Instruction Set Principles and Examples

1Q. Explain various Instruction Set Architectures
2.1 Classifying Instruction Set Architectures

There are four types of internal storages uses by the processor to store operands explicitly and implicitly for execution of a program

Stack
Accumulator
Set of Registers (Register-Memory)
Set of Registers (Register-Register/load-store)

The operands in stack architecture are implicitly on the top of the stack, and in an accumulator architecture one operand is implicitly the accumulator. The general-purpose register architectures (Register-Memory and Register-Register) have only explicit operands, either in registers or memory locations.

Figure 2.1 shows a block diagram of such architectures

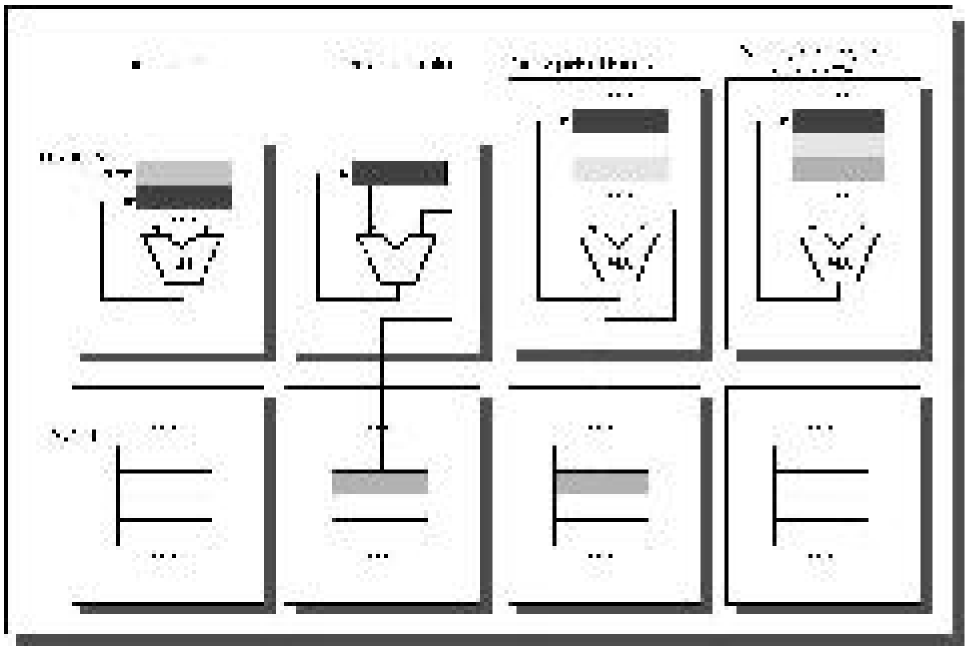


FIGURE 2.1 Operand locations for four instruction set architecture classes.

Figure 2.2 shows how the code sequence C=A+B would typically appear in these four classes of instruction sets. The explicit operands may be accessed directly from memory or may need to be first loaded into temporary storage, depending on the class of architecture and choice of specific instruction.

Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A	Load A	Load R1,A	Load R1,A
Push B	Add B	Add R3,R1,B	Load R2,B
Add	Store C	Store R3,C	Add R3,R1,R2
Pop C			Store R3,C

FIGURE 2.2 The code sequence for C = A + B for four classes of instruction sets.

As the figures show, there are really two classes of register computers.

1. Register-memory architecture: Access memory as part of any instruction,
2. Load-Store or Register-Register architecture: Access memory only with load and store instructions

A third class, memory-memory architecture: all operands stored in memory.

Some instruction set architectures have more registers than a single accumulator, but place restrictions on uses of these special registers. Such architecture is sometimes called an extended accumulator or special-purpose register computer.

Early computers used stack or accumulator-style architectures, virtually every new architecture designed after 1980 uses load-store register architecture. The major reasons for the emergence of general-purpose register (GPR) computers are twofold. First, registers—like other forms of storage internal to the processor are faster than memory. Second, registers are more efficient for a compiler to use than other forms of internal storage.

For example, on a register computer the expression (A*B)–(B*C)–(A*D) may be evaluated by doing the multiplications in any order, which may be more efficient because of the location of the operands or because of pipelining concerns.

Two major instruction set characteristics divide GPR architectures. Both characteristics concern the nature of operands for a typical arithmetic or logical instruction (ALU instruction). The first concerns whether an ALU instruction has two or three operands. In the three-operand format, the instruction contains one result operand and two source operands. In the two-operand format, one of the operands is both a source and a result for the operation. The second distinction among GPR architectures concerns how many of the operands may be memory addresses in ALU instructions. The number of memory operands supported by a typical ALU instruction may vary from none to three. Figure 2.3 shows combinations of these two attributes with examples of computers. Although there are seven possible combinations, three serve to classify nearly all existing computers. As we mentioned earlier, these three are register-register (also called load-store), register-memory, and memory-memory.

Number of memory addresses	Maximum number of operands allowed	Type of architecture	Examples
0	3	Register-register	Alpha, ARM, MIPS, PowerPC, SPARC, SuperH, Trimedia TM5200
1	2	Register-memory	IBM 360/370, Intel 80x86, Motorola 68000, TI TMS320C54x
2	2	Memory-memory	VAX (also has three-operand formats)
3	3	Memory-memory	VAX (also has two-operand formats)

FIGURE 2.3 Typical combinations of memory operands and total operands per typical ALU instruction with examples of computers.

