CS6303 – COMPUTER ARCHITECTURE

LESSION NOTES

UNIT I

OVERVIEW & INSTRUCTIONS

Embedded Computer: Performs single function on a microprocessor

- Embedded within a product (e.g. microwave, car, cell phone)
- Objective: Low cost
- Increasingly written in a hardware description language, like Verilog or VHDL
- Processor core allows application-specific hardware to be fabricated on a single chip.

Desktop Computer: Designed for individual use

- Also called personal computer, workstation

Server: Runs large, specialized program(s)

- Shared by many users: more memory, higher speed, better reliability
- Accessed via a network using a request-response (client-server) interface
- Example: File server, Database server, Web server

Supercomputer: Massive computing resources and memory

- Hundreds to thousands of processors within single computer
- Terabytes of memory
- Program uses multiple processors simultaneously
- Rare due to extreme expense
- Applications: Weather forecasting, military simulations, etc.

What types of applications are concerned about:

- Memory?
- Processing speed?
- Usability?
- Maintainability?

How can the following impact performance?

- A selected algorithm?
- A programming language?
- A compiler?
- An operating system?
- A processor?
- I/O system/devices?

Computer Architect must balance speed and cost across the system

- System is measured against specification
- Benchmark programs measure performance of systems/subsystems
- Subsystems are designed to be in balance between each other

Usage:

- Normal: Data communications, time, clock frequencies
- Power of 2: Memory (often)

Memory units:

- **Bit** (b): 1 binary digit
- **Nibble:** 4 binary digits
- **Byte** (B): 8 binary digits
- **Word**: Commonly 32 binary digits (but may be 64).
- **Half Word**: Half the binary digits of a word
- **Double Word**: Double the binary digits of a word

Common Use:

- 10 Mbps = 10 Mb/s = 10 Megabits per second
- 10 MB = 10 Megabytes
- 10 MIPS = 10 Million Instructions Per Second

Moore's Law:

- Component density increase per year: 1.6

- Processor performance increase: 1.5 more recently 1.2 and < 1.2
- Memory capacity improvement: 4/3: 1.33

Tradeoffs in Power versus Clock Rate

- Faster Clock Rate = Faster processing = More power
- More transistors = More complexity = More power

Example Problems:

A disk operates at 7200 Revolutions per minute (RPM). How long does it take to revolve once?

7200 Revs = 1 Rev

60 seconds x secs

7200/60 x = 1

120x = 1

x = 1/120 = 0.00833 second = 8.33milliseconds or 8.33 ms

A disk holds 600 GB. How many bytes does it hold?

600 GB = 600 x 2^{30} = 600 x 1,073,741,824 = 644,245,094,400

A LAN operates at 10 Mbps. How long will it take to transfer a packet of 1000 bytes? (Optimistically assuming 100% efficiency)

10 Mb = 8 bits	10 Mb = 8000
1 sec x sec	1 sec x sec
10,000,000x = 8	10,000,000x = 8000
x = 8/10,000,000 = 0.000,000,8 = 800ns	x = 8000/10,000,000=8/10,000
1000 x 800 ns = 800us	x = 0.0008 = 800us

8 GREAT IDEAS

- **Design for *Moore's Law***

one constant for computer designers is rapid change, which is driven largely by Moore's Law. It states that integrated circuit resources double every 18–24 months. Moore's Law resulted from a 1965 prediction of such growth in IC capacity made by Gordon Moore, one of the founders of Intel. As computer designs can take years, the resources available per chip can easily double or quadruple between the start and finish of the project. Like a skeet shooter, computer architects must anticipate where the technology will be when the design finishes rather than design for where it starts. We use an "up and to the right" Moore's Law graph to represent designing for rapid change.

- **Use *abstraction* to simplify design**
  . Both computer architects and programmers had to invent techniques to make themselves more productive, for otherwise design time would lengthen as dramatically as resources grew by Moore's Law. A major productivity technique for hardware and soft ware is to use abstractions to represent the design at different levels of representation; lower-level details are hidden to off er a simpler model at higher levels. We'll use the abstract painting icon to represent this second great idea

- **Make the *common case fast***
  . Making the common case fast will tend to enhance performance better than optimizing the rare case. Ironically, the common case is oft en simpler than the rare case and hence is oft en easier to enhance. This common sense advice implies that you know what the common case is, which is only possible with careful experimentation and measurement. We use a sports car as the icon for making the common case fast, as the most common trip has one or two passengers, and it's surely easier to make a fast sports car than a fast minivan

- **Performance *via parallelism***
  Since the dawn of computing, computer architects have offered designs that get more performance by performing operations in parallel. We'll see many examples of

parallelism in this book. We use multiple jet engines of a plane as our icon for parallel performance.

- **Performance *via pipelining***
  Following the saying that it can be better to ask for forgiveness than to ask for permission, the next great idea is prediction. In some cases it can be faster on average to guess and start working rather than wait until you know for sure, assuming that the mechanism to recover from a misprediction is not too expensive and your prediction is relatively accurate. We use the fortune-teller's crystal ball as our prediction icon.

- **Performance *via prediction***

  A particular pattern of parallelism is so prevalent in computer architecture that it merits its own name: pipelining. For example, before fire engines, a "bucket brigade" would respond to a fire, which many cowboy movies show in response to a dastardly act by the villain. Th e townsfolk form a human chain to carry a water source to fi re, as they could much more quickly move buckets up the chain instead of individuals running back and forth. Our pipeline icon is a sequence of pipes, with each section representing one stage of the pipeline**.**

- ***Hierarchy* of memories**
  Programmers want memory to be fast, large, and cheap, as memory speed often shapes performance, capacity limits the size of problems that can be solved, and the cost of memory today is often the majority of computer cost. Architects have found that they can address these conflicting demands with a hierarchy of memories, with the fastest, smallest, and most expensive memory per bit at the top of the hierarchy and the slowest, largest, and cheapest per bit at the bottom. Caches give the programmer the illusion that main memory is nearly as fast as the top of the hierarchy and nearly as big and cheap as

the bottom of the hierarchy. We use a layered triangle icon to represent the memory hierarchy. The shape indicates speed, cost, and size: the closer to the top, the faster and more expensive per bit the memory; the wider the base of the layer, the bigger the memory.

- ***Dependability via* redundancy**
  Computers not only need to be fast; they need to be dependable. Since any physical device can fail, we make systems dependable by including redundant components that can take over when a failure occurs and to help detect failures. We use the tractor-trailer as our icon, since the dual tires on each side of its rear axels allow the truck to continue driving even when one tire fails. (Presumably, the truck driver heads immediately to a repair facility so the fl at tire can be fixed, thereby restoring redundancy!)

<u>COMPONENTS OF A COMPUTER SYSTEM</u>

Computer components include:

Input: keyboard, mouse, network, disk
Output: printer, video screen, network, disk
Memory: DRAM, magnetic disk
CPU: Intelligence: Includes Datapath and Control

Input/Output

Mouse:

Electromechanical: Rolling ball indicates change in position as (x,y) coordinates.
Optical: Camera samples 1500 times per second. Optical processor compares images and determines distance moved.

