

Today's Topics

- Viewing the computer system as a hierarchy of levels
- How operations at one level are translated into a lower level
- RISC vs CISC computer architectures
- RISC design principles
- Overview of the textbook's three sample computer architectures
- Organization of a computer system
- How programs are executed
- Units of measurement
- Instruction- and Processor-level parallelism
- Memory organization
- Memory addresses
- Memory words
- Byte ordering

Exercise 1 - What is a Computer?

1.1.2



A - Something that can calculate?



B - Something that responds to user input?



C - Something that follows instructions?



D - Something that stores data?

A Computer System Is:

Of the choices given, the best one is:



Something that follows instructions

But a better definition would be the one given by Wikipedia:

A machine for manipulating data according to a list of instructions.

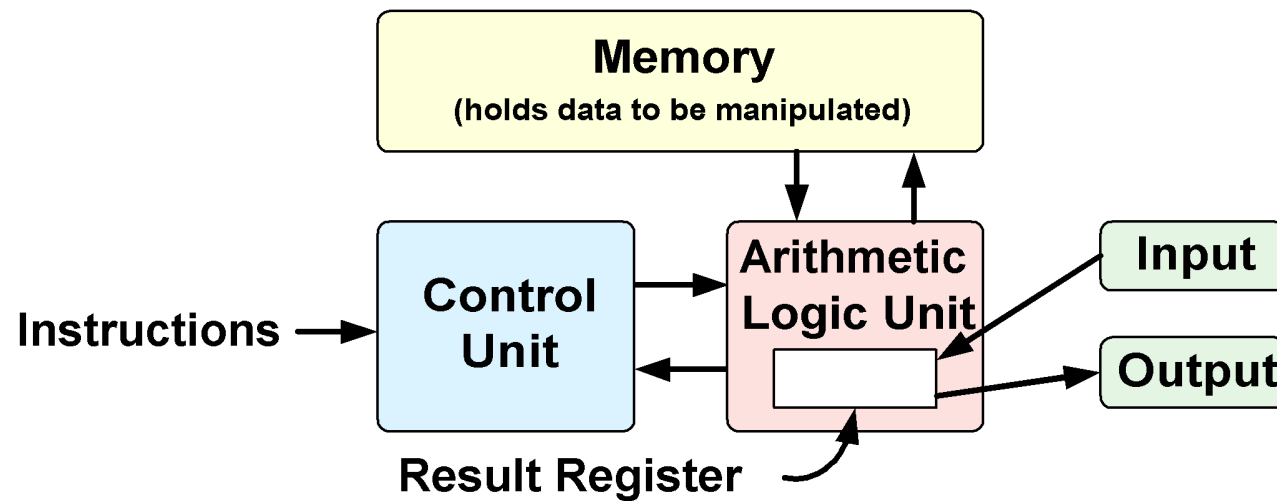
Computer System Capabilities

A “machine for manipulating data according to a list of instructions” must have:

- The ability to store data (memory)
- The ability to manipulate data (an “Arithmetic Logic Unit”)
- A means to understand and execute instructions (a “Control Unit”)

And, although it’s not strictly required by this definition, a computer is not particularly useful unless it can also:

- Accept data from an input device
- Deliver results to an output device



What are “Instructions”?

In a computer system, instructions are signals that direct the computer to manipulate data. Examples include instructions which can:

- Accept data from an **input** device
- Perform an **arithmetic operation** on data
- **Compare** data
- **Move data** from one storage location to another
- Deliver data to an **output** device

In the 1940's a mathematician named Alan Turing showed that a computer with the right kinds of instructions can perform any algorithm. Such a computer is said to be “**Turing Complete**”.

To be “Turing Complete”, a computer system must also understand and execute “goto” instructions that can choose a particular sequence of instructions to be executed. These allow the algorithm to include:

- **Loops** (sequences of instructions that are executed repeatedly)
- **Conditional execution** (instructions that may or may not be executed depending on the value of data)

A computer system that includes these capabilities can perform any algorithm on the data it contains.

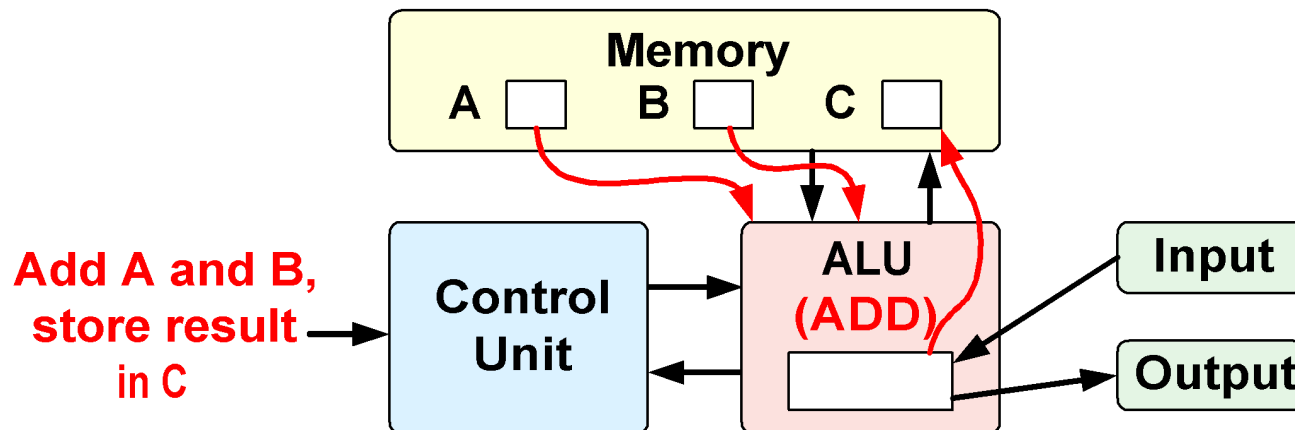
Executing an Instruction

The verb “execute” is used to describe the action that a computer takes to perform the work indicated by an instruction. For example, consider an instruction that directs the computer to:

Add data item A and data item B together, and store the result in data item C

To do this, the control unit would have to perform the following steps:

- Signal the memory to move data item A to the Arithmetic Logic Unit (ALU)
- Signal the memory to move data item B to the ALU
- Signal the ALU to add the two data items together
- Signal the memory to move the ALU result to data item C



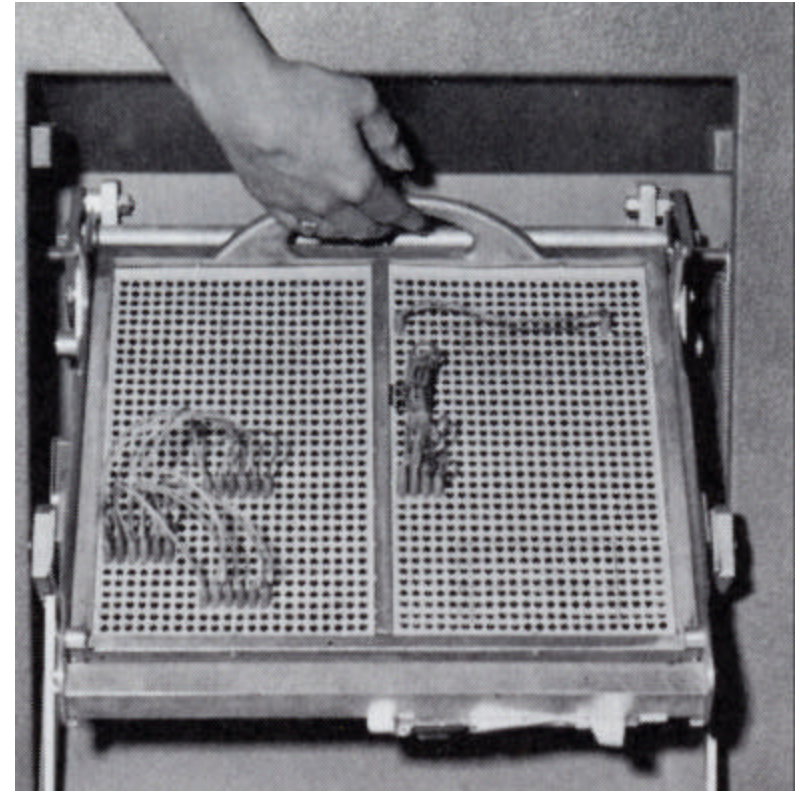
When the control unit takes these actions, we say that it's “executing” the instruction.

Where do the Instructions come from?

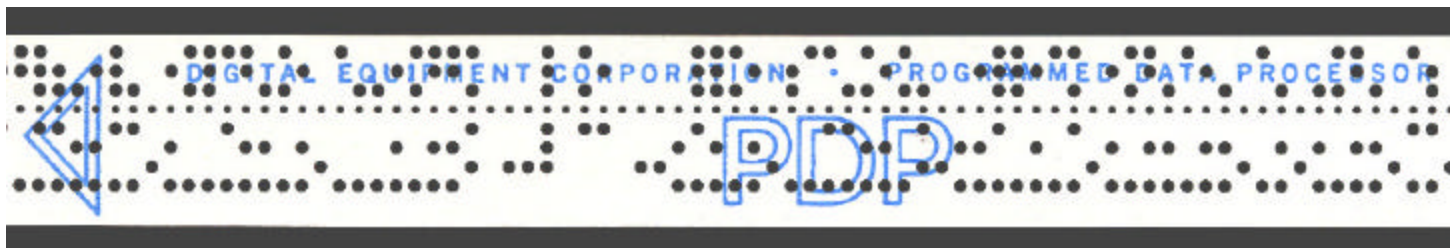
Sequences of instructions are prepared by people in the form of **computer programs**. A program is a complete set of instructions required to perform some specific task. Before the computer can execute a program, it must be encoded into a form that can be accepted and recognized by the control unit.

In early computers, the instructions were punched onto paper tape or wired into plugboards. These had to be physically inserted into the machine or fed into a reader in order to run the program.

These methods were difficult to work with and made it difficult or impossible to include instructions such as loops (although some enterprising programmers actually spliced the ends of a paper tape to itself to form physical loop!)



A Plugboard



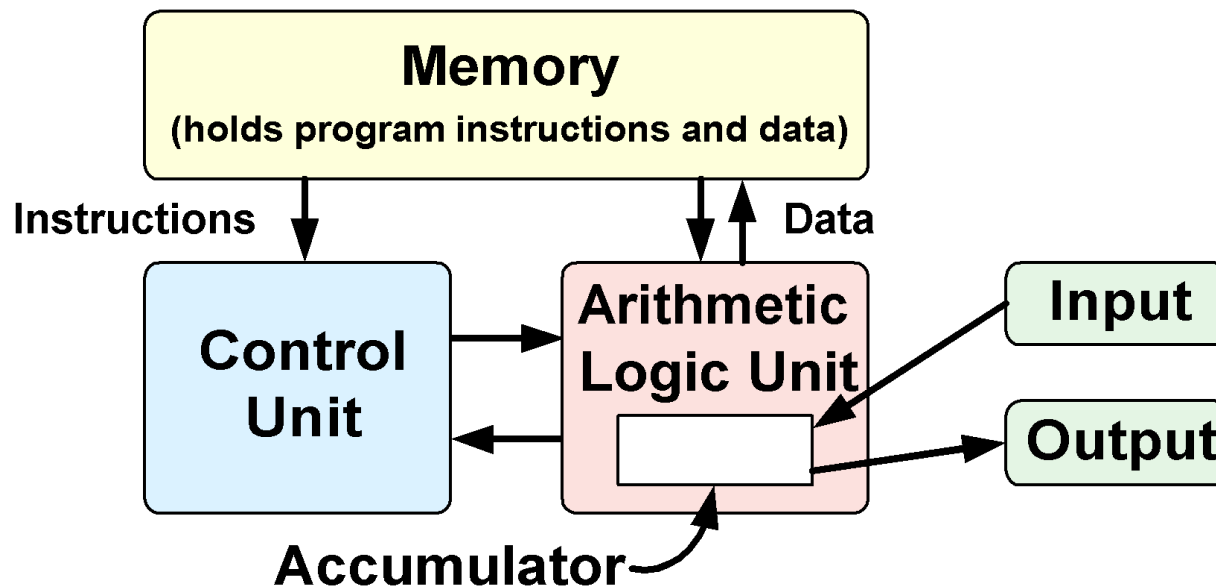
A Paper Tape

Von Neumann Architecture

1.2.2

In the mid 1940's John Von Neumann, another key figure in the early development of computers, came up with the idea of storing the instructions in the same memory that was used to store data.

This freed the computer from the constraints of sequentially reading instructions from a linear medium such as paper tape. The program could still be encoded onto paper tape, but it would now be transferred into memory for execution. This allowed the computer to execute instructions more quickly, and since the entire program was always accessible in memory, it meant that the program could include loops.

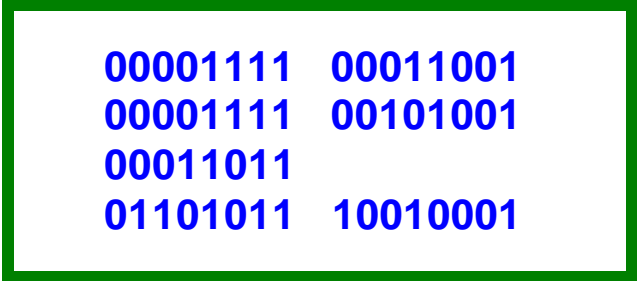


This key concept revolutionized the design of computers and since then virtually all digital computers have used this design, which is now known as the [Von Neumann Architecture](#).

The Instruction Set Architecture

Instructions must be given to a computer in a format that it can understand. In most modern digital computers, they're delivered as binary signals (signals that can be TRUE or FALSE, 1 or 0). Each individual signal is known as a “bit” (Binary digIT).

A series of instructions might consist of several dozen bits that look something like this:



```
00001111 00011001
00001111 00101001
00011011
01101011 10010001
```

This could be interpreted by the computer as a sequence of instructions to (for example):

- Move data item A from memory to the ALU (“load A”)
- Move data item B from memory to the ALU (“load B”)
- Subtract
- Move the ALU result to data item C in memory (“store to C”)

Each type of computer system has it's own way of encoding instructions – instructions that are recognized by one system may not work on a different type of system. The instruction formats and types of data that a particular computer is able recognize and execute is known as it's Instruction Set Architecture, often abbreviated to just “ISA”.

Assembly Language

Programs in the form of binary numbers such as:

```
00001111 00011001
00001111 00101001
00011011
01101011 10010001
```

...are extremely difficult to write, understand, and debug. Rather than writing the 1 and 0 bits directly, programmers use a symbolic representation known as “[Assembly Language](#)”. The above program might look something like this when written symbolically:

```
LD      Gross_Pay
LD      Deductions
SUB
ST      NetPay
```

In this example, the instructions and data items have been given symbolic names such as “LD” (load) and “ST” (store) to make the program easier to understand. A programmer would create the instructions using this symbolic notation and use a special program called an Assembler to convert it to the binary form for execution by the computer.

