# RISC versus CISC

Complex instruction set computers evolved over time

The desire was to close the “semantic gap”

- the difference in the way operations are specified in an expressive high-level language such as Java or C++ and the their hardware implementation

more sophisticated instructions were included in the instruction set such as:

- string search, string compare, string translation
- automating looping

# RISC versus CISC

An instruction that just adds to integers to produce a sum is considered a simple instruction

One that copies an element from one array to another and automatically updates both array subscripts is considered a complex instruction

CISC systems have intricate addressing modes

- to ease processing high level data structures

A goal was to simplify high level instruction translation

- by making the CISC machine instructions more closely resemble the high level functions

# RISC versus CISC

Microprogramming provides these more complex features
- microinstructions carry out the many intricate steps
- easier to implement than hardwired logic

A machine instruction is interpreted by a microprogram
- may be as many as a dozen or more microinstructions
- they direct hardware in performing the required actions

Microprograms are stored in an internal control memory
- the same chip as the CPU
- take additional time to process compared to hardwired logic

# RISC versus CISC

The use of these complex features reached a peak during the 1970s and early 1980s.

Processors of that era following this design approach included:

- the Intel 8086, 80286, 80386 and 80486
- Motorola 68K series
- Digital Equipment (DEC) VAX processors

Memory was very expensive and relatively small in capacity
Using complex instructions made programs smaller

# RISC versus CISC

CISC type instructions may use multiple memory operands
- each operand may be referenced using a different addressing mode

The instructions support a large and flexible set of operations

They can vary in size from one byte to 15 or more bytes

They must be partly decoded to know how many additional bytes makeup the instruction

This prevents pre-fetching instructions to speed processing

# Example CISC Instructions

x86 CISC type instructions include:

scas (string scan) - scans a block of memory looking for a match to the contents of a register and automatically updates counter and pointers.

xlat (translate) – performs a table lookup to convert the contents of a register to a number stored in a memory table.

loop – decrements and tests a counter and jumps based on the outcome

# Example CISC Instructions

VAX CISC type instructions include:

PUSHR #^M<R0, R1, R2, R3, R4, R5>
- which saves a series of registers on the stack based on a specified bit mask.

POPR #^M<R0, R1, R2, R3, R4, R5>
- which restores a series of registers from the stack based on a specified bit mask.

These are functions that would be performed as part of a call to a procedure or in returning from a procedure

# Example CISC Instructions

A Motorola 68K  CISC type instruction is:

CAS (compare and swap) -  which compares the value in a memory location with the value in a data register, and copies a second data register into the memory location if the compared values are equal, otherwise copies the contents of the memory location into the first register

# CISC Disadvantages

The instructions are difficult to implement in hardware and require microprogramming

microcode slows down the system – after fetching each machine instruction, a series of microinstructions must be retrieved and executed to interpret the machine instruction

Compiler writers often avoid using CISC instructions that are unique to one machine in favor of generating groups of more standard simple instructions to perform the same task

# CISC Disadvantages

Intricate addressing modes and variable length instructions require a more complex and thus slower control unit

Allowing most instructions to reference memory operands makes the instruction take longer to execute

Complex instructions make pipelining more difficult

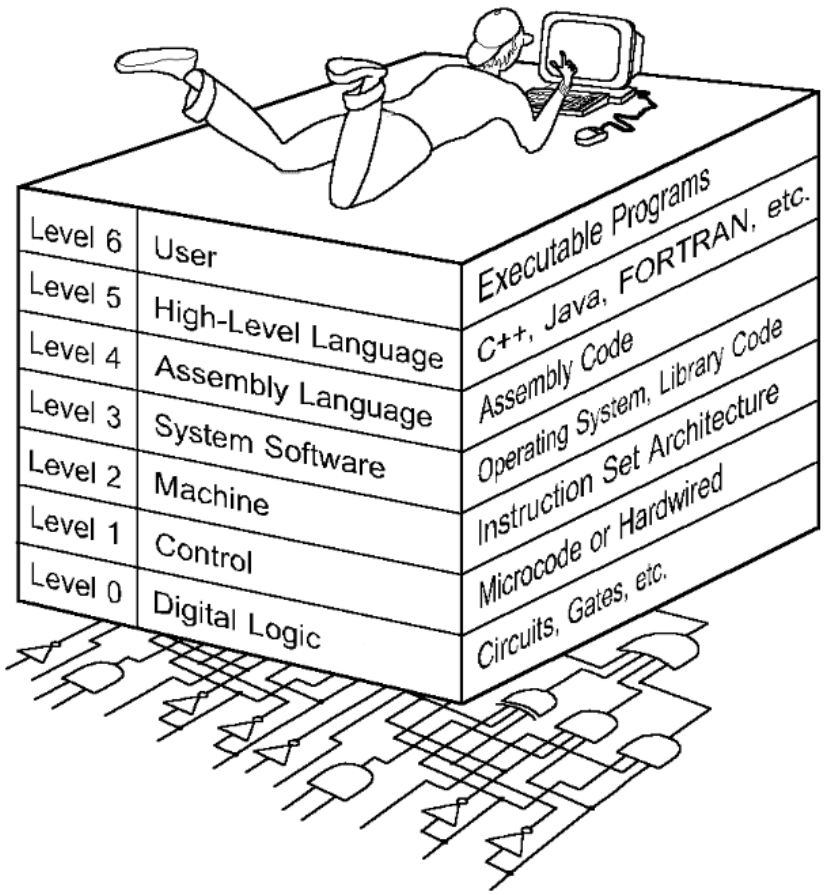
# The Computer Hierarchy

- Computers can be viewed at different levels
  - Each layer corresponds to a *"virtual machines"*
  - Each layer provides services to the level above
  - Each layer abstracts away the details of the level below

- "Programs" at each layer can be:
  - translated into the form of the next lower level
  - interpreted by a program at the next lower

# The Computer Hierarchy

- Each virtual machine layer is an abstraction of the level below it.
- The machines at each level execute their own particular instructions, calling upon machines at lower levels to perform tasks as required.
- Computer circuits ultimately carry out the work.



# The Computer Hierarchy

- Level 4: Assembly Language Level
  - Acts upon assembly language produced from Level 5, as well as instructions programmed directly at this level.
- Level 3: System Software Level
  - Controls executing processes on the system.
  - Protects system resources.
  - Assembly language instructions often pass through Level 3 without modification.

# The Computer Hierarchy

- Level 2: Machine Level
  - Also known as the Instruction Set Architecture (ISA) Level.
  - Consists of instructions that are particular to the architecture of the machine.
  - Programs written in machine language need no compilers, interpreters, or assemblers.

# The Computer Hierarchy

- Level 1: Control Level
  - A *control unit* decodes and executes instructions and moves data through the system.
  - Control units can be *microprogrammed* or *hardwired*.
  - A microprogram is a program written in a low-level language that is implemented by the hardware.
  - Hardwired control units consist of hardware that directly executes machine instructions.

# The Computer Hierarchy

- Level 0: Digital Logic Level
  - This level is where we find digital circuits (the chips).
  - Digital circuits consist of gates and wires.
  - These components implement the mathematical logic of all other levels.

# Overview of the MIPS Architecture

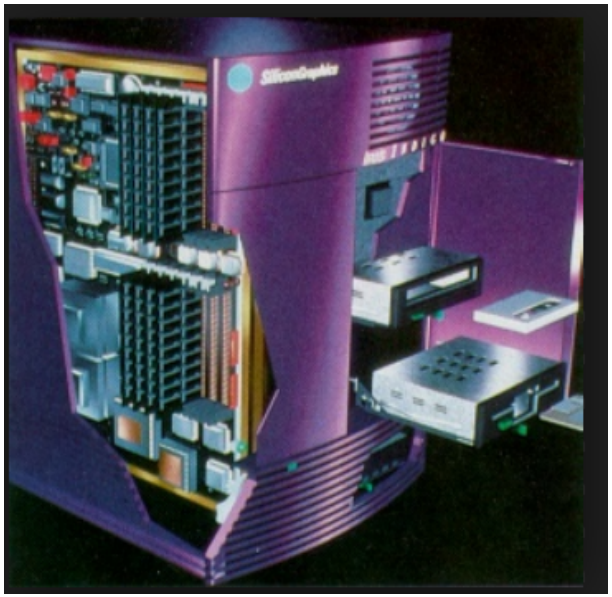
This course is based on the MIPS R3000 processor as an example of a RISC System

# Sample applications of MIPS processors

MIPS processors are commonly found in:

- Set-top boxes
- Game consoles
- VoIP SoCs
- Digital TVs and DVD recorders/players
- Workstations

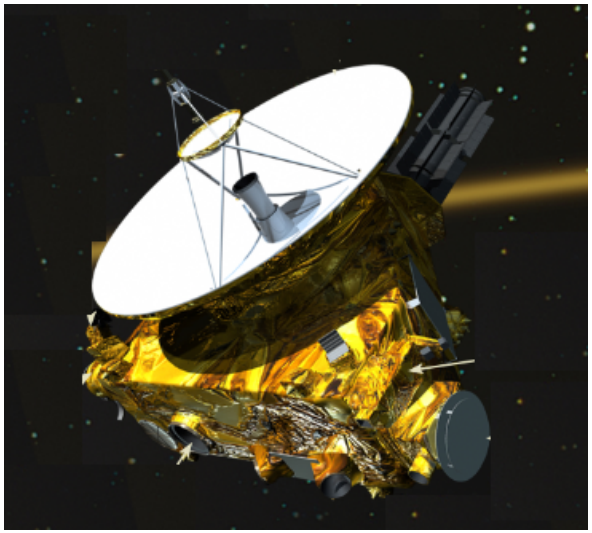
# Sample applications of MIPS processors

