COMPUTER ARCHITECTURE SIMULATION USING A REGISTER TRANSFER LANGUAGE

by

LESTER BARTEL

B. A., Tabor College, 1983

A MASTER'S THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1986

Approved by:

Major Professor

Table of Contents

List of Figures v

Acknowledgements vi

1 . Introduction 1

1 .1 . Purpose of an Architecture Simulator 1

1 .2 . Instruction Set Processor 3

1 .3 . Machine Cycle Simulator 3

1 .4 . Register Transfer Language 4

1 .5 . Silicon Compilers 6

1 .6 . Definitions 8

2 . Review of Existing CHDLs 10

2 .1 . Representative CHDLs 10

2 .1 .1 . CDL 10

2 .1 .2 . ISP 11

2 .1 .3 . AHPL 11

2 .1 .4 . DDL 13

2 .1 .5 . ADLIB 14

2 .1 .6 . DTMS 15
2 .1 .7 . CONLAN 17
2 .2 . Levels of Hardware Description 17
2 .2 .1 . Circuit Level 18
2 .2 .2 . Logic Gate Level 18
2 .2 .3 . Register Transfer Level 19
2 .2 .3 .1 . Structure Level 19
2 .2 .3 .2 . Functional Level 19
2 .2 .3 .3 . Behavior Level 20
2 .2 .4 . Instruction Set Level 20
2 .2 .5 . Processor Memory Switch Level 21
2 .2 .6 . Algorithmic Level 21
2 .3 . Applications of CHDLs 21
2 .3 .1 . Descriptive Tool 22
2 .3 .2 . Simulation and Design Verification 23
2 .3 .3 . Design Automation and Hardware Synthesis 24
3 . Introduction to ASIM II 25
3 .1 . Purpose of ASIM II 25
3 .2 . Description of Components 27
4 . ASIM II Operation 30

4 .1 . Primitives 30

4 .2 . Description of Primitives 31

4 .2 .1 . ALU 31

4 .2 .2 . Selector 32

4 .2 .3 . Memory 33

4 .3 . Implementation Notes 34

4 .4 . Optimization Notes 37

4 .5 . Input and Output 38

5 . Conclusion 39

5 .1 . Benefits 39

5 .2 . Execution Speed 39

5 .3 . Hardware Construction 41

5 .4 . Future Considerations 42

Bibliography 45

Appendix A ASIM II Documentation A1

Appendix B ASIM II Syntax B1

Appendix C ASIM II Source Code C1

Appendix D Example Stack Machine Simulator Specification

(Sieve of Eratosthenes) D1

Appendix E Pascal Code for Example Generated by ASIM II E1

Appendix F Specification and Hardware Circuit F1

List of Figures

Figure 3.1. Bit Concatenation..... 28

Figure 4.1. Alu specification and code generated by ASIM II..... 31

Figure 4.2. Selector specification and code generated by ASIM II..... 32

Figure 4.3. Memory specification and code generated by ASIM II..... 34

Figure 5.1. Execution time comparison of ASIM and ASIM II.....39

Acknowledgements

I wish to thank Dr. Thomas Pittman for his suggestion of this topic, and his patient assistance. His suggestions were invaluable in the preparation of this thesis.

Secondly, I wish to thank my major professor Dr. Virgil Wallentine, and my committee members Dr. David Gustafson and Dr. Maarten vanSwaay for serving on my committee, for their encouragement and input to this thesis.

I also wish to thank my friend Naomi Regier for proofreading this document and providing thoughtful suggestions to improve this thesis.

Finally, I wish to thank my parents, Albert and Lavina Bartel for their prayers and encouragement during my graduate studies. This work would not have been possible without their support.

Chapter One

Introduction

1.1. Purpose of an Architecture Simulator

As computer chips have grown in complexity, the need for tools to aid the engineer in designing and testing a proposed architecture has increased greatly. Many of these tools are available as computer programs, some of which are silicon compilers, RTL (Register Transfer Language), or ISP (Instruction Set Processor). A silicon compiler deals with the actual hardware mask generation and can be quite difficult to use. ISP defines each opcode as an instruction to be executed, and thus is a one to one replacement of the assembly code to a "high level language". RTL, which is classified as a CHDL (Computer Hardware Description Language), lies between silicon compilers and ISP in terms of the level of abstraction it deals with. This thesis will concentrate on examining and using an RTL.

Using a software simulator can greatly reduce the cost of designing a new microprocessor by avoiding physical construction. This physical construction is usually more error-prone and labor intensive than a

software system due to the need to actually build the circuit under consideration. Great care must be taken to ensure that wires are hooked up correctly and that the individual chips are working properly. Also, as complexity increases, the length and type of connections becomes critical and may cause problems which are not readily apparent. Some of these problems can be ignored during the simulation phase without serious consequence; they can be dealt with during the production of a prototype. Since the project can be simulated on an existing computer, building a prototype is not necessary until later in the design phase.

In the past three years, only brief mention is made of CHDLs in the literature, partially due to manufacturers developing their own CHDL and retaining it as a trade secret [50]. In light of this, it may be quite difficult for a single CHDL to become a standard for computer architecture design, implementation, and comparison. Some of the current CHDL implementations are relatively old and are more difficult to port over to a newer and faster computer. In implementing a new CHDL, it is desirable to promote porting to other computers and to promote standardization of hardware representation at the register transfer language level. It is desirable to provide a complete set of primitives with which a designer can define a circuit.

1.2. Instruction Set Processor

ISP translates each opcode of the test architecture to an expression in a high level language where the instruction execution is often implemented as a large case structure [67]. This level of translation does not lend itself well to actual circuitry generation, but is useful in designing an instruction set for the microprocessor to execute and for simulating that execution. After the instruction set has been generated and tested, it can be converted to an RTL for further testing. The RTL provides a circuit level definition of the microprocessor.

1.3. Machine Cycle Simulator

The machine cycle simulator, a technique especially useful in microprocessor design, is somewhat related to the ISP. In addition to the capabilities of the ISP, the machine cycle simulator traces the events at each cycle, thus providing timing information not available via an ISP. This type of simulator is quite close to an RTL simulator in the type of information provided by the simulation run. However, the machine cycle simulator specification is more difficult to convert to an actual circuit than the RTL specification because it still does not represent the hardware in detail.

1.4. Register Transfer Language

RTL machine simulation is the process whereby a program written in the assembly code of the machine to be simulated is read and executed by a different machine. The result is the same as executing the code on the architecture being simulated [67]. Other forms of RTL simulation employ a machine which has characteristics that are not physically present. This virtual machine may have more memory or a different instruction set, among other characteristics.

An RTL defines the instruction set of a microprocessor for simulation in terms of a small set of register transfer statements. If the RTL statements are carefully chosen, they will remain unchanged as instruction sets are defined for various microprocessors. An RTL is not limited to microprocessors, but can also be used for almost any other piece of digital electronic hardware. This extends its usefulness beyond computer architecture to the electrical engineering field.

The basic composition of an RTL for a microprocessor is as follows. The microprocessor is defined in terms of a set of register transfers which carry out the intended function in the same way the actual microprocessor would perform it. These register transfers are a mapping of the data flow through the microprocessor. A translator then reads these instructions to simulate the proposed architecture. The actual circuitry and

the register transfer code generate exactly the same final output. In addition the register transfer execution will typically produce statistics about the actual simulation, such as execution cycles required, memory accesses, and other related information. This extra output is invaluable when the designer desires to view the internal states of a microprocessor. With the RTL model, this is easily accomplished; however, with the hardware prototype, this can be difficult or impossible due to the need to attach additional hardware to monitor the various signals.

A model of a real (or proposed) microprocessor can be defined using an RTL. Tests may be run comparing different implementations of the same languages, or of various languages. Studies of this sort could then be applied to various compilers running on this microprocessor. Comparisons could be made on execution efficiency of the compiler. In this way, a compiler writer could ascertain the statements for which the compiler has difficulty generating efficient code, and possibly deal with these inefficiencies at the RTL level. These inefficiencies can be dealt with by changing the register transfer sequences within the microprocessor to make it more closely conform to the way compilers tend to generate code. More importantly, the architecture could be changed with no hardware modifications.

1.5. Silicon Compilers

In this study, an RTL compiler written in Pascal is presented that will read the RTL specification and produce a Pascal program. This resulting program will then simulate the target architecture and produce output at each cycle of the simulation. There exists an RTL interpreter with the same input specification. The compiler written in this study generates code which executes approximately an order of magnitude faster than the current simulator. The present interpreter is used in the classroom at Kansas State University, and is too slow for use in large projects.

Silicon compilers can be used to produce a mask used in the actual production of the individual chips (microprocessor, controllers, and other specialized chips), and thus find their way into the CAD/CAM category. However, the silicon compiler is not very well suited for the testing of the design functionality of microprocessor chips. Also, the silicon compilers available today are quite inefficient both in terms of ease of use and production efficiency (in terms of physical layout of the gates on the actual chip surface) of the final chip specification [47]. An RTL is needed in the early stages of development to ease the design of the chip that actually performs the desired function.

Whereas a silicon compiler is concerned with the actual connection and layout of the gates in a chip, an RTL avoids the gate level. Since the RTL is a higher level of abstraction, a correct design is easier to build with an RTL than with a silicon compiler. Occasionally gates must be simulated in a microprocessor description, for example by saving certain internal flags or operation results in flip-flops or to describe a specific interface from memory to an ALU (Arithmetic Logic Unit). After the RTL specification has been designed and rigorously tested, the design may then be converted to a language suitable for a silicon compiler which will produce the actual layout diagrams for production.

1.6. Definitions

In simulation languages, just as in programming languages, the terms "procedural", "nonprocedural", "serial", "parallel", and "concurrent" are very common. These terms are discussed below.

In a programming language, strict timing constraints seldom need to be enforced. In contrast, for a simulation language, timing is critical to the analysis of a design. Many times the designer must know the exact cycle in which a particular event occurs so that the information can be captured and analyzed properly. Each simulated unit has a "simulation time" associated with it. The simulation time (or clock cycle) carries with

it the values of different components (registers, gates, flip-flops, etc.) corresponding to that particular simulation time.

Depending on the sequencing structure and control structure of the underlying language, a language can be classified as procedural or nonprocedural. A set of statements is procedural if the statements are executed in the same order as they are specified. This is typical of most programming languages in use today. In a nonprocedural language, the order of execution is not necessarily the same order as the statements are specified.

The terms parallel and serial are commonly used in conventional programming languages as well as CHDLs. In CHDLs the term **parallel** means that the actions specified are to be executed simultaneously or in the same simulation time. The term **serial** means one after the other. Two sequencing mechanisms exist in programming languages. **Procedural** means that the statements are executed in the same order as specified. The term **nonprocedural** refers to the absence of a sequencing mechanism. **Concurrent** means that two (or more) processes can be executing at the same time, and that any communication between the processes must be explicitly specified. The term concurrent is not synonymous with the term parallel. If statements are executed in parallel, all are executed in the same simulation time. No concept of simulation time exists in concurrent

operations. Parallel blocks are synchronous, while concurrent blocks are asynchronous. In some CHDLs, a condition is associated with each statement or statement body. In these languages, all statement bodies with a true condition are executed in a nonprocedural and parallel fashion.

Chapter Two

Review of Existing CHDLs

2.1. Representative CHDLs

A number of CHDLs have been designed to describe different levels of hardware. Each has been developed for a specific use at a certain level of abstraction. Some CHDLs lie on the border of two abstraction levels and use features from both levels, but may not be able to fully implement both levels of abstraction. The following section presents a brief review of several existing CHDLs. The levels of abstraction are described later in this chapter.

2.1.1. CDL

CDL (Computer Design Language) is an Algol-like hardware description language developed by Yaohan Chu at the University of Maryland. CDL describes the structural and functional parts of a digital system [50]. The structural components (memory, registers, clocks, and switches) are declared explicitly at the beginning of the description. The functional behavior of each element is described using operators. The

system can be described at only one level of abstraction. There is no subroutine facility in CDL, thus making it unsuitable for modular description of a system. However, its simple structure and portability (implemented in Fortran) have made CDL a popular language. The language makes use of Fortran's operators as well as user defined operators. CDL can be used to describe complex digital systems. CDL was implemented in two parts: a translator and a simulator. The translator performs a syntax check of the description and translates it into a set of tables and a polish string program. The simulator executes the output of the translator and can accept simulation parameters through the commands: LOAD, OUTPUT, SWITCH, RESET, and SIMULATE. CDL does have some drawbacks. It does not easily lend itself to hardware generation. Also, because of the nonmodular description feature of CDL and the difficulty in using the polish string output of the translator to generate logic diagram level, it is unsuitable for a Computer-Aided Design And Test (CADAT) system.

2.1.2. ISP

ISP (Instruction Set Processor), as designed by Bell and Newell, was initially intended to be used only for documentation purposes [10, 24]. Now, however, it is being used for design automation, software

generation, program verification, and architecture evaluation. ISPS (Instruction Set Processor Specifications) is a computer language based on ISP for which a compiler and simulator have been produced. ISPS is a procedural language to describe instruction sets. Since it simulates an architecture at the instruction set level, ISPS does not provide any data concerning concurrency, timing, or interconnection of processors. A user can arbitrarily stop, start, and count events during the execution of the simulator, or examine and modify the contents of registers. By comparing the results of the simulation with expected results, the user can detect errors in the design of the system. ISPS is best suited to provide performance information for a hypothetical architecture.

2.1.3. AHPL

AHPL (A Hardware Programming Language) is a procedural language developed by F. J. Hill at the University of Arizona. Like CDL, AHPL is popular and well documented [37]. AHPL has applications in three areas: documentation, design verification and automatic design. It is supported by a simulator which provides design verification. A hardware compiler provides automatic design by translating AHPL descriptions to wiring lists specifying the interconnections of gates. The principle weakness in AHPL is its difficulty to express parallelism.

2.1.4. DDL

DDL (Digital system Design Language) is a complex and powerful block oriented nonprocedural language [25]. It is based on finite state machines and is designed to describe digital systems at the boolean equation, register transfer and algorithmic levels. Many of its features are similar to CDL, but in addition it allows the design to be specified in a block oriented fashion. This allows the designer to construct a portion of the system, test it, and use that block as a module which is already debugged. These blocks are expanded to their boolean equation equivalents for actual simulation [50]. By incorporating this capability, DDL has lost some of its flexibility as a simulator. DDL is well supported by software. A translator and simulator were implemented in IFTRAN on a Harris 6024 machine. This translator converts DDL descriptions to a set of boolean equations and register transfer expressions which can be used for hardware compilation. DDL is so well documented that it is used as an instructional example in two text books [15, 23].

2.1.5. ADLIB

ADLIB (A Design Language for Indicating Behavior) is a part of the SABLE (Structure And Behavior Linking Environment) simulation and design automation system developed at Stanford University [64].

ADLIB was developed to describe the computer component types. The interconnection of the component types is specified in a separate language, SDL (Structure Design Language). This ADLIB/SABLE/SDL system is designed to describe digital systems at different levels of abstraction. This is an important feature which allows the designer to describe a subsystem, and use that description in several other places without having to replicate the description. This saves time by reducing the amount of work necessary in the description, as well as reducing the number of potential design errors. The multi-level description and simulation approach also aids in validating a lower level design. For example, a system can be described at the behavior level and also at the structure level. Both simulation results can then be compared to assure the designer of similar descriptions.

ADLIB is a superset of Pascal and thus is strongly typed like Pascal. It extends Pascal with constructs to specify synchronous and asynchronous timing, extendable data types, subprocesses and intercomponent signaling. It is well supported in software.

The strength of ADLIB is also its weakness. ADLIB works well at the behavior level, but does not work well at the structure level. Structure level simulation is an important aspect of CHDLs which are used to design a circuit for the specification.

2.1.6. DTMS

DTMS (Descriptive Techniques for Modules and Systems) was developed at Kansas State University to better describe digital systems consisting of the interconnection of complex MSI (Medium Scale Integration) and LSI (Large Scale Integration) modules [55]. Descriptions in DTMS reflect the modular hardware of modern digital systems. Module functions are described at a high behavior level, and their interconnections are specified at the structural level. These modules communicate to each other through busses. DTMS allows both procedural and nonprocedural constructs to be expressed through PROCESS and NONPROCESS sections. It is not implemented in software; instead it is intended as a documentation tool.

2.1.7. CONLAN

The development of CONLAN (CONsensus LANguage) goes back to the first Symposium on Hardware Descriptions Languages (HDL) at Rutgers University [43, 44, 45]. The lack of an industry acceptance of the dozens of HDLs then in existence prompted a team of people from the United States and Europe to develop a language in which HDLs could be standardized. There are several reasons why acceptance of existing HDLs is so low:

1. None of the languages alone is sufficient to describe all aspects of a system and cover all phases of the design process.
2. Languages of different scope are syntactically and semantically unrelated.
3. Few of the languages are formally defined.
4. Only a few languages are implemented.
5. Descriptions are represented by character strings rather than graphically.
6. There exists no complete hardware design methodology to tell how to use HDLs effectively.

The main emphasis of CONLAN is to address the first four deficiencies. CONLAN itself is not an HDL, but its primary objectives are to 1) provide a common formal syntactic and semantic base for all levels of hardware description, 2) provide a means for the derivation of user languages from this common base, and 3) support CAD tools for documentation, certification, synthesis, and so on. Although many concepts of existing CHDLs appear in CONLAN, it is not intended as a language standard, but instead is a formal system which allows designers to construct HDLs of their choice in a consistent and unambiguous way. In this way, the industry will be able to more efficiently use HDLs in the entire phase of hardware development.

2.2. Levels of Hardware Description

CHDLs have become vital in the design of microprocessors and related electronic equipment. As computers have become more complex, tools have been developed to aid the engineer in the development of the circuitry which makes up a computer system at different levels. Six levels of hardware description are widely recognized among hardware designers [50]. These are 1) circuit level, 2) logic gate level, 3) register transfer level, 4) instruction set level or programming level, 5) processor memory switch level, and 6) algorithmic level. A number of CHDLs have been developed for use at the various levels of abstraction of the design of a microprocessor.

2.2.1. Circuit Level

The components described at the circuit level are the transistors, diodes, resistors, etc. which make up the various gates of an electronic system. An electronic design is described as the interconnection of these components and is the lowest level of abstraction recognized among computer hardware designers. (There exist lower levels which are of interest to engineers who build and test these components, however, these levels are not usually used by the computer hardware developer.)

2.2.2. Logic Gate Level

The logic gate level is the description of a piece of hardware at the logic gate (AND, OR, NOT, etc.) level. Its primary purpose is to describe SSI (Small Scale Integration), MSI (Medium Scale Integration), and some LSI (Large Scale Integration) circuits. The behavior is given by a set of boolean equations, and timing is on the order of a gate delay. With the advent of VLSI (Very Large Scale Integration), the computer design engineer will have less use for this type of representation, and will need a system that begins to model the behavior of the circuit instead of the structure.

