists of value in register 4 plus 76

Operand Tradeoffs

- No single style of operands optimal for all purposes

- Tradeoffs among

  - Ease of programming

  - Fewer instructions

  - Smaller instructions

  - Larger range of immediate values

  - Faster operand fetch and decode

  - Decreased hardware size

Operands In Memory And Indirect Reference

- Operand can specify

  – Value in memory (*memory reference*)

  – Location in memory that contains the address of the operand (*indirect reference*)

- Note: accessing memory is relatively expensive

Types Of Indirection

- Indirection through a register

    – Operand specifies register number, *R*

    – Obtain A, the current value from register *R*

    – Interpret A as a memory address, and fetch the operand from memory location A

- Indirection through a memory location

    – Operand specifies memory address, A

    – Obtain M, the value in memory location A

    – Interpret M as a memory address, and fetch the operand from memory location M

Illustration Of Operand Addressing Modes
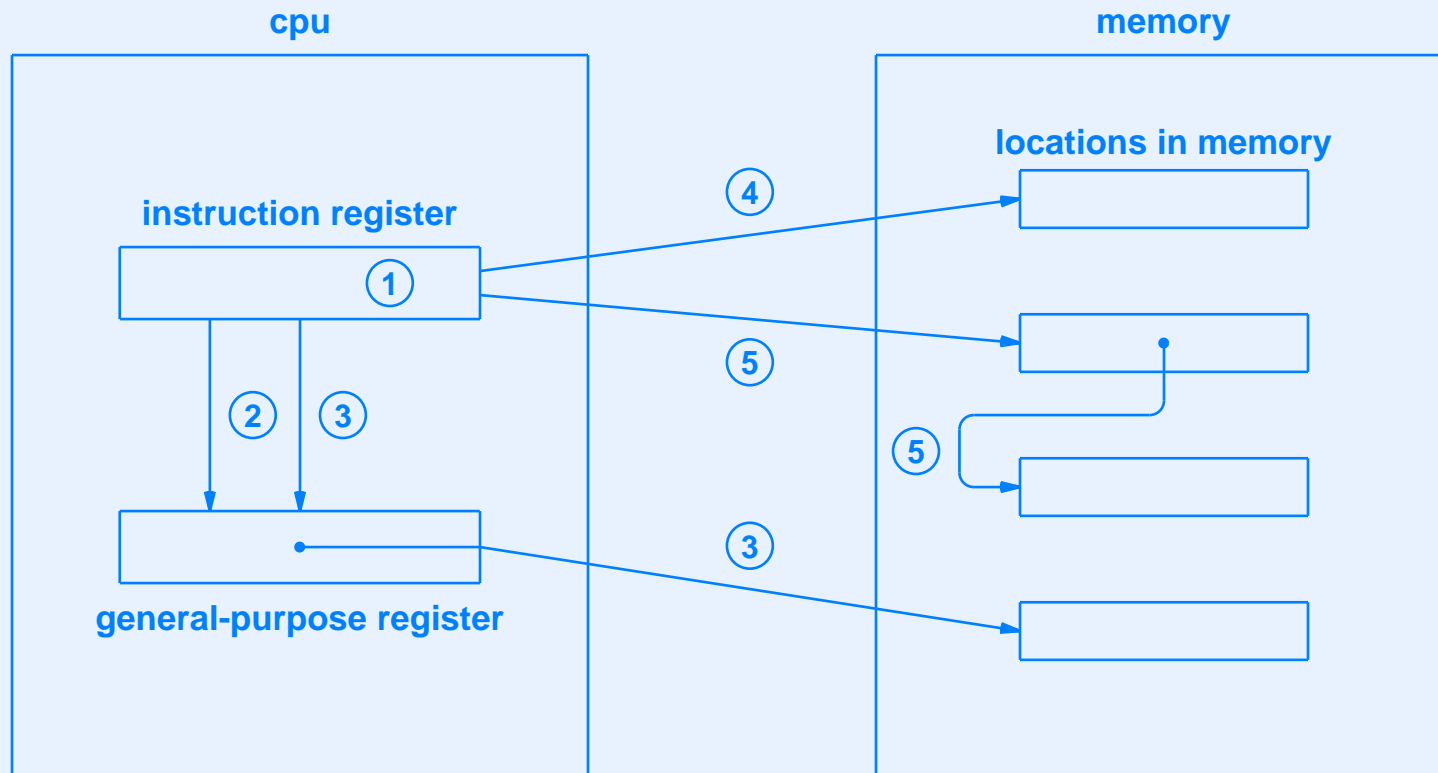
1 Immediate value (in the instruction)

2 Direct register reference

3 Indirect through a register

4 Direct memory reference

5 Indirect memory reference

Summary

- Architect chooses the number and types of operands for each instruction

- Possibilities include

  – Immediate (constant value)

  – Contents of register

  – Value in memory

  – Indirect reference to memory

Summary
(continued)

- Type of operand can be encoded

  - Implicitly (opcode determines types of operands)

  - Explicitly (extra bits in each operand specify the type)

- Many variations exist; each represents a tradeoff

Questions?

VII

CPUs:
Microcode, Protection,
And
Processor Modes

Evolution Of Computers

- Early systems

  - Single *Central Processing Unit (CPU)* controlled entire computer

  - Responsible for all I/O as well as computation

- Modern computer

  - Decentralized architecture

  - Each I/O device (e.g., a disk) contains processor

  - CPU performs computation and controls other processors

CPU Complexity

- Designed for wide variety of control and processing tasks

- Many special-purpose subunits for speed

- Example: one multi-core Intel processor contains more than a billion transistors

CPU Characteristics

- Completely general

- Can perform control functions as well as basic computation

- Offers multiple levels of protection and privilege

- Provides mechanism for hardware priorities

- Handles large volumes of data

- Uses parallelism to achieve high speed

Modes Of Execution

- CPU hardware has several possible *modes*

- At any time, CPU operates in one mode

- Mode dictates

  - Instructions that are valid

  - Regions of memory that can be accessed

  - Amount of privilege

  - Backward compatibility with earlier models

- CPU behavior varies widely among modes

An Observation About Modes

A CPU uses an execution mode to determine the current operational characteristics. In some CPUs, the characteristics of modes differ so widely that we think of the CPU as having separate hardware subsystems and the mode as determining which piece of hardware is used at the current time.

Changing Modes

- Automatic mode change

    - Initiated by hardware

    - Programmer can specify code for new mode

- Manual mode change

    - Program makes explicit request

    - Typically used when program calls the operating system

Changing Modes
(continued)

- Mechanisms vary among architectures

- Possibilities

  – Invoke a special instruction to change mode

  – Assign a value to a mode register

  – Mode change is a side-effect of another instruction

Privilege And Protection

Privilege Level

- Determines what resources program can use

- Linked to mode

- Basic scheme: two levels, one for operating system and one for applications

- Advanced scheme: multiple levels

Illustration Of Basic Privilege Scheme
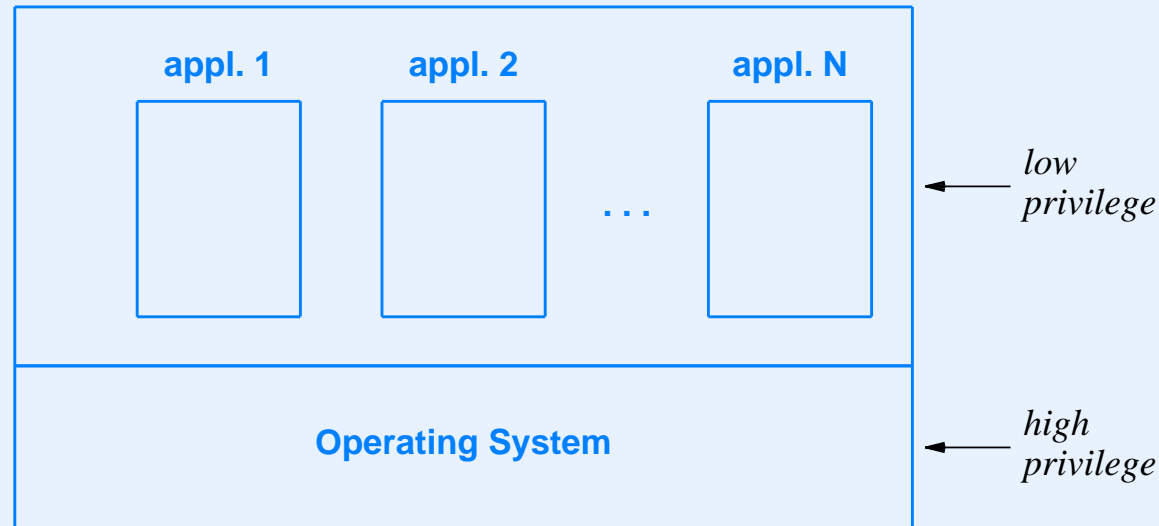
- Application-level privilege available to arbitrary program

- System-level privilege restricted to operating system

Microcode

Microcoded Instructions

- Hardware technique used to build complex processors

- Employs two levels of processor hardware

    - Microcontroller (*microprocessor*) provides basic operations

    - Macro instruction set built on microinstructions

- Key concept: it is easier to construct complex processors by writing programs then by building hardware from scratch

Illustration Of Microcoded Instruction Set
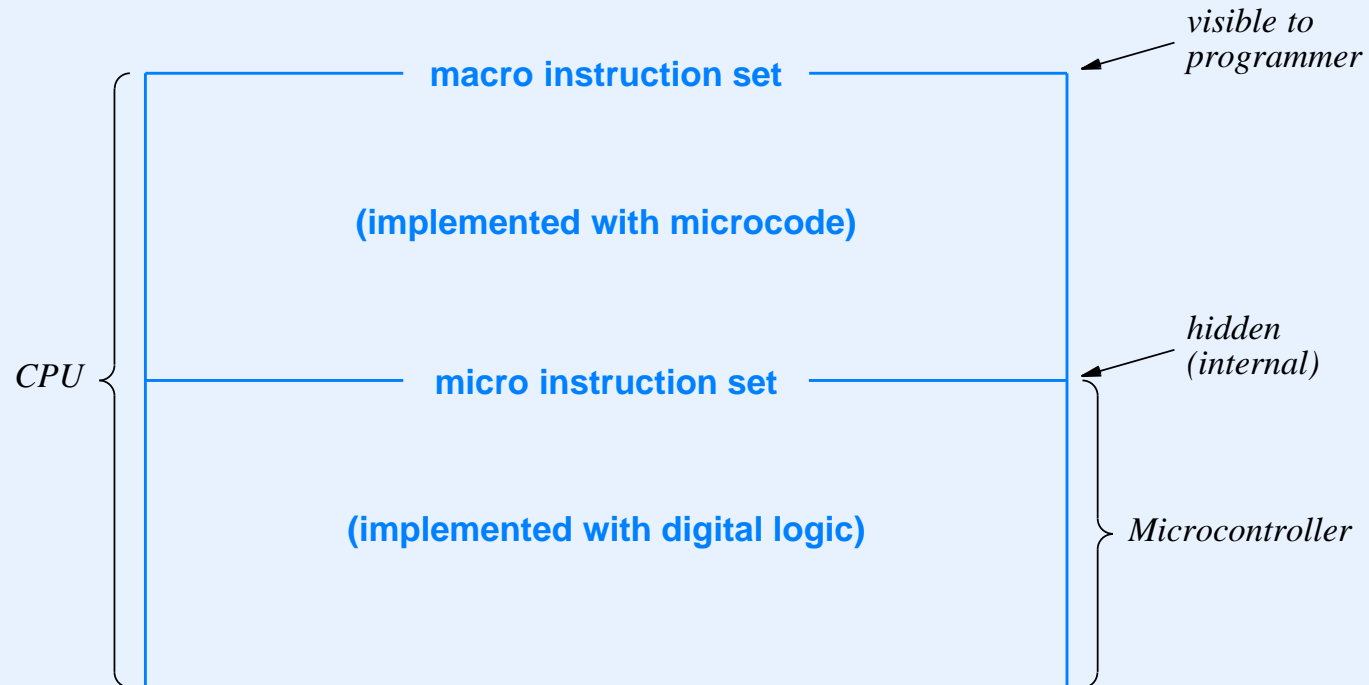
Implementation Of Microcoded Instructions

- Microcontroller

  – Lowest level of processor

  – Implemented with digital logic

  – Offers basic instructions

- Macro instructions

  – Implemented as microcode subroutines

  – Can be entirely different than micro instructions

Data And Register Sizes

- Data size used by micro instructions can differ from size used by macro instructions

- Example

  - 16-bit hardware used for micro instructions

  - 32-bit hardware used for macro instructions

Example Of Microcoded Arithmetic

- Assume

  - Macro registers

    * Each 32 bits wide

    * Named R0, R1, ...

  - Micro registers

    * Each 16 bits wide

    * Named r0, r1, ...

- Devise microcode to add values from R5 and R6

Example Of Microcoded Arithmetic
(continued)

add32:

move low-order 16 bits from R5 into r2

move low-order 16 bits from R6 into r3

add r2 and r3, placing result in r1

save value of the carry indicator

move high-order 16 bits from R5 into r2

move high-order 16 bits from R6 into r3

add r2 and r3, placing result in r0

copy the value in r0 to r2

add r2 and the carry bit, placing the result in r0

check for overflow and set the condition code

move the thirty-two bit result from r0 and r1
   to the desired destination

Microcode Variations

- Restricted or full scope

    - Special-purpose instructions only (e.g., extensions to normal instruction set)

    - All instructions

- Partial or complete use

    - Entire fetch-execute cycle

    - Instruction fetch and decode

    - Opcode processing

    - Operand decode and fetch

Why Use Microcode Instead Of Circuits?

- Higher level of abstraction

- Easier to build and less error prone

- Easier to change

    - Easy upgrade to next version of chip

    - Can allow field upgrade

Disadvantages Of Microcode

- More overhead

- Macro instruction performance depends on micro instruction set

- Microcontroller hardware must run at extremely high clock rate to accommodate multiple micro instructions per macro instruction

Visibility To Programmers

- Fixed microcode

    – Approach used by most CPUs

    – Microcode only visible to CPU designer

- Alterable microcode

    – Microcode loaded dynamically

    – May be restricted to extensions (creating new macro instructions)

    – User software written to use new instructions

    – Known as a *reconfigurable CPU*

Reconfigurable CPU

Some CPUs provide a mechanism that allows microcode to be rewritten. The motivation for allowing such change arises from the desire for flexibility and optimization: the CPU's owner can create a macro instruction set that is optimized for a specific task.

In Practice

- Writing microcode is tedious and time-consuming

- Results are difficult to test

- Performance of microcode can be much worse than performance of dedicated hardware

- Result: reconfigurable CPUs have not enjoyed much success

Two Fundamental Types Of Microcode

- What programming paradigm is used for microcode?

- Two fundamental types

    – Vertical

    – Horizontal

Vertical Microcode

- Microcontroller similar to standard processor

- Vertical microcode similar to conventional assembly language

- Typically performs one operation at a time

- Has access to all facilities macro instruction set uses

  – ALU

  – General-purpose registers

  – Memory

  – I/O buses

Example Of Vertical Microcode

- Macro instruction set is CISC

- Microcontroller is fast RISC processor

- Programmer writes microcode for each macro instruction

- Hardware decodes macro instruction and invokes correct microcode routine

Advantages And Disadvantages
Of Vertical Microcode

- Easy to read

- Programmers are comfortable using it

- Unattractive to hardware designers because higher clock rates needed

- Generally has low performance (many micro instructions needed for each macro instruction)

Horizontal Microcode

- Alternative to vertical microcode

- Exploits parallelism in underlying hardware

- Controls functional units and data movement

- Extremely difficult to program

The Important Tradeoff With Horizontal Microcode

Horizontal microcode allows the hardware to run faster, but is more difficult to program.

Horizontal Microcode Paradigm

- Each instruction controls a set of hardware units

- Instruction specifies

    – Transfer of data

    – Which hardware units operate

Horizontal Microcode Example

- Consider the internal structure of a CPU

- Data can only move along specific paths between functional units

- Example:
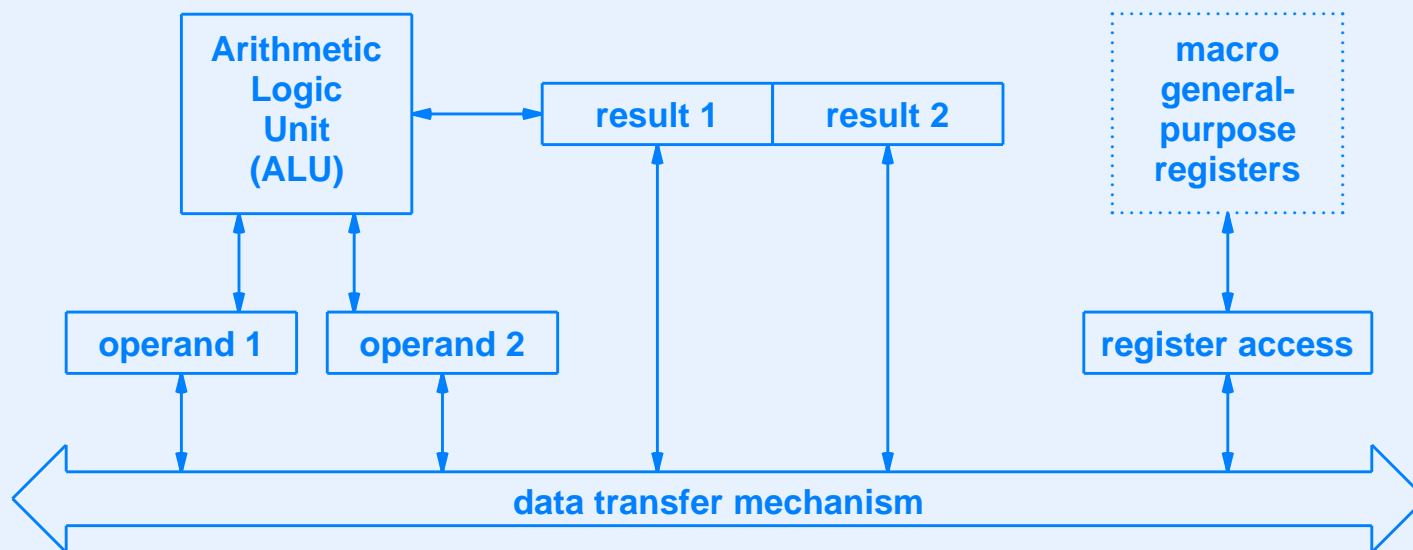
Example Hardware Control Commands

Unit	Command	Meaning
ALU	0 0 0	No operation
	0 0 1	Add
	0 1 0	Subtract
	0 1 1	Multiply
	1 0 0	Divide
	1 0 1	Left shift
	1 1 0	Right shift
	1 1 1	Continue previous operation
operand 1 or 2	0	No operation
	1	Load value from data transfer mechanism
result 1 or 2	0	No operation
	1	Send value to data transfer mechanism
register interface	0 0 x x x x	No operation
	0 1 x x x x	Move register xxxx to data transfer mechanism
	1 0 x x x x	Move data transfer mechanism to register xxxx
	1 1 x x x x	No operation

Microcode Instruction Format For Example

ALU   Oper. 1   Oper. 2   Res. 1   Res. 2   Register interface

x   x   x   x   x   x   x   x   x   x   x   x   x
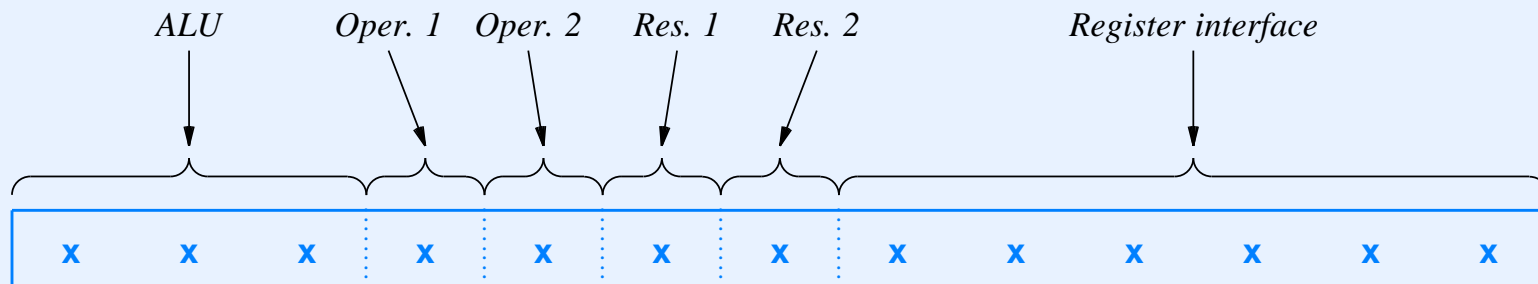
- Diagram shows how instruction is interpreted

- Bit fields in instruction encode hardware control commands

Example Horizontal Microcode Steps

- Move the value from register 4 to the hardware unit for operand 1

- Move the value from register 13 to the hardware unit for operand 2

- Arrange for the ALU to perform addition

- Move the value from the hardware unit for result 2 (the low-order bits of the result) to register 4

Example Horizontal Microcode
(In Binary)

Instr.	ALU			OP$_1$	OP$_2$	RES$_1$	RES$_2$	REG. INTERFACE					
1	0	0	0	1	0	0	0	0	1	0	1	0	0
2	0	0	0	0	1	0	0	0	1	1	1	0	1
3	0	0	1	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	1	1	0	0	1	0	0

- Note: code is unlike a conventional program

Horizontal Microcode And Timing

- Each microcode instruction takes one cycle

- Functional unit may require more than one cycle to complete an operation

- Programmer must accommodate hardware timing or errors can result

- To wait for functional unit, insert microcode instructions that continue the operation

Example Of Continuing An Operation

ALU			OP$_1$	OP$_2$	RES$_1$	RES$_2$	REG. INTERFACE					
1	1	1	0	0	0	0	0	0	0	0	0	0
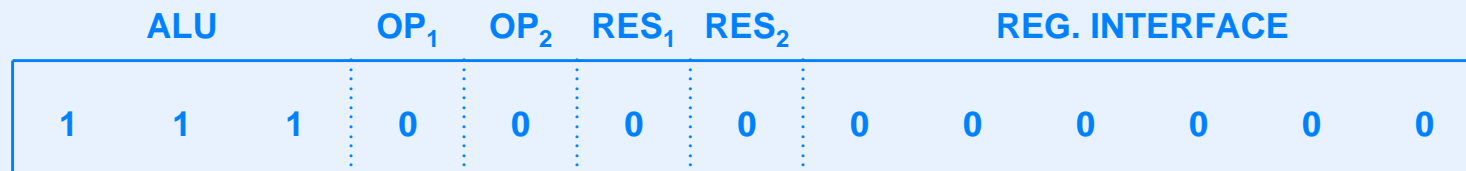
- Assume ALU operation *1 1 1* acts as a delay to continue the previous operation

Example Of Parallel Execution

ALU			OP$_1$	OP$_2$	RES$_1$	RES$_2$	REG. INTERFACE				
1	1	1	1	0	0	0	0	1	0	1	1	1

- A single microcode instruction can continue the ALU operation and also load the value from register 7 into operand unit 1.

Horizontal Microcode And Parallel Execution

Because an instruction contains separate fields that each correspond to one hardware unit, horizontal microcode makes it easy to specify simultaneous, parallel operation of multiple hardware units.

Intelligent Microcontroller

- Schedules instructions by assigning work to functional units

- Handles operations in parallel

- Performs *branch prediction* by beginning to execute both paths of a branch

- Constrains results so instructions have sequential semantics

  - Keeps results separate

  - Decides which path to use when branch direction finally known

The Important Concept Of Branch Prediction

A CPU that offers parallel instruction execution can handle conditional branches by precomputing values on both branches and choosing which values to use at a later time when the computation of the branch condition completes.

Taming Parallel Execution Units

- Parallel hardware runs wild

- CPU must preserve sequential execution semantics (as expected by programmer)

- Mechanisms used

    - Scoreboard

    - Re-Order Buffer (ROB)

- Note: when results computed from two paths, CPU eventually discards results that are not needed

Summary

- CPU offers modes of execution that determine protection and privilege

- Complex CPU usually implemented with microcode

- Vertical microcode uses conventional instruction set

- Horizontal microcode uses unconventional instructions

Summary
(continued)

- Each horizontal microcode instruction controls underlying hardware units

- Horizontal microcode offers parallelism

- Most complex CPUs have mechanism to schedule instructions on parallel execution units

- Scoreboard and Re-Order Buffer used to maintain sequential semantics

Questions?

VIII

Assembly Languages
And
Programming Paradigm

The Two Types Of Programming Languages

- Low-level (close to hardware)

- High-level (abstracted away from hardware)

Characteristics Of High-Level Language

- One-to-many translation

- Hardware independence

- Application orientation

- General-purpose

- Powerful abstractions

Characteristics Of Low-Level Language

- One-to-one translation

- Hardware dependence

- Systems programming orientation

- Special-purpose

- Few abstractions

A Note About Abstractions

- A low-level language forces a programmer to construct abstractions from low-level mechanisms

- Computer scientist Alan Perlis once said that a programming language is low-level if programming requires attention to irrelevant details

- Perlis' point: because most applications do not need direct control of hardware, a low-level language increases programming complexity without providing benefits

Terminology

- *Assembly language*

    – Refers to a special type of low-level language

    – Specific to given processor

- *Assembler*

    – Refers to software that translates assembly language into binary code

    – Analogous to compiler

An Important Concept

Because an assembly language is a low-level language that incorporates specific characteristics of a processor, such as the instruction set, operand addressing, and registers, one assembly language exists for each type of processor.

- All assembly languages share the same general structure

- A programmer who understands one assembly language can learn another quickly

Our Approach

- We will discuss general concepts in class

- You will use a specific assembly language in lab

Assembly Statement Format

- General format is:

$$\text{label:} \quad \text{opcode} \quad \text{operand}_1, \quad \text{operand}_2, \quad ...$$

- Label is optional

- Opcode and operands are processor specific

Opcode Names

- Specific to each assembly language

- Most assembly languages use short mnemonics

- Examples

  - *ld* instead of *load_value_into_register*

  - *jsr* instead of *jump_to_subroutine*

Comment Syntax

- Typically

  - Character reserved to start a comment

  - Comment extends to end of line

- Examples of comment characters

  - Pound sign (#)

  - Semicolon (;)

Commenting Conventions

- Block comment to explain overall purpose of large section of code

- One comment per line explaining purpose of the instruction

Block Comment Example

```
###################################################################
#                                                                 #
#     Search linked list of free memory blocks to find a block    #
#     of size N bytes or greater.  Pointer to list must be in      #
#     register 3 and N must be in register 4.  The code also       #
#     destroys the contents of register 5, which is used to        #
#     walk the list.                                               #
#                                                                 #
###################################################################
```

Per-Line Comment Example

```
         ld        r5, r3     # load the address of list into r5
loop_1:  cmp       r5, r0     # test to see if at end of list
         bz        notfnd     # if reached end of list go to notfnd
```

- It is typical to find a comment on each line of an assembly language program

Operand Order

- Annoying fact: assembly languages differ on operand order

- Example

    - Instruction to load a register

    - Possible orders are:

```
ld      r5, r3    # load the address of list into r5

ld      r3, r5    # load the address of list into r5
```

- Note: in one historic case, two assembly languages for the same processor used opposite orders for operands!

Remembering Operand Order

- When programming assembly language that uses

  (*source, destination*)

  remember that we read left-to-right

- When programming assembly language that uses

  (*destination, source*),

  remember that the operands are in the same order as an assignment statement

Names For General-Purpose Registers

- Registers used heavily

- Most assembly languages use short names for registers

- Typical format is letter *r* followed by a number

- Syntax that has been used in various assembly languages

    – reg10

    – r10

    – R10

    – $10

Symbolic Definitions

- Some assemblers use long names, but permit a programmer to define abbreviations

- Example definitions

```
#
#  Define register names used in the program
#
r1      register 1          # define name r1 to be register 1
r2      register 2          #   and so on for r2, r3, and r4
r3      register 3
r4      register 4
```

Using Meaningful Names

- Symbolic definition also allows meaningful names

- Example: registers used for a linked list

```
#
#  Define register names for a linked list program
#
listhd    register 6      # holds starting address of list
listptr   register 7      # moves along the list
```

Denoting Operands

- Assembly language provides a way to code each possible operand type (e.g., immediate, register, memory reference, indirect memory reference)

- Typically, compact syntax is used

- Example

```
mov     r2, r1      # copy contents of reg. 1 into reg. 2
mov     r2, (r1)    # treat r1 as a pointer to memory and
                    # copy from the mem. location to reg. 2
```

Assembly Language And Idioms

- No high-level abstractions

- Programmer writes sequence of code instead

- Best if programmer follows idioms

Assembly Language For Conditional Execution

if (condition) {   code to test condition and
    body   set condition code
}   branch not true to label
next statement   code to perform body
                                    label: code for next statement

Assembly Language For If-Then Else

if (condition) {

    then_part

} $else$ {

    else_part

}

next statement

                code to test condition and

                    set condition code

                branch not true to label1

                code to perform then_part

                branch to label2

label1: code for else_part

label2: code for next statement

Assembly Language For Definite Iteration

for (i=0; i<10; i++) {
    body
}
next statement

                  set r4 to zero
label1: compare r4 to 10
                  branch to label2 if >=
                  code to perform body
                  increment r4
                  branch to label1
label2: code for next statement

Assembly Language For Indefinite Iteration

while (condition) {

    body

}

next statement

label1: code to compute condition

        branch to label2 if not true

        code to perform body

        branch to label1

label2: code for next statement

Assembly Language For Procedure Call

x ( ) {   x: code for body of x

    body of procedure x   ret

}

x ( );   jsr x

other statement;   code for other statement

x ( );   jsr x

next statement   code for next statement

Argument Passing

- Hardware possibilities

    - Stack in memory used for arguments

    - Register windows used to pass arguments

    - Special-purpose argument registers used

- Assembly language depends on hardware

Consequence For Programmers

No single argument passing paradigm is used in assembly languages because a variety of hardware mechanisms exist for argument passing. In addition, programmers sometimes use alternatives to the basic mechanism to optimize performance (e.g., passing values in registers).

Example Procedure Invocation
(Using Registers 1 - 8)

x (a, b) {

    body of function x

}

x (-4, 17);

other statement;

x (71, 27);

next statement

x: code for body of x that assumes

      reg. 1 contains parameter *a*

      and reg. 2 contains *b*

  ret

load -4 into register 1

load 17 into register 2

jsr x

code for other statement

load 71 into register 1

load 27 into register 2

jsr x

code for next statement

Function Invocation

- Like procedure invocation

- Also returns result

- Hardware exists that returns value

  – On a stack in memory

  – In a special-purpose register

  – In a general-purpose register

Interaction With High-Level Language

- Assembly language program can call procedure written in high-level language (e.g., to avoid writing in assembly language)

- High-level language program can call procedure written in assembly language

  - When higher speed is needed

  - When access to special-purpose hardware is required

- Assembly language coded to follow *calling conventions* of high-level language

In Practice

Because writing application programs in assembly language is difficult, assembly language is reserved for situations where a high-level language has insufficient functionality or results in poor performance.

Declaration Of Variable In Assembly Language

- Most assembly languages have no *declarations* or *typing*

- Programmer can reserve blocks of storage (for variables) and use labels

- Typical directives

  – .word

  – .byte or .char

  – .long

Examples Of Equivalent Declarations

```
int     x, y, z;          x:    .long
                          y:    .long
                          z:    .long
short   w, q;             w:    .word
                          q:    .word
statement                       code for statement
```

Specifying Initial Values

- Usually allowed as arguments to directives

- Example to declare 16-bit storage with initial value 949:

```
x:      .word    949
```

Assembler

- Software component

- Accepts assembly language program as input

- Produces binary form of program as output

- Uses *two-pass* algorithm

Difference Between Assembler And Compiler

Although both a compiler and an assembler translate a source program into equivalent binary code, a compiler has more freedom to choose which values are kept in registers, the instructions used to implement each statement, and the allocation of variables to memory. An assembler merely provides a one-to-one translation of each statement in the source program to the equivalent binary form.

What An Assembler Provides

- Statements are 1-to-1 with hardware instructions

- Assembler

  - Computes relative location for each label

  - Fills in branch offsets automatically

- Consequence: programmer can insert or delete statements without recomputing offsets manually

Example Of Code Offsets

```
     locations                      assembly code
_____

0x00   -   0x03        x:          .word
0x04   -   0x07        label1:  cmp     r1, r2
0x08   -   0X0B                    bne     label2
0X0C   -   0x0F                    jsr     label3
0x10   -   0x13        label2:  load    r3, 0
0x14   -   0x17                    br      label4
0x18   -   0x1B        label3:  add     r5, 1
0X1C   -   0x1F                    ret
0x20   -   0x23        label4:  load    r1, 1
0x24   -   0x27                    ret
```

General Concept

Conceptually, an assembler makes two passes over an assembly language program. During the first pass, the assembler assigns a location to each statement. During the second pass, the assembler uses the assigned locations to generate code.

Assembly Language Macros

- Syntactic substitution

- Parameterized for flexibility

- Programmer supplies *macro definitions*

- Code contains *macro invocations*

- Assembler handles *macro expansion* in extra pass

- Known as *macro assembly language*

Macro Syntax

- Varies among assembly languages

- Typical definition bracketed by keywords

- Example keywords

  – *macro*

  – *endmacro*

- Typical invocation uses macro name

Example Of Macro Definition

```
macro   addmem(a, b, c)
load    r1, a     # load 1st arg into register 1
load    r2, b     # load 2nd arg into register 2
add     r1, r2    # add register 2 to register 1
store   r3, c     # store the result in 3rd arg
endmacro
```

- Invocation has arguments that correspond to parameters *a*, *b*, and *c*

Example Of Macro Expansion

```
#
# note: code below results from addmem(xxx, YY, zqz)
#
load    r1, xxx  # load 1st arg into register 1
load    r2, YY   # load 2nd arg into register 2
add     r1, r2   # add register 2 to register 1
store   r3, zqz  # store the result in 3rd arg
```

Programming With Macros

- Many assembly languages use syntactic substitution

    - Parameters treated as string of characters

    - Arbitrary text permitted

    - No error checking performed

- Consequences for programmers

    - Macro can generate invalid code

    - May be difficult to debug

Example Of Illegal Code That Results From A Macro Expansion

```
#
# note: code below results from addmem(1+, %*J , +)
#
load    r1, 1+    # load 1st arg into register 1
load    r2, %*J   # load 2nd arg into register 2
add     r1, r2    # add register 2 to register 1
store   r3, +     # store the result in 3rd arg
endmacro
```

- Assembler substitutes macro arguments literally

- Error messages refer to expanded code, not macro definition

Summary

- Assembly language is low-level and incorporates details of a specific processor

- Many assembly languages exist, one per processor

- Each assembly language statement corresponds to one machine instruction

- Same basic programming paradigm used in most assembly languages

- Programmers must code assembly language equivalents of abstractions such as

  – Conditional execution

  – Definite and indefinite iteration

  – Procedure call

Summary
(continued)

- Assembler translates assembly language program into binary code

- Assembler uses two-pass processing

  - First pass assigns relative locations

  - Second pass generates code

- Some assemblers have additional pass to expand macros

Questions?

IX

Memory And Storage

Key Aspects Of Memory

- Technology

  – Type of underlying hardware

  – Differ in cost, persistence, performance

  – Many variants available

- Organization

  – How underlying hardware is used to build memory system

  – Directly visible to programmer

Memory Characteristics

- Volatile or nonvolatile

- Random or sequential access

- Read-write or read-only

- Primary or secondary

Memory Volatility

- Volatile memory

  - Contents disappear when power is removed

  - Least expensive

- Nonvolatile memory

  - Contents remain without power

  - More expensive than volatile memory

  - May have slower access times

  - One possibility: "cheat" by using a battery to maintain contents

Memory Access Paradigm

- Random access

  - Typical for most applications

- Sequential access

  - Special purpose hardware

  - Known as *FIFO* (*First-In-First-Out*)

Permanence Of Values

- ROM (Read Only Memory)

    – Values can be read, but not changed

    – Useful for firmware

- PROM (Programmable Read Only Memory)

    – Contents can be altered, but doing so is time-consuming

    – Change may involve removal from a circuit and exposure to ultraviolet light

Permanence Of Values
(continued)

- EEPROM

    – Form of PROM that can be changed while installed

    – Variants such as *Flash ROM* used in digital cameras

Primary And Secondary Memory

- Broad classification of memory technologies

- Terminology is qualitative

Traditional Terminology

- Primary memory

    – Highest speed

    – Most expensive, therefore smallest

    – Typically solid state technology

- Secondary memory

    – Lower speed

    – Less expensive, therefore can be larger

    – Typically magnetic media and electromechanical drive mechanism

In Practice

- Distinction between primary and secondary storage blurred

- Solid state technology replacing electromechanical technology

- Examples

    - Memory cards used in digital cameras

    - Solid-state hard drives used in laptop computers

Memory Hierarchy

- Key concept to memory design

- Related to definitions of *primary/secondary* memory

- Arise as tradeoff

    - Highest performance memory costs the most

    - Architect chooses set of memories to satisfy both performance and cost constraints

Memory Hierarchy
(continued)

- Small amount of memory has highest performance

- Slightly larger amount of memory has somewhat lower performance

- Large amount of memory has lowest performance

- Example hierarchy

    – Dozens of general-purpose registers

    – One or two gigabyte of main memory

    – Hundreds of gigabytes of secondary storage

General Principle

To optimize memory performance for a given cost, a set of technologies are arranged in a hierarchy that contains a relatively small amount of fast memory and larger amounts of less expensive, but slower memory.

