

## Appendix B

### Instruction Set Principles and Examples

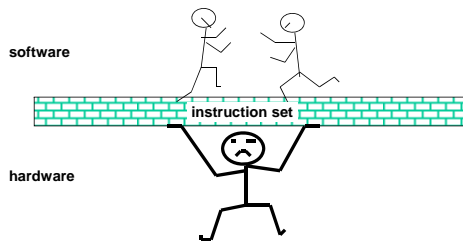
1

## Computer Architecture's Changing Definition

- 1950s to 1960s:  
Computer Architecture Course = Computer Arithmetic
- 1970s to mid 1980s:  
Computer Architecture Course = Instruction Set Design, especially ISA appropriate for compilers
- 1990s:  
Computer Architecture Course = Design of CPU, memory system, I/O system, Multiprocessors

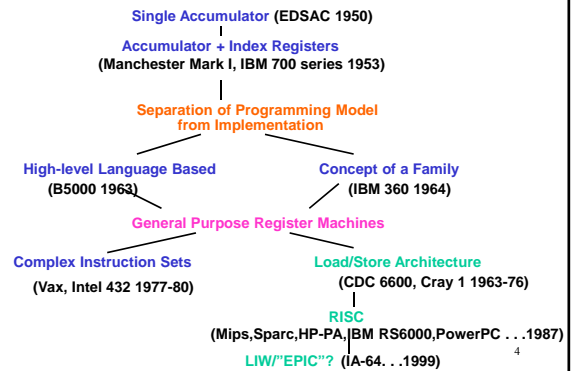
2

## Instruction Set Architecture (ISA)



3

## Evolution of Instruction Sets



4

## Instructions Can Be Divided into 3 Classes (I)

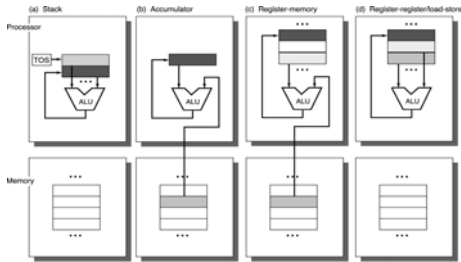
- Data movement instructions
  - Move data from a memory location or register to another memory location or register without changing its form
  - **Load**—source is memory and destination is register
  - **Store**—source is register and destination is memory
- Arithmetic and logic (ALU) instructions
  - Change the form of one or more operands to produce a result stored in another location
  - **Add, Sub, Shift**, etc.
- Branch instructions (control flow instructions)
  - Alter the normal flow of control from executing the next instruction in sequence
  - **Br Loc, Brz, Loc2**—unconditional or conditional branches

## Classifying ISAs

- Accumulator (before 1960):  
1 address    add A    acc ← acc + mem[A]
- Stack (1960s to 1970s):  
0 address    add    tos ← tos + next
- Memory-Memory (1970s to 1980s):  
2 address    add A, B    mem[A] ← mem[A] + mem[B]  
3 address    add A, B, C    mem[A] ← mem[B] + mem[C]
- Register-Memory (1970s to present):  
2 address    add R1, A    R1 ← R1 + mem[A]  
              load R1, A    R1 ← mem[A]
- Register-Register (Load/Store) (1960s to present):  
3 address    add R1, R2, R3    R1 ← R2 + R3  
              load R1, R2    R1 ← mem[R2]  
              store R1, R2    mem[R1] ← R2

6

## Classifying ISAs



© 2003 Elsevier Science (USA). All rights reserved.

7

## Load-Store Architectures

- **Instruction set:**

```
add R1, R2, R3    sub R1, R2, R3    mul R1, R2, R3
load R1, R4       store R1, R4
```
- **Example:  $A*B - (A+C*B)$** 

```
load R1, &A
load R2, &B
load R3, &C
load R4, R1
load R5, R2
load R6, R3
mul R7, R6, R5    /*    C*B    */
add R8, R7, R4    /*    A + C*B  */
mul R9, R4, R5    /*    A*B    */
sub R10, R9, R8   /*    A*B - (A+C*B) */
```

8

## Load-Store: Pros and Cons

- **Pros**
  - Simple, fixed length instruction encoding
  - Instructions take similar number of cycles
  - Relatively easy to pipeline
- **Cons**
  - Higher instruction count
  - Not all instructions need three operands
  - Dependent on good compiler