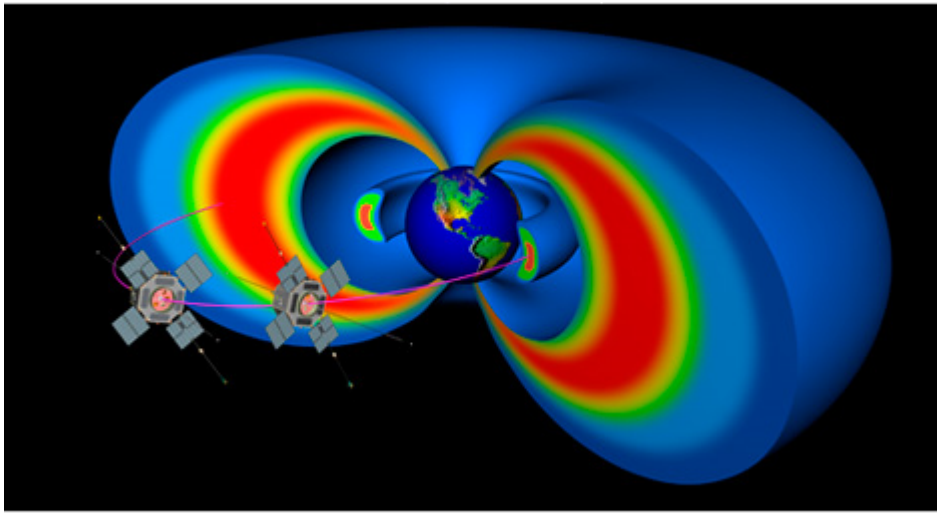



Sillicon Graphics (SGI) Workstations were based on MIPS R4000 processors, as were Nintendo 64 game consoles.

The R4000 was the first commercially available 64-bit microprocessor.

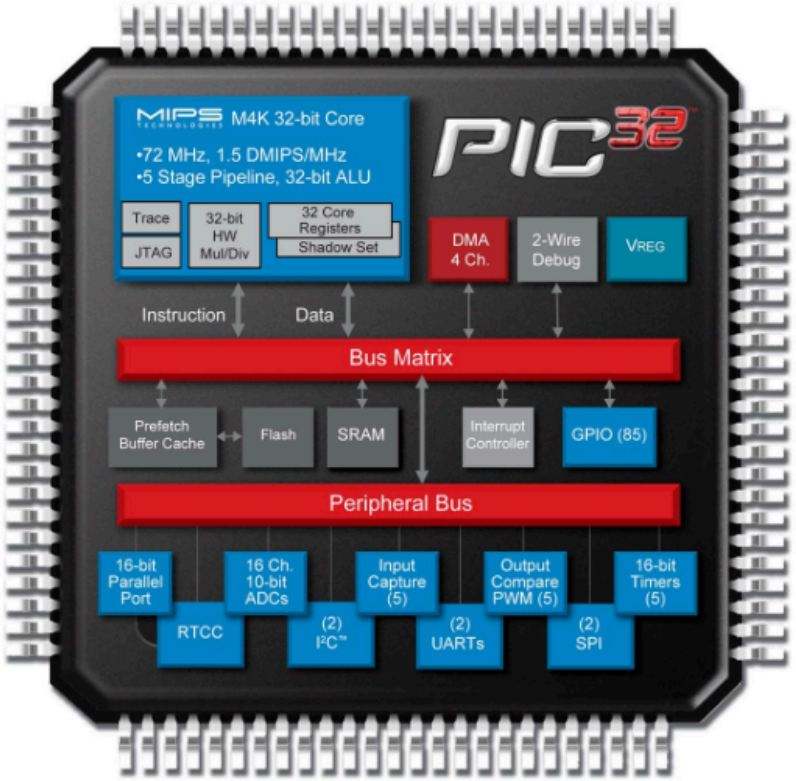
# Sample applications of MIPS processors





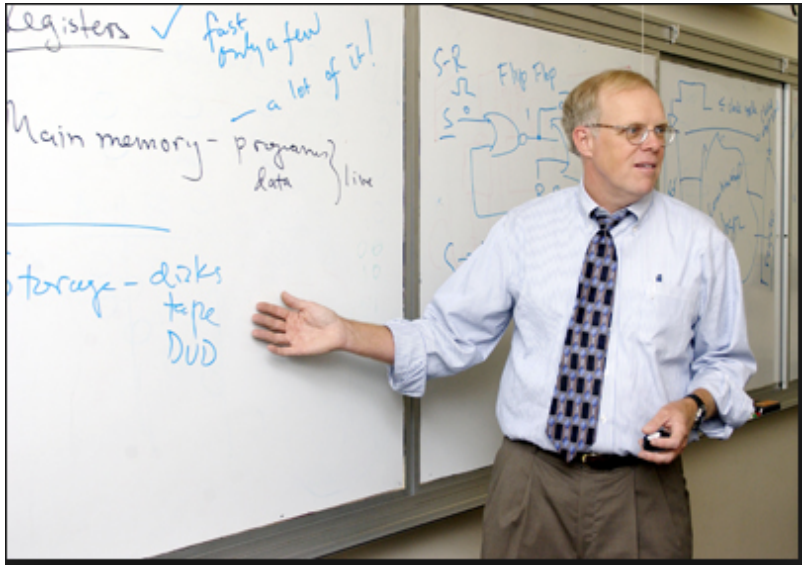
MIPS processors control the subsystems on the New Horizons Spacecraft to Pluto and were used to implement Software Defined Radios on the twin Van Allen Probes.

# MIPS targets high-performance embedded designs




Used in digital consumer products and has a growing presence in mobile devices

MIPS has become a standard and performance leader in the embedded industry.




The design of the original MIPS processor was developed by Stanford University professor of engineering John Hennessy

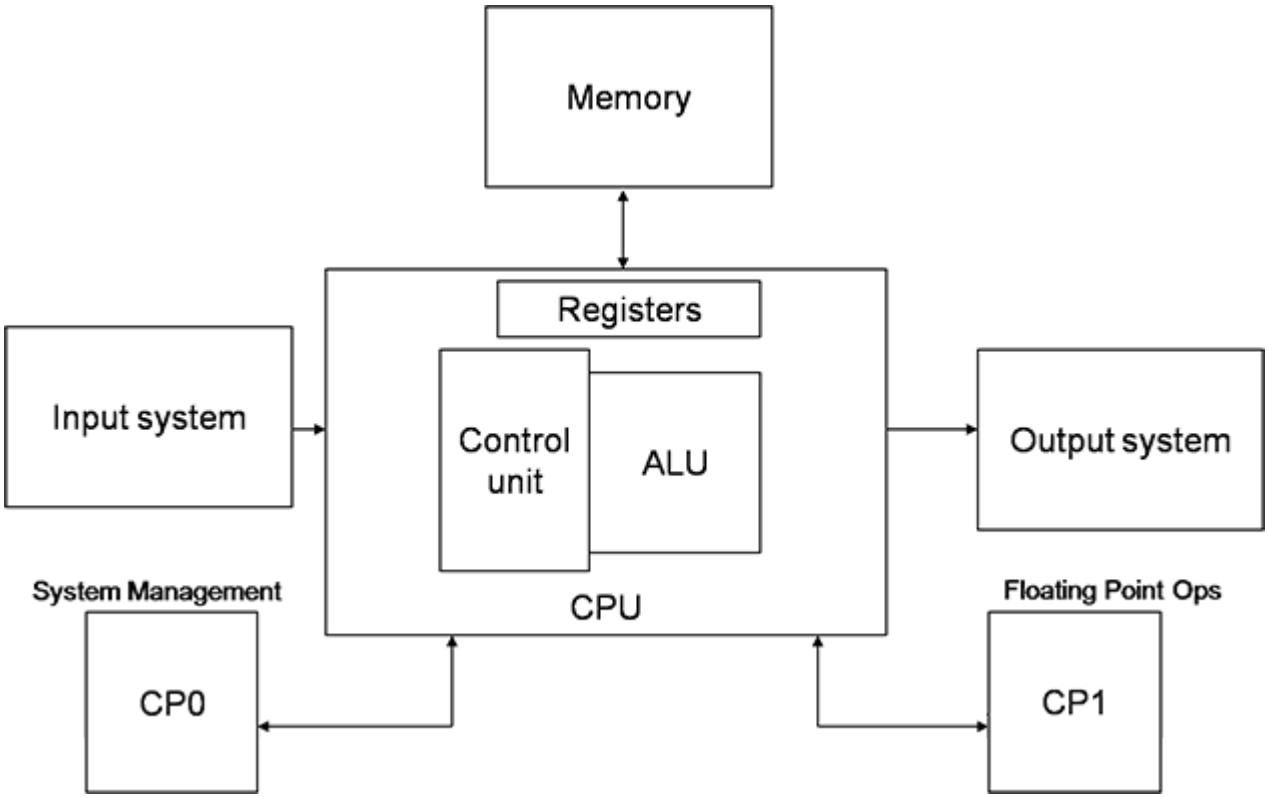
The MIPS R3000 processor is a 32-bit machine with the following RISC-like features:

- Fixed size machine instructions (32 bits)
- Only load and store instructions can reference memory (load/store architecture)
- Uses 32-bit addresses
- Pipelined functional units

# MIPS RISC-like features:

- only three formats for machine instructions, all of which are 32 bits
- only four addressing modes
- 32-bit registers ($0, $1, ... , $31, Hi and Lo)

# MIPS Organization



# MIPS Registers

- CPU contains 32 registers, each 32 bits wide
- Registers may be used for general purposes
- MIPS calling convention should be adhered to if the user program is to be compatible with library routines and system software.