Two Possibilities For Computer Memory

- Separate memories, one for programs and another for data

- A single memory that holds both programs and data

Instruction Store And Data Store

- Early computers had separate memories known as

    - *Instruction store*

    - *Data store*

- Most modern computers

    - One memory for both instructions and data

- Note: single memory design is known as a *Von Neumann architecture*

Consequence Of A Von Neumann Architecture

Consequence Of A Von Neumann Architecture

- Instructions and data occupy the same memory

Consequence Of A Von Neumann Architecture

- Instructions and data occupy the same memory

- Consider the following C code:

```
short main[] = {
-25117, -16480, 16384, 28, -28656, 8296, 16384, 26, -28656, 8293, 16384,
24, -28656, 8300, 16384, 22, -28656, 8300, 16384, 20, -28656, 8303,
16384, 18, -28656, 8224, 16384, 16, -28656, 8311, 16384, 14, -28656,
8303, 16384, 12, -28656, 8306, 16384, '\n', -28656, 8300, 16384, '\b',
-28656, 8292, 16384, 6, -28656, 8238, 16384, 4, -28656, 8202, -32313,
-8184, -32280, 0, -25117, -16480, 4352, 5858, -18430, 8600, -4057,
-24508, -17904, 8192, -17913, 24577, -32601, 16412, 9919, -1, -17913,
24577, -27632, 8193, -28656, 8193, 16384, 4, -28153, -24505, -32313,
-8184, -32280, 0, -32240, 8196, -28208, 8192, 6784, 4, 6912, '\b', -26093,
24800, -32317, 16384, 256, 0, -32317, -8184, 256, 0, 0, 0, -32240, 8193,
-28208, 8192, 768, '\b', -12256, 24816, -32317, -8184, -28656, 16383
};
```

Consequence Of A Von Neumann Architecture

- Instructions and data occupy the same memory

- Consider the following C code:

```
short main[] = {
-25117, -16480, 16384, 28, -28656, 8296, 16384, 26, -28656, 8293, 16384,
24, -28656, 8300, 16384, 22, -28656, 8300, 16384, 20, -28656, 8303,
16384, 18, -28656, 8224, 16384, 16, -28656, 8311, 16384, 14, -28656,
8303, 16384, 12, -28656, 8306, 16384, '\n', -28656, 8300, 16384, '\b',
-28656, 8292, 16384, 6, -28656, 8238, 16384, 4, -28656, 8202, -32313,
-8184, -32280, 0, -25117, -16480, 4352, 5858, -18430, 8600, -4057,
-24508, -17904, 8192, -17913, 24577, -32601, 16412, 9919, -1, -17913,
24577, -27632, 8193, -28656, 8193, 16384, 4, -28153, -24505, -32313,
-8184, -32280, 0, -32240, 8196, -28208, 8192, 6784, 4, 6912, '\b', -26093,
24800, -32317, 16384, 256, 0, -32317, -8184, 256, 0, 0, 0, -32240, 8193,
-28208, 8192, 768, '\b', -12256, 24816, -32317, -8184, -28656, 16383
};
```

- Does the code specify instructions or data?

A Note About Memory Types

- Some special-purpose processors require separate instruction and data store

- Motivation

    – Separate caches (described later)

    – Allows memory technology to be optimized for pattern of use

- Access patterns

    – Instruction store: typically sequential

    – Data store: typically random

The Fetch-Store Paradigm

- Access paradigm used by memory

- Two operations

    – *Fetch* a value from a specified location

    – *Store* a value into a specified location

- Two operations also called

    – *Read*

    – *Write*

- We will discuss the implementation and consequences of fetch / store later

Summary

- The two key aspects of memory are

    – Technology

    – Organization

- Memory can be characterized as

    – Volatile or nonvolatile

    – Random or sequential access

    – Permanent or nonpermanent

    – Primary or secondary

Summary
(continued)

- Memory systems use fetch / store paradigm

- Only two operations available

  – *Fetch* (*Read*)

  – *Store* (*Write*)

Questions?

X

Physical Memory
And
Physical Addressing

Computer Memory

- Main memory known as *Random Access Memory* (*RAM*)

- Usually volatile

- Two basic technologies available

  - Static RAM

  - Dynamic RAM

Static RAM (SRAM)

- Easiest to understand

- Similar to flip-flop

Illustration Of Static RAM
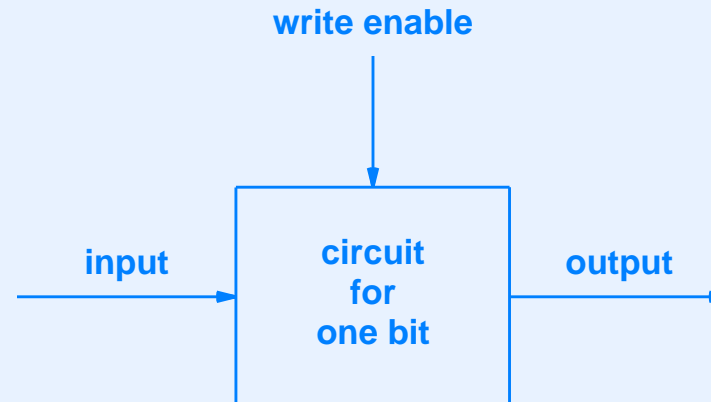
- When *enable* is high, output is same as input

- Otherwise, output holds last value

Advantages And Disadvantages Of SRAM

- Chief advantage

    – High speed

    – No extra refresh circuitry required

- Chief disadvantages

    – Power consumption

    – Heat

    – High cost

Dynamic RAM (DRAM)

- Alternative to SRAM

- Consumes less power

- Acts like a *capacitor* that stores an electrical charge

The Facts Of Electronic Life

The Facts Of Electronic Life

- Entropy increases

The Facts Of Electronic Life

- Entropy increases

- Any electronic storage device gradually loses charge

The Facts Of Electronic Life

- Entropy increases

- Any electronic storage device gradually loses charge

- When left for a long time, a bit in DRAM changes from logical 1 to logical 0

The Facts Of Electronic Life

- Entropy increases

- Any electronic storage device gradually loses charge

- When left for a long time, a bit in DRAM changes from logical 1 to logical 0

- Discharge time can be less than a second

The Facts Of Electronic Life

- Entropy increases

- Any electronic storage device gradually loses charge

- When left for a long time, a bit in DRAM changes from logical 1 to logical 0

- Discharge time can be less than a second

- Conclusion: although it is inexpensive, DRAM is an imperfect memory device!

Making DRAM Work

- Need extra hardware that operates independently

- Repeatedly steps through each location of DRAM

- Reads value from location in DRAM

- Writes value back into same location (recharges the memory bit)

- Extra hardware known as a *refresh circuit*

Illustration Of Bit In DRAM
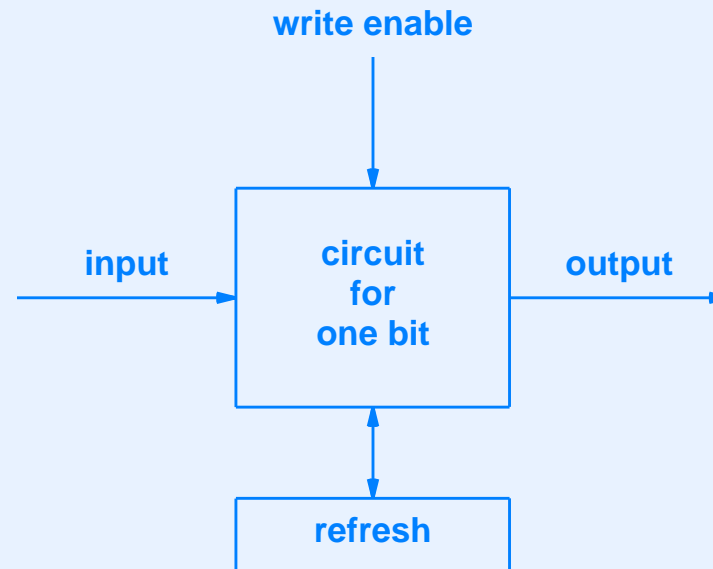
DRAM Refresh Circuit

- More complex than figure implies

- Must coordinate with normal *read* and *write* operations

- Needed for all bits in memory

Measures Of Memory Technology

- Density

- Latency and cycle time

Measuring Memory

- Density

    - Refers to memory cells per square area of silicon

    - Usually stated as number of bits on standard size chip

    - Example: *4 meg chip* holds four megabits of memory

    - Note: higher density chip generates more heat

- Latency

    - Time that elapses between the start of an operation and the completion of the operation

    - Not a constant

Separation Of Read And Write Latency

In many memory technologies, the time required to fetch information from memory differs from the time required to store information in memory, and the difference can be dramatic. Therefore, any measure of memory performance must give two values: the performance of read *operations and the performance of* write *operations.*

Memory Organization

- Hardware unit connects processor to physical memory chips

- Called a *memory controller*


```
┌──────────────┐      ┌──────────┐      ┌──────────────┐
│              │      │ control- │      │   physical   │
│  processor   │◄────►│   ler    │◄────►│    memory    │
│              │      │          │      │              │
└──────────────┘      └──────────┘      └──────────────┘
```
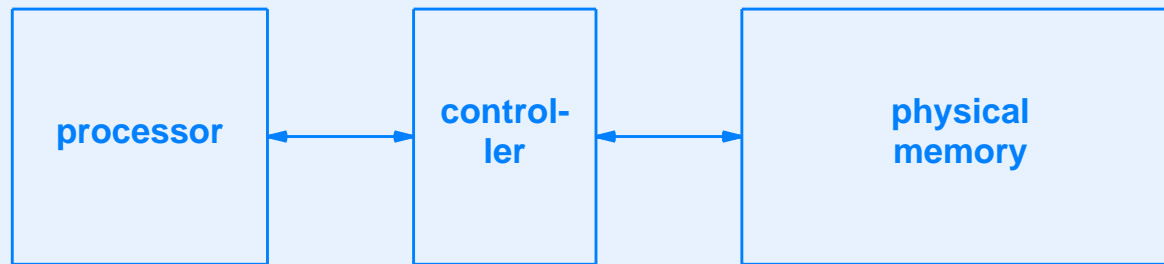
- Main point: because all memory requests go through the controller, the interface a processor "sees" can differ from the underlying hardware organization

Honoring A Memory Request

- Processor

  – Presents request to controller

  – Waits for response

- Controller

  – Translates request into signals for physical memory chips

  – Returns answer to processor immediately

  – Sends signals to reset physical memory for next request

Consequence Of The Need To Reset Memory

Because a memory controller may need extra time between operations to reset the underlying physical memory, latency is an insufficient measure of performance; a performance measure needs to measure the time required for successive operations.

Memory Cycle Time

- Time that must elapse between two successive memory operations

- More accurate measure than latency

- Two separate measures

  - Read cycle time (tRC)

  - Write cycle time (tWC)

The Point About Cycle Times

The read cycle time and write cycle time are used as measures of memory system performance because they measure how quickly the memory system can handle successive requests.

Synchronized Memory Technologies

- Both memory and processor use a clock

- Synchronized memory uses same hardware clock as processor

- Avoids unnecessary delays

- Technique can be used with SRAM or DRAM

- Terminology

  - *Synchronized Static Random Access Memory (SSRAM)*

  - *Synchronized Dynamic Random Access Memory (SDRAM)*

- Note: the RAM in many computers is SDRAM

Multiple Data Rate Memory Technologies

- Technique to improve memory performance

- Avoids a memory bottleneck

- Memory hardware runs at a multiple of CPU clock

- Examples

  - Double Data Rate SDRAM (DDR-SDRAM)

  - Quad Data Rate SRAM (QDR-SRAM)

Example Memory Technologies

Technology	Description
DDR-DRAM	Double Data Rate Dynamic RAM
DDR-SDRAM	Double Data Rate Synchronized Dynamic RAM
FCRAM	Fast Cycle RAM
FPM-DRAM	Fast Page Mode Dynamic RAM
QDR-DRAM	Quad Data Rate Dynamic RAM
QDR-SRAM	Quad Data Rate Static RAM
SDRAM	Synchronized Dynamic RAM
SSRAM	Synchronized Static RAM
ZBT-SRAM	Zero Bus Turnaround Static RAM
RDRAM	Rambus Dynamic RAM
RLDRAM	Reduced Latency Dynamic RAM

- Many others exist

Memory Organization

parallel interface

processor | control-ler | physical memory
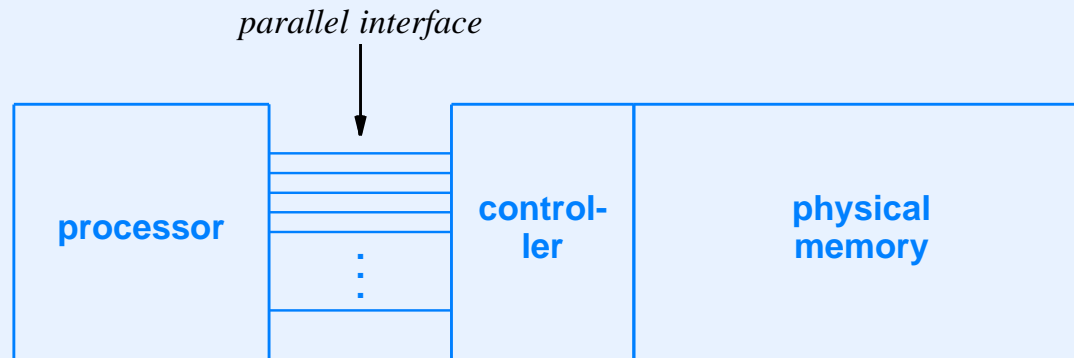
- Parallel interface used between computer and memory

- Called a *bus* (more later in the course)

Memory Transfer Size

- Amount of memory that can be transferred to computer simultaneously

- Determined by bus between computer and controller

- Example memory transfer sizes

    - 16 bits

    - 32 bits

    - 64 bits

- Important to programmers

Physical Memory And Word Size

- Bits of physical memory are divided into blocks of *N* bits each

- Terminology

    – Group of *N* bits is called a *word*

    – *N* is known as the *width* of a word or the *word size*

- Computer is often characterized by its word size (e.g., one might speak of a 64-bit computer)

Physical Memory Addresses

- Each word of memory is assigned a unique number known as a *physical memory address*

- Underlying hardware views physical memory as an array of words

- Note: hardware must transfer an entire word

Illustration Of Physical Memory
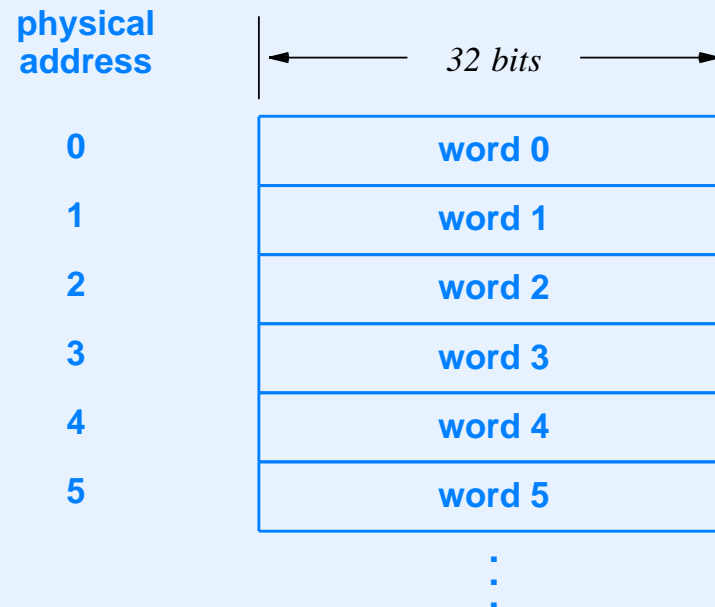
- Figure depicts a 32-bit word size

Summary of Physical Memory Organization

Physical memory is organized into words, where a word is equal to the memory transfer size. *Each* read *or* write *operation applies to an entire word.*

Choosing A Physical Word Size

- Word size represents a tradeoff

- Larger word size

    - Results in higher performance

    - Requires more parallel wires and circuitry

    - Has higher cost and more power consumption

- Note: architect usually designs all parts of computer to use one size for:

    - Memory word

    - Integers (general-purpose registers)

    - Floating point numbers

Byte Addressing

- View of memory presented to processor

- Each byte of memory assigned an address

- Convenient for programmers

- Underlying memory can still use word addressing

Translation Between Byte And Word Addresses

- Performed by *memory controller*

- Allows processor to use byte addressing (convenient)

- Allows physical memory to use word addressing (efficient)

Illustration Of Address Translation

- Assume 32-bit physical word

- Four 8-bit bytes per word

- Bytes numbered sequentially as follows

physical address	32 bits			
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15
4	16	17	18	19
5	20	21	22	23

⋮
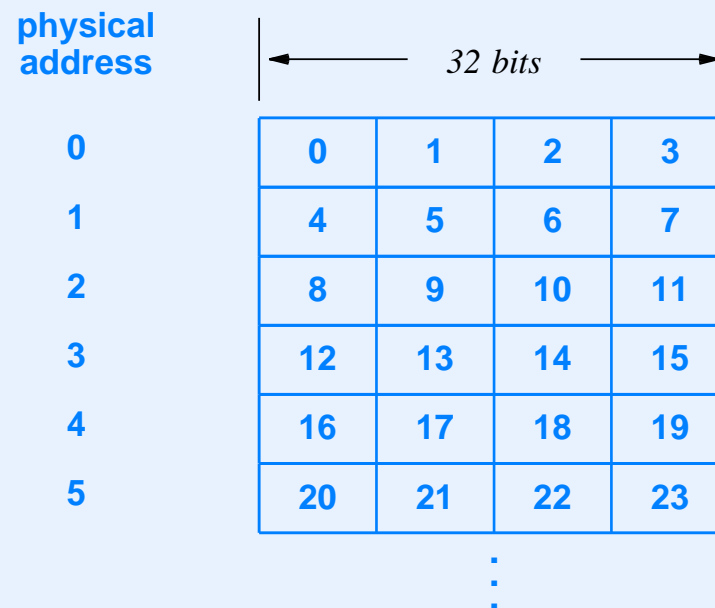
Mathematics Of Translation

- Word address given by:

$$W = \left\lfloor \frac{B}{N} \right\rfloor$$

- Offest given by:

$$O = B \bmod N$$

- Example

  - $N = 4$

  - Byte address 11

  - Found in word 2 at offset 3

Efficient Translation

- Choose word size as power of 2

- Word address computed by extracting high-order bits

- Offset computed by extracting low-order bits

The Important Point

To avoid arithmetic calculations such as division or remainder, physical memory is organized such that the number of bytes per word is a power of two, which means the translation from a byte address to word address and offset can be performed by extracting bits.

Example Of Byte-To-Word Translation

Byte Address, B (17)

| 0 | . . . | 0 | 1 | 0 | 0 | 0 | 1 |

Word Address, W (4)   *Offset, O (1)*
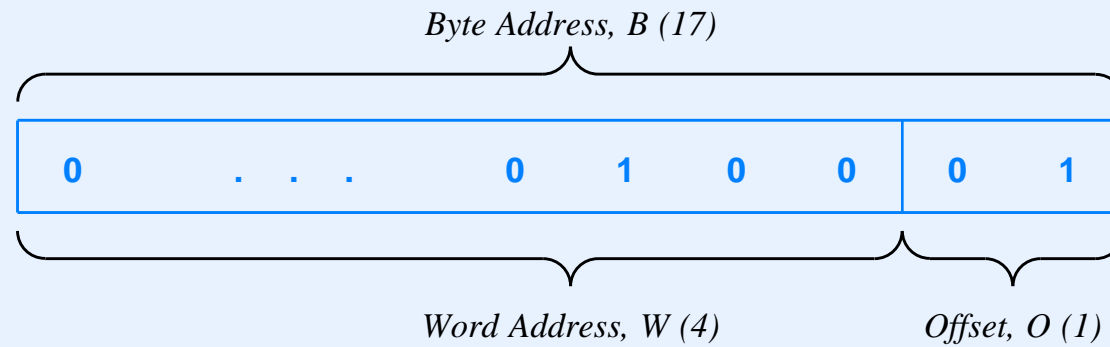
Byte Alignment

- Refers to integer storage in memory

- In some architectures

  - Integer in memory must correspond to word in underlying physical memory

- In other architectures

  - Integer can be unaligned, but *fetch* and *store* operations are much slower

The Point For Programmers

The organization of physical memory affects programming: even if a processor allows unaligned memory access, aligning data on boundaries that correspond to the physical word size can improve program performance.

Memory Size And Address Space

- Size of address limits maximum memory

- Example: 32-bit address can represent

$$2^{32} = 4,294,967,296$$

  unique addresses

- Known as *address space*

- Note: word addressing allows larger memory than byte addressing

Programming On A Computer
That Uses Word Addressing

- To obtain a single byte

    – Fetch word from memory

    – Extract byte from word

- To store a single byte

    – Fetch word from memory

    – Replace byte in word

    – Write entire word back to memory

- Programmer can optimize performance by keeping word in a register until no longer needed

Measures Of Physical Memory Size

Physical memory is organized into a set of M *words that each contain* N *bytes; to make controller hardware efficient,* M *and* N *are each chosen to be powers of two.*

- Consequence of the above: memory sizes expressed as powers of two, not powers of ten

    - Kilobyte defined to be 2^{10} bytes

    - Megabyte defined to be 2^{20} bytes

Consequence To Programmers

- Speeds of data networks and other I/O devices are usually expressed in powers of ten

    – Example: a *Gigabit Ethernet* operates at 10^9 bits per second

- Programmer must accommodate differences between measures for storage and transmission

C Programming And Memory Addressability

- C has a heritage of both byte and word addressing

- Example of byte pointer declaration

$$char \quad *iptr;$$

- Example of integer pointer declaration

$$int \quad *iptr;$$

- If integer size is four bytes, iptr++ increments by four

Memory Dump

- Used for debugging

- Printable representation of bytes in memory

- Each line of output specifies memory address and bytes starting at that address

Example Memory Dump

- Assume linked list in memory

- Head consists of pointer

- Each node has the following structure:

```
struct  node  {
        int count;
        struct node *next;
}
```

Example Memory Dump

Address	Contents Of Memory			
0001bde0	00000000	0001bdf8	deadbeef	4420436f
0001bdf0	6d657200	0001be18	000000c0	0001be14
0001be00	00000064	00000000	00000000	00000002
0001be10	00000000	000000c8	0001be00	00000006

Example Memory Dump

head

Address	Contents Of Memory			
0001bde0	00000000	0001bdf8	deadbeef	4420436f
0001bdf0	6d657200	0001be18	000000c0	0001be14
0001be00	00000064	00000000	00000000	00000002
0001be10	00000000	000000c8	0001be00	00000006

- Assume head located at address 0x0001bde4

Example Memory Dump

Address	Contents Of Memory			
0001bde0	00000000	0001bdf8	deadbeef	4420436f
0001bdf0	6d657200	0001be18	000000c0	0001be14
0001be00	00000064	00000000	00000000	00000002
0001be10	00000000	000000c8	0001be00	00000006

head

node 1

- Assume head located at address 0x0001bde4

- First node at 0x0001bdf8 contains 192 (0xc0)

Example Memory Dump

Address	Contents Of Memory			
0001bde0	00000000	0001bdf8	deadbeef	4420436f
0001bdf0	6d657200	0001be18	000000c0	0001be14
0001be00	00000064	00000000	00000000	00000002
0001be10	00000000	000000c8	0001be00	00000006

- Assume head located at address 0x0001bde4

- First node at 0x0001bdf8 contains 192 (0xc0)

- Second node at 0x0001be14 contains 200 (0xc8)

Example Memory Dump

Address	Contents Of Memory			
0001bde0	00000000	0001bdf8	deadbeef	4420436f
0001bdf0	6d657200	0001be18	000000c0	0001be14
0001be00	00000064	00000000	00000000	00000002
0001be10	00000000	000000c8	0001be00	00000006

(head → 0001bdf8, node 1 → 000000c0 0001be14, node 2 → 000000c8 0001be00, node 3 → 00000064)

- Assume head located at address 0x0001bde4

- First node at 0x0001bdf8 contains 192 (0xc0)

- Second node at 0x0001be14 contains 200 (0xc8)

- Last node at 0x001be00 contains 100 (0x64)

Increasing Memory Performance

- Two major techniques

    - Memory banks

    - Interleaving

- Both employ parallel hardware

Memory Banks

- Alternative to single memory and single memory controller

- Processor connects to multiple controllers

- Each controller connects to separate physical memory

- Controllers and memories can all operate simultaneously

Programming With Memory Banks

- Two approaches

- Transparent

    – Programmer is not concerned with banks

    – Hardware automatically finds and exploits parallelism

- Opaque

    – Banks visible to programmer

    – To optimize performance, programmer must place items that will be accessed simultaneously in separate banks

Interleaving

- Related to memory banks

- Transparent to programmer

- Hardware places consecutive bytes in separate physical memory

- Technique: use low-order bits of address to choose module

- Known as *N-way interleaving*, where *N* is number of physical memories

Illustration Of 4-Way Interleaving
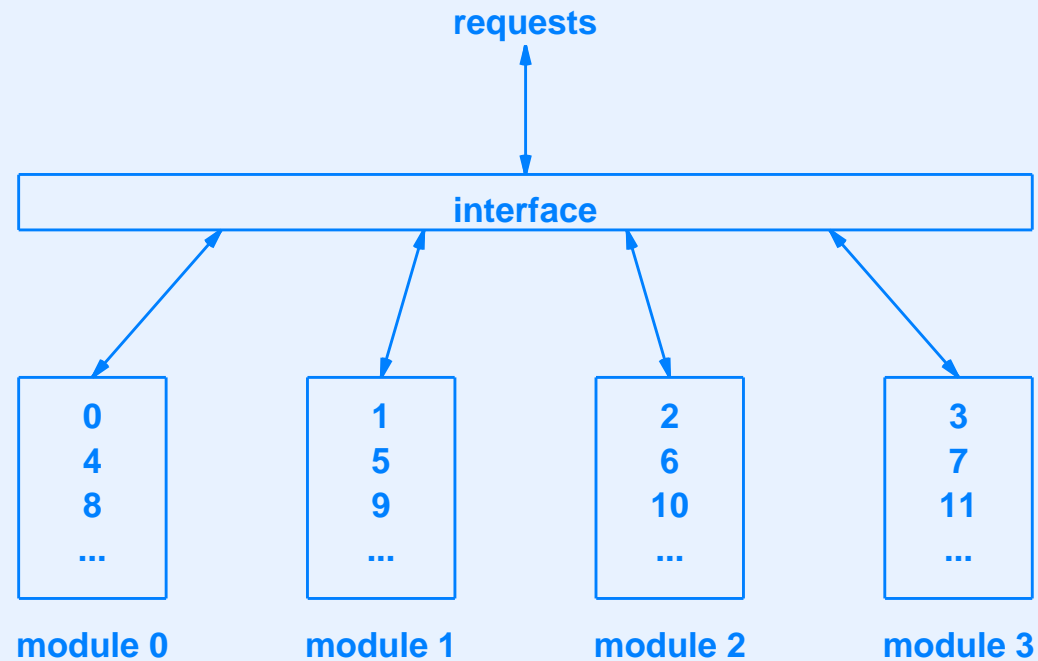
- Consecutive bytes stored in separate physical memory

Content Addressable Memory (CAM)

- Blends two key ideas

  - Memory technology

  - Memory organization

- Includes parallel hardware for high-speed search

CAM

- Think of memory as a two-dimensional array

- Row in the array is called a *slot*

- Special hardware

    – Can answer the question: "is *X* stored in any row of the CAM?"

    – Operates in parallel to make search fast

- Query is known as a *key*

Illustration Of CAM
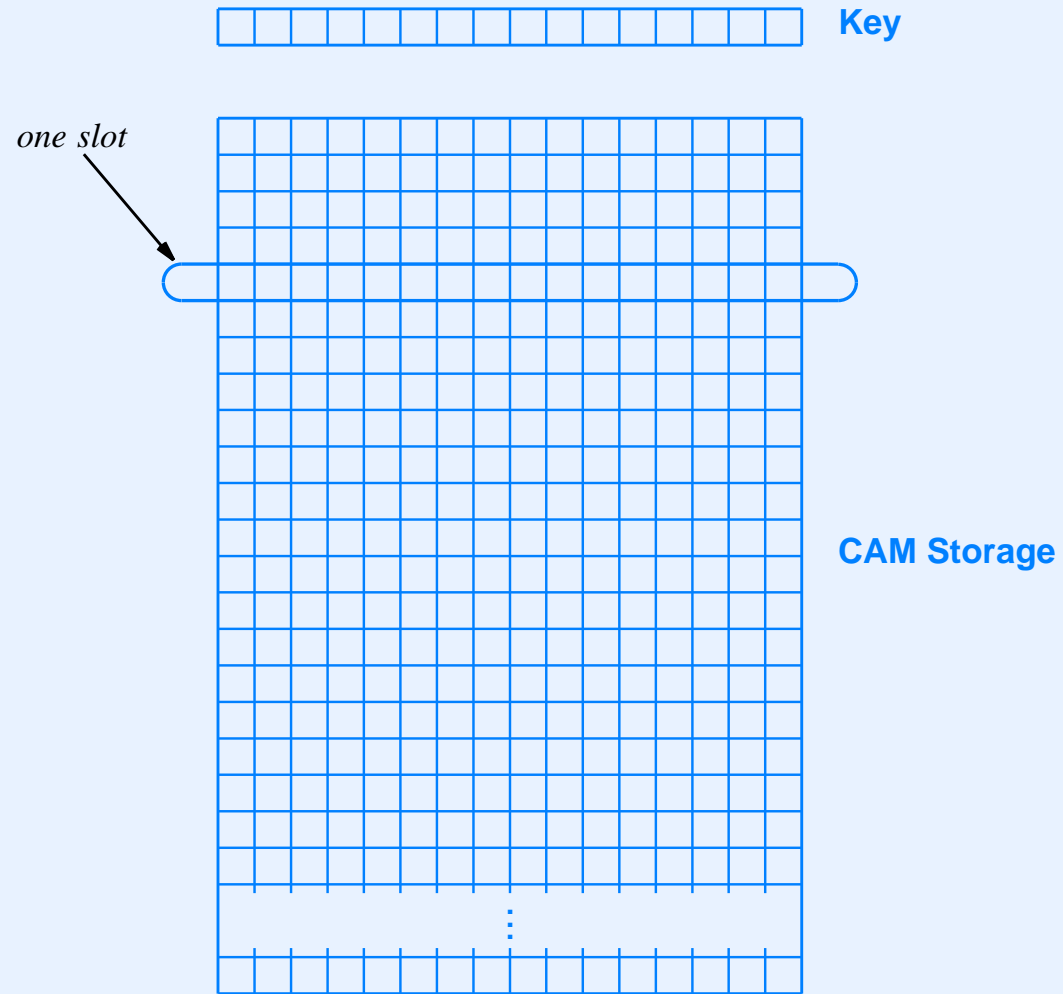
Lookup In A CAM

- CAM presented with key for lookup

- Hardware searches slots to determine whether key is present

    - Search operation performed in parallel on all slots

    - Result is index of slot where value found

- Note: parallel search hardware makes CAM expensive

Ternary CAM (T-CAM)

- Variation of CAM that adds *partial match searching*

- Each bit in slot can have one of three possible values:

    – Zero

    – One

    – Don't care

- T-CAM ignores "don't care" bits and reports match

- T-CAM can either report

    – First match

    – All matches (bit vector)

Summary

- Physical memory

  - Organized into fixed-size words

  - Accessed through a controller

- Controller can use

  - Byte addressing when communicating with a processor

  - Word addressing when communicating with a physical memory

- To avoid arithmetic, use powers of two for

  - Address space size

  - Bytes per word

Summary
(continued)

- Many memory technologies exist

- A memory dump that shows contents of memory in a printable form can be an invaluable tool

- Two techniques used to optimize memory access

    – Separate memory banks

    – Interleaving

- Content Addressable Memory (CAM) permits parallel search; variation of CAM known as Ternary Content Addressable Memory (T-CAM) allows partial match retrieval

Questions?

XI

Virtual Memory Technologies
And
Virtual Addressing

Virtual Memory

- Broad concept

- Hides the details of the underlying physical memory

- Provides a view of memory that is more convenient to a programmer

- Can overcome limitations of physical memory and physical addressing

A Basic Example: Byte Addressing

- CPU uses byte addresses

- Underlying physical memory uses word addresses

- Memory controller translates automatically

- Fits our definition of *virtual memory*

Virtual Memory Terminology

- *Memory Management Unit* (*MMU*)

    – Hardware unit

    – Provides translation between virtual and physical memory schemes

- *Virtual address*

    – Address generated by processor

    – Translated into corresponding physical address by MMU

Virtual Memory Terminology
(continued)

- *Virtual address space*

    – Set of all possible virtual addresses

    – Can be larger or smaller than physical memory

- *Virtual memory system*

    – All of the above

Multiple Physical Memory Systems

- Many computers have more than one physical memory system

- Each physical memory

    - Can be optimized for a specific purpose

    - Can use a unique technology (e.g., SRAM or DRAM)

- Virtual memory system can provide uniform address space for all physical memories

Illustration Of Virtual Address Space
That Covers Two Memories
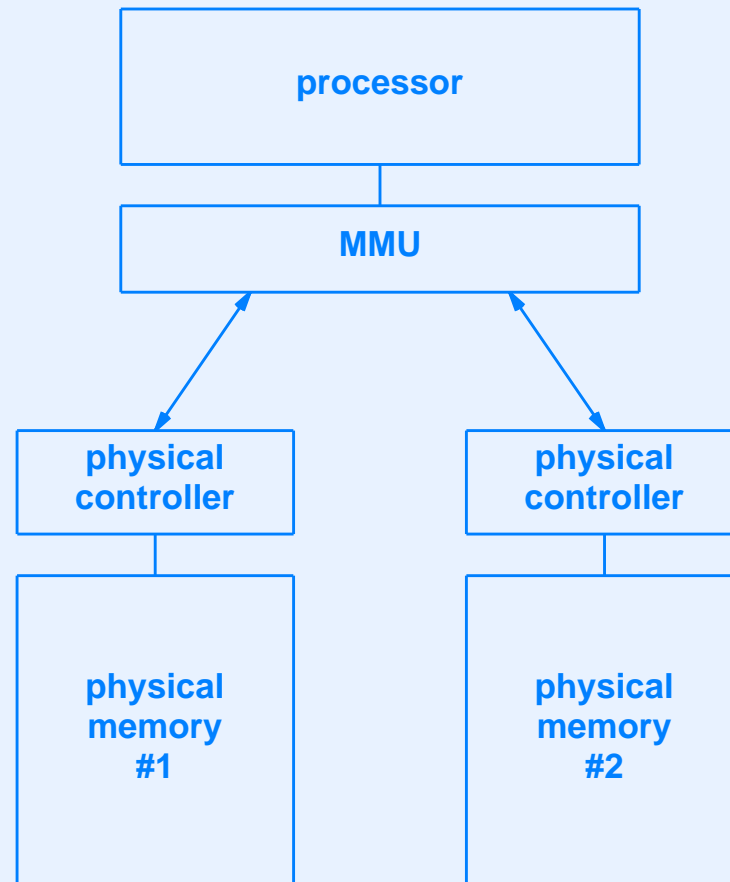
Virtual Addressing

- Processor must have unique address for each location in memory

- MMU translates from virtual space to underlying physical memories

- Example:

    – Two physical memories with 1000 bytes each

    – Virtual addresses 0 through 999 correspond to memory 1

    – Virtual addresses 1000 through 1999 correspond to memory 2

Illustration Of Virtual Addressing
That Spans Two Physical Memories

Address

0

memory 1

999
1000

memory 2

1999

Processor sees a single contiguous memory
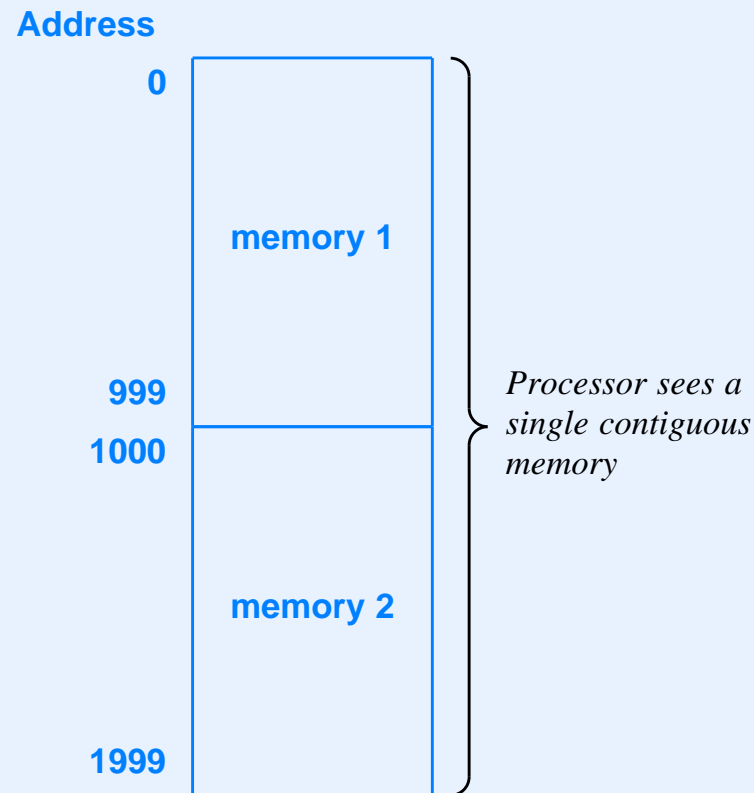
Address Translation

- Performed by MMU

- Also called *address mapping*

- For our example

  - To determine which physical memory, test if address is above 999

  - Subtract 1000 from address when forwarding a request to memory 2

Algorithm To Perform The Example Address Translation

receive memory request from processor;
let A be the address in the request;
if (A >= 1000) {
        A = A − 1000;
        pass the modified request to memory 2;
} else {
        pass the unmodified request to memory 1;
}

Avoiding Arithmetic Calculation

- Arithmetic computation

    – Is expensive

    – Can be avoided

- Divide virtual address space along boundaries that correspond to powers of two

- Select bits of virtual address to

    – Choose among underlying physical memories

    – Specify an address in the physical memory

Example Using Powers Of Two

- Two physical memories

- Each memory contains 1024 bytes

- Virtual addresses 0 through 1023 map to memory 1

- Virtual addresses 1024 through 2047 map to memory 2

- No arithmetic is required

Example Addresses In Binary

Addresses	Values In Binary
0	0 0 0 0 0 0 0 0 0 0 0
to	to
1023	0 1 1 1 1 1 1 1 1 1 1
1024	1 0 0 0 0 0 0 0 0 0 0
to	to
2047	1 1 1 1 1 1 1 1 1 1 1

- Values above 1023 are the same as previous set except for high-order bit

- High-order bit determines physical memory (0 or 1)

The Important Point

Dividing a virtual address space on a boundary that corresponds to a power of two allows the MMU to choose a physical memory and perform the necessary address translation without requiring arithmetic operations.