Figure 2.4 shows the advantages and disadvantages of each of these alternatives.

Type	Advantages	Disadvantages
Register-register (0,3)	Simple, fixed-length instruction encoding. Simple code-generation model. Instructions take similar numbers of clocks to execute	Higher instruction count than architectures with memory references in instructions. More instructions and lower instruction density leads to larger programs.
Register-memory (1,2)	Data can be accessed without a separate load instruction first. Instruction format tends to be easy to encode and yields good density.	Operands are not equivalent since a source operand in a binary operation is destroyed. Encoding a register number and a memory address in each instruction may restrict the number of registers. Clocks per instruction vary by operand location.
Memory-memory (2,2) or (3,3)	Most compact. Doesn't waste registers for temporaries.	Large variation in instruction size, especially for three-operand instructions. In addition, large variation in work per instruction. Memory accesses create memory bottleneck.

FIGURE 2.4 Advantages and disadvantages of the three most common types of general-purpose register computers. The notation (m, n) means m memory operands and n total operands.

2Q. Explain Memory addressing concept using in Instruction set Design

2.2 Memory Addressing

The memory addressing means how the object is accessed as a function of the address the length. There are two memory address issues, first one is ordering of bytes and words.

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:

7	6	5	4	3	2	1	0

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:

0	1	2	3	4	5	6	7

Little Endian ordering also fails to match normal ordering of words when strings are compared. Strings appear “SDRAWKCAB” (backwards) in the registers.

A second memory issue is that 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 2.5 shows the addresses at which an access is aligned or misaligned.

Width of object:	**Value of 3 low order bits of byte address: 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		Misal igned		Misaligned		Misalig.
4 Bytes (Word)	Aligned				Aligned			
4 Bytes (Word)		Misaligned				Misaligned		
4 Bytes (Word)			Misaligned				Misaligned	
4 Bytes (Word)				Misaligned				Misalig.
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)
Misalig.

FIGURE 2.5 Aligned and misaligned addresses of byte, half word, word, and double word objects for byte addressed computers.

Figure 2.5 above, suppose we read a byte from an address with its three low order bits having the value 4. We will need shift right 3 bytes to align the byte to the proper place in a 64-bit register. Depending on the instruction, the computer may also need to sign-extend the quantity. Stores are easy: only the addressed bytes in memory may be altered. On some computers a byte, half word, and word operation does not affect the upper portion of a register. Although all the computers discussed in this book permit byte, half-word, and word accesses to memory, only the IBM 360/370, Intel 80x86, and VAX supports ALU operations on register operands narrower than the full width.

The way addresses are specified by instructions is called addressing modes. Addressing mode 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.

Figure 2.6 shows the most common names for the addressing modes, though the names differ among architectures. In this figure, only one non-C feature is used: The left arrow (←) is used for assignment. We also use the array Mem as the name for main memory and the array Regs for registers. Thus, Mem[Regs[R1]]refers to the contents of the memory location whose address is given by the contents of register 1 (R1). Later, we will introduce extensions for accessing and transferring data smaller than a word.

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.

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 R3is 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. R2points to start of array; each reference increments R2by 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 2.6 Selection of addressing modes with examples, meaning, and usage.

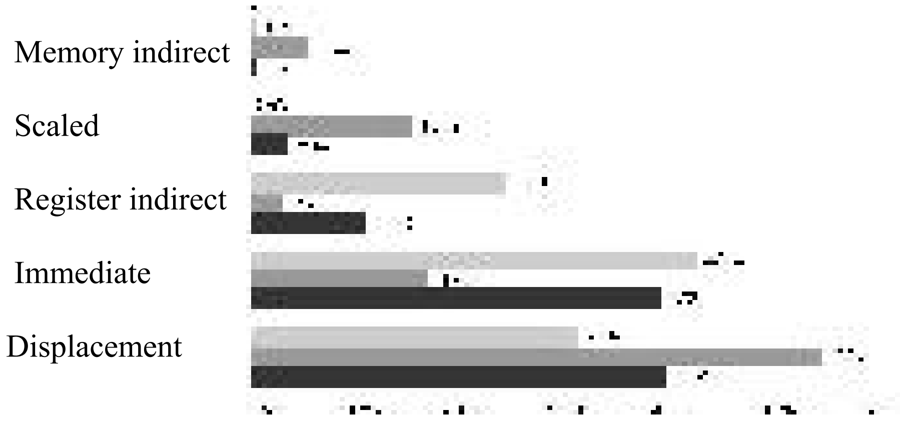
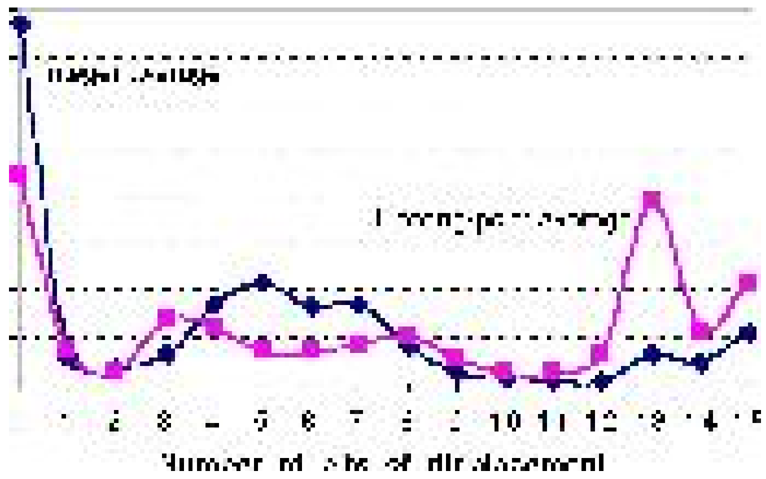


