Order Number 8825898

M2: An architectural system for computer design

Anderson, Noel W., Ph.D.

Iowa State University, 1988

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

M2: An architectural system for computer design

by

Noel W. Anderson

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Department: Electrical Engineering and Computer Engineering
Major: Computer Engineering

Approved:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Iowa State University
Ames, Iowa

1988

TABLE OF CONTENTS

		Page
I.	FOUNDATION	1
	A. Introduction	1
	B. Background	2
	1. Embedded Computer Systems	4
	2. The Modula-2 Programming Language	8
	3. Reduced Instruction Set Computers	15
	4. Implementation Considerations	18
	C. Foreground	22
	1. The Larger Picture: Product Development	22
	2. The Narrower Picture: The M2 Project	23
II.	DESCRIPTION OF M2	28
	A. Introduction	28
	B. Silicon Compilation	30
	1. Design Automation Assistant (DAA)	35
	2. Yorktown Silicon Compiler (YSC)	36
	3. Cathedral II	37
	4. Genesil Silicon Compiler	38
	5. Concorde Silicon Compiler	39
	6. Summary	39
	C. CC: The CPU Compiler	40
	1. Description of the IL3_4S1D architecture	41
	2. Description of CC	48
	D. The Stanford Computer Architect's Workbench	52
	E. Architecture Specification and Implementation Plausibility	58
	F. A Frame-based View of System Design	60

G. AKS: The Architecture Knowledge System 63

1. Knowledge base structure 63
2. Using AKS 68
3. Extending AKS 69

H. Summary 69

III. M2 LANGUAGE TOOLS 72

A. Cmpl2: An analytic compiler 72

1. Compiler structure 73
2. Compile-time analysis 78

B. Compiling Modula-2 to IL3 79

1. Storage allocation 81
2. Procedure calls 82
3. Processes 85
4. Interrupts 86

C. IL3Sim: An IL3 simulator 87

1. Simulator description 87
2. Simulator analysis 88

IV. EVALUATION OF M2 91

A. Introduction 91

B. An Example of the Use of M2 94

1. Static analysis 94
2. Dynamic analysis 98
3. AKS analysis 98

V. CONCLUSIONS 104

A. Research Contributions 104

B. Future Work 107

VI. REFERENCES 109

VII. ACKNOWLEDGEMENTS 113

I. FOUNDATION

It was a wise and useful provision of the ancients to transmit their thoughts to posterity by recording them in treatises, so that they should not be lost, but, being developed in succeeding generations through publications in books, should gradually attain in later times, to the highest refinement of learning. And so the ancients deserve no ordinary, but unending thanks, because they did not pass on in envious silence, but took care that their ideas of every kind should be transmitted to the future in their writings.

Vitruvius

from the introduction to book 7 of *The Ten Books on Architecture*

A. Introduction

The evolution of computers over the last four decades can be characterized by increasing performance and reliability concurrent with decreasing size and cost. During the 1980s, these four parameters have reached levels that encourage the use of processors in products as inexpensive and mundane as household appliances and as expensive and vital as multi-million dollar defense systems. While the popular image of a computer remains a box with a video display and a keyboard, the reality increasingly lies in microwave ovens and the family car.

The 1980s have also seen advances in computer architecture, in implementation technology, and in languages for systems programming and logic programming. The research presented here attempts to combine advances

in these areas to improve the design of embedded processors.

This chapter provides the reader with fundamental knowledge about embedded computer systems, languages for systems programming, reduced instruction set computers (RISCs), and current implementation options. Against such a backdrop, a vision of product development in the 1990s is presented. The place this research holds in the bridge between current practice and the future possibility is then described.

Chapter 2 describes the M2 tools CC and AKS. CC is a silicon compiler surrogate while AKS is a knowledge system which attempts to make optimal choices about processor architecture and processor implementation for a particular application. Chapter 3 discusses the M2 language tools and the relation between the Modula-2 programming language and IL3_4S1D architecture. Chapter 4 considers the utility of the M2 tools through a design example. Conclusions and topics for future research are presented in Chapter 5.

B. Background

Aguero and Dasgupta [1987, p. 924] have written that "an architectural design should be viewed as a specification of constraints that are to be met by a system that will be implemented by a combination of

hardware and firmware." They go on to identify four kinds of constraints:

> Design facts define the actual features of the artifact being designed. These features may suffer alterations in the course of the design process . . . .
>
> Design objectives define desired features of the artifact as well as desired characteristics of the design process itself. Note that design facts may become objectives that might be satisfied by lower level design facts (and assumptions).
>
> Assumptions are statements about the environment or the design process itself that may help reduce the universe of solutions . . . .
>
> A style is a conglomerate of characteristics that distinguish one type of (abstract) objects from another type, and also a particular way of doing something. Since the creation of a new style is time consuming, designers frequently use a given *universe of design styles* as a source of choices in order to satisfy design objectives and refine design facts. Therefore, in order to simplify the design process, designers generally constrain their design decisions to existing styles unless the circumstances demand the creation of a new style.

The M2 style includes a wholistic view of the design process. Design optimization is accomplished by making tradeoffs across all components rather than by assembling components which have been independently optimized. Major components of a computer system are (1) the target application, (2) the language used to express algorithms for the application, (3) the language's compiler, (4) the architecture of the computer executing the algorithms, and (5) the technology used to implement the architecture.

While it is desirable to design as generally as possible, limited resources often require implementations to be focussed on particular cases. Such is the situation with this research. An attempt has been made to keep the structure of M2 extensible so that it can be readily used in studies with assumptions different from those listed below:

1. The application area is limited to embedded computer systems
2. The programming language used is Modula-2
3. The architecture used is a 4-stage pipelined RISC
4. The implementation is in bipolar transistor-transistor logic (TTL).

1. Embedded Computer Systems

An embedded computer system is a computer which provides services as part of a larger system. These services may include processing information from other subsystems, communicating information between subsystems, and/or controlling physical processes. Embedded computer systems must often support the following features:

1. Multiple tasks
2. Time critical interrupt processing
3. Efficient run-time error handling
4. Economical development of reliable software

Multitasking and external interrupt handling. When multiple tasks are executed by a processor, control

may be passed between tasks when specifically prescribed by software or in response to an interrupt. The frequency of task transfers and the processing needed to actually make the transfer are factors which proportionally decrease the throughput of a processor. Thus systems should be designed with minimal transfers and/or transfer overhead.

An interrupt is an unscheduled request to transfer processor control from one task to another. The interrupt may originate outside the processor as a response to an external event or from within the processor as a response to an execution problem. A task invoked by an external interrupt must often complete its work by a specified time or within a given time interval. It should be able to retain the processor when interrupts for lower priority tasks occur in order to expedite its completion.

a. **Internal interrupts** Internal interrupts typically result from arithmetic errors, attempted execution of an illegal instruction, page fault, hardware failure, or execution of an explicit software interrupt instruction. Arithmetic errors may occur when the result of an arithmetic operation cannot be represented with the available bits or when the operation is not defined for one of the operands. Overflow and divide-by-zero are respective examples. Detection of arithmetic errors can

be done implicitly in the object code generated by the compiler, explicitly in the source code written by the programmer, or implicitly by hardware in the processor.

The best method of arithmetic error detection is application dependent. Some factors influencing the choice include speed requirements of the application, software development costs, hardware development costs, the ease of recovering from detected errors, and the consequences of not adequately recovering from an error. Compiler generated checks ensure that error detection is always performed, but can degrade throughput by a factor of 2-3 [Powell, 1984] [Anderson, 1985]. Programmer generated checks can perform tests only when needed, but this approach increases the complexity of the source code and can lower programmer productivity. Hardware checks increase processor complexity and may decrease throughput by lengthening the basic cycle time (see section on RISCs).

For purposes of this research, the other internal sources of interrupts are presumed away. The attempted execution of an illegal instruction is often the result of an error in assembly language programming or an attempt by a human user to circumvent operating system resource protection. Earlier in this chapter it was stated that all software would be written in Modula-2. When all executable code is generated by a compiler,

instruction legality checking can be moved from hardware to the compiler as was done on the IBM 801 [Radin, 1982]. Since the processor is embedded in a larger system, it is assumed that it is either impossible or pointless for a human user to attempt to execute an illegal instruction.

Furthermore, it is assumed that embedded systems will not require virtual memory and not have to handle page faults; that internal hardware failure is not visible to the processor; and that services from a run-time support library will be accomplished through a procedure call interface rather than software interrupt interface.

b. Software modularity Source program modularity is seen as a significant way to support the economical development of reliable embedded system software. Programs partitioned into modules promote reuse of software in other projects requiring the same functionality and lowers development costs. Encapsulation of software and data aid reliability by supporting the use of previously debugged software and by requiring access to code and data to be through compiler-enforced interfaces. More will be said about modules in the next subsection.

By contrast, unmodularized software may contain code which acts on globally accessible data and bypasses procedures written specifically to act on the data.

Changes in data structures may require more extensive changes to software than would otherwise be necessary and increases the likelihood that the modifications would introduce bugs rather than eliminate them.

2. The Modula-2 Programming Language

Modula-2 is a structured, high-level programming language developed by Niklaus Wirth [Wirth, 1983]. It is a descendent of two other languages developed by Wirth, Pascal and Modula. The syntax and semantics are very similar to those of Pascal with the improvements listed in Figure 1-1.

- Added data types BITSET, CARDINAL, and PROC
- Open array parameters for procedures
- Short circuit evaluation of Boolean expressions
- CASE statement supports ranges as labels
- CASE statement supports default ELSE clause
- FOR statement supports index increments other than 1
- LOOP..EXIT..END statement for generalized iteration
- Support for systems programming
  - Multitasking through co-routines
  - Interrupt support
  - Low-level facilities
  - Modules which separate implementation from definition

Figure 1-1. Modula-2 improvements over Pascal

There is a growing number of introductory texts for Modula-2. The discussion here will focus on support for systems programming in general and embedded systems programming in particular.

a. **Multitasking and interrupt handling** The multitasking support provided by Modula-2 is directed towards loosely coupled processes running on a single processor [Wirth 1983, p. 128]. Concurrent execution of tasks is not possible on a single processor, but quasi-concurrent execution is obtained by implementing tasks as co-routines. Co-routines do not have a hierarchical relationship like that typically found between procedures in a program. Rather, they are on an equal level and are able to synchronize activity and to share data.

Co-routines are used to implement the Modula-2 data type PROCESS. The data type is exported from the module SYSTEM and is realized as a parameterless procedure declared at the outermost level of a program. Also exported from SYSTEM are a procedure to instantiate processes, NEWPROCESS, and one to allow transfer of control between processes, TRANSFER. If any process ends, the whole program ends. Thus no procedure is provided to dynamically kill processes and reclaim their resources.

Synchronization and communication between processes is supported through a module called Processes. Whereas

the type PROCESS and procedures NEWPROCESS and TRANSFER are generally inaccessible to the programmer, the module Processes can be readily tailored to an application. Ford and Weiner [Ford and Wiener 1986] provide detailed examples.

Interrupt service routines (ISRs) can be written completely in Modula-2. They are viewed as conventional processes with two extensions. The first is the procedure IOTRANSFER exported from the module SYSTEM. ISRs differ from conventional processes in that control can be transferred to an ISR from any procedure at any time. Control between conventional processes is accomplished solely through the TRANSFER procedure as specified in the source program. The IOTRANSFER procedure is used to initialize the data structures of a given ISR and prepare it for invocation, say, by entering its address in a table of interrupt vectors. The ISR is dormant until its interrupt occurs. Control is transferred to the ISR which services the interrupt and then transfers control back to the interrupted process.

The second extension is the support of the monitor concept for preventing processes from being interrupted while operating on data shared with another process. It is implemented by collecting shared data and procedures which act on that data into a module and then associating a priority with the module. The meaning of the priority

and the ability of interrupts to pre-empt other interrupts is implementation dependent.

c. **Data types for systems programming** Modula-2 also supports system programming through the inclusion of the types BYTE, WORD, and BITSET as well as the ability to assign variables to absolute locations. Communication between a processor and external devices is usually accomplished through a parallel or serial port. Integrated circuits implementing ports have control and data registers with specific addresses. In Modula-2, these registers can be declared as variables of a specified type.

The type BITSET treats a word as an array of bits and is an elegant and efficient way to represent control data. Values of type BYTE are eight bits in size while those of type WORD are one word long. The only operation permitted with each of the two types is assignment, but values with either type can be transferred to a variable of any type which has the same size. Thus data from an external devices can be manipulated before their type is established. Another use of the types BYTE and WORD is in the writing of generic procedures to operate on open arrays. Generic procedures decrease the amount of software needed for a particular program and improve the reusability of the code.

d. Software modularity Modules which do not contain a main program or are not contained within another module are split into definition and implementation parts. These parts are compiled separately with the requirement that the definition module must be compiled before the implementation module is.

A compiled definition module provides to other modules the fact that the module actually exists and the names of the constants, variables, types, and procedures it makes available to client modules. Information on type sizes, variable types, parameter bindings, and also the date of definition module compilation ensure that external procedure calls are consistent with external procedure implementations. This checking allows inconsistency problems to be detected at compile and link time by software rather than at test and integrate time by the software engineer. Development costs are reduced and software reliability improved.

The separation of definition and implementation parts facilitates software development in several additional ways. The actual data structures and algorithms used to implement an abstract data type can be truly hidden from client modules. This prevents programmers from circumventing the prescribed interfaces and writing code dependent on unspecified implementation

details. Hardware dependencies can be isolated from abstract interfaces, improving program structure and portability.

Modules facilitate the partitioning of software for implementation by a number of programmers. The partitioning also facilitates software maintenance by collecting and isolating related implementation details. The implementation can be changed with minimal, perhaps zero, side effects on code outside the module. Furthermore, fixing a problem in a module fixes it simultaneously in all programs which use the module after re-linking the client programs.

Modules are not without their drawbacks. The need for separate definition and implementation parts slightly increases source program size. Second, object code size can increase significantly if all code for a module is linked into client programs. Intelligent linkers which include only code needed by a program are becoming more common [Logitech 1987] [Borland 1987]. Finally, current programming environments do not provide much support for module cataloging and management. Programmers can easily spend more time leafing through notebooks for procedure names and parameter lists than is spent in writing code. Computer-aided programming is a current area of research in software engineering.

e. Comparison with C and ADA The final topic to be considered in this brief look at Modula-2 is how it compares with two other systems programming languages: C and Ada (tm). Wiatrowski and Wiener [1987] provide a detailed comparison of C and Modula-2. The areas in which C is currently superior are execution speed, portability of source code, and coding flexibility. The execution speed and portability benefits are due more to market factors than to the languages themselves. The execution speed advantage comes principally from better code generators needed by C compilers to stay competitive in the market place. Increased portability comes from the fact that a C compiler is typically one of the first compilers to be developed for a new computer because of the large base of extant programs, including the UNIX (tm) operating system.

The flexibility of C is due, in part, to its weak type checking relative to Modula-2. Modula-2 goes beyond independent compilation of source code files often available with C compilers and supports separate compilation of modules. This adds procedure and type checking across module boundaries to independent compilation [Wirth 1983] and allows module interface errors to be detected at compile time rather than at system integration time. The result is software which is more economical to develop and more reliable when in use.

Modules also provide superior support for data abstraction and other modern software development paradigms.

Comparing Modula-2 to Ada is comparing a small and simple language to a large and complex one. Ada is superior to Modula-2 in its support for floating and fixed point arithmetic, exception handling, generics, concurrent processing, and embedded systems support [Ford and Wiener 1986]. The widespread adoption of the IEEE 754 standard for floating point arithmetic is reducing the need to support a diverse collection of machine-dependent floating-point specifications. Ada compilers are typically slow and require a relatively large amount of memory because of the complexity of the language. For many applications, the speed and space penalty is more significant than the added functionality which is seldom, if ever, used.

