The synthesis of application-specific machines using the Euler language

Russell Dean Meier
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

Part of the Computer Sciences Commons, and the Electrical and Electronics Commons

Recommended Citation
https://lib.dr.iastate.edu/rtd/11633

This Dissertation is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6” x 9” black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
The synthesis of application-specific machines using the Euler language

by

Russell Dean Meier

A dissertation submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering
Major Professor: Charles T. Wright, Jr.

Iowa State University
Ames, Iowa

1998

Copyright © Russell Dean Meier. 1998. All rights reserved.
This is to certify that the Doctoral dissertation of

Russell Dean Meier

has met the dissertation requirements of Iowa State University

Signature was redacted for privacy.

Major Professor

Signature was redacted for privacy.

For the Major Program

Signature was redacted for privacy.

For the Graduate College
DEDICATION

This dissertation is dedicated to my best friend Heidi T. Meier. The past eight years of my life have been enriched by her love and friendship. She has patiently supported my work as both student and teacher. I am grateful to her for so many things, but most of all for her unconditional love for me. Without Heidi, my life would be so empty.
TABLE OF CONTENTS

CHAPTER 1. INTRODUCTION

The Design Abstraction ................................................................. 1
The Level View of Computers .......................................................... 3
Pedagogy ......................................................................................... 8
Dissertation Outline ........................................................................ 9

CHAPTER 2. THE SAMUEL ENVIRONMENT ........................................ 11

Why is SAMUEL Novel? ................................................................. 14
Advantages ................................................................................... 20
Disadvantages ............................................................................... 21

CHAPTER 3. EULER ................................................................. 23

Language Perspectives .................................................................... 23
Example: A Shift Cipher ................................................................. 28
Example: Insertion Sort .................................................................. 34
Example: Recursive Factorial .......................................................... 36
Euler-97 ......................................................................................... 37

CHAPTER 4. THE SAMUEL MACHINE TEMPLATE .............................. 41

The Set of Required Circuits ............................................................ 45
The Set of Opcode Circuits ............................................................... 58
Example: Mathematical Opcodes ..................................................... 62
Example: Delimiters ....................................................................... 66
Example: Data Types ........................................................................................................... 67

CHAPTER 5. EXAMPLE MACHINES ......................................................................................... 69

CHAPTER 6. CONCLUSIONS AND FUTURE WORK ................................................................. 79

Future Work .......................................................................................................................... 81

APPENDIX A. THE EULER-97 COMPILER ........................................................................ 83

APPENDIX B. VERILOG OPCODES ....................................................................................... 117

APPENDIX C. EXAMPLE MACHINES .................................................................................. 130

REFERENCES ......................................................................................................................... 177
CHAPTER 1. INTRODUCTION

This dissertation documents a rapid prototyping environment for creating application-specific custom computing machines that can be implemented in field programmable gate arrays (FPGAs). The environment is called SAMUEL, an acronym for the Synthesis of Application-specific Machines Using the Euler Language. The fundamental premise of this project is that an algorithm expressed in a high-level language, such as Euler, defines an application-specific custom computing machine. As suggested by the acronym, the environment will allow a user to enter an algorithm in Euler and, in some sense, synthesize the resulting machine. The synthesis task is reduced to selecting library components, called opcodes, from an available set and interfacing these components together to form the final computing machine.

This dissertation developed around three main project goals: create an environment that raises the level of the design abstraction, apply architectural support for a high-level language, and provide an environment that could be beneficial to pedagogy. These combined goals create a rapid prototyping environment that is useful to a broad user base.

The Design Abstraction

The first goal was to create an environment that raises the level of abstraction at which the user describes custom computing machines. In fact, a system was envisioned
where a user would enter an algorithm. This specification of the custom computing machine would be all that was needed to actually have a working FPGA in just a few minutes. Thus, the user would not need any knowledge of digital design techniques. This approach differs from the current trend.

A review of current conference proceedings shows that the majority of custom computing machines designed for FPGAs are created schematically and behaviorally [1,2,3]. Schematic design techniques require a high degree of digital design knowledge. Behavioral design methods allow users to enter descriptions in a hardware description language. VHDL and Verilog are currently the industry standard hardware description languages. VHDL and Verilog both have language constructs that do allow circuits to be described at a level of abstraction that is similar to general algorithms [4,5,6]. But, it can be argued that both languages also, unfortunately, force the traditional "black-box" digital design methodology on users. In other words, in order to create a good VHDL or Verilog description, the user must be familiar with input and output ports, modular design techniques (hierarchy), registers, wires, signals, events, processes, concurrency, sequential behavior, and timing constraints, just to name a few!

If SAMUEL could provide an environment where the user would not need this background knowledge, users from broad areas might be able to use FPGAs to quickly test algorithms, speed up computations, or even to experiment with electronics and circuit design. For example, a biomedical engineer is not classically trained in digital design techniques, yet that engineer might desire an FPGA in a system that monitors body signals and produces appropriate outputs. The computational behavior might be easily
described algorithmically, and the FPGA is a single chip solution (as opposed to a microcontroller and its support chips). Or, consider a secondary school teacher in the United States that wants to encourage her students to become interested in science, electronics, mathematics, etc. The desktop PC is a great environment for software development, but there is still something to be said for creating your own “chips” that do the work for you!

Additionally, even if a user does possess the background knowledge in digital design techniques, SAMUEL still provides an alternative environment to demonstrate algorithm performance improvements in hardware. Using SAMUEL, the algorithm would be quickly transformed into the resulting FPGA for testing. However, the schematic design of a custom computing machine for the algorithm could potentially take a significant amount of layout and testing. Similarly, a hardware description of the algorithm might take longer to create since the actual algorithm must be implemented using the constructs of a language that more directly supports hardware design abstractions.

The Level View of Computers

The second goal was to create an environment that directly supported a high-level language in the architecture of the custom computing machine. Computing circuits optimized for the instructions of the high-level language used by the SAMUEL system would exhibit better performance than non-optimized circuits. This idea is not new. In
fact, architectural support for high-level languages is well documented. See, for example, the excellent collection of papers [7]. A quick review of data paths and levels of machines can help to put this part of the dissertation work in context. For reference, see [8].

Consider a custom computing machine as a set of machines called level 0, level 1, level 2, and so on. Level 0 is the physical data path. Here, the data path elements are all coordinated by a control unit. The data path elements would consist of computational units, support registers, and input/output units for example. For the example data path in Figure 1.1, let the nanoinstruction be defined to be the logic 1 and 0 values on data path element control signals \{LDA, LDB, OP1, OP0, DRALU, DRX\}. The control unit

![Figure 1.1: Example data path with control unit](image)
generates a sequence of nanoinstructions through time and thus manipulates data. This sequence results in computation in the sense that the data path is left in a final state when the sequence terminates. This level 0 machine defines the architecture and data path organization that is viewed by a programmer of the nanoinstructions. A programmer writing code for the level 0 machine of Figure 1.1 would write an instruction segment similar to Figure 1.2. This instruction segment is called a nanoprogram since it consists of a sequence (i.e. a program) of nanoinstructions. It can also be said that instructions of the form in Figure 1.2 are written in machine language. Now, abstract these nanoprograms upward one level to level 1. Each nanoprogram can become an instruction in the level 1 machine. A programmer at level 1 could then program MULT2(X), for example, instead of the code in Figure 1.2, provided that an appropriate translation mechanism exists to convert level 1 programs to level 0 programs. Perhaps we could call the level 1 instruction a microinstruction. Sequences of microinstructions executing on the level 1 machine could thus be called microprograms. These microinstructions might potentially be considered the base level for “somewhat readable” programs, and thus form

Nanoinstructions for \( \text{REG B} = x+x \)

\{LDA,LDB,OP1,OP0,DRALU,DRX\}
1,1,1,1,1,0 comment load a and b
0,1,0,1,0,1 comment drive \( x+x \) into b

Figure 1.2: A nanoprogram for the level 0 data path
the assembly language of the data path. Note, however, that nothing anchors the assembly language to the level 1 instructions. A data path designer could decide that the instructions of a higher machine level were more appropriate for the assembly language of the data path. The creation of more abstract languages continues as levels are added to hide the details of the data path. The most important point is that the expressive power of the languages is increasing as the machine level increases. In turn, the readability of programs is increasing due to a higher semantic level and expressive power given by the higher-level syntactic constructs.

Most modern programming languages have robust semantics supported by syntactical constructs. Thus, programs written in modern programming languages are many levels away from the nanoprograms of the microprocessors (general purpose data paths) on which they execute. Figure 1.3 diagrams this view of a computing machine.

Level 3
Programs written in the high-level language Pascal

Level 2
P-code

Level 1
Assembly

Level 0
Machine code

Figure 1.3: Language levels
From this discussion, it is clear that a large semantic gap can potentially exist between most high-level programming languages and the data paths on which they execute. This semantic gap must be bridged.

One possible bridging technique is translation from level to level. Translation converts a program from level n into a program at level n-1. If this process stops before level 0, say at level k, the resulting translated code is ready for interpretation by the level k machine. Thus, an interpretation program, called an interpreter, must be created in the language of level k. This process of translation from one abstraction level to some other abstractly lower level is commonly referred to as compilation. The program at level n is said to be compiled to the machine code of level k [8.9]. Clearly, compilation can proceed from any arbitrary level to level 0 given appropriate translation rules.

Translation across the semantic gap generally leads to more instructions that need to be executed at the lower level than at the starting level. Recall that the higher-level languages are much more expressive than the level 0 language. It is clear, then, that a program written in j instructions at level 3 will need more instructions at level 0. Thus, translation introduces overhead into the computation in two important ways: program storage space and performance.

A second bridging technique is to directly decrease the semantic gap by making the level 0 language more sophisticated. This approach is used by SAMUEL. This can be done by implementing level k instructions directly at level 0. Thus, all levels between the original levels 0 and k disappear and the semantic gap narrows. This is the basic idea of architectural support for directly executed high-level languages architectures [7].
seems logical, therefore, that since the overhead of translation has been reduced, performance should increase. It is also logical to assume that since the data path at level 0 has increased in sophistication, it will require more circuitry to implement.

SAMUEL bridges the semantic gap by using only two levels. A high-level language that is fairly robust is translated once to a set of somewhat less sophisticated operation codes (opcodes). These opcodes are then implemented as the data path nanoinstructions. Under one classification scheme, SAMUEL data paths could be classified as software translated, language corresponding, complex, directly executing architectural support for high-level languages [7].

Pedagogy

The third project goal was to create an environment that would be beneficial to the teaching of certain interesting computer science and engineering topics. Such an environment would encourage instructors to use rapid prototyping and FPGAs as learning aids in laboratory-based classrooms. Example instructional topics for laboratory work include parameter passing mechanisms, block structure, stack-based mathematics, virtual machines, stacks, typed versus untyped data objects, procedures as first-class data objects, and run-time error generation. Currently, SAMUEL uses a language that illustrates these topics very well. The SAMUEL environment synthesizes data paths that further illustrate hardware support for these topics.
Dissertation Outline

The remaining chapters of this dissertation will outline the components of the SAMUEL system.

Chapter 2 begins by discussing the basic components that form SAMUEL, as well as the design flow experienced by users of the system. This dissertation is then placed in context by comparing and contrasting the SAMUEL components to the efforts documented by other researchers. The chapter finishes with a discussion of advantages and disadvantages introduced into the design cycle by SAMUEL.

Chapter 3 surveys the Euler algorithmic language. Historical perspectives are given, and reasons for including Euler as a SAMUEL component are discussed. Three example Euler algorithms are provided to facilitate discussion of language semantics and syntax. The chapter concludes by discussing the version of the Euler language that has been implemented as a SAMUEL component.

Two chapters discuss the custom computing machines that are created by SAMUEL. Every custom computing machine built by SAMUEL follows a machine template that includes certain required circuits and the language opcodes that are needed to implement the algorithm. Chapter 4 begins by examining the machine template chosen for this version of SAMUEL. The chapter continues by documenting the required circuits that are needed to support any algorithm written in Euler. Finally, the opcode library and its role in SAMUEL machines is introduced.
Chapter 5 provides example custom computing machines created using SAMUEL and shows simulation results of the created machines.

Chapter 6 provides concluding remarks and proposes future work.
CHAPTER 2. THE SAMUEL ENVIRONMENT

The SAMUEL environment consists of three main components that work together to generate the final custom computing machine. The manner in which the components cooperate is called the design flow of the environment. This design flow is diagrammed in Figure 2.1. The first component is a general-purpose algorithmic language. It is in this language that the user programs algorithms that are destined for implementation as custom computing machines. As mentioned in the introductory comments, this language is not a hardware description language, and thus does not contain language constructs (such as wires, signals, or registers) that naturally support the description of hardware. Instead, in the spirit of true generality, this language is abstract and mathematical in nature. This allows the broadest set of input algorithms to be entered. A language that is syntactically simple, yet semantically rich would be ideal. For SAMUEL, an historically-early programming language called Euler was chosen.
The choice of Euler will be discussed in a separate chapter of this dissertation. Note that this choice was somewhat arbitrary. Any language with well defined behavior, such as Java or Pascal, could be used. An Euler interpreter has been created that allows users to enter algorithms and interpret them. This provides a pure software environment which can be used for algorithm debugging before actual hardware implementation.

The second SAMUEL component is the opcode library. The Euler definition completely describes a translation from Euler syntax to operation codes one level lower [11]. The behavior of each opcode has been created as a hardware object available from a library. Each opcode can then be instantiated as a data path element of the level 0 machine. This provides an architecture that directly supports Euler. By removing the additional levels that normally define the semantic gap, the resulting custom computing machine will more closely resemble the computation entered algorithmically by the user. Note that the addition of opcodes to the physical data path could be done by an engineer using a schematic design environment or by an automated environment such as SAMUEL.

The opcode library also transfers the burden of hardware design from the user to the creator of the library components. The actual digital design knowledge is known to the engineer that creates the opcode library, and not by the user. It is the engineer that must optimize the performance of the opcodes, and hence the data path. The engineer is also responsible for updating the library components as needed, since the user of SAMUEL may not possess the digital design knowledge to affect changes.
The third SAMUEL component is an analysis and generation tool that takes an Euler description and outputs a hardware description of the level 0 data path using the opcodes provided in the library. In SAMUEL, this tool is an extension (in the form of a set of functions) added to the Euler translator and interpreter. This extension executes only when requested by command line options to the Euler translator. The tool uses the translated level 0 instructions to determine which opcodes to use from the library and then generates a Verilog description of the resulting data path. This description can then become input to one of many commercially available Verilog simulation and synthesis tools. Note that these synthesis tools will create netlist descriptions of the Verilog data path. The netlist description, in turn, becomes input to the proprietary FPGA vendor placement and routing tools.

Figure 2.2 shows the SAMUEL design flow again. In this figure, the actual components replace the abstract functionality previously shown. Note that the user only interacts with SAMUEL by entering the algorithm written in Euler. The central role played by the analysis and synthesis functions is emphasized.

Figure 2.2: SAMUEL components
Why is SAMUEL Novel?

SAMUEL is novel for three main reasons. First, SAMUEL allows the user to describe a custom computing machine using a general-purpose algorithmic language. Thus, users do not need to learn a sophisticated hardware description language. The novelty exists because the original algorithmic language definition has not been modified in any way, and does not include language constructs to describe hardware.

Other attempts to use general purpose programming languages have been documented. Galloway discusses the use of a C subset, called Transmogrifier C, as a hardware description language [13]. The Transmogrifier C subset is actually quite good. However, the lack of pointer support, as well as user-defined structures, changes the nature in which most C programs are written. Additionally, many algorithms are expressed more cleanly in a recursive fashion, but Transmogrifier C does not support recursion. SAMUEL, on the other hand, does not destroy the recursive ability defined in the Euler language. The approach taken by SAMUEL also retains Euler’s lists of dynamically typed data to provide a powerful data structuring mechanism.

Transmogrifier C adds extensions to the standard C definition that enhance hardware implementations. Thus, Transmogrifier C is a hardware description language. Euler has not been modified in this way. Transmogrifier C also generates a netlist description of the combinational and sequential circuits needed to create the behavior in a Xilinx FPGA. This synthesis is different from the SAMUEL environment. SAMUEL synthesizes to a data path described using Verilog. While this requires additional
software tools to translate the Verilog to the actual FPGA netlist for placement and routing, it does provide a convenient simulation point. Additionally, it provides an intermediate hardware description that is human-readable, portable between many different vendor tools, and not targeted to one specific FPGA manufacturer.

The C++ to netlist compiler (nlc) documented by Iseli and Sanchez was also created to speed the creation of custom computing machines from general purpose languages [14]. However, as with Transmogrifier C, nlc is extended with constructs to support hardware design techniques. Also, in contrast to SAMUEL, nlc produces an output at the netlist level for Xilinx FPGAs. Additionally, the generality supported by abstract data types in C++ is not supported in nlc. Instead, the only allowed data type is the static bit-vector. SAMUEL does not restrict the user to a single data type. In addition, Euler data types can change type dynamically at runtime.

The PRISM system described by Athanas uses C as an input language [15]. While not modifying the language with specific hardware constructs, certain key features of C were not implemented. Iteration constructs and global variable dereferencing are not included. SAMUEL leaves all Euler language constructs intact, thus providing a more complete language implementation. Additionally, PRISM synthesizes Boolean expressions from the C source code, while SAMUEL synthesizes data path descriptions in Verilog.

Other sources will site the HardwareC language as an example of using a modern language, C, to compile hardware. However, the authors of HardwareC acknowledge that it is not C in many ways, and is in fact a hardware description language that is only
syntactically similar to C. The features provided to a programmer to describe hardware

circuitry in HardwareC are very different from the standard C function set [16]. Again,
this differs from the use of the original, unchanged Euler language within SAMUEL.

The Hardware Promela Compiler (HPC) is described in [17]. This compiler is

very close in spirit to SAMUEL's goal of maintaining an original language. The authors
have made very minor changes to the language to support hardware parallelism. HPC
diffsers from Euler, however, in two key ways. First, Euler is a general-purpose
algorithmic language meant to describe sequential behavior. HPC, however, is a
language that is directed to supporting interprocess communication. This changes the
types of algorithms that each system would target. Secondly, the semantic level of HPC
appears to be less than that of Euler. Promela programs read less like English text [17].
Euler programs, on the other hand, are easily read and understood without additional
thought (i.e. human translation from the written text to the proper abstraction level).

The use of C++ to describe bit-serial pipelines has been described in [18]. Here, a
library of available components is used to create FPGA circuits that compute functions
that can be naturally expressed as pipelines. The available component set is quite small.
and the goal of the work is not to produce circuits for any algorithm that can be expressed
in C++. SAMUEL, on the other hand, does provide an environment for synthesizing any
algorithm written in Euler.

Smalltalk-80 has also been used to create FPGA circuits that implement language
behavior [19]. In this work, Smalltalk-80 blocks (similar to procedures) become input to
a system that generates a corresponding logic circuit that can be routed into an FPGA.
Again, however, this differs from SAMUEL in that the language is restricted in some sense. Only a restricted subset of objects are allowed as inputs to the Smalltalk-80 blocks, and thus the full power of the language does not appear to be realized.

Another use of the C language is documented in [20]. Instead of compiling the algorithm into a single FPGA, this work focuses on using a sophisticated tool suite to analyze the dataflow graphs of a C program and appropriately schedule and map the functional units on a collection of FPGAs. While related to SAMUEL by the use of a high-level language, it still targets a different set of goals.

The BEDROC system uses Pascal as the input language [21]. The designers openly acknowledge, however, that the language has been modified with constructs that support the description of hardware. The resulting language, HardwarePal, is thus a hardware description language that becomes input to synthesis tools that focus on the generation of a net list description of the circuit. Again, this differs from the SAMUEL approach of using a virtual machine opcode library.

Many of these language-based systems, SAMUEL included, have historical roots in the large amount of high level synthesis research work that has been published. High level synthesis has, for the most part, focused effort on mapping problems given in some high level description (be it algorithmically, pictorially, etc.) into intermediate representations that through mathematical manipulation can be reduced into circuit form. Two excellent references on high level synthesis are [22,23]. Each of these references provides bibliographies with numerous papers for interested readers. Note that SAMUEL, at this point in time, is not concerned with data flow graphs, scheduling
problems, or many of the other interesting research areas covered in these references. SAMUEL has a different stopping point — a Verilog description of the data path built from opcodes that will interpret the Euler source code. Many of the issues and solutions covered in these references, however, will clearly be used by commercial synthesis tools that implement the SAMUEL Verilog output.

The second reason that SAMUEL is novel is the opcode library. Rather than synthesizing a general purpose data path, SAMUEL raises the abstraction of the level 0 machine by generating a data path using circuits that directly implement the behavior of a language construct. A complete literature review of architectural support for high-level languages is not needed to support this dissertation. However, interested readers are referred to the excellent collection of papers edited by Milutinovic [7]. An examination of recent FPGA trends in architectural support is appropriate to place SAMUEL within current methodologies.

One current trend is to implement only certain parts of a computation in the FPGA at a single time. Then, as computation proceeds, a host computer can download additional circuits (parts of the computation) to the FPGA to dynamically reconfigure the FPGA during the run-time [24,25]. This is a powerful method for increasing the throughput of time-expensive algorithms running on host computers. Run-time reconfiguration differs from SAMUEL in the sense that SAMUEL synthesizes a data path to execute the entire Euler algorithm. SAMUEL data paths are complete, and hence do not need to dynamically reconfigure.
A second current trend finds FPGA designers rapidly creating stand-alone circuits that implement part or all of an algorithm using VHDL, Verilog, or schematic entry tools. For example designs, refer to proceedings from current FPGA conferences [1,2,3].

A third reason that SAMUEL is novel is the attempt to create an environment that encourages the instruction of programming language theory at the undergraduate level. The three SAMUEL components provide tools with which many important computing concepts can be clearly demonstrated. The Euler language has many features that illustrate issues in programming language design. For example, dynamically typed data and procedures as first-class data objects. The opcode library can be used, in some sense schematically, by computer science and engineering students to create data paths manually. This reinforces the concepts of data path design and architectural support for high-level languages. Additionally, automatically generated Verilog data paths can be examined to reinforce behavioral modeling using hardware description languages. Finally, the Euler interpreter can be used as a stand-alone software environment to reinforce such basic ideas as language syntax, language design, translation, machine levels, parameter passing mechanisms, and block structure. In a keynote address, Bouldin focused on curricular areas that could be enhanced through the use of FPGAs [26]. The majority of his paper focuses on traditional digital logic instruction as well as systems-oriented courses. The paper did fail to mention the power FPGA demonstration could add to a programming language course or a digital logic course that chooses to examine language concepts.
Advantages

It can be argued that SAMUEL has some advantages over current rapid prototyping methods. First, SAMUEL eliminates schematic design. Schematic editing tools force a knowledge of digital design and lower the design abstraction (at least at the lowest levels of hierarchy). Implementing a circuit using a schematic editor also takes a significant amount of time since components must be placed, wired, and design-rule checked. If truly rapid prototyping of algorithms is the goal, then a system such as SAMUEL that automatically implements the circuit would be preferable to human schematic layout.

Second, SAMUEL dramatically reduces the computer-aided design tool learning curve. Users do not need to learn a schematic editor. Users do not need to learn sophisticated hardware description languages like VHDL or Verilog. Users do not need to learn placement and wiring techniques. Instead, users must learn a simple, general-purpose language to describe algorithms.

Third, SAMUEL does not require a background in electric circuits or digital logic design. Users enter algorithms in a natural way with no regard to the actual circuits that will implement the custom computing machine. This advantage allows a broad user base to experiment with algorithm prototyping.

Finally, SAMUEL returns to the idea of supporting a high-level language at the architectural level, but in a modular fashion using FPGAs. Since the synthesis tools only add opcodes to the data path if they are needed to complete the computation, the resulting
data paths will vary in size. Thus, in the environment of FPGAs where space is a hard design constraint, SAMUEL provides another method to attempt maximal use of available resources using a minimum of required elements.

Disadvantages

It is important to ask what disadvantages are present in SAMUEL. Let it be stated that all of the disadvantages depend upon the design constraints imposed by the user. An engineer will probably consider the loss of implementation control as a major disadvantage. Since the implementation of the data path is synthesized from the opcodes in the library (which in turn are implemented in Verilog a priori), a digital engineer will not be able to "tweak" the data path for resource use and real-estate benefits. Instead, the size of the synthesized data path must be accepted. This places an optimization burden on the designer of the opcodes.

An engineer may also find the loss of timing control to be a disadvantage. Many engineers are highly adept at using timing parameters to finesse performance from circuits, as well as to maximize the amount of work done by the circuit. Again, since control of the implementation is given to the opcode designer, the user will not have the ability to make changes.

Finally, the very nature of architectural support for a high-level language leads to circuits that are potentially larger than a general purpose data path executing a less sophisticated language. This disadvantage will make SAMUEL inappropriate for designs
where FPGAs must maximize their computing use. Instead, SAMUEL will find its most application in designs where algorithm verification is most important, but performance measures, as well as space savings are less relevant.
CHAPTER 3. EULER

A programming language called Euler was chosen for use in this rapid prototyping system. It should be noted that Euler is only one possible language for use. As the chosen language, it provides an existence proof to the fundamental premise of this dissertation: a high-level language with a well defined set of virtual machine opcodes (bytecodes) can be used to write algorithms that can be rapidly transformed into custom computing machines given a translation system and a library of circuit objects that implement the opcodes. Other languages with similarly defined bytecodes could also be chosen.

Three attributes make Euler an attractive choice. First, Euler is a language with a high semantic level. Second, Euler has a well-defined, small instruction set. Finally, the Euler language is a good compromise between languages with extraneous functions and those with minimal functional sets. In addition, the Euler functional set allows an interesting mix of pedagogical topics to be explored. This chapter will provide a brief justification for the use of Euler, and then give example programs in order to familiarize the reader with Euler syntax and its translation to intermediate opcodes.

Language Perspectives

Euler is a general purpose, algorithmic language defined by Niklaus Wirth [10,11,12]. It is an ALGOL-60 mutant described during a time period when language
engineering was still in its infancy. At that time, many programming language theorists were debating the advantages and disadvantages of the two dominant languages, FORTRAN and ALGOL, and were proposing modifications (mutations) that would enhance programmer usability [27]. Wirth generalized ALGOL to produce a language with many interesting modifications. His motivation was, in his words, a "yearning for simplicity" supported by a paper by A. van Wijngaarden that professed the then current languages as too complex [28]. Thus, Euler is ALGOL-like, but quite different. Variables are untyped, procedures are elevated to first-class data objects, the address reference is introduced into the language, and iteration structures are minimized. Interested readers may find a description of ALGOL-60 useful for comparison [29].

It should be noted that Euler is the first language published by Wirth [28]. From an historical perspective, Wirth is now considered one of the fathers of modern programming languages. Many of the ideas he explored developing Euler, and his more famous languages Pascal and Modula, shaped the way programming languages matured. Indeed, Wirth intimated the concepts of strongly typed variables, structured programming, and program modules into Pascal and Modula, producing languages that were extensively used throughout the 1970s and 1980s. Modern languages, such as Ada and Java, have continued to refine these concepts.

Ironically, some of the language characteristics that would be called "in the spirit of Niklaus Wirth languages", are not present in the Euler definition. For example, it is commonly believed in modern language design that the use of unconditional branches is poor programming style, and that strongly typed languages offer the best method for
formal program verification [9,30,31]. Wirth is a strong supporter of highly typed languages, yet, Euler does not restrict variables to be statically typed. Also, the unconditional branch (goto) statement is a predominant language feature in Euler. In fact, since the iterative constructs for and while are not defined in Euler, unconditional branches must be used to implement them! Additionally, the procedure is a first class data structure that can be assigned to a variable. Thus, unexpected program side-effects may unintentionally modify a procedure definition by overwriting the variable to which the procedure is assigned. This results in programs that can be difficult to debug, the anti-thesis of structured programming. Wirth discusses these pitfalls, as well as others, in a paper that attempts to document how his beliefs in language theory evolved from his work on Euler and Algol mutants [28].

However, the Euler advantages discussed in the opening paragraph of this chapter are more important to SAMUEL than these disadvantages. First, Euler is defined at a very high semantic level. In fact, the semantic level of Euler could be argued as higher than that of the currently popular C programming language [32]. Euler allows variables to be dynamically typed, while C enforces statically typed variables. Euler includes call-by-name as a parameter passing mechanism, while C does not. Euler allows procedures to be dynamically redefine, dynamically renamed, and passed as parameters while C enforces static definition of functions and procedures. Euler's block structure mechanism is robust, allowing hierarchical name spaces. The block structure mechanism of C is not as robust. The C language restricts procedures and functions to those defined within the
same name space as the main program. All Euler blocks return values, generalizing the return mechanism, while C allows blocks to be defined that do not return values.

Secondly, Euler is well documented and small. A Backus-Naur form grammar definition is provided, as well as a complete description of opcode behavior [10,11,12]. These papers discuss key language features in good detail. Little guesswork is left when implementing the language. The only missing definition is input and output behavior. These missing definitions can be viewed as merely a language allowance that implementations will be built on many platforms — each with unique I/O capabilities. A few errors were present in the initial publication, and corrections have been published [12].

The language syntax is quite small. There are only seventy-five basic symbols, of which many are simply numbers and mathematical operators. The learning curve for most users (especially the users with previous programming experience) should be minimal. The virtual execution engine expected by the grammar consists of a basic push-pop stack used to support the results of the fifty-six executable opcodes. Results are held on the stack until absorbed as input to an opcode.

The third advantage considered when choosing Euler for SAMUEL was the pedagogical ideas supported by the language constructs. As a good compromise between languages that are overloaded with functions and those with small functional sets, Euler is rich in language theoretic abstractions. Dynamically typed variables provide an interesting programming paradigm for students more familiar with the languages of the 1990's. The ability to dynamically assign any data object to a variable provides an unique
opportunity for students to compare and contrast the use of statically typed languages versus dynamically typed languages. Students can explore ideas such as storage methodologies, type checking, and type casting.

Euler supports many parameter passing mechanisms that also make it attractive as an instructional language. Call-by-value, call-by-reference, as well as call-by-name are all provided in the language syntax. Students can explore the opcodes that result when each parameter passing mechanism is used. The storage space needed to store the opcodes can be compared. The side-effects of assignment to passed variables can be analyzed.

The ability to dynamically change procedures is also a striking paradigm. Some functional languages, such as Lisp, do retain the procedure as a first class data object, and thus have mechanisms in place to support dynamic redefinition [33]. These mechanisms are finding interesting uses in genetic programming (the use of evolutionary techniques to evolve programs to solve problems more optimally) [34]. Most modern declarative languages, such as C, C++, or Ada, statically define procedures that exist throughout the life of the program. These declarative languages are the ones most often taught in universities [35]. Thus, many students could benefit from an environment that would let them explore the uniqueness of a non-static procedure space, as well as the difficulties that dynamically changeable procedures introduce into program correctness and error tracking.