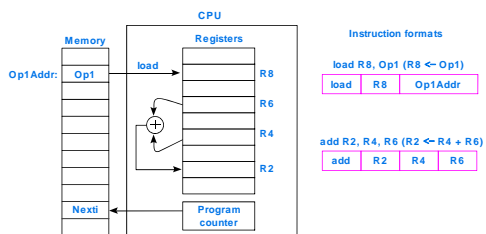
9

## Registers: Advantages and Disadvantages

- **Advantages**
  - Faster than cache (no addressing mode or tags)
  - Deterministic (no misses)
  - Can replicate (multiple read ports)
  - Short identifier (typically 3 to 8 bits)
  - Reduce memory traffic
- **Disadvantages**
  - Need to save and restore on procedure calls and context switch
  - Can't take the address of a register (for pointers)
  - Fixed size (can't store strings or structures efficiently)
  - Compiler must manage

10

## General Register Machine and Instruction Formats



11

## General Register Machine and Instruction Formats

- It is the most common choice in today's general-purpose computers
- Which register is specified by small "address" (3 to 6 bits for 8 to 64 registers)
- Load and store have one long & one short address: One and half addresses
- Arithmetic instruction has 3 "half" addresses

12

## Real Machines Are Not So Simple

- Most real machines have a mixture of 3, 2, 1, 0, and 1- address instructions
- A distinction can be made on whether arithmetic instructions use data from memory
- If ALU instructions only use registers for operands and result, machine type is **load-store**
  - Only load and store instructions reference memory
- Other machines have a mix of register-memory and memory-memory instructions

13

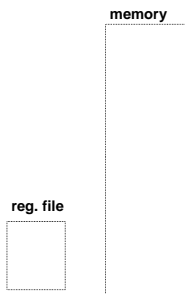
## Alignment Issues

- If the architecture does not restrict memory accesses to be aligned then
  - Software is simple
  - Hardware must detect misalignment and make 2 memory accesses
  - Expensive detection logic is required
  - All references can be made slower
- Sometimes unrestricted alignment is required for backwards compatibility
- If the architecture restricts memory accesses to be aligned then
  - Software must guarantee alignment
  - Hardware detects misalignment access and traps
  - No extra time is spent when data is aligned
- Since we want to make the common case fast, having restricted alignment is often a better choice, unless compatibility is an issue

14

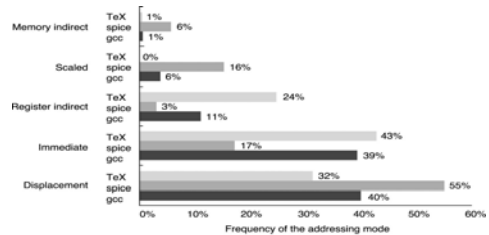
## Types of Addressing Modes (VAX)

1. Register direct  $R_i$
2. Immediate (literal)  $\#n$
3. Displacement  $M[R_i + \#n]$
4. Register indirect  $M[R_i]$
5. Indexed  $M[R_i + R_j]$
6. Direct (absolute)  $M[\#n]$
7. Memory Indirect  $M[M[R_i]]$
8. Autoincrement  $M[R_i++]$
9. Autodecrement  $M[R_i--]$
10. Scaled  $M[R_i + R_j * d + \#n]$



15

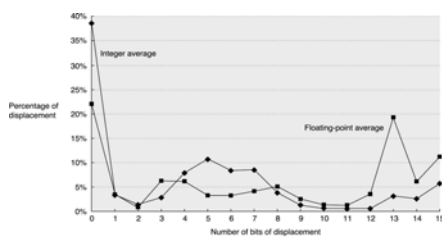
## Summary of Use of Addressing Modes



© 2003 Elsevier Science (USA). All rights reserved.

16

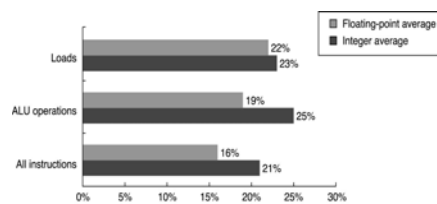
## Distribution of Displacement Values



© 2003 Elsevier Science (USA). All rights reserved.

17

## Frequency of Immediate Operands



© 2003 Elsevier Science (USA). All rights reserved.