3. Reduced Instruction Set Computers

Reduced Instruction Set Computers (RISCs) are defined as machines which generally meet the following criteria proposed by David Patterson and Carlo Sequin from the University of California, Berkeley [Patterson and Sequin 1982]:

1. Execute one instruction per cycle.
2. Have all instructions the same size.

3. Access memory with only load and store instructions; the rest operate between registers.

4. Support high-level languages.

As C. Gordon Bell has noted, RISC architectures have been around since the earliest days of electronic computing [Bell 1986]. In the two decades between Eniac and the IBM 360, processor architectures had to be simple to keep implementation costs down and reliability up. The development of integrated circuits and the flexibility of implementing instruction sets with microprogramming initiated a trend towards larger instruction sets operating on more data types with more addressing modes. In the late 1970s and early 1980s, the assumptions underlying the trend were called into question and research was conducted and is continuing at a growing number of universities and companies. Furthermore, RISC architectures have been the basis for recent commercial products from firms such as IBM, Hewlett-Packard, and Motorola [Gimarc and Milutinovic 1987].

a. **Pipelining** A 4-stage pipeline was selected for this research based on reports in the literature. Pipelining achieves temporal parallelism of instruction execution. "To achieve pipelining, one must subdivide the input task (process) into a sequence of subtasks, each of which can be executed by a specialized hardware

stage that operates concurrently with other stages in the pipeline" [Hwang and Briggs 1984, p. 145]. The maximum speedup of an n-stage pipelined processor relative to a processor with the same instruction set which executes one instruction at a time is roughly equal to n.

It would seem, then, that larger values of n should be better and this is true up to a point. The actual speedup is always less than the maximum speedup for two reasons. The first reason is dependencies between data being processed in various stages of the pipeline. An earlier stage cannot perform an operation on data which is being modified by a later stage. In such cases, earlier stages must be idle until the needed data is available. Larger values of n increase the likelihood of data dependencies.

The second reason is due to program branches. When a branch is detected in a particular stage, instructions being executed in earlier stages must be flushed and the pipeline refilled. Larger values of n result in more partially executed instructions being flushed when a branch occurs. They also make techniques such as delayed branches less effective since there are a larger number of instructions to be rearranged.

Katevenis [1985] has described the trade-offs made between the 2-stage pipeline of RISC I and the 3-stage pipeline of RISC II as well as trade-offs made between

the 3-stage RISC II and the 4-stage IBM 801. The RISC II pipeline allows one instruction or data access to memory during each clock cycle. LOAD and STORE operation force the pipeline to stall one cycle while data memory is being accessed. This causes the speedup to be significantly less than the maximum possible value of three. In SPUR, a successor to RISC II, the pipeline was divided into four stages which eliminated the need to stall for LOAD and STORE instructions and thus increased instruction throughput [Hill et al. 1986, p. 16].

Researchers at McDonnell Douglas Corporation settled on a 4-stage pipeline after studies indicated that potential benefits from a longer pipeline were offset by the drawbacks described above. More specifically, they found that "With a four-stage pipeline, we can run the microprocessor's clock at least 25 percent slower than with a six-stage pipeline and still achieve the same net throughput" [Rasset et al. 1986, p. 64]. A recent survey of research and commercial RISC processors showed most had three to five stages in their pipelines [Gimarc and Milutinovic 1987, p. 64].

4. Implementation Considerations

Implementing a design involves organizing people who use tools to fabricate the finished product from available materials. Ideally, the specification of an

architecture would be completely isolated from its implementation to facilitate a top-down design methodology. Different architectures would be evaluated using abstract criteria with the guarantee that selecting the best abstract design would imply the selection of the best possible implementation.

The actual situation is quite different: architecture and implementation are intimately related. This is because the criteria used to evaluate architectures are physical rather than abstract in nature. Speed, size, and power measures have primacy over cycle counts, orthogonality, and regularity. RISC research has shown that these two domains do not need to be in opposition, but, in fact, can be complementary.

The computer architect needs to have knowledge of implementation options and of the effect each has on the architecture and final performance. This is in addition to the knowledge the architect must have about the synergy between the software and architecture/implementation. Because of the breadth of design factors, recent architectures have tended to be cross-disciplinary committee efforts. Their end products can reflect the background of the members as much as current technical possibilities. Committee efforts also suffer from loss of conceptual integrity in the

architecture [Brooks 1975] as well as high personnel costs.

Compounding the problem are the diversity of implementation options and the rate at which their particulars change. This difficulty is endemic to the entire computer field and makes it hard for professionals to be both productive on a project and current in their knowledge. Silicon technologies improve, gallium-arsinide offers high speed with a different set of design constraints, and fiber optics may solve the pin-number explosion on integrated circuits. What is a good choice early in the design process might be unacceptable when a processor goes into production.

While one benefit of the M2 system is better management of changing implementation technology, the emphasis in this research is on demonstrating principles rather than on producing a state-of-the-art processor. The implementation technology considered here is bipolar transistor-transistor-logic (TTL). TTL was most popular in the early to mid-1970s and has become obsolescent as LSI and VLSI MOSFET fabrication became the technology of choice. There are three major reasons why TTL has been selected.

First, there is a large and stable set of information on available components. The components used in the research are generally 7400 series TTL which has a

number of sub-families with different speed, power, and cost attributes. These data are widely available and its format not only helped popularize TTL, but was used to popularize a successor technology [Birkner 1987]. In contrast, there is no analog to a TTL data book for VLSI fabrication.

Second, the author is more familiar with TTL than with other implementation options. Limited personnel resources favored going with a known technology rather than exploring and adopting others.

Finally, M2 is destined for the academic rather than commercial environment. Most introductory computer engineering courses focus on the gate and register transfer level of computers with corresponding laboratory work done using 7400 series small scale integration (SSI) and medium scale integration (MSI) parts. VLSI design is typically a senior elective and often does not include a laboratory because of large workstation and fabrication costs. Thus M2 is well suited for use in upper level architecture and design courses in most universities. As hands-on VLSI laboratories become economically feasible [Soma 1988], the M2 database can be updated to include data for the added technology [Heinbuch 1988].

C. Foreground

1. The Larger Picture: Product Development

Computers are embedded in products for a variety of reasons including those listed below.

1. Improved capability
2. Improved reliability
3. Improved integration with other sub-systems
4. Improved user interface
5. Reduced manufacturing cost
6. Reduced operating costs
7. Reduced space and energy requirements

As the use of embedded computers continues to increase into the 1990s, the demand for new designs will press the supply of people to design them, leading to higher design costs and longer design times. This comes in a period when market conditions are mandating lower product costs and shorter development cycles [Bussey and Sease 1988].

Action is being taken on a number of fronts to overcome these problems. Efforts include more efficient management practices [Cortes-Comerer 1987], improved computer-aided design (CAD) tools [Katz 1987], and the

capturing and dissemination of design knowledge as expert systems [Kim 1988]. These diverse fields are coming together with a new science of design as the foundation [Gajski and Thomas 1988].

As computers supplant humans in the selection, placement, and connection of computer components, human designers can focus their attention to higher level concerns such as customer needs and the specification of system behavior. Thus, in the next decade, a design engineer and customer may work together to specify the behavior of an embedded computer system by developing a program in Modula-2 and by defining constraints that the system must meet. The program and constraints will then be processed by a number of software tools to design an application specific integrated circuit (ASIC) which is the embedded computer. Furthermore, the tools may develop several implementations which meet the constraints and calculate how cost and performance will very with anticipated changes in markets and technology. Thus a family of systems can be derived from a single specification.

2. The Narrower Picture: The M2 Project

At the beginning of the M2 Project, the goal was to develop a single architecture optimized for embedded computer systems. This raised two questions: what it

meant for an architecture to be optimized and what characterized the archetypical embedded computer system. Traditionally these issues are addressed by identifying a set of representative benchmark programs for an environment and then measuring selected static and dynamic characteristics of those programs. These characteristics often include execution speed, program code size, and bus activity. Conclusions about processor architectures are drawn using the data gathered from particular implementations.

This approach lacks generality. First of all, optimization is often given a narrow definition limited to code size and/or execution speed. Power, physical size, compatibility with older designs, anticipated advances in technology, and life cycle costs are typically excluded. Depending on the application, the relative weights of these and other factors will vary. Their inclusion in architecture evaluation increases the complexity of the design process.

Second, benchmark data are often sensitive to the benchmark workload. Processor behavior is typically a non-linear function of the workload and should be evaluated for best, worst, and typical cases. A gradual degradation of performance as the workload moves from typical to worst cases is usually, but not always, desired. The requirements of the application determines

which is best and influences what architecture should be considered the best. The inclusion of data for multiple workloads and performance fall-off increases the complexity of the design process.

Third, the benchmark program(s) and data may not be representative of a particular application. In the case of embedded processors which run only a single program, the best benchmark is the actual application program. The problem is more significant for a general purpose processor which might be called upon to execute programs with diverse characteristics. Collecting and interpreting data for various programs executed on a variety of candidate architectures increases the complexity of the design process.

Finally, the confidence that a particular architecture is optimal for a particular application is often low. This is because the associated "proof" is often presented informally and may contain restrictive, ambiguous, or implicit assumptions [Aguero and Dasgupta 1987, p. 922]. This complicates the assessment of a particular case and makes comparisons of various cases difficult, especially when the cases were analyzed with different simplifying assumptions. Maintaining a record of assumptions, their implications, and the relationships between the two categories increases the complexity of the design process.

The confident specification of an architecture requires that included features be consistent with the needs of the application and be realizable with the available technology. The choices must be based on a clear definition of the design requirements, accurate data on application behavior, and complete information about implementation options. Furthermore, the design process must be conducted in a logical and reproducible fashion so the specification is correct and its rationale communicable. This entails the collection and evaluation of information which increases the complexity of design process.

Unless adequately managed, the complexity can reach a point where its effects corrupt the optimization process. The corruption can take a number of forms. If the optimization is done entirely by humans, restrictive simplifying assumptions may be introduced or logical errors may occur. A lack of analysis tools could limit the amount of data available to those making the decisions or the decision makers might not have the expertise to use all the data which are available.

In light of this analysis of the optimization of computer architectures, it was seen that the solution to the stated difficulties could be applied to more than embedded computer systems. The original goal of an optimized architecture is now qualified to be for a

particular application, but the overall research result is a general means for finding optimized architectures. More specifically, the research shows how application software can be analyzed statically at compile-time and dynamically through simulation. A knowledge-based system implemented in Prolog can use the application software data and knowledge about implementation options to determine which elements of a design space meet user-specified constraints. The next two chapters describe more fully the tools needed to accomplish this task.

II. DESCRIPTION OF M2

Owing to this favour I need have no fear of want to the end of my life, and being thus laid under obligation I began to write this work for you [Imperator Caesar], because I saw that you have built and are now building extensively, and that in the future also you will take care that our public and private buildings shall be worthy to go down to posterity by the side of your other splendid achievements. I have drawn up definite rules to enable you, by observing them, to have personal knowledge of the quality both of existing buildings and of those yet to be constructed. For in the following books I have disclosed all the principles of the art.

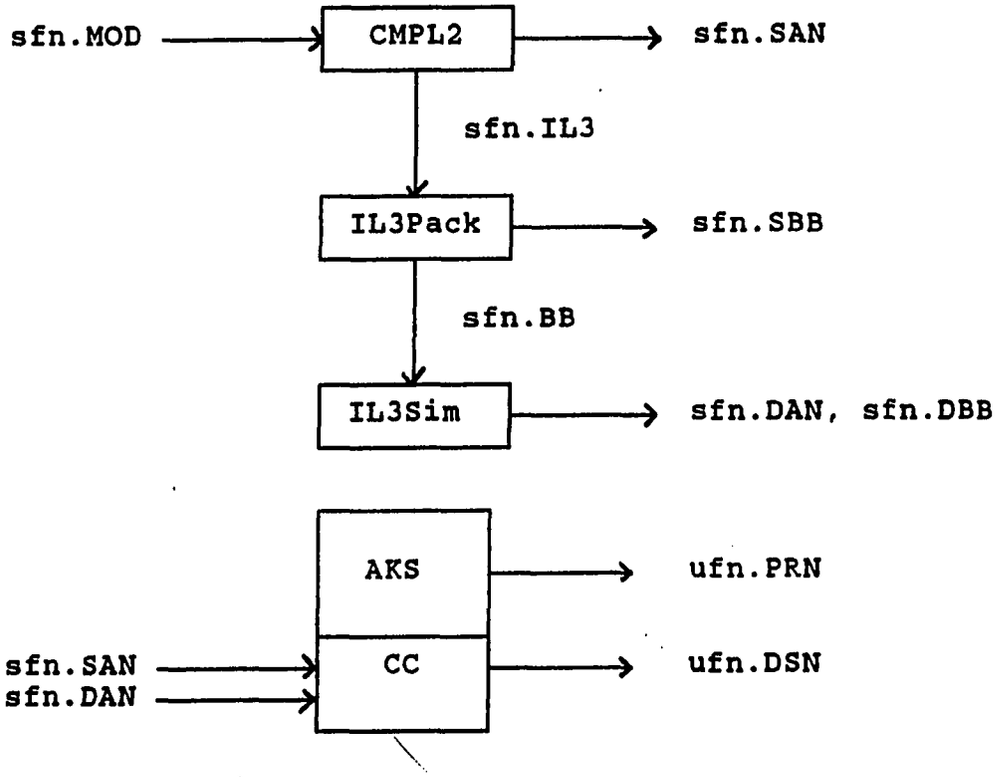
Vitruvius

from the introduction to book 1 of *The Ten Books on Architecture*

A. Introduction

The principal programs comprising M2 and the files they read and write are shown in Figure 2-1. The programs can be grouped into three classes: language tools (Cmpl2, IL3Pack, and IL3Sim), architectural tools (AKS), and implementation tools (CC). Cmpl2 translates Modula-2 source code into an intermediate language called IL3 which is optimized and packed into basic blocks by IL3Pack. The execution of the IL3 code in basic block format is done by IL3Sim.

The results of analyzing the source file, sfn.MOD, at compile time are stored in the file sfn.SAN. Simulation of the program with typical data sets provides a dynamic analysis of the program which is



Name	Meaning
sfn	Source file name
ufn	User-supplied file name
.BB	Basic block
.DAN	Dynamic analysis
.DSN	CC Design file
.PRN	ASCII report data
.MOD	Modula-2
.SAN	Static analysis

Figure 2-1. M2 programs and data files

recorded in the file sfn.DAN. The static and dynamic analysis files are used by AKS, the architectural knowledge system, to determine which algorithm-architecture-implementation triples meet constraints specified by the user. Information about implementation options is supplied by CC, a CPU compiler. CC performs the functions that a silicon compiler would in a commercial version of M2.

This chapter has two foci. It starts with an overview of silicon compilers, briefly considers some actual implementations, and goes on to examine CC and the role it plays in this research. The second focus of this chapter is the architecture knowledge system, AKS. Its consideration is preceded by a discussion of expert systems, and the concept of a computer architect's workbench.

B. Silicon Compilation

Silicon compilation has been around since the late 1970s [Johannsen 1979] and is to hardware what program compilation is to software. Traditional software compilers take a behavioral description in the form of a program as input, translate the program to an intermediate form which is optimized, and then translate the optimized intermediate form into a target machine language. Silicon compilers start with a high-level

functional description of a VLSI (Very Large Scale Integration) system, translate the source description into an intermediate structural form consisting of modules and their interconnections, perform optimization, and then produce a geometric description of the IC.