Displays:

Raster Refresh Buffer: Holds the bitmap or matrix of pixel values.

Matrix of Pixels: low resolution: 512 x 340 pixels to high resolution: 2560 x 1600 pixels
- Black & White: 1 bit per pixel
- Grayscale: 8 bits per pixel
- Color: (one method): 8 bits each for red, blue, green = 24 bits

Required: Refresh the screen periodically to avoid flickering

Two types of Displays:

Cathode Ray Tube (CRT): Pixel is source of light

Scans one line at a time with a refresh rate of 30-75 times per second
Liquid Crystal Display (LCD): LCD pixel control or bends the light for the display.

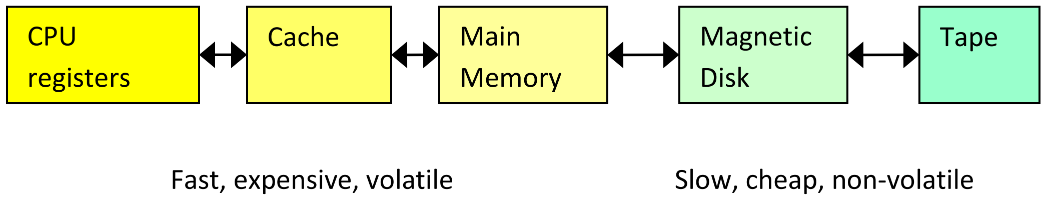
Color active matrix LCD: Three transistor switches per pixel

Networking: Communications between computers

Local Area Network (LAN): A network which spans a small area: within a building

Wide Area Network (WAN): A network which extends hundreds of miles; typically managed by a communications service provider

Memory Hierarchy:



Secondary Memory: Nonvolatile memory used to store programs and data when not running

Nonvolatile: Does not lose data when powered off
Includes:
- Magnetic Disk: Access time: 5-15 ms
- Tape: Sometimes used for backup
- Optical Disk: CD or DVD
- FLASH: Removable memory cards attach via USB
- Floppy and Zip: Removable form of magnetic disk

Magnetic Disk: Movable arm moves to concentric circle then writes

Disk diameter: 1 to 3.5 inches
Latency: Moving head to 'cylinder' or concentric track
Rotation Time: Rotating cylinder to correct location on 'track'
Transfer Time: Reading or writing to disk on 'track'
Access time: 5-20 ms

Optical Disk: Laser uses spiral pattern to write bits as pits or flats.

Compact Disc (CD): Stores music
Digital Versatile Disc (DVD): Multi-gigabyte capacity required for films
Read-write procedure similar to Magnetic Disk (but optical write, not magnetic)

Flash Memory: Semiconductor memory is nonvolatile

- More expensive than disk, but also more rugged and faster latency.
- Good for 100,000-1M writes.
- Common in cameras, portable music players, memory sticks

Primary or Main Memory: Programs are retained while they are running. Uses:

Dynamic Random Access Memory (**DRAM**): Built as an integrated circuit, equal speed to any location in memory Access time: 50-70 ns.
SIMM (Single In-line Memory Module): DRAM memory chips lined up in a row, often on a daughter card
DIMM (Dual In-line Memory Module): Two rows of memory chips
ROM (Read Only Memory) or **EPROM** (Erasable Programmable ROM)

Cache: Buffer to the slower, larger main memory. Uses:

Static Random Access Memory (**SRAM)**
Faster, less dense and more expensive than DRAM
Uses multiple transistors per bit instead of the single transistor for DRAM

Registers: Fastest memory within the CPU.

Central Processing Unit (**CPU**) or **Processor**: Intelligence

Data Path: Performs arithmetic operations using registers
Control: Management of flow of information through the data path
Bus: Connects the CPU, Memory, I/O Devices

Bits are transmitted between the CPU, Memory, I/O Devices in a timeshared way
Serial buses transmit one bit at a time.
Parallel buses transmit many bits simultaneously: one bit per line

One bus system: Memory, CPU, I/O Subsystem on same bus

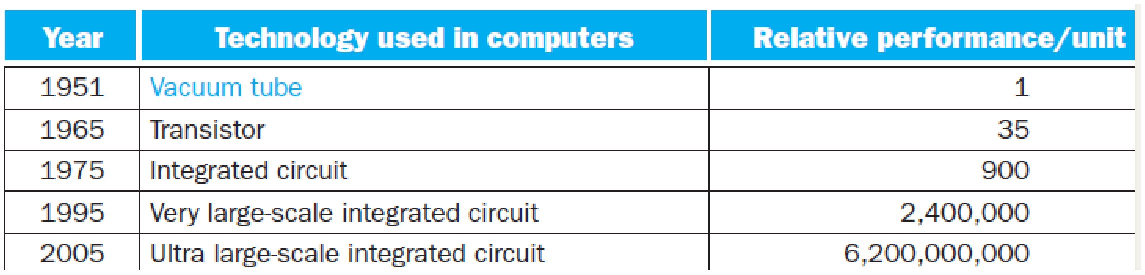
Two bus system:

One bus: CPU←→Memory
One bus: CPU←→I/O Subsystem

Example: Universal Serial Bus (USB 2.0)

Hot-pluggable: can be plugged and unplugged without damage to the system
Operates at 0.2, 1.5 or 60 MB/sec
Can interface to printer or other slow devices

TECHNOLOGY



Year	Technology used in computers	Relative performance/unit
1951	Vacuum tube	1
1965	Transistor	35
1975	Integrated circuit	900
1995	Very large-scale integrated circuit	2,400,000
2005	Ultra large-scale integrated circuit	6,200,000,000

A **transistor** is simply an on/off switch controlled by electricity. The *integrated circuit* (IC) combined dozens to hundreds of transistors into a single chip. To describe the tremendous increase in the number of transistors from hundreds to millions, the adjective *very large scale* is added to the term, creating the abbreviation *VLSI,* for **very large-scale integrated circuit**.

CLASSES OF COMPUTERS

Desktop computers

- Designed to deliver good performance to a single user at low cost usually executing 3rd party software, usually incorporating a graphics display, a keyboard, and a mouse

Servers

- Used to run larger programs for multiple, simultaneous users typically accessed only via a network and that places a greater emphasis on dependability and (often) security
  - Modern version of what used to be called mainframes, minicomputers and supercomputers
  - Large workloads
  - Built using the same technology in desktops but higher capacity
    - Gigabytes to Terabytes to Peta bytes of storage
      - Expandable
      - Scalable
      - Reliable
  - Large spectrum: from low-end (file storage, small businesses) to supercomputers (high end scientific and engineering applications) Examples: file servers, web servers, database servers

Supercomputers

- A high performance, high cost class of servers with hundreds to thousands of processors, terabytes of memory and petabytes of storage that are used for high-end scientific and engineering applications

Embedded computers (processors)

- A computer inside another device used for running one predetermined application
  - Microprocessors everywhere! (washing machines, cell phones, automobiles, video games)
  - Run one or a few applications
  - Specialized hardware integrated with the application (not your common processor)
  - Usually stringent limitations (battery power)
  - High tolerance for failure (don't want your airplane avionics to fail!)
  - Becoming ubiquitous
  - Engineered using *processor cores*

- The core allows the engineer to integrate other functions into the processor for fabrication on the same chip
- Using hardware description languages: Verilog, VHDL

Embedded Processor Characteristics

The largest class of computers spanning the widest range of applications and performance

- Often have minimum performance requirements.
- Often have stringent limitations on cost.
- Often have stringent limitations on power consumption.
- Often have low tolerance for failure.