2.2.3. Register Transfer Level

The register transfer level is where most CHDLs are implemented. At this level, registers are the basic component of the system. These languages model the transfer of data between registers, and logical and arithmetic expressions. Some examples of CHDLs at this level are AHPL, DDL, and CDL. Most CHDLs operate at the register transfer level. This level has been broken down into three sub-levels [9]. These are 1) structure level, 2) function level, and 3) behavior level.

2.2.3.1. Structure Level

Structure level description consists of describing the system using actual hardware components. Operators have physical counterparts. The structure level partially corresponds to the logic gate level. The structure level can describe hardware at the logic gate level, but generally only does so when necessary. It typically uses components at a higher level of abstraction.

2.2.3.2. Functional Level

Descriptions at the functional level consist of using actual hardware components and their functional relationship rather than the connections between them. Unlike the structure level, operators do not have physical counterparts.

2.2.3.3. Behavior Level

The behavior level describes the external behavior of the system. Its purpose is to describe the algorithm used by the hardware in terms of its input and output functions. This level is similar to some of the higher levels of abstraction.

2.2.4. Instruction Set Level

ISP is an example of a CHDL at the instruction set level. At this level, the instruction set of the microprocessor is simulated. The bits that make up the program being simulated are interpreted by a specific set of rules, and results are produced based on these rules. This is a functional description of a microprocessor whereas the previous levels describe the structure of the underlying hardware. Using the instruction set level, an engineer can first develop a workable instruction set (behavior level), and then concentrate on the hardware (structure) level.

2.2.5. Processor Memory Switch Level

The Processor Memory Switch (PMS) level describes the system in terms of processing units, memory components, peripherals, and switching networks. Only the major properties of the system are defined at this level. These properties include costs, memory capacities, system peripherals, and information flow rates. One use for this level is to determine the cost effectiveness of a particular design for a microprocessor. Another use is its application as a formal feasibility study tool for the engineer.

2.2.6. Algorithmic Level

The highest level of abstraction is the algorithmic level. At this level, only the algorithm executed by the hardware is important. This is an important concept, especially considering the complexity of circuitry in modern microprocessors. The engineer can build a module using a lower level of abstraction. When the module is completed, it can be expressed as an algorithm in order to reduce the complexity of the simulator specification.

2.3. Applications of CHDLs

Most of the work in the area of CHDLs deals with the functional and behavior level of the register transfer language level. Several simulators exist on a variety of computers which operate at the functional and behavior level. The operations which define the hardware in some of these CHDLs are quite complex both in implementation and in the way they are used by the hardware engineer.

CHDLs have important applications in the computer aided design, documentation, and design automation systems. The applications fall into three main categories. These are 1) descriptive tool, 2) simulation and design verification, and 3) design automation and hardware synthesis [63]. A discussion of each application area is presented next.

2.3.1. Descriptive Tool

In any complex system development cycle, a product needs to be described before producing it, and while building and testing it. Many CHDLs are intended as a descriptive tool. Later some of these have been modified for use with the other two application areas. After describing the product, the description must be communicated to other members of the design team and documented. The efficiency of the design team's efforts can be increased with the use of a standard tool. Desirable features include a consistency check of the documentation, as well as an automated documentation revision system. As these tools become more sophisticated, they will generate additional useful information.

2.3.2. Simulation and Design Verification

Another role of a CHDL is simulation and verification. As a system is being developed, its actions can be simulated with a CHDL simulator. The simulator output can then be checked with the expected results. Since no actual hardware has been built in this phase, it is relatively easy to make a change to the design, to correct a problem or experiment with a different design. Simulating a design can also assist the engineer in verification of the design. As designs become more complex, formal proof that it will work with all possible sets of input data becomes more difficult. With the

help of a CHDL, the accuracy of a design can be tested. One way to do this is by fault injection, the process of inserting a fault in the specification to cause errors (by design) in the simulation run. Thus if a catastrophic failure occurs on a certain type of fault, additional design work is necessary to reduce the impact of the fault or reduce the chances of that kind of fault occurring.

2.3.3. Design Automation and Hardware Synthesis

Additionally, the design process and hardware synthesis needs to be automated to increase designer productivity. With a CHDL, design automation is a realistic goal. As well as providing a standard method of describing a particular system, the CHDL can be designed to produce a specification for physical construction. In many cases the hardware generated by these systems is not optimal, and may need hand optimization. However, as CHDLs become more sophisticated they will produce better specifications for the physical construction.

Chapter Three

Introduction to ASIM II

3.1. Purpose of ASIM II

Although the existing CHDLs have a large command set, a small command set which provides a maximum of functionality is most desirable. Thus it is the purpose of ASIM II (Architecture SIMulator II) to realize this goal. ASIM II (and its predecessor ASIM) has a very small command set, namely ALU, selector, and memory operations. With these three primitives, it is possible to represent nearly any hardware device. The primary advantage of the small command set is the ease of remembering the different commands. If the entire command set is easily learned by the hardware designer, a specification can be easily written without the need to consult the users manual for the description of the semantics and syntax of an exotic command utilized in certain designs. This ease of use enables the designer to use ASIM II in place of a larger language.

When writing a simulator it is important to consider that simulation at this level is very likely to take considerably more execution time than the

actual architecture would require. The machine doing the simulation as well as the machine being simulated will, of course, make a great deal of difference. Since most work will likely involve simulating relatively new architectures on an existing machine, simulation is expected to be quite slow in comparison to what the actual execution speed of the simulated device would be.

The major benefit of a RTL is to allow the circuit designer to produce a circuit using a software system to evaluate the performance characteristics of the design. Another benefit is in the classroom situation where a student will be able to use the simulator instead of or in addition to the actual hardware lab. ASIM II is motivated by the need for a simulator which would run at an acceptable speed for a significant specification. The existing simulator, ASIM, provides an acceptable degree of usefulness, but its simulation time is too slow to simulate a usable microprocessor specification. ASIM was written in Pascal by Dr. Thomas Pittman on the Macintosh computer. It was ported to a Harris computer and later to the Vax 11/780. ASIM reads the specification into tables, and produces a simulation run by interpreting the symbols in the table. ASIM II, on the other hand, produces Pascal code from the specification which is then compiled by a standard Pascal compiler and executed. The execution time

of ASIM II is less than that of ASIM by approximately an order of magnitude. This reduction is offset in part by a longer compile time.

3.2. Description of Components

The three functional units in ASIM II are 1) ALU (Arithmetic and Logic Unit), 2) Selector, and 3) Memory. These three components are sufficient to describe many different hardware projects ranging from a simple counter to a stack machine and beyond. A description of each of the components follows. See Appendix A for the formal documentation of the simulator, and Appendix B for the syntax diagrams which define the language.

The format for an ALU is:

```
A name function left right
```

where `name` is the name of the ALU, `function` is an expression which determines the operation to be performed on `left` and `right`, the two operands.

The format for a Selector is:

```
S name selector value0 value1 value2 ... valuen
```

where `name` is the name of the Selector, and the value of `selector` forms an index to the appropriate value in the value list.

The format for a Memory is:

```
M name address data operation number [initial
  values]
```

where `name` is the name of the memory, `address` is the address (0 based) of the memory, `data` is the expression which gets stored in the memory (for a write operation), `operation` is the operation which is performed on the memory, `number` is the number of memory cells, and `initial values` is an optional list of values from which the memory gets its initialization data.

Most fields in the components may contain a complex expression (see Appendix B for a complete description of an expression and where the expression can occur). This expression can be composed of the concatenation of several components, and numeric constants. Thus `mem.3.4,#01,count.1` means to concatenate the fourth and fifth bits (bit positions are zero based) of `mem` with the binary string `01` and the second bit of `count` giving the result shown in Figure 3.1.

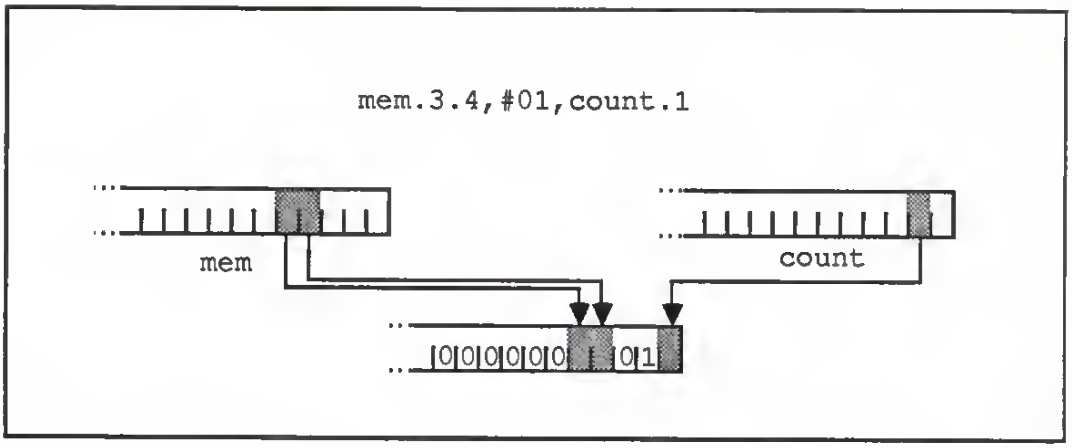


Figure 3.1 Bit Concatenation

Chapter Four

ASIM II Operation

4.1. Primitives

ASIM II has a small instruction set with which it is possible to express nearly any piece of hardware. It lacks no primitive which is needed to form a more complete command set to describe digital electronic equipment. The primitives ALU, selector, and memory have been used to describe a small stack machine which is able to execute a set of stack operators. The popular Sieve of Eratosthenes (a prime number generator implemented with a standard algorithm to assure similar test conditions among the various machines being benchmarked) has been implemented as a series of stack commands and is simulated using this simulator specification. The stack machine implementation of the Sieve of Eratosthenes is shown in Appendix D.

4.2. Description of Primitives

Each component in the specification can be replaced with a hardware component when constructing the prototype of the specification.

These hardware components can be purchased and easily connected together in the way described by the device description. A description of each primitive follows.

4.2.1. ALU

The ALU primitive is a software representation of a hardware ALU. In some cases, the ALU describes gates (NAND, AND, OR, NOT, etc.) and in other cases an actual arithmetic unit capable of addition, subtraction, multiplication, division, and comparison. Each ALU has three inputs (see Section 3.2). The function input tells the ALU which operation to perform. If the operation is a constant, the ALU may be implemented as a series of gates which perform that one function; otherwise, an actual ALU may be used, with the function bits determining its actual function. The left and right operands form the data inputs to the ALU, while the name contains the output of the ALU for use as input to another component. See Figure 4.1 for an example of an ALU specification and the code which ASIM II generates to simulate that specification. The ALU named "alu" shows the generic code generated for an ALU. The ALU named "add" shows the optimized code for a function whose value is a constant and whose function is to add the left and right operands. The

Specification:

```
A alu compute left 3048
A add 4 left 3048
```

Code generated:

```
alu := dologic(compute, left, 3048) ;
add := left + 3048;
```

Figure 4.1 ALU Specification and Code Generated by ASIM II

function `dologic` is shown in Appendix E with the code generated for the stack machine.

4.2.2. Selector

A selector is usually implemented as a data selector/multiplexor when the description is used to construct a hardware circuit. The selector input selects the input (value0, value1, ..., valuen) which is connected to the output. Again, the name of the selector is used to hold the output value for use as input to another component. See Figure 4.2 for an example of the code generated by ASIM II.

Specification:

```
S selector index value0 value1 value2 value3
```

Code generated:

```
case index of
  0: selector := value0;
  1: selector := value1;
  2: selector := value2;
  3: selector := value3
end;
```

Figure 4.2 Selector Specification and Code Generated by ASIM II

4.2.3. Memory

A memory is a much more complicated device. The hardware representation can be implemented in a variety of ways depending on the actual use of the memory, and the number of cells in that memory. If the memory is a single location, it will typically be represented as a flip-flop, or a set of flip-flops to hold several bits, i.e. a register. If the memory is composed of several cells, it may be represented as a ROM or RAM, depending on the type of operations that may be performed on it. Automatic logic generation of a circuit from the specification can be quite difficult if the designer wishes to have the circuit optimized to a reasonable

extent. However, the primary goal of ASIM II is to simulate a design in preparation of building a prototype. See Figure 4.3 for an example of the code generated from an ASIM II specification of a memory. (See next section for a description of the various temporary variables used.)

4.3. Implementation Notes

An ALU is implemented as a procedure call which accepts as inputs the function, left, and right operands. A case statement then decodes the function, and computes the result of the function applied to the left and right operands.

A selector, consisting of a case statement with the selector operand, provides the index to the list of cases. If the value of the selector exceeds the number of cases, a runtime error will result. It is up to the programmer to ensure that there are enough cases for the range of selector values.

Memories are implemented as a zero-based array of integers. If the number of memory locations is specified as negative, the memory is initialized with the initializer list. A memory in ASIM II, just like real hardware, has a delay from when it is accessed to when it actually provides the result of that access. This delay is one cycle.

Specification:

```
M memory address data operation -4 12 34 56 78
```

Code generated in initialization procedure:

```
memory[0]  := 12;
memory[1]  := 34;
memory[2]  := 56;
memory[3]  := 78;
```

Code generated in main program:

Compute new value and handle input and output

```
case land(operation, 3) of
  0: tempmemory := memory[address];
  1: begin
       tempmemory := data;
       memory[address] := data
     end;
  2: tempmemory := sinput(address);
  3: begin
       tempmemory := data;
       soutput(address, data);
     end
end; {case}
```

Figure 4.3 Memory Specification and Code Generated by ASIM II

Trace writes

```
if land(operation, 5) = 5 then
  writeln('  Write to memory at ', address:1, ' : ',
          tempmemory:1);
```

Trace reads

```
if land(operation, 9) = 8 then
  writeln('  Read from memory at ', address:1, ' : ',
          tempmemory:1);
```

Figure 4.3 (continued)

To eliminate the need for actual parallel processing of the components, the components are sorted in a dependency order. If the value of a selector requires the result of an ALU, the ALU's value is computed first, etc. Memories are not sorted. Instead, their results are stored in temporary memories (similar to the memory buffer register in actual hardware) while the new value is being computed.

The values of the components (if traced) are printed after their new values have been computed. In the case of memories, the value used in the computation is printed before it is updated with the new value it may have received during that cycle.

4.4. Optimization Notes

In implementing ASIM II, an emphasis was placed on optimization of the code produced by the compiler in an effort to reduce execution time. This optimization has a goal of reducing the number of procedure calls in the program. The generic procedure takes the function of the the ALU, and returns the value based on the three inputs, namely function, left, and right. If the function is a constant, code is generated which performs the function inline, rather than call the procedure. Similarly, if the memory operation is a constant, the case structure is eliminated and only the appropriate action is performed on the memory.

4.5. Input and Output

Input and output is possible with ASIM II, and models actual hardware designs in use today by using memory mapped I/O. Memory mapped I/O treats the input and output as a special case of memory. The I/O is received from the standard input or sent to the standard output in this simulator. ASIM II utilizes a procedure for the I/O, thus making changes in the handling of I/O easy to implement. A memory in the specification can contain the input values for execution as well. For example, the stack machine description takes its input directly from the specification. A RAM in the description contains the code which is executed. Its output is sent to

the standard output and consists of the prime numbers generated by the simulator.

Chapter Five

Conclusion

5.1. Benefits

ASIM II is an important tool for the design of a digital electronic circuit. It provides all of the necessary primitives to express nearly any circuit in a form suitable for simulation. Since the design can be simulated on a software system, actual hardware prototypes need not be built until later in the design phase. Secondly, it is a documentation tool (at a low level) useful for generating hardware that performs the function described in the specification. A team of designers can use this specification format to convey various designs to one another in a standard way.

5.2. Execution speed

The simulation time of ASIM II has been reduced significantly over that of its predecessor ASIM (see Figure 5.1). The data in Figure 5.1 was taken from the compile and execution time of the stack machine example in appendices D and E.

ASIM	
Generate tables	10.8
Simulation time	310.6
ASIM II	
Generate code	34.2
Pascal Compile	43.2
Simulation time	15.0
Traditional Methods	
Generate Prototype	100000
Run Prototype	0.01

Figure 5.1 Execution time comparison (in seconds) of ASIM and ASIM II

Both ASIM and ASIM II executed the specification for 5545 cycles (the maximum number of cycles allowable in this specification of the stack machine). The best of 5 time trials was taken with 3 of the timings taken very early in the morning to reduce the effect of other user activity causing excessive variations. ASIM uses 10.8 seconds of cpu time to prepare the tables necessary for simulation. 310.6 seconds are used in the actual simulation . ASIM II used 34.2 seconds to generate the Pascal code from the specification, and 43.2 seconds to compile the Pascal code to the object code. The simulation used only 15.0 seconds. Thus if the simulation

preparation times are not considered, (the simulator is used more often than code is generated and compiled for it) ASIM II runs approximately 20 times faster than ASIM. Including the preparation times it is nearly 2.5 times faster. This reduction has come at the cost of increased preparation time for large specifications. This includes the Pascal compile time to translate the code generated by the simulator into machine code. In contrast to the specification compile time, hand wiring of a prototype would take several days. Execution of the prototype would be so fast (real time) that monitoring states in the circuit would be difficult. However, a prototype is a necessary phase of any hardware project.

5.3 Hardware Construction

A hardware circuit can be easily built from a hardware specification in ASIM II. Essentially, ASIM II is a list of hardware components with the wiring interconnection specified by the names of the components and their bit fields. If the bit field exists in the specification, then it is known that only the particular pins corresponding to that component are hooked up to to the named component. The specification is most like a block diagram of the circuit. The connections between components are not labeled with pin numbers nor actual component types. Enough information exists so that the engineer can choose appropriate

components which perform the function of the specified component. See Appendix F for an example of a hardware specification and circuit for a small 10 bit microprocessor with five instructions (load, store, branch, branch on borrow, and subtract) and 128 bytes of program and data memory. Note that each of the components in the specification has a hardware component represented in the diagram. It should be noted that this is not an optimum circuit, but rather a reflection of the ASIM II specification and demonstrates the ease of translating the specification to an actual hardware circuit. Actual hardware generation is beyond the scope of this thesis, however, it is appropriate to show a small example to demonstrate the value of ASIM II for larger projects.

5.4. Future Considerations

The ALUs, selectors, and memories provide a slightly higher level of abstraction than most other CHDLs provide. This makes hardware description somewhat more modular in that the lower level primitives such as gates are not expressed unless they are necessary in the description. Thus, a complete description is smaller in ASIM II than in most of the other CHDLs.

As with all software, there is always room for improvement. Further optimization of the code is possible using heuristics to determine

which memories do not need temporary variables in which to store results while the new values are computed for the memories.

Modularity is an important concept in today's programming languages. ASIM II, however, does not have any high level modularity construct. The behavior of an electronic circuit is difficult to express in a modular fashion without providing the the actual description of the module and expanding that description at compile time. As semiconductor prices continue to fall and microprocessors gain more features, modularity will become more important. The designer will be less concerned over the amount of silicon required for the circuit than the additional time needed to produce a specification with fewer gates (typically less modular due to the gate reduction techniques).