- Either register names or numbers may be used within assembly language statements
- Format: $n, where n is the register number (0 – 31)
- Register numbers are encoded in 5-bit fields within machine instructions
- Register names serve as more mnemonic aliases for the register numbers within assembly language statements

# MIPS Registers

- Register $0 ($zero) is hardwired to zero
- (reads as 0 and cannot be overwritten)

- Serves a convenient source or a zero constant

Example use: add $3,$2,$0 has same effect as
move $3,$2 to copy $2 into $3

- Register $1 ($at) is used by the assembler to implement pseudo-instructions (i.e. synthetic instructions)

# MIPS Registers

- Temporary registers are not preserved across function calls:

  $t0 - $t7  (register numbers 8 – 15)

  $t8 - $t9  (register numbers 24 – 25)

- Saved registers must be saved and restored if used within a function:

  $s0 - $s7 (register numbers 16 – 23)

# Role of Registers in Procedure Calling

Steps required:

1. Place parameters in registers
2. Transfer control to procedure
3. Acquire storage for procedure
4. Perform procedure's function
5. Place result in register for caller
6. Return to place of call

# Role of Registers in Procedure Calling

$a0 – $a3: arguments (reg's 4 – 7)

$v0, $v1: result values (reg's 2 and 3)

$t0 – $t9: temporaries (can be overwritten by callee)

$s0 – $s7: Must be saved/restored by callee

$gp: global pointer for static data (reg 28)

$sp: stack pointer (reg 29)

$fp: frame pointer (reg 30)

$ra: return address (reg 31)

# MIPS Instruction Types

- R-type employs register operands
- I-type contains a literal (immediate value)
- J-type contains part of the jump address

# MIPS Instruction Summary

| Instruction type | Examples |
|---|---|
| arithmetic | add, sub, addu, subu, mult, multu, div, divu |
| Logical | and, andi, or, ori, xor, xori, nor, sll, srl, sra, sllv, srlv, srav |
| Data transfer | lw, lh, lb, sw, sh, sb, lui, mfhi, mflo, mthi, mtlo |
| Conditional branch | beq, bne, bltz, blez, bgtz, bgez |
| Unconditional branch | b, j, jr, jal |
| Comparison | slt, slti, sltu, sltiu |

# Addressing Modes

The MIPS only supports the following addressing modes:

1. Register mode (operands in registers)

2. Immediate mode (literal contained in instruction)

3. Base relative mode (offset + contents of base register give the memory address of the operand)

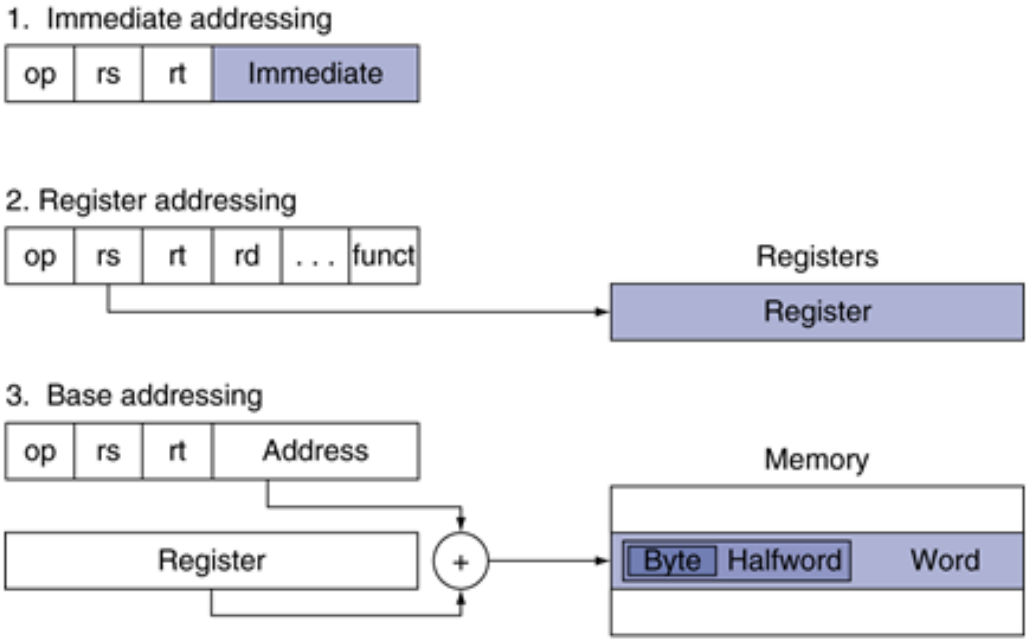
4. PC-relative: PC + (4*displacement)  gives the branch target address)

5. Pseudo-direct (rightmost 26 bits with machine instruction is multiplied by 4 and concatenated with upper 4 bits of PC to yield the jump address)

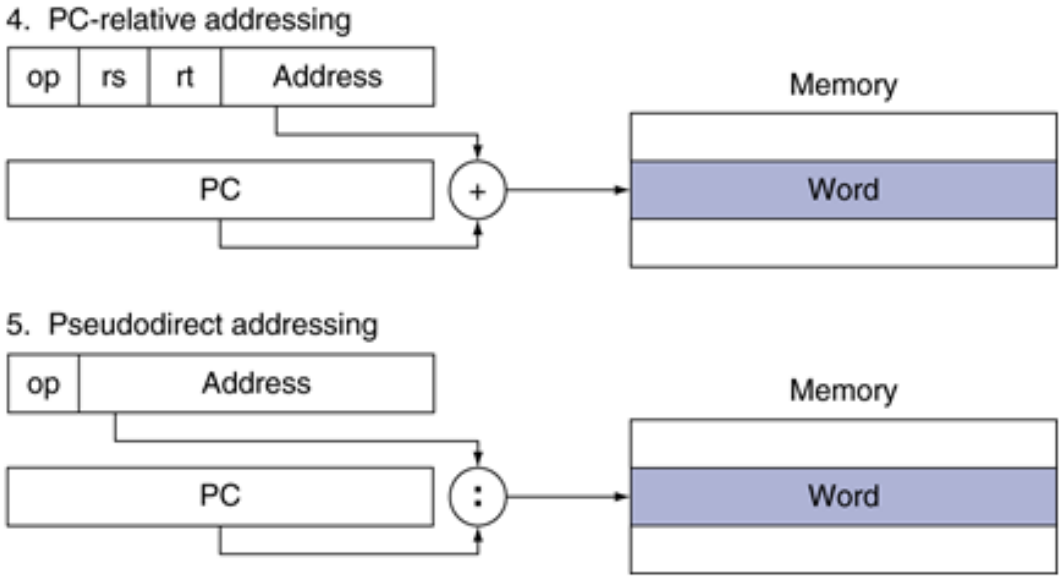
- The 16-bit immediate values are sign-extended to 32 bits by the arithmetic, load/store and branch instructions.

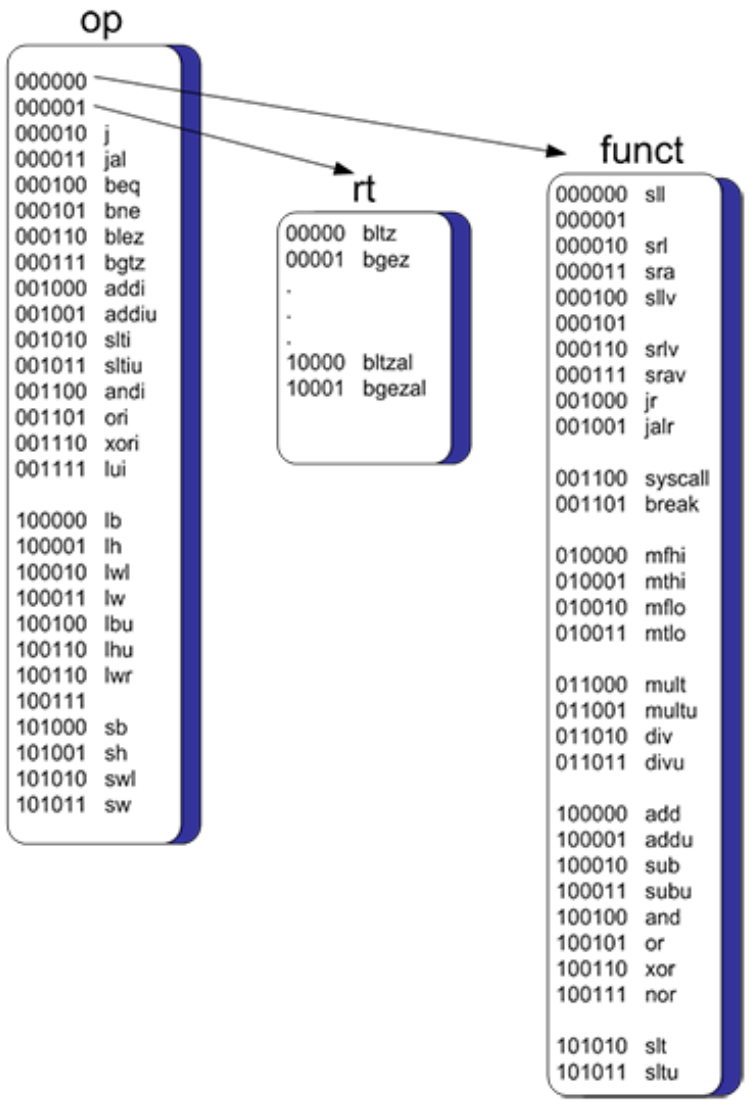
- The logical instructions (e.g. AND, OR, etc.) generate the 32-bit immediate operand by zero-extending the 16-bit immediate field contained in the machine instruction.