All of these language-theoretic abstractions help to define a set of teaching topics that can be supported nicely by Euler. Note, as well, that these same abstractions
introduce hardware complexity into the level 0 data path. Thus, in addition to the software education that is enhanced by Euler, the hardware education can also be greatly expanded. Students can be instructed in the creation of systems that support dynamic data types, block structure, and result stacks for example. Additionally, since the primary goal of this work is to create a rapid prototyping environment that will automatically use given opcode circuits to synthesize a level 0 data path, students can integrate the concepts of data path design with language support. For example, they can compare and contrast the resulting data paths when certain parameter passing mechanisms are used. They can ask many questions. Will the data path be any more complex if it needs to support call-by-name? Will the data path need extra opcodes to support recursion when compared to an iterative version of the same algorithm? How does the lack of a for/while iteration construct affect the complexity of the data paths that result?

These comments are intended to provide insight into the Euler language. However, Euler is merely the language chosen for SAMUEL, and thus this dissertation will not attempt to fully describe all the nuances of it. Instead, the reader is referred [10,11,12].

Example: A Shift Cipher

A shift cipher algorithm can be used to provide a first glimpse at the syntax of Euler. Shift ciphers, while not providing a great deal of security, have application in computer cryptography [36]. The shift cipher algorithm shown in Figure 3.1 requests a
shift value from the user and then outputs the message string after shifting each character of the string by the shift value. Important points illustrated in this example that should be discussed include blocks, variable declarations, the generalized data structure, and branch constructs.

In Euler blocks are delimited with the `begin` and `end` language constructs. Note that each statement within a block is terminated with a semicolon. The final block statement is not terminated with a semicolon. Each semicolon corresponds to a removal of the statement "return value" from a runtime stack. Thus, the final statement does not have a semicolon so that the return value of the block will be available on the runtime stack for use by statements in the dynamically-enclosing block. The dynamically-enclosing block is the block that executed a statement that called the terminating block.
This may differ at runtime from the statically enclosing block, which is the block in which the terminating block was defined.

Euler supports unique name spaces for each delimited block. This implies that the Euler interpretation mechanism must be able to find the appropriate variable during variable dereferencing. The Euler language includes a support structure for each delimited block. This structure is called a mark. Marks are created and pushed onto the runtime stack whenever block execution is initiated. The index of the runtime stack corresponding to the current mark is held in a mark pointer. Each mark consists of six fields and contains the information shown in Table 3.1. Note that the fields marked with an asterisk were added in the Euler implementation created for SAMUEL (Euler-97). It should be stated that the Euler definition describes the use of the mark, as well as a set of mark access functions. It is these access functions that allow the mark to be modified to fit implementation needs. For example, the original definition has the mark storing all the variables as a list. The Euler-97 implementation does not store the list in the mark. Instead, the mark contains a pointer to the start of the list in the data memory. Note that

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>block number</td>
<td>a numerical identifier for this block</td>
</tr>
<tr>
<td>dynamic link</td>
<td>the block which started this block executing</td>
</tr>
<tr>
<td>static link</td>
<td>the block in which this block is defined</td>
</tr>
<tr>
<td>variable list start</td>
<td>the address at which the variables for the block</td>
</tr>
<tr>
<td>number of variables</td>
<td>the number of variables defined within this block</td>
</tr>
<tr>
<td>return address</td>
<td>resume execution at this address upon block exit</td>
</tr>
</tbody>
</table>
others have expanded or abandoned the mark in other Euler implementations. For example, see [37,38].

The shift cipher algorithm has only one non-recursive delimited block. Thus, only one mark will be created during runtime and pushed onto the runtime stack. The mark will be (1,0,0,0,3.0). The field values imply block number 1 was not called by any other block, was not defined within any other block, has three variables stored in the memory beginning at varstart, and will not return to a dynamically enclosing block.

Turning now to variable declarations, the shift cipher algorithm demonstrates that variables are declared within a block by the new symbol. Since Euler is dynamically typed, no characteristics are attached to the variable at declaration and no storage size information can be inferred. Each new encountered will cause a corresponding intermediate opcode at runtime to allocate an initial storage location (which can grow as needed) and increment the recorded number of variables on the mark for that block. An Euler variable can dynamically take on a value that is a number, boolean constant, program label, list, procedure, symbol, variable reference, or a symbol representing that the variable currently is undefined.

Next, the shift cipher algorithm also illustrates the generalized data structure available in Euler. This structure is a list of dynamically typed data objects. In the shift cipher algorithm, a list is created by the assignment (:=) of the collection of numbers to the variable called MessageString. A second list constructor is provided by the list symbol, which would allow a statement such as “MessageString := list 5” to be written. This statement creates a list of five elements of type undefined. Member access is
demonstrated in the algorithm as well. Note that members are accessed by indexing the list with a bracketed index value. Since a member of a list can be another list, multiple indexes can be used to access elements of the nested list. For example, if a list called A is defined as ((1 2 3) 4 5), then A[2] would return 4, while A[1][3] would return 3. Note that indices begin at one in Euler. A length symbol is also provided to return the current size of the list. Other list symbols are defined as well. Euler-97 currently does not allow the concatenation behavior defined by Wirth. This is a major limitation to list behavior and should be a high priority for future enhancement to Euler-97.

A final concept illustrated by the shift cipher algorithm is the unfortunate reliance on the unconditional branch. Note that the use of goto requires a point in the program that is labeled. This introduces a label symbol, and a corresponding label data type into Euler. The use of unconditional branches is generally considered poor programming style, and can lead to code that is not easily debugged or verifiable [28]. However, as the only non-procedural branch construct, goto plays an extremely large role in Euler and cannot be avoided. More structured language constructs, such as for loops and while loops, are not defined in Euler. They can, of course, be written as Euler procedures that use goto, but their absence in the Euler language is a negative.

Finally, the translation system defined by Wirth for the Euler language will produce the set of intermediate opcodes shown in Figure 3.2. Note that the twenty-one lines of the algorithm result in sixty-three intermediate opcodes. It is these intermediate opcodes that are implemented as library components (circuits) in SAMUEL. Note two operands may be present with an instruction in the program string.
<table>
<thead>
<tr>
<th>SourceLine</th>
<th>Opcode</th>
<th>Operand1</th>
<th>Operand2</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td><code>ebegin</code></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td><code>enew</code></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td><code>enew</code></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td><code>enew</code></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td><code>eref</code></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td><code>enumber</code></td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td><code>enumber</code></td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td><code>enumber</code></td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td><code>enumber</code></td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td><code>enumber</code></td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td><code>enumber</code></td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td><code>enumber</code></td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td><code>enumber</code></td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td><code>enumber</code></td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td><code>enumber</code></td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td><code>enumber</code></td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td><code>enumber</code></td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td><code>enumber</code></td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td><code>enumber</code></td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td><code>enumber</code></td>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td><code>enumber</code></td>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td><code>esparen</code></td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td><code>eassign</code></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td><code>esemi</code></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td><code>eref</code></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td><code>ein</code></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td><code>eassign</code></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td><code>esemi</code></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td><code>eref</code></td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td><code>enumber</code></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td><code>eassign</code></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td><code>esemi</code></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td><code>eref</code></td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td><code>eref</code></td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td><code>esemi</code></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td><code>evalue</code></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td><code>enumber</code></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td><code>eadd</code></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td><code>eassign</code></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td><code>esemi</code></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td><code>eref</code></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td><code>eref</code></td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td><code>evalue</code></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td><code>erbrak</code></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td><code>evalue</code></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td><code>eref</code></td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td><code>evalue</code></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td><code>eadd</code></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td><code>enumber</code></td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td><code>emod</code></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td><code>ewut</code></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td><code>esemi</code></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td><code>eref</code></td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td><code>evalue</code></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td><code>eref</code></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td><code>elength</code></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td><code>elthan</code></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td><code>etran</code></td>
<td>62</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td><code>elabel</code></td>
<td>33</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td><code>evalue</code></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td><code>egoco</code></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td><code>else</code></td>
<td>63</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td><code>enumber</code></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td><code>eend</code></td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 3.2: Intermediate opcodes for the shift cipher algorithm
Example: Insertion Sort

Figure 3.3 is an Euler implementation of an insertion sort algorithm for sorting a list of numbers. For a reference on sorting algorithms, see the excellent text by Corbin, Leiserson, and Rivest [39]. The procedures named InsertionSort, ListOut, EnterList, and while are shown in Figure 3.4. Many more concepts of the Euler language are illustrated

begin
  new InsertionSort;
  new ListOut;
  new EnterList;
  new ListForSorting;
  new while;

  /* procedures EnterList, InsertionSort, ListOut, and while */
  /* are defined here, but will be shown in figure 3.4 due to */
  /* space limitations */
  EnterList(@ListForSorting);
  InsertionSort(@ListForSorting);
  ListOut(@ListForSorting)
end

Figure 3.3: Insertion sort

by the insertion sort algorithm. First, refer to Figure 3.3 and note that the variable named ListForSorting is preceeded by the symbol @ in the call to each procedure. This is the memory reference operator, and is used to pass the address of variable ListForSorting to each procedure. This style of parameter passing is called call-by-reference, and is especially useful in this algorithm to avoid copying long lists as parameters. Instead, the procedure receives the starting memory location of the list as the parameter, and modifies the values of each list element directly through further dereferencing. The changes are clearly visible to the outer block since the procedures will modify the original list.
Next, the procedure definitions shown in Figure 3.4 demonstrate that procedures are assigned to variables using the assignment symbol := in the same manner that a number or any other data type would be assigned. The procedure is delimited by the symbols ` and `. Note that these delimiters do define a new name space, and thus will

```
InsertionSort := 'formal inList;
begin
  new i; new j; new key;
  j := 2;
  while('j <= length inList',
  begin
    key := inList[j];
    i := i - 1;
    while('i>0 and inList[i] > key',
    begin
      inList[i-1] := inList[i];
      i := i - 1
    end');
    inList[i-1] := key;
    j := j - 1
  end');
end';

while := formal Conditional; formal CodeSegment;
begin
  label WhileLoop;
  WhileLoop: if Conditional then
    begin
      CodeSegment;
      goto WhileLoop
    end;
  else 0
end';

ListOut := 'formal inList;
begin
  new index;
  label ListOutLoop;
  index:=0;
  ListOutLoop: index := index - 1;
  out inList[index];
  if index < length inList then goto ListOutLoop else 0
end';

EnterList := 'formal inList;
begin
  new index;
  index := 1;
inList := list in;
  while('index <= length inList',
  'begin
    inList[index] := in;
    index := index + 1
  end');
  length inList
end';
```

Figure 3.4: Insertion sort procedures
have a corresponding mark on the runtime stack while the procedure is executing. Also note the formal parameter declarations. These formal parameters are used to transmit information between the blocks. It should be noted, however, that since Euler is designed to allow inner blocks to dereference and access a variable in an outer block, parameter passing is not explicitly needed in these examples. In fact, the variable ListForSorting could have been used directly. However, the parameters in the procedure of Figure 3.4 allow them to be used on any list, regardless of name.

The procedure named while also illustrates the call-by-procedure parameter passing mechanism (also known as call-by-name). Procedures to be passed as parameters are enclosed in procedure delimiters, and can be executed within the procedure to which they are passed.

Example: Recursive Factorial

It is sometimes easier to express functions using recursion. Euler supports recursion whose depth is limited only by the size of the runtime stack. Figure 3.5 is a recursive factorial algorithm written in Euler. Note that since a single statement can be used to describe the factorial behavior, the begin and end delimiters were not needed within the procedure. Abstractly, when a program is executed a mark will be stacked that corresponds to the outermost block. This will be block number one. User input is sampled and passed to the factorial procedure. The recursion will continue to stack marks with block numbers 2, 3, 4, ..., n+1 before the recursion terminates. As each
recursive call completes, the corresponding \_ symbol will remove a mark from the stack. This process continues until the final recursive result is available.

**Euler-97**

The version of Euler implemented for SAMUEL was written in ANSI C on a Hewlett-Packard workstation. It was compiled with the gcc compiler, and debugged using the gdb debugger [40]. The grammar was translated from BNF to C using the yacc tool [41]. A lexical analyzer was translated from the Flex input language to C using the Flex tool [41]. Approximately 5000 lines of C code are compiled to form the final Euler translator, interpreter, and data path generation tool. The code is portable, and has been recompiled on a CyberMax (IBM PC compatible) Pentium™ system running the Windows 95™ operating system using the Borland C compiler.
Euler-97 does not implement the entire language specified by Wirth. Only integer mathematics has been implemented. It was believed that integer mathematics was sufficient to demonstrate the concepts of SAMUEL. The floating point mathematical operators can easily be finished as future work. Next, the concatenation list operator has not been implemented. This is a major oversight that should be corrected as soon as possible so that the list structure can be utilized at its full potential. Finally, only very basic behaviors for the \texttt{in} and \texttt{out} symbols have been created. Much more sophisticated input and output behaviors could be implemented if desired.

Table 3.2 lists the various command line options and the associated actions. Note that multiple command line options can be specified on the same command line. Some explanation is probably in order. The big tables option increases the size of the symbol and translated program tables generated by the translator. The table size increases from 500 to 10000 entries. This table size should be more than is ever needed for most

<table>
<thead>
<tr>
<th>Command line</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>euler &lt;fname&gt;</td>
<td>translate and interpret &lt;fname&gt;</td>
</tr>
<tr>
<td>euler -b &lt;fname&gt;</td>
<td>translate and interpret &lt;fname&gt; using big tables</td>
</tr>
<tr>
<td>euler -c &lt;fname&gt;</td>
<td>translate (syntax check) &lt;fname&gt; only</td>
</tr>
<tr>
<td>euler -d &lt;fname&gt;</td>
<td>translate and interpret &lt;fname&gt; in a debugging mode</td>
</tr>
<tr>
<td>euler -o &lt;fname&gt;</td>
<td>translate and interpret &lt;fname&gt;, \texttt{out} directed to &lt;fname.out&gt;</td>
</tr>
<tr>
<td>euler -r &lt;fname&gt;</td>
<td>translate and interpret &lt;fname&gt;, report machine state</td>
</tr>
<tr>
<td>euler -synth &lt;fname&gt;</td>
<td>translate and synthesize data path for &lt;fname&gt;</td>
</tr>
<tr>
<td>euler -synth -sim</td>
<td>translate and synthesize, including simulation extras</td>
</tr>
<tr>
<td>euler -t &lt;filename&gt;</td>
<td>translate and interpret &lt;filename&gt; and create translated listing</td>
</tr>
</tbody>
</table>
programs written in Euler.

The debugging mode causes the interpreter to stop before every goto symbol and present a menu of choices to the user. The user can choose to examine the stack, the variable values, execute the next instruction, or continue the program with no further stops. This primitive debugger provides a little insight into the virtual machine while a program is executing, but is not very sophisticated. The skeleton is in place, but many of the debugging features are not yet implemented. Future work could enhance the functionality of this mode of operation.

The output redirection option causes the interpreter to route all data outputs to a file called <fname.out>. This provides a convenient way to collect the results of program runs without enhancing the language with sophisticated file I/O routines.

The report option causes the interpreter to output machine state information to the user display before executing the next instruction. The machine state, in this context, is the instruction about to be executed, the contents of the stack, and the variable values. This provides a mechanism to view the machine state without pausing for user input at a debugger prompt.

The synthesis option causes the translator to synthesize a data path description file using the intermediate opcode library. The data path will be described in a file called <filename.synth>. Currently, a Verilog description is produced by default. Note that this synthesis task is not difficult. It is a translation from the set of intermediate opcodes needed for the algorithm to a set of module instantiations in Verilog. Each intermediate opcode has been created as a Verilog module, and thus the port definitions are known.
and recorded in the library file. The translation functions reference this library file to
correctly interconnect the modules to the data path.

The second synthesis option adds a simulation flag (-sim). When these two flags
are used together, SAMUEL produces a Verilog data path that is ready for simulation by a
Verilog simulator. A Verilog test bench is included, with a system clock defined, and
simulation models of a stack and instruction ROM ported to the data path. In contrast,
without the -sim option, the data path is ready for synthesis. The testbench is not
included, and the stack and instruction ROMs are assumed to be stand-alone ICs, and thus
are not added to the data path.

Finally, the translated listing option instructs the translator to produce a file called
<fname.cmp> which contains the program in bytecode form.
CHAPTER 4. THE SAMUEL MACHINE TEMPLATE

Every algorithm programmed in the Euler language will be transformed into a custom computing machine that is built according to the machine template shown in Figure 4.1. Boldfaced lines represent information busses, either data or instruction. A bus is a group of signals collected together because they carry information that is related in abstraction. The number of signals in each bus (the bus width) is represented in the figure by hashing the bus and noting the size. Non-boldfaced lines represent address busses feeding the instruction and data storage memories. It is important to note that the bus widths for these address busses are not shown in this figure. This template can be represented completely in set notation by the 4-tuple $M = \{R, O, d, i\}$, where capitalized letters represent sets and non-capitalized letters represent integer variables.
The first element of the four-tuple is the set of circuits, $R$, that is required in every custom computing machine regardless of the algorithm under consideration. Since every custom computing machine requires the circuits from set $R$, the cardinality of $R$ is fixed. Certain properties of the Euler language and its virtual environment force the inclusion of these circuit elements. For example, every Euler algorithm will have at least one block delimited by `begin` and `end`. Thus, every Euler algorithm will require support for the mark structure. The elements of set $R$ will be discussed later.

The second element of the four-tuple is the set, $O$, of opcode circuits. This set depends on the algorithm under consideration and is the set of opcodes that result from the translation process from the algorithm written in Euler to the intermediate opcodes. For example, consider the recursive factorial algorithm first given in Figure 3.5 and repeated in this chapter as Figure 4.2. This program requires 19 basic language symbols to describe its behavior. After translation to the level of the Euler virtual machine, 20

```euler
#**************
#* fact.euler *
#* a factorial *
#* test of *
#* recursion *
#**************
begin
  new factorial;
  factorial := 'formal n;
  if n <= 1 then 1 else n * factorial(n-1)';
  out factorial(in);
  out factorial(in)
end
```

Figure 4.2: Recursive factorial
machine opcodes result. Table 4.1 lists the basic symbols, as well as the translated opcodes. Note that many of the basic symbols have a one-to-one correspondence with a translated opcode. The semantic gap between the language and virtual machine is small.

Table 4.1: Basic symbols and translated opcodes for recursive factorial

<table>
<thead>
<tr>
<th>Basic Symbol</th>
<th>Basic Symbol</th>
<th>Opcodes</th>
<th>Opcodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</td>
<td>new</td>
<td>ebegin</td>
<td>enew</td>
</tr>
<tr>
<td>;</td>
<td>:=</td>
<td>esemi</td>
<td>eassign</td>
</tr>
<tr>
<td>`</td>
<td>formal</td>
<td>elquote</td>
<td>eformal</td>
</tr>
<tr>
<td>if</td>
<td>&lt;=</td>
<td>elethan</td>
<td>ethen</td>
</tr>
<tr>
<td>then</td>
<td>else</td>
<td>eelse</td>
<td>emul</td>
</tr>
<tr>
<td>*</td>
<td></td>
<td>esub</td>
<td>erparen</td>
</tr>
<tr>
<td>-</td>
<td>)</td>
<td>erquote</td>
<td>eout</td>
</tr>
<tr>
<td>`</td>
<td>out</td>
<td>ein</td>
<td>eend</td>
</tr>
<tr>
<td>in</td>
<td>end</td>
<td>enumber</td>
<td>eref</td>
</tr>
<tr>
<td>l</td>
<td></td>
<td>ecall</td>
<td>evalue</td>
</tr>
</tbody>
</table>

Now, compare these numbers with the numbers for the insertion sort algorithm shown in Figures 3.3 and 3.4 (these figures will not be repeated in this chapter). The insertion sort algorithm is written with 31 basic symbols and translates to 28 opcodes. The small semantic gap that exists between basic symbols and opcodes keeps the numbers close. Notice, however, that the number of opcodes required to implement insertion sort is eight greater than the number required to implement recursive factorial. Thus, the cardinality of set O depends on the algorithm.

The variation in the cardinality of O is a property that SAMUEL exploits to minimize the use of available circuit real-estate. Rather than building a machine that supports all fifty-six of the opcodes needed to support the Euler language, SAMUEL will
generate a machine that contains just the \textit{required} opcode circuits to complete the
algorithm. Let each opcode be characterized in terms of the number of logic gates and 1-
bit registers needed to implement it as a logic circuit. Call this parameter, $O_{GF}$. Let $\mu$
represent the average of $O_{GF}$ over all fifty-seven opcodes:

$$\mu = \frac{\sum O_{GF}}{57}$$

Then, an average real-estate parameter for any machine generated by SAMUEL which
requires $n$ Euler intermediate opcodes is

$$M_{GF} = \mu \times n$$

and the corresponding average real-estate saved is:

$$S_{GF} = \sum O_{GF} - M_{GF} = \mu (57 - n)$$

The recursive factorial machine generated by SAMUEL, with $n = 20$, will thus use $37\mu$
less average real-estate than a machine built to execute \textit{all} Euler opcodes. The recursive
factorial machine has lost the generality of a complete Euler machine, but it is only a
small percentage of the (averaged) size.

The third element of the 4-tuple is an integer constant, $d$, representing the width of
the shared data bus in bits. The default data bus width of all machines created by
SAMUEL is currently fixed at 48 bits. These 48 bits are divided into a data type flag and
data. The most significant byte stores the data type flag.

The final element of the 4-tuple is an integer constant, $i$, representing the width of
the instruction bus. The default instruction bus width of all machines created by
SAMUEL is currently fixed at 40 bits. The lower 32-bits hold the first and second
operands as 16-bit quantities, while the upper byte stores the opcode. Since an instruction flows on a different bus than the data, SAMUEL generates architectures that avoid the von Neumann bottleneck [8]. However, SAMUEL does not currently create machines that pre-fetch the next instruction while executing the current instruction.

The default 4-tuple for all machines that are currently generated by SAMUEL is $M = \{ R, 0.48, 40 \}$. Note, however, that it would be straightforward to allow the bus widths to be passed as command line parameters to the Euler-97 compiler. Allowed bus width parameters could be 16 bits, 24 bits, 32 bits, and 48 bits, for example. Of course, the compiler would also need to truncate numerical constants to the appropriate field width given the bus width. Additionally, certain aspects of the Verilog opcode implementation depend upon the bus width, and data field positions within data values. These dependencies are currently not completely written as parameters within the Verilog modules. Thus, some additional work will need to be done to the Verilog opcode library to support user-defined bus widths.

**The Set of Required Circuits**

The Euler virtual machine described in the original papers contain certain state information and storage areas that are required for the execution of all programs written in the language. The state information includes the instruction address register, the stack pointer, the mark pointer, and a counter for formal procedure parameters. The storage
requirements include an area for a stack, an area for the program string, as well as variable storage [10,11,12].

SAMUEL creates custom computing machines that are, effectively, subsets of the original virtual machine. Thus, the state information and storage areas will be required within the required set, $R$, of a machine generated by SAMUEL. Therefore, $R$ includes the instruction address register, stack pointer, formal counter, stack, instruction memory and variable storage. However, $R$ also includes a stack address multiplexer, input data register, output data register, variable storage address register, the mark register, the begin opcode, and the end opcode. Note that the mark pointer of the original description has been replaced by the mark register, which holds a copy of the current block mark and mark pointer.

A general-purpose SAMUEL machine register (mreg) is used to implement the instruction address register, stack pointer, mark register, variable storage address register, and formal counter. The mreg Verilog description is given in Appendix B. Figure 4.2 shows the mreg with control signals and associated behavior.

![Figure 4.2: The SAMUEL machine register](image-url)
The only part of the state information that is not required in all Euler programs is the formal procedure parameter counter (FPPC). The Euler-97 compiler could be programmed to remove the FPPC from the custom computing machine if no Euler procedures with formal parameters are written and called in the algorithm. The FPPC is still included within the set R, however, since it cannot be characterized as a language opcode, and thus does not fit within the set O.

To begin study of the circuits of set R, consider Figure 4.3. The program string is the sequence of opcodes and numerical constants that make up the algorithm to be executed. This program string is created by the Euler-97 compiler, and is the result of translating an algorithm written in Euler down one level to the virtual machine. SAMUEL generates custom computing machines under the assumption that this program string has been stored in an instruction read-only memory (IMEM) that can be accessed.

Figure 4.3: The instruction address register and associated instruction memory
through the required instruction address register (IADR). It is the responsibility of the user to transfer the program created by the Euler-97 compiler into the memory, and to port the memory to the instruction address register of the custom computing machine. If the custom computing machine is implemented in an FPGA, the porting task will be completed through input pins for the instruction bus, and output pins from the instruction address register. If the custom computing machine is intended for simulation only, then an appropriate simulation model of the memory must be ported before simulation. The register control signals (sliadr, s0iadr, and reset) are also shown in the figure.

The stack, stack pointer (sp), and stack multiplexor (smux) are members of the required set, R. These circuits are shown in Figure 4.4. The stack takes input from the shared data bus, and drives output to the shared data bus. The stack should thus be ported

![Figure 4.4: The stack and associated circuitry](image-url)
to the data bus as well as the stack address bus (sabus). In an FPGA implementation, the stack would be a stand-alone integrated circuit again ported through pins on the FPGA.

The stack pointer value is manipulated by the opcode circuits. Opcodes either increment, decrement, or load the register through encodings on the slsp and s0sp control signals. SAMUEL generates machines under the assumption that the stack pointer is pointing to the value stored at the top of the stack, and not to the next empty space. A stack pop is achieved in two states of a state machine. Thus, a pop begins with a read of the value pointed to by the stack pointer. The second state decrements the stack pointer. Similarly, a stack push is a state machine in which the stack pointer is incremented, and the data is written to the stack. The only other behavior performed by the opcode circuits is the loading of the stack pointer with the current mark pointer value when a block is terminating. Thus, the mark pointer value is the data input of the stack pointer.

The smux1 and smux0 control signals are used by the opcode circuits to select which input bus is passed as the address to the stack as shown. In most accesses, the smux will be set to pass the stack pointer as the address. Certain opcodes will require the ability to randomly access the stack contents at the mark dynamic link value, mark static link value, and the mark pointer. For example, the eoref opcode will use the static links of each mark to chain backward through the marks to find specific variables in enclosing name spaces. Another example opcode is eend, which will use the dynamic link to restore the block that called the currently executing block.

The mark register is a required circuit, and stores a copy of the current mark. Recall from Chapter 3 that Euler supports name spaces by storing information about the
space in the mark data structure. In the Euler papers, the mark is pushed onto the stack whenever a name space is dynamically created during program execution. SAMUEL has been designed to create machines that not only push the mark onto the stack, but also store a local copy in the mark register. This register does use chip real-estate, but also simplifies access to the current mark by avoiding random access read cycles of the mark from the stack during runtime. Figure 4.5 illustrates the mark register. Note that the mark register drives output to a dedicated bus called the mark bus, and not to the shared data bus. This mark bus is used as an input by many of the opcode circuits. As shown previously, fields of the mark bus, such as the dynamic link, are used as input to the stack multiplexer.

Table 4.2 show the positional relationships of the mark data structure fields within the mark register. Refer to Chapter 3 for a discussion of each field. Note that the least significant byte of the mark register stores the mark pointer. The mark pointer is the address in the stack where the current mark is also stored.
Table 4.2: The mark data structure fields within the mark register

<table>
<thead>
<tr>
<th>mark register bits</th>
<th>mark data structure field</th>
</tr>
</thead>
<tbody>
<tr>
<td>47:40</td>
<td>block number</td>
</tr>
<tr>
<td>39:32</td>
<td>dynamic link</td>
</tr>
<tr>
<td>31:24</td>
<td>static link</td>
</tr>
<tr>
<td>23:16</td>
<td>varend</td>
</tr>
<tr>
<td>15:8</td>
<td>return address</td>
</tr>
<tr>
<td>7:0</td>
<td>mark pointer</td>
</tr>
</tbody>
</table>

Two registers are always included in SAMUEL-generated machines to handle user input and machine output. The input register and the output register are instantiations of the general SAMUEL machine register, and should be ported appropriately to pins of an FPGA at implementation. Figure 4.6 diagrams the registers with control bus signals shown. Note that the control signals of the input register are tied to the logic levels to cause a load on every clock cycle. It is isolated from the data bus by a three-state buffer. The output register control signal \( ldor \) is used to request loading of

![Figure 4.6: The input and output registers](image)
the register from the data bus. The Euler-97 compiler could scan the program string for \texttt{ein} and \texttt{eout} opcodes, and base the inclusion of the input register and output register on the presence of these opcodes. The current version of the compiler, however, always includes the registers since an FPGA without them would not be very useful since no information could be exchanged with the user.

The data memory and data memory address register are members of set R, and are included to provide storage for list elements, as well as parameters passed to procedures. Again, it is assumed that a random access memory will be ported to the machine in actual implementations, or a simulation model for simulation. Figure 4.7 shows the interface details.

The final two elements of set R are the \texttt{ebegin} and \texttt{eend} opcode circuits. Note that these opcodes are required in any SAMUEL-generated circuit since the Euler language definition requires at least one name space. Figure 4.8 shows both the \texttt{ebegin} and \texttt{eend}

![Figure 4.7: The data memory circuitry](image)
opcode circuits with their inputs and outputs ports.

Some basic principles of all opcode circuits can help facilitate the rest of this discussion. First, all of the SAMUEL opcode circuits are built as finite state machines that create control bus (cbus) signals as outputs. Table 4.3 lists the control bus signals.

Table 4.3: The control bus signals

<table>
<thead>
<tr>
<th>Signal</th>
<th>Signal</th>
<th>Signal</th>
<th>Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit 16: error</td>
<td>bit 11: mcs</td>
<td>bit 6: smux0</td>
<td>bit 1: drir</td>
</tr>
<tr>
<td>bit 15: s1iadr</td>
<td>bit 10: mwe</td>
<td>bit 5: scs</td>
<td>bit 0: ldor</td>
</tr>
<tr>
<td>bit 14: s0iadr</td>
<td>bit 9: s1sp</td>
<td>bit 4: swe</td>
<td></td>
</tr>
<tr>
<td>bit 13: s1madr</td>
<td>bit 8: s0sp</td>
<td>bit 3: s1mp</td>
<td></td>
</tr>
<tr>
<td>bit 12: s0madr</td>
<td>bit 7: smuxl</td>
<td>bit 2: s0mp</td>
<td></td>
</tr>
</tbody>
</table>

Single control signals are asserted low, while encodings such as \{s1iadr,s0iadr\} do not have assertion. Seventeen control signals are needed for the custom computing machines currently created by SAMUEL.

All SAMUEL opcode circuits are ported to the instruction bus. The classical fetch-decode-execute cycle, which is usually coordinated by a controller, is eliminated in SAMUEL-generated custom computing machines by asynchronous opcode snooping.

Figure 4.8: The \texttt{ebegin} and \texttt{eend} opcode port information
Each SAMUEL opcode remains in an orbit state monitoring the instruction bus for that opcode's presence in the instruction stream. When the instruction on the instruction bus matches, the opcode executes its state machine and updates the instruction address register upon completion.

