**CO UNIT 3 PT 3**

**RISC**

Since the development of the stored-program computer around 1950, there have

been remarkably few true innovations in the areas of computer organization and

architecture. The following are some of the major advances since the birth of the

computer:

■ The family concept: Introduced by IBM with its System/360 in 1964, followed

shortly thereafter by DEC, with its PDP- 8. The family concept decouples

the architecture of a machine from its implementation. A set of computers

is offered, with different price/performance characteristics, that presents the

same architecture to the user. The differences in price and performance are

due to different implementations of the same architecture.

■ Microprogrammed control unit: Suggested by Wilkes in 1951 and introduced

by IBM on the S/360 line in 1964. Microprogramming eases the task of designing and implementing the control unit and provides support for the family

concept.

■ Cache memory: First introduced commercially on IBM S/360 Model 85 in

1968. The insertion of this element into the memory hierarchy dramatically

improves performance.

■ Pipelining: A means of introducing parallelism into the essentially sequential

nature of a machine-instruction program. Examples are instruction pipelining

and vector processing.

■ Multiple processors: This category covers a number of different organizations

and objectives.

■ Reduced instruction set computer (RISC) architecture: This is the focus of

this chapter.

When it appeared, RISC architecture was a dramatic departure from the historical trend in processor architecture. An analysis of the RISC architecture brings

into focus many of the important issues in computer organization and architecture.

Although RISC architectures have been defined and designed in a variety of

ways by different groups, the key elements shared by most designs are these:

■ A large number of general-purpose registers, and/or the use of compiler technology to optimize register usage.

■ A limited and simple instruction set.

■ An emphasis on optimizing the instruction pipeline.

Table 15.1 compares several RISC and non-RISC systems.

We begin this chapter with a brief survey of some results on instruction sets,

and then examine each of the three topics just listed. This is followed by a description of two of the best-documented RISC designs.



One of the most visible forms of evolution associated with computers is that of programming languages. As the cost of hardware has dropped, the relative cost of software has risen. Along with that, a chronic shortage of programmers has driven up

software costs in absolute terms. Thus, the major cost in the life cycle of a system is

software, not hardware. Adding to the cost, and to the inconvenience, is the element

of unreliability: it is common for programs, both system and application, to continue

to exhibit new bugs after years of operation.

The response from researchers and industry has been to develop ever more

powerful and complex high- level programming languages. These high- level languages (HLLs): (1) allow the programmer to express algorithms more concisely;

(2) allow the compiler to take care of details that are not important in the programmer’s expression of algorithms; and (3) often support naturally the use of structured

programming and/or object-oriented design.

Alas, this solution gave rise to a perceived problem, known as the semantic gap, the difference between the operations provided in HLLs and those provided in computer architecture. Symptoms of this gap are alleged to include

execution inefficiency, excessive machine program size, and compiler complexity. Designers responded with architectures intended to close this gap. Key

features include large instruction sets, dozens of addressing modes, and various HLL statements implemented in hardware. An example of the latter is

the CASE machine instruction on the VAX. Such complex instruction sets are

intended to:

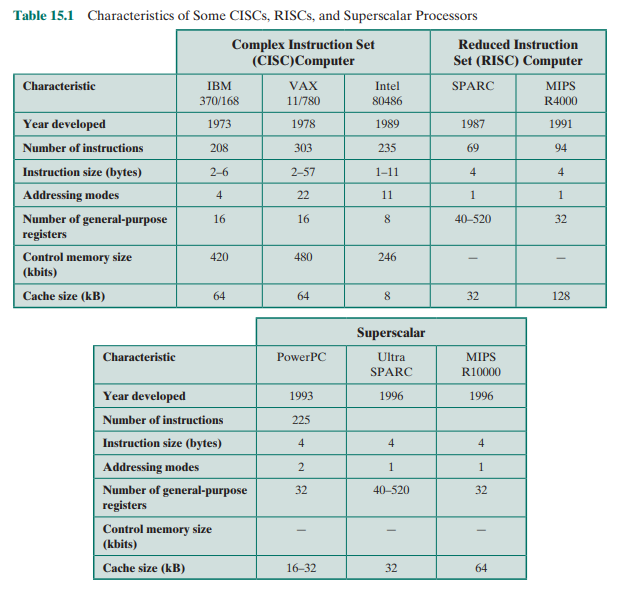
Ease the task of the compiler writer.

■ Improve execution efficiency, because complex sequences of operations can

be implemented in microcode.

■ Provide support for even more complex and sophisticated HLLs.

Meanwhile, a number of studies have been done over the years to determine the characteristics and patterns of execution of machine instructions generated from HLL programs. The results of these studies inspired some researchers to look



for a different approach: namely, to make the architecture that supports the HLL

simpler, rather than more complex.

To understand the line of reasoning of the RISC advocates, we begin with a

brief review of instruction execution characteristics. The aspects of computation of

interest are as follows:

■ Operations performed: These determine the functions to be performed by the

processor and its interaction with memory.

■ Operands used: The types of operands and the frequency of their use determine the memory organization for storing them and the addressing modes for

accessing them.