PERFORMANCE AND POWERWALL

■Purchasing perspective

■ given a collection of machines, which has the

- best performance ?
- least cost ?
- best cost/performance?

■Design perspective

faced with design options, which has the

- best performance improvement ?
- least cost ?
- best cost/performance?

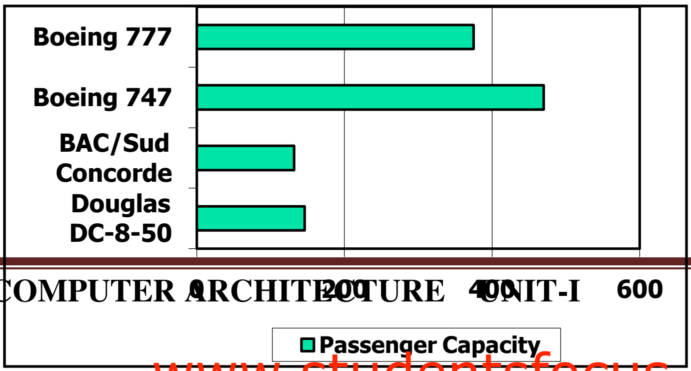
■Both require

■ basis for comparison

■ metric for evaluation

■Our goal is to understand what factors in the architecture contribute to overall system performance and the relative importance (and cost) of these factors

■Which airplane has the best performance?



Response Time and Throughput

- Response time
    - How long it takes to do a task
- Throughput
    - Total work done per unit time
        - e.g., tasks/transactions/… per hour
- How are response time and throughput affected by
    - Replacing the processor with a faster version?
    - Adding more processors?

Relative Performance

Define Performance = 1/Execution Time

"X is *n* time faster than Y"

$$\text{Performance}_X/\text{Performance}_Y = \text{Execution time}_Y/\text{Execution time}_X = n$$

- Example: time taken to run a program
    - ✓ 10s on A, 15s on B
    - ✓ Execution Time_B / Execution Time_A
      = 15s / 10s = 1.5

So A is 1.5 times faster than B

Measuring Execution Time

- Elapsed time
    - Total response time, including all aspects
        - ✓ Processing, I/O, OS overhead, idle time
    - Determines system performance
- CPU time
    - Time spent processing a given job
        - ✓ Discounts I/O time, other jobs' shares
    - Comprises user CPU time and system CPU time
    - Different programs are affected differently by CPU and system performance

CPU Clocking

- Operation of digital hardware governed by a constant-rate clock
- Clock period: duration of a clock cycle
    - e.g., 250ps = 0.25ns = 250×10^{-12}s

- Clock frequency (rate): cycles per second
  - e.g., 4.0GHz = 4000MHz = 4.0×10^9Hz

CPU Time

$$\text{CPU Time} = \text{CPU Clock Cycles} \times \text{Clock Cycle Time}$$
$$= \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}}$$

- Performance improved by
  - Reducing number of clock cycles

$$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$$
$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$
$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

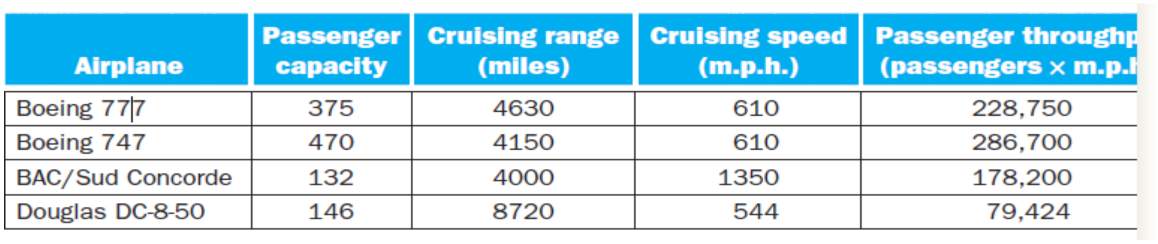
ainst cycle count

- Instruction Count for a program
  - Determined by program, ISA and compiler
- Average cycles per instruction
  - Determined by CPU hardware
  - If different instructions have different CPI
    - Average CPI affected by instruction mix

PERFORMANCE

Defining Performance

Let's suppose we define performance in terms of speed. This still leaves two possible definitions. You could define the fastest plane as the one with the highest cruising speed, taking a single passenger from one point to another in the least time. If you were interested in transporting 450 passengers from one point to another, however, the 747 would clearly be the fastest, as the last column of the figure shows. Similarly, we can define computer performance in several different ways.



Airplane	Passenger capacity	Cruising range (miles)	Cruising speed (m.p.h.)	Passenger throughp (passengers × m.p.h
Boeing 777	375	4630	610	228,750
Boeing 747	470	4150	610	286,700
BAC/Sud Concorde	132	4000	1350	178,200
Douglas DC-8-50	146	8720	544	79,424

Throughput and Response Time Do the following changes to a computer system increase throughput, decrease response time, or both?

1. Replacing the processor in a computer with a faster version

2. Adding additional processors to a system that uses multiple processors for separate tasks—for example, searching the World Wide Web

$$\text{Performance}_X = \frac{1}{\text{Execution time}_X}$$

This means that for two computers X and Y, if the performance of X is greater than the performance of Y, we have

$$\text{Performance}_X > \text{Performance}_Y$$

$$\frac{1}{\text{Execution time}_X} > \frac{1}{\text{Execution time}_Y}$$

$$\text{Execution time}_Y > \text{Execution time}_X$$

That is, the execution time on Y is longer than that on X, if X is faster than Y.

