# Principle of Instruction Set Architecture

Jaynarayan T Tudu

Lecture 3, 3.1

IIT Tirupati, India

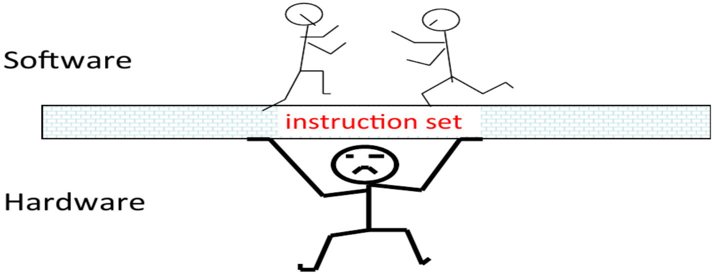
11th, 12th Feb, 2021

Computer System Architecture

## Review From Last Lectures

- Trends in Computer Architecture
- The Architecture and Design Parameters
- Measuring Performances: Benchmark and Metric
- Modeling of Energy and Dependability

# Instruction Set Architecture



- ISA is also known as programmer's model of machine.
- For a programmer or compiler designer the only thing visible is instruction set architecture.
- Different applications requires different set of instruction set.

| | | |
|---|---|---|
| Desktop system | File Server/Data Center | Mobile/Embedded application |
| Integer/FP operation | File Operation | Energy |

## Component of ISA

- Storage cell (the place where to keep the things)
  - General and special purpose registers in the CPU
  - Many general purpose cells of same size in memory
  - Storage associated with I/O devices
- The machine instruction set
  - The instruction set is the entire repertoire of machine operations
  - Makes use of storage cells, formats, and results of the fetch/execute cycle

## Component of ISA

- The instruction format
  - Size, field, and meaning of the field within the instruction
- Fetch and execute procedure
  - Things that are performed prior to knowing the instruction

## From C to Assembly view

- a 'C' programming language statement: $f = (g + h) - (i + j)$;
  - The set of instruction (called assembly instructions)
    add t0, g, h ; t0 $\leftarrow$ g + h
    add t1, i, j ; t1 $\leftarrow$ i + j
    sub f, t0, t1 ; f $\leftarrow$ t0 - t1
  - Opcode/mnemonic, operand (source and destination)

The instruction specifies the operation (and operand) to be performed

## What an Instruction Specifies

1. What operation to perform
   - Example: add r0, r1, r3
   - Arithmetic, logical etc.
2. Where to find operands
   - CPU registers
   - Memory cells
   - I/O location
   - Within instruction
3. Place to store the results
   - CPU registers
   - Memory cells
   - I/O location
4. Location of the next instruction
   - Memory location (pointed by a register called Program Counter)

| | |
|---|---|
| Operation | Operands |

There could be numerous ways to arrange and specify operands and operations!

## Classification of Instructions

Classification based on behavior:

- 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 Instructions
  - Change the form of one or more operands to produce a result stored in another location
- Control flow instructions
  - Alter the normal flow of control from executing the next instruction in sequence
  - Two type: conditional and unconditional

## Classification of Instructions

Classification of underlying architecture based on Internal storage:

- Stack Architecture
- Accumulator Architecture
- Register-Memory Architecture
- General Purpose Register Architecture (load-store)

## Classification of Instructions

Classification of Architecture based on Internal storage:

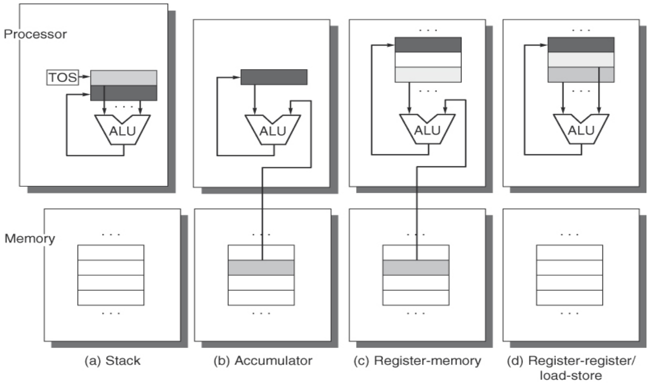


Figure : Operand location for diff class of architecure

## Classification of Instructions Set Architecture

| #Memory Address | Max # Oprnd | Arch type | Examples |
|---|---|---|---|
| 0 | 3 | Load-Store | Alpha, ARM, MIPS, PowerPC |
| 1 | 2 | Reg-Mem | IBM360, Intel x86, 68000 |
| 2 | 2 | Mem-Mem | VAX (DEC) |
| 3 | 3 | Mem-Mem | VAX (DEC) |

VAX: Virtual Address Extension architecure developed by Digital Equipement Corp in 1977.