Address Space Continuity

- *Contiguous address space*

    – All locations correspond to physical memory

    – Inflexible: requires all memory sockets to be populated

- *Discontiguous address space*

    – One or more blocks of address space do not correspond to physical memory

    – Called *hole*

    – Fetch or store to address in a hole causes an error

    – Flexible: allows owner to decide how much memory to install

Illustration Of Discontiguous Address Space
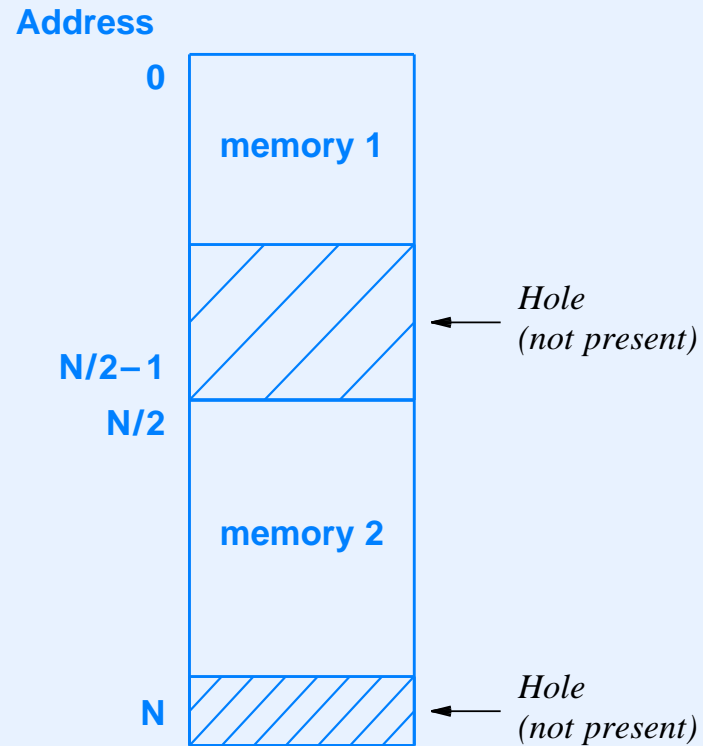
Consequence To A Programmer

A virtual address space can be contiguous, in which case every address maps to a location of an underlying physical memory, or noncontiguous, in which case the address space contains one or more holes. If a processor attempts to read *or* write *any address that does not correspond to physical memory, an error results.*

Motivations For Virtual Memory

- Homogeneous integration of hardware

- Programming convenience

- Support for multiprogramming

- Protection of programs and data

Multiple Virtual Spaces And Multiprogramming

- Goal: allow multiple application programs to run concurrently

- Prevent one program from interfering with another

- Trick: provide each program with a separate virtual address space

Illustration Of Four Virtual Address Spaces Mapped To A Single Physical Address Space
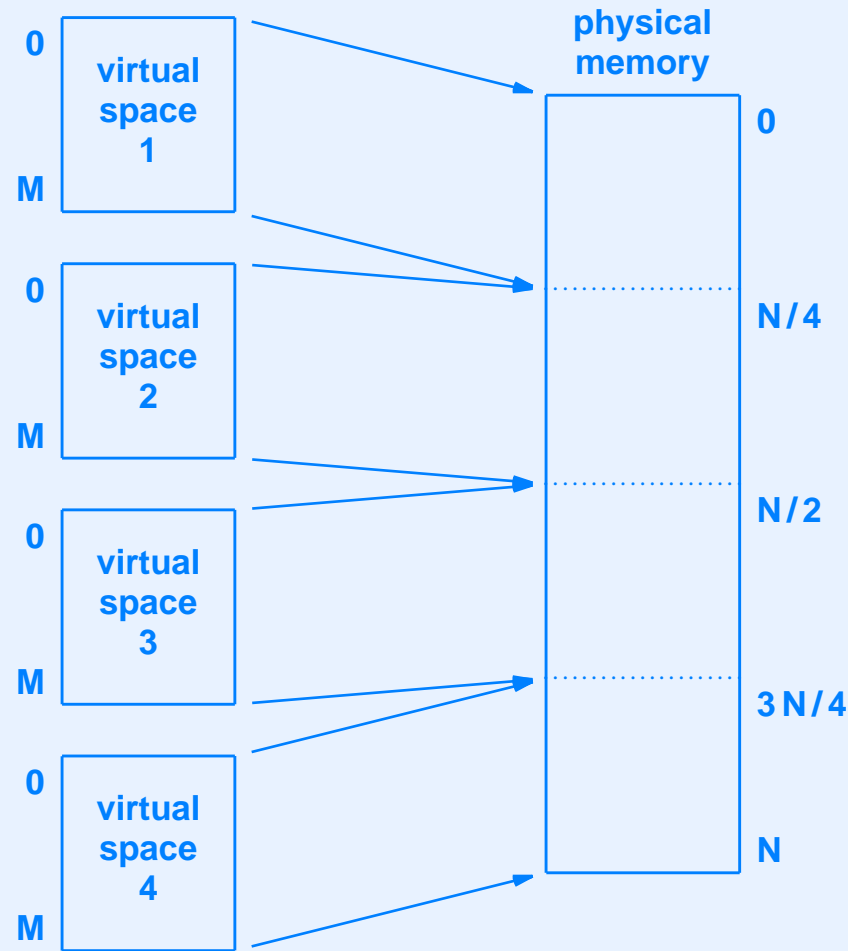
Dynamic Address Space Creation

- Processor configures MMU

- Address space mapping can be changed at any time

- Typically

    – Access to MMU restricted to operating system

    – OS runs in *real mode* (access to physical address space)

    – Changes to virtual memory only affect application programs

Technologies For Dynamic Address Space Manipulation

- Base-bound registers

- Segmentation

- Demand paging

Base-Bound Registers

- Two hardware registers in MMU

- Base register specifies starting address

- Bound register specifies size of address space

- Values changed by operating system

    - Set before application runs

    - Changed by operating system when switching to another application

Illustration Of Virtual Memory Using Base-Bound Registers
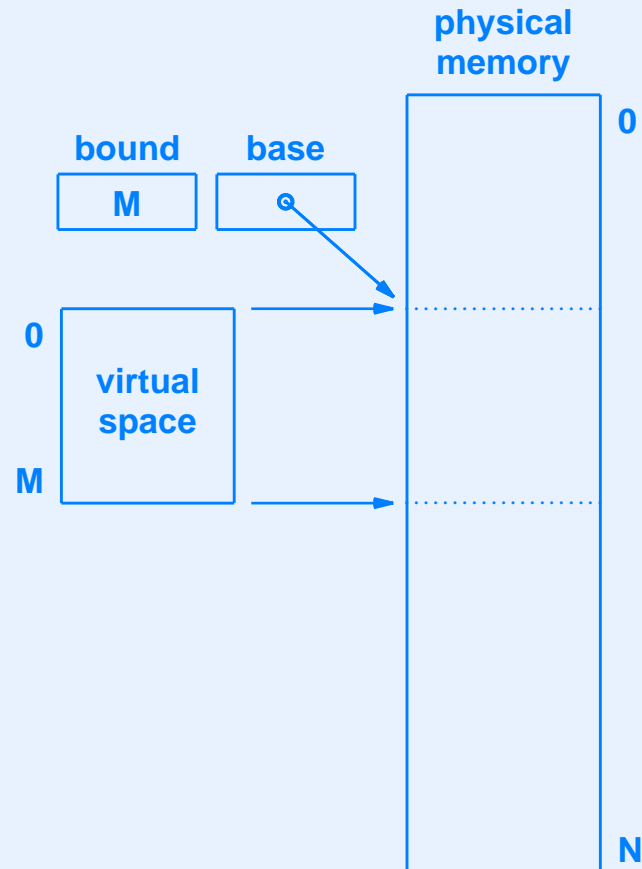
Protection

- Multiple applications each allocated separate area of physical memory

- OS sets base-bound registers before application runs

- MMU hardware checks each memory reference

- Reference to any address outside the valid range results in an error

The Concept Of Protection

A virtual memory system that supports multiprogramming must also provide protection that prevents one program from reading or altering memory that has been allocated to another program.

Segmentation

- Alternative to base-bound

- Provides *fine-granularity* mapping

    - Divides program into *segments* (typical segment corresponds to one procedure)

    - Maps each segment to physical memory

- Key idea

    - Segment is only placed in physical memory when needed

    - When segment is no longer needed, OS moves it to disk

Problems With Segmentation

- Need hardware support to make moving segments efficient

- Two choices

  – Variable-size segments cause memory *fragmentation*

  – Fixed-size segments may be too small or too large

Summary Of Segmentation

Segmentation refers to a virtual memory scheme in which programs are divided into variable-size blocks, and only the blocks currently needed are kept in memory. Because it leads to a problem known as memory fragmentation, segmentation is seldom used.

Demand Paging

- Alternative to segmentation and base-bounds

- Most popular virtual memory technology

- Divides program into fixed-size pieces called *pages*

- No attempt to align page boundary with procedure

- Typical page size 4K bytes

Support Needed For Demand Paging

- Hardware that handles address mapping and detects missing pages

- Software that moves pages between external store and physical memory

Paging Hardware

- Part of MMU

- Intercepts each memory reference

- If referenced page is present in memory, translate address

- If referenced page not present in memory, generate a *page fault* (error condition)

- Allow operating system to handle the fault

Demand Paging Software

- Part of the operating system

- Works closely with hardware

- Responsible for overall memory management

- Determines which pages of each application to keep in memory and which to keep on disk

- Records location of all pages

- Fetches pages on demand

- Configures the MMU

Page Replacement

- Initially

  - Applications reference pages

  - Each referenced page is placed in physical memory

- Eventually

  - Memory is full

  - Existing page must be written to disk before memory can be used for new page

- Choosing a page to expel is known as *page replacement*

- Should replace a page that will not be needed soon

Paging Terminology

- *Page*: fixed-size piece of program's address space

- *Frame*: slot in memory exactly the size of one page

- *Resident*: a page that is currently in memory

- *Resident set*: pages from a given application that are present in memory

Paging Data Structure

- Page table

    - One per application

    - Think of each as one-dimensional array indexed by page number

    - Stores the location of each page in the application (either in memory or on disk)

Illustration Of A Page Table
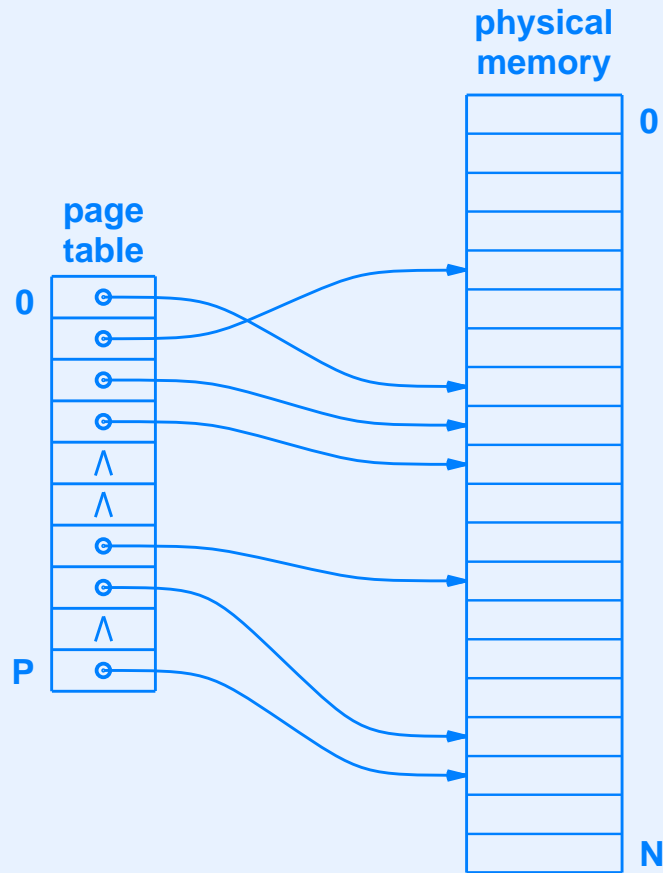
- Typical system has 4K bytes per page

Address Translation

- Given virtual address *V*, find physical memory address *P*

- Three conceptual steps

  - Determine the number of the page on which address *V* lies

  - Use the page number as an index into the page table to find the location in memory that corresponds to the first byte of the page

  - Determine how far into the page *V* lies, and convert to a position in the frame in memory

Mathematical View Of Address Translation

- Page number computed by dividing the virtual address by the number of bytes per page, *K*:

$$N = \left\lfloor \frac{V}{K} \right\rfloor$$

- Offset within the page, *O*, can be computed as the remainder:

$$O = V \; modulo \; K$$

Mathematical View Of Address Translation
(continued)

- Use N and O to translate virtual address V to physical address P:

$$P = pagetable\,[N] + O$$

Using Powers Of Two

- Cannot afford division or remainder operation for each memory reference

- Use powers of two to eliminate arithmetic

- Let number of bytes per page be 2^k

    - Offset O given by low-order k bits

    - Page number given by remaining (high-order) bits

- Computation is:

$$P = pagetable \, [\, high\_order\_bits \, (V) \,] \;\; or \;\; low\_order\_bits \, (V)$$

Illustration Of Translation With MMU Hardware
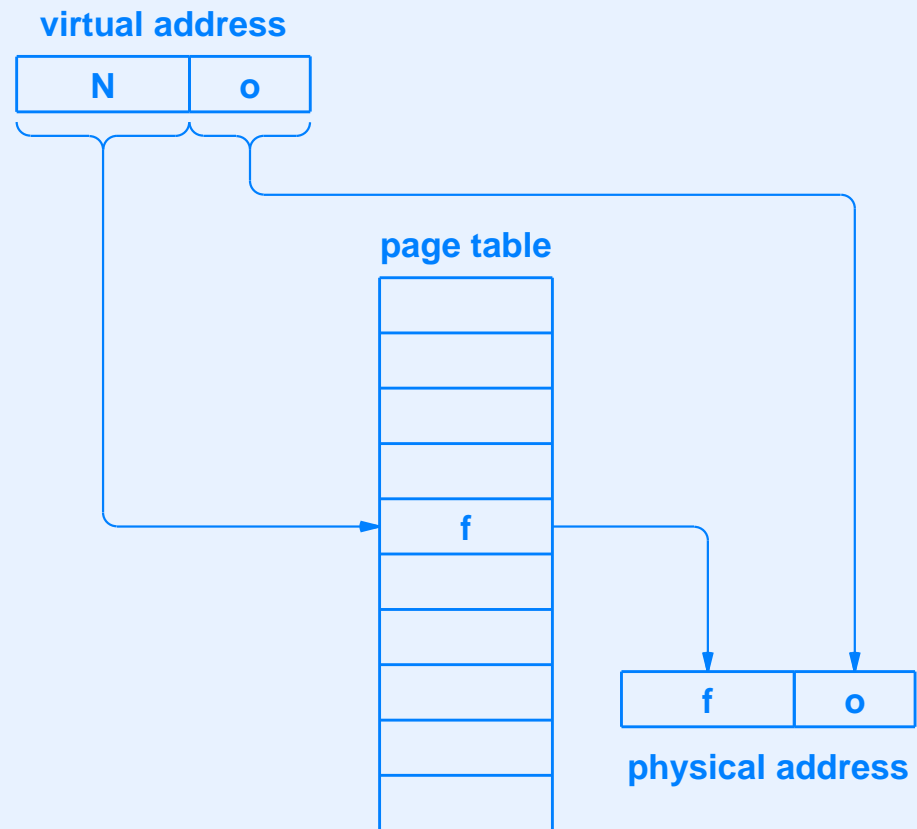
Presence, Use, And Modified Bits

- Found in most paging hardware

- Shared by hardware and software

- Purpose of each bit:

Control Bit	Meaning
Presence bit	Tested by hardware to determine whether page is currently present in memory
Use bit	Set by hardware whenever page is referenced
Modified bit	Set by hardware whenever page is changed

Page Table Storage

- Page tables occupy space

- Two possibilities for page table storage

    - In MMU

    - In main memory

Illustration Of Page Tables
Stored In Physical Memory
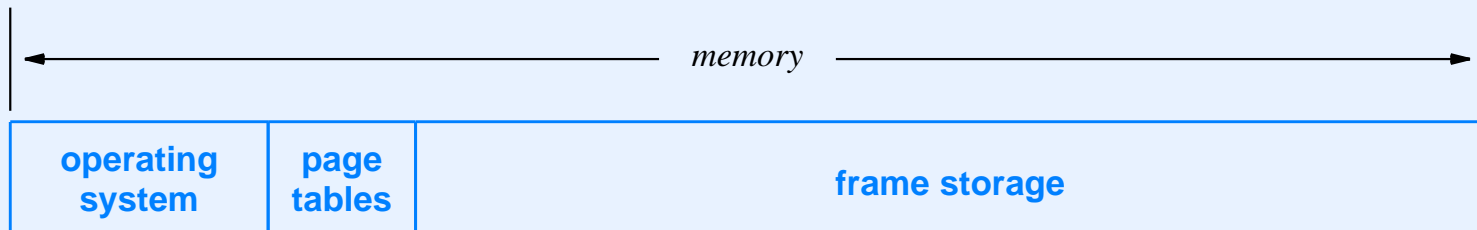
Paging Efficiency

- Paging must be used

    – For each instruction fetch

    – For each data reference

- Can become a bottleneck

- Must be optimized

Translation Lookaside Buffer (TLB)

- Hardware mechanism

- Optimizes paging system

- Form of Content Addressable Memory (CAM)

- Stores pairs of

    (virtual address, physical address)

- If mapping in TLB

    – No page table reference needed

    – MMU can return mapping quickly

In Practice

- Virtual memory system without TLB is unacceptable

- TLB works well because application programs tend to reference given page many times

The Importance Of A TLB

A special high-speed hardware device called a Translation Lookaside Buffer (TLB) is used to optimize performance of a paging system. A virtual memory that does not have a TLB can be unacceptably slow.

Consequences For Programmers

- Can optimize performance by accommodating paging system

- Examples

    – Group related data items on same page

    – Reference arrays in order that accesses contiguous memory locations

Array Reference

- Illustration of array in *row-major order*

row 0	row 1	row 2	row 3	row 4	row 5		row N

. . .

- Location of A[i, j] given by:

$$location(A) \ + \ i{\times}Q \ + \ j$$

where *Q* is number of bytes per row

Programming To Optimize Array Access

- Optimal

```
for  i  =  1  to  N  {
    for  j  =  1  to  M  {
        A [ i, j ]  =  0;
    }
}
```

- Nonoptimal

```
for  j  =  1  to  M  {
    for  i  =  1  to  N  {
        A [ i, j ]  =  0;
    }
}
```

Summary

- Virtual memory systems present illusion to processor and programs

- Many virtual memory architectures are possible

- Examples include

    - Hiding details of word addressing

    - Create uniform address space that spans multiple memories

    - Incorporate heterogeneous memory technologies into single address space

Summary
(continued)

- Virtual memory offers

    – Convenience for programmer

    – Support for multiprogramming

    – Protection

- Three technologies used for virtual memory

    – Base-bound registers

    – Segmentation

    – Demand paging

Summary
(continued)

- Demand paging

    – The most popular technology

    – Combination of hardware and software

    – Uses page tables to map virtual addresses to physical addresses

    – High-speed lookup mechanism known as TLB makes demand paging practical

Questions?

XII

Caches And Caching

Caching

- Key concept in computing

- Used in hardware and software

- Memory cache essential to reduce the Von Neumann bottleneck

Cache

- Acts as an intermediary

- Located between source of requests and source of replies

requester   cache   large data storage
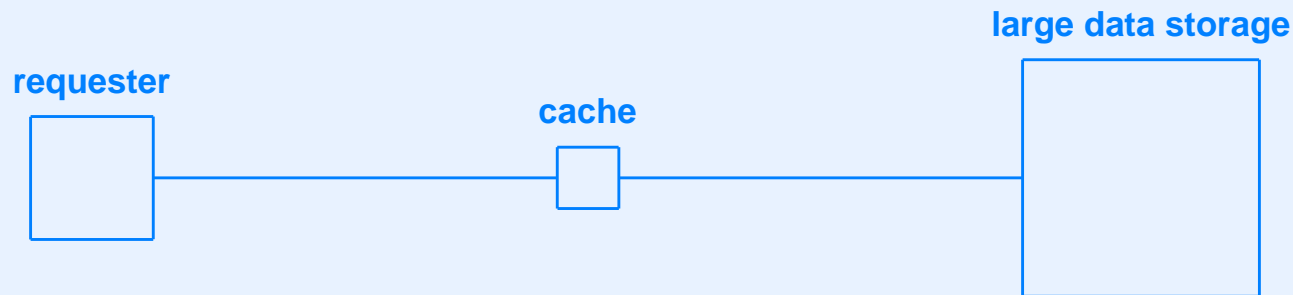
- Cache contains temporary local storage

  – Very high-speed

  – Limited size

- Copy of selected items kept in local storage

- Cache answers requests from local copy when possible

Cache Characteristics

- Small (usually much smaller than storage needed for entire set of items)

- Active (makes decisions about which items to save)

- Transparent (invisible to both requester and data store)

- Automatic (uses sequence of requests; does not receive extra instructions)

Generality Of Caching

- Implemented in hardware, software, or a combination

- Small or large data items (a byte of memory or a complete file)

- Generic data items (e.g., disk block)

- Specific data item (e.g., document from a word processor)

- Textual data (e.g., an email message)

- Nontextual data (e.g., an image, an audio file, or a video clip)

Generality Of Caching
(continued)

- A single computer system (e.g., between a processor and a memory)

- Many computer systems (e.g., between a set of desktop computers and a database server)

- Systems that are designed to retrieve data (e.g., the World Wide Web)

- Systems that store as well as retrieve data (e.g., a physical memory)

The Importance Of Caching

Caching is a fundamental optimization technique used throughout most hardware and software systems that retrieve information. Caching is a broad concept; data items kept in a cache are not limited to a specific type, form, or size.

Cache Terminology

- *Cache hit*

    – Request that can be satisfied from cache

    – No need to access data store

- *Cache miss*

    – Request cannot be satisfied from cache

    – Cache retrieves item from data store

Cache Terminology
(continued)

- *Locality of reference*

  – Refers to repetitions of same request

  – High locality means many repetitions

  – Low locality means few repetitions

- Note: cache works well with high locality of reference

Cache Performance

- Cost measured with respect to requester
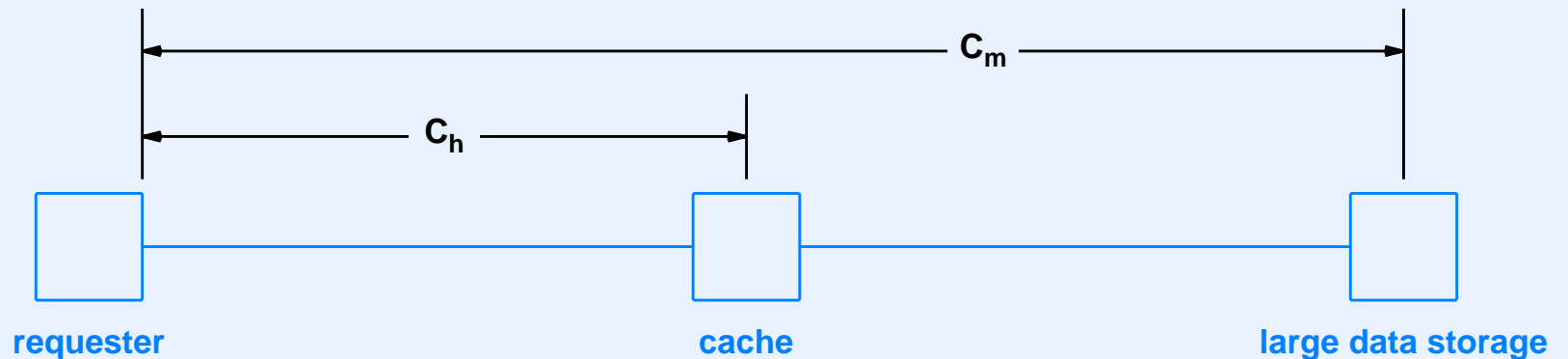
- C_h is the cost of an item found in the cache (hit)

- C_m is the cost of an item not found in the cache (miss)

Analysis Of Cache Performance

Analysis Of Cache Performance

- Worst case for sequence of *N* requests

Analysis Of Cache Performance

- Worst case for sequence of N requests

$$C_{worst} = N \, C_m$$

Analysis Of Cache Performance

- Worst case for sequence of N requests

$$C_{worst} = N \, C_m$$

- Best case for sequence of N requests

Analysis Of Cache Performance

- Worst case for sequence of N requests

$$C_{worst} = N\ C_m$$

- Best case for sequence of N requests

$$C_{best} = C_m + (N - 1)\ C_h$$

Analysis Of Cache Performance

- Worst case for sequence of N requests

$$C_{worst} = N\ C_m$$

- Best case for sequence of N requests

$$C_{best} = C_m + (N - 1)\ C_h$$

- For best cast, the average cost per request is:

$$\frac{C_m + (N - 1)\ C_h}{N} = \frac{C_m}{N} - \frac{C_h}{N} + C_h$$

Analysis Of Cache Performance

- Worst case for sequence of N requests

$$C_{worst} = N\ C_m$$

- Best case for sequence of N requests

$$C_{best} = C_m + (N-1)\ C_h$$

- For best cast, the average cost per request is:

$$\frac{C_m + (N-1)\ C_h}{N} = \frac{C_m}{N} - \frac{C_h}{N} + C_h$$

- Note: as $N \to \infty$, average cost becomes C_h

Summary Of Costs

If we ignore overhead, in the worst case, the performance of caching is no worse than if the cache were not present. In the best case, the cost per request is approximately equal to the cost of accessing the cache, which is lower than the cost of accessing the data store.

Definition Of Hit and Miss Ratios

- *Hit ratio*

    – Percentage of requests satisfied from cache

    – Given as value between 0 and 1

- *Miss ratio*

    – Percentage of requests not satisfied from cache

    – Equal to 1 minus hit ratio

Cache Performance On A Typical Sequence

- Access cost depends on hit ratio

$$Cost \ = \ r \ C_h \ + \ (1-r) \ C_m$$

  where r is the hit ratio

- Note: performance improves if hit ratio increases or cost of access from cache decreases

Cache Replacement Policy

- Recall: a cache is smaller than data store

- Once cache is full, existing item must be ejected before another can be inserted

- *Replacement policy* chooses items to eject

- Most popular replacement policy known as *Least Recently Used* (*LRU*)

    - Easy to implement

    - Tends to retain items that will be requested again

    - Works well in practice

Multi-level Cache Hierarchy

- Can use multiple caches to improve performance

- Arranged in hierarchy by speed

- Example: insert an extra cache in previous diagram

requester   **new cache**   **original cache**   **large data storage**
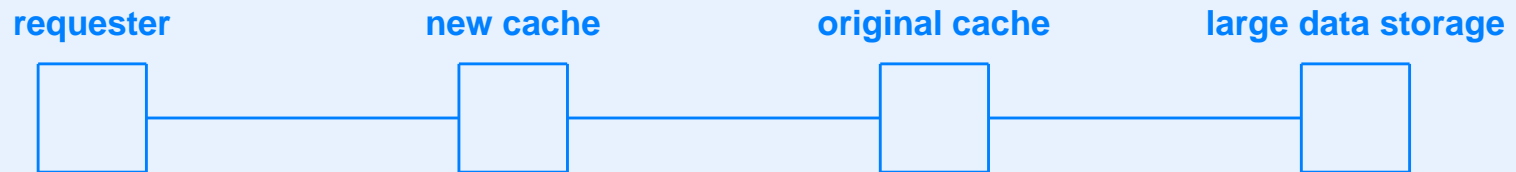
Analysis Of Two-Level Cache

- Cost is:

$$Cost \; = \; r_1 \, C_{h1} \; + \; r_2 \, C_{h2} \; + \; (1 - r_1 - r_{2)} C_m$$

- r_1 is fraction of hits for the new cache

- r_2 is fraction of hits for the original cache

- C_{h1} is cost of accessing the new cache

- C_{h2} is cost of accessing the original cache

Preloading Caches

- Optimization technique

- Stores items in cache *before* requests arrive

- Works well if data accessed in related groups

- Examples

    - When web page is fetched, web cache can preload images that appear on the page

    - When byte of memory is fetched, memory cache can preload succeeding bytes

Memory Cache

- Several memory mechanisms operate as a cache

    - TLB used in a virtual memory system

    - Pages in a demand paging system

    - Words of memory in a physical memory system

Demand Paging Performance

Cache analysis shows that using demand paging on a computer system with a small physical memory can perform almost as well as if the computer had a physical memory large enough for the entire virtual address space.

Physical Memory Cache

- Located between processor and physical memory

- Smaller than physical memory

- Note: sophisticated cache hardware operates in parallel to achieve high performance:

    - Search local cache

    - Send request to underlying memory

- If answer found in cache, cancel request to memory

Two Basic Types Of Cache

- Differ in how the caches handle a *write* operation

- *Write-through*

    – Place a copy of item in cache

    – Also send (*write*) a copy to physical memory

- *Write-back*

    – Place a copy of item in cache

    – Only write the copy to physical memory when necessary

    – Works well for frequent updates (e.g., a loop increments a value)

Writes On A System With Multiple Caches
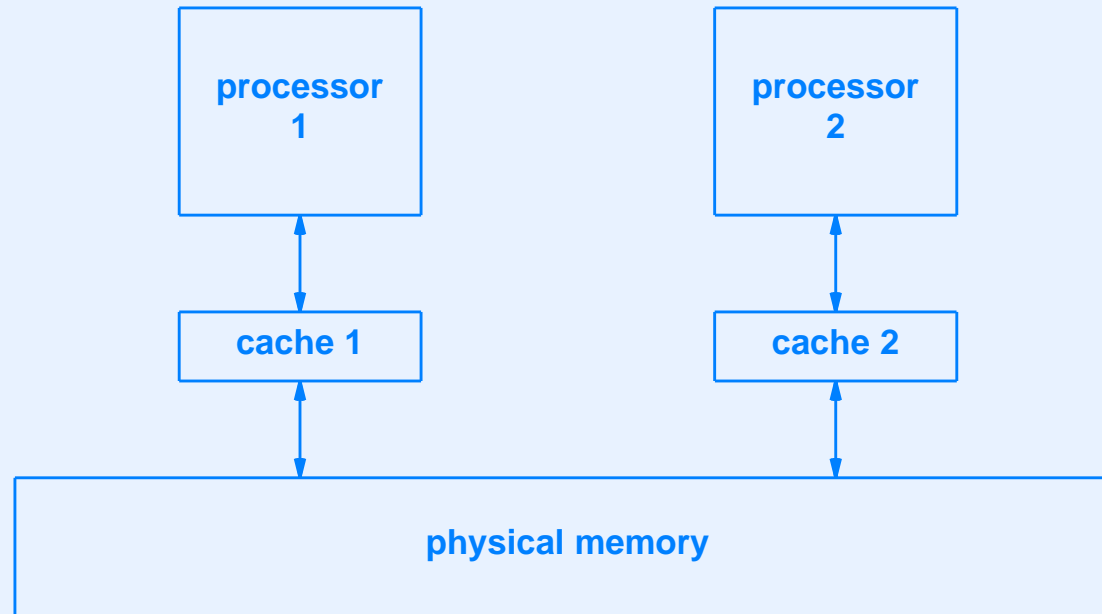
- Write-back means each cache can retain copy of item

- *Cache coherence* needed to ensure correctness

Motivation For Multi-Level Memory Cache

- Traditional memory cache was separate from both the memory and the processor

- To access traditional memory cache, a processor used pins that connect the processor chip to the rest of the computer

- Using pins to access external hardware takes much longer than accessing functional units that are internal to the processor chip

- Advances in technology have made it possible to increase the number of transistors per chip, which means a processor chip can contain a larger cache

Multi-Level Memory Cache Terminology

- *Level 1 cache* (*L1 cache*) on the processor chip

- *Level 2 cache* (*L2 cache*) external to the processor

- *Level 3 cache* (*L3 cache*) built into the physical memory

Cost Of Accessing Memory

Computer systems use a multi-level cache hierarchy in which an L1 cache is embedded on the processor chip, an L2 cache is external to the processor, and an L3 cache is built into the physical memory. In the best case, a multi-level cache makes the cost of accessing memory approximately the same as the cost of accessing a register.

Instruction And Data Caches

- Instruction references are typically sequential

    – High locality of reference

    – Amenable to prefetching

- Data references typically exhibit more randomness

    – Lower locality of reference

    – Prefetching does not work well

- Question: does performance improve with separate caches for data and instructions?

Instruction And Data Caches
(continued)

- Cache tends to work well with sequential references

- Adding many random references tends to lower cache performance

- Therefore, separating instruction and data caches can improve performance

- However: if cache is "large enough", separation doesn't help

Virtual Memory Caching

- Can build a system that caches

  - Physical memory address and contents

  - Virtual memory address and contents

- Notes

  - If MMU is off-chip, L1 cache must use virtual addresses

  - Multiple applications use *same* virtual address space

Handling Overlapping Virtual Addresses

- Each application uses virtual addresses 0 through N

- System must insure that an application does not receive data from another application's memory

- Two possible approaches

  - OS performs cache *flush* operation when changing applications

  - Cache includes disambiguating tag with each entry (i.e., an application ID)

Illustration Of ID Register
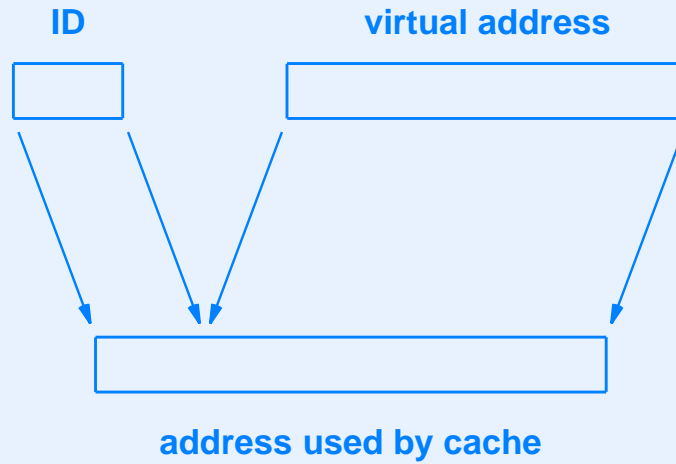
Two Technologies For Memory Caching

- Direct mapping memory cache

- Set associative memory cache

Direct Mapping Memory Cache

- Divides memory into *K* numbered blocks, where *K* is number of slots in cache

- Tag used to distinguish among blocks

- Example: block size of 4 bytes

memory				block
0	1	2	3	0
4	5	6	7	1
8	9	10	11	2
12	13	14	15	3

⋮

- Only block numbered *i* can be placed in cache slot *i*

Illustration Of Tags

memory   **block**

cache

tag	value	
		0
		1
		2
		3
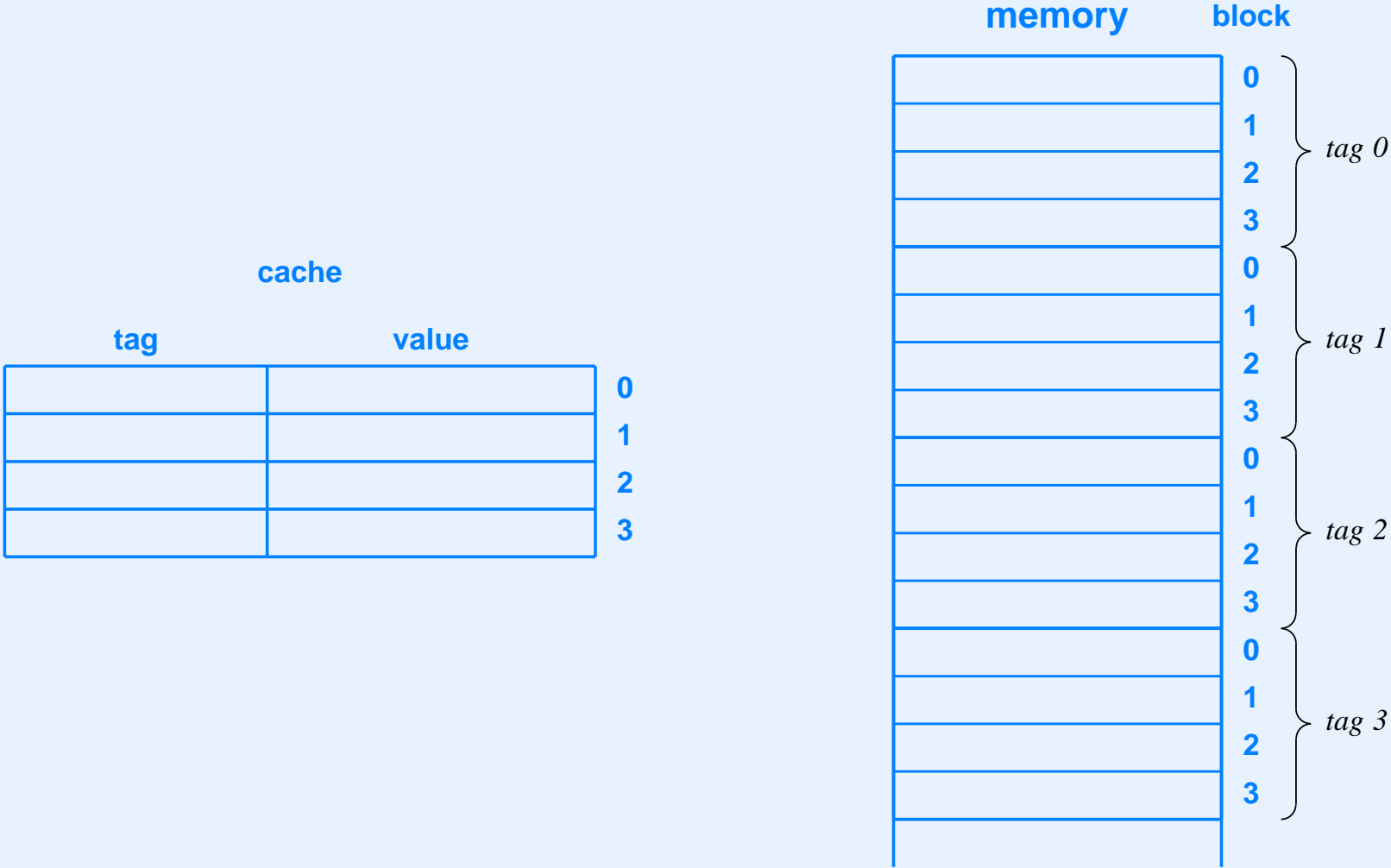
- Use of tags saves space

Using Powers Of Two

- If all values are powers of two, bits of an address can be used to specify a *tag*, *block*, and *offset*

tag	block	offset

Algorithm For Cache Lookup

Given:

    A memory address

Find:

    The data byte at that address

Method:

Extract the tag number, t, block number, b, and offset, o, from the address.