# Addressing Mode Summary



# Addressing Mode Summary






Their fixed size and regularity make it easier and faster for the control unit to interpret the machine instructions.

The opcode is always the leftmost 6 bits.

When the opcode is 0, the operation is determined by rt field.

When the opcode is 1, the operation is determined by function code field in the rightmost 6 bits.

# Load/Store Architecture

- Compilers that generate code for RISC processors try to maximize and optimize the use of registers
- Only the load and store instructions can access memory operands
- Registers are faster to access than memory
- Less frequently used variables are spilled to memory
- This improves performance and makes the common case fast

# Factors affecting Performance

Programs are easier to write in high level languages
- examples are C, C++ and Java

Compilers must translate high level language programs
- target programs can be assembly language
- or machine code

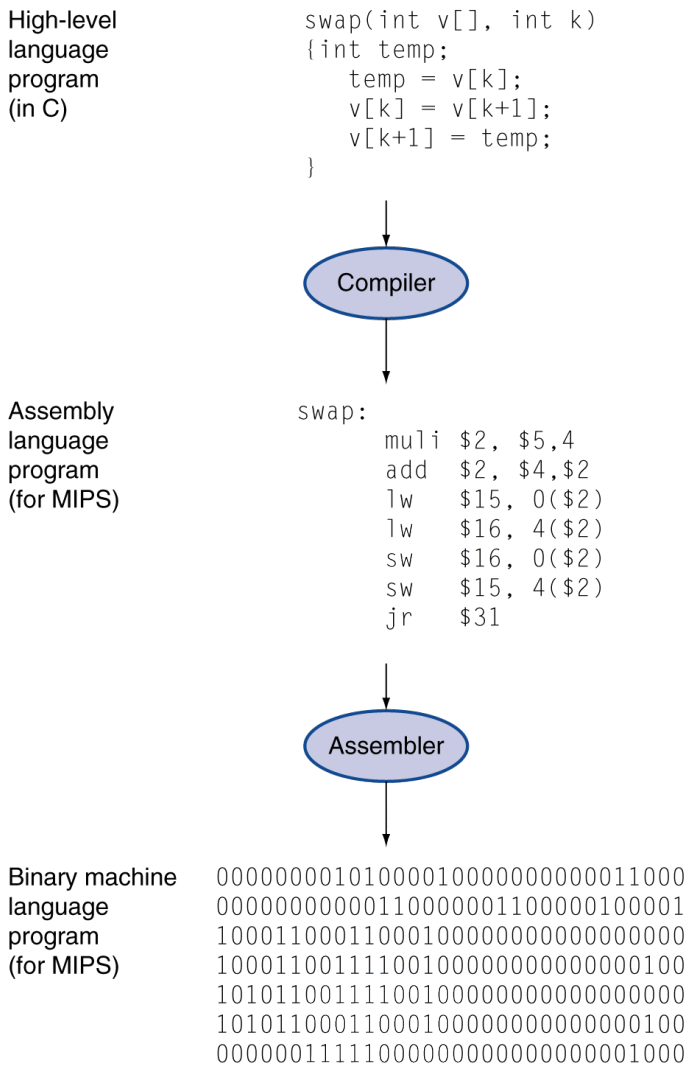
Assemblers generate machine code from assembly programs
- which are symbolic representations of machine code

Processors can only execute machine code

The *execution cycle* specifies how instructions are carried out

# Factors affecting Performance

- High-level language
  - Level of abstraction closer to problem domain
  - Provides for productivity and portability
- Assembly language
  - Textual representation of instructions
- Hardware representation
  - Binary digits (bits)
  - Encoded instructions and data



# Factors affecting Performance

The execution cycle consists of one or more steps

Each step requires a clock cycle

cycle time is the duration of a clock cycle (clock period)
- Instructions that take fewer cycles execute faster
- The shorter the cycle time the faster the execution

The clock rate is the number of cycles per second (Hz)
Clock rate = 1/cycle time

Simple RISC type instructions take fewer cycles

CISC type instructions tend to take more cycles

# Factors affecting Performance

Programs that contain fewer instructions are faster

Programs that use mainly simple instructions are faster

Ways to improve performance include using:

- Efficient algorithms
- Efficient fast hardware
- Fast memory
- Parallel operations

Metrics are needed to access performance improvements

# Measuring Performance

- 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?
- We'll focus on response time for now…

# Measuring Performance

- Define Performance = 1/Execution Time

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

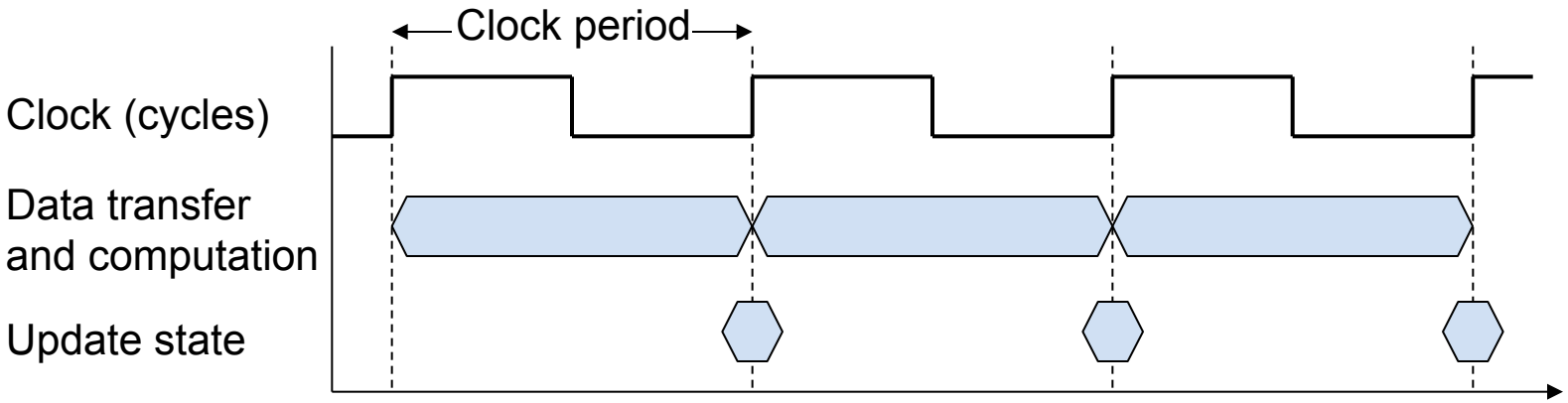
  - "X is $n$ times as fast as than Y"
- Example: time taken to run a program
  - 10s on computer A, 15s on computer B
  - $\text{Execution Time}_B$ / $\text{Execution Time}_A$ = 15s / 10s = 1.5
  - So A is 1.5 times as fast as B

# Measuring Performance

- 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
  - includes user CPU time and system CPU time
  - Different programs are affected differently by CPU and system performance

# Measuring Performance

- Digital hardware is driven by a constant-rate clock




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

# CPU Time

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

$$= \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}}$$

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

# CPU time example

- Computer A: 2GHz clock, program takes 10s CPU time
- Designing Computer B
  - Desire is for 6s CPU time
  - Suppose using a faster clock requires 1.2 × clock cycles
- How fast must Computer B clock be?

$$\text{Clock Rate}_B = \frac{\text{Clock Cycles}_B}{\text{CPU Time}_B} = \frac{1.2 \times \text{Clock Cycles}_A}{6\text{s}}$$

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

$$= 10\text{s} \times 2\text{GHz} = 20 \times 10^9$$

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

# Instruction count and CPI

$$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$$

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$

$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

- 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 is affected by instruction mix

# 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?

$$\text{CPU Time}_A = \text{Instruction Count} \times \text{CPI}_A \times \text{Cycle Time}_A$$

$$= I \times 2.0 \times 250\text{ps} = I \times 500\text{ps}$$

A is faster…

$$\text{CPU Time}_B = \text{Instruction Count} \times \text{CPI}_B \times \text{Cycle Time}_B$$

$$= I \times 1.2 \times 500\text{ps} = I \times 600\text{ps}$$

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

…by this much

# Average CPI example

- With different instruction classes:

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

  - Weighted average CPI

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

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

# Performance Summary

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

- Performance depends on
  - Algorithm: affects IC, possibly CPI
  - Programming language: affects IC, CPI
  - Compiler: affects IC, CPI
  - Instruction set architecture: affects IC, CPI, $T_c$

# Performance Metrics

- MIPS: Millions of Instructions Per Second
  - Doesn't account for
    - Differences in ISAs between computers
    - Differences in complexity between instructions

$$\text{MIPS} = \frac{\text{Instruction count}}{\text{Execution time} \times 10^6}$$

$$= \frac{\text{Instruction count}}{\frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}} \times 10^6} = \frac{\text{Clock rate}}{\text{CPI} \times 10^6}$$

