Memory Addressing

- Independent of whether the architecture is load-store or allows any operand to be a memory reference, it must define how memory addresses are interpreted and how they are specified.
- The measurements presented are largely, but not completely, computer independent.
- In some cases the measurements are significantly affected by the compiler technology.
- These measurements have been made using an optimizing compiler, since compiler technology plays a critical role.

Interpreting Memory Addresses

- How is a memory address interpreted? That is, what object is accessed as a function of the address and the length?

  All the instruction sets discussed are byte addressed and provide access for bytes (8 bits), half words (16 bits), and words (32 bits).

- Most of the computers also provide access for double words (64 bits).

Interpreting Memory Addresses

There are two different conventions for ordering the bytes within a larger object.

- Little Endian byte order puts the byte whose address is "x . . . x000" at the least-significant position in the double word (the little end). The bytes are numbered

- Big Endian byte order puts the byte whose address is "x . . . x000" at the most significant position in the double word (the big end). The bytes are numbered

Interpreting Memory Addresses

- When operating within one computer, the byte order is often unnoticeable— only programs that access the same locations as both words and bytes can notice the difference.
- However, byte order is a problem when exchanging data among computers with different orderings.
- Little Endian ordering also fails to match normal ordering of words when strings are compared. Strings appear "SDRAWKCAB" (backwards) in the registers.

Interpreting Memory Addresses

- A second memory issue is that in many computers, accesses to objects larger than a byte must be aligned.
- An access to an object of size s bytes at byte address A is aligned if A mod s = 0. Figure 12 shows the addresses at which an access is aligned or misaligned.

Interpreting Memory Addresses

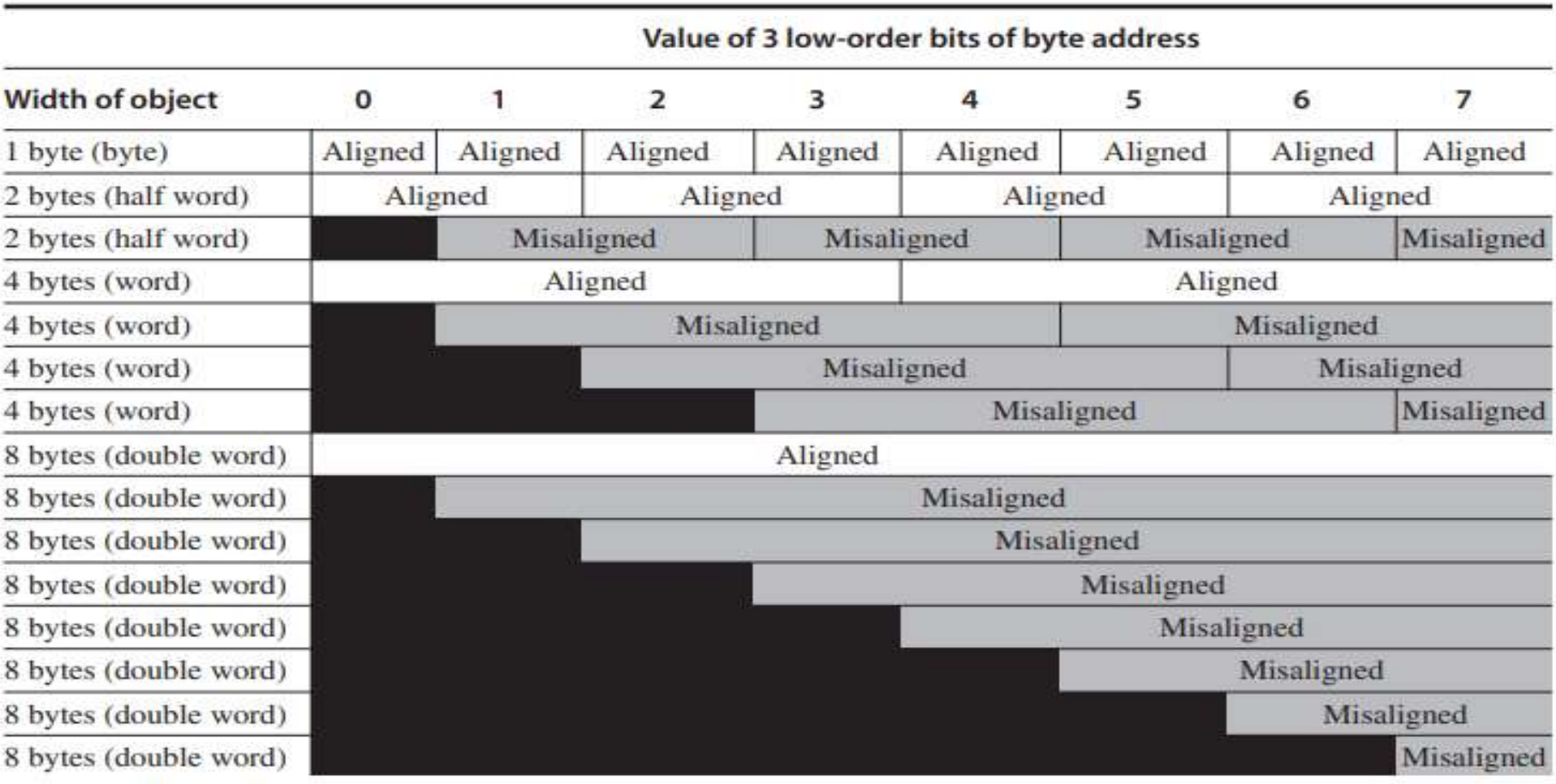


Figure 12 Aligned and misaligned addresses of byte, half-word, word, and double-word objects for byte-addressed computers. For each misaligned example some objects require two memory accesses to complete. Every aligned object can always complete in one memory access, as long as the memory is as wide as the object. The figure shows the memory organized as 8 bytes wide. The byte offsets that label the columns specify the low-order 3 bits of the address.

[J. Hennessy and D. Patterson, "Computer Architecture, A quantitative approach", 5th Edition.]

Interpreting Memory Addresses

- Why would someone design a computer with alignment restrictions?
- Misalignment causes hardware complications, since the memory is typically aligned on a multiple of a word or double-word boundary.
- A misaligned memory access may take multiple aligned memory references.
- Thus, even in computers that allow misaligned access, programs with aligned accesses run faster.

Interpreting Memory Addresses

- Even if data are aligned, supporting byte, half-word, and word accesses requires an alignment network to align bytes, half words, and words in 64-bit registers.
- Depending on the instruction, the computer may also need to sign-extend the quantity.
- On some computers a byte, halfword, and word operation does not affect the upper portion of a register.

Addressing Modes

- The ways addresses are specified by instructions are called as addressing modes.
- Addressing modes—how architectures specify the address of an object they will access.
- Addressing modes specify constants and registers in addition to locations in memory.
- When a memory location is used, the actual memory address specified by the addressing mode is called the effective address.

Addressing Modes

- Addressing modes have the ability to significantly reduce instruction counts.
- They also add to the complexity of building a computer and may increase the average CPI (clock cycles per instruction) of computers that implement those modes.
- Thus, the usage of various addressing modes is quite important in helping the architect choose what to include.

Addressing Modes

Hamacher, Vranesic & Zaky, "Computer Organization" (5th Ed), McGraw Hill.

- A program operates on data that reside in the computer's memory.
- These data can be organized in a variety of ways. For Eg: - if we want to keep track of students' names, we can write them in a list.
- Programmers use organizations called data structures to represent the data used in computations. These include lists, linked lists, arrays, queues, and so on.
- Programs are normally written in a high-level language, which enables the programmer to use constants, local and global variables, pointers, and arrays.
- The different ways in which the location of an operand is specified in an instruction are referred to as addressing modes.

Addressing Modes

Hamacher, Vranesic & Zaky, "Computer Organization" (5th Ed), McGraw Hill.

Table 1 Generic addressing modes

Name	Assembler syntax	Addressing function
Immediate	# Value	Operand = Value
Register	Ri	EA = Ri
Absolute (Direct)	LOC	EA = LOC
Indirect	(Ri)	EA = [Ri]
	(LOC)	EA = [LOC]
Index	X(Ri)	EA = [Ri] + X
Base with index	(Ri, Rj)	EA = [Ri] + [Rj]
Base with index and offset	X (Ri, Rj)	EA = [Ri] + [Rj] + X
Relative	X(PC)	EA = [PC] + X
Autoincrement	(Ri)+	EA = [Ri]; Increment Ri
Autodecrement	-(Ri)	Decrement Ri; EA = [Ri]

EA = effective address

Value = a signed number

Addressing Modes

Hamacher, Vranesic & Zaky, "Computer Organization" (5th Ed), McGraw Hill.

IMPLEMENTATION OF VARIABLE AND CONSTANTS:-

- Variables and constants are the simplest data types and are found in almost every computer program.
- In assembly language, a variable is represented by allocating a register or memory location to hold its value. Thus, the value can be changed as needed using appropriate instructions.
- We can access an operand by specifying the name of the register or the address of the memory location where the operand is located.

Addressing Modes

Hamacher, Vranesic & Zaky, "Computer Organization" (5th Ed), McGraw Hill.

IMPLEMENTATION OF VARIABLE AND CONSTANTS:-

- Register mode - The operand is the contents of a processor register; the name (address) of the register is given in the instruction.

- Absolute mode – The operand is in a memory location; the address of this location is given explicitly in the instruction. (In some assembly languages, this mode is called Direct).

