Computer Architecture
Basic Concepts and Performance (Week#4)

Benchmark and SPEC

Benchmark suite: is a collection of programs, defined in a high-level language, that together attempt to provide a representative test of a computer in a particular application or system programming area. The best known such collection of benchmark suites is defined and maintained by the Standard Performance Evaluation Corporation (SPEC), an industry consortium.

Other SPEC suites include the following:

SPECviewperf: Standard for measuring 3D graphics performance based on professional applications.

SPECwpc: benchmark to measure all key aspects of workstation performance based on diverse professional applications, including media and entertainment, product development, life sciences, financial services, and energy.

SPECjvm2008: Intended to evaluate performance of the combined hardware and software aspects of the Java Virtual Machine (JVM) client platform.

SPECjbb2013 (Java Business Benchmark): A benchmark for evaluating server-side Java-based electronic commerce applications.

Benchmark and SPEC

SPECsfs2008: Designed to evaluate the speed and request-handling capabilities of file servers.

SPECvirt_sc2013: Performance evaluation of datacenter servers used in virtualized server consolidation. Measures the end-to-end performance of all system components including the hardware, virtualization platform, and the virtualized guest operating system and application software. The benchmark supports hardware virtualization, operating system virtualization, and hardware partitioning schemes.

To better understand results of a system using CPU2006, we define the following terms used in the SPEC documentation:

Benchmark: A program written in a high-level language that can be compiled and executed on any computer that implements the compiler.

System under test: This is the system to be evaluated.

Reference machine: This is a system used by SPEC to establish a baseline performance for all benchmarks. Each benchmark is run and measured on this machine to establish a reference time for that benchmark. A system under test is evaluated by running the CPU2006 benchmarks and comparing the results for running the same programs on the reference machine.

Benchmark and SPEC

Base metric: These are required for all reported results and have strict guidelines for compilation. In essence, the standard compiler with more or less default settings should be used on each system under test to achieve comparable results.

Peak metric: This enables users to attempt to optimize system performance by optimizing the compiler output. For example, different compiler options may be used on each benchmark, and feedback-directed optimization is allowed.

Speed metric: This is simply a measurement of the time it takes to execute a compiled benchmark. The speed metric is used for comparing the ability of a computer to complete single tasks.

Rate metric: This is a measurement of how many tasks a computer can accomplish in a certain amount of time; this is called a throughput, capacity, or rate measure. The rate metric allows the system under test to execute simultaneous tasks to take advantage of multiple processors.

Benchmark and SPEC

For the integer benchmarks, there are 12 programs in the test suite. Calculation is a three-step process (Figure 2.7):

1. The first step in evaluating a system under test is to compile and run each program on the system three times.

For each program, the runtime is measured and the median value is selected.

The reason to use three runs and take the median value is to account for variations in execution time that are not intrinsic to the program, such as disk access time variations, and OS kernel execution variations from one run to another.

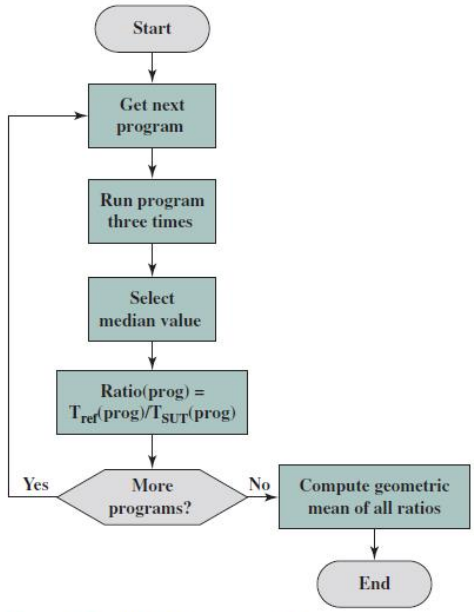


Figure 2.7 SPEC Evaluation Flowchart

Benchmark and SPEC

2. Next, each of the 12 results is normalized by calculating the runtime ratio of the reference run time to the system run time. The ratio is calculated as follows:

$$r_i = \frac{Tref_i}{Tsut_i}$$

Trefi is the execution time of benchmark program i on the reference system

Tsuti is the execution time of benchmark program i on the system under test. Thus, ratios are higher for faster machines

3.Finally, the geometric mean of the 12 runtime ratios is calculated to yield the overall metric.

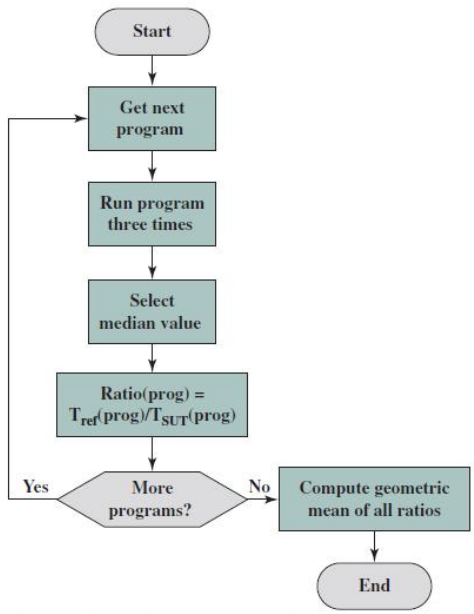


Figure 2.7 SPEC Evaluation Flowchart

Benchmark and SPEC

Example 2.9: The results for the Sun Blade 1000 are shown in Table 2.7a. One of the SPEC CPU2006 integer benchmark is 464.h264ref. This is a reference implementation of H.264/ AVC (Advanced Video Coding), the latest state-of-the-art video compression standard.