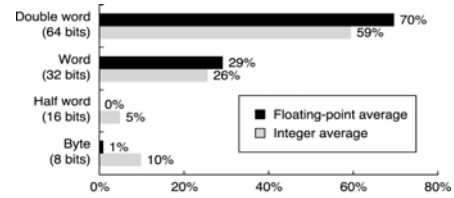
18

## Types of Operations

- Arithmetic and Logic: AND, ADD
- Data Transfer: MOVE, LOAD, STORE
- Control: BRANCH, JUMP, CALL
- System: OS CALL, VM
- Floating Point: ADDF, MULF, DIVF
- Decimal: ADDD, CONVERT
- String: MOVE, COMPARE
- Graphics: (DE)COMPRESS

19

## Distribution of Data Accesses by Size



© 2003 Elsevier Science (USA). All rights reserved.

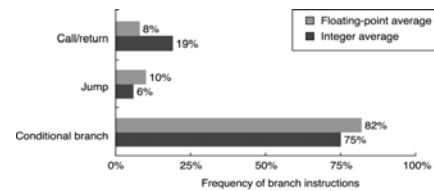
20

## 80x86 Instruction Frequency (SPECint92, Fig. B.13)

Rank	Instruction	Frequency
1	load	22%
2	branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	register move	4%
9	call	1%
10	return	1%
<b>Total</b>		<b>96%</b>

21

## Relative Frequency of Control Instructions



© 2003 Elsevier Science (USA). All rights reserved.

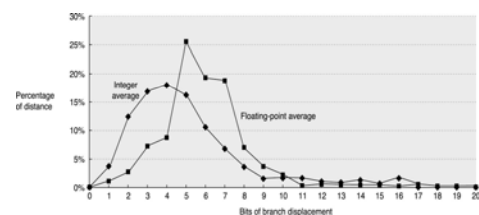
22

## Control instructions (cont'd)

- Addressing modes
  - PC-relative addressing (independent of program load & displacements are close by)
    - Requires displacement (how many bits?)
    - Determined via empirical study. [8-16 works!]
  - For procedure returns/indirect jumps/kernel traps, target may not be known at compile time.
    - Jump based on contents of register
    - Useful for switch/(virtual) functions/function ptrs/dynamically linked libraries etc.

23

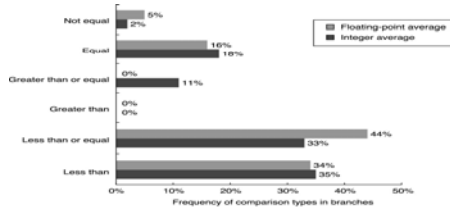
## Branch Distances (in terms of number of instructions)



© 2003 Elsevier Science (USA). All rights reserved.

24

## Frequency of Different Types of Compares in Conditional Branches



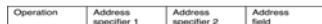
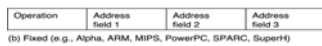
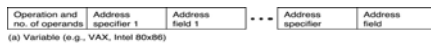
© 2003 Elsevier Science (USA). All rights reserved. 25

## Encoding an Instruction set

- a desire to have as many registers and addressing mode as possible
- the impact of size of register and addressing mode fields on the average instruction size and hence on the average program size
- a desire to have instruction encode into lengths that will be easy to handle in the implementation

26

## Three choice for encoding the instruction set



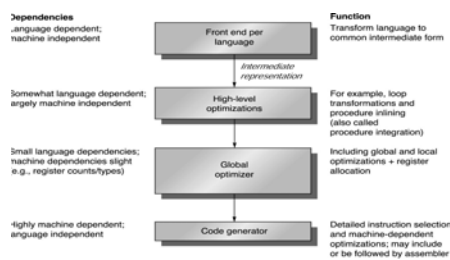
© 2003 Elsevier Science (USA). All rights reserved. 27

## Compilers and ISA

- Compiler Goals
  - All correct programs compile correctly
  - Most compiled programs execute quickly
  - Most programs compile quickly
  - Achieve small code size
  - Provide debugging support
- Multiple Source Compilers
  - Same compiler can compile different languages
- Multiple Target Compilers
  - Same compiler can generate code for different machines

28

## Compilers Phases



© 2003 Elsevier Science (USA). All rights reserved. 29

## Compiler Based Register Optimization

- Assume small number of registers (16-32)
- Optimizing use is up to compiler
- HLL programs have no explicit references to registers
  - usually – is this always true?
- Assign symbolic or virtual register to each candidate variable
- Map (unlimited) symbolic registers to real registers
- Symbolic registers that do not overlap can share real registers
- If you run out of real registers some variables use memory
- Uses graph coloring approach

30

## Designing ISA to Improve Compilation

- Provide enough general purpose registers to ease register allocation ( more than 16).
- Provide regular instruction sets by keeping the operations, data types, and addressing modes orthogonal.
- Provide primitive constructs rather than trying to map to a high-level language.
- Simplify trade-off among alternatives.
- Allow compilers to help make the common case fast.