Addressing Modes

Hamacher, Vranesic & Zaky, "Computer Organization" (5th Ed), McGraw Hill.

IMPLEMENTATION OF VARIABLE AND CONSTANTS:-

- The instruction Move LOC, R2
  Uses these two modes. Processor registers are used as temporary storage locations where the data in a register are accessed using the Register mode.

- Declaration such as Integer A, B;
  The Absolute mode can represent global variables in a program.

Addressing Modes

Hamacher, Vranesic & Zaky, "Computer Organization" (5th Ed), McGraw Hill.

IMPLEMENTATION OF VARIABLE AND CONSTANTS:-

- Address and data constants can be represented in assembly language using the immediate mode

- Immediate mode – The operand is given explicitly in the instruction.
- For example, the instruction Move $200_{immediate}$, R0

  Places the value 200 in register R0. Clearly, the Immediate mode is only used to specify the value of a source operand.

Addressing Modes

Hamacher, Vranesic & Zaky, "Computer Organization" (5th Ed), McGraw Hill.

IMPLEMENTATION OF VARIABLE AND CONSTANTS:-

- Using a subscript to denote the Immediate mode is not appropriate in assembly languages.
- A common convention is to use the sharp sign (#) in front of the value to indicate that this value is to be used as an immediate operand.
- Hence, we write the instruction Move $200_{immediate}$, R0 in the form Move #200, R0
- The statement in high level language is A = B+6 may be accessed using absolute mode as

```
Move B,R1
Add #6,R1
Move R1,A
```

Addressing Modes

Hamacher, Vranesic & Zaky, "Computer Organization" (5th Ed), McGraw Hill.

INDIRECTION AND POINTERS

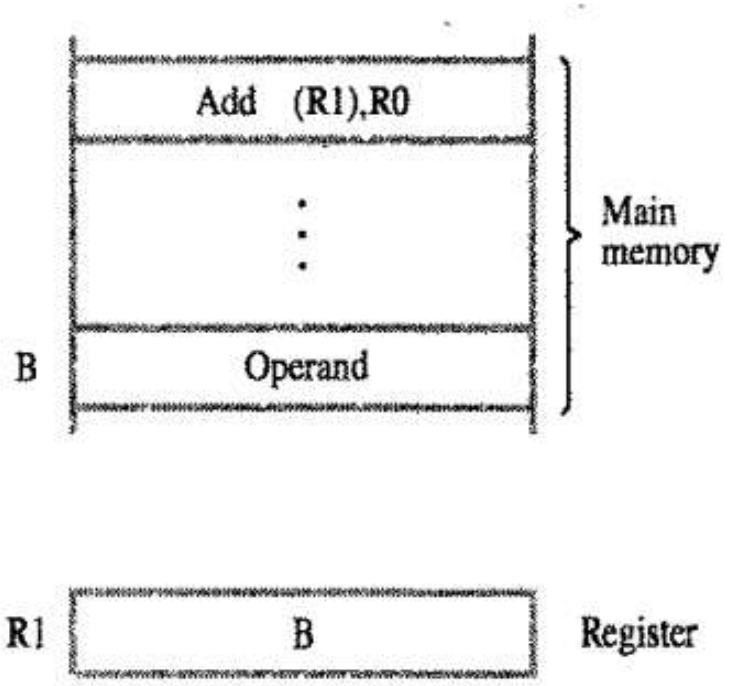
- In certain addressing modes, the instruction does not give the operand or its address explicitly.
- Instead, it provides information from which the memory address of the operand can be determined. This address is known as the effective address (EA) of the operand.
- Indirect mode – The effective address of the operand is the contents of a register or memory location whose address appears in the instruction.
- We denote indirection by placing the name of the register or memory address given in the instruction in the parenthesis.

Addressing Modes

Hamacher, Vranesic & Zaky, "Computer Organization" (5th Ed), McGraw Hill.

INDIRECTION AND POINTERS

- To execute the Add instruction in fig 18(a), the processor uses the value B, which is in register R1, as the effective address of the operand.
- It requests a read operation from the memory to read the contents of location B.
- The value read is the desired operand, which the processor adds to the contents of register R0.



(a) Through a general-purpose register

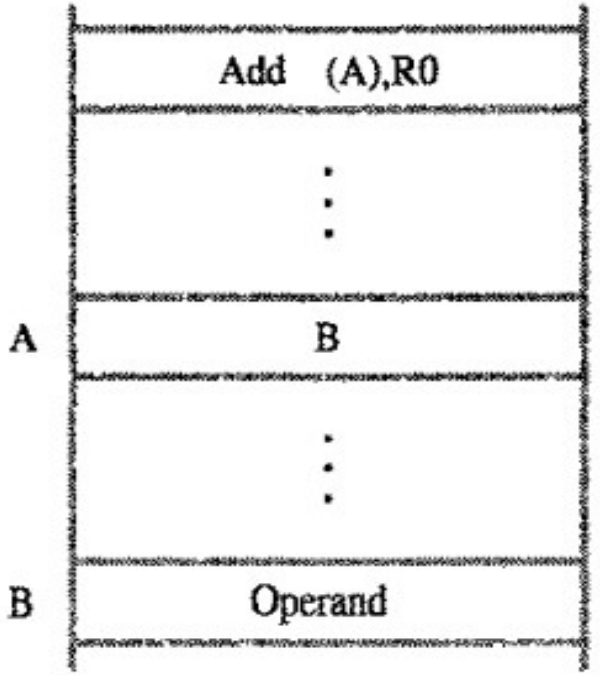
Figure 18 Indirect addressing.

Addressing Modes

Hamacher, Vranesic & Zaky, "Computer Organization" (5th Ed), McGraw Hill.

INDIRECTION AND POINTERS

- Indirect addressing through a memory location is also possible as shown in fig 18(b).
- In this case, the processor first reads the contents of memory location A, then requests a second read operation using the value B as an address to obtain the operand.



(b) Through a memory location

Figure 18 Indirect addressing.

Addressing Modes

Hamacher, Vranesic & Zaky, "Computer Organization" (5th Ed), McGraw Hill.

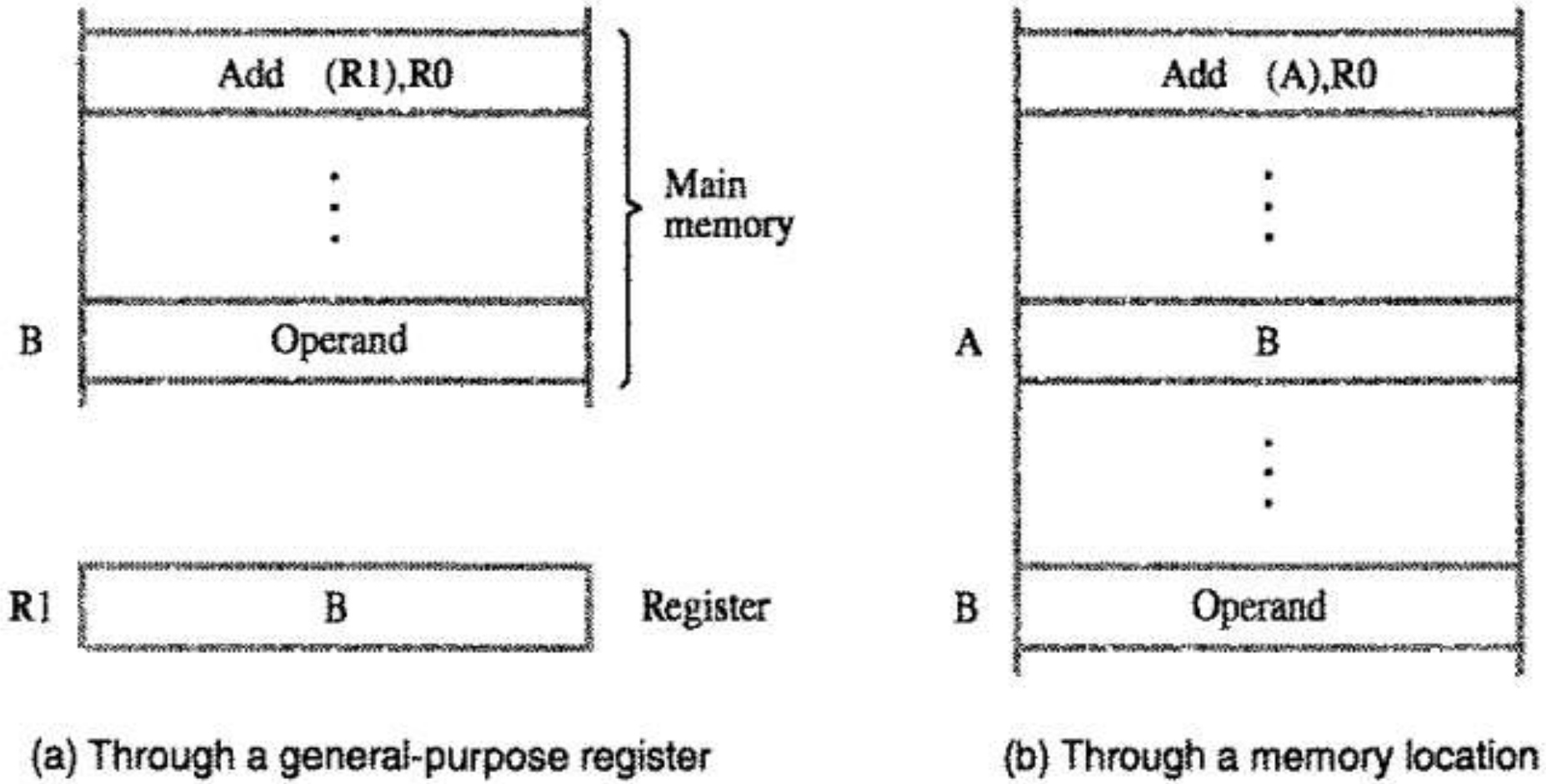


Figure 18 Indirect addressing.