ASIM II has become a valuable tool for small digital hardware projects. With improved speed, and modularity, it could become even more useful for simulating hardware components at the behavior and structure levels.

Bibliography

1. *Computer Hardware Description Languages and their Applications.* IFIP 1983.

2. *Microprocessor Development and Development Systems.* 1982

3. *Microprocessor Systems: Software and Hardware Architecture.* 1984

4. Alexandridis, Nikitas A., *Microprocessor System Design Concepts.* 1984.

5. Baer, Jean-Loup, *Computer Systems Architecture.* Computer Science Press, Rockville, Maryland 1980.

6. Ballieu, G., Lewi, J., and Willems, Y. D., "A Microprogramming Language at the Register Transfer Level", Microprocessing and Microprogramming, Oct, Nov, Dec 1981. p179-188.

7. Bara, J. Born, "A CDL Compiler for Designing and Simulating Digital Systems at the Register Level". p96-102. Proceedings of: 1975 International Symposium on Computer Hardware Description Languages and their Applications. Sept 3-5, 1975. New York NY.

8. Barbacci, Mario. "An Architectural Research facility - ISP Descriptions, Simulation, Data Collection". pg 161-173. AFIPS 1977.

9. Barbacci, Mario R. "A Comparison of Register Transfer Languages for Describing Computers and Digital Systems", IEEE Transactions on Computers, V24 N2 (Feb 1975) p137-150.

10. Barbacci, M., "Instruction Set Processor Specifications (ISPS): The Notation and Its Applications", IEEE Transactions on Computers, Vol C-30, No. 1, January 1981.

11. Barbacci, M., "Instruction Set Processor Specifications for Simulation, Evaluation, and Synthesis". 64-72. Proceedings of: The Sixteenth Annual Conference on Design Automation. June 25-27, 1979. San Diego, Calif. Hightower, D. W. (Ed.)

12. Barbacci, Mario R., and Parker, Alan., "Using Emulation to Verify Formal Architecture Descriptions". Computer, V11 N5 (May 1978) p51-56.

13. Blasewitz, Robert M., *Microcomputer Systems: Hardware/Software Design.* 1982.

14. Borrine, D. "LASCAR: A Language for Simulation of Computer Architecture". p143-152. Proceedings of: 1975 International Symposium on Computer Hardware Description Languages and their Applications. Sept 3-5, 1975. New York NY.

15. Breuer, Melvin A., *Design Automation of Digital Systems.* Prentice Hall, Inc., Englewood Cliffs, N.J. 1972.

16. Burris, Harrison R. "Instrumented Architectural Level Emulation Technology". pg 937-946. AFIPS 1977.

17. Chang, S. J. "Some Applications of Hardware Description Languages in Real-time Digital System Development". p181-182. Proceedings of: 1975 International Symposium on Computer Hardware Description Languages and their Applications. Sept 3-5, 1975. New York NY.

18. Chiang, Mike, and Palkovic, Richard, "LCC Simulators Speed Development of Synchronous Hardware", Computer Design, March 1, 1986, p87-92.

19. Chu, Yaohan, "Why do we Need Computer Hardware Description Languages?". Computer, Chu, Yaohan, Dec 1974. p18-22.

20. Dasgupta, Subrata, "On the Verification of Computer Architectures Using an Architecture Description Language", Department of Computer Science, University of Southwestern Louisiana, Lafayette, Louisiana.

21. Dasgupta, Subrata, and Olafsson, Marius, "Towards a Family of Languages for The Design and Implementation of Machine Architectures", Department of Computing Science, University of Alberta, Edmonton, Alberta Canada.

22. Dennis, Jack B., "Computer Architecture and the Cost of Software", Computer Architecture News, Vol 5, No 1, April 1976, pg 17.

23 Dietmeyer, D.L., *Logic Design of Digital Systems*, 2nd. ed., Allyn and Bacon, 1978.

24. Djordjevic, J., Ibbett, R. N., and Barbacci, M. R., "Evaluation of Computer Architectures Using ISPS", Proceedings of the Institute of Electronics Engineering, Part E, 127 (4), 1980, p 126.

25. Duley, J.R., and Dietmeyer, D. L., "A Digital System Design Language (DDL)", IEEE Transactions on Computers, C-17 (9), 1968, p850.

26. Evangelisti, C. J., Goertzel, G., Ofek, H., "Designing with LCD: Language for Computer Design", IEEE 1977 Design Automation Conference, p 369.

27. Franta, W. R., and Giloi, W.K. "APL*DS: A Hardware Description Language for Design and Simulation". p45-52. Proceedings of: 1975 International Symposium on Computer Hardware Description Languages and their Applications. Sept 3-5, 1975. New York NY.

28. Fuller, Samuel H. "Evaluation of Computer Architectures Via Test Programs". pg 147-160. AFIPS 1977.

29. Gardner, R. I. Jr., Estrin, G., and Potash, H. "A Structural Modeling Language for Architecture of Computer Systems". p161-171. Proceedings of: 1975 International Symposium on Computer Hardware Description Languages and their Applications. Sept 3-5, 1975. New York NY.

30. Hafer, L., and Parker, A. C. "A Formal Method for the Specification, Analysis, and Design of Register Transfer Level Digital Logic". p846-853. ACM IEEE 18th Design Automation Conference Proceedings. Nashville, Tenn. June 29-July 1, 1981.

31. Hafer, L. J., and Parker, A. C., "Automated Synthesis of Digital Hardware", IEEE Transactions on Computers, Vol C-31, no 2, p93-109, Feb 1982.

32. Hafer, L. J., and Parker, A. C. "Register-Transfer Level Digital Design Automation: The Allocation Process". p213-219. Proceedings of: Fifteenth Annual Design Automation Conference. June 19-21, 1978. Las Vegas Nev.

33. Hahn, Winfried, "Computer Design Language - Version Munich (CDLM) A Modern Multi-level Language", 1983 Design Automation Conference, p 4.

34. Hakozaki, Katusya. "Design and Evaluation System for Computer Architecture". pg 81, AFIPS 1973.

35. Hartenstein, R. W., *Fundamentals of Structured Hardware Design: A Design Language Approach at Register Transfer Level.* Elsevier North-Holland, Inc., New York, 1977.

36. Hayes, John P. *Computer Architecture and Organization.* (McGraw-Hill 1978).

37. Hill, Fredrick J., and Peterson, Gerald R., *Digital Systems: Hardware Organization and Design.* New York, John Wiley & Sons, Inc. 1973.

38. Huen, W. H., and Siewiorek, D. P., "Intermodule protocol for Register Transfer Level Modules: Representation and Analytic Tools". p56-62. Proceedings of: Second Annual Symposium on Computer Architecture. Jan 2-22, 1975. Houston TX.

39. Kumar, K. S. DTMS II - "An Improved Computer Hardware Description Language for Modules and Systems", Phd Dissertation Kansas State University, 1982.

40. Ma, Perng-Yi. "The Design of a Firmware Engineering Tool: The Microcode Compiler." pg 87 AFIPS 1981.

41. Mano, M. Morris, *Digital Logic and Computer Design.* Prentice Hall, Inc., Englewood Cliffs, N.J. 1979.

42. Marczynski, R. W., Pulczyn, W. T., and Sochacki, J. M. "OSM - Microprogrammed Hardware Structure Description Language". p181-182. Proceedings of: 1975 International Symposium on Computer Hardware Description Languages and their Applications. Sept 3-5, 1975. New York NY.

43. Piloty, Robert. "CONLAN - A Formal Construction Method for Hardware Description Languages: Basic Principles". pg 209 AFIPS 1980.

44. Piloty, Robert. "CONLAN - A Formal Construction Method for Hardware Description Languages: Language Derivation". pg 219 AFIPS 1980.

45. Piloty, Robert. "CONLAN - A Formal Construction Method for Hardware Description Languages: Language Application". pg 229 AFIPS 1980.

46. Rauscher, Tomlinson Gene. "Developing Application Oriented Computer Architectures on General Purpose Microprogrammable Machines". pg 715-722. AFIPS 1976.

47. Schindler, Max, "Silicon Compilers Travel Rough Roads to Acceptance", Electronic Design, Vol 34, No 10, May 1, 1986.

48. Schindler, Max, "New Languages Help Create and Test Systems With no Need for Breadboards", Electronic Design, Vol 34, No 23, October 2, 1986, p90.

49. Schorr, H., "Computer Aided Digital System Design and Analysis Using a Register Transfer Language" IEEE Transactions Electronic Computers, vol EC-13, pp. 730-737, Dec. 1964.

50. Shiva, Sajjan G., "Computer Hardware Description Languages - A Tutorial", Proceedings of the IEEE, Vol. 67, No 12, December 1979.

51. Shiva, Sajjan G., "Combinational Logic Synthesis from an HDL Description", Proceedings of the 17th Design Automation Conference, 1980, p550-551.

52. Shiva, Sajjan G., Covington, J. A., "Modular Descriptoin/Simulation/Synthesis Using DDL", Proceedings of the 19th Design Automation Conference, 1982, p321-325.

53. Shiva, Sajjan G., "Use of DDL in an Automatic LSI Design and Test System", International Symposium on CHDLs, 1979, p28-31.

54. Shiva, Sajjan G., Patel, D. C., "Simulation Attributs of Computer Hardware Description Languages", The Radio and Electronic Engineer, Vol 54, No 1, January 1984, p45-50.

55. Singh, Anil Kumar, "Descriptive Techniques for Digital System Containing Complex Hardware Components", Phd Dissertation, Kansas State University, 1981.

56. Smith, William R., "AADC Computer Family Architecture Questions and Answers", Computer Architecture News, Vol 4, No 3, September 1975.

57. Su, Stephen Y. H., "A Survey of Computer Hardware Description Languages in the U.S.A.", Computer, December 1974, p45.

58. Su, Stephen Y. H., "Hardware Description Language Applications: An Introduction and Prognosis", Computer, June 1977, p10.

59. Su, Stgephen Y. H. *Computer Hardware Description Languages & Applications*. IEEE New York, NY. 1975.

60. Svobodova, L., *Computer Performance Measurement and Evaluation Methods: Analysis and Applications*, New York, Elsevier North-Holland, 1976.

61. Teng, Albert Y., *Experiments in Logic and Computer Design*. 1984

62. VanCleemput, W. M. IEEE New York, NY. 1979.

63. VanCleemput, W. M., "Computer Hardware Description Languages and their Applications", Proceedings of the 16th Design Automation Conference, June 1979, p554-560.

64. VanCleemput, W. M., "An Hierarchical Language for the Structural Description of Digital Systems", IEEE 1977 Design Automation Conference, p 377.

65. Villar, E., and Bracho, S., "Checking Sequences for the Control Unit in Digital Circuits Described by means of Register Transfer Languages". International Journal of Electronics, V59 N1 19-31.

66. Wakerly, John F., "Pascal Extensions for Describing Computer Instruction Sets", Computer Architecture News, Vol 8, No 7, December 15, 1980, pg 15.

67. Williams, M. H., and Stewart, A. J., "Machine Cycle Simulator for a Microprocessor". Microprocessors and Microsystems, April 1982, p131-134.

Appendix A

ASIM II Documentation

To invoke ASIM II, type `sim [file]`. If `file` is not specified, you will be prompted for an input file. `File` contains the specification to be compiled. After successful compilation, type `pc simulator.p` in order to generate executable code (a.out) from the specification.

File format:

The first line must be a comment line starting with the '#' character. This line is echoed to the code file as a comment. Any number of macros may follow. A macro begins with a '~' and is followed by a name, followed by a text string which will be substituted for the macro name in the definition of components. (No whitespace between '~' and name, and no whitespace in macro string). A macro may be placed anywhere in an expression as long as the macro string can legally replace the macro name. The macro name is entered in the component specification with a '~' immediately followed by the macro name, and may be part of any string. Any character except letters and numbers will delimit a macro name from

the rest of the string. A macro may contain a macro name, as long as that name has already been defined (cannot be circular or recursive).

The number of cycles the simulator is to run can, but need not be specified. The format is an '=' sign followed by a decimal integer. (Whitespace between '=' and integer). If the number of cycles is not specified, you will be asked how many cycles to execute at the beginning of the simulation. After those cycles have been executed, you will again be prompted for the cycle number to continue to.

A list of all component names follows with an '*' immediately following each name that is to be traced at each cycle. A period ends the list of names. The names may be listed in any order. Those followed by an '*', will be printed in the order listed. The component specification follows the list of names. Components consist of ALUs (A), Selectors (S), and Memories (M), and may appear in any order. The components will be sorted to resolve any dependencies. Circular dependencies will generate an error message providing a clue to the component(s) which are involved in the dependency. Following is a list of the components:

```
A name function left right
S name address value0 value1 ... valuen
M name address data operation number [value0 value1
  ... valuen]
```

It is up to the user to provide enough values for all possible address values in a selector. Otherwise a runtime error will result. If `number` < 0 then the memory is initialized with the values listed. The specification must end with a period.

Notes:

Fields must not contain whitespace. Comments may be placed anywhere in the file where whitespace is permitted. A comment starts with a '{' and ends with a '}'. Nested comments are not supported. A runtime error will occur if the number of sources for the selector is less than the value of the address, or if a memory address falls outside the declared range which starts at 0. If the number in memory is less than zero, then there must be exactly that many initial values provided. All components are initialized to zero before simulation begins (except memories with initial values listed).

I/O Memory operations take the memory data from standard input, or send the memory data to standard output, depending on the read/write bit. If the address (for both read and write) is 0, then the data is treated as character data, if the address is 1, the data is treated as an integer, otherwise the data is treated as an integer, and the address read or written is

printed. These input and output functions are handled as a procedure in ASIM II, and may be modified by the user if some other action is desired.

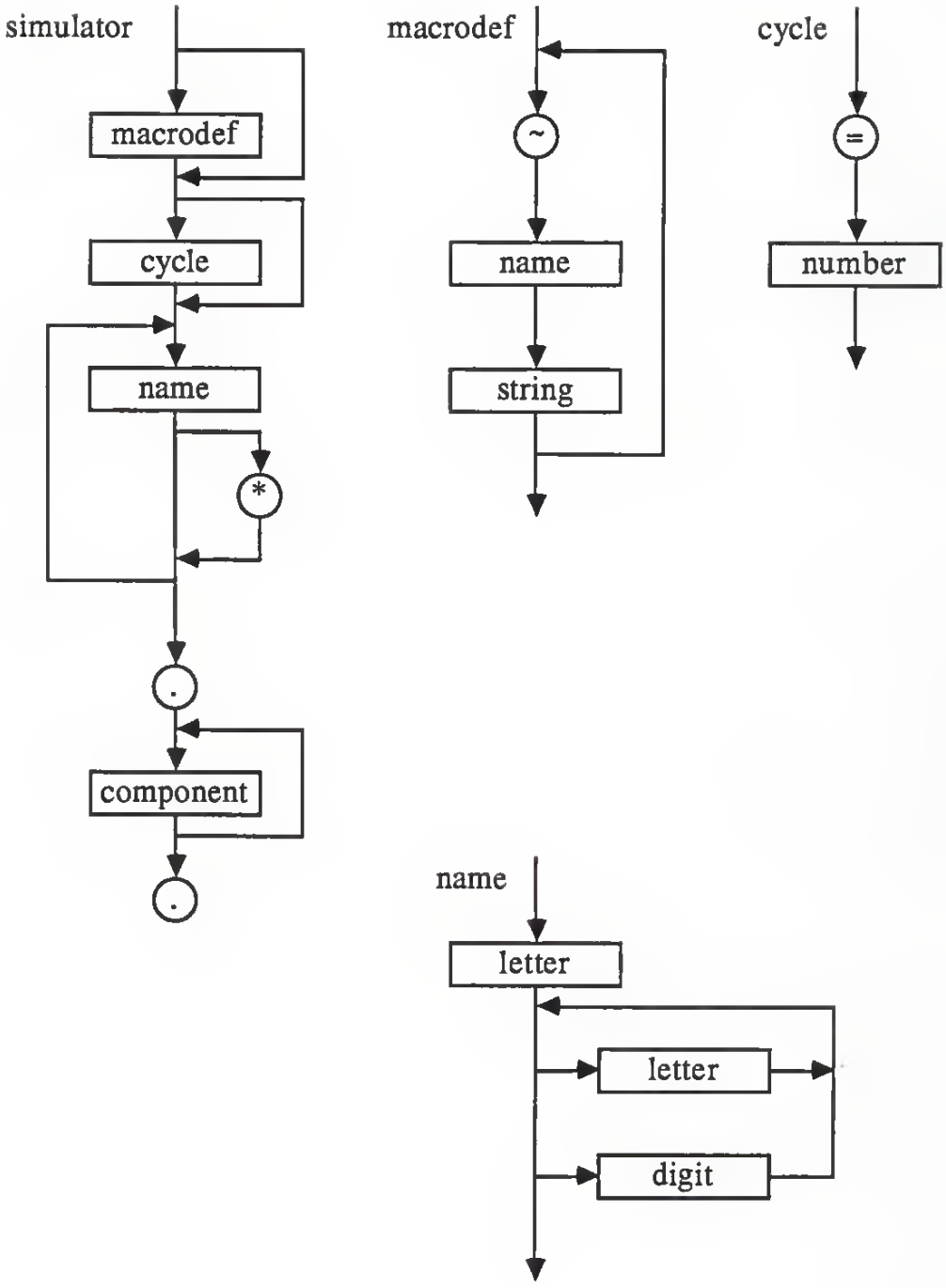
Below is a list of ALU functions and memory operations

