EEL 4768
Computer Architecture

Instruction Set Architecture (2)

Outline

- Type and Size of Operands
- Instruction Types
- Control Flow
- Procedure Invocation

Type and Size of Operands

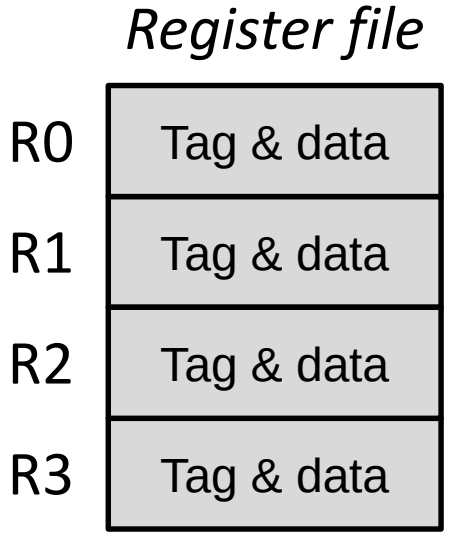
- The CPU should be able to do operations on multiple types and sizes of operands

Types: character, integer (arithmetic, logical), floating point

Size: 8-bit, 16-bit, 32-bit, 64-bit

- Opcode vs. Tagging

The opcode is the same (eg. ADD) whether the data is integer, floating-point, character, …