## Classification of Instructions Set Architecture

- Register - Register
  - Advantages: Simple, fixed length encoding, simple code generation, all instr. Take same no. of cycles
  - Disadvantages: Higher instruction count, lower instruction density
- Register - Memory
  - Advantages: Data can be accessed without separate load instruction first, instruction format tend to be easy to encode and yield good density
  - Disadvantages: Encoding register no and memory address in each instruction may restrict the no. of registers.
- Memory- Memory
  - Advantages: Most compact, doesn't waste registers for temporaries
  - Disadvantages: Large variation in instruction size, large variation in in amount of work (NOT USED TODAY)

## Specifying Memory Address

- Interpreting memory address:
  - Big Endian
  - Little Endian
- Byte addressability and instruction misalignment
- Addressing mode

## Specifying Memory Address

Interpreting memory address: How do you order the bytes?

Example:

| Address | Bytes |
|---|---|
| $A_0$ | $B_0$ |
| $A_1$ | $B_1$ |
| $A_2$ | $B_2$ |
| $A_3$ | $B_3$ |
| $A_4$ | $B_4$ |
| $A_5$ | $B_5$ |
| $A_6$ | $B_6$ |
| $A_7$ | $B_7$ |

Ordering in word:

Little Endian:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $B_7$ | $B_6$ | $B_5$ | $B_4$ | $B_3$ | $B_2$ | $B_1$ | $B_0$ |

Big Endian:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $B_0$ | $B_1$ | $B_2$ | $B_3$ | $B_4$ | $B_5$ | $B_6$ | $B_7$ |

## Specifying Memory Address

Interpreting memory address: How do you order the bytes?

The other way to look at the problem:

Given to me a Word, how am I going to keep them in memory?

Word size: 64 bit (hypothetical computer)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $B_0$ | $B_1$ | $B_2$ | $B_3$ | $B_4$ | $B_5$ | $B_6$ | $B_7$ |

BigEndian Ordering

| Address | Bytes |
|---|---|
| $A_0$ | $B_0$ |
| $A_1$ | $B_1$ |
| $A_2$ | $B_2$ |
| $A_3$ | $B_3$ |
| $A_4$ | $B_4$ |
| $A_5$ | $B_5$ |
| $A_6$ | $B_6$ |
| $A_7$ | $B_7$ |

LittleEndian Ordering

| Address | Bytes |
|---|---|
| $A_0$ | $B_7$ |
| $A_1$ | $B_6$ |
| $A_2$ | $B_5$ |
| $A_3$ | $B_4$ |
| $A_4$ | $B_3$ |
| $A_5$ | $B_2$ |
| $A_6$ | $B_1$ |
| $A_7$ | $B_0$ |

## Specifying Memory Address

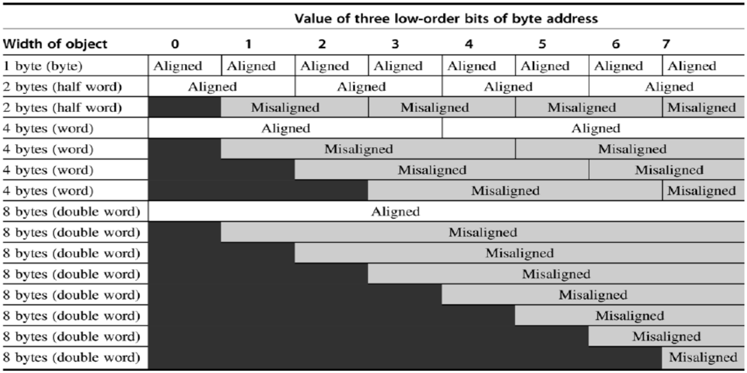
Interpreting memory address: How do you order the bytes?

Big Endian: A kind of natural ordering; bigger address byte at LSB!
Little Endian: A kind of reverse, smaller address byte at LSB!