(a) Sun Blade 1000

Benchmark	Execution time (secs)	Execution time (secs)	Execution time (secs)	Reference time (secs)	Ratio
400.perlbench	3077	3076	3080	9770	3.18
401.bzip2	3260	3263	3260	9650	2.96
403.gcc	2711	2701	2702	8050	2.98
429.mcf	2356	2331	2301	9120	3.91
445.gobmk	3319	3310	3308	10,490	3.17
456.hmmer	2586	2587	2601	9330	3.61

(a) Sun Blade 1000

Benchmark	Execution time (secs)	Execution time (secs)	Execution time (secs)	Reference time (secs)	Ratio
458.sjeng	3452	3449	3449	12,100	3.51
462.libquantum	10,318	10,319	10,273	20,720	2.01
464.h264ref	5246	5290	5259	22,130	4.21
471.omnetpp	2565	2572	2582	6250	2.43
473.astar	2522	2554	2565	7020	2.75
483.xalancbmk	2014	2018	2018	6900	3.42

Benchmark and SPEC

The rate metrics take into account a system with multiple processors. To test a machine, a number of copies N is selected—usually this is equal to the number of processors or the number of simultaneous threads of execution on the test system. Each individual test program's rate is determined by taking the median of three runs. Each run consists of N copies of the program running simultaneously on the test system. The execution time is the time it takes for all the copies to finish (i.e., the time from when the first copy starts until the last copy finishes). The rate metric for that program is calculated by the following formula:

$$rate_i = N \times \frac{Tref_i}{Tsut_i}$$

The rate score for the system under test is determined from a geometric mean of rates for each program in the test suite.

Benchmark and SPEC

2.5 The following table, based on data reported in the literature [HEAT84], shows the execution times, in seconds, for five different benchmark programs on three machines.

Benchmark	Processor		
	R	M	Z
E	417	244	134
F	83	70	70
H	66	153	135
I	39,449	35,527	66,000
K	772	368	369

a. Compute the speed metric for each processor for each benchmark, normalized to machine R. That is, the ratio values for R are all 1.0. Other ratios are calculated using Equation (2.5) with R treated as the reference system. Then compute the arithmetic mean value for each system using Equation (2.3). This is the approach taken in [HEAT84].
b. Repeat part (a) using M as the reference machine. This calculation was not tried in [HEAT84].
c. Which machine is the slowest based on each of the preceding two calculations?
d. Repeat the calculations of parts (a) and (b) using the geometric mean, defined in Equation (2.6). Which machine is the slowest based on the two calculations?

Instruction Set Architecture

(Interface between hardware and software)

Machine Instruction Characteristic

Machine Instruction: The operation of the processor is determined by the instructions it executes, referred to as machine instructions or computer instructions.

Processor Instruction: The collection of different instructions that the processor can execute is referred to as the processor's instruction set.

Elements of Machine Instruction: Each instruction must contain the information required by the processor for execution.Figure 12.1 shows the steps involved in instruction execution and defines the elements of a machine instruction.

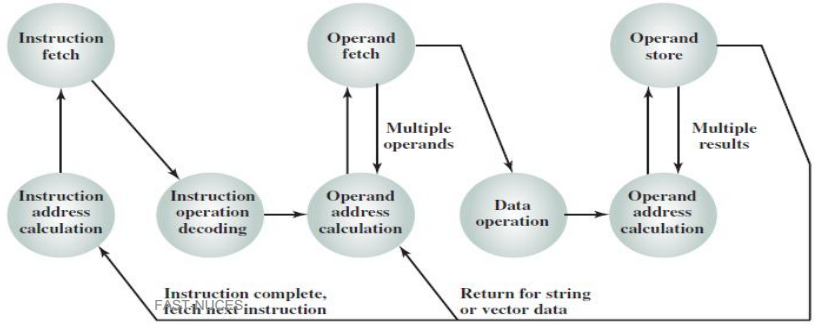


Figure 12.1 Instruction Cycle State Diagram

Machine Instruction Characteristic

These elements are as follows:

Operation code: Specifies the operation to be performed (e.g., ADD, I/O). The operation is specified by a binary code, known as the operation code, or opcode.

Source operand reference: The operation may involve one or more source operands, that is, operands that are inputs for the operation.

Result operand reference: The operation may produce a result.

Next instruction reference: This tells the processor where to fetch the next instruction after the execution of this instruction is complete.

Source and result operands can be in one of four areas:

Main or virtual memory: As with next instruction references, the main or virtual memory address must be supplied.

Processor register: contains one or more registers that may be referenced by machine instructions. If only one register exists, reference to it may be implicit. If more than one register exists, then each register is assigned a unique name or number, and the instruction must contain the number of the desired register.

Machine Instruction Characteristic

Immediate: The value of the operand is contained in a field in the instruction being executed.

I/O device: The instruction must specify the I/O module and device for the operation. If memory- mapped I/O is used, this is just another main or virtual memory address.

Instruction Representation: each instruction is represented by a sequence of bits. The instruction is divided into fields, corresponding to the constituent elements of the instruction. A simple example of an instruction format is shown in Figure 12.2.

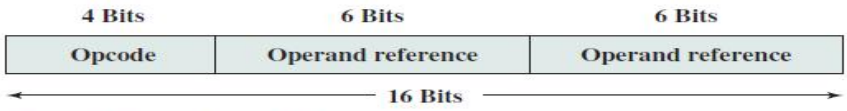


Figure 12.2 A Simple Instruction Format

During instruction execution, an instruction is read into an instruction register (IR) in the processor. The processor must be able to extract the data from the various instruction fields to perform the required operation.