The shared data bus is ported to opcode circuits that need to create data for stacking, or to send data to one of the circuits from set R. Since the data is dynamically typed, the opcode circuit adds a type flag to any data value it drives onto the shared data bus. It is important to note that some opcodes do not create output for the data bus. For example, the ethen and eelse opcodes only modify the instruction address register through control bus signals, and do not drive data to the data bus.

Each opcode circuit is ported to the system clock and the global reset signal. The system clock is used by an opcode circuit as an edge trigger for transitions in state machines, as well as register latching. The reset signal is not asynchronous. Rather, it is positive edge-triggered, and causes an opcode circuit to clear any internal storage registers, as well as to return to the orbit state.

Recall that SAMUEL generates a Verilog description of the machine. This description can then be input to other tools to actually perform synthesis. Thus, a certain style of Verilog description has been used to provide for the best portability across tools. For example, output values are never specified in more than one always block. Also, state transitions are only defined within one always block. Additionally, default values are added to all case statements even if all binary possibilities have already been covered. In most opcodes, multiple assign statements are used to create the bit fields of larger
outputs, rather than using the concatenation operator. Outputs are not aliased by registers. Rather, outputs are *assigned* register values using continuous assignment statements. The edge-triggered reset is included in all states, rather than as an *if* statement followed by an *else* clause containing the state transitions. Bus widths are specified rather than left as implied.

With these basic principles in mind, Figure 4.9 gives the Verilog description of the *ebegin* opcode and Figure 4.10 gives the Verilog description of the *eend* opcode. The Verilog source code for *ebegin* snoops the bus waiting for the presence of *ebegin* in the opcode field of the instruction bus. Then, the state machine transitions through the next five states, incrementing the stack pointer, writing the new mark to the stack, writing the new mark to the mark register, and finally incrementing the instruction address register. Similarly, the Verilog source code for *eend* snoops the bus waiting for the presence of *eend* in the opcode field of the instruction bus. It then proceeds through seven states in which the control signals sequentially save the block return value, restore the previously stacked mark (if one exists) to the mark register, returns the stack pointer to the value it held before the block was executed, and pushes the block return value.
module ebegin(clk, markbus, opcode, reset, spbus, cbus, dbus);
  input clk;
  input [47:0] markbus;
  input [7:0] opcode;
  input reset;
  input [7:0] spbus;
  output [16:0] cbus;
  output [47:0] dbus;
  reg [2:0] statebits;
  reg [16:0] nanoword;

assign dbus["BLOCK"] = (statebits==3 || statebits==4)?markbus["BLOCK"]+8'b1:8'b0;
assign dbus["DYNAMIC"] = (statebits==3 || statebits==4)?markbus[7:0]:8'b0;
assign dbus["STATIC"] = (statebits==3 || statebits==4)?markbus[7:0]:8'b0;
assign dbus["VAR-END"] = (statebits==3 || statebits==4)?spbus:8'b0;
assign dbus["RETURN"] = (statebits==3 || statebits==4)?8'b0:8'b0;
assign dbus["MP"] = (statebits==3 || statebits==4)?spbus:8'b0;
assign cbus = (statebits == 0)?17'b0:nanoword;

always @(posedge clk)
begin : stateTransition
  case (statebits)
    0: if (reset == 0) statebits = 0;
        else if (opcode == 'ebegin) statebits = 1;
        else statebits = 0;
    1: if (reset == 0) statebits = 0;
        else statebits = 2;
    2: if (reset == 0) statebits = 0;
        else statebits = 3;
    3: if (reset == 0) statebits = 0;
        else statebits = 4;
    4: if (reset == 0) statebits = 0;
        else statebits = 5;
    5: if (reset == 0) statebits = 0;
        else statebits = 0;
    default: statebits = 0;
  endcase
end

always @(statebits)
begin : combLogic
  case (statebits)
    0: nanoword = 17'h1FFFF;
    1: nanoword = 17'h1FDFF;
    2: nanoword = 17'h1FFEF;
    3: nanoword = 17'h1FEEF;
    4: nanoword = 17'h1FEBB;
    5: nanoword = 17'h1F7FF;
    default: nanoword = 17'h1FFFF;
  endcase
end
endmodule

Figure 4.9: The Verilog description of the ebegin opcode
module eend(clk, opcode, reset, cbus, dbus);
input clk;
input [7:0] opcode;
input reset;
output [16:0] cbus;
inout [47:0] dbus;
reg [2:0] statebits;
reg [16:0] nanoword;
reg [47:0] temp;

assign dbus = (statebits == 4) || (statebits == 5) ? temp : 48'bz;
assign cbus = (statebits == 0) ? 17'bz : nanoword;

always @(negedge clk)
begin : tempLatch
  case (statebits)
    1: temp = dbus;
    5: temp = dbus;
    default: temp = temp;
  endcase
end

always @(posedge clk)
begin : stateTransition
  case (statebits)
    0: if (reset == 0) statebits = 0;
      else if (opcode == 'eend) statebits = 1;
      else statebits = 0;
    1: if (reset == 0) statebits = 0;
      else statebits = 2;
    2: if (reset == 0) statebits = 0;
      else statebits = 3;
    3: if (reset == 0) statebits = 0;
      else statebits = 4;
    4: if (reset == 0) statebits = 0;
      else statebits = 5;
    5: if (reset == 0) statebits = 0;
      else statebits = 6;
    6: if (reset == 0) statebits = 0;
      else statebits = 7;
    7: if (reset == 0) statebits = 0;
      else statebits = 0;
    default: statebits = 0;
  endcase
end

always @(statebits)
begin : combLogic
  case (statebits)
    0: nanoword = 17'hFFFF;
    1: nanoword = 17'hFFDF;
    2: nanoword = 17'hFEFF;
    3: nanoword = 17'hFFEF;
    4: nanoword = 17'hFFCF;
    5: nanoword = 17'hFF9F;
    6: nanoword = 17'hFFBF;
    7: nanoword = 17'hFFF7;
    default: nanoword = 17'hFFFF;
  endcase
end
endmodule

Figure 4.10: The Verilog description of the eend opcode
The Set of Opcode Circuits

The set, $O$, of opcodes needed to create the custom computing machine, $M$, for an Euler algorithm are extracted from the opcode library by the Euler-97 compiler during synthesis. The opcode library is a collection of Verilog descriptions that has been written for each opcode. The opcodes are instantiated into the final custom computing machine by using the module instantiation constructs of the Verilog language.

Table 4.4 lists the opcodes that are currently implemented as Verilog descriptions, and stored in the opcode collection. Note that since the Euler-97 compiler does not support floating point mathematics, the cardinality of the implemented set differs from the cardinality given in the language definition. Integer multiplication and division are supported as byte-sized operations on the least significant byte of the data operands.

Table 4.4: Implemented opcodes

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebegin</td>
<td>eend</td>
<td>esemi</td>
<td>elquote</td>
</tr>
<tr>
<td>erquote</td>
<td>enew</td>
<td>eref</td>
<td>eassign</td>
</tr>
<tr>
<td>eout</td>
<td>ein</td>
<td>eformal</td>
<td>enumber</td>
</tr>
<tr>
<td>elogval</td>
<td>eomega</td>
<td>elabel</td>
<td>eadd</td>
</tr>
<tr>
<td>esub</td>
<td>emod</td>
<td>eiinteger</td>
<td>elogical</td>
</tr>
<tr>
<td>emin</td>
<td>emax</td>
<td>elthan</td>
<td>elethan</td>
</tr>
<tr>
<td>egthanh</td>
<td>egthan</td>
<td>eequal</td>
<td>enequal</td>
</tr>
<tr>
<td>evalue</td>
<td>erparen</td>
<td>ecall</td>
<td>egoto</td>
</tr>
<tr>
<td>ethen</td>
<td>eelse</td>
<td>eand</td>
<td>eor</td>
</tr>
<tr>
<td>enot</td>
<td>elist</td>
<td>elength</td>
<td>erbrak</td>
</tr>
<tr>
<td>emul</td>
<td>ediv</td>
<td>eisb</td>
<td>eisl</td>
</tr>
<tr>
<td>eisli</td>
<td>eism</td>
<td>eisn</td>
<td>eisp</td>
</tr>
<tr>
<td>eisr</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Additionally, some functionality associated with the list data structure is not supported. For example, concatenation of two lists is not currently supported.

Each opcode has input and output signals that must be attached to the signals interconnecting the components of the custom computing machine. Circuit modules described in Verilog support this interconnection through a port list specified in the module description. For example, the Verilog description for a four-bit up counter is given in Figure 4.11. The port signals are \( ar, \) \( clk, \) and \( value. \) Note that the widths of busses can be specified with parameters, and that the direction of the port is declared as

```verilog
module upcounter(ar, clk, value);
    parameter valueWidth = 8;
    input ar, clk;
    output [valueWidth-1:0] value;

    reg [valueWidth-1:0] value;

    always @(negedge ar)
    begin : asynchReset
        assign value = 0;
    end

    always @(posedge ar)
    begin : releaseReset
        deassign value;
    end

    always @(posedge clk)
    begin : countUp
        value = value + 1;
    end
endmodule
```

Figure 4.11: A Verilog description of an up counter
input or output. Values in Verilog that must maintain their values through time (i.e. non-combinational) are declared as register storage units using the `reg` keyword. In this case, the `value` register is also aliasing the port named `value`, and thus any change in register value will be visible on the port. A complete explanation of the Verilog code will not be given in this dissertation. Instead, refer to a number of good references on the Verilog hardware description language, including [6].

The counter, as a stand-alone module, can now be used hierarchically as a component within a larger system. This is the way the Euler opcodes will be used hierarchically as modules within the SAMUEL-generated custom computing machine. Figure 4.12 is a Verilog description of a larger circuit that instantiates two counters. Note how signals are attached to the port interface to allow information flow from the counters

```verilog
module biggerCircuit(biggerAsynchReset, biggerClk, ...)

parameter counterWidth = 4;

input biggerAsynchReset, biggerClk;
wire [counterWidth-1:0] counter1Value;
wire [counterWidth-1:0] counter2Value;

/* instantiate two counters named value1 and value2 */
upcounter #(counterWidth) value1(biggerAR,biggerCLK,counter1Value);
upcounter #(counterWidth*3) value2(biggerAR,biggerCLK,counter2Value);

always @( ... )

Figure 4.12: Hierarchical instantiation of the up counter
```
into the enclosing circuit. Also note that parameter values can be overridden by passing new parameter values (in order of declaration in the instantiated module) to each instantiated module. Thus, the counter instantiated as `counter1` will be a four-bit counter, while the counter instantiated as `counter2` will be a 12-bit counter. If no parameter values are specified at instantiation, the default parameter values are used as declared in the module definition. In this case, an instantiated counter without parameters specified would default to be an eight-bit counter.

The basic behavior of every Euler opcode written in Verilog is sequential. As mentioned in the discussion of the `ebegin` and `eend` opcodes, once activated an opcode circuit will proceed through a number of states, asserting control signals and analyzing data when appropriate. This basic behavior is diagrammed in Figure 4.13. The number of states varies between opcodes, since clearly the behavior performed by each opcode

![Figure 4.13: Basic opcode sequence](image)

can vary in complexity. For example, the `esemi` opcode does nothing but pop the stack, while the `eval` opcode must first dereference a piece of stacked data and then determine if the dereference resulted in a procedure that needs to be called. Recall from the previous section that each opcode has an `orbit` state that monitors the instruction bus. If
the opcode encoded on the instruction bus matches this opcode circuit, then the opcode transitions to the state named State 1 to begin the behavior. Each opcode updates the instruction address register when finished by either incrementing or loading a new value. When the behavior terminates, the *orbit* state is entered again, and the process repeats. The state transitions are synchronously controlled by the system clock. Examples of some Verilog descriptions will be provided next to illustrate the opcode library. A representative set of opcodes is provided for reference in Appendix B. The entire set of opcodes has not been included.

**Example: Mathematical Opcodes**

The 16 arithmetic and logical opcodes are listed in Table 4.5. Each arithmetic opcode circuit must check the types of the popped data to guarantee that two numbers are being operated upon. If this is not the case, a type error should occur at runtime as defined in the Euler language. The control bus contains the signal *error* which is asserted

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>eadd</td>
<td>esub</td>
<td>emod</td>
</tr>
<tr>
<td>emin</td>
<td>emax</td>
<td>elthan</td>
</tr>
<tr>
<td>elethan</td>
<td>egethan</td>
<td>egthan</td>
</tr>
<tr>
<td>eequal</td>
<td>enequal</td>
<td>eand</td>
</tr>
<tr>
<td>eor</td>
<td>enot</td>
<td>emul</td>
</tr>
</tbody>
</table>
if one of the opcodes enters the runtime error state. The *error* signal could be used to light a light-emitting diode (LED) for the user, or to stop the system clock and halt processing. However, at the present time, this interface is not well defined. All runtime errors are signaled in the exact same way, with a single *error* signal, rather than an error bus encoding which runtime signal occurred.

The abstract behavior of each mathematical opcode is diagrammed in Figure 4.14 as a state machine, with states labeled to reflect the actions taken during each state. Note that the logical opcodes `eand`, `eor`, and `enot` do not pop the stack twice, as they only need the top value to compute their outcome. This state machine behavior mirrors the behavior suggested by the original Euler language definition for mathematical opcodes [10,11].

The `eadd` opcode will be used to demonstrate a basic mathematical opcode

![State Machine Diagram](image)

*Figure 4.14: Basic mathematical opcode behavior*
Verilog description. The nanoregister transfer level (nanoRTL) description for the control bus outputs from eadd is listed in Table 4.6. The nanoinstructions needed for each nanoRTL instruction are also shown. This nanoRTL is very simple, and a lexical analyzer and nanoinstruction generator have been written in Flex to provide software support for quickly generating the control signal values within each state. Note that all control signals in the SAMUEL-generated custom computing machines are asserted low, except for encodings, and thus a logic zero in the nanoinstruction represents an asserted signal, while a logic one represents a signal that is not asserted during that state.

Table 4.6: Nanoregister transfer language instructions for eadd

<table>
<thead>
<tr>
<th>nanoRTL</th>
<th>nanoinstruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>POP:</td>
<td>scs IFFDF</td>
</tr>
<tr>
<td></td>
<td>decsp IFCFF</td>
</tr>
<tr>
<td>POP:</td>
<td>scs IFFDF</td>
</tr>
<tr>
<td>PUSH:</td>
<td>swe IFFEF</td>
</tr>
<tr>
<td></td>
<td>swe,scs IFFCF, out = numb1 + numb2</td>
</tr>
<tr>
<td>IADR:</td>
<td>inciadr 17FFF</td>
</tr>
<tr>
<td>ERROR:</td>
<td>error 0FFF, entered if numb1 or numb2 is not number</td>
</tr>
</tbody>
</table>

Figure 4.15 shows the Verilog implementation storing the current state in a register named statebits, and the combinational output logic generated whenever the value in statebits changes. This is controlled by the event binding always @(statebits) placed on the combinational logic block. Also note that a rising edge on the opcode clock causes the state transitions to occur. The orbit state is equivalent to statebits = 0, with the remaining states only entered if the value on the ibus port matches the defined binary value (eadd) from the list of opcodes defined in the file <defines.v>.
module eadd(clk, opcode, reset, cbus, dbus);
input clk;
input [7:0] opcode;
input reset;
output [16:0] cbus;
inout [47:0] dbus;
reg [2:0] statebits;
reg [15:0] nanoword;
reg [39:0] number1;
reg [39:0] number2;
assign dbus["TYPEFLAG1" = (statebits == 5) ? number1 : "bz;
assign dbus["NUMBER1" = (statebits == 5) ? number1["NUMBER"] + number2["NUMBER"] : "bz;
assign cbus = (statebits == 0) ? 17'hz : nanoword;

always @(negedge clk)
begin : tempLatch
  case (statebits)
    0: number1 = dbus;
    1: number1 = dbus;
    default: begin number1 = number1; number2 = number2; end
  endcase
end

always @(posedge clk)
begin : stateTransistion
  case (statebits)
    0: if (reset == 0) statebits = 0;
      else if (opcode == 'eadd) statebits = 1;
      else statebits = 0;
    1: if (reset == 0) statebits = 0;
      else statebits = 2;
    2: if (reset == 0) statebits = 0;
      else statebits = 3;
    3: if (reset == 0) statebits = 0;
      else statebits = 4;
    4: if (reset == 0) statebits = 0;
      else statebits = 5;
    5: if (reset == 0) statebits = 0;
      else statebits = 6;
    6: if (reset == 0) statebits = 0;
      else if (number1["TYPEFLAG"] == numb && number2["TYPEFLAG"] == numb)
      statebits = 0;
      else statebits = 7;
    7: if (reset == 0) statebits = 0;
      else statebits = 7;
    default: statebits = 0;
  endcase
end

always @(statebits)
begin : combLogic
  case (statebits)
    0: nanoword = 17'hFFFF;
    1: nanoword = 17'h00FF;
    2: nanoword = 17'h00FD;
    3: nanoword = 17'h00FF;
    4: nanoword = 17'h00FD;
    5: nanoword = 17'h00FF;
    6: nanoword = 17'hFFFF;
    7: nanoword = 17'h00FF;
    default: nanoword = 17'hFFFF;
  endcase
end
endmodule

Figure 4.15: Verilog implementation of the eadd opcode
Example: Delimiters

The five delimiter opcodes are listed in Table 4.7. Note that the opcode esemi is listed here as a delimiter since it delimits the end of an Euler language statement. The other delimiters, ebegin, eend, elquote, and erquote delimit blocks and procedures. The delimiters that start a block cause the creation of a mark, while the terminators restore the

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebegin</td>
<td>eend</td>
</tr>
<tr>
<td>esemi</td>
<td>elquote</td>
</tr>
<tr>
<td>erquote</td>
<td></td>
</tr>
</tbody>
</table>

previous environment data. Recall that delimiters ebegin and eend belong to the set of required circuits, R. Table 4.8 gives nanoRTL and nanoinstructions for the ebegin opcode to demonstrate delimiter state behavior. The ebegin opcode drives a new mark data value onto the stack, as well as into the mark register. The ebegin opcode does not add a type flag to the data value. Verilog code for ebegin is given in Figure 4.9. As

Table 4.8: Nanoregister transfer language instruction for ebegin

<table>
<thead>
<tr>
<th>nanoRTL</th>
<th>nanoinstruction (refer to Table 4.3 for signals and bit positions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH: incsp</td>
<td>1FDFF</td>
</tr>
<tr>
<td>swe</td>
<td>1FFE5</td>
</tr>
<tr>
<td>swe,scs</td>
<td>1FFCF</td>
</tr>
<tr>
<td>MREG:ldmp</td>
<td>1FFFB</td>
</tr>
<tr>
<td>IADR: inciadr</td>
<td>17FFF</td>
</tr>
</tbody>
</table>
mentioned previously, the new mark is pushed onto the stack in the first three states, stored into the mark pointer in the fourth state, and the instruction address is advanced in the final state. Note that the opcode circuit drives the new mark onto the data bus during the push to the stack, as well as the write to the mark register.

Example: Data Types

The opcodes used to create basic data types (with the exception of marks and procedures) are listed in Table 4.9. The symbol data type from the Euler language definition is not implemented in this version of the Euler opcode library. The `enumber` and `elabel` opcode circuits will be used to demonstrate the behavior of opcode circuits from this class.

Table 4.9: Opcodes that create basic data types

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>enumber</td>
<td>elabel</td>
</tr>
<tr>
<td>elist</td>
<td>eref</td>
</tr>
<tr>
<td>elogval</td>
<td>eomega</td>
</tr>
</tbody>
</table>

A number in the current version of SAMUEL is stored as a 16-bit quantity in bits 15:0 with the 8-bit `numb` type flag stored in bits 47:40. The numerical constant created by the `enumber` opcode is extracted as a 16-bit quantity from the instruction word. Table 4.10 gives the nanoRTL segment for the `enumber` opcode circuit. During the third state,
Table 4.10: Nanoregister transfer language instruction for enumber

<table>
<thead>
<tr>
<th>nanoRTL</th>
<th>nanoinstruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH: incsp</td>
<td>1FDFF</td>
</tr>
<tr>
<td>swe</td>
<td>1FFE8</td>
</tr>
<tr>
<td>swe,scs</td>
<td>1FFCF</td>
</tr>
<tr>
<td>IADR: inciadr</td>
<td>17FFF</td>
</tr>
</tbody>
</table>

(refer to Table 4.3 for signals and bit positions)

the opcode outputs the numerical data value onto the data bus for writing onto the stack.

Refer to Appendix B for the Verilog description.

A label is an Euler data type which is used to name program instructions. When encountered in the program string, `elabel` creates a stacked data item that can be used later by `egoto` to reset the instruction address register to the address of the named instruction. The nanoRTL segment for `elabel` will appear identical to that of `enumber`. However, the data value pushed to the stack in state three will be the type flag `labl` in bits 39:32, followed by two 8-bit fields in bits 31:24 and 23:16 used to locate the instruction in the program string when used by `egoto`. Refer to Appendix B for the Verilog description.

Hopefully, these example descriptions have provided insight into how the Verilog library is written, and into the type of sequential circuits that will result for simulation or synthesis. It should be noted that a lot of optimization can be done to the current opcode library.
CHAPTER 5. EXAMPLE MACHINES

Consider the Euler algorithm given in Figure 5.1. This algorithm is the polynomial calculation \( y = ax^3 + bx^2 + cx + d \). Such expressions are widely used in mathematics and engineering. In some applications, the potential performance gains from hardware implementation may warrant investigating a custom computing machine to calculate this type of expression. The algorithm begins by declaring storage space for the coefficients, as well as variable \( x \). The coefficient values are assigned. The input value is sampled into variable \( x \), and the result of the polynomial calculation is provided as output. As Figure 5.2 shows, the translated program consists of 57 instructions to implement the polynomial calculation on the Euler virtual machine. Thirty-six of the instructions are needed to support the variables and name space. This clearly shows that most Euler algorithms will result in machines that spend a large amount of the execution time managing the overhead of variable name spaces.

```
begin
  new a;
  new b;
  new c;
  new d;
  new x;
  a:=3;
  b:=4;
  c:=5;
  d:=1;
  x:=in;
  out a*x*x*x + b*x*x + c*x + d
end
```

Figure 5.1: A polynomial calculation written in Euler
<table>
<thead>
<tr>
<th>Source Line</th>
<th>Opcode</th>
<th>Operand1</th>
<th>Operand2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ebegin</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>enew</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>enew</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>enew</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>enew</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>enew</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>eref</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>enumber</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>eassign</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>esemi</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>eref</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>enumber</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>eassign</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>esemi</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>eref</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>enumber</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>eassign</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>esemi</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>eref</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>enumber</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>eassign</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>esemi</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>eref</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>ein</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>eassign</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>esemi</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>eref</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>eval</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>eref</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>eval</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>emul</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>eref</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>eval</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>emul</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>eref</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>eval</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>eref</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>eval</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>emul</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>eref</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>eval</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>emul</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>eadd</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>eref</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>eval</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>eref</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>eval</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>emul</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>eadd</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>eref</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>eval</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>eadd</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>eout</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>eend</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 5.2: The polynomial expression translated to the virtual machine opcodes
Figure 5.3 gives the *synthesis personality* created by the Euler-97 compiler for the polynomial calculation. The opcodes needed to implement the calculation are summarized in this synthesis personality. The first column is the library number corresponding to the opcode given in the second column. Thus, the `evalue` opcode is the fifty-seventh opcode in the opcode library. During custom computing machine generation, the Euler-97 compiler will include library opcode 57 in the final machine. Again, remember that these opcodes belong to the set \( O \), and are the opcodes that uniquely define the described algorithm. Only the opcodes in the synthesis personality are required in a machine built to execute the polynomial calculation. The final column is the total number of times the opcode appears in the program string.

Twelve opcodes are needed to implement the polynomial calculation as a custom computing machine. The cardinality of set \( O \) is 10, since the `ebegin` and `eend` opcodes are included within the required set \( R \). The sets \( R \) and \( O \) have been instantiated into the top-

```
<table>
<thead>
<tr>
<th>EULER Synthesis Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>5   eassign      5</td>
</tr>
<tr>
<td>6   esemi         5</td>
</tr>
<tr>
<td>12 eref          15</td>
</tr>
<tr>
<td>16  eadd         3</td>
</tr>
<tr>
<td>18  ebegin       1</td>
</tr>
<tr>
<td>22  eend         1</td>
</tr>
<tr>
<td>26  ein          1</td>
</tr>
<tr>
<td>44  emul         6</td>
</tr>
<tr>
<td>46  enew         5</td>
</tr>
<tr>
<td>48  enumber      4</td>
</tr>
<tr>
<td>51  eout         1</td>
</tr>
<tr>
<td>57  evalue       10</td>
</tr>
</tbody>
</table>
```

---

Total Opcodes Required: 12
Total Opcode Calls: 57

---

Figure 5.3: The synthesis personality for the polynomial calculation
level module of the custom computing machine by the Euler-97 compiler as shown in
Figure 5.4. Recall from chapter 3 that the -synth command line option causes the Euler-
97 compiler to generate a data path, and the -sim command line option causes the addition
of Verilog stack and instruction ROM modules. Note that Figure 5.4 only shows a very
small portion of the actual Verilog custom computing machine. The compiler actually
creates one file, called <fname.v>, which contains all of the Verilog opcode modules, as
well as this top-level machine module. Additionally, if the -sim option is used, a test
module is included in <fname.v> to facilitate simulation. The single Verilog file is all
that is needed for cross-platform use of the custom computing machine description. The
opcode library does not need to be on the same computer as the simulation environment
or the synthesis environment. The complete Verilog source code for the SAMUEL-
generated polynomial machine can be found in Appendix C, along with a schematic data
path created by one commercial synthesis tool from the Verilog source code.

Consider some simple analysis on the resulting Verilog file for the polynomial
calculation. Returning to the saved real-estate expression from Chapter 4, the custom
computing machine with just the required 12 opcodes (two in set R, and ten in set O)
results in an averaged savings of 45μ when compared to a machine that would support
every Euler opcode regardless of the algorithm. Suppose any other polynomial
expression were desired. What kind of custom computing machine description would
result? Would it be a completely different description, with a completely different
averaged savings parameter? Or, would the same custom computing machine result?
module machine(clk, error, inport, outport, reset);
    input clk;
    output error;
    output [7:0] iabus;
    input[47:0] inport;
    output [47:0] outport;
    input reset;
    output [7:0] spbus;
    wire [47:0] dbus;
    wire [7:0] spbus; /* busses */
    wire [7:0] sabus;
    wire [7:0] iabus;
    wire [47:0] markbus;
    wire [39:0] ibus;
    wire [47:0] cbus;
    wire [47:0] inputbus;

    /* instantiate members of R from M = {R,0} */
    mreg inputreg(clk, inport, reset, 1'b1, 1'b0, inputbus);
    tsbuf inputbuf(inputbus, cbus[1'dir], dbus);
    mreg #8 sp(clk, markbus[1'VAREND], reset, cbus[1'slsp], cbus[1's0sp], spbus);
    mreg markreg(clk, dbus, reset, cbus[1'simp], ebus[1's0mp], markbus);
    mreg #(8,8'b00000000) iadr(clk, dbus[7:0], reset, cbus[1'sliadr],
                              cbus[1's0iadr], iabus);
    mux41 #(8) smux(spbus, markbus[1'DYNAMIC], markbus[1'STATIC],
                   markbus[1'MP], sabus, cbus[1'smux1], cbus[1'smux0]);
    mem stack(sabus, cbus[1'scs], dbus, cbus[1'swe]);
    irom imemory(iabus, ibus);

    assign error = cbus[1'error];

    /* instantiate members of O from M = {R,0} */
    eassign assignop(clk, ibus[1'_OPCODE], markbus, reset, cbus, dbus);
    esemi semiop(clk, ibus[1'_OPCODE], reset, cbus);
    eref refop(clk, ibus, markbus, reset, cbus, dbus);
    eadd addop(clk, ibus[1'_OPCODE], reset, cbus, dbus);
    ebegin beginop(clk, markbus, ibus[1'_OPCODE], reset, spbsbus, cbus, dbus);
    end endop(clk, ibus[1'_OPCODE], reset, cbus, dbus);
    ein inop(clk, ibus[1'_OPCODE], reset, cbus, dbus);
    emul mulop(clk, ibus[1'_OPCODE], reset, cbus, dbus);
    enew newop(clk, markbus, ibus[1'_OPCODE], reset, cbus, dbus);
    enumber numberop(clk, ibus, reset, cbus, dbus);
    eout outop(clk, ibus[1'_OPCODE], reset, cbus, dbus);
    evalue valuop(clk, ibus[1'_OPCODE], markbus, reset, cbus, dbus);

endmodule

Figure 5.4: The machine generated from the Euler polynomial calculation
The fundamental premise of this work is that unique algorithms will produce unique circuits. This is true, of course, only in the sense that the algorithms differ in abstraction and opcode needs. It is possible for two completely different algorithms to have the same set of opcodes in their synthesis personalities. These two algorithms will be executed by identical SAMUEL custom computing machines.

Consider the modified polynomial calculation shown in Figure 5.5. In this calculation, the degree of the polynomial is one greater than previously. The synthesis

```
begin
new a;
new b;
new c;
new d;
new x;
a:=3;
b:=4;
c:=5;
d:=1;
x:=in;
out a*x*x*x*x + b*x*x*x + c*x + d
end
```

Figure 5.5: A new polynomial calculation

personality for the polynomial of Figure 5.5 is exactly the same as the synthesis personality for the original polynomial. Thus, identical custom computing machines will result. In fact, the same custom computing machine will be generated for a polynomial of any degree (degree greater than or equal to 1) that consists of only multiplication and addition. A subtraction within the polynomial, however, would add a subtraction opcode to the synthesis personality, and thus generate a slightly different machine.
The new polynomial calculation does differ in one important way. The synthesis personality of the new polynomial indicates that the total number of instructions needed to implement the polynomial has increased to 63. This increase is expected, as additional multiplications are required to calculate the polynomial value.

Before leaving the polynomial examples, it will be beneficial to explore the type of machine that results if variable use is minimized. Clearly, the polynomial calculations shown so far did not need to store the coefficients in variables. Instead, Figure 5.6 provides an alternative Euler implementation of the original polynomial calculation. In

```
begin
  new x;
  x := in;
  out 3*x*x*x + 4*x*x + 5*x + 1
end
```

Figure 5.6: Another alternative polynomial calculation

this alternative, only the input value is stored in a variable. The coefficients have been placed directly into the output expression. The synthesis personality of this alternative polynomial calculation again reveals that the same 12 opcodes are required in the final custom computing machine. This alternative polynomial, however, only requires 33 instructions to complete calculation. Minimizing the overhead associated with dereferencing and assignment of the coefficient variables has led to less program instructions. The inclusion of variable x, as well as the assignment to variable x, force
the inclusion of the opcodes associated with variable assignment and dereference in the final machine.