- Does the order really matter in terms of performance or other parameters?
- Can we order the byte in order? (think from security point of view!)
- How does the exchange of information takes place between two machines: one with BigEnd and the other with LittleEnd?

# Specifying Memory Address

## Byte addressability and Alignment Issue



| | Value of three low-order bits of byte address | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Width of object | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 byte (byte) | Aligned | Aligned | Aligned | Aligned | Aligned | Aligned | Aligned | Aligned |
| 2 bytes (half word) | Aligned | | Aligned | | Aligned | | Aligned | |
| 2 bytes (half word) | ■ | Misaligned | | Misaligned | | Misaligned | | Misaligned |
| 4 bytes (word) | Aligned | | | | Aligned | | | |
| 4 bytes (word) | ■ | Misaligned | | | | Misaligned | | |
| 4 bytes (word) | ■ | ■ | Misaligned | | | | Misaligned | |
| 4 bytes (word) | ■ | ■ | ■ | Misaligned | | | | Misaligned |
| 8 bytes (double word) | Aligned | | | | | | | |
| 8 bytes (double word) | ■ | Misaligned | | | | | | |
| 8 bytes (double word) | ■ | ■ | Misaligned | | | | | |
| 8 bytes (double word) | ■ | ■ | ■ | Misaligned | | | | |
| 8 bytes (double word) | ■ | ■ | ■ | ■ | Misaligned | | | |
| 8 bytes (double word) | ■ | ■ | ■ | ■ | ■ | Misaligned | | |
| 8 bytes (double word) | ■ | ■ | ■ | ■ | ■ | ■ | Misaligned | |
| 8 bytes (double word) | ■ | ■ | ■ | ■ | ■ | ■ | ■ | Misaligned |

Figure : Aligned and misaligned bytes

## Specifying Memory Address: Intriguing Questions

Natural questions to ask with respect to byte ordering and alignment:

- What is the model of computer memory? How do I visualize the computer's memory?
- How is memory address specified for an object?
- How is memory accessed? Why do we break memory into elements like "bytes" and "words"?
- Why are there variable word size for different architectures?
- Why did endian-ness arise? Are there any advantages to one over the other?
- Should a programmer worry about all these things at all?
- How does these issues affects overall performance? (a research question)
- Does programming language suffer from these issues, particularly memory alignment?

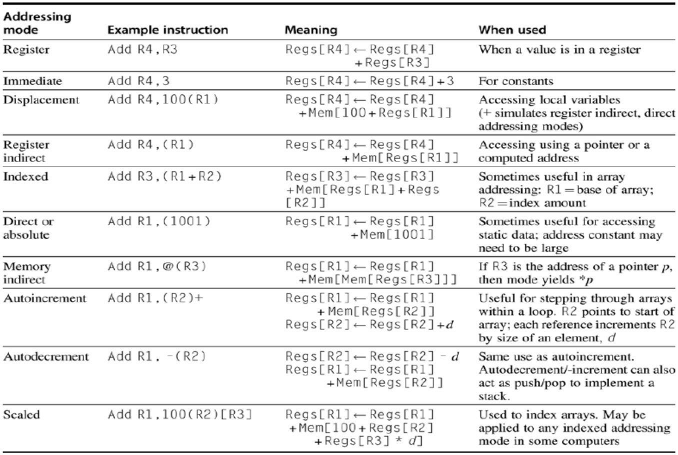
## Addressing Modes: Specifying Addrs in Instruction

What are the different ways addresses can be specified in an instruction?

- Register
- Immediate
- Displacement
- Register Indirect
- Indexed
- Direct/Absolute
- Memory indirect
- Autoincrement
- Autodecrement

## Addressing Modes: Specifying Addrs in Instruction

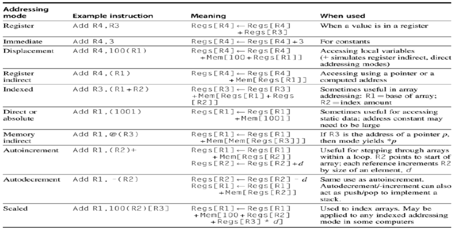
What are the different ways operand address can be specified?