■ Execution sequencing: This determines the control and pipeline organization.

In the remainder of this section, we summarize the results of a number of

studies of high-level-language programs. All of the results are based on dynamic

measurements. That is, measurements are collected by executing the program and

counting the number of times some feature has appeared or a particular property

has held true. In contrast, static measurements merely perform these counts on

the source text of a program. They give no useful information on performance,

because they are not weighted relative to the number of times each statement is

executed.

**Operations**

A variety of studies have been made to analyze the behavior of HLL programs.

Table 4.7, discussed in Chapter 4, includes key results from a number of studies.

There is quite good agreement in the results of this mixture of languages and applications. Assignment statements predominate, suggesting that the simple movement of data is of high importance. There is also a preponderance of conditional

statements (IF, LOOP). These statements are implemented in machine language

with some sort of compare and branch instruction. This suggests that the sequence

control mechanism of the instruction set is important.

These results are instructive to the machine instruction set designer, indicating

which types of statements occur most often and therefore should be supported in

an “optimal” fashion. However, these results do not reveal which statements use

the most time in the execution of a typical program. That is, we want to answer the

question: Given a compiled machine- language program, which statements in the

source language cause the execution of the most machine-language instructions and

what is the execution time of these instructions?

To get at this underlying phenomenon, the Patterson programs [PATT82a],

described in Appendix 4A, were compiled on the VAX, PDP- 11, and Motorola

68000 to determine the average number of machine instructions and memory references per statement type. The second and third columns in Table 15.2 show the relative frequency of occurrence of various HLL statements in a variety of programs;

the data were obtained by observing the occurrences in running programs rather

than just the number of times that statements occur in the source code. Hence

these metrics capture dynamic behavior. To obtain the data in columns four and

five (machine-instruction weighted), each value in the second and third columns is

multiplied by the number of machine instructions produced by the compiler. These

results are then normalized so that columns four and five show the relative frequency of occurrence, weighted by the number of machine instructions per HLL

statement. Similarly, the sixth and seventh columns are obtained by multiplying the

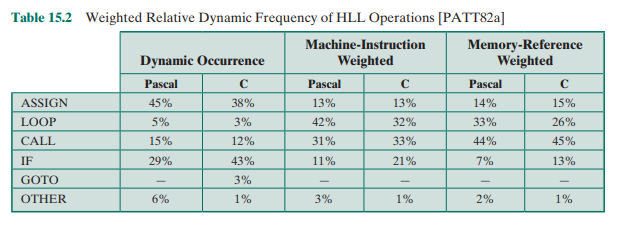
frequency of occurrence of each statement type by the relative number of memory

references caused by each statement. The data in columns four through seven provide surrogate measures of the actual time spent executing the various statement

types. The results suggest that the procedure call/return is the most time-consuming

operation in typical HLL programs.

The reader should be clear on the significance of Table 15.2. This table indicates the relative performance impact of various statement types in an HLL, when



that HLL is compiled for a typical contemporary instruction set architecture. Some

other architecture could conceivably produce different results. However, this study

produces results that are representative for contemporary complex instruction set

computer (CISC) architectures. Thus, they can provide guidance to those looking

for more efficient ways to support HLLs.

**Operands**

Much less work has been done on the occurrence of types of operands, despite the

importance of this topic. There are several aspects that are significant.

The Patterson study already referenced [PATT82a] also looked at the dynamic

frequency of occurrence of classes of variables (Table 15.3). The results, consistent

between Pascal and C programs, show that most references are to simple scalar

variables. Further, more than 80% of the scalars were local (to the procedure) variables. In addition, each reference to an array or a structure requires a reference to

an index or pointer, which again is usually a local scalar. Thus, there is a preponderance of references to scalars, and these are highly localized.

The Patterson study examined the dynamic behavior of HLL programs,

independent of the underlying architecture. As discussed before, it is necessary

to deal with actual architectures to examine program behavior more deeply. One

study, [LUND77], examined DEC- 10 instructions dynamically and found that

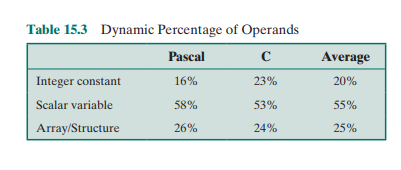
each instruction on the average references 0.5 operand in memory and 1.4 registers. Similar results are reported in [HUCK83] for C, Pascal, and FORTRAN

programs on S/370, PDP- 11, and VAX. Of course, these figures depend highly

on both the architecture and the compiler, but they do illustrate the frequency of

operand accessing

YES LUND LMAOOOO



These latter studies suggest the importance of an architecture that lends itself

to fast operand accessing, because this operation is performed so frequently. The

Patterson study suggests that a prime candidate for optimization is the mechanism

for storing and accessing local scalar variables.

**Procedure Calls**

We have seen that procedure calls and returns are an important aspect of HLL programs. The evidence (Table 15.2) suggests that these are the most time-consuming

operations in compiled HLL programs. Thus, it will be profitable to consider ways

of implementing these operations efficiently. Two aspects are significant: the number of parameters and variables that a procedure deals with, and the depth of

nesting.

