HIGH·LEVEL

LANGUAGE COMPUTER ARCHITECTURE

EDITED BY

VELJKO M. MILUTINOVIĆ

COMPUTER SCIENCE PRESS

HIGH-LEVEL LANGUAGE COMPUTER ARCHITECTURE

Edited by
Veljko M. Milutinović

with a Foreword by
Michael Flynn

COMPUTER SCIENCE PRESS

Library of Congress Cataloging-in-Publication Data

High-level language computer architecture.
(Advances in VLSI series; 1)
Bibliography: p.
Includes index.
1. Computer architecture. 2. Programming languages
(Electronic computers) I. Milutinović, Veljko.
II. Series.
QA76.9.A73T66 1988 004.2′2 87-38193
ISBN 0-88175-132-4

Printed in the United States of America

1 2 3 4 5 6 7 8 9 0 RRD 6 5 4 3 2 1 0 8 9 8

HIGH-LEVEL LANGUAGE COMPUTER ARCHITECTURE

PREFACE

High-Level Language Computer Architecture is intended for computer system architects and designers, as well as for computer managers or users, who need a deep, broad, and up-to-date knowledge of HLL computer architecture. It can be used for graduate university courses and as a research and development reference in the industrial environment.

The book's organization grew out of the following classification of high-level language computer architectures:

- Control-Driven Approach
  - Reduced Instruction Set
  - Complex Instruction Set
    - Language-Directed
    - Language-Corresponding
      - Translation in Software (type A)
      - Translation in Hardware (type B)
- Dataflow
- Reduction

This is a combination of the classifications of Treleaven (control-driven, dataflow, and reduction), Patterson (reduced and complex), and Myers (language-directed and language-corresponding types A and B). Note that various classes in the above classification are correlated. For example, sometimes it is difficult to distinguish between language-directed and language-corresponding approaches. Also, dataflow and reduction approaches often work best when combined. Finally, all existing classes can be further subdivided. However, I feel most comfortable with the above classification;

where appropriate, references are made to other related classes or possible subclasses within the classification. Consequently, the topics covered in this book, in the order of increased hardware complexity, are as follows:

Reduced Instruction Set Computer Architecture (Chapters 1–3)
Language-Directed Computer Architecture (Chapters 4–6)
Language-Corresponding Computer Architecture (Chapters 7 and 8)
Dataflow Architecture (Chapters 9 and 10)
Reduction Architecture (Chapters 11 and 12)

Each topic is represented by at least two chapters: one on the general overview and related theory and practice, the other on a selected case study. The overview chapter includes a number of small pedagogical examples. The case study chapter includes the details of the design and/or implementation for a particular project. For reduced and complex language-directed architectures, an additional application-oriented chapter has been included to reflect the increased popularity of the two topics.

In some cases, the same problem is treated in several chapters; this is because the topics are correlated, as indicated earlier. It is often useful to read about the same problem from different perspectives.

Each contributing author has been encouraged to express individual opinions and approaches. Therefore, some controversial issues are discussed in different chapters from different points of view, and sometimes with different conclusions. This presentation points clearly to the fact that the field is rapidly expanding and that final solutions to many problems have not yet been found.

Finally treating RISC architecture as a class of high-level language architecture is sometimes controversial. I believe that the primary classification criteria should include the efficiency of the compiled high-level language code, among other issues; the number of machine-level instructions that corresponds to a "typical" high-level language statement should be of secondary importance.

Veljko M. Milutinović, Editor
Purdue University

2. Making a large number of registers available to the instruction set reduces the number of data state transitions required of memory, but large register sets may increase processor cycle time as well as the number of cycles when a context switch is required.

The overall objective of modern processor design is to improve program execution: to reduce the number of cycles required to execute a program and to reduce the time it takes to execute each cycle. Considering the previously mentioned tradeoffs, we see the nature of the processor architecture design space. Minimizing the number of cycles (state transitions) may be accomplished by a rich vocabulary of operations together with large register sets, yet this may be disadvantageous to cycle time itself and hence to overall program execution time. Moreover, there is an issue of predictability: minimizing the number or location of possible alternative actions within an instruction or sequence of instructions. Highly predictable code improves the possibility for efficient pipelining of instructions. If much of instruction execution can be overlapped among individual instructions, then rapid program execution can be assured. While it is difficult to make broad generalizations, the support of improved cycle time and pipelining is seen in many of the RISC (reduced instruction set computer) approaches, while attempts to provide better encoding of instruction sets and hence smaller program size and fewer instructions to execute a program are seen in more complex instruction set (CISC) computers. These are two competing approaches for improving processor architectures.