Addressing Modes

Hamacher, Vranesic & Zaky, "Computer Organization" (5th Ed), McGraw Hill.

- The register or memory location that contains the address of an operand is called a pointer.
- Indirection and the use of pointers are important and powerful concepts in programming.

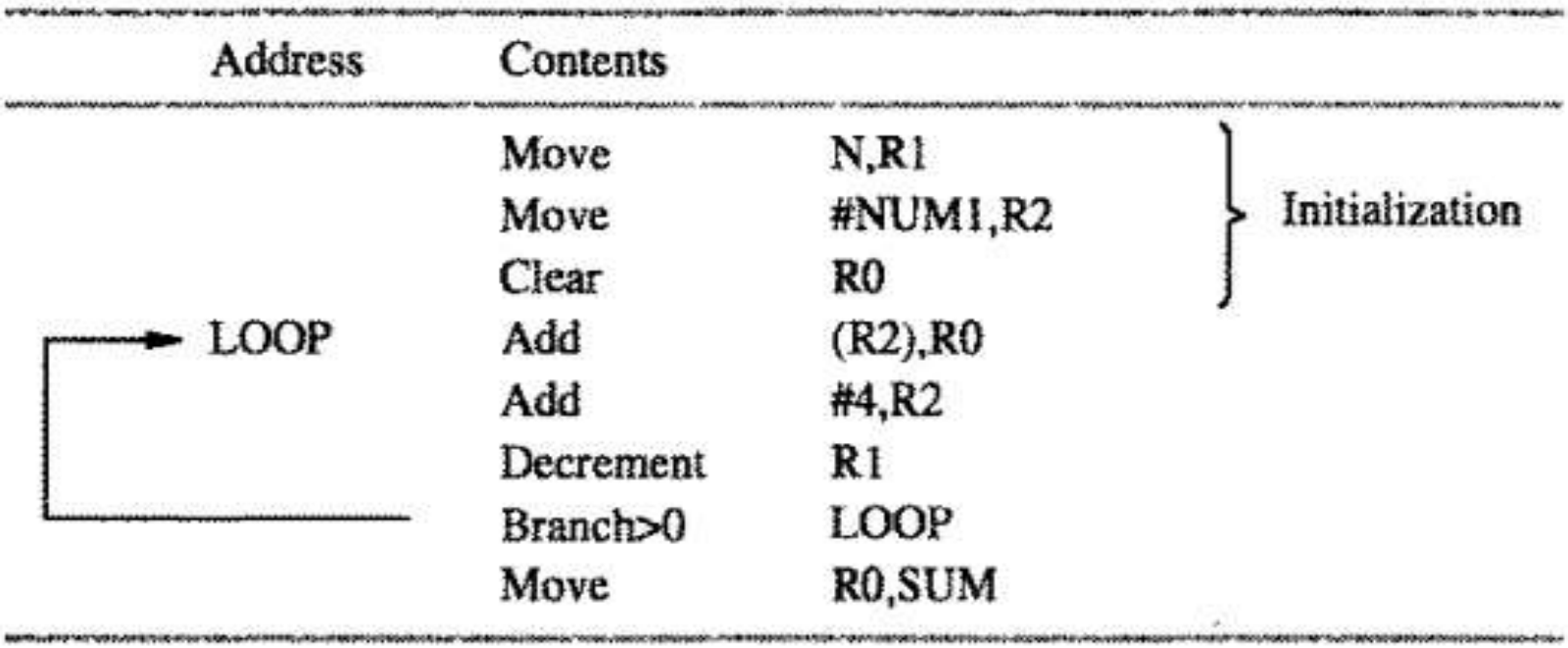


Figure 19 Use of indirect addressing in the program [Using a loop to add n numbers]

Addressing Modes

Hamacher, Vranesic & Zaky, "Computer Organization" (5th Ed), McGraw Hill.

- Register R2 is used as a pointer to the numbers in the list, and the operands are accessed indirectly through R2.
- The initialization section of the program loads the counter value n from memory location N into R1.
- Uses the immediate addressing mode to place the address value NUM1, which is the address of the first number in the list, into R2.
- Then it clears R0 to 0.

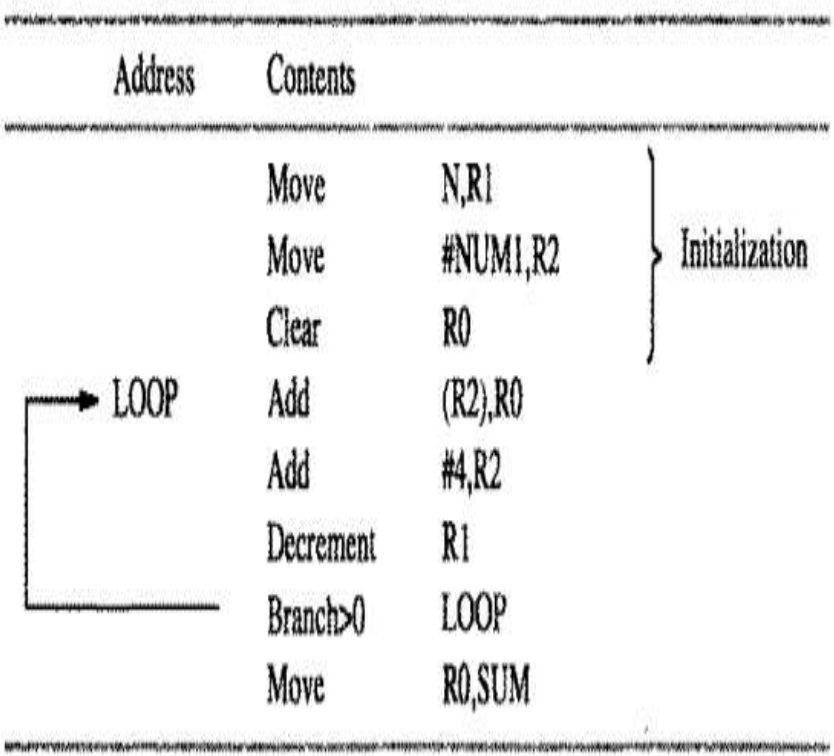


Figure 19 Use of indirect addressing in the program

[Using a loop to add n numbers]

Addressing Modes

Hamacher, Vranesic & Zaky, "Computer Organization" (5th Ed), McGraw Hill.

- The first time through the loop, the instruction Add (R2), R0 fetches the operand at location NUM1 and adds it to R0.

- The second Add instruction adds 4 to the contents of the pointer R2, so that it will contain the address value NUM2 when the above instruction is executed in the second pass through the loop.

[Using a loop to add n numbers]

Address	Contents		
	Move	N,R1	Initialization
	Move	#NUM1,R2	
	Clear	R0	
LOOP	Add	(R2),R0	
	Add	#4,R2	
	Decrement	R1	
	Branch>0	LOOP	
	Move	R0,SUM	

Figure 19 Use of indirect addressing in the program

[Address Offset: Adding 4 to the address in the register increments the address by 4 bytes. This is used when accessing consecutive memory locations, as in an array of 32-bit integers (4 bytes per integer). By adding 4 to the address, the register points to the next integer in the array.]

Addressing Modes

Hamacher, Vranesic & Zaky, "Computer Organization" (5th Ed), McGraw Hill.

- Consider the C statement

  A=*B; Where B is a pointer variable.

  A pointer is a variable that stores the memory address of another variable as its value

- This statement may be compiled into

  Move B, R1

  Move (R1), A [Indirect addressing through registers]

  Using indirect addressing through memory, the same action can be achieved with

  Move (B),A

- Indirect addressing through registers is used extensively. However indirect addressing through memory is rarely used in modern computers since it is not suited to pipelined execution.
- When absolute addressing is not available, indirect addressing through registers makes it possible to access global variables by first loading the operand's address in a register.

Addressing Modes

Hamacher, Vranesic & Zaky, "Computer Organization" (5th Ed), McGraw Hill.

INDEXING AND ARRAYS

- A different kind of flexibility for accessing operands is useful in dealing with lists and arrays.
- Index mode – the effective address of the operand is generated by adding a constant value to the contents of a register.
- The register used may be either a special register provided for this purpose, or, more commonly, it may be any one of a set of general-purpose registers in the processor.
- In either case, it is referred to as index register and indicate the Index mode symbolically as

X (Ri)

Where X denotes the constant value contained in the instruction and Ri is the name of the register involved.

Addressing Modes

Hamacher, Vranesic & Zaky, "Computer Organization" (5th Ed), McGraw Hill.