In this example, note that the assembler would translate the “LD” statements (LOAD the ALU inputs) to a machine language code of “00001111”.

The Computer as a Multilevel Architecture

1.1.2

The organization of a computer system can be thought of as having multiple levels. A computer that's programmed using a symbolic Assembly language can be thought of as a system with these three levels:

Assembly Level

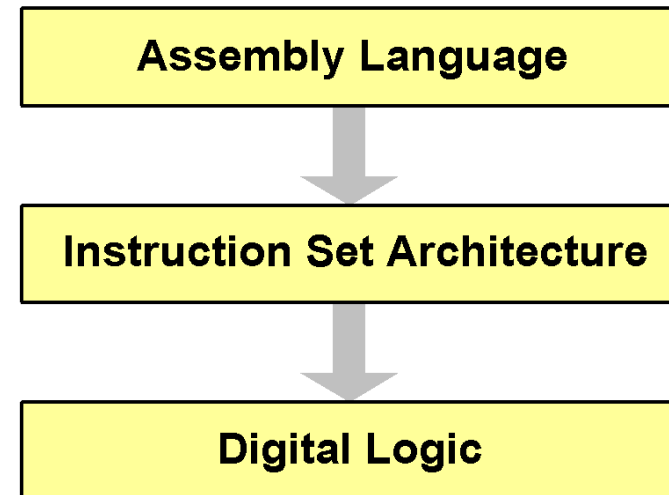
The symbolic language in which the computer is programmed.

Instruction Set Architecture Level

Binary codes created from the symbolic instructions which the computer hardware can understand and execute

Digital Logic

The circuits which can store data, manipulate it, and which recognize and execute the ISA instructions.



The earliest computer systems consisted of just these three levels and nothing more. This meant:

- Programs were difficult to write because they had to be written using very simple instructions that the hardware could understand.
- Programs had to be manually loaded into the computer, and only one program could be run at a time.

The Invention of the Operating System

1.1.3

The manual loading of programs and data into early computers was a waste of time and money. To reduce manual intervention, simple “supervisor” programs were developed to automate the process.

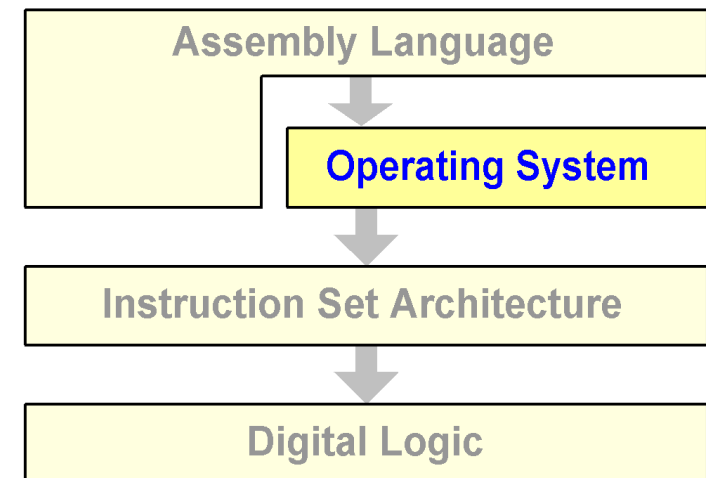
The supervisor program would be loaded when the computer was turned on and it would always reside in memory. It’s job was to read an “application” program into memory from a card reader or other input device, then allow it to execute it’s own instructions and read it’s own data. When the application program finished, it would branch back to the supervisor program which would then load then next program from the input device.

When more sophisticated I/O devices such as tape and disk drives became available, the supervisor evolved into an Operating System which managed storage and automatic loading of program and data files, and provided many additional capabilities to application programs.

The development of the operating system added a fourth level to the architecture of computer systems.



A Card Reader



Operating System Capabilities

1.1.3

Early Operating Systems merely provided the ability to run “batch” jobs, where the program and its data were read and processed automatically with no human intervention. Modern operating systems are far more sophisticated, and usually include:

- **Multitasking**, the ability to run multiple programs simultaneously.
- **Security** measures to prevent errors or malicious code from affecting the system.
- **Memory management** to coordinate the use of memory by multiple programs.
- **Networking**, to allow the exchange of information between separate computers.
- A **User Interface** (UI) which allows the user to start programs and interact with them. Most modern operating systems use a **Graphical User Interface** (GUI).
- An **Application Program Interface** (API) to allow programs to access system resources such as files, networks, and other I/O devices.

Operating Systems have now become as important as the CPU hardware itself, and they are an integral part of every general-purpose computer system.

The only modern computers without an operating system are typically “embedded” systems which are built into a device (such as a VCR or microwave oven). And, in fact, even many of them use simple operating systems to simplify the job of programming them.

Exercise 2 - What is an Operating System?

1.1.3

Which of these is an Operating System?

- A – Internet Explorer**
- B – Microsoft Word**
- C – Microsoft Windows**
- D – Norton Anti-Virus**
- E – Microsoft Media Player**

The Invention of Microprogramming

1.1.3

The simple circuitry used in early computers meant that they could only execute very simple instructions. For example, it took four instructions to add two numbers together:

LOAD	A	// Move memory variable A to the ALU
LOAD	B	// Move memory variable B to the ALU
ADD		// Add the two variables together
STORE	C	// Move the ALU result to memory variable C

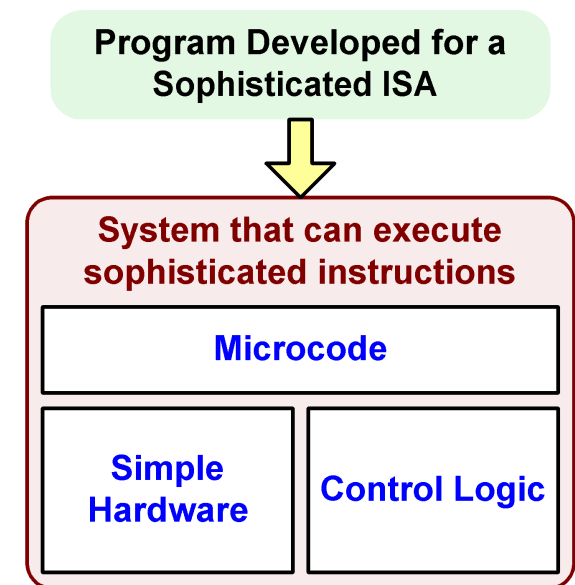
In 1951 Maurice Wilkes envisioned a machine whose control logic could automatically step through several of these simple operations when it executed an ISA instruction. For example, the programmer could code an ISA instruction such as:

ADD	A, B, C	// Add memory variables A and B, store result in C
-----	---------	--

When the control unit executes the “ADD” instruction, it would automatically perform the four simple LOAD, ADD and STORE instructions to do the equivalent work.

This technique is called “**microprogramming**”, because each ISA instruction in turn causes a series of one or more of these simple “micro-instructions” to be executed by the CPU.

Microprogramming allows a computer system with very simple hardware to understand more sophisticated ISA instructions. For example, such a system might be able to execute division or multiplication instructions even if the machine’s hardware is only able to perform addition and subtraction.



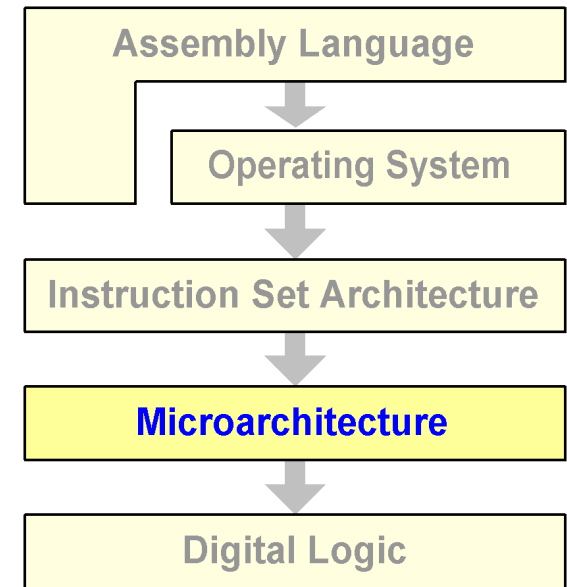
The Microarchitecture Level

1.1.3

A computer using micro-instructions introduces a new layer of organization called the **Microarchitecture** level.

The microarchitecture level is a very powerful concept. In the 1960's IBM introduced a family of mainframe computer systems called the System/360. They ranged from a (for the time) relatively inexpensive 8-bit machine all the way up to a very expensive 64-bit system.

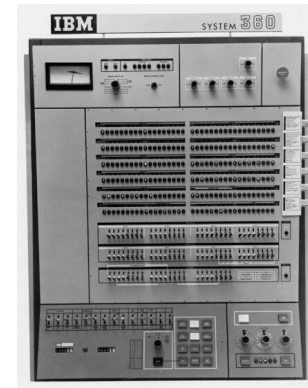
Although the hardware in each member of the family was very different, microprogramming allowed all of these systems to be able to run exactly the same programs – the first time such a thing had been done.



IBM 360/22



IBM 360/50



IBM 360/65

(Photos: http://www-1.ibm.com/ibm/history/exhibits/mainframe/mainframe_album.html)

By the 1980's almost all computer systems used microprogramming

Problem Oriented Languages

1.1.3

Even with the sophisticated instructions made possible by a microarchitecture, writing programs in assembly language is difficult and requires a deep understanding of the computer. In the mid 1950's a team lead by John Backus worked to create a “**Problem Oriented Language**” to overcome this hurdle.

The result was **FORTRAN** (“FORmula TRANslator”), generally considered to be the first problem oriented language. FORTRAN is a special piece of software that allows a programmer to write a program using algebraic-like statements such as:

$$A = B + (C * D)$$

...and converts them to the simple ISA instructions that can be executed by the computer hardware.

This allowed programs to be written with fewer lines of code using a notation that more closely matched the “problem” being worked on.

```
SUBROUTINE QUADR(A,B,C,DISCR,X1,X2,VX,VY,FL,FPY)
REAL A,B,C,DISCR,X1,X2,VX,VY,FL,FPY

C DISCRIMINANT, VERTEX, FOCAL LENGTH, FOCAL POINT Y
DISCR = B**2.0 - 4.0*A*C
VX = -B / (2.0*A)
VY = A*VX**2.0 + B*VX + C
FL = 1.0 / (A * 4.0)
FPY = VY + FL
FL = ABS(FL)

C COMPUTE THE ROOTS BASED ON THE DISCRIMINANT
IF(DISCR) 110,120,130

C -VE DISCRIMINANT, TWO COMPLEX ROOTS, REAL=X1, IMG=+/-X2
110 X1 = -B / (2.0*A)
X2 = SQRT(-DISCR) / (2.0*A)
RETURN

C ZERO DISCRIMINANT, ONE REAL ROOT
120 X1 = -B / (2.0*A)
X2 = X1
RETURN

C +VE DISCRIMINANT, TWO REAL ROOTS
130 X1 = (-B + SQRT(DISCR)) / (2.0*A)
X2 = (-B - SQRT(DISCR)) / (2.0*A)
RETURN

C
C NEXT STORE SUBROUTINE ON DISK USING DUP
END
```

High Level Languages

1.1.3

FORTRAN was designed to make it easy to encode formulas and so it was heavily used by the scientific and engineering communities, but languages to solve other types of specific problems soon followed, such as:

- COBOL (Common Business Oriented Language)
- SNOBOL (String Oriented Symbolic Language)
- LISP (List Processing Language)

However general purpose languages are now the most commonly used, and include:

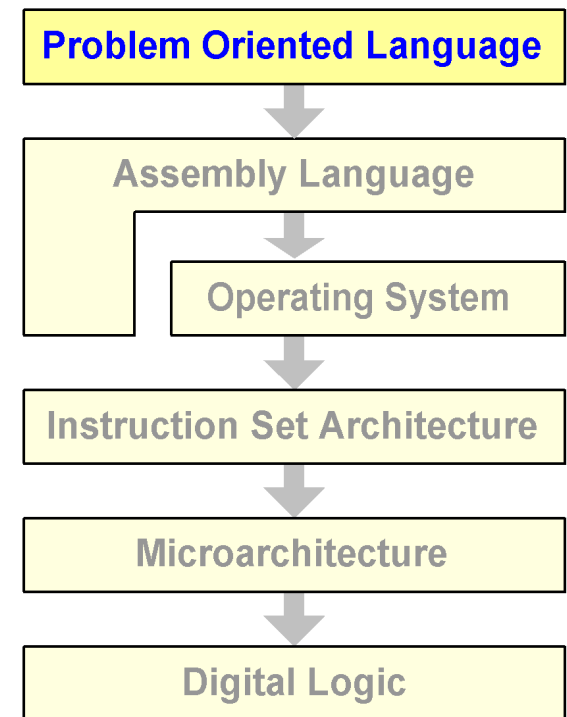
- C / C++ / C#
- Java
- BASIC

Virtually all formal programming is now done using Problem Oriented Languages (also known as **High Level Languages**), and they have become an important architectural level in modern computer systems.

Note that the computer hardware still only understands simple ISA instructions, so before high level language statements such as:

C = A + B;

...can actually be executed they must first be converted into simple ISA instructions such as **LOAD**, **ADD**, and **STORE**.