Machine Instruction Characteristic

Opcodes: are represented by abbreviations, called mnemonics, that indicate the operation. Common examples include

ADD ----------Add

SUB ----------Subtract

MUL ----------Multiply

DIV ----------Divide

LOAD---------- Load data from memory

STOR---------- Store data to memory

Operands are also represented symbolically. For example, the instruction

ADD R, Y

add the value contained in data location Y to the contents of register R. Y refers to the address of a location in memory, and R refers to a particular register. Note that the operation is performed on the contents of a location, not on its address.

Machine Instruction Characteristic

Instruction Types: Consider a high- level language instruction that could be expressed in a language such as BASIC or FORTRAN. For example,

X = X + Y

This statement instructs the computer to add the value stored in Y to the value stored in X and put the result in X.

How might this be accomplished with machine instructions?

Let us assume that the variables X and Y correspond to locations 513 and 514. If we assume a simple set of machine instructions, this operation could be accomplished with three instructions:

1. Load a register with the contents of memory location 513.

2. Add the contents of memory location 514 to the register.

3. Store the contents of the register in memory location 513

Machine Instruction Characteristic

we can categorize instruction types as follows:

Data processing (Arithmetic and logic instructions): provide computational capabilities for processing numeric data. Logic (Boolean) instructions operate on the bits of a word as bits rather than as numbers; they provide capabilities for processing any other type of data the user may wish to employ. These operations are performed primarily on data in processor registers.

Data storage: Movement of data into or out of register and or memory locations.

Data movement (I/O instructions): I/O instructions are needed to transfer programs and data into memory and the results of computations back out to the user.

Control (Test and branch instructions): Test instructions are used to test the value of a data word or the status of a computation. Branch instructions are then used to branch to a different set of instructions depending on the decision made.

Machine Instruction Characteristic

Branch Instruction:

A branch, jump or transfer is an instruction in a computer program that can cause a computer to begin executing a different instruction sequence and thus deviate from its default behavior of executing instructions in order.

Test Instruction:

A 'Test Instruction' in Computer Science refers to an atomic operation that allows a task to check and modify a memory location based on the value of a bit. It sets the bit to 1 if it is 0 and returns 0, or returns 1 if the bit is already 1.

Machine Instruction Characteristic

Number of Addresses: One of the traditional ways of describing processor architecture is in terms of the number of addresses contained in each instruction. This dimension has become less significant with the increasing complexity of processor design.

Virtually all arithmetic and logic operations are either unary (one source operand) or binary (two source operands). We would need a maximum of two addresses to reference source operands.

The result of an operation must be stored, suggesting a third address, which defines a destination operand. Finally, after completion of an instruction, the next instruction must be fetched, and its address is needed.

Instruction could required four address references: two source operands, one destination operand, and the address of the next instruction. In most architectures, many instructions have one, two, or three operand addresses, with the address of the next instruction being implicit (obtained from the program counter).

Machine Instruction Characteristic

Figure 12.3 compares typical one-, two-, and three- address instructions that could be used to compute Y = (A - B)/[C + (D * E)]. With three addresses, each instruction specifies two source operand locations and a destination operand location.

Instruction	Comment
SUB Y, A, B	Y ← A − B
MPY T, D, E	T ← D × E
ADD T, T, C	T ← T + C
DIV Y, Y, T	Y ← Y ÷ T

(a) Three-address instructions

Instruction	Comment
MOVE Y, A	Y ← A
SUB Y, B	Y ← Y − B
MOVE T, D	T ← D
MPY T, E	T ← T × E
ADD T, C	T ← T + C
DIV Y, T	Y ← Y ÷ T

(b) Two-address instructions

Instruction	Comment
LOAD D	AC ← D
MPY E	AC ← AC × E
ADD C	AC ← AC + C
STOR Y	Y ← AC
LOAD A	AC ← A
SUB B	AC ← AC − B
DIV Y	AC ← AC ÷ Y
STOR Y	Y ← AC

(c) One-address instructions

Figure 12.3 Programs to Execute $Y = \frac{A - B}{C + (D \times E)}$

Machine Instruction Characteristic

Zero-address instructions: are applicable to a special memory organization called a stack. A stack is a last- in- first- out set of locations. The stack is in a known location and, often, at least the top two elements are in processor registers. Thus, zero- address instructions would reference the top two stack elements.

Table 12.1 summarizes the interpretations to be placed on instructions with zero, one, two, or three addresses. it is assumed that the address of the next instruction is implicit, and that one operation with two source operands and one result operand is to be performed.

Table 12.1 Utilization of Instruction Addresses (Nonbranching Instructions)

Number of Addresses	Symbolic Representation	Interpretation
3	OP A, B, C	A ← B OP C
2	OP A, B	A ← A OP B
1	OP A	AC ← AC OP A
0	OP	T ← (T − 1) OP T

AC = accumulator
T = top of stack
(T − 1) = second element of stack
A, B, C = memory or register locations

Machine Instruction Characteristic

The number of addresses per instruction is a basic design decision. Fewer addresses per instruction result in instructions that are more primitive, requiring a less complex processor. It also results in instructions of shorter length.

On the other hand, programs contain more total instructions, which in general results in longer execution times and longer, more complex programs. There is an important threshold between one-address and multiple-address instructions.