INDEXING AND ARRAYS

X (Ri)

- The effective address of the operand is given by

  EA = X + [Ri]

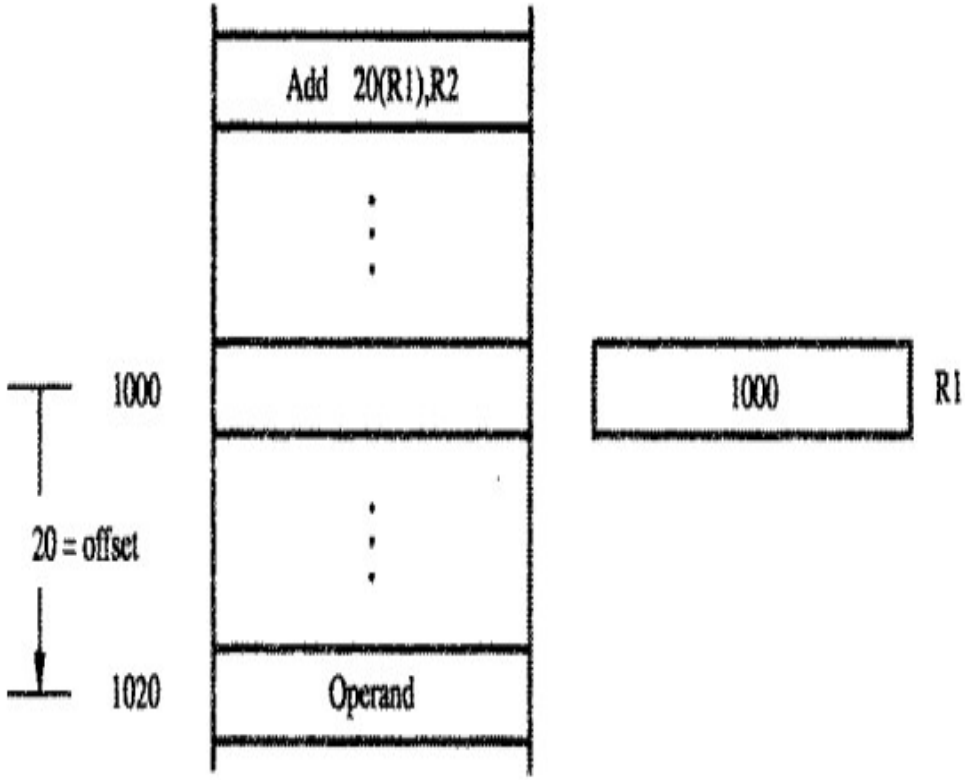
- The contents of the index register are not changed in the process of generating the effective address.
- In an assembly language program, the constant X may be given either as an explicit number or as a symbolic name representing a numerical value.

Addressing Modes

Hamacher, Vranesic & Zaky, "Computer Organization" (5th Ed), McGraw Hill.

INDEX MODES

- Figure 20 illustrates two ways of using the Index mode(offset is given explicitly in the instruction, and the other is stored in a register.)
- In figure 20a, the index register, R1, contains the address of a memory location, and the value X defines an offset (also called a displacement) from this address to the location where the operand is found.



(a) Offset is given as a constant

Figure 20 Indexed addressing.

Addressing Modes

Hamacher, Vranesic & Zaky, "Computer Organization" (5th Ed), McGraw Hill.

INDEX MODES

- An alternative use is illustrated in figure 20b. (offset is stored in a register.)
- The constant X corresponds to a memory address, and the contents of the index register define the offset to the operand.
- In both cases, the effective address is the sum of two values; one is given explicitly in the instruction, and the other is stored in a register.

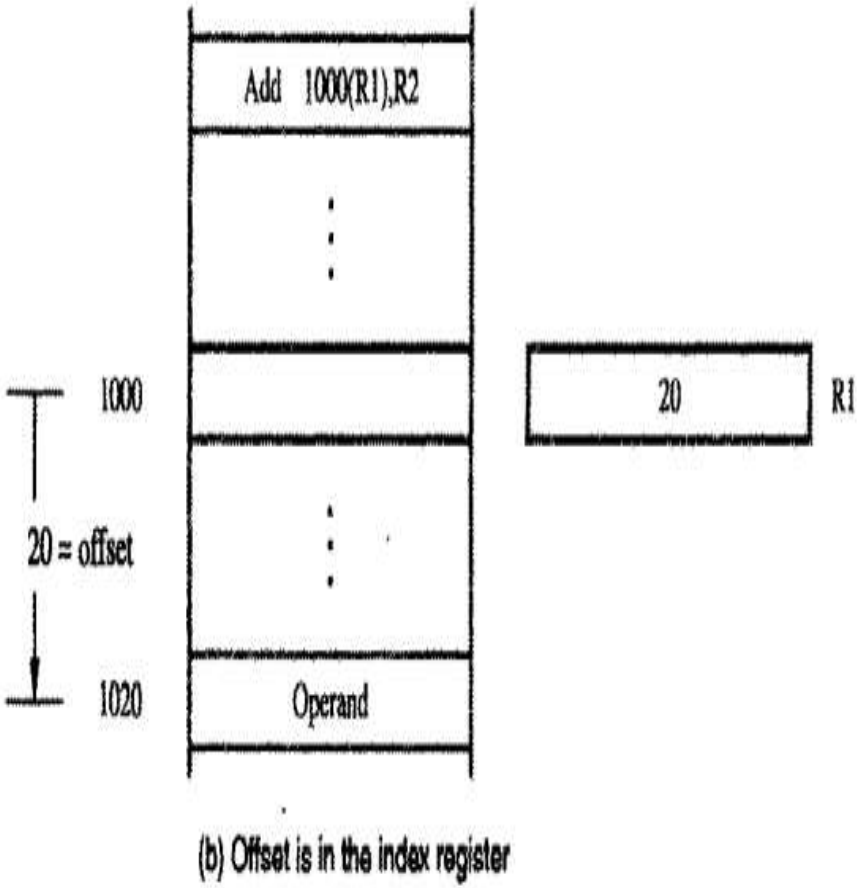
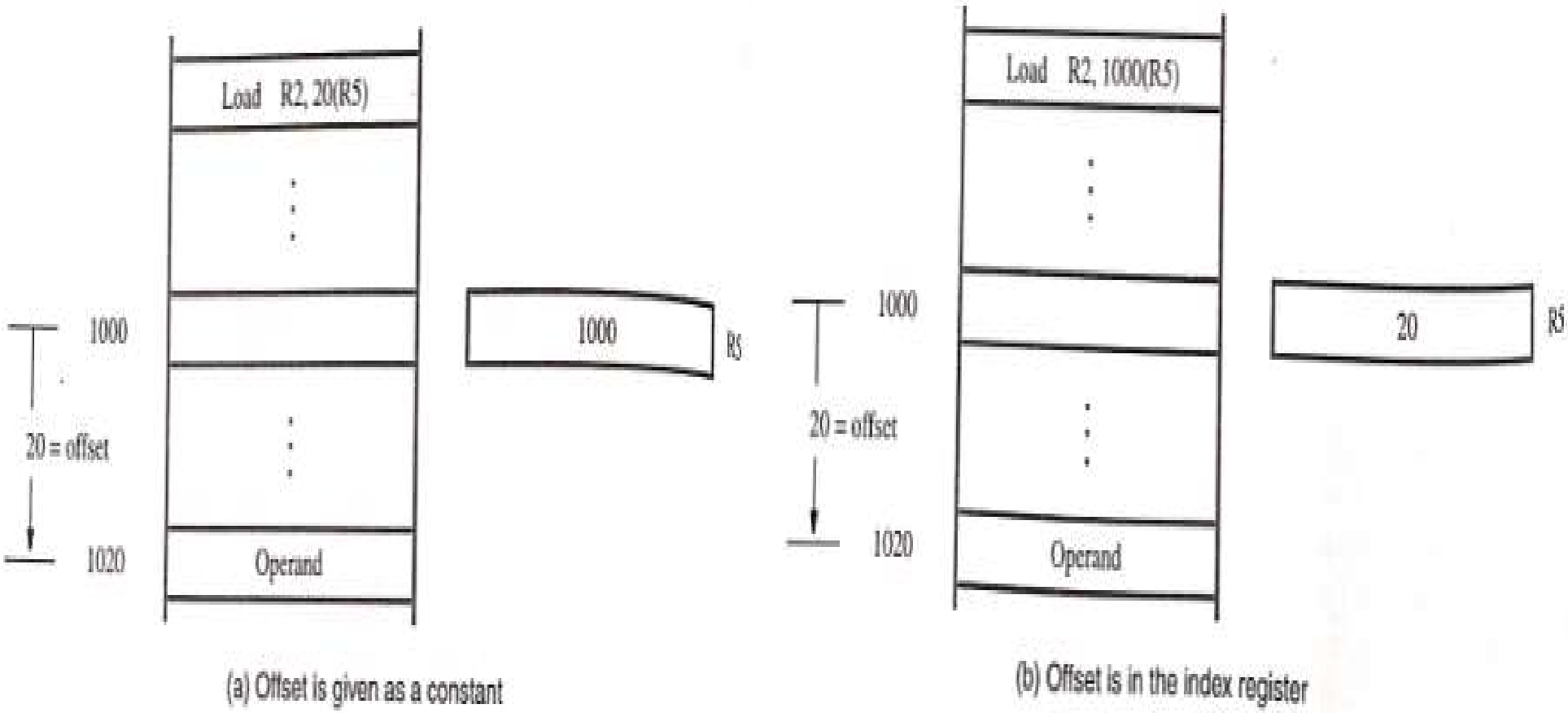


Figure 20 Indexed addressing.

Addressing Modes

Hamacher, Vranesic & Zaky, "Computer Organization and Embedded Systems" (4th Ed), McGraw Hill.



Indexed addressing.