| Addressing mode | Example instruction | Meaning | When used |
|---|---|---|---|
| Register | Add R4,R3 | Regs[R4]←Regs[R4] +Regs[R3] | When a value is in a register |
| Immediate | Add R4,3 | Regs[R4]←Regs[R4]+3 | For constants |
| Displacement | Add R4,100(R1) | Regs[R4]←Regs[R4] +Mem[100+Regs[R1]] | Accessing local variables (+ simulates register indirect, direct addressing modes) |
| Register indirect | Add R4,(R1) | Regs[R4]←Regs[R4] +Mem[Regs[R1]] | Accessing using a pointer or a computed address |
| Indexed | Add R3,(R1+R2) | Regs[R3]←Regs[R3] +Mem[Regs[R1]+Regs [R2]] | Sometimes useful in array addressing: R1 = base of array; R2 = index amount |
| Direct or absolute | Add R1,(1001) | Regs[R1]←Regs[R1] +Mem[1001] | Sometimes useful for accessing static data; address constant may need to be large |
| Memory indirect | Add R1,@(R3) | Regs[R1]←Regs[R1] +Mem[Mem[Regs[R3]]] | If R3 is the address of a pointer *p*, then mode yields **p* |
| Autoincrement | Add R1,(R2)+ | Regs[R1]←Regs[R1] +Mem[Regs[R2]] Regs[R2]←Regs[R2]+*d* | Useful for stepping through arrays within a loop. R2 points to start of array; each reference increments R2 by size of an element, *d* |
| Autodecrement | Add R1,–(R2) | Regs[R2]←Regs[R2]–*d* Regs[R1]←Regs[R1] +Mem[Regs[R2]] | Same use as autoincrement. Autodecrement/-increment can also act as push/pop to implement a stack. |
| Scaled | Add R1,100(R2)[R3] | Regs[R1]←Regs[R1] +Mem[100+Regs[R2] +Regs[R3]**d*] | Used to index arrays. May be applied to any indexed addressing mode in some computers |

## Addressing Modes: Probing Further

Exercises:

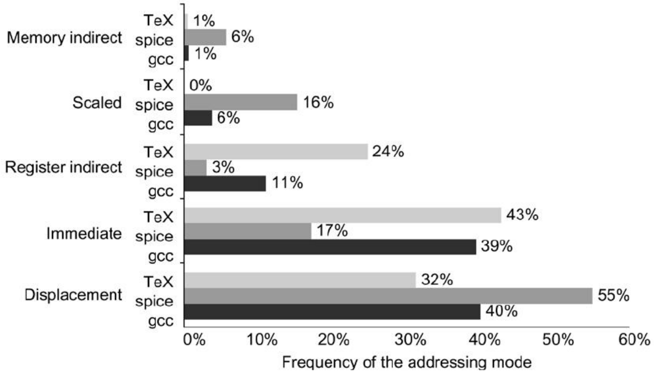


| Addressing mode | Example instruction | Meaning | When used |
|---|---|---|---|
| Register | `Add R4,R3` | `Regs[R4] ← Regs[R4] + Regs[R3]` | When a value is in a register |
| Immediate | `Add R4,3` | `Regs[R4] ← Regs[R4]+3` | For constants |
| Displacement | `Add R4,100(R1)` | `Regs[R4] ← Regs[R4] +Mem[100+Regs[R1]]` | Accessing local variables (+ simulates register indirect, direct addressing modes) |
| Register indirect | `Add R4,(R1)` | `Regs[R4] ← Regs[R4] +Mem[Regs[R1]]` | Accessing using a pointer or a computed address |
| Indexed | `Add R3,(R1+R2)` | `Regs[R3] ← Regs[R3] +Mem[Regs[R1]+Regs [R2]]` | Sometimes useful in array addressing: R1 = base of array; R2 = index amount |
| Direct or absolute | `Add R1,(1001)` | `Regs[R1] ← Regs[R1] +Mem[1001]` | Sometimes useful for accessing static data; address constant may need to be large |
| Memory indirect | `Add R1,@(R3)` | `Regs[R1] ← Regs[R1] +Mem[Mem[Regs[R3]]]` | If R3 is the address of a pointer *p*, then mode yields *p |
| Autoincrement | `Add R1,(R2)+` | `Regs[R1] ← Regs[R1] +Mem[Regs[R2]]` `Regs[R2] ← Regs[R2]+d` | Useful for stepping through arrays within a loop. R2 points to start of array; each reference increments R2 by size of an element, *d* |
| Autodecrement | `Add R1, –(R2)` | `Regs[R2] ← Regs[R2] – d` `Regs[R1] ← Regs[R1] +Mem[Regs[R2]]` | Same use as autoincrement. Autodecrement/-increment can also act as push/pop to implement a stack. |
| Scaled | `Add R1,100(R2)[R3]` | `Regs[R1] ← Regs[R1] +Mem[100+Regs[R2] +Regs[R3] * d]` | Used to index arrays. May be applied to any indexed addressing mode in some computers |

- How does these addressing mode affects performance?
- When shall we call an addressing mode to be power efficient?
- How does they impact the hardware complexity?
- How can we build a secure ISA? Can we make authorization for every instructions?

