COMPUTER ORGANIZATION AND DESIGN

The Hardware/Software Interface

5th Edition

# Chapter 2

## The MIPS Processor and Instruction Set

## Chapter 1 Recap

- In my opinion, knowledge of hardware improves software quality – compilers, OS, threaded programs, memory management.
- Important trends to follow:
  - Transistor sizing.
  - Move to multi-core.
  - Slowing rate of performance improvement.
  - Power/thermal constraints.
  - Long memory/disk latencies.
- Reasoning about performance – clock speeds, CPI, benchmark suites, performance equations.

## Performance Equation Review

- Basic performance equations:

$$CPU\ time = (clock\ cycle\ time)(instruction\ count)(CPI)$$

or

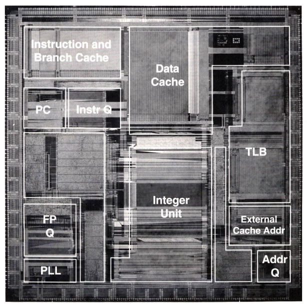
$$CPU\ time = \frac{(instruction\ count)(CPI)}{clock\ rate}$$

- These equations separate the key factors that affect performance:
  - The CPU execution time is measured by running the program.
  - The clock rate is usually given.
  - The overall instruction count is measured by using profilers or simulators.
  - CPI varies by instruction type and the instruction set architecture.

## The MIPS Processor

- Used as an example throughout the text book.
- Decent share of embedded core market:
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …
  - https://en.wikipedia.org/wiki/MIPS_architecture

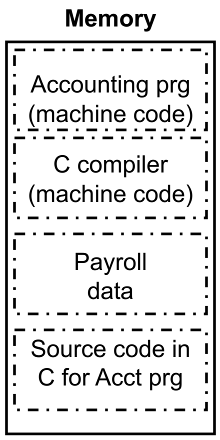
MIPS R8000 (TFP IU)



2.6 million transistors
17.2 × 17.3 mm
First silicon: May 1994

## Two Key Principles of Computer Design

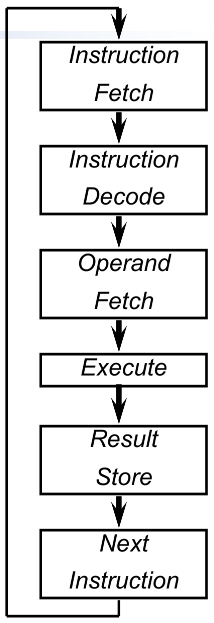
1. Instructions are represented as numbers and, as such, are indistinguishable from data.
2. Programs are stored in alterable memory (that can be read or written to) just like data.


Memory
Accounting prg (machine code)
C compiler (machine code)
Payroll data
Source code in C for Acct prg

- Stored-program concept
  - Programs can be shipped as files of binary numbers.
  - Computers can inherit current and legacy software provided they are compatible with an existing Instruction Set Architecture (ISA) – leads industry to align around a small number of ISA's.

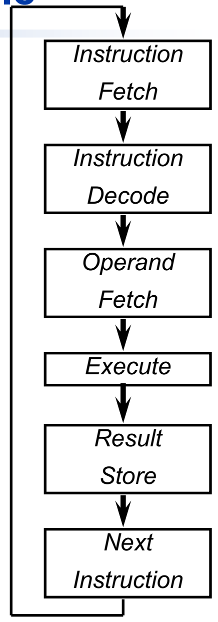
## Instruction Execution

- Get the instruction.
- Decide what kind of instruction it is.
- Get necessary data.
- Execute the instruction.
- Store the result.
- Repeat forever.