High Level vs. Low Level Languages

1.1.3

Here is a comparison of some of the characteristics of a high level languages vs. Assembler and Machine language (which are considered to be “Low Level” languages):

Low Level

- closely linked with machine language
- specific to a particular CPU / OS, difficult or impossible to move to a different type of machine
- requires more lines of code to do a given amount of work
- costs more (time and money) to write a program
- requires understanding of how the hardware works
- more direct control over how the work is done

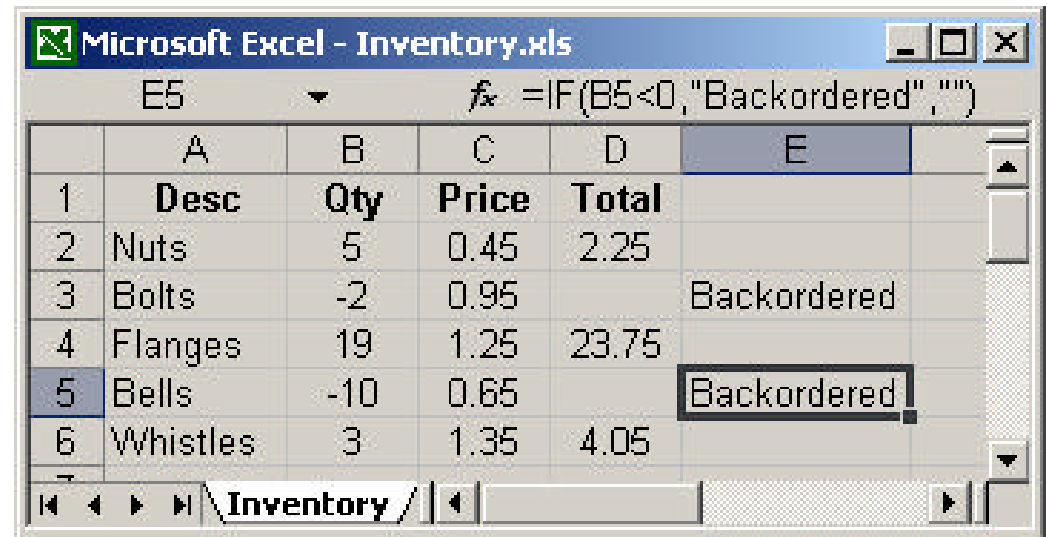


High Level

- independent of machine language
- more usable with any CPU / OS, can move to any type of machine that supports the language
- requires fewer lines of code to do a given amount of work
- costs less (time and money) to write a program
- no need to understand how the hardware works
- less direct control over how the work is done

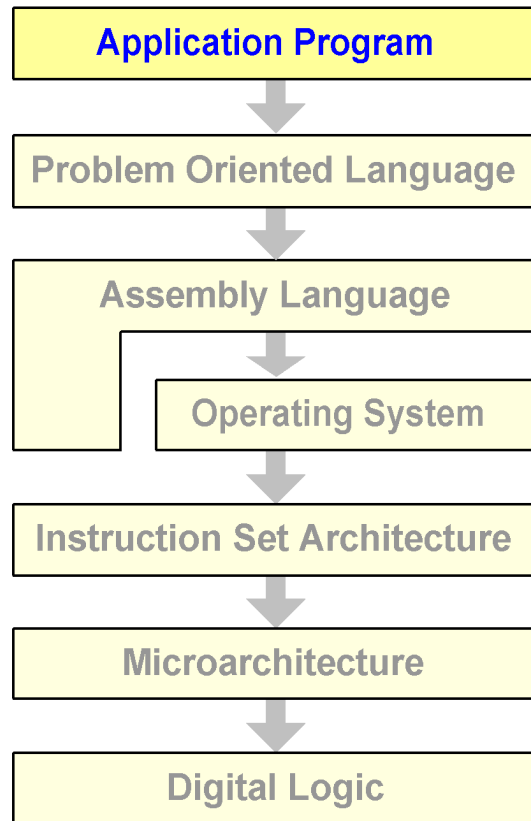
Application Programs

In the 1980s Personal Computers created the opportunity for application programs that interacted directly with the users. Many application programs now allow the user to “program” the computer in application-specific terms. An example is a spreadsheet, whose individual cells can contain instructions to manipulate and present data as required by the user.



The screenshot shows a Microsoft Excel window titled "Microsoft Excel - Inventory.xls". The active cell is E5, containing the formula `=IF(B5<0,"Backordered","")`. The spreadsheet has the following data:

	A	B	C	D	E
1	Desc	Qty	Price	Total	
2	Nuts	5	0.45	2.25	
3	Bolts	-2	0.95		Backordered
4	Flanges	19	1.25	23.75	
5	Bells	-10	0.65		Backordered
6	Whistles	3	1.35	4.05	

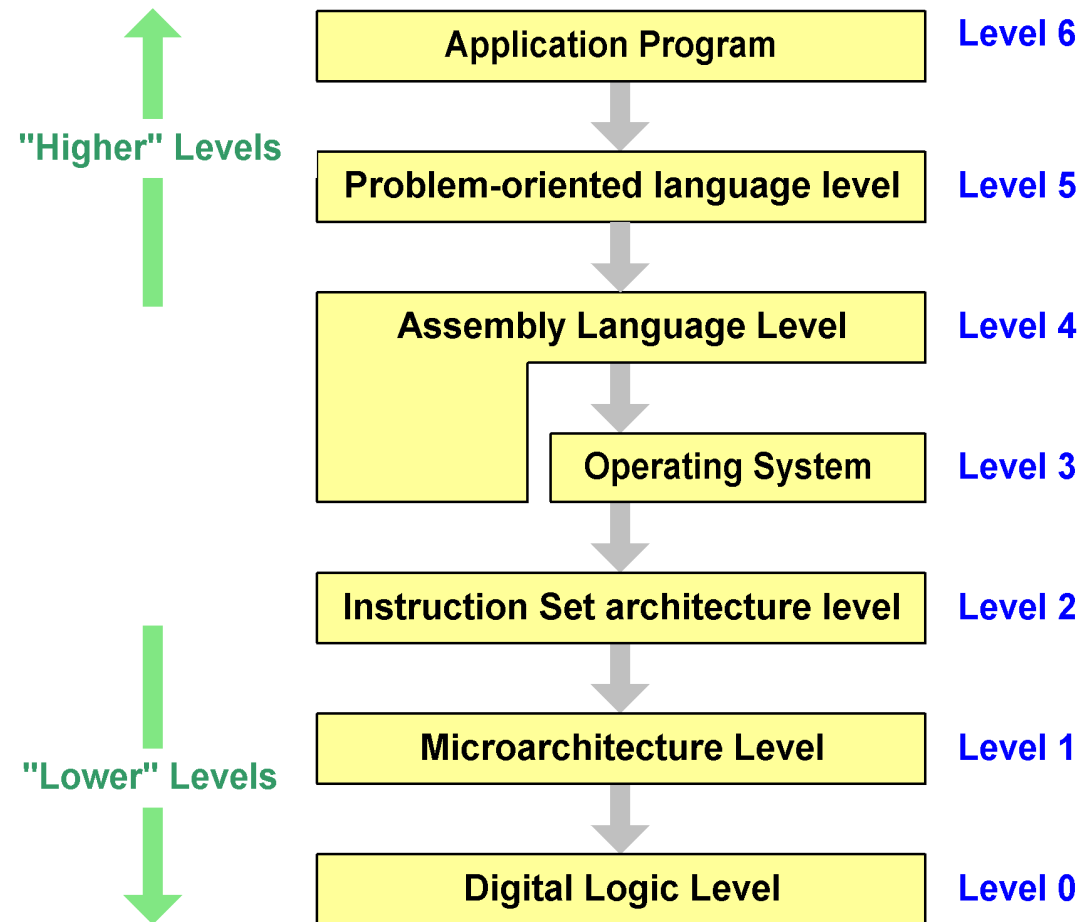


Not all applications provide a means of programming, and of the ones that do not all are Turing Complete. But many applications have provided a whole new level at which users can give instructions to the computer system without any special knowledge of traditional computer languages or the underlying computer hardware.

Modern Multilevel Architectures

1.1.2

So we've seen that modern computer systems can be thought of as having many distinct levels of organization:



Different computer systems can implement different layers in different ways. For example, the Digital Logic level could be built using relays, vacuum tubes, or transistors.

Hardware vs. Software

1.1.2

Note that the Instruction Set Architecture forms the boundary between “**Hardware**” and “**Software**”.

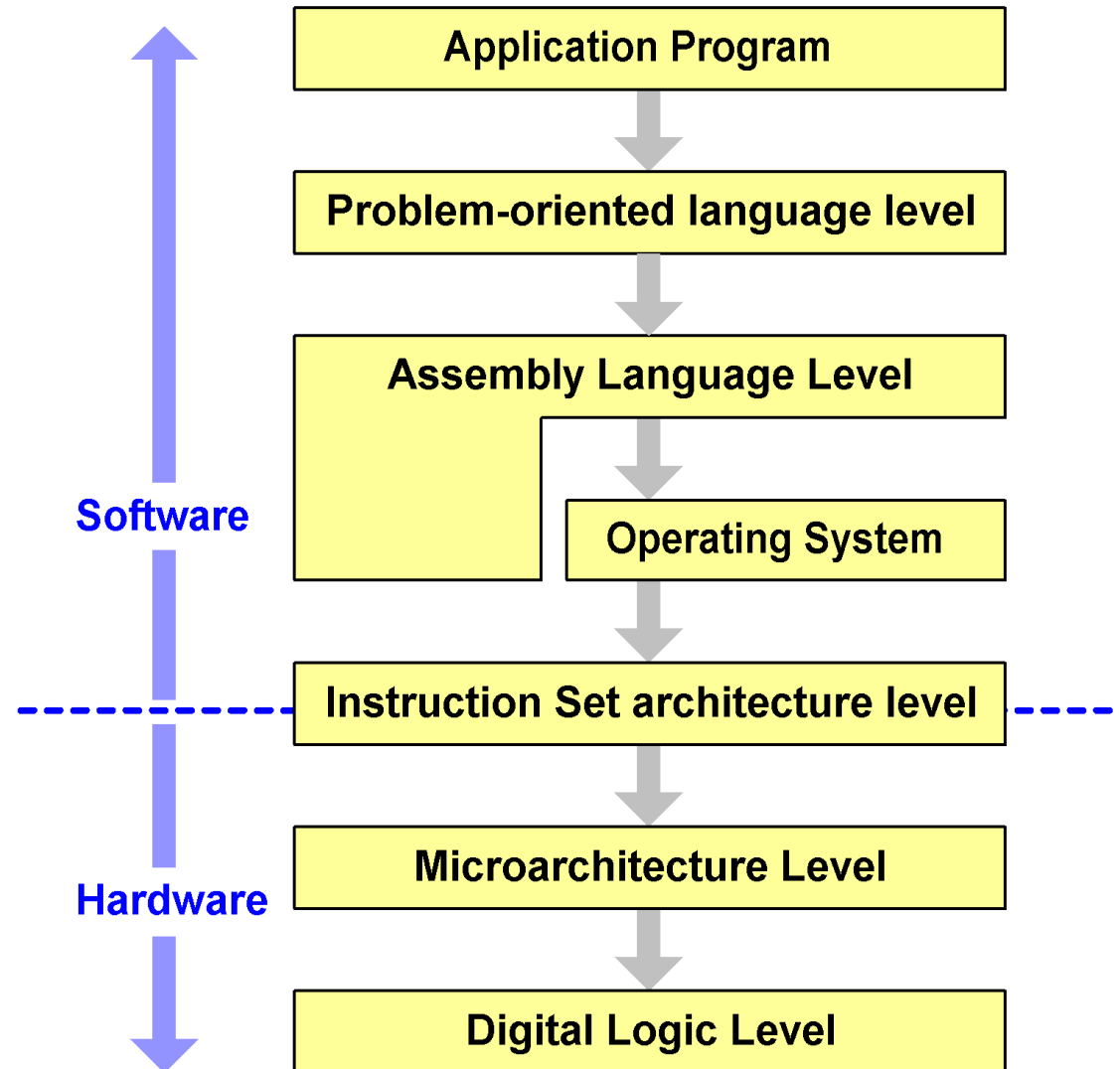
“**Hardware**” consists of physical devices such as transistors, data paths, and memory.

“**Software**” consists of algorithms expressed in a language that hardware can understand and contained on some sort of media (disk, memory).

The hardware executes the software (i.e., it follows the instructions given by the software).

The boundary between which functions are performed by software and which are built into the hardware can change from one machine to another.

We’ll now take a look at the development of RISC systems, which provides an excellent example of this.



The Zenith of Microcode

1.1.3

The popularity of microcoded architectures continued to grow through into the early 1980s because it was easy to create systems with very sophisticated capabilities.

The peak of this trend came with the Digital Equipment Corporation VAX (Virtual Address eXtension) computer family, which included ISA-level instructions to perform tasks such as:

- Evaluating polynomials
- Calculating CRCs (Cyclic Redundancy Checksums) used in error detection codes
- Inserting and removing items in linked lists

Digital Equipment Corp VAX



(Photo: http://www.montagar.com/dfwcug/VMS_HTML/timeline/1977-3.htm)

RISC is Developed

1.1.3

In the 1980s researchers started to question the effectiveness of microcoded architectures. They analyzed actual computer programs and found:

- Complex instructions like “CRC” or “Evaluate Polynomial” make up only a very tiny percentage of most programs and are only very rarely executed.
- The most frequently executed instructions are very simple arithmetic, comparison, and data movement instructions
- Heavily microcoded architectures weren’t as efficient for these simple instructions as a direct hardware implementation could be

The researchers reasoned that computers which eliminated the microcode used to provide complex instructions and instead used pure hardware to implement the commonly-used simple instructions would be faster.

Systems designed this way were indeed faster, and as a result, most of the new computer architectures introduced since the mid 1980’s use this design. These systems are now known as “**RISC**”:

Reduced Instruction Set Computer

...while the older microcoded architectures have now become known as “**CISC**”:

Complex Instruction Set Computer

RISC Effectiveness

2.1.3

It takes more simple ISA-level instructions for a RISC processor to do the same work as the more complex instructions of a CISC processor, but because the instructions are able to run faster it takes less overall time for the RISC processor to do the work:

RISC Instructions

LOAD A	(1ns)	(Total = 4ns)
LOAD B	(1ns)	
ADD	(1ns)	
STORE C	(1ns)	

CISC Instructions

ADD A, B, C	(6 ns)
-------------	--------

RISC systems have become popular because:

- Most programs are now written using problem-oriented languages, so there's no longer much of a benefit for having complex instructions.
- The modern computers have improved in speed faster than memory has, so memory has become a bottleneck. The instructions on RISC processors don't require memory to be accessed as often, and this helps to boost speed even further.
- RISC processors usually provide a lot of temporary storage space ("**registers**") to hold data being worked on which further reduces the impact of slow memory.

RISC Design Principles

2.1.4

Modern processors, even CISC processors such as the Intel Pentium, have adopted a common set of general design principles which help the system run very quickly:

Execute most instructions directly by hardware

This is faster than using microcode and helps make all instructions take the same time to execute – important in a pipelined architecture (discussed below).

Maximize instruction issue rate

In a pipelined architecture, it's much more important to be able to start (“issue”) instructions quickly than it is to be able to complete them quickly.

Make instructions easy to decode

If it takes a long time to figure out what an instruction is supposed to do, you can't maximize issue rate.

Eliminate memory references except for Loads and Stores

This encourages the use of registers which are fast, and it also allows memory references to take place in separate instructions which can execute in parallel with register-based instructions.