With one-address instructions, the programmer generally has available only one general-purpose register, the accumulator. With multiple-address instructions, it is common to have multiple general- purpose registers.

This allows some operations to be performed solely on registers. Because register references are faster than memory references, this speeds up execution.

For reasons of flexibility and ability to use multiple registers,most contemporary machines employ a mixture of two- and three- address instructions.

Machine Instruction Characteristic

Instruction Set Design: Instruction set defines many of the functions performed by the processor and has a significant effect on the implementation of the processor.

Instruction set is the programmer's means of controlling the processor. Programmer requirements must be considered in designing the instruction set.

Fundamental design issues include the following:

Operation repertoire: How many and which operations to provide, and how complex operations should be.

Data types: The various types of data upon which operations are performed.

Instruction format: Instruction length (in bits), number of addresses, size of various fields, and so on.

Registers: Number of processor registers that can be referenced by instructions, and their use.

Addressing: The mode or modes by which the address of an operand is specified.

Types of Operand

Machine instructions operate on data. The most important general categories of data are:

- ✓ Addresses
- ✓ Numbers
- ✓ Characters
- ✓ Logical data

1.Numbers: All machine languages include numeric data types. Even in nonnumeric data processing,there is a need for numbers to act as counters, field widths.

First, there is a limit to the magnitude of numbers representable on a machine and second, in the case of floating- point numbers, a limit to their precision. Programmer is faced with understanding the consequences of rounding, overflow, and underflow.

Three types of numerical data are common in computers:

- ✓ Binary integer or binary fixed point
- ✓ Binary floating point
- ✓ Decimal

Types of Operand

There is a necessity to convert from decimal to binary on input and from binary to decimal on output. For applications in which there is a great deal of I/O and comparatively little, comparatively simple computation, it is preferable to store and operate on the numbers in decimal form. The most common representation for this purpose is packed decimal.

Packed decimal: Each decimal digit is represented by a 4-bit code with two digits stored per byte.

A packed decimal representation stores decimal digits in each "nibble" of a byte. Each byte has two nibbles, and each nibble is indicated by a hexadecimal digit. For example, the value 15 is stored in two nibbles, using the hexadecimal digits 1 and 5

Negative numbers can be represented by including a 4-bit sign digit at either the left or right end of a string of packed decimal digits.

Sign Magnitude:

- ✓ In a Signed Number, the MSB (the sign bit) is 0 for a Positive number.
- ✓ In a Signed Number, the MSB (the sign bit) is 1 for a Negative number.

Types of Operand

12.1 Show in hex notation:
- a. The packed decimal format for 23.
- b. The ASCII characters 23.

12.3 A given microprocessor has words of 1 byte. What is the smallest and largest integer that can be represented in the following representations:
- a. Unsigned.
- b. Sign-magnitude.
- c. Ones complement.
- d. Twos complement.
- e. Unsigned packed decimal.
- f. Signed packed decimal.

Types of Operand

2.Characters: A common form of data is text or character strings. While textual data are most convenient for human beings, they cannot, in character form, be easily stored or transmitted by data processing and communications systems.

Such systems are designed for binary data. A number of codes have been devised which characters are represented by a sequence of bits.

Earliest common example of this is the **Morse code.**

Chart of the Morse code 26 letters and 10 numerals[1]

Types of Operand

ASCII: International Reference Alphabet (IRA), referred to in the United States as ASCII.

Each character in this code is represented by a unique 7-bit pattern; 128 different characters can be represented.

IRA- encoded characters are almost always stored and transmitted using 8 bits per character. The eighth bit may be set to 0 or used as a parity bit for error detection.

- ✓ *With even parity, you would add a parity bit to make the total count of ones even. So, the parity bit would be set to 1, resulting in the code 11011.*
- ✓ *With odd parity, the parity bit would be set to 0 to make the total count of one's odds, resulting in the code 11010 .*

Extended Binary Coded Decimal Interchange Code (EBCDIC): used on IBM mainframes. It is an 8-bit code. As with IRA, EBCDIC is compatible with packed decimal. In the case of EBCDIC, the codes 11110000 through 11111001 represent the digits 0 through 9.

Types of Operand

3. Logical Data: Each word or other addressable unit (byte, halfword, and so on) is treated as a single unit of data. An n- bit unit as consisting of n 1-bit items of data, each item having the value 0 or 1 considered to be logical data.

There are two advantages to the bit- oriented view.

1. store an array of Boolean or binary data items, in which each item can take on only the values 1 (true) and 0 (false). With logical data, memory can be used most efficiently for this storage.

2.to manipulate the bits of a data item. For example, if floating-point operations are implemented in software, we need to be able to shift significant bits in some operations.

Another example: To convert from IRA to packed decimal, we need to extract the rightmost 4 bits of each byte.

Intel x86 and ARM Data Types

A) x86 Data Types: The x86 can deal with data types of 8 (byte), 16 (word), 32 (doubleword), 64 (quadword), and 128 (double quadword) bits in length.

To allow maximum flexibility in data structures and efficient memory utilization, words need not be aligned at even-numbered addresses; doublewords need not be aligned at addresses evenly divisible by 4; quadwords need not be aligned at addresses evenly divisible by 8; and so on.

When data are accessed across a 32-bit bus, data transfers take place in units of doublewords, beginning at addresses divisible by 4.

The processor converts the request for misaligned values into a sequence of requests for the bus transfer.

As with all of the Intel 80x86 machines, the x86 uses the little- endian style; the least significant byte is stored in the lowest address.