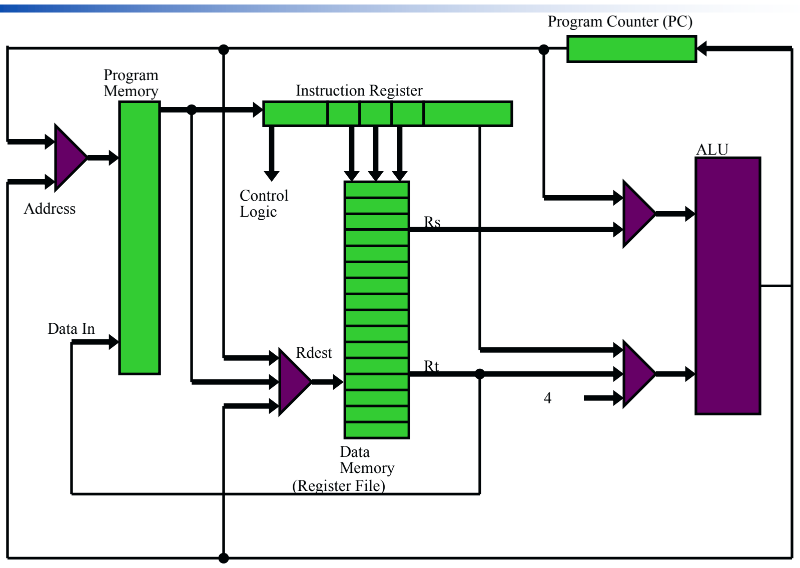


## Instruction Execution Example

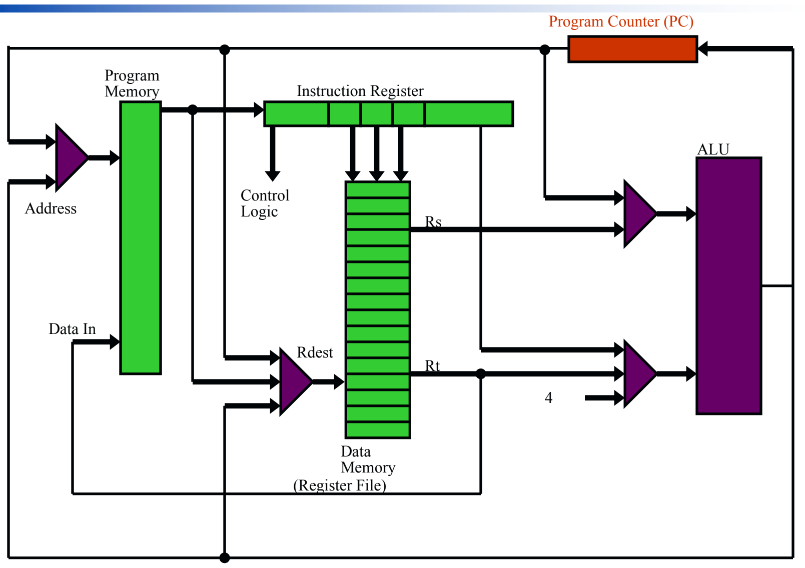
- C instruction: x = a+b;
- Assembly instruction: add a,b,x
  - Step 1: Fetch the instruction
  - Step 2: Determine it is an add instruction
  - Step 3: Fetch the data items a and b
  - Step 4: Do the addition
  - Step 5: Store the result in x
  - Step 6: Go to step 1



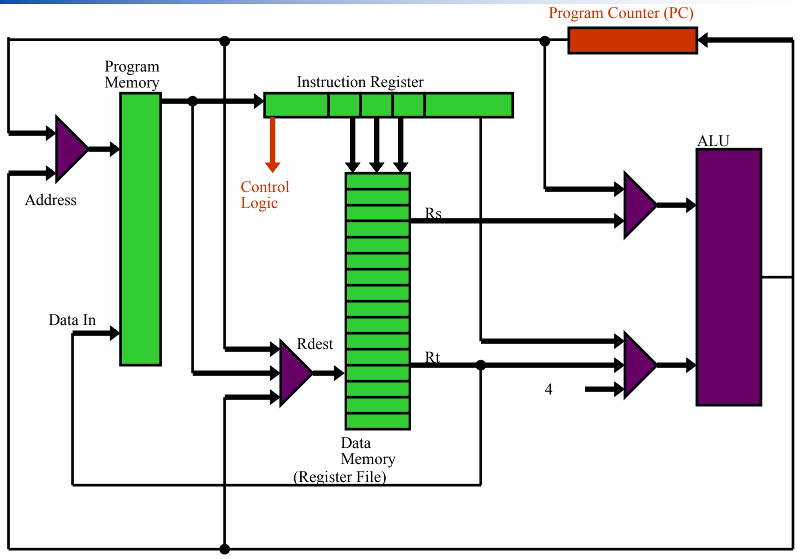
## Typical Fetch-Execute Processor Architecture



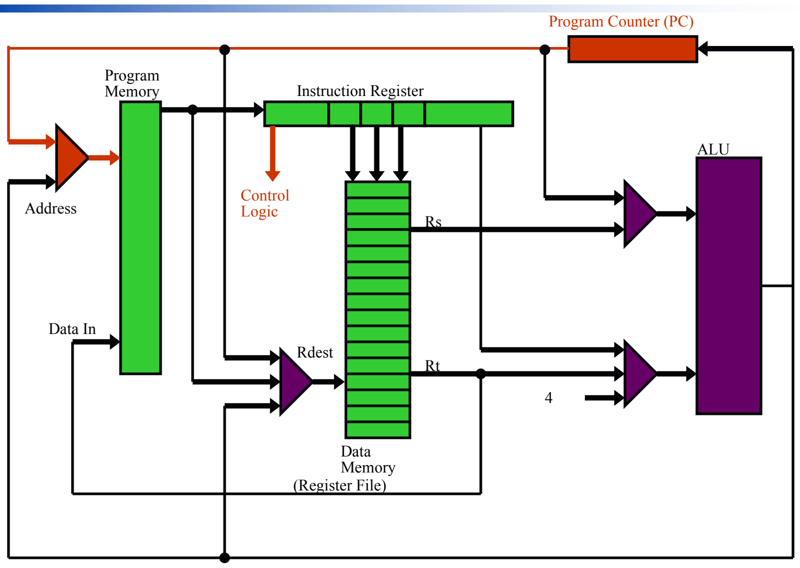
## Initialize Program Counter (PC) to Point to First Instruction