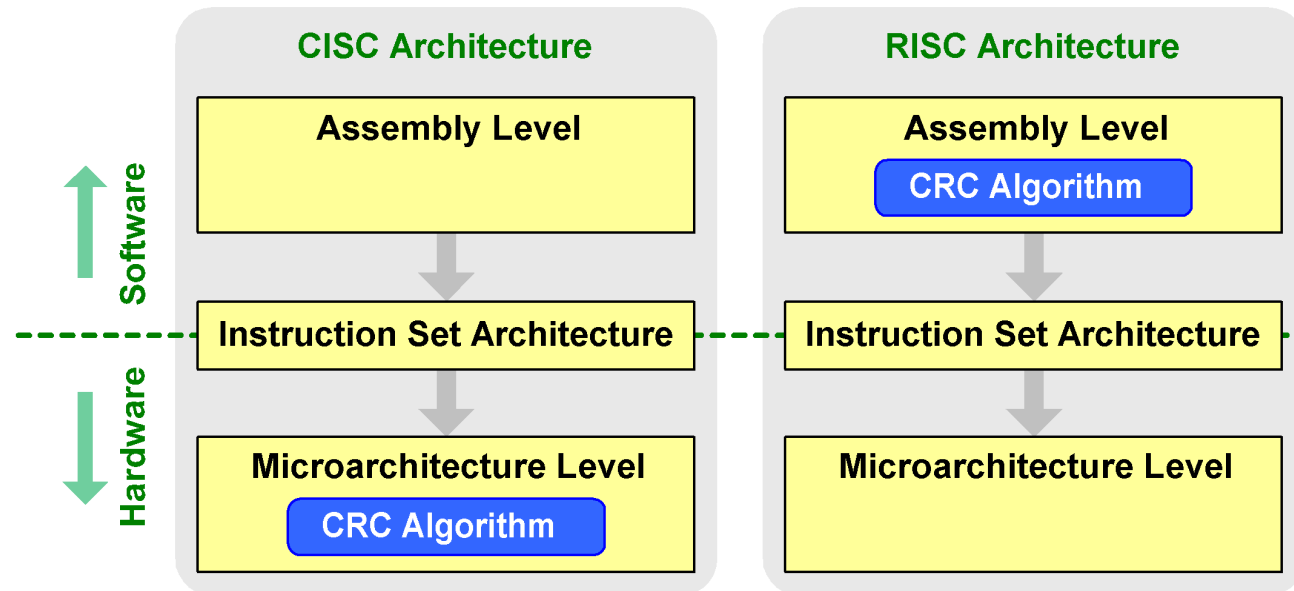
Provide Plenty of Registers

If you discourage the use of memory, you have to make up for it with lots of high-speed registers which allow data to be loaded directly into the ALU for fast access.

Multilevel Machine Evolution

1.1.3

You can see that some functions (such as the calculation of CRC checksums) which are handled in hardware for a CISC system have been relegated to software in a RISC system:



From this we can see that hardware and software are logically equivalent. It's much harder to change an algorithm in hardware than in software, but hardware can often produce results much faster than software.

“Hardware is Just Petrified Software”

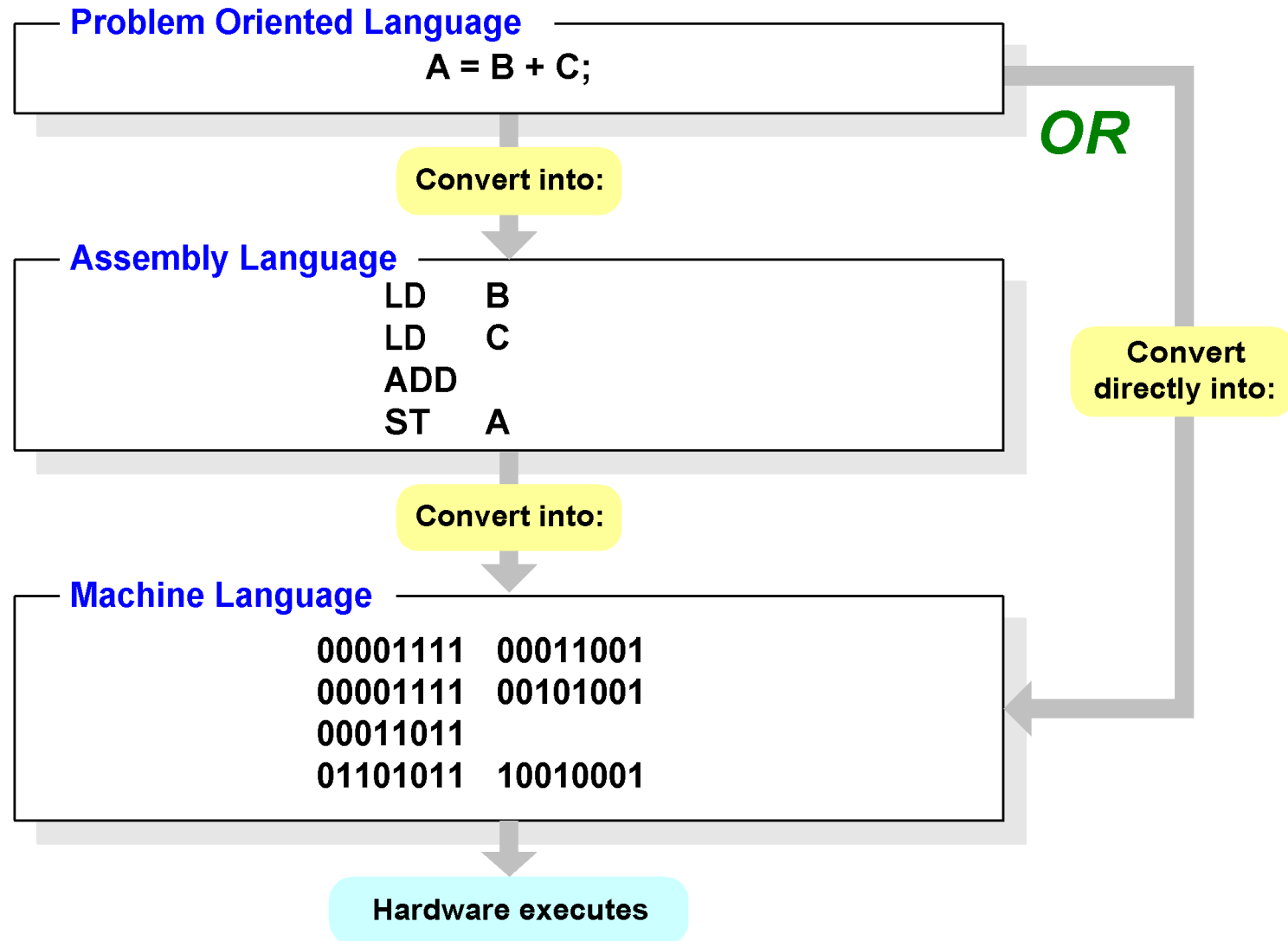
(Karen Panetta Lentz)

As the cost of hardware falls, and as the algorithms for delivering standard computing solutions are better understood, the trend is for the “dividing line” between hardware and software to move up to higher levels in a computer system (but RISC is a notable exception to this general trend).

Conversion Between Language Levels

1.1.1

A program written in anything other than machine language must be converted to machine language before the computer system can execute it.



Interpretation vs. Translation

1.1.1

There are two ways to convert a program from a higher level language to a lower level language “[Interpet](#)” it or “[Translate](#)” it:

Interpretation

- Program is converted as it executes (at “run time”)
- “Interpreter” is a special program which reads the high level language and executes equivalent machine language instructions
- Interpreters often use an intermediate language format for efficiency
- “Interpretation” takes longer to run the program but doesn’t require it to be translated ahead of time
- Programs can run on any CPU for which an interpreter is available
- Analogy: Having an interpreter read a foreign-language book to you
- Examples: Java

Translation

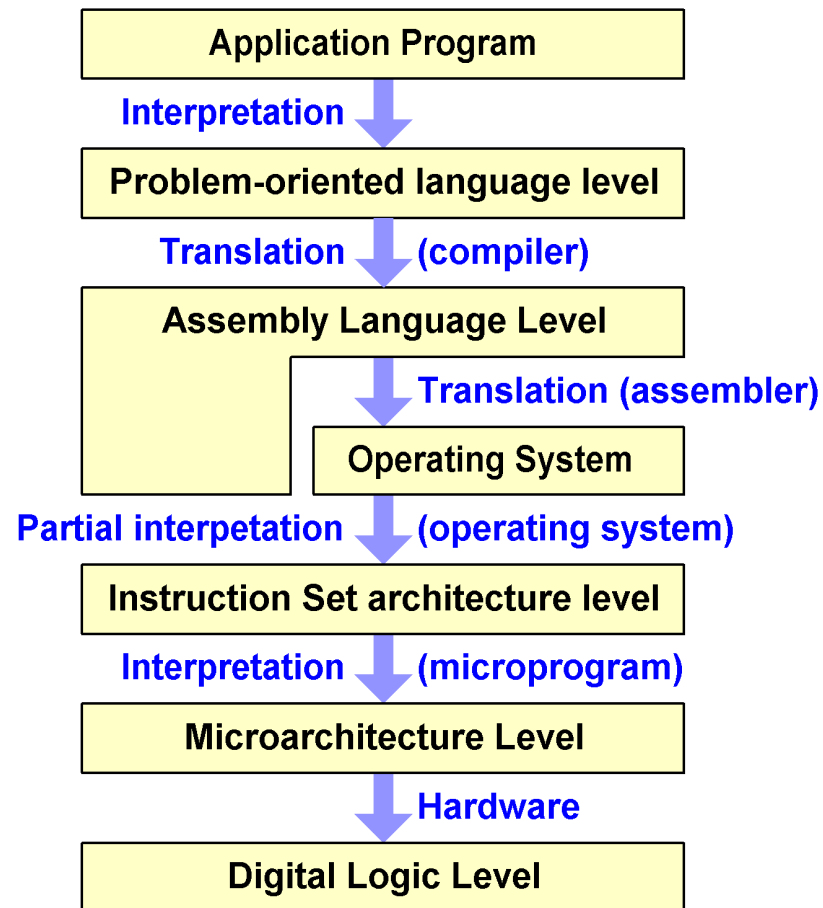
- Also known as “compiling”
- Program is converted once by a “compiler” program and the result is saved for repeated use
- The saved (“executable”) form is actual machine instructions which can be loaded into memory and executed directly by the CPU
- Translated programs run faster, but changes require re-translation
- Translated programs are CPU-type-dependent
- Analogy: Reading a translation of a foreign-language book
- Examples: C, C++ (but not with Microsoft.Net)

A method known as “[Just-in-Time](#)” compilation combines the best of both methods by translating the program immediately before it’s executed.

Multi-Level Execution

1.1.1

The various layers in a multi-level architecture often use a combination of translation and interpretation in the process of executing a program:



This diagram is not universal, and the execution method for each level varies from language to language and machine to machine.

Virtual Machines

1.1.1

A computer programmer doesn't have to be aware of whether his program is translated or interpreted, and he doesn't even have to be aware that there are lower language levels involved in the execution of his program. To the programmer, the language he works in can be considered as a “**virtual machine**” which provides the facilities he needs.

- Example – “Windows”, “buttons” and “menu bars” in modern computer systems are high-level constructs created by the operating system. Programmers don't have to be aware of what makes them work, only the interface that is used to control them.

A common term for this concept is “black box”. A “black box” is any device that provides some defined function, but whose internal operation is unknown or irrelevant.

- Example – You can drive a car even if you don't know whether it uses a gas, diesel, or electric engine. The engine can be thought of as a “black box” which provides more power when you press down harder on the “gas” pedal.

The user of a virtual machine doesn't have to know how it operates, only what inputs it needs to be provided with and what outputs to expect.

- The person who implements (builds) a virtual machine must come up with a way to make it do what it's supposed to do
- Knowing the internal workings of virtual machine can help get the most out of it

A Programming Example of the Black Box Concept

1.1.1

Let us suppose you've written a subroutine which can calculate the average of a list of numbers. Once this routine has been written it can be catalogued to a library and made available to other programmers as well:

**“avg”- a subroutine containing
instructions to average some numbers**

```
int avg(int *list, int count)
{
    int *cur, sum, loop_ctr;
    loop_ctr = count;
    for(cur = list; loop_ctr; loop_ctr--)
        sum = sum + *cur++;
    if (count > 0)
        return (sum/count);
    else return 0;
}
```

instructions that use “avg”

```
if ( avg(gradelist, count) > 50 )
    status = Passed;
else
    status = Failed;
```

Once the subroutine has been written, other programmers can use it as if it's a “black box”. They don't have to know how it works, only what parameters to pass and what result to expect.

What a programmer needs to know to use the subroutine is called it's specification.

How the subroutine does it's work is called it's implementation.

Implementation vs. Specification

1.1.1

A particular level of a multilevel computer system must perform work in accordance with its **specification**. The specification describes **what is supposed to be done** with a given set of inputs or instructions.

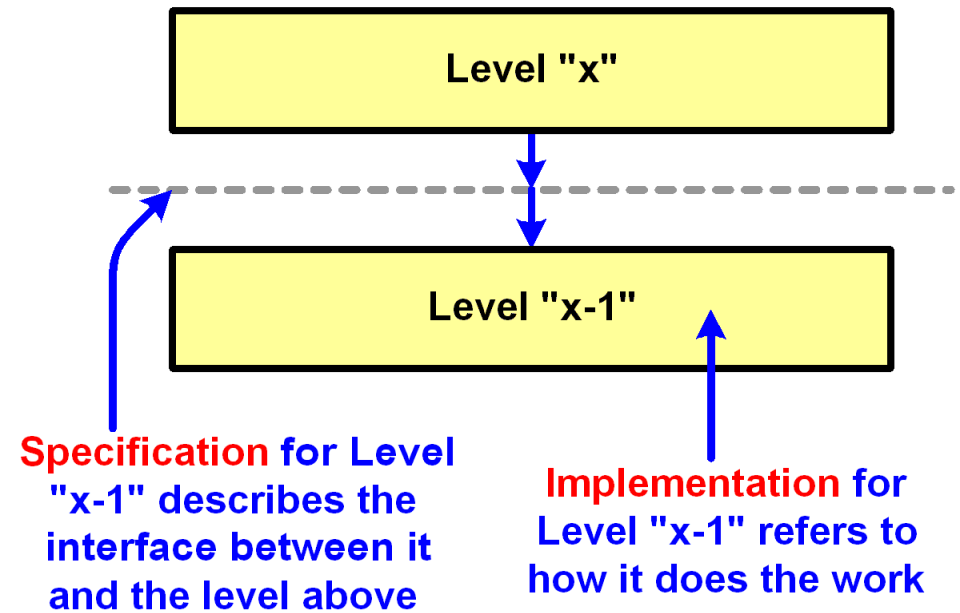
There are usually many different ways that a given level **could** give the specified results. The **actual** hardware or software that does the work is known as the **implementation**.

You can't change the specification of a particular level without also affecting the level above. But it's quite possible to change the implementation without affecting the upper level.

As an example of the difference between specification and implementation, consider a calculator:

- The **specification** describes the keyboard and the operations you can perform (to add: enter a number, press "+", enter another number, press "=").
- The **implementation** describes how the calculator does its work (electronic calculator vs. electromechanical calculator, vs. a software calculator on a computer system).