Addressing Modes

Hamacher, Vranesic & Zaky, "Computer Organization and Embedded Systems" (4th Ed), McGraw Hill.

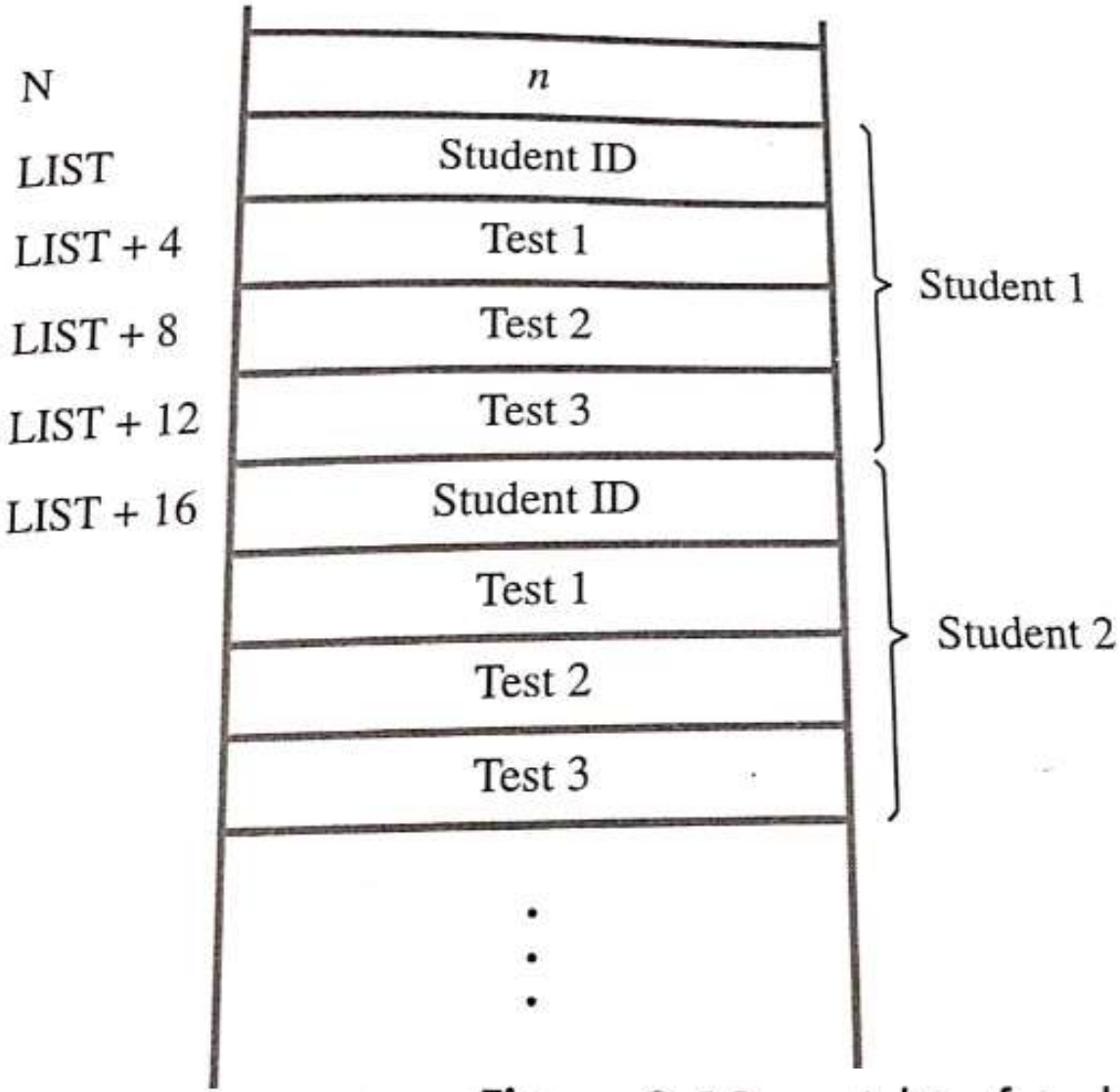
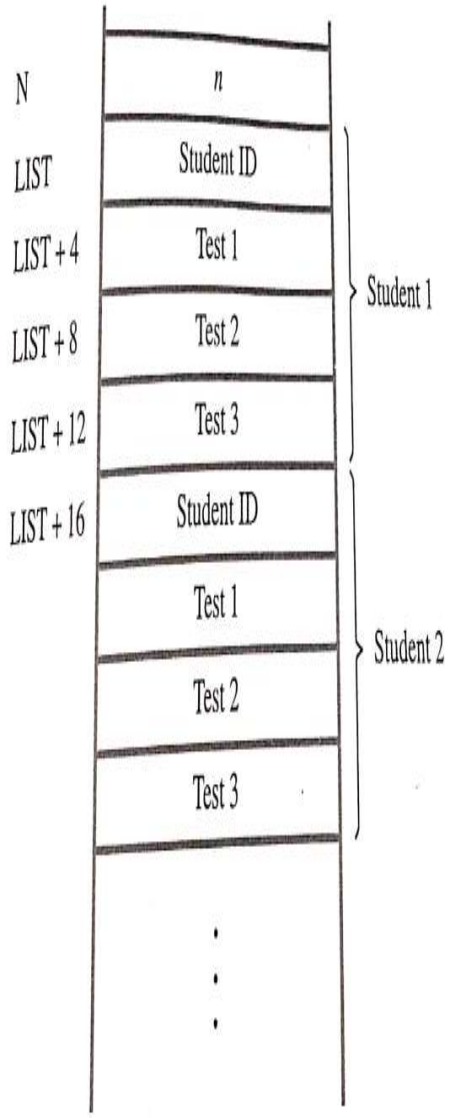


Figure 2.10 A list of students' marks.

Addressing Modes

Hamacher, Vranesic & Zaky, "Computer Organization and Embedded Systems" (4th Ed), McGraw Hill.




```
        Move        R2, #LIST       Get the address LIST.
        Clear       R3
        Clear       R4
        Clear       R5
        Load        R6, N           Load the value n.
LOOP:   Load        R7, 4(R2)       Add the mark for next student's
        Add         R3, R3, R7        Test 1 to the partial sum.
        Load        R7, 8(R2)       Add the mark for that student's
        Add         R4, R4, R7        Test 2 to the partial sum.
        Load        R7, 12(R2)      Add the mark for that student's
        Add         R5, R5, R7        Test 3 to the partial sum.
        Add         R2, R2, #16     Increment the pointer.
        Subtract    R6, R6, #1      Decrement the counter.
        Branch_if_[R6]>0 LOOP       Branch back if not finished.
        Store       R3, SUM1        Store the total for Test 1.
        Store       R4, SUM2        Store the total for Test 2.
        Store       R5, SUM3        Store the total for Test 3.
```

Figure 2.11 Indexed addressing used in accessing test scores in the list in Figure 2.10.

Addressing Modes

Hamacher, Vranesic & Zaky, "Computer Organization" (5th Ed), McGraw Hill.

INDEXING AND ARRAYS

- In the most basic form of indexed addressing several variations of the basic form provide a very efficient access to memory operands in practical programming situations.
- For example, a second register may be used to contain the offset X, in which case we can write the Index mode as

  (Ri, Rj)

  The effective address is the sum of the contents of registers Ri and Rj.
- The second register is usually called the base register.
- This form of indexed addressing provides more flexibility in accessing operands, because both components of the effective address can be changed.

Addressing Modes

Hamacher, Vranesic & Zaky, "Computer Organization" (5th Ed), McGraw Hill.

INDEXING AND ARRAYS

- Another version of the Index mode uses two registers plus a constant, which can be denoted as

  X(Ri, Rj)

- In this case, the effective address is the sum of the constant X and the contents of registers Ri and Rj.
- This added flexibility is useful in accessing multiple components inside each item in a record, where the beginning of an item is specified by the (Ri, Rj) part of the addressing mode.
- In other words, this mode implements a three-dimensional array.

Addressing Modes

Hamacher, Vranesic & Zaky, "Computer Organization" (5th Ed), McGraw Hill.

Table 1 Generic addressing modes

Name	Assembler syntax	Addressing function
Immediate	# Value	Operand = Value
Register	Ri	EA = Ri
Absolute (Direct)	LOC	EA = LOC
Indirect	(Ri)	EA = [Ri]
	(LOC)	EA = [LOC]
Index	X(Ri)	EA = [Ri] + X
Base with index	(Ri, Rj)	EA = [Ri] + [Rj]
Base with index and offset	X (Ri, Rj)	EA = [Ri] + [Rj] + X
Relative	X(PC)	EA = [PC] + X
Autoincrement	(Ri)+	EA = [Ri]; Increment Ri
Autodecrement	-(Ri)	Decrement Ri; EA = [Ri]

EA = effective address

Value = a signed number

Addressing Modes

Hamacher, Vranesic & Zaky, "Computer Organization" (5th Ed), McGraw Hill.

RELATIVE ADDRESSING

- A useful version of Index mode is obtained if the program counter, PC, is used instead of a general purpose register.
- Then, X(PC) can be used to address a memory location that is X bytes away from the location presently pointed to by the program counter.
- The addressed location is identified "relative" to the program counter, which always identifies the execution point in a program, the name "relative mode" is associated with this type of addressing.
- Relative mode – The effective address is determined by the Index mode using the program counter in place of the general-purpose register Ri.

  [The program counter (PC) is a register that manages the memory address of the instruction to be executed next.]