In discussing a computer design, we often want to relate the performance of two different computers quantitatively. We will use the phrase "X is *n* times faster than Y"—or equivalently "X is *n* times as fast as Y"—to mean

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = n$$

If X is *n* times faster than Y, then the execution time on Y is *n* times longer than it is on X:

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = \frac{\text{Execution time}_Y}{\text{Execution time}_X} = n$$

Example

Time taken to run a program = 10s on A, 15s on B

Relative performance =Execution Time$_B$ / Execution Time$_A$

=15s/10s

=1.5

So A is 1.5 times faster than B

Measuring Execution Time

Elapsed time

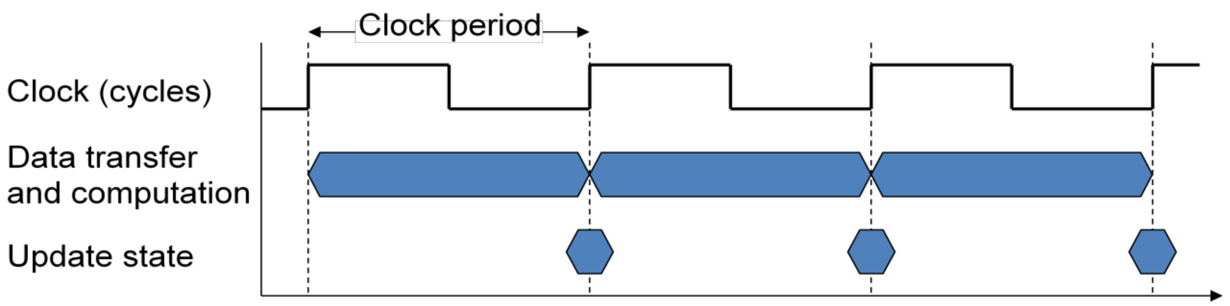
- Total response time, including all aspects Processing, I/O, OS overhead, idle time
- Determines system performance

CPU time

- Time spent processing a given job Discounts I/O time, other jobs' shares
- Comprises user CPU time and system CPUtime
- Different programs are affected differently byCPU and system performance

CPU Clocking

Operation of digital hardware governed by a constant-rate clock



- ❑ Clock period: duration of a clock cycle
    - l e.g., 250ps = 0.25ns = 250×10^{-12}s
- ❑ Clock frequency (rate): cycles per second
    - l e.g., 4.0GHz = 4000MHz = 4.0×10^{9}Hz

CPU TIME

- ❑ Performance improved by
    - Reducing number of clock cycles
    - Increasing clock rate
    - Hardware designer must often trade off clock rate against cycle count

$$\text{CPU Time} = \text{CPU Clock Cycles} \times \text{Clock Cycle Time}$$
$$= \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}}$$

Example

- ❑ Computer A: 2GHz clock, 10s CPU time
- ❑ Designing Computer B
  - l Aim for 6s CPU time
  - l Can do faster clock, but causes 1.2 × clock cycles
- ❑ How fast must Computer B clock be?

$$\text{Clock Rate}_B = \frac{\text{Clock Cycles}_B}{\text{CPU Time}_B} = \frac{1.2 \times \text{Clock Cycles}_A}{6s}$$
$$\text{Clock Cycles}_A = \text{CPU Time}_A \times \text{Clock Rate}_A$$
$$= 10s \times 2\text{GHz} = 20 \times 10^9$$
$$\text{Clock Rate}_B = \frac{1.2 \times 20 \times 10^9}{6s} = \frac{24 \times 10^9}{6s} = 4\text{GHz}$$

Instruction Count and CPI

- ❑ Instruction Count for a program
  - l Determined by program, ISA and compiler
- ❑ Average cycles per instruction
  - l Determined by CPU hardware
  - l If different instructions have different CPI
    - - Average CPI affected by instruction mix

$$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$$
$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$
$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

CPI Example

- ❑ Computer A: Cycle Time = 250ps, CPI = 2.0
- ❑ Computer B: Cycle Time = 500ps, CPI = 1.2
- ❑ Same ISA
- ❑ Which is faster, and by how much?

$$\text{CPU Time}_A = \text{Instruction Count} \times \text{CPI}_A \times \text{Cycle Time}_A$$
$$= I \times 2.0 \times 250ps = I \times 500ps$$

A is faster…

$$\text{CPU Time}_B = \text{Instruction Count} \times \text{CPI}_B \times \text{Cycle Time}_B$$
$$= I \times 1.2 \times 500ps = I \times 600ps$$

$$\frac{\text{CPU Time}_B}{\text{CPU Time}_A} = \frac{I \times 600ps}{I \times 500ps} = 1.2$$

…by this much

CPI in More Detail

- ❑ If different instruction classes take different numbers of cycles

$$\text{Clock Cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{Instruction Count}_i)$$

- ❑ Weighted average CPI

$$\text{CPI} = \frac{\text{Clock Cycles}}{\text{Instruction Count}} = \sum_{i=1}^{n} \left(\text{CPI}_i \times \frac{\text{Instruction Count}_i}{\text{Instruction Count}} \right)$$

CPI Example

- ❑ Alternative compiled code sequences using instructions in classes A, B, C

Class	A	B	C
CPI for class	1	2	3
IC in sequence 1	2	1	2
IC in sequence 2	4	1	1

- ❑ Sequence 1: IC = 5
  - l Clock Cycles
    = 2×1 + 1×2 + 2×3
    = 10
  - l Avg. CPI = 10/5 = 2.0
- ❑ Sequence 2: IC = 6
  - l Clock Cycles
    = 4×1 + 1×2 + 1×3
    = 9
  - l Avg. CPI = 9/6 = 1.5

Performance depends on

- l Algorithm: affects IC, possibly CPI

- Programming language: affects IC, CPI
- Compiler: affects IC, CPI
- Instruction set architecture: affects IC, CPI, T_c

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

computers are constructed using a clock that determines when events take place in the hardware. These discrete time intervals are called **clock cycles** (or **ticks**, **clock ticks**, **clock periods**, **clocks**, **cycles**). Designers refer to the length of a **clock period** both as the time for a complete *clock cycle* (e.g., 250 picoseconds, or 250 ps) and as the *clock rate* (e.g., 4 gigahertz, or 4 GHz), which is the inverse of the clock period. In the next subsection, we will formalize the relationship between the clock cyclesof the hardware designer and the seconds of the computer user.

Computer Performance and its Factors

$$\begin{matrix}\text{CPU execution time}\\ \text{for a program}\end{matrix} = \begin{matrix}\text{CPU clock cycles}\\ \text{for a program}\end{matrix} \times \text{Clock cycle time}$$

Alternatively, because clock rate and clock cycle time are inverses,

$$\begin{matrix}\text{CPU execution time}\\ \text{for a program}\end{matrix} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}}$$

Instruction Performance

$$\text{CPU clock cycles} = \text{Instructions for a program} \times \begin{matrix}\text{Average clock cycles}\\ \text{per instruction}\end{matrix}$$

The term **clock cycles per instruction**, which is the average number of clock cycles each instruction takes to execute, is often abbreviated as **CPI**