The tag indicates the size and type of the operand

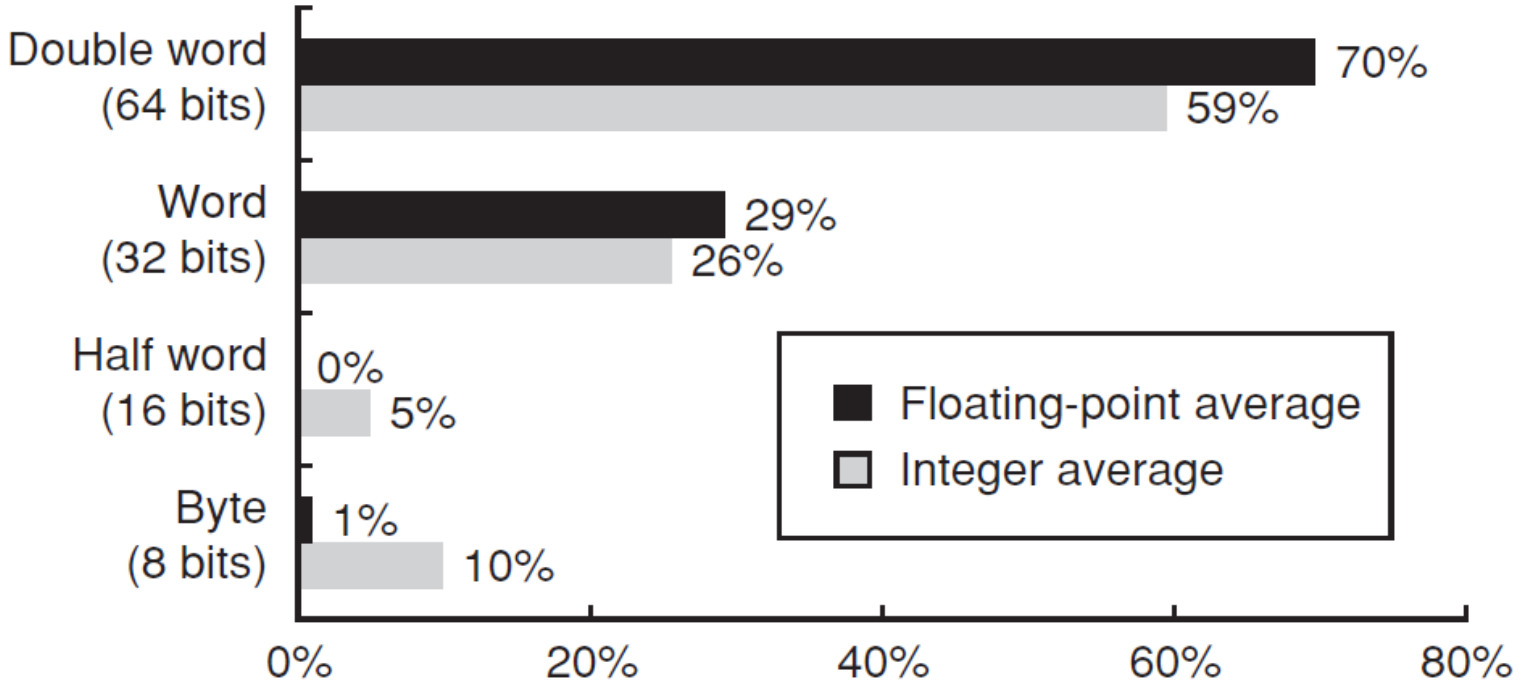
Type and Size of Operands

- Common data types and sizes:
  - Integer: 2's complement word (32-bit)
  - Character: 8-bit (ASCII), 16-bit (Unicode) Java
  - Floating point: IEEE 754 Single-precision (32-bit) double-precision (64-bit)

- Some others:
  - String: compare or move of chars in some architectures
  - BCD: binary-coded decimal

Type and Size of Operands

- The SPEC benchmark uses:



- 64-bit integer → to reduce the overflow possibility
- 64-bit floating point → better precision
- Takeaway: 64-bit datapath would fetch in 1 clock cycle

Operations in the Instruction Set

Operations in the Instruction Set

Operator type	Examples
Arithmetic and logical	Integer arithmetic and logical operations: add, subtract, and, or, multiply, divide
Data transfer	Loads-stores (move instructions on computers with memory addressing)
Control	Branch, jump, procedure call and return, traps
System	Operating system call, virtual memory management instructions
Floating point	Floating-point operations: add, multiply, divide, compare
Decimal	Decimal add, decimal multiply, decimal-to-character conversions
String	String move, string compare, string search
Graphics	Pixel and vertex operations, compression/decompression operations

Well-supported

Exists, not standardized

Rare

- If not supported: synthesized by the compiler from simpler instructions
  - For example, if MIPS doesn't support 'load half' and 'load byte', we can still process 16-bit and 8-bit numbers, but it would take multiple instructions to do the task

Operations in the Instruction Set

- **Simplicity wins!**
- Integer programs on an Intel 80x86 processor:
  - 10 simple instructions account for 96%

Rank	80x86 instruction	Integer average (% total executed)
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
Total		96%

Instructions for Control Flow

- Branch and jump instructions
- Can be **‘conditional’** or **‘unconditional’**
- 50s: were called **‘transfers’**
- 60s: the name **‘branch’** started to be used
- In the book:
  - Branch refers to conditional instructions
  - Jump refers to unconditional instructions
- Some architectures, like Intel x86, call everything ‘jump’, whether the instruction is conditional or unconditional.