Addressing Modes

Hamacher, Vranesic & Zaky, "Computer Organization" (5th Ed), McGraw Hill.

RELATIVE ADDRESSING

- This mode can be used to access data operands.
- But, its most common use is to specify the target address in branch instructions.
- An instruction such as

  Branch>0 LOOP

  causes program execution to go to the branch target location identified by the name LOOP if the branch condition is satisfied.
- This location can be computed by specifying it as an offset from the current value of the program counter.
- Since the branch target may be either before or after the branch instruction, the offset is given as a signed number.

Addressing Modes

Hamacher, Vranesic & Zaky, "Computer Organization" (5th Ed), McGraw Hill.

ADDITIONAL MODES

- Autoincrement mode – the effective address of the operand is the contents of a register specified in the instruction. After accessing the operand, the contents of this register are automatically to point to the next item in a list.
- The autoincrement mode is written as (Ri)+

- Autodecrement mode – the contents of a register specified in the instruction are first automatically decremented and are then used as the effective address of the operand.
- The autodecrement mode is written as -(Ri)

Addressing Modes

Hamacher, Vranesic & Zaky, "Computer Organization" (5th Ed), McGraw Hill.

Table 1 Generic addressing modes

Name	Assembler syntax	Addressing function
Immediate	# Value	Operand = Value
Register	Ri	EA = Ri
Absolute (Direct)	LOC	EA = LOC
Indirect	(Ri)	EA = [Ri]
	(LOC)	EA = [LOC]
Index	X(Ri)	EA = [Ri] + X
Base with index	(Ri, Rj)	EA = [Ri] + [Rj]
Base with index and offset	X (Ri, Rj)	EA = [Ri] + [Rj] + X
Relative	X(PC)	EA = [PC] + X
Autoincrement	(Ri)+	EA = [Ri]; Increment Ri
Autodecrement	-(Ri)	Decrement Ri; EA = [Ri]

EA = effective address

Value = a signed number

Addressing Modes

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.

Figure 13 Selection of addressing modes with examples, meaning, and usage. In autoincrement/-decrement and scaled addressing modes, the variable *d* designates the size of the data item being accessed (i.e., whether the instruction is accessing 1, 2, 4, or 8 bytes). These addressing modes are only useful when the elements being accessed are adjacent in memory. RISC computers use displacement addressing to simulate register indirect with 0 for the address and to simulate direct addressing using 0 in the base register. In our measurements, we use the first name shown for each mode.

[J. Hennessy and D. Patterson, "Computer Architecture, A quantitative approach", 5th Edition.]

Addressing Modes

- We have kept addressing modes that depend on the program counter, called PC relative addressing, separate.
- PC-relative addressing is used primarily for specifying code addresses in control transfer instructions.

Relative	X(PC)	EA = [PC] + X

- The left arrow (←) is used for assignment.
- We use the array Mem as the name for main memory and the array Regs for registers.
- Mem[Regs[R1]] refers to the contents of the memory location whose address is given by the contents of register 1 (R1).

Addressing Modes

- Figure 14 shows the results of measuring addressing mode usage patterns in three programs on the VAX architecture.
- We use the old VAX architecture for a few measurements because it has the richest set of addressing modes and the fewest restrictions on memory addressing.
- As Figure 14 shows, displacement and immediate addressing dominate addressing mode usage.

Addressing Modes

[J. Hennessy and D. Patterson, “Computer Architecture, A quantitative approach”, 5th Edition.]

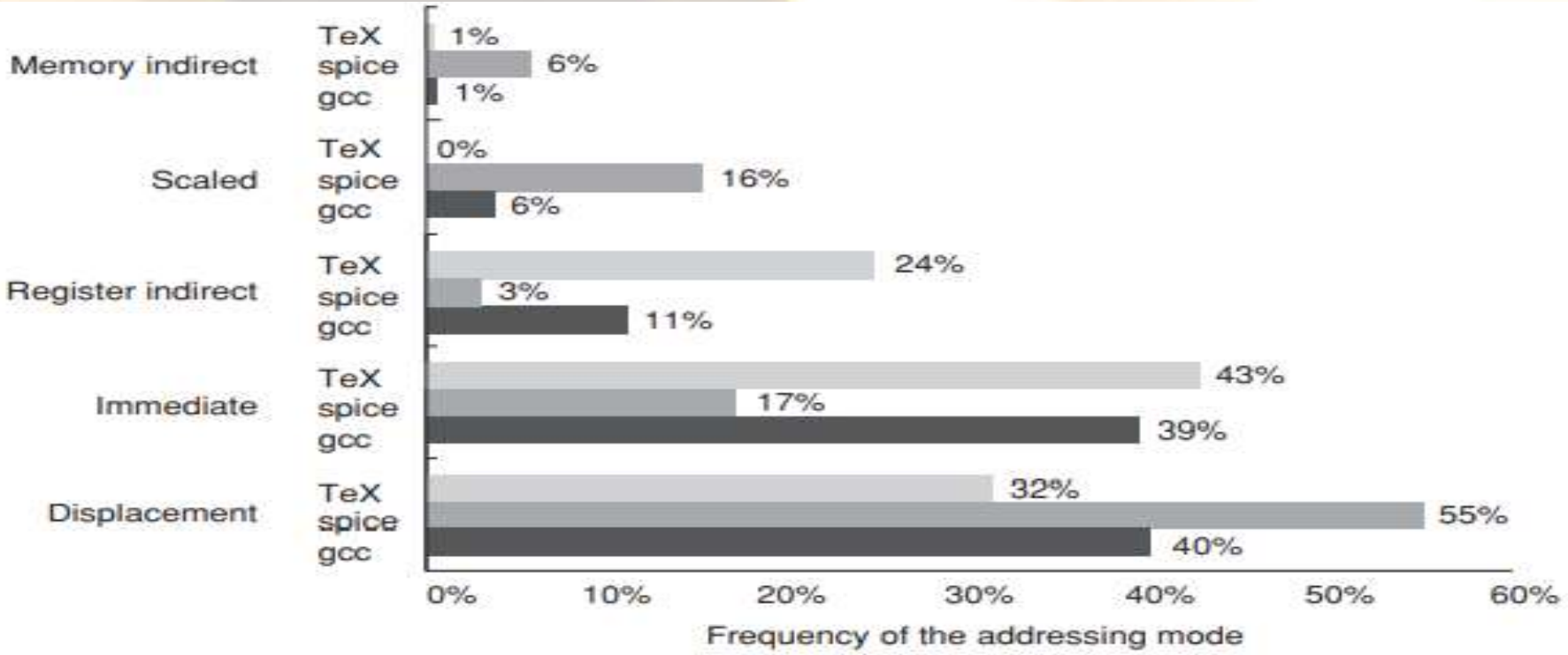


Figure 14 Summary of use of memory addressing modes (including immediates). These major addressing modes account for all but a few percent (0% to 3%) of the memory accesses. Register modes, which are not counted, account for one-half of the operand references, while memory addressing modes (including immediate) account for the other half. Of course, the compiler affects what addressing modes are used;

The memory indirect mode on the VAX can use displacement, autoincrement, or autodecrement to form the initial memory address; in these programs, almost all the memory indirect references use displacement mode as the base. Displacement mode includes all displacement lengths (8, 16, and 32 bits). The PC-relative addressing modes, used almost exclusively for branches, are not included. Only the addressing modes with an average frequency of over 1% are shown.