Figure 5.7 shows an iterative addition algorithm. This slightly more sophisticated example can be used to illustrate the type of Verilog machines that result when more complex behaviors are described algorithmically. This example contains named program labels, as well as branches and boolean inequalities. The algorithm samples the input

```verilog
begin
    new a;
    new b;

    label loop;
    loop: a := in;
    b := in;
    if a != 0 and b != 0 then out a+b else out 35565;
    if a=0 or b = 0 then out 0 else goto loop

end
```

Figure 5.7: An iterative addition algorithm

until zeros are received. During each iteration of the loop, the sum of the last two input samples is output. As Figure 5.8 demonstrates, the synthesis personality of this algorithm includes nineteen opcodes (two within R, 17 within O). Implementing these nineteen opcodes would result in an averaged real-estate savings of 38μ. The generated custom computing machine can be found in Appendix C.

As a final example, consider the iterative factorial algorithm shown in Figure 5.9. This algorithm is even more sophisticated than the previous examples because multiple name spaces are used. Note the `begin-end` delimiters. It is easy to see, however, that the additional name spaces will not add any opcodes to the final machine, since every
machine already requires the \texttt{ebegin} and \texttt{eend} opcodes. The synthesis personality for this iterative factorial algorithm places 17 opcodes into the final machine (2 within \texttt{R}, 15 within \texttt{O}), and notes 41 instructions in the program string. Again, contrast this final machine with a machine built to support every opcode in Euler. The potential real-estate savings easily justify this approach to supporting high-level languages.

Appendix C contains a complete set of SAMUEL-generated Verilog code for the examples provided in this chapter, as well as simulation runs and synthesis personalities for reference.
begin
new invalue;
new result;

label LOOP;

invalue := in;
result := 1;

LOOP: if invalue > 1 then
  begin
    result := result * invalue;
invalue := invalue - 1;
goto LOOP
  end
else out result

end

Figure 5.9: An iterative factorial algorithm
CHAPTER 6. CONCLUSIONS AND FUTURE WORK

A rapid prototyping environment, called SAMUEL, for creating custom computing machines has been described. Custom computing machines are generated by a compiler from a general purpose algorithmic language and a library of opcode circuits. The custom computing machines have successfully simulated, and commercial synthesis tools have been able to produce net list files. The net list files can be presented to commercial FPGA software for placement and routing.

Simulation of SAMUEL-generated custom computing machines has demonstrated the feasibility of this approach. SAMUEL-generated custom computing machines are competitive with other documented systems in the sense that these custom computing machines have been shown to simulate correctly. While the current Verilog descriptions are quite large, future work will focus on optimizing the opcode library, as well as changing the way block structure is supported. Implementation in FPGAs also remains as future work.

The opcode library makes SAMUEL unique among other research work that has been documented by raising the semantic level of the level 0 circuits. SAMUEL is also unique because the algorithmic language used is not a hardware description language, and has not been modified in any way from the original language definition.

SAMUEL is also unique because the language chosen supports dynamic procedure definition. This allows a procedure to transform into a completely different procedure at runtime. This is language-supported reconfigurability, which SAMUEL
uses to enhance the current research trends in reconfigurable devices. The language ability to dynamically reconfigure procedures at runtime makes SAMUEL different among other documented methods that instead must be host-supported, single instruction or single procedure-based dynamic instruction set FPGAs [15,19,20,23,24].

Custom computing machines generated by SAMUEL can be described using the scheme given by Milutinovic as software translated, language corresponding, complex, directly executing architectural support for the high-level language Euler [7]. This approach differs from other work, however, by exploiting the field programmability of gate arrays (and the freedom guaranteed by a simulation environment) to create custom computing machines that only support the required language opcodes. This is important within the design space of limited real-estate programmable logic. Average real-estate savings can be achieved by not implementing support for the entire language in every custom computing machine.

SAMUEL-generated custom computing machines will not be competitive if the system is used in an attempt to speed up highly parallel data manipulation. Any parallelism inherent in the algorithm is not recognized by the Euler-97 compiler for exploitation. Concurrently active opcodes are also not supported, thus compounding the inability to exploit parallelism.
Future Work

Additional work can be done to make SAMUEL a more useful tool, as well as to generalize the custom computing machines generated. An integer-only Euler-97 compiler was built to serve as an existence proof of the feasibility of the SAMUEL approach to custom computing. Thus, since the Euler-97 compiler does not currently support floating point mathematics, the set of algorithms that can be implemented is clearly smaller. An appropriate starting point for future work would be to build an Euler-97 compiler that supports floating point mathematics, as well as the other Euler non-integral types, such as characters (called symbols in the Euler literature). By nature, this work would then require additional work in the design and testing of new library opcode circuits to process the non-integral types.

Extending SAMUEL so that it can support parallelism is another area of work that can be pursued. This can be done in a variety of ways. For example, the Euler-97 compiler can be extended to extract statements within blocks that can be executed in parallel. Then, an appropriate mechanism must be added to the machine template to allow multiple opcodes to be active at the same time. This would, of course, then require an appropriate memory access protocol to preserve the integrity of the storage areas.

Another approach to supporting parallelism would be to extend the Euler language with parallel constructs. Then, appropriate opcode circuits would be designed to support these constructs. Parallel extensions to Euler have been documented by others [38]. Novel parallel extensions might also prove useful.
Another area of future work would be to enhance the Euler language with constructs that would support event-driven behavior. This would allow a larger class of algorithms to be expressed, as well as raise the semantic level of the language and corresponding custom computing machine. It is interesting to note that events are strongly supported in Verilog, and thus an easy transition between the language constructs and the Verilog opcodes may exist.

Additional languages could be supported by SAMUEL. Potential languages include Java, BASIC, and small subsets of Common-Lisp. This would require a general front-end compiler capable of translating the given language to an appropriate opcode set, and then generating the machine description from the set, $O_L$, which contains the opcode circuits that support the input language. Studies of the performance and real-estate usage of the generated custom computing machines, as well as the user acceptance of the language, could enhance the efficacy of SAMUEL.

A final area of future work that might prove interesting would be a complete redesign of the SAMUEL custom computing machine template. Resulting machines could be compared and contrasted to current machines.
APPENDIX A. THE EULER-97 COMPILER

```c
# include <stdio.h>
# include <string.h>
# include <stdlib.h>

#define undefined 0
#define TOTAL_OPCODES 58
#define EULER_VERSION_STRING "Euler 1997 v1.0"
#define LIBRARY_PATH "/home/phd/RESEARCH/EULER/LIB"

typedef enum
  { OPequal, OPrem, OPconcat, OPquote, OPparen, OPassign, OPsemi, OPltthan, OPlechcin, OPlechcin, OPgechan, OPref, OPbrak, OPlquote, OPabs, OPadd, OPand, OPbegin, OPcall, OPdiv, OPelse, OPend, OPexp, OPcall, OPgoco, OPin, OPinceger, OPisb, OPisl, OPisli, OPisn, OPisp, OPisr, OPisu, OPsw, OPlabel, OPlength, OPlist, OPlogical, OPlogval, OPmax, OPmin, OPmod, OPmul, OPneg, OPnew, OPnot, OPnumber, OPomega, OPor, OPout, OPreal, OPSsub, OPSsymbol, OPcall, OPthen, OPvalue} EulerOpcodeType;

typedef enum
  { undf = 0, bool, labl, list, mark, numb, proc, symb, vref, garb} EulerDataTypes:

  /* undf = 0 is used in this enum so that when */
  /* calloc is used to create the variable heap */
  /* all objects will initially have a typeflag */
  /* of undf. This avoids the loop that would */
  /* otherwise be required to init variables to */
  /* be undefined. A neat little use of bad C! */

typedef struct
  { EulerDataTypes typeflag;
    int value[8];
  } EulerDataObject;

typedef struct
  { EulerOpcodeType opcode;
    int operand1;
    int operand2;
    int sourceline;
  } EulerInstruction;

typedef struct
  { char *value;
    int block_number;
    int offset;
    char *type;
  } SymTabEntry;

#ifndef EULERMAIN
 /* for Che parser */
 extern int BlockNumber, Offset, ProgramIndex, MarkValue, SymTabIndex;
```
extern int Stemp, Scale, Temp, Linenum;
extern EulerInstruction *CompiledProgram;
extern SymTabEntry *SymTab;
extern FILE *yyin, *RedirectedOutputFile;

/* the flags */
extern int EulerReportFlag, EulerRedirectedOutputFlag, EulerDebugFlag;
extern int EulerCompiledProgramOutputFlag, EulerCompileOnlyFlag;
extern int EulerSynthesisFlag;

/* for the interpretation */
extern EulerDataObject *EulerStack;
extern EulerDataObject *EulerVarHeap;
extern int EulerFCT, EulerMP, EulerPC, EulerSP, EulerVTP;
extern float EulerCompiledProgramLength, EulerStackSize, EulerVarHeapSize;

/* for the system */
extern char *OPCODE_STRINGS[];
extern int EulerOpcodeCounts;
#endif

/* from parser.c */
void NewInstruction(int, EulerOpcodeType, int, int);
void NewSymTabEntry(int, char*, int, int, char*);
void ChangeSymTabOffset(int, int);
void ChangeSymTabType(int, char*);
void ChangeInstructionOperand(int, int);
void var_error(void);
void lab_error(void);
int yyerror(char*);
int yyparse(void);
int yywrap(void);

/* from cond.c */
void elsee(int operandi);
void else(int operandi);

/* from const.c */
void eval(void);
void ecall(void);
void egoto(void);

/* from cycle.c */
void Cycle(void);

/* from debug.c */
void monitor(void);

/* from defin.c */
void ebegin(void);
void eendr(void);
void esemi(void);
void elquote(int operandi);
void erquote(void);

/* from files.c */
void FatalError(const char *in_message);
void RuntimeWrong(char *in_message);

/* from list.c */
void ebegin(void);
void eparen(int operandi);
void elen(void);
EulerDataObject ecopylist(EulerDataObject inlistdescriptor);
void ebrak(void);

/* from logics.c */
void eand(int operandi);
void eordn(int operandi);
void enot(void);

/* from marlc.c */
int CurrentMarkBlockNumber(void);
int CurrentMarkDynamicLink(void);
int CurrentMarkStaticLink(void);
int CurrentMarkVarListStart(void);
int CurrentMarkReturnAddress(void);
int FindMark(int);

/* from emach.c */
void eadd(void);
void esub(void);
void emul(void);
void ediv(void);
void emod(void);
void eexpt(void);
void enegate(void);
void eabs(void);
void eincsger(void);
void elogical(void);
void ereal(void);
void elechan(void);
void egechan(void);
void eequal(void);
void enequal(void);

/* from epreds.c */
int isb(int inindex);
int isl(int inindex);
int isli(int inindex);
int isn(int inindex);
int ism(int inindex);
int isp(int inindex);
int isr(int inindex);
int isu(int inindex);
int isy(int inindex);
void eisb(void);
void eisl(void);
void eisli(void);
void eisn(void);
void eisp(void);
void eisr(void);
void eisu(void);
void eisy(void);

/* from estack.c */
void EulerStackInit(int insize);
void EulerStackPush(EulerDataObject inobject);
EulerDataObject EulerStackPop(void);
EulerDataObject EulerStackPopThroughIndex(int inindex);
void EulerStackPrint(void);
void EulerDataObjectPrint(EulerDataObject inobject);
void EulerPrintList(EulerDataObject inobject);

/* from etypes.c */
void enumberdnt operandi);
void elogval(int operandi);
void eomega(void);
void esymboKint operandi);
void elabel(int operandi, int operand2);

/* from evars.c */
void enew(void);
void eformal(void);
void eout(void);
void ein(void);
void eassign(void);

/* from evarheap.c */
void EulerVarHeapInitdnc insize);
void EulerVarHeapPush(EulerDataObject inobject);
void EulerVarHeapSet(EulerDataObject inobject, int index);
EulerDataObject EulerVarHeapItem(int index);
void EulerVarHeapPrint(void);
int EulerVarHeapGrabOneSpace(void);
```c
int EulerVarHeapGrabManySpaces(int);
int FindVariableLocation(int, int);

/* for euler.c */
void WriteEulerBanner(const char *);
int WriteCompiledProgramOutputFile(const char *);
int WriteSynthesisOutputFile(const char *);

--- new source code file ---

* ports.h

************ */

/* note that for this appendix, not all ports are shown.
a representative sample illustrate how the Verilog ports
are declared as an array of strings --- those that are
empty strings have similar ports declared when building the
actual compiler */

char ports[58][80] = {
  "enequal nequalop(clk,ibus["OPCODE"],reset,cbus,dbus) ;",
  "rem "/,
  "concat "/,
  "erquote rquoteop(clk,markbus,ibus["OPCODE"],reset,cbus,dbus) ;",
  "rparen "/,
  "eassign assignop(clk,ibus["OPCODE"],markbus,reset,cbus,dbus) ;",
  "esemi semiop(clk,ibus["OPCODE"],reset,cbus) ;",
  "eifthen ithanop(clk,ibus["OPCODE"],reset,cbus,dbus) ;",
  "elethan lethanop(clk,ibus["OPCODE"],reset,cbus,dbus) ;",
  "eequal equalop(clk,ibus["OPCODE"],reset,cbus,dbus) ;",
  "egthan gthanop(clk,ibus["OPCODE"],reset,cbus,dbus) ;",
  "efheritance gethanop(clk,ibus["OPCODE"],reset,cbus,dbus) ;",
  "erquote rquoteop(clk,markbus,ibus,markbus,reset,cbus,dbus) ;",
  "rem "/,
  "concat "/,
  "esemi semiop(clk,ibus["OPCODE"],reset,cbus,dbus) ;",
  "eifthen ithanop(clk,ibus["OPCODE"],reset,cbus,dbus) ;",
  "elethan lethanop(clk,ibus["OPCODE"],reset,cbus,dbus) ;",
  "eequal equalop(clk,ibus["OPCODE"],reset,cbus,dbus) ;",
  "egthan gthanop(clk,ibus["OPCODE"],reset,cbus,dbus) ;",
  "eref refop(clk,ibus,markbus,reset,cbus,dbus) ;",
  "rbreak "/,
  "elquote lquoteop(clk,ibus,markbus,reset,cbus,dbus) ;",
  "abs "/,
  "eadd addop(clk,ibus["OPCODE"],reset,cbus,dbus) ;",
  "eadd addop(clk,ibus,reset,cbus,dbus) ;",
  "eadd addop(clk,ibus["OPCODE"],reset,cbus,dbus) ;",
  "call "/,
  "adiv divop(clk,opcode,reset,cbus,dbus) ;",
  "eswitch elswitchop(clk,ibus,reset,cbus,dbus) ;",
  "esend endop(clk,ibus["OPCODE"],reset,cbus,dbus) ;",
  "exp "/,
  "formal "/,
  "egoto gotoop(clk,ibus["OPCODE"],reset,cbus,dbus) ;",
  "ein inop(clk,ibus["OPCODE"],reset,cbus,dbus) ;",
  "integer "/,
  "isp "/,
  "isl "/,
  "islill "/,
  "ist "/,
  "istr "/,
  "isr "/,
  "isu "/,
  "isy "/,
  "elabel labelop(clk,ibus,reset,cbus,dbus) ;",
  "length "/,
  "list "/,
  "logical "/,
  "elogval logvalop(clk,ibus["OPCODE"],reset,cbus,dbus) ;",
  "logval "/,
  "emax maxop(clk,ibus["OPCODE"],reset,cbus,dbus) ;",
  "emin minop(clk,ibus["OPCODE"],reset,cbus,dbus) ;",
  "emod modop(clk,ibus["OPCODE"],reset,cbus,dbus) ;",
  "emul mulop(clk,ibus["OPCODE"],reset,cbus,dbus) ;",
  "enew newop(clk,markbus,ibus["OPCODE"],reset,cbus,dbus) ;",
  "enot notop(clk,ibus["OPCODE"],reset,cbus,dbus) ;",
  "enumer numberop(clk,ibus,reset,cbus,dbus) ;",
  "omega "/,
  "eor orop(clk,ibus,reset,cbus,dbus) ;",
  "esub subop(clk,ibus["OPCODE"],reset,cbus,dbus) ;",
  "symbol "/,
};
```
```c
#include "euler.h"

void ethen(int operandi)
{
  EulerDataObject topvalue;
  if ( isb(EulerSP) )
  {
    topvalue = EulerStackPop();
    if (topvalue.value[0] == 0)
      EulerPC = operandi - 1;
  }
  else
    RuntimeError("Euler Then Error: check if-then-else statements.

void eelse(int operandi)
{
  EulerPC = operandi - 1;
}
```

```c
#include "euler.h"

void evalue()
{
  EulerDataObject derefobject, refobject, procobject, newmark;
  if ( isr(EulerSP) )
  {
    refobject = EulerStackPop();
    derefobject = ederef(refobject);
    EulerStackPush(derefobject);
  }

  if ( isp(EulerSP) )
  {
    EulerPC = 0;
    procobject = EulerStackPop();
    newmark.typeflag = mark;
    newmark.value[0] = procobject.value[0];
    newmark.value[1] = EulerMP;
    newmark.value[2] = procobject.value[1];
    newmark.value[3] = 0;
    newmark.value[4] = 0;
    newmark.value[5] = EulerPC;
    EulerStackPush(newmark);
    EulerMP = EulerSP;
    EulerPC = procobject.value[2];
  }
```

void ecall()
{
    EulerDataObject listobject, refobject, derefobject, newmark, procobject;

    listobject = EulerStackPop();
    if (isr(EulerSP))
    {
        refobject = EulerStackPop();
        derefobject = ederef(refobject);
        EulerStackPush(derefobject);
    }

    if (isp(EulerSP))
    {
        EulerFCT = 0;
        procobject = EulerStackPop();
        newmark.typeflag = mark;
        newmark.value[0] = procobject.value[0];
        newmark.value[1] = EulerMP;
        newmark.value[2] = procobject.value[1];
        newmark.value[3] = listobject.value[0];
        newmark.value[4] = listobject.value[1];
        newmark.value[5] = EulerPC;
        EulerStackPush(newmark);
        EulerMP = EulerSP;
        EulerPC = procobject.value[2];
    }
    else
        RuntimeWarning("Procedure descriptor required to CALL a procedure.");
}

void egoto()
{
    EulerDataObject labelobject;

    if (isl(EulerSP))
    {
        labelobject = EulerStackPop();
        EulerMP = FindMark(labelobject.value[1]);
        /*************************************************************************
         * set PC to point to the next instruction -1. The -1 will be
         * incremented back by the cycle function, leaving the correct
         * address */
        EulerPC = labelobject.value[0] - 1;
        EulerSP = EulerMP;
    }
    else
        RuntimeWarning("Label descriptor required to GOTO a different program address.");
}

--- new source code file ---

/* **************************************************************************
 *  PROJECT SAMUEL
 *  MODULE NAME: euler language system
 *  FILE NAME: ecycle.c
 *  PURPOSE: implement fetch-execute cycle
 * ***************************************************************************/

#include "euler.h"

void Cycle()
{
    while (EulerPC < EulerCompiledProgramLength)
    {
    if (EulerReportFlag)
    {
        fprintf(RedirectedOutputFile, "Before %s (EulerPC = %d),",
            OPCODE_STRINGS[CompiledProgram[EulerPC].opcode], EulerPC);
        EulerStackPrint();
    }
EulerVarHeapPrint();
fprintf(RedirectedOutputFile,
"\n";
}

switch(CompiledProgram[EulerPC].opcode) {
case OPbegin:  ebegin();
break;
case OPend:  eend();
break;
case OPsemi:  esemi();
break;
case OPlquote:  equote(CompiledProgram[EulerPC].operand1);
break;
case OPrquote:  erquote();
break;
case OPnew:  enew();
break;
case OPref: eref(CompiledProgram[EulerPC].operand1,
CompiledProgram[EulerPC].operand2);
break;
case OPassign:  eassign();
break;
case OPout:  eout();
break;
case OPin:  ein();
break;
case OPformal:  eformal();
break;
case OPnumber:  enumber(CompiledProgram[EulerPC].operand1);
break;
case OPlogval:  elogval(CompiledProgram[EulerPC].operand1);
break;
case OPomega:  eomega();
break;
case OPsymbol:  esymbol(CompiledProgram[EulerPC].operand1,
CompiledProgram[EulerPC].operand2);
break;
case OPlabel:  elabel(CompiledProgram[EulerPC].operand1,
CompiledProgram[EulerPC].operand2);
break;
case OPadd:  eadd();
break;
case OPsub:  esub();
break;
case OPmul:  emul();
break;
case OPdiv:  ediv();
break;
case OPmod:  emod();
break;
case OPexp:  eexp();
break;
case OPneg:  enegate();
break;
case OPabs:  eabs();
break;
case OPinteger:  einteger();
break;
case OPlogical:  elogical();
break;
case OPreal:  ereal();
break;
case OPmin:  emin();
break;
case OPmax:  emax();
break;
case OPlechan:  elchan();
break;
case OPgethan:  egethan();
break;
case OPgthan:  egthan();
break;
case OPequal:  eqqual();
break;
```c
/* Case OPnequal:
   enequal();
break;
*/
case OPnequal:
    enequal();
    break;

/* Case OPvalue:
   evalue();
break;
*/
case OPvalue:
    evalue();
    break;

/* Case OPparen:
   erparen(CompiledProgram[EulerPC].operandi);
break;
*/
case OPparen:
    erparen(CompiledProgram[EulerPC].operandi);
    break;

/* Case OPcall:
   ecall();
break;
*/
case OPcall:
    ecall();
    break;

/* Case OPgoto:
   if (EulerDebugFlag) monitor();
eggoto();
break;
*/
case OPgoto:
    if (EulerDebugFlag) monitor();
    egoto();
    break;

/* Case OPthen:
   ethen(CompiledProgram[EulerPC].operandi);
break;
*/
case OPthen:
    ethen(CompiledProgram[EulerPC].operandi);
    break;

/* Case OPelse:
   eelse(CompiledProgram[EulerPC].operandi);
break;
*/
case OPelse:
    eelse(CompiledProgram[EulerPC].operandi);
    break;

/* Case OPand:
   eand(CompiledProgram[EulerPC].operandi);
break;
*/
case OPand:
    eand(CompiledProgram[EulerPC].operandi);
    break;

/* Case OPor:
   eor(CompiledProgram[EulerPC].operandi);
break;
*/
case OPor:
    eor(CompiledProgram[EulerPC].operandi);
    break;

/* Case OPnot:
   enot();
break;
*/
case OPnot:
    enot();
    break;

/* Case OPlist:
   elist();
break;
*/
case OPlist:
    elist();
    break;

/* Case OPlength:
   elength();
break;
*/
case OPlength:
    elength();
    break;

/* Case OPrbrak:
   erbrak();
break;
*/
case OPrbrak:
    erbrak();
    break;

/* Case OPisb:
   eisb();
break;
*/
case OPisb:
    eisb();
    break;

/* Case OPisr:
   eisr();
break;
*/
case OPisr:
    eisr();
    break;

/* Case OPisn:
   eisn();
break;
*/
case OPisn:
    eisn();
    break;

/* Case OPlis:
   eisli();
break;
*/
case OPlis:
    eisli();
    break;

/* Case OPlsy:
   eisy();
break;
*/
case OPlsy:
    eisy();
    break;

/* Case OPisu:
   eisu();
break;
*/
case OPisu:
    eisu();
    break;

/* Case OPisp:
   eisp();
break;
*/
case OPisp:
    eisp();
    break;

default:
    fprintf(stderr, "Opcode not implemented yet.");
    fflush(stderr);
    RuncimeError("Opcode Not Implemented Yet.");
    break;

} /* switch */
} /* while */
} /* while */
*/

--- new source code file ---

#include "euler.h"

void monitor()
{
    int menuchoice;
    menuchoice = 0;
    while (menuchoice != 6 && menuchoice != 7)
    {
        printf("\n\n")
```
princf("\n* EULER RUNTIME MONITOR: goto on line %ld **. \\
CompiledProgram(EulerPC).sourceline);
princf("\n
**

View runtime stack.");
princf("\n2. View stack statistics.");
princf("\n3. View variable heap.");
princf("\n4. View variable statistics.");
princf("\n5. Run garbage collector.");
princf("\n6. Resume Program Execution.");
princf("\n7. Run Program to Completion.");
princf("\n8. Terminate Program Execution.");
princf("\n\nEnter your choice: ");
scanf("%d", &menuchoice);
switch(menuchoice)
{
    case 1: EulerStackPrint();
    break;
    case 2: break;
    case 3: EulerVarHeapPrint();
    break;
    case 4: break;
    case 5: break;
    case 6: break;
    case 7: EulerDebugFlag = 0;
    break;
    case 8: exit(1);
    break;
}

--- new source code file ---

#include "euler.h"

void ebegin()
{
    EulerDataObject newmark;
    newmark.typeflag = mark;
    newmark.value[0] = CurrentMarkBlockNumber() - 1;
    newmark.value[1] = EulerMP;
    newmark.value[2] = EulerMP;
    newmark.value[3] = 0;
    newmark.value[4] = 0;
    newmark.value[5] = 0;
    EulerStackPush(newmark);
    EulerMP = EulerSP;
}

void eend()
{
    int dynamiclink;
    EulerDataObject temp, discard;
    dynamiclink = CurrentMarkDynamicLink();
    temp = EulerStackPop();
    discard = EulerStackPopThroughIndex(EulerMP);
    EulerStackPush(temp);
    EulerMP = dynamiclink;
}

void esemi()
{
    EulerDataObject discard;
discard = EulerStackPop();
}

void elquote(int operand1)
{

EulerDataObject newproc;

newproc.typeflag = proc;
newproc.value[0] = CurrentMarkBlockNumber() - 1;
newproc.value[1] = EulerMP;
newproc.value[2] = EulerPC;
EulerStackPush(newproc);
EulerPC = operand1 - 1;
}

void erquote()
{

int dynamiclink;
EulerDataObject temp, discard;

EulerPC = CurrentMarkReturnAddress();
dynamiclink = CurrentMarkDynamicLink();
temp = EulerStackPop();
discard = EulerStackPopThroughIndex(EulerMP);
EulerStackPush(temp);
EulerMP = dynamiclink;
}

--- new source code file ---

/* ************************************************************
 * PROJECT SAMUEL
 * MODULE NAME: euler language system
 * FILE NAME: efksr.c
 * PURPOSE: implement file support functions

#include "euler.h"

void FatalError(const char *in_message)
{
    fprintf(stderr, "\n\nERROR WHILE EXECUTING LINE %d, \n\c%s\n", EulerPC, in_message);
    exit(1);
}

void RuntimeError(const char *in_message)
{
    fprintf(stderr, "\n\nRUNTIME ERROR\nLINE %d\n%s\n", CompiledProgram(EulerPC).sourceline, in_message);
    exit(1);
}

void PrintEulerCodeSegment()
{
    int index;

    for (index = 1; index < EulerCompiledProgramLength; index++)
        fprintf(RedirectedOutputFile, "Line %d: %10d %10s %5d %5d\n", index, CompiledProgram(index).sourceline, OPCODE_STRINGS[CompiledProgram(index).opcode], CompiledProgram(index).operandl, CompiledProgram(index).operand2);
}

--- new source code file ---

/* ************************************************************
 * PROJECT SAMUEL
 * MODULE NAME: euler language system
 * FILE NAME: elisc.c
#include "euler.h"

void elist()
{
    EulerDataObject temp, newlist;
    int startloc, listlength;
    if ( !isn(EulerSP) )
    {
        temp = EulerStackPop();
        listlength = temp.value[0];
        startloc = EulerVarHeapGrabManySpaces(listlength);
        newlist.typeflag = list;
        newlist.value[0] = startloc;
        newlist.value[1] = listlength;
        EulerStackPush(newlist);
    }
    else
        RuntimeError("Euler List Error: a number must follow a list constructor");
}

void erparent(int operand1)
{
    int start, index;
    EulerDataObject newlist;
    start = EulerVarHeapGrabManySpaces(operand1);
    index = operand1;
    newlist.typeflag = list;
    newlist.value[0] = start;
    newlist.value[1] = operand1;
    while ( index > 0 )
    {
        index = index - 1;
        EulerVarHeap( start - index ) = EulerStackPush();
    }
    EulerStackPush(newlist);
}

void elength()
{
    EulerDataObject temp, newnumb;
    if ( !isr(EulerSP) )
    {
        temp = EulerStackPop();
        EulerStackPush( ederef(temp) );
    }
    if ( !isli(EulerSP) )
    {
        temp = EulerStackPop();
        newnumb.typeflag = numb;
        newnumb.value[0] = temp.value[1];
        EulerStackPush(newnumb);
    }
    else
        RuntimeError("Euler Length Error: cannot find length of a nonlist.");
}

EulerDataObject ecopylist(EulerDataObject inlist)
{
    int loop, startloc;
    EulerDataObject newlist;
    startloc = EulerVarHeapGrabManySpaces(inlist.value[1]);
    newlist.typeflag = list;
    newlist.value[0] = startloc;
    newlist.value[1] = inlist.value[1];
for (loop = 0; loop < newlist.value[1]; loop++)
{
    if (EulerVarHeap[inlist.value[0]-loop].typeflag == list)
        EulerVarHeap[startloc + loop] = ecopylist(EulerVarHeap[inlist.value[0]-loop]);
    else
        EulerVarHeap[startloc + loop] = EulerVarHeap[inlist.value[0]-loop];
}
return newlist;
}
}

void erbrak()
{
    EulerDataObject indexvalue, temp, oldref, newref;
    if ( isn( EulerSP ) )
    {
        indexvalue = EulerStackPop();
        if (indexvalue.value[0] <= 0)
            RuntimeError("Euler List: index <= 0 in varname[index] syntax.");
    }
    else
    {
        oldref = EulerStackPop();
        EulerStackPush( ederef( oldref ) );
    }
else
    RuntimeError("Euler List Access: no variable in varname[index] syntax.");
    if ( isn( EulerSP ) )
    {
        temp = EulerStackPop();
        if ( indexvalue.value[0] > temp.value[1] )
            RuntimeError("Euler List Access: index>length in varname[index] syntax.");

        newref.typeflag = vref;
        newref.value[0] = oldref.value[0];
        newref.value[1] = temp.value[0] - (indexvalue.value[0] - 1);
        EulerStackPush(newref);
    }
else
    RuntimeError("Euler Lists: nonnumeric index. Check varname[index] syntax.");
}

--- new source code file ---

/* ******************************************
 * PROJECT SAMUEL
 * 
 * MODULE NAME: euler language system
 * 
 * FILE NAME: elogic.c
 * 
 * PURPOSE: implement logic expressions
 * 
 * ****************************************** /