## Addressing Modes: Where they are used?

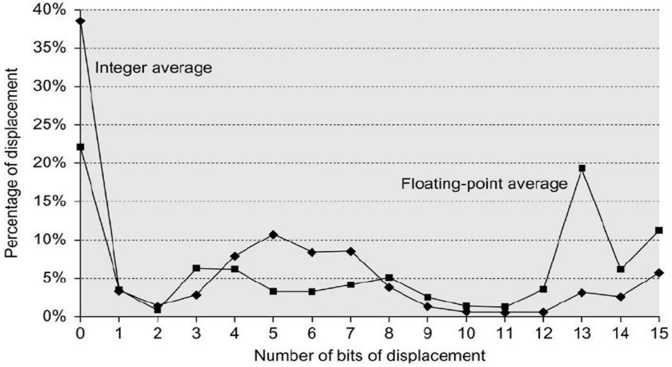
Statistics on usage of various Addressing modes in different benchmark



## Addressing Modes: Displacement Addr Mode

Add R4, 100(R1)

The displacement field affects the instruction length size!



## Addressing Modes: Immediate Addressing Mode

Add R4, 108 ; Regs[R4] $\leftarrow$ Regs[R4] + 108

Where the Immediate values are being used most?

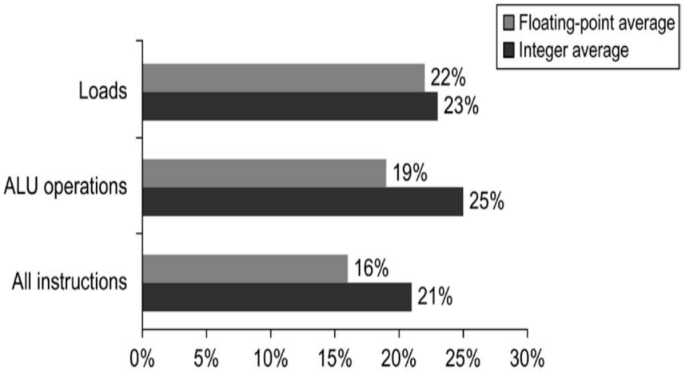


Figure : Usage of immediate operands across the instructions

## Addressing Modes: Immediate Addressing Mode

Add R4, 108

The immediate and displacement field affects the instruction length size!

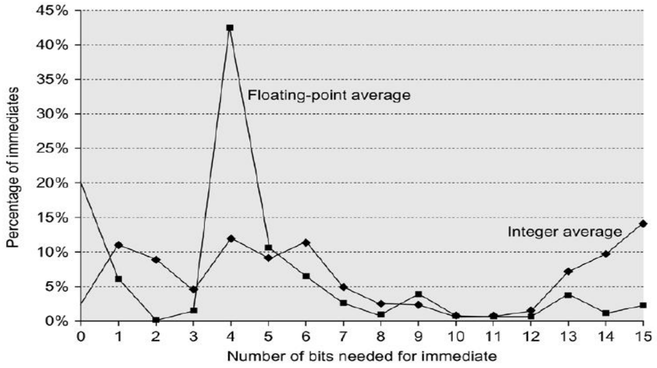


Figure : Number of bits (bit width) used for operations

## Operations and Operands

Add R4, 100(R1)

Operation types and Example instructions

- Arithmetic and Logic: add, subtract, and, or, multiply, divide, shift
- Data Transfer: move, load, store
- Control: branch, jump, call, traps
- System: OS CALL, VM, printer etc, network packet
- Floating Point: ADDF, MULF, DIVF
- Decimal: arithmetic, dec to char convert
- String: move, compare, search
- Graphics: compress, decomp, pixel, vertex

## Operations and Operands

Add R4, (1001) ; Regs[R4] $\leftarrow$ Regs[R4] + Mem[1001]