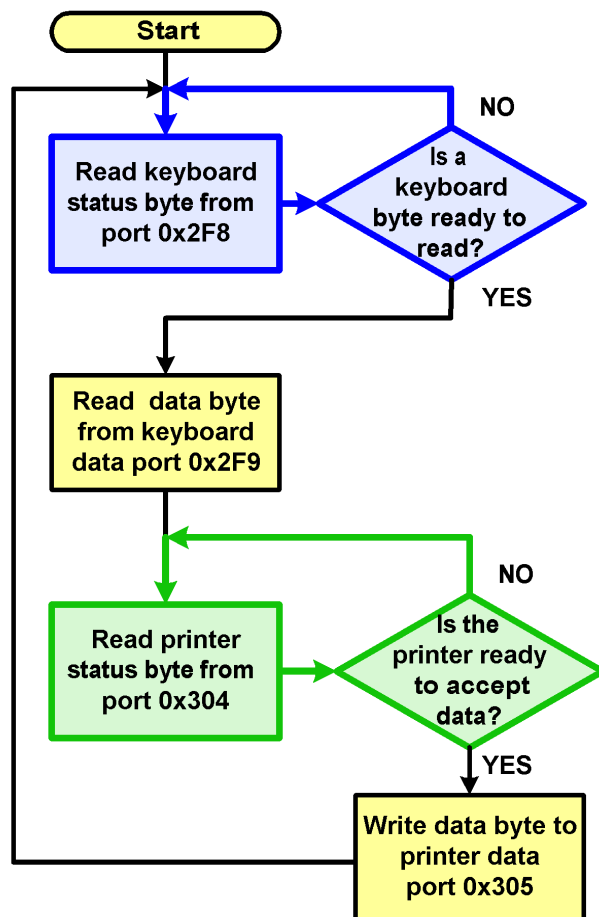
Each type of calculator would give the same results for the same combination of inputs (button presses), although the performance may vary.



Exercise 3 - Implementation or Specification?

1.1.1

Are the picture and diagram showing an Implementation (A) or a Specification (B)?

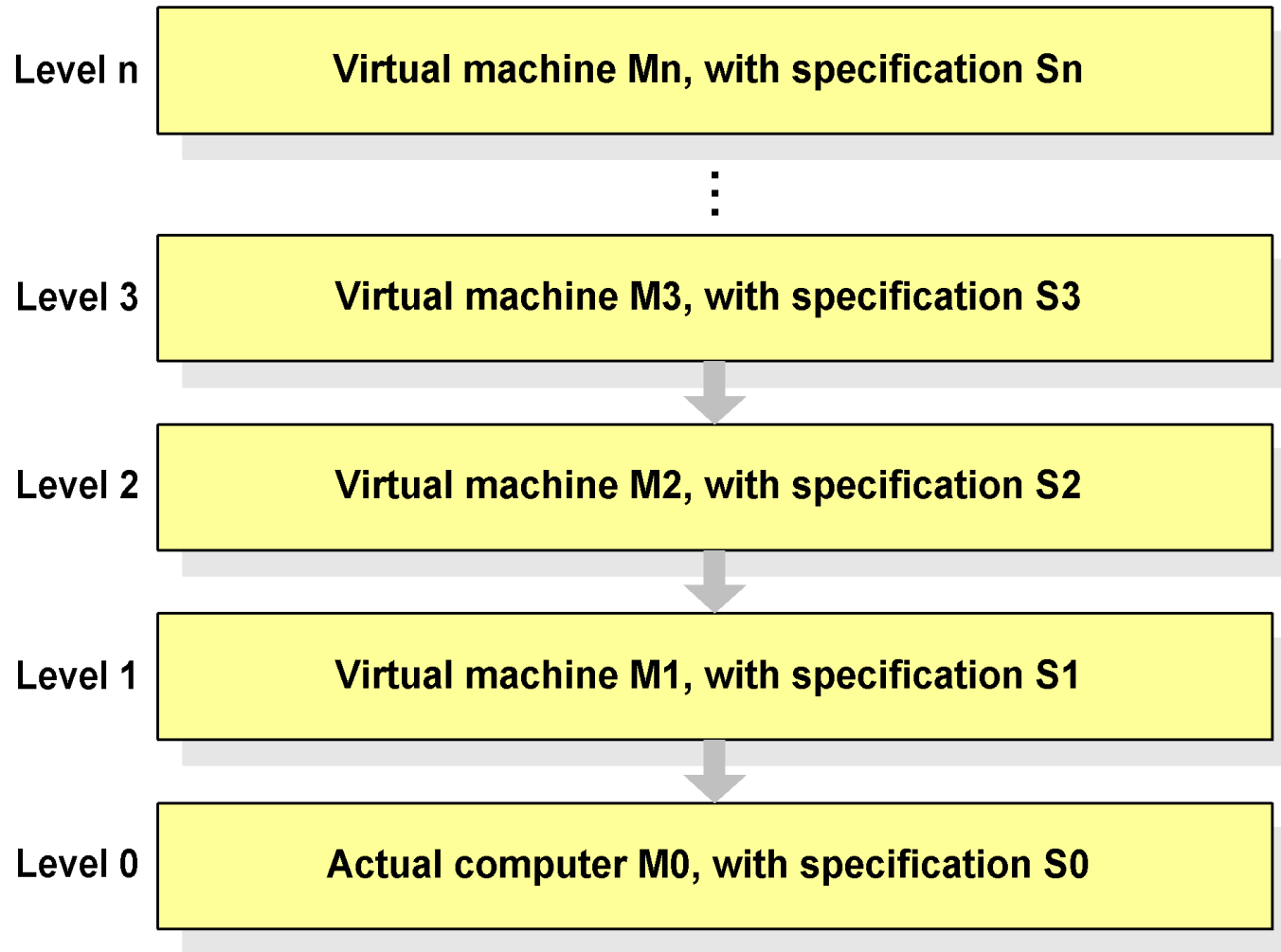


Note that the word “**Interface**” is often used in the same sense as “**Specification**”, particularly between two layers or subsystems which exchange data.

Virtual Machines in a Multilevel Architecture

1.1.1

In a multilevel computer architecture, each level acts like a virtual machine for the levels above. It has a specification that defines what it's supposed to do, and an implementation that depends on lower levels to function:



Sample Architectures

1.4

The textbook uses three real-world chip designs as samples of different architectural approaches to building a computer system:

Pentium-4

A 32-bit CISC design with a long lineage dating back to the early 1980's

UltraSPARC-III

A modern 64-bit RISC design

8051

A low-cost chip designed to be used in embedded systems

Pentium-4 Overview

1.4.1

The Pentium 4 is a recent entry in a family of compatible CPUs starting with original 8088 used in original IBM PC.

It's the last major CISC processor that's is still being designed, because backward compatibility with older versions is even more important than the best possible performance.

It's a classic microprogrammed design with a CISC instruction set, which uses a RISC core in order to boost performance.

Because of it's tremendous popularity, other manufacturers have also created their own CPUs which can run the same programs. Pentium's main competitor is currently Advanced Micro Devices.

Chip	Year	MHz	Memory	Notes
4004	1971	0.108	640	First single-chip CPU
8008	1972	0.108	16K	First 8-bit microprocessor
8080	1974	2	64K	1 st general-purpose μ proc
8086	1978	5-10	1M	First 16-bit CPU on a chip
8088	1979	5-8	1M	Used in original IBM PC
80286	1982	8-12	16M	Incl. memory protection
80386	1985	16-33	4G	First 32-bit CPU
80486	1989	25-100	4G	Incl. 8K cache memory
Pentium	1993	60-233	4G	2 pipelines, MMX intro'd
Pentium Pro	1995	150-200	4G	Two levels of cache
Pentium-II	1997	233-400	4G	Pentium Pro plus MMX
Pentium-III	1999	450-1G	4G	Adds SSE instructions
Pentium-4	2000	1.4-3G+	4G	Hyperthreading, SSE2
Core	2006	1-2.6G+	4G	Multiple CPUs on a chip
Core 2	2006	1.9-3G	16E	64-bit Instruction Set

UltraSPARC-III Overview

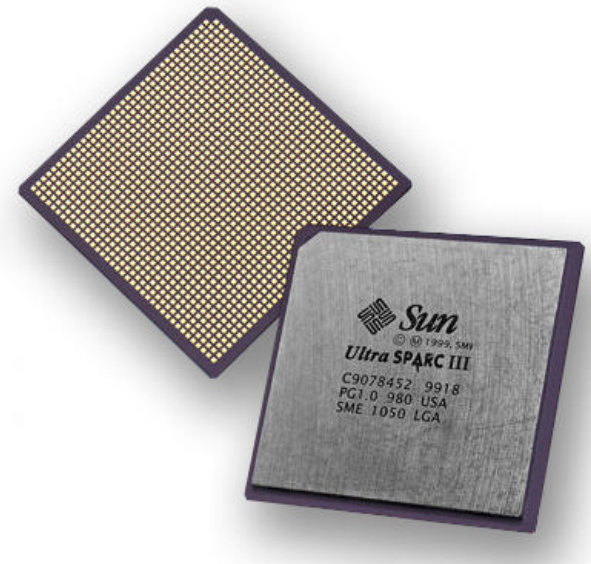
1.4.2

SPARC (**S**calable **P**rocessor **A**rchitecture) was designed by Sun Microsystems from the ground up as a RISC architecture to replace the CPU in their Sun workstations (which had previously used Motorola microprocessors).

The original SPARC was a 32-bit CPU with an architecture designed for expansion to 64 bits. The UltraSPARC-II and -III are full 64-bit CPUs, and more recent versions have included multiple CPU cores.

The architecture includes a parallel processing instruction set called VIS (Visual Instruction Set), similar to the MMX instructions on Pentium processors.

Unlike the Pentium design which is closely guarded by Intel, Sun has licensed the SPARC design to several semiconductor manufacturers (including Texas Instruments and Fujitsu) who build the actual chips.



8051 Overview

1.4.3

The 8051 is a member of a chip family created in 1976 for use in embedded systems such as microwaves, coffee makers, etc.

It's designed to be extremely cheap both in and of itself, (10-15¢ per chip in large quantities), and in terms of the overall circuitry needed to make a working system. It includes memory and I/O pins so that a complete computer system can be built very cheaply without having to add separate chips.

Chip	Mem Size	Mem Type	RAM	Timers	Interrupts
8031	0K		128	2	5
8051	4K	ROM	128	2	5
8751	8K	EPROM	128	2	5
8032	0K		256	3	6
8052	8K	ROM	256	3	6
8752	8K	EPROM	256	3	6

The table shows a few of the many configurations that are available for different needs.

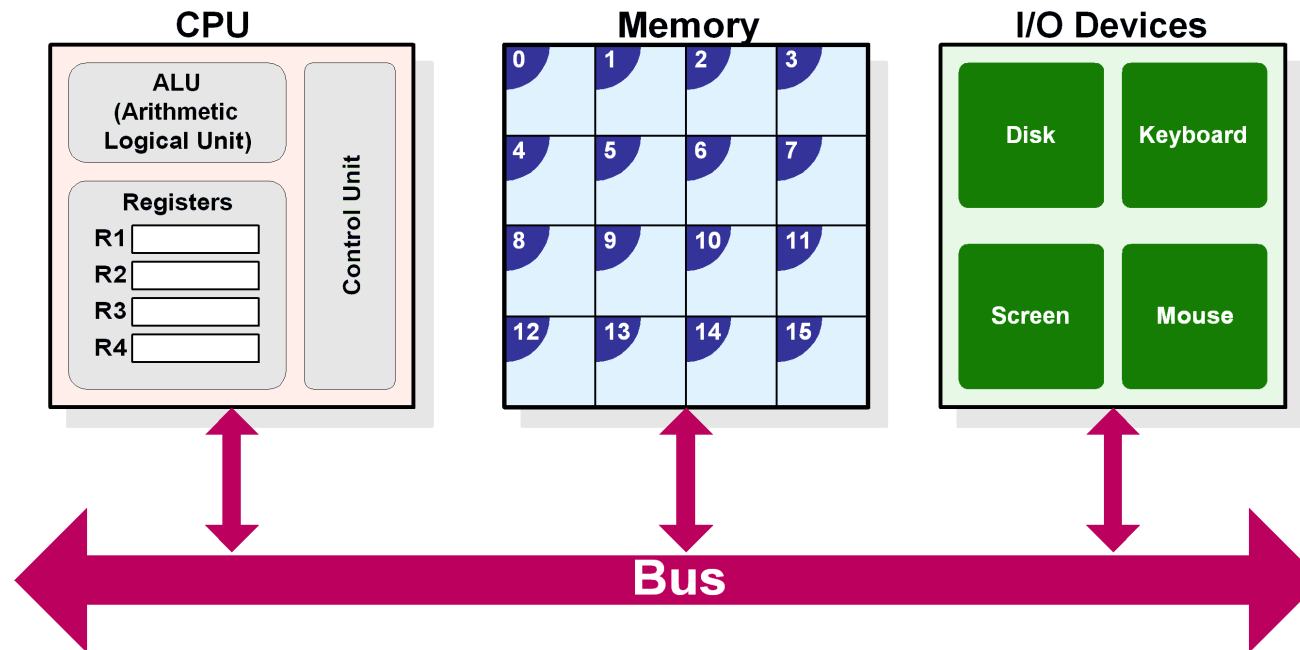
The chip is a CISC design, but this isn't an issue for the uses to which it's typically put.

Despite being very old, it's still very popular because it's readily available from many manufacturers and over the years a large number of software development tools have been created for it.

Computer System Organization

2.1

Computer systems consist of a **CPU** (Central Processing Unit), **Memory**, and **I/O devices**. In modern systems, these are connected by a **Bus** which moves data between them.



The **CPU** directs the operation of the system and controls when information is read or written to/from the **Memory** and the **I/O devices**. The CPU in turn follows the actions dictated by the instructions in the program.

Memory stores the data to be manipulated.

I/O devices are used to exchange information with the outside world.

CPU Organization

2.1.1

The CPU itself consists of Registers, an ALU (Arithmetic Logic Unit), and a Control Unit, as well as internal data paths which connect them.