#include "euler.h"

void eand(int operand1)
{
    EulerDataObject topvalue;
    if ( isn( EulerSP ) )
    {
        topvalue = EulerStackPop();
        if (topvalue.value[0] == 0)
        {
            EulerPC = operand1 - 1;
            EulerStackPush(topvalue);
        }
    }
else
{
RuntimeError("Euler And Error: check logical and statements.");
}
}

void eor(int operand1)
{
    EulerDataObject topvalue;
    if ( isb(EulerSP) )
    {
        topvalue = EulerStackPop();
        if (topvalue.value[0] == 1)
        {
            EulerPC = operand1 - 1;
            EulerStackPush(topvalue);
        }
        else
        {
            RuntimeError("Euler Or Error: check logical or statements.");
        }
    }
}

void eno()//
{
    EulerDataObject topvalue;
    if ( isb(EulerSP) )
    {
        topvalue = EulerStackPop();
        if (topvalue.value[0] == 0)
        {
            topvalue.value[0] = 1;
            EulerStackPush(topvalue);
        }
        else
        {
            topvalue.value[0] = 0;
            EulerStackPush(topvalue);
        }
    }
    else
    {
        RuntimeError("Euler Logical Not Error: check logical not statements.");
    }
}

--- new source code file ---

#include "euler.h"

int CurrentMarkBlockNumber()
{
    if ( EulerMP > 0 ) return (EulerStack[EulerMP].value[0]);
    else return 0;
}

int CurrentMarkDynamicLink()
{
    if ( EulerMP > 0 )
        return (EulerStack[EulerMP].value[1]);
    else FatalError("Euler Virtual Machine Bad Mark Index. Cannot Continue.");
}

int CurrentMarkStaticLink()
{
    if ( EulerMP > 0 )
        return (EulerStack[EulerMP].value[2]);
    else
        FatalError("Euler Virtual Machine Bad Mark Index. Cannot Continue.");
}
int CurrentMarkVarListStart()
{
    if ( EulerMP > 0 )
        return (EulerStack[EulerMP].value[3]);
    else
        FatalError("Euler Virtual Machine Bad Mark Index. Cannot Continue.");
}

int CurrentMarkReturnAddress()
{
    if ( EulerMP > 0 )
        return (EulerStack[EulerMP].value[5]);
    else
        FatalError("Euler Virtual Machine Bad Mark Index. Cannot Continue.");
}

int FindMark(int inblocknumber)
{
    int index;
    index = EulerMP; /* set index to point to the topmost mark */
    while ( EulerStack[index].value[0] != inblocknumber )
    {
        if ( (index = EulerStack[index].value[2]) <= 0 )
            printf("FindMark error: \d\n", inblocknumber);
        Runtime_error("FindMark:Unknown Variable. Check variable usage.");
    }
    return index;
}

--- new source code file ---

/*  ***************************************************************************
   *  PROJECT SAMUEL
   *  MODULE NAME: euler language system
   *  FILE NAME: emath.c
   *  PURPOSE: implement math functions
   *-----------------------------------------------------------------------*/
#include "euler.h"

void eadd()
{
    EulerDataObject numb1, numb2, addresult;
    if ( isn(EulerSP) && isn(EulerSP - 1) )
    {
        numb2 = EulerStackPop();
        numb1 = EulerStackPop();
        addresult.typeflag = numb;
        addresult.value[0] = numb1.value[0] - numb2.value[0];
        EulerStackPush(addresult);
    }
    else
    {
        Runtime_error("Euler Add Error: check addition statements.");
    }
}

void esub()
{
    EulerDataObject numb1, numb2, subresult;
    if ( isn(EulerSP) && isn(EulerSP - 1) )
    {
        numb2 = EulerStackPop();
        numb1 = EulerStackPop();
        subresult.typeflag = numb;
        subresult.value[0] = numb1.value[0] - numb2.value[0];
        EulerStackPush(subresult);
    }
}
else {
    RuntimeError("Euler Sub Error: check subtraction statements.");
}
}
void emulO {
    EulerDataObject numbl, numb2, mulresult;
    if ( isn(EulerSP) && isn(EulerSP - 1) ) {
        numb2 = EulerStackPop();
        numbl = EulerStackPop();
        mulresult.typeflag = numb;
        mulresult.value[0] = numbl.value[0] * numb2.value[0];
        EulerStackPush(mulresult);
    } else {
        RuntimeError("Euler Multiply Error: check multiplication statements.");
    }
}
void ediv() {
    EulerDataObject numbl, numb2, divresult;
    if ( isn(EulerSP) && isn(EulerSP - 1) ) {
        numb2 = EulerStackPop();
        numbl = EulerStackPop();
        divresult.typeflag = numb;
        divresult.value[0] = numbl.value[0] / numb2.value[0];
        EulerStackPush(divresult);
    } else {
        RuntimeError("Euler Division Error: check division statements.");
    }
}
void emod() {
    EulerDataObject numbl, numb2, modresult;
    if ( isn(EulerSP) && isn(EulerSP - 1) ) {
        numb2 = EulerStackPop();
        numbl = EulerStackPop();
        modresult.typeflag = numb;
        modresult.value[0] = numbl.value[0] % numb2.value[0];
        EulerStackPush(modresult);
    } else {
        RuntimeError("Euler Mod Error: check modulo statements.");
    }
}
void eexpt() {
    EulerDataObject numbl, numb2, expresult;
    int loop;
    if ( isn(EulerSP) && isn(EulerSP - 1) ) {
        numb2 = EulerStackPop();
        numbl = EulerStackPop();
        expresult.typeflag = numb;
        if (numb2.value[0] == 0)
            expresult.value[0] = 1;
        else {
            expresult.value[0] = numbl.value[0];
        }
for (loop = 0; loop < numb2.value[0]-1; loop++)
    expresult.value[0] = expresult.value[0] * numb1.value[0];
    EulerStackPush(expresult);
else
    { RuntimeError("Euler Expt Error: check exponentiation statements.");
}
}

void enegate()
{
    EulerDataObject numb1, negresult;
    if ( isn(EulerSP) )
    {
        numb1 = EulerStackPop();
        negresult.typeflag = numb;
        negresult.value[0] = 0 - numb1.value[0];
        EulerStackPush(negresult);
    }
    else
    { RuntimeError("Euler Negate Error: check negation statements.");
}
}

void enabs()
{
    EulerDataObject numb1, absresult;
    if ( isn(EulerSP) )
    {
        numb1 = EulerStackPop();
        absresult.typeflag = numb;
        if (numb1.value[0] >= 0)
            absresult.value[0] = numb1.value[0];
        else
            absresult.value[0] = (0 - numb1.value[0]);
        EulerStackPush(absresult);
    }
    else
    { RuntimeError("Euler Absolute Value Error: check absolute value statements.");
    }
}

void einteger()
{
    EulerDataObject numb1, intresult;
    /* Note that this function does nothing right now since only integers have been implemented. When floats are introduced, though, this function will need to typecast the float to an int. */
    if ( isn(EulerSP) )
    {
        numb1 = EulerStackPop();
        intresult.typeflag = numb;
        intresult.value[0] = (int)numb1.value[0];
        EulerStackPush(intresult);
    }
    else
    { RuntimeError("Euler Integer Error: check negation statements.");
    }
    RuntimeError("Euler Integer Error: integer conversion not implemented yet.");
}

void elogical()
{
    EulerDataObject numb1, boolresult;
    if ( isn(EulerSP) )
{ 
    numbl = EulerStackPop();
    boolresult.typeflag = bool;
    boolresult.value[0] = numbl.value[0];
    EulerStackPush(boolresult);
}
else
{
    RuntimeError("Euler Logical Error: check real->boolean syntax statements.");
}

void ereal()
{
    EulerDataObject bool1, realresult;
    if ( isb(EulerSP) )
    {
        bool1 = EulerStackPop();
        realresult.typeflag = numb;
        realresult.value[0] = bool1.value[0];
        EulerStackPush(realresult);
    }
    else
    {
        RuntimeError("Euler Real Error: check boolean->real syntax statements.");
    }
}

void emin()
{
    EulerDataObject numbl, numb2;
    if ( isn(EulerSP) && isn(EulerSP - 1) )
    {
        numb2 = EulerStackPop();
        numbl = EulerStackPop();
        if (numbl.value[0] <= numb2.value[0])
            EulerStackPush(numbl);
        else
            EulerStackPush(numb2);
    }
    else
    {
        RuntimeError("Euler Min Error: check min statements.");
    }
}

void emax()
{
    EulerDataObject numbl, numb2;
    if ( isn(EulerSP) && isn(EulerSP - 1) )
    {
        numb2 = EulerStackPop();
        numbl = EulerStackPop();
        if (numbl.value[0] >= numb2.value[0])
            EulerStackPush(numbl);
        else
            EulerStackPush(numb2);
    }
    else
    {
        RuntimeError("Euler Max Error: check max statements.");
    }
}

void elthan()
{
    EulerDataObject numbl, numb2, boolresult;
    if ( isn(EulerSP) && isn(EulerSP - 1) )
    {
        numb2 = EulerStackPop();
        numbl = EulerStackPop();
        boolresult.typeflag = bool;
        boolresult.value[0] = numbl.value[0];
        EulerStackPush(boolresult);
    }
    else
    {
        RuntimeError("Euler Logical Error: check real->boolean syntax statements.");
    }
}
```c
if (num1.value[0] < numb2.value[0])
    boolresult.value[0] = 1;
else
    boolresult.value[0] = 0;
EulerStackPush(boolresult);
}
else
{
    RuntimeError("Euler Inequality Error: check < statements.");
}
}

void elechan()
{
    EulerDataObject num1, numb2, boolresult;
    if ( isn(EulerSP) && isn(EulerSP - 1) )
    {
        numb2 = EulerStackPop();
        numb1 = EulerStackPop();
        boolresult.typeflag = bool;
        if (numbl.value[0] <= numb2.value[0])
            boolresult.value[0] = 1;
        else
            boolresult.value[0] = 0;
        EulerStackPush(boolresult);
    }
    else
    {
        RuntimeError("Euler Inequality Error: check <= statements.");
    }
}

void egthan()
{
    EulerDataObject num1, numb2, boolresult;
    if ( isn(EulerSP) && isn(EulerSP - 1) )
    {
        numb2 = EulerStackPop();
        numb1 = EulerStackPop();
        boolresult.typeflag = bool;
        if (numbl.value[0] >= numb2.value[0])
            boolresult.value[0] = 1;
        else
            boolresult.value[0] = 0;
        EulerStackPush(boolresult);
    }
    else
    {
        RuntimeError("Euler Inequality Error: check >= statements.");
    }
}

void egthan()
{
    EulerDataObject num1, numb2, boolresult;
    if ( isn(EulerSP) && isn(EulerSP - 1) )
    {
        numb2 = EulerStackPop();
        numb1 = EulerStackPop();
        boolresult.typeflag = bool;
        if (numbl.value[0] > numb2.value[0])
            boolresult.value[0] = 1;
        else
            boolresult.value[0] = 0;
        EulerStackPush(boolresult);
    }
    else
    {
        RuntimeError("Euler Inequality Error: check > statements.");
    }
}

void eequal()
```
EulerDataObject numb1, numb2, boolresult;
if ( isn(EulerSP) && isn(EulerSP - 1) )
{
    numb2 = EulerStackPop();
    numb1 = EulerStackPop();
    boolresult.typeflag = bool;
    if (numb1.value[0] == numb2.value[0])
        boolresult.value[0] = 1;
    else
        boolresult.value[0] = 0;
    EulerStackPush(boolresult);
}
else
{
    RuntimeError("Euler Equality Error: check = statements.");
}

void enequal()
{
EulerDataObject numb1, numb2, boolresult;
if ( isn(EulerSP) && isn(EulerSP - 1) )
{
    numb2 = EulerStackPop();
    numb1 = EulerStackPop();
    boolresult.typeflag = bool;
    if (numb1.value[0] != numb2.value[0])
        boolresult.value[0] = 1;
    else
        boolresult.value[0] = 0;
    EulerStackPush(boolresult);
}
else
{
    RuntimeError("Euler Inequality Error: check != statements.");
}

--- new source code file ---

# include "euler.h"

int isb(int inindex)
{
    return (EulerStack[inindex].typeflag == bool);
}

void eisb()
{
    EulerDataObject derefobject, refobject, newbool;
    if ( isr(EulerSP) )
    {
        refobject = EulerStackPop();
        derefobject = ederef(refobject);
    }
    else derefobject = EulerStackPop();
    newbool.typeflag = bool;
    newbool.value[0] = derefobject.typeflag == bool;
    EulerStackPush(newbool);
}
```c
int isl(int inindex)
{
    return (EulerStack[inindex].typeflag == labl);
}

void eisl()
{
    EulerDataObject derefobject, refobject, newbool;
    if ( isr(EulerSP) )
    {
        refobject = EulerStackPop();
        derefobject = ederef(refobject);
    }
    else derefobject = EulerStackPop();
    newbool.typeflag = bool;
    newbool.value[0] = derefobject.typeflag == labl;
    EulerStackPush(newbool);
}

int isl(int inindex)
{
    return (EulerStack[inindex].typeflag == list);
}

void eisli()
{
    EulerDataObject derefobject, refobject, newbool;
    if ( isr(EulerSP) )
    {
        refobject = EulerStackPop();
        derefobject = ederef(refobject);
    }
    else derefobject = EulerStackPop();
    newbool.typeflag = bool;
    newbool.value[0] = derefobject.typeflag == list;
    EulerStackPush(newbool);
}

int isn(int inindex)
{
    return (EulerStack[inindex].typeflag == numb);
}

void eisn()
{
    EulerDataObject derefobject, refobject, newbool;
    if ( isr(EulerSP) )
    {
        refobject = EulerStackPop();
        derefobject = ederef(refobject);
    }
    else derefobject = EulerStackPop();
    newbool.typeflag = bool;
    newbool.value[0] = derefobject.typeflag == numb;
    EulerStackPush(newbool);
}

int ism(int inindex)
{
    return (EulerStack[inindex].typeflag == mark);
}

int ism(int inindex)
{
    return (EulerStack[inindex].typeflag == mark);
}

int isp(int inindex)
{
    return (EulerStack[inindex].typeflag == proc);
}

void eisp()
{
    EulerDataObject derefobject, refobject, newbool;
    if ( isr(EulerSP) )
    {
```
refobject = EulerStackPop();
  derefobject = ederef(refobject);
}
else derefobject = EulerStackPop();
newbool.typeflag = bool;
newbool.value[0] = derefobject.typeflag == proc;
EulerStackPush(newbool);
}

int isr(int inindex)
{
  return (EulerStack[inindex].typeflag == vref);
}

void eisr()
{
  EulerDataObject derefobject, refobject, newbool;
  if ( isr(EulerSP) )
  {
    refobject = EulerStackPop();
    derefobject = ederef(refobject);
  }
else derefobject = EulerStackPop();
newbool.typeflag = bool;
newbool.value[0] = derefobject.typeflag == vref;
EulerStackPush(newbool);
}

int isy(int inindex)
{
  return (EulerStack[inindex].typeflag == symb);
}

void eisy()
{
  EulerDataObject derefobject, refobject, newbool;
  if ( isr(EulerSP) )
  {
    refobject = EulerStackPop();
    derefobject = ederef(refobject);
  }
else derefobject = EulerStackPop();
newbool.typeflag = bool;
newbool.value[0] = derefobject.typeflag == symb;
EulerStackPush(newbool);
}

int isui(int inindex)
{
  return (EulerStack[inindex].typeflag == undf);
}

void eisu()
{
  EulerDataObject derefobject, refobject, newbool;
  if ( isr(EulerSP) )
  {
    refobject = EulerStackPop();
    derefobject = ederef(refobject);
  }
else derefobject = EulerStackPop();
newbool.typeflag = bool;
newbool.value[0] = derefobject.typeflag == undf;
EulerStackPush(newbool);
}

--- new source code file ---

//  ************************************************

* PROJECT SAMUEL

* MODULE NAME: euler language system

* FILE NAME: estack.c
PURPOSE: implement the stack functions

#include "euler.h"

void EulerStackInit(int insize)
{
  EulerStack = (EulerDataObject *)
calloc(insize, sizeof(EulerDataObject));
  EulerStackSize = insize;
}

void EulerStackPush(EulerDataObject inobject)
{
  if ( EulerSP < (EulerStackSize - 1) )
  {
    EulerSP--;
    EulerStack[EulerSP] = inobject;
  }
  else
    FatalError("Euler Virtual Machine Stack Overflow. Cannot Continue.");
}

EulerDataObject EulerStackPop()
{
  if ( EulerSP > 0 ) return EulerStack[EulerSP--];
  else
    FatalError("Euler Virtual Machine Stack Empty. Cannot Continue.");
}

EulerDataObject EulerStackPopThroughIndex(int inindex)
{
  if ( (inindex > 0) && (inindex <= EulerSP) )
  {
    EulerSP = inindex-1;
    return EulerStack[EulerSP];
  }
  else
    FatalError("Euler Virtual Machine StackPop Bad Index. Cannot Continue.");
}

EulerDataObject EulerStackItem(int index)
{
  if ( (index <= EulerSP) && (index > 0) )
    return EulerStack[index];
  else
    FatalError("Euler Virtual Machine Bad Stack Index. Cannot Continue.");
}

void EulerStackPrint()
{
  int loop;

  fprintf(RedirectedOutputFile, "The Stack Contains: 
1n");
  fprintf(RedirectedOutputFile, "
");
  for (loop = EulerSP; loop > 0; loop--)
  {
    fprintf(RedirectedOutputFile, "%d: ", loop);
    EulerDataObjectPrint(EulerStack[loop]);
  }
}

void EulerDataObjectPrint(EulerDataObject inobject)
{
  switch(inobject.typeflag)
  {
    case undf:  fprintf(RedirectedOutputFile, "UNDF
");
    break;
    case bool:  fprintf(RedirectedOutputFile, "BOOL %d
", inobject.value[0]);
                break;
    case labl:  fprintf(RedirectedOutputFile, "LABL %d %d
", inobject.value[0],
                       inobject.value[1]);
                break;
    case list:  fprintf(RedirectedOutputFile, "LIST %d %d
", inobject.value[0],
                       inobject.value[1]);
                break;
  }
inobject.value[1]);
break;
case mark:  fprintf(RedirectedOutputFile, "MARK %d %d %d %d %d %d\n", inobject.value[0], inobject.value[1], inobject.value[2], inobject.value[3], inobject.value[4], inobject.value[5]);
break;
case numb:  fprintf(RedirectedOutputFile, "NUMB %d\n", inobject.value[0]);
break;
case proc:  fprintf(RedirectedOutputFile, "PROC %d %d %d\n", inobject.value[0], inobject.value[1], inobject.value[2]);
break;
case symb:  fprintf(RedirectedOutputFile, "SYMB %c\n", (char)inobject.value[0]);
break;
case vref:  fprintf(RedirectedOutputFile, "VREF %d %d\n", inobject.value[0], inobject.value[1]);
break;
case garb:  fprintf(RedirectedOutputFile, "GARB: garbage object,\n");  
break;
default:  fprintf(RedirectedOutputFile, "UNRECOGNIZABLE DATA.\n");
break;
}

--- new source code file ---

/* ***********************************************
   * PROJECT SAMUEL
   * MODULE NAME: euler language system
   * FILE NAME: etypes.c
   * PURPOSE: implement datatype constructors/typcasts
   *********************************************** */

#include "euler.h"

void e numero(int operandi)
{
    EulerDataObject newnumber;
    newnumber.typeFlag = num;
    newnumber.value[0] = operandi;
    EulerStackPush(newnumber);
}

void e logval(int operandi)
{
    EulerDataObject newbool;
    newbool.typeFlag = bool;
    newbool.value[0] = operandi;
    EulerStackPush(newbool);
}

void e omega()
{
    EulerDataObject newundf;
    newundf.typeFlag = undf;
    EulerStackPush(newundf);
}

void e symbol(int operandi)
{
    EulerDataObject newsymb;
    newsymb.typeFlag = symb;
```c
newsymb.value[0] = operand1;
EulerStackPush(newsym);
}

void elabel(int operand1, int operand2)
{
    EulerDataObject newlabl;
    newlabl.typeflag = labl;
    newlabl.value[0] = operand1;
    newlabl.value[1] = operand2;
    EulerStackPush(newlabl);
}

--- new source code file ---

/* ***********************************************************************************/
* PROJECT SAMUEL
* MODULE NAME: euler language system
* FILE NAME: euler.c
* PURPOSE: Implement the main euler function
* ***********************************************************************************/

#define EULERMAIN
#include "euler.h"
#include "ports.h"
#undef EULERMAIN

/* *******************************************************************************/
* define the opcode strings
* *******************************************************************************/
const char 'OPCODE_STRING[] =
{ "anequal", "erem", "ecmpcat", "erquote", "erparen", "eassign", "esemi",
  "elthan", "elethan", "equal", "egthan", "eref", "erbrak",
  "erquote", "eabs", "eadd", "eand", "ebegin", "ecall", "edi", "eelse",
  "eend", "eexp", "eformal", "egoto", "ein", "einteger", "eisb", "eis!",
  "eisii", "eisn", "eisp", "eisr", "eisv", "eisv", "elabel", "elength",
  "elist", "elogical", "elogval", "enmax", "emin", "enmul", "eneg",
  "enew", "enot", "enumber", "enomega", "eor", "eout", "ersai", "esub",
  "esymbol", "etail", "ethen", "evalue" };

/* *******************************************************************************/
* define the parser globals
* *******************************************************************************/
int BlockNumber=0, Offset=0, ProgramIndex=0, MarkValue=0, SymTabIndex=0;
int StempSb, Scale=0, Temp=0, Linenum=1;
EulerInstruction *CompiledProgram;
SymTabEntry *SymTab;
FILE *yyin, *RedirectedOutputFile;

/* *******************************************************************************/
* define the global flags
* *******************************************************************************/
int EulerReportFlag=0, EulerDebugFlag=0, EulerRedirectedOutputFlag=0;
int EulerCompiledProgramFlag=0, EulerCompileOnlyFlag=0;
int EulerSynthesisFlag=0, EulerSimulationFlag = 0, EulerShortFormFlag = 0;

/* *******************************************************************************/
* define the run time system
* variables
* *******************************************************************************/
EulerDataObject *EulerStack;
EulerDataObject *EulerVarHeap;
int EulerFCT=0, EulerMP=0, EulerPC=0, EulerSP=0, EulerVTP=0;
int EulerCompiledProgramLength, EulerStackSize, EulerVarHeapSize;
int *EulerOpcodeCounts;

/* *******************************************************************************/
* the main function
* *******************************************************************************/
```
```c
int main(int argc, char *argv[]) {

    /* ............................
        * local variables for main *
    ............................ *
    
    char *RedirectedOutputFilename;
    FILE *SourceFile;
    int CmdLineArguments, FilenameLength, TableSize = 500;

    /* process the command line */
    ............................ *
    
    CmdLineArguments = argc;
    if ( CmdLineArguments == 1 ) {
        fprintf(stderr, "\n\n\n%s
\r\n", EULER_USAGE_STRING);
        return -1;
    }

    while (--CmdLineArguments > 1) {
        if ( strcmp(argv[CmdLineArguments-1],"-v") == 0 )
            EulerReportFlag = 1;
        else if ( strcmp(argv[CmdLineArguments-1],"-o") == 0 )
            EulerRedirectedOutputFlag = 1;
        else if ( strcmp(argv[CmdLineArguments-1],"-d") == 0 )
            EulerDebugFlag = 1;
        else if ( strcmp(argv[CmdLineArguments-1],"-t") == 0 )
            EulerCompiledProgramOutputFlag = 1;
        else if ( strcmp(argv[CmdLineArguments-1],"-c") == 0 )
            EulerCompileOnlyFlag = 1;
        else if ( strcmp(argv[CmdLineArguments-1],"-b") == 0 )
            TableSize = 50000;
        else if ( strcmp(argv[CmdLineArguments-1],"-synch") == 0 )
            EulerSynthesisFlag = 1;
        else if ( strcmp(argv[CmdLineArguments-1],"-sim") == 0 )
            EulerSimulationFlag = 1;
        else {
            fprintf(stderr, "\n\n\n%s
\r\n", EULER_USAGE_STRING);
            return -1;
        }

        /* redirect system output? */
        ............................ *
        
        FilenameLength = strlen(argv[argc-1]);
        if ( EulerRedirectedOutputFlag ) {
            RedirectedOutputFilename = (char *)malloc(strlen(argv[argc-1]));
            strcpy(RedirectedOutputFilename,argv[argc-1]);
            freopen(RedirectedOutputFilename,"w*");
            if ( (RedirectedOutputFile = fopen(RedirectedOutputFilename,"w")) == NULL) {
                fprintf(stderr, "\n\nThe output file could not be opened: %s\n\n", argv[argc-1]);
                return -1;
            }
        } else RedirectedOutputFile = stdout;

        /* allocate the tables */
        ............................ *
        
        if ( (SymTab = calloc(TableSize,sizeof(SymTabEntry))) == 0 )
```
if (CompiledProgram = calloc(TableSize, sizeof(EulerInstruction))) == 0) {
    fprintf(stderr, "ERROR: Euler could not allocate a compiled program table.\n\n") ;
    return -1;
}
EulerOpcodeCounts = (int *)calloc(TOTAL_OPCODES.sizeof(int));

/* open the source and parse */
Tools::Euler::Euler::parse();
EulerStackInit(TableSize);
EulerVarHeapInit(TableSize); EulerCompiledProgramLength = ProgramIndex + 1;

/* run the compiled program */
Cycle();
109

```c
/* ****************************
 * exit safely
 *****************************/

return(0);
}

void ShowEulerBanner(const char *infilename)
{
    int index, length;
    length = strlen(EULER_VERSION_STRING);
    fprintf(RedirectedOutputFile, "%\n");
    fprintf(RedirectedOutputFile, "****\n");
    fprintf(RedirectedOutputFile, "%\n");
    fprintf(RedirectedOutputFile, "****\n");
    fprintf(RedirectedOutputFile, "%\n");
    fprintf(RedirectedOutputFile, "****\n");
    if (EulerCompileOnlyFlag)
        fprintf(RedirectedOutputFile, "nCompiling: "filename, infilename);
    else fprintf(RedirectedOutputFile, "nRunning: "filename, infilename);
}

int WriteSynthesisOutputFile(const char *infilename)
{
    FILE *SynthesisFile, *VerilogFile, *TemplateFile;
    FILE *ObjectCodeFile;
    char *SynthesisFilename, *VerilogFilename, *TemplateFilename, temp;
    char *ObjectCodeFilename;
    int index = 0, totalopcodes = 0, totalcalls = 0, current = 0;
    SynthesisFilename = (char *)malloc(strlen(infilename)-5);
    strcpy(SynthesisFilename, infilename);
    strcat(SynthesisFilename, "syn");
    if ((SynthesisFile = fopen(SynthesisFilename, "w")) == NULL)
    {
        free(SynthesisFilename);
        return -1;
    }
    fprintf(SynthesisFile,"EULER Synthesis Information\n");
    fprintf(SynthesisFile,"\n");
    while ( index < TOTAL_OPCODES )
    {
        if (EulerOpcodeCounts[index] != 0)
        {
            fprintf(SynthesisFile, "%d %10s %5d\n", index, OPCODE_STRINGS[index], EulerOpcodeCounts[index]);
            totalcalls = totalcalls + EulerOpcodeCounts[index];
            totalopcodes = totalopcodes + 1;
            /* I want to know how many total opcodes, and how many times */
            /* those opcodes are called to complete the program execution*/
        }
        index = index + 1;
    }
    fprintf(SynthesisFile, "Total Opcodes Required: %3d\n", totalopcodes);
    fprintf(SynthesisFile, "Total Opcode Calls: %3d\n", totalcalls);
    fclose(SynthesisFile);
    free(SynthesisFilename);
    if (EulerSimulationFlag)
    {
        ObjectCodeFilename = (char *)malloc(strlen(infilename) + 5);
        strcpy(ObjectCodeFilename, infilename);
        strcat(ObjectCodeFilename, "rom");
        if ((ObjectCodeFile = fopen(ObjectCodeFilename, "w")) == NULL)
        {
```
free(ObjectCodeFilename);
            return -1;
        }

        fprintf(ObjectCodeFile,"module irom(abus, dbus): \n\n");
        fprintf(ObjectCodeFile," input [7:0] abus: \n");
        fprintf(ObjectCodeFile," output [39:0] dbus: \n\n");
        fprintf(ObjectCodeFile," reg [39:0] values[0:255]: \n\n");
        fprintf(ObjectCodeFile," assign dbus = values[abus]; \n\n");
        fprintf(ObjectCodeFile," initial\n")
        fprintf(ObjectCodeFile," begin : compiledProgram\n");

        while(--current <= ProgramIndex)
            fprintf(ObjectCodeFile," \n\n" values[\d] = 40'h02x_\%04x_\%04x; \n\n" current,
            CompiledProgram[current].opcode,
            CompiledProgram[current].operand1,
            CompiledProgram[current].operand2);

        fprintf(ObjectCodeFile," \n\n" values[\d] = 40'hFF_0000_0000; \n\n"
            fprintf(ObjectCodeFile," end\n");
            fprintf(ObjectCodeFile," endmodule \n");
            fclose(ObjectCodeFile);
        }

        VerilogFilename = (char *)malloc(strlen(infilename)-5) ;
        strcpy(VerilogFilename, infilename);
        strcat(VerilogFilename,".v");
        if ((VerilogFile = fopen(VerilogFilename,"w")) == NULL) {
            free(VerilogFilename);
            return -1;
        }

        fprintf(VerilogFile,"/* \n\n%s: created by SAMUEL\n\n*/ \n\n", VerilogFilename);
        index = 0;

        TemplateFilename = malloc(50); /* a char buffer of fifty */
        strcpy(TemplateFilename,LIBRARY_PATH);
        strcat(TemplateFilename,"/*");
        strcat(TemplateFilename,"defines.v");
        if ((TemplateFile = fopen(TemplateFilename,"r")) == NULL) {
            free(TemplateFilename);
            return -1;
        }

        while (fscanf(TemplateFile, "%c", &temp) != EOF)
            fprintf(VerilogFile,"%c", temp);
        fprintf(VerilogFile,"\n\n");
        fclose(TemplateFile);

        strcpy(TemplateFilename, LIBRARY_PATH);
        strcat(TemplateFilename, "/\n\n");
        strcat(TemplateFilename,"mux41.v");
        if ((TemplateFile = fopen(TemplateFilename,"r")) == NULL) {
            free(TemplateFilename);
            return -1;
        }

        while (fscanf(TemplateFile, "%c", &temp) != EOF)
            fprintf(VerilogFile,"%c", temp);
        fprintf(VerilogFile,"\n\n");
        fclose(TemplateFile);

        strcpy(TemplateFilename, LIBRARY_PATH);
        strcat(TemplateFilename, "/\n\n");
        strcat(TemplateFilename,"mux41.v");
if ((TemplateFile = fopen(TemplateFilename, "r")) == NULL)
    {
        free(TemplateFilename);
        return -1;
    }

while (fscanf(TemplateFile, "%c", &temp)! = EOF)
    fprintf(VerilogFile, "%c", temp);

fclose(TemplateFile);

if ((TemplateFile = fopen(TemplateFilename, "r")) == NULL)
    {
        free(TemplateFilename);
        return -1;
    }

while (fscanf(TemplateFile, "%c", &temp)! = EOF)
    fprintf(VerilogFile, "%c", temp);

fclose(TemplateFile);

if (EulerSimulationFlag)
    {
        strcpy(TemplateFilename, LIBRARY_PATH);
        strcat(TemplateFilename, "/");
        strcat(TemplateFilename, "mem.v");
        if ((TemplateFile = fopen(TemplateFilename, "r")) == NULL)
            {
                free(TemplateFilename);
                return -1;
            }
        while (fscanf(TemplateFile, "%c", &temp)! = EOF)
            fprintf(VerilogFile, "%c", temp);
        fclose(TemplateFile);
        fprintf(VerilogFile, "\n\n");
    }

while (index < TOTAL_OPCODES)
    {
        if (EulerOpcodeCounts[index] != 0)
            {
                strcpy(TemplateFilename, LIBRARY_PATH);
                strcat(TemplateFilename, "/");
                strcat(TemplateFilename, OPCODE_STRINGS[index]);
                strcat(TemplateFilename, ".v");
                if ((TemplateFile = fopen(TemplateFilename, "r")) == NULL)
                    {
                        free(TemplateFilename);
                        return -1;
                    }
                while (fscanf(TemplateFile, "%c", &temp)! = EOF)
                    fprintf(VerilogFile, "%c", temp);
                fclose(TemplateFile);
                index = index + 1;
            }
        fprintf(VerilogFile, "\module machine(clk, error, import, output, reset);\n");
        fprintf(VerilogFile, "input clk;\n");
```c
112