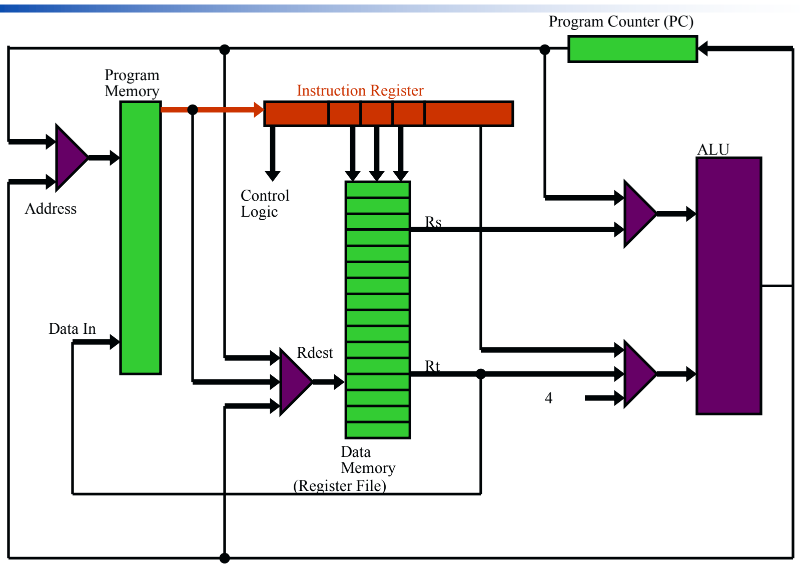
## Activate Control Logic

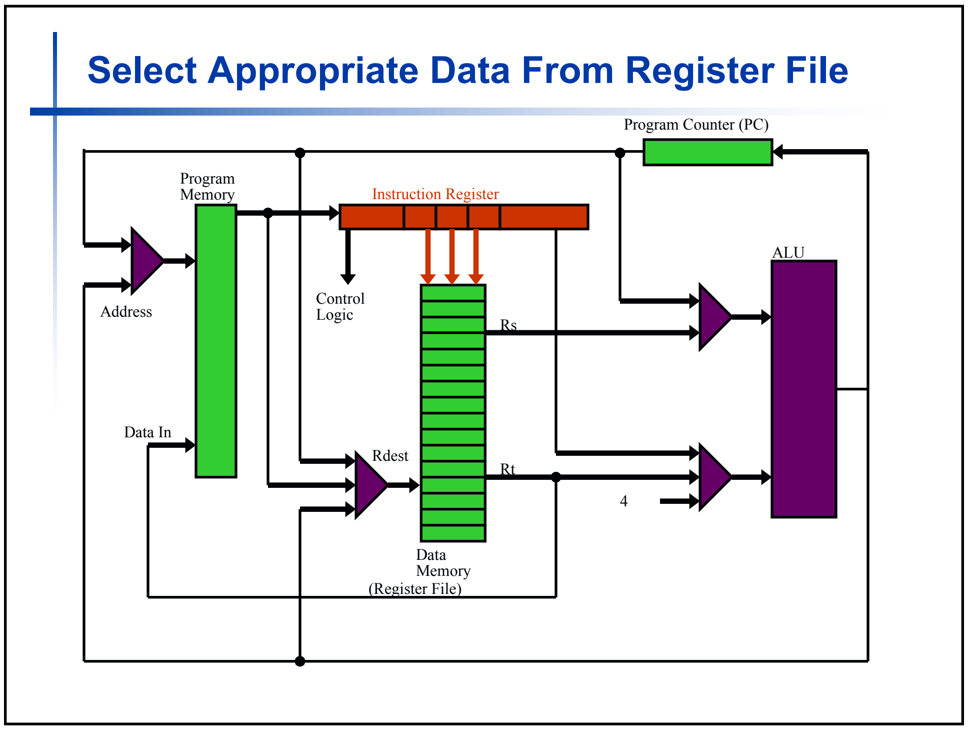


## Route Address to Program Memory

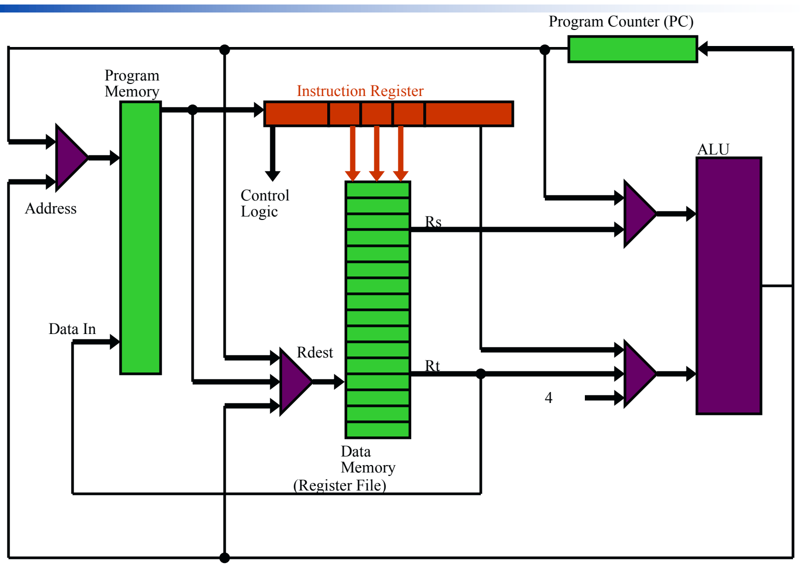


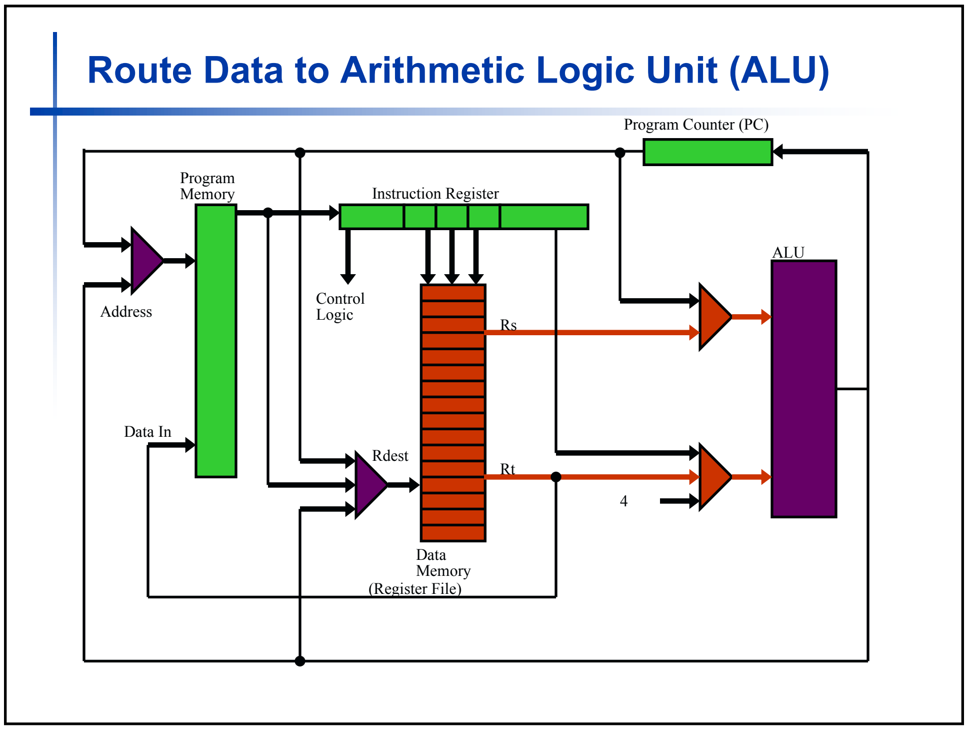
## Route Instruction to Instruction Register (IR)