Gajski and Kuhn [1983] have introduced a Y-diagram, shown in Figure 2-2, which indicates the three representations and levels of detail within each representation. The diagram is useful not only in understanding silicon compilation, but also in comparing the methodology applied in different compilers. There is a significant variation in methodologies because of the differing internal representations of design data, algorithms which act on the data, and interactions with human experts or computer expert systems.

The complexity of software compilers is much less than that of silicon compilers because their output has more fixed constraints. That is, the sequential nature of programs, the sequential execution model of von Neumann architectures, and a finite set of fixed-sized machine instructions limit the number of choices to be made by a compiler at any instant in the translation process. By contrast, silicon compilers must consider not only the target electronic devices, but their physical size, physical shape, and also physical

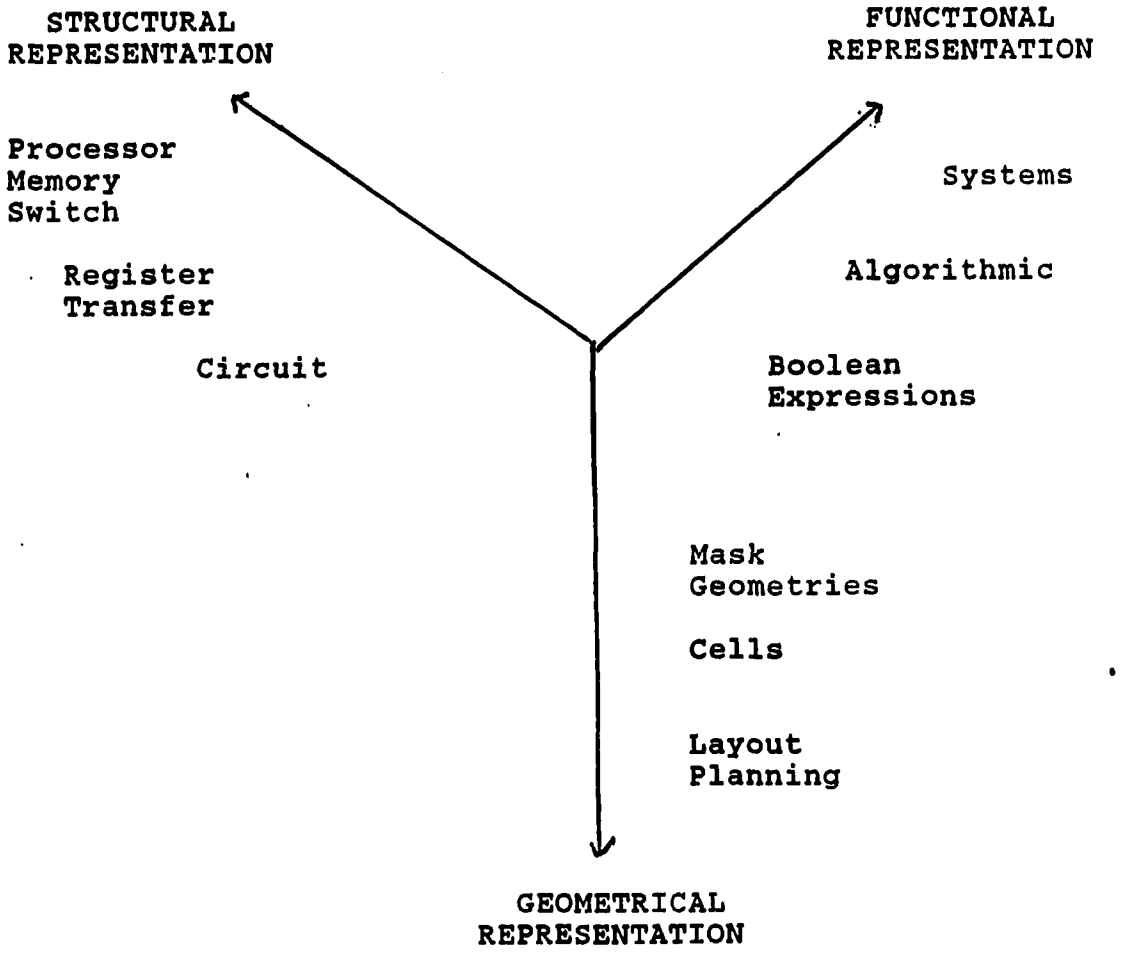


Figure 2-2. Levels of silicon compilation

interconnection and electrical interaction with neighboring devices. The step of optimally placing devices and interconnecting them is NP-Hard [Gajski and Thomas 1988].

The functional specification of a VLSI system can include an instruction set, if the system includes a processor; logic functions; and signals at the IC boundary and their relationships in time. Using a conventional programming language at this stage has several benefits. Such a specification can be executable and provide information to drive the design process. The specification process can make use of tools already developed for the language and serve as part of the design's documentation. On the negative side, conventional languages can lack features useful for concisely specifying the behavior of hardware components.

Features on the structural level include logic subsystems, data paths, memory, and I/O modules. They are often represented as parameterized procedures which can be considered instances from a library of component classes. By carefully selecting the components in the library, user needs can be met while reducing the design space the compiler must manage and reducing the effort needed to target the compiler to a different fabrication technology.

On the geometric level, the modules are placed on a map of the silicon die to form a floor plan. The cells that make up the module are placed and interconnected. Finally, the cells are compacted to minimize their area, delay, and power. The computation needed to do this can be reduced by using standardized modules and/or cells which may not minimize size and maximize speed of the final design.

In general, silicon compilation has the potential to increase the usability, quality, and profitability of embedded computer systems. Application Specific Integrated Circuits (ASICs) may account for 50% of all VLSI systems by 1990 [Rabaey et al. 1988, p. 315] and the market for silicon compilers may grow from $20 million per year in the mid-1980s to $500 million per year in 1990 [Gajski and Thomas 1988, p. vii].

There are a number of obstacles which must be overcome for silicon compilers to come into widespread use. VLSI design speed and production cost are very sensitive to the size of cells and their placement. Until recently, compiled designs were less desirable than conventional ones because of the former's inefficient use of die area. There has also been a problem with targeting the compilers to a specific fabrication process before the process became obsolete. On a larger scale, silicon compilers share a problem with other CAD/CAE

areas: Design tools have been adopted for various stages of the design process in an *ad hoc* way. Silicon compilers work best in an integrated environment where the tools work well together. Integration of old tools can be inefficient while porting older designs to new environments can be expensive.

The current state-of-the-art in silicon compilers is surveyed below. The five systems were each given a chapter in a recent anthology edited by Daniel Gajski [Gajski 1988]. The summaries for each of the systems will list their goals, describe their features, and comment on one or more design examples.

1. Design Automation Assistant (DAA)

The DAA has been developed at AT&T Bell Laboratories and Carnegie-Mellon University [Thomas et al. 1983] with the aim of "...aiding the designer by producing data paths and control sequences that implement the algorithmic system description within supplied constraints. Thus the designer can consider many alternatives before deciding on a final design" [Kowalski 1988, p. 122].

The algorithmic description of the architecture is supplied in ISPS format. A rule base written in OPS5 translates the high-level description into a hardware description. The system initially had 70 rules to guide

the translation process, but after several refining iterations the number grew to 300.

Two applications have been reported: a processor with a MOS Technology 6502 architecture and one with an IBM System/370 architecture [Thomas et al. 1983] [Kowalski 1988]. The rule base developed for the 6502 was used for the S/370. Both designs were evaluated by expert human designers and found to be good but with some room for improvement. The 6502 design required 5 hours of VAX 11/750 CPU time while the more complex System/370 required 47 hours of CPU time on a VAX 11/780.

2. Yorktown Silicon Compiler (YSC)

The YSC was developed at the IBM Thomas J. Watson Research Center at Yorktown Heights, New York, with three goals in mind. "The first goal is to achieve automatic silicon compilation from a behavioral high-level description into a chip image....The second goal is to be able to design chips that are competitive with custom manual design in performance and silicon area....As a final goal, the Yorktown silicon compiler provides a design environment whose backbone is the automatic synthesis operation" [Brayton et al. 1988, p. 207].

The functional description of the architecture is specified in the Yorktown Logic Language (YLL) which is an extension of APL. YLL is translated into Yorktown

Intermediate Format (YIF) in which parallelism and physical constraints can be expressed. The final translation step is from YIF to a circuit image which has been optimized for space and speed in the target technology. YSC provides two modes of operation: one completely automatic and one which allows human intervention into the design synthesis.

The reported application of the YSC is particularly relevant to the M2 project: it was the automatic design of a 4-stage pipelined RISC processor. The architecture is that of the IBM 801 which was originally implemented in ECL at the Watson Research Center in the late 1970s [Radin 1982]. Compared to a design done by hand, the YSC design was slightly smaller and faster.

3. Cathedral II

The Cathedral II silicon compiler is a multi-company effort which was under-written by the European Economic Community. The goal of the project was "...the automatic synthesis of synchronous multiprocessor system chips starting from a high-level behavioral description...the target application is a subset of digital signal processing (DSP) algorithms to be architecturally realized by a set of concurrent dedicated bit-parallel processors on a single chip" [Rabaey et al. 1988, pp. 311-312].

The functional description of the processor is specified in a language called SILAGE which was developed at the University of California-Berkeley for the specification of DSP algorithms. A rule-based system containing 105 rules and written in Prolog translates the specification into a network of parameterized modules. The second phase of translation takes the parameterized modules and generates the IC layout.

The translation from behavioral description to chip image is not done in a purely top-down manner. The first stage of the compiler can consult a module knowledge data base which contains implementation details in order to make better decisions. This strategy is called "meet-in-the-middle" since it has elements of both top-down and bottom-up design methodologies. Such an approach can lead to a design space explosion, but in Cathedral II it is prevented by targeting the translation to a single base architecture which has enough flexibility to satisfy the needs of many DSP applications.

4. Genesil Silicon Compiler

The Genesil Silicon Compiler is a commercial product first released by Silicon Compiler Systems in 1985 [Cheng and Mazor 1988]. It is evolving into a complete system for ASIC design and at present allows the designer to specify designs on the module level using forms and

menus. The compiler generates four types of output: IC layout, timing model, functional model, and power model.

5. Concorde Silicon Compiler

The Concorde Silicon Compiler was developed by the Seattle Silicon Corporation "to automate application-specific IC design for the system-level logic designer and to produce higher-quality designs with higher levels of productivity and production economics than standard cell and gate array design automation software" [Corbin and Snapp 1988, p. 406].

The user specifies the design on the module level through a tree of menus and forms. Analysis of the design can be performed on the block level and changes made, if desired. Once a design meets space, time, and power requirements, it is translated into an IC description in either GDSII or CIF format.

6. Summary

In less than a decade, silicon compilers have moved from an area of inchoate research to the marketplace. Design quality does not yet exceed that of the best human designers, but that may change as more human expertise is captured by expert systems within the compilers. Of particular note to the M2 project is the success of the YSC in compiling a description of the IBM 801, a 4-stage pipelined RISC, into a circuit image.

Factors which affect compiler capability and performance are the level on which the architecture is specified and the constraints placed on the specification. Providing a great deal of flexibility in the specification can cause the design space to become so large that the compiler is unacceptably slow. Limiting flexibility can increase compiler performance, but hurt the performance of the resulting design or limit the applications for which it may be used. This trade-off between speed and generality is just one of a number of areas of on-going research in silicon compilation.

C. CC: The CPU Compiler

It was shown in the preceding section that silicon compilers have reached a point where good VLSI implementations can be achieved from behavioral descriptions. These descriptions often take the form of parameterized modules, but how are the parameters determined? In most cases, there is a human designer who establishes them. In M2, they are determined by AKS from analysis of a Modula-2 program.

In order to determine parameter values, AKS uses information about candidate implementations which can only be derived from knowledge of the implementation technology. The role of CC, then, is to provide this information. For reasons discussed in Chapter 1, the

technology considered is TTL. In a full implementation, CC would be capable of generating a detailed design including the placement and interconnection of the TTL packages. In the present implementation, CC returns "reasonable" values for space, speed, power, cost, and reliability for specified implementations. They are reasonable, as will be shown shortly, because they are calculated from actual TTL data. They are unreasonable in the sense that some data and calculations include simplifying assumptions that are not valid for an actual implementation.

1. Description of the IL3_4S1D architecture

The compiler to be described in the next chapter translates programs written in Modula-2 to an intermediate language called IL3. IL3_4S1D is a 4-stage pipelined RISC processor with 1-stage branch delay whose instruction set, shown in Figure 2-3, is a subset of IL3.

CC returns implementation information for the registers, memories, and functional units in the IL3_4S1D data path which is shown in Figure 2-4. Control signals and logic, buses, and result forwarding hardware are not considered. While in many processor architectures this omission would be significant, it is less so for a RISC design. For example, in the RISC II processor, the opcode decoder occupied 0.5% of the chip area, used 0.7%

```
DATA OPERATIONS

    add, sub
    and, or, not
    asr, asl

DATA MOVEMENT

    lod, ldi, ldx
    sto, stx

    parin, parini, parinx
    parout, paroutx


COMPARISON AND CONDITIONAL BRANCHES

    gt
    ge
    eq
    ne
    le
    lt
    t
    f


CONTROL AND MISCELANEOUS

    call
    retc
    retf
    jmp
    nop
    halt
```

Figure 2-3. IL3_4S1D Instruction set

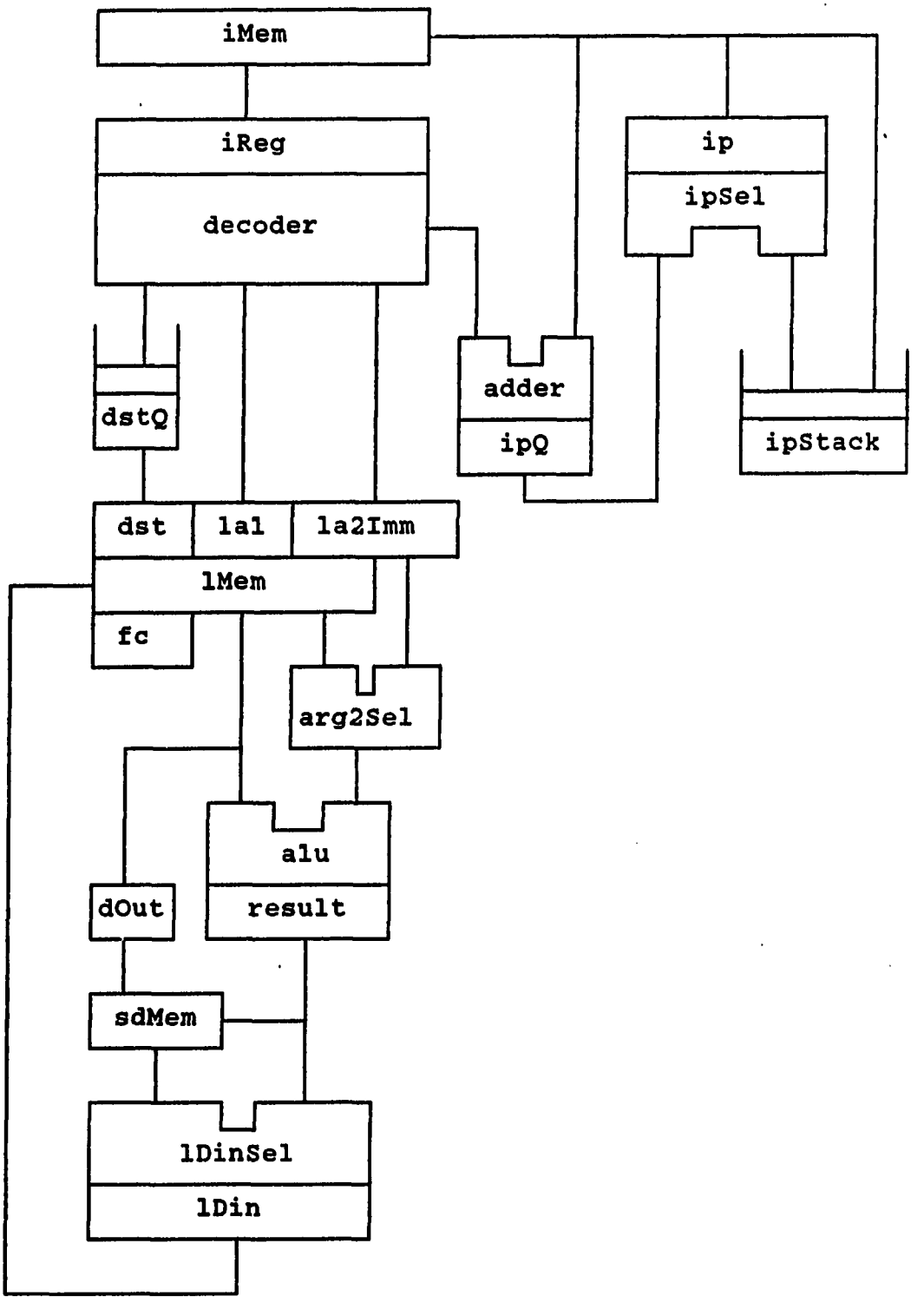


Figure 2-4. IL3_4S1D data path

of the transistors, and took less than 2% of the total design and layout time. In the M68000, a CISC architecture, 68% of the chip area is dedicated to control [Katevenis 1985].