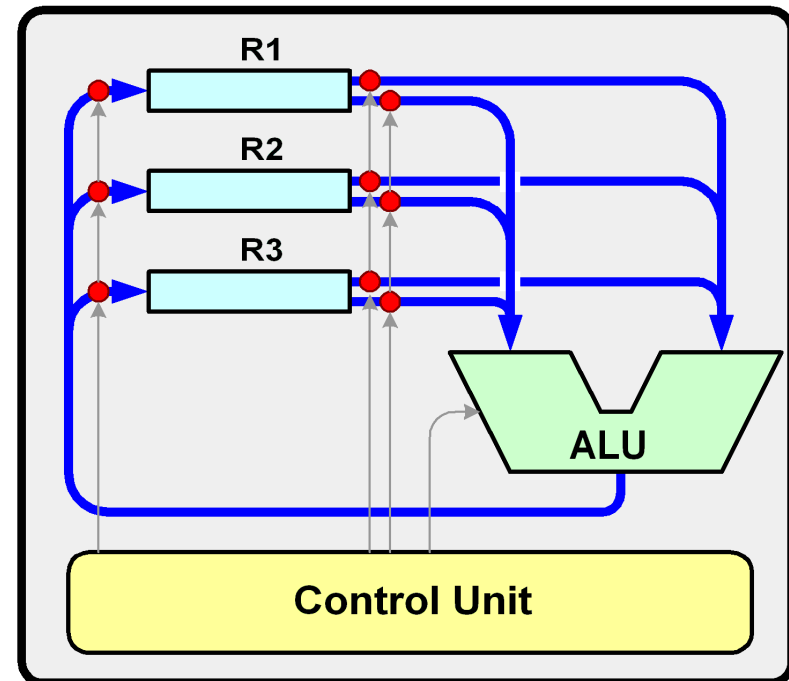
Each register is a temporary storage area that contains one item of data which can be fed into the ALU or receive its results.

The ALU is capable of performing various arithmetic operations such as Add, Subtract, Compare, etc.

The Control Unit recognizes program instructions and activates switches which allow the registers to be connected or disconnected to the ALU inputs and output, as well as the function (Add, Subtract, etc.) that the ALU is to perform.

By switching the connections in the correct sequence, the control unit causes the correct actions to take place as required by the program instructions.

The CPU



(Note – not shown here is the CPU's connection to the Bus so it can move information to and from Memory and I/O Devices)

Registers

2.1.1

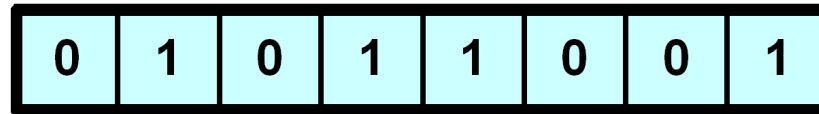
Registers are high-speed storage areas within the CPU chip used to stage data being manipulated by the ALU.

Like memory, Registers consist of a group of Binary digiTs (“bits”).

Depending on the CPU, Registers can hold a varying number of bits. Typical register sizes include 8, 16, 32, or 64 bits.

(Analogy: different calculators often use different numbers of digits)

An 8-bit Register



In order for a program to perform operations on data (ie, Add, Compare, etc.), the data to be used must first be loaded into a register.

RISC Processors usually have many more registers (dozens or hundreds) than CISC processors (a dozen or less).

Some registers can be used by the program for any purpose, while others are special registers that are needed to help control the operation of the CPU.

Executing a Program – Part 1

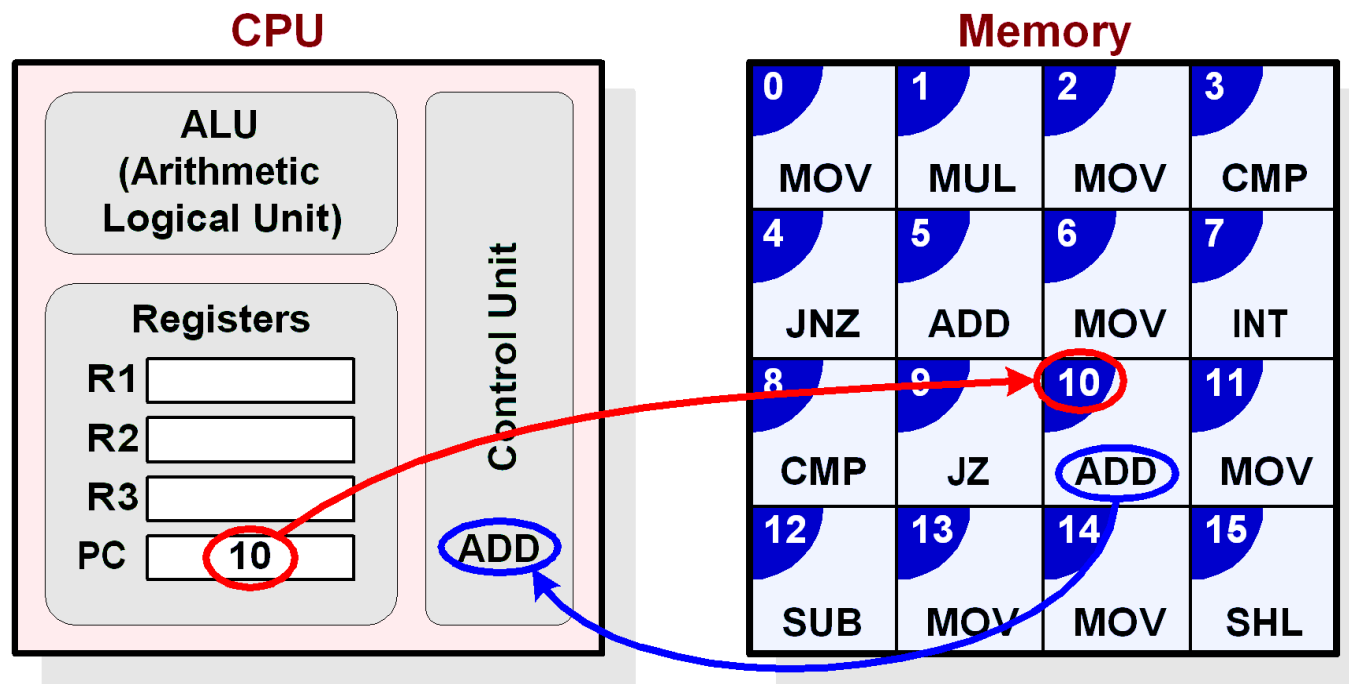
2.1.2

In modern Von Neumann-based computers, the program instructions are stored in memory. Therefore, to “execute” a program the CPU must fetch each program instruction from memory.

The CPU keeps track of which instruction to execute using a special **PC (Program Counter)** register. This register holds the memory address of the next instruction to be executed.

The CPU starts the process of instruction execution asking memory for the information stored at the address indicated by the PC register.

The actual instructions stored in memory are special bit patterns (ie, “00010100”) that are recognized by the Control Unit.

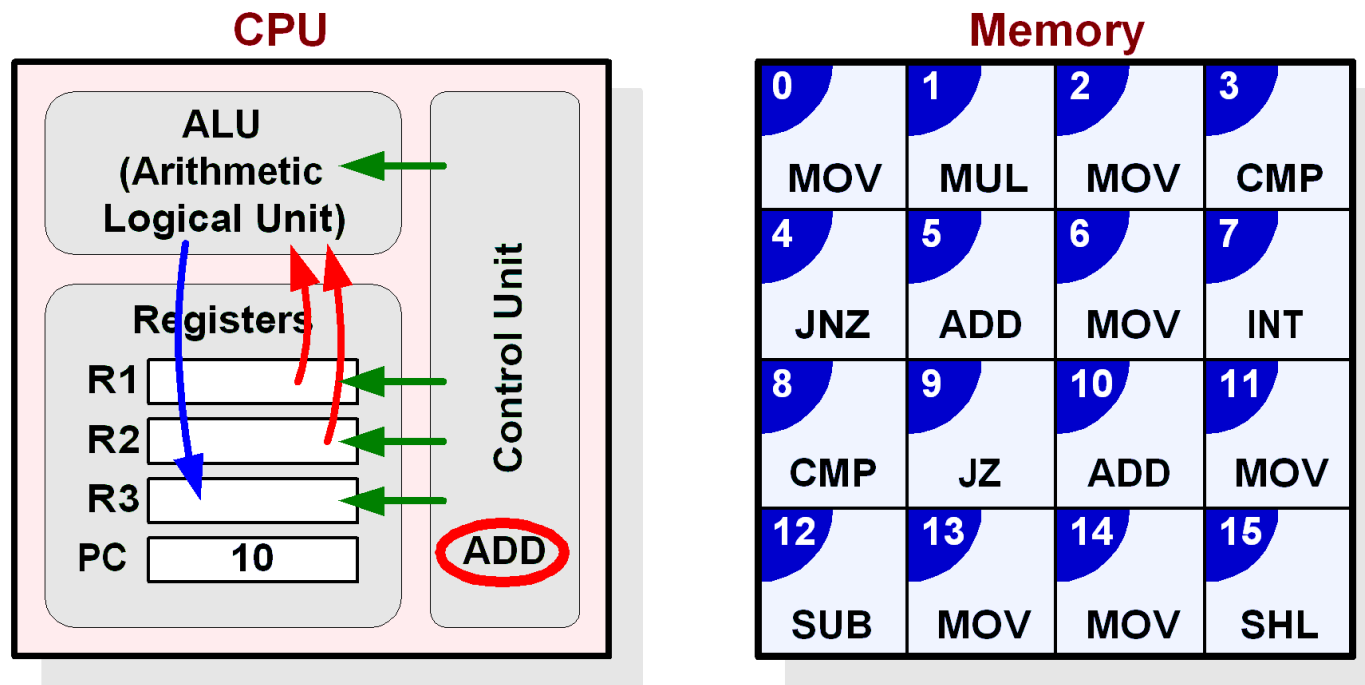


Executing a Program – Part 2

2.1.2

Once the instruction has been fetched, the Control Unit “decodes” it and closes the appropriate switches to allow the registers and ALU to perform the requested operation.

When the switches are closed, data flows into the ALU and out again, and the result is stored in one of the registers.



Executing a Program – Part 3

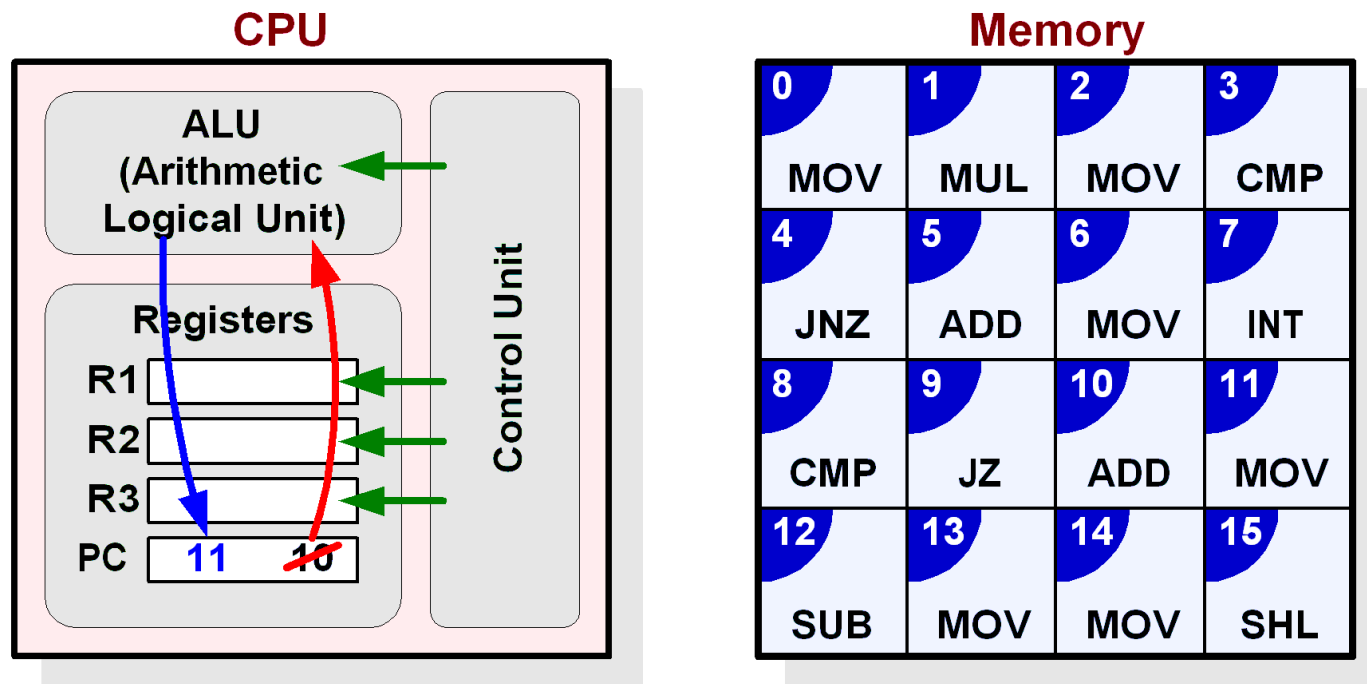
2.1.2

Finally, the CPU prepares for the execution of the next instruction by adding one to the PC register (“incrementing” it).

The PC register is sent to the ALU with the command to “ADD 1” to it. The result is stored back in the PC register again.

The resulting PC value points to the next program instruction in memory.

The CPU is now ready to begin execution of the next instruction.



Executing a Program

2.1.2

A CPU executes a program by using the PC (Program Counter) register to track where the next instruction is. Each instruction is executed by following a simple algorithm:

- **Fetch** an instruction from memory at the address contained in the PC register.
- **Decode** the instruction and set the switches which connect the Registers and ALU to **execute** the appropriate operation.
- **Increment the PC** so it points at the following instruction.

Since this sequence of steps is itself an algorithm, it can either be done directly by hardware, or software can be built to act as an interpreter of the program instructions:

- In a **RISC** computer, the bit patterns in the program instruction are filtered through digital logic circuits to directly activate the switches which connect the registers and ALU and cause the system to perform the correct operation.
- In a **CISC** computer, the control store executes a sequence of microinstructions. Each microinstruction controls the switches which connect the registers and ALU. When there is more than one microinstruction then the amount of work done can be more complex than for a simple single “hardwired” instruction.

Improving Performance with Parallelism

2.1.5

In order to improve performance beyond the general level of faster circuitry, modern computer systems use parallelism to increase throughput. Most of the digital logic in a modern CPU is dedicated to doing work in parallel.

There are two general forms of parallelism:

Instruction-Level Parallelism

This increases the amount of work done by a single CPU executing a stream of instructions (a “program”).

Advantage: Works without changing the basic sequential programming model.

Drawback: Difficult to extract parallelism from a sequential instruction stream.

Processor-Level Parallelism

This adds additional CPUs to a system so that more than one instruction stream (“program”) can be executed simultaneously.

Advantage: Easy to increase performance by adding CPUs. Works well to provide parallelism between separate instruction streams in different programs.