Intel x86 and ARM Data Types

Endianness: refers to the order in which bytes are arranged in memory.

Big-endian (BE): The mapping on the left stores the most significant byte in the lowest numerical byte address; this is known as big endian and is equivalent to the left- to- right order of writing.

Address	Value
184	12
185	34
186	56
187	78

Address	Value
184	78
185	56
186	34
187	12

Little-endian (LE): The mapping on the right stores the least significant byte in the lowest numerical byte address ; this is known as little endian and is reminiscent of the right- to- left order of arithmetic operations in arithmetic units.

Intel x86 and ARM Data Types

Table 12.2 x86 Data Types

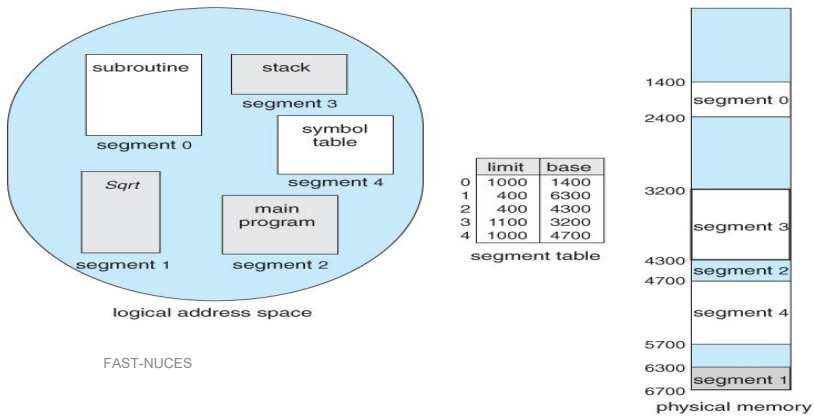
Data Type	Description
General	Byte, word (16 bits), doubleword (32 bits), quadword (64 bits), and double quadword (128 bits) locations with arbitrary binary contents.
Integer	A signed binary value contained in a byte, word, or doubleword, using twos complement representation.
Ordinal	An unsigned integer contained in a byte, word, or doubleword.
Unpacked binary coded decimal (BCD)	A representation of a BCD digit in the range 0 through 9, with one digit in each byte.
Packed BCD	Packed byte representation of two BCD digits; value in the range 0 to 99.
Near pointer	A 16-bit, 32-bit, or 64-bit effective address that represents the offset within a segment. Used for all pointers in a nonsegmented memory and for references within a segment in a segmented memory.
Far pointer	A logical address consisting of a 16-bit segment selector and an offset of 16, 32, or 64 bits. Far pointers are used for memory references in a segmented memory model where the identity of a segment being accessed must be specified explicitly.
Bit field	A contiguous sequence of bits in which the position of each bit is considered as an independent unit. A bit string can begin at any bit position of any byte and can contain up to 32 bits.
Bit string	A contiguous sequence of bits, containing from zero to $2^{23} - 1$ bits.
Byte string	A contiguous sequence of bytes, words, or doublewords, containing from zero to $2^{23} - 1$ bytes.
Floating point	See Figure 12.4.
Packed SIMD (single instruction, multiple data)	Packed 64-bit and 128-bit data types.

Intel x86 and ARM Data Types

Near pointer: is used to address memory within the same memory segment as the pointer itself.

Far pointer: is used to address memory within any segment, not just the one the pointer belongs to.

Memory segmentation: is an operating system memory management technique of dividing a computer's primary memory into segments or sections. In a computer system using segmentation, a reference to a memory location includes a value that identifies a segment and an offset (memory location) within that segment.



Intel x86 and ARM Data Types

Figure 12.4 illustrates the x86 numerical data types.

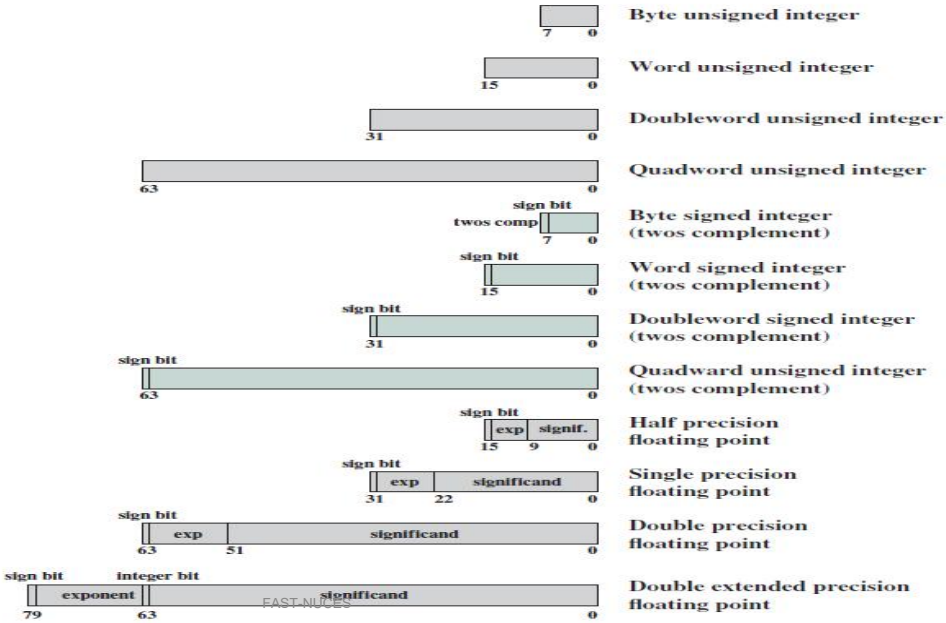


Figure 12.4 x86 Numeric Data Formats

Intel x86 and ARM Data Types

Signed integers: are in twos complement representation and may be 16, 32, or 64 bits long.