IL3_4S1D is a Harvard architecture, namely, it has separate buses and memories for instructions (iMem) and data (sdMem). It also has a large register file with overlapping register windows (lMem) which is used to store local data. The relation between Modula-2, the IL3 execution model, and IL3_4S1D can be found in the next chapter.

Each pipeline stage consists of a set of registers which are loaded with data from the previous stage during ϕ1 of a 4-phase execution cycle. In the case of stage 1, an instruction from iMem is loaded. Functional units operate on the register contents for the remaining 3 phases as shown in Figure 2-5.

a. **Stage 1** The function of stage 1 is to fetch and decode instructions from iMem. The instruction pointer, ip, can contain a value calculated in one of four ways: (1) incrementing the previous ip value, (2) popping an address from the ipStack on return from a procedure call, (3) using the absolute address from a Jmp or call instruction, or (4) calculating an ip-relative address using an offset from a branch instruction. Since IL3_4S1D implements a delayed branch with a delay of 1

Stage	Phase 1	Phase 2	Phase 3	Phase 4
1	Load ip and read iMem [ip] Load ipQ		Load iReg Decode iReg contents	
2	Load dstQ Load dst Load la1 Load la2Imm	Read lMem to get operand	Select arg2 and perform operation	
3	Load dOut Load result	sdMem write access Arithmetic shift or idle		
4	Load lData	Idle	Write to lMem	

Figure 2-5. IL3_4S1D Timing diagram

instruction, absolute and ip-relative addresses are held in the ipQ until needed.

The instruction register, iReg, contains the last instruction fetched from memory. The instruction is decoded by decoder. Instruction fields are passed on to stage 2 and the ip. If the instruction includes an lMem destination field, the field is passed to the dstQ which has a length of 2. The delay is necessary because the result of the operation is not available for writing until the rest of the instruction reaches stage 4.

b. Stage 2 One of three actions can occur in stage 2 of IL3_4S1D: (1) operands can be fetched and an operation performed by the alu, (2) the address for an sdMem read operation can be calculated, or (3) the address and value for an sdMem write operation can be obtained.

In the case of a binary alu operation, the first operand must be located in lMem while the second may either be an immediate value or located in lMem. The address of the first operand is contained in register la1 while the address or value of the second operand is contained in the register la2Imm. Reading of lMem is performed during ø1 and ø2 while writing a previous alu operation result is done during ø3 and ø4 using an address in the register dst. The register fc is the frame counter whose use is explained in Chapter 3.

Including the alu in stage 2 lengthens the delay of the stage, but allows delayed branches to be implemented with a delay of 1 rather than with a delay of 2 which is required if the alu is located in stage 3. The trade-off between a longer stage delay and a branch delay of 1 versus a shorter stage delay and a branch delay of 2 are a subject for further study.

When calculating the address for an sdMem read, the alu may be used to do the addition for indexed and base-indexed addressing modes. When preparing for an sdMem write, the value to be written is transferred from lMem to the register dOut while the address is transferred from lMem through the alu with no operation performed on it.

c. **Stage 3** Stage 3 is used for sdMem accesses and shifting in the register result. In all other cases, no operations are performed. In the case of an sdMem read, the address is contained in the register result and value read passes through lDinSel to the register lDin. In the case of an sdMem write, the datum stored in the register dOut is written to the address located in the register result.

d. **Stage 4** Stage 4, if needed, is used to transfer values into lMem. During ø1, an lMem address is removed from dstQ and stored in register dst while the value is read into the register lDin. The stage is idle

during ø2 while 1Mem reads are completed. 1Mem writes are performed during ø3 and ø4 as described earlier.

2. Description of CC

CC is implemented in two sections: a parts database and a set of rules for combining parts into the components which comprise the IL3_4S1D architecture. The format of the part database and the sources of field values are shown in Figure 2-6. The parts database itself is shown in Figure 2-7.

CC composition rules work with register and functional unit widths which are multiples of four between 4 and 32. This choice was made because the TTL parts used are 4 bits wide. In use, AKS passes a component name, fabrication technology, and component size (width and, if appropriate, depth) to the *make* predicate which returns the size, delay, power, cost, and failure rate for the component.

The composition rules for simple registers are shown in Figure 2-8. The *member* predicate (2) verifies that the named component is in the list of registers for which the rule applies. The *slice4* predicate (3) determines the number of components needed for the register. The *part* predicate (4) is a calls to the parts database. In other predicates (5, 6, 7), the size, power, and cost of the component are calculated by simply multiplying the

FORMAT: part (ID, FAB, SIZE, DELAY, POWER, COST, REL)

ID If part is a 7400 series TTL part, ID is series number less the 74- prefix and any sub-family designator such as LS or S. Otherwise part number is unaltered.

FAB is the fabrication technology for the part:
- std for standard
- ls for low-power Schottky
- s for Schottky.
- sram for static RAM
- dram for dynamic RAM

SIZE (in^2) is based on number of pins as follows:

PINS	AREA
8	0.16
14	0.28
16	0.32
18	0.36
20	0.40
22	0.55
24	0.84
28	0.98
40	1.40

DELAY (ns) is the worst case propagation delay as given in a TTL data book. Set-up and hold times for signals are ignored here.

POWER (mW) is the maximum power supply current (Icc) times 5 Volts.

COST (dollars) is the cost for a single part as listed in a recent parts catalog or, where needed, an estimate based on prices of similar parts.

RELiability (failures/1000 hrs) is the failure rate. Failures are assumed to be independent of each other. The failure rate is assumed to be time-independent. For integrated circuits, the rate is taken from Blakeslee.

Figure 2-6. Part database format

```
/*     Num   Fab   Space   Time   Power   Cost   Rel    */
/*     ===   ===   =====   ====   =====   ====   ===    */

part (157,  std,   0.32,    14,    240,  0.59,  0.01).
part (157,   ls,   0.32,    14,     80,  0.45,  0.01).
part (157,    s,   0.32,     8,    405,  0.79,  0.01).

part (169,  std,   0.32,    23,    170,  1.09,  0.01). /*ls*/
part (169,   ls,   0.32,    23,    170,  1.09,  0.01).
part (169,    s,   0.32,    15,    800,  3.95,  0.01).

part (181,  std,   0.84,     0,    750,  1.95,  0.01).
part (181,   ls,   0.84,     0,    185,  2.49,  0.01).
part (181,    s,   0.84,     0,   1100,  2.95,  0.01).

part (182,  std,   0.32,     0,    360,  0.99,  0.01).
part (182,   ls,   0.32,     0,    360,  0.99,  0.01). /*std*/
part (182,    s,   0.32,     0,    545,  1.19,  0.01).

part (194,  std,   0.32,    26,    315,  0.79,  0.01).
part (194,   ls,   0.32,    26,    115,  0.69,  0.01).
part (194,    s,   0.32,    17,    550,  1.29,  0.01).

part (195,  std,   0.32,    26,    315,  0.89,  0.01).
part (195,  ls,    0.32,    26,    105,  0.69,  0.01).
part (195,  s,     0.32,    17,    545,  1.29,  0.01).

part (2068, std,   0.40,    40,    100,  4.95,  0.01).
part (2068, ls,    0.40,    40,    100,  4.95,  0.01).
part (2068, s,     0.40,    40,    100,  4.95,  0.01).
```

Figure 2-7. Part database

```
(1)  make (PIPO, Fab, Width, 1,
                 Size, Delay, Power, Cost, Rel) :-
(2)    member (PIPO,
              [iReg, dst, la1, la2Imm, dOut, lDin]),
(3)    slice4 (Width, W4),
(4)    part (195, Fab, S, Delay, P, C, R),
(5)    Size  = W4 * S,
(6)    Power = W4 * P,
(7)    Cost  = W4 * C,
(8)    serRelN (Rel, W4, R),
(9)    !.


(1)  make (BSR, Fab, Width, 1,
                 Size, Delay, Power, Cost, Rel) :-
(2)    member (BSR, [result]),
(3)    slice4 (Width, W4),
(4)    part (194, Fab, S, Delay, P, C, R),
(5)    Size  = W4 * S,
(6)    Power = W4 * P,
(7)    Cost  = W4 * C,
(8)    serRelN (Rel, W4, R),
(9)    !.


(1)  make (CNT, Fab, Width, 1,
                 Size, Delay, Power, Cost, Rel) :-
(2)    member (CNT, [ip, fc]),
(3)    slice4 (Width, W4),
(4)    part (169, Fab, S, Delay, P, C, R),
(5)    Size  = W4 * S,
(6)    Power = W4 * P,
(7)    Cost  = W4 * C,
(8)    serRelN (Rel, W4, R),
(9)    !.
```

Figure 2-8. Composition rules for simple registers

values for an individual part by the number of parts in the component. The calculation of component failure rates from part failure rates with the serRelN predicate (8) is based on a series model of reliability. It is assumed that part failures are independent and that the failure of any one component causes the whole system to fail [Pohm 1983, p. 133].

When parts work in parallel with other parts, such as a look-ahead carry generator working with an adder or alu, the calculation of delay or other parameters can be complicated. In such cases, the value in the parts database is ignored and a special rule for calculating the parameter is specified (Figure 2-9).

D. The Stanford Computer Architect's Workbench

Silicon compilers, as described in the previous section, take architectural specifications and translate them into circuit images. A computer architect could set parameters for an architecture based on experience, but it would be better if they could be set using information about the application program and its behavior when executed on the target architecture.

The computer architect's workbench, shown in Figure 2-10, is being developed at Stanford University to assist in the evaluation of architectures when a top-down design methodology is being employed. It "...is a set of

```
make (alu, Fab, Width, 1,
          Size, Delay, Power, Cost, Rel) :-
  Width <= 8,  /* No look-ahead carry */
  slice4 (Width, W),
  part (181, Fab, S181, _, P181, C181, R181),
  addTime (Fab, Width, Delay),
  Size  = W * S181,
  Power = W * P181,
  Cost  = W * C181,
  serRelN (Rel, W, R181),
  !.
make (alu, Fab, Width, 1,
          Size, Delay, Power, Cost, Rel) :-
  Width >= 8,  /* Look-ahead carry used */
  slice4 (Width, W),
  laCarry (Width, L),
  part (181, Fab, S181, _, P181, C181, R181),
  part (182, Fab, S182, _, P182, C182, R182),
  addTime (Fab, Width, Delay),
  Size  = (W * S181) + (L * S182),
  Power = (W * P181) + (L * P182),
  Cost  = (W * C181) + (L * C182),
  serRelN (Ra, W, R181),
  serRelN (Rb, L, R182),
  serRel2 (Rel, Ra, Rb),
  !.

addTime (std, N, 24) :- N <=  4, !.
addTime (std, N, 36) :- N <= 16, !.
addTime (std, N, 60) :- N <= 64, !.

addTime (ls,  N, 24) :- N <=  4, !.
addTime (ls,  N, 40) :- N <=  8, !.
addTime (ls,  N, 44) :- N <= 16, !.
addTime (ls,  N, 68) :- N <= 68, !.

addTime (s,   N, 11) :- N <=  4, !.
addTime (s,   N, 18) :- N <=  8, !.
addTime (s,   N, 19) :- N <= 16, !.
addTime (s,   N, 28) :- N <= 64, !.

laCarry (W, 0) :- W <=  8, !.
laCarry (W, 1) :- W <= 16, !.
laCarry (W, 2) :- W <= 32, !.
laCarry (W, 3) :- W <= 48, !.
laCarry (W, 5) :- W <= 64, !.
```

Figure 2-9. Composition rules for alu

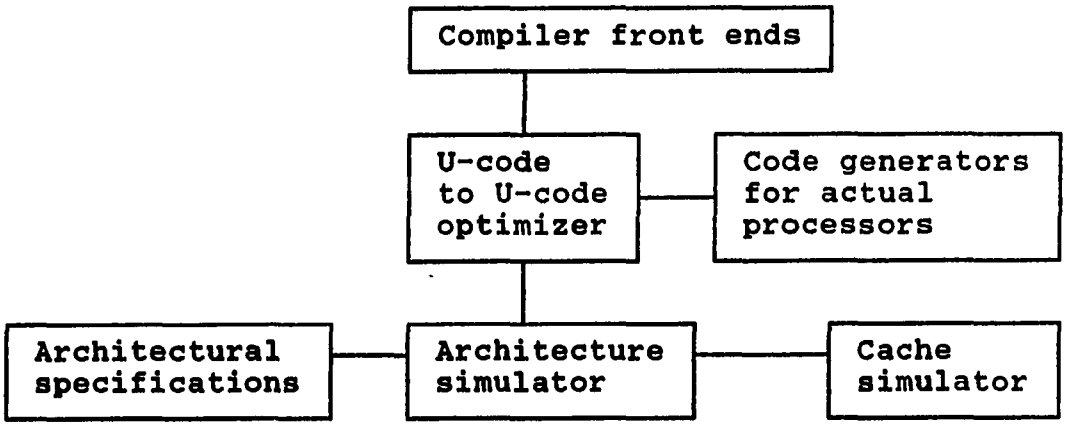


Figure 2-10. Stanford computer architect's workbench

tools...which allows the evaluation of architecture and memory system parameters for a variety of different instruction sets using a common compiler front end" [Flynn, Mitchell, and Mulder 1987, p. 72]. Benchmarks, object code optimizations, and implementation technology can all be fixed so the focus of study can be the comparison of architectural features.

Compiler front ends compile source programs to an intermediate language called U-Code which is then optimized. U-Code can then be translated to machine language for an existing processor such as the Motorola 68020 and VAX 11/780 or to a U-Code format which can be used for simulating a proposed processor. Both static and dynamic analysis can be performed with the U-Code.

The computer architect's workbench supports a technique called CARA (Compiler-Aided Research on Architectures) [Mitchell and Flynn 1988]. The workbench generates performance data using a combination of measurement, simulation, and analytical methods which can be used in the design process. It is an improvement over older methods since the analysis can be made without constructing a full compiler or simulator for proposed architecture.