Examine the tag in slot b of the cache. If the tag matches t, extract the value from slot b of the cache.

If the tag in slot b of the cache does not match t, use the memory address to extract the block from memory, place a copy in slot b of the cache, replace the tag with t, and use o to select the appropriate byte from the value.

Set Associative Memory Cache

- Alternative to direct mapping memory cache

- Uses parallel hardware

- Maintains multiple, independent caches
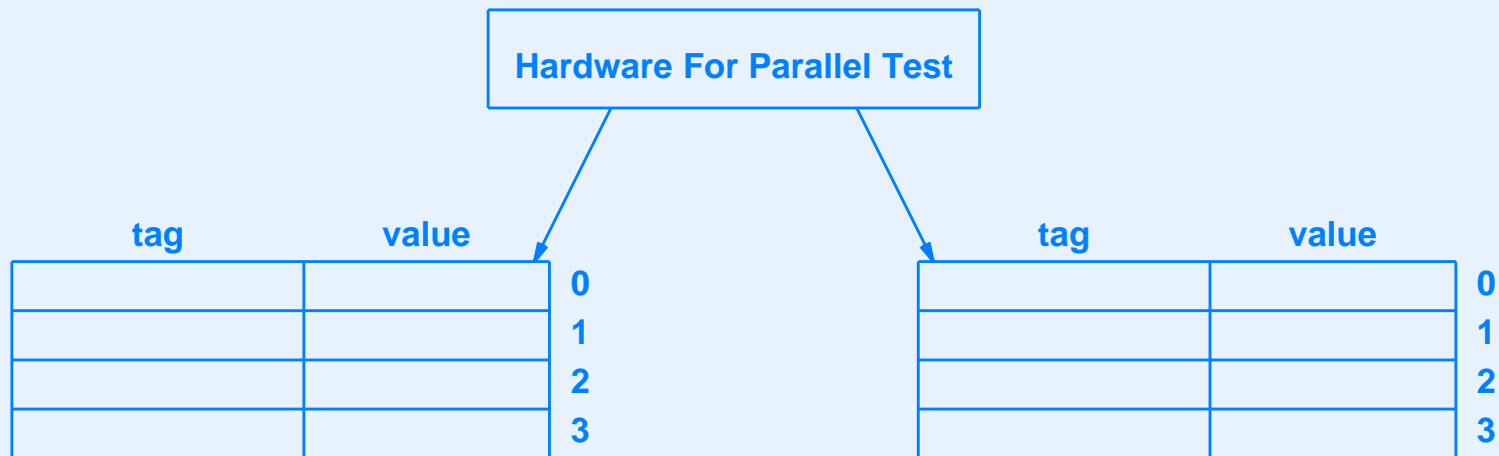
Advantage Of Set Associative Cache

- Assume two memory addresses A_1 and A_2

    – Both have block number zero

    – Have different tags

- In direct mapped cache

    – A_1 and A_2 contend for single slot

    – Only one can be cached at a given time

- In set associative cache

    – A_1 and A_2 can be placed in separate caches

    – Both can be cached at a given time

Fully Associative Cache

- Generalization of set associative cache

- Many parallel caches

- Each cache has exactly one slot

- Slot can hold arbitrary item

Conceptual Continuum Of Caches

- No parallelism corresponds to direct mapped cache

- Some parallelism corresponds to set associative cache

- Full parallelism corresponds to Content Addressable Memory

Consequences For Programmers

- In many programs caching works well without extra work

- To optimize cache performance

    - Group related data items into same cache line (e.g., related bytes into a word)

    - Perform all operations on one data item before moving to another data item

Summary

- Caching is fundamental optimization technique

- Cache intercepts requests, automatically stores values, and answers requests quickly, whenever possible

- Caching can be used with both physical and virtual memory addresses

- Memory cache uses hierarchy

    – L1 onboard processor

    – L2 between processor and memory

    – L3 built into memory

Summary
(continued)

- Two technologies used for memory cache

    – Direct mapped

    – Set associative

Questions?

XIII

Input / Output
Concepts
And
Terminology

I/O Devices

- Third major component of computer system

- Wide range of types

  - Keyboards

  - Mice

  - Monitors

  - Hard disks

  - Printers

  - Cameras

  - Audio speakers

Conceptual Organization Of Basic I/O Device

- Operates independent of processor

- Separate power supply

- Digital signals used for control

- Example: panel lights
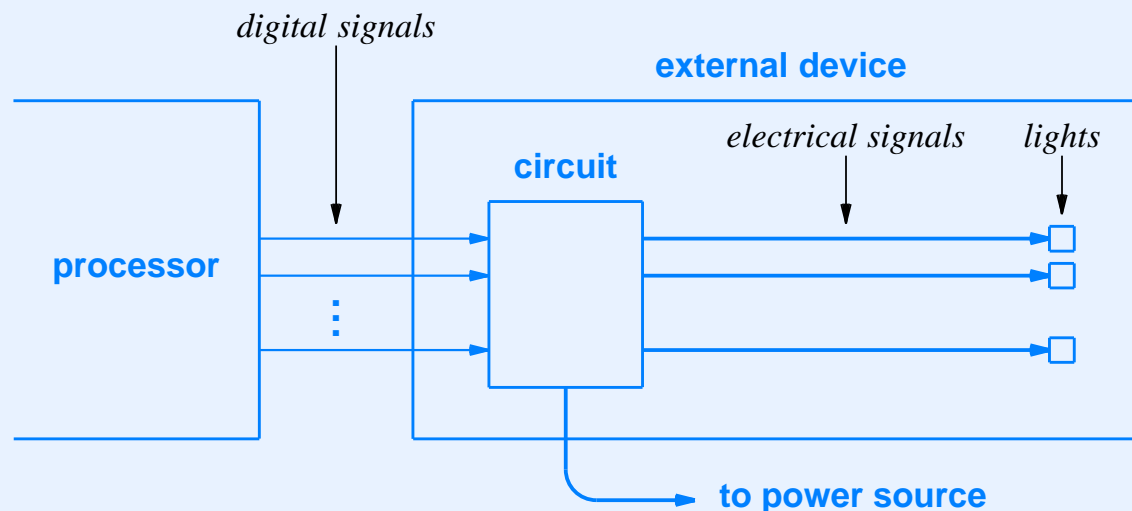
Illustration Of Modern Interface Controller

- Controller placed at each end of physical connection

- Allows arbitrary voltage and signals to be used
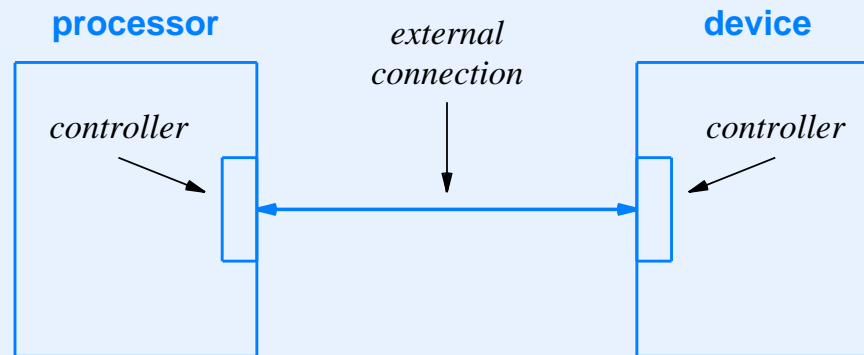
Two Types Of Interfaces

- Parallel interface

    – Composed of many wires

    – Each wire carries one bit at any time

    – *Width* is number of wires

    – Complex hardware with higher cost

- Serial interface

    – Single signal wire (also need ground)

    – Bits sent one-at-a-time

    – Slower than parallel interface

    – Less complex hardware with lower cost

Self-Clocking Data

- Ends of connection use separate clocks

  - Processor

  - I/O device

- Transmission is *self-clocking* if signal is encoded in such a way that receiver can determine boundary of bits without knowing sender's clock

Duplex Terminology

- Full-duplex

  - Simultaneous, bi-directional transfer

  - Example: disk drive supports simultaneous *read* and *write* operations

- Half-duplex

  - Transfer in one direction at a time

  - Interfaces must negotiate access before transmitting

  - Example: processor can *read* or *write* to a disk, but can only perform one operation at a time

Latency And Throughput

The latency *of an interface is a measure of the time required to perform a transfer, the* throughput *of an interface is a measure of the data that can be transferred per unit time.*

Data Multiplexing

- Fundamental idea

- Arises from hardware limits on parallelism (pins or wires)

- Allows sharing

- Multiplexor

  – Accepts input from many sources

  – Sends small amount from one source before accepting another

- Demultiplexor

  – Receives transmission of pieces

  – Sends each piece to appropriate destination

Illustration Of Multiplexing

- Example: sixty-four bits of data multiplexed over 16-bit path

64 bits of data to be transferred

unit 1	unit 2	unit 3	unit 4

multiplexing hardware

parallel interface sends 16 at a time →

demultiplexing hardware

unit 1	unit 2	unit 3	unit 4

data reassembled after transfer
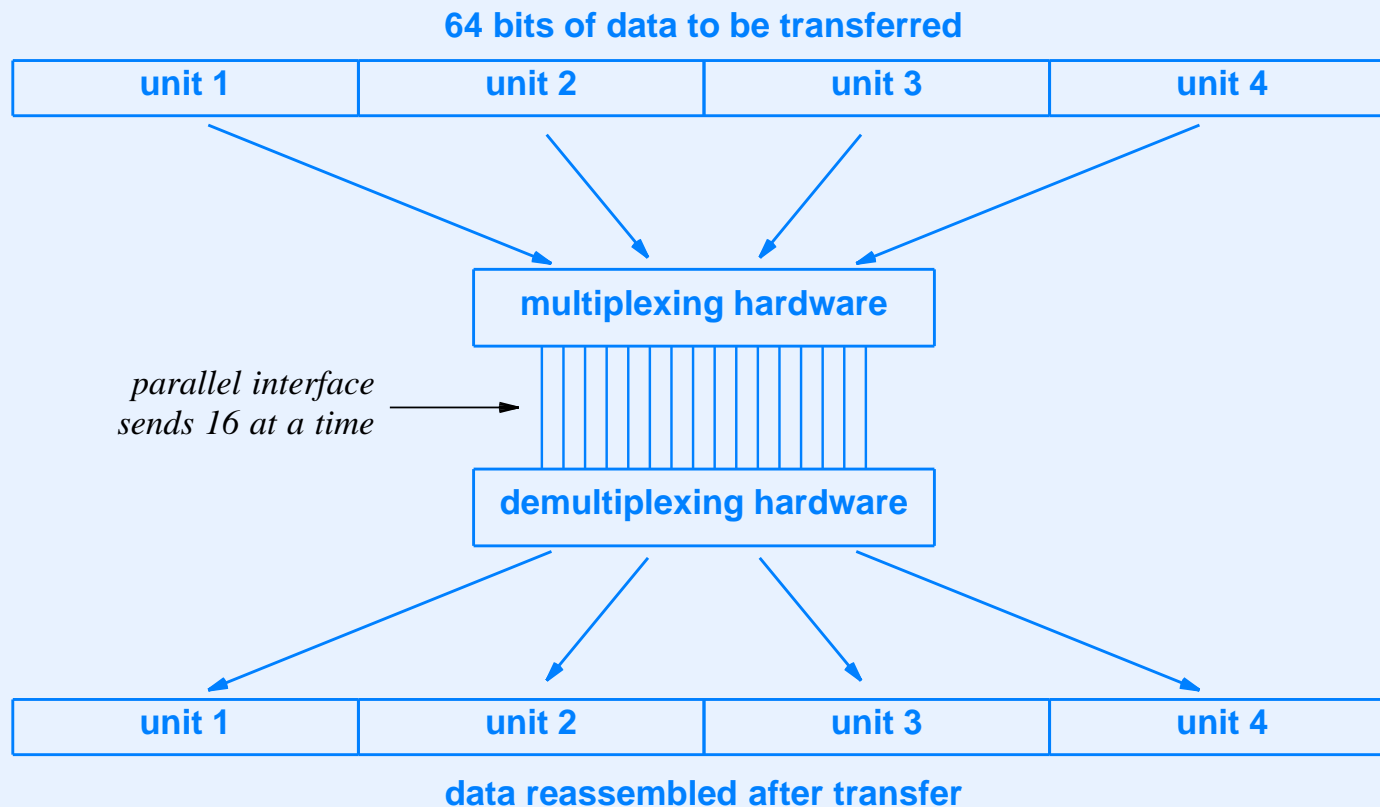
Multiplexing And I/O Interfaces

Multiplexing is used to construct an I/O interface that can transfer arbitrary amounts of data over a fixed number of parallel wires. Multiplexing hardware divides the data into blocks, and transfers each block independently.

Multiple Devices Per External Interface

- Cannot have a separate physical interconnect per device

    – Too many physical wires

    – Not enough pins on processor chip

    – Interface hardware adds economic cost

- Sharing allows multiple devices to use a given interconnection

- The next section of the course discusses connection sharing

Processor's View Of I/O

A processor does not access external devices directly. Instead, the processor uses a programming interface to pass requests to an interface controller, which translates the requests into the appropriate external signals.

Questions?

XIV

Buses
And
Bus Architecture

Definition Of A Bus

- Digital interconnection mechanism

- Allows two or more functional units to transfer data

- Typical use: connect processor to

    – Memory

    – I/O devices

- Design can be

    – Proprietary (owned by one company)

    – Open standard (available to many companies)

Illustration Of A Bus

- Double-headed arrow often used to denote a bus

- Connection to bus shown from components

- Example

```
┌──────────┐        ┌────────┐      ┌────────┐
│          │        │        │      │        │
│ processor│        │ memory │      │ device │
│          │        │        │      │        │
└────┬─────┘        └───┬────┘      └───┬────┘
     │                  │               │
◄════╧══════════════════╧═══════════════╧═══════►
                      bus
```
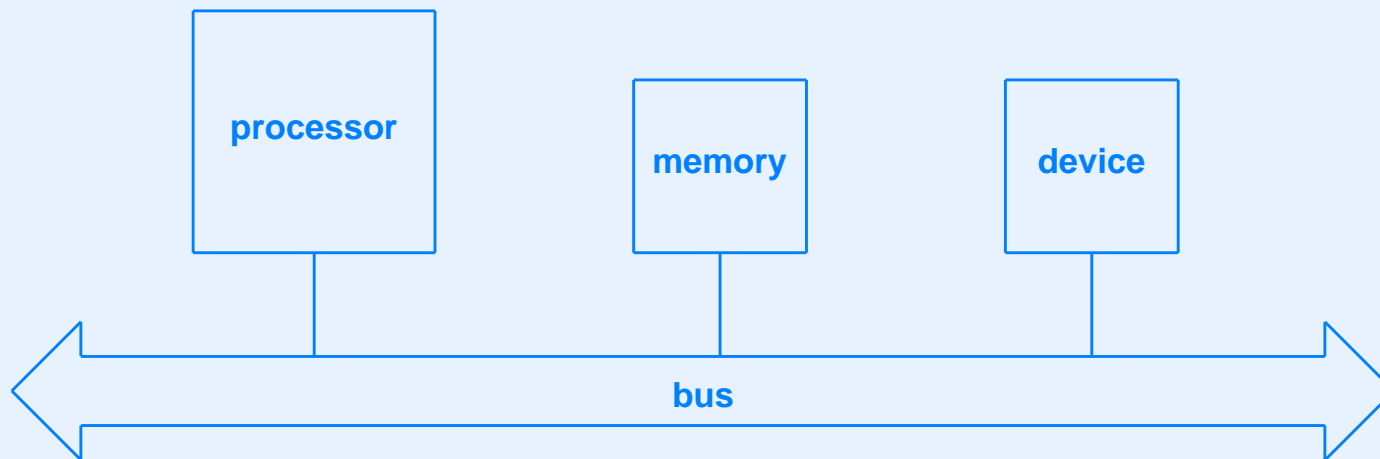
Sharing

- Most buses shared by multiple devices

- Need an *access protocol*

    – Determines which device can use the bus at any time

    – All attached devices follow the protocol

- Note: it is possible to have multiple buses in one computer

Characteristics Of A Bus

- Parallel data transfer

    - Hardware to transfer multiple bits at the same time

    - Typical width is 32 or 64 bits

- Essentially passive

    - Bus does not contain many electronic components

    - Attached devices handle communication

- Conceptual view: think of a bus as parallel wires

- Bus may have *arbiter* that manages sharing

Physical Bus Connections

- Several possibilities

- Can consist of

    - Physical wires

    - Traces on a circuit board

- Usually, bus has sockets into which devices plug

Illustration Of Bus On A Motherboard

mother board

area on mother board for the processor, memory, and other units

sockets placed near the edge of the board

bus formed from parallel wires
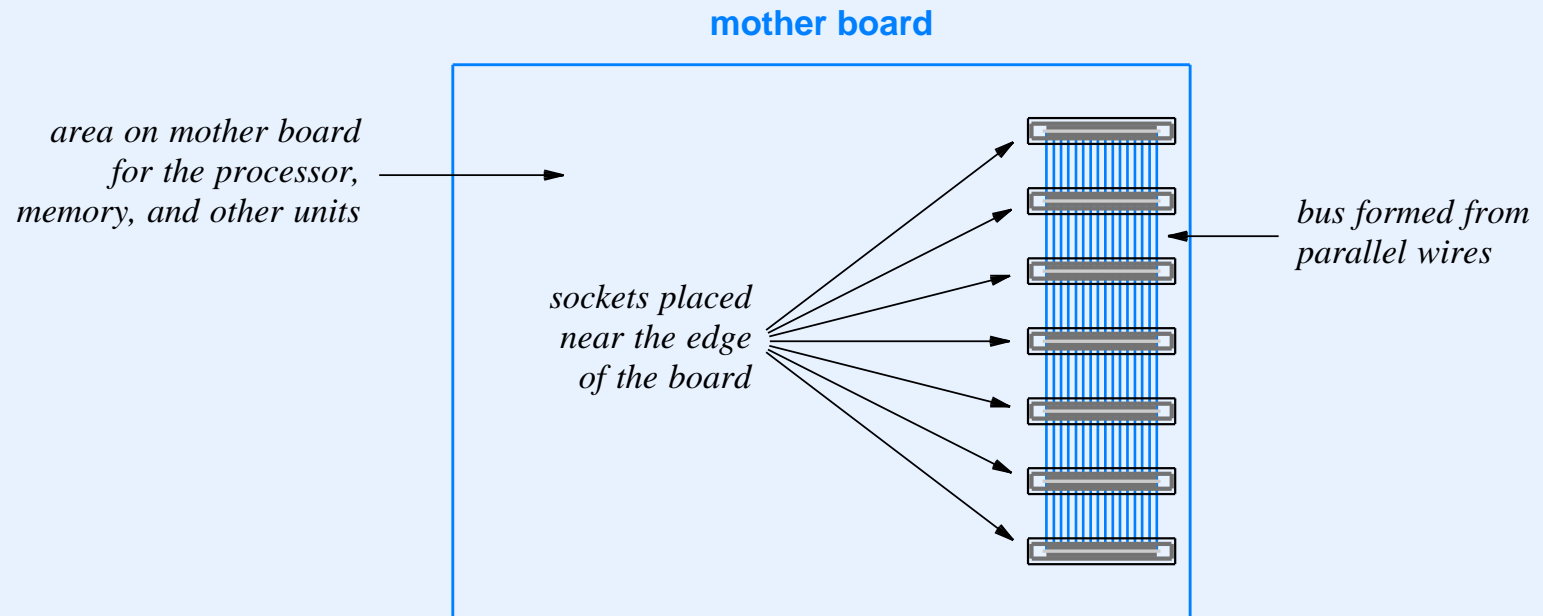
Side View Of Circuit Board
And Corresponding Sockets

- Each I/O device on a circuit board

- I/O devices plug into sockets on the mother board

circuit board
(device interface)

mother board

socket
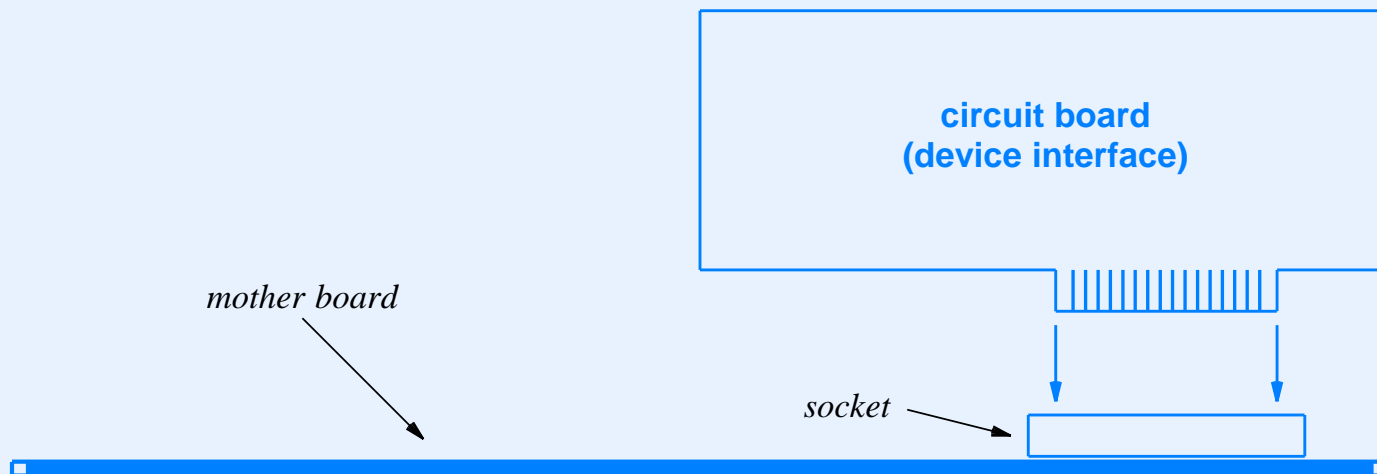
Bus Interface

- Access protocol is nontrivial

- Controller circuit required

- Circuitry part of each I/O device

Conceptual Division Of A Bus

- Three functions

    – Control (devices determine whether bus is currently in use and which device will use it next)

    – Address specification (requester specifies an address)

    – Data being transferred (responder uses bus to send requested item)

- Conceptually separate group of wires (*lines*) for each function

Illustration Of Lines In A Bus

- In simplest case, separate hardware exists for control, address, and data

control lines   **address lines**   **data lines**

- To lower cost, some bus designs arrange to share address and data lines (value sent in request is address; value sent in response is data)

Bus Access

- Bus only supports two operations

  - *Fetch* (also called *read*)

  - *Store* (also called *write*)

- Access paradigm known as *fetch-store paradigm*

- Obvious for memory access

- Surprise: all bus operations, including communication between a processor and an I/O device must be performed using fetch-store paradigm

Fetch-Store Over A Bus

- Fetch

  - Place an address on the address lines

  - Use control line to signal *fetch* operation

  - Wait for control line to indicate *operation complete*

  - Extract data item from the data lines

- Store

  - Place an address on the address lines

  - Place data item on the data lines

  - Use control line to signal *store* operation

  - Wait for control line to indicate *operation complete*

Width Of A Bus

- Larger width

  – Higher performance

  – Higher cost

  – Requires more pins

- Smaller width

  – Lower cost

  – Lower performance

  – Requires fewer pins

- Compromise: multiplex transfers to reduce cost

Multiplexing

- Same as multiplexing on a parallel interface

- Reuses lines for multiple purposes

- Extreme case

  - Serial bus has one line

- Typical case

  - Bus has K lines

  - Address can be K bits wide

  - Data can be K bits wide

Illustration Of A Bus Using Multiplexing

- Bus has control and value lines

- Value lines used to pass both addresses and data

control
lines

address or data
lines
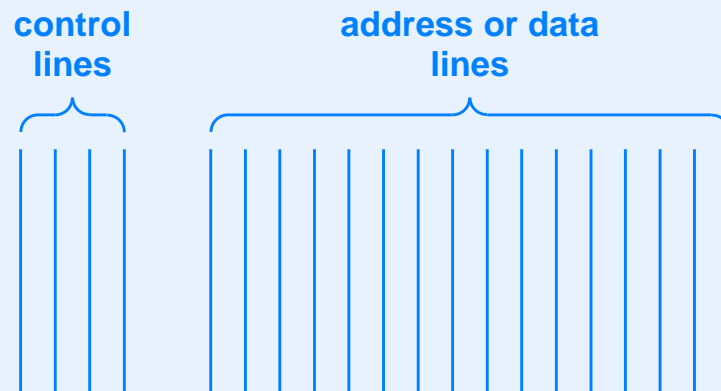
- Transfer takes longer with multiplexing

- Controller hardware is more sophisticated

Effect Of Bus Multiplexing Architecture

Addresses and data values can be multiplexed over a bus. To optimize performance of the hardware, an architect chooses a single size for both data items and addresses.

Illustration Of Connection To Memory Bus

- Bus provides path between processor and memory

- Memory hardware includes bus controller
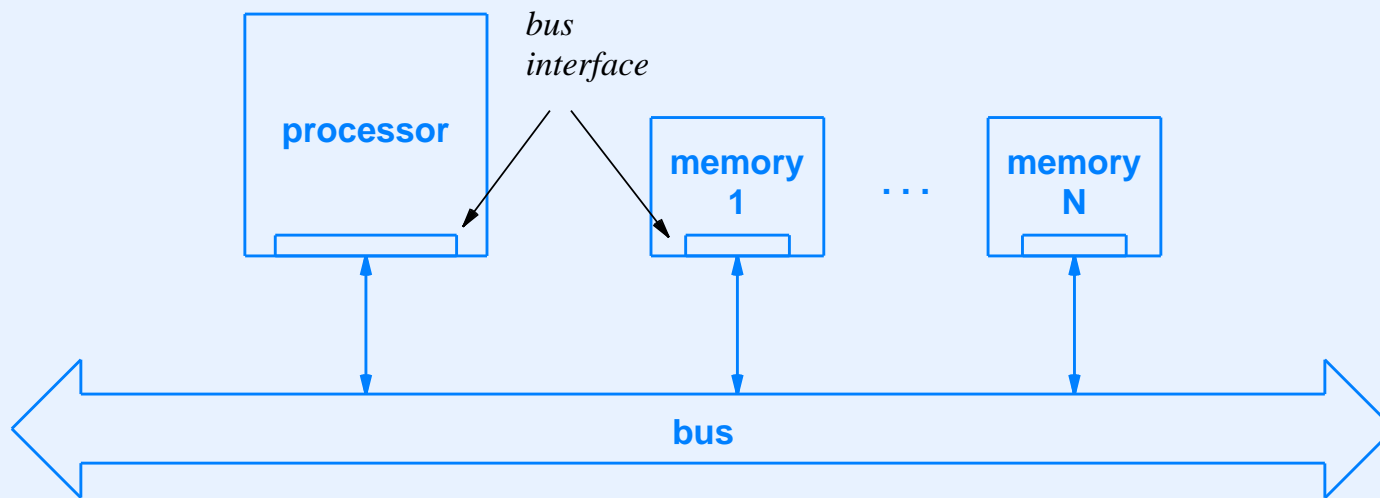
- Bus defines an *address space*

Control Hardware And Addresses

Although bus interface hardware receives all requests that pass across the bus, the interface only responds to requests that contain an address for which the interface has been configured.

Example Of Steps A
Memory Interface Takes

Let R be the range of addresses assigned to the memory
Repeat forever {
    Monitor the bus until a request appears;
    if (the request specifies an address in range R) {
      respond to the request
    } else {
      ignore the request
    }
}

Potential Errors On A Bus

- Address conflict

  - Two devices attempt to respond to a given address

- Unassigned address

  - No device responds to a given address

Address Configuration And Sockets

- Two options for address configuration

    - Configure each interface with set of addresses

    - Arrange sockets so that wiring limits each socket to a range of addresses

- Latter avoids misconfiguration: owner can plug in additional boards without configuring the hardware

- Note: some systems allow MMU to detect and configure boards automatically

Example Of Using Fetch-Store With Devices

- Imagine a device with lights used to display status

    – Contains sixteen separate lights

    – Connects to 32-bit bus

- Desired functions are

    – Turn the display on

    – Turn the display off

    – Set the display brightness

    – Turn the ith status light on or off

Example Of Meaning Assigned To Addresses

- Device designer chooses semantics for *fetch* and *store*

- Example

Address	Oper.	Meaning
100 – 103	store	nonzero data value turns the display on, and a zero data value turns the display off
100 – 103	fetch	returns zero if display is currently off, and nonzero if display is currently on
104 – 107	store	Change brightness. Low-order four bits of the data value specify brightness value from zero (dim) through sixteen (bright)
108 – 111	store	The low order sixteen bits each control a status light, where a zero bit sets the corresponding light off and one sets it on.

Interpretation Of Operations

- Semantics are

    if (address == 100 && op == store && data != 0)
        turn_on_display ;

- And

    if (address == 100 && op == store && data == 0)
        turn_off_display ;

Asymmetry

- *Fetch* and *store* operations

  – Are defined independently

  – Do not always mean "fetch data" or "store data"

- Note: operations do not need to be defined for all addresses

Unification Of Memory And Device Addressing

- Single bus can attach

    - Multiple memories

    - Multiple devices

- Bus address space includes all units

Example System

- Bus connects processor to

  - Multiple physical memory units

  - Multiple I/O devices

- Architectural illustration
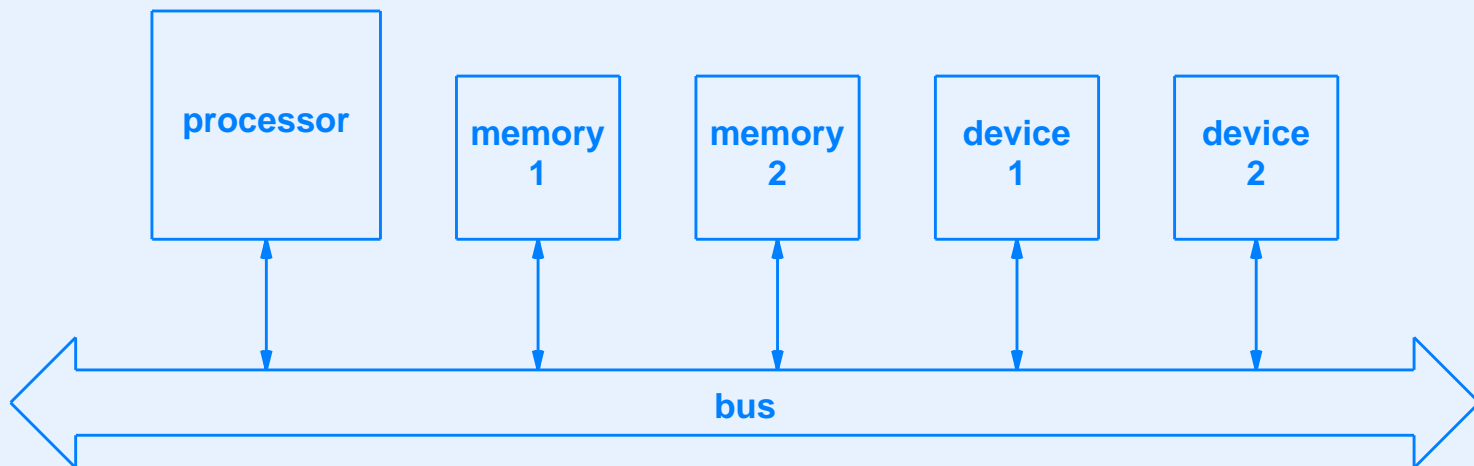
Address Assignments
For Example System

Device	Address Range		
Memory 1	0x000000	through	0x0fffff
Memory 2	0x100000	through	0x1fffff
Device 1	0x200000	through	0x20000b
Device 2	0x20000c	through	0x200017

Illustration Of Bus Address Space For Example System
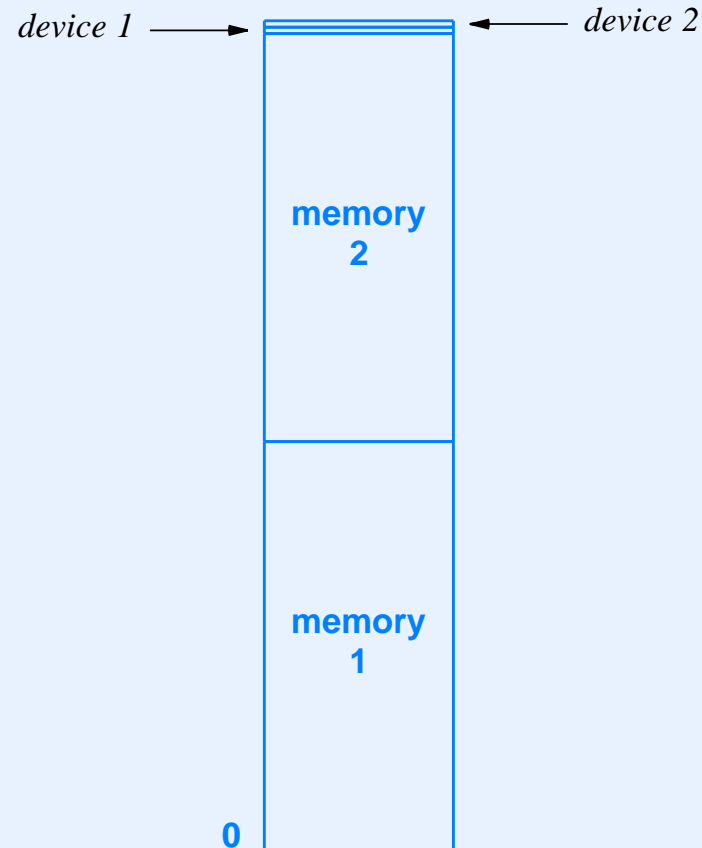
- Bus address space may contain *holes*

Example Address Map For 16-Bit Bus

0xffff	**available for devices**
0xdfff	*Hole (not available)*
0xbfff	**available for memory**
0x7fff	*Hole (not available)*
0x3fff	**available for memory**
0x0000	
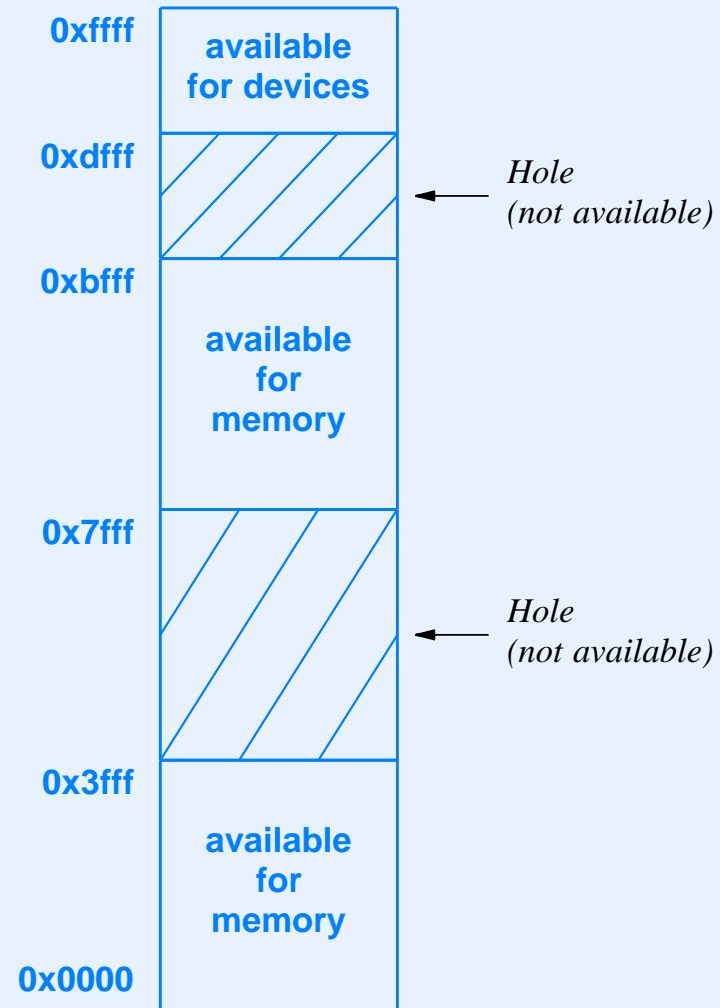
A Note About The Bus Address Space

In a typical computer, the part of the address space available to devices is sparsely populated — only a small percentage of possible addresses are used.