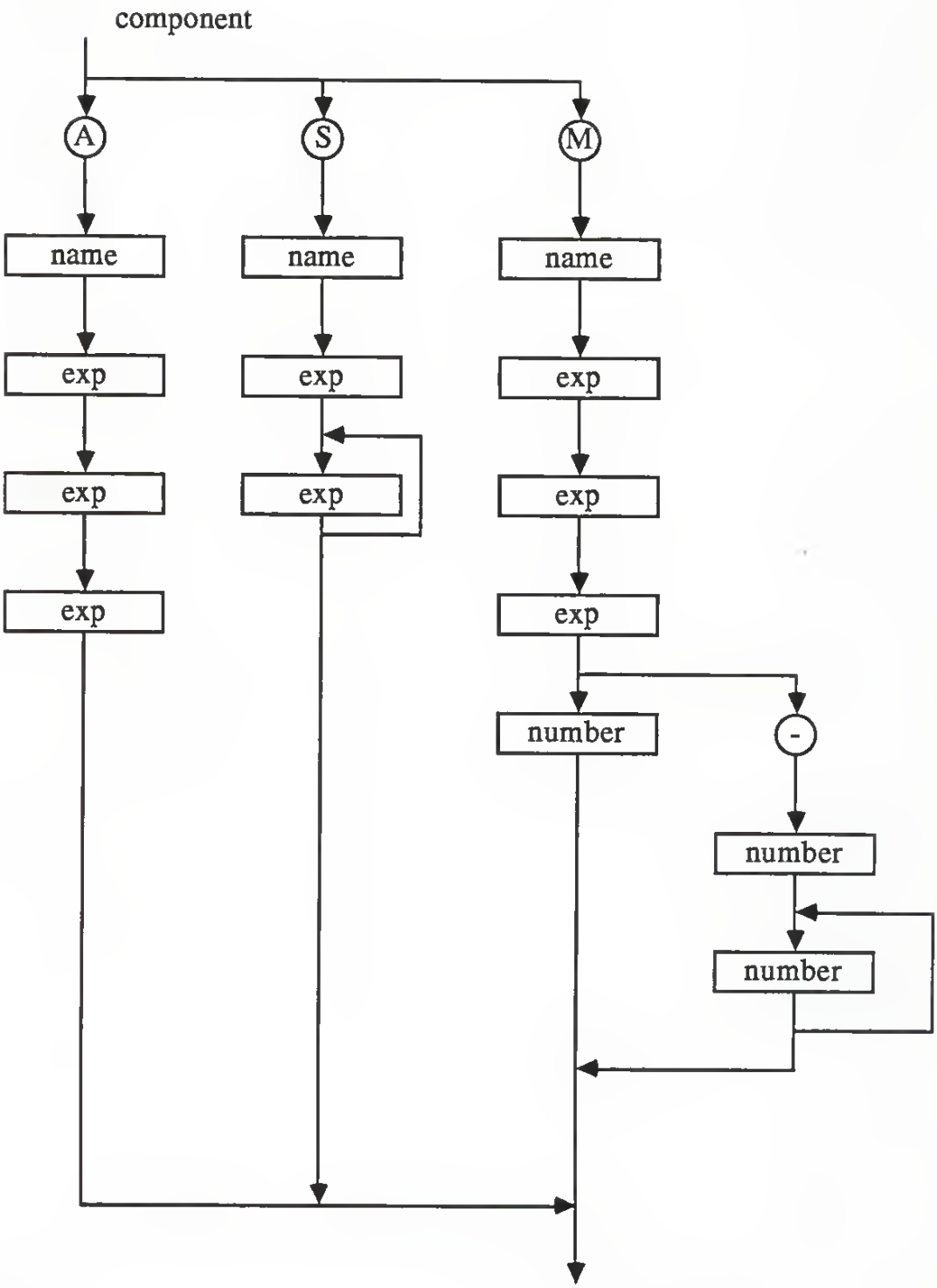
ALU functions:
0 0
1 right
2 left
3 NOT(left)
4 left + right
5 left - right
6 left * 2^right (shift left)
7 left * right
8 AND(left, right)
9 OR(left, right)
10 XOR(left, right)
11 unused
12 left = right
13 left < right

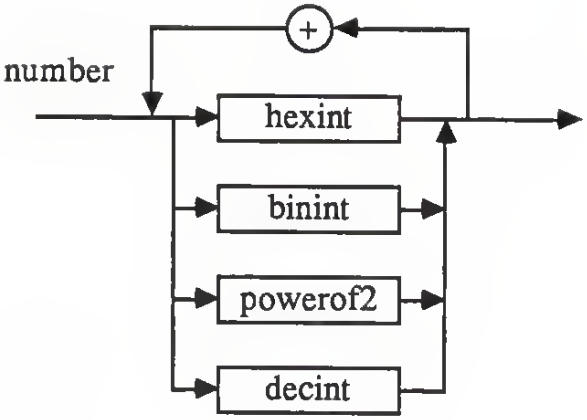
Memory operations:
0 read
1 write
2 input
3 output
4 trace writes
8 trace reads

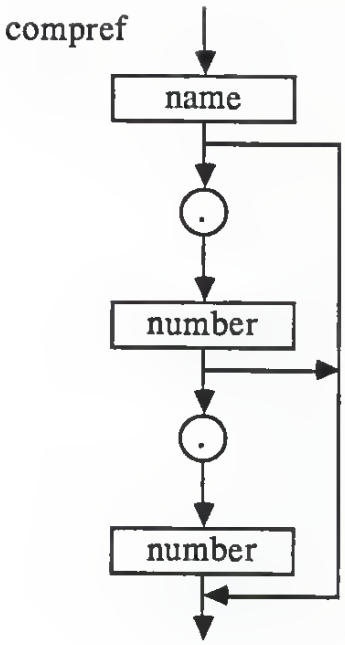
Functions 12 and 13 evaluate to 1 if true, 0 if false.

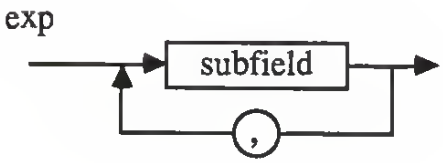
Appendix B
ASIM II Syntax

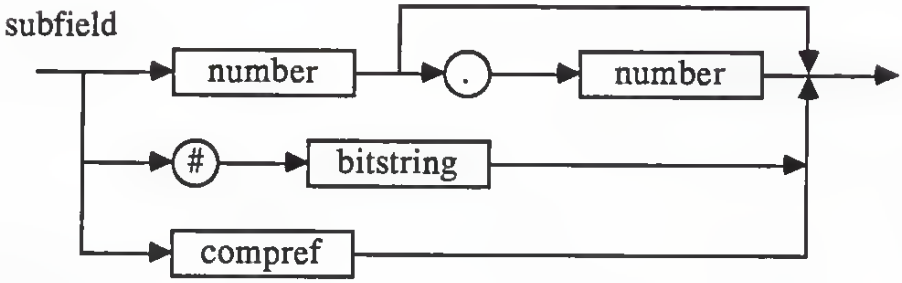


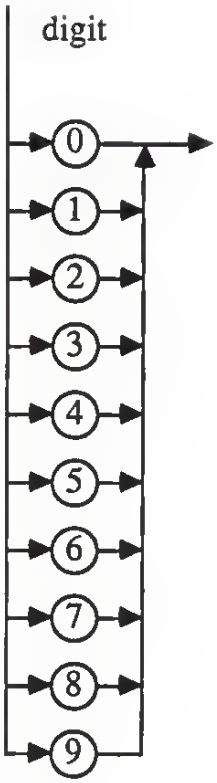

component
A
S
M
name
name
name
exp
exp
exp
exp
exp
exp
exp
exp
number
-
number
number