FIGURE 2.7 Summary of use of memory addressing modes

Displacement Addressing Mode
In Displacement-style addressing mode filed size is important because they directly affect the instruction length. Figure 2.8 shows the measurements taken on the data access on a load-store architecture using our benchmark programs.



Immediate or Literal Addressing Mode:

Immediates can be used in arithmetic operations, in comparisons and in moves where a constant is wanted in a register. Figure 2.9 show the frequency of immeidates for the general classes of integer operations in an instruction set. As figure 2.10 shows, small immediate values are most heavily used. Large immediates are sometimes used, however most likely in addressing instructions.

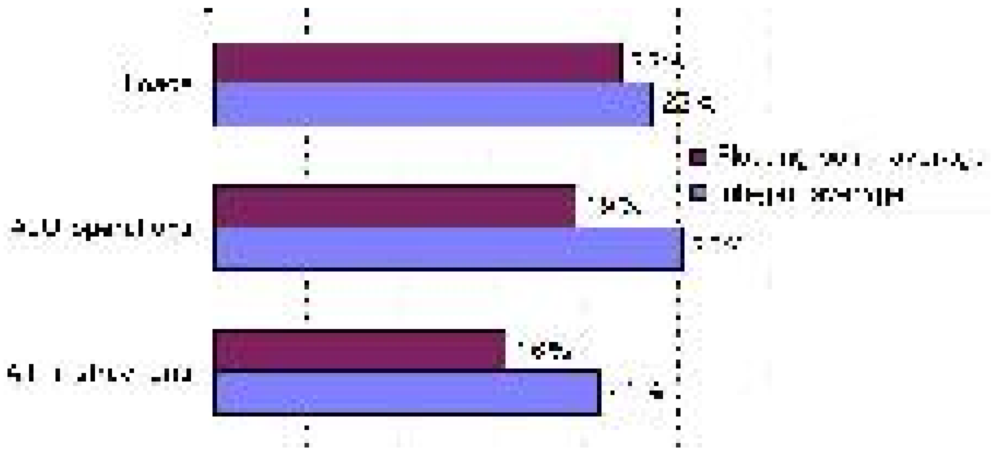
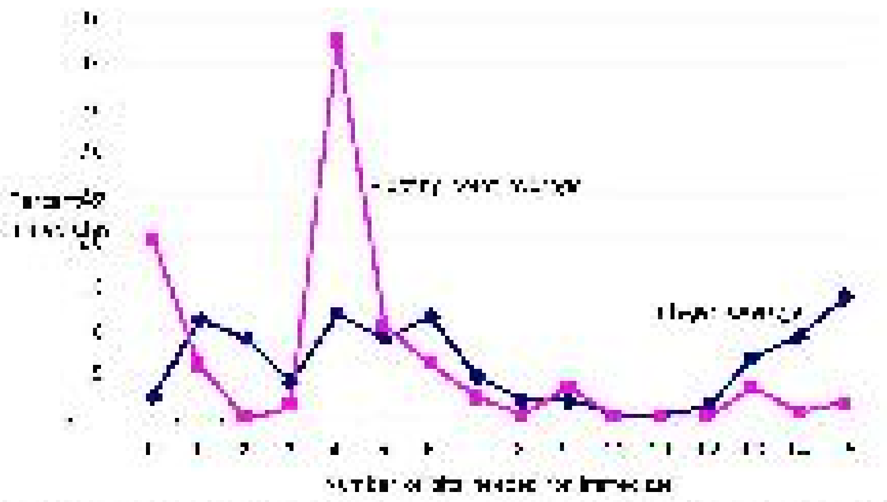


FIGURE 2.9 About one-quarter of data transfers and ALU operations have an immediate operand.



[illegible]

3Q. Explain various addressing model used for signal procesing
2.3 Addressing Modes for Signal Processing

Digital Signal Processors (DSPs) deal with infinite, continuous streams of data, they routinely rely on circular buffers. Hence, as data is added to the buffer, a pointer is checked to see if it is pointing at the end of the buffer. If not, it increments the pointer to the next address; if it is, the pointer is set instead to the start of the buffer. Similar issues arise when emptying a buffer.

Every DSP has a modulo or circular addressing mode to handle this case automatically, our first novel DSP addressing mode. It keeps a start register and an end register with every address register, allowing the auto-increment and auto-decrement addressing modes to reset when the reach the end of the buffer. One variation makes assumptions about the buffer size starting at an address that ends in “xxx00.00” and so uses just a single buffer length register per address register.