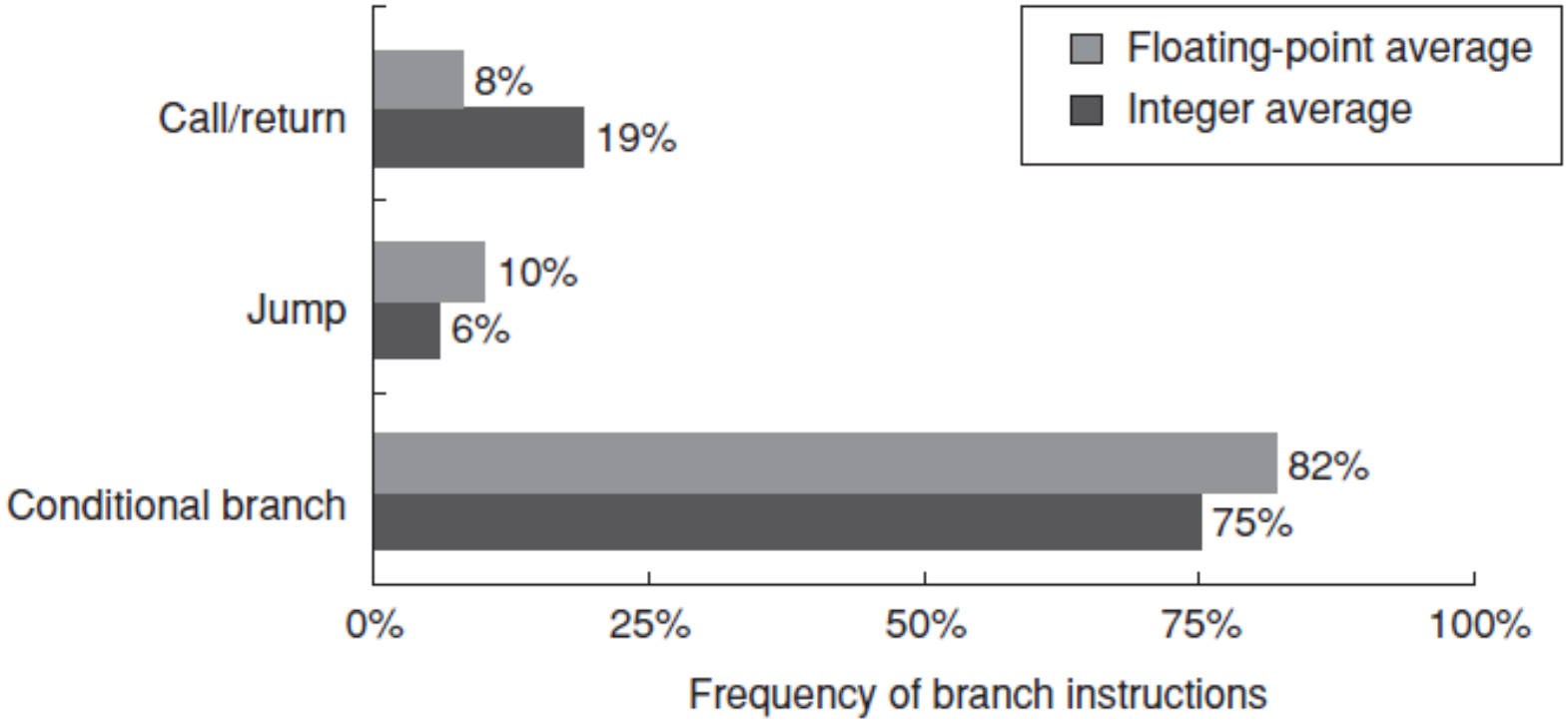
Instructions for Control Flow

- Four different types of control flow instructions:
  - Conditional branches
  - Jumps
  - Procedure calls
  - Procedure returns

- The procedure call links the return address
- The procedure return jumps back to the calling code (possibly the ‘main’ function)