```
free(TemplateFilename);
return 0;
}

int WriteCompiledProgramOutputFile(const char *infilename)
{
    FILE *CompiledProgramFile;
    char *CompiledProgramFilename;
    int current = 0;
    CompiledProgramFilename = (char *)malloc(strlen(infilename)-5);  
    strcpy(CompiledProgramFilename, infilename);
    strcat(CompiledProgramFilename, ".cmp");
    if ((CompiledProgramFile = fopen(CompiledProgramFilename, "w")) == NULL)
    {
        free(CompiledProgramFilename);
        return -1;
    }

    fprintf(CompiledProgramFile, "%10s %10s %10s %10s\n",
            "SourceLine", "Opcode", "Operand1", "Operand2");

    while (current <= ProgramIndex)
    {
        fprintf(CompiledProgramFile, "%10d %10s %10d %10d\n",
                CompiledProgram[current].sourceLine, 
                OPCODE_STRINGS[ CompiledProgram[current].opcode ],
                CompiledProgram[current].operand1, 
                CompiledProgram[current].operand2);
    }

    fclose(CompiledProgramFile);
    free(CompiledProgramFilename);
    return 0;
}

--- new source code file ---

#include "euler.h"

void EulerVarHeapInit(int insize)
{
    EulerVarHeap = (EulerDataObject*)
        calloc(insize, sizeof(EulerDataObject));
    EulerVarHeapSize = insize;
}

void EulerVarHeapPush(EulerDataObject inobject)
{
    if ( EulerVTP < (EulerVarHeapSize - 1) )
        EulerVarHeap[--EulerVTP] = inobject;
    else
    {
        FatalError("\nEuler Variable Table Overflow. Cannot Continue.\n");
    }
}

void EulerVarHeapSet(EulerDataObject inobject, int index)
{
    if ( index < (EulerVarHeapSize - 1) )
    {
        EulerVarHeap[index] = inobject;
    }
}
else
{
    FatalError("\nEuler Variable Table Set: Bad Index. Cannot Continue.");
}
}

EulerDataObject EulerVarHeapItem(int index)
{
    if ( (index <= EulerVTP) && (index > 0) )
        return EulerVarHeap[index];
    else
    {
        FatalError("Euler Variable Table: Bad Index. Cannot Continue.");
    }
}

int EulerVarHeapGrabOneSpace()
{
    if ( EulerVTP < (EulerVarHeapSize - 1) )
        return ++EulerVTP;
    else
        FatalError("No More Variable Memory Space.");
}

int EulerVarHeapGrabManySpaces(int insize)
{
    int outvalue;
    /* can I have insize locations? -1 is for the -- */
    if ( EulerVTP < (EulerVarHeapSize - insize ) )
    {
        outvalue = EulerVTP - 1;
        EulerVTP = EulerVTP - insize;
        return outvalue;
    }
    else
        FatalError("No More Variable Memory Space.");
}

void EulerVarHeapPrint()
{
    int loop;
    fprintf(RedirectedOutputFile, "\nThe Variable Table Contains:
");
    fprintf(RedirectedOutputFile, "---------------------------\n");
    for (loop = 1; loop <= EulerVTP; loop++)
    {
        fprintf(RedirectedOutputFile, "%d: , loop);
        EulerDataObjectPrint(EulerVarHeap[loop]);
    }
}

int FindVariableLocation(int inmarkptr, int invarnumber)
{
    if ( (invarnumber <= EulerStack[inmarkptr].value[4]) )
        return EulerStack[inmarkptr].value[3] - (invarnumber - 1);
    else
        FatalError("FVL: Unknown Variable. Check variable usage.");
}

--- new source code file ---

/ * ************************************************************************** *
 * PROJECT SAMUEL
 *
 * MODULE NAME: euler language system
 * FILE NAME: evars.c
 * PURPOSE: implement variable manipulation functions
 * ************************************************************************** */
#include "euler.h"

void enew()
{
    int temp;
temp = EulerVarHeapGrabOneSpace();
EulerStack[EulerMP].value[1]--;
if (EulerStack[EulerMP].value[3] == 0)
  EulerStack[EulerMP].value[3] = temp;
}

void eref(int operand1, int operand2)
{
  EulerDataObject newref;
  newref.typeflag = vref;
  /* get location of this mark on EulerStack */
  newref.value[0] = FindMark(operand2);
  /* get location of this variable on EulerVarHeap */
  newref.value[1] = FindVariableLocation(newref.value[0], operand1);
  /* push reference */
  EulerStackPush(newref);
}

EulerDataObject ederef(EulerDataObject inref)
{
  return EulerVarHeap[ inref.value[1] ];
}

void eformal()
{
  if (--EulerFCT > EulerStack[EulerMP].value[4])
    enew();
}

void ecout()
{
  printf(RedirectedOutputFile,"OUTPUT: ");
  EulerDataObjectPrint(EulerStack[EulerSP]);
}

void ein()
{
  EulerDataObject inputresult;
  int innumber;
  printf(RedirectedOutputFile,"INPUT VALUE: ");
  scanf("%d", &innumber);
  inputresult.typeflag = numb;
  inputresult.value[0] = innumber;
  EulerStackPush(inputresult);
}

void eassign()
{
  EulerDataObject rvalue, lvalue;
  if (isr(EulerSP-1))
  {
    if (issi(EulerSP))
      rvalue = ecopylist(EulerStackPop());
    else
      rvalue = EulerStackPop();
    lvalue = EulerStackPop();
    EulerVarHeap[ lvalue.value[1] ] = rvalue;
    EulerStackPush(rvalue);
  }
  else
    {RuntimeError(“Euler Assignment Error: check := statements.”);
  }
}

--- new source code file ---
/
************************************
PROJECT SAMUEL

MODULE NAME: euler language system
FILE NAME: parser.c
PURPOSE: implement support functions for the parser

#include "euler.h"

void NewSymTabEntry(int in_index, char *in_value, int in_block,
                     int in_offset, char* in_type)
{
    static int total_symbols = 0;
    SymTab[in_index].value = strdup(in_value);
    SymTab[in_index].block_number = in_block;
    SymTab[in_index].offset = in_offset;
    SymTab[in_index].type = strdup(in_type);
    total_symbols++;
}

void ChangeSymTabOffset(int in_index, int in_offset)
{
    SymTab[in_index].offset = in_offset;
}

void ChangeSymTabType(int in_index, char *in_type)
{
    SymTab[in_index].type = strdup(in_type);
}

void NewInstruction(int in_index, EulerOpcodeType in_opcode,
                     int in_operand1, int in_operand2)
{
    static int total_program_lines = 0;
    CompiledProgram[in_index].opcode = in_opcode;
    CompiledProgram[in_index].operand1 = in_operand1;
    CompiledProgram[in_index].operand2 = in_operand2;
    CompiledProgram[in_index].source_line = Linenum;
    total_program_lines++;
    EulerOpcodeCounts[in_opcode]++;
}

void ChangeInstructionOperand1(int in_index, int in_op1)
{
    CompiledProgram[in_index].operand1 = in_op1;
}

void var_error(void)
{
    fprintf(stderr, "Error occurred in Rule 4: t<i
");
    fprintf(stderr, "Cannot recover.\n");
    exit(1);
}

void lab_error(void)
{
    fprintf(stderr, "Error occurred in Rule 113: label error\n");
    fprintf(stderr, "Cannot recover.\n");
}

int yyerror(char *s)
{
    fprintf(stderr, "Line:%10d
"), Linenum;
    fprintf(stderr, "%s\n", s);
    return 0;
}

int yywrap(void)
{
    fclose(yyin);
    return 1;
APPENDIX B. VERILOG OPCODES

This appendix does not contain Verilog source code for all 57 Euler opcodes. Rather, a representative sample is provided for reference.

/* typeflags */
#define mark 8'b00000000
#define bool 8'b00000001
#define labl 8'b00000010
#define list 8'b00000011
#define undf 8'b00000100
#define proc 8'b00000110
#define vref 8'b00000111

/* data info */
#define DATA 31:0
#define TYPE 47:40

/* stack info */
#define STACKMAX 255
#define SPBITS 8

/* fields for readability */
#define BLOCK 47:40
#define DYNAMIC 39:32
#define STATIC 31:24
#define VAREND 23:16
#define RETURN 15:8
#define MP 7:0
#define UNDF 39:0
#define NUMBER 15:0
#define NUMBERLSB 7:0
#define LABELOPl 31:16
#define LABELOP2 15:0
#define LABELUNUSED 39:32

/* control bus info */
#define CBITS 17

/* instruction info */
#define OPCODEBITS 8
#define OPCODE 39:32
#define OPERAND1 31:16
#define OPERAND2 15:0

/* opcode encodings on ibus */
define enequal 8'h00
#define erem 8'h01
#define econcat 8'h02
#define erquote 8'h03
#define erparen 8'h04
#define eassign 8'h05
#define esemi 8'h06
#define elthan 8'h07
#define elethan 8'h08
#define eequal 8'h09
#define egthan 8'h0A
#define egethan 8'h0B
#define eref 8'h0C
#define erbrak 8'h0D
#define elquote 8'h0E
#define eabs 8'h0F
#define eadd 8'h10
#define eand 8'h11
#define ebegin 8'h12
#define ecall 8'h13
#define ediv 8'h14
#define eelse 8'h15
#define eend 8'h16
#define eexp 8'h17
#define eformal 8'h18
module eadd(clk, opcode, reset, cbus, dbus);
  input clk;
  input [7:0] opcode;
  input reset;
  output [16:0] cbus;
  inout [47:0] dbus;

  reg [2:0] statebits;
  reg [16:0] nanoword;
  reg [47:0] number1;
  reg [47:0] number2;

  assign dbus['TYPEFLAG] = (statebits == 5)?8'b0:8'bz;
  assign dbus[35:16] = (statebits == 5)?2'b0:24'b0;
  assign dbus['NUMBER] = (statebits == 5)?number1['NUMBER]+number2['NUMBER]:16'b0;
  assign cbus = (statebits == 0)?17'b0:nanoword;

always @(negedge clk)
begin : cempLacch
  case (statebits)
  ...
always @(posedge clk)
begin : stateTransition
  case (statebits)
  0: if (reset == 0) statebits = 0;
     else if (opcode == 'eadd) statebits = 1;
     else statebits = 0;
  1: if (reset == 0) statebits = 0;
     else statebits = 2;
  2: if (reset == 0) statebits = 0;
     else statebits = 3;
  3: if (reset == 0) statebits = 0;
     else statebits = 4;
  4: if (reset == 0) statebits = 0;
     else if (number1['TYPEFLAG'] == numb && number2['TYPEFLAG'] == num)
     statebits = 5;
     else statebits = 7;
  5: if (reset == 0) statebits = 0;
     else statebits = 6;
  6: if (reset == 0) statebits = 0;
     else statebits = 7;
  7: if (reset == 0) statebits = 0;
     else statebits = 7;
  default: statebits = 0;
  endcase
end
always @(statebits)
begin : combLogic
  case (statebits)
  0: nanoword = 17'h1FFF;
  1: nanoword = 17'h1FFDF;
  2: nanoword = 17'h1FCFF;
  3: nanoword = 17'h1FFDF;
  4: nanoword = 17'h1FEFF;
  5: nanoword = 17'h1FFCF;
  6: nanoword = 17'h1FFFF;
  7: nanoword = 17'h0FFFF;
  default: nanoword = 17'h1FFFF;
  endcase
end
//always @(statebits == 7) $display("n1=%h,n2=%h", number1,number2);
endmodule

--- new module ---

module eassign(clk,opcode,markbus,reset,cbus,dbus);
input clk;
input [7:0] opcode;
input [47:0] markbus;
input reset;
output [16:0] cbus;
inout [47:0] dbus;
reg [3:0] statebits;
reg [16:0] nanoword;
reg [47:0] lvalue;
reg [47:0] rvalue;
reg [47:0] temp;

assign dbus = (statebits == 5)?(40'b0,lvalue{7:0}) : 
             (statebits == 7 || statebits == 10)? rvalue :
             (statebits == 8)? temp : 48'bz;
assign cbus = (statebits == 0)?17'bz:nanoword;
always @(negedge clk)
begin : tempLatch
  case (statebits)
  1: rvalue = dbus;
  endcase
endmodule
always @(posedge elk) begin : stateTransistion
    case (statebits)
    begin
        0: if (reset == 0) statebits = 0;
           else if (opcode == "eassign") statebits = 1;
           else statebits = 0;
        1: if (reset == 0) statebits = 0;
           else statebits = 2;
        2: if (reset == 0) statebits = 0;
           else statebits = 3;
        3: if (reset == 0) statebits = 0;
           else statebits = 4;
        4: if (reset == 0) statebits = 0;
           else if (lvalue[TYPEFLAG]!="vref) statebits = 12;
           else statebits = 5;
        5: if (reset == 0) statebits = 0;
           else statebits = 6;
        6: if (reset == 0) statebits = 0;
           else statebits = 7;
        7: if (reset == 0) statebits = 0;
           else statebits = 8;
        8: if (reset == 0) statebits = 0;
           else statebits = 9;
        9: if (reset == 0) statebits = 0;
           else statebits = 10;
        10: if (reset == 0) statebits = 0;
           else statebits = 11;
        11: if (reset == 0) statebits = 0;
           else statebits = 0;
        12: if (reset == 0) statebits = 0;
           else statebits = 12;
           default: statebits = 0;
    endcase
end
always @(statebits) begin : combLogic
    case (statebits)
    begin
        0: nanoword = 17'h1FFFF;
        1: nanoword = 17'h1FFDF;
        2: nanoword = 17'h1FCFF;
        3: nanoword = 17'h1FFDF;
        4: nanoword = 17'h1FFFF;
        5: nanoword = 17'h1FFFF;
        6: nanoword = 17'h1FFFF;
        7: nanoword = 17'h1FFFF;
        8: nanoword = 17'h1FFFF;
        9: nanoword = 17'h1FFFF;
        10: nanoword = 17'h1FFFF;
        11: nanoword = 17'h1FFFF;
        12: nanoword = 17'h0FFFF;
           default: nanoword = 17'h1FFFF;
    endcase
end
endmodule
--- new module ---
module ebegin(clk,markbus,opcode,reset,spbus,cbus,dbus);
    input clk;
    input [47:0] markbus;
    input [7:0] opcode;
    input reset;
    input [7:0] spbus;
    output [16:0] cbus;
    output [47:0] dbus;
    reg [2:0] statebits;
    reg [16:0] nanoword;
always @(posedge clk)
begin : stacTransition
  case (statebits)
    0: if (reset == 0) statebits = 0;
      else if (ibus[‘OPCODE] == ‘else) statebits = 1;
      else statebits = 0;
    1: if (reset == 0) statebits = 0;
      else statebits = 0;
    2: if (reset == 0) statebits = 0;
      else statebits = 3;
    3: if (reset == 0) statebits = 0;
      else statebits = 4;
    4: if (reset == 0) statebits = 0;
      else statebits = 5;
    5: if (reset == 0) statebits = 0;
      else statebits = 0;
    default: statebits = 0;
  endcase
end
always @(statebits)
begin : combLogic
  case (statebits)
    0: nanoword = 17’h1FFFF;
    1: nanoword = 17’h1FDFF;
    2: nanoword = 17’h1FEFF;
    3: nanoword = 17’h1FFCF;
    4: nanoword = 17’h1FFFF;
    5: nanoword = 17’h1FFFF;
    default: nanoword = 17’h1FFFF;
  endcase
end
endmodule

--- new module ---

module else(clk, ibus, reset, cbus, dbus);
  input clk;
  input [39:0] ibus;
  input reset;
  output [16:0] cbus;
  output [47:0] dbus;
  reg [1:0] statebits;
  reg [15:0] nanoword;

  assign dbus = (statebits == 1)?[32:0, ibus[‘OPERANDO]':48:’bz;
  assign cbus = (statebits == 0)?17’bz:nanoword;

always @(posedge clk)
begin : stateTransition
  case (statebits)
    0: if (reset == 0) statebits = 0;
      else if (ibus[‘OPCODE] == ‘else) statebits = 1;
      else statebits = 0;
    1: if (reset == 0) statebits = 0;
      else statebits = 0;
    2: if (reset == 0) statebits = 0;
      else statebits = 0;
    3: if (reset == 0) statebits = 0;
      else statebits = 0;
    4: if (reset == 0) statebits = 0;
      else statebits = 0;
    default: statebits = 0;
  endcase
end
always @(statebits)
begin : combLogic
  case (statebits)
    0: nanoword = 17’h1FFFF;
    1: nanoword = 17’h1F3FF;
    default: nanoword = 17’h1FFFF;
  endcase
module eend(clk,opcode,reset,cbus,dbus);
  input clk;
  input [7:0] opcode;
  input reset;
  output [16:0] cbus;
  inout [47:0] dbus;

  reg [3:0] statebits;
  reg [16:0] nanoword;
  reg [47:0] temp;

  assign dbus = (statebits == 4) || (statebits == 7) ? temp : 48'bz;
  assign cbus = (statebits == 0)?17'b0:nanoword;

  always @(negedge clk)
  begin : tempLatch
    case (statebits)
      1: temp = dbus;
      6: temp = dbus;
      default: temp = temp;
    endcase
  end

  always @(posedge clk)
  begin : stateTransition
    case (statebits)
      0: if (reset == 0) statebits = 0;
         else if (opcode == 'eend) statebits = 1;
      1: if (reset == 0) statebits = 0;
         else statebits = 2;
      2: if (reset == 0) statebits = 0;
         else statebits = 3;
      3: if (reset == 0) statebits = 0;
         else statebits = 4;
      4: if (reset == 0) statebits = 0;
         else statebits = 5;
      5: if (reset == 0) statebits = 0;
         else statebits = 6;
      6: if (reset == 0) statebits = 0;
         else statebits = 7;
      7: if (reset == 0) statebits = 0;
         else statebits = 8;
      8: if (reset == 0) statebits = 0;
         else statebits = 0;
      default: statebits = 0;
    endcase
  end

  always @(statebits)
  begin : combLogic
    case (statebits)
      0: nanoword = 17'hFFFF;
      1: nanoword = 17'hFDFF;
      2: nanoword = 17'hFFEF;
      3: nanoword = 17'hFFBF;
      4: nanoword = 17'hFFCF;
      5: nanoword = 17'hFFBB;
      6: nanoword = 17'hFFF9;
      7: nanoword = 17'hFFFB;
      8: nanoword = 17'hFFFF;
      default: nanoword = 17'hFFFF;
    endcase
  end
endmodule

--- new module ---

module egethan(clk,opcode,reset,cbus,dbus);
  input clk;
input [7:0] opcode;
input reset;
output [16:0] cbus;
inout [47:0] dbus;

reg [2:0] statebits;
reg [16:0] nanoword;
reg [47:0] number1;
reg [47:0] number2;

assign dbus['TYPEFLAG1 = (statebits == 5)?'bool:3'bz;
assign dbus[39:16] = (statebits == 5)?24'b0:24'b1;
assign dbus[‘NUMBER] = (statebits == 5)?number1[‘NUMBER]:16’bz;
assign cbus = (statebits == 0)?17’bz:nanoword;

always @(negedge clk)
begin : tempLatch
  case (statebits)
    1: number2 = dbus;
    3: number1 = dbus;
    default: begin number1 = number1; number2 = number2; end
  endcase
end

always @(posedge clk)
begin : stateTransition
  case (statebits)
    0: if (reset == 0) statebits = 0;
      else if (opcode == 1egethan) statebits = 1;
      else statebits = 0;
    1: if (reset == 0) statebits = 0;
      else statebits = 1;
    2: if (reset == 0) statebits = 0;
      else statebits = 2;
    3: if (reset == 0) statebits = 0;
      else statebits = 3;
    4: if (reset == 0) statebits = 0;
      else if (number1[‘TYPEFLAG] == numb && number2[‘TYPEFLAG] == numb)
          statebits = 5;
      else statebits = 4;
    5: if (reset == 0) statebits = 0;
      else statebits = 5;
    6: if (reset == 0) statebits = 0;
      else statebits = 6;
    7: if (reset == 0) statebits = 0;
      else statebits = 7;
    default: statebits = 0;
  endcase
end

always @(statebits)
begin : combLogic
  case (statebits)
    0: nanoword = 17’hlFFFF;
    1: nanoword = 17’hlFFDF;
    2: nanoword = 17’hlFCFF;
    3: nanoword = 17’hlFFDF;
    4: nanoword = 17’hlFFEF;
    5: nanoword = 17’hlFFCF;
    6: nanoword = 17’h1FFFF;
    7: nanoword = 17’h0FFFF;
    default: nanoword = 17’h1FFFF;
  endcase
end
endmodule

--- new module ---

module ehalt(clk,opcode,reset,cbus);
input clk;
input [7:0] opcode;
input reset;
output [16:0] cbus;

reg statebits;
reg [16:0] nanoword;
assign cbus = (statebits == 0)?17’bz:nanoword;

always @(posedge clk)
begin : stateTransition
    case (statebits)
        0: if (reset == 0) statebits = 0;
           else if (opcode == 'ehalc) statebits = 1;
           else statebits = 0;
        1: statebits = 1;
           default: statebits = 0;
    endcase
end

always @(statebits)
begin : combLogic
    case (statebits)
        0: nanoword = 17’hFFFF;
        1: nanoword = 17’hFFFF;
           default: nanoword = 17’hFFFF;
    endcase
end
endmodule

--- new module ---

module ein(clk, opcode, reset, cbus);
    inpuc elk;
    inpuc [7:0] opcode;
    input reset;
    output [16:0] cbus;
    reg [2:0] scatebics;
    reg [16:0] nanoword;

    assign ebus = (scatebics == 0)?17’bz:nanoword;

    always @(posedge clk)
    begin : stateTransiscion
        case (statebits)
            0: if (reset == 0) statebits = 0;
               else if (opcode == 'ein) statebits = 1;
               else statebits = 0;
            1: if (reset == 0) statebits = 0;
               else statebits = 2;
            2: if (reset == 0) statebits = 0;
               else statebits = 3;
            3: if (reset == 0) statebits = 0;
               else statebits = 4;
            4: if (reset == 0) statebits = 0;
               else statebits = 0;
               default: statebits = 0;
        endcase
    end

    always @(statebits)
    begin : combLogic
        case (statebits)
            0: nanoword = 17’hFFFF;
            1: nanoword = 17’hFFFF;
            2: nanoword = 17’hFFFF;
            3: nanoword = 17’hFFFF;
            4: nanoword = 17’hFFFF;
               default: nanoword = 17’hFFFF;
        endcase
    end
endmodule

--- new module ---

module elabel(clk, ibus, reset, cbus, dbus);
    inpuc elk;
    inpuc [39:0] ibus;
    input reset;
    output [16:0] cbus;
    output [47:0] dbus;
reg [2:0] statebits;
reg [16:0] nanoword;

assign dbus["TYPEFLAG"] = (statebits == 1)?1'b1:8'b0;
assign dbus["LABELUNUSED"] = (statebits == 3)?8'b0:8'b0;
assign dbus["LABELOP1"] = (statebits == 3)?ibus["OPERAND1":16]':8'b0;
assign dbus["LABELOP2"] = (statebits == 3)?ibus["OPERAND2":16]':8'b0;
assign cbus = (statebits == 0)?17'b0:nanoword;

always @(posedge clk)
begin : stateTransition
  case (statebits)
    0: if (reset == 0) statebits = 0;
      else if (ibus["OPCODE"] == 'elabo) statebits = 1;
      else statebits = 0;
    1: if (reset == 0) statebits = 0;
      else statebits = 2;
    2: if (reset == 0) statebits = 0;
      else statebits = 3;
    3: if (reset == 0) statebits = 0;
      else statebits = 4;
    4: if (reset == 0) statebits = 0;
      else statebits = 0;
      default: statebits = 0;
  endcase
end

always @(posedge clk)
begin : combLogic
  case (statebits)
    0: nanoword = 17'h1FFFF;
    1: nanoword = 17'h1FPFF;
    2: nanoword = 17'h1FPEF;
    3: nanoword = 17'h1FPEF;
    4: nanoword = 17'h17FFF;
    default: nanoword = 17'h1FFFF;
  endcase
end

endmodule

--- new module ---

module elquoce(clk, iabus, ibus, markbus, reset, cbus, dbus);
  input clk;
  input [7:0] iabus;
  input [39:0] ibus;
  input [47:0] markbus;
  input reset;
  output [16:0] cbus;
  output [47:0] dbus;

  reg [2:0] statebits;
  reg [16:0] nanoword;

  assign dbus = (statebits == 3)?"proc.markbus["BLOCK]+1.markbus["MP]","iabus,8'b0,8'b0":
    (statebits == 4)?"40'b0.ibus["OPERAND1"]":48'b0;
  assign cbus = (statebits == 0)?17'b0:nanoword;

  always @(posedge clk)
  begin : stateTransition
    case (statebits)
      0: if (reset == 0) statebits = 0;
        else if (opcode == 'elquoce) statebits = 1;
        else statebits = 0;
      1: if (reset == 0) statebits = 0;
        else statebits = 2;
      2: if (reset == 0) statebits = 0;
        else statebits = 3;
      3: if (reset == 0) statebits = 0;
        else statebits = 4;
      4: if (reset == 0) statebits = 0;
        else statebits = 0;
        default: statebits = 0;
    endcase
  end
always @(statebits)
begin : combLogic
    case (statebits)
        0: nanoword = 17'h1FFFF;
        1: nanoword = 17'h1FFFD;
        2: nanoword = 17'h1FFFD;
        3: nanoword = 17'h1FFFD;
        default: nanoword = 17'h1FFFF;
    endcase
end
endmodule

--- new module ---

module emin(clk, opcode, reset, cbus, dbus);
    input clk;
    input [7:0] opcode;
    input reset;
    output [16:0] cbus;
    inout [47:0] dbus;
    reg [2:0] statebits;
    reg [16:0] nanoword;
    reg [47:0] number1;
    reg [47:0] number2;
    assign dbus = (statebits == 5) ?
        (number1['NUMBER] <= number2['NUMBER] ? number1 : number2) :
        48'bz;
    assign cbus = (statebits == 0)?17'bz:nanoword;
always @(negedge clk)
begin : tempLatch
    case (statebits)
        1: number2 = dbus;
        3: number1 = dbus;
        default: begin number1 = number1; number2 = number2; end
    endcase
end
always @(posedge clk)
begin : stateTransition
    case (statebits)
        0: if (reset == 0) statebits = 0;
           else if (opcode == 'emin) statebits = 1;
           else statebits = 0;
        1: if (reset == 0) statebits = 0;
           else statebits = 2;
        2: if (reset == 0) statebits = 0;
           else statebits = 3;
        3: if (reset == 0) statebits = 0;
           else statebits = 4;
        4: if (reset == 0) statebits = 0;
           else if (number1['TYPEFLAG] == numb && number2['TYPEFLAG] == numb)
                statebits = 5;
           else statebits = 7;
        5: if (reset == 0) statebits = 0;
           else statebits = 6;
        6: if (reset == 0) statebits = 0;
           else statebits = 0;
        7: if (reset == 0) statebits = 0;
           else statebits = 7;
        default: statebits = 0;
    endcase
end
always @(statebits)
begin : combLogic
    case (statebits)
        0: nanoword = 17'h1FFFF;
        1: nanoword = 17'h1FFFD;
        2: nanoword = 17'h1FFFD;
        3: nanoword = 17'h1FFFD;
```verilog
4: nanoword = 17'hFFFF;
5: nanoword = 17'hFFFC;
6: nanoword = 17'hFFFF;
7: nanoword = 17'h0FFF;
default: nanoword = 17'hFFFF;
endcase
end
endmodule

--- new module ---

module enew(clk, markbus, opcode, reset, cbus, dbus);
input clk;
input [47:0] markbus;
input [7:0] opcode;
input reset;
output [16:0] cbus;
output [47:0] dbus;
reg [2:0] statebits;
reg [47:0] temp;
reg [16:0] nanoword;
assign dbus = (statebits == 3)?"undf.40'b0":
               (statebits == 4 || statebits == 5)?temp : 48'bz;
assign cbus = (statebits == 0)?17'bz:nanoword;

always @(negedge clk)
begin : latchTemp
  case (statebits)
    1: temp = markbus;
    2: begin
      temp['BLOCK] = temp['BLOCK];
      temp['DYNAMIC] = temp['DYNAMIC];
      temp['STATIC] = temp['STATIC];
      temp['VAREND] = temp['VAREND] - 1;
      temp['RETURN] = temp['RETURN];
      temp['MP] = temp['MP];
    end
    default: temp = temp;
  endcase
end

always @(posedge clk)
begin : stateTransition
  case (statebits)
    0: if (reset == 0) statebits = 0;
       else if (opcode == 'enew) statebits = 1;
       else statebits = 0;
    1: if (reset == 0) statebits = 0;
       else statebits = 2;
    2: if (reset == 0) statebits = 0;
       else statebits = 3;
    3: if (reset == 0) statebits = 0;
       else statebits = 4;
    4: if (reset == 0) statebits = 0;
       else statebits = 5;
    5: if (reset == 0) statebits = 0;
       else statebits = 6;
    6: if (reset == 0) statebits = 0;
       else statebits = 7;
    7: if (reset == 0) statebits = 0;
       else statebits = 0;
    default: statebits = 0;
  endcase
end

always @(statebits)
begin : combLogic
  case (statebits)
    0: nanoword = 17'hFFFF;
    1: nanoword = 17'hFFFD;
    2: nanoword = 17'hFFFE;
    3: nanoword = 17'hFFFF;
  endcase
end
```
module eref (elk, ibus, markbus, reset, cbus, dbus);
    input elk;
    input [39:0] ibus;
    input [47:0] markbus;
    input reset;
    output [15:0] cbus;
    inout [47:0] dbus;
    reg [3:0] statebits;
    reg [16:0] nanoword;
    reg [47:0] temp1;
    reg [47:0] temp2;
    assign dbus['TYPEFLAG] = (statebits == 6)?temp2['TYPEFLAG] :
        (statebits == 7)?temp1['TYPEFLAG] :
        (statebits == 10)?'vref : 8'bz;
    assign dbus[39:8] = (statebits == 6)?temp2[39:8] :
        (statebits == 7)?temp1[39:8] :
        (statebits == 10)?32'b0 : 32'bz;
    assign dbus[7:0] = (statebits == 6)?temp2[7:0] :
        (statebits == 7)?temp1[7:0] :
        (statebits == 10)?ibus['OPERAND1] : 8'bz;
    assign cbus = (statebits == 0)?17'bz:nanoword;
always @(negedge clk)
    begin : tempLatch
        case (statebits)
            1: begin temp1 = markbus; temp2 = markbus; end
            5: temp2 = dbus;
            default: begin temp1=temp1; temp2=temp2; end
        endcase
    end
always @(posedge clk)
    begin : stateTransition
        case (statebits)
            3: if (reset == 0) statebits = 0;
                else if (ibus['OPCODE] == 'eref) statebits = 1;
                else statebits = 0;
            1: if (reset == 0) statebits = 0;
                else statebits = 2;
            2: if (reset == 0) statebits = 0;
                else if (markbus['BLOCK] == ibus['OPERAND2]) statebits = 7;
                else statebits = 3;
            3: if (reset == 0) statebits = 0;
                else if (markbus['STATIC] == 0) statebits = 12;
                else statebits = 4;
            4: if (reset == 0) statebits = 0;
                else statebits = 5;
            5: if (reset == 0) statebits = 0;
                else statebits = 6;
            6: if (reset == 0) statebits = 0;
                else statebits = 2;
            7: if (reset == 0) statebits = 0;
                else statebits = 8;
            8: if (reset == 0) statebits = 0;
                else statebits = 9;
            9: if (reset == 0) statebits = 0;
                else statebits = 10;
            10: if (reset == 0) statebits = 0;
                else statebits = 11;
            11: if (reset == 0) statebits = 0;
                else statebits = 0;
        endcase
    end
12: if (reset == 0) statebits = 0;
    else statebits = 12;
    default: statebits = 0;
endcase
end
always @(statebits)
begin : combLogic
    case (statebits)
        0: nanoword = 17'hFFFFF;
        1: nanoword = 17'hFFFFF;
        2: nanoword = 17'hFFFFF;
        3: nanoword = 17'hFFFFF;
        4: nanoword = 17'hFFFFF;
        5: nanoword = 17'hFFFFF;
        6: nanoword = 17'hFFFFF;
        7: nanoword = 17'hFFFFF;
        8: nanoword = 17'hFFFFF;
        9: nanoword = 17'hFFFFF;
        10: nanoword = 17'hFFFFF;
        11: nanoword = 17'hFFFFF;
        12: nanoword = 17'hFFFFF;
        default: nanoword = 17'hFFFFF;
    endcase
end //always@(dbus) Sdisplay("eref statebits = %h dbus=%h markreg=%h", statebits, dbus, markbus);
endmodule