Flynn, Mitchell, and Mulder [1987] have used the workbench to evaluate tradeoffs between RISC architectures and complex instruction set computer (CISC)

architectures. They chose five Pascal benchmarks believed to be representative of workstation programs. They fixed the implementation technology, the ALU design, and the data path width across the architectures they examined. They also assumed that all instructions executed in unit time and they did not consider effects of pipelining.

In general, CARA focuses on six architectural performance parameters whose values are determined from the source programs and user specified ISA values (see Figure 2-11). As will be described later in this chapter, M2 determines many of the values the user must supply for the Stanford workbench. It also determines optimal data path and instruction widths and considers the available implementation options. On the other hand, the workbench goes into more detail with instruction decoding and cache structure.

The differences between M2 and the workbench can be largely attributed to two factors: design objective and state of development. The workbench is directed toward the study of processor architectures running a number of programs whereas M2 is directed towards the selection of algorithm-architecture-implementation combinations which execute a single program as an embedded processor within a set of constraints. The fact that M2 is younger than

FOCI

- Instruction bandwidth
- Instruction decoder complexity
- Storage requirements
- Data bandwidth
- Size of cache
- Cache management policies

PARAMETERS

- Bits for simple RR opcode
- Alignment of branch targets
- Additional bits for small branch constant
- Additional bits for each memory reference
- Additional bits for each branch constant
- Registers available for temporaries
- Registers available for local variables
- Use exact register allocation?
- Number of operand sources allowed (0, 1, 2)
- Op desitination allowed in memory?
- Special register window simulation?

Figure 2-11. CARA foci and parameters

the workbench explains, in part, why the latter has a larger number of previously defined architectures.

E. Architecture Specification and Implementation Plausibility

Aquero and Dasgupta [1987] have recently addressed the difficulty of making plausible statements about the performance of a proposed computer without actually implementing the architecture. Two sources they identified for the difficulty are the informal nature of architectural proposals and the uncertain relation of architectures to their implementations. They went on to develop "plausibility-driven approach to computer architecture design" which provides a public, reproducible proof that an implementation will meet architectural constraints. As mentioned in Chapter 1, they view the constraints as the architecture.

They focussed on three aspects of the architecture as well as the implementation level. Their terms of *exoarchitecture* and *endoarchitecture* are called *architecture* and *organization, behavior* and *structure,* or *programmer's view* and *implementer's view* elsewhere. They also recognize *microarchitecture* as an instance of endoarchitecture when microcode is used in the implementation. Their work built on the S*M architecture

description language (ADL) and also resulted in its extension.

Constraints can be assigned to any of their four design aspects (see Chapter 1) and then as design progresses from one stage to another, it must be shown that the constraints are satisfied and consistent. This is done by maintaining a history for each constraint by generating a 4-tuple for each one at key steps of the design process. The 4-tuple contains the constraint itself, the current design step, the development of the constraint, and the constraint's plausibility state.

While the user specifies an initial set of constraints, additional constraints may be generated, modified, propagated, and discarded in the design process. Furthermore, an initial constraint that was assumed to be valid may be refuted during the design process. Aguero and Dasgupta have developed laws of plausibility for specifying the logically correct plausibility state of a design at a given step.

The reader is referred to the paper cited for a more extensive and detailed description of plausibility-driven computer design. At the time the paper was published, a decision had already been made for M2 to address what was called the "proof of design goodness" issue through logic programming. Aguero and Dasgupta noted that their theoretical treatment of the

plausibility problem could lead to research in architectural tools and expert systems to support their methodology. In M2, the view of "architecture as specification of constraints" has been adopted from the paper and expanded to include more than the architecture and implementation levels of design. Furthermore, the notion of plausibility logic was seminal in developing the perspectives presented in the next section.

F. A Frame-based View of System Design

In the first chapter, it was stated that the M2 design style breaks a design into four components: application (embedded systems), algorithm (Modula-2 programs), architecture (4-stage pipelined RISC), and implementation (TTL). Each of the components can be considered a model of some physical thing which may or may not exist. Each model exists at some level of abstraction above the thing. The application manifests the highest level of abstraction and the implementation design the lowest.

Analysis, synthesis, manufacturing, and testing can be added to this view as shown in Figure 2-12. Creating a physical instance of an implementation design is the act of manufacturing while verifying that a physical item is an instance of the implementation design is the act of testing. Determining the attributes of a design object

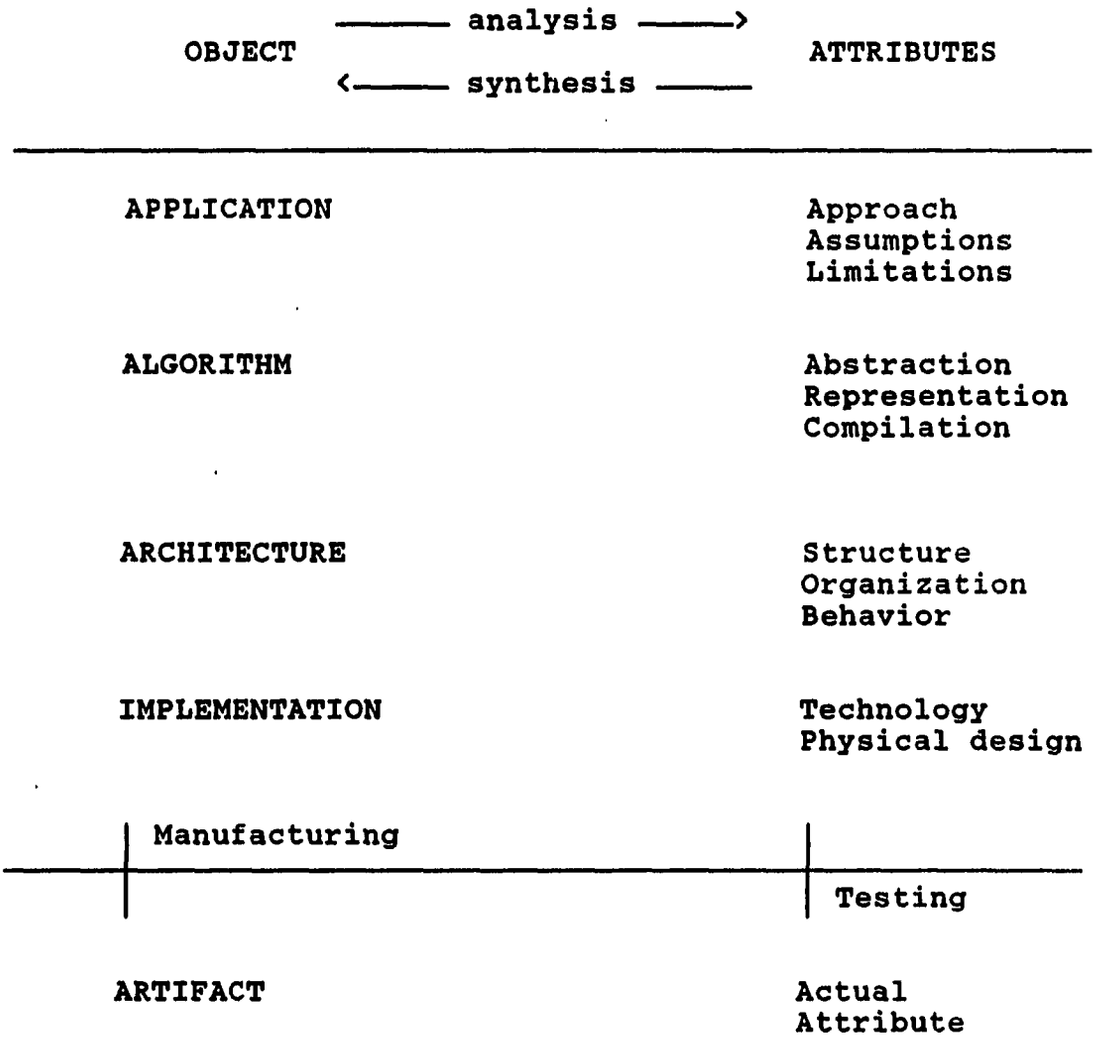


Figure 2-12. Frame-based view of system design

is analysis while generating a design object from attributes is synthesis.

M2 currently supports a top-down design methodology (conceivably it could also be extended to support abottom-up methodology). First, an application model is specified. This is followed by the specification of one or more candidate algorithms, architectures, and implementation designs. Succeeding levels contain increasingly detailed information which may be derived from known facts, obtained empirically, or obtained from consulting an external source.

Attributes for each level can be maintained in frames. A frame is an object consisting of a name and a list of attributes called slots. Frames with similar slots are instances of the same class. Classes, in M2, are the four design levels. Thus a frame for a sorting program would be an instance of the class algorithm.

The practice of organizing attributes into frames has a number of benefits. Frames have a close correspondence to the way human experts externally organize knowledge, facilitating their creation and use [Wolfgram, Dear, and Galbraith 1987, p. 56]. Frames are also readily integrated with rule-based expert systems. They provide a concise yet general means of representing knowledge while the rules concisely express how slot values are determined [Fikes and Kehler 1985]. Frames

also integrate well with truth-maintenance tools [Filman 1988].

G. AKS: The Architecture Knowledge System

AKS is a Prolog implementation of the frame-based design paradigm described in the preceding section. Its main components are a menu-driven user interface, a host file-system interface, and a frame-oriented knowledge base. This section will describe the knowledge base, how it is used, and useful extensions. A complete design example is given in Chapter 4.

1. Knowledge base structure

As described in the previous section, each instance of a design object consists of a name and a list of attributes called slots. Each slot contains four fields: a name, a facet, a value, and a constraint (see Figure 2-13). Facets indicate how slot values have been determined. Values have standard types such as string and real.

Constraints placed on slots arise from one of two sources. The first is limitations of AKS, other M2 tools, and the host computer. Respective examples include available design styles, available implementation technologies, and the maximum length of a file name. The second source of constraints is the AKS user who defines a limit on a design attribute. For example, the design

```
slot (Name, Facet, Value, Constraint)

FACETS

   system       - Value set by system
   fromList     - Value is selected by user from list
   userConstr   - No value, only constraint field is used.
   default      - Value is inherited from class
   fromUser     - Value is entered by user

VALUES

   real (real)
   str  (string)
   strL (stringList)
   fn   (string)

CONSTRAINTS

     File
       exist            - File must exist
       new              - File must not exist
       verifyOverwrite  - If file exists, verify overwrite
     List
       excl ([List])    - Value must not be in List
       incl ([List])    - Value must be in List
     Numeric
       in (Val, Val)    - Value must be in range
       out (Val, Val)   - Value must be outside range
       relOp (Val)      - Value must satisfy relational
                            operator where relOp is one of
                            >, >=, =, <>, <=, <
     Other
       none             - Value is not constrained
       system           - Constraint cannot be set by user
```

Figure 2-13. AKS slot structure

might be constrained to occupy less than 100 square inches and consume less than 5 Watts of power. At present, constraints are set when classes are initialized and apply to all instances of the class.

Some of the more significant slots for each of the four design levels are shown in Figure 2-14. For applications, the only design style supported is top-down. The composition rule (compRule) field indicates whether all instances should be considered in the design process or if some composite instance, such as a worst case, should be created from class instances and used in lower design levels (at the time of this writing, composition is not supported within AKS but is available externally). This gives the user control over the design search space.

The slots in the algorithm class may be split into three distinct classes as M2 increases in capability. These new classes would be language/compiler (language, compiler front end, and compiler back end), compiler configuration (target instruction set, compiler options, and execution model), and target execution environment (target architecture and input file).

At present, only the slots grouped under target execution environment have more than one option. The language/compiler slots can only be assigned the values Modula-2 for language, Cmp12 as the compiler front end,

APPLICATION SLOTS

- Name
- Style
- Composition rule
- Space constraint
- Time constraint
- Power constraint
- Cost constraint
- Reliability constraint
- Algorithm names

ALGORITHM SLOTS

- Name
- Language
- Compiler front end
- Compiler back end
- Target ISA
- Compiler optimizations
- Execution model
- Target architecture
- Input generation

ARCHITECTURE SLOTS

- Name
- Implementation technologies

IMPLEMENTATION SLOTS

- Name
- Space
- Time
- Power
- Cost
- Reliability

Figure 2-14. Key AKS slots

and IL3Pack as the compiler back end. Cmpl2 compiles to the IL3 instruction set with fixed optimizations and a fixed execution model which are described in the next chapter.

Distinguishing between the target instruction set and the target architecture becomes important when studying how a program developed for one processor performs on a different architecture with an enhanced instruction set. A current example of this would be executing programs compiled for the 8086 ISA on an 80386 processor. Compilers for segmented architectures such as the 8086 generate code for a number of execution models whose suitability determine and are determined by static and dynamic memory requirements.

An architecture instance is generated for each target architecture/input pair specified for each algorithm. Information on dynamic behavior of the algorithm/architecture/input triple is stored in the .DAN file for each algorithm. This dynamic data and static data gathered at compile time are used by CC to generate the space, time, power, cost, and reliability values for each implementation instance. The implementation values are compared to the application constraints to determine if the implementation is acceptable.

Each object has a time stamp slot which is used to support a rudimentary truth maintenance system [Filman

1988]. The default time stamp for a new instance is "0". When all slots have been assigned values or when values are modified, the timestamp indicates the corresponding day and time.

For the knowledge base to be valid, all objects must have a timestamp that is later than that of any ancestor and earlier than that of any descendant. Timestamps can also be compared to timestamps of files used to derive slot values. Files must be older than the objects which use them. When a timestamp violation is detected, the values of child objects can be recalculated and new timestamps assigned.

2. Using AKS

Prior to executing AKS, the user should have written and compiled the candidate algorithms. The user should also have executed the algorithms with the relevant input files. These actions generate the SAN and DAN files used by AKS.

When AKS is invoked, the only objects present are the four design classes which are in an uninitialized state. Assuming that AKS is being used in a new application, the user selects "Modify All" from the menu. Starting with the application level, the user is asked to set constraints on instances of each class.

From the main menu, the user then selects "Design" which causes AKS to generate class instances that lead to implementation candidates. At that point the knowledge base can be examined, modified (leading to recalculation when implemented), or saved. Saving can be done for either future use with M2 (not yet implemented) or with report generation software.

3. Extending AKS

The method of extending AKS is typical for extending knowledge bases using frames. Slots are added to the classes along with rules for initializing and otherwise managing them. Four types of extensions and instances of those extensions are shown in Figure 2-15. They include increasing what can be done with knowledge residing in the knowledge base, reducing the amount of user intervention needed in the design process, providing the user with a better explanation of reasoning leading to particular values or conclusions, and expanding the number of sources from which data can be obtained.

H. Summary

The first part of this chapter described the current state of silicon compilation. A knowledge-based system such as AKS can be used to determine values assigned to module parameters which serve as the input to the silicon compiler. AKS guides the user through a four-level top-

INCREASED INTERNAL FUNCTIONALITY

- Addition of bottom-up methodology which would suggest which known applications could effectively use a given processor implementation
- Increase the number of languages and target instruction sets supported
- Enhance memory design capabilities, particularly with respect to cache memory
- Increase the number of implementation parameters which can be handled (eg radiation tolerance, manufacturing cost)
- Refine the rules used at all levels of design

REDUCED USER INTERVENTION

- Automatically detecting the need and then performing compilation and simulation
- Improved detection of constraint violation and automatic pruning of design tree

EXPLANATION FACILITIES

- Implementation of an explanation facility
- Enhanced generation and management of relevant design documents such as graphs and reports

INTERFACES WITH OTHER TOOLS

- Algorithm analysis tools such as Balsa-II and the Hewlett-Packard Software Performance Analyzer (SPA)
- Architecture analysis tools such as the Stanford computer architect's workbench
- Implementation tools such as actual silicon compilers

Figure 2-15. AKS extensions

down design process. At the application level, the user specifies constraints which candidate implementations must meet to be acceptable. Between application and implementation levels, the user specifies candidate algorithms and architectures along with examples of input data streams. Compile and simulation time analysis provide data for AKS. The silicon compiler, or in the case of this research CC, returns calculated physical parameters for candidate implementations which are then compared to constraints set at the application level.