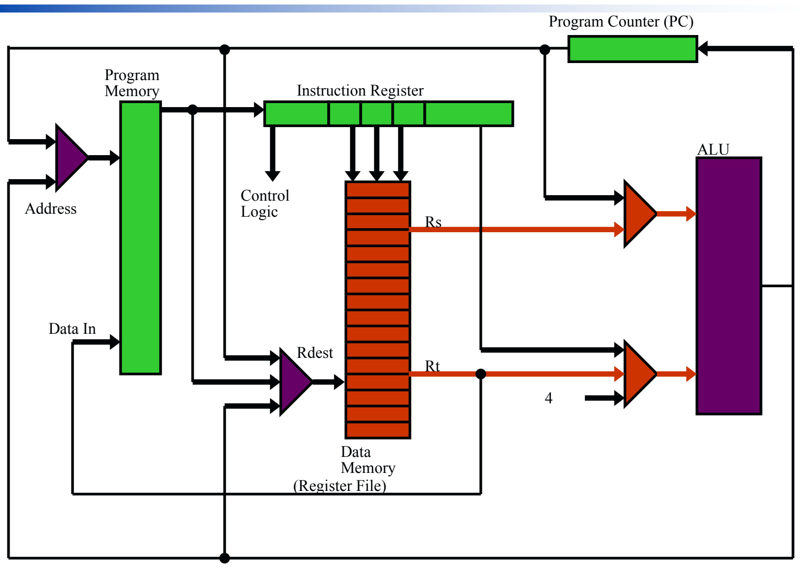


## Select Appropriate Data From Register File

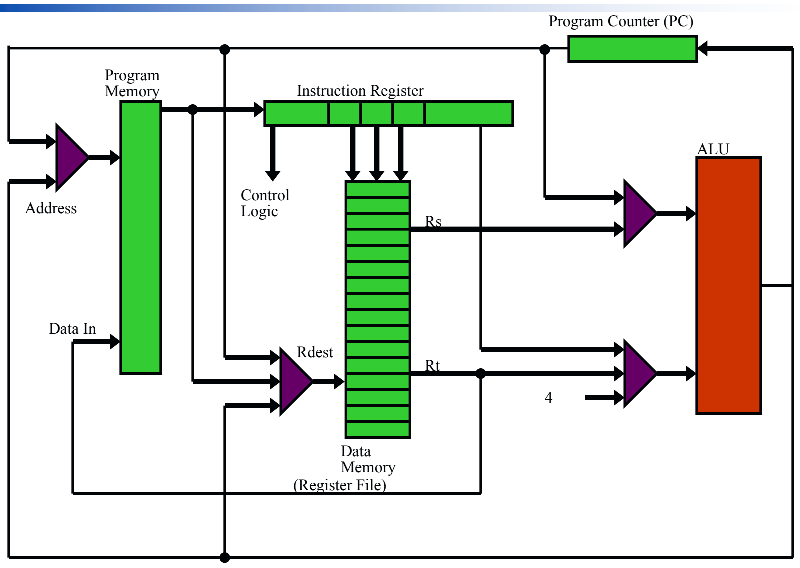




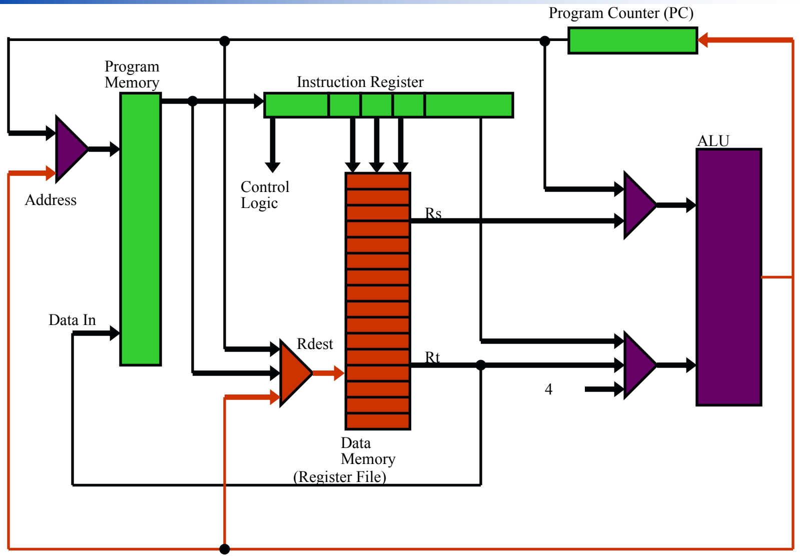
## Route Data to Arithmetic Logic Unit (ALU)