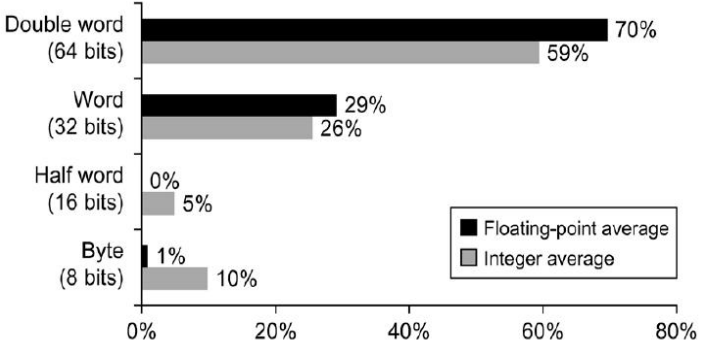
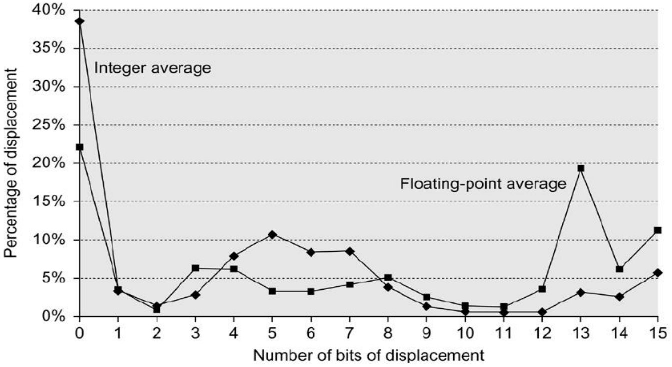


Figure : Distribution of data accesses

## Addressing Modes: Displacement Addr Mode

Add R4, 100(R1)

The displacement field affects the instruction length size!



## Analysis of Instruction Frequency

x86 instruction frequency for SPECint92 suit.

| 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% |
| Total | | 96% |

96% of the executed code is dominated by simple instructions! Why this analysis is important?

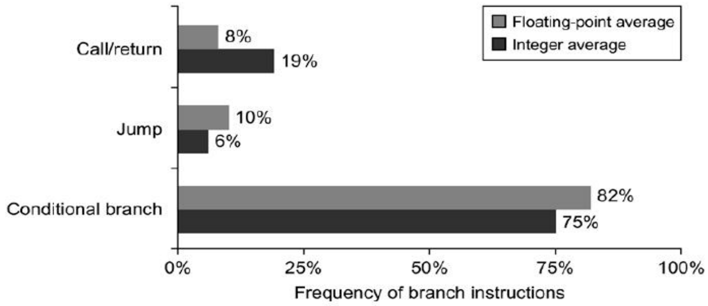
How does it appears for SPEC2017 suit?

## Analysis of Control Flow Instructions

There are different ways a program control can be changed, which is achieved by a set of control instructions.

- Four different class of control instructions:
    - Conditional branches
    - Jump (unconditional branches)
    - Procedure calls
    - Procedure returns

# Frequency of Control Flow Instructions



## Addressing Modes for Control Flow Instructions

Which type of addressing modes are suitable for control flow instructions? How to specify the target address?

- PC-relative addressing (this is similar to displacement mode)
- PC-relative is position independent
- How many bits of displacement?

What if the address is not known at the compile time?

## Addressing Modes for Control Flow Instructions

What if the address is not known at the compile time?

- Specify the target dynamically
- Need a register to specify the address dynamically
- Register indirect addressing mode is commonly used

Scenario in programming language:

- *Switch-case*: Selecting one among many cases!
- *Virtual function or Method*: Different routines to be called!
- *funtion pointer*
- *dynamically linked libraries*