Even though DSPs are tightly targeted to a small number of algorithms. Fast Fourier Transform (FFT). FFTs start or end their processing with data shuffled in a particular order. For eight data items in a radix-2 FFT, the transformation is listed below, with addresses in parentheses shown in binary:

0 (000_2) => 0 (000_2)
1 (001_2) => 4 (100_2)
2 (010_2) => 2 (010_2)
3 (011_2) => 6 (110_2)
4 (100_2) => 1 (001_2)
5 (101_2) => 5 (101_2)
6 (110_2) => 3 (011_2)
7 (111_2) => 7 (111_2)

Without special support such address transformation would take an extra memory access to get the new address, or involve a fair amount of logical instructions to transform the address.

The DSP solution is based on the observation that the resulting binary address is simply the reverse of the initial address! For example, address 100_2 (4) becomes 001_2(1). Hence, many DSPs have this second novel addressing mode— bit reverse addressing— whereby the hardware reverses the lower bits of the address, with the number of bits reversed depending on the step of the FFT algorithm.

Figure 2.11 shows the static frequency of data addressing modes in a DSP for a set of 54 library routines. This architecture has 17 addressing modes, yet the 6 modes also found in Figure 2.6 on page 108 for desktop and server computers account for 95% of the DSP addressing. Despite measuring hand-coded routines to derive Figure 2.11, the use of novel addressing mode is sparse.

Addressing Mode	**Assembly Symbol**	**Percent**
Immediate	#num	30.02%
Displacement	ARx(num)	10.82%
Register indirect	*ARx	17.42%
Direct	num	11.99%
Autoincrement, pre increment (increment register before use contents as address)	*+ARx	0
Autoincrement, post increment (increment register after use contents as address)	*ARx+	18.84%
Autoincrement, pre increment with 16b immediate	*+ARx(num)	0.77%
Autoincrement, pre increment, with circular addressing	*ARx+%	0.08%
Autoincrement, post increment with 16b immediate, with circular addressing	*ARx+(num)%	0
Autoincrement, post increment by contents of AR0	*ARx+0	1.54%
Autoincrement, post increment by contents of AR0, with circular addressing	*ARx+0%	2.15%
Autoincrement, post increment by contents of AR0, with bit reverse addressing	*ARx+0B	0
Autodecrement, post decrement (decrement register after use contents as address	*ARx	6.08%
Autodecrement, post decrement, with circular addressing	*ARx-%	0.04%
Autodecrement, post decrement by contents of AR0	*ARx-0	0.16%
Autodecrement, post decrement by contents of AR0, with circular addressing	*ARx-0%	0.08%
Autodecrement, post decrement by contents of AR0, with bit reverse addressing	*ARx-0B	0
Total		100.00%

FIGURE 2.11 Frequency of addressing modes for TI TMS320C54x DSP.

4Q. Explain various types and sizes of Operands using in Instruction Set Design
2.4 Type and Size of Operands

The type of an operand designates by encoding in the opcode or the data can be annotated with tags that are interpreted by the hardware. These tags specify the type of the operand, and the operation is chosen accordingly.

In desktop and server architectures the type of an operand usually
Integer
Single-precision floating point
Character, and so on

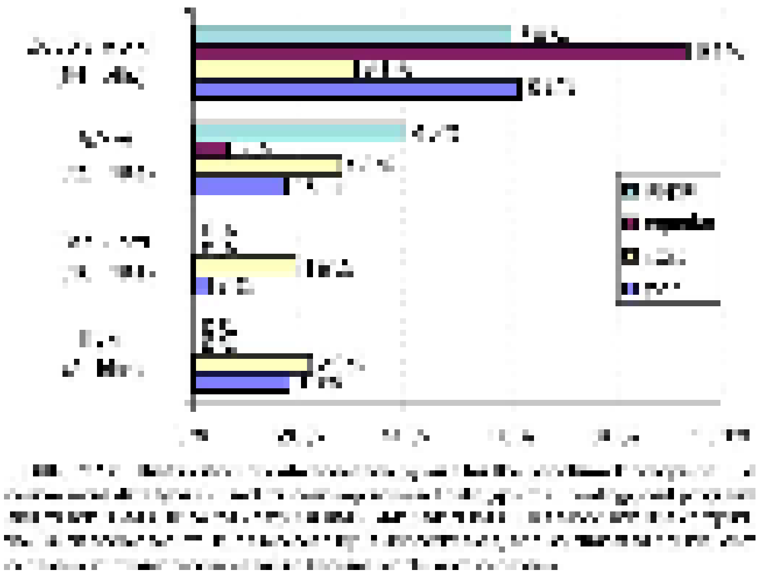
The size of operands generally
Character (8 bits)
Half word (16 bits)
Word (32 bits)
Single-precision floating point (also 1 word)
Double-precision floating point (2 words)

Integers are almost universally represented as two's complement binary numbers. Characters are usually in ASCII, but the 16bit Unicode is gaining popularity with the internationalization of computers. Until the early 1980s, most computer manufacturers chose their own floating-point representation. Almost all computers since that time follow the same standard for floating point, the IEEE standard 754.

Some architectures provide operations on character strings, although such operations are usually quite limited and treat each byte in the string as a single character. Typical operations supported on character strings are comparisons and moves.