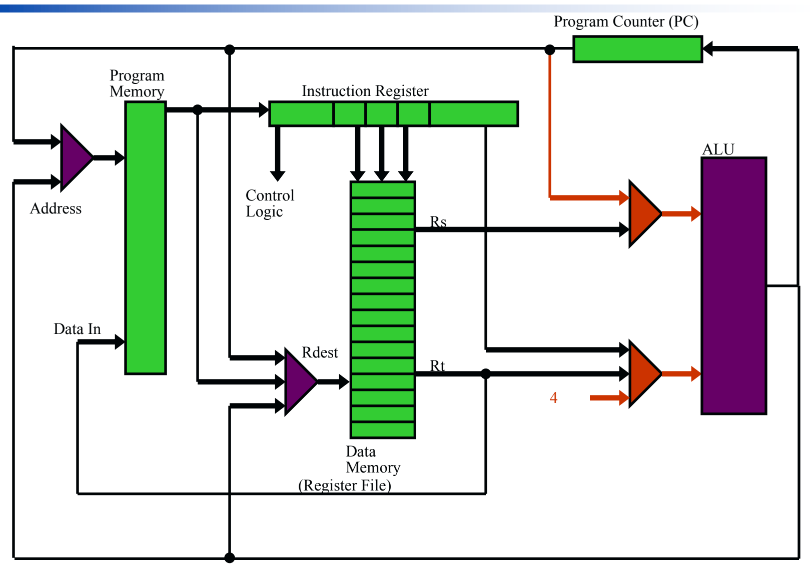
## Do the Computation



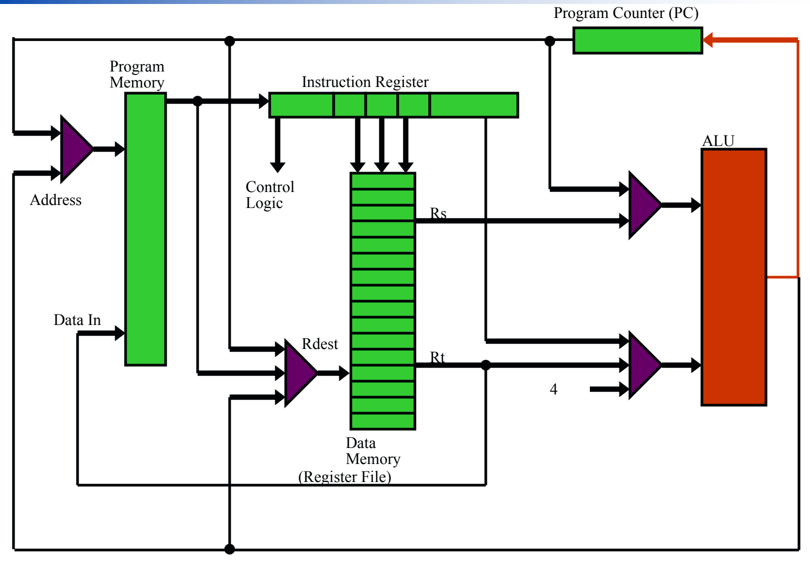
## Store the Result



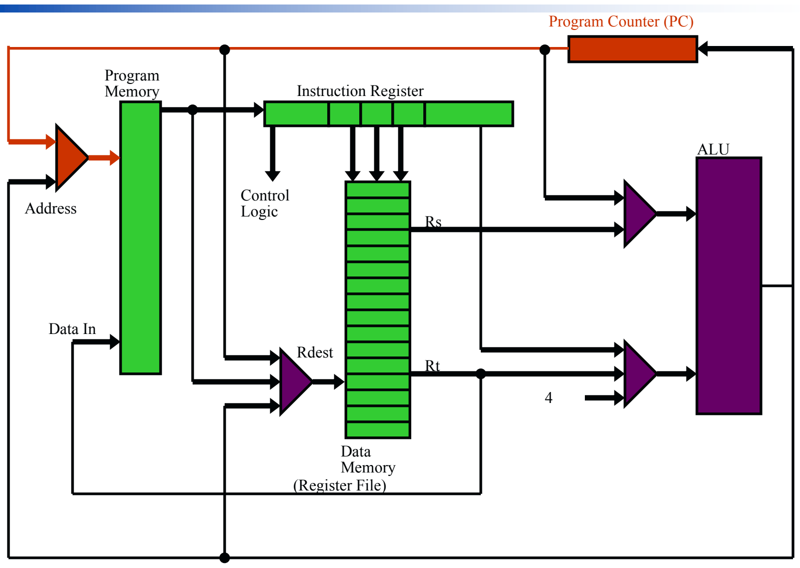
## Increment Program Counter to Point to Next Instruction