Beyond all this is the issue of language itself. Processors execute programs that are translated from a high-level language, thus, the information available to the processor can be no more than the information available in the high-level language form of the program. The processor deals merely with a surrogate for the high-level language source program. If the original semantics of program expression are not sensitive to issues of concurrency —if they do not promote the specification of independent actions—then it is difficult for the processor to identify actions that may be executably independent. In an ensemble of processors, execution would still be limited by the sequentiality (lack of concurrency) implied by the semantics of the source program. Concurrent execution of a single source program may be limited by many things, but especially by

1. Precedence—the need for an action to be completed before a successor action can be applied to its result.
2. Global traffic—modifications to memory updating data structures which are visible to the ensemble of processors.

There has been much recent effort in the development of programming languages and processor ensembles to eliminate unneeded or implied precedence and global traffic from programs. Dataflow languages and reduction languages, as well as the machines which support these, are examples of this effort.

The common thread through all of this is the sensitivity of initial program representation (the language), the support of the attributes of language in architecture (instruction set and storage), and the creation of efficient processor designs.

Among the authors of this book are leading contributors to the respective architectural topics. Their work should shed light on issues of efficiency in language and correspondence between architecture and language, processor and architecture, and be a welcome addition to the literature in the field.

Michael Flynn
Stanford University

CONTENTS

Chapter 1

RISC PRINCIPLES, ARCHITECTURE, AND DESIGN

Charles E. Gimarc and Veljko M. Milutinović

1.1 INTRODUCTION

One of the fundamental goals of computer design is to maximize system performance. Computer performance may be thought of as the rate at which a computer can produce the desired results, given a detailed task specification, and the appropriate data. From the first stored program machines, performance increases were often realized through increased system complexity. As will be seen in this chapter, the reduced instruction set computer (RISC) approach attacks the same performance problems, using a conceptually different methodology.

1.1.1 Background Information and Initial Premises

In many computer applications, programs written in assembly language exhibit the shortest execution times. Assembly language programmers often know the computer architecture more intimately, and can write more efficient programs than compilers can generate from high level language (HLL) code. The disadvantage to this method of increasing program performance is the diverging cost of computer hardware and software. On one hand, it is now possible to construct an entire computer on a single chip of semiconductor material, its cost being very small compared to the cost of a programmer's time. On the other hand, assembly language programming is perhaps the most time-consuming method of writing software.

One way to decrease software costs is to provide assembly language instructions that perform complex tasks similar to those existing in HLLs. These tasks, such as the *character select* instruction, can be executed in one powerful assembly language instruction. A result of this philosophy is that computer instruction sets become relatively large, with many complex, special purpose, and often slow instructions. Another way to decrease software costs is to program in a HLL, and then let a compiler translate the program into assembly language. This method does not always produce the most efficient code. It has been found that it is extremely difficult to write an efficient optimizing compiler for a computer that has a very large instruction set.

How can the HLL program execute more quickly? One approach is to narrow the semantic distance between HLL concepts and the underlying architectural concepts [Meyers81]. This closing of the semantic gap supports lower software costs, since the computer more closely matches the HLL, and is therefore easier to program in an HLL. Various aspects of this issue are discussed elsewhere in the book, and will not be further elaborated here.

As instruction sets became more complex, significant increases in performance and programming efficiency occurred. However, some designers began to question whether computers with complex instruction sets (commonly referred to as CISCs, or complex instruction set computers) are as fast as they could be, having in mind the capabilities of the underlying technology. A few designers hypothesized that increased performance should be possible through a streamlined design, and instruction set simplicity. Thus, research efforts began in order to investigate how processing performance could be increased through simplified architectures. This is the root of the reduced instruction set computer (RISC) design philosophy.

Seymour Cray has been credited with some of the very early RISC concepts [Mashey86], [MIPS86b]. In an effort to design a very high speed vector processor (CDC 6600), a simple instruction set with pipelined execution was chosen. The CDC 6600 computer was register based, and all operations used data from registers local to the arithmetic units. Cray realized that all operations must be simplified for maximal performance. One complication or bottleneck in processing may cause all other operations to have degraded performance.

Starting in the mid 1970s, the IBM 801 research team investigated the effect of a small instruction set and optimizing compiler design on computer performance. They performed dynamic studies of the frequency of use of different instructions in actual application programs. In these studies, they found that approximately 20 percent of the available instructions were used 80 percent of the time. Also, complexity of the control unit necessary to support rarely used instructions, slows the execution of all instructions. Thus, through careful study of program characteristics, one can specify a smaller instruction set consisting only of instructions which are used most of the time, and execute quickly.