Instructions for Control Flow

- The figure shows which control flow instructions are the most popular



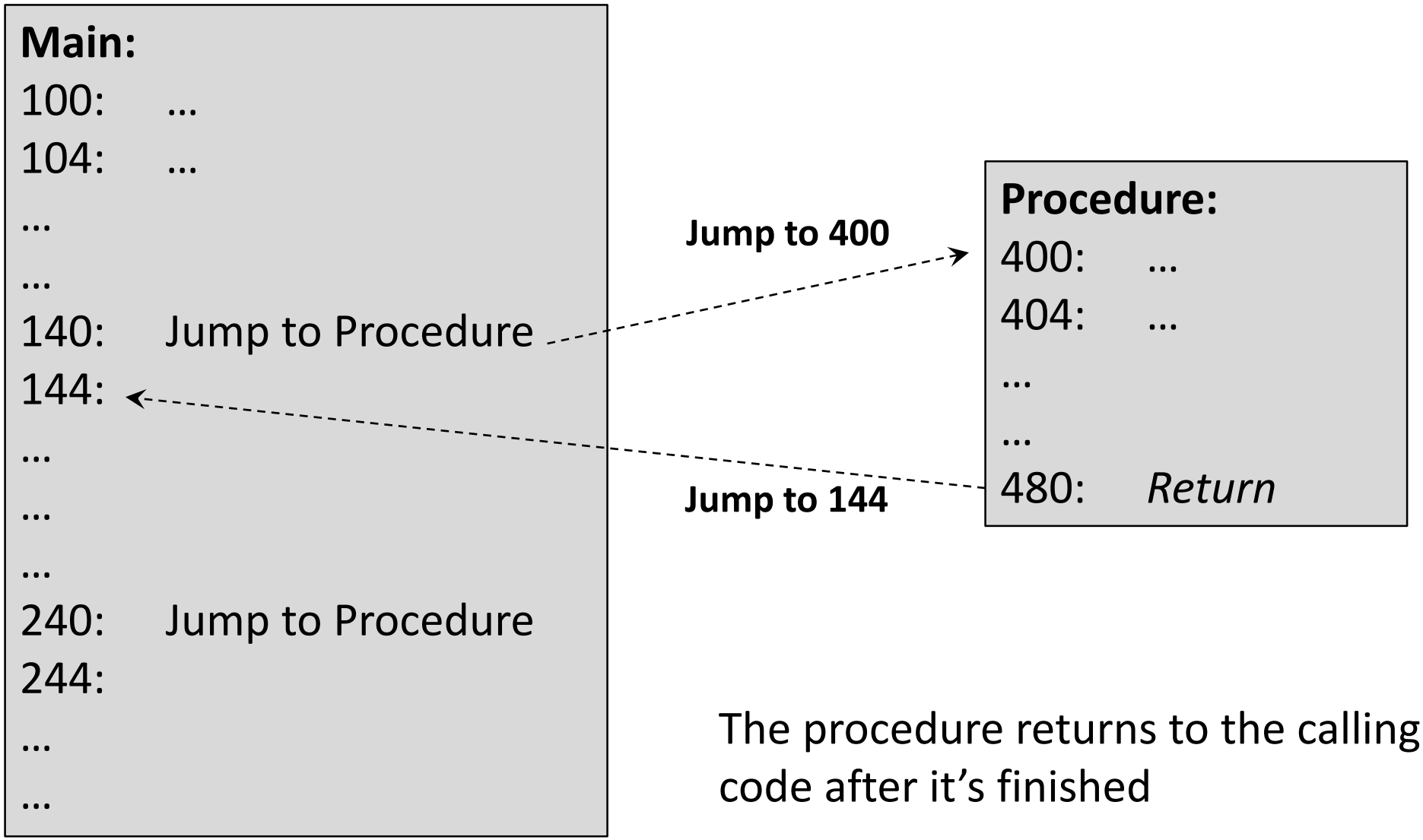
- The measurements were taken from a benchmark running on a load-store architecture

Instructions for Control Flow

- The destination address of a control flow instruction is usually specified in the instruction