--- new module ---
module mreg(clk, d, reset, sl, s0, q);
    parameter RWIDTH = 48;
    parameter RESETVALUE = 'b0;
    input clk, reset, sl, s0;
    input [RWIDTH-1:0] d;
    output [RWIDTH-1:0] q;
    reg [RWIDTH-1:0] qvalue;
assign q = qvalue;
always @(negedge clk)
begin : behavior
    if (reset == 0) qvalue = RESETVALUE;
    else
    case ((sl, s0))
        0: qvalue = qvalue - 1;
        1: qvalue = qvalue - 1;
        2: qvalue = d;
        default: qvalue = qvalue;
    endcase
end
endmodule
APPENDIX C. EXAMPLE MACHINES

```haskell
#* ****************************
#*  \* adder
#*  \* an Euler algorithm to continually sample the input
#*  \* for two numbers and produce the addition as output
#*  \* ****************************
begin
new a;
new b;
label loop;
loop:
a := in;
b := in;
if a != 0 and b != 0 then out a - b else out 35565;
if a = 0 or b = 0 then out 0 else goto loop
end

#*  *************************
#*  \* adder.cmp
#*  \* the translated opcodes and operands
#*  \* created by the Euler-97 compiler according to the
#*  \* translation rules given in the Wirth description
#*  \* of Euler
#*  *************************

<table>
<thead>
<tr>
<th>SourceLine</th>
<th>Opcode</th>
<th>Operand1</th>
<th>Operand2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ebegin</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>enew</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>enew</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>eref</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>ein</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>eassign</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>esemi</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>eref</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>ein</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>eassign</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>esemi</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>eref</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>evalue</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>enumber</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>eneual</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>eand</td>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>eref</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>evalue</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>enumber</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>eneual</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>ethen</td>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>eref</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>evalue</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>eref</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>evalue</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>eadd</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>eout</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>eelse</td>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>enumber</td>
<td>35565</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>eout</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>esemi</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>eref</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>evalue</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>enumber</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>equal</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>eor</td>
<td>41</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>eref</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>evalue</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>enumber</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>eequal</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```
#* adder.syn
#* the synthesis personality for the adder algorithm
#* created by the Euler-97 compiler

**EULER Synthesis Information**

-------------
0  enequal  2
  eassign  2
  esemi   3
  enequal  2
  eref   8
  eadd   1
  eand   1
  ebegin  1
  else   2
  eend   1
  egoto  1
  ein    2
  elabel 1
  enew   2
  enumber 5
  eor    1
  eout   3
  egen   2
  eend  1
  evalue 7
-------------
Total Opcodes Required: 19
Total Opcode Calls: 48

/* adder.v: created by SAMUEL using the -sim option
   this file is ready for simulation, not synthesis
   to remove simulation components and prepare for
   synthesis, recompile using the -sim -synth options
*/

/* typeflags */
'define mark 8'b00000000
'define bool 8'b00000000
'define labl 8'b00000000
'define list 8'b00000000
'define undf 8'b00000000
'define numb 8'b00000000
'define proc 8'b00000000
'define vref 8'b00000000

/* data info */
'define DATAVALE 31:0
'define TYPEFLAG 47:40

/* stack info */
'define STACKMAX 255
'define SPBsns 8

/* fields for readability */
'define BLOCVk 47:40
'define DYNAMIC 39:32
'define VAREND 31:24
'define RETURN 15:0
'define MP 7:0
'define UNDF 39:0
'define NUMBER 15:0
/* control bus info */
#define CBITS 17

/* instruction info */
#define OPCODEBITS 8
#define OPCODE 39:32
#define OPERAND1 31:16
#define OPERAND2 15:0

/* opcode encodings on ibus */
#define enequal 8'h00
#define ebegin 8'h12
#define ehex 8'h22
#define eisn 8'h23
#define eisp 8'h24
#define ein 8'h25
#define einteger 8'h26
#define elisb 8'h27
#define ecall 8'h33
#define edm 8'h34
#define enew 8'h35
#define eprint 8'h36
#define enumber 8'h39
#define eomega 8'h3A
#define eor 8'h3C
#define enext 8'h3D
#define euser 8'h3E
#define etask 8'h3F
#define evalue 8'h40
#define ehalt 8'hFF

/* command bus bits */
module mreg(clk, d, reset, sl, s0, q);  
  parameter HWIDTH = 48;  
  parameter RESETVALUE = 'b0;  
  input clk, reset, sl, s0;  
  input [HWIDTH-1:0] d;  
  output [HWIDTH-1:0] q;  
  reg [HWIDTH-1:0] qvalue;  
  assign q = qvalue;  
  always @(negedge clk)  
  begin : behavior  
  if (reset == 0) qvalue = RESETVALUE;  
  else  
  case ((sl, s0))  
  0: qvalue = qvalue - 1;  
  1: qvalue = qvalue + 1;  
  2: qvalue = d;  
  default: qvalue = qvalue;  
  endcase  
  end  
endmodule  
module mux41(in3, in2, in1, in0, outdata, sl, s0);  
  parameter datawidth = 8;  
  input [datawidth-1:0] in3;  
  input [datawidth-1:0] in2;  
  input [datawidth-1:0] in1;  
  input [datawidth-1:0] in0;  
  output [datawidth-1:0] outdata;  
  input sl;  
  input s0;  
  assign outdata = (sl, s0) == 2'b11 ? in3 :  
  (sl, s0) == 2'b10 ? in2 :  
  (sl, s0) == 2'b01 ? in1 :  
  (sl, s0) == 2'b00 ? in0 : in0;  
endmodule  
module tsbuf(d, dr, q);  
  parameter bwidth = 48;  
  input [bwidth-1:0] d;  
  input dr;  
  output [bwidth-1:0] q;  
  assign q = (dr == 0) ? d : 48'bz;  
endmodule  
module mem(abus, cs, dbus, we);  
  input [7:0] abus;  

input cs;
inout [47:0] dbus;
input we;
reg [47:0] values[0:255];
assign dbus = (cs || (we == 0)) ? 48'b2 : values[abus];
always @(negedge cs)
begin : writeMe
  if (we==0)
    begin
      values[abus] = dbus;
      //display("cs=%b, we=%b, abus = %h, dbus=%h", cs, we, abus, dbus);
    end
end
//always @(dbus) $display("cs=%b,we=%b.abus = %h, dbus=%h", cs, we, abus, dbus);
initial
begin : memInit
  integer i;
  for (i=0; i<256; i=i+1) values[i] = 48'b0;
end
endmodule

`include "adder.rom"

module enequal(clk, opcode, reset, cbus, dbus);
input clk;
inout [7:0] opcode;
input reset;
output [16:0] cbus;
inout [47:0] dbus;
reg [2:0] statebits;
reg [16:0] nanoword;
reg [47:0] number1;
reg [47:0] number2;
assign dbus["TYPEFLAG1 = (statebits == 5) ? 8'b1; else : 8'b0;
assign dbus[39:16] = (statebits == 5)?24'b0:24'b1;
assign dbus["NUMBER1 = (statebits == 5)?number1["NUMBER1 =number2["NUMBER1 :16'b1;
assign cbus = (statebits == 0)?17'b1:nanoword;
always @(negedge clk)
begin : tempLatch
  case (statebits)
    1: number2 = dbus;
    3: number1 = dbus;
    default: begin number1 = number1; number2 = number2; end
  endcase
end
always @(posedge clk)
begin : stateTransition
  case (statebits)
    0: if (reset == 0) statebits = 0;
    else if (opcode == 'enequal) statebits = 1;
    else statebits = 0;
    1: if (reset == 0) statebits = 0;
    else statebits = 2;
    2: if (reset == 0) statebits = 0;
    else statebits = 3;
    3: if (reset == 0) statebits = 0;
    else statebits = 4;
    4: if (reset == 0) statebits = 0;
    else if (number1["TYPEFLAG] == numflag & number2["TYPEFLAG] == numflag)
      statebits = 5;
    else statebits = 7;
    5: if (reset == 0) statebits = 0;
    else statebits = 6;
    6: if (reset == 0) statebits = 0;
  endcase
end
module eassign(clk, opcode, markbus, reset, cbus, dbus);
input clk;
input [7:0] opcode;
input [47:0] markbus;
input reset;
output [16:0] cbus;
inout [47:0] dbus;
reg [3:0] statebits;
reg [16:0] nanoword;
reg [47:0] lvalue;
reg [47:0] rvalue;
reg [47:0] temp;
assign dbus = (statebits == 5)?40'b0,lvalue[7:0]} : (statebits == 7 || statebits == 10)? rvalue : (statebits == 8)? temp : 48'bz;
assign cbus = (statebits == 0)?17'bz:nanoword;
always @(posedge clk)
begin : tempLatch
  case (statebits)
    1: rvalue = dbus;
    3: lvalue = dbus;
    4: temp = markbus;
    default: begin lvalue=lvalue; rvalue=rvalue; temp=temp; end
  endcase
end
always @(posedge clk)
begin : stateTransistion
  case (statebits)
    0: if (reset == 0) statebits = 0;
    else statebits = 1;
    else statebits = 2;
    2: if (reset == 0) statebits = 3;
    else statebits = 4;
    4: if (reset == 0) statebits = 0;
    else if (lvalue['TYPEFLAG]!='vref) statebits = 12;
    else statebits = 5;
    5: if (reset == 0) statebits = 0;
    else statebits = 6;
    6: if (reset == 0) statebits = 0;
    else statebits = 7;
    7: if (reset == 0) statebits = 0;
    else statebits = 8;
    8: if (reset == 0) statebits = 0;
  endcase
end
else statebits = 9;
9: if (reset == 0) statebits = 0;
else statebits = 10;
10: if (reset == 0) statebits = 0;
else statebits = 11;
11: if (reset == 0) statebits = 0;
else statebits = 12;
12: if (reset == 0) statebits = 0;
else statebits = 12;
default: statebits = 0;
endcase
end
always @(statebits)
begin : combLogic
case (statebits)
0: nanoword = 17'hFFFF;
1: nanoword = 17'hFFDF;
2: nanoword = 17'hFFEF;
3: nanoword = 17'hFFEF;
4: nanoword = 17'hFFEF;
5: nanoword = 17'hFFEF;
6: nanoword = 17'hFFEF;
7: nanoword = 17'hFFEF;
8: nanoword = 17'hFFEF;
9: nanoword = 17'hFFEF;
10: nanoword = 17'hFFEF;
11: nanoword = 17'hFFEF;
12: nanoword = 17'hFFEF;
default: nanoword = 17'hFFEF;
endcase
end
endmodule

module esemi(clk,opcode,reset,cbus);
input clk;
input [7:0] opcode;
input reset;
output [16:0] cbus;
reg [1:0] statebits;
reg [16:0] nanoword;
assign cbus = (statebits == 0) ? 17'b0 : nanoword;
always @(posedge clk)
begin : stateTransistion
case (statebits)
0: if (reset == 0) statebits = 0;
else if (opcode == 'esemi) statebits = 1;
else statebits = 0;
1: if (reset == 0) statebits = 0;
else statebits = 2;
2: if (reset == 0) statebits = 0;
else statebits = 0;
default: statebits = 0;
endcase
end
always @(statebits)
begin : combLogic
case (statebits)
0: nanoword = 17'hFFFF;
1: nanoword = 17'hFFDF;
2: nanoword = 17'hFFEF;
default: nanoword = 17'hFFFF;
endcase
end
endmodule

module equal(clk,opcode,reset,cbus,dbus);
input clk;
input [7:0] opcode;
input reset;
output [16:0] cbus;
module eref(clk, ibus, markbus, reset, cbus, dbus);
    input clk;
    input [39:0] ibus;
    input [47:0] markbus;
    input reset;
    output [16:0] cbus;
    input [47:0] dbus;
    reg [3:0] statebits;
    reg [16:0] nanoword;
    reg [47:0] number1;
    reg [47:0] number2;
    assign dbus['TYPEFLAGI = (statebits == 5)?'bool:9'bz;
    assign dbus[39:16] = (statebits == 5)?24'b0:24'bz;
    assign dbus['NUMBER] = (statebits == 5)?number1['NUMBER]==number2['NUMBER]:16'bz;
    assign cbus = (statebits == 0)?17'b0:nanoword;
    always @(negedge clk)
    begin : tempLatch
        case (statebits)
            1: number2 = dbus;
            3: number1 = dbus;
        default: begin number1 = number1; number2 = number2; end
        endcase
    end
    always @(posedge clk)
    begin : stateTransition
        case (statebits)
            0: if (reset == 0) statebits = 0;
            else if (opcode == 'equal) statebits = 1;
                else statebits = 0;
            1: if (reset == 0) statebits = 3;
                else statebits = 2;
            2: if (reset == 0) statebits = 0;
                else statebits = 3;
            3: if (reset == 0) statebits = 0;
                else statebits = 4;
            4: if (reset == 0) statebits = 0;
                else if (number1['TYPEFLAG] == numb & & number2['TYPEFLAG] == numb)
                    statebits = 5;
                else statebits = 7;
            5: if (reset == 0) statebits = 0;
                else statebits = 6;
            6: if (reset == 0) statebits = 0;
                else statebits = 0;
            7: if (reset == 0) statebits = 0;
                else statebits = 7;
        default: statebits = 0;
        endcase
    end
    always @(statebits)
    begin : combLogic
        case (statebits)
            0: nanoword = 17'h1FFFF;
            1: nanoword = 17'h1FFDF;
            2: nanoword = 17'h1FCFF;
            3: nanoword = 17'h1FFDF;
            4: nanoword = 17'h1FFEF;
            5: nanoword = 17'h1FFCF;
            6: nanoword = 17'h1FFFF;
            7: nanoword = 17'h0FFFF;
        default: nanoword = 17'hFFFF;
        endcase
    end
endmodule
assign dbus[\`TYPEFLAG] = (statebits == 5)?temp2[\`TYPEFLAG] : (statebits == 7)?temp1[\`TYPEFLAG] :
(statebits == 10)?'0':bz;
assign dbus[39:8] = (statebits == 5)?temp2[39:8] :
(statebits == 7)?temp1[39:8] :
(statebits == 10)?'32'b0 : '32'bz;
assign dbus[7:0] = (statebits == 5)?temp2[7:0] :
(statebits == 7)?temp1[7:0] :
(statebits == 10)?temp2[7:0]-ibus[\`OPERAND1] : '8'bz;
assign cbus = (statebits == 0)?17'h0:nanoword;

always @(negedge clk)
begin : tempLatch
  case (statebits)
    1: begin temp1 = markbus; temp2 = markbus; end
    5: temp2 = dbus;
    default: begin temp1=temp1; temp2=temp2; end
  endcase
end

always @(posedge clk)
begin : stateTransition
  case (statebits)
    0: if (reset == 0) statebits = 0;
       else if (ibus[\`OPCODE] == '0ref') statebits = 1;
       else statebits = 5;
    1: if (reset == 0) statebits = 0;
       else statebits = 2;
    2: if (reset == 0) statebits = 0;
       else if (markbus[\`BLOCK] == ibus[\`OPERAND2]) statebits = 7;
       else statebits = 3;
    3: if (reset == 0) statebits = 0;
       else if (markbus[\`STATIC] == 0) statebits = 12;
       else statebits = 4;
    4: if (reset == 0) statebits = 0;
       else statebits = 5;
    5: if (reset == 0) statebits = 0;
       else statebits = 6;
    6: if (reset == 0) statebits = 0;
       else statebits = 7;
    7: if (reset == 0) statebits = 0;
       else statebits = 8;
    8: if (reset == 0) statebits = 0;
       else statebits = 9;
    9: if (reset == 0) statebits = 0;
       else statebits = 10;
    10: if (reset == 0) statebits = 0;
       else statebits = 11;
    11: if (reset == 0) statebits = 0;
       else statebits = 12;
    12: if (reset == 0) statebits = 0;
       else statebits = 12;
    default: statebits = 0;
  endcase
end

always @(statebits)
begin : combLogic
  case (statebits)
    0: nanoword = 17'hFFFF;
    1: nanoword = 17'hFFFF;
    2: nanoword = 17'hFFFF;
    3: nanoword = 17'hFFFF;
    4: nanoword = 17'hFFFF;
    5: nanoword = 17'hFFFF;
    6: nanoword = 17'hFFFF;
    7: nanoword = 17'hFFFF;
    8: nanoword = 17'hFFFF;
    9: nanoword = 17'hFFFF;
    10: nanoword = 17'hFFFF;
    11: nanoword = 17'hFFFF;
    12: nanoword = 17'hFFFF;
    default: nanoword = 17'hFFFF;
module eadd(clk, opcode, reset, cbus, dbus);
input clk;
inint [7:0] opcode;
inint reset;
output [16:0] cbus;
inout [47:0] dbus;

reg [2:0] statebits;
reg [16:0] nanoword;
reg [47:0] number1;
reg [47:0] number2;

assign dbus['TYPEFLAG! = (statebits == 5) ? 'numb: 3 ' bs;
assign dbus[39:16] = (scacebics == 5)? 24 'bO:24 • bz,
assign number1 = (statebits == 5)? number1['NUMBER]-number2['NUMBER]:16 'b;
assign cbus = (statebits == 0)? 17 'bz: nanoword;

always @(negedge clk)
begin : tempLatch
  case (statebits)
    1: number2 = dbus;
    3: number1 = dbus;
    default: begin number1 = number1; number2 = number2; end
  endcase
end

always @(posedge die)
begin : stateTransistion
  case (statebits)
    0: if (reset == 0) statebits = 0;
        else if (opcode == 'eadd) statebits = 1;
        else statebits = 0;
    1: if (reset == 0) statebits = 0;
        else statebits = 2;
    2: if (reset == 0) statebits = 0;
        else statebits = 3;
    3: if (reset == 0) statebits = 0;
        else statebits = 4;
    4: if (reset == 0) statebits = 0;
        else if (number1['TYPEFLAG] == 'numb && number2['TYPEFLAG] == 'numb)
          statebits = 5;
        else statebits = 6;
    5: if (reset == 0) statebits = 0;
        else statebits = 6;
    6: if (reset == 0) statebits = 0;
        else statebits = 7;
    7: if (reset == 0) statebits = 0;
        else statebits = 7;
    default: statebits = 0;
  endcase
end

always @(statebits)
begin : combLogic
  case (statebits)
    0: nanoword = 17'hFFFF;
    1: nanoword = 17'h1FFF;
    2: nanoword = 17'h1FCF;
    3: nanoword = 17'h1FFD;
    4: nanoword = 17'h1FEE;
    5: nanoword = 17'h1FFCF;
    6: nanoword = 17'h1FFFF;
    7: nanoword = 17'h0FFFF;
    default: nanoword = 17'h1FFFF;
  endcase
end

module eadd(clk, ibus, reset, cbus, dbus);
input clk;
input [39:0] ibus;
input reset;
output [16:0] cbus;
inout [47:0] dbus;

reg [3:0] statebits;
reg [16:0] nanoword;
reg [47:0] temp;

assign dbus = (statebits == 5 || statebits == 6) ? temp : (statebits == 7) ? (32'bo, ibus['OPCODE']) : 48'bz;
assign cbus = (statebits == 0)?17'bz:nanoword;

always @(negedge clk)
begin : tempLatch
  case (statebits)
    1: temp = dbus;
    default: temp = temp;
  endcase
end

always @(posedge clk)
begin : stateTransition
  case (statebits)
    0: if (reset == 0) statebits = 0;
    else if (ibus['OPCODE] == 'eand) statebits = 1;
    else statebits = 0;
    1: if (reset == 0) statebits = 0;
    else statebits = 1;
    2: if (reset == 0) statebits = 0;
    else if (temp['TYPEFLAG] != 'bool) statebits = 9;
    else statebits = 3;
    3: if (reset == 0) statebits = 0;
    else if (temp['NUMBER] != 0) statebits = 8;
    else statebits = 4;
    4: if (reset == 0) statebits = 0;
    else statebits = 5;
    5: if (reset == 0) statebits = 0;
    else statebits = 6;
    6: if (reset == 0) statebits = 0;
    else statebits = 7;
    7: if (reset == 0) statebits = 0;
    else statebits = 0;
    8: if (reset == 0) statebits = 0;
    else statebits = 0;
    9: if (reset == 0) statebits = 0;
    else statebits = 9;
    default: statebits = 0;
  endcase
end

always @(statebits)
begin : combLogic
  case (statebits)
    0: nanoword = 17'hFFFF;
    1: nanoword = 17'hFFFD;
    2: nanoword = 17'hFFDF;
    3: nanoword = 17'hFDFD;
    4: nanoword = 17'hFDDF;
    5: nanoword = 17'hFDFE;
    6: nanoword = 17'hFFFF;
    7: nanoword = 17'hFFFF;
    8: nanoword = 17'hFDFD;
    9: nanoword = 17'hFDDF;
    default: nanoword = 17'hFFFF;
  endcase
end
endmodule
assign dbus('TYPEFLAG') = (statebits==3 || statebits==4) ? markbus('BLOCK'): 'b1:8' 'bz;
assign dbus('DYNAMIC') = (statebits==3 || statebits==4) ? markbus('MP'): 'b1:8' 'bz;
assign dbus('STATIC') = (statebits==3 || statebits==4) ? markbus('MP'): 'b1:8' 'bz;
assign dbus('VAREND') = (statebits==3 || statebits==4) ? spbus: 'b1:8' 'bz;
assign dbus('RETURN') = (statebits==3 || statebits==4) ? spbus: 'b1:8' 'bz;
always @(posedge clk)
begin : stateTransition
  case (statebits)
    0: if (reset == 0) statebits = 3;
      else if (opcode == 'e') statebits = 1;
      else statebits = 0;
    1: if (reset == 0) statebits = 0;
      else statebits = 2;
    2: if (reset == 0) statebits = 0;
      else statebits = 3;
    3: if (reset == 0) statebits = 0;
      else statebits = 4;
    4: if (reset == 0) statebits = 0;
      else statebits = 5;
    5: if (reset == 0) statebits = 0;
      else statebits = 0;
    default: statebits = 0;
  endcase
end
always @(statebits)
begin : combLogic
  case (statebits)
    0: nanoword = 17'h1FFFF;
    1: nanoword = 17'h1FDFF;
    2: nanoword = 17'h1FFEF;
    3: nanoword = 17'h1FFCF;
    4: nanoword = 17'h1FFFB;
    5: nanoword = 17'h17FFF;
      default: nanoword = 17'h1FFFF;
  endcase
end
module else(clk, ibus, reset, cbus, dbus);
  input clk;
  input [39:0] ibus;
  input reset;
  output [16:0] cbus;
  output [47:0] dbus;
  reg [1:0] statebits;
  reg [16:0] nanoword;
assign dbus = (statebits == 1)?(32'b0, ibus['OPERAND1]): 48'b1;
assign cbus = (statebits == 0)?17'b0: nanoword;
always @(posedge clk)
begin : stateTransition
  case (statebits)
    0: if (reset == 0) statebits = 0;
     else if (ibus['OPCODE'] == 'else) statebits = 1;
      else statebits = 0;
    1: if (reset == 0) statebits = 0;
    2: if (reset == 0) statebits = 0;
    default: statebits = 0;
  endcase
end
always @(statebits)
begin : combLogic
  case (statebits)
    0: nanoword = 17'h1FFFF;
    1: nanoword = 17'h1FDFF;
    2: nanoword = 17'h1FFEF;
    3: nanoword = 17'h1FFCF;
    4: nanoword = 17'h1FFFB;
    5: nanoword = 17'h17FFF;
    default: nanoword = 17'h1FFFF;
  endcase
end
module eend(clk,opcode,reset,cbus,dbus);
  input clk;
  input [7:0] opcode;
  input reset;
  output [16:0] cbus;
  inout [47:0] dbus;

  reg [3:0] statebits;
  reg [16:0] nanoword;
  reg [47:0] temp;

  assign dbus = (statebits == 4) || (statebits == 7) ? temp : 48'bz;
  assign cbus = (statebits == 0)?17'hFFFF:nanoword;

always @(negedge clk)
begin : tempLatch
  case (statebits)
    1: temp = cbus;
    2: temp = dbus;
    default: temp = temp;
  endcase
end

always @(posedge clk)
begin : stateTransistion
  case (statebits)
    0: if (reset == 0) statebits = 0;
      else if (opcode == 'eend) statebits = 1;
      else statebits = 0;
    1: if (reset == 0) statebits = 0;
      else statebits = 2;
    2: if (reset == 0) statebits = 0;
      else statebits = 3;
    3: if (reset == 0) statebits = 0;
      else statebits = 4;
    4: if (reset == 0) statebits = 0;
      else statebits = 5;
    5: if (reset == 0) statebits = 0;
      else statebits = 6;
    6: if (reset == 0) statebits = 0;
      else statebits = 7;
    7: if (reset == 0) statebits = 0;
      else statebits = 8;
    8: if (reset == 0) statebits = 0;
      else statebits = 9;
    9: if (reset == 0) statebits = 0;
      else statebits = 0;
      default: statebits = 0;
  endcase
end

always @(statebits)
begin : combLogic
  case (statebits)
    0: nanoword = 17'hFFFF;
    1: nanoword = 17'h1FFDF;
    2: nanoword = 17'h1FFEF;
    3: nanoword = 17'h1FFEF;
    4: nanoword = 17'h1FFCF;
    5: nanoword = 17'h1FFBF;
    6: nanoword = 17'h1FF9F;
    7: nanoword = 17'h1FFFB;
    8: nanoword = 17'h17FFF;
      default: nanoword = 17'h1FFFF;
  endcase
end
endmodule

module egoto(clk,opcode,reset,cbus,dbus);
input clk;
input [7:0] opcode;
input reset;
output [16:0] dbus;
input [47:0] cbus;

reg [3:0] statebits;
reg [16:0] nanoword;
reg [47:0] label;
reg [47:0] temp;

assign dbus = (statebits == 4)?(32'b0,label["LABELP2]) : 
(statebits == 7)?temp : 
(statebits == 9)?(32'b0,label["LABELP1]) :
4'b0;

assign cbus = (statebits == 0)?17'b0:nanoword;

//always @(statebits) $display("egoto: statebits = %h, temp=%h",statebits,temp);
always @(negedge clk)
begin : tempLatch
  case (statebits)
    1: label = dbus;
    5 : temp = dbus;
    default: begin label = label; temp = temp; end
  endcase
end

always @(posedge clk)
begin : stateTransition
  case (statebits)
    0: if (reset == 0) statebits = 0;
    else if (opcode == egoto) statebits = 1;
    else statebits = 0;
    1: if (reset == 0) statebits = 0;
    else statebits = 2;
    2: if (reset == 0) statebits = 0;
    else statebits = 3;
    3: if (reset == 0) statebits = 0;
    else if (label["TYPEFLAG]!="labl) statebits = 10;
    else statebits = 4;
    4: if (reset == 0) statebits = 0;
    else statebits = 5;
    5: if (reset == 0) statebits = 0;
    else statebits = 6;
    6: if (reset == 0) statebits = 0;
    else statebits = 7;
    7: if (reset == 0) statebits = 0;
    else statebits = 8;
    8: if (reset == 0) statebits = 0;
    else statebits = 9;
    9: if (reset == 0) statebits = 0;
    else statebits = 0;
    10: if (reset == 0) statebits = 0;
    else statebits = 10;
    default: statebits = 0;
  endcase
end

always @(statebits)
begin : combLogic
  case (statebits)
    0: nanoword = 17'hFFFFFF;
    1: nanoword = 17'h1FFDF;
    2: nanoword = 17'h1FCFF;
    3: nanoword = 17'h1FFFF;
    4: nanoword = 17'h1FFFF;
    5: nanoword = 17'h1FFFD;
    6: nanoword = 17'h1FFBF;
    7: nanoword = 17'h1FFFB;
    8: nanoword = 17'h1FFFE;
    9: nanoword = 17'h1BFFFF;
    10: nanoword = 17'h0FFFF;
    default: nanoword = 17'h1FFFF;
  endcase
end
module ein(clk, opcode, reset, cbus);
  input clk;
  input [7:0] opcode;
  input reset;
  output [16:0] cbus;

  reg [2:0] statebits;
  reg [16:0] nanoword;

  assign cbus = (statebits == 0)?17'bz:nanoword;

  always @(posedge clk)
  begin : stateTransition
    case (statebits)
      0: if (reset == 0) statebits = 0;
      else if (opcode == 'ein) statebits = 1;
      else statebits = 0;
      1: if (reset == 0) statebits = 0;
      else statebits = 2;
      2: if (reset == 0) statebits = 0;
      else statebits = 3;
      3: if (reset == 0) statebits = 0;
      else statebits = 4;
      4: if (reset == 0) statebits = 0;
      else statebits = 0;
      default: statebits = 0;
    endcase
  end

  always @(statebits)
  begin : combLogic
    case (statebits)
      0: nanoword = 17'h1FFF;
      1: nanoword = 17'h1FFF;
      2: nanoword = 17'h1FFF;
      3: nanoword = 17'h1FFF;
      4: nanoword = 17'h1FFF;
      default: nanoword = 17'h1FFF;
    endcase
  end
endmodule

module elabel(elk, ibus, reset, cbus, dbus);
  input [39:0] ibus;
  input reset;
  output [16:0] cbus;
  output [47:0] dbus;

  reg [2:0] statebits;
  reg [16:0] nanoword;

  assign dbus['TYPEFLAG] = (statebits==3)? 'labl:8'bz;
  assign dbus['LABELUNUSED] = (statebits == 3)? 'b0:8'bz;
  assign dbus['LABELOP1] = (statebits==3)?ibus['OPERAND1]:16'bz;
  assign dbus['LABELOP2] = (statebits==3)?ibus['OPERAND2]:16'bz;
  assign cbus = (statebits == 0)?17'bz:nanoword;

  always @(posedge clk)
  begin : stateTransition
    case (statebits)
      0: if (reset == 0) statebits = 0;
      else if (ibus['OPCODE] == 'elabel) statebits = 1;
      else statebits = 0;
      1: if (reset == 0) statebits = 0;
      else statebits = 2;
      2: if (reset == 0) statebits = 0;
      else statebits = 3;
      3: if (reset == 0) statebits = 0;
      else statebits = 4;
4: if (reset == 0) statebits = 0;
   else statebits = 0;
   default: statebits = 0;
endcase
end

always @(statebits)
begin : comblogic
   case (statebits)
      0: nanoword = 1'hFFFF;
      1: nanoword = 1'hFDFF;
      2: nanoword = 1'hFFE7;
      3: nanoword = 1'hFFC7;
      4: nanoword = 1'h7FFF;
      default: nanoword = 1'hFFFF;
   endcase
end
endmodule

defmodule enew(clk,markbus,opcode,reset,cbus,dbus);
   input clk;
   input [47:0] markbus;
   input [7:0] opcode;
   input reset;
   output [16:0] cbus;
   output [47:0] dbus;
   reg [2:0] statebits;
   reg [47:0] temp;
   reg [16:0] nanoword;

   assign dbus = (statebits == 3) ? 'undf, 40'b0 : (statebits == 4 || statebits == 6) ? temp : 48'bz;
   assign cbus = (statebits == 0) ? 17'b0 : nanoword;
always @(negedge clk)
begin : latchTemp
   case (statebits)
      1: temp = markbus;
      2: begin
         temp[ 'BLOCK] = temp[ 'BLOCK];
         temp[ 'DYNAMIC] = temp[ 'DYNAMIC];
         temp[ 'STATIC] = temp[ 'STATIC];
         temp[ 'VAREND] = temp[ 'VAREND] - 1;
         temp[ 'RETURN] = temp[ 'RETURN];
         temp[ 'MP] = temp[ 'MP];
      end
      default: temp = temp;
   endcase
end
always @(posedge clk)
begin : stateTransition
   case (statebits)
      0: if (reset == 0) statebits = 0;
         else if (opcode == 'enew) statebits = 1;
         else statebits = 0;
      1: if (reset == 0) statebits = 0;
         else statebits = 2;
      2: if (reset == 0) statebits = 0;
         else statebits = 3;
      3: if (reset == 0) statebits = 0;
         else statebits = 4;
      4: if (reset == 0) statebits = 0;
         else statebits = 5;
      5: if (reset == 0) statebits = 0;
         else statebits = 6;
      6: if (reset == 0) statebits = 0;
         else statebits = 7;
      7: if (reset == 0) statebits = 0;
         else statebits = 0;
      default: statebits = 0;
   endcase
end
always @(statebits)
begin : combLogic
  case (statebits)
  0: nanoword = 17'h1FFFF;
  1: nanoword = 17'h1FDF;
  2: nanoword = 17'h1FEE;
  3: nanoword = 17'h1FFCF;
  4: nanoword = 17'h1FFFB;
  5: nanoword = 17'h1FF2F;
  6: nanoword = 17'h1FF0F;
  7: nanoword = 17'h1FFF;
  default: nanoword = 17'h1FFFF;
  endcase
end
endmodule

module enumber(clk,ibus,reset,cbus,dbus);
  input clk;
  input [39:0] ibus;
  input reset;
  output [16:0] cbus;
  output [47:0] dbus;
  reg [2:0] statebits;
  reg [16:0] nanoword;

  assign dbus['TYPEFLAG] = (statebits==3) ? "numb:8'bz;
  assign dbus[39:16] = (statebits==3) ? 24'b0:24'bz;
  assign dbus['NUMBER] = (statebits==3) ? ibus['OPERAND1] : 16'bz;
  assign cbus = (statebits == 0) ? 17'b2: nanoword;

  always @(posedge clk)
  begin : stateTransition
    case (statebits)
    0: if (reset == 0) statebits = 0;
    else if (ibus['OPCODE] == 'enumber) statebits = 1;
    else statebits = 0;
    1: if (reset == 0) statebits = 0;
    else statebits = 2;
    2: if (reset == 0) statebits = 0;
    else statebits = 3;
    3: if (reset == 0) statebits = 0;
    else statebits = 4;
    4: if (reset == 0) statebits = 0;
    else statebits = 0;
    default: statebits = 0;
    endcase
  end

  always @(statebits)
  begin : combLogic
    case (statebits)
    0: nanoword = 17'h1FFFF;
    1: nanoword = 17'h1FDF;
    2: nanoword = 17'h1FEE;
    3: nanoword = 17'h1FFCF;
    4: nanoword = 17'h1FFFB;
    default: nanoword = 17'h1FFFF;
    endcase
  end
endmodule

module eor(clk,ibus,reset,cbus,dbus);
  input clk;
  input [39:0] ibus;
  input reset;
  output [16:0] cbus;
  inout [47:0] dbus;
  reg [3:0] statebits;
  reg [16:0] nanoword;
  reg [47:0] temp;

  assign dbus = (statebits == 5 || statebits == 6) ? temp :
    (statebits == 7) ? (32'b0,ibus['OPERAND1]) : 48'bz;
assign cbus = (statebits == 0)?17'bz:nanoword;

always @(negedge clk)
begin : tempLatch
  case (statebits)
    1: temp = dbus;
    default: temp = temp;
  endcase
end

always @(posedge clk)
begin : stateTransistion
  case (statebits)
    0: if (reset == 0) statebits = 0;
       else if ([bus[OPCODE] == 'eor) statebits = 1;
       else statebits = 0;
    1: if (reset == 0) statebits = 0;
       else statebits = 2;
    2: if (reset == 0) statebits = 0;
       else if (temp[TYPEFLAG] != 'bool) statebits = 9;
       else statebits = 3;
    3: if (reset == 0) statebits = 0;
       else if (temp[NUMBER] != 0) statebits = 9;
       else statebits = 4;
    4: if (reset == 0) statebits = 0;
       else statebits = 5;
    5: if (reset == 0) statebits = 0;
       else statebits = 6;
    6: if (reset == 0) statebits = 0;
       else statebits = 7;
    7: if (reset == 0) statebits = 0;
       else statebits = 9;
    8: if (reset == 0) statebits = 0;
    9: if (reset == 0) statebits = 0;
       else statebits = 9;
       default: statebits = 0;
  endcase
end

always @(statebits)
begin : combLogic
  case (statebits)
    0: nanoword = 17'h1FFFF;
    1: nanoword = 17'h1FFDF;
    2: nanoword = 17'h1FCEF;
    3: nanoword = 17'h1FFFE;
    4: nanoword = 17'h1FDFF;
    5: nanoword = 17'h1FFFE;
    6: nanoword = 17'h1FCHF;
    7: nanoword = 17'h1BFFFF;
    8: nanoword = 17'h17FFFF;
    9: nanoword = 17'h0FFFF;
    default: nanoword = 17'h1FFFF;
  endcase
end
endmodule

module eout(clk,opcode,reset,cbus);
  input clk;
  input [7:0] opcode;
  input reset;
  output [16:0] cbus;

  reg [11:0] statebits;
  reg [16:0] nanoword;

assign cbus = (statebits == 0)?17'bz:nanoword;

always @(posedge clk)
begin : stateTransistion
  case (statebits)
    0: if (reset == 0) statebits = 0;
       else if (opcode == 'eout) statebits = 1;
       else statebits = 0;
    1: ...
if (reset == 0) statebits = 0;
    else statebits = 2;
2: if (reset == 0) statebits = 0;
    else statebits = 0;
default: statebits = 0;
endcase
end

always @(statebits)
begin : combLogic
  case (statebits)
    0: nanoword = 17'h1FFFF;
    1: nanoword = 17'h1FFDF;
    2: nanoword = 17'h1FFFF;
    default: nanoword = 17'h1FFFF;
  endcase
end

module even(clk, ibus, reset, cbus, dbus);
  input clk;
  input [39:0] ibus;
  input reset;
  output [16:0] cbus;
  inout [47:0] dbus;
  reg [2:0] statebits;
  reg [16:0] nanoword;
  reg [47:0] temp;

  assign dbus[47:16] = (statebits == 5)?32'b0:32'bz;
  assign dbus[15:0] = (statebits == 5)?ibus['OPERAND1]:16'b0;
  assign cbus = (statebits == 0)?17'b0:nanoword;

  always @(negedge clk)
  begin : tempLatch
    case (statebits)
      1: temp = dbus;
    default: begin temp = temp; end
  endcase
end

always @(posedge clk)
begin : stateTransistion
  case (statebits)
    0: if (reset == 0) statebits = 0;
        else if (ibus['OPCODE] == 'ethen) statebits = 1;
        else statebits = 0;
    1: if (reset == 0) statebits = 0;
        else statebits = 0;
    2: if (reset == 0) statebits = 0;
        else if (temp['TYPEFLAG] != 'bool) statebits = 6;
        else statebits = 3;
    3: if (reset == 0) statebits = 0;
        else if (temp[15:0] == 16'b0) statebits = 5;
        else statebits = 4;
    4: if (reset == 0) statebits = 0;
        else statebits = 0;
    5: if (reset == 0) statebits = 0;
        else statebits = 0;
    6: if (reset == 0) statebits = 0;
        else statebits = 6;
    default: statebits = 0;
  endcase
end

always @(statebits)
begin : combLogic
  case (statebits)
    0: nanoword = 17'h1FFFF;
    1: nanoword = 17'h1FFDF;
    2: nanoword = 17'h1FCCF;
    3: nanoword = 17'h1FFFF;
    4: nanoword = 17'h17FFFF;
    5: nanoword = 17'h1BFFFF;
    6: nanoword = 17'h0FFFF;
module evalue (elk, opcode, markbus, reset, cbus, dbus);
    input elk;
    input [7:0] opcode;
    input [47:0] markbus;
    input reset;
    output [16:0] cbus;
    inout [47:0] dbus;

    reg [3:0] statebits;
    reg [16:0] nanoword;
    reg [47:0] temp1;
    reg [47:0] temp2;

    assign dbus = (statebits == 4) ? {40'h0, temp1[7:0]} : (statebits == 7) ? temp2 : (statebits == 9) ? temp1 : 48'hbz;
    assign cbus = (statebits == 0) ? 17'hFFFF

always @(negedge elk)
begin : tempLatch
    case (statebits)
        1: temp1 = dbus;
        3: temp2 = markbus;
        6: temp1 = dbus;
        default: begin temp1 = temp1; temp2 = temp2; end
    endcase
end

always @(posedge elk)
begin : stateTransistion
    case (statebits)
        0: if (reset == 0) statebits = 0;
            else if (opcode == 'evalue) statebits = 1;
            else statebits = 0;
        1: if (reset == 0) statebits = 0;
            else statebits = 2;
        2: if (reset == 0) statebits = 0;
            else if (temp1[TYPEFLAG]!='vref) statebits = 10;
            else statebits = 1;
        3: if (reset == 0) statebits = 0;
            else statebits = 4;
        4: if (reset == 0) statebits = 0;
            else statebits = 5;
        5: if (reset == 0) statebits = 0;
            else statebits = 6;
        6: if (reset == 0) statebits = 0;
            else statebits = 7;
        7: if (reset == 0) statebits = 0;
            else statebits = 8;
        8: if (reset == 0) statebits = 0;
            else statebits = 9;
        9: if (reset == 0) statebits = 0;
            else statebits = 10;
        10: if (reset == 0) statebits = 0;
            else if (temp1[TYPEFLAG]!='proc) statebits = 12;
            else statebits = 11;
        11: if (reset == 0) statebits = 0;
            else statebits = 0;
        12: if (reset == 0) statebits = 0;
            else statebits = 12;
        default: statebits = 0;
    endcase
end

always @(statebits)
begin : combLogic
    case (statebits)
        0: nanoword = 17'hFFFF;
        1: nanoword = 17'hFFDF;
        2: nanoword = 17'hFFFF;
module machine(clk, error, inport, outport, reset);
input clk;
output error;
input[47:0] inport;
output [47:0] outport;
input reset;
wire [47:0] dbus;
wire [7:0] spbus; /* busses */
wire [7:0] sabus;
wire [7:0] iabus;
wire [7:0] markbus;
wire [15:0] dbus;
wire [47:0] inputbus;
/* instantiate members of R from M = {R,0} */
mreg inputreg(clk, inport, reset, 1'b1, 1'b0, inputbus);
tsbuf inputbuf(inputbus, dbus);  
mreg outputreg(clk, dbus, reset, 1'b1, 1'b0, outputbus);
msel smux(spbus, markbus["DYNAMIC"], markbus["STATIC"], markbus["MP"], sabus, dbus["smux1", dbus["smux0"]]);
mem stack(sabus, dbus["scs", dbus["swe"]]);
from imemory(iabus, dbus);
assign error = dbus["error"];  
/* instantiate members of O from M = {R,0} */
esemi semiop(clk, iabus["OPCODE"], reset, dbus);
eassign assignop(clk, iabus["OPCODE"], markbus, reset, iabus, dbus);
eadd addop(clk, iabus["OPCODE"], reset, efaus, dbus);
eand andop(clk, iabus, reset, cbus, dbus);
ebegin beginop(clk, markbus, iabus["OPCODE"], reset, spbus, dbus);
eelse elseop(clk, iabus, reset, cbus, dbus);
eend endop(clk, iabus["OPCODE"], reset, efaus, dfaus);
egoto gotoop(clk, iabus["OPCODE"], reset, dbus);
ein inop(clk, iabus["OPCODE"], reset, cbus);
e label labelop(clk, iabus, reset, cbus, dbus);
enew newop(clk, markbus, iabus["OPCODE"], reset, iabus, dbus);
enumber numberop(clk, iabus, reset, cbus, dbus);
eor orop(clk, iabus, reset, cbus, dbus);
eout outop(clk, iabus["OPCODE"], reset, cbus, dbus);
ethen thenop(clk, iabus, reset, cbus, dbus);
evalue valuop(clk, iabus["OPCODE"], markbus, reset, cbus, dbus);
endmodule
/* simulation module follows: modify */
* events within this module as needed *
***************************************************************************
module testmachine;

reg clk;
wire error;
reg [47:0] inport;
wire [47:0] outport;
reg reset;

machine undertest(clk, error, inport, outport, reset);

initial
begin : stimuli
$monitor($time, "reset= %b, error= %b, outport= %h", reset, error, outport):

// initial values for simulation
// clock at "power off"
// input port at "switch settings for power on"
reset = 0;
clk = 0;
inport = 48'h050000000000008;

// exercise ports through time
#20 reset = 1;
#1000 inport = 48'h050000000000000a;
#30000 $finish;
end

always #5 clk = -clk;
endmodule

g* ******************************************************
g* adder.log
g* ******************************************************
g* an output file created by the Cadence Verilog
*g* simulator while simulating adder.v
*g* ******************************************************

Host command: verilog
Command arguments:
adder.v

--- some header lines removed to save space in this appendix ---

Compiling source file "adder.v"
Compiling included source file "adder.rom"
Continuing compilation of source file "adder.v"

Highest level modules:
testmachine

0reset= 0, error= x, outport= 0000000000000000
5reset= 0, error= z, outport= 0000000000000000
20reset= 1, error= 1, outport= 0000000000000000
25reset= 1, error= 1, outport= 0000000000000000
145reset= 1, error= 1, outport= 0000000000000000
155reset= 1, error= 1, outport= 0000000000000000
1770reset= 1, error= 1, outport= 0500000000010
3920reset= 1, error= 1, outport= 0500000000014

L1496 "adder.v": $finish at simulation time 31020

All results are displayed in hexadecimal. This demonstrates that
8+8 = 16 , and 10 + 10 = 20.

#* ******************************************************
#* fact
#* 
#* an Euler algorithm describing an iterative
#* factorial calculation based on the sampled input
#* ******************************************************

begin
new invalue;
new result;

label LOOP:

invalue := in;
result := 1;
LOOP: if invalue > 1 then
  begin
    result := result * invalue;
    invalue := invalue - 1;
    goto LOOP
  end
  else out result
end

#*  ****************************************************************************
#*  * fact.cmp
#*  ****************************************************************************
#*  * the translated opcodes and operands
#*  * created by the Euler-97 compiler according to the
#*  * translation rules given in the Wirth description
#*  * of Euler
#*  ****************************************************************************

SourceLine  Opcode  Operand1  Operand2
-----------  -------  ---------  ---------
    1      ebeg   0         0
    2       enew   0         0
    3       enew   0         0
    7      eref   1         1
    7       ein   0         0
    7      eassign   0         0
    7      esemi   0         0
    8      eref   2         1
    8      enumber   1         0
    8      eassign   0         0
    8      esemi   0         0
   10      ebegin   0         0
   10      ethen   38        0
   11      ebeg   0         0
   12      ebegin   0         0
   12      eeref   2         1
   12      eeref   2         1
   12      evalue   0         0
   12      evalue   1         1
   12      evalue   0         0
   12      emul   0         0
   12      eassign   0         0
   12      esemi   0         0
   13      eref   1         1
   13      eref   1         1
   13      evalue   0         0
   13      enumber   0         0
   13      esub   0         0
   13      eassign   0         0
   13      esemi   0         0
   14      elabel   12        1
   15      evalue   0         0
   15      egoto   0         0
   15      eend   0         0
   16      else     41        0
   16      ebegin   0         0
   18      evalue   0         0
   18      eout   0         0
   18      eend   0         0

#*  ****************************************************************************
#*  * fact.syn
#*  ****************************************************************************
#*  * the synthesis personality for the fact algorithm
#*  * created by the Euler-97 compiler
#*  ****************************************************************************

EULER Synthesis Information
-------------------------------
   5      eassign   4
   6      esemi   4
  10      egchan   1
  12      eref   9
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47  
48  
49  
50  
51  
52  
53  
54  
55  
56  
57  

---

Total Opcodes Required: 17
Total Opcode Calls: 41