Example Code To Manipulate A Bus

- Software such as an OS that has access to the bus address space can fetch or store to a device

- Example code:

```
int *p;              /* declare p to be a pointer to an integer */

p = (*int)100;      /* set pointer to address 100 */

*p = 1;             /* store nonzero value in addresses 100 - 103 */
```

A Note About Programming With Multiple Buses

A processor that has multiple buses provides special instructions to access each; a processor that has one bus interprets normal memory operations as referencing locations in the bus address space.

Illustration Of Bridge Between Two Buses

- Bus interconnection device is called a *bridge*
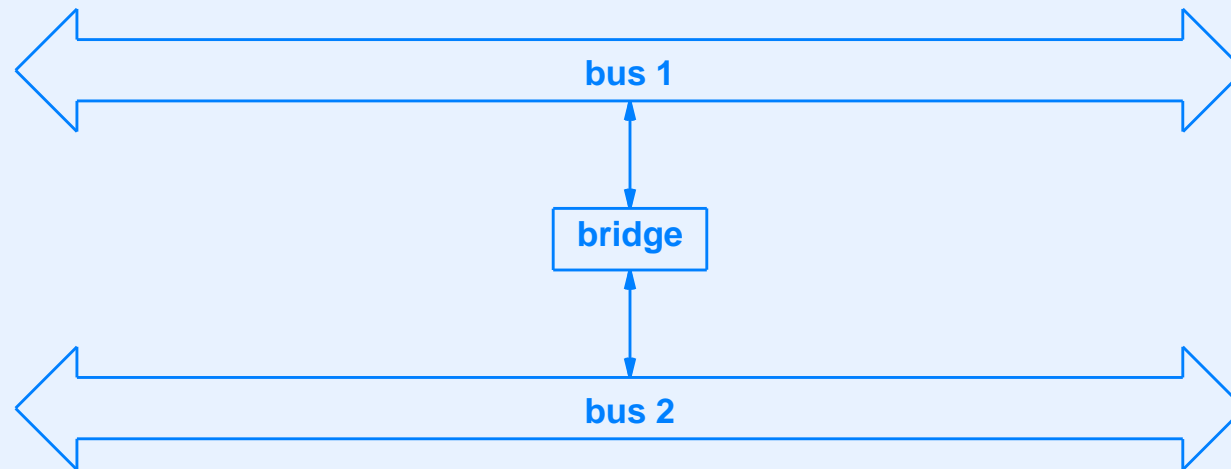
- Maps range of addresses

- Forwards operations and replies from one bus to the other

- Especially useful for adding an auxiliary bus

Illustration Of Address Mapping
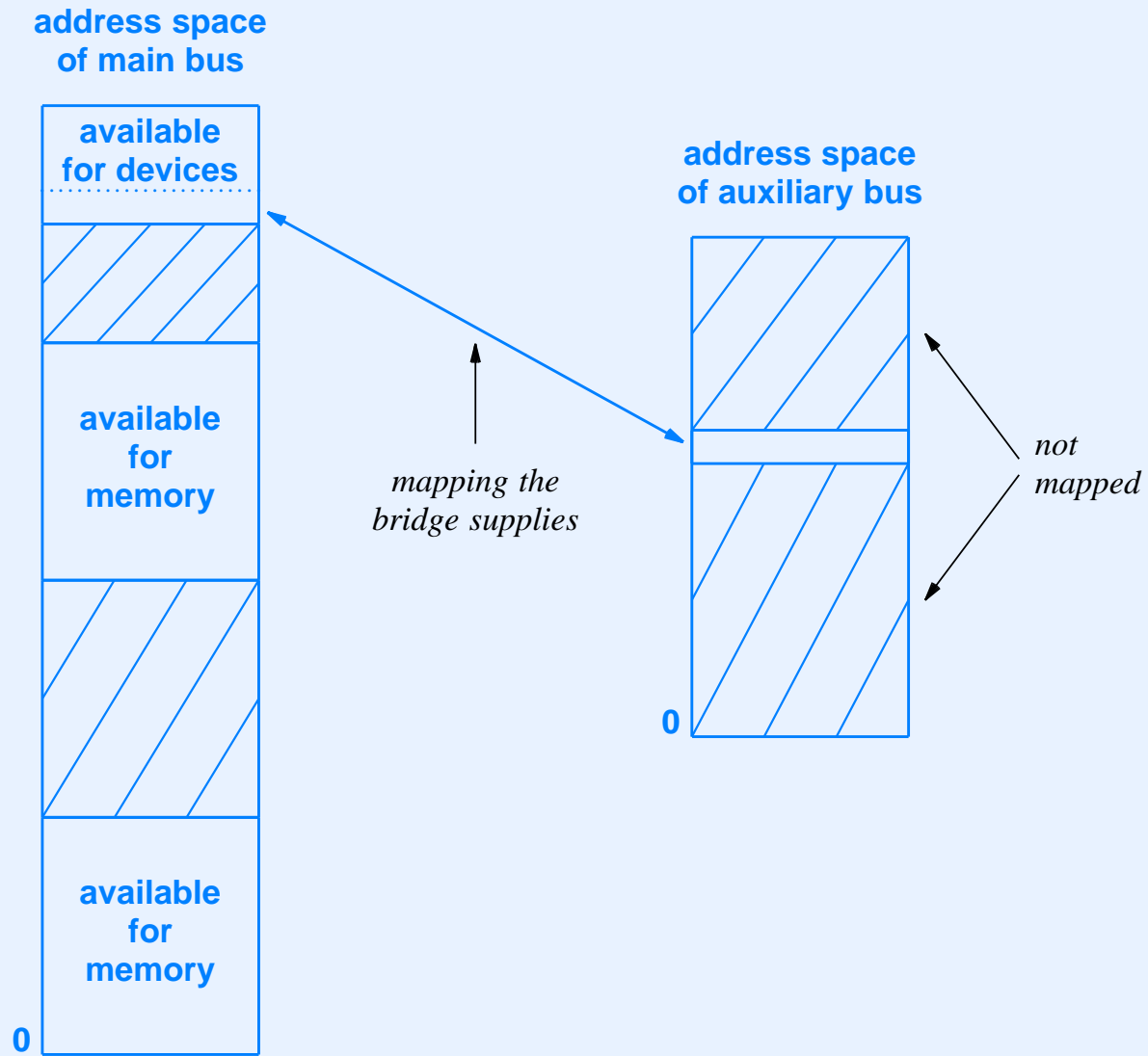
Switching Fabric

- Alternative to bus

- Connects multiple devices

- Sender supplies data and destination device

- Fabric delivers data to specified destination

Conceptual Crossbar Fabric
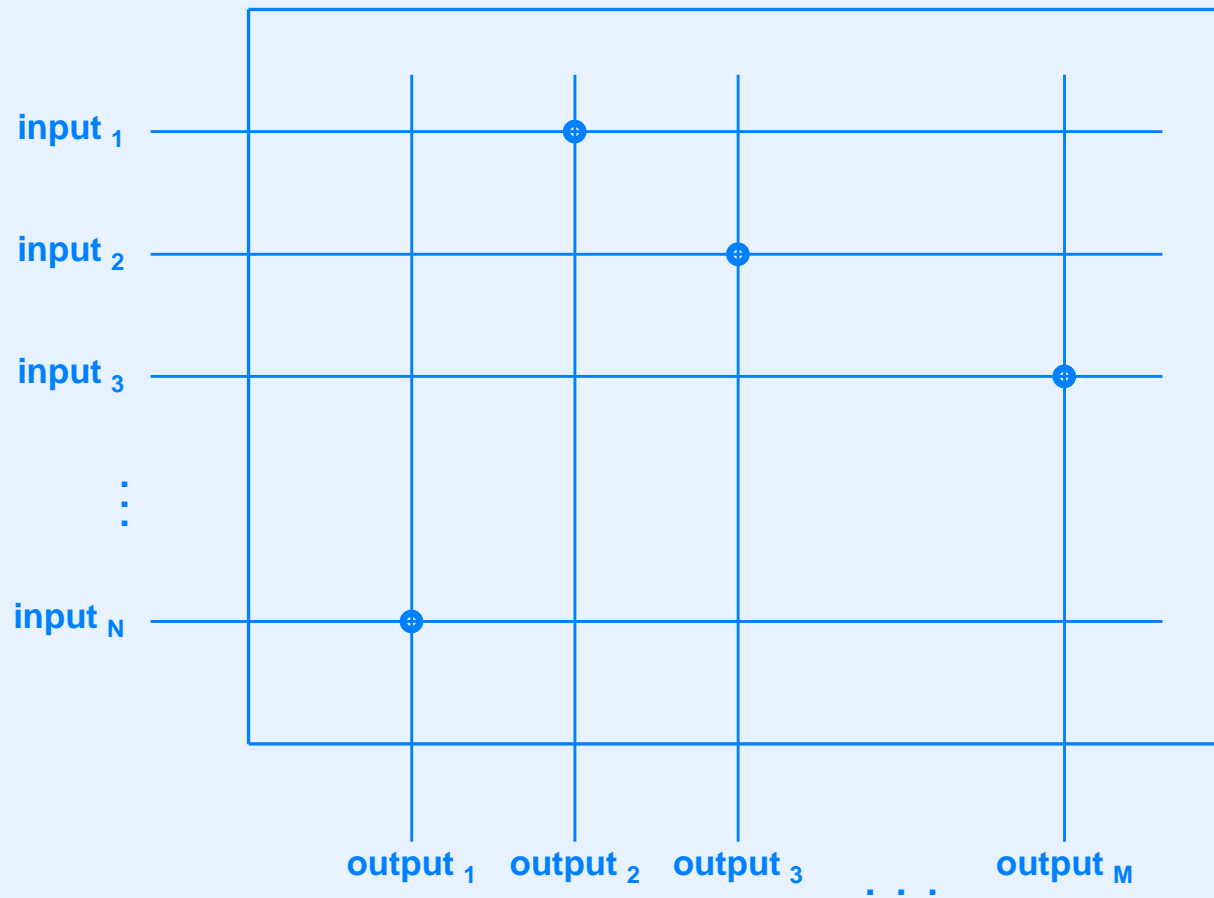
- Solid dot indicates a connection

Summary

- Bus is fundamental mechanism that interconnects

    – Processor

    – Memory

    – I/O devices

- Bus uses fetch-store paradigm for all communication

- Each unit assigned set of addresses in bus address space

- Bus address space can contain holes

- Bridge maps subset of addresses on one bus to another bus

Summary
(continued)

- Programmer uses conventional memory address mechanism to communicate over a bus

- Switching fabric is alternative to bus that allows parallelism

Questions?

XV

Programmed
And
Interrupt-driven
I / O

Two Basic Approaches To I/O

- Programmed I/O

- Interrupt-driven I/O

Programmed I/O

- Used in earliest computers and smallest embedded systems

- CPU does all the work

- Device has no intelligence (called *dumb*)

- Processor

  – Handles all synchronization

  – Is much faster than device

  – Starts operation on device

  – Waits for device to complete

Polling

- Technique used when processor waits for a device

- Processor executes a loop

    – Repeatedly requests status from device

    – Loop continues until device indicates "ready"

Example Of Polling

- Cause the printer to advance the paper

- Poll to determine when paper has advanced

- Move the print head to the beginning of the line

- Poll to determine when the print head reaches the beginning of the line

- Specify a character to print

- Start the ink jet spraying ink

- Poll to determine when the ink jet has finished

- Cause the print head to move to the next character position

. . . Continue with each successive character

Example Specification Of Addresses
Used For Device Polling

Addresses	Oper.	Meaning
0 through 3	store	Nonzero starts paper advance
4 through 7	store	Nonzero starts head moving to left margin
8 through 11	store	Character to print (low-order byte)
9 through 12	store	Nonzero starts inkjet spraying
13 through 16	fetch	Busy: nonzero when device is busy

Example C Code For Device Polling

```
int        *p;              /* declare an integer pointer */

p = 0x110000;               /* point to lowest address of device */
*p = 1;                     /* start paper advance */
while (*(p+4) != 0)         /* poll for paper advance */
        ;
*(p+1) = 1;                 /* start print head moving */
while (*(p+4) != 0)         /* poll for print head movement */
        ;
*(p+2) = 'C';              /* select character "C" */
while (*(p+4) != 0)         /* poll for character selection */
        ;
*(p+3) = 1;                 /* start inkjet spraying */
while (*(p+4) != 0)         /* poll for inkjet */
        ;
```

- Note: code does *not* contain any infinite loops!

C Code Rewritten To Use A Struct

```
struct    dv        {                    /* device control structure */
          int       d_adv;               /* nonzero starts paper advance */
          int       d_strt;              /* nonzero starts head moving */
          int       d_char;              /* character to print */
          int       d_strk;             /* nonzero starts inkjet spraying */
          int       d_busy;             /* nonzero when device busy */
}
struct    dv        *p;                   /* pointer to use */

p = (struct dv *)0x110000;  /* initialize pointer */

p->d_adv = 1;                            /* start paper advance */
while (p->d_busy)   ;                    /* poll for paper advance */

p->d_strt = 1;                           /* start print head moving */
while (p->d_busy)   ;                    /* poll for print head movement */

p->d_char = 'C';                         /* select character "C" */
while (p->d_busy)   ;                    /* poll for character selection */

p-> = 1;                                 /* start inkjet spraying */
while (p->d_busy)   ;                    /* poll for inkjet */
```

Control And Status Registers

- Terminology for the set of bus addresses a device uses

- Abbreviated *CSRs*

- Each CSR can respond to

  - *Fetch* operation

  - *Store* operation

  - Both

- Individual CSR bits may be assigned meaning

- Operations on CSRs control the device

A Note About Polling And Speed

Because a typical processor is much faster than an I/O device, the speed of a system that uses polling depends only on the speed of the I/O device; using a fast processor will not increase the rate at which I/O is performed.

- Bottom line: polling wastes processor cycles

Generations Of Computers

Generation	Description
1	Vacuum tubes used to build digital circuits
2	Transistors used to build digital circuits
3	Interrupt mechanism used to control I/O

Interrupt-Driven I/O

- Eliminates polling

- Allows processor to perform computation *while* I/O occurs

- Affects

  - I/O device hardware

  - Bus architecture and functionality

  - Processor architecture

  - Programming paradigm

Bus Architecture For Interrupts

- Must support two-way communication

- Processor controls device

- Device informs processor when task is complete

Programming Paradigms

- Polling uses *synchronous* paradigm

    – Code is sequential

    – Programmer includes device polling for each I/O operation

- Interrupts use *asynchronous* paradigm

    – Device temporarily "interrupts" processor

    – Processor services device and returns to computation in progress

    – Programmer creates separate piece of software to handle interrupts

Hardware Interrupt Mechanism

As the name implies, an interrupt mechanism temporarily borrows the processor to handle an I/O device. When an interrupt occurs, the hardware saves the state of the computation, and restarts the computation when interrupt processing finishes.

Fetch-Execute Cycle With Interrupts

Repeat forever {

Test: if any device has requested interrupt, handle the interrupt and then continue with the next iteration of the loop.

Fetch: access the next step of the program from the location in which the program has been stored.

Execute: Perform the step of the program.
}

- Note: interrupt appears to occur *between* two instructions

Handling An Interrupt

- Save the current execution state

- Determine which device interrupted

- Call the procedure that handles the device

- Clear the interrupt signal on the bus

- Restore the current execution state

Saving And Restoring State

- Processor

    – Saves state when interrupt occurs

    – Provides a *return from interrupt* instruction to restore hardware state

- State includes

    – Values in registers

    – Program counter

    – Condition code

- Usually, a *return from interrupt* accepts the same format the processor uses when saving state

Interrupt Vectors

- Array of addresses

- Stored at known location

- Point to software handler for each of the devices

- Used when device interrupts

- When device i interrupts, hardware follows pointer i

Illustration Of Interrupt Vectors
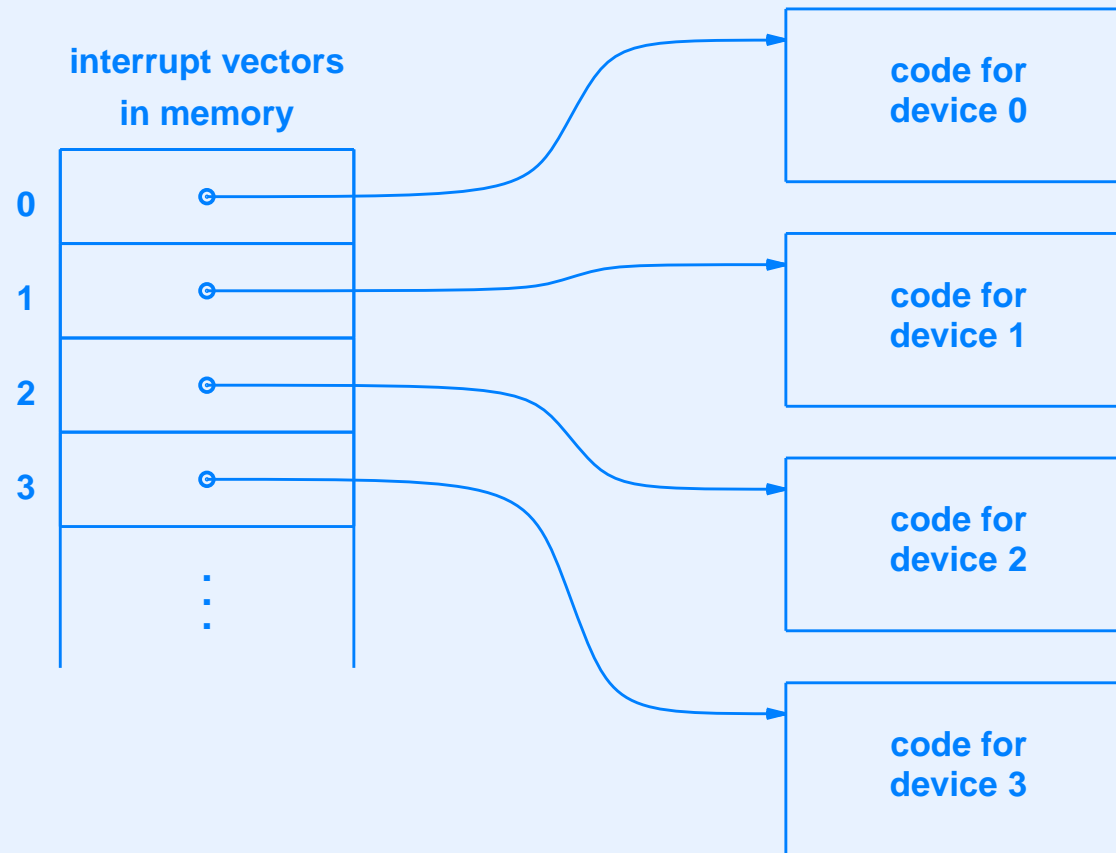
Initialization Of Interrupt Vectors

- Performed by software

- Usually performed by operating system

- Notes

  – Processor mode determines whether interrupts are permitted

  – Processor begins running with interrupts *disabled*

  – After interrupts initialized, operating system *enables* interrupts

Preventing Interrupt Code From Being Interrupted

- Multiple devices can interrupt

- Need to prevent simultaneous interrupts

- Technique: temporarily *disable* further interrupts while handling an interrupt

- Usually occurs when state saved / restored

- Consequence: only one interrupt can occur at any time

Multiple Interrupt Levels

- Advanced technique used in systems with many devices

- Assigns each device a priority level: devices that need faster service are assigned a higher priority

- General rule: at most one device at each level can be interrupting

- Important in real-time systems

- Note: lowest priority (usually zero) used when executing an application program

Priority Rule

When operating at priority level K, a processor can only be interrupted by a device that has been assigned to level K+1 or higher.

Interrupt Assignments

- Each device assigned unique interrupt vector

- Usually, device is assigned a small integer (think of it as an index into the interrupt vector array)

- Possibility 1: fixed, manual assignment

    – Tedious and prone to human error

    – Used on small, embedded systems

- Possibility 2: automated assignment at system startup

    – Flexible and less error prone

    – Only feasible if hardware allows each device to be assigned a unique number dynamically

Automated Interrupt Assignment

- Only possible if devices are *smart*

- Handled at system startup, usually by the operating system

- Either

    – Processor probes devices on the bus

    – Devices notify the processor of their presence

Dynamic Bus Connections And Pluggable Devices

- Some bus architectures allow devices to be connected at run-time

- Example *Universal Serial Bus* (*USB*)

- Single hardware bus controller handles interrupts for all USB devices

- Software to handle specific device linked at run-time

- No need for separate hardware interrupt vector

Advantage Of Interrupts

A computer that uses interrupts is both easier to program and offers better I/O performance than a computer that uses polling.

Dumb Device

- Processor performs all the work

- Example of interaction

  – Processor starts the disk spinning

  – Disk interrupts when it reaches full speed

  – Processor starts disk arm moving to the desired location

  – Disk interrupts when arm is in position

  – Processor starts a *read* operation to transfer data to memory

  – Disk interrupts when the transfer completes

Smart Devices

- Device contains embedded processor

- Offloads work from CPU

- Allows each device to operate independently

- Improves performance of both I/O and processor

- Example of interaction

  - Processor requests a *read* operation by specifying the location on the disk and the location in memory

  - Disk performs all steps of the operation and interrupts when the operation completes

Further I/O Optimizations

- Direct Memory Access (DMA)

- Buffer Chaining

- Operation Chaining

Direct Memory Access (DMA)

- Important optimization

- Needed for high-speed I/O

- Device moves data across the bus to/from memory without using processor

- Requires smart device

- Example: disk device can read an entire block and place in a specified buffer in memory

Buffer Chaining

- Handles multiple transfers without the processor

- Device given linked list of buffers

- Device hardware uses next buffer on list automatically
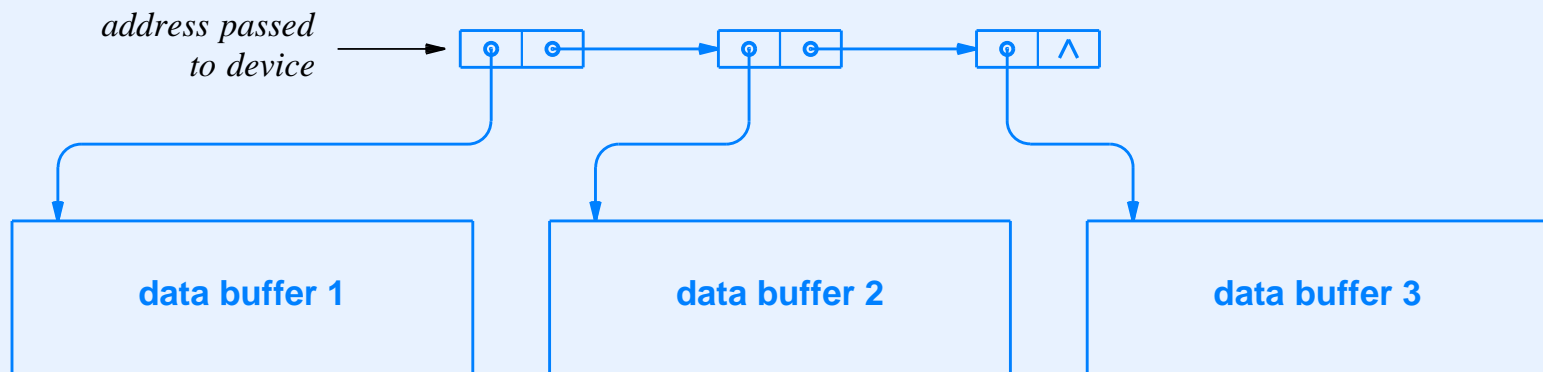
Scatter Read And Gather Write

- Special case of buffer chaining

- Large data transfer formed from separate blocks

- Example: to write a network packet, combine packet header from buffer 1 and packet data from buffer 2

- Eliminates application program from copying data into single, large buffer

Operation Chaining

- Further optimization for smart device

- Processor gives series of commands to device

- Device carries out successive commands automatically

- Illustration
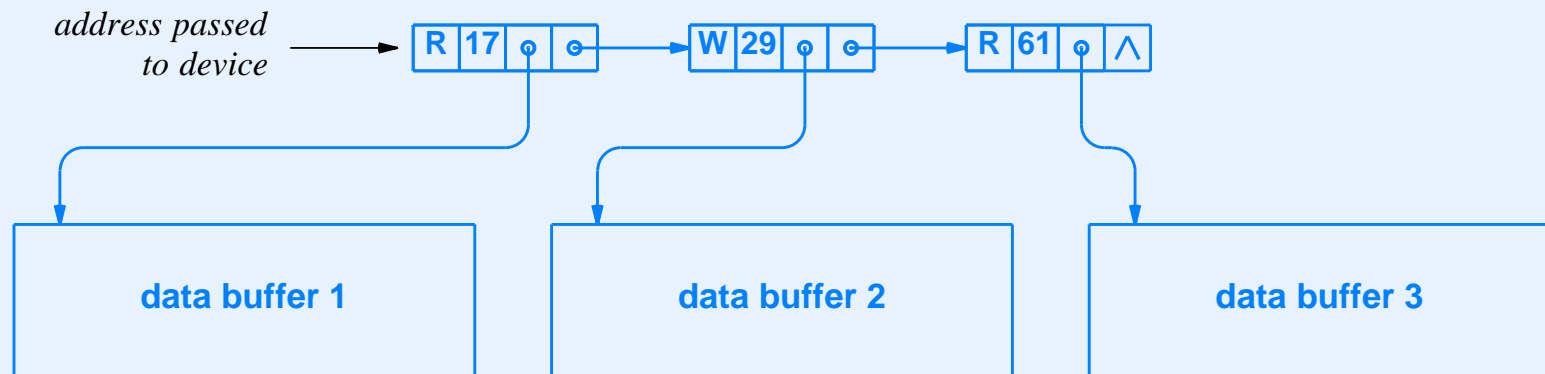
Summary

- Devices can use

    - Programmed I/O

    - Interrupt-driven I/O

- Interrupts

    - Introduced in third-generation computers

    - Allow processor to continue running while waiting for I/O

    - Use vector (usually in memory)

    - Occur "between" instructions in fetch-execute cycle

Summary
(continued)

- Multi-level interrupts handle high-speed and low-speed devices on same bus

- Smart device has some processing power built-in

- Optimizations include

    – Direct Memory Access (DMA)

    – Buffer chaining

    – Operation chaining

Questions?

XVI

A Programmer's View
Of I / O
And
Buffering

Device Driver

- Piece of software

- Responsible for communicating with specific device

- Usually part of operating system

- Classified as *low-level code*

- Performs basic functions

    - Manipulates device's CSRs to start operations when I/O is needed

    - Handles interrupts from device

Purposes Of Device Driver

- Device independence: application is not written for specific device(s)

- Encapsulation and hiding: details of device hidden from other software

Conceptual Parts Of A Device Driver

- Lower half

    – Handler code that is invoked when the device interrupts

    – Communicates directly with device (e.g., to reset hardware)

- Upper half

    – Set of functions that are invoked by applications

    – Allows application to request I/O operations

- Shared variables

    – Used by both halves to coordinate

Illustration Of Device Driver Organization

applications programs

upper half
invoked by
applications

shared
variables

lower half
invoked by
interrupts

device hardware
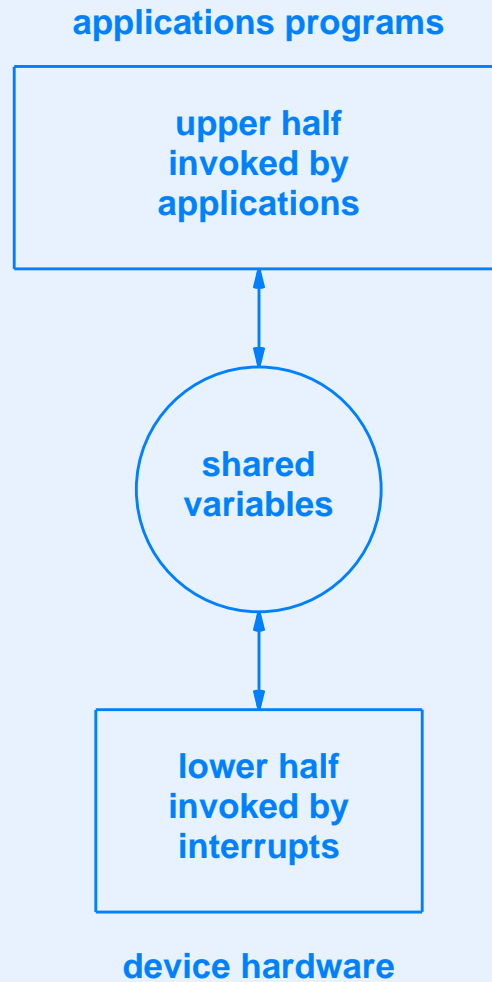
Types Of Devices

- Character-oriented

  – Transfer one byte at a time

  – Examples

    * Keyboard

    * Mouse

- Block-oriented

  – Transfer block of data at a time

  – Examples

    – Disk

    – Network interface

Example Flow Through A Device Driver

computer
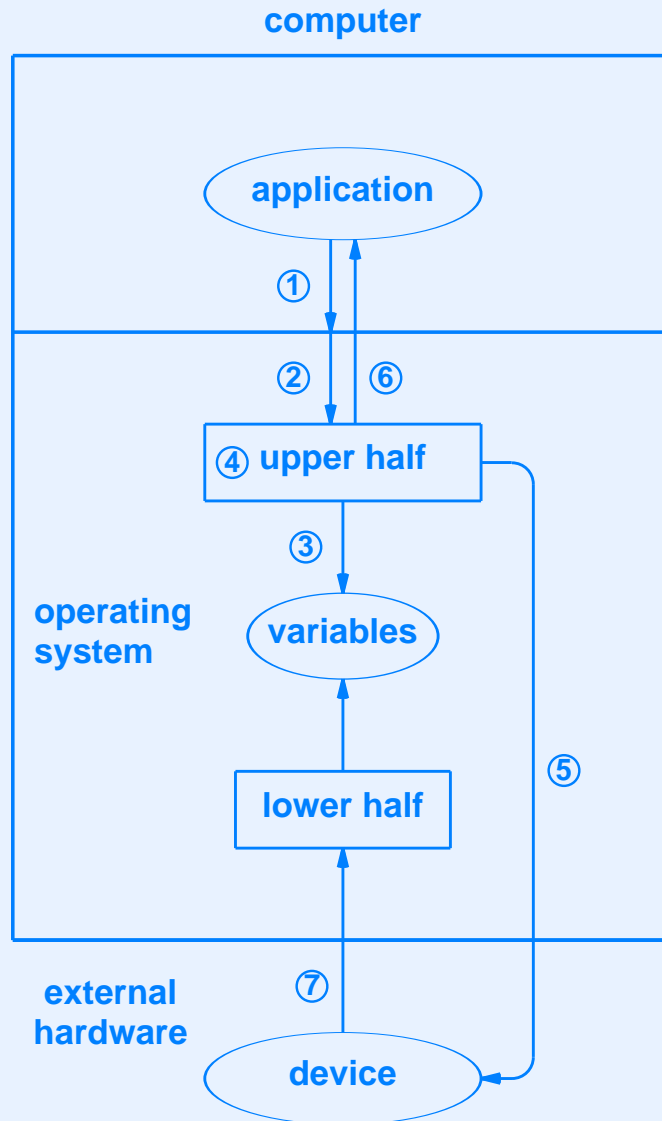
Steps Taken

1. The application writes data

2. The OS passes control to the driver

3. The driver records information

4. The driver waits for the device

5. The driver starts the transfer

6. The driver returns to the application

7. The device interrupts

Queued Output Operations

- Used by most device drivers

- Shared variable area contains queue of requests

- Upper-half places request on queue

- Lower-half moves to next request on queue when an operation completes

Illustration Of A Device Driver Request Queue
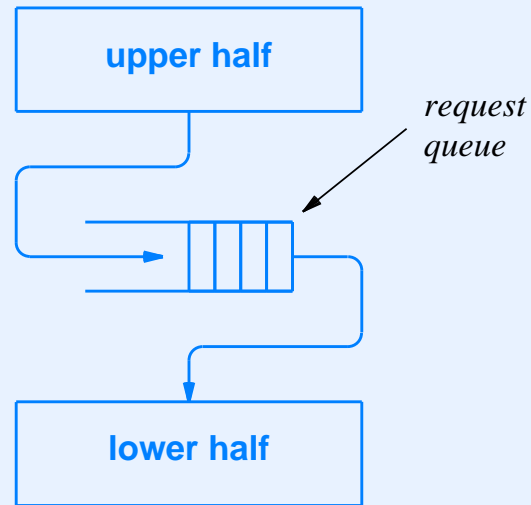
- Queue is shared among both halves

- Each half must insure that the other half will not attempt to examine or change the queue at the same time

Steps Taken On Output

- Initialization (computer system starts)

    – Initialize input queue to empty

- Upper half (application performs *write*)

    – Deposit data item in queue

    – Force the device to interrupt

    – Return to application

- Lower half (interrupt occurs)

    – If the queue is empty, stop the device from interrupting

    – If the queue is nonempty, extract the next item from the queue and start output

    – Return from interrupt

Forcing An Interrupt

- Many devices have a CSR bit, B, that can be used to force the device to interrupt

- If the device is idle, setting bit B causes the device to generate an interrupt

- If the device is currently performing an operation, setting bit B has no effect

- Above makes device driver code especially elegant

Queued Input Operations

- Initialization (computer system starts)

    – Initialize input queue to empty

    – Force the device to interrupt

- Upper half (application performs *read*)

    – If input queue is empty, temporarily stop the application

    – Extract the next item from the input queue

    – Return the item to the application

- Lower half (interrupt occurs)

    – If the queue is not full, start another input operation

    – If an application is stopped waiting for input, allow the application to run

    – Return from interrupt

Devices That Support Bi-Directional Transfer

- Most devices include two-way communication

- Example: although printer is primarily an output device, most printers allow the processor to check status

- Drivers can

  - Treat device as two separate devices, one used for input and one used for output

  - Treat the device as a single device that handles two types of commands, one for input and one for output

Asynchronous Vs. Synchronous Paradigm

- Synchronous programming

    – Used for many applications

    – Processor follows single path through the code

- Asynchronous programming

    – Used for interrupts

    – Programmer writes set of handlers

    – Each handler invoked when corresponding event occurs

    – More challenging than synchronous programming

- Device drivers use the asynchronous paradigm

Mutual Exclusion

- Needed when events occur asynchronously

- Guarantees only one operation will be performed at a time

- For device drivers: must provide mutual exclusion between processor and smart device that can each change shared data

I/O Interface For Applications

- Few programmers write device drivers

- Most programmers use high-level abstractions

    – Files

    – Windows

    – Documents

- Compiler generates calls to *run-time library* functions

- Chief advantage: I/O hardware and/or device drivers can change without changing applications

Conceptual Arrangement Of Library And OS
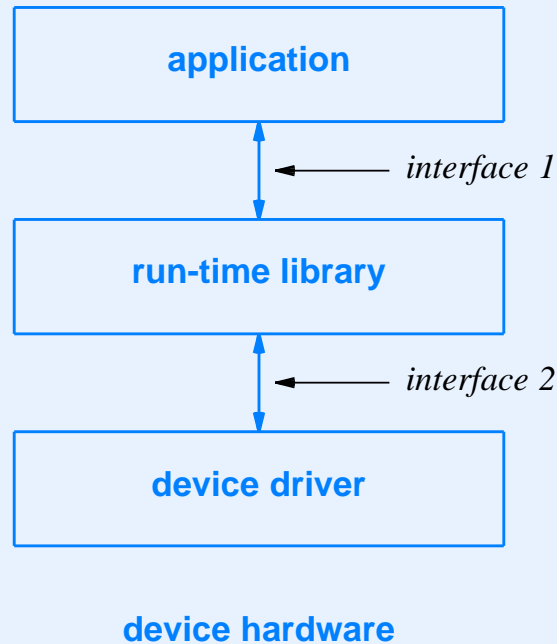
- Example

  - Interface 1: standard I/O library

  - Interface 2: system calls in the Unix kernel

Example Of Interfaces

- UNIX library functions

Operation	Meaning
printf	Generate formatted output from a set of variables
fprintf	Generate formatted output for a specific file
scanf	Read formatted data into a set of variables

- UNIX system calls

Operation	Meaning
open	Prepare a device for use (e.g., power up)
read	Transfer data from the device to the application
write	Transfer data from the application to the device
close	Terminate use of the device
seek	Move to a new location of data on the device
ioctl	Misc. control functions (e.g., change volume)

Reducing The Cost Of I/O Operations

- Two principles

The overhead involved in using a system call to communicate with a device driver is extremely high; a system call is much more expensive than a conventional procedure call, such as the call used to invoke a library function.

To reduce overhead and optimize I/O performance, a programmer must reduce the number of system calls that an application invokes. The key to reducing system calls involves transferring more data per call.

Buffering

- Important optimization

- Used heavily

- Automated and usually invisible to programmer

- Key idea: make large I/O transfers

  - Accumulate outgoing data before transfer

  - Transfer large block of incoming data and then extract individual items

Hiding Buffering From Programmers

- Typically performed with *library functions*

- Application

    – Uses functions in the library for all I/O

    – Transfers data in arbitrary size blocks

- Library functions

    – Buffer data from applications

    – Transfer data to underlying system in large blocks

Example Library Functions For Output

Operation	Meaning
setup	Initialize the buffer
input	Perform an input operation
output	Perform an output operation
terminate	Discontinue use of the buffer
flush	Force contents of buffer to be written

- Note: an operating system may also perform buffering

Use Of Library

- *Setup*

  – Called to initialize buffer

  – May allocate buffer

  – Typical buffer sizes 8K to 128K bytes

- *Output*

  – Called when application needs to emit data

  – Places data item in buffer

  – Only writes to I/O device when buffer is full

- *Terminate*

  – Called when all data has been emitted

  – Forces remaining data to be written

Implementation Of Output Buffer Functions

Setup(N)

    1. Allocate a buffer of N bytes.

    2. Create a global pointer, p, and initialize p to the address of the first byte of the buffer.

Output(D)

    1. Place data byte D in the buffer at the position given by pointer p, and move p to the next byte.

    2. If the buffer is full, make a system call to write the contents of the entire buffer, and reset pointer p to the start of the buffer.