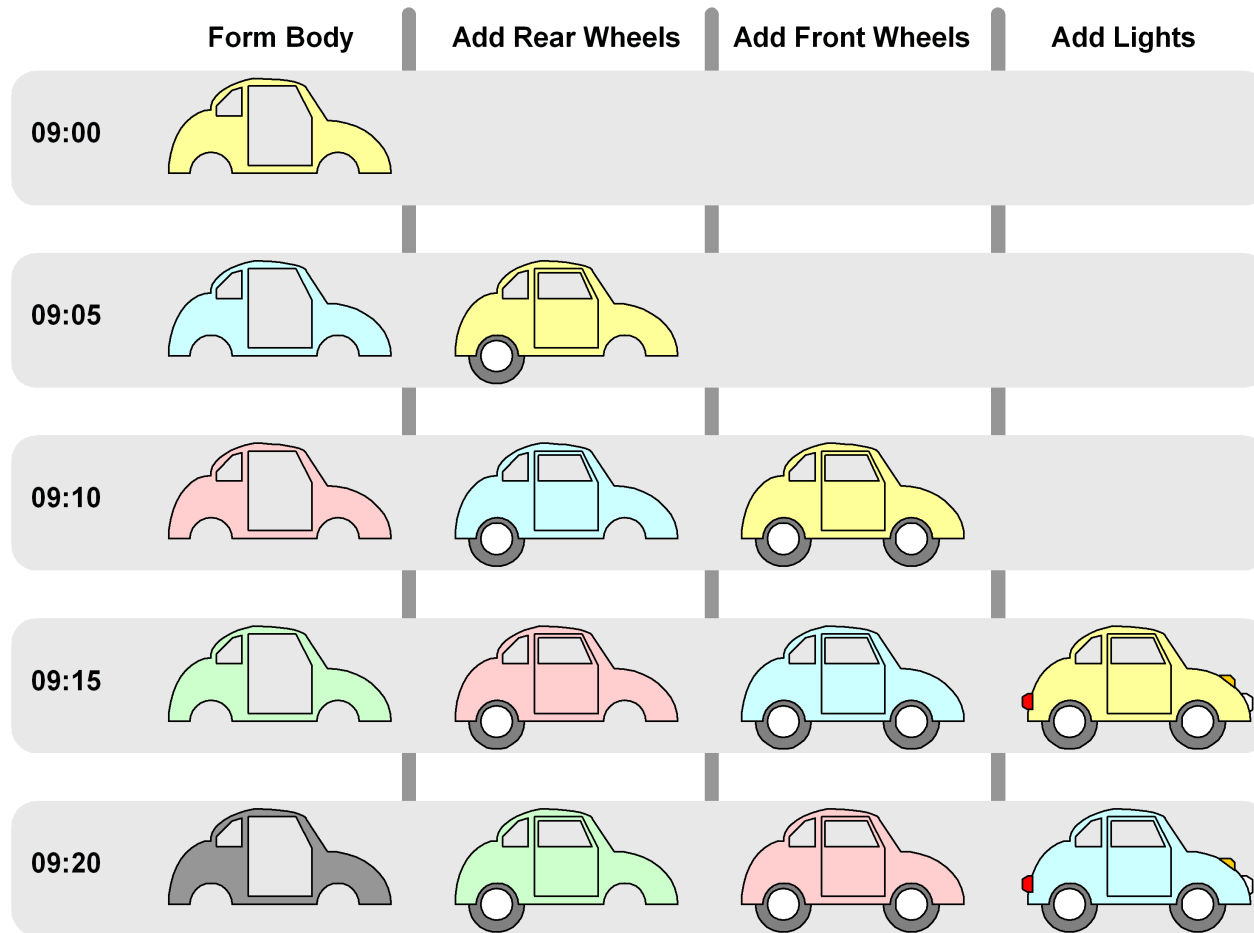
Drawback: Difficult to write a single program with parallel instruction streams to work on multiple CPUs.

Instruction-Level Parallelism

Pipelining

2.1.5

A pipeline is a way to break down the work done for each instruction and execute parts of it in parallel with adjacent instructions. It's very much like a car assembly line:



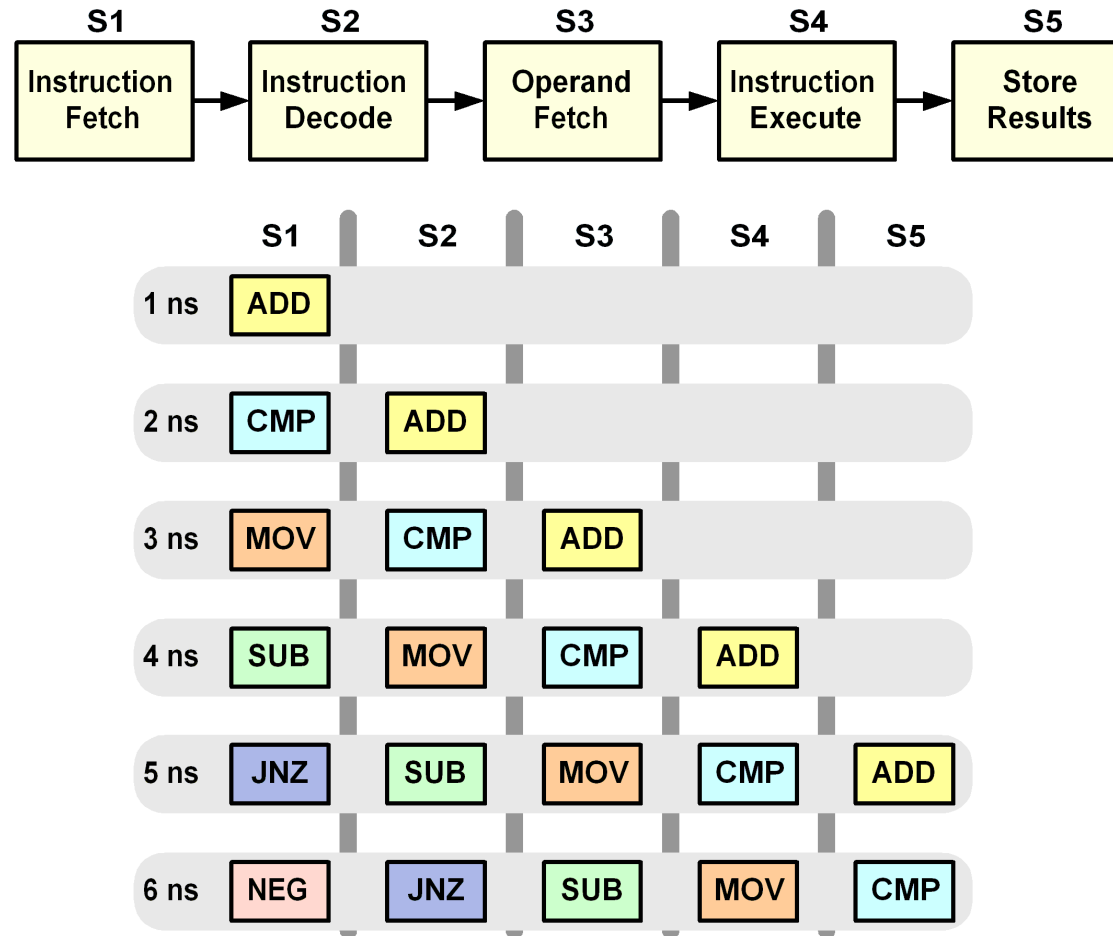
A car is produced every 5 minutes, even though it takes 20 minutes to build any one car.

Instruction-Level Parallelism

Pipelining

2.1.5

A CPU pipeline divides the work needed to execute instructions into several “pipeline stages”:



In this example, 1 billion instructions complete every second (1ns/instruction) - this is the CPU's bandwidth. It takes 5ns for any one instruction to execute (to go all the way through the pipeline) - this is the pipeline's latency.

Instruction-Level Parallelism

Pipelining

2.1.5

Pipeline latency is the sum of the time required for each pipeline stage.

CPU bandwidth is calculated as:

(1 second / cycle time) → gives the number of cycles per second

cycles / sec X instructions per cycle → gives the no. of instructions per second

MIPS (millions of instructions per second) is a common unit for CPU throughput. To get MIPS, just divide the number of instructions per second by 1,000,000.

For example, the pipeline on the previous page had a bandwidth of 1,000,000,000 (a billion) instructions per second, which is the same as $1,000,000,000 / 1,000,000 = \underline{1000 \text{ MIPS}}$

Note

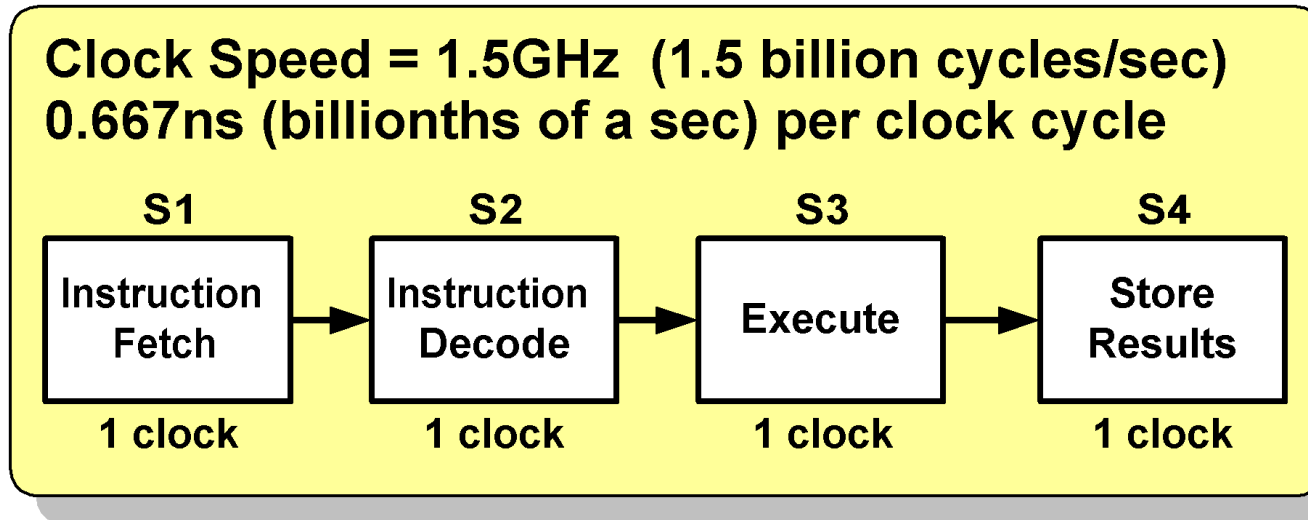
In order for a simple pipeline (as shown above) to be effective, all of the pipeline stages must be able to complete in about the same amount of time.

Any pipeline stage which takes longer than the others is a bottleneck.

Exercise 4 - Pipeline

2.1.5

What is the latency and the bandwidth of the following pipeline?



Latency:

- A** – 0.667 ns
- B** – 1.5 ns
- C** – 2.667 ns
- D** – 4 ns

Bandwidth:

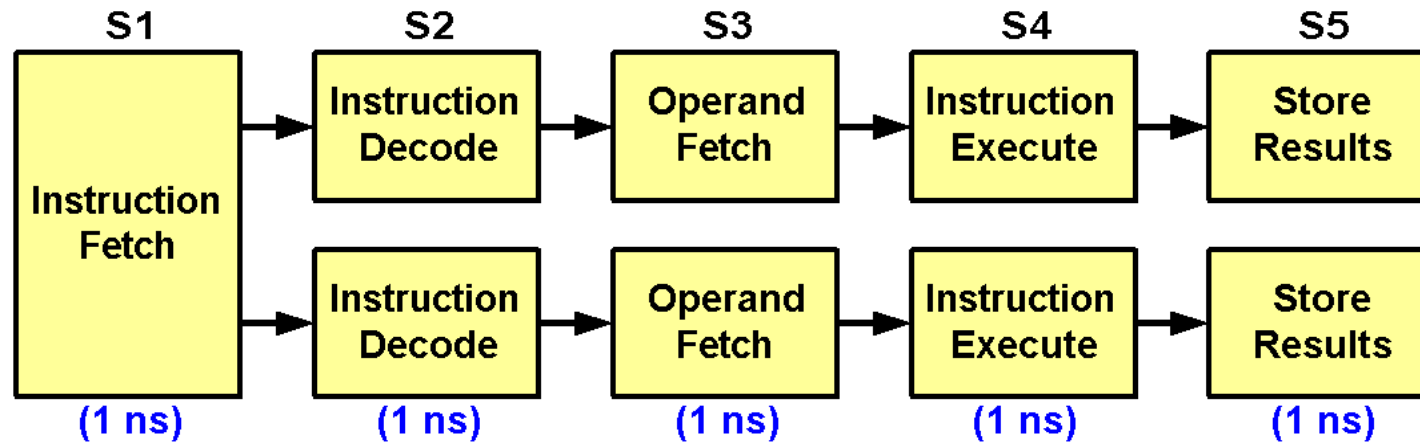
- A** – 1,500 Instructions / sec
- B** – 1,500,000,000 instruction / sec
- C** – 667,000 instructions / sec
- D** – 4,000,000 instructions / sec

Instruction-Level Parallelism

Superscalar Architectures

2.1.5

Superscalar architectures have multiple execution units and so they have the ability to execute more than one instruction with every cycle:



In this example with a 1ns cycle time:

- the pipeline has a 2000 MIPS bandwidth (two instructions can enter the pipeline each clock cycle, so 2000 Million instructions can enter the pipeline each second), and
- a 5ns latency (it takes 5ns for each instruction to go through the pipeline).

Note that in this dual-pipeline example, the Instruction Fetch unit must be able to fetch two instructions per cycle in order to keep the other pipeline stages busy.