Classic cpu performance

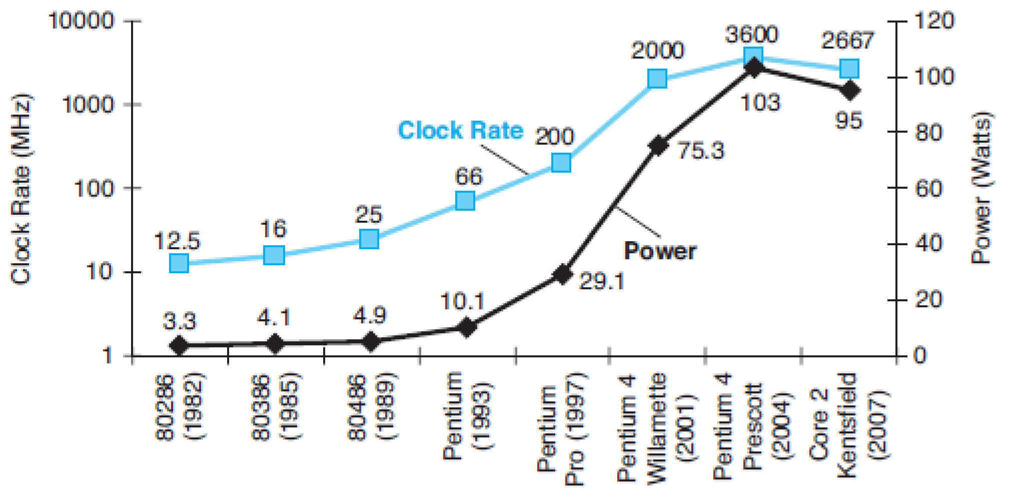
$$\text{CPU time} = \text{Instruction count} \times \text{CPI} \times \text{Clock cycle time}$$

or, since the clock rate is the inverse of clock cycle time:

$$\text{CPU time} = \frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}}$$

Components of performance	Units of measure
CPU execution time for a program	Seconds for the program
Instruction count	Instructions executed for the program
Clock cycles per instruction (CPI)	Average number of clock cycles per instruction
Clock cycle time	Seconds per clock cycle

POWERWALL



The dominant technology for integrated circuits is called CMOS (complementary metal oxide semiconductor). For CMOS, the primary source of power dissipation is so-called dynamic power—that is, power that is consumed during switching. The dynamic power dissipation depends on the capacitive loading of each transistor, the voltage applied, and the frequency that the transistor is switched:

$$\text{Power} = \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency switched}$$

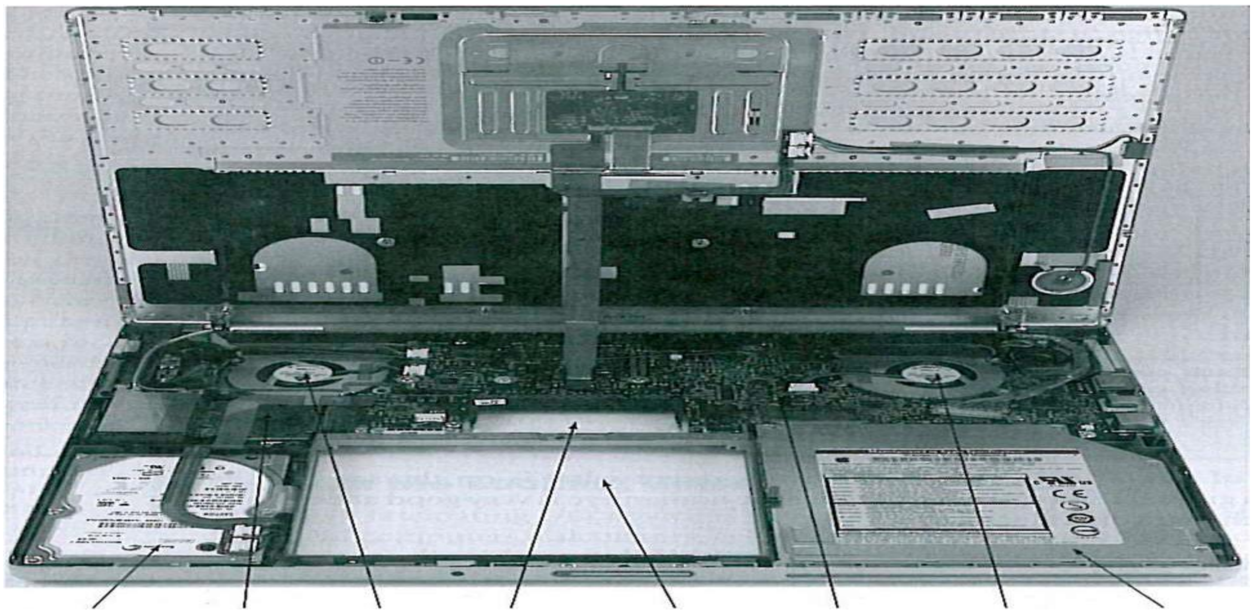
UNIPROCESSORS TO MULTIPROCESSORS

If we open the box containing the computer, we see a fascinating board of thin plastic, covered with dozens of small gray or black rectangles.

The motherboard is shown in the upper part of the photo. Two disk drives are in front—the hard drive on the left and a DVD drive on the right. The hole in the middle is for the laptop battery. The small rectangles on the motherboard contain the devices that drive our advancing technology, called integrated circuits and nicknamed chips.

The board is composed of three pieces: the piece connecting to the I/O devices mentioned earlier, the memory, and the processor.

The memory is where the programs are kept when they are running; it also contains the data needed by the running programs. Figure 1.8 shows that memory is found on the two small boards, and each small memory board contains eight integrated circuits



The *processor* is the active part of the board, following the instructions of a program

to the letter. It adds numbers, tests numbers, signals I/O devices to activate, and so on.. Occasionally, people call the processor the CPU, for the more bureaucratic-sounding central processor unit.

Descending even lower into the hardware, The processor logically comprises two main components: datapath and control, the respective brawn and brain of the processor.

The datapath performs the arithmetic operations, and control tells the datapath, memory, and I/O devices what to do according to the wishes of the instructions of the program. This explains the datapath and control for a higher-performance design Descending into the depths of any component of the hardware reveals insights into the computer. Inside the processor is another type of memory—cache memory.

Cache memory consists of a small, fast memory that acts as a buffer for the DRAM memory. (The nontechnical definition of *cache* is a safe place for hiding things.)

Cache is built using a different memory technology, **static random access memory (SRAM).** SRAM is faster but less dense, and hence more expensive, than DRAM You may have noticed a common theme in both the software and the hardware descriptions: delving into the depths of hardware or software reveals more information or, conversely, lower-level details are hidden to offer a simpler model at higher levels. The use of such layers, or **abstractions,** is a principal technique for designing very sophisticated computer systems.

The power limit has forced a dramatic change in the design of microprocessors. Figure shows the improvement in response time of programs for desktop microprocessors over time. Since 2002, the rate has slowed from a factor of 1.5 per year to less than a factor of 1.2 per year.