For business applications, some architectures support a decimal format, usually called packed decimal or binary-coded decimal—4 bits are used to encode the values 0–9, and 2 decimal digits are packed into each byte. Numeric character strings are sometimes called unpacked decimal, and operations—called packing and unpacking—are usually provided for converting back and forth between them.

Our SPEC benchmarks use byte or character, half word (short integer), word (integer), double word (long integer) and floating-point data types. Figure 2.12 shows the dynamic distribution of the sizes of objects referenced from memory for these programs. Figure 2.12 uses memory references to examine the types of data being accessed.



5Q. Explain various operations used in the instruction set
2.5 Operations in the Instruction Set

The operators supported by most instruction set architectures can be categorized as in Figure 2.15. One rule of thumb across all architectures is that the most widely executed instructions are the simple operations of an instruction set. For example Figure 2.16 shows 10 simple instructions that account for 96% of instructions executed for a collection of integer programs running on the popular Intel 80x86. Hence, the implementor of these instructions should be sure to make these fast, as they are the common case.

Operator type	Examples
Arithmetic and logical	Integer arithmetic and logical operations: add, subtract, and, or, multiple, 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

FIGURE 2.15 Categories of instruction operators and examples of each.

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%

Figure 2.16 : The top 10 instructions for the 80x86

6Q. Explain various types of control flow and address modes used for control flow in instruction ser architecture
2.6 Instructions for Control Flow

There are four different types of control-flow change:

1 Conditional branches
2 Jumps
3 Procedure calls
4 Procedure returns

We want to know the relative frequency of these events, as each event is different, may use different instructions, and may have different behavior. Figure 2.19 shows the frequencies of these control-flow instructions for a load-store computer running our benchmarks.

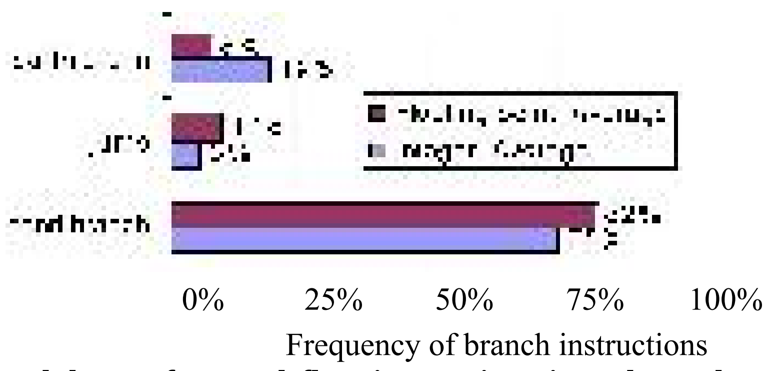


FIGURE 2.19 Breakdown of control flow instructions into three classes: calls or returns, jumps, and conditional branches.

Address modes for control flow instructions:

The destination address of a control flow instruction must always be specified. This destination is specified explicitly in the instruction in the vast majority of cases—procedure return being the major exception—since for return the target is not known at compile time. The most common way to specify the destination is to supply a displacement that is added to the program counter, or PC.

To implement returns and indirect jumps when the target is not known at compile time, a method other than PC-relative addressing is required. Here, there must be a way to specify the target dynamically, so that it can change at runtime. This dynamic address may be as simple as naming a register that contains the target address; alternatively, the jump may permit any addressing mode to be used to supply the target address.

These register indirect jumps are also useful for four other important features:

- case or switch statements found in most programming languages (which select among one of several alternatives);
- virtual functions or methods in object-oriented languages like C++ or Java (which allow different routines to be called depending on the type of the argument);
- high order functions or function pointers in languages like C or C++ (which allows functions to be passed as arguments giving some of the flavor of object oriented programming), and
- dynamically shared libraries (which allow a library to be loaded and linked at runtime only when it is actually invoked by the program rather than loaded and linked statically before the program is run).

In all four cases the target address is not known at compile time, and hence is usually loaded from memory into a register before the register indirect jump. Figure 2.20 shows the distribution of displacements for PC-relative branches in instructions. About 75% of the branches are in the forward direction.

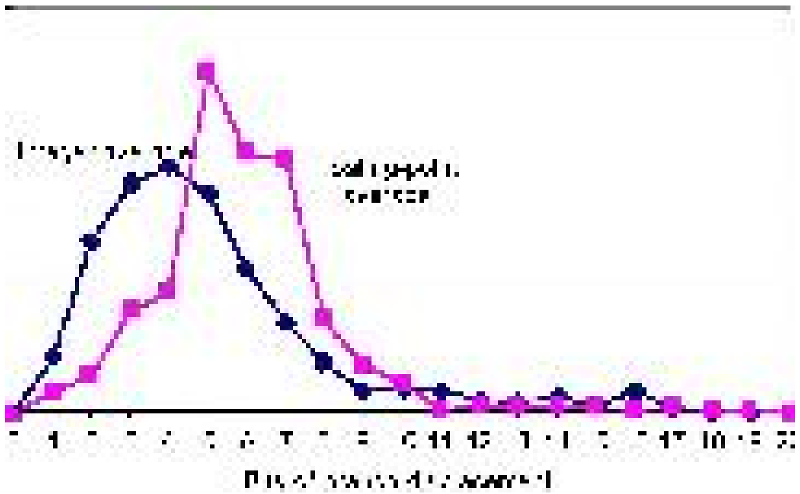


FIGURE 2.20 Branch distances in terms of number of instructions between the target and the branch instruction.