## Increment Program Counter to Point to Next Instruction



## Execute Next Instruction



## Instruction Set

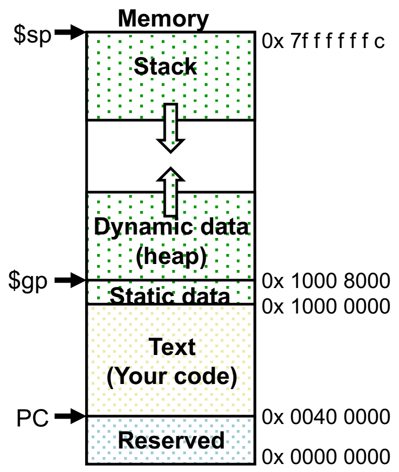
- The repertoire of instructions of a computer.
- Different computers have different instruction sets, but with many aspects in common.
- Important design principles when defining an ISA:
  - Keep the hardware simple – the CPU is most efficient when executing basic operations.
  - Keep the instruction formats regular – simplifies the decoding/scheduling of instructions.

## MIPS (RISC) Design Principles

- Simplicity favors regularity:
  - Fixed size instructions.
  - Small number of instruction formats.
  - Opcode always the first 6 bits in an instruction.
- Smaller is faster:
  - Limited instruction set.
  - Limited number of registers in the register file.
  - Limited number of addressing modes.
- Make the common case fast:
  - Arithmetic operands taken only from the register file.
  - Allow instructions to contain immediate operands.

## MIPS Memory Map

- The static data segment is for constants and other static variables.
- The dynamic data segment (*heap*) is for structures that grow and shrink (e.g., linked lists)
  - Allocate space on the heap with `malloc()` and free it with `free()` in C.



## MIPS Instruction Class Distribution

- Frequency of MIPS instruction classes for SPEC2006.

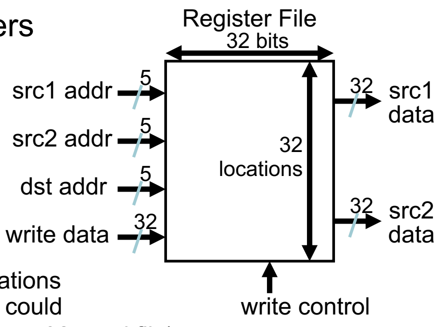
| **Instruction Class** | **Frequency** | |
|---|---|---|
| | Integer | Float Pt. |
| Arithmetic | 16% | 48% |
| Data transfer | 35% | 36% |
| Logical | 12% | 4% |
| Cond. Branch | 34% | 8% |
| Jump | 2% | 0% |

## MIPS Architecture

- The MIPS architecture is considered to be a typical RISC architecture:
  - Simplified instruction set => easier to study
- Programmable storage:
  - 32 x 32-bit General Purpose Registers (r0 = 0)
  - 32 x 32-bit Floating Point registers
  - Special purpose registers - HI, LO, PC
  - 2^32 bytes of addressable main memory
- Memory is byte addressable:
  - Words are 32 bits = 4 bytes
  - Words start at multiple of 4 address, referred to as *word-aligned.*

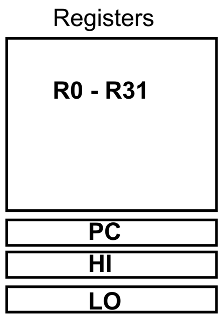
## MIPS Register File

- Holds thirty-two 32-bit registers
  - Two read ports and
  - One write port
- Registers
  - Are faster than main memory:
    - But register files with more locations are slower (e.g., a 64 word file could be as much as 50% slower than a 32 word file).
  - Are easier for a compiler to use:
    - e.g., (A*B) – (C*D) – (E*F) can do multiplies in any order vs. using data on a stack.
  - Can hold variables so that:
    - Code density improves (since registers are named with fewer bits than a memory location).