Implementation Of Output Buffer Functions
(continued)

Terminate

    1. If the buffer is not empty, make a system call to write the contents of the buffer prior to pointer p.

    2. If the buffer was dynamically allocated, deallocate it.

Flushing An Output Buffer

- Allows a programmer to force data out

- Needed for interactive programs

- When *flush* is called

  – If buffer contains data, write to I/O device

  – If buffer is empty, *flush* has no effect

Implementation Of Flush And Terminate

Flush

1. If the buffer is currently empty, return to the caller without taking any action.

2. If the buffer is not currently empty, make a system call to write the contents of the buffer and set the global pointer p to the address of the first byte of the buffer.

Terminate

1. Call *flush* to insure that any remaining data is written.

2. Deallocate the buffer.

Summary Of Buffer Flushing

A programmer uses a flush *function to specify that outgoing data in a buffer should be sent to the device. A* flush *operation has no effect if a buffer is currently empty.*

Buffering On Input

Setup(N)

1. Allocate a buffer of N bytes.

2. Create a global pointer, p, and initialize p to indicate that the buffer is empty.

Input(N)

1. If the buffer is empty, make a system call to fill the entire buffer, and set pointer p to the start of the buffer.

2. Extract a byte, D, from the position in the buffer given by pointer p, move p to the next byte, and return D to the caller.

Terminate

1. If the buffer was dynamically allocated, deallocate it.

Important Note About Implementation

- Both input and output buffering are straightforward

- Only a trivial amount of code needed

Effectiveness Of Buffering

- Buffer of size *N* reduces number of system calls by a factor of *N*

- Example

  – Minimum size buffer is typically 8K bytes

  – Resulting number of system calls is $S/8192$, where S is the original number of system calls

Buffering In An Operating System

- Buffering is used extensively inside the OS

- Important part of device drivers

- Goal: reduce number of external transfers

- Reason: external transfers are slower than system calls

Relation Between Buffering And Caching

- Closely related concepts

- Chief difference

  – Cache handles random access

  – Buffer handles sequential access

Example Of I/O Functions That Buffer

- Standard I/O library in UNIX

Function	Meaning
fopen	Set up a buffer
fgetc	Buffered input of one byte
fread	Buffered input of multiple bytes
fwrite	Buffered output of multiple bytes
fprintf	Buffered output of formatted data
fflush	Flush operation for buffered output
fclose	Terminate use of a buffer

- Each function buffers extensively

- Dramatically improves I/O performance

Summary

- Two aspects of I/O pertinent to programmers

    - Device details important to systems programmers who write device drivers

    - Application programmer must understand relative costs of I/O

- Device driver divided into three parts

    - Upper-half called by application

    - Lower-half handles device interrupts

    - Shared data area accessed by both halves

Summary
(continued)

- Buffering

  – Fundamental technique used to enhance performance

  – Useful with both input and output

- Buffer of size N reduces system calls by a factor of N

Questions?

XVII

Parallelism

Techniques Used To Increase Performance

Techniques Used To Increase Performance

- Software:

Techniques Used To Increase Performance

- Software: many techniques available

  - Caching

  - Buffering

  - Ordering of references (e.g., in arrays)

  - Data placement

  - New algorithms

  - . . . and many more

Techniques Used To Increase Performance

- Software: many techniques available

  - Caching

  - Buffering

  - Ordering of references (e.g., in arrays)

  - Data placement

  - New algorithms

  - . . . and many more

- Hardware:

Techniques Used To Increase Performance

- Software: many techniques available

    - Caching

    - Buffering

    - Ordering of references (e.g., in arrays)

    - Data placement

    - New algorithms

    - . . . and many more

- Hardware: only two techniques

    - Parallelism

    - Pipelining

Parallelism

- Employs multiple copies of a hardware unit

- All copies can operate simultaneously

- General idea

    - Distribute data items among parallel hardware units

    - Gather (and possibly combine) results

- Occurs at many levels of architecture

- Term *parallel computer* applied when parallelism dominates the entire architecture

Characterizations Of Parallelism

- Microscopic vs. macroscopic

- Symmetric vs. asymmetric

- Fine-grain vs. coarse-grain

- Explicit vs. implicit

Microscopic Vs. Macroscopic Parallelism

Parallelism is so fundamental that virtually all computer systems contain some form of parallel hardware. We use the term microscopic parallelism *to characterize parallel facilities that are present, but not especially visible, and* macroscopic parallelism *to describe parallelism of which a programmer is aware.*

Examples Of Microscopic Parallelism

- Parallel operations in an ALU

- Parallel access to general-purpose registers

- Parallel data transfer to/from physical memory

- Parallel transfer across an I/O bus

Examples Of Macroscopic Parallelism

- Symmetric parallelism

    – Refers to multiple, identical processors

    – Example: computer with quad-core processor

- Asymmetric parallelism

    – Refers to multiple, dissimilar processors

    – Example: computer with a CPU and a graphics processor

Level Of Parallelism

- Fine-grain

    - Parallelism among individual instructions or data elements

- Coarse-grain parallelism

    - Parallelism among programs or large blocks of data

Explicit And Implicit Parallelism

- Explicit

  - Visible to programmer

  - Requires programmer to initiate and control parallel activities

- Implicit

  - Invisible to programmer

  - Hardware runs multiple copies of program or instructions automatically

Parallel Architectures

- Design in which computer has reasonably large number of processors

- Intended for *scaling*

- Example: computer with thirty-two processors

- Not generally classified as parallel computer

    – Computer with dual-core or quad-core processor

    – Computer with multiple hard drives

Types Of Parallel Architectures

Name	Meaning
SISD	Single Instruction Single Data stream
SIMD	Single Instruction Multiple Data streams
MIMD	Multiple Instructions Multiple Data streams

- Known as the *Flynn classification*

- Only provides a broad, intuitive classification

SISD: A Conventional (Nonparallel) Architecture

- Processor executes

  – One instruction at a time

  – Operation applies to one set of data items

- Synonyms include

  – *Sequential architecture*

  – *Uniprocessor*

SIMD: Single Instruction Multiple Data

- Each instruction specifies a single operation

- Hardware applies operation to multiple data items

- Example

  - Add operation performs pairwise addition on two one-dimensional arrays

  - Store operation can be used to clear a large block of memory

Vector Processor

- Special case of SIMD

- Usual focus is on floating point operations

- Applies a given operation to an entire array of values

- Example use: normalize values in a set

Normalization Example

- On a conventional computer

$$\text{for } i \text{ from } 1 \text{ to } N \; \{$$
$$V[i] \leftarrow V[i] \times Q;$$
$$\}$$

- On a vector processor

$$V \leftarrow V \times Q;$$

- Vector code is trivial (no iteration)

- Special *vector instruction* is invoked

- If vector *V* contains more items than the hardware has parallel execution units, multiple steps are required

Graphics Processors

- Graphics hardware uses sequential bytes in memory to store pixels

- To move a window, the image must be copied from one location in the frame buffer to another

- SIMD architecture allows copies to proceed in parallel

- Special-purpose graphics processors are available that offer parallel hardware for graphics operations

- Sometimes called a *Graphics Processing Unit* (*GPU*)

MIMD: Multiple Instructions Multiple Data

- Parallel architecture with many physical processors

- Each processor

  - Can run an independent program

  - May have dedicated I/O devices (e.g., its own disk)

- Visible to programmer

- Works best for applications where computation can be divided into separate, independent pieces

Two Popular Categories Of Multiprocessors

- Symmetric

- Asymmetric

Symmetric Multiprocessor (SMP)

- Most well-known MIMD architecture

- Set of *N* identical processors

- Examples of groups that built SMP computers

  - Carnegie Mellon University (C.mmp)

  - Sequent Corporation (Balance 8000 and 21000)

  - Encore Corporation (Multimax)

- Current example: Intel multicore designs

Illustration Of A Symmetric Multiprocessor
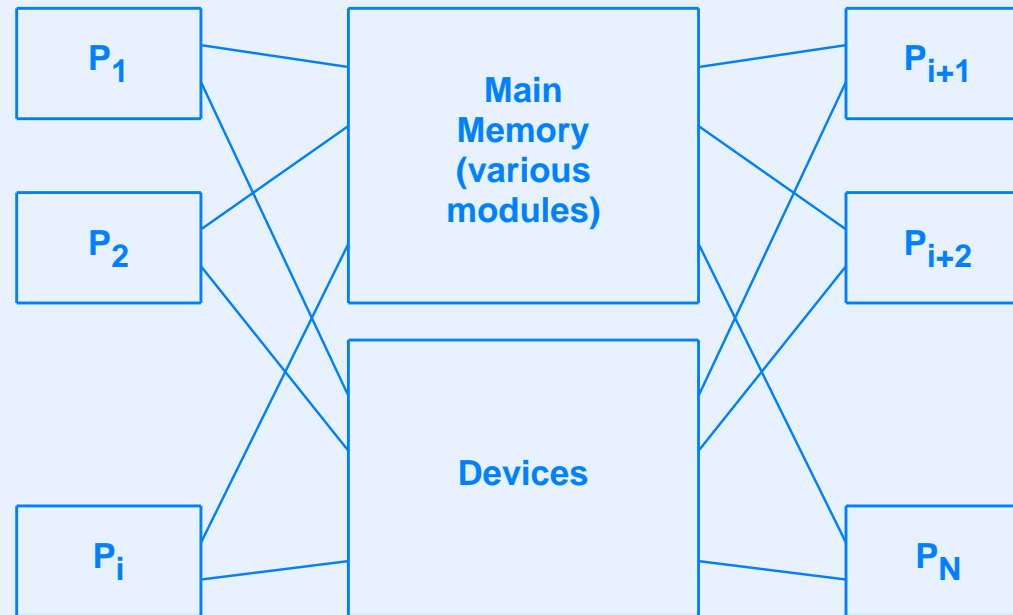
- Major problem with SMP architecture: *contention* for memory and I/O devices

- To improve performance: provide each processor with its own copy of a device

Asymmetric Multiprocessor (AMP)

- Set of *N* processors

- Multiple types of processors

- Processors optimized for specific tasks

- Often use master-slave paradigm

Example AMP Architectures

- Math (or graphics) coprocessor

  - Special-purpose processor

  - Handles floating point (or graphics) operations

  - Called by main processor as needed

- I/O Processor

  - Optimized for handling interrupts

  - Programmable

Examples Of Programmable I/O Processors

- Channel (IBM mainframe)

- Peripheral Processor (CDC mainframe)

Multiprocessor Overhead

- Having many processors is not always a clear win

- Overhead arises from

    - Communication

    - Coordination

    - Contention

Communication

- Needed

    - Among processors

    - Between processors and I/O devices

- As number of processors increases, communication becomes a bottleneck

Coordination

- Needed when processors work together

- May require one processor to wait for another to compute a result

- One possibility: designate a processor to perform coordination tasks

Contention

- Processors contend for resources

    - Memory

    - I/O devices

- Speed of resources can limit overall performance

    - Example: $N-1$ processors wait while one processor accesses memory

Performance Of Multiprocessors

- Has been disappointing

- Bottlenecks include

    – Contention for operating system (only one copy of OS can run)

    – Contention for memory and I/O

- Another problem: caching

    – One centralized cache means contention problems

    – Coordinated caches means complex interaction

- Many applications are I/O bound

According To John Harper

"Building multiprocessor systems that scale while correctly synchronising the use of shared resources is very tricky, whence the principle: with careful design and attention to detail, an N-processor system can be made to perform nearly as well as a single-processor system. (Not nearly N times better, nearly as good in total performance as you were getting from a single processor). You have to be very good — and have the right problem with the right decomposability — to do better than this."

http://www.john-a-harper.com/principles.htm

Definition Of Speedup

- Defined relative to single processor

$$Speedup \; = \; \frac{\tau_N}{\tau_1}$$

- τ_N denotes the execution time on a multiprocessor

- τ_1 denotes the execution time on a single processor

- Goal: speedup that is linear in number of processors

Ideal And Typical Speedup
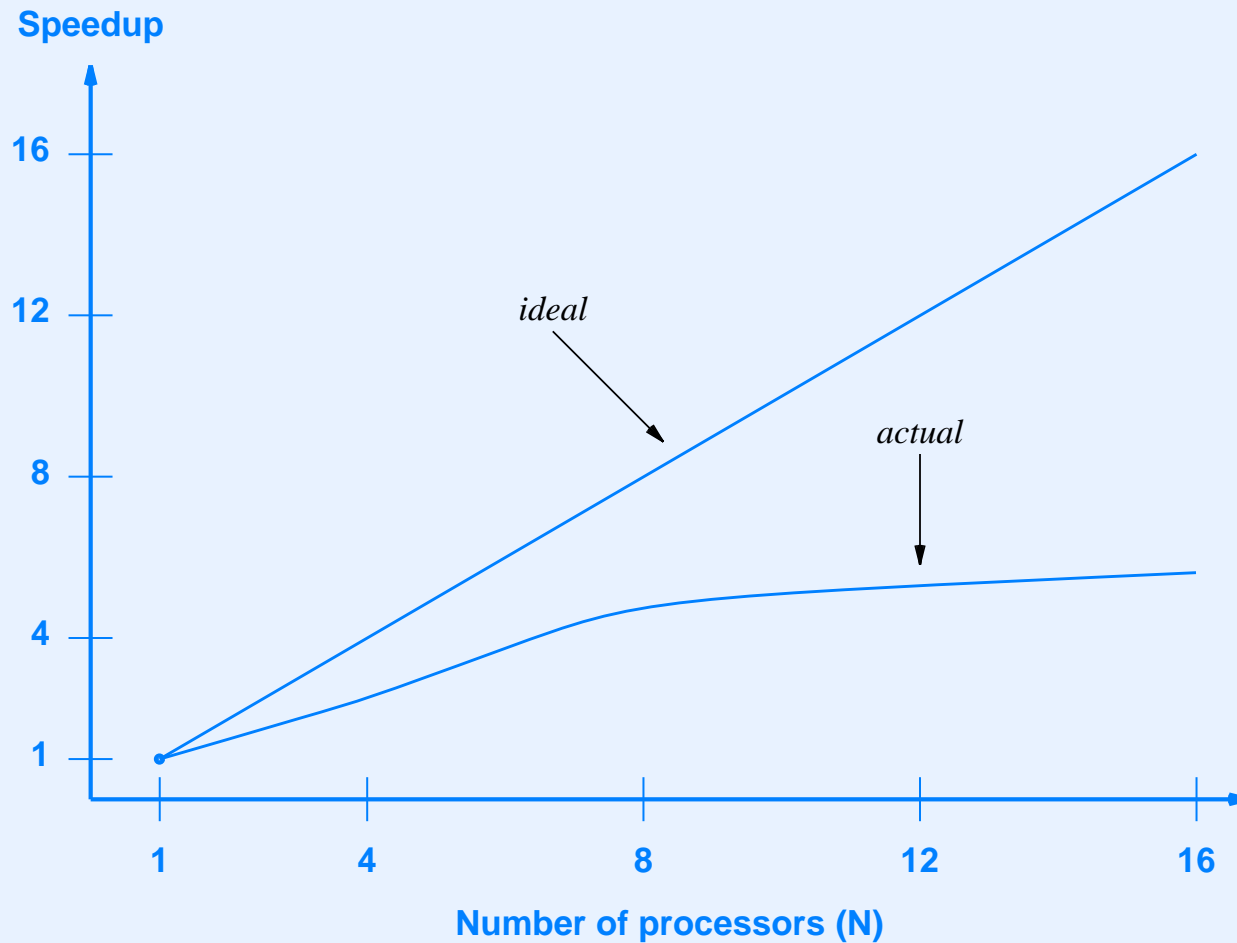
Speedup For N>>1 Processors
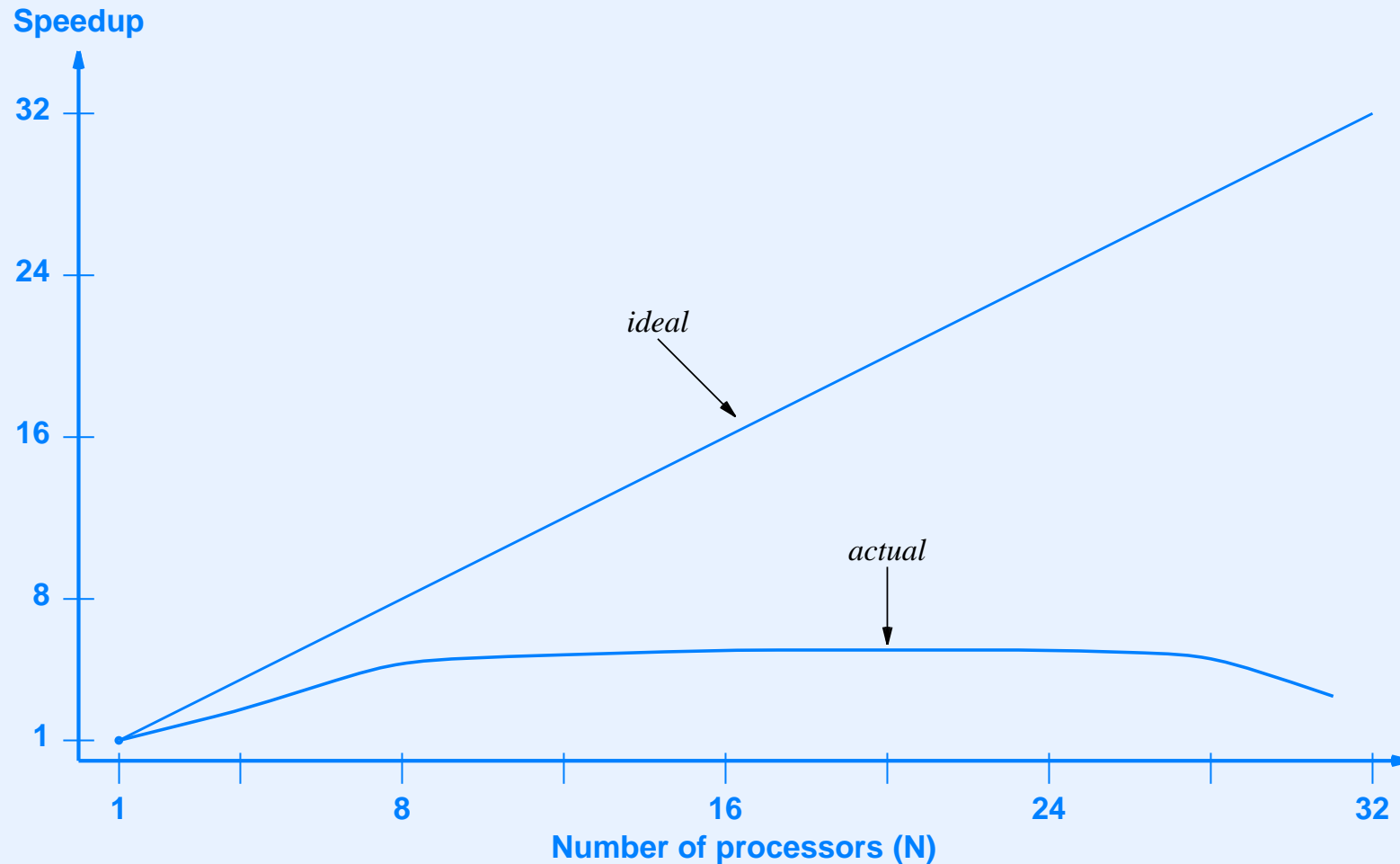
- At some point, performance begins to decrease!

Summary Of Speedup

When used for general-purpose computing, a multiprocessor may not perform well. In some cases, added overhead means performance decreases as more processors are added.

Consequences For Programmers

- Writing code for multiprocessors is difficult

    - Need to handle mutual exclusion for shared items

    - Typical mechanism: locks

The Need For Locking

- Consider a trivial assignment statement:

$$\{x \ = \ x \ + \ 1;\}$$

- Typical code might be something like this:

  load x, R5
  incr   R5
  store R5, x

- On a uniprocessor, no problems arise

- Consider a multiprocessor

The Need For Locking
(continued)

- Suppose two processors attempt to increment item x

- The following sequence can result

  - Processor 1 loads x into its register 5

  - Processor 1 increments its register 5

  - Processor 2 loads x into its register 5

  - Processor 1 stores its register 5 into x

  - Processor 2 increments its register 5

  - Processor 2 stores its register 5 into x

Hardware Locks

- Prevent simultaneous access

- Separate lock assigned to each item

- Each lock assigned an ID

- If lock 17 is used, code becomes

```
lock   17
load   x, R5
incr   R5
store  R5, x
release 17
```

- Hardware allows one processor to hold a given lock at a given time, and blocks others

Programming Parallel Computers

- Implicit parallelism

    – Programmer writes sequential code

    – Hardware runs many copies automatically

- Explicit parallelism

    – Programmer writes code for parallel architecture

    – Code must use locks to prevent interference

- Conclusion: explicit parallelism makes computers extremely difficult to program

The Point About Parallel Programming

From a programmer's point of view, a system that uses explicit parallelism is significantly more complex to program than a system that uses implicit parallelism.

Programming Symmetric And Asymmetric Multiprocessors

- Both types can be difficult to program

- Symmetric has two advantages

  - One instruction set

  - Programmer does not need to choose processor type for each task

- Asymmetric has an advantage

  - Programmer can use processor that is best-suited to a given task

  - Example: using a special-purpose graphics processor may be easier than implementing graphics operations on a standard processor

Redundant Parallel Architectures

- Used to increase reliability

- Do not improve performance

- Multiple copies of hardware perform *same* function

- Watchdog circuitry detects whether all units computed the same result

- Can be used to

    – Test whether hardware is performing correctly

    – Serve as backup in case of hardware failure

Loose And Tight Coupling

- *Tightly coupled multiprocessor*

    – Multiple processors in single computer

    – Buses or switching fabrics used to interconnect processors, memory, and I/O

    – Usually one operating system

- *Loosely coupled multiprocessor*

    – Multiple, independent computer systems

    – Computer networks used to interconnect systems

    – Each computer runs its own operating system

    – Known as *distributed computing*

Cluster Computer

- Special case of distributed computer system

- All computers work on a single problem

- Works best if problem can be partitioned into pieces

- Currently popular in large *data centers*

Grid Computing

- Form of loosely-coupled distributed computing

- Uses computers on the Internet

- Popular for large, scientific computations

- One application: *Search for Extra-terrestrial Intelligence* (*SETI*)

Summary

- Parallelism is fundamental

- Flynn scheme classifies computers as

    - SISD (e.g., conventional uniprocessor)

    - SIMD (e.g., vector computer)

    - MIMD (e.g., multiprocessor)

- Multiprocessors can be

    - Symmetric or asymmetric

    - Explicitly or implicitly parallel

- Multiprocessor speedup usually less than linear

Summary
(continued)

- Programming multiprocessors is usually difficult

    – Programmer must divide tasks onto multiple processors

    – Locks needed for shared items

- Parallel systems can be

    – Tightly-coupled (single computer)

    – Loosely-coupled (computers connected by a network)

Questions?

XVIII

Pipelining

Concept Of Pipelining

- One of the two major hardware optimization techniques

- Information flows through a series of stations (processing components)

- Each station can

    – Inspect

    – Interpret

    – Modify

- Station known as a *stage* of the pipeline

Illustration Of Pipelining

information arrives

information leaves

stage 1

stage 2

stage 3

stage 4
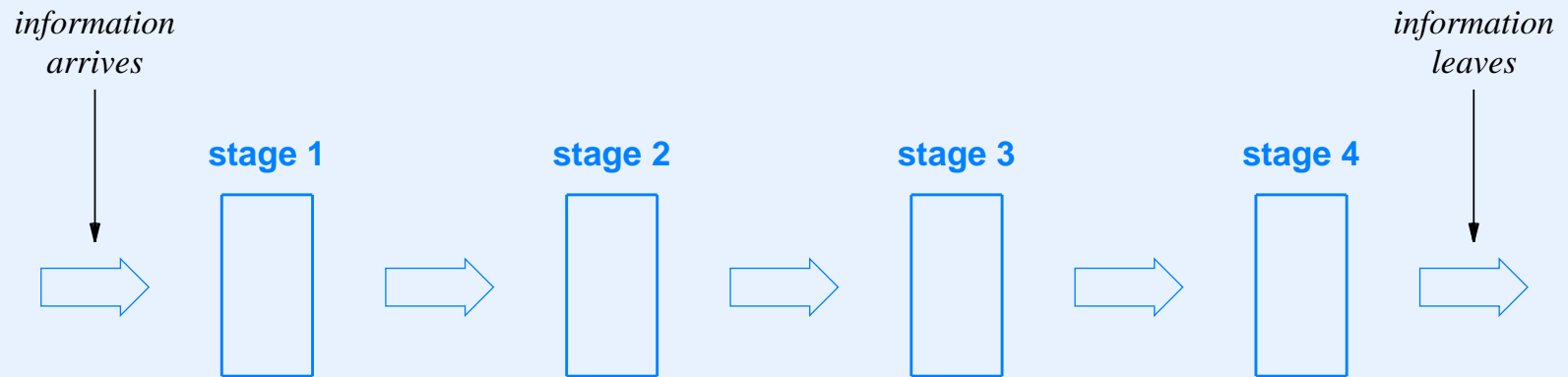
Characteristics Of Pipelines

- Hardware or software implementation

- Large or small scale

- Synchronous or asynchronous flow

- Buffered or unbuffered flow

- Finite chunks or continuous bit streams

- Automatic data feed or manual data feed

- Serial or parallel path

- Homogeneous or heterogeneous stages

Implementation

- Pipeline can be implemented in hardware or software

- Software pipeline

  - Programmer convenience

  - More efficient than intermediate files

  - Output from one process becomes input of another

- Hardware pipeline

  - Separate hardware units in each stage

  - Much higher performance

  - Stages can be tightly integrated (e.g., within a chip)

Scale

- Range of scales

- Example of small scale: pipeline within an ALU

- Example of large scale: pipeline composed of programs running on separate computers connected by the Internet

Synchrony

- Synchronous pipeline

    - Operates like an assembly line

    - Items move between stages at exactly the same time

    - Cannot work faster than slowest stage

- Asynchronous pipeline

    - Allows variable-processing time at each stage

    - Each station forwards whenever it is ready

    - Slow stage may block previous stages

Buffering

- Buffered flow

  – Buffer placed between each pair of stages

  – Useful when processing time per item varies

- Unbuffered flow

  – Stage blocks until next stage can accept item

  – Works best if processing time per stage is constant

Size Of Items

- Finite chunks

  - Discrete items pass through pipeline

  - Example: sequence of Ethernet packets

- Continuous bit stream

  - Stream of bits flows through pipeline

  - Example: video feed

Data Feed Mechanism

- Automatic

  – Built into pipeline itself

  – Integrates hardware for computation and data movement

- Manual

  – Separate hardware to move items between stages

  – Separate computation from data movement

Width Of Data Path

- Serial

  - One bit at a time

- Parallel

  - N bits at a time

Homogeneity Of Stages

- Homogeneous

  – All stages use the same hardware

  – Example: five identical processors

- Heterogeneous

  – Can have unique hardware at each stage

  – Example: each stage optimized for one function

Software Pipelining

- Popularized by Unix command interpreter (shell)

- User can specify pipeline as a command

- Example:

  cat x | sed 's/friend/partner/g' | sed '/W/d' | more

  - Substitutes "partner" for "friend"

  - Deletes lines that contain "W"

  - Passes result to *more* for display

- Note: example can be optimized by swapping the order of the two *sed* commands

Implementation Of Software Pipeline

- Uniprocessor

    – Each stage is a *process* or *task*

- Multiprocessor

    – Each stage executes on separate processor

    – Hardware assist can speed inter-stage data transfer

Hardware Pipelining

- Two broad categories

    - Instruction pipeline

    - Data pipeline

Instruction Pipeline

- Recall that instruction pipelining was covered in Chapter 5

- General idea

  - Optimizes performance

  - Heavily used with RISC architecture

  - Each instruction processed in stages

  - Exact details and number of stages depend on instruction set and operand types

Data Pipeline

- Data passes through pipeline

- Each stage handles data item and passes item to next stage

- Requires designer to divide work into stages

- Among the most interesting uses of pipelining

Hardware Pipelining And Performance

- A data pipeline can dramatically increase performance (throughput)

- To see why, consider an example

    – Internet router handles packets

    – Assume that a router

    * Processes one packet at a time

    * Performs six functions on each packet

Example Of Internet Router Processing

1. Receive a packet (i.e., transfer the packet into memory).

2. Verify packet integrity (i.e., verify that no changes occurred between transmission and reception).

3. Check for routing loops (i.e., decrement a value in the header, and reform the header with the new value).

4. Route the packet (i.e., use the destination address field to select one of the possible output networks and a destination on that network).

5. Prepare for transmission (i.e. compute information that will be used to verify packet integrity).

6. Transmit the packet (i.e., transfer the packet to the output device).

Illustration Of A Processor In A Router And The Algorithm Used

processor

input from one network
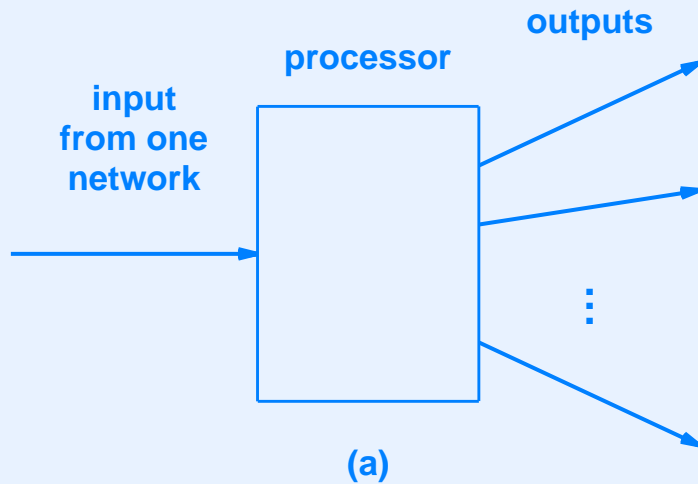
outputs

```
do forever  {
        Wait to receive packet
        Verify integrity
        Check for loops
        Route packet
        Prepare for transmission
        Enqueue packet for output
}
```

(a)

(b)

- (a) is an Internet router with multiple outgoing network connections

- (b) describes the computational steps the router must take for each packet

Example Pipeline Implementation

- Consider a router that uses a data pipeline

Example Pipeline Implementation

- Consider a router that uses a data pipeline
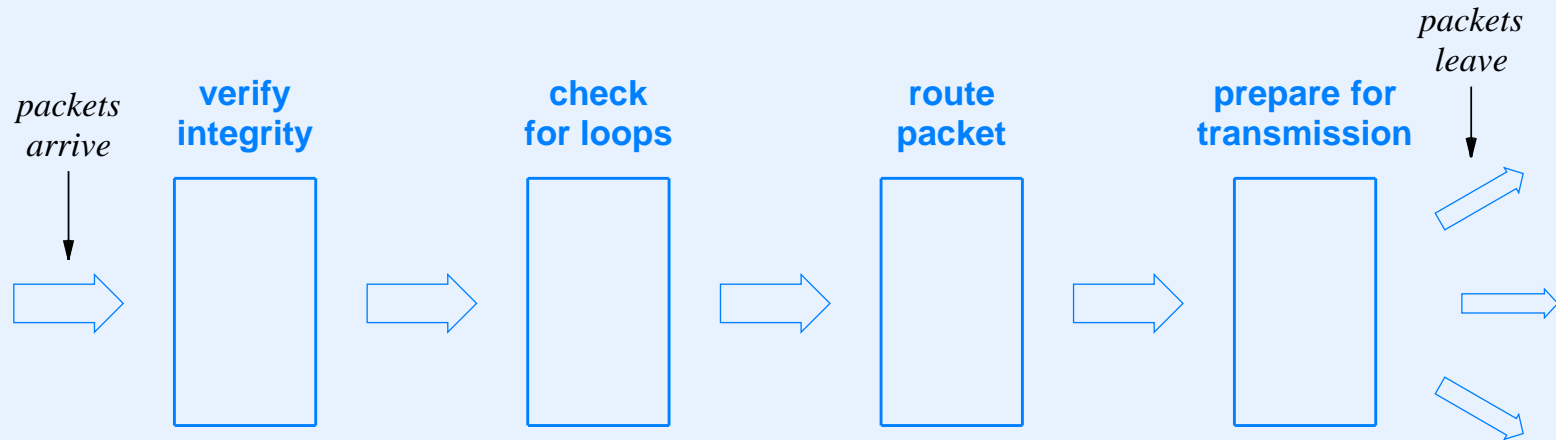
Example Pipeline Implementation

- Consider a router that uses a data pipeline
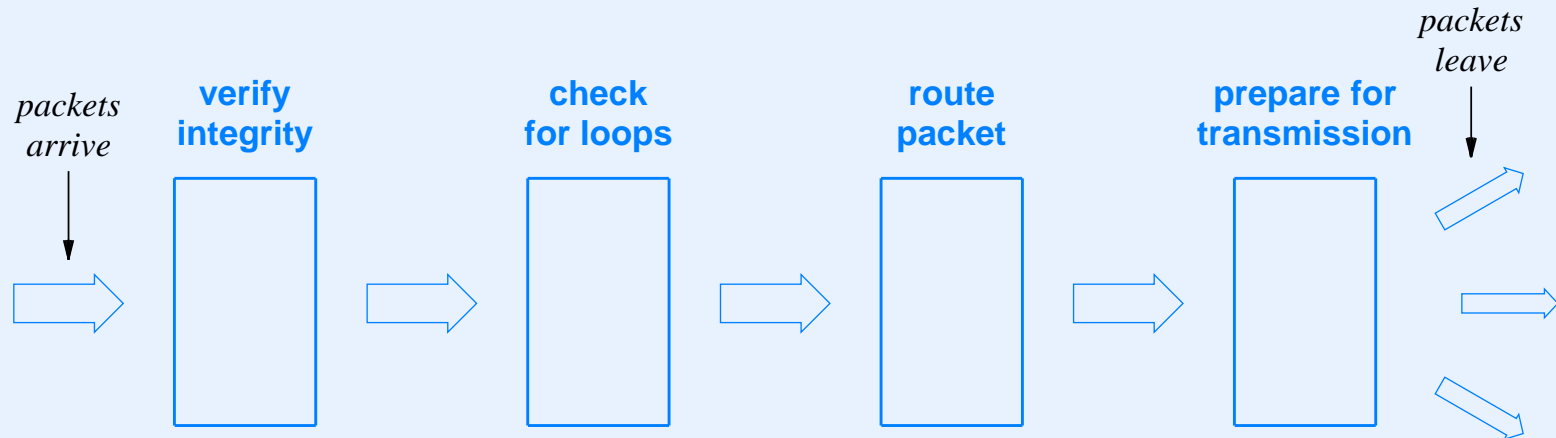
- Imagine a packet passing through the pipeline

Example Pipeline Implementation

- Consider a router that uses a data pipeline
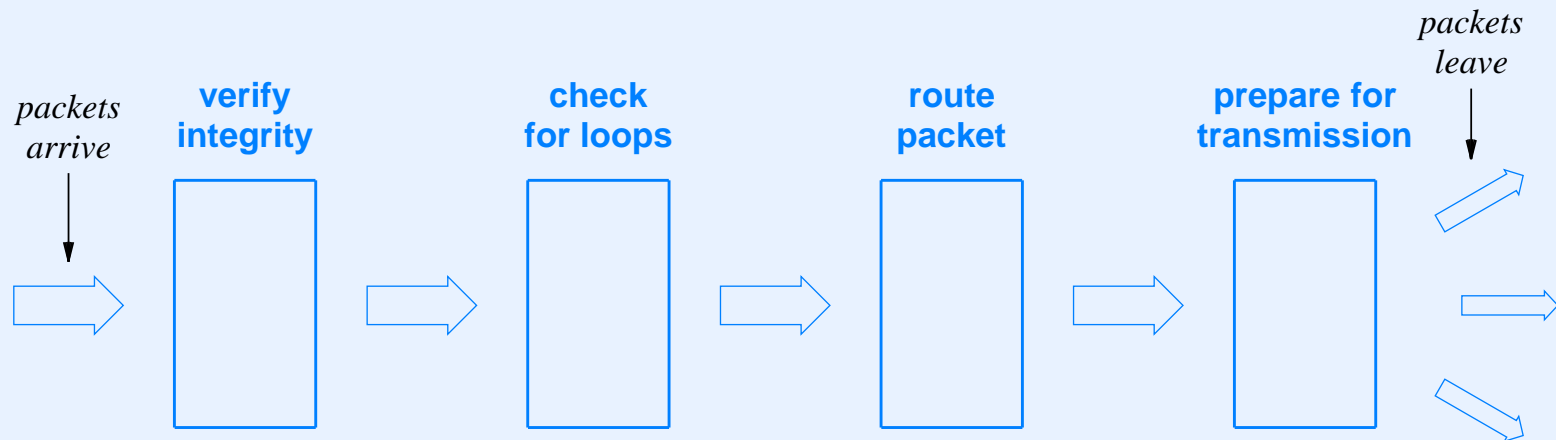
- Imagine a packet passing through the pipeline

- Assume zero delay between stages

Example Pipeline Implementation
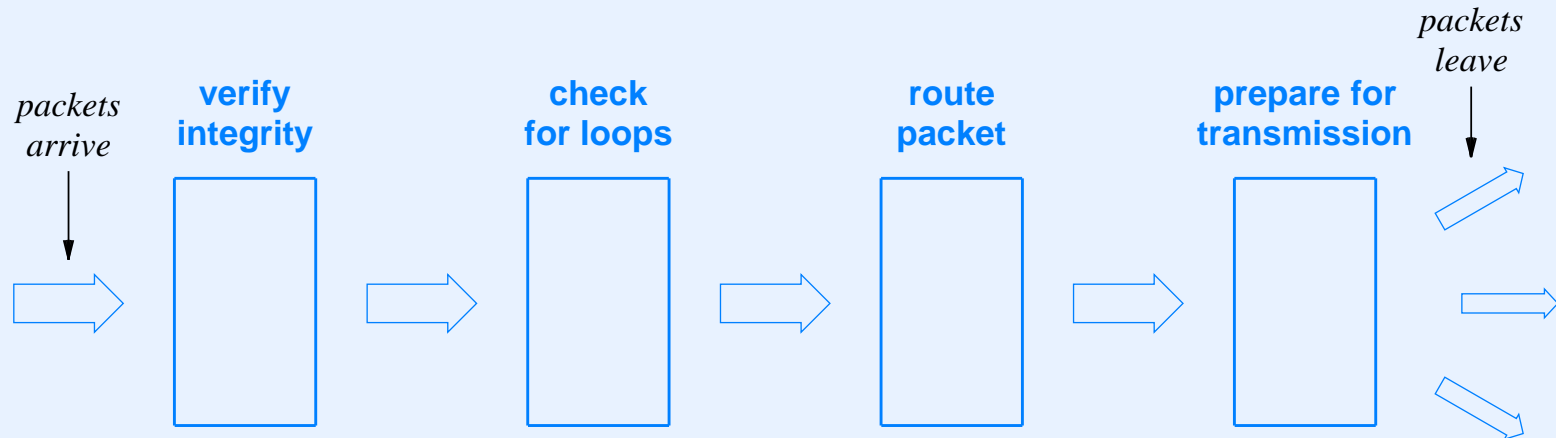
- Consider a router that uses a data pipeline

- Imagine a packet passing through the pipeline

- Assume zero delay between stages

- Question: how long will the pipeline take to process the packet?