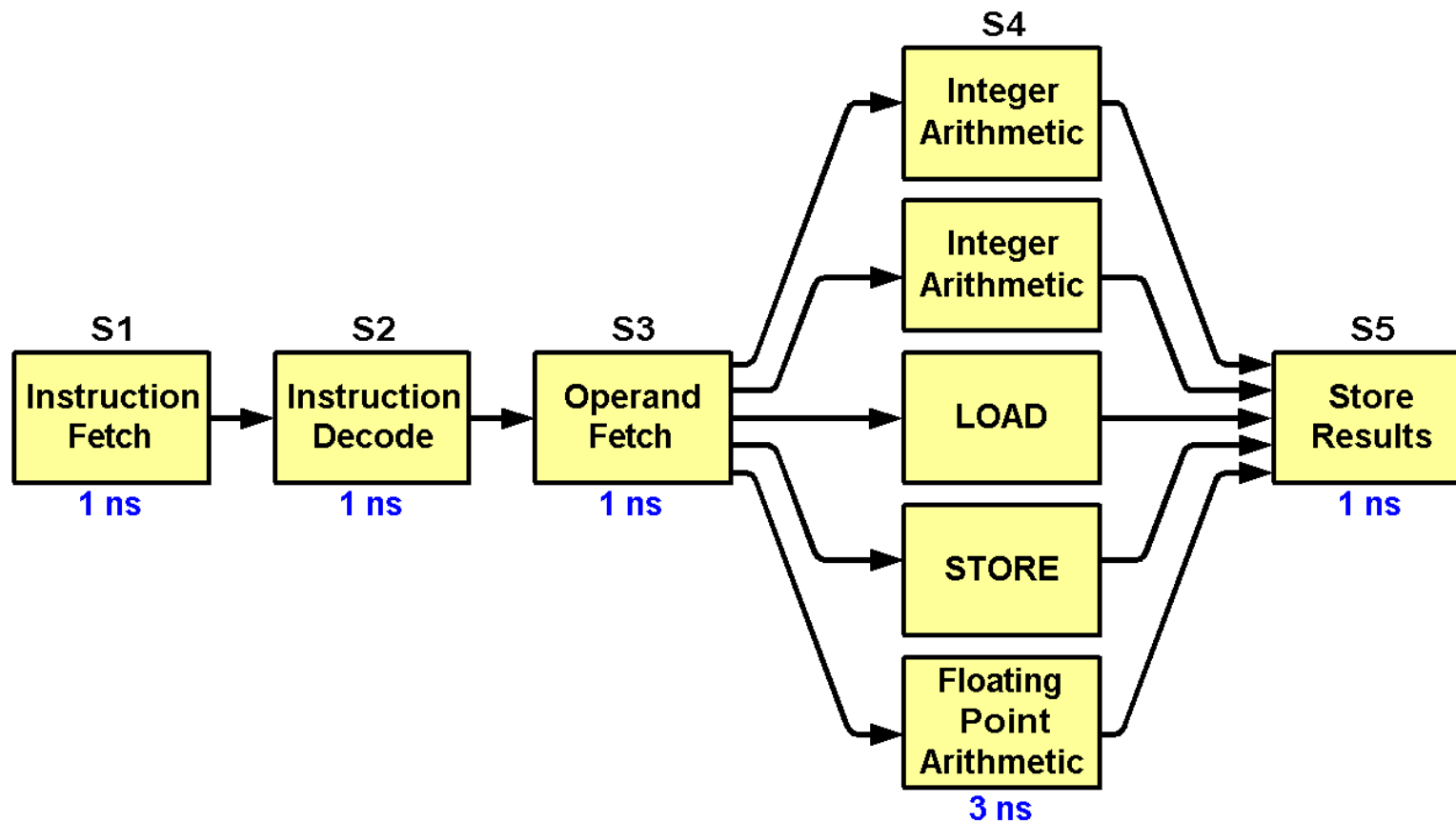
Parallel pipelines such as this don't usually operate at full bandwidth because of dependencies between instructions – we'll learn more about this in a latter lesson.

Instruction-Level Parallelism

Superscalar Architectures

2.1.5

Not much extra performance is gained by going beyond two complete pipelines. To add more parallelism, a common pipeline usually includes multiple execution units:



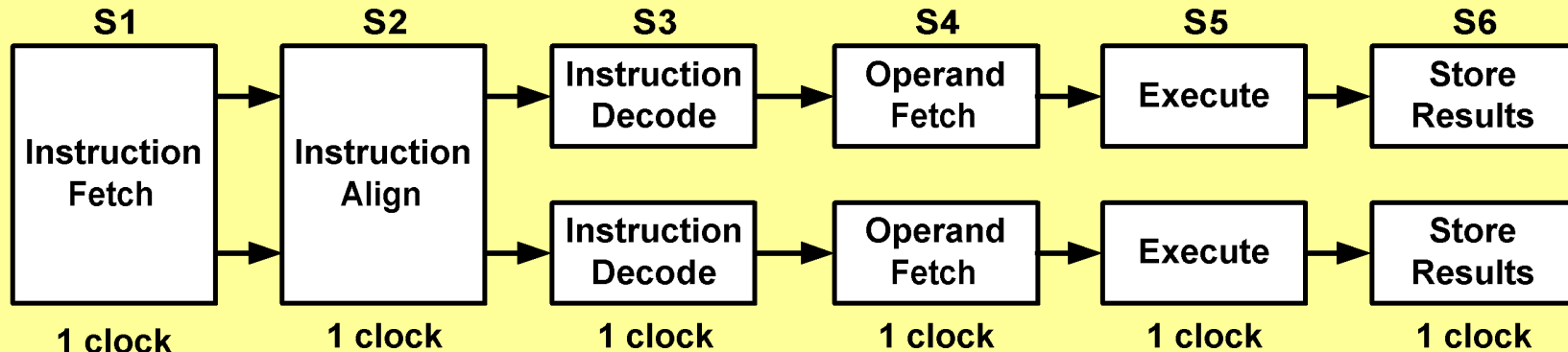
The execution (S4) pipeline stages take longer to execute than the other stages. As instructions are processed through the pipeline they're directed into the appropriate execution unit. This pipeline has a latency of 7ns (1+1+1+3+1), and a bandwidth of 1000 MIPS.

Exercise 5 - Superscalar Pipeline

2.1.5

What is the latency and the bandwidth of the following pipeline?

Clock Speed = 500MHz (500 million cycles per second)
2ns (billionths of a second) per clock cycle



Latency:

- A** – 2 ns
- B** – 6 ns
- C** – 10 ns
- D** – 12 ns

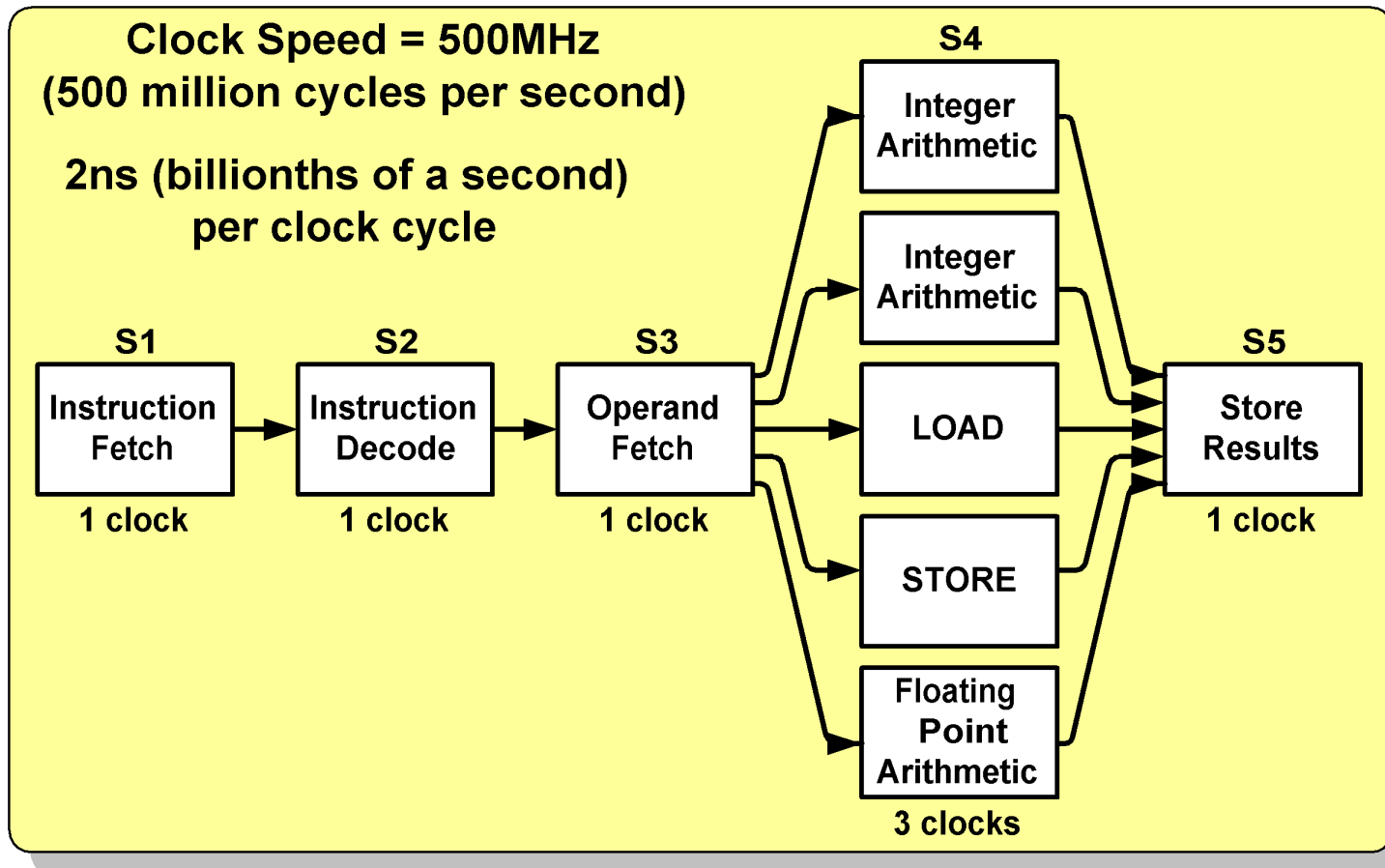
Bandwidth:

- A** – 500 MIPS
- B** – 1000 MIPS
- C** – 2000 MIPS
- D** – 5000 MIPS

Exercise 6 - Superscalar Pipeline

2.1.5

What is the latency and the bandwidth of the following pipeline?



Latency:

A – 5 ns

B – 10 ns

C – 14 ns

D – 18 ns

Bandwidth:

A – 500 MIPS

B – 1000 MIPS

C – 2000 MIPS

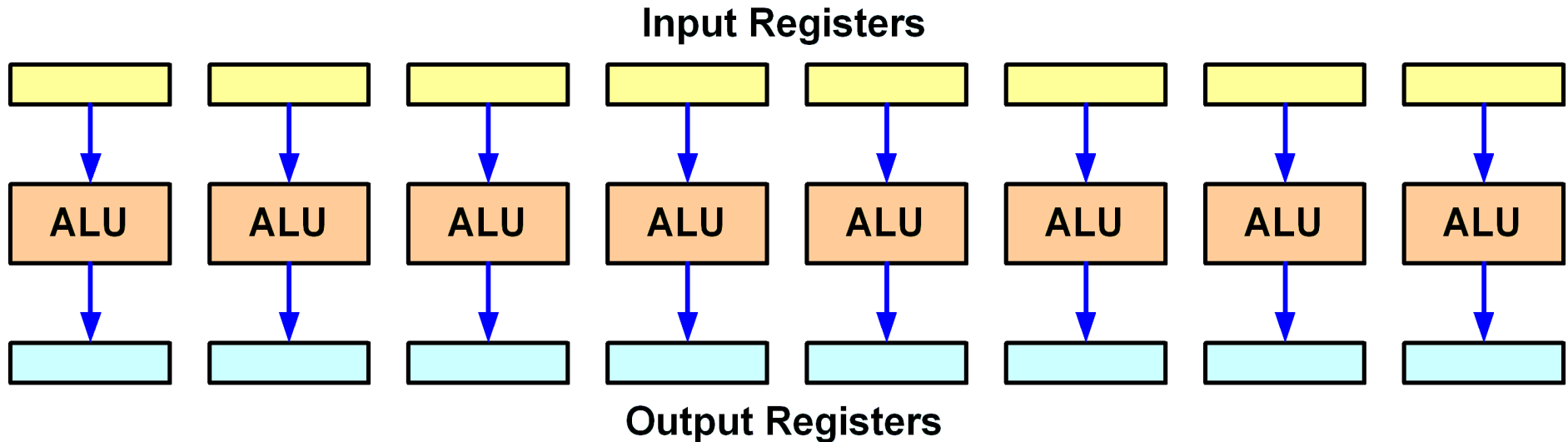
D – 5000 MIPS

Processor-Level Parallelism

Vector Processors

2.1.6

Vector processors use separate arithmetic units (or a single heavily pipelined unit) to perform operations in parallel on a more limited set of data.



This is different than a multistage pipeline, because in a vector processor all of the ALUs are performing the same instruction on different data values at the same time.

Many modern CPU chips include vector processors (ie, Pentium's "MMX" and AMD's "3DNow" instruction sets).

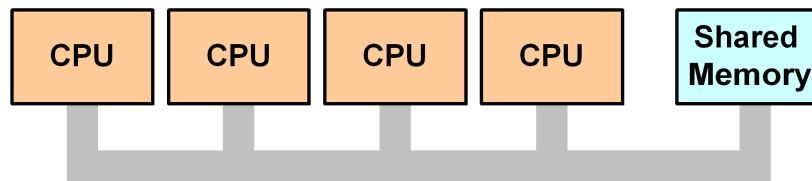
Vector or array instructions are very efficient at computing arrays of data in applications such as image processing. But much more programming skill is needed to take advantage of them, and fewer language tools are available.

Processor-Level Parallelism

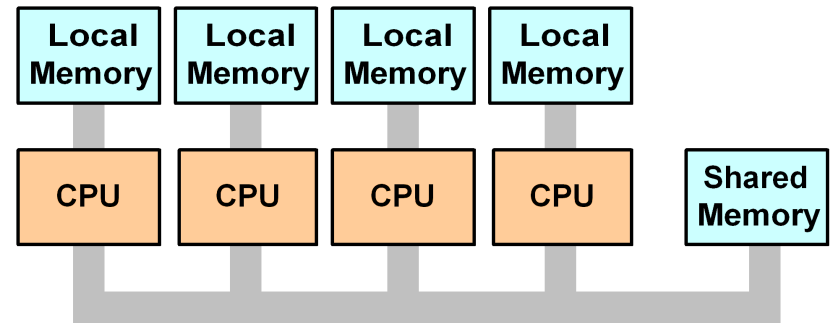
Multiprocessors

2.1.6

Multiprocessors are built using more than one conventional CPU. Various architectures differ primarily around how memory is connected and shared to the various CPUs.



The simplest design adds additional CPUs to a bus with a single shared memory. This architecture is known as “**Symmetrical MultiProcessing**” (**SMP**). The problem with this design is that the single memory becomes a bottleneck, so this design doesn’t scale well to more than a handful of CPUs.



A way to solve the memory bottleneck is to provide each CPU with its own local memory. This requires a smarter program and also creates a “partitioning” problem (If CPU “A” needs more memory, it can’t use CPU “B”’s memory even CPU “B” doesn’t need it).

Most general-purpose multiprocessor systems and operating systems use the **SMP** architecture.

Key Concepts

- **Computer system levels – high vs. low level, the concept of treating a lower level as a “black box”, Implementation vs. Interface**
- **How programs written at one level are translated or interpreted for execution at a lower level.**
- **The Von Neumann architecture.**
- **The three sample architectures: Pentium-4, UltraSPARC III, and 8051**
- **The three major components of a computer system**
- **How a pipeline works, and pipeline measurements**
- **Instruction vs. Processor level parallelism**

What's Next

- **Look at Review Questions for this week**
- **Sign on to WebCT and do Module 1 – “Measurements and Numbers”**
- **Pre-read week 2 material**