III. M2 LANGUAGE TOOLS

All these must be built with due reference to durability, convenience, and beauty. Durability will be assured when the foundations are carried down to the solid ground and materials wisely and liberally selected; convenience, when the arrangement of the apartments is faultless and presents no hindrance to use, and when each class of building is assigned to its suitable and appropriate exposure; and beauty, when the appearance of the work is pleasing and in good taste, and when its members are in due proportion according to correct principles of symmetry.

Vitruvius

from book 1, chapter 3, section 2 of *The Ten Books on Architecture*

A. Cmpl2: An analytic compiler

Compilers can be divided into three categories depending on their purposes: developmental compilers, optimizing compilers, and analytic compilers. Developmental compilers are geared for use by programmers in developing software. Emphasis is placed on compilation speed and ease of debugging while the size and speed of generated code is a secondary consideration. When a program's source code has been fixed and the program is about to be released, the final compilation can be done with an optimizing compiler. Code size is minimized and speed is maximized using techniques which can be time consuming and produce code which is difficult to trace with a source-level debugger.

The focus of an analytic compiler is the analysis of the source program and object code produced during compilation. The resulting information can be used to suggest changes in the source program which will improve speed or size. In the case of the M2 project, the analytic compiler is used to guide the design and selection of an architecture to execute the program. The structure of Cmpl2 and the analysis it performs will be discussed in the following paragraphs.

1. Compiler structure

A block diagram of Cmpl2 is shown in Figure 3-1. With the exception of the lexical analyzer (scanner) which is written in Modula-2, Cmpl2 is written in Prolog. It is a recursive-descent compiler with backtracking and requires three passes to generate the file used by IL3Sim.

a. **Pass 1** In the first pass, the lexical analyzer reads and tokenizes the source file sfn.MOD and places the tokens in the file sfn.TOK. Each token has four fields: the number of the source file line in which the token appears, the position of the first character of the token in the source file, the class to which the token belongs (keyword, identifier, character string, integer, real), and the lexeme for the token.

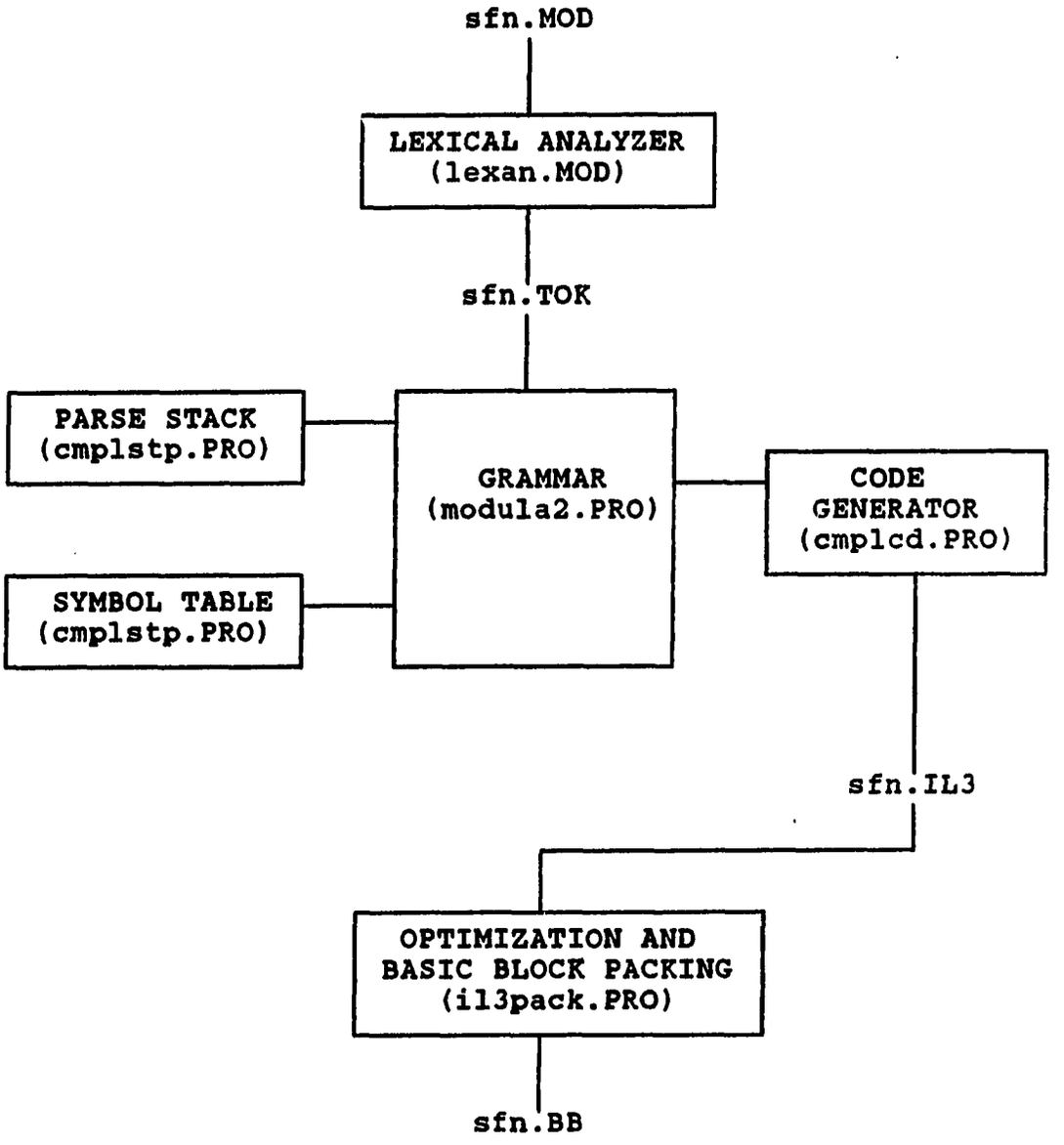


Figure 3-1. Cmpl2 and IL3Pack block diagram

b. Pass 2 In the second pass, Cmpl2 reads the file sfn.TOK and generates the intermediate language file sfn.IL3. Parsing is directed by a BNF-like representation of the Modula-2 grammar which contains terminal symbols, non-terminal symbols, the start symbol, and epsilon. It also contains semantic actions which direct identifier manipulation, type checking, and object code generation. The symbol table contains information about identifiers while the parse stack (pStack) is used to retain general information.

The activity of the token stream, parse stack, and code stream, can be observed on the video display as compilation progresses. The symbol table can also be examined during compilation. This "user view" provides a good environment for the incremental development of the compiler. Semantic actions can be added and tested with a delay of about a minute between the end of editing and the start of testing.

This short turn-around stands in stark contrast to the initial compiler development environment for the M2 project. Cmpl1 was designed as a nonrecursive predictive parser requiring a parse table to direct the compilation. A compiler generator, CG, was written in Prolog based on work by Cohen and Hickey [Cohen and Hickey 1987] as well as the standard reference on compilers by Aho, Sethi, and Ullman [1986]. It generated a parse table for a grammar

with 10 productions in a few seconds, but required 30 minutes for the 200 production Modula-2 grammar. This was deemed an undesirably long delay and consequently Cmpl2 came into existence. While Cmpl2 provides a good compiler development environment, its compilation speed is unacceptably slow for use in program development.

An effort has been made to make Cmpl2 semantic actions parameterless. When all the actions are parameterless and Cmpl2 is generally complete, modifying CG to generate a parse table from the Cmpl2 grammar and modifying Cmpl1 to support the Cmpl2 semantic actions should be a straightforward task. Cmpl1 will be the M2 compiler of choice for general use as it will be a smaller, faster, and equally functional version of Cmpl2.

An effort has also been made to define Cmpl2 semantic actions so they can be used with other imperative languages. Since the Modula-2 dependent syntax and parsing aspects of Cmpl2 are isolated in a single file, it should be fast and easy to adapt Cmpl2 for languages such as Pascal and C. The current (incomplete) Modula-2 version of Cmpl2 was developed in about 6 person-weeks. Developing a full Pascal version would take perhaps 4 person-weeks.

Programming in Modula-2, 2nd edition served as the basis for Cmpl2. Features in the standard which are not currently supported by Cmpl2 are shown in Figure 3-2.

GENERAL

Qualification of identifiers
Open array parameters for procedures
Short-circuit expression evaluation
Definition and implementation modules
Embedded modules

TYPES

Anonymous
Set
Record
Pointer
Procedure

STATEMENTS

Case
Exit
Return
With

Figure 3-2. Modula-2 features not currently implemented in Cmpl2

The completion of the compiler and bringing it into conformance with the third edition of *Programming in Modula-2* has been given a high priority as the M2 project continues.

c. Pass 3 In the third pass, IL3Pack reads the file sfn.IL3 and generates the files sfn.SAN, sfn.SBB, and sfn.BB. Il3Pack performs optimization and groups instructions into basic blocks for execution by IL3Sim. Optimizations include algebraic simplification, reduction in strength, use of IL3 idioms, and peephole optimization [Aho, Sethi, and Ullman 1986, pp. 554-557].

2. Compile-time analysis

The selection of data to be gathered at compile time and simulation time has been driven by the needs of CC. Since CC currently supports a template for only one architecture, IL3_4S1D, the data gathered is that which is needed to generate candidate IL3_4S1D implementations.

The compile time, or static, analysis is done after the optimizations of pass 3 are performed with two exceptions: the measurement of maximum frame size and the measurement of static memory. These data are gathered during pass 2.

The results of static analysis for sfn.MOD are stored in file sfn.SAN which contains records of the form

san (SANkey, Itemname, Value, Clog (Value))

in which SANkey is the name of the source file (sfn), Itemname is the name of the datum, Value is the numeric value of the datum, and Clog (Value) is the ceiling of the base 2 log of the Value field. A list of items stored in this format is shown in Figure 3-3 along with IL3_4S1D values synthesized from them.

Data for the basic blocks generated by IL3Pack are stored in the file sfn.SBB. The form for the records in the file is

sanBB (SANkey, BlkNum, Length, MaxDelay)

in which SANkey is the name of the source file (sfn), BlkNum is the number assigned to the basic block by IL3Pack, Length is the number of IL3 instructions in the block, and MaxDelay is the maximum delayed branch length which can be supported by the block without the insertion of null (nop) instructions.

B. Compiling Modula-2 to IL3

An important aspect of compilation is the effective mapping of high-level language constructs to the target architecture. A wide range of views on the subject exist and examination of them can fill a book [Milutinovic

NAME	DESCRIPTION
iMemLength_0D	Static program size, no branch delay
iMemLength_1D	Static program size, branch delay of one
iMemLength_2D	Static program size, branch delay of two
sMemLength	Number of words of static memory
maxFrameSize	Maximum procedure activation frame size
maxImmValue	Maximum immediate operand value

Figure 3-3. Data gathered at compile time (static analysis)

1986]. As mentioned in Chapter 1, M2 adopts the RISC approach for the standard reasons [Patterson and Ditzel 1980]. This section considers the relation of Modula-2 to the intermediate language IL3 and the IL3_4S1D architecture when used in embedded computer systems.

1. Storage allocation

Variables in a Modula-2 program can be declared statically within a module, locally when a procedure is activated, and dynamically through calls to procedures such as NEW and DISPOSE. Modula-2 also allows procedures to be treated as variables. Thus IL3 was designed to support four address spaces: static, local, dynamic, and instruction.

IL3_4S1D was designed with three physical memories: iMem for instructions, lMem for local variables, and sdMem for static and dynamic variables (refer to Figure 2-4). Programs for embedded computers are typically stored in ROM while data are stored in RAM. The instruction memory could be implemented as a single level of semiconductor ROM, but at present the fastest ROMs are slower than the fastest RAMs and processors. If this speed difference is a problem, the contents of ROM can be copied to RAM and executed from there, or else a cache can be placed between the ROM and the processor. This design decision is not currently supported by AKS. All

that is provided is a constraint on iMem access and cycle times imposed by the processor design.

Static data would be called global data in languages such as C and Pascal. In Modula-2, however, data that persist for the entire program are not visible outside of the module in which they are declared unless they are explicitly exported from the module. The enforcement of data hiding is done by the compiler.

Dynamic data are typically allocated from the heap and returned to it by calls to the operating system or run-time support library. A number of algorithms exist for these tasks, each with particular strengths and weaknesses [Aho, Hopcroft, and Ullman 1983]. An informed choice can be made after studying the allocation and deallocation call data gathered by IL3Sim.

In IL3_4S1D, the static and dynamic data share sdMem. As with iMem, AKS only constrains the cycle and access times of this memory. Local data are stored in lMem as described in the next subsection.

2. Procedure calls

The template for IL3 activation records is shown in Figure 3-4. Local data are accessed by their offset within the current activation record. Offset 0 is not used with the IL3_4S1D architecture and is reserved for future use. In another architecture, offset 0 could be

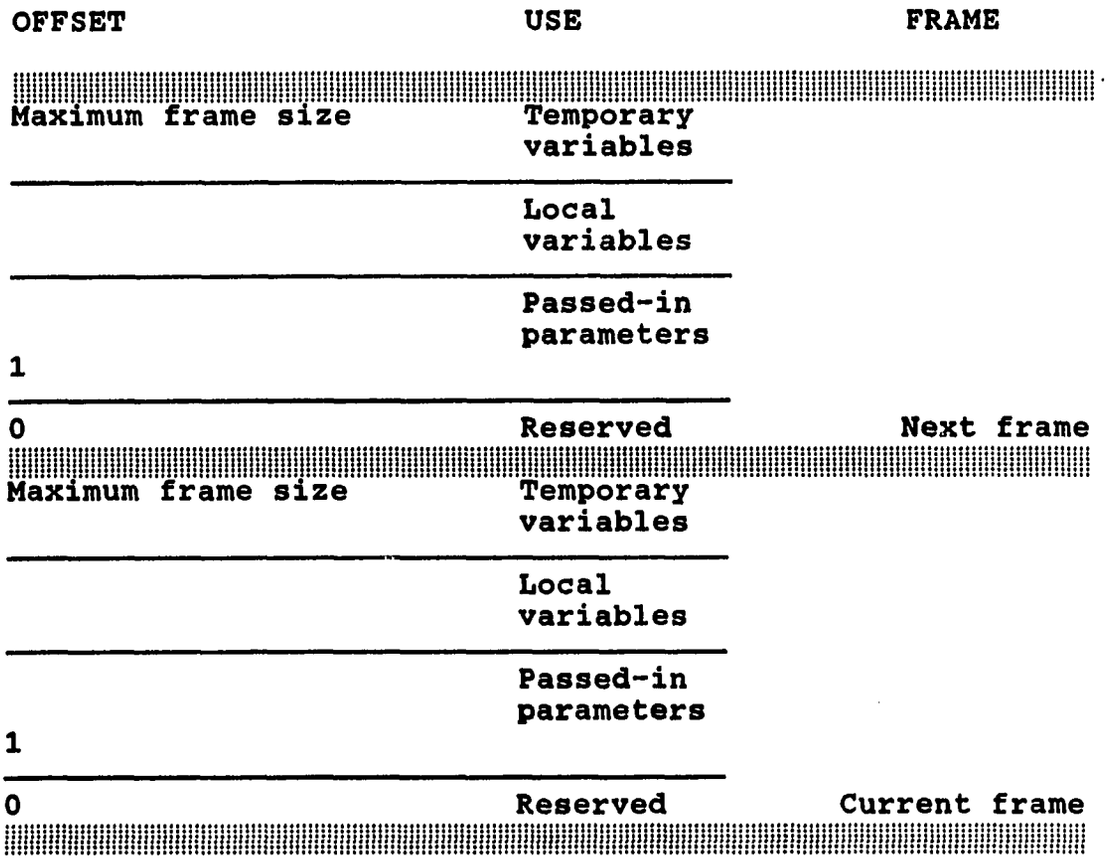


Figure 3-4. IL3 procedure activation template

used to save the instruction pointer at the time of a procedure call or could have a constant value of zero with benefits described by Katevenis [Katevenis 1985].