  - CPI is the average for a program
  - and yields the "*native MIPS*"

# Performance Metrics

- MIPS can be misleading
  - Varies between programs on the same machine
  - Does not reflect the instruction set complexity
  - May vary inversely with performance

- The following example illustrates this last point:

# MIPS Example

Assume a program is translated by two different compilers for the same machine:

$$MIPS = \frac{\text{Clock rate}}{CPI \times 10^6}$$

| | Instruction counts (in millions) for each instruction class | | |
|---|---|---|---|
| Instruction Class | A | B | C |
| Code from compiler 1 | 4 | 2 | 1 |
| Code from compiler 2 | 9 | 1 | 1 |

$CPI_1 = ((4*1 + 2*2 + 1*3)*10^6) / ((4+2+1)*10^6) = 1.57$

Thus $MIPS_1 = 100 \text{ MHz} / 1.57*10^6 = 63.64$

$\text{CPU time}_1 = ((4+2+1)*10^6* 1.57) /100 * 10^6 = 0.11 \text{ sec}$

# MIPS Example

Assume a program is translated by two different compilers for the same machine:

| | Instruction counts (in millions) for each instruction class | | |
|---|---|---|---|
| Instruction Class | A | B | C |
| Code from compiler 1 | 4 | 2 | 1 |
| Code from compiler 2 | 9 | 1 | 1 |

$CPI_2 = ((9*1 + 1*2 + 1*3)*10^6) / ((9+1+1)*10^6) = 1.27$

Thus $MIPS_2 = 100\ MHz / 1.27*10^6 = 78.57$

$CPU\ time_2 = ((9+1+1)*10^6* 1.27) /100 * 10^6 = 0.14\ sec$

Yields a higher MIPS rating but takes longer to execute

# Performance Metrics

- *Native MIPS* was defined above
- *Peak MIPS* is based on minimum CPI
  - Assumes all instructions in program have minimum CPI
  - Minimum possible CPI = 1
  - All instructions require at least 1 clock cycle

- Relative MIPS is based on a standard machine

$$MIPS_{relative} = \frac{Time_{reference}}{Time_{unrated}} \times MIPS_{reference}$$

# Bench Marks

- *Benchmarks are suites of programs*
  - Chosen to be typical of workloads
  - Systems are rated based on benchmark execution times
  - Average time is used as a measure of performance

- Arithmetic mean is not used as average
  - Outliers can have undue influence on result
  - Geometric mean is used instead (defined below)

# Performance Metrics

Geometric Mean →

$$\sqrt[n]{\prod_{i=1}^{n} \text{Execution time}}$$

- Standard Performance Evaluation Corp (SPEC)
  - Develops benchmarks for CPU, I/O, Web, …
- SPEC CPU2006
  - Elapsed time to execute a selection of programs
    - Negligible I/O, so focuses on CPU performance
  - Normalize relative to reference machine
  - Summarize as geometric mean of performance ratios
    - CINT2006 (integer) and CFP2006 (floating-point)

# Performance Metrics

- SPEC CPU2017
  - Comprised of a group of 43 programs
  - Organized into 4 suites

- Another possibility is the Harmonic Mean
  - More appropriate for averaging rates such as MIPS
  - Average MIPS for suite gives a single performance measure

Harmonic Mean 

$$\overline{S_H} = \frac{N}{\sum_1^N \frac{1}{S_i}}$$

# Amdahl's Law

- Improving an aspect of a computer and expecting a proportional improvement in overall performance

$$T_{improved} = \frac{T_{affected}}{improvement\ factor} + T_{unaffected}$$

- Example: multiply accounts for 80s/100s
  - How much must multiply be improved to get 5× the overall performance?

$$20 = \frac{80}{n} + 20$$

  - Can't be done!
- Corollary: make the common case fast

# Floating Point Performance

MIPS measures integer performance

Engineering Applications employ floating point

- So the appropriate metric is Mega-flops (MFLOPS)
- Floating point support varies even more
- trig and exponentiation functions may be available
- In other cases, only simple arithmetic  is supported

MFLOPS can be very unreliable in predicting performance

It is best to use execution time to assess performance

# Computer Architecture versus Computer Organization

The architecture of a computer defines:

the view of the computer from the perspective of an assembly language or machine language programmer.

The instruction set architecture (ISA)

defines the software hardware boundary

It includes:

- the instruction set
- the machine instruction formats
- the available addressing techniques
- the operational register set
- the format of the available data types

The architecture specifies what the computer can do

# Computer Architecture versus Computer Organization

"Computer Organization" is sometimes used interchangeably with "Computer Architecture"

but they have different meanings

Computer Organization is also called the microarchitecture

describes how the capability defined by the architecture is implemented

Architecture may define 32-bit memory word transfers

the organization may internally perform two 16-bit transfers

Computer models that share a common architecture may have different microarchitectures (i.e., organizations).

# Analogy

The architecture of a clock is defined by the movement of hands on a marked dial to indicate the time

However, one clock may be driven by a wind-up spring, while another, with the same architecture, may be driven by a crystal oscillator

The user may be unaware of the internal timing mechanism

A machine code program can run on different machines with the same architecture without change

- organizational details may differ

# Questions answered by the **Architecture**

- what types of operations are available?
  (integer, floating point, etc.)
- which instructions are allowed to reference memory?
- do operands have to reside in registers?

  or can one or more reside in memory?
- are vector type instructions and vector registers available?
- how many bits do the instruction operands require?
- is a segmented or flat memory model used?
- how many operands can instructions employ?

# Questions answered by the Organization

- are the instructions pipelined?
- is a single-cycle or multi-cycle datapath used?
- how many cache levels are employed?
- is secondary storage provided by magnetic disks or by flash?
- are there multiple buses?
- is the control unit microprogrammed or is hardwired logic used?
- are multiple execution units included?
- how many memory accesses are used to retrieve data or instructions?
- are vector type instructions provided by an array processor or by a vector processor?

# Low versus High Level Languages

Computers with different **organizations**, but with a common shared architecture can execute the same machine code program.

Machine code programs are what the compiler produces when it translates a high level language source program (such as C or C++).

High level languages are designed to abstract away or hide the architecture

The underlying machine architecture is only revealed at the assembly language and machine language levels

# Low versus High Level Languages

Assemblers translate symbolic assembly language into machine code

- Assembly instructions are  a higher level representation of machine instructions
- machine instructions are binary patterns understood by the computer hardware

Assembly language instructions and native built-in machine instructions have a one-to-one correspondence

A set of computers that share a common architecture is referred to as a computer family

# Computer Families

Examples of computer families include:
the Intel x86, Motorola 68000 and MIPS processor families

Members of a computer family can all run the same machine language programs although they tend to vary greatly in the speed with which the programs are executed.

Different technologies may be used to implement the various models within the family.

Computer families usually include lower cost lower performance models as well as more powerful, faster and more expensive models.

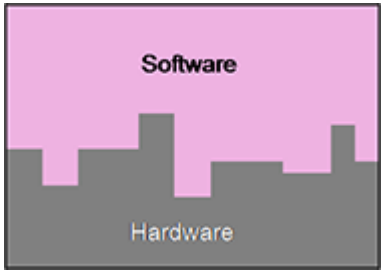
# Computer Families

New members of a computer family perform better
    but execute programs written for older members
    this is called "*backwards compatibility*"

Backwards compatibility is desirable
    runs existing software base on newer more powerful models allows for easier hardware upgrades

Programs must be re-complied to run on a machine with a different architecture

# Instruction Set Architecture (ISA)

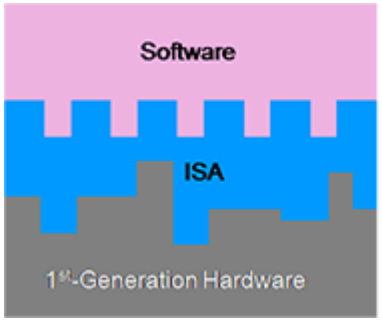


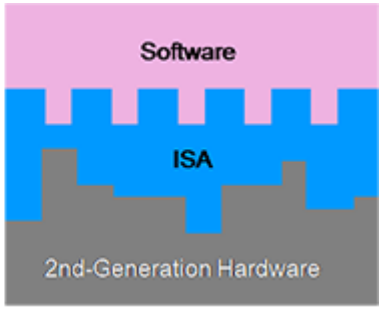

Initially programs were written to interact directly with a machine's unique hardware

Each update or improvement in the hardware required rewriting programs that had been previously produced

# Instruction Set Architecture (ISA)

Machine programs run on all machines with a standard shared ISA




This saves on development time and on cost

manufacturers can innovate and fine-tune the hardware for performance without breaking the existing software base

## Von Neumann Machines

Most modern computers are based on this design
Developed by John von Neumann at Princeton
At the Institute for Advanced Studies in the 1940's.