Back

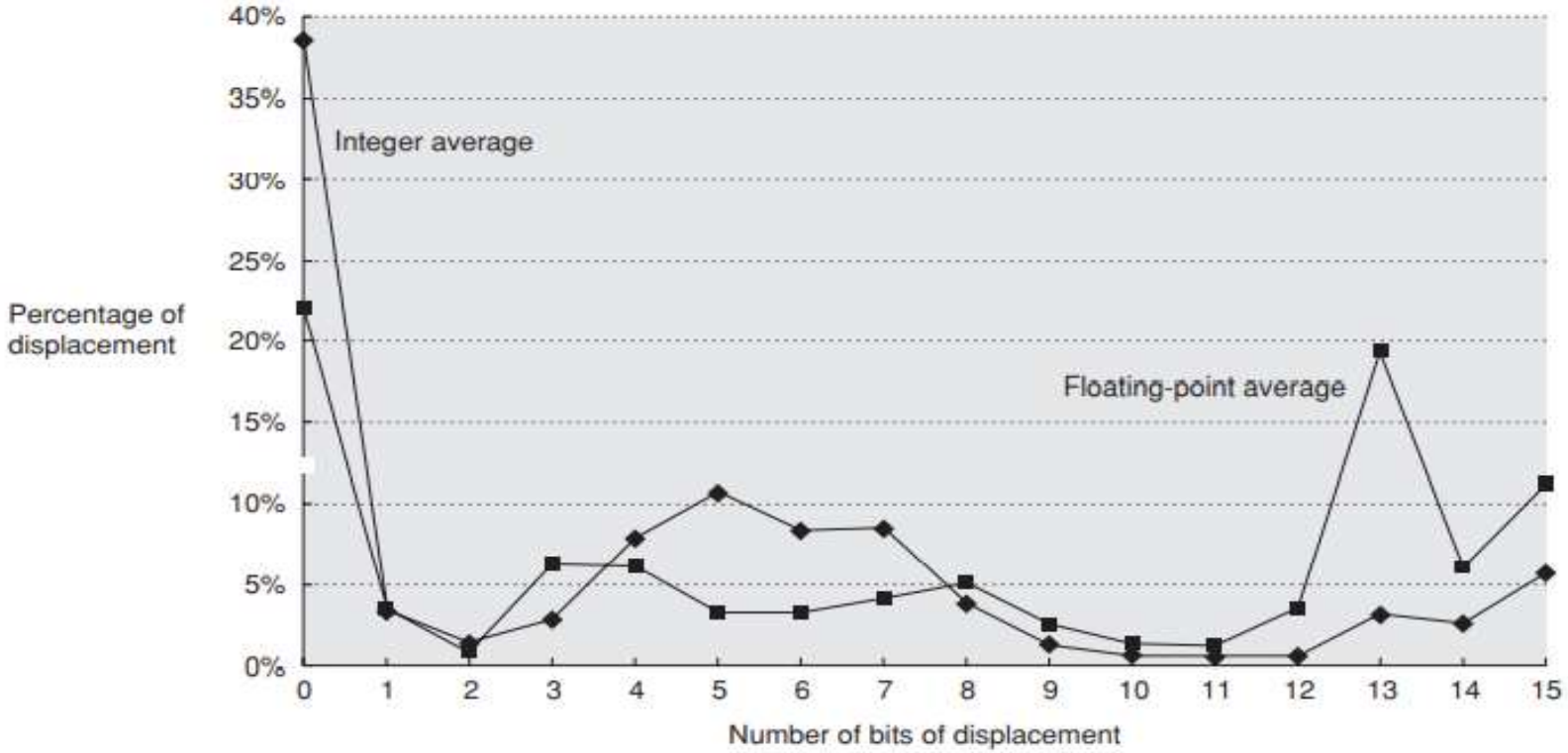
Addressing Modes

Displacement Addressing Mode

- The major question that arises for a displacement-style addressing mode is that of the range of displacements used.
- Based on the use of various displacement sizes, a decision of what sizes to support can be made.
- Choosing the displacement field sizes is important because they directly affect the instruction length.
- Figure 15 shows the measurements taken on the data access on a load-store architecture using the benchmark programs.
- We look at branch offsets - data accessing patterns and branches are different; little is gained by combining them, although in practice the immediate sizes are made the same for simplicity.

Addressing Modes

Displacement Addressing Mode



Back

Figure 15 **Displacement values are widely distributed.** There are both a large number of small values and a fair number of large values. The wide distribution of displacement values is due to multiple storage areas for variables and different displacements to access them as well as the overall addressing scheme the compiler uses. The *x*-axis is $\log_2$ of the displacement; that is, the size of a field needed to represent the magnitude of the displacement. Zero on the *x*-axis shows the percentage of displacements of value 0. The graph does not include the sign bit, which is heavily affected by the storage layout. Most displacements are positive, but a majority of the largest displacements (14+ bits) are negative. Since these data were collected on a computer with 16-bit displacements, they cannot tell us about longer displacements. These data were taken on the Alpha architecture with full optimization for SPEC CPU2000, showing the average of integer programs (CINT2000) and the average of floating-point programs (CFP2000).

[J. Hennessy and D. Patterson, "Computer Architecture, A quantitative approach", 5th Edition.]

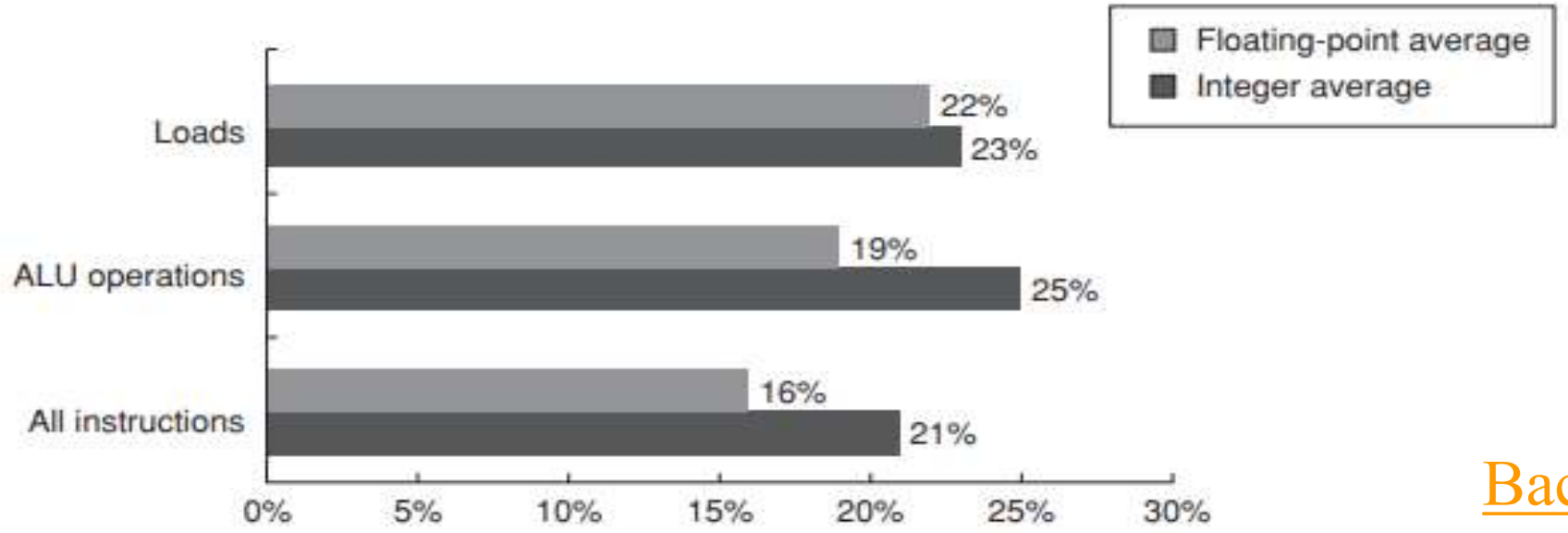
Addressing Modes

Immediate or Literal Addressing Mode

- Immediates can be used in arithmetic operations, in comparisons (primarily for branches), and in moves where a constant is wanted in a register.
- The last case occurs for constants written in the code—which tend to be small—and for address constants, which tend to be large.
- For the use of immediates it is important to know whether they need to be supported for all operations or for only a subset.
- Figure 16 shows the frequency of immediates for the general classes of integer and floating-point operations in an instruction set.

Addressing Modes

Immediate or Literal Addressing Mode



Back

Figure 16 About one-quarter of data transfers and ALU operations have an immediate operand. The bottom bars show that integer programs use immediates in about one-fifth of the instructions, while floating-point programs use immediates in about one-sixth of the instructions. For loads, the load immediate instruction loads 16 bits into either half of a 32-bit register. Load immediates are not loads in a strict sense because they do not access memory. Occasionally a pair of load immediates is used to load a 32-bit constant, but this is rare. (For ALU operations, shifts by a constant amount are included as operations with immediate operands.)

[J. Hennessy and D. Patterson, "Computer Architecture, A quantitative approach", 5th Edition.]