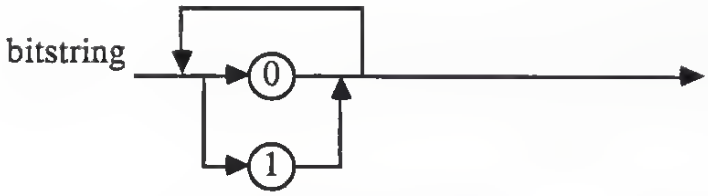

number
+
hexint
binint
powerof2
decint


compref
name
.
number
.
number

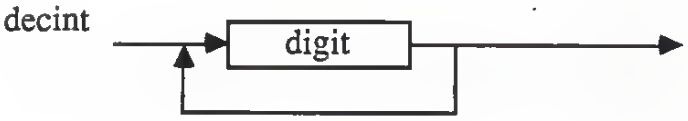

exp
subfield
,

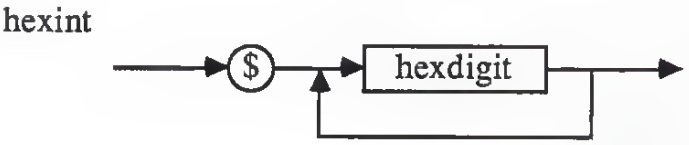

subfield
number
.
number
#
bitstring
compref


digit
0
1
2
3
4
5
6
7
8
9


bitstring
0
1


binint
%
bitstring

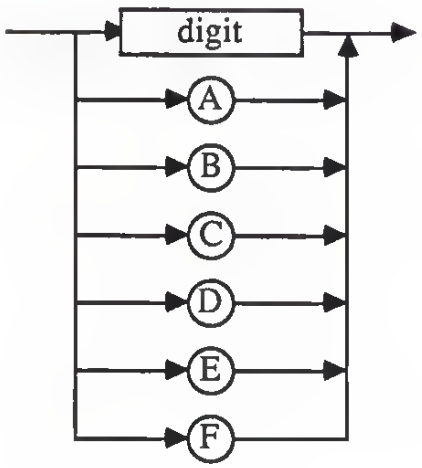

decint
digit


hexint
$
hexdigit


powerof2
^
decint


macro
~
name

hexdigit


digit
A
B
C
D
E
F

Appendix C

ASIM II Source Code

```
(*****************************************************************)
(*                                                               *)
(*  Author:          Lester Bartel                               *)
(*  Program name:    ASIM II                                     *)
(*  Date completed:  7 October 1986                              *)
(*                                                               *)
(*  Description:     ASIM II is an architecture simulator that   *)
(*    reads an input description from a file and produces        *)
(*    Pascal code which will simulate the specification.         *)
(*    If an error occurs in the specification, the code          *)
(*    generation ceases, and an error is reported describing     *)
(*    the error condition.                                       *)
(*                                                               *)
(*****************************************************************)

program simulator(input, output);

label
  1;

const
  strsize = 127; {max length of string}
  maxcomponents = 500; {max number of components}

type
  charset = set of char;
  string = array[0..strsize] of char;
  kindtype = (alu, sel, mem);
  caseptr = ^casetype;
  casetype = record
    casevalue: string;
    link: caseptr
  end; {record}
  valueptr = ^valuetype;
  valuetype = record
    value: integer;
    link: valueptr
  end; {record}
  nameptr = ^nametype;
  nametype = record
```

```
    name: string;
    print, used: boolean;
    link: nameptr
  end; {record}
  compptr = ^comptype;
  comptype = record
    used: boolean;
    name: string;
    link: compptr;
    case kind: kindtype of
      alu: (funct, left, right: string);
      sel: (select: string;
            cases: caseptr);
      mem: (addr, data, opn: string;
            number: integer;
            values: valueptr)
  end; {record}
  macroptr = ^macrotype;
  macrotype = record
    name, macro: string;
    link: macroptr
  end; {record}

var
  token, filename, comment: string;
  ch: char;
  donereading, err, varflag, endmacrodef, gettokenend: boolean;
  inf, sim: text;
  numbers, hexnums, letters, whitespace: charset;
  numcomponents, numcycles, i: integer;
  nametable, nptr: nameptr;
  comptable, ptr: compptr;
  macrotable: macroptr;
  highbits: array[0..31] of integer;

function length (a: string): integer;
(*************************************************************)
(*                                                           *)
(*  This function returns the length of the string passed in.  *)
(*                                                           *)
(*************************************************************)

begin
  length := ord(a[0])
end; (length)
```

```
procedure concat (var a: string; b: string);
(*****************************************************************)
(*                                                               *)
(*  This procedure concatenates the string b to the end of       *)
(*  string a, returning the string a.                            *)
(*                                                               *)
(*****************************************************************)

var index, lena, lenb: integer;
begin
  lena:= length(a);
  lenb:= length(b);
  if lena + lenb > strsize then
    lenb:= strsize - lena;
  a[0] := chr(lena+lenb);
  for index := 1 to lenb do
    a[lena+index]:= b[index]
end; {concat}

procedure concat1 (var a: string; b: char);
(*****************************************************************)
(*                                                               *)
(*  This procedure concatenates the character b to the end of    *)
(*  the string a returning the string a.                         *)
(*                                                               *)
(*****************************************************************)

begin
  if length(a) < strsize then a[0] := chr(length(a)+1);
  a[length(a)] := b
end; {concat1}

function strcmp (str1, str2: string): boolean;
(*****************************************************************)
(*                                                               *)
(*  This function compares the two strings passed to for         *)
(*  equality.  It returns a boolean value.                       *)
(*                                                               *)
(*****************************************************************)

var i, len1: integer;
begin
  if length(str1) <> length(str2) then
    strcmp := false
  else begin
    i := 1;
    len1 := length(str1);
    while (str1[i] = str2[i]) and (i < len1) do
      i := i + 1;
    if str1[i] = str2[i] then
```

```
      strcmp := true
    else
      strcmp := false
    end; (if)
end; (strcmp)

procedure swrt (var fil: text; a: string);
(*****************************************************************)
(*                                                               *)
(*  This procedure prints the string a to the file fil.          *)
(*                                                               *)
(*****************************************************************)

var i: integer;
begin
  for i:= 1 to length(a) do
    write(fil, a[i]);
end; (swrt)

function max(a, b: integer): integer;
(*****************************************************************)
(*                                                               *)
(*  This function returns the largest of a and b.                *)
(*                                                               *)
(*****************************************************************)

begin
  if a > b then
    max := a
  else
    max := b
end; (max)

procedure printcomperr;
(*****************************************************************)
(*                                                               *)
(*  This procedure prints the last component read in.  It is     *)
(*  used to report which component caused an error condition.    *)
(*                                                               *)
(*****************************************************************)

var ptr: compptr;
begin
  if not donereading then begin
    ptr := comptable;
    while ptr^.link <> nil do
      ptr := ptr^.link;
    write('Last component read is <');
    swrt(output, ptr^.name);
    writeln('> (error is in this or the next component).');
```

```
  end; {if}
end; {printcomperr}

function numberofbits(str: string): integer;
(*************************************************************)
(*                                                           *)
(*  This function returns the number of bits represented by  *)
(*  the expression in str.  It is used for code optimization.*)
(*                                                           *)
(*************************************************************)

var i, n, m: integer;
begin
  n := 0;
  m := 0;
  i := 1;
  while (i <= length(str)) and (n < 31) do begin
    if str[i] in ['%', '#'] then begin
      i := i + 1;
      m := 0;
      while (i <= length(str)) and (str[i] in ['0', '1'])
      do begin
        m := m + 1;
        i := i + 1
      end; {while}
      if str[i] <> '.' then
        n := n + m
      else begin
        i := i + 1;
        m := 0;
        while (str[i] in numbers) and (i <= length(str)) do begin
          m := m * 10 + ord(str[i]);
          i := i + 1
        end; {while}
        n := n + m;
      end; {if}
    end {if %}
    else if (str[i] = '$') or (str[i] in numbers) then begin
      i := i + 1;
      while not(str[i] in ['.', ',']) and (i <= length(str)) do
        i := i + 1;
      if (str[i] = '.') and (i <= length(str)) then begin
        m := 0;
        i := i + 1;
        while (str[i] in numbers) and (i <= length(str)) do begin
          m := m * 10 + ord(str[i]);
          i := i + 1
        end; {while}
        n := n + m
      end
```

```
      else
        n := 31
    end (if $)
    else if str[i] = '^' then begin
      i := i + 1;
      m := 0;
      while (str[i] in numbers) and (i <= length(str)) do begin
        m := m * 10 + ord(str[i]);
        i := i + 1;
      end; (while)
      n := max(n, m + 1)
    end {if ^)
    else if str[i] in letters then begin
      i := i + 1;
      while ((str[i] in letters) or (str[i] in numbers)) and
            (i <= length(str)) do
        i := i + 1;
      if (str[i] = '.') and (i <= length(str)) then begin
        m := 0;
        i := i + 1;
        while (str[i] in numbers) and (i <= length(str)) do begin
          m := m * 10 + ord(str[i]);
          i := i + 1
        end; (while)
        if (str[i] = '.') and (i <= length(str)) then begin
          n := n - m;
          m := 0;
          i := i + 1;
          while (str[i] in numbers) and (i <= length(str)) do begin
            m := m * 10 + ord(str[i]);
            i := i + 1
          end; (while)
          n := n + m + 1
        end {if)
        else
          n := n + 1
      end (if)
      else
        n := 31
    end; (if letters)
    i := i + 1;
  end; (main while}
  if n > 31 then
    n := 31;
  numberofbits := n
end; {numberofbits)
```

```
function str2num (a: string): integer;
(*************************************************************)
(*                                                           *)
(*  This function returns the integer equivalent of the string *)
(*  passed to it.  The string may consist of the summation of  *)
(*  any combination of numbers as defined in the syntax chart. *)
(*                                                           *)
(*************************************************************)

var i, j, k, l, m, len: integer;
begin
  j := 0;
  i := 1;
  len := length(a);
  while i <= len do begin
    if (a[i] in numbers) or (a[i] in ['^', '$', '%']) then begin
      case a[i] of
        '0', '1', '2', '3', '4', '5', '6', '7', '8', '9' : begin
                k := 0;
                while (a[i] in numbers) and (i <= len) do begin
                  k := k * 10 + ord(a[i]) - ord('0');
                  i := i + 1
                end; {while)
                j := j + k;
              end; (case numbers)
        '%' : begin
                k := 0;
                i := i + 1;
                while (a[i] in ['1', '0']) and (i <= len) do begin
                  k := k * 2;
                  if a[i] = '1' then k := k + 1;
                  i := i + 1
                end; {while)
                j := j + k;
              end; (case %)
        '$' : begin
                k := 0;
                i := i + 1;
                while (a[i] in hexnums) and (i <= len) do begin
                  k := k * 16;
                  if a[i] in numbers then
                    k := k + ord(a[i]) - ord('0')
                  else
                    k := k + ord(a[i]) - ord('A') + 10;
                  i := i + 1
                end; {while}
                j := j + k;
              end; {case $}
        '^' : begin
                k := 0;
```

```
                i := i + 1;
                while (a[i] in numbers) and (i <= len) do begin
                  k := k * 10 + ord(a[i]) - ord('0');
                  i := i + 1
                end; {while}
                l := 1;
                for m := 1 to k do
                  l := l * 2;
                j := j + 1;
              end {case ^}
      end; {case}
      if (i <= len) and (a[i] <> '+') then begin
        write('Error.  Malformed number ');
        swrt(output, a);
        writeln('.');
        err := true;
        printcomperr;
        goto 1
      end
    end
    else begin
      write('Error.  Malformed number ');
      swrt(output, a);
      writeln('.');
      err := true;
      printcomperr;
      goto 1
    end; {if}
    i := i + 1
  end; {while}
  str2num := j
end; {str2num}

function numeric (str: string): boolean;
(*************************************************************)
(*                                                           *)
(*  This function determines if the expression passed in as  *)
(*  string is a numeric constant.  It is used for optimization.*)
(*                                                           *)
(*************************************************************)

var i: integer;
begin
  numeric := true;
  for i := 1 to length(str) do
    if not (str[i] in ['+', '%', '$', '^', '0'..'9', 'A'..'F']) then
      numeric := false
end; (numeric)
```

```
function land (a, b: integer): integer;
(*****************************************************************)
(*                                                               *)
(*  This function performs the bit and function on the values    *)
(*  of a and b.                                                  *)
(*                                                               *)
(*****************************************************************)

type bitnos = 0..31;
     bigset = set of bitnos;
var intset: record case boolean of
              false: (i, j: integer);
              true:  (x, y: bigset)
            end;
begin
with intset do begin
  i := a;
  j := b;
  x := x * y;
  land := i
  end
end {land};

function findname (name: string): compptr;
(*****************************************************************)
(*                                                               *)
(*  This function finds the component name and returns a         *)
(*  pointer to it.                                               *)
(*                                                               *)
(*****************************************************************)

var ptr: compptr;
begin
  findname := nil;
  ptr := comptable;
  while (ptr <> nil) do begin
    if strcmp(ptr^.name, name) then findname := ptr;
    ptr := ptr^.link
  end; {while}
end; {findname}
```

```
procedure expr(str: string; tempflag: boolean);
(*************************************************************)
(*                                                           *)
(*  This procedure generates the Pascal code for the expr in *)
(*  str.  If tempflag is true, it generates code for the     *)
(*  temporary values for the memories.  Otherwise, code is   *)
(*  generated using the subscripted memory value.  The str   *)
(*  may be any valid expression derivable from the syntax    *)
(*  diagrams.                                                *)
(*                                                           *)
(*************************************************************)

var fptr: compptr;
    a, b, name: string;
    i, j, k, l, m, n, o, p, q, frombit, tobit, bits, numbits,
      consttotal: integer;
    fromflag, toflag, addflag, dirtyflag: boolean;
begin
  i := length(str);
  j := i;
  numbits := 0;
  consttotal := 0;
  addflag := false;
  dirtyflag := false;
  repeat
    while (i>0) and (str[i] <> ',') do
      i := i - 1;
    l := i;
    a[0] := chr(j - i);
    for k := i+1 to j do
      a[k-l] := str[k];
    if a[1] in ['$', '%', '^', '0'..'9'] then begin  {number}
      m := 1;
      while (m <= length(a)) and (a[m] <> '.') do begin
        b[m] := a[m];
        m := m + 1
      end; {while}
      b[0] := chr(m - 1);
      q := 0;
      o := str2num(b);
      if (a[m] = '.') and (m <= length(a)) then begin
        m := m + 1;
        n := m;
        while m <= length(a) do begin
          b[m - n + 1] := a[m];
          m := m + 1
        end; {while}
        b[0] := chr(length(a) - n - 1);
        for p := 1 to str2num(b) do
          q := q + highbits[p];
```

```
    consttotal := consttotal + land(o, q) * highbits[numbits];
    numbits := numbits + str2num(b)
  end
  else begin
    consttotal := consttotal + o * highbits[numbits];
    numbits := 31
  end {if}
end {case numbers}
else if a[1] = '#' then begin  {binary string}
  a[1] := '%';
  consttotal := consttotal + str2num(a) * highbits[numbits];
  numbits := numbits + length(a) - 1
end (case #}
else if a[1] in ['a'..'z', 'A'..'Z'] then begin
  {component reference}
  dirtyflag := true;
  fromflag := false;
  toflag := false;
  frombit := 0;
  tobit := 0;
  if addflag then write(sim, ' + ');
  m := 1;
  while (m <= length(a)) and (a[m] <> '.') do begin
    name[m] := a[m];
    m := m + 1
  end; (while)
  name[0] := chr(m - 1);
  fptr := findname(name);
  if fptr = nil then begin
    err := true;
    write('Error.  Component <');
    swrt(output, name);
    writeln('> not found.');
    goto 1
  end; (if)
  m := m + 1;
  n := 1;
  if m <= length(a) then begin
    while (m <= length(a)) and (a[m] <> '.') do begin
      b[n] := a[m];
      n := n + 1;
      m := m + 1
    end; {while}
    b[0] := chr(n - 1);
    fromflag := true;
    frombit := str2num(b);
    m := m + 1;
    n := 1;
  end; {if}
  if m <= length(a) then begin
```

```
  while m <= length(a) do begin
    b[n] := a[m];
    n := n + 1;
    m := m + 1
  end; {while}
  b[0] := chr(n - 1);
  tobit := str2num(b);
  toflag := true;
end; {if}
if fromflag then
  write(sim, 'land(');
fptr := findname(name);
if fptr <> nil then
  if fptr^.kind = mem then
    if tempflag then begin
      write(sim, 'temp');
      swrt(sim, fptr^.name)
    end
    else
    begin
      write(sim, 'ljb');
      swrt(sim, name);
      write(sim, '[');
      expr(fptr^.addr, tempflag);
      write(sim, ']')
    end {if}
  else begin
    write(sim, 'ljb');
    swrt(sim, name)
  end; {if}
if fromflag then begin
  write(sim, ', ');
  bits := highbits[frombit];
  if toflag then begin
    for m := frombit + 1 to tobit do
      bits := bits + highbits[m];
  end; {if}
  write(sim, bits:1, ')');
end; (if)
if frombit > numbits then
  write(sim, ' div ', highbits[frombit - numbits]:1)
else
  if numbits - frombit <> 0 then
    write(sim, ' * ', highbits[numbits - frombit]:1);
if fromflag then
  if toflag then
    numbits := numbits + tobit - frombit + 1
  else
    numbits := numbits + 1
else
```

```
        numbits := 31;
      addflag := true
    end {case letters}
    else begin
      err := true;
      write('Error.  Malformed expression ');
      swrt(output, str);
      writeln('.');
      printcomperr;
      goto 1
    end; {case}
    if numbits > 31 then begin
      err := true;
      write('Error.  Too many bits in ');
      swrt(output, str);
      writeln('.');
      goto 1
    end; {if}
    j := i - 1;
    i := i - 1
  until i < 0;
  if dirtyflag then
    if consttotal <> 0 then
      write(sim, ' + ', consttotal:1)
    else
  else
    write(sim, consttotal:1)
end; {expr}

procedure genfunctions;
(*****************************************************************)
(*                                                               *)
(*  This procedure generates the variable declarations, memory *)
(*  initialization procedure, alu function, input function,    *)
(*  and output procedure.                                        *)
(*                                                               *)
(*****************************************************************)

var vptr: valueptr;
begin
  {generate variable declarations}
  ptr := comptable;
  write(sim, 'var ');
  varflag := false;
  repeat
    if (varflag) then
      if (ptr^.kind in [alu, sel]) then begin
        write(sim, ', ljb');
        swrt(sim, ptr^.name)
      end
```

```
    else begin
      write(sim, ', temp');
      swrt(sim, ptr^.name);
      write(sim, ', adr');
      swrt(sim, ptr^.name);
      write(sim, ', data');
      swrt(sim, ptr^.name);
      write(sim, ', opn');
      swrt(sim, ptr^.name)
    end
  else
    if (ptr^.kind in [alu, sel]) then begin
      varflag := true;
      write(sim, 'ljb');
      swrt(sim, ptr^.name)
    end
    else begin
      write(sim, 'temp');
      swrt(sim, ptr^.name);
      write(sim, ', adr');
      swrt(sim, ptr^.name);
      write(sim, ', data');
      swrt(sim, ptr^.name);
      write(sim, ', opn');
      swrt(sim, ptr^.name);
      varflag := true
    end;
  ptr := ptr^.link
until ptr = nil;
writeln(sim, ': integer;');
writeln(sim, '    cycles, cyclecount: integer;');
ptr := comptable;
repeat
  if ptr^.kind = mem then begin
    write(sim, '    ljb');
    swrt(sim, ptr^.name);
    writeln(sim, ': array[0..', abs(ptr^.number) - 1:1, '] of
      integer;')
  end; (if)
  ptr := ptr^.link
until ptr = nil;

(generate bit and function)
writeln(sim);
writeln(sim, 'function land (a, b: integer): integer;');
writeln(sim, 'type bitnos = 0..31;');
writeln(sim, '     bigset = set of bitnos;');
writeln(sim, 'var intset: record case boolean of');
writeln(sim, '              false: (i, j: integer);');
writeln(sim, '              true:  (x, y: bigset)');
```

```
writeln(sim, '            end;');
writeln(sim, 'begin');
writeln(sim, 'with intset do begin');
writeln(sim, '  i := a;');
writeln(sim, '  j := b;');
writeln(sim, '  x := x * y;');
writeln(sim, '  land := i');
writeln(sim, '  end');
writeln(sim, 'end {land};');

(generate procedure to initialize memories)
writeln(sim);
writeln(sim, 'procedure initvalues;');
writeln(sim, 'var i: integer;');
writeln(sim, 'begin');
ptr := comptable;
while ptr <> nil do begin
  if ptr^.kind = mem then begin
    if ptr^.number < 0 then begin
      vptr := ptr^.values;
      for i := 0 to -ptr^.number - 1 do begin
        write(sim, '  ljb');
        swrt(sim, ptr^.name);
        writeln(sim, '[', i:1, '] := ', vptr^.value:1, ';');
        vptr := vptr^.link
      end; (for)
    end
    else begin
      writeln(sim, 'for i := 0 to ', ptr^.number - 1:1, ' do');
      write(sim, '  ljb');
      swrt(sim, ptr^.name);
      writeln(sim, '[i] := 0;')
    end; (if)
    write(sim, '  temp');
    swrt(sim, ptr^.name);
    writeln(sim, ' := 0;')
  end; (if)
  ptr := ptr^.link
end; (while)
writeln(sim, 'end; (initvalues)');

(generate function to calculate alu functions)
writeln(sim);
writeln(sim, 'function dologic(funct, left, right: integer):
  integer;');
writeln(sim, 'const mask = ', highbits[30] - 1 + highbits[30]:1,
  ';');
writeln(sim, 'var value : integer;');
writeln(sim, 'begin');
writeln(sim, '  value := 0;');
```

```
writeln(sim, '  case funct of');
writeln(sim, '    0 : value := 0;');
writeln(sim, '    1 : value := right;');
writeln(sim, '    2 : value := left;');
writeln(sim, '    3 : value := mask - left;');
writeln(sim, '    4 : value := left + right;');
writeln(sim, '    5 : value := left - right;');
writeln(sim, '    6 : while (right > 0) and (left <> 0) do begin');
writeln(sim, '          left := land(left + left, mask);');
writeln(sim, '          value := left;');
writeln(sim, '          right := right - 1;');
writeln(sim, '        end;');
writeln(sim, '    7 : value := left * right;');
writeln(sim, '    8 : value := land(left, right);');
writeln(sim, '    9 : value := left + right - land(left, right);');
writeln(sim, '    10: value := left + right - land(left, right) *
  2;');
writeln(sim, '    11: value := 0;');
writeln(sim, '    12: if left = right then value := 1;');
writeln(sim, '    13: if left < right then value := 1');
writeln(sim, '  end; (case)');
writeln(sim, '  dologic := value;');
writeln(sim, 'end; (dologic)');

{generate function for input to memory)
writeln(sim);
writeln(sim, 'function sinput(address: integer): integer;');
writeln(sim, 'var datum: char;');
writeln(sim, '    data: integer;');
writeln(sim, 'begin');
writeln(sim, '  if address = 0 then begin');
writeln(sim, '    read(input, datum);');
writeln(sim, '    sinput := ord(datum)');
writeln(sim, '  end');
writeln(sim, '  else if address = 1 then begin');
writeln(sim, '    read(input, data);');
writeln(sim, '    sinput := data');
writeln(sim, '  end');
writeln(sim, '  else begin');
writeln(sim, '    write(output, ''Input from address '', address:1,
  '': '');');
writeln(sim, '    readln(input, data);');
writeln(sim, '    sinput := data;');
writeln(sim, '  end');
writeln(sim, 'end; (sinput)');

(generate procedure for output from memory)
writeln(sim);
writeln(sim, 'procedure soutput(address, data: integer);');
writeln(sim, 'begin');
```

```
  writeln(sim, '  if address = 0 then writeln(output, chr(data))');
  writeln(sim, '  else if address = 1 then writeln(output, data)');
  writeln(sim, '  else writeln(output, ''Output to address '',
    address:1, '': '', data:1)');
  writeln(sim, 'end; (soutput)');
end; (genfunctions)

procedure checkname (name: string);
(*****************************************************************)
(*                                                               *)
(*  This procedure checks all names for valid characters.  If    *)
(*  the first character is not a letter, or the remaining        *)
(*  characters are not letters or numbers, then an error is      *)
(*  reported.                                                    *)
(*                                                               *)
(*****************************************************************)

var start, i: integer;
begin
  if name[1] = '~' then start := 2
  else start := 1;
  if not (name[start] in letters) then err := true;
  for i := start + 1 to length(name) do
    if not (name[i] in letters) and not (name[i] in numbers) then
      err := true;
  if err then begin
    write('Error.  Component name ');
    swrt(output, name);
    writeln(' invalid, use letters and numbers only.');
    goto 1
  end; {if}
end; (checkname)

procedure gettoken;
(*****************************************************************)
(*                                                               *)
(*  This procedure gets each whitespace delimited string of      *)
(*  characters and returns them in the global variable token.    *)
(*  Macro substitution is done here as well.  If a macro name    *)
(*  is found in the string, the actual text of the macro is      *)
(*  substituted immediately.                                     *)
(*                                                               *)
(*****************************************************************)

var macro: string;
    ptr : macroptr;
begin
  token[0] := chr(0);