Tanenbaum’s study [TANE78] found that 98% of dynamically called procedures were passed fewer than six arguments and that 92% of them used fewer

than six local scalar variables. Similar results were reported by the Berkeley RISC

team [KATE83], as shown in Table 15.4. These results show that the number of

words required per procedure activation is not large. The studies reported earlier indicated that a high proportion of operand references is to local scalar variables. These studies show that those references are in fact confined to relatively few

variables.

The same Berkeley group also looked at the pattern of procedure calls and

returns in HLL programs. They found that it is rare to have a long uninterrupted

sequence of procedure calls followed by the corresponding sequence of returns.

Rather, they found that a program remains confined to a rather narrow window of

procedure-invocation depth. This is illustrated in Figure 4.21, which was discussed

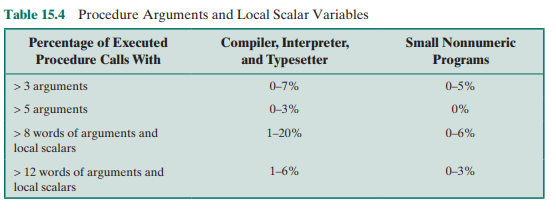
in Chapter 4. These results reinforce the conclusion that operand references are

highly localized.

**Implications**

A number of groups have looked at results such as those just reported and have concluded that the attempt to make the instruction set architecture close to HLLs is not

the most effective design strategy. Rather, the HLLs can best be supported by optimizing performance of the most time-consuming features of typical HLL programs.



Generalizing from the work of a number of researchers, three elements emerge

that, by and large, characterize RISC architectures. First, use a large number of

registers or use a compiler to optimize register usage. This is intended to optimize

operand referencing. The studies just discussed show that there are several references per HLL statement and that there is a high proportion of move (assignment)

statements. This, coupled with the locality and predominance of scalar references,

suggests that performance can be improved by reducing memory references at the

expense of more register references. Because of the locality of these references, an

expanded register set seems practical.

Second, careful attention needs to be paid to the design of instruction pipelines. Because of the high proportion of conditional branch and procedure call

instructions, a straightforward instruction pipeline will be inefficient. This manifests itself as a high proportion of instructions that are prefetched but never

executed.

Finally, an instruction set consisting of high- performance primitives is indicated. Instructions should have predictable costs (measured in execution time,

code size, and increasingly, in energy dissipation) and be consistent with a high performance implementation (which harmonizes with predictable execution-time

cost).



The results summarized in Section 15.1 point out the desirability of quick access to

operands. We have seen that there is a large proportion of assignment statements

in HLL programs, and many of these are of the simple form A d B. Also, there is

a significant number of operand accesses per HLL statement. If we couple these

results with the fact that most accesses are to local scalars, heavy reliance on register

storage is suggested.

The reason that register storage is indicated is that it is the fastest available

storage device, faster than both main memory and cache. The register file is physically small, on the same chip as the ALU and control unit, and employs much

shorter addresses than addresses for cache and memory. Thus, a strategy is needed

that will allow the most frequently accessed operands to be kept in registers and to

minimize register-memory operations.

Two basic approaches are possible, one based on software and the other

on hardware. The software approach is to rely on the compiler to maximize register usage. The compiler will attempt to assign registers to those variables that

will be used the most in a given time period. This approach requires the use

of sophisticated program-analysis algorithms. The hardware approach is simply

to use more registers so that more variables can be held in registers for longer

periods of time.

In this section, we will discuss the hardware approach. This approach has been

pioneered by the Berkeley RISC group [PATT82a]; was used in the first commercial RISC product, the Pyramid [RAGA83]; and is currently used in the popular

SPARC architecture.

**Register Windows**

On the face of it, the use of a large set of registers should decrease the need to access

memory. The design task is to organize the registers in such a fashion that this goal

is realized.

Because most operand references are to local scalars, the obvious approach

is to store these in registers, with perhaps a few registers reserved for global variables. The problem is that the definition of local changes with each procedure call

and return, operations that occur frequently. On every call, local variables must

be saved from the registers into memory, so that the registers can be reused by the

called procedure. Furthermore, parameters must be passed. On return, the variables of the calling procedure must be restored (loaded back into registers) and

results must be passed back to the calling procedure.

The solution is based on two other results reported in Section 15.1. First,

a typical procedure employs only a few passed parameters and local variables

(Table 15.4). Second, the depth of procedure activation fluctuates within a relatively narrow range (Figure 4.21). To exploit these properties, multiple small sets

of registers are used, each assigned to a different procedure. A procedure call automatically switches the processor to use a different fixed- size window of registers,

rather than saving registers in memory. Windows for adjacent procedures are overlapped to allow parameter passing.