Conditional Branch Options

Since most changes in control flow are branches, deciding how to specify the branch condition is important. Figure 2.21 shows the three primary techniques in use today and their advantages and disadvantages.

One of the most noticeable properties of branches is that a large number of the comparisons are simple tests, and a large number are comparisons with zero. Thus, some architectures choose to treat these comparisons as special cases, especially if a compare and branch instruction is being used. Figure 2.22 shows the frequency of different comparisons used for conditional branching.

Name	**Examples**	**How condition is tested**	**Advantages**	**Disadvantages**
Condition code (CC)	80x86, ARM, PowerPC, SPARC, SuperH	Special bits are 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.

FIGURE 2.21 The major methods for evaluating branch conditions, their advantages, and their disadvantages.

FIGURE 2.22 Frequency of different types of compares in conditional branches.

Procedure Invocation Options:

Procedure calls and returns include control transfer and possibly some state saving; at a minimum the return address must be saved somewhere, sometimes in a special link register or just a GPR. Some older architectures provide a mechanism to save many registers, while newer architectures require the compiler to generate stores and loads for each register saved and restored.

There are two basic conventions in use to save registers: either at the call site or inside the procedure being called. Caller saving means that the calling procedure must save the registers that it wants preserved for access after the call, and thus the called procedure need not worry about registers. Callee saving is the opposite: the called procedure must save the registers it wants to use, leaving the caller is unrestrained.

There are times when caller save must be used because of access patterns to globally visible variables in two different procedures. For example, suppose we have a procedure P1 that calls procedure P2, and both procedures manipulate the global variable x. If P1 had allocated x to a register, it must be sure to save x to a location known by P2 before the call to P2. A compiler's ability to discover when a called procedure may access register-allocated quantities is complicated by the possibility of separate compilation. Suppose P2 may not touch x but can call another procedure, P3, that may access x, yet P2 and P3 are compiled separately. Because of these complications, most compilers will conservatively caller save any variable that may be accessed during a call.

2.7 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..

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.

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 the register field and addressing mode field may appear many times in a single instruction. There are 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.

Figure 2.23 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. 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.

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

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

(c) Hybrid (e.g., IBM 360/70, MIPS16, Thumb, TI TMS320C54x)

FIGURE 2.23 Three basic variations in instruction encoding: variable length, fixed length, and hybrid.

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. The narrow instructions support fewer operations, smaller address and immediate fields, fewer registers, and two-address format rather than the classic three-address format of RISC computers.

2.8 Crosscutting Issues: The Role of Compilers

Today almost all programming is done in high-level languages for desktop and server applications. This development means that since most instructions executed are the output of a compiler, an instruction set architecture is essentially a compiler target.

The Structure of Recent Compilers

A compiler writer's first goal is correctness—all valid programs must be compiled correctly. The second goal is usually speed of the compiled code. Typically, a whole set of other goals follows these two, including fast compilation, debugging support, and interoperability among languages. Normally, the passes in the compiler transform higher-level, more abstract representations into progressively lower-level representations. Eventually it reaches the instruction set. This structure helps manage the complexity of the transformations and makes writing a bug-free compiler easier.

The complexity of writing a correct compiler is a major limitation on the amount of optimization that can be done. Although the multiple-pass structure helps reduce compiler complexity, it also means that the compiler must order and perform some transformations before others. In the diagram of the optimizing compiler in Figure 2.24,

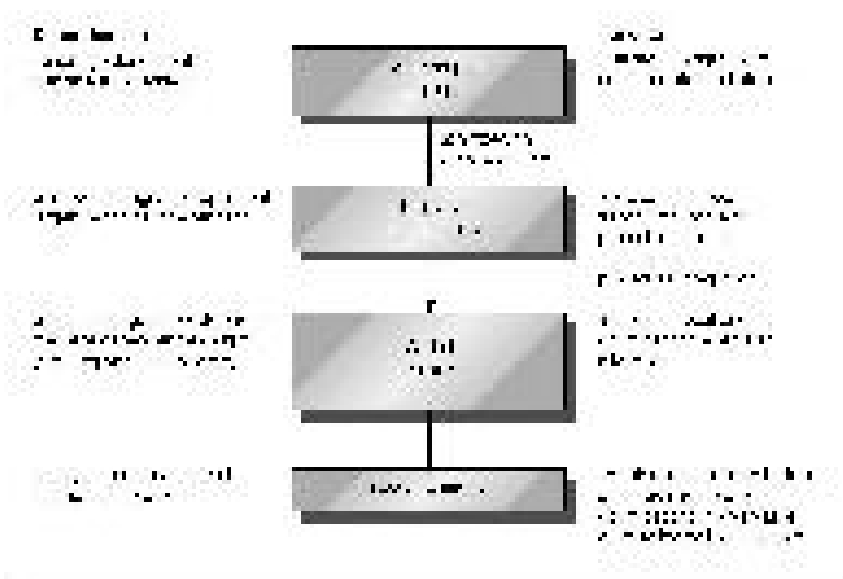


FIGURE 2.24 Compilers typically consist of two to four passes, with more highly optimizing compilers having more passes.