Example Pipeline Implementation
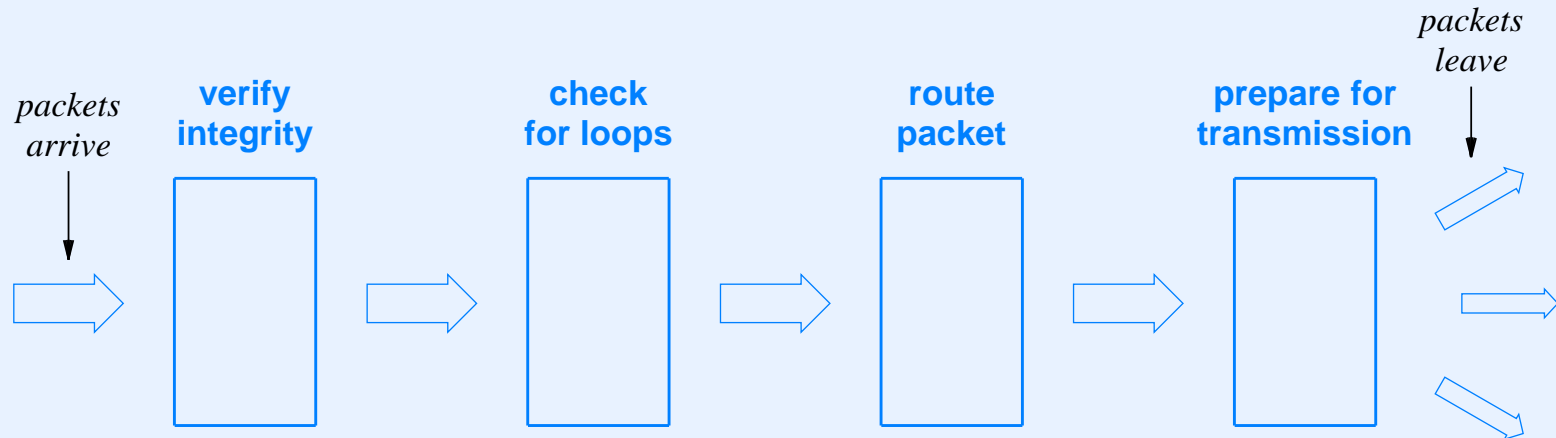
- Consider a router that uses a data pipeline

- Imagine a packet passing through the pipeline

- Assume zero delay between stages

- Question: how long will the pipeline take to process the packet?

- Answer: the same amount of time as a conventional router!

The Bad News

In a pipeline system, data passes through a series of stages that each examine or modify the data. If it uses the same speed processors as a nonpipeline architecture, a data pipeline will not improve the overall time needed to process a given data item.

The Good News

- If a pipeline takes the same total time to process a given item, how does it help?

The Good News

- If a pipeline takes the same total time to process a given item, how does it help?

- Surprise: by overlapping computation on multiple items, a pipeline increases throughput

- To summarize:

Even if a data pipeline uses the same speed processors as a nonpipeline architecture, a data pipeline has higher overall throughput (i.e., number of data items processed per second).

Pipelining Can Only Be Used If

- It is possible to partition processing into independent stages

- Overhead required to move data from one stage to another is insignificant

- The slowest stage of the pipeline is faster than a single processor

Understanding Pipeline Speed

- Assume

    - The task is packet processing

    - Processing a packet requires exactly 500 instructions

    - A processor executes 10 instructions per μ*sec*

- Total time required for one packet is:

$$time = \frac{500 \; instructions}{10 \; instr. \; per \; \mu sec} = 50 \; \mu sec$$

- Throughput for a non-pipelined system is:

$$T_{np} = \frac{1 \; packet}{50 \; \mu sec} = \frac{1 \; packet \times 10^6}{50 \; sec} = 20,000 \; packets \; per \; second$$

Understanding Pipeline Speed
(continued)

- Suppose the problem can be divided into four stages and that the stages require:

    - 50 instructions

    - 100 instructions

    - 200 instructions

    - 150 instructions

- The slowest stage takes 200 instructions
- So, the time required for the slowest stage is:

$$total\ time = \frac{200\ inst}{10\ inst\ /\ \mu sec} = 20\ \mu sec$$

Understanding Pipeline Speed
(continued)

- Throughput of the pipeline is limited by the slowest stage

- Overall throughput can be calculated:

$$T_p = \frac{1\ packet}{20\ \mu sec} = \frac{1\ packet \times 10^6}{20\ sec} = 50,000\ packets\ per\ second$$

- Note: throughput of pipelined version is 150% greater than throughput of the non-pipelined version!

Pipeline Architectures

- Term refers to architectures that are primarily formed around data pipelining

- Most often used for special-purpose systems

- Pipeline usually organized around functions

- Less relevant to general-purpose computers

Organization Of A Data Pipeline

- Easiest to form one stage per function

- Illustration

(a)   (b)
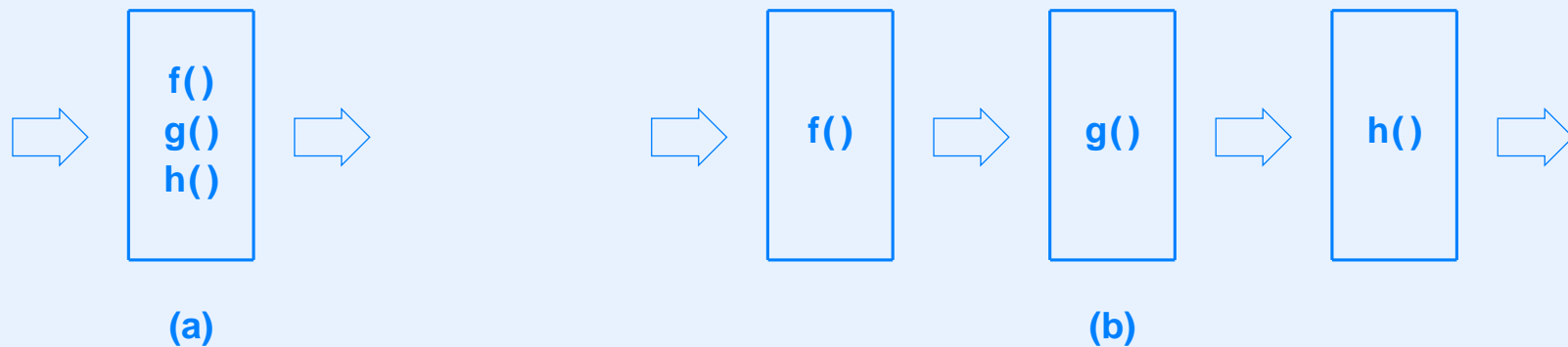
- (a) shows a single processor handling three functions

- (b) shows processing divided into a 3-stage pipeline with each stage handling one function

Pipeline Terminology

- *Setup time*

  – Refers to time required to start the pipeline initially

- *Stall time*

  – Refers to time required to restart the pipeline after a stage blocks to wait for a previous stage

- *Flush time*

  – Refers to time that elapses between the cessation of input and the final data item emerging from the pipeline (i.e., the time required to shut down the pipeline)

Superpipelining

- Most often used with instruction pipelining

- Subdivides a stage into smaller stages

- Example: subdivide operand processing into

  - Operand decode

  - Fetch immediate value or value from register

  - Fetch value from memory

  - Fetch indirect operand

- Technique: subdivide the slowest pipeline stage

Summary

- Pipelining

  - Broad, fundamental concept

  - Can be used with hardware or software

  - Applies to instructions or data

  - Can be synchronous or asynchronous

  - Can be buffered or unbuffered

Summary
(continued)

- Pipeline performance

  – Unless faster processors are used, data pipelining does not decrease the overall time required to process a single data item

  – Using a pipeline does increase the overall throughput (items processed per second)

  – The stage of a pipeline that requires the most time to process an item limits the throughput of the pipeline

Questions?

XIX

Assessing Performance

Measuring Computational Power

Measuring Computational Power

- Difficult to assess computer performance

Measuring Computational Power

- Difficult to assess computer performance

- Chief problems

Measuring Computational Power

- Difficult to assess computer performance

- Chief problems

    - Flexibility: computer can be used for wide variety of computational tasks

Measuring Computational Power

- Difficult to assess computer performance

- Chief problems

    – Flexibility: computer can be used for wide variety of computational tasks

    – Architecture that is optimal for some tasks is suboptimal for others

Measuring Computational Power

- Difficult to assess computer performance

- Chief problems

    - Flexibility: computer can be used for wide variety of computational tasks

    - Architecture that is optimal for some tasks is suboptimal for others

    - Memory and I/O costs can dominate processing

The Point About Performance

Because a computer is designed to perform a wide variety of tasks and no architecture is optimal for all tasks, the performance of a system depends on the task being performed.

Consequences

- Many groups try to assess computer performance

- A variety of performance measures exist

- No single measure suffices for all situations

Measures Of Computational Power

- Two primary measures

- Integer computation speed

    - Pertinent to most applications

    - Example measure is millions of instructions per second (*MIPS*)

- Floating point computation speed

    - Used for scientific calculations

    - Typically involve matrices

    - Example measure is floating point operations per second (*FLOPS*)

Average Floating Point Performance

- Assume

  - Addition or subtraction takes Q nanoseconds

  - Multiplication or division takes $2Q$ nanoseconds

- Average cost of floating point operation is:

$$T_{avg} = \frac{Q + Q + 2Q + 2Q}{4} = 1.5\,Q \quad ns\ per\ instr.$$

- Notes

  - Addition or subtraction costs 33% less than average, and multiplication or division costs 33% more

  - A typical program may not have equal numbers of multiply and add operations

A Note About Average Execution Times

Because some instructions take substantially longer to execute than others, the average time required to execute an instruction only provides a crude approximation of performance. The actual time required depends on which instructions are executed.

Application Specific Instruction Counting

- More accurate assessment of performance for specific application

- Examine application to determine how many times each instruction occurs

- Example: multiplication of two $N \times N$ matrices

  - N^3 floating point multiplications

  - $N^3 - N^2$ floating point additions

  - Time required is:

$$T_{total} = 2 \times Q \times N^3 + Q \times (N^3 - N^2)$$

Weighted Average

- Alternative to precise count of operations

- Typically obtained by instrumentation

- Program run on many input data sets and each instruction counted

- Counts averaged over all runs

- Example

Instruction Type	Count	Percentage
Add	8513508	72
Subtract	1537162	13
Multiply	1064188	9
Divide	709458	6

Computation Of Weighted Average

- Uses instruction counts and cost of each instruction

- Example

$$T_{avg}' = .72 \times Q + .13 \times Q + .09 \times 2\,Q + .06 \times 2\,Q$$

- Or

$$T_{avg}' = 1.16\,Q \ \ ns \ per \ instruction$$

- Note: the weighted average given here is %23 less than uniform average obtained above

Instruction Mix

- Measure a large set of programs

- Obtain relative weights for each type of instruction

- Use relative weights to assess the performance of a given architecture on the example set

- Try to choose set of programs that represent a "typical" workload

Use Of Instruction Mix

An instruction mix consists of a set of instructions along with relative weights that have been obtained by counting instruction execution in example programs. An architect can use an instruction mix to assess how a proposed architecture will perform.

Standardized Benchmarks

- Provides workload used to measure computer performance

- Represent "typical" applications

- Independent corporation formed in 1980s to create benchmarks

  - Named *Standard Performance Evaluation Corporation* (*SPEC*)

  - Not-for-profit

  - Avoids having each vendor choose benchmark that is tailored to their architecture

Examples Of Benchmarks Developed By SPEC

- SPEC cint2000

    – Used to measure integer performance

- SPEC cfp2000

    – Used to measure floating point performance

- Result of measuring performance on a specific architecture is known as the computer's *SPECmark*

I/O And Memory Bottlenecks

- CPU performance is only one aspect of system performance

- Bottleneck can be

    – Memory

    – I/O

- Some benchmarks focus on memory operations or I/O performance rather than computational speed

Increasing Overall Performance

To optimize performance, move operations that account for the most CPU time from software into hardware.

Which Items Should Be Optimized?

- Of course, adding additional hardware increases cost

- An architect cannot use high-speed hardware for all operations

- Computer architect Gene Amdahl observed that it is a waste of resources to optimize functions that are seldom used

Amdahl's Law

The performance improvement that can be realized from faster hardware technology is limited to the fraction of time the faster technology can be used.

Amdahl's Law And Parallel Systems

- Amdahl's law

    - Applies directly to parallel systems

    - Explains why adding processors does not always increase performance

Summary

- A variety of performance measures exist

- Simplistic measures include MIPS and FLOPS

- More sophisticated measures use a weighted average derived by counting the instructions in a program or set of programs

- Set of weights corresponds to an instruction mix

- Benchmark refers to a standardized program or set of programs used to measure performance

- Best-known benchmarks, known as SPECmarks, are produced by the SPEC Corporation

- Amdahl's Law helps architects select functions to be optimized (moved from software to hardware)

Questions?

XX

Architecture Examples
And
Hierarchy

General Idea

- Computer architecture can be presented at multiple levels of abstraction

- Known as *architectural hierarchy*

Architecture Range

- Macroscopic

  – Example: entire computer system

- Microscopic

  – Single integrated circuit

Architecture Terminology

Level	Description
System	A complete computer with processor(s), memory, and I/O devices. A typical system architecture describes the interconnection of components with buses.
Board	An individual circuit board that forms part of a computer system. A typical board architecture describes the interconnection of chips and the interface to a bus.
Chip	An individual integrated circuit that is used on a circuit board. A typical chip architecture describes the interconnection of functional units and gates.

Example System-Level Architecture
(A Personal Computer)

- Functional units

  - Processor

  - Memory

  - I/O interfaces

- Interconnections

  - High-speed buses for high-speed devices and functional units

  - Low-speed buses for lower-speed devices

Bus Interconnection And Bridging

- *Bridge* technology used to interconnect buses

- Allows

    – Multiple buses in a computer system

    – Processor only connects to one bus

- Bridge maps between bus address spaces

- Permits *backward compatibility* (e.g., old I/O device can connect to old bus and still be used with newer processor and newer bus)

Example Of Bridging

- Consider a PC

- Processor uses *Peripheral Component Interconnect* bus (*PCI*)

- I/O devices use older *Industry Standard Architecture* (*ISA*)

- Buses are incompatible (cannot be directly connected)

- Solution: have two buses connected by a bridge

Illustration Of PC Architecture Using A Bridge
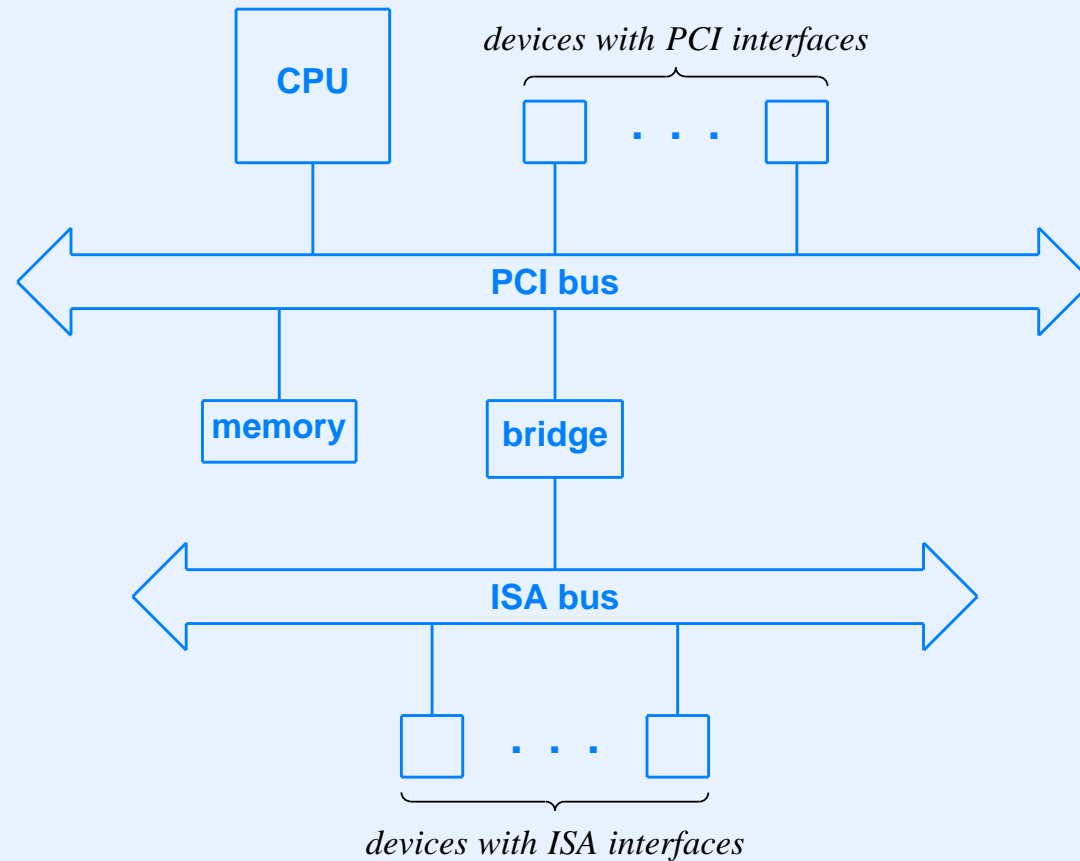
- Interconnection can be *transparent*

Physical Architecture

- Implementation of bridge is more complex than our conceptual diagram implies

- Uses special-purpose controller chips

- Separates high-speed and low-speed units onto separate chips

Controller Chips And Interconnections

Architects use a controller chip to provide interconnection among components in a computer because doing so is less expensive than equipping each unit with a set of interfaces or building a set of discrete bridges to interconnect buses.

A controller chip can provide the illusion of a bus over a direct connection; the wiring and sockets normally used to construct a bus are optional.

Typical PC Architecture

- Two controller chips used

- *Northbridge* chip connects higher-speed units

    – Processor

    – Memory

    – Advanced Graphics Port (AGP) interface

- *Southbridge* chip connects lower-speed units

    – Local Area Network (LAN) interface

    – PCI bus

    – Keyboard, mouse, or printer ports

Illustration Of Physical Interconnections
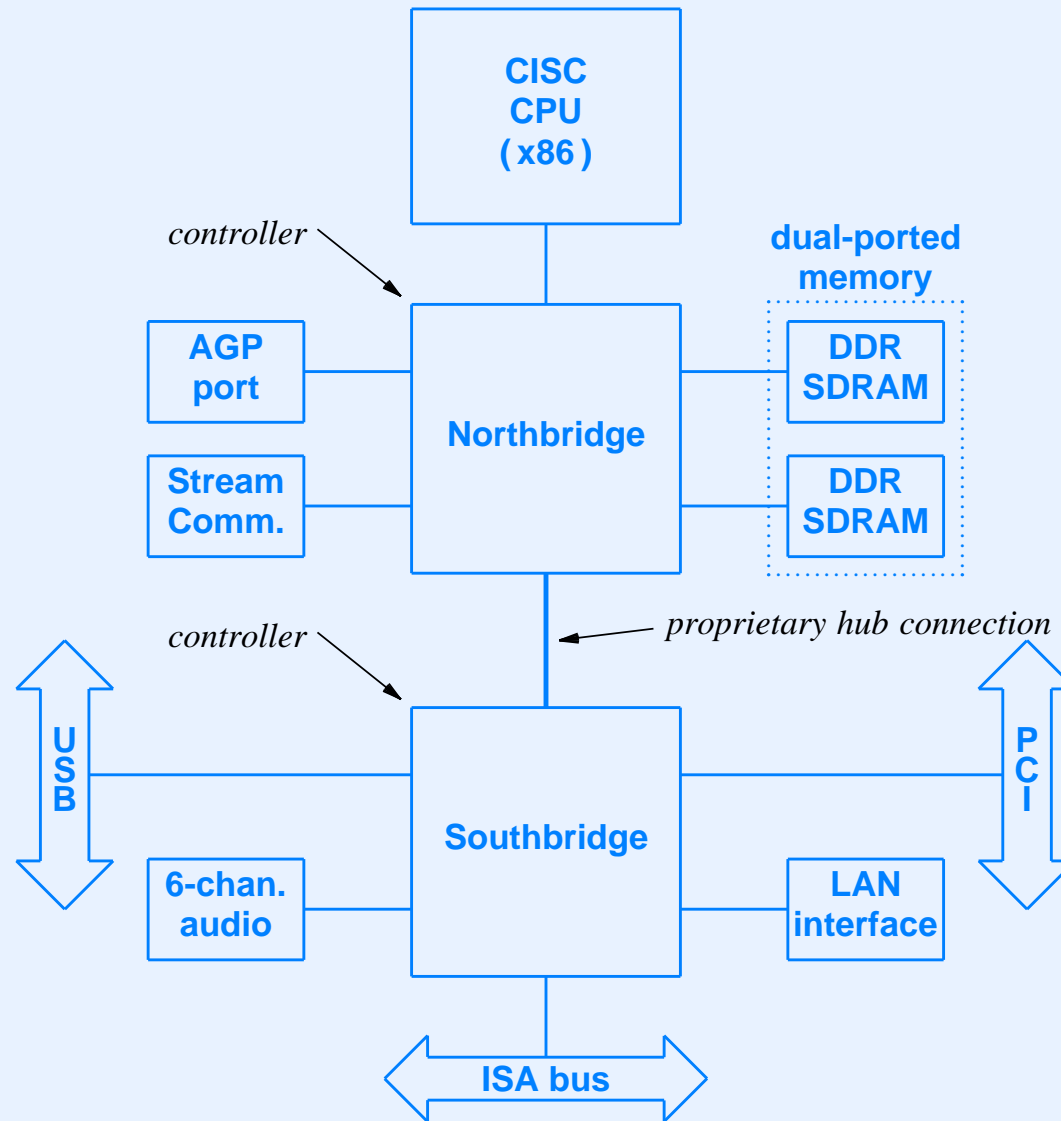
Example Products

- Northbridge: Intel 82865 PE

- Southbridge: Intel ICH5

Example Connection Speeds

- Rates increase over time

- Look at relative speeds, not absolute numbers in the following examples

Connection	Clock Rate	Width	Throughput
AGP	100-200 MHz	64-128 bits	2.0 GBps
Memory	200-800 MHz	64-128 bits	6.4 GBps
CPU	400-800 MHz	64-128 bits	3.2-6.4 GBps
Hub	100-200 MHz	64 bits	800 MBps
USB	33 MHz	32 bits	133 MBps
PCI	33 MHz	32 bits	133 MBps

Bridging Functionality And Virtual Buses

- Controller chips can virtualize hardware

- Example: controller presents the illusion of multiple buses to the processor

- One possible form: controller presents three virtual buses

  - Bus 1 contains the host and memory

  - Bus 2 contains a high-speed graphics device

  - Bus 3 corresponds to the external PCI slots for I/O devices

Example Board-Level Architecture

- LAN interface

  - Connects computer to Local Area Network

  - Transfers data between computer and network

  - Physically consists of separate circuit board

  - Usually contains a processor and buffer memory

Example Board-Level Architecture
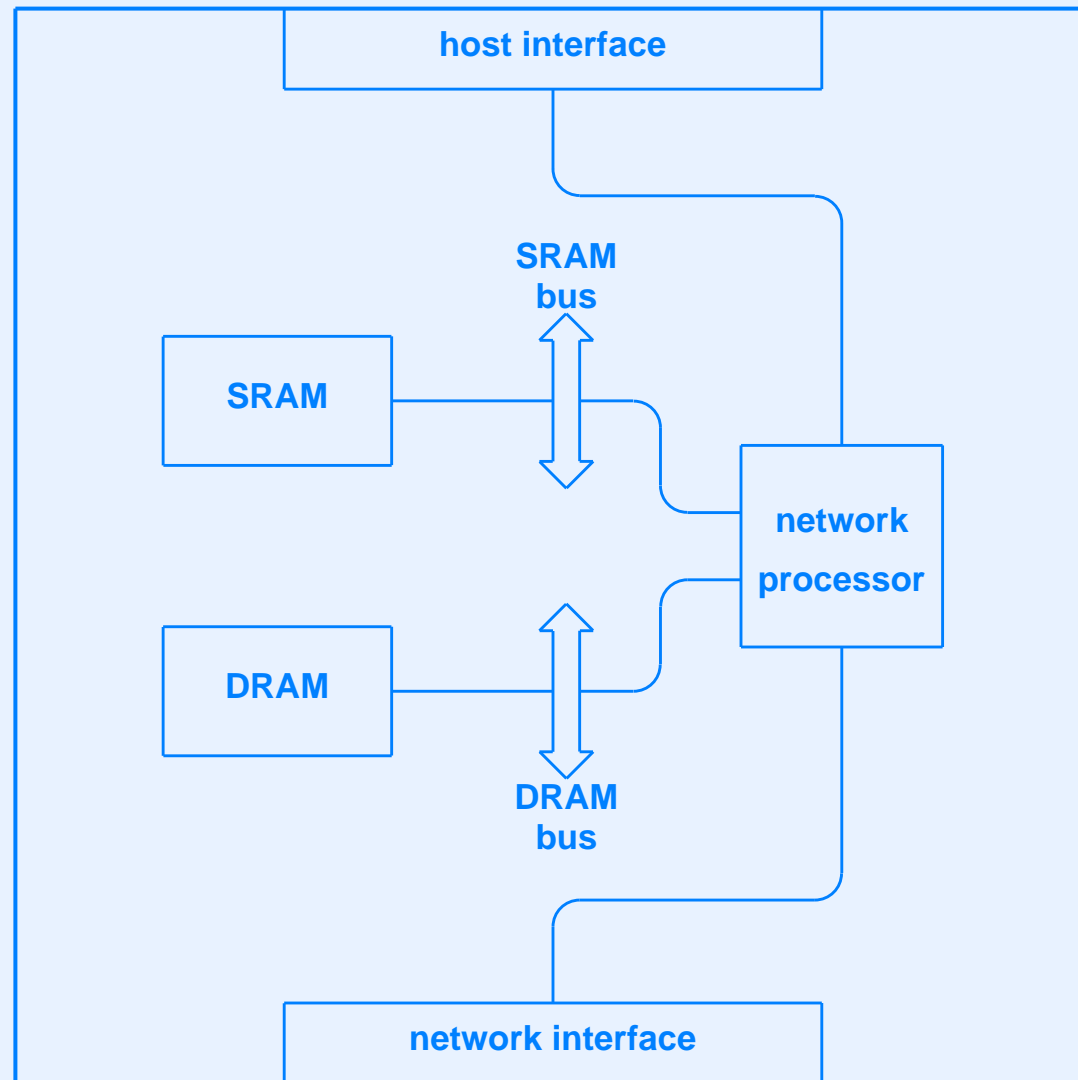
- LAN interface

Memory On A LAN Interface

- SRAM

    - Highest speed

    - Typically used for instructions

    - May be used to hold packet headers

- DRAM

    - Lower speed

    - Typically used to hold packet

- Designer decides which data items to place in each memory

Chip-Level Architecture

- Describes structure of single integrated circuit

- Components are functional units

- Can include on-board processors, memory, or buses

Example Chip-Level Architecture
(Intel Network Processor)
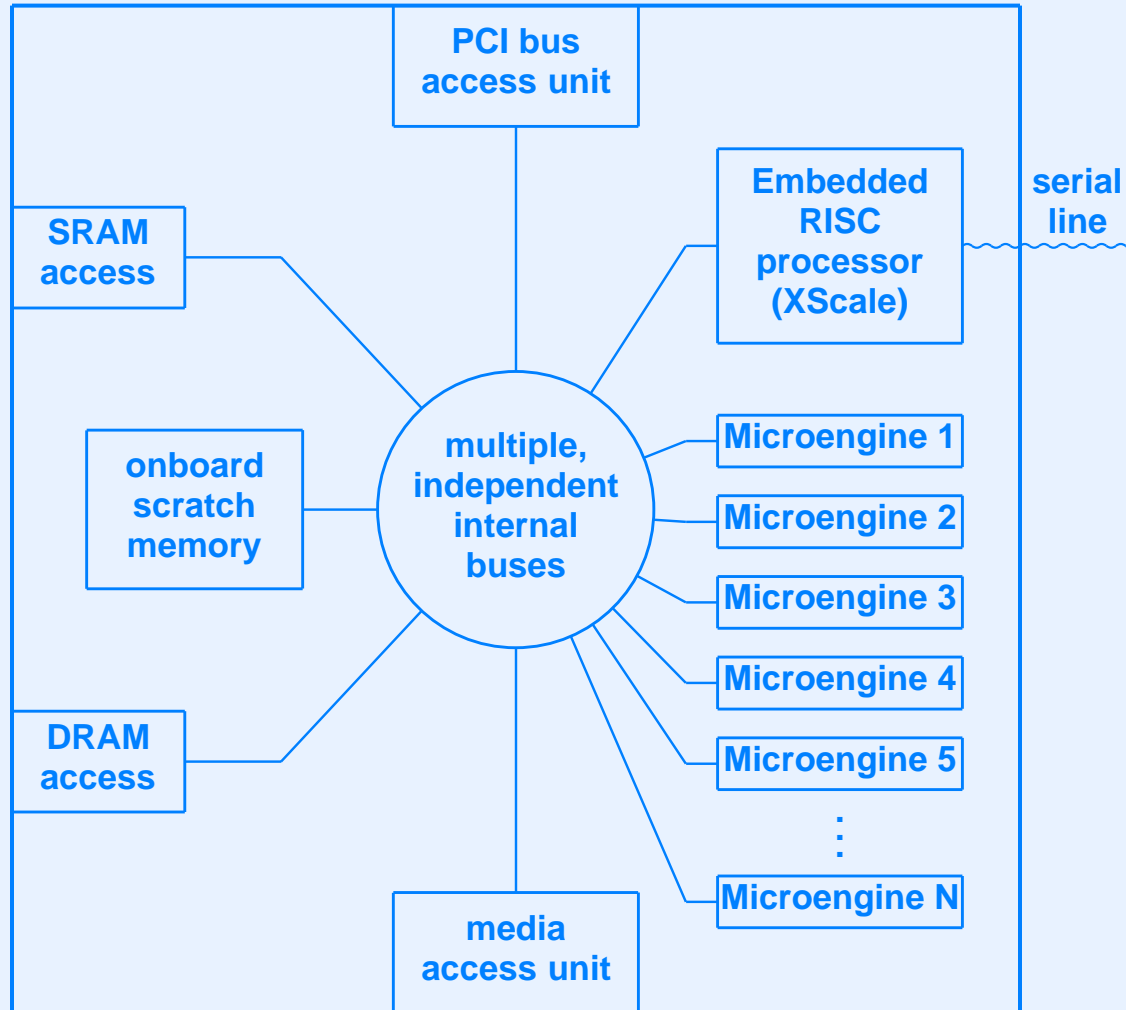
The Point Of Architectural Level

   A chip-level architecture reveals details about the internal structure of an integrated circuit that are hidden in a board-level architecture.

Structure Of Functional Units On A Chip
(Example Of Further Detail)
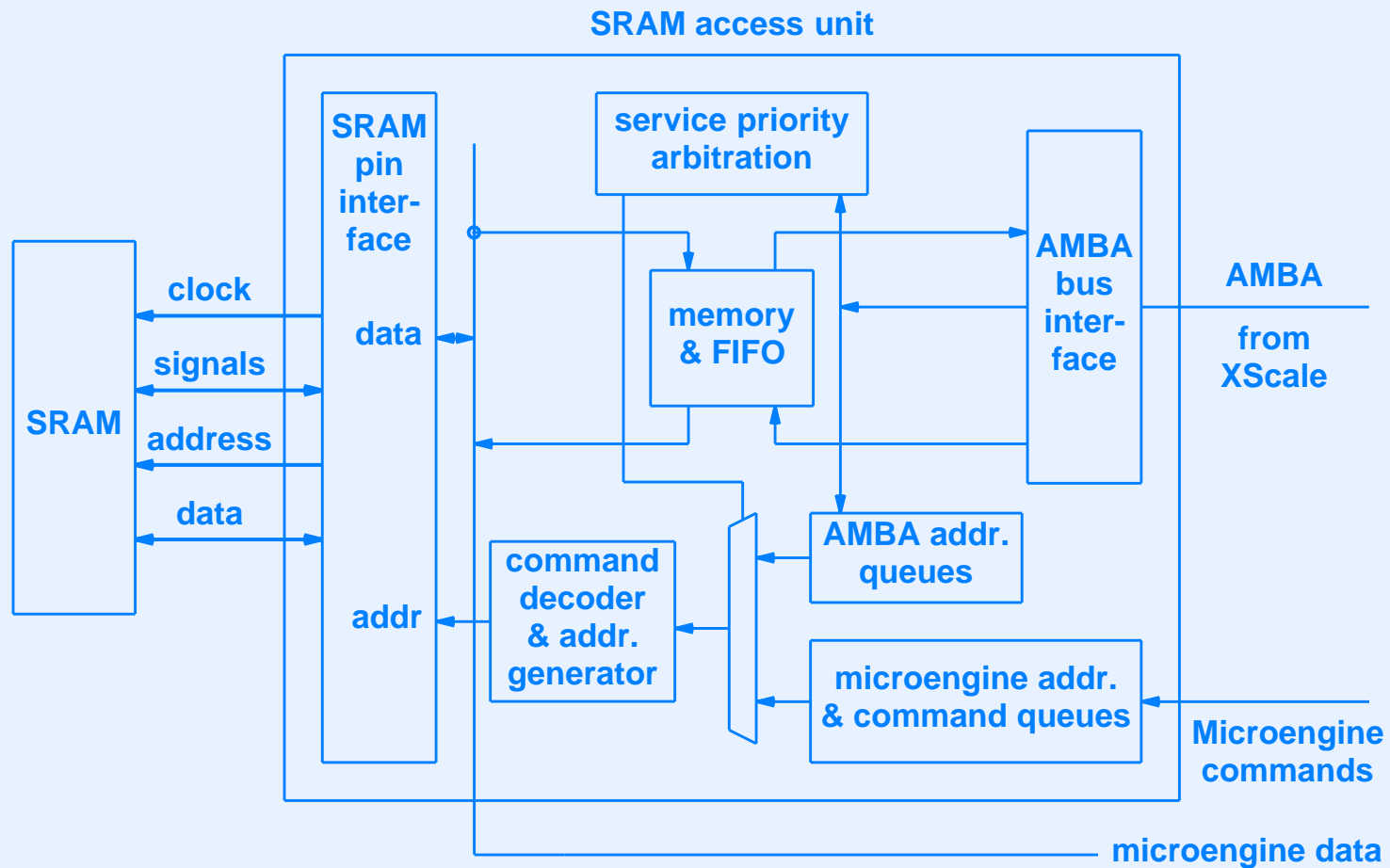
- Note: each item composed of logic gates

Summary

- Architecture of a digital system can be viewed at several levels of abstraction

- System architecture shows entire computer system

- Board architecture shows individual circuit board

- Chip architecture shows individual IC

- Functional unit architecture shows individual functional unit on an IC

Summary
(continued)

- We examined an example hierarchy

    - Entire PC

    - Physical interconnections of a PC

    - LAN interface in a PC

    - Network processor chip on a LAN interface

    - SRAM access unit on a network processor chip

Questions?