In IL3_4S1D, local memory (lMem) is organized as a register file with non-overlapping windows. Two windows are visible at any given time: the current window and the next window. Parameters are passed using a copy-restore method. Aho, Sethi, and Ullman [1986, p. 427] note a danger in the method if a variable local to the caller can be accessed in more than one way from the callee. The burden of this check is currently on the programmer and should be shifted to Cmpl2 which would flag the situation with a warning message.

Storage for local arrays and records is allocated in dynamic memory with a pointer to the structure stored in local memory. This keeps frame sizes small and allows all local items to occupy a single word. This approach may incur relatively significant time penalties in applications whose references to structured data exhibits a high degree of temporal locality.

The call sequence for procedures starts with the copying of parameter values into the parameter area of the next frame using parin instructions. In IL3_4S1D, a call instruction performs three simultaneous actions: (1) the current instruction pointer is pushed from register ip onto ipStack, (2) the new instruction pointer

value is loaded into register ip, and (3) the frame counter fc is incremented.

The return sequence for IL3_4S1D is carried out in three steps. First, the execution of a retc instruction causes the instruction pointer for the caller to be popped from ipStack back into the ip. Second, the values of any variable parameters are copied back to the caller's frame using parout instructions. Finally, the retf instruction removes the frame of the just completed procedure.

3. Processes

Processes are simulated in Modula-2 as quasi-concurrent co-routines with their own control thread and local data. When control is passed between co-routines using the Modula-2 procedure TRANSFER, a number of actions must be performed. First, the state of the current co-routine must be saved. Second, the state of the destination co-routine must be restored. Finally, control must be passed to the destination co-routine.

If the processor has a large register file, state saving and restoration can take a considerable amount of time. Thus a program dominated by process transfers is best executed on a processor with a small register file while one dominated by procedure calls is best executed on a processor with a large register file such as

IL3_4S1D. An alternative is to permit each process to have its own large register file which is selected by loading register with a process id.

A process id (pid) register could be added to the IL3_4S1D architecture and is already supported by the IL3 simulator IL3Sim. The detailed study of the relation between processes and architecture is well within the scope of M2, but has not been initiated at the time of this writing.

4. Interrupts

Like processes, interrupt handling is an important feature which is not yet supported by M2 tools. What follows is a brief description of how they may be treated for the IL3_4S1D architectures.

An interrupt is an unscheduled, parameterless procedure call. When an interrupt is detected, the current ip value is pushed onto ipStack while the address of a call instruction is used in fetching the next instruction from iMem. An interrupt service routine is then executed whose activation record is the next record in lMem, just as with a normal procedure call. When the service routine is completed, a reti instruction restores the previous state of the processor.

IL3 does not have a "test and set" instruction which can be used to manage program critical sections.

This is because conventional implementations require two indivisible cycles to execute (one to test a memory location and the other to set a value at the location) which violates the RISC principle of single-cycle instruction execution. What is proposed for IL3_4S1D is a set of "test and set" registers external to the processor and mapped into sdMem which automatically set when they are read. External system support in RISC architectures dates back to the IBM 801 [Radin 1982].

C. IL3Sim: An IL3 simulator

1. Simulator description

An IL3 program encapsulated into basic blocks is loaded from disk and is executed one instruction at a time. The user can observe simulation on the screen. At the end of the simulation, a file is generated which contains data from the simulation run.

The simulator supports four system functions: open, close, read, and write. These procedures act as interfaces to what would normally be services provided by an operating system or run-time library. The simulator does not assign any time cost to these functions, but does record the number of times they are called so a suitable implementation may be selected at a later time. Instructions executed by the caller to move parameters and call the function are counted.

At present, input data must be entered from the keyboard and output goes only to the screen. The open and close system functions have no effect on the simulation. Ideally, the open function would attach an input generator to an input channel which would be accessed with the read procedure. Such an approach is taken with Balsa-II [Brown 1988] and could be extended to include interrupt generators as well. At that point, the input file slot in the AKS architecture frame would be replaced by a script file containing a list of input and interrupt generators for a program.

2. Simulator analysis

The number of times each block is executed is saved in the .DBB file in records whose format is

danBB (DANkey, BlkNum, Count)

where DANkey is a key for the simulation, BlkNum is the block number, and Count is the number of times the block was executed.

Researchers at Stanford University have pointed out that retaining this information permits the execution times for multiple processors to be determined by running the simulation only once [Mitchell and Flynn 1988]. The number of instructions executed in the course of the

program is determined by summing the product of the static block size for a given branch delay for each block and multiplying the size by the number of times the block was executed. Total execution time for the program is the sum of the products of the time to execute each block and the number of times the block was executed.

The simulator also gathers data used by CC and saves it in the file sfn.DAN in the format

dan (DANkey, Itemname, Value, Clog (Value)

where DANkey is the key for the record and Itemname, Value, and Clog(Value) have the same meanings as for san records. Figure 3-5 shows the values gathered at simulation time and values which are derived from both static and dynamic analysis.

NAME	DESCRIPTION
iMemLength_0D	Dynamic program size, no delayed branch
iMemLength_1D	Dynamic program size, branch delay one
iMemLength_2D	Dynamic program size, branch delay two
maxdMemLength	Maximum allocated dynamic memory
maxlMemValue	Maximum value written to local memory
maxsdMemValue	Maximum value written to static or dynamic memory
maxDynamicLevel	Maximum number of procedure activation records

Figure 3-5. Data gathered at simulation time (dynamic analysis)

IV. EVALUATION OF M2

In all matters, but particularly in architecture, there are two points: - the thing signified, and that which gives significance. That which is signified is the subject of which we may be speaking; and that which gives significance is a demonstration on scientific principles. It appears, then, that one who professes himself an architect should be well versed in both directions.

Vitruvius

from book 1, chapter 1, section 3 of *The Ten Books on Architecture*

A. Introduction

How does one evaluate an architectural system, particularly when there are no other systems quite like it? The problem is further complicated by the fact that M2 is still under development and what is presented here is a snapshot of it as the parts are coming together for the first time (June 1988). One could wait for the system to be completed, but since M2 has been established as an open, evolving system there may never be a final completion.

The evolutionary nature of the system also limits the time span for which some measurements would be valid. Values for the code size and execution speed of the tools could be gathered and presented in this work, but in the near future a new version of the Prolog compiler used in this research is to be installed. The M2 tools as

currently implemented can then be recompiled without modification into smaller, faster executable files.

Computer systems are traditionally evaluated with benchmark programs. The results often have more promotional than technical value. To have technical value, a benchmark should meet at least the following requirements: (1) be meaningful, (2) be accurate, (3) be repeatable within a specified tolerance, and (4) be discriminatory of system differences [Nicholls 1988, p. 207].

For a benchmark to be meaningful, there should be a high degree of correspondence between reported measurements and the attributes the benchmark is attempting to illuminate. In reviewing Prolog programs and expert systems, attention is frequently given to the number of rules, the number of rule firings, and database size. While such values can be determined accurately and repeatably, their meaning and discriminatory usefulness is doubtful.

R1, a rule-based configurer of DEC computers, is similar in many ways to AKS and frequently appears in the literature. In 1980, the existing rule base size was cut in half while maintaining functionality through a restructuring of the rules. The size grew steadily for the next four years as the ability to configure models other than the VAX 11/780 was added. The rule base size

does not seem a good measure of system capability [Bachant and McDermott 1984].

Furthermore, it is not clear how general figures for number of rules and number of rule firings is useful in evaluating performance. Are a few "big" rules better than many "small" rules? Is a large table search better than a recursive calculation using two rules? Since Prolog searches for rule matches are linear, the ordering of the rules in the program can cause speed to range from n to rn where *n* is the number of times a search for a matching instance of a rule is initiated and *r* is the number of instances of the rule. The same speed range exists for database searches.

Given that there are no systems known to be comparable to M2, that some measureable values can reflect the state of M2 for only a short time, and that other measureable values do not have a clear interpretation, how is M2 to be evaluated?

The approach adopted here will be to show that M2 can perform the task set our for it in Chapter 1, viz., that it can analyze software statically at compile-time and dynamically through simulation and that the resulting information can be used by an expert system to determine which elements of a design space meet user specified constraints.

B. An Example of the Use of M2

The application to be considered is a simple one: a processor is to read eight values from a 10-bit analog-to-digital (A/D) converter, to sort the values, and then to report the average, median, and range of the values. Two Modula-2 programs, RS1 and RS2 (RS for Read Sensor), have the functionality needed for the task. IL3_4S1D is the only architecture to be considered and it may be implemented in either standard, low-power Schottky, or Schottky TTL. No constraints are set on space, time, power, cost, or reliability.

1. Static analysis

The Modula-2 source code for RS1 and RS2 is shown in Figures 4-1 and 4-2. The two programs are identical except for the procedure used to sort the array of sensor readings. RS1 uses a simple bubblesort algorithm while RS2 uses a straight insertion algorithm. Both algorithms are from Wirth [1986] who also provides typical and worst case analysis for them.

Normally the two programs would be compiled by Cmpl2 and IL3Pack. Cmpl2 does not support all the constructs in the programs, so portions of the .IL3 files had to be generated by hand to Cmpl2 specifications. The .IL3 files were then fed to IL3Pack which generated .BB and .SAN files. SAN values are shown in Figure 4-3.

```
MODULE RS1;  (* 21 May 88 *)

FROM M2System IMPORT sysOpen, sysClose, sysRead,
sysWrite;

CONST  BufferSize = 8;
TYPE   Buffer = ARRAY [1..BufferSize] OF CARDINAL;
VAR    B : Buffer;

PROCEDURE ReadBuffer (VAR B : Buffer);
VAR I : CARDINAL;
BEGIN
  FOR I := 1 TO BufferSize DO sysRead (1, B[I]); END;
END ReadBuffer;

PROCEDURE SortBuffer (VAR B : Buffer);
(* From _Algorithms_ by N. Wirth, pp 81-82 *)
VAR I, J, X : CARDINAL;
BEGIN
  FOR I := 2 TO BufferSize DO
    FOR J := BufferSize TO I BY -1 DO
      IF B[J-1] > B[J] THEN
        X       := B[J];
        B [J]   := B[J-1];
        B [J-1] := X;
      END;
    END;
  END;
END SortBuffer;

PROCEDURE ProcessBuffer (B : Buffer);
VAR I, Sum : CARDINAL;
BEGIN
  Sum := 0;
  FOR I := 1 TO BufferSize DO
    Sum := Sum + B [I];
  END;
  sysWrite (2, Sum DIV 8);
  I := (BufferSize + 1) DIV 2;   sysWrite (2, B [I]);
  sysWrite (2, B [BufferSize]);  sysWrite (2, B [1]);
END ProcessBuffer;

BEGIN
  sysOpen (1, "R"); sysOpen (2, "W");
  ReadBuffer (B); SortBuffer (B); ProcessBuffer (B);
  sysClose (1); sysClose (2);
END RS1.
```

Figure 4-1. Listing for program RS1

```
MODULE RS2;  (* 21 May 88 *)

FROM M2System IMPORT sysOpen, sysClose, sysRead,
sysWrite;

CONST  BufferSize = 8;
TYPE   Buffer = ARRAY [0..BufferSize] OF CARDINAL;
VAR    B : Buffer;

PROCEDURE ReadBuffer (VAR B : Buffer);
VAR I : CARDINAL;
BEGIN
  FOR I := 1 TO BufferSize DO
    sysRead (1, B[I]);
  END;
END ReadBuffer;

PROCEDURE SortBuffer (VAR B : Buffer);
(* StraightInsertion from _Algorithms_ by
   N. Wirth pp 77                          *)
VAR I, J, X : CARDINAL;
BEGIN
  FOR I := 2 TO BufferSize DO
    X := B[I]; B[0] := X;  J := I;
    WHILE (X < B [J-1]) DO
      B [J] := B [J-1];
      J := J - 1;
    END;
    B [J] := X;
  END;
END SortBuffer;

PROCEDURE ProcessBuffer (B : Buffer);
VAR I, Sum : CARDINAL;
BEGIN
  Sum := 0;
  FOR I := 1 TO BufferSize DO
    Sum := Sum + B [I];
  END;
  sysWrite (2, Sum DIV 8);
  I := (BufferSize + 1) DIV 2;   sysWrite (2, B [I]);
  sysWrite (2, B [BufferSize]);  sysWrite (2, B [1]);
END ProcessBuffer;

BEGIN
  sysOpen (1, "R"); sysOpen (2, "W");
  ReadBuffer (B); SortBuffer (B); ProcessBuffer (B);
  sysClose (1); sysClose (2);
END RS1.
```

Figure 4-2. Listing for program RS2

SAN NAME	RS1	RS2
iMemLength_0D	92	83
iMemLength_1D	100	90
iMemLength_2D	112	99
sMemLength	8	9
maxFrameSize	6	6
maxImmValue	87	87

Figure 4-3. SAN values for programs RS1 and RS2

2. Dynamic analysis

The execution time of the two sorting algorithms is dependent on the initial order of the values to be sorted and is worst when the readings are in reverse order. The largest value of the compution results from summing the readings in order to find their average. Thus the worst case input would be eight readings near the maximum value which are arranged in reverse order.

If it is assumed that the variation in sensor readings is due to noise and that the permutations of the values are equally likely, then a "mixed" set of readings can serve as a typical case. The order of the typical case data shown in Figure 4-4 was obtained by arbitrary selection of the values.

Each program was executed by IL3Sim twice, once with the worst-case data set, wst, and once with the typical data set, typ. The resulting DAN file values for each case are shown in Figure 4-5.