## Analysis of Branch Distance

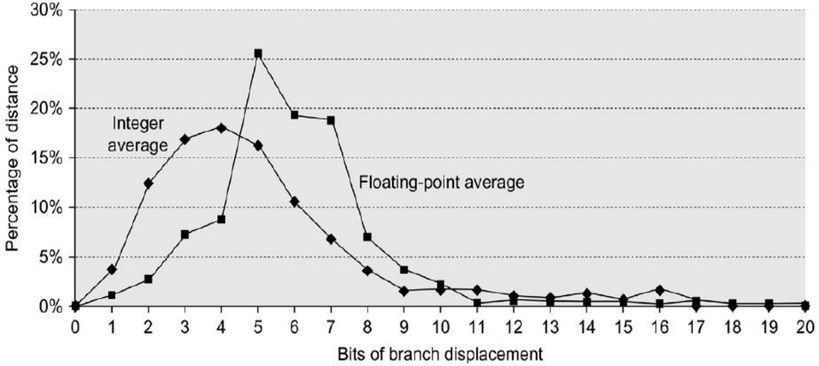


Figure : Branch distance interms of number of instructions between the target and the branch

The key point to observe is the number of bits

## Analysis of Conditional Branch Instruction

The truth value of condition is: {TRUE, FALSE}

There are three ways to specify the condition:

- Condition code (CC)
  - Test special bits (flag) set by ALU
  - 80x86, ARM, PowerPC, SPARC, SuperH
- Condition register/Limited Comparison
  - Test arbitrary register with the result of simple comparison (for equality)
  - Alpha, MIPS
- Compare and branch
  - Compare is part of branch.
  - RISC-V, VAX

## Frequency of Conditional Control Instruction

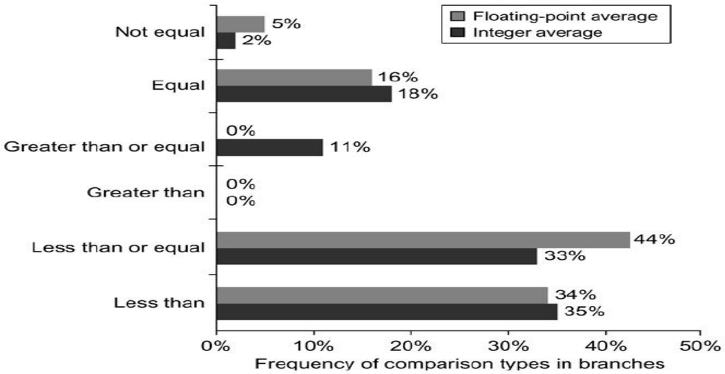


Figure : Frequency of compares in conditional branch

## So far on Instruction Set Analysis

– We have so far seen all the instructions which are visible to the assembly programmer
– Now, we need to take decision, based on these instruction set, on designing a hardware.

## Encoding an Instruction Set

The basic principles while encoding the instruction set.
The architect must balance several competing forces:

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

## Encoding an Instruction Set

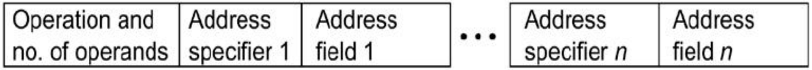
Definition: To represent the instructions in such a way that it could be decoded by the hardware.

Three choices to encode the instructions:

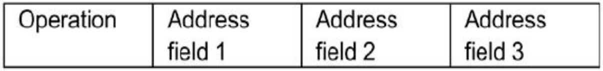
- Variable Length Encoding (All)
- Fixed Length Encoding (All)
- Variable + Fixed

## Encoding an Instruction Set

Three choices to encode the instructions:



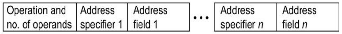
(A) Variable (e.g., Intel 80x86, VAX)



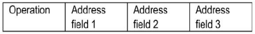
(B) Fixed (e.g., RISC V, ARM, MIPS, PowerPC, SPARC)

# Encoding an Instruction Set

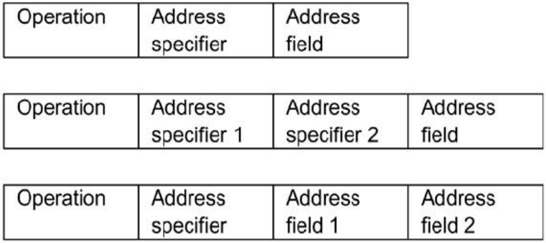
Three choices to encode the instructions:



(A) Variable (e.g., Intel 80x86, VAX)



(B) Fixed (e.g., RISC V, ARM, MIPS, PowerPC, SPARC)



(C) Hybrid (e.g., RISC V Compressed (RV32IC), IBM 360/370, microMIPS, Arm Thumb2)

## References

- Appendix A, Instruction Set Principle; Computer Architecture: Quant approach; Hennessy n Patterson; 6th Ed.

thank you