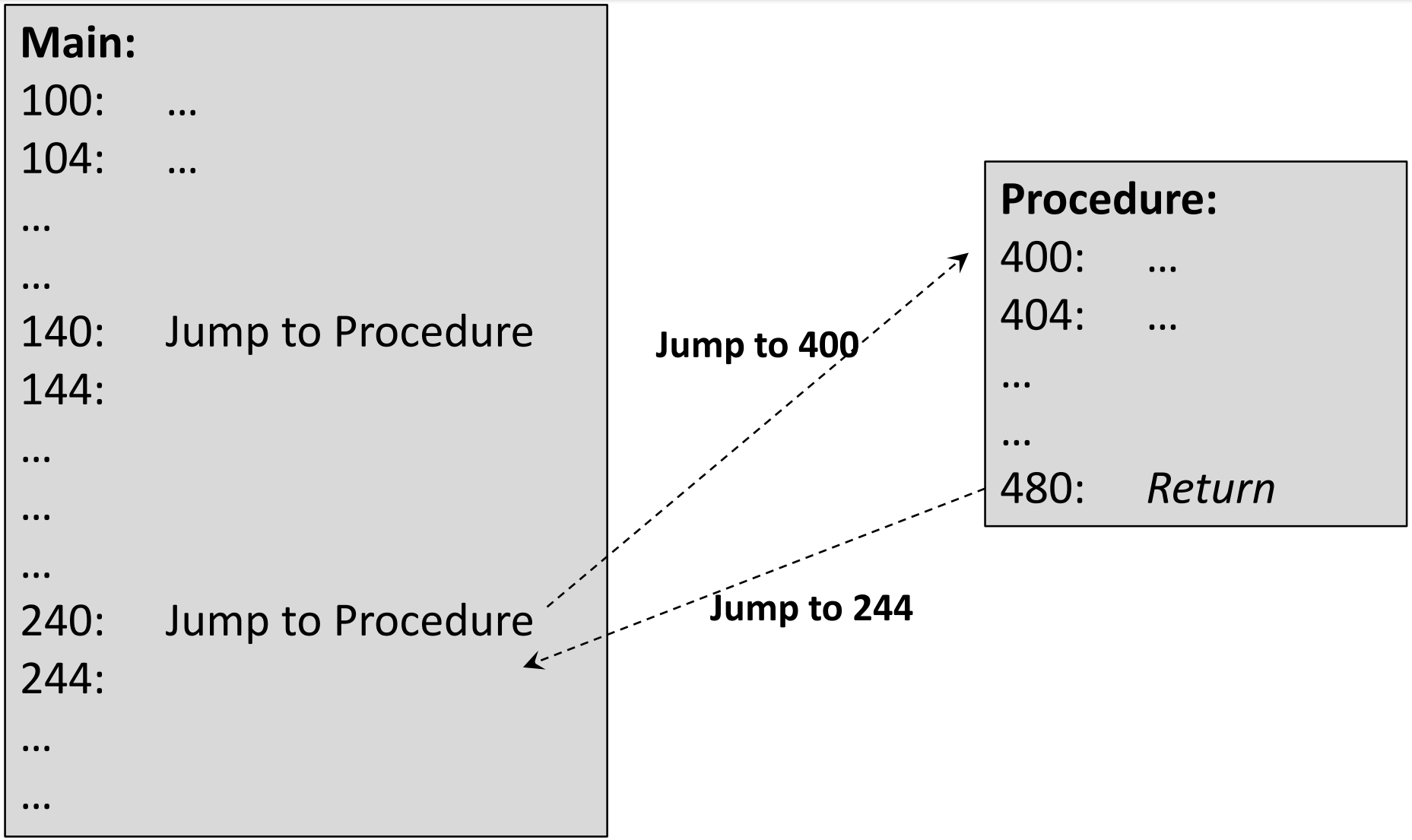
- The exception to this is the ‘procedure return’ case
  - When the procedure returns, the program should go back to the calling code
  - The procedure could be called from multiple places in the code
  - Therefore, at compile time, the return address is not known

Instructions for Control Flow



The procedure returns to the calling code after it's finished

Instructions for Control Flow



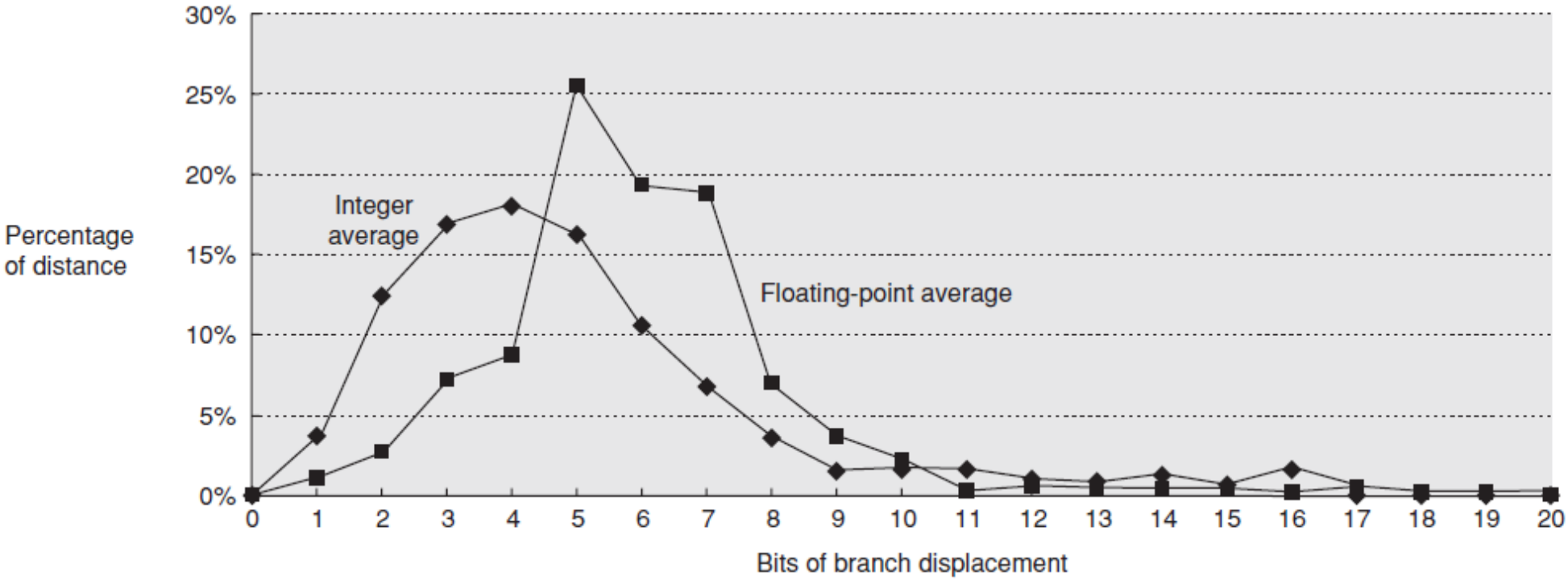
- The return address is usually saved in a register and procedure returns use a jump-to-register instruction (e.g., jr $ra), where $ra has the return address