Floating- point: refers to a set of types that are used by the floating-point unit and operated on by floating-point instructions.

Packed SIMD (single-instruction- multiple-data) data types: were introduced to the x86 architecture as part of the extensions of the instruction set to optimize performance of multimedia applications.

These extensions include MMX (multimedia extensions) and SSE (streaming SIMD extensions).

The basic concept is that multiple operands are packed into a single referenced memory item and that these multiple operands are operated on in parallel. The data types are as follows:

Packed byte and packed byte integer: Bytes packed into a 64-bit quadword or 128-bit double quadword, interpreted as a bit field or as an integer

Intel x86 and ARM Data Types

Packed word and packed word integer: 16-bit words packed into a 64-bit quadword or 128-bit double quadword, interpreted as a bit field or as an integer.

Packed doubleword and packed doubleword integer: 32-bit doublewords packed into a 64-bit quadword or 128-bit double quadword, interpreted as a bit field or as an integer.

Packed quadword and packed quadword integer: Two 64-bit quadwords packed into a 128-bit double quadword, interpreted as a bit field or as an integer.

Packed single-precision floating- point and packed double-precision floatingpoint: Four 32-bit floating- point or two 64-bit floating-point values packed into a 128-bit double quadword.

Intel x86 and ARM Data Types

B) ARM Data Types: ARM processors support data types of 8 (byte), 16 (halfword), and 32 (word) bits in length. Normally, halfword access should be halfword aligned and word accesses should be word aligned. For nonaligned access attempts, the architecture supports three alternatives.

Default case: The address is treated as truncated, with address bits[1:0] treated as zero for word accesses, and address bit[0] treated as zero for halfword accesses.

Load single word ARM instructions are architecturally defined to rotate right the word-aligned data transferred by a non word-aligned address one, two, or three bytes depending on the value of the two least significant address bits.

Alignment checking: When the appropriate control bit is set, a data abort signal indicates an alignment fault for attempting unaligned access.

Unaligned access: When this option is enabled, the processor uses one or more memory accesses to generate the required transfer of adjacent bytes transparently to the programmer.

Intel x86 and ARM Data Types

For all three data types (byte, halfword, and word) an unsigned interpretation is supported, in which the value represents an unsigned, nonnegative integer. All three data types can also be used for twos complement signed integers.

The majority of ARM processor implementations do not provide floating-point hardware, which saves power and area.

If floating- point arithmetic is required in such processors, it must be implemented in software.

ARM does support an optional floating-point coprocessor that supports the single-and double- precision floating point data types defined in IEEE 754.

Intel x86 and ARM Data Types

ENDIAN Support: A state bit (E-bit) in the system control register is set and cleared under program control using the SETEND instruction. The E-bit defines which endian to load and store data.

Figure 12.5 illustrates the functionality associated with the E- bit for a word load or store operation. This mechanism enables efficient dynamic data load/store for system designers who know they need to access data structures in the opposite endianness to their OS/environment. Address of each data byte is fixed in memory. However, the byte lane in a register is different.

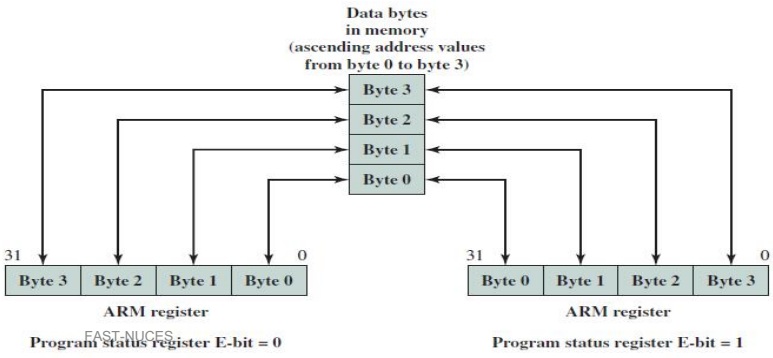


Figure 12.5 ARM Endian Support—Word Load/Store with E-Bit

Types of Operations

The number of different opcodes varies widely from machine to machine. Same general types of operations are found on all machines. A useful and typical categorization is the following:

- ✓ Data transfer
- ✓ Arithmetic
- ✓ Logical
- ✓ Conversion
- ✓ I/O
- ✓ System control
- ✓ Transfer of control

Types of Operations

Table 12.3 Common Instruction Set Operations

Type	Operation Name	Description
Data transfer	Move (transfer)	Transfer word or block from source to destination
	Store	Transfer word from processor to memory
	Load (fetch)	Transfer word from memory to processor
	Exchange	Swap contents of source and destination
	Clear (reset)	Transfer word of 0s to destination
	Set	Transfer word of 1s to destination
	Push	Transfer word from source to top of stack
	Pop	Transfer word from top of stack to destination
Arithmetic	Add	Compute sum of two operands
	Subtract	Compute difference of two operands
	Multiply	Compute product of two operands
	Divide	Compute quotient of two operands
	Absolute	Replace operand by its absolute value
	Negate	Change sign of operand
	Increment	Add 1 to operand
	Decrement	Subtract 1 from operand