3. AKS analysis

AKS was initialized to the state shown in Figure 4-6 using the "Modify All" command. CC accessed the .SAN and .DAN files for the two programs and calculated space, time, power, cost, and reliability figures for the twelve implementation candidates. The values are shown in Figure 4-7. Had constraints been set and an

POSITION	INPUT VALUES	
	TYPICAL	WORST
1	1016	1023
2	1018	1022
3	1022	1021
4	1021	1020
5	1023	1019
6	1020	1018
7	1017	1017
8	1019	1016

Figure 4-4. Input data sets TYP and WST

DAN NAME	RS1/Typ	RS1/Wst	RS2/Typ	RS2/Wst
iMemLength_0D	625	805	373	523
iMemLength_1D	717	897	422	587
iMemLength_2D	846	1026	473	653
maxdMemLength	0	0	0	0
maxlMemValue	8156	8156	8156	8156
maxsdMemValue	1023	1023	1023	1023
maxDynamicLevel	3	3	3	3

Figure 4-5. DAN values for programs RS1 and RS2

```
APPLICATION SLOTS

     Name                          -
     Style                         Top-down
     Composition rule              None
     Space constraint              None
     Time constraint               None
     Power constraint              None
     Cost constraint               None
     Reliability constraint        None
     Algorithm names               RS1, RS2

ALGORITHM SLOTS

     Name                          -
     Language                      Modula-2
     Compiler front end            Cmpl2
     Compiler back end             IL3Pack

     Target ISA                    IL3_8805
     Compiler optimizations        -
     Execution model               -

     Target architecture           IL3_4S1D
     Input generation              Typ, Wst

ARCHITECTURE SLOTS

     Name                          -
     Implementation technologies   Std, LS, S

IMPLEMENTATION SLOTS

     Name                          -
     Space                         -
     Time                          -
     Power                         -
     Cost                          -
     Reliability                   -
```

Figure 4-6. AKS initialization

IMPLEMENTATION NAME	SPACE (sq in)	TIME (us)	POWER (W)	COST	REL[a]
\rs1\il3_4s1d\typ\std	34	109	2.9	$116	1
\rs1\il3_4s1d\typ\ls	34	115	1.1	$112	1
\rs1\il3_4s1d\typ\s	34	73	4.9	$160	1
\rs1\il3_4s1d\wst\std	34	136	2.9	$116	1
\rs1\il3_4s1d\wst\ls	34	144	1.1	$112	1
\rs1\il3_4s1d\wst\s	34	91	4.9	$160	1
\rs2\il3_4s1d\typ\std	34	64	2.9	$116	1
\rs2\il3_4s1d\typ\ls	34	67	1.1	$112	1
\rs2\il3_4s1d\typ\s	34	43	4.9	$160	1
\rs2\il3_4s1d\wst\std	34	89	2.9	$116	1
\rs2\il3_4s1d\wst\ls	34	94	1.1	$112	1
\rs2\il3_4s1d\wst\s	34	60	4.9	$160	1

[a]Failures per 1000 hours

Figure 4-7. Key attributes of implementation candidates

implementation failed to meet one of them, the implementation would have been highlighted on the video display when viewed with the "Examine Implementation" command.

All the candidates in this example have identical component sizes. Consequently, the size and reliability values for all the candidates are the same. Differences in speed, cost, and power consumption are due to the algorithms and fabrication technology. It is up to the user to determine which of those factors should dominate the selection of an implementation.

V. CONCLUSIONS

With regard to scorpiones, catapults, and ballistae, likewise with regard to tortoises and towers, I have set forth, as seemed to me especially appropriate, both by whom they were invented and in what manner they should be constructed. But I have not considered it as necessary to describe ladders, cranes, and other things, the principles of which are simpler, for the soldiers usually construct these by themselves, nor can these very machines be useful in all places nor in the same way, since fortifications differ from each other, and so also the bravery of nations. For seige works against bold and venturesome men should be constructed on one plan, on another against cautious men, and on still another against the cowardly.

Vitruvius

from book 10, chapter 16, section 1 of *The Ten Books on Architecture*

A. Research Contributions

This work has presented M2: a collection of software tools which can be used to determine the realizable points of a processor design space which meet user-specified constraints. A design is viewed as a model of an artifact which may or may not exist. The design style used is the top-down development of four models of the artifact: the application, the algorithm, the architecture, and the implementation. The implementation model is used to guide the manufacturing of an artifact while testing verifies that the artifact is a realization of the implementation model.

Top-down design is viewed as being non-monotonic. Design constraints and values can be altered at any stage of the design process, but results at lower design levels will be invalidated and will need to be redetermined. Changes at a given level will not affect higher levels. In M2, design validation is done by AKS, the Architectural Knowledge System. AKS is an extension of research done on a plausibility-driven approach to computer architecture design by Aguero and Dasgupta at the University of Southwestern Louisiana.

Current assumptions underlying M2 and their influence on the research were examined in chapter 1. The assumptions included embedded computer systems as the application area, Modula-2 as the language used to express algorithms, a 4-stage pipelined RISC as the target architecture, and a TTL implementation.

Chapter 2 presented CC, a CPU Compiler which is intended to be a silicon compiler surrogate. Silicon compilers stand to be a widespread design tool in the 1990s. They take architectural specifications and translate them into masks which can be used to fabricate integrated circuits. Silicon compilers are currently expensive both in terms of initial cost and operating cost. CC has been designed to demonstrate how a silicon compiler might interact with AKS in the design process

while using fewer monetary and computer resources than a true silicon compiler would.

Chapter 2 also discusses a computer architect's workbench being developed at Stanford University that takes programs written in Pascal, C, or FORTRAN and compiles them into an intermediate language called U-Code. Simulation of the U-code provides data useful in studying processor architecture features and memory structures. As presented in the literature, the Stanford workbench leaves data analysis to the researcher. AKS extends the workbench concept by using production rules to automatically analyze the data and provide parameters for candidate architectures which are evaluated by CC.

AKS also maintains the four design models for a project and guides their construction. The retention of models that have failed allows the designer to examine the implication of specified constraints on the design process.

Chapter 3 examined M2's language tools: CG, an LL(1) compiler generator; the analytic compilers Cmp11 and Cmp12; and IL3SIM, an intermediate language simulator. The chapter also considers the Modula-2 language and its consequences for compilers and computer architectures.

The M2 project is evaluated in Chapter 4. A discussion of possible approaches to its evaluation is

followed by a design example which shows M2 in use and the improvements in processor design which can be realized from its use.

It is believed that M2 is a novel synthesis of design theory, design tools, and artificial intelligence paradigms that can be used to facilitate the design process. Design can be done at the application and algorithmic levels by humans while software tends to the details of the architecture and implementation levels. The result is better designs in less time with lower cost than with manual methods. Computers can manage more design details with less error and, consequently, implementation speed, space, power, cost, and reliability can all be considered in detail.

B. Future Work

M2 currently exists as a bare-bones prototype. The integration of the tools is not always seamless and the user interfaces are minimal. Some refinements will need to be made as M2 moves from a research prototype to a tool for instruction and research in an academic environment.

Extensions can be made to all four design levels of M2. Applications can be extended beyond embedded systems and compilers can be created for additional languages. Knowledge bases are open-ended entities. Like human

knowledge, the knowledge in AKS can be refined in depth and expanded in breadth. Added depth will result in better designs while added breadth will increase the number of areas which will benefit from the use of M2.

Contemplated expansion of M2 on the architectural level includes the design of mixed digital and analog systems, determination of optimum pipelining for uniprocessors, comparison of multiprocessor and uniprocessor systems, and further study of RISC vs CISC architectures. On the implementation level, cache design could be included as well as technologies other than TTL.

VI. REFERENCES

Aguero, Ulises, and Dasgupta, Subrata. "A Plausibility-Driven Approach to Computer Architecture Design." *Communications of the ACM,* 30, No. 11 (1987): 922-932.

Aho, Alfred V., Hopcroft, John E., and Ullman, Jeffrey D. *Data Structures and Algorithms.* Reading, MA: Addison-Wesley, 1983.

Aho, Alfred V., Sethi, Ravi, and Ullman, Jeffrey D. *Compilers: Principles, Techniques, and Tools.* Reading, MA: Addison-Wesley, 1986.

Anderson, Noel. "Workload Partitioning in a Microcomputer Graphics System." Master's thesis, Iowa State University, 1985.

Bachant, Judith, and McDermott, John. "R1 Revisited: Four Years in the Trenches." *AI Magazine,* 5, No. 3 (1984): 21-32.

Bell, C. Gorden. "RISC: Back to the Future?" In *Reduced Instruction Set Computers,* William Stallings, ed. Washington, D.C.: The Computer Society Press, 1986.

Birkner, John Martin. "The Evolution of PALs." *BYTE,* 12, No. 1 (1987): 208.

Blakeslee, Thomas R. *Digital Design with Standard MSI and LSI. 2nd edition.* New York: John Wiley & Sons, 1979.

Borland International. *Turbo Pascal Owner's Handbook.* Scotts Valley, CA: Borland International, 1987.

Brayton, R. K., Camposano, R., De Micheli, G., Otten, R. H. J. M., and van Eijndhoven, J. "The Yorktown Silicon Compiler System". In *Silicon Compilation,* Daniel D. Gajski, ed. Reading, MA: Addison-Wesley, 1988.

Brooks, Frederick P., Jr. *The Mythical Man-Month.* Reading, MA: Addison-Wesley, 1975.

Brown, Marc H. "Exploring Algorithms Using Balsa-II." *IEEE Computer,* 21, No. 5 (1988): 14-36.

Bussey, John, and Sease, Douglas R. "Speeding Up: Manufacturers Strive To Slice Time Needed to Develop Products", *The Wall Street Journal*, 69, No. 91, Tue Feb 23, 1988, p 1, c 6 and p 18, c 1&2.

Cheng, E., and Mazor, S. "The Genesil Silicon Compiler." In *Silicon Compilation*, Daniel D. Gajski, ed. Reading, MA: Addison-Wesley, 1988.

Cohen, Jacques, and Hickey, Timothy J. "Parsing and Compiling Using Prolog." *ACM Transactions on Programming Languages and Systems*, 9, No. 2 (1987): 125-163.

Corbin, V., and Snapp, W. "Design Methodologies of the Concorde Silicon Compiler." In *Silicon Compilation*, Daniel D. Gajski, ed. Reading, MA: Addison-Wesley, 1988.

Cortes-Comerer, Nhora. "Motto for Specialists: Give Some, Get Some." *IEEE Spectrum*, 24, No. 5 (1987): 41-46.

Fikes, Richard, and Kehler, Tom. "The Role of Frame-Based Representation in Reasoning." *Communications of the ACM*, 28, No. 9 (1985): 904-920.

Filman, Robert E. "Reasoning with Worlds and Truth Maintenance in a Knowledge-Based Programming Environment." *Communications of the ACM*, 31, No. 4 (1988): 382-401.

Flynn, Michael J., Mitchell, Chad L., and Mulder, Johannes M. "And Now a Case for More Complex Instruction Sets." *IEEE Computer*, 20, No. 9 (1987): 71-83.

Ford, Gary A., and Wiener, Richard S. *Modula-2: A Software Development Approach*. New York: John Wiley & Sons, 1986.

Gajski, Daniel D., ed. *Silicon Compilation*. Reading, MA: Addison-Wesley, 1988.

Gajski, Daniel D., and Kuhn, Robert H. "New VLSI Tools." *IEEE Computer*, 16, No. 12 (1983): 11-14.

Gajski, D. and Thomas, D. "Introduction to Silicon Compilation." In *Silicon Compilation*, Daniel D. Gajski, ed. Reading, MA: Addison-Wesley, 1988.

Gimarc, Charles, E., and Milutinovic, Veljko M. "A Survey of RISC Processors and Computers of the Mid 1980's." *IEEE Computer,* **20**, No. 9 (1987): 59-69.

Heinbuch, Dennis, ed. CMOS3 Cell Library. Reading, MA: Addison-Wesley, 1988.

Hill, Mark et al. "Design Decisions in SPUR." *IEEE Computer,* **19**, No. 11 (1986): 8-22.

Hwang, Kai, and Briggs, Faye, A. *Computer Architecture and Parallel Processing.* New York: McGraw-Hill Book Company, 1984.

Johannsen, D. "BristleBlocks: A Silicon Compiler." *Proc. 16th Design Automation Conference,* 1979, 310-313.

Katevenis, Manolis G.H. *Reduced Instruction Set Architectures for VLSI.* Cambridge, MA: MIT Press, 1985.

Katz, Isadore. "Automating Electronic Design." *IEEE Spectrum,* **24**, No. 5 (1987): 55-58.

Kim, J. H. "Knowledge-Based System for IC Design." In *Silicon Compilation,* Daniel D. Gajski, ed. Reading, MA: Addison-Wesley, 1988.

Kowalski, Thaddeus J. "The VLSI Design Automation Assistant: An Architecture Compiler." In *Silicon Compilation,* Daniel D. Gajski, ed. Reading, MA.: Addison-Wesley, 1988.

Logitech, Inc. *Logitech Modula-2 Version 3.0 User's Manual.* Fremont, CA: Logitech, Inc., 1987.

Milutinovic, Veljko, ed. *Tutorial on Advanced Microprocessors and High-Level Language Computer Architecture.* Washington, D.C.: IEEE Computer Society Press, 1986.

Mitchell, Chad L., and Flynn, Michael J. "A Workbench for Computer Architects." *IEEE Design & Test of Computers,* **5**, No. 1 (February 1988): 19-29.

Nicholls, Bill. "That "B" Word!" *BYTE,* **13**, No. 6 (1988): 207-212.

Patterson, David A., and Ditzel, David R. "The Case for the Reduced Instruction Set Computer." *Computer Architecture News*, 8, No. 6 (1980): 25-33.

Patterson, David A. and Sequin, Carlo H. "A VLSI RISC." *IEEE Computer*, 15, No. 9 (1982): 8-21.

Pohm, A. V., and Agrawal, O. P. *High Speed Memory Systems*. Reston, VA: Reston Publishing, 1983.

Powell, Michael L. "A Portable Optimizing Compiler for Modula-2." *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction. SIGPLAN Notices*, 19, No. 6 (1984): 310-318.

Rabaey, J., De Man, H., Vanhoof, J., Goossens, G., and Catthoor, F. "CATHEDRAL-II: A Synthesis System for Multiprocessor DSP Systems." In *Silicon Compilation*, Daniel D. Gajski, ed. Reading, MA.: Addison-Wesley, 1988.

Radin, George. "The 801 Minicomputer." *SIGPLAN Notices*, 17, No. 4 (1982): 39-47.

Rasset, Terrence L., Niederland, Roger A., Lane, John H., and Geideman, William A. "A 32-bit RISC Implemented in Enhancement-Mode JFET GaAs." *IEEE Computer*, 19, No. 10 (1986): 60-68.

Soma, Mani. "A PC-Based VLSI Design and Test System for Education." *IEEE Transactions on Education*, 31, No. 1 (1988): 26-31.

Thomas, Donald E., Hitchcock III, Charles Y., Kowalski, Thaddeus J., Rajan, Jayanth V., and Walker, Robert. "Automatic Data Path Synthesis." *IEEE Computer*, 16, No. 12 (1983): 59-70.

Wiatrowski, Claude A., and Wiener, Richard S. *From C to Modula-2 and Back: Bridging the Language Gap*. New York: John Wiley & Sons, 1987.

Wirth, Niklaus. *Programming in Modula-2*. New York: Springer-Verlag, 1983.

Wirth, Niklaus. *Algorithms and Data Structures*. Englewood Cliffs, NJ: Prentice-Hall, 1986.

Wolfgram, Deborah D., Dear, Teresa J., and Galbraith, Craig S. *Expert Systems for the Technical Professional*. New York: John Wiley & Sons, 1987.

VII. ACKNOWLEDGEMENTS

I would like to acknowledge the contributions of the following to my research and also to my life. The listing is more or less in order of appearance.

God has given His providence and my parents have given their support throughout my life.

Kevin Powell and Joyce House have shared study and research breaks through yet another degree (the last for a while).

Dr. Terry Smay, my major professor, has overseen both my master's and doctoral work and has allowed me considerable freedom in pursuing both.

The members of my program of study committee, Dr. James Bieman, Dr. Paul Bond, Dr. Arthur Pohm, and Dr. Charles Wright, have reviewed this work and have participated in oral examinations. Dr. Chip Comstock reviewed this work and participated in the oral final examination.

Ricardo Salvador has enriched my Ph.D. studies through our dialogs on philosophy, theology, music, science, and life. May the world eat better and know more fully as a result of his continuing research on maize physiology.