The compiler consists of four phases

1. Font end per language: The functionality of this phase is to transform high level language into common intermediate form. This phase functionally dependent on language and machine independent.
2. High-level optimization : The functionality of this phase is loop transformations and procedure integration. This phase some what language dependent and largely machine independent.
3. Global Optimizer : The functionality of this phase is local and global optimizations and register allocation. This phase contains small language dependency and machine dependencies slightly
4. Code generator: The functionality of this phase machine dependent optimization. This phase highly machine dependent and language independent.

Optimizations performed by modern compilers can be classified by the style of the transformation, as follows:

- High-level optimizations are often done on the source with output fed to later optimization passes.
- Local optimizations optimize code only within a straight-line code fragment (called a basic block by compiler people).
- Global optimizations extend the local optimizations across branches and introduce a set of transformations aimed at optimizing loops.
- Register allocation.
- processor-dependent optimizations attempt to take advantage of specific architectural knowledge.

Register Allocation

Register allocation algorithms today are based on a technique called graph coloring. The basic idea behind graph coloring is to construct a graph representing the possible candidates for allocation to a register and then to use the graph to allocate registers. Roughly speaking, the problem is how to use a limited set of colors so that no two adjacent nodes in a dependency graph have the same color. The emphasis in the approach is to achieve 100% register allocation of active variables. The problem of coloring a graph in general can take exponential time as a function of the size of the graph (NP-complete). There are heuristic algorithms, however, that work well in practice yielding close allocations that run in near linear time.

Graph coloring works best when there are at least 16 (and preferably more) general-purpose registers available for global allocation for integer variables and additional registers for floating point. Unfortunately, graph coloring does not work very well when the number of registers is small because the heuristic algorithms for coloring the graph are likely to fail.

Impact of Optimizations on Performance

The typical optimizations are given in Figure 2.25. The last column of Figure 2.25 indicates the frequency with which the listed optimizing transforms were applied to the source program.

Optimization name	**Explanation**	**Percentage of the total number of optimizing transforms**
High-level Procedure integration	**At or near the source level; processor-independent** Replace procedure call by procedure body	N.M.
Local Common subexpression elimination	**Within straight-line code** Replace two instances of the same computation by single copy	18%
Constant propagation	Replace all instances of a variable that is assigned a constant with the constant	22%
Stack height reduction	Rearrange expression tree to minimize resources needed for expression evaluation	N.M.

Global Global common subexpression elimination Copy propagation Code motion Induction variable elimination	**Across a branch** Same as local, but this version crosses branches Replace all instances of a variable A that has been assigned X (i.e., A = X) with X Remove code from a loop that computes same value each iteration of the loop Simplify/eliminate array-addressing calculations within loops	13% 11% 16% 2%
Processor-dependent Strength reduction Pipeline scheduling Branch offset optimization	**Depends on processor knowledge** Many examples, such as replace multiply by a constant with adds and shifts Reorder instructions to improve pipeline performance Choose the shortest branch displacement that reaches target	N.M. N.M. N.M.

FIGURE 2.25 Major types of optimizations and examples in each class.

The Impact of Compiler Technology on the Architect's Decisions

The interaction of compilers and high-level languages significantly affects how programs use an instruction set architecture.

The stack is used to allocate local variables. The stack is grown and shrunk on procedure call or return, respectively.

The global data area is used to allocate statically declared objects, such as global variables and constants. A large percentage of these objects are arrays or other aggregate data structures

The heap is used to allocate dynamic objects that do not adhere to a stack discipline.

Register allocation is much more effective for stack-allocated objects than for global variables, and register allocation is essentially impossible for heap-allocated objects because they are accessed with pointers.

The variable a could not be register allocated across the assignment to *p without generating incorrect code. Aliasing causes a substantial problem because it is often difficult or impossible to decide what objects a pointer may refer to. A compiler must be conservative; some compilers will not allocate any local variables of a procedure in a register when there is a pointer that may refer to one of the local variables.

How the Architect Can Help the Compiler Writer

Today, the complexity of a compiler does not come from translating simple statements like A = B + C. Most programs are locally simple, and simple translations work fine. Rather, complexity arises because programs are large and globally complex in their interactions, and because the structure of compilers means decisions are made one step at a time about which code sequence is best.

Compiler writers often are working under their own corollary of a basic principle in architecture: Make the frequent cases fast and the rare case correct. That is, if we know which cases are frequent and which are rare, and if generating code for both is straightforward, then the quality of the code for the rare case may not be very important—but it must be correct!

Some instruction set properties help the compiler writer. These properties should not be thought of as hard and fast rules, but rather as guidelines that will make it easier to write a compiler that will generate efficient and correct code.

1 Regularity—Whenever it makes sense, the three primary components of an instruction set—the operations, the data types, and the addressing modes— should be orthogonal. Two aspects of an architecture are said to be orthogonal if they are independent. For example, the operations and addressing modes are orthogonal if for every operation to which one addressing mode can be applied, all addressing modes are applicable. This regularity helps simplify code generation and is particularly important when the decision about what code to generate is split into two passes in the compiler. A good counterexample of this property is restricting what registers can be used for a certain class of instructions. Compilers for special-purpose register architectures typically get stuck in this dilemma. This restriction can result in the compiler finding itself with lots of available registers, but none of the right kind!