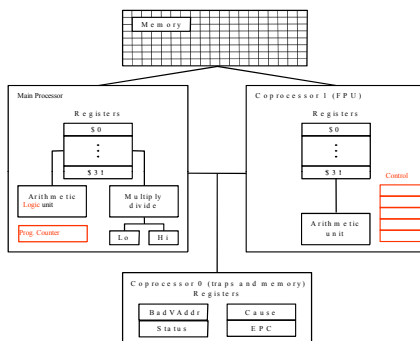
31

## ISA Metrics

- Orthogonality
  - No special registers, few special cases, all operand modes available with any data type or instruction type
- Completeness
  - Support for a wide range of operations and target applications
- Regularity
  - No overloading for the meanings of instruction fields
- Streamlined Design
  - Resource needs easily determined. Simplify tradeoffs.
- Ease of compilation (programming?), Ease of implementation, Scalability

32

## MIPS Processor



33

## MIPS Registers

- **Main Processor (integer manipulations):**
  - 32 64-bit general purpose registers – GPRs ( $R_0 - R_{31}$ );  $R_0$  has fixed value of zero. Attempt to writing into  $R_0$  is not illegal, but its value will not change;
  - two 64-bit registers – Hi & Lo, hold results of integer multiply and divide
  - 64-bit program counter – PC;
- **Coprocessor 1 (Floating Point Processor – real numbers manipulations):**
  - 32 64-bit floating point registers – FPRs ( $f_0 - f_{31}$ );
  - five control registers;
- **Coprocessor 0 – CP0** is incorporated on the MIPS CPU chip and it provides functions necessary to support operating system: exception handling, memory management scheduling and control of critical resources.

34

## MIPS Registers (continued)

- **Coprocessor 0 (CP0) registers (partial list):**
  - Status register (CP0reg12) – processor status and control;
  - Cause register (CP0reg13) – cause of the most recent exception;
  - EPC register (CP0reg14) – program counter at the last exception;
  - BadVAddr register (CP0reg08) – the address for the most recent address related exception;
  - Count register (CP0reg09) – acts as a timer, incrementing at a constant rate that is a function of the pipeline clock;
  - Compare register (CP0reg11) – used in conjunction with Count register;
  - Performance Counter register (CP0reg25);

35

## MIPS Data Types

- MIPS64 operates on:
  - 64-bit (unsigned or 2's complement) integers,
  - 32-bit (single precision floating point) real numbers,
  - 64-bit (double precision floating point) real numbers;
- 8-bit bytes, 16-bit half words and 32-bit words loaded into GPRs are either zero or sign bit expanded to fill the 64 bits.
- only 32- or 64-bit real numbers can be loaded into FPRs.
- 32-bit real number loaded into FPRs is zero-appended.

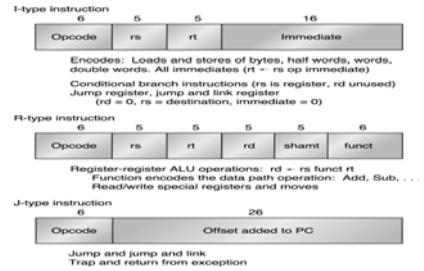
36

## MIPS Addressing Modes

- register addressing;
- immediate addressing;
- register indexed is the only memory data addressing; (in MIPS terminology called **base addressing**):
  - memory address = register content plus 16-bit offset
- since  $R_0$  always contains value 0:
  - $R_0 + 16\text{-bit offset} \rightarrow$  **absolute addressing**;
  - 16-bit offset = 0  $\rightarrow$  **register indirect**;
- branch instructions use **PC relative addressing**:
  - branch address =  $[PC] + 4 + 4 \times 16\text{-bit offset}$
- jump instructions use:
  - **pseudo-direct addressing** with 28-bit addresses (jumps inside 256MB regions),
  - **direct (absolute) addressing** with 64-bit addresses.

37

## Instruction Layout for MIPS



© 2003 Elsevier Science (USA). All rights reserved.