EXTRA
Examples Of Chip-Level Architecture
(Network Processors)

Definition Of A Network Processor

A network processor *is a special-purpose, programmable hardware device that combines the low cost and flexibility of a RISC processor with the speed and scalability of custom silicon (i.e., ASIC chips), and is designed to provide computational power for packet processing systems such as Internet routers.*

Network Processor Architectures

- What is known

  – Must partition packet processing into separate functions

  – To achieve highest speed, must handle each function with separate hardware

- Still being researched

  – Overall chip organization

  – Hardware building blocks

  – Interconnect of building blocks

  – Programming paradigm

Commercial Network Processors

Commercial Network Processors

- First emerged in late 1990s

Commercial Network Processors

- First emerged in late 1990s

- Used in products in 2000

Commercial Network Processors

- First emerged in late 1990s

- Used in products in 2000

- By 2003, more than thirty vendors existed

Commercial Network Processors

- First emerged in late 1990s

- Used in products in 2000

- By 2003, more than thirty vendors existed

- Currently, only a handful of vendors remain viable

Commercial Network Processors

- First emerged in late 1990s

- Used in products in 2000

- By 2003, more than thirty vendors existed

- Currently, only a handful of vendors remain viable

- Large variety of architectures

Commercial Network Processors

- First emerged in late 1990s

- Used in products in 2000

- By 2003, more than thirty vendors existed

- Currently, only a handful of vendors remain viable

- Large variety of architectures

- Optimizations: parallelism and pipelining

Augmented RISC (Alchemy)
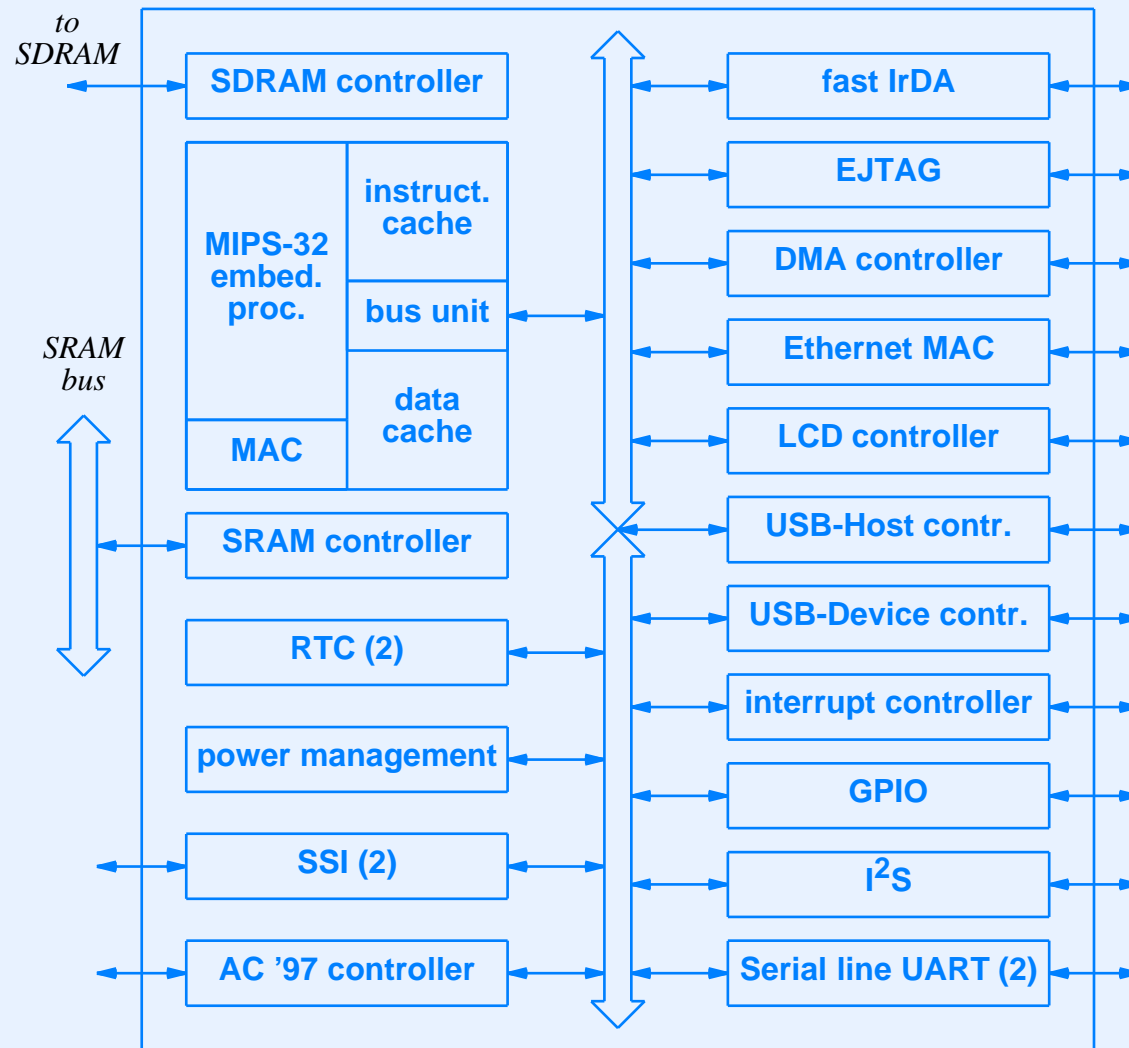
Parallel Processors Plus Coprocessors (AMCC)
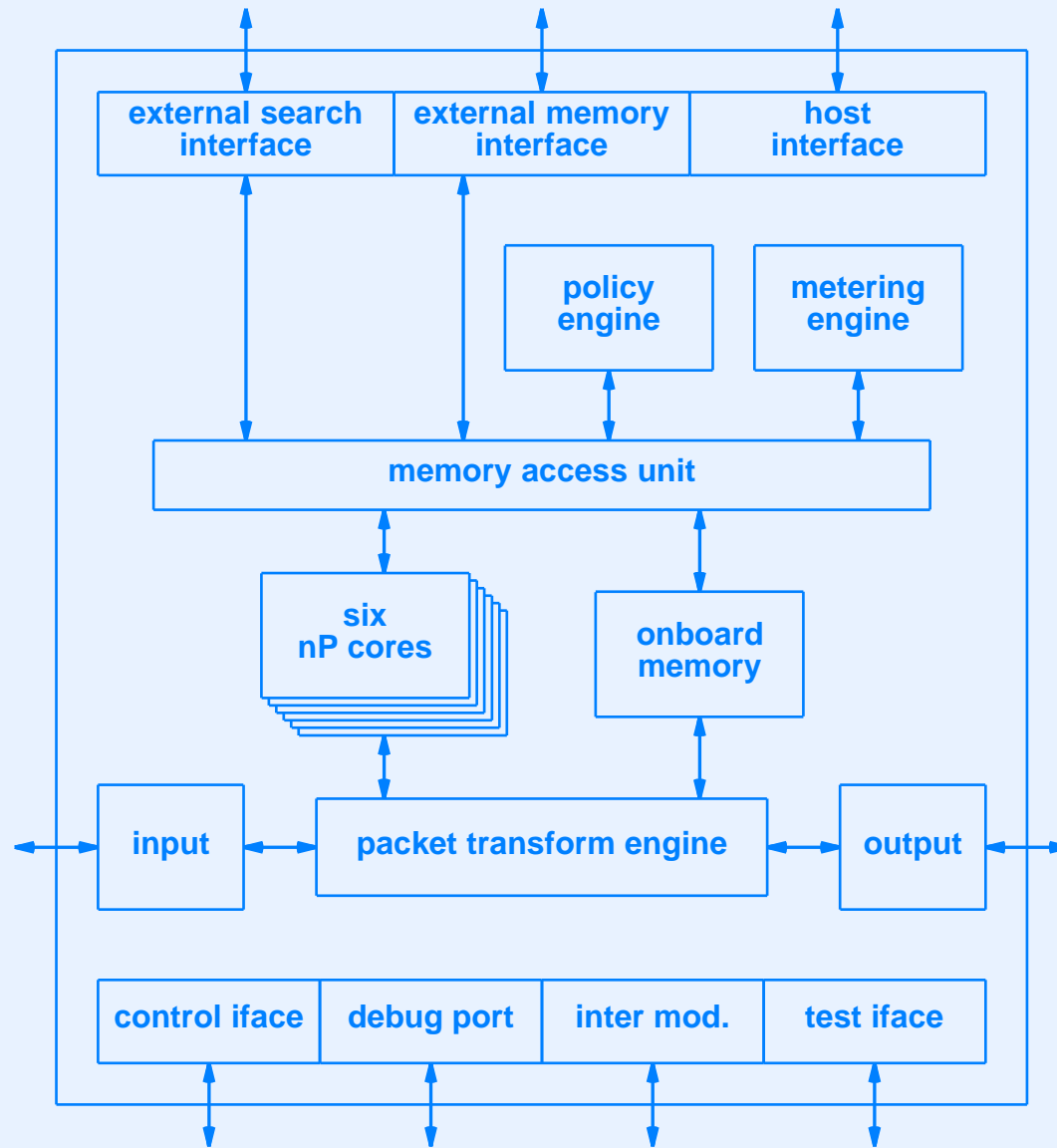
Pipeline Of Homogeneous Processors (Cisco)

- MAC classify
- Accounting & ICMP
- FIB & Netflow
- MPLS classify
- Access Control
- CAR
- MLPPP
- WRED
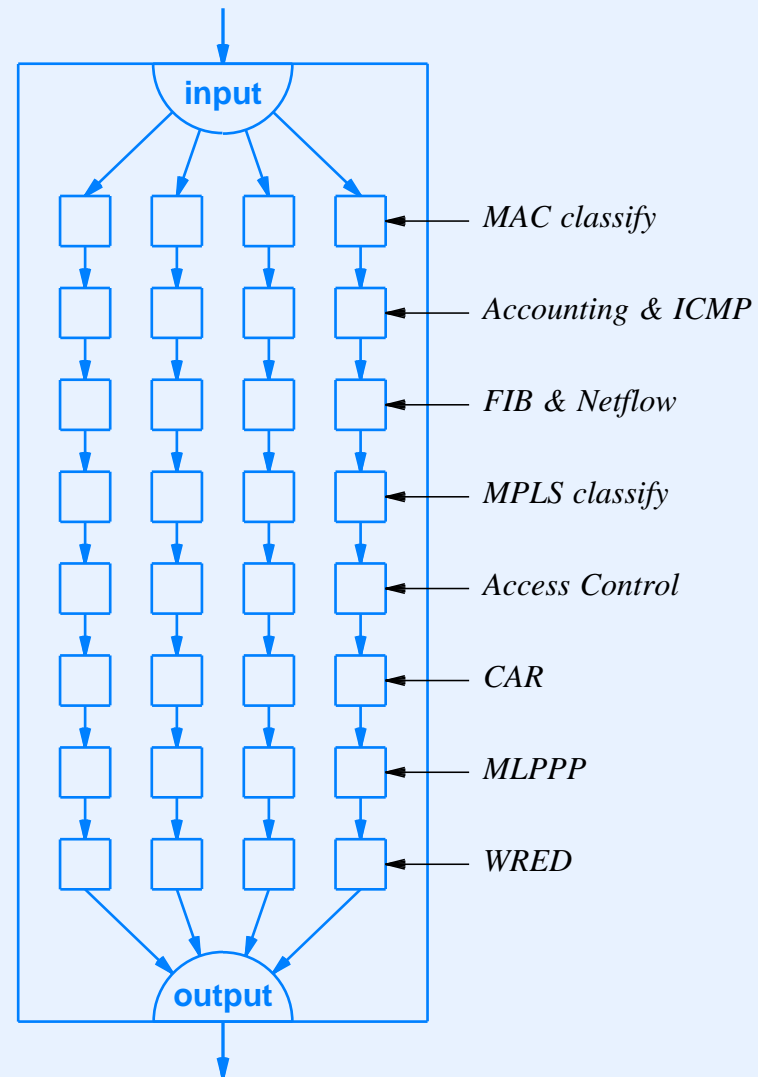
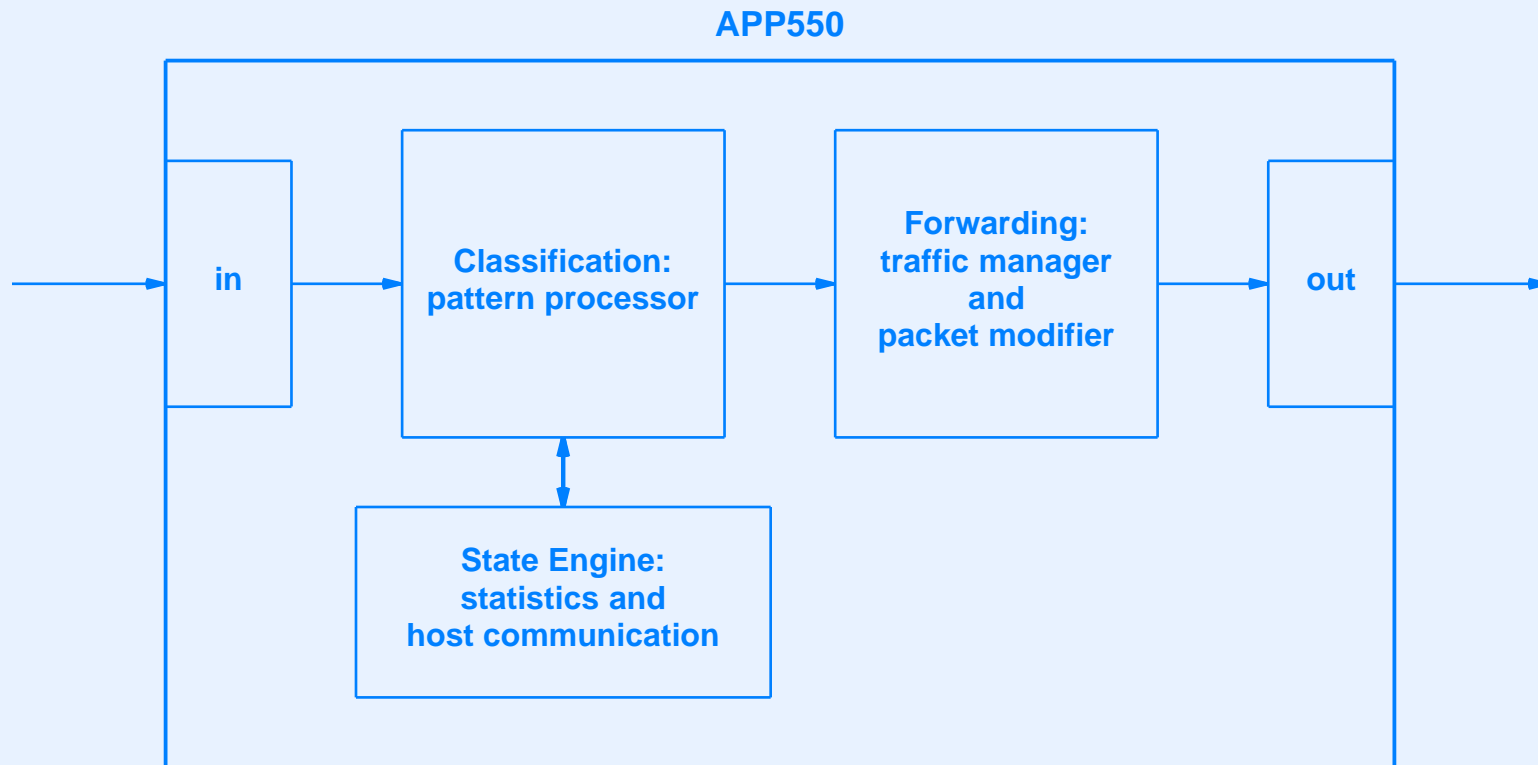# Pipeline Of Parallel Heterogeneous Processors (EZchip)
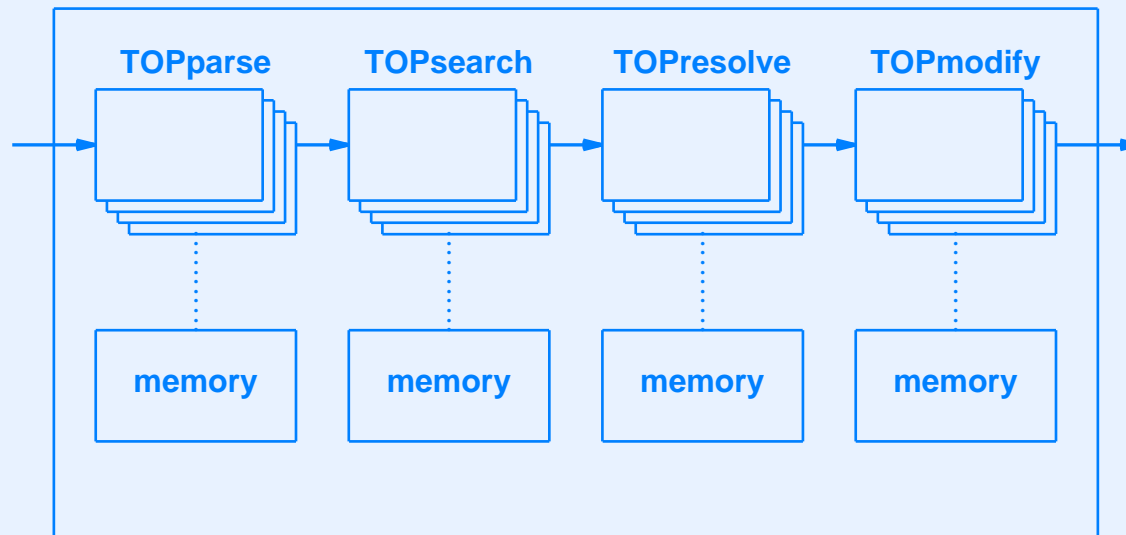
Extensive And Diverse Processors (Hifn)
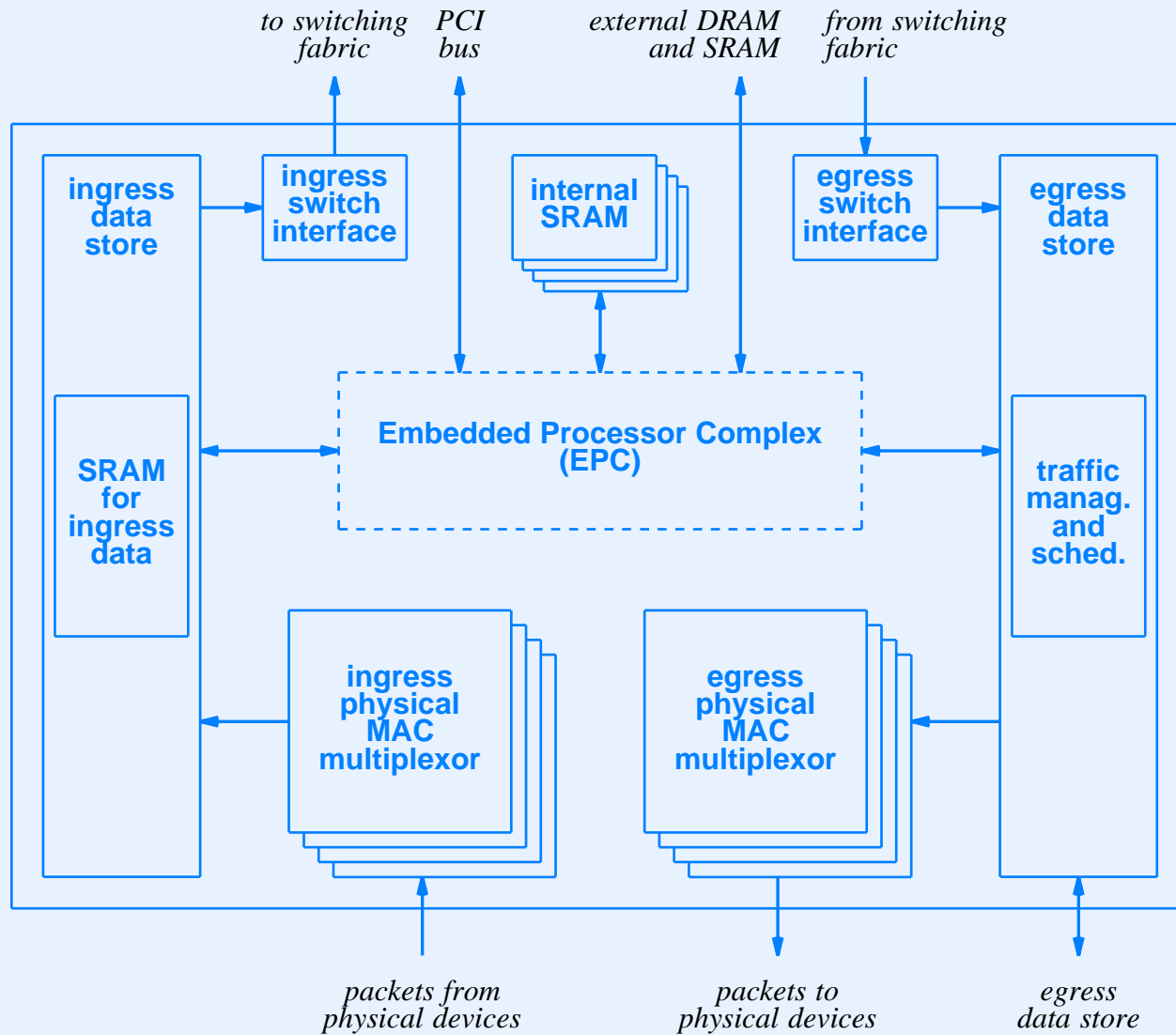
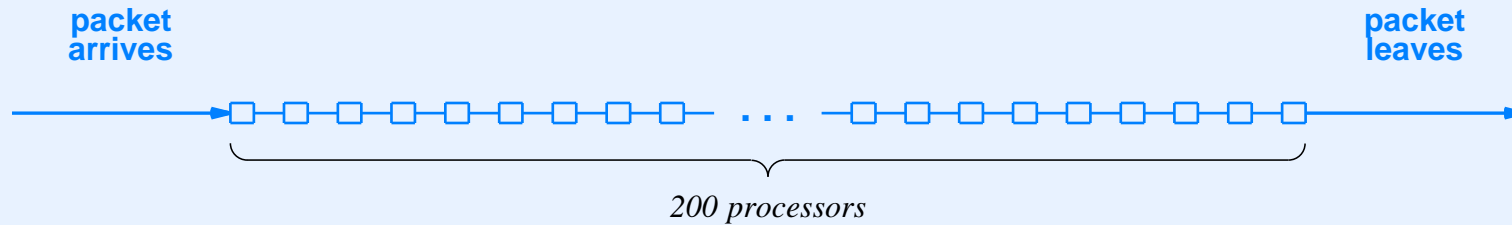# Hifn's Embedded Processor Complex
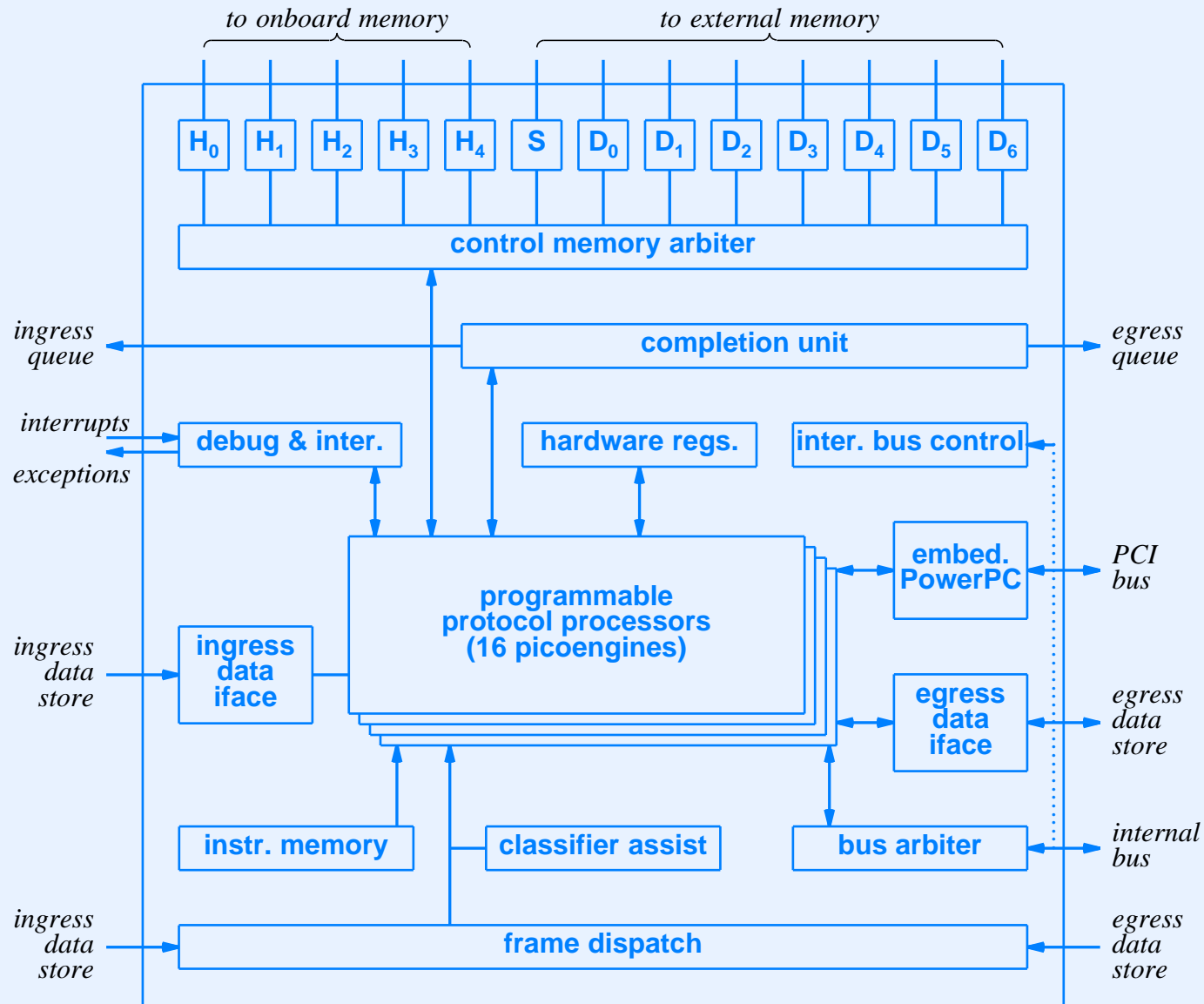
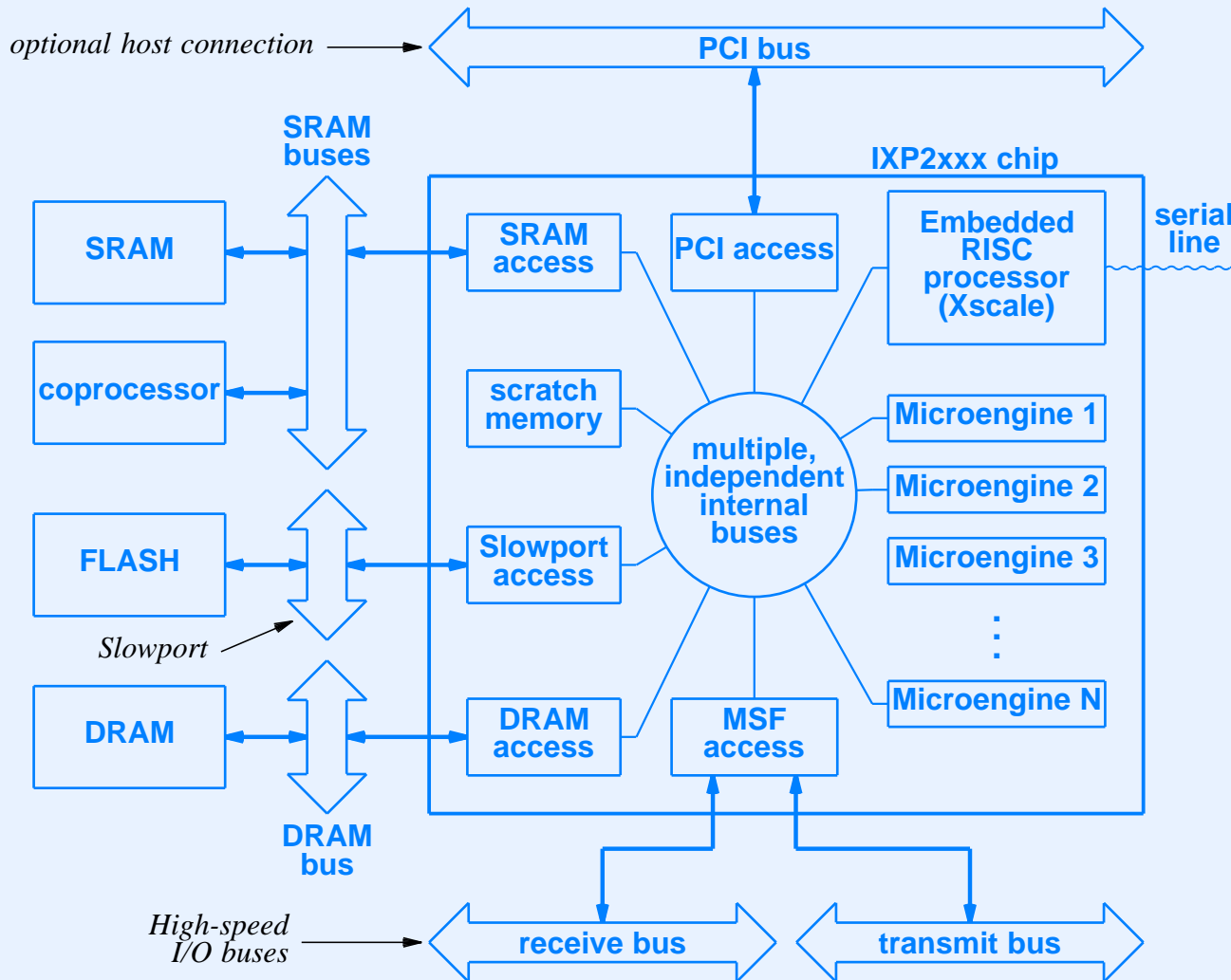# Short Pipeline Of Unconventional Processors (Agere)

- Classifier uses programmable pattern matching engine

- Traffic manager includes 256,000 queues
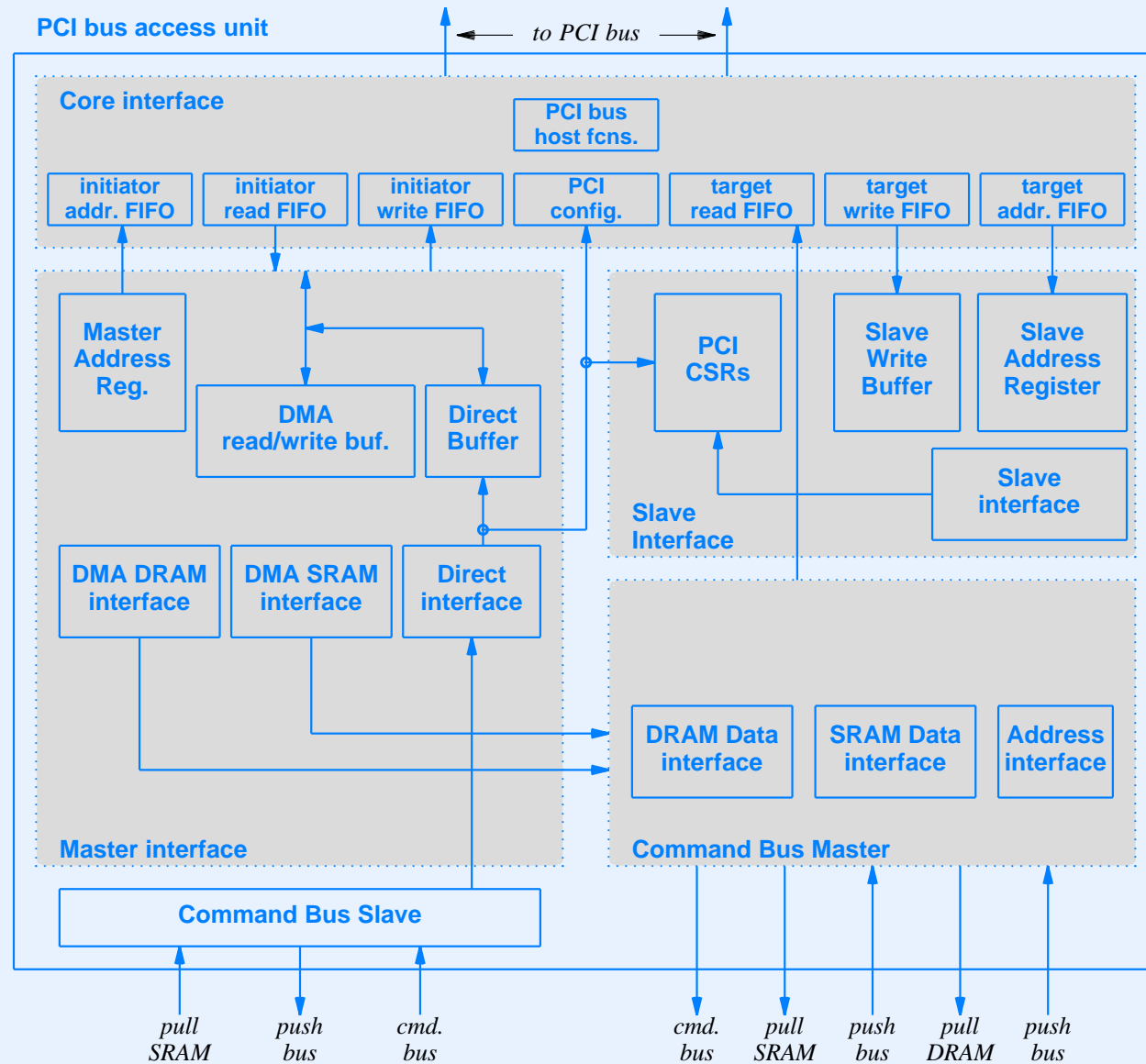
Extremely Long Pipeline (Xelerated)

- Each processor executes four instructions per packet

- External coprocessor calls used to pass state

Parallel Packet Processors (Intel)

Example Of Complexity (PCI Access Unit)

Programming Network Processors

- The bad news

  - Hardware determines programming paradigm

  - Low-level hardware often means low-level programming

- Most network processors are really microcontrollers

- Typically, programmer must

  - Control parallelism explicitly

  - Allocate data to registers

  - Position data items in specific memories

  - Be concerned with banks and interleaving for memory and registers

Programming A Network Processor
(continued)

- Unexpected surprises

  – Assembly language usually required

  – Programmer must be concerned with hardware contexts

  – Memory access costs many cycles

  – Separate interface hardware for each external connection

  – Simulator/emulator usually not accurate nor complete

Programming A Network Processor
(continued)

- More unexpected surprises

    – Code is optimized by hand

    – Assembler cannot resolve all register conflicts

    – No easy way to find bottlenecks

Programming Network Processors
(The Good News)

- A few exceptions exist

- *Agere*

  – Focuses on pattern matching and packet classification

  – Uses high-level languages

  – Offers implicit parallelism

- *IP Fabrics*

  – Focuses on programming simplicity

  – Uses very high-level language

Questions?

CS250
SEMESTER WRAP-UP

What You Learned

- Overall organization of hardware

- How digital circuits work

- Basics of

  – Processors

  – Memory

  – I/O devices

- Fundamental ideas of

  – Parallelism

  – Pipelining

Reasons For Studying Architecture

Reasons For Studying Architecture

- Understand how a computer works

Reasons For Studying Architecture

- Understand how a computer works

- Know what's going on underneath a program

Reasons For Studying Architecture

- Understand how a computer works

- Know what's going on underneath a program

- Relate programming constructs to hardware

Reasons For Studying Architecture

- Understand how a computer works

- Know what's going on underneath a program

- Relate programming constructs to hardware

- Appreciate reasons for hardware quirks

Reasons For Studying Architecture

- Understand how a computer works

- Know what's going on underneath a program

- Relate programming constructs to hardware

- Appreciate reasons for hardware quirks

- Write optimized code

Reasons For Studying Architecture

- Understand how a computer works

- Know what's going on underneath a program

- Relate programming constructs to hardware

- Appreciate reasons for hardware quirks

- Write optimized code

- Be able to tell when a compiler contains an error

Reasons For Studying Architecture

- Understand how a computer works

- Know what's going on underneath a program

- Relate programming constructs to hardware

- Appreciate reasons for hardware quirks

- Write optimized code

- Be able to tell when a compiler contains an error

- Prepare for later courses

Reasons For Studying Architecture

- Understand how a computer works

- Know what's going on underneath a program

- Relate programming constructs to hardware

- Appreciate reasons for hardware quirks

- Write optimized code

- Be able to tell when a compiler contains an error

- Prepare for later courses

The Key Idea You Have Mastered

- Computing involves multiple levels of abstraction

    - High-level language

    - Assembly language

    - Instruction set

    - Program in memory

    - Individual bits and bytes

    - Logic gates

- Note: to debug or optimize at one level, need to know the next lower level

- Example: to optimize a C program, use the *asm( )* function

What Comes Next

What Comes Next

- Compilers: you will learn how to parse a high-level language and translate into assembly language

What Comes Next

- Compilers: you will learn how to parse a high-level language and translate into assembly language

- Operating systems: you will learn how an operating system uses the hardware to provide concurrent execution

What Comes Next

- Compilers: you will learn how to parse a high-level language and translate into assembly language

- Operating systems: you will learn how an operating system uses the hardware to provide concurrent execution

- Computer networks: you will learn about computer communication and see how the Internet works

What You Should Do

What You Should Do

- Think about your career, not just a degree

What You Should Do

- Think about your career, not just a degree

- Don't assume courses are enough

What You Should Do

- Think about your career, not just a degree

- Don't assume courses are enough

- Practice on your own: program, program, program

What You Should Do

- Think about your career, not just a degree

- Don't assume courses are enough

- Practice on your own: program, program, program

- Get experience configuring hardware and software

What You Should Do

- Think about your career, not just a degree

- Don't assume courses are enough

- Practice on your own: program, program, program

- Get experience configuring hardware and software

    - Get an old, slow computer

What You Should Do

- Think about your career, not just a degree

- Don't assume courses are enough

- Practice on your own: program, program, program

- Get experience configuring hardware and software

  - Get an old, slow computer

  - Put Linux on it

What You Should Do

- Think about your career, not just a degree

- Don't assume courses are enough

- Practice on your own: program, program, program

- Get experience configuring hardware and software

  - Get an old, slow computer

  - Put Linux on it

  - Add a second Ethernet NIC

What You Should Do

- Think about your career, not just a degree

- Don't assume courses are enough

- Practice on your own: program, program, program

- Get experience configuring hardware and software

    – Get an old, slow computer

    – Put Linux on it

    – Add a second Ethernet NIC

    – Increase or decrease memory size

What You Should Do

- Think about your career, not just a degree

- Don't assume courses are enough

- Practice on your own: program, program, program

- Get experience configuring hardware and software

    – Get an old, slow computer

    – Put Linux on it

    – Add a second Ethernet NIC

    – Increase or decrease memory size

    – Program: see what you can make it do

Questions?

STOP