These machines have 3 major components:

- a CPU
- a main-memory system
- an I/O system

# von Neumann Characteristics

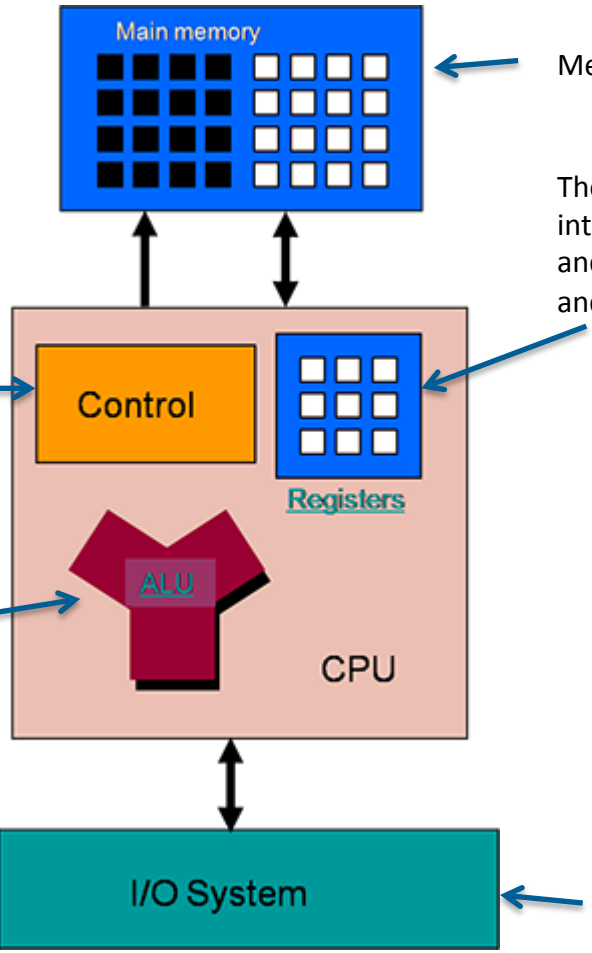
- Both programs and data are stored in a single memory
  program instructions can be manipulated like data

- The program counter is used to fetch instructions
  data operands are fetched during the execute cycle
  based on the operand addressing mode

- Instructions execute sequentially
- flow of control may be altered by a branch type instruction

- CPU accesses memory over a single path
  Which is a potential bottle neck
  Called the "von Neumann bottleneck”

The diagram below depicts a system that adheres to the von Neumann architecture:




Memory holds both instructions and data.

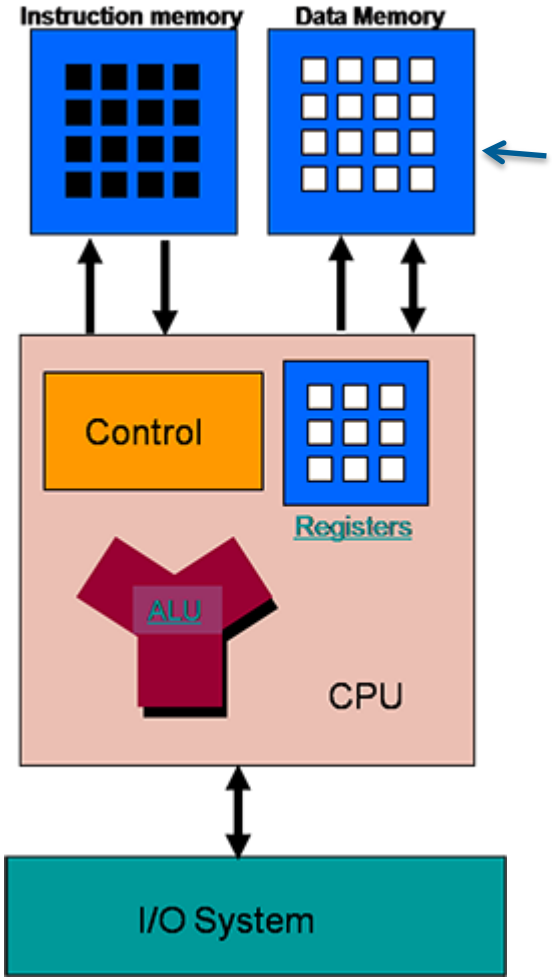
The register set (also called register file) serves as internal storage within the CPU to hold operands and intermediate values during instruction fetch and execution.

The major subsystems within the CPU are: The control unit (CU) which selects and interprets machine instructions and coordinates the various parts of the computer in executing these instructions.

The arithmetic logic unit (ALU) which performs arithmetic, logical, and shift operations on the operands and generates the results.

Allows data to be brought into the system and results to be sent out of the system.

As an alternative, illustrated below is a “Harvard architecture”:

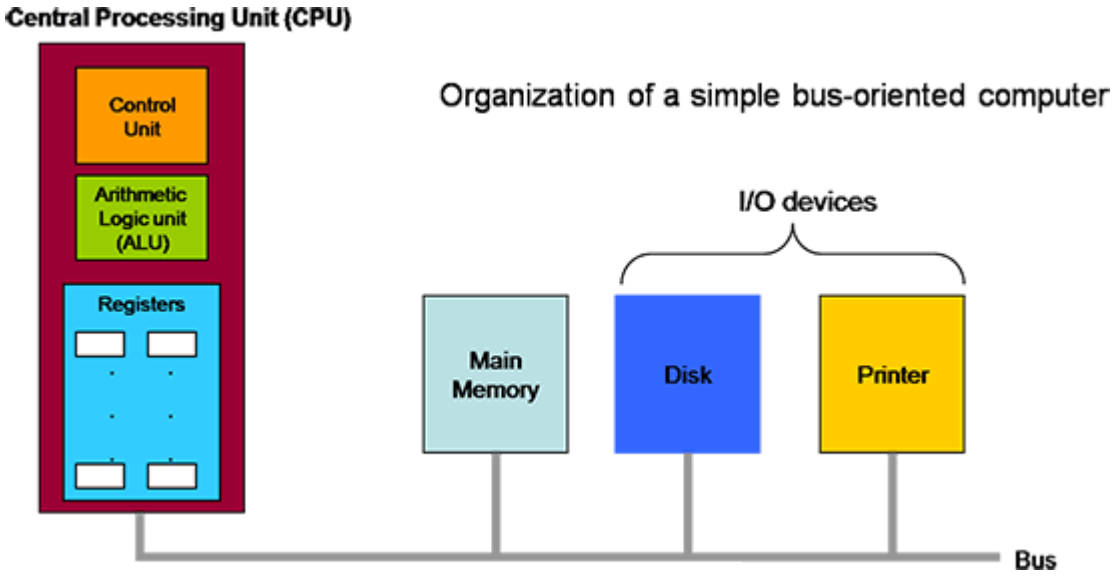



The distinguishing feature is the presence of separate instruction and data memories.

This allows one instruction to be fetched while another stores or reads an operand.

This design was developed at Harvard University by Howard Aiken and others.

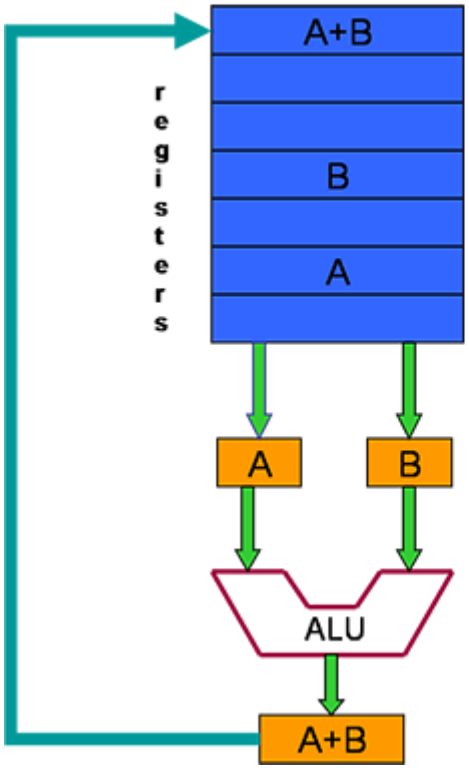
Computer components exchange data and communicate over a “system bus”:




A bus is a collection of parallel wires that carry address, data, and control signals
External buses connect the CPU to memory and I/O devices
Internal buses carry signals between registers, the ALU and the control unit

# The Datapath

## Different organizations have different datapaths

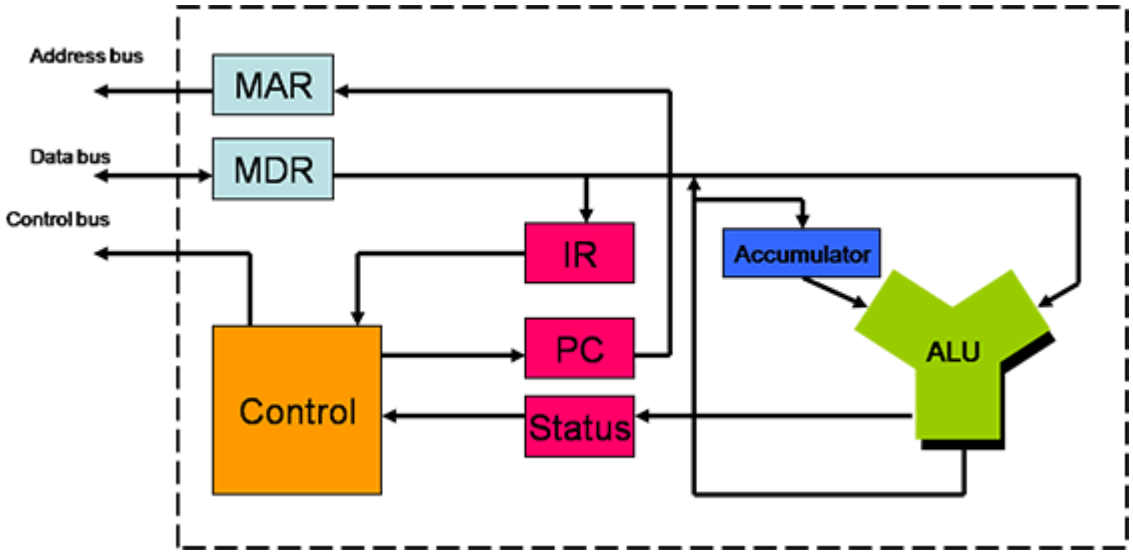



The registers, ALU and interconnecting buses constitute the data path for the machine.