The concept is illustrated in Figure 15.1. At any time, only one window of registers is visible and is addressable as if it were the only set of registers (e.g., addresses

0 through N - 1). The window is divided into three fixed- size areas. Parameter

registers hold parameters passed down from the procedure that called the current

procedure and hold results to be passed back up. Local registers are used for local

variables, as assigned by the compiler. Temporary registers are used to exchange

parameters and results with the next lower level (procedure called by current procedure). The temporary registers at one level are physically the same as the parameter

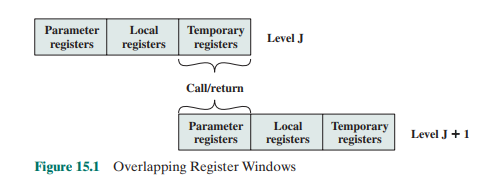
registers at the next lower level. This overlap permits parameters to be passed without the actual movement of data. Keep in mind that, except for the overlap, the registers at two different levels are physically distinct. That is, the parameter and local

registers at level J are disjoint from the local and temporary registers at level J + 1.

To handle any possible pattern of calls and returns, the number of register

windows would have to be unbounded. Instead, the register windows can be used

to hold the few most recent procedure activations. Older activations must be saved



in memory and later restored when the nesting depth decreases. Thus, the actual

organization of the register file is as a circular buffer of overlapping windows. Two

notable examples of this approach are Sun’s SPARC architecture, described in Section 15.7, and the IA-64 architecture used in Intel’s Itanium processor.

The circular organization is shown in Figure 15.2, which depicts a circular

buffer of six windows. The buffer is filled to a depth of 4 (A called B; B called C;

C called D) with procedure D active. The current- window pointer (CWP) points

to the window of the currently active procedure. Register references by a machine

instruction are offset by this pointer to determine the actual physical register. The

saved-window pointer (SWP) identifies the window most recently saved in memory.

If procedure D now calls procedure E, arguments for E are placed in D’s temporary registers (the overlap between w3 and w4) and the CWP is advanced by one

window.

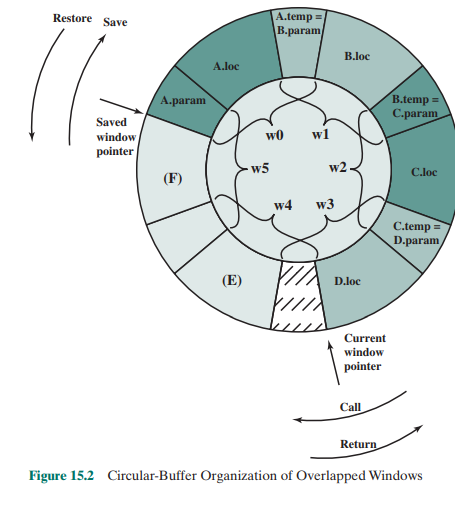
If procedure E then makes a call to procedure F, the call cannot be made with

the current status of the buffer. This is because F’s window overlaps A’s window. If

F begins to load its temporary registers, preparatory to a call, it will overwrite the

parameter registers of A (A.in). Thus, when CWP is incremented (modulo 6) so

that it becomes equal to SWP, an interrupt occurs, and A’s window is saved. Only



the first two portions (A.in and A.loc) need be saved. Then, the SWP is incremented

and the call to F proceeds. A similar interrupt can occur on returns. For example,

subsequent to the activation of F, when B returns to A, CWP is decremented and

becomes equal to SWP. This causes an interrupt that results in the restoration of

A’s window.

From the preceding, it can be seen that an N- window register file can hold

only N - 1 procedure activations. The value of N need not be large. As was mentioned in Appendix 4A, one study [TAMI83] found that, with 8 windows, a save or

restore is needed on only 1% of the calls or returns. The Berkeley RISC computers

use 8 windows of 16 registers each. The Pyramid computer employs 16 windows of

32 registers each.

**Global Variables**

The window scheme just described provides an efficient organization for storing

local scalar variables in registers. However, this scheme does not address the need

to store global variables, those accessed by more than one procedure. Two options

suggest themselves. First, variables declared as global in an HLL can be assigned

memory locations by the compiler, and all machine instructions that reference these

variables will use memory-reference operands. This is straightforward, from both the

hardware and software (compiler) points of view. However, for frequently accessed

global variables, this scheme is inefficient.

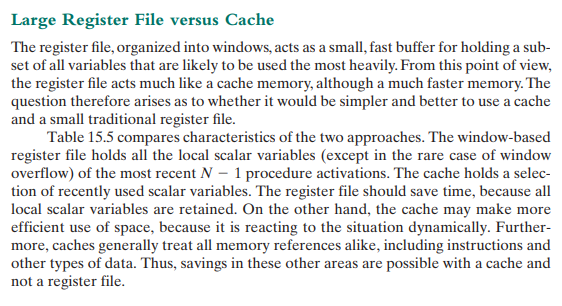
An alternative is to incorporate a set of global registers in the processor. These

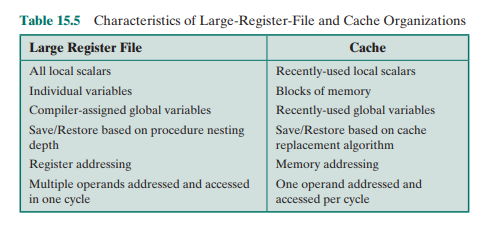
registers would be fixed in number and available to all procedures. A unified numbering scheme can be used to simplify the instruction format. For example, references to registers 0 through 7 could refer to unique global registers, and references

to registers 8 through 31 could be offset to refer to physical registers in the current

window. There is an increased hardware burden to accommodate the split in register addressing. In addition, the linker must decide which global variables should be

assigned to registers.