Instructions for Control Flow

- PC-relative addressing mode:
  - Destination as a displacement
- Works great:
  - Destination is typically close by: a full address is not necessary
  - Can run independent of the absolute position.
  - Linking is easy.

Instructions for Control Flow

- Branch Offset: signed number
- Branch Distance = | Branch Offset |
- How far is the branch address is from the branch instruction?
  - # of bits needed for the offset in the instruction



- 8 bits or less seem to be enough for most cases
- The measurements from a load-store architecture

Conditional Branch Instructions

- *How is the condition mechanism implemented?*

Name	Examples	How condition is tested	Advantages	Disadvantages
Condition code (CC)	80x86, ARM, PowerPC, SPARC, SuperH	Tests special bits set by ALU operations, possibly under program control.	Sometimes condition is set for free.	CC is extra state. Condition codes constrain the ordering of instructions since they pass information from one instruction to a branch.
Condition register	Alpha, MIPS	Tests arbitrary register with the result of a comparison.	Simple.	Uses up a register.
Compare and branch	PA-RISC, VAX	Compare is part of the branch. Often compare is limited to subset.	One instruction rather than two for a branch.	May be too much work per instruction for pipelined execution.

Conditional Branch: Condition Code

- Condition Code Register (CCR) contains multiple 1-bit fields:
- Example: CCR: [Z, S, …] (Z: zero bit, S: signed bit)
- A conditional branch is done in two instructions:
  - an arithmetic (or logic) operation that sets the condition code register
  - the following instruction inspects CCR and decides if the branch should be taken

- The code below branches to ‘Label’ if R1 is equal to R2
- There should not be any instruction between SUB and BRZ that changes the condition codes!

```
SUB   R3, R1, R2    // will do (R1–R2) and sets the flags
BRZ   Label         // will branch if Z flag =1 (which means
                    // R1 – R2=0)
```

Conditional Branch: Condition Register

- How can we use the CCR approach to branch if R1=R2?

```
SUB   R3, R1, R2    // will do: R3 = R1 – R2
BEQ   R3, Label     // will branch if R3=0 (which means R1=R2)
```

Conditional Branch: Condition Register

- Alpha architecture uses the **condition register approach**
- Its format of the branch instruction is:

Bxx <Reg> <Offset>

- The term ‘Bxx’ is replaced with ‘BEQ’ (branch on equal), ‘BGT’ (branch on greater than), etc.
- There is only one register in the branch instruction
- The other register is implied as the value zero
- Therefore, this instruction branches if R1 is equal to zero