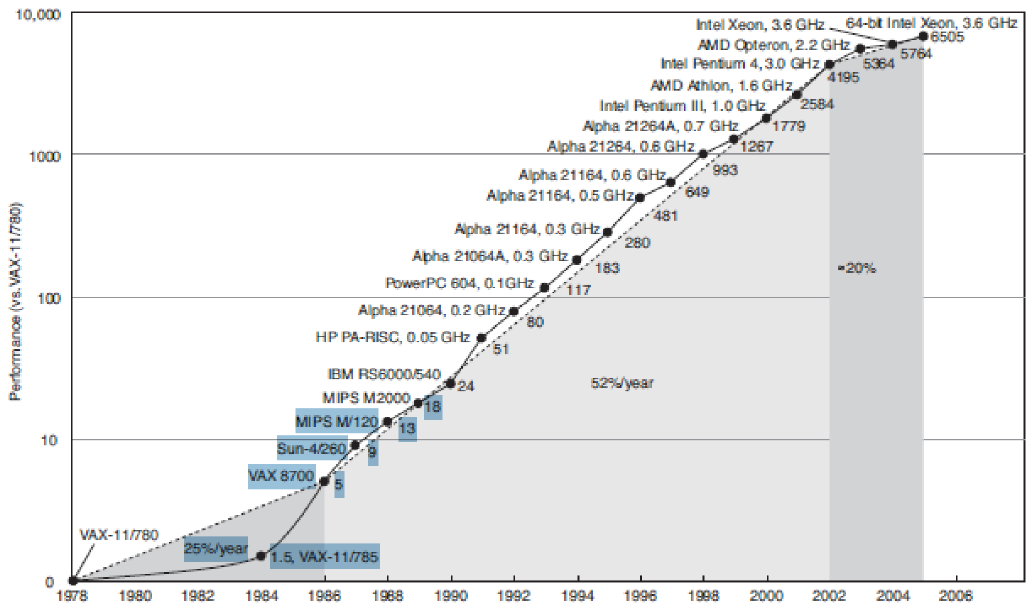


FIGURE 1.16 Growth in processor performance since the mid-1980s. This chart plots performance relative to the VAX 11/780 as measured by the SPECint benchmarks (see Section 1.8). Prior to the mid-1980s, processor performance growth was largely technology-driven and averaged about 25% per year. The increase in growth to about 52% since then is attributable to more advanced architectural and organizational ideas. By 2002, this growth led to a difference in performance of about a factor of seven. Performance for floating-point-oriented calculations has increased even faster. Since 2002, the limits of power, available instruction-level parallelism, and long memory latency have slowed uniprocessor performance recently, to about 20% per year.

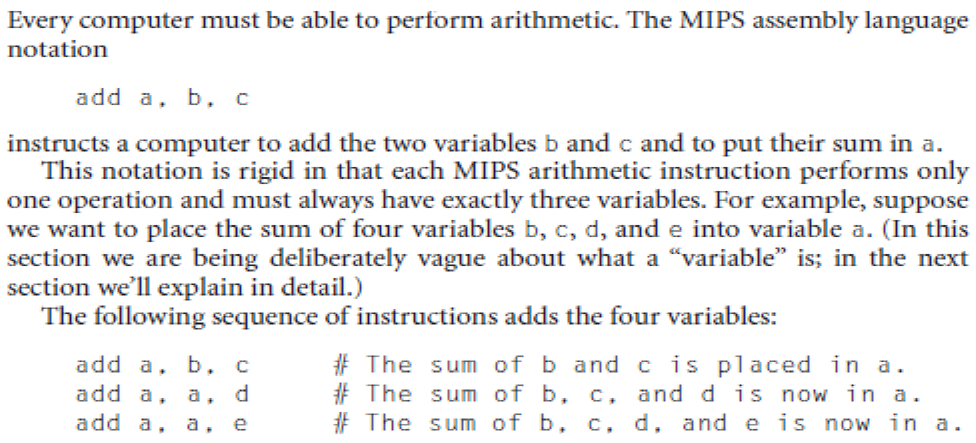
Product	AMD Opteron X4 (Barcelona)	Intel Nehalem	IBM Power 6	Sun Ultra SPARC T2 (Niagara 2)
Cores per chip	4	4	2	8
Clock rate	2.5 GHz	~ 2.5 GHz ?	4.7 GHz	1.4 GHz
Microprocessor power	120 W	~ 100 W ?	~ 100 W ?	94 W

As an analogy, suppose the task was to write a newspaper story. Eight reporters working on the same story could potentially write a story eight times faster. To achieve this increased speed, one would need to break up the task so that each reporter had something to do at the same time. Thus, we must *schedule* the subtasks. If anything went wrong and just one reporter took longer than the seven

others did, then the benefits of having eight writers would be diminished. Thus, we must *balance the load* evenly to get the desired speedup. Another danger would be if reporters had to spend a lot of time talking to each other to write their sections. You would also fall short if one part of the story, such as the conclusion, couldn't be written until all of the other parts were completed. Thus, care must be taken to *reduce communication and synchronization overhead*. For both this analogy and parallel programming, the challenges include scheduling, load balancing, time for synchronization, and overhead for communication between the parties. As you might guess, the challenge is stiffer with more reporters for a newspaper story and more processors for parallel programming

INSTRUCTIONS – OPERATIONS AND OPERANDS – REPRESENTING INSTRUCTIONS- LOGICAL OPERATIONS – CONTROL OPERATIONS

operations of computer hardware



Every computer must be able to perform arithmetic. The MIPS assembly language notation