A register file may make inefficient use of space, because not all procedures

will need the full window space allotted to them. On the other hand, the cache

suffers from another sort of inefficiency: Data are read into the cache in blocks.

Whereas the register file contains only those variables in use, the cache reads in a

block of data, some or much of which will not be used.

The cache is capable of handling global as well as local variables. There are

usually many global scalars, but only a few of them are heavily used [KATE83]. A

cache will dynamically discover these variables and hold them. If the window-based

register file is supplemented with global registers, it too can hold some global scalars. However, when program modules are separately compiled, it is impossible for

the compiler to assign global values to registers; the linker must perform this task.

With the register file, the movement of data between registers and memory is

determined by the procedure nesting depth. Because this depth usually fluctuates

within a narrow range, the use of memory is relatively infrequent. Most cache memories are set associative with a small set size. Thus, there is the danger that other

data or instructions will compete for cache residency.

Based on the discussion so far, the choice between a large window-based register file and a cache is not clear-cut. There is one characteristic, however, in which

the register approach is clearly superior and which suggests that a cache-based system will be noticeably slower. This distinction shows up in the amount of addressing

overhead experienced by the two approaches.

Figure 15.3 illustrates the difference. To reference a local scalar in a windowbased register file, a “virtual” register number and a window number are used.

These can pass through a relatively simple decoder to select one of the physical registers. To reference a memory location in cache, a full-width memory address must

be generated. The complexity of this operation depends on the addressing mode. In

a set associative cache, a portion of the address is used to read a number of words

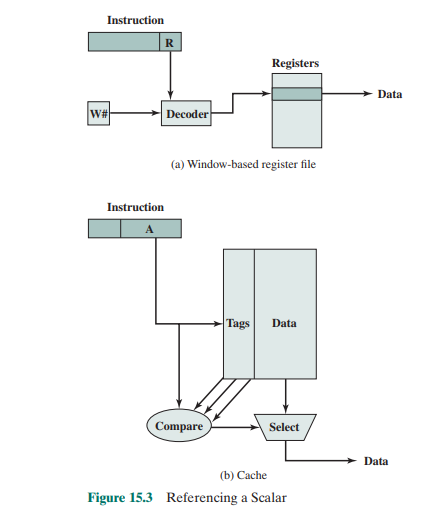
and tags equal to the set size. Another portion of the address is compared with the

tags, and one of the words that were read is selected. It should be clear that even if

the cache is as fast as the register file, the access time will be considerably longer.

Thus, from the point of view of performance, the window-based register file is superior for local scalars. Further performance improvement could be achieved by the

addition of a cache for instructions only.



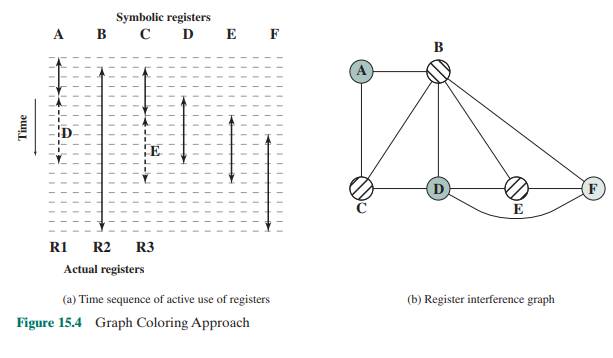


Let us assume now that only a small number (e.g., 16–32) of registers is available on the target RISC machine. In this case, optimized register usage is the responsibility of the compiler. A program written in a high-level language has, of course, no explicit references to registers (the C-language keyword register notwithstanding). Rather, program quantities are referred to symbolically. The objective of the compiler is to keep the operands for as many computations as possible in registers rather than main memory, and to minimize load-and-store operations. In general, the approach taken is as follows. Each program quantity that is a candidate for residing in a register is assigned to a symbolic or virtual register. The compiler then maps the unlimited number of symbolic registers into a fixed number of real registers. Symbolic registers whose usage does not overlap can share the same real register. If, in a particular portion of the program, there are more quantities to deal with than real registers, then some of the quantities are assigned to memory locations. Load-and-store instructions are used to position quantities in registers temporarily for computational operations.