  if gettokenend then begin
    token[0] := chr(1);
```

```
    token[1] := '.';
    gettokenend := false
  end
  else begin
    while (ch in whitespace) and (not eof(inf)) do
      if ch = '(' then
        while ch <> ')' do
          read(inf, ch)
      else
        read(inf, ch);
    (end if  end while)
    while not (ch in whitespace) and (not eof(inf)) do begin
      {substitute macro name for actual macro text)
      if endmacrodef and (ch = '~') then begin
        macro[0] := chr(0);
        concat1(macro, ch);
        read(inf, ch);
        while not (ch in whitespace) and (not eof(inf)) and
              ((ch in letters) or (ch in numbers)) do begin
          concat1(macro, ch);
          read(inf, ch)
        end; (while)
        ptr := macrotable;
        while (ptr <> nil) and (not strcmp(macro, ptr^.name)) do
          ptr := ptr^.link;
        if ptr = nil then begin
          write('Error.  Macro <');
          swrt(output, macro);
          writeln('> not defined.');
          err := true;
          goto 1
        end
        else
          concat(token, ptr^.macro)
      end
      else begin
        concat1(token, ch);
        read(inf, ch)
      end; (if)
    end; (while)
  end; (if)
  if (token[ord(token[0])] = '.') and (ord(token[0]) <> 1) then begin
    token[0] := chr(ord(token[0]) - 1);
    gettokenend := true
  end;
end; (gettoken)
```

```
procedure addname (namein: string; printflag: boolean);
(*************************************************************)
(*                                                           *)
(*  This procedure adds the component name and all of the data *)
(*  associated with it to the data structure.                *)
(*                                                           *)
(*************************************************************)

var ptr: nameptr;
begin
  checkname(namein);
  if nametable = nil then begin
    new(nametable);
    with nametable^ do begin
      link := nil;
      name := namein;
      print := printflag;
      used := false
    end; {with}
  end
  else begin
    ptr := nametable;
    while ptr^.link <> nil do begin
      ptr := ptr^.link;
    end;
    new(ptr^.link);
    with ptr^.link^ do begin
      link := nil;
      name := namein;
      print := printflag;
      used := false
    end; {with}
  end; {if}
end; {addname}

procedure name;
(*************************************************************)
(*                                                           *)
(*  This procedure reads the component name from the input   *)
(*  file.                                                    *)
(*                                                           *)
(*************************************************************)

begin
  while token[1] <> '.' do begin
    if token[length(token)] = '*' then begin
      token[0]:= chr(ord(token[0])-1);
      addname(token, true)
    end
    else
```

```
      addname(token, false);
    gettoken;
  end; (while}
  gettoken;
end; (name}

procedure macrodef;
(*****************************************************************)
(*                                                               *)
(*  This procedure reads and stores the macro definition in      *)
(*  the macro data structure.                                    *)
(*                                                               *)
(*****************************************************************)

var ptr: macroptr;
begin
  checkname(token);
  new(macrotable);
  with macrotable^ do begin
    link := nil;
    name := token;
    gettoken;
    macro := token
  end; (with}
  ptr := macrotable;
  gettoken;
  while token[1] = '~' do begin
    checkname(token);
    new(ptr^.link);
    ptr := ptr^.link;
    with ptr^ do begin
      link := nil;
      name := token;
      endmacrodef := true;
      gettoken;
      endmacrodef := false;
      macro := token
    end; (with}
    gettoken
  end; (while}
end; (macrodef}

procedure cycle;
(*****************************************************************)
(*                                                               *)
(*  This procedure gets the number of cycles which are to        *)
(*  be simulated.                                                *)
(*                                                               *)
(*****************************************************************)
```

```
begin
  gettoken;
  numcycles := str2num(token);
  gettoken
end; {cycle}

procedure readit;
(*****************************************************************)
(*                                                               *)
(*  This procedure reads the input file, calling any necessary *)
(*  support procedures.  It places the specification in the    *)
(*  data structure for processing by other procedures.         *)
(*                                                               *)
(*****************************************************************)

var nptr: nameptr;
    flag: boolean;
    vptr, ovptr: valueptr;
    cptr, ocptr: caseptr;
begin
  endmacrodef := false;
  comment[0] := chr(0);
  repeat
    read(inf, ch);
    concatl(comment, ch)
  until eoln(inf);
  writeln(sim, 'program simulator(input, output);');
  if comment[1] <> '#' then begin
    err:= true;
    writeln('Error.  Comment required.');
    goto 1
  end
  else begin
    write(sim, '(');
    swrt(sim, comment);
    writeln(sim, '}');
  end; {if}
  ch:= ' ';
  gettoken;
  if token[1] = '~' then macrodef;
  endmacrodef := true;
  numcycles := 0;
  if token[1] = '=' then cycle;
  name;
  while token[1] <> '.' do begin
    if (length(token) =1) and (token[1] in ['A', 'S', 'M']) then begin
      ptr := comptable;
      if ptr = nil then begin
        new(ptr);
        comptable := ptr
```

```
end
else begin
  while ptr^.link <> nil do
    ptr := ptr^.link;
  new(ptr^.link);
  ptr := ptr^.link
end; {if}
with ptr^ do begin
  link := nil;
  used := false;
  i := 0;
  numcomponents := numcomponents + 1;
  case token[1] of
    'A' : kind := alu;
    'S' : kind := sel;
    'M' : kind := mem
  end; {case}
  case kind of
    alu : begin
            gettoken;
            name := token;
            gettoken;
            funct := token;
            gettoken;
            left := token;
            gettoken;
            right := token;
            gettoken;
          end;
    sel : begin
            gettoken;
            name := token;
            gettoken;
            select := token;
            i:= 0;
            gettoken;
            repeat
              new(cptr);
              cptr^.casevalue := token;
              cptr^.link := nil;
              if i = 0 then
                ptr^.cases := cptr
              else
                ocptr^.link := cptr;
              ocptr := cptr;
              i := i + 1;
              gettoken
            until (token[1] in ['A', 'S', 'M', '.']) and
                  (length(token) = 1);
          end;
```

```
      mem : begin
              gettoken;
              name := token;
              gettoken;
              addr := token;
              gettoken;
              data := token;
              gettoken;
              opn := token;
              gettoken;
              if token[1] = '-' then begin
                for i := 2 to length(token) do
                  token[i-1] := token[i];
                token[0] := chr(ord(token[0]) - 1);
                number := -(str2num(token));
                for i := 0 to abs(number) - 1 do begin
                  gettoken;
                  new(vptr);
                  if i = 0 then
                    values := vptr
                  else
                    ovptr^.link := vptr;
                  ovptr := vptr;
                  vptr^.value := str2num(token);
                  vptr^.link := nil
                end; {for}
              end
              else
                number := str2num(token);
              gettoken;
            end;
    end; {case}
  nptr := nametable;
  repeat
    flag := strcmp(nptr^.name, name);
    if flag then begin
      used := true;
      nptr^.used := true
      end
      else
        nptr := nptr^.link;
    until flag or (nptr = nil);
  end; {with}
end
else begin
  err:= true;
  write('Error.  Component expected.  Got <');
  swrt(output, token);
  writeln('> instead.');
  printcomperr;
```

```
      goto 1
    end;
  end; (while);
  writeln(numcomponents:1, ' components read.');
end; (readit)

procedure checkdcl;
(*****************************************************************)
(*                                                               *)
(*  This procedure checks the declarations of the components.    *)
(*  if a component is declared but not used, or used, but not    *)
(*  declared, a warning message is issued.  Code generation      *)
(*  continues.                                                   *)
(*                                                               *)
(*****************************************************************)

var nptr: nameptr;
    cptr: compptr;
begin
  nptr := nametable;
  repeat
    with nptr^ do begin
      if not used then begin
        write('Warning:  ');
        swrt(output, name);
        writeln(' declared but not defined.')
      end; (if)
      nptr := nptr^.link
    end; (with)
  until nptr = nil;
  cptr := comptable;
  repeat
    with cptr^ do begin
      if not used then begin
        write('Warning:  ');
        swrt(output, name);
        writeln(' defined but not declared.')
      end; (if)
      cptr := cptr^.link
    end; (with)
  until cptr = nil
end;
```

```
function compare (a, b: string): boolean;
(*************************************************************)
(*                                                           *)
(*  This function breaks the expression in b into the        *)
(*  individual pieces, and compares these pieces with the    *)
(*  string a.  If the any part of b is in a, then compare    *)
(*  returns true.                                            *)
(*                                                           *)
(*************************************************************)

var i, j: integer;
    c: string;
begin
  i := 1;
  compare := false;
  while i <= length(b) do begin
    j := 1;
    if b[i] in letters then begin
      while ((b[i] in letters) or (b[i] in numbers)) and
            (i <= length(b)) do begin
        c[j] := b[i];
        j := j + 1;
        i := i + 1
      end; {while}
      c[0] := chr(j - 1);
      if strcmp(a, c) then
        compare := true;
    end; {if}
    i := i + 1;
  end; {while}
end; {compare}

function dependent (a, b: compptr): boolean;
(*************************************************************)
(*                                                           *)
(*  This function returns true if the component a depends on *)
(*  the value of component b.                                *)
(*                                                           *)
(*************************************************************)

var c: string;
    cptr: caseptr;
begin
  dependent := false;
  c:= b^.name;
  with a^ do
    case kind of
      alu: if compare(c, funct) or compare(c, left) or
              compare(c, right) then
             dependent := true;
```

```
      sel: begin
             if compare(c, select) then dependent := true;
             cptr := cases;
             while cptr <> nil do begin
               if compare(c, cptr^.casevalue) then
                 dependent := true;
               cptr := cptr^.link
             end; {while}
           end; {case sel}
      mem: dependent := false;
    end; {case}
end; {dependent}

procedure orderit;
(*************************************************************)
(*                                                           *)
(*  This procedure sorts the alu and selector components in  *)
(*  dependency order.  It also checks for circular           *)
(*  dependencies.                                            *)
(*                                                           *)
(*************************************************************)

var i, j, k, num: integer;
    templ, ptr: compptr;
    a, b: array[1..maxcomponents] of compptr;
begin
  i := 1;
  j := 1;
  num := 0;
  ptr := comptable;
  {break component list into 2 pieces: one with alus and
   selectors, and the other with memories}
  a[1] := nil;
  a[2] := nil;
  while ptr <> nil do begin
    case ptr^.kind of
      alu, sel: begin
                  a[i] := ptr;
                  i := i + 1;
                end;
      mem: begin
             b[j] := ptr;
             j := j + 1
           end
    end; {case}
    ptr := ptr^.link
  end; {while}
  a[i] := nil;
  b[j] := nil;
  num := i - 1;
```

```
  {sort alus and selectors}
  for k := 1 to num do
    for i := 1 to num do
      for j := i to num do
        if dependent(a[i], a[j]) then begin
          templ := a[j];
          a[j] := a[i];
          a[i] := templ
        end; {if}
  {check for remaining dependencies}
  for i := 1 to num do
    for j := i to num do
      if dependent(a[i], a[j]) then begin
        err := true;
        write('Error.  Circular dependency with ');
        swrt(output, a[j]^.name);
        write(' and/or ');
        swrt(output, a[i]^.name);
        writeln('.');
        goto 1
      end; {if}
  j := 1;
  if a[1] <> nil then
    comptable := a[1]
  else begin
    j := 2;
    comptable := b[1]
  end; {if}
  ptr := comptable;
  i := 2;
  while a[i] <> nil do begin
    ptr^.link := a[i];
    ptr := a[i];
    i := i + 1
  end; {while}
  while b[j] <> nil do begin
    ptr^.link := b[j];
    ptr := b[j];
    j := j + 1
  end; {while}
  ptr^.link := nil
end; {orderit}

procedure init;
{*****************************************************************}
{*                                                               *}
{*  This procedure initializes variables.                        *}
{*                                                               *}
{*****************************************************************}
```

```
begin
  donereading := false;
  nametable := nil;
  comptable := nil;
  macrotable := nil;
  numcomponents := 0;
  gettokenend := false;
  varflag := false;
  numbers := ['0'..'9'];
  letters := ['a'..'z', 'A'..'Z'];
  hexnums := ['0'..'9', 'A'..'F'];
  whitespace := [chr(9), chr(10), chr(13), ' ', '(', ')'];
  rewrite(sim, 'simulator.p');
  if argc = 2 then
    argv(1, filename)
  else begin
    writeln('Enter name of input file.');
    i:= 0;
    repeat
      read(filename[i]);
      i:=i+1
    until eoln
  end;
  reset(inf, filename);
  write('Reading file ');
  for i := 0 to 20 do
    write(filename[i]);
  writeln;
  highbits[0] := 1;
  for i := 1 to 31 do
    highbits[i] := highbits[i-1] * 2;
end; {init}

procedure gencode;
(*****************************************************************)
(*                                                               *)
(*  This procedure generates the Pascal code for the main        *)
(*  program of the simulator.                                    *)
(*                                                               *)
(*****************************************************************)

var fptr: compptr;
    flag, flag2: boolean;
    cptr: caseptr;
    mask: integer;
begin
  mask := highbits[30] - 1 + highbits[30];
  (generate main program)
  writeln(sim);
  writeln(sim, 'begin');
```

```
writeln(sim, 'initvalues;');
write(sim, 'cycles := ');
writeln(sim, numcycles:1, ';');
writeln(sim, 'if cycles = 0 then begin');
writeln(sim, '  writeln(''Number of cycles to trace'');');
writeln(sim, '  read(cycles);');
writeln(sim, 'end;');
writeln(sim, 'cyclecount := 0;');

{start of main loop}
writeln(sim, 'while cyclecount <= cycles do begin');

{generate assignments to components}
ptr := comptable;
while ptr <> nil do begin
  case ptr^.kind of
    alu: begin
           with ptr^ do begin
             if numeric(funct) then
               case str2num(funct) of
                 0: begin
                      write(sim, 'ljb');
                      swrt(sim, name);
                      writeln(sim, ' := 0;')
                    end;
                 1: begin
                      write(sim, 'ljb');
                      swrt(sim, name);
                      write(sim, ' := ');
                      expr(right, true);
                      writeln(sim, ';')
                    end;
                 2: begin
                      write(sim, 'ljb');
                      swrt(sim, name);
                      write(sim, ' := ');
                      expr(left, true);
                      writeln(sim, ';')
                    end;
                 3: begin
                      write(sim, 'ljb');
                      swrt(sim, name);
                      write(sim, ' := ', mask:1, ' - ');
                      expr(left, true);
                      writeln(sim, ';')
                    end;
                 4: begin
                      write(sim, 'ljb');
                      swrt(sim, name);
                      write(sim, ' := ');
```

```
      expr(left, true);
      write(sim, ' + ');
      expr(right, true);
      writeln(sim, ';');
    end;
5:  begin
      write(sim, 'ljb');
      swrt(sim, name);
      write(sim, ' := ');
      expr(left, true);
      write(sim, ' - ');
      expr(right, true);
      writeln(sim, ';');
    end;
6:  begin
      write(sim, 'ljb');
      swrt(sim, name);
      write(sim, ' := dologic(6, ');
      expr(left, true);
      write(sim, ', ');
      expr(right, true);
      writeln(sim, ');')
    end;
7:  begin
      write(sim, 'ljb');
      swrt(sim, name);
      write(sim, ' := ');
      expr(left, true);
      write(sim, ' * ');
      expr(right, true);
      writeln(sim, ';')
    end;
8:  begin
      write(sim, 'ljb');
      swrt(sim, name);
      write(sim, ' := ');
      write(sim, 'land(');
      expr(left, true);
      write(sim, ', ');
      expr(right, true);
      writeln(sim, ');')
    end;
9:  begin
      write(sim, 'ljb');
      swrt(sim, name);
      write(sim, ' := ');
      expr(left, true);
      write(sim, ' + ');
      expr(right, true);
      write(sim, ' - land(');
```

```
            expr(left, true);
            write(sim, ', ');
            expr(right, true);
            writeln(sim, ');')
          end;
        10:begin
            write(sim, 'ljb');
            swrt(sim, name);
            write(sim, ' := ');
            expr(left, true);
            write(sim, ' + ');
            expr(right, true);
            write(sim, ' - land(');
            expr(left, true);
            write(sim, ', ');
            expr(right, true);
            writeln(sim, ');')
          end;
        11:begin
            write(sim, 'ljb');
            swrt(sim, name);
            writeln(sim, ' := 0;')
          end;
        12:begin
            write(sim, 'if ');
            expr(left, true);
            write(sim, ' = ');
            expr(right, true);
            write(sim, ' then ljb');
            swrt(sim, name);
            writeln(sim, ' := 1');
            write(sim, '  else ljb');
            swrt(sim, name);
            writeln(sim, ' := 0;')
          end;
        13:begin
            write(sim, 'if ');
            expr(left, true);
            write(sim, ' < ');
            expr(right, true);
            write(sim, ' then ljb');
            swrt(sim, name);
            writeln(sim, ' := 1');
            write(sim, '  else ljb');
            swrt(sim, name);
            writeln(sim, ' := 0;')
          end
        end {case optimize alu functions}
    else begin
      write(sim, 'ljb');
```

```
                swrt(sim, name);
                write(sim, ' := dologic(');
                expr(funct, true);
                write(sim, ', ');
                expr(left, true);
                write(sim, ', ');
                expr(right, true);
                writeln(sim, ');');
              end; {if}
            end; {with}
          end; {case alu}
    sel: begin
           write(sim, 'case ');
           expr(ptr^.select, true);
           writeln(sim, ' of');
           i := 0;
           cptr := ptr^.cases;
           while cptr <> nil do begin
             write(sim, '  ', i:1, ' : ljb');
             swrt(sim, ptr^.name);
             write(sim, ' := ');
             expr(cptr^.casevalue, true);
             writeln(sim, ';');
             cptr := cptr^.link;
             i := i + 1
           end; {while}
           writeln(sim, 'end;');
         end; {case sel}
    mem: ;
  end; {case}
  ptr := ptr^.link
end; {while}

{generate trace statements}
nptr := nametable;
writeln(sim, 'write(''Cycle '', cyclecount:3);');
while nptr <> nil do begin
  if nptr^.print then begin
    write(sim, 'write('' ');
    swrt(sim, nptr^.name);
    write(sim, '= '', ');
    fptr := findname(nptr^.name);
    if fptr <> nil then
      if fptr^.kind = mem then begin
        write(sim, 'temp');
        swrt(sim, nptr^.name)
      end
      else begin
        write(sim, 'ljb');
        swrt(sim, nptr^.name)
```

```
      end; {if}
    writeln(sim, ':1);')
  end; {if}
  nptr := nptr^.link
end; (while)
writeln(sim, 'writeln;');

{assign temporary memory values (adr, data, opn)}
ptr := comptable;
while ptr <> nil do begin
  if ptr^.kind = mem then begin
    write(sim, 'adr');
    swrt(sim, ptr^.name);
    write(sim, ' := ');
    expr(ptr^.addr, true);
    writeln(sim, ';');
    write(sim, 'data');
    swrt(sim, ptr^.name);
    write(sim, ' := temp');
    swrt(sim, ptr^.name);
    writeln(sim, ';');
    write(sim, 'opn');
    swrt(sim, ptr^.name);
    write(sim, ' := ');
    expr(ptr^.opn, true);
    writeln(sim, ';')
  end; {if}
  ptr := ptr^.link;
end; (while)

{assign memories their new values}
ptr := comptable;
while ptr <> nil do begin
  if ptr^.kind = mem then begin
    if numeric(ptr^.opn) then begin
      case land(str2num(ptr^.opn), 3) of
        0 : begin
              write(sim, 'temp');
              swrt(sim, ptr^.name);
              write(sim, ' := ljb');
              swrt(sim, ptr^.name);
              write(sim, '[adr');
              swrt(sim, ptr^.name);
              writeln(sim, '];')
            end;
        1 : begin
              write(sim, 'temp');
              swrt(sim, ptr^.name);
              write(sim, ' := ');
              expr(ptr^.data, true);
```

```
            writeln(sim, ';');
            write(sim, 'ljb');
            swrt(sim, ptr^.name);
            write(sim, '[adr');
            swrt(sim, ptr^.name);
            write(sim, '] := temp');
            swrt(sim, ptr^.name);
            writeln(sim, ';')
          end;
      2 : begin
            write(sim, 'temp');
            swrt(sim, ptr^.name);
            write(sim, ' := sinput(');
            expr(ptr^.data, true);
            writeln(sim, ');')
          end;
      3 : begin
            write(sim, 'temp');
            swrt(sim, ptr^.name);
            write(sim, ' := ');
            expr(ptr^.data, true);
            writeln(sim, ';');
            write(sim, 'soutput(');
            expr(ptr^.data, true);
            write(sim, ', temp');
            swrt(sim, ptr^.name);
            writeln(sim, ');')
          end
    end (case)
  end
  else begin
    write(sim, 'case land(opn');
    swrt(sim, ptr^.name);
    writeln(sim, ', 3) of');
    write(sim, '  0: temp');
    swrt(sim, ptr^.name);
    write(sim, ' := ljb');
    swrt(sim, ptr^.name);
    write(sim, '[adr');
    swrt(sim, ptr^.name);
    writeln(sim, '];');
    writeln(sim, '  1: begin');
    write(sim, '       temp');
    swrt(sim, ptr^.name);
    write(sim, ' := ');
    expr(ptr^.data, true);
    writeln(sim, ';');
    write(sim, '       ljb');
    swrt(sim, ptr^.name);
    write(sim, '[adr');
```

```
  swrt(sim, ptr^.name);
  write(sim, '] := temp');
  swrt(sim, ptr^.name);
  writeln(sim, ';');
  writeln(sim, '     end;');
  write(sim, '  2: temp');
  swrt(sim, ptr^.name);
  write(sim, ' := sinput(adr');
  swrt(sim, ptr^.name);
  writeln(sim, ');');
  writeln(sim, '  3: begin');
  write(sim, '       temp');
  swrt(sim, ptr^.name);
  write(sim, ' := ');
  expr(ptr^.data, true);
  writeln(sim, ';');
  write(sim, '       soutput(adr');
  swrt(sim, ptr^.name);
  write(sim, ', temp');
  swrt(sim, ptr^.name);
  writeln(sim, ');');
  writeln(sim, '     end');
  writeln(sim, 'end; (case}')
end;

(generate code to trace writes}
flag := false;
flag2 := false;
if not (numeric(ptr^.opn)) and
       (numberofbits(ptr^.opn) >= 3) then
  flag := true
else
  if numeric(ptr^.opn) then
    if land(str2num(ptr^.opn), 4) = 4 then
      flag2 := true;
if flag then begin
  write(sim, 'if land(');
  expr(ptr^.opn, true);
  writeln(sim, ', 5) = 5 then');
  write(sim, '  ')
end;
if flag or flag2 then begin
  write(sim, 'writeln('' Write to ');
  swrt(sim, ptr^.name);
  write(sim, ' at '', ');
  expr(ptr^.addr, true);
  write(sim, ':1, '' : '', ');
  expr(ptr^.name, true);
  writeln(sim, ':1);')
end; {if}
```

```
      {generate code to trace reads)
      flag := false;
      flag2 := false;
      if not (numeric(ptr^.opn)) and (numberofbits(ptr^.opn) >=4) then
        flag := true
      else
        if numeric(ptr^.opn) then
          if land(str2num(ptr^.opn), 8) = 8 then
            flag2 := true;
      if flag then begin
        write(sim, 'if land(');
        expr(ptr^.opn, true);
        writeln(sim, ', 9) = 8 then');
        write(sim, '  ')
      end;
      if flag or flag2 then begin
        write(sim, 'writeln(''  Read from ');
        swrt(sim, ptr^.name);
        write(sim, ' at '', ');
        expr(ptr^.addr, true);
        write(sim, ':1, '' : '', ');
        expr(ptr^.name, true);
        writeln(sim, ':1);')
      end; {if)

    end; {if}
    ptr := ptr^.link
  end; {while)

  {generate code to check for end of  loop)
  writeln(sim, 'cyclecount := cyclecount + 1;');
  writeln(sim, 'if cyclecount = cycles + 1 then begin');
  writeln(sim, '  writeln(''Continue to cycle (0 to quit)'');');
  writeln(sim, '  read(cycles);');
  writeln(sim, 'end;');
  writeln(sim, 'end; {while)');
  {end of main loop}

  writeln(sim, 'end.');
end; {gencode)

begin {main}
  init;
  readit;
  donereading := true;
  writeln('Sorting components.');
  orderit;
  checkdcl;
  writeln('Generating code.');
```

```
  genfunctions;
  gencode;
1:
  if err then
    writeln('Error in program (no code generated).');
end.
```