Addressing Modes

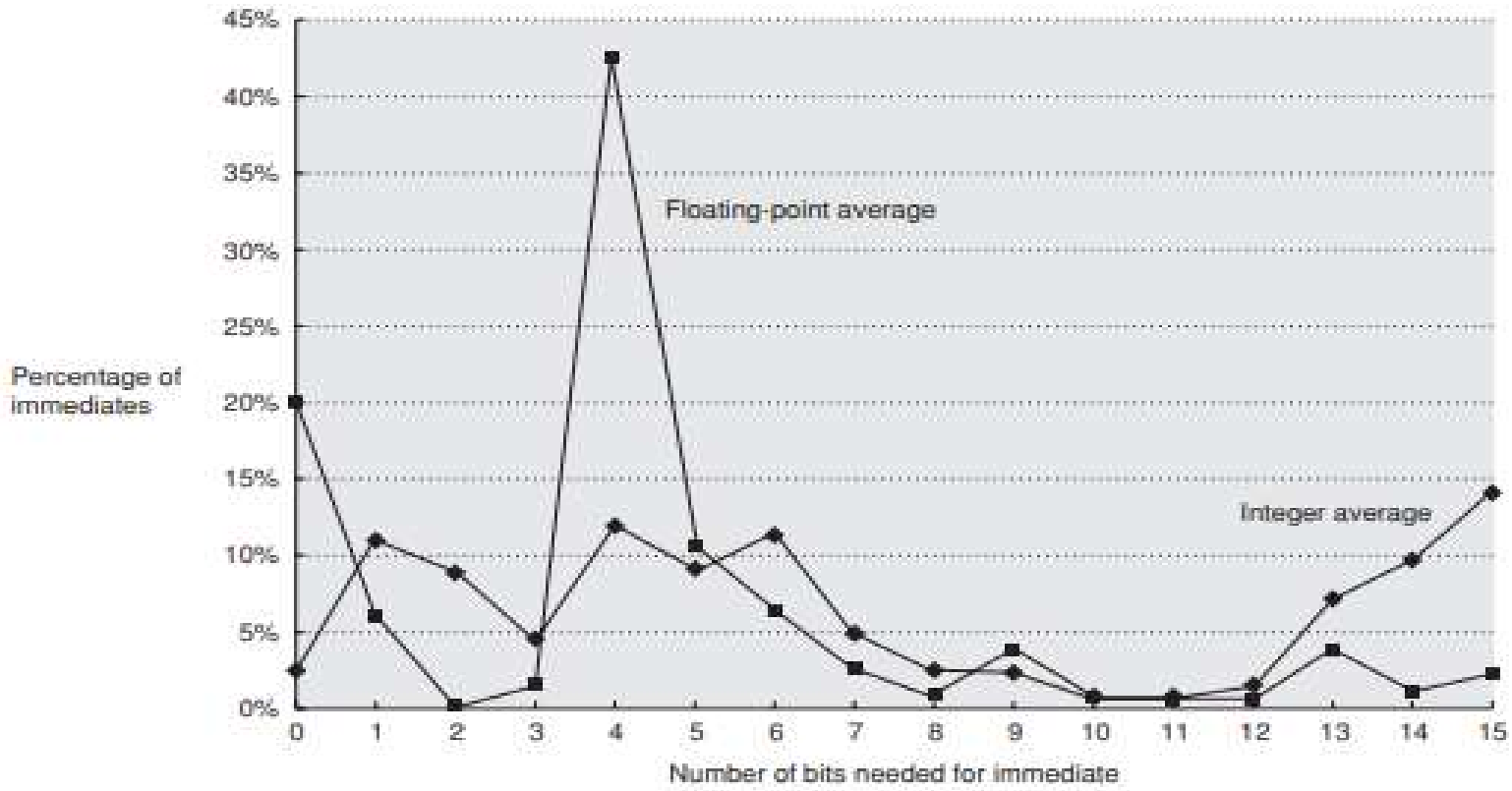
Immediate or Literal Addressing Mode

- Another important instruction set measurement is the range of values for immediates.
- Like displacement values, the size of immediate values affects instruction length.
- As Figure 17 shows, small immediate values are most heavily used.
- Large immediates are sometimes used most likely in addressing calculations

Addressing Modes

[J. Hennessy and D. Patterson, “Computer Architecture, A quantitative approach”, 5th Edition.]

Immediate or Literal Addressing Mode



Back

Figure 17 The distribution of immediate values. The *x*-axis shows the number of bits needed to represent the magnitude of an immediate value—0 means the immediate field value was 0. The majority of the immediate values are positive. About 20% were negative for CINT2000, and about 30% were negative for CFP2000. These measurements were taken on an Alpha, where the maximum immediate is 16 bits, for the same programs . A similar measurement on the VAX, which supported 32-bit immediates, showed that about 20% to 25% of immediates were longer than 16 bits. Thus, 16 bits would capture about 80% and 8 bits about 50%.

Addressing Modes

Summary: Memory Addressing

- First, because of their popularity, we would expect a new architecture to support at least the following addressing modes: displacement, immediate, and register indirect. [They represent 75% to 99% of the addressing modes used in our measurements(refer figure 14)]
- Second, we would expect the size of the address for displacement mode to be at least 12–16 bits. [Figure 15 suggests these sizes would capture 75% to 99% of the displacements]
- Third, we would expect the size of the immediate field to be at least 8–16 bits.

Encoding an Instruction Set

- The instructions are encoded into a binary representation for execution by the processor.
- This representation affects not only the size of the compiled program; it affects the implementation of the processor, which must decode this representation to quickly find the operation and its operands.
- The operation is typically specified in one field, called the opcode.
- The important decision is how to encode the addressing modes with the operations.

Encoding an Instruction Set

- This decision depends on the range of addressing modes and the degree of independence between opcodes and addressing modes.
- Some older computers have one to five operands with 10 addressing modes for each operand.
- For such a large number of combinations, typically a separate address specifier is needed for each operand.
- The address specifier tells what addressing mode is used to access the operand.
- At the other extreme are load-store computers with only one memory operand and only one or two addressing modes; obviously, in this case, the addressing mode can be encoded as part of the opcode.

Encoding an Instruction Set

- When encoding the instructions, the number of registers and the number of addressing modes both have a significant impact on the size of instructions, as the register field and addressing mode field may appear many times in a single instruction.
- In fact, for most instructions many more bits are consumed in encoding addressing modes and register fields than in specifying the opcode.

Encoding an Instruction Set

- The architect must balance several competing forces when encoding the instruction set.
    1. The desire to have as many registers and addressing modes as possible.
    2. The impact of the size of the register and addressing mode fields on the average instruction size and hence on the average program size.
    3. A desire to have instructions encoded into lengths that will be easy to handle in a pipelined implementation.
- As a minimum, the architect wants instructions to be in multiples of bytes, rather than an arbitrary bit length. Many desktop and server architects have chosen to use a fixed-length instruction to gain implementation benefits while sacrificing average code size.

Encoding an Instruction Set

- Figure 21 shows three popular choices for encoding the instruction set.
- The first we call variable, since it allows virtually all addressing modes to be with all operations.
- This style is best when there are many addressing modes and operations.

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

Figure 21 Three basic variations in instruction encoding: variable length, fixed length, and hybrid.

[J. Hennessy and D. Patterson, "Computer Architecture, A quantitative approach", 5th Edition.]

Encoding an Instruction Set

- The second choice we call fixed, since it combines the operation and the addressing mode into the opcode.
- Often fixed encoding will have only a single size for all instructions; it works best when there are few addressing modes and operations.
- The trade-off between variable encoding and fixed encoding is size of programs versus ease of decoding in the processor.
- Variable tries to use as few bits as possible to represent the program, but individual instructions can vary widely in both size and the amount of work to be performed.

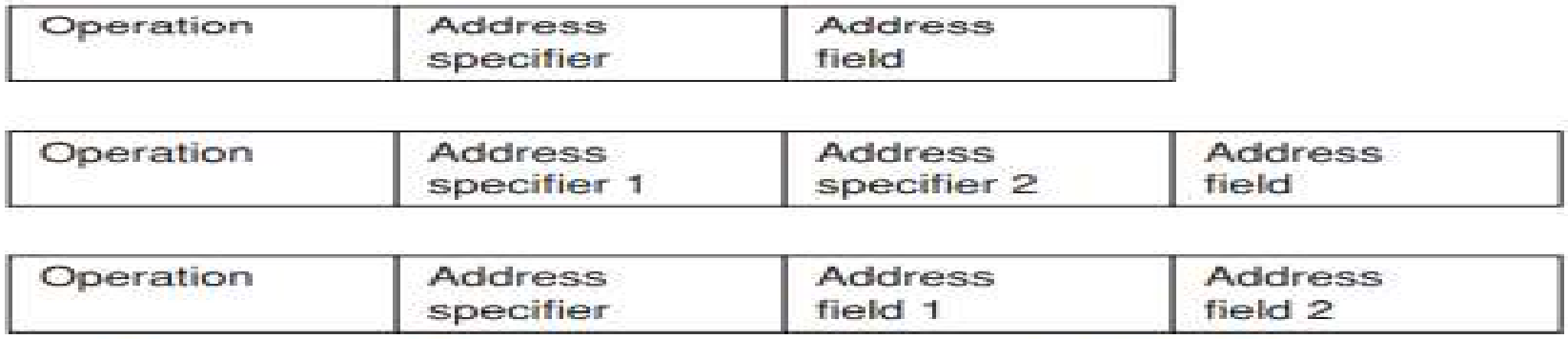
(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)

Figure 21 Three basic variations in instruction encoding: variable length, fixed length, and hybrid.

[J. Hennessy and D. Patterson, "Computer Architecture, A quantitative approach", 5th Edition.]

Encoding an Instruction Set

- Given the two poles of instruction set design of variable and fixed, the third alternative immediately springs to mind.
- Reduce the variability in size and work of the variable architecture but provide multiple instruction lengths to reduce code size.
- This hybrid approach is the third encoding alternative.



(c) Hybrid (e.g., IBM 360/370, MIPS16, Thumb, TI TMS320C54x)

Figure 21 Three basic variations in instruction encoding: variable length, fixed length, and hybrid.

[J. Hennessy and D. Patterson, "Computer Architecture, A quantitative approach", 5th Edition.]

Encoding an Instruction Set

- The variable format can support any number of operands, with each address specifier determining the addressing mode and the length of the specifier for that operand.
- It generally enables the smallest code representation, since unused fields need not be included.
- The fixed format always has the same number of operands, with the addressing modes (if options exist) specified as part of the opcode.
- It generally results in the largest code size. Although the fields tend not to vary in their location, they will be used for different purposes by different instructions.
- The hybrid approach has multiple formats specified by the opcode, adding one or two fields to specify the addressing mode and one or two fields to specify the operand address.

Encoding an Instruction Set

Reduced Code Size in RISCs

- As RISC computers started being used in embedded applications, the 32-bit fixed format became a liability since cost and hence smaller code are important.
- In response, several manufacturers offered a new hybrid version of their RISC instruction sets, with both 16-bit and 32-bit instructions.
- In contrast to the instruction set extensions, IBM simply compresses its standard instruction set, and then adds hardware to decompress instructions as they are fetched from memory on an instruction cache miss.
- Thus, the instruction cache contains full 32-bit instructions, but compressed code is kept in main memory, ROMs, and the disk.
- Hitachi simply invented a RISC instruction set with a fixed 16-bit format, called SuperH, for embedded applications.

RISC(Reduced Instruction Set Computers)-each instruction must fit into a single word- reduce complexity

Encoding an Instruction Set

Summary

- Decisions made in the components of instruction set design determine whether the architect has the choice between variable and fixed instruction encodings.

- The architect more interested in code size than performance will pick variable encoding, and the one more interested in performance than code size will pick fixed encoding.