The essence of the optimization task is to decide which quantities are to be assigned to registers at any given point in the program. The technique most commonly used in RISC compilers is known as graph coloring, which is a technique borrowed from the discipline of topology [CHAI82, CHOW86, COUT86, CHOW90]. The graph coloring problem is this. Given a graph consisting of nodes and edges, assign colors to nodes such that adjacent nodes have different colors, and do this in such a way as to minimize the number of different colors. This problem is adapted to the compiler problem in the following way. First, the program is analyzed to build a register interference graph. The nodes of the graph are the symbolic registers. If two symbolic registers are “live” during the same program fragment, then they are joined by an edge to depict interference. An attempt is then made to color the graph with n colors, where n is the number of registers. Nodes that share the same color can be assigned to the same register. If this process does not fully succeed, then those nodes that cannot be colored must be placed in memory, and loads and stores must be used to make space for the affected quantities when they are needed.

Figure 15.4 is a simple example of the process. Assume a program with six symbolic registers to be compiled into three actual registers. Figure 15.4a shows the time sequence of active use of each symbolic register. The dashed horizontal lines indicate successive instruction executions. Figure 15.4b shows the register interference graph (shading and stripes are used instead of colors). A possible coloring with three colors is indicated. Because symbolic registers A and D do not interfere, the compile can assign both of these to physical register R1. Similarly, symbolic registers C and E can be assigned to register R3. One symbolic register, F, is left uncolored and must be dealt with using loads and stores.

In general, there is a trade-off between the use of a large set of registers and compiler-based register optimization. For example, [BRAD91a] reports on a study



that modeled a RISC architecture with features similar to the Motorola 88000 and the MIPS R2000. The researchers varied the number of registers from 16 to 128, and they considered both the use of all general-purpose registers and registers split between integer and floating-point use. Their study showed that with even simple register optimization, there is little benefit to the use of more than 64 registers. With reasonably sophisticated register optimization techniques, there is only marginal performance improvement with more than 32 registers. Finally, they noted that with a small number of registers (e.g., 16), a machine with a shared register organization executes faster than one with a split organization. Similar conclusions can be drawn from [HUGU91], which reports on a study that is primarily concerned with optimizing the use of a small number of registers rather than comparing the use of large register sets with optimization efforts.



In this section, we look at some of the general characteristics of and the motivation for a reduced instruction set architecture. Specific examples will be seen later in this chapter. We begin with a discussion of motivations for contemporary complex instruction set architectures.

**Why CISC**

We have noted the trend to richer instruction sets, which include a larger number

of instructions and more complex instructions. Two principal reasons have motivated this trend: a desire to simplify compilers and a desire to improve performance.

Underlying both of these reasons was the shift to HLLs on the part of programmers;

architects attempted to design machines that provided better support for HLLs.

It is not the intent of this chapter to say that the CISC designers took the

wrong direction. Indeed, because technology continues to evolve and because architectures exist along a spectrum rather than in two neat categories, a black-and-white

assessment is unlikely ever to emerge. Thus, the comments that follow are simply

meant to point out some of the potential pitfalls in the CISC approach and to provide some understanding of the motivation of the RISC adherents.

The first of the reasons cited, compiler simplification, seems obvious, but it is

not. The task of the compiler writer is to build a compiler that generates good (fast,

small, fast and small) sequences of machine instructions for HLL programs (i.e., the

compiler views individual HLL statements in the context of surrounding HLL statements). If there are machine instructions that resemble HLL statements, this task is

simplified. This reasoning has been disputed by the RISC researchers ([HENN82],

[RADI83], [PATT82b]). They have found that complex machine instructions are

often hard to exploit because the compiler must find those cases that exactly fit

the construct. The task of optimizing the generated code to minimize code size,

reduce instruction execution count, and enhance pipelining is much more difficult

with a complex instruction set. As evidence of this, studies cited earlier in this chapter indicate that most of the instructions in a compiled program are the relatively

simple ones.

The other major reason cited is the expectation that a CISC will yield smaller,

faster programs. Let us examine both aspects of this assertion: that programs will be

smaller and that they will execute faster.

There are two advantages to smaller programs. Because the program takes up

less memory, there is a savings in that resource. With memory today being so inexpensive, this potential advantage is no longer compelling. More important, smaller

programs should improve performance, and this will happen in three ways. First,

fewer instructions means fewer instruction bytes to be fetched. Second, in a paging

environment, smaller programs occupy fewer pages, reducing page faults. Third,

more instructions fit in cache(s).

The problem with this line of reasoning is that it is far from certain that a CISC

program will be smaller than a corresponding RISC program. In many cases, the

CISC program, expressed in symbolic machine language, may be shorter (i.e., fewer

instructions), but the number of bits of memory occupied may not be noticeably

smaller. Table 15.6 shows results from three studies that compared the size of compiled C programs on a variety of machines, including RISC I, which has a reduced

instruction set architecture. Note that there is little or no savings using a CISC over

a RISC. It is also interesting to note that the VAX, which has a much more complex

instruction set than the PDP-11, achieves very little savings over the latter. These

results were confirmed by IBM researchers [RADI83], who found that the IBM 801

(a RISC) produced code that was 0.9 times the size of code on an IBM S/370. The

study used a set of PL/I programs.