The datapath contains all of the devices and pathways needed to execute instructions.

**Typical Data path for executing instructions**

Early designs used a special accumulator register
to hold an input operand for the instruction
and to received the instruction's result




Such systems employed one-address instructions
a single memory operand
the second implicit operand was the accumulator register

# Example One-address Instructions

| | | |
|---|---|---|
| **LDA** | **X** | **# load the variable X from memory into the accumulator** |
| **ADD** | Y | # add the memory variable Y to the accumulator |
| **STA** | Z | # store the accumulator result into memory variable Z |
| **BGE** | P1 | # branch to location P1 if result is not negative |

Most instructions incur a penalty
- the time required to access the memory operand

The alternative is a load/store architecture
- allows only a few instructions to use memory operands
- all others have to use operands that reside within registers
- registers takes a fraction of the memory access time (better performance)

# 2 and 3 address instructions

Accumulators limit the number of high cost CPU registers

Other designs use two or more instruction memory operands
This gives rise to two-address and three-address instructions:

```
Mul   x,y   # the variable x is multiplied by y and the product is stored back in x
Add   z,w   # the sum of z plus w is stored in z
```

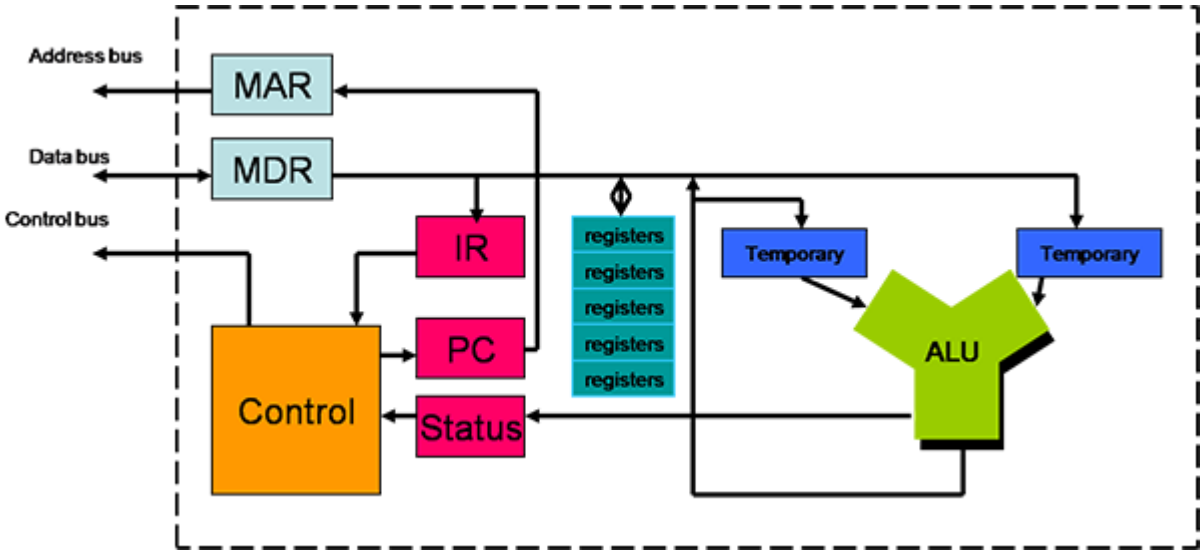
Three-address instructions:

```
Mul   z,x,y   # the product of x times y is stored into z
Add   w,x,y   # the sum of x plus y is stored into w
```

such instructions use even more memory accesses
therefore take longer to execute.

# Multi-register Systems

Over time, register cost declined and reliability increased




More on-chip registers allowed fast register-to-register operations

indexed and register indirect addressing could be used

# Multi-register Systems

MAR and MDR serve as the CPU to memory interface

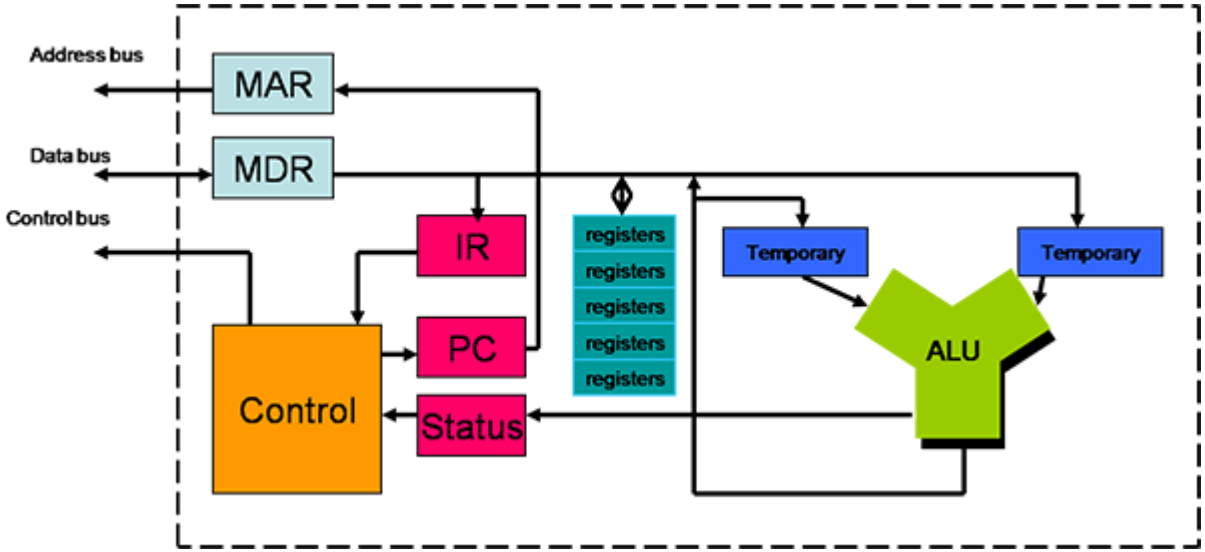
MAR (memory address register)

holds the address of the item in memory to be accessed.

MDR (memory data register)

receives the item read from memory

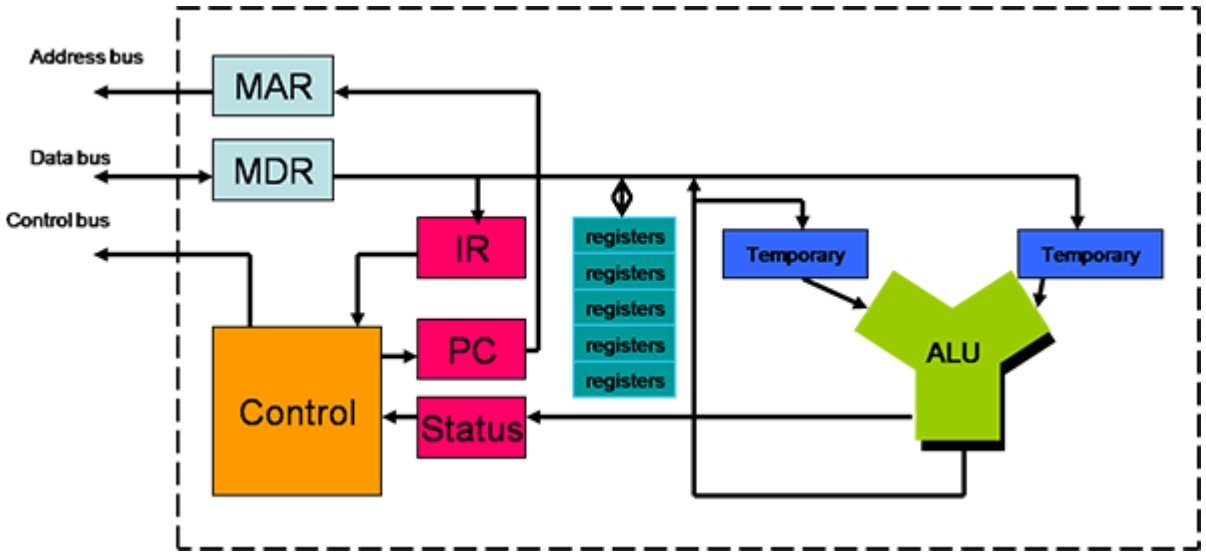
or holds the item to be written into memory



# Multi-register Systems

IR is the instruction register
Holds the instruction while the control unit processes it