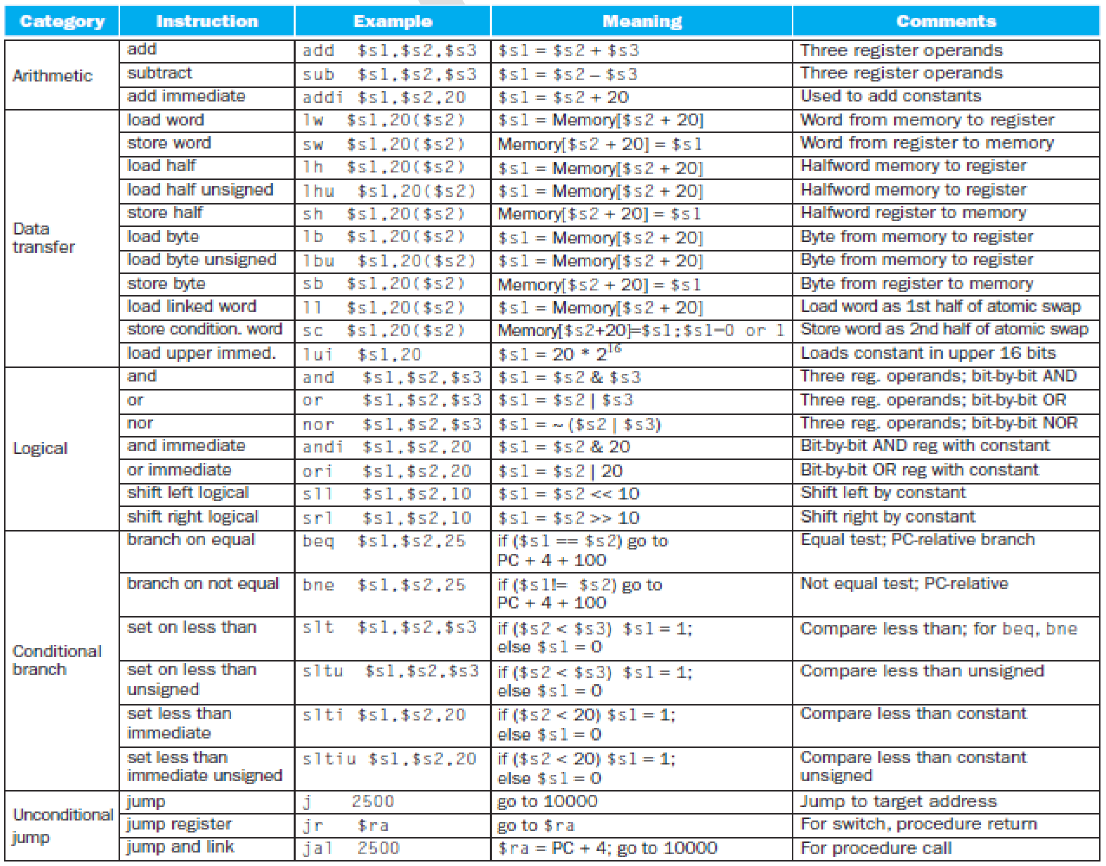
```
add a, b, c
```

instructs a computer to add the two variables b and c and to put their sum in a.

This notation is rigid in that each MIPS arithmetic instruction performs only one operation and must always have exactly three variables. For example, suppose we want to place the sum of four variables b, c, d, and e into variable a. (In this section we are being deliberately vague about what a "variable" is; in the next section we'll explain in detail.)

The following sequence of instructions adds the four variables:

```
add a, b, c    # The sum of b and c is placed in a.
add a, a, d    # The sum of b, c, and d is now in a.
add a, a, e    # The sum of b, c, d, and e is now in a.
```



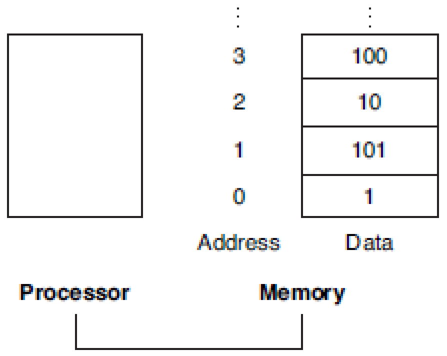
Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	`lh $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	`lb $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	`beq $s1,$s2,25`	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if ($s1!= $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne
	set on less than unsigned	`sltu $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to $ra	For switch, procedure return
	jump and link	`jal 2500`	$ra = PC + 4; go to 10000	For procedure call

operands of computer hardware

A very large number of registers may increase the clock cycle time simply because it takes electronic signals longer when they must travel farther. Guidelines such as “smaller is faster” are not absolutes; 31 registers may not be faster than 32. Yet, the truth behind such observations causes computer designers to take them seriously. In this case, the designer must balance the craving of programs for more registers with the designer’s desire to keep the clock cycle fast.Another reason for not using more than 32 is the number of bits it would take in the instruction forma

Memory Operands

As explained above, arithmetic operations occur only on registers in MIPS instructions; thus, MIPS must include instructions that transfer data between memory and registers. Such instructions are called **data transfer instructions**. To access a word in memory, the instruction must supply the memory **address**. Memory is just a large, single-dimensional array, with the address acting as the index to that array, starting at 0. For example, in Figure the address of the third data element is 2, and the value of Memory[2] is 10.



Given the importance of registers, what is the rate of increase in the number of registers in a chip over time?

1. Very fast: They increase as fast as Moore’s law, which predicts doubling the number of transistors on a chip every 18 months.

2. Very slow: Since programs are usually distributed in the language of the computer, there is inertia in instruction set architecture, and so the number of registers increases only as fast as new instruction sets become viable.

representing instruction

The decimal representation is

0	17	18	8	0	32

000000	10001	10010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

This layout of the instruction is called the instruction format. As you can see from counting the number of bits, this MIPS instruction takes exactly 32 bits—the same size as a data word. In keeping with our design principle that simplicity favors regularity, all MIPS instructions are 32 bits long.

- *op:* Basic operation of the instruction, traditionally called the opcode.
- *rs:* The first register source operand.
- *rt:* The second register source operand.
- *rd:* The register destination operand. It gets the result of the operation.
- *shamt:* Shift amount. (Section 2.6 explains shift instructions and this term; it will not be used until then, and hence the field contains zero in this section.)
- *funct:* Function. This field, often called the *function code,* selects the specific variant of the operation in the op field.

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32_{ten}	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34_{ten}	n.a.
add immediate	I	8_{ten}	reg	reg	n.a.	n.a.	n.a.	constant
lw (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address

LOGICAL INSTRUCTION

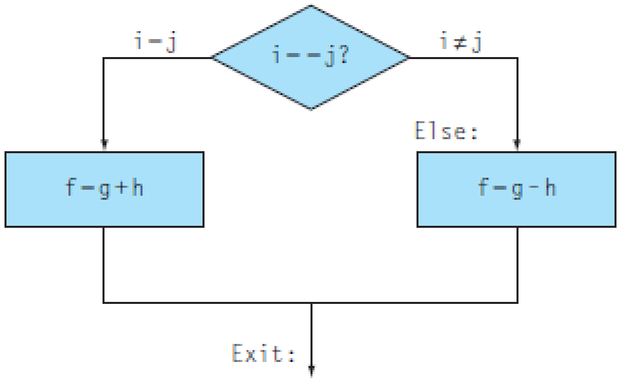
Logical operations	C operators	Java operators	MIPS instructions
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit NOT	~	~	nor

```
beq register1, register2, L1
```

This instruction means go to the statement labeled `L1` if the value in `register1` equals the value in `register2`. The mnemonic `beq` stands for *branch if equal*. The second instruction is

```
bne register1, register2, L1
```

It means go to the statement labeled `L1` if the value in `register1` does *not* equal the value in `register2`. The mnemonic `bne` stands for *branch if not equal*. These two instructions are traditionally called conditional branches.



Case/Switch Statement

Most programming languages have a *case* or *switch* statement that allows the programmer to select one of many alternatives depending on a single value. The simplest way to implement *switch* is via a sequence of conditional tests, turning the *switch* statement into a chain of *if-then-else* statements. Sometimes the alternatives may be more efficiently encoded as a table of addresses of alternative instruction sequences, called a **jump address table** or **jump table**, and the program needs only to index into the table and then jump to the appropriate sequence. The jump table is then just an array of words containing addresses that correspond to labels in the code. The program loads the appropriate entry from the jump table into a register. It then needs to jump using the address in the register. To support such situations, computers like MIPS include a *jump register* instruction (jr), meaning an unconditional jump to the address specified in a register. Then it jumps to the proper address using this instruction

Nested Loop

Procedures that do not call others are called *leaf* procedures. Life would be simple if all procedures were leaf procedures, but they aren't. Just as a spy might employ other spies as part of a mission, who in turn might use even more spies, so do procedures invoke other procedures. Moreover, recursive procedures even invoke "clones" of themselves. Just as we need to be careful when using registers in procedures, more care must also be taken when invoking nonleaf procedures.