There are several reasons for these rather surprising results. We have already

noted that compilers on CISCs tend to favor simpler instructions, so that the conciseness of the complex instructions seldom comes into play. Also, because there

are more instructions on a CISC, longer opcodes are required, producing longer

instructions. Finally, RISCs tend to emphasize register rather than memory references, and the former require fewer bits. An example of this last effect is discussed

presently.

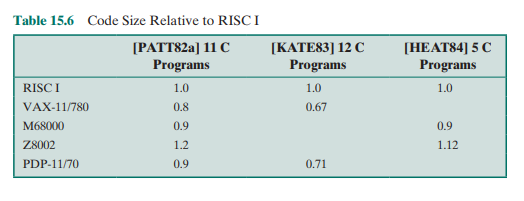
So the expectation that a CISC will produce smaller programs, with the attendant advantages, may not be realized. The second motivating factor for increasingly

complex instruction sets was that instruction execution would be faster. It seems to

make sense that a complex HLL operation will execute more quickly as a single

machine instruction rather than as a series of more primitive instructions. However,

because of the bias toward the use of those simpler instructions, this may not be so.



The entire control unit must be made more complex, and/or the microprogram control store must be made larger, to accommodate a richer instruction set. Either factor

increases the execution time of the simple instructions.

In fact, some researchers have found that the speedup in the execution of complex functions is due not so much to the power of the complex machine instructions

as to their residence in high- speed control store [RADI83]. In effect, the control

store acts as an instruction cache. Thus, the hardware architect is in the position of

trying to determine which subroutines or functions will be used most frequently and

assigning those to the control store by implementing them in microcode. The results

have been less than encouraging. On S/390 systems, instructions such as Translate

and Extended-Precision-Floating-Point-Divide reside in high-speed storage, while

the sequence involved in setting up procedure calls or initiating an interrupt handler

are in slower main memory.

Thus, it is far from clear that a trend to increasingly complex instruction sets is

appropriate. This has led a number of groups to pursue the opposite path.

**Characteristics of Reduced Instruction Set Architectures**

Although a variety of different approaches to reduced instruction set architecture

have been taken, certain characteristics are common to all of them:

■ One instruction per cycle

■ Register-to-register operations

■ Simple addressing modes

■ Simple instruction formats

Here, we provide a brief discussion of these characteristics. Specific examples are

explored later in this chapter.

The first characteristic listed is that there is one machine instruction per

machine cycle. A machine cycle is defined to be the time it takes to fetch two operands from registers, perform an ALU operation, and store the result in a register.

Thus, RISC machine instructions should be no more complicated than, and execute

about as fast as, microinstructions on CISC machines (discussed in Part Four). With

simple, one-cycle instructions, there is little or no need for microcode; the machine

instructions can be hardwired. Such instructions should execute faster than comparable machine instructions on other machines, because it is not necessary to access a

microprogram control store during instruction execution.

A second characteristic is that most operations should be register to register,

with only simple LOAD and STORE operations accessing memory. This design

feature simplifies the instruction set and therefore the control unit. For example, a

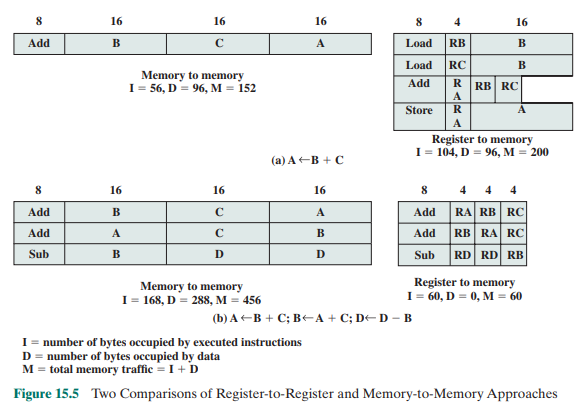
RISC instruction set may include only one or two ADD instructions (e.g., integer

add, add with carry); the VAX has 25 different ADD instructions. Another benefit

is that such an architecture encourages the optimization of register use, so that frequently accessed operands remain in high-speed storage.

This emphasis on register-to-register operations is notable for RISC designs.

Contemporary CISC machines provide such instructions but also include memory-to- memory and mixed register/memory operations. Attempts to compare these



approaches were made in the 1970s, before the appearance of RISCs. Figure 15.5a

illustrates the approach taken. Hypothetical architectures were evaluated on program size and the number of bits of memory traffic. Results such as this one led

one researcher to suggest that future architectures should contain no registers at

all [MYER78]. One wonders what he would have thought, at the time, of the RISC

machine once produced by Pyramid, which contained no less than 528 registers!

What was missing from those studies was a recognition of the frequent access

to a small number of local scalars and that, with a large bank of registers or an optimizing compiler, most operands could be kept in registers for long periods of time.

Thus, Figure 15.5b may be a fairer comparison.

A third characteristic is the use of simple addressing modes. Almost all RISC

instructions use simple register addressing. Several additional modes, such as displacement and PC-relative, may be included. Other, more complex modes can be

synthesized in software from the simple ones. Again, this design feature simplifies

the instruction set and the control unit.