BEQ R1, Label

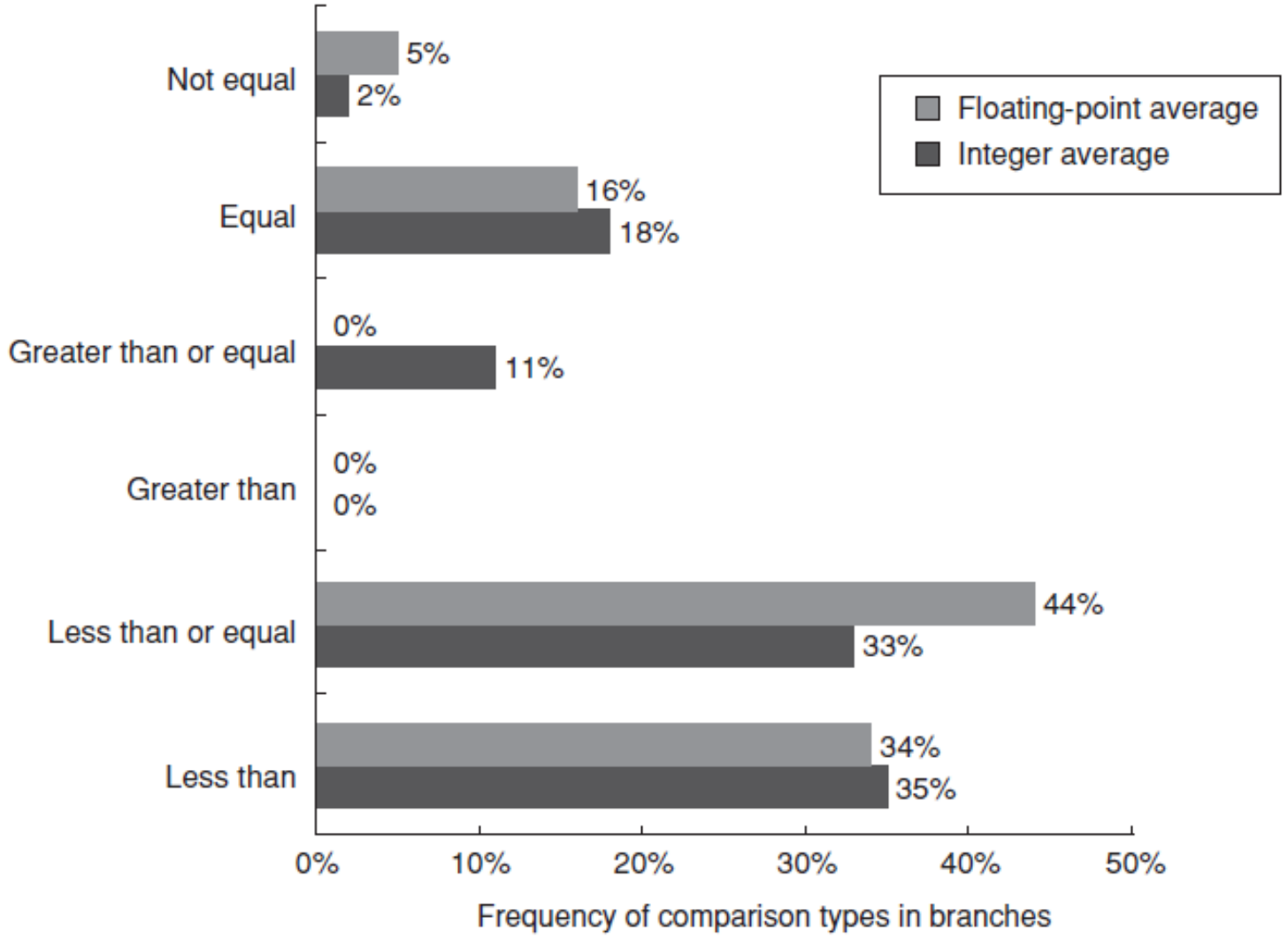
Conditional Branch: Condition Register

- Compilers also take advantage of the **implicit zero register**
- Let's say we write a loop in a high-level language that increments the counter from 0 to 9
- One way to check for loop termination is to do the subtraction (counter–10), if the result is zero, the loop stops
- *Therefore, the comparison is done with zero*

Conditional Branch: Compare & Branch

- MIPS' 'beq' and 'bne' instructions fall in this category
  - Uses many clock cycles: hard to pipeline
- For floating-point comparisons, MIPS uses CCR
- Practically, the number of branches based on the floating-point comparisons is small
  - Branches are usually based on counters which are integers
- This may be the biggest disadvantage of MIPS!