Types of Operations

Logical	AND	Perform logical AND
	OR	Perform logical OR
	NOT	(complement) Perform logical NOT
	Exclusive-OR	Perform logical XOR
	Test	Test specified condition; set flag(s) based on outcome
	Compare	Make logical or arithmetic comparison of two or more operands; set flag(s) based on outcome
	Set Control Variables	Class of instructions to set controls for protection purposes, interrupt handling, timer control, etc.
	Shift	Left (right) shift operand, introducing constants at end
	Rotate	Left (right) shift operand, with wraparound end
Transfer of control	Jump (branch)	Unconditional transfer; load PC with specified address
	Jump Conditional	Test specified condition; either load PC with specified address or do nothing, based on condition
	Jump to Subroutine	Place current program control information in known location; jump to specified address
	Return	Replace contents of PC and other register from known location
	Execute	Fetch operand from specified location and execute as instruction; do not modify PC
	Skip	Increment PC to skip next instruction
	Skip Conditional	Test specified condition; either skip or do nothing based on condition
	Halt	Stop program execution
	Wait (hold)	Stop program execution; test specified condition repeatedly; resume execution when condition is satisfied
	No operation	No operation is performed, but program execution is continued

Types of Operations

Type	Operation Name	Description
Input/output	Input (read)	Transfer data from specified I/O port or device to destination (e.g., main memory or processor register)
	Output (write)	Transfer data from specified source to I/O port or device
	Start I/O	Transfer instructions to I/O processor to initiate I/O operation
	Test I/O	Transfer status information from I/O system to specified destination
Conversion	Translate	Translate values in a section of memory based on a table of correspondences
	Convert	Convert the contents of a word from one form to another (e.g., packed decimal to binary)

Types of Operations

Table 12.4 Processor Actions for Various Types of Operations

Data transfer	Transfer data from one location to another
	If memory is involved: Determine memory address Perform virtual-to-actual-memory address transformation Check cache Initiate memory read/write
Arithmetic	May involve data transfer, before and/or after
	Perform function in ALU
	Set condition codes and flags
Logical	Same as arithmetic
Conversion	Similar to arithmetic and logical. May involve special logic to perform conversion
Transfer of control	Update program counter. For subroutine call/return, manage parameter passing and linkage
I/O	Issue command to I/O module
	If memory-mapped I/O, determine memory-mapped address

Types of Operations

Table 12.5 Examples of IBM EAS/390 Data Transfer Operations

Operation Mnemonic	Name	Number of Bits Transferred	Description
L	Load	32	Transfer from memory to register
LH	Load Halfword	16	Transfer from memory to register
LR	Load	32	Transfer from register to register
LER	Load (short)	32	Transfer from floating-point register to floating-point register
LE	Load (short)	32	Transfer from memory to floating-point register
LDR	Load (long)	64	Transfer from floating-point register to floating-point register
LD	Load (long)	64	Transfer from memory to floating-point register
ST	Store	32	Transfer from register to memory
STH	Store Halfword	16	Transfer from register to memory
STC	Store Character	8	Transfer from register to memory
STE	Store (short)	32	Transfer from floating-point register to memory
STD	Store (long)	64	Transfer from floating-point register to memory

Types of Operations

Data Transfer: Amount of data to be transferred (8, 16, 32, or 64 bits).

There are different instructions for register to register, register to memory, memory to register, and memory to memory transfers.

VAX has a move (MOV) instruction with variants for different amounts of data to be moved, but it specifies whether an operand is register or memory as part of the operand.

VAX approach is somewhat easier for the programmer, who has fewer mnemonics to deal with.

Less compact than the IBM EAS/390 approach because the location (register versus memory) of each operand must be specified separately in the instruction.

Types of Operations

Arithmetic: Most machines provide the basic arithmetic operations of add, subtract, multiply, and divide. These are invariably provided for signed integer (fixed-point) numbers. Often they are also provided for floating-point and packed decimal numbers.

Other possible operations include a variety of single-operand instructions; for example,

- ✓ **Absolute:** Take the absolute value of the operand.
- ✓ **Negate:** Negate the operand.
- ✓ **Increment:** Add 1 to the operand.
- ✓ **Decrement:** Subtract 1 from the operand.

The execution of an arithmetic instruction may involve data transfer operations to position operands for input to the ALU, and to deliver the output of the ALU.

Figure 3.5 illustrates the movements involved in both data transfer and arithmetic operations.

Types of Operations

Figure 3.5 illustrates the movements involved in both data transfer and arithmetic operations.

Adds the contents of the memory word at address 940 to the contents of the memory word at address 941 and stores the result.

Three fetch and three execute cycles, are required:

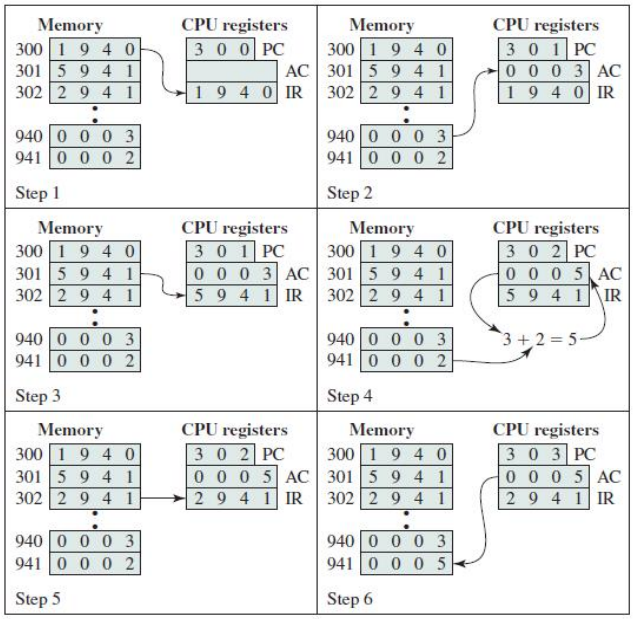


 Figure 3.5 Example of Program Execution (contents of memory and registers in hexadecimal)

Types of Operations

1.PC contains 300, the address of the first instruction. This instruction (the value 1940 in hexadecimal) is loaded into the instruction register IR, and the PC is incremented.