A final common characteristic is the use of simple instruction formats. Generally, only one or a few formats are used. Instruction length is fixed and aligned on

word boundaries. Field locations, especially the opcode, are fixed. This design feature has a number of benefits. With fixed fields, opcode decoding and register operand accessing can occur simultaneously. Simplified formats simplify the control unit.

Instruction fetching is optimized because word-length units are fetched. Alignment on

a word boundary also means that a single instruction does not cross page boundaries.

Taken together, these characteristics can be assessed to determine the potential performance benefits of the RISC approach. A certain amount of “circumstantial

evidence” can be presented. First, more effective optimizing compilers can be developed. With more- primitive instructions, there are more opportunities for moving

functions out of loops, reorganizing code for efficiency, maximizing register utilization, and so forth. It is even possible to compute parts of complex instructions at

compile time. For example, the S/390 Move Characters (MVC) instruction moves a

string of characters from one location to another. Each time it is executed, the move

will depend on the length of the string, whether and in which direction the locations

overlap, and what the alignment characteristics are. In most cases, these will all be

known at compile time. Thus, the compiler could produce an optimized sequence of

primitive instructions for this function.

A second point, already noted, is that most instructions generated by a compiler are relatively simple anyway. It would seem reasonable that a control unit built

specifically for those instructions and using little or no microcode could execute

them faster than a comparable CISC.

A third point relates to the use of instruction pipelining. RISC researchers feel

that the instruction pipelining technique can be applied much more effectively with

a reduced instruction set. We examine this point in some detail presently.

A final, and somewhat less significant, point is that RISC processors are more

responsive to interrupts because interrupts are checked between rather elementary operations. Architectures with complex instructions either restrict interrupts to

instruction boundaries or must define specific interruptible points and implement

mechanisms for restarting an instruction.

The case for improved performance for a reduced instruction set architecture

is strong, but one could perhaps still make an argument for CISC. A number of

studies have been done, but not on machines of comparable technology and power.

Further, most studies have not attempted to separate the effects of a reduced

instruction set and the effects of a large register file. The “circumstantial evidence,”

however, is suggestive.

**CISC versus RISC Characteristics**

After the initial enthusiasm for RISC machines, there has been a growing realization

that (1) RISC designs may benefit from the inclusion of some CISC features and that

(2) CISC designs may benefit from the inclusion of some RISC features. The result is

that the more recent RISC designs, notably the PowerPC, are no longer “pure” RISC

and the more recent CISC designs, notably the Pentium II and later Pentium models,

do incorporate some RISC characteristics.

An interesting comparison in [MASH95] provides some insight into this issue.

Table 15.7 lists a number of processors and compares them across a number of characteristics. For purposes of this comparison, the following are considered typical of

a classic RISC:

1. A single instruction size.

2. That size is typically 4 bytes.

3. A small number of data addressing modes, typically less than five. This

parameter is difficult to pin down. In the table, register and literal modes

are not counted and different formats with different offset sizes are counted

Separately.

4. No indirect addressing that requires you to make one memory access to get

the address of another operand in memory.

5. No operations that combine load/store with arithmetic (e.g., add from memory, add to memory).

6. No more than one memory-addressed operand per instruction.

7. Does not support arbitrary alignment of data for load/store operations.

8. Maximum number of uses of the memory management unit (MMU) for a data

address in an instruction.

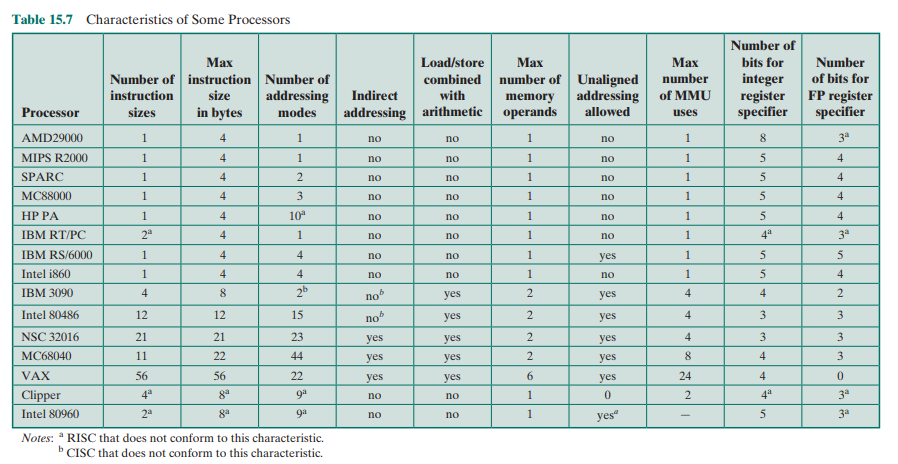
9. Number of bits for integer register specifier equal to five or more. This means

that at least 32 integer registers can be explicitly referenced at a time.

10. Number of bits for floating-point register specifier equal to four or more. This

means that at least 16 floating-point registers can be explicitly referenced at a

time.

****

**UNIT - 3 OVER FFFFFFF**

GO TAKE A BREAK DUDE LMAO