<u>ADDRESSING AND ADDRESSING MODES</u>

1. *Immediate addressing*, where the operand is a constant within the instruction itself

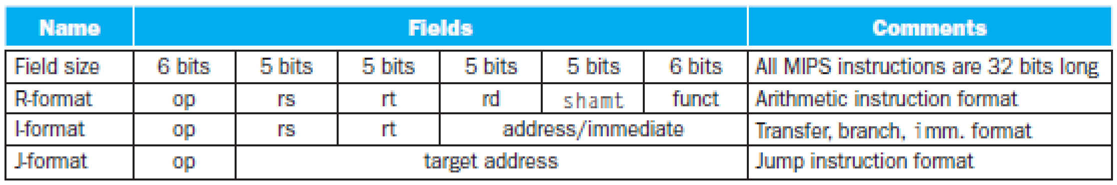
2. *Register addressing*, where the operand is a register

3. *Base* or *displacement addressing*, where the operand is at the memory location whose address is the sum of

a register and a constant in the instruction

4. *PC-relative addressing*, where the branch address is the sum of the PC and a Constant in the instruction

5. *Pseudodirect addressing*, where the jump address is the 26 bits of the instruction concatenated with the upper bits of the PC



Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

Implementation of Variables and Constants:

Variables:

The value can be changed as needed using the appropriate instructions.

There are 2 accessing modes to access the variables. They are

- Register Mode
- Absolute Mode

Register Mode:

The operand

The operand is in new location.

The address of this location is given explicitly in the instruction.

Eg: MOVE LOC,R2

The above instruction uses the register and absolute mode.

The processor register is the temporary storage where the data in the register are accessed using register mode.

The absolute mode can represent global variables in the program.

Mode Assembler Syntax Addressing Function

Register mode Ri EA=Ri

Absolute mode LOC EA=LOC

Where EA-Effective Address

Constants:

Address and data constants can be represented in assembly language using Immediate Mode.

Immediate Mode.

The operand is given explicitly in the instruction.

Eg: Move 200 immediate ,R0

It places the value 200 in the register R0.The immediate mode used to specify the value of source operand.

In assembly language, the immediate subscript is not appropriate so # symbol is used.

It can be re-written as

Move #200,R0

Assembly Syntax: Addressing Function

Immediate #value Operand =value

Indirection and Pointers:

Instruction does not give the operand or its address explicitly.Instead it provides information from which the new address of the operand can be determined.This address is called effective Address (EA) of the operand.

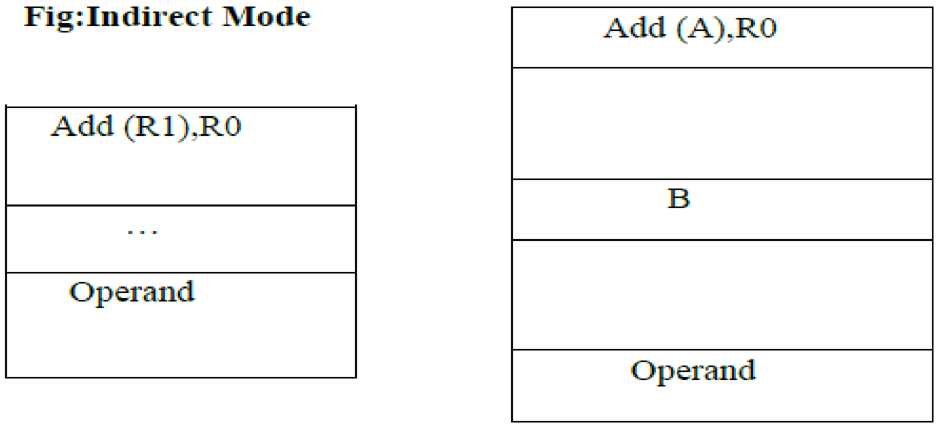
Indirect Mode:

The effective address of the operand is the contents of a register .

We denote the indirection by the name of the register or new address given in the instruction.

instruction.

Fig:Indirect Mode



Address of an operand(B) is stored into R1 register.If we want this operand,we can get it through register R1(indirection).

The register or new location that contains the address of an operand is called the **pointer.**

Mode Assembler Syntax Addressing Function

Indirect Ri , LOC EA=[Ri] or EA=[LOC]

Indexing and Arrays:

Index Mode:

The effective address of an operand is generated by adding a constant value to the contents of a register.

The constant value uses either special purpose or general purpose register. We indicate the index mode symbolically as, **X(Ri)**

Where **X** – denotes the constant value contained in the instruction

Ri – It is the name of the register involved.

The Effective Address of the operand is,

EA=X + [Ri]

The index register R1 contains the address of a new location and the value of X defines an offset(also called a displacement).

To find operand,

First go to Reg R1 (using address)-read the content from R1-1000

Add the content 1000 with offset 20 get the result.

1000+20=1020

Here the constant X refers to the new address and the contents of index register define the offset to the operand.

The sum of two values is given explicitly in the instruction and the other is stored in register.

Eg: Add 20(R1) , R2 (or) EA=>1000+20=1020

Index Mode	Assembler Syntax	Addressing Function
Index	X(Ri)	EA=[Ri]+X
Base with Index	(Ri,Rj)	EA=[Ri]+[Rj]
Base with Index and offset	X(Ri,Rj)	EA=[Ri]+[Rj] +X

Relative Addressing:

It is same as index mode. The difference is, instead of general purpose register, here we can use program counter(PC).

Relative Mode:

The Effective Address is determined by the Index mode using the PC in place of the general purpose register (gpr).

This mode can be used to access the data operand. But its most common use is to specify the target address in branch instruction.Eg. Branch>0 Loop

It causes the program execution to goto the branch target location. It is identified by the name loop if the branch condition is satisfied.

Mode Assembler Syntax Addressing Function

Relative X(PC) EA=[PC]+X

Additional Modes:

There are two additional modes. They are

- Auto-increment mode
- Auto-decrement mode

Auto-increment mode:

The Effective Address of the operand is the contents of a register in the instruction.

After accessing the operand, the contents of this register is automatically incremented to point to the next item in the list.

Mode Assembler syntax Addressing Function

Auto-increment (Ri)+ EA=[Ri];

Increment Ri

Auto-decrement mode:

The Effective Address of the operand is the contents of a register in the instruction.

After accessing the operand, the contents of this register is automatically decremented to point to the next item in the list.

Mode Assembler Syntax Addressing Function

Auto-decrement -(Ri) EA=[Ri];

Decrement Ri