Note that this process involves the use of a memory address register and a memory buffer register.

For simplicity, these intermediate registers are ignored.

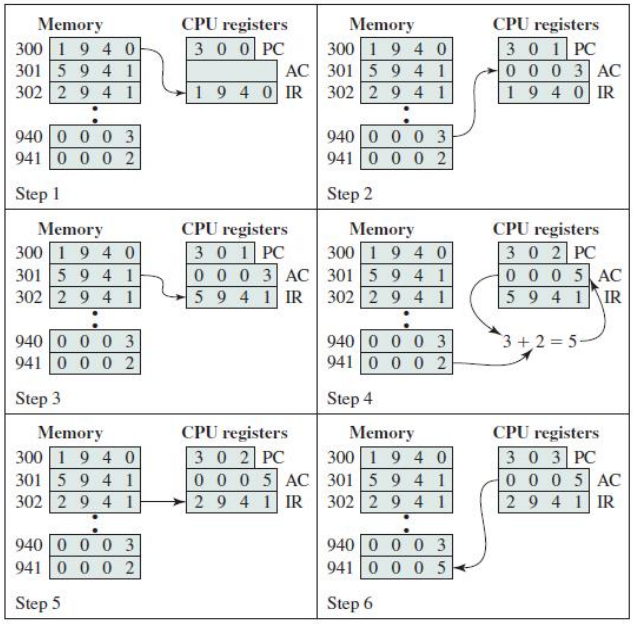


 Figure 3.5 Example of Program Execution (contents of memory and registers in hexadecimal)

Types of Operations

2. The first 4 bits (first hexadecimal digit) in the IR indicate that the AC is to be loaded. The remaining 12 bits (three hexadecimal digits) specify the address (940) from which data are to be loaded.

3. The next instruction (5941) is fetched from location 301, and the PC is incremented.

4. The old contents of the AC and the contents of location 941 are added, and the result is stored in the AC.

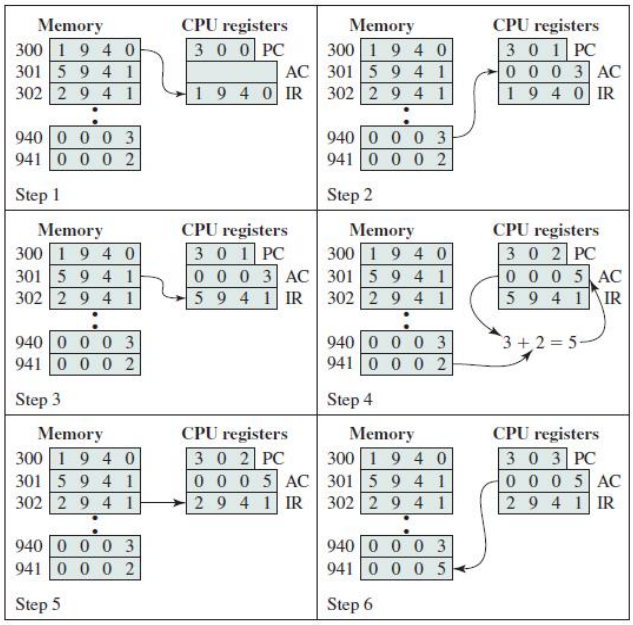


 Figure 3.5 Example of Program Execution (contents of memory and registers in hexadecimal)

Types of Operations

5. The next instruction (2941) is fetched from location 302, and the PC is incremented.

6. The contents of the AC are stored in location 941.

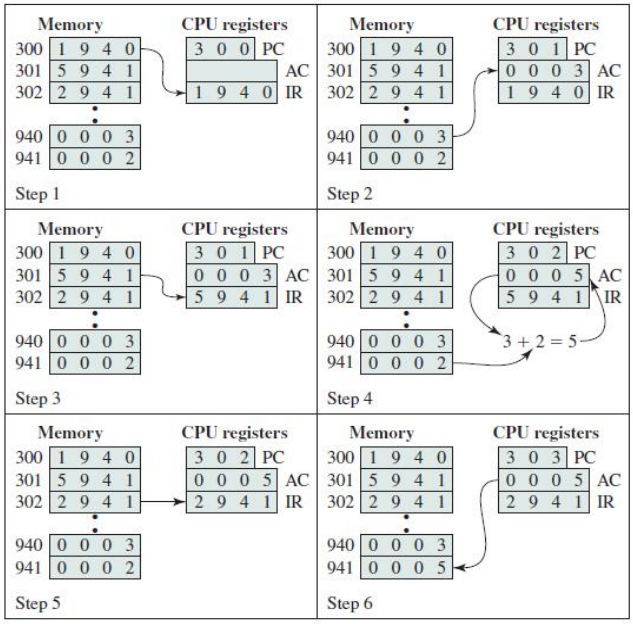


 Figure 3.5 Example of Program Execution (contents of memory and registers in hexadecimal)