```plaintext
module irom(abus, dbus);

input [7:0] abus;
output [39:0] dbus;
reg [39:0] values[0:255];
assign dbus = values[abus];

initial
  begin : compiledProgram
    values[1] = 40'h12_0000_0000;
    values[2] = 40'h2e_0000_0000;
    values[3] = 40'h2e_0000_0000;
    values[4] = 40'h0c_0001_0001;
    values[5] = 40'h1a_0000_0000;
    values[6] = 40'h05_0000_0000;
    values[7] = 40'h06_0000_0000;
    values[8] = 40'h0c_0002_0001;
    values[9] = 40'h30_0000_0000;
    values[10] = 40'h05_0000_0000;
    values[11] = 40'h06_0000_0000;
    values[12] = 40'h0c_0001_0001;
    values[13] = 40'h39_0000_0000;
    values[14] = 40'h30_0000_0000;
    values[15] = 40'h0a_0000_0000;
    values[16] = 40'h38_0000_0000;
    values[17] = 40'h12_0000_0000;
    values[18] = 40'h0c_0002_0001;
    values[19] = 40'h0c_0002_0001;
    values[20] = 40'h38_0000_0000;
    values[21] = 40'h0c_0001_0001;
    values[22] = 40'h39_0000_0000;
    values[23] = 40'h2c_0000_0000;
    values[24] = 40'h05_0000_0000;
    values[25] = 40'h06_0000_0000;
    values[26] = 40'h0c_0001_0001;
    values[27] = 40'h0c_0000_0000;
    values[28] = 40'h39_0000_0000;
    values[29] = 40'h30_0001_0000;
    values[30] = 40'h15_0000_0000;
    values[31] = 40'h05_0000_0000;
    values[32] = 40'h06_0000_0000;
    values[33] = 40'h24_0000_0001;
    values[34] = 40'h39_0000_0000;
    values[35] = 40'h19_0000_0000;
    values[36] = 40'h16_0000_0000;
    values[37] = 40'h15_0000_0000;
    values[38] = 40'h0c_0000_0000;
    values[39] = 40'h39_0000_0000;
    values[40] = 40'h33_0000_0000;
    values[41] = 40'h16_0000_0000;

end
```
```
values[42] = 40'hFF_0000_0000;
endmodule

/* fact.v: created by SAMUEL */
/* this file is ready for simulation, not synthesis */
/* to remove simulation components and prepare for */
/* synthesis, recompile using the -sim -synth options */

/* typeflags */
'define mark 8'b00000000
'define bool 8'b00000001
'define labl 8'b00000010
'define list 8'b00000011
'define undf 8'b00000100
'define numb 8'b00000101
'define proc 8'b00000110
'define vref 8'b00000111

/* data info */
'define DATAVALUE 31:0
'define TYPEFLAG 47:40

/* stack info */
'define STACKMAX 255
'define SPBITS 8

/* fields for readability */
'define BLOCK 47:40
'define DYNAMIC 39:32
'define STATIC 31:24
'define VAREND 23:16
'define RETURN 15:8
'define MP 7:0
'define UNDF 39:0
'define NUMBER 15:0
'define NUMBERLSB 7:0
'define LABEL0P1 31:16
'define LABEL0P2 15:0
'define LABELUNUSED 39:32

/* control bus info */
'define CBITS 17

/* instruction info */
'define OPCODEBITS 8
'define OPCODE 39:32
'define OPERAND1 31:16
'define OPERAND2 15:0

/* opcode encodings on ibus */
'define enequal 8'h00
'define erem 8'h01
'define econcat 8'h02
'define erquote 8'h03
'define eassign 8'h04
'define esemi 8'h05
'define eabs 8'h06
'define ebegin 8'h07
'define ecall 8'h08
'define eexp 8'h09
'define eadd 8'h0A
'define eand 8'h0B
'define elquote 8'h0C
'define ebegin 8'h0D
'define equote 8'h0E
'define eabs 8'h0F
'define eadd 8'h10
'define eand 8'h11
'define ebegin 8'h12
'define ecall 8'h13
'define ediv 8'h14
'define eelse 8'h15
'define eend 8'h16
'define eexp 8'h17
module mreg(clk, d, reset, sl, s0, q);
parameter RWIDTH = 48;
parameter RESETVALUE = 'b0;
input clk, reset, sl, s0;
input [RWIDTH-1:0] d;
output [RWIDTH-1:0] q;
reg [RWIDTH-1:0] qvalue;
assign q = qvalue;
always @(negedge clk)
begin : behavior
  if (reset == 0) qvalue = RESETVALUE;
  else
    case ((sl,s0))
      0: qvalue = qvalue-1;
      1: qvalue = qvalue + 1;
      2: qvalue = d;
    endcase
end
```
default: qvalue = qvalue;
endcase
endmodule

module miix41(in3, in2, in1, in0, outdata, sl, s0);
parameter datawidth = 8;

input [datawidth-1:0] in3;
input [datawidth-1:0] in2;
input [datawidth-1:0] in1;
input [datawidth-1:0] in0;
output [datawidth-1:0] outdata;
input sl;
input s0;

assign outdata = {sl,s0} == 2'b11 ? in3 :
                 {sl,s0} == 2'b10 ? in2 :
                 {sl,s0} == 2'b01 ? in1 :
                 {sl,s0} == 2'b00 ? in0 : in0;
endmodule

module tsbuf(d,dr,q);
parameter bwidth = 48;

input [bwidth-1:0] d;
input dr;
output [bwidth-1:0] q;

assign q = (dr == 0) ? d : 48'bz;
endmodule

module mem(abus,cs,dbus,we);
input [7:0] abus;
input cs;
inout [47:0] dbus;
input we;

reg [47:0] values[0:255];

assign dbus = (cs || (we == 0)) ? 4a'b2 : values[abus];

always @(negedge cs)
begin : writeMe
if (we==0)
begin
values[abus] = dbus;
//Sdisplay("cs=%b,we=%b,abus = %h, dbus=%h",cs,we,abus,dbus);
end
end

//always @(dbus) Sdisplay("cs=%b,we=%b,abus = %h, dbus=%h",cs,we,abus,dbus);

initial
begin : memInit
integer i;
for (i=0; i<256; i=i+1) values[i] = 4a'b0;
end
endmodule

#include "fact.rom"

module eassign(clk,opcode,markbus,reset,cbus,dbus);
input clk;
input [7:0] opcode;
input [47:0] markbus;
input reset;
output [16:0] cbus;
inout [47:0] dbus;

reg [3:0] statebits;
reg [16:0] nanoword;
reg [47:0] lvalue;
reg [47:0] rvalue;
reg [47:0] temp;

assign dbus = (statebits == 5)?{40'b0,lvalue[7:0]} :
            (statebits == 1) && statebits == 10)? rvalue :
            (statebits == 8)? temp : 48'bz;

assign dbus = (statebits == 0)?17'bz:nanoword;

always @(negedge clk)
begin : tempLatch
    case (statebits)
        1: rvalue = dbus;
        3: lvalue = dbus;
        4: temp = markbus;
        default: begin lvalue=lvalue; rvalue=rvalue; temp=temp; end
    endcase
end

always @(posedge clk)
begin : stateTransition
    case (statebits)
        0: if (reset == 0) statebits = 0;
            else if (opcode == "assign") statebits = 1;
            else statebits = 0;
        1: if (reset == 0) statebits = 0;
            else statebits = 2;
        2: if (reset == 0) statebits = 0;
            else statebits = 3;
        3: if (reset == 0) statebits = 0;
            else statebits = 4;
        4: if (reset == 0) statebits = 0;
            else if (lvalue["TYPEFLAG"]!='vref) statebits = 12;
            else statebits = 5;
        5: if (reset == 0) statebits = 0;
            else statebits = 6;
        6: if (reset == 0) statebits = 0;
            else statebits = 7;
        7: if (reset == 0) statebits = 0;
            else statebits = 8;
        8: if (reset == 0) statebits = 0;
            else statebits = 9;
        9: if (reset == 0) statebits = 0;
            else statebits = 10;
        10: if (reset == 0) statebits = 0;
            else statebits = 11;
        11: if (reset == 0) statebits = 0;
            else statebits = 12;
        default: statebits = 0;
    endcase
end

always @(statebits)
begin : combLogic
    case (statebits)
        0: nanoword = 17'hFFFFF;
        1: nanoword = 17'hFFDF;
        2: nanoword = 17'hF0FF;
        3: nanoword = 17'hFFFF;
        4: nanoword = 17'hF0FD;
        5: nanoword = 17'hFFFFB;
        6: nanoword = 17'hF0FE;
        7: nanoword = 17'hFF0F;
        8: nanoword = 17'hFFEB;
        9: nanoword = 17'hFFFFB;
        10: nanoword = 17'hFFFF;
        11: nanoword = 17'hFFFF;
        12: nanoword = 17'hFFFFF;
        default: nanoword = 17'hFFFFF;
    endcase
end
```verilog
module esemi(clk, opcode, reset, cbus);
    input clk;
    input [7:0] opcode;
    input reset;
    output [16:0] cbus;
    reg [1:0] statebits;
    reg [16:0] nanoword;
    assign cbus = (statebits == 0)?17'b1FFFF : nanoword;
    always @(posedge clk)
    begin : stateTransition
        case (statebits)
            0: if (reset == 0) statebits = 0;
                else if (opcode == 'esemi) statebits = 1;
                else statebits = 0;
            1: if (reset == 0) statebits = 0;
                else statebits = 1;
            2: if (reset == 0) statebits = 0;
                else statebits = 2;
            default: statebits = 0;
        endcase
    end
    always @(statebits)
    begin : combLogic
        case (statebits)
            0: nanoword = 17'hlFFFF;
            1: nanoword = 17'hlFCFF;
            2: nanoword = 17'hl7FFF;
            default: nanoword = 17'hlFFFF;
        endcase
    end
endmodule

module egthan(clk, opcode, reset, cbus, dbus);
    input clk;
    input [7:0] opcode;
    input reset;
    output [16:0] cbus;
    inout [47:0] dbus;
    reg [2:0] statebits;
    reg [16:0] nanoword;
    reg [47:0] number1;
    reg [47:0] number2;
    assign dbus["TYPEFLAG"] = (statebits == 5)?8'b1 : 8'b0;
    assign dbus[39:16] = (statebits == 5)?24'b0:24'b1;
    assign dbus["NUMBER"] = (statebits == 5)?number1["NUMBER">number2["NUMBER]:16'b1;
    assign cbus = (statebits == 0)?17'b1FFFF : nanoword;
    always @(negedge clk)
    begin : tempLatch
        case (statebits)
            1: number2 = dbus;
            3: number1 = dbus;
            default: begin number1 = number1; number2 = number2; end
        endcase
    end
    always @(posedge clk)
    begin : stateTransition
        case (statebits)
            0: if (reset == 0) statebits = 0;
                else if (opcode == 'egthan) statebits = 1;
                else statebits = 0;
            1: if (reset == 0) statebits = 0;
                else statebits = 2;
            2: if (reset == 0) statebits = 0;
                else statebits = 3;
            3: if (reset == 0) statebits = 0;
        endcase
    end
endmodule
```
else statebits = 4;
4: if (reset == 0) statebits = 0;
else if (number1["TYPEFLAG"]=='numb && number2["TYPEFLAG"]=='numb)
   statebits = 5;
else statebits = 7;
5: if (reset == 0) statebits = 0;
else statebits = 6;
6: if (reset == 0) statebits = 0;
else statebits = 0