The first major university RISC research project was at the University of California, Berkeley (UCB). David Patterson, Carlos Séquin, and a group of graduate students investigated the effective use of VLSI in microprocessor design. To fit a powerful processor on a single chip of silicon, they looked at ways to simplify the processor design. Much of the circuitry of a modern computer CPU is dedicated to the decoding of instructions and to controlling their execution. Microprogrammed CISC computers typically dedicate over half of their circuitry to the control section. However, UCB researchers realized that a small instruction set requires a smaller area for control circuitry, and the area saved could be used by other CPU functions to boost performance. Extensive studies of application programs were performed to determine what kind of instructions are typically used, how often they execute, and what kind of CPU resources are needed to support them. These studies indicated that a large register set enhanced performance, and pointed to specific instruction classes that should be optimized for better performance. The UCB research effort produced two RISC designs that are very widely referenced in the literature. The two processors developed at UCB will be referred to as UCB-RISC I and UCB-RISC II, and are discussed in detail in Section 1.3. The mnemonics RISC and CISC emerged at this time.

Shortly after the UCB group began its work, researchers at Stanford University (SU), under the direction of John Hennessy, began looking into the relationship between computers and compilers. Their research evolved into the design and implementation of optimizing compilers, and single-cycle instruction sets. Since this research pointed to the need for single-cycle instruction sets, issues related to complex, deep pipelines were also investigated. This research resulted in a RISC processor for VLSI that will be referred to here as the SU-MIPS, and is discussed in detail in Section 1.3.3.

The result of these initial investigations was the establishment of a design philosophy for a new type of von Neumann architecture computer. Reduced instruction set computer design resulted in computers that execute instructions faster than other computers built of the same technology. It was seen that a study of the target application programs is vital in designing the instruction set and datapath. Also, it was made evident that all facets of a computer design must be considered together.

1.1.2 Essence of the RISC Design Philosophy

The design of reduced instruction set computers does not rely upon inclusion of a set of required features, but rather, is guided by a design philosophy. Since there is no strict definition of what constitutes a RISC design, a significant amount of controversy exists in categorizing a computer as RISC or CISC. This controversy is discussed in section 1.2.3.

The RISC philosophy can be stated as follows: *The effective speed of a computer can be maximized by migrating all but the most frequently used functions into software, thereby simplifying the hardware, and allowing it to be faster. Therefore, included in hardware are only those performance features* *that are pointed to by dynamic studies of HLL programs. The same philosophy applies to the instruction set design, as well as to the design of all other on-chip resources. Thus, a resource is incorporated in the architecture only if its incorporation is justified by its frequency of use, as seen from the language studies, and if its incorporation does not slow down other resources that are used more frequently.*

Common features of this design philosophy can be observed in several examples of RISC designs. The instruction set is based upon a load/store approach. Only *load* and *store* instructions access memory. No arithmetic, logic, or I/O instruction operates directly on memory contents. This is the key to single-cycle execution of instructions. Operations on register contents are always faster than operations on memory contents* (memory references usually take multiple cycles). Simple instructions and simple addressing modes are used. This simplification results in an instruction decoder that is small, fast, and relatively easy to design. It is easier to develop an optimizing compiler for a small, simple instruction set than for a complex instruction set. With few addressing modes, it is easier to map instructions onto a pipeline, since the pipeline can be designed to avoid a number of computation related conflicts. Little or no microcode is found in many RISC designs. The absence of microcode implies that there is no complex micro-CPU within the instruction decode/control section of a CPU. Pipelining is used in all RISC designs to provide simultaneous execution of multiple instructions. The depth of the pipeline (number of stages) depends upon how execution tasks are subdivided, and the time required for each stage to perform its operation. A carefully designed memory hierarchy is required for increased processing speed. This hierarchy permits fetching of instructions and operands at a rate that is high enough to prevent pipeline stalls. A typical hierarchy includes high-speed registers, cache, and/or buffers located on the CPU chip, and complex memory management schemes to support off-chip cache and memory devices. Most RISC designs include an optimizing compiler as an integral part of the computer architecture. The compiler provides an interface between the HLL and the machine language. Optimizing compilers provide a mechanism to prevent or reduce the number of pipeline faults by reorganizing code. The reorganization part of many compilers moves code around to eliminate redundant or useless statements, and to present instructions to the pipeline in the most efficient order. All instructions typically execute in the minimum possible number of CPU cycles. In some RISC designs, only load/store instructions require more than one cycle in which to execute. If all instructions take the same amount of time, the pipeline can be designed

*However, references to cached or buffered operands may be as rapid as register references, if the desired operand is in the cache, and the cache is on the CPU chip.

to recover from faults more easily, and it is easier for the compiler to reorganize and optimize the instruction sequence.