## MIPS-32 ISA

- Instruction Categories
  - Computational
  - Load/Store
  - Jump and Branch
  - Floating Point



**3 Instruction Formats: all 32 bits wide**

| | | | | | | |
|---|---|---|---|---|---|---|
| **op** | **rs** | **rt** | **rd** | **sa** | **funct** | R format |
| **op** | **rs** | **rt** | **immediate** | | | I format |
| **op** | **jump target** | | | | | J format |

## MIPS Instruction Fields

- MIPS fields are given names to make them easier to refer to. For R-type instructions:

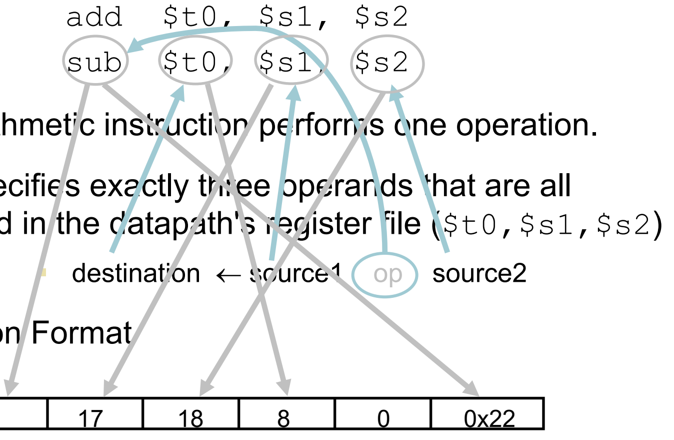
| | | | | | |
|---|---|---|---|---|---|
| op | rs | rt | rd | shamt | funct |

| | | |
|---|---|---|
| op | 6-bits | opcode that specifies the operation |
| rs | 5-bits | register file address of the first source operand |
| rt | 5-bits | register file address of the second source operand |
| rd | 5-bits | register file address of the result’s destination |
| shamt | 5-bits | shift amount (for shift instructions) |
| funct | 6-bits | function code augmenting the opcode |

## MIPS R-type Instructions

- MIPS assembly language arithmetic instructions:

```
add  $t0, $s1, $s2
```



- Each arithmetic instruction performs one operation.
- Each specifies exactly three operands that are all contained in the datapath's register file (`$t0,$s1,$s2`)
  - destination ← source1 op source2
- Instruction Format

| | | | | | |
|---|---|---|---|---|---|
| 0 | 17 | 18 | 8 | 0 | 0x22 |

## MIPS Data Types and Literals

- Data types:
  - Byte, halfword (2 bytes), word (4 bytes).
  - A character requires 1 byte of storage (stored as ASCII).
  - An integer requires 1 word (4 bytes) of storage.
- Literals:
  - Numbers entered as is: <u>example</u> 4
  - Characters enclosed in single quotes: <u>example</u> 'b'
  - Strings enclosed in double quotes: <u>example</u> "A string"

## Example: MIPS Add Instruction

- C code: a = b + c; # (friendly to programmers)
- Assembly code: add a, b, c # (friendly to ???)
- Machine code: 00000010001100100100000000100000 (friendly to hardware)

## Code Example

- C code: a = b + c + d + e;
- Assembly code:

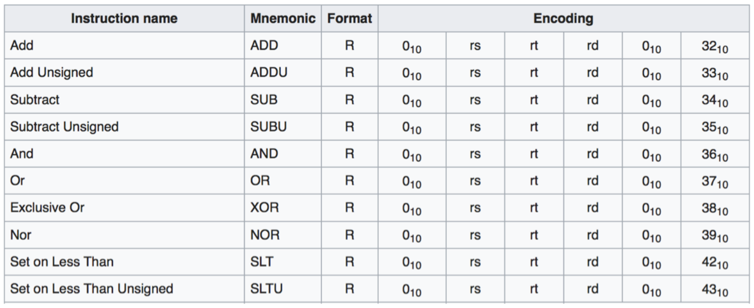
```
add  a, b, c               add  a, b, c
add  a, a, d       or      add  f, d, e
add  a, a, e               add  a, a, f
```

- Assembly instructions are simple – fixed number of operands (unlike C).
- Some sequences are better than others … the second sequence needs one more temporary variable f.

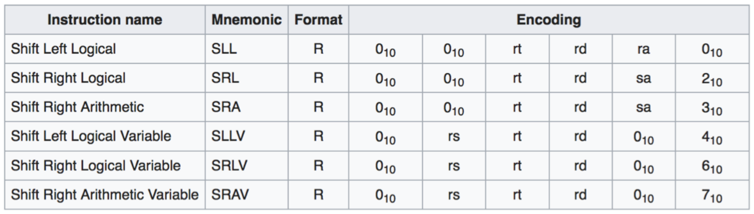
## Operands

- In C, each "variable" is a location in memory.
- In hardware, each memory access is expensive in terms of time – if variable ***a*** is accessed repeatedly, it increases performance if the variable is stored nearby, like an on-chip scratchpad (register file). Thank of it as *L0 cache*.
- To simplify instructions, MIPS requires that every R-type instruction (add, sub, etc.) operates only on register data. This was a significant departure from Complex Instruction Set Machines (CISC).
- The number of operands in a C program can be very large; the number of operands in assembly is fixed … there can be only so many scratchpad registers.