2 Provide primitives, not solutions—Special features that “match” a language construct or a kernel function are often unusable. Attempts to support high-level languages may work only with one language, or do more or less than is required for a correct and efficient implementation of the language. An example of how such attempts have failed is given in section 2.14.

3 Simplify trade-offs among alternatives—One of the toughest jobs a compiler writer has is figuring out what instruction sequence will be best for every segment of code that arises. In earlier days, instruction counts or total code size might have been good metrics, but—as we saw in the last chapter—this is no longer true. With caches and pipelining, the trade-offs have become very complex. Anything the designer can do to help the compiler writer understand the costs of alternative code sequences would help improve the code. One of the most difficult instances of complex trade-offs occurs in a register-memory architecture in deciding how many times a variable should be referenced before it is cheaper to load it into a register. This threshold is hard to compute and, in fact, may vary among models of the same architecture.

4 Provide instructions that bind the quantities known at compile time as constants—A compiler writer hates the thought of the processor interpreting at runtime a value that

was known at compile time. Good counterexamples of this principle include instructions that interpret values that were fixed at compile time. For instance, the VAX procedure call instruction (calls) dynamically interprets a mask saying what registers to save on a call, but the mask is fixed at compile time (see section 2.14).

Compiler Support (or lack thereof) for Multimedia Instructions

Alas, the designers of the SIMD instructions that operate on several narrow data times in a single clock cycle consciously ignored the prior subsection. These instructions tend to be solutions, not primitives, they are short of registers, and the data types do not match existing programming languages. Architects hoped to find an inexpensive solution that would help some users, but in reality, only a few low-level graphics library routines use them.

The SIMD instructions are really an abbreviated version of an elegant architecture style that has its own compiler technology. As explained in Appendix F, vector architectures operate on vectors of data. Invented originally for scientific codes, multimedia kernels are often vectorizable as well. Hence, we can think of Intel's MMX or PowerPC's AltiVec as simply short vector computers: MMX with vectors of eight 8-bit elements, four 16-bit elements, or two 32-bit elements, and AltiVec with vectors twice that length. They are implemented as simply adjacent, narrow elements in wide registers

These abbreviated architectures build the vector register size into the architecture: the sum of the sizes of the elements is limited to 64 bits for MMX and 128 bits for AltiVec. When Intel decided to expand to 128 bit vectors, it added a whole new set of instructions, called SSE.

The missing elegance from these architectures involves the specification of the vector length and the memory addressing modes. By making the vector width variable, these vectors seemlessly switch between different data widths simply by increasing the number of elements per vector. For example, vectors could have, say, 32 64-bit elements, 64 32-bit elements, 128 16-bit elements, and 256 8-bit elements. Another advantage is that the number of elements per vector register can vary between generations while remaining binary compatible. One generation might have 32 64-bit elements per vector register, and the next have 64 64-bit elements. (The number of elements per register is located in a status register.) The number of elements executed per clock cycle is also implementation dependent, and all run the same binary code. Thus, one generation might operate 64bits per clock cycle, and another at 256-bits per clock cycle.

A major advantage of vector computers is hiding latency of memory access by loading many elements at once and then overlapping execution with data transfer. The goal of vector addressing modes is to collect data scattered about memory, place them in a compact form so that they can be operated on efficiently, and then place the results back where they belong.

Over the years traditional vector computers added strided addressing and gather/scatter addressing to increase the number of programs that can be vectorized. Strided addressing skips a fixed number of words between each access, so sequential addressing is often called unit stride addressing. Gather and scatter find their addresses in another vector register: think of it as register indirect addressing for vector computers. From a vector perspective, in contrast these short-vector SIMD computers support only unit strided accesses: memory accesses load or store all elements at once from a single wide memory location. Since the data for multimedia applications are often streams that start and end in memory, strided and gather/scatter addressing modes such are essential to successful vectoization.

3 vector stores (to store YUV).

The total is 20 instructions to perform the 20 operations in the C code above to convert 8 pixels [Kozyrakis 2000]. (Since a vector might have 32 64-bit elements, this code actually converts up to 32 x 8 or 256 pixels.)

In contrast, Intel's web site shows a library routine to perform the same calculation on eight pixels takes 116 MMX instructions plus 6 80x86 instruc tions [Intel 2001]. This sixfold increase in instructions is due to the large num
ber of instructions to load and unpack RBG pixels and to pack and store YUV
pixels, since there are no strided memory accesses.

n

Having short, architecture limited vectors with few registers and simple memory addressing modes makes it more difficult to use vectorizing compiler technology. Another challenge is that no programming language (yet) has support for operations on these narrow data. Hence, these SIMD instructions are commonly found only in hand coded libraries.

Summary: The Role of Compilers

This section leads to several recommendations. First, we expect a new instruction set architecture to have at least 16 general-purpose registers—not counting separate registers for floating-point numbers—to simplify allocation of registers using graph coloring. The advice on orthogonality suggests that all supported addressing modes apply to all instructions that transfer data. Finally, the last three pieces of advice—provide primitives instead of solutions, simplify trade-offs between alternatives, don't bind constants at runtime—all suggest that it is better to err on the side of simplicity. In other words, understand that less is more in the design of an instruction set. Alas, SIMD extensions are more an example of good marketing than outstanding achievement of hardware/software co-design.