Appendix D

Example Stack Machine Simulator Specification

```
# Itty Bitty Stack Machine Simulator Specification

{Macro bit functions}
~pack #000000000000 {=zero.0.~k}
~k 11       {high bit position in LDC loop}
~m 11       {high bit of RAM address}
~n 12       {I/O select bit in RAM address}
~d 5        {~d selects right operand for alu}
            {~d selects fp to ALU right}
~dd 7       {~d..~dd selects write data}
~st 4       {0..~st selects next state}
~a 10       {~a signals absolute addressing}
~f 11       {~f signals frame pointer update}
~g 9        {~g signals goto, not increment}
~i 6        {~i signals increment or branch}
~l 3        {~l loads left from ram}
~o 1        {~o signals pop, not push; ~z adds, not loads}
~p 7        {~p signals stack pointer update}
~r 4        {~r loads right from ram}
~s 12       {~s selects state from opcode+n}
~v 0        {~v selects frame pointer to load, not 1 to add}
~w 8        {~w writes into stack ram}
            {~w selects LDC loop test, not BZ}
~x 13       {~x enables condition test}
~y 5        {~y selects frame offset for ram address}
~z 2        {~z indicates escape; current opcode}

{Component list ...}
state rom parm relpc offset psp sp pushpop selfp fp afp addr
ram op left right neg selr alu exit write newpc pc prog ir
data newst.

{Component specifications...}
M state 0 newst 1 1          {write next state (at end)}

A exit %110,rom.~w ram rom.~w,~pack

S newst rom.~s.~x,exit.0     {~s selects state from opcode+n}
  {000} parm.0.~st           {next state from rom}
```

```
  {001} parm.0.~st
  {010} 1,rom.~z,prog.0.3     {~z indicates escape; current
                               opcode}
  {011} 1,rom.~z,prog.0.3
  {100} 0                     {~x enables condition test}
  {101} parm.0.~st            {~w selects LDC loop test, not
                               BZ}
  {110} 0
  {111} 1,rom.~z,prog.0.3

M pc 0 newpc rom.~i 1         {~i signals increment or branch}

A newpc %100 relpc offset

S relpc rom.~a pc 0           {~a signals absolute addressing}

S offset rom.~g 1 left        {~g signals goto, not increment}

M sp 0 pushpop rom.~p 1       {~p signals stack pointer
                               update}

A pushpop rom.~z,#0,rom.~o sp psp

S psp rom.~v.~z               {~v selects frame pointer to
                               load, not 1 to add}
  {0-2} 0 0 0                 {~o signals pop, not push; ~z
                               adds, not loads}
  {011} fp
  {100} 1
  {101} left
  {110} 1
  {111} right

M fp 0 selfp rom.~f 1         {~f signals frame pointer
                               update}

S selfp ir.0 sp ram           {load from current sp or from
                               stack}

S addr rom.~y sp afp          {~y selects frame offset for ram
                               address}

A afp %100 fp left            {frame offset = fp+left}

M left 0 ram rom.~l 1         {~l loads left from ram}

M right 0 ram rom.~r 1        {~r loads right from ram}

A neg %101 0 ram              {negative of right}

S selr parm.~d                {~d selects right operand for
                               alu}
```

```
  {0} right
  {1} fp

A alu op ram selr

M ir 0 prog rom.~s 1 {So that prog isn't held up, hack
                      remembers the value of prog at fetch
                      time.  Note that prog must be used to
                      calculate newst because ir won't be
                      valid until the cycle following the
                      fetch}

M data 0 prog parm.8 1        { gets prog's value when it is
                                data }

S write parm.~d.~dd           {~d..~dd selects write data}
  {000} alu
  {001} alu                   {~d selects fp to ALU right}
  {010} fp
  {011} pc
  {100} ir.0
  {101} ram.0.~k,data.0.3
  {110} left
  {111} neg

M ram addr.0.~m write addr.~n,rom.~w 4096 {~w writes into
             stack ram the ll sets trace reads & writes }

S op ir.0.3                  {Opcode-ALU function ROM follows}
  {0} 0 {1} 0 {2} %1 {3} %100 {4} %1 {5} %1000 {6} %1101
  {7} %1100 {8} %11 {9} 0 {10} %100 {11} %111 {12} %10
  {13} %1 {14} %1100 {15} %101

S rom state.0.5               {decode rom follows}
  {00 fetch}    ^~s+^~l+^~r+^~i
  {01 LDZ}      ^~w
  {02 LD0}      ^~w {~dd=4}  ^~w {~dd=5}
  {04 ST...}    ^~w+^~y {goto $19 so addr and ram can bounce
                         back}
  {05 =NOT}     ^~w
  {06 =NEG}     ^~w {~dd=7}
  {07 EQUAL}    ^~w {goto NEG}
  {08 INDEX...} ^~y+^~w+^~1 {goto SWAP+2}
  {09 SWAP...}  ^~w {~dd=6}
  {0A EXIT...}  ^~z+^~p+^~o+^~v+^~l  ^~a+^~g
  {0C LD}       ^~w
  {0D ST}       ^~r+^~z+^~p+^~o
  {0E BZ}       ^~x+^~z+^~p+^~o  ^~i+^~g
  {10 LDC}      ^~w {parm=^5+^7+^8, ~dd=5}
  {11 LDC}      ^~w {parm=^5+^7+^8}
  {12 SWAP}     ^~l+^~z+^~p+^~w
  {13 INDEX}    ^~r
```

```
  {14 LDC}      ^~w+^~i {parm=^5+^7+^8}
  {15 EXIT}     ^~f+^~p+^~o+^~z
  {16 CALL}     ^~w+^~a+^~g {~dd=3}
  {17 LDC}      ^~w+^~i {parm=^5+^7+^8, ~dd=5}
  {18 LDC}      ^~w+^~i {parm=^7+^8, ~dd=4}
  {19-1E}       0 0 0 0 0 0 {19-1E are interim states (see
                             parm)}
  {1F esc}      ^~s+^~z+^~i
  {20 esc}      0
  {21 LDZ}      ^~z+^~p
  {22 LD0}      ^~z+^~p+^~i
  {23 LD1}      ^~z+^~p+^~i {goto LD0+1}
  {24 DUPE}     ^~z+^~p {goto LD+1}
  {25 AND}      ^~z+^~p+^~o {goto NOT, goto 1E}
  {26 LESS}     ^~z+^~p+^~o {goto EQUAL+1, goto 1D}
  {27 EQUAL}    ^~z+^~p+^~o {goto 1D}
  {28 NOT}      ^~w
  {29 NEG}      ^~w {~dd=7}
  {2A ADD}      ^~z+^~p+^~o {goto NOT, goto 1E}
  {2B MPY}      ^~z+^~p+^~o {goto NOT, goto 1E}
  {2C LD}       ^~y
  {2D ST}       ^~z+^~p+^~o {goto 1C}
  {2E BZ}       ^~z+^~p+^~o {goto 1B}
  {2F GLOB}     ^~w {~dd=1}
  {30 NOP}      0
  {31 LDC}      ^~z+^~p+^~i
  {32 SWAP}     ^~z+^~p+^~o {goto 1A}
  {33 INDEX}    ^~z+^~p+^~o
  {34 ENTER}    ^~w+^~f+^~p+^~z+^~v {~dd=2}
  { enter does the following: write fp to ^sp, fp gets sp,
    sp gets sp + left, increment pc }
  {35 EXIT}     ^~p+^~o+^~v
  {36 CALL}     ^~i
  {37-3F} 0 0 0 0 0 0 0 0 0

S parm state.0.5        {part 2 of decode rom}
  {00} 0 0 128+3+^8 160 25 0 224 6 9 192 11 0 0 4 15 25
  {10} 0+^5+^7+^8 16+^5+^7+^8 9 8 17+^5+^7+^8 10 96
       20+^5+^7+^8
  {18} 23+^7+^8 0 18 14 13 7 5 0
  {20} 31 1 2 2 12 30 29 29 0 224 30 30 12 28 27 32
  {30} 0 24 26 19 64 21 22 0 0 0 0 0 0 0 0 0

M prog pc 0 0 -133 {-1024}          {program rom follows}
 0 0 3 10               { 000: 1d1 10 ; 1dc 26 }
 0  4                   {  enter }
                        {  ; count = 0 }
 1                      {  1dz }
 2  4                   {  1d0 4 }
13                      {  st }
                        { ; for(i=0; i<=size; i++)
                          ; flags[i] = true }
```

```
 2  5                    {  ld0 5  ; one less than array addr
                            since 1 is added immediately }
 2  1                   { FOR1     ld0 1 }
10                       {  add }
 4                       {  dupe }
 2  1                    {  ld0 1 }
 0  2                    {  swap }
13                       {  st }
 4                       {  dupe }
 3 10                    {  ld1 10 ; ldc 26 = size + array
                            offset }
 7                       {  equal }
 3  1                    {  ld1 1; ldc 17 = endfor1 - for1 }
 9                       {  neg }
14                       {  bz }
 2  5                   { ENDFOR1  ld0 5 ; loc 5 unused, get
                           junk off stack }
13                       {  st }
                         { ; for (i=0; i <= size; i++) }
 1                       {  ldz }
 2  1                    {  ld0 1 }
13                       {  st }
                         { ; if (flags[i]) }
 2  1                   { FOR2     ld0 1 ; get i }
12                       {  ld }
 2  6                    {  ld0 6 ; get start of flags }
10                       {  add }
12                       {  ld ; get flags[i] }
 0  1 0 0 3 10           {  ldc 58=$3a     ; INC - IF }
14                       {  bz }
                         { ; prime = i + i + 3 }
 2  1                   { IF       ld0 1 ; get i }
12                       {  ld }
 4                       {  dupe }
 4                       {  dupe }
10                       {  add ; i + i }
 2  3                    {  ld0 3 }
10                       {  add ; this is the prime number }
 4                       { dupe }
 0  1  1 0 0 0          { ld0 4096; output prime }
13                       { st }
 4                       { dupe }
 2  2                    { ld0 2 ; store prime }
13                       { st }
                         { ; for(k=i+prime; k <= size;
                             k+=prime} ; flags[k] = false }
10                       { add ; k=i+prime }
 4                       { FOR3    dupe }
 2  6                    { ld0 6 }
10                       { add ; add array address }
 1                       { ldz }
 0  2                    { swap }
```

```
13                      { st ; flags[k]=0 }
 2  2                   { ld0 2; get prime }
12                      { ld }
10                      { add ; k = k + prime }
 4                      { dupe }
 3  5                   { ld1 5 ; ldc 21 = size + 1 }
 6                      { less }
 2  5                   { ld0 ; ENDFOR3 - SKIP }
14                      { bz }
 1                      { SKIP    ldz }
 3  8                   { ld1 8 ; ldc 24=ENDFOR3-FOR3 }
 9                      { neg }
14                      { bz ; forced jump }
 2  5                   { ENDFOR3 ld0 5 ; here if
                          not(k<size+1) }
13                      { st ; store it to get it off the
                          stack }
                        { ;count++ }
 2  4                   { ld0 4 }
12                      { ld }
 2  1                   { ld0 1 }
10                      { add }
 2  4                   { ld0 4 }
13                      { st }
                        { ; i++ }
 2  1                   { INC     ld0 1 }
12                      { ld }
 2  1                   { ld0 1 }
10                      { add }
 4                      { dupe }
 2  1                   { ld0 1 }
13                      { st }
 3  5                   { ld1 5; ldc 21=size+1 }
 7                      { equal }
 0  1 0 0 5 13          { ldc 93=$5d ;ENDFOR2 - FOR2 }
 9                      { neg }
14                      { bz; goto beginning of for loop }
0 0                     { ENDFOR2 nop }
0 0
.
```