## MIPS R-type Instructions



| Instruction name | Mnemonic | Format | Encoding | | | | | |
|---|---|---|---|---|---|---|---|---|
| Add | ADD | R | $0_{10}$ | rs | rt | rd | $0_{10}$ | $32_{10}$ |
| Add Unsigned | ADDU | R | $0_{10}$ | rs | rt | rd | $0_{10}$ | $33_{10}$ |
| Subtract | SUB | R | $0_{10}$ | rs | rt | rd | $0_{10}$ | $34_{10}$ |
| Subtract Unsigned | SUBU | R | $0_{10}$ | rs | rt | rd | $0_{10}$ | $35_{10}$ |
| And | AND | R | $0_{10}$ | rs | rt | rd | $0_{10}$ | $36_{10}$ |
| Or | OR | R | $0_{10}$ | rs | rt | rd | $0_{10}$ | $37_{10}$ |
| Exclusive Or | XOR | R | $0_{10}$ | rs | rt | rd | $0_{10}$ | $38_{10}$ |
| Nor | NOR | R | $0_{10}$ | rs | rt | rd | $0_{10}$ | $39_{10}$ |
| Set on Less Than | SLT | R | $0_{10}$ | rs | rt | rd | $0_{10}$ | $42_{10}$ |
| Set on Less Than Unsigned | SLTU | R | $0_{10}$ | rs | rt | rd | $0_{10}$ | $43_{10}$ |

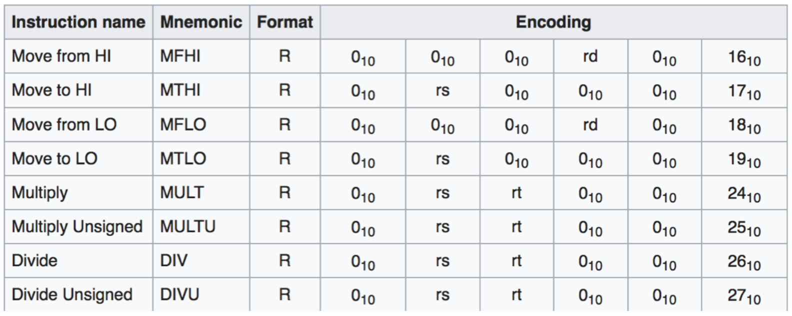


| Instruction name | Mnemonic | Format | Encoding | | | | | |
|---|---|---|---|---|---|---|---|---|
| Shift Left Logical | SLL | R | $0_{10}$ | $0_{10}$ | rt | rd | ra | $0_{10}$ |
| Shift Right Logical | SRL | R | $0_{10}$ | $0_{10}$ | rt | rd | sa | $2_{10}$ |
| Shift Right Arithmetic | SRA | R | $0_{10}$ | $0_{10}$ | rt | rd | sa | $3_{10}$ |
| Shift Left Logical Variable | SLLV | R | $0_{10}$ | rs | rt | rd | $0_{10}$ | $4_{10}$ |
| Shift Right Logical Variable | SRLV | R | $0_{10}$ | rs | rt | rd | $0_{10}$ | $6_{10}$ |
| Shift Right Arithmetic Variable | SRAV | R | $0_{10}$ | rs | rt | rd | $0_{10}$ | $7_{10}$ |

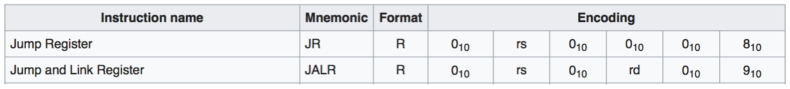
## MIPS Bit-wise Logical Operations

| **Logical ops** | **C operators** | **Java operators** | **MIPS instr** |
|---|---|---|---|
| Shift Left | << | << | sll |
| Shift Right | >> | >>> | srl |
| Bit-by-bit AND | & | & | and, andi |
| Bit-by-bit OR | \| | \| | or, ori |
| Bit-by-bit NOT | ~ | ~ | nor |

## MIPS R-type Instructions



| Instruction name | Mnemonic | Format | Encoding | | | | | |
|---|---|---|---|---|---|---|---|---|
| Move from HI | MFHI | R | $0_{10}$ | $0_{10}$ | $0_{10}$ | rd | $0_{10}$ | $16_{10}$ |
| Move to HI | MTHI | R | $0_{10}$ | rs | $0_{10}$ | $0_{10}$ | $0_{10}$ | $17_{10}$ |
| Move from LO | MFLO | R | $0_{10}$ | $0_{10}$ | $0_{10}$ | rd | $0_{10}$ | $18_{10}$ |
| Move to LO | MTLO | R | $0_{10}$ | rs | $0_{10}$ | $0_{10}$ | $0_{10}$ | $19_{10}$ |
| Multiply | MULT | R | $0_{10}$ | rs | rt | $0_{10}$ | $0_{10}$ | $24_{10}$ |
| Multiply Unsigned | MULTU | R | $0_{10}$ | rs | rt | $0_{10}$ | $0_{10}$ | $25_{10}$ |
| Divide | DIV | R | $0_{10}$ | rs | rt | $0_{10}$ | $0_{10}$ | $26_{10}$ |
| Divide Unsigned | DIVU | R | $0_{10}$ | rs | rt | $0_{10}$ | $0_{10}$ | $27_{10}$ |



| Instruction name | Mnemonic | Format | Encoding | | | | | |
|---|---|---|---|---|---|---|---|---|
| Jump Register | JR | R | $0_{10}$ | rs | $0_{10}$ | $0_{10}$ | $0_{10}$ | $8_{10}$ |
| Jump and Link Register | JALR | R | $0_{10}$ | rs | $0_{10}$ | rd | $0_{10}$ | $9_{10}$ |

## MIPS Recap

- MIPS: typical of RISC ISAs
  - Keep it simple.
  - Keep it small.
  - Make the common case fast.
- MIPS Architecture
- MIPS Instruction set
  - R-type
- Next class period: Immediate type instructions