38

## MIPS Alignment

- MIPS supports **byte addressability**:
  - it means that a byte is the smallest unit with its own address;
- MIPS restricts memory accesses to be aligned as follows:
  - 64-bit word has to start at byte address which is multiple of 8; thus, 64-bit word at address  $8x$  includes eight bytes with addresses  $8x, 8x+1, 8x+2, \dots, 8x+6, 8x+7$ .
  - 32-bit word has to start at byte address that is multiple of 4; thus, 32-bit word at address  $4n$  includes four bytes with addresses:  $4n, 4n+1, 4n+2$ , and  $4n+3$ .
  - 16-bit half word has to start at byte address that is multiple of 2; thus, 16-bit word at address  $2n$  includes two bytes with addresses:  $2n$  and  $2n+1$ .
- MIPS supports **64-bit addresses**:
  - it means that an address is given as 64-bit unsigned integer;

39

## MIPS Instruction

- **Instructions that move data**:
  - load to register from memory (only base addressing),
  - store from register to memory (only base addressing),
  - move between registers in same and different coprocessors.
- ALU integer instructions; register – register and register-immediate computational instructions.
- Floating point instructions; register – register computational instructions and floating point to/from integer conversions.
- Control-related instruction:
  - (simple) branch instructions use PC relative addressing
  - jump instructions with 28-bit addresses (jumps inside 256MB regions), or absolute 64-bit addresses.
- Special control-related instructions.

40

## Load/Store Instructions

Example instruction	Instruction name	Meaning
$LD R1, 30(R2)$	Load double word	$Regs[R1] \leftarrow Mem[30+Regs[R2]]$
$LD R1, 1000(R0)$	Load double word	$Regs[R1] \leftarrow Mem[1000+0]$
$LR R1, 40(R2)$	Load word	$Regs[R1] \leftarrow Mem[40+Regs[R2]]_{31:0} \# Mem[40+Regs[R2]]$
$LB R1, 40(R3)$	Load byte	$Regs[R1] \leftarrow Mem[40+Regs[R3]]_{31:24} \# Mem[40+Regs[R3]]$
$LBU R1, 40(R3)$	Load byte unsigned	$Regs[R1] \leftarrow Mem[40+Regs[R3]]_{31:24} \# Mem[40+Regs[R3]]$
$LD R1, 40(R2)$	Load half word	$Regs[R1] \leftarrow Mem[40+Regs[R2]]_{31:16} \# Mem[40+Regs[R2]] \# Mem[41+Regs[R2]]$
$L.S F0, 50(R3)$	Load FP single	$Regs[F0] \leftarrow Mem[50+Regs[R3]] \# D^{32}$
$L.D F0, 50(R2)$	Load FP double	$Regs[F0] \leftarrow Mem[50+Regs[R2]]$
$SD R3, 500(R4)$	Store double word	$Mem[500+Regs[R4]] \leftarrow Regs[R3]$
$SW R3, 500(R4)$	Store word	$Mem[500+Regs[R4]]_{31:0} \leftarrow Regs[R3]$
$S.S F0, 40(R3)$	Store FP single	$Mem[40+Regs[R3]]_{31:0} \leftarrow Regs[F0]_{31:0}$
$S.D F0, 40(R3)$	Store FP double	$Mem[40+Regs[R3]] \leftarrow Regs[F0]$
$SH R3, 502(R2)$	Store half	$Mem[502+Regs[R2]]_{31:16} \leftarrow Regs[R3]_{31:16}$
$SB R2, 41(R3)$	Store byte	$Mem[41+Regs[R3]]_{31:24} \leftarrow Regs[R2]_{31:24}$

Figure B.23 The load and store instructions in MIPS. All use a single addressing mode and require that the memory value be aligned. Of course, both loads and stores are available for all the data types shown.

41

## Sample ALU Instructions

Example instruction	Instruction name	Meaning
$DADDU R1, R2, R3$	Add unsigned	$Regs[R1] \leftarrow Regs[R2] + Regs[R3]$
$DADDIU R1, R2, #3$	Add immediate unsigned	$Regs[R1] \leftarrow Regs[R2] + 3$
$LUI R1, #42$	Load upper immediate	$Regs[R1] \leftarrow 0^{22} \# \# 42 \# 0^{16}$
$DSLL R1, R2, #5$	Shift left logical	$Regs[R1] \leftarrow Regs[R2] \ll 5$
$DSLT R1, R2, R3$	Set less than	$\text{if } (Regs[R2] < Regs[R3])$ $Regs[R1] \leftarrow 1 \text{ else } Regs[R1] \leftarrow 0$

Figure B.24 Examples of arithmetic/logical instructions on MIPS, both with and without immediates.

42