Appendix E

Pascal Code for Example Specification

```
program simulator(input, output);
{# Itty Bitty Stack Machine Simulator Specification}
var ljbrom, ljbexit, ljbrelpc, ljboffset, ljbnewpc, ljbpsp,
    ljbpushpop, ljbselfp, ljbafp, ljbaddr, ljbneg, ljbparm,
    ljbop, ljbselr, ljbalu, ljbnewst, ljbwrite, tempstate,
    adrstate, datastate, opnstate, temppc, adrpc, datapc,
    opnpc, tempsp, adrsp, datasp, opnsp, tempfp, adrfp,
    datafp, opnfp, templeft, adrleft, dataleft, opnleft,
    tempright, adrright, dataright, opnright, tempir, adrir,
    datair, opnir, tempdata,  adrdata, datadata, opndata,
    tempram, adrram, dataram, opnram,  tempprog, adrprog,
    dataprog, opnprog: integer;
    cycles, cyclecount: integer;
    ljbstate: array[0..0] of integer;
    ljbpc: array[0..0] of integer;
    ljbsp: array[0..0] of integer;
    ljbfp: array[0..0] of integer;
    ljbleft: array[0..0] of integer;
    ljbright: array[0..0] of integer;
    ljbir: array[0..0] of integer;
    ljbdata: array[0..0] of integer;
    ljbram: array[0..4095] of integer;
    ljbprog: array[0..132] of integer;

function land (a, b: integer): integer;
type bitnos = 0..31;
     bigset = set of bitnos;
var intset: record case boolean of
              false: (i, j: integer);
              true:  (x, y: bigset)
            end;
begin
with intset do begin
  i := a;
  j := b;
  x := x * y;
  land := i
  end
end {land};
```

```
procedure initvalues;
var i: integer;
begin
for i := 0 to 0 do
  1jbstate[i] := 0;
  tempstate := 0;
for i := 0 to 0 do
  ljbpc[i] := 0;
  temppc := 0;
for i := 0 to 0 do
  1jbsp[i] := 0;
  tempsp := 0;
for i := 0 to 0 do
  ljbfp[i] := 0;
  tempfp := 0;
for i := 0 to 0 do
  ljbleft[i] := 0;
  templeft := 0;
for i := 0 to 0 do
  ljbright[i] := 0;
  tempright := 0;
for i := 0 to 0 do
  ljbir[i] := 0;
  tempir := 0;
for i := 0 to 0 do
  1jbdata[i] := 0;
  tempdata := 0;
for i := 0 to 4095 do
  ljbram[i] := 0;
  tempram := 0;
  ljbprog[0] := 0;
  ljbprog[1] := 0;
  ljbprog[2] := 3;
  ljbprog[3] := 10;
  1jbprog[4] := 0;
  1jbprog[5] := 4;
  ljbprog[6] := 1;
  ljbprog[7] := 2;
  ljbprog[8] := 4;
  ljbprog[9] := 13;
  ljbprog[10] := 2;
  1jbprog[11] := 5;
  ljbprog[12] := 2;
  ljbprog[13] := 1;
  ljbprog[14] := 10;
  ljbprog[15] := 4;
  ljbprog[16] := 2;
  ljbprog[17] := 1;
  ljbprog[18] := 0;
  ljbprog[19] := 2;
  ljbprog[20] := 13;
```

```
ljbprog[21] := 4;
ljbprog[22] := 3;
ljbprog[23] := 10;
ljbprog[24] := 7;
ljbprog[25] := 3;
ljbprog[26] := 1;
ljbprog[27] := 9;
ljbprog[28] := 14;
ljbprog[29] := 2;
ljbprog[30] := 5;
ljbprog[31] := 13;
ljbprog[32] := 1;
ljbprog[33] := 2;
ljbprog[34] := 1;
ljbprog[35] := 13;
ljbprog[36] := 2;
ljbprog[37] := 1;
ljbprog[38] := 12;
ljbprog[39] := 2;
ljbprog[40] := 6;
ljbprog[41] := 10;
ljbprog[42] := 12;
ljbprog[43] := 0;
ljbprog[44] := 1;
ljbprog[45] := 0;
ljbprog[46] := 0;
ljbprog[47] := 3;
ljbprog[48] := 10;
ljbprog[49] := 14;
ljbprog[50] := 2;
ljbprog[51] := 1;
ljbprog[52] := 12;
ljbprog[53] := 4;
ljbprog[54] := 4;
ljbprog[55] := 10;
ljbprog[56] := 2;
ljbprog[57] := 3;
ljbprog[58] := 10;
ljbprog[59] := 4;
ljbprog[60] := 0;
ljbprog[61] := 1;
ljbprog[62] := 1;
ljbprog[63] := 0;
ljbprog[64] := 0;
ljbprog[65] := 0;
ljbprog[66] := 13;
ljbprog[67] := 4;
ljbprog[68] := 2;
ljbprog[69] := 2;
ljbprog[70] := 13;
ljbprog[71] := 10;
ljbprog[72] := 4;
```

```
ljbprog[73] := 2;
ljbprog[74] := 6;
ljbprog[75] := 10;
ljbprog[76] := 1;
ljbprog[77] := 0;
ljbprog[78] := 2;
ljbprog[79] := 13;
ljbprog[80] := 2;
ljbprog[81] := 2;
ljbprog[82] := 12;
ljbprog[83] := 10;
ljbprog[84] := 4;
ljbprog[85] := 3;
ljbprog[86] := 5;
ljbprog[87] := 6;
ljbprog[88] := 2;
ljbprog[89] := 5;
ljbprog[90] := 14;
ljbprog[91] := 1;
ljbprog[92] := 3;
ljbprog[93] := 8;
ljbprog[94] := 9;
ljbprog[95] := 14;
ljbprog[96] := 2;
ljbprog[97] := 5;
ljbprog[98] := 13;
ljbprog[99] := 2;
ljbprog[100] := 4;
ljbprog[101] := 12;
ljbprog[102] := 2;
ljbprog[103] := 1;
ljbprog[104] := 10;
ljbprog[105] := 2;
ljbprog[106] := 4;
ljbprog[107] := 13;
ljbprog[108] := 2;
ljbprog[109] := 1;
ljbprog[110] := 12;
ljbprog[111] := 2;
ljbprog[112] := 1;
ljbprog[113] := 10;
ljbprog[114] := 4;
ljbprog[115] := 2;
ljbprog[116] := 1;
ljbprog[117] := 13;
ljbprog[118] := 3;
ljbprog[119] := 5;
ljbprog[120] := 7;
ljbprog[121] := 0;
ljbprog[122] := 1;
ljbprog[123] := 0;
ljbprog[124] := 0;
```

```
  ljbprog[125] := 5;
  ljbprog[126] := 13;
  ljbprog[127] := 9;
  ljbprog[128] := 14;
  ljbprog[129] := 0;
  ljbprog[130] := 0;
  ljbprog[131] := 0;
  ljbprog[132] := 0;
  tempprog := 0;
end; {initvalues}

function dologic(funct, left, right: integer): integer;
const mask = 2147483647;
var value : integer;
begin
  value := 0;
  case funct of
    0 : value := 0;
    1 : value := right;
    2 : value := left;
    3 : value := mask - left;
    4 : value := left + right;
    5 : value := left - right;
    6 : while (right > 0) and (left <> 0) do begin
          left := land(left + left, mask);
          value := left;
          right := right - 1;
        end;
    7 : value := left * right;
    8 : value := land(left, right);
    9 : value := left + right - land(left, right);
    10: value := left + right - land(left, right) * 2;
    11: value := 0;
    12: if left = right then value := 1;
    13: if left < right then value := 1
  end; {case}
  dologic := value;
end; {dologic}

function sinput(address: integer): integer;
var datum: char;
    data: integer;
begin
  if address = 0 then begin
    read(input, datum);
    sinput := ord(datum)
  end
  else if address = 1 then begin
    read(input, data);
    sinput := data
  end
  else begin
```

```
    write(output, 'Input from address ', address:1, ': ');
    readln(input, data);
    sinput := data;
  end
end; {sinput}

procedure soutput(address, data: integer);
begin
  if address = 0 then writeln(output, chr(data))
  else if address = 1 then writeln(output, data)
  else writeln(output, 'Output to address ', address:1,
    ': ', data:1)
end; {soutput}

begin
initvalues;
cycles := 0;
if cycles = 0 then begin
  writeln('Number of cycles to trace');
  read(cycles);
end;
cyclecount := 0;
while cyclecount <= cycles do begin
case land(tempstate, 63) of
  0 : ljbrom := 4184;
  1 : ljbrom := 256;
  2 : ljbrom := 256;
  3 : ljbrom := 256;
  4 : ljbrom := 288;
  5 : ljbrom := 256;
  6 : ljbrom := 256;
  7 : ljbrom := 256;
  8 : ljbrom := 296;
  9 : ljbrom := 256;
  10 : ljbrom := 143;
  11 : ljbrom := 1536;
  12 : ljbrom := 256;
  13 : ljbrom := 150;
  14 : ljbrom := 8326;
  15 : ljbrom := 576;
  16 : ljbrom := 256;
  17 : ljbrom := 256;
  18 : ljbrom := 396;
  19 : ljbrom := 16;
  20 : ljbrom := 320;
  21 : ljbrom := 2182;
  22 : ljbrom := 1792;
  23 : ljbrom := 320;
  24 : ljbrom := 320;
  25 : ljbrom := 0;
  26 : ljbrom := 0;
  27 : ljbrom := 0;
```

```
  28 : ljbrom := 0;
  29 : ljbrom := 0;
  30 : ljbrom := 0;
  31 : ljbrom := 4164;
  32 : ljbrom := 0;
  33 : ljbrom := 132;
  34 : ljbrom := 196;
  35 : ljbrom := 196;
  36 : ljbrom := 132;
  37 : ljbrom := 134;
  38 : ljbrom := 134;
  39 : ljbrom := 134;
  40 : ljbrom := 256;
  41 : ljbrom := 256;
  42 : ljbrom := 134;
  43 : ljbrom := 134;
  44 : ljbrom := 32;
  45 : ljbrom := 134;
  46 : ljbrom := 134;
  47 : ljbrom := 256;
  48 : ljbrom := 0;
  49 : ljbrom := 196;
  50 : ljbrom := 134;
  51 : ljbrom := 134;
  52 : ljbrom := 2437;
  53 : ljbrom := 131;
  54 : ljbrom := 64;
  55 : ljbrom := 0;
  56 : ljbrom := 0;
  57 : ljbrom := 0;
  58 : ljbrom := 0;
  59 : ljbrom := 0;
  60 : ljbrom := 0;
  61 : ljbrom := 0;
  62 : ljbrom := 0;
  63 : ljbrom := 0;
end;
ljbexit := dologic(land(ljbrom, 256) div 256 + 12, tempram,
                   land(ljbrom, 256) * 16);
case land(ljbrom, 1024) div 1024 of
  0 : ljbrelpc := temppc;
  1 : ljbrelpc := 0;
end;
case land(ljbrom, 512) div 512 of
  0 : ljboffset := 1;
  1 : ljboffset := templeft;
end;
ljbnewpc := ljbrelpc + ljboffset;
case land(ljbrom, 7) of
  0 : ljbpsp := 0;
  1 : ljbpsp := 0;
  2 : ljbpsp := 0;
```

```
  3 : ljbpsp := tempfp;
  4 : ljbpsp := 1;
  5 : ljbpsp := templeft;
  6 : ljbpsp := 1;
  7 : ljbpsp := tempright;
end;
ljbpushpop := dologic(land(ljbrom, 2) div 2 + land(ljbrom,
  4), tempsp, ljbpsp);
case land(tempir, 1) of
  0 : ljbselfp := tempsp;
  1 : ljbselfp := tempram;
end;
ljbafp := tempfp + templeft;
case land(ljbrom, 32) div 32 of
  0 : ljbaddr := tempsp;
  1 : ljbaddr := ljbafp;
end;
ljbneg := 0 - tempram;
case land(tempstate, 63) of
  0 : ljbparm := 0;
  1 : ljbparm := 0;
  2 : ljbparm := 387;
  3 : ljbparm := 160;
  4 : ljbparm := 25;
  5 : ljbparm := 0;
  6 : ljbparm := 224;
  7 : ljbparm := 6;
  8 : ljbparm := 9;
  9 : ljbparm := 192;
  10 : ljbparm := 11;
  11 : ljbparm := 0;
  12 : ljbparm := 0;
  13 : ljbparm := 4;
  14 : ljbparm := 15;
  15 : ljbparm := 25;
  16 : ljbparm := 416;
  17 : ljbparm := 432;
  18 : ljbparm := 9;
  19 : ljbparm := 8;
  20 : ljbparm := 433;
  21 : ljbparm := 10;
  22 : ljbparm := 96;
  23 : ljbparm := 436;
  24 : ljbparm := 407;
  25 : ljbparm := 0;
  26 : ljbparm := 18;
  27 : ljbparm := 14;
  28 : ljbparm := 13;
  29 : ljbparm := 7;
  30 : ljbparm := 5;
  31 : ljbparm := 0;
  32 : ljbparm := 31;
```

```
  33 : ljbparm := 1;
  34 : ljbparm := 2;
  35 : ljbparm := 2;
  36 : ljbparm := 12;
  37 : ljbparm := 30;
  38 : ljbparm := 29;
  39 : ljbparm := 29;
  40 : ljbparm := 0;
  41 : ljbparm := 224;
  42 : ljbparm := 30;
  43 : ljbparm := 30;
  44 : ljbparm := 12;
  45 : ljbparm := 28;
  46 : ljbparm := 27;
  47 : ljbparm := 32;
  48 : ljbparm := 0;
  49 : ljbparm := 24;
  50 : ljbparm := 26;
  51 : ljbparm := 19;
  52 : ljbparm := 64;
  53 : ljbparm := 21;
  54 : ljbparm := 22;
  55 : ljbparm := 0;
  56 : ljbparm := 0;
  57 : ljbparm := 0;
  58 : ljbparm := 0;
  59 : ljbparm := 0;
  60 : ljbparm := 0;
  61 : ljbparm := 0;
  62 : ljbparm := 0;
  63 : ljbparm := 0;
end;
case land(tempir, 15) of
  0 : ljbop := 0;
  1 : ljbop := 0;
  2 : ljbop := 1;
  3 : ljbop := 4;
  4 : ljbop := 1;
  5 : ljbop := 8;
  6 : ljbop := 13;
  7 : ljbop := 12;
  8 : ljbop := 3;
  9 : ljbop := 0;
  10 : ljbop := 4;
  11 : ljbop := 7;
  12 : ljbop := 2;
  13 : ljbop := 1;
  14 : ljbop := 12;
  15 : ljbop := 5;
end;
case land(ljbparm, 32) div 32 of
  0 : ljbselr := tempright;
```

```
  1 : ljbselr := tempfp;
end;
ljbalu := dologic(ljbop, tempram, ljbselr);
case land(ljbexit, 1) + land(ljbrom, 12288) div 2048 of
  0 : ljbnewst := land(ljbparm, 31);
  1 : ljbnewst := land(ljbparm, 31);
  2 : ljbnewst := land(tempprog, 15) + land(ljbrom, 4) * 4 +
        32;
  3 : ljbnewst := land(tempprog, 15) + land(ljbrom, 4) * 4 +
        32;
  4 : ljbnewst := 0;
  5 : ljbnewst := land(ljbparm, 31);
  6 : ljbnewst := 0;
  7 : ljbnewst := land(tempprog, 15) + land(ljbrom, 4) * 4 +
        32;
end;
case land(ljbparm, 224) div 32 of
  0 : ljbwrite := ljbalu;
  1 : ljbwrite := ljbalu;
  2 : ljbwrite := tempfp;
  3 : ljbwrite := temppc;
  4 : ljbwrite := land(tempir, 1);
  5 : ljbwrite := land(tempdata, 15) + land(tempram, 4095) *
        16;
  6 : ljbwrite := templeft;
  7 : ljbwrite := ljbneg;
end;
write('Cycle ', cyclecount:3);
writeln;
adrstate := 0;
datastate := tempstate;
opnstate := 1;
adrpc := 0;
datapc := temppc;
opnpc := land(ljbrom, 64) div 64;
adrsp := 0;
datasp := tempsp;
opnsp := land(ljbrom, 128) div 128;
adrfp := 0;
datafp := tempfp;
opnfp := land(ljbrom, 2048) div 2048;
adrleft := 0;
dataleft := templeft;
opnleft := land(ljbrom, 8) div 8;
adrright := 0;
dataright := tempright;
opnright := land(ljbrom, 16) div 16;
adrir := 0;
datair := tempir;
opnir := land(ljbrom, 4096) div 4096;
adrdata := 0;
datadata := tempdata;
```

```
opndata := land(ljbparm, 256) div 256;
adrram := land(ljbaddr, 4095);
dataram := tempram;
opnram := land(ljbrom, 256) div 256 + land(ljbaddr, 4096)
  div 2048;
adrprog := temppc;
dataprog := tempprog;
opnprog := 0;
tempstate := ljbnewst;
ljbstate[adrstate] := tempstate;
case land(opnpc, 3) of
  0: temppc := ljbpc[adrpc];
  1: begin
       temppc := ljbnewpc;
       ljbpc[adrpc] := temppc;
     end;
  2: temppc := sinput(adrpc);
  3: begin
       temppc := ljbnewpc;
       soutput(adrpc, temppc);
     end
end; {case}
case land(opnsp, 3) of
  0: tempsp := ljbsp[adrsp];
  1: begin
       tempsp := ljbpushpop;
       ljbsp[adrsp] := tempsp;
     end;
  2: tempsp := sinput(adrsp);
  3: begin
       tempsp := ljbpushpop;
       soutput(adrsp, tempsp);
     end
end; {case}
case land(opnfp, 3) of
  0: tempfp := ljbfp[adrfp];
  1: begin
       tempfp := ljbselfp;
       ljbfp[adrfp] := tempfp;
     end;
  2: tempfp := sinput(adrfp);
  3: begin
       tempfp := ljbselfp;
       soutput(adrfp, tempfp);
     end
end; {case}
case land(opnleft, 3) of
  0: templeft := ljbleft[adrleft];
  1: begin
       templeft := tempram;
       ljbleft[adrleft] := templeft;
     end;
```

```
  2: templeft := sinput(adrleft);
  3: begin
       templeft := tempram;
       soutput(adrleft, templeft);
     end
end; (case}
case land(opnright, 3) of
  0: tempright := ljbright[adrright];
  1: begin
       tempright := tempram;
       ljbright[adrright] := tempright;
     end;
  2: tempright := sinput(adrright);
  3: begin
       tempright := tempram;
       soutput(adrright, tempright);
     end
end; {case}
case land(opnir, 3) of
  0: tempir := ljbir[adrir];
  1: begin
       tempir := tempprog;
       ljbir[adrir] := tempir;
     end;
  2: tempir := sinput(adrir);
  3: begin
       tempir := tempprog;
       soutput(adrir, tempir);
     end
end; {case}
case land(opndata, 3) of
  0: tempdata := ljbdata[adrdata];
  1: begin
       tempdata := tempprog;
       ljbdata[adrdata] := tempdata;
     end;
  2: tempdata := sinput(adrdata);
  3: begin
       tempdata := tempprog;
       soutput(adrdata, tempdata);
     end
end; {case}
case land(opnram, 3) of
  0: tempram := ljbram[adrram];
  1: begin
       tempram := ljbwrite;
       ljbram[adrram] := tempram;
     end;
  2: tempram := sinput(adrram);
  3: begin
       tempram := ljbwrite;
       soutput(adrram, tempram);
```

```
    end
end; {case}
tempprog := ljbprog[adrprog];
cyclecount := cyclecount + 1;
if cyclecount = cycles + 1 then begin
  writeln('Continue to cycle (0 to quit)');
  read(cycles);
end;
end; {while}
end.
```

Appendix F

Example of specification translation to a hardware diagram

Specification in ASIM II

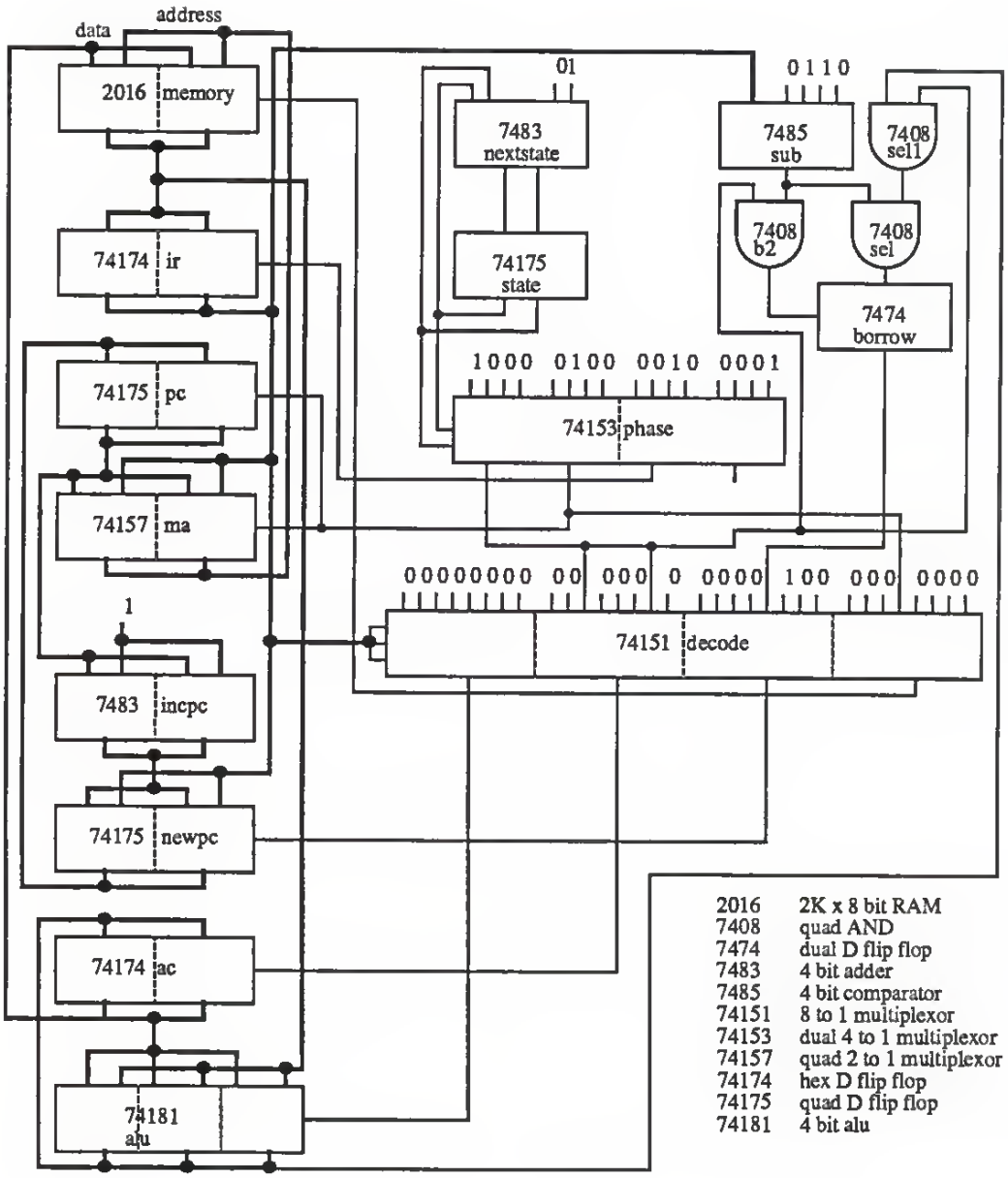
```
#  tiny computer specification 1986 June 12

{macro definition of instrucions}
~LD 256  ~ST 384  ~BB 512  ~BR 640  ~SU 768

state* nextstate phase pc* incpc newpc ir decode ma
memory ac* borrow alu sel sel1 b2 sub.

M state 0 nextstate.0.1 1 1 {state counter}
A nextstate %0100 state 1
S phase state.0.1 %0001 %0010 %0100 %1000
A incpc %0100 pc 1          {add pc+1 (or load ir) in
                              phase.2}
S newpc decode.1 incpc ir
M pc 0 newpc.0.6 phase.2 1
M ir 0 memory phase.1 1
S decode ir.7.9
  0
  0
  phase.3,#00
  phase.2
  borrow,#0
  #10                              {unconditional only}
  1,phase.3,#00
  0
A alu decode.3,#01 ac memory.0.9  {subtract or load}
M ac 0 alu.0.10 decode.2 1
M borrow 0 sel b2 1     {borrow flag set only on
                          subtract}
```

```
A b2 8 phase.3 sub       {operation and during phase 3}
A sub 12 %110 ir.7.9     {is this a subtract operation?}
A sel 8 sub sel1         { select based on subtract
                           operation and}
A sel1 8 alu.10 phase.3     {phase 3 and bit 10 of alu}
S ma phase.2 pc ir
M memory ma.0.6 ac decode.0 -128
{decimal memory data follows}
~LD+30    {load accumulator from location 30}
~SU+31    {subtract value in location 31 from
           accumulator}
~LD+30
~SU+32
~LD+32
~SU+30
~LD+34
~SU+33
~LD+30
~SU+31
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0
5 7 7 10 5
0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
.
```


data
address
2016 memory
74174 ir
74175 pc
74157 ma
1
7483 incpc
74175 newpc
74174 ac
74181 alu
01
7483 nextstate
74175 state
0110
7485 sub
7408 sel1
7408 b2
7408 sel
7474 borrow
1000 0100 0010 0001
74153 phase
00000000 00 000 0 0000 100 000 0000
74151 decode
2016 2K x 8 bit RAM
7408 quad AND
7474 dual D flip flop
7483 4 bit adder
7485 4 bit comparator
74151 8 to 1 multiplexor
74153 dual 4 to 1 multiplexor
74157 quad 2 to 1 multiplexor
74174 hex D flip flop
74175 quad D flip flop
74181 4 bit alu

COMPUTER ARCHITECTURE SIMULATION USING A REGISTER TRANSFER LANGUAGE

by

LESTER BARTEL

B. A., Tabor College, 1983

AN ABSTRACT OF A MASTER'S THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1986

ABSTRACT

ASIM II (Architecture Simulator II) is a compiler which compiles an electronic hardware description to Pascal. When executed, this Pascal code simulates the hardware described in the specification. The components of an electronic system are described by three primitives: ALU, selector, and memory. These three primitives are sufficient to describe any piece of digital electronic equipment and resemble their hardware counterparts in a digital electronic system. ASIM II is different from other computer hardware description languages in that it uses only these three primitives. It is not based upon an underlying programming language on which it is implemented, or on a more complex set of primitives. ASIM II significantly reduces the simulation time over an interpreter while maintaining the same functionality.