1.1.3 A Comparison of RISCs and CISCs

Design of a computer based upon RISC philosophy is not always the best solution to all computer architecture problems. RISC designs have some inherent advantages and disadvantages when compared to CISC designs.

With RISC designs, it is possible to increase performance (i.e., the speed at which instructions can be executed) through careful design of the datapath, pipeline, and other CPU resources. This possible increase is primarily a result of the simplified instruction set and physically smaller controller area. Optimizing compilers that are capable of organizing the assembly instruction stream in the most efficient order, are easier to develop when the target instruction set is small and simple. RISCs have an architectural advantage over CISCs in requiring fewer clock cycles per instruction. This is due to the relative simplicity of the control section. Thus, for a given technology or clock speed, a RISC will always execute an instruction stream faster.

Because of the design philosophy, some aspects of RISCs may be seen as a disadvantage. Most RISC instruction sets are designed after a statistical study of target application programs. As a result, RISCs may be more narrowly defined than CISCs, and not as useful in the most general-purpose computing environments. This can also be seen as an advantage since it is possible to define a RISC that is targeted for a carefully defined application class, and therefore is more efficient than a general-purpose CISC. Optimizing compilers for RISCs usually require more time to execute than the nonoptimizing compilers commonly used with CISCs. This is due to the additional compile-time tasks of pipeline management, branch prediction, and code reorganization. Since RISCs generally have fewer instructions than CISCs, each one must be more primitive, requiring more of them to perform a single HLL task. Thus, for a given HLL program, RISCs usually require more assembly instructions than CISCs. This small memory penalty is not seen as important in view of the low cost of semiconductor memory. Consequently, because of the reduced number of instructions required to represent a HLL program, CISCs have decreased memory traffic with respect to instruction fetches. Also, the penalty of increased program length is compensated by the fact that RISCs can execute most instructions in a single CPU cycle, while most CISC instructions require several cycles. Typically, a CPU cycle for a RISC requires less time than a CISC CPU cycle (assuming implementation in similar technologies).

In the area of software reliability, RISC designs may be at a disadvantage. Typically, CISCs use hardware to perform run-time tests on the validity of instructions, data, and memory accesses. Privileged instructions, protected memory, and error correction schemes in firmware are used to improve software reliability. Implementation of many reliability schemes in RISC designs is contrary to the RISC philosophy. A hardware resource should not be included in the design unless its frequency of use is relatively high. Hardware to support software reliability is not used very frequently, thus should it be included? RISCs tend to depend upon compilers for software reliability related issues.

1.1.4 Relationship of RISCs and HLL Architectures

The most important issue of the relationship between RISCs and HLL architectures is how fast the compiled HLL programs execute. It is not necessary to design an instruction set that is similar to the HLL instructions. In fact, this is often difficult to do, especially if one wants to make the assembly language useful for more than one high level language. RISC philosophy requires that the instruction set provides primitive solutions that can be combined to perform HLL programming tasks.

Can RISC architecture be treated as a subclass of HLL computer architectures? We believe: Yes! What is of primary relevance is the execution speed of compiled HLL code, not the number of machine instructions corresponding to one HLL statement.

1.1.5 Relationship of RISCs and Implementational Technologies

RISC philosophy can be applied to both VLSI and SSI/MSI designs, but a computer architecture based upon the RISC philosophy is better suited for implementation in VLSI. Through a distribution of on-chip resources that allows a minimization of the datapath cycle time, a VLSI implementation can have high performance. The reduction of hardware resources allows the entire CPU to be built on a single, small chip. In contrast, CISC designs are typically hardware intensive, and single chip implementations imply large, complex chips.

Most of the RISC designs are implemented in silicon technology. However, the last three examples in this chapter are based on Gallium Arsenide (GaAs) technology. GaAs integrated circuit designs place many severe constraints upon computer architecture, because of the nature of the technology. Useful chip area limits the size of a circuit that can be built on a single substrate. Delays for off-chip communications indicate that there are severe penalties for going off-chip. Thus RISC design, which is inherently more compact than CISC design, is an excellent candidate for GaAs implementation. In silicon, one can argue for or against RISC philosophy. In GaAs, there is no dilemma: RISC is the only choice [Milut86].

1.2 DEFINITION OF ESSENTIAL ISSUES

One of the fundamental concerns for a RISC architect is to design the maximum performance for the desired applications mix, given a finite