Conditional Branch Instructions



Procedure Invocation Options

- Control transfer:
  - Branching to the procedure and returning to the calling code
- State saving
  - A procedure might alter a register that the calling code needs to use after the procedure returns
  - Such registers should be saved before the procedure is called; then, they should be restored when the procedure returns

Procedure Invocation Options

- Procedure call: 'return address' must be saved
- MIPS: saves in $ra, a link register
  - The instruction 'jal' (jump and link) jumps to the label and saves the return address automatically in register $ra

Jal Procedure1

- Alpha: saves in any register

BSR R1, Procedure

  - 'BSR' (Branch to SubRoutine) saves in register R1
- Procedure return in both: a 'jump-to-register' instruction is used:

jr $ra (MIPS)

jr R1 (Alpha)

Procedure Invocation

- Who saves the state?
- **Caller saving:**
  - The calling code saves the registers that it wants to use after the procedure returns
  - The procedure code doesn't save any registers
- **Callee saving:**
  - The procedure saves any register that it plans to use
  - At the end of the procedure, the saved registers are restored before returning to the calling code
- Some older CPUs automatically save all the registers when a procedure is called
  - Used to be a reasonable approach for small # of registers

Procedure Invocation

- When the calling code and the procedure access the same global variable, ‘caller saving’ is preferred to ‘callee saving’

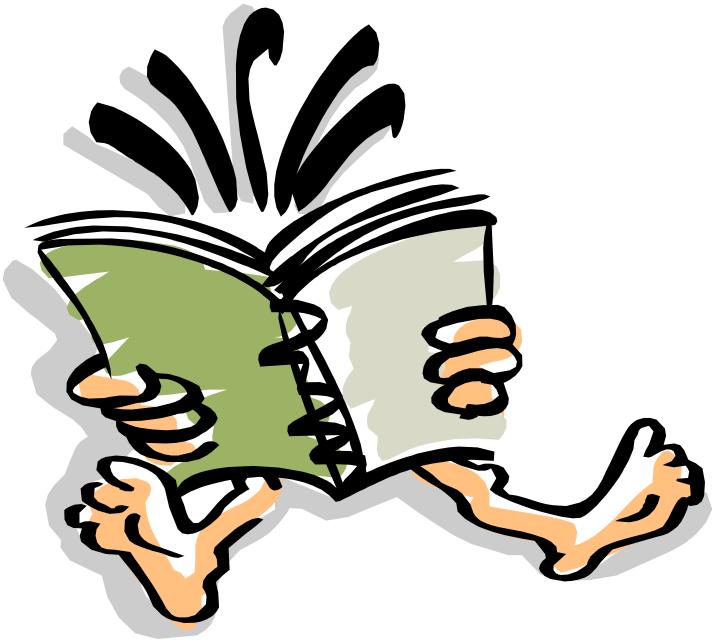
- *Consider a global variable ‘x’*
- Let’s say the calling code has mapped ‘x’ to register R1
- If we do ‘callee saving’, the calling code keeps ‘x’ in R1
- This means the procedure has to know that ‘x’ is now mapped to R1 and use the value from there

- However, with ‘caller saving’, the calling code saves ‘x’ at a memory location before calling the procedure
- The procedure can get ‘x’ from the memory and use it
- This is the preferred approach since the procedure doesn’t have to know the variable-to-register mapping of the calling code

Summary

- The instruction set includes:
  - Simple operations
  - PC-relative conditional branch
  - Jump-and-link instruction for procedure calling
  - Register indirect jump for procedure return
- We are leaning towards:
  - A load-store architecture
  - Addressing modes: displacement, immediate, register indirect
  - Integer data types are: 8-bit, 16-bit, 32-bit, 64-bit
  - Floating-point types are: 32-bit, 64-bit

Readings



- H&P CA
  - App A
  - App K