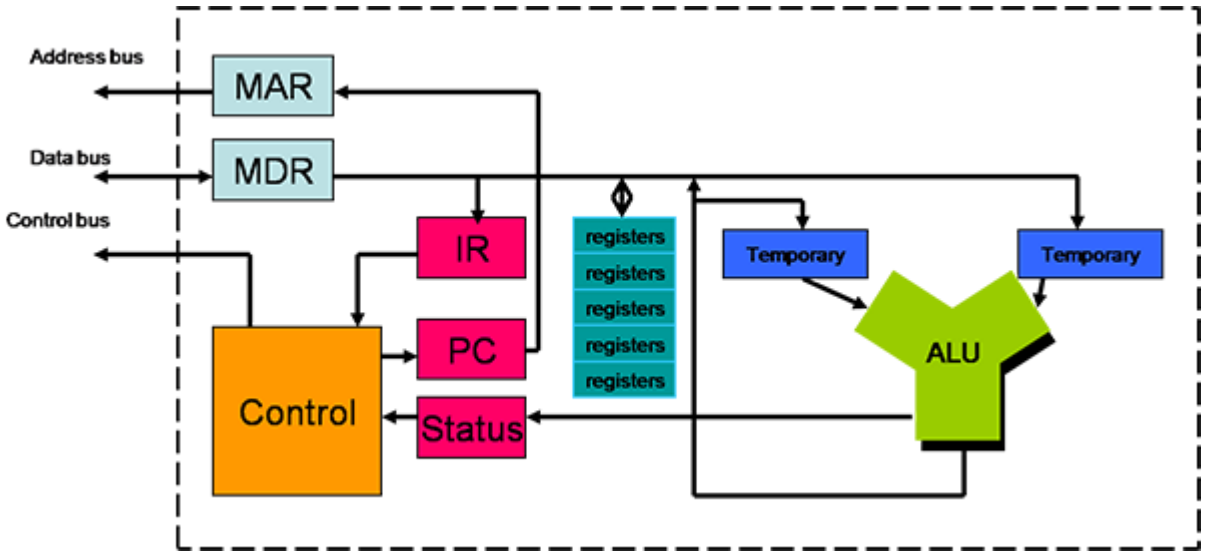
PC is the program counter register
holds the address of the next instruction to fetch from memory



# Multi-register Systems

STATUS register holds condition codes
( negative, zero, positive, a carry, etc.)

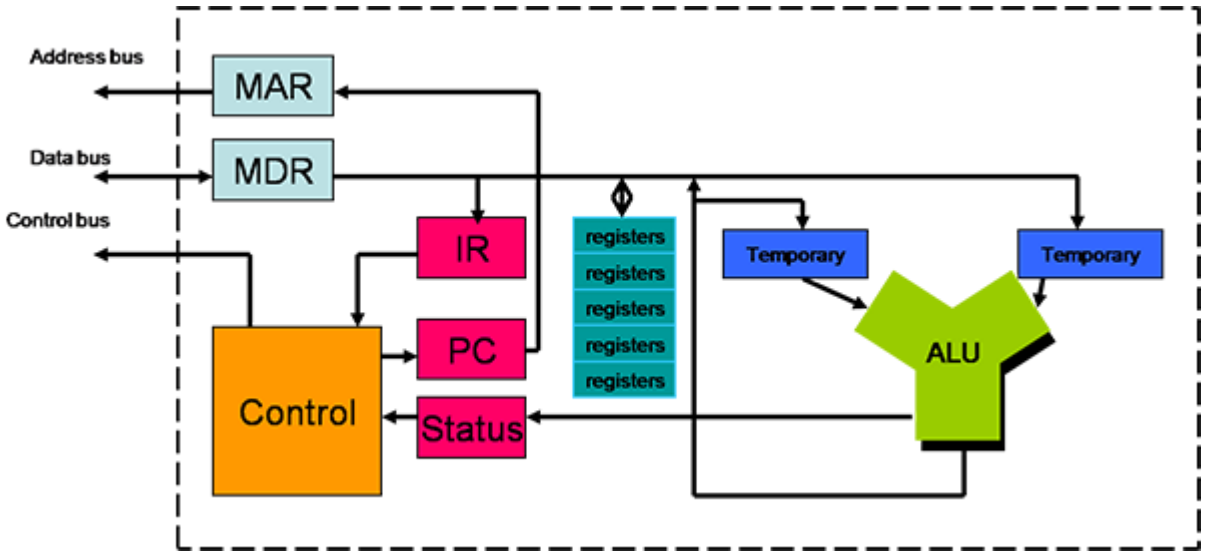
Registers serve as internal storage cells
hold operands and results for the instructions



# Multi-register Systems

Control unit generates signals that direct operations
For example, which registers to use, what ALU operation to perform, etc.

The ALU actually performs the operations
(such as addition, multiplication, shifting, etc.)



# Multi-register Systems

Multi-register designs allow explicit use of one or more register operands

Some systems have instructions that use 1 memory operand and 1 or more registers: (for example the Intel x86 systems and their clones)

```
add    eax,m        # add the 32-bit variable m to the 32-bit eax register
mov    ebx,data1    # copy the contents of the variable data1 into the ebx register
```

These are sometimes called one and a half address instructions.

# Stack Architecture

Some classes of computers use mostly implicit stack operands
These are known as stack-based machines
they use zero-address instructions

the instructions make no explicit reference to operands

Examples of such instructions are:

```
add       # pop the top two stack elements, add them and push the sum onto the stack
dup       # make a copy of the top element (i.e., duplicate it)
sub       # pop the top two elements and push the difference back onto the stack
pop   x   # remove the top element and store it into memory variable x
push  5   # push the constant 5 onto the stack
```

these instructions incur a penalty for each operand access
unless the stack is implemented using registers

# Load/Store Architecture

MIPS machines are reduced instruction set computers (RISC)
- require that most instructions use only registers operands
- only the load and store instructions use memory operands
- That is, they employ a load/store architecture

An example of a MIPS instruction that uses 3 register operands:

```
sub    $8,$9,$10    # subtract register $10 from $9 and put the difference in $8
```

# RISC Properties

- Instruction sets that only included simple straight forward instructions
- Simple addressing modes
- Fixed size machine instructions that can be fetched with one memory access
- Instruction pipelines that overlap the execution of instructions and complete one instruction per clock cycle

# RISC Properties

- Restricting the use of memory operands to only the load and store type instructions
- Hardwired logic implementation as opposed to microprogramming
- The use of optimizing compilers to generate the code to perform more complex tasks
- May have large instruction sets but the instructions are simple in nature

# RISC Disadvantages

Require multiple instructions to accomplish what CISC machines perform in a single complex instruction.

The compiler (or assembly language programmer) has to generate a sequence of simple instructions that perform the same actions of the individual CISC instructions.

Can result in code expansion which can be a problem.

# RISC Disadvantages

Code expansion refers to the increase in size that you get when you take a program that had been compiled for a CISC machine and re-compile it for a RISC machine

The amount of expansion depends primarily on the compiler
- higher quality compilers generate less code
- the nature of the machine's instruction set has an effect

Typically, code expansion can range up to 100% or more

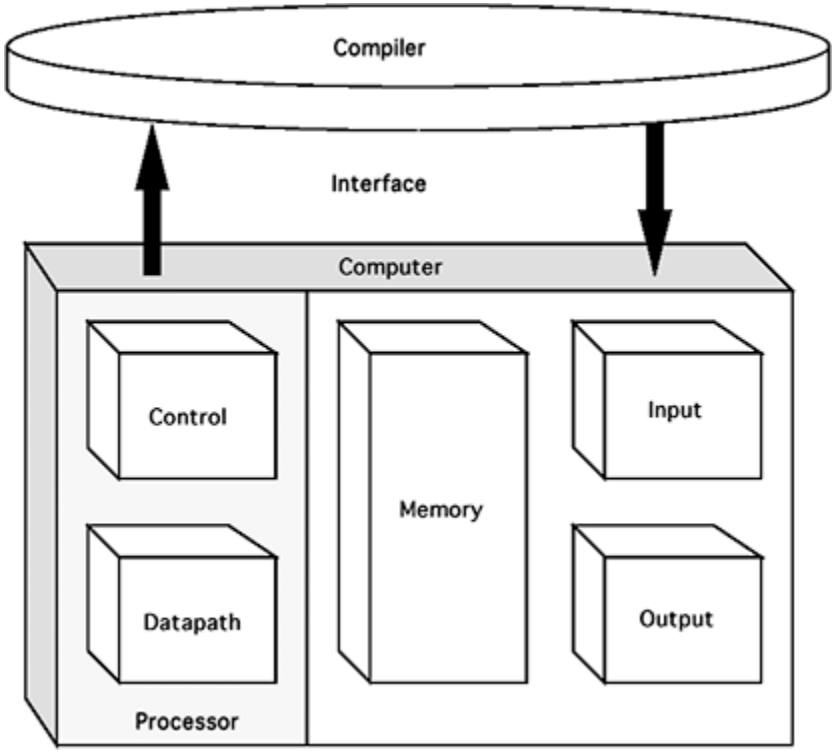
# Example RISC Machines

- MIPS
- Sparc
- ARM
- PowerPC

The MIPS architecture will be explained in detail later in the course as a prime example of a RISC machine.

# Moving Complexity to the Compiler

much of the complexity is handled by the compiler not the hardware




optimizations such as  instruction rearrangement take care of pipeline dependencies

# RISC Registers

The simpler nature of the RISC control unit leaves space (real estate) on the CPU chip

This extra space can be used to implement additional registers

RISC machines tend to have 32 or more registers

CISC machines may have 8 or fewer registers

Sparc processors have more than a hundred CPU registers
organized into logical groups called register windows

# RISC versus CISC Summary

| RISC | CISC |
|---|---|
| Simple instructions | Complex instructions |
| Completes one instruction per cycle | Requires many cycles to complete an instruction |
| Load/Store Architecture (only load and store instructions can reference memory) | Most instructions can reference memory |
| Relies heavily on pipelining | Relies less on pipelining |
| Instructions executed in hardware | Microcode interprets instructions |
| Fixed size instructions with a small number of formats | Variable size instructions with many formats |
| Simpler addressing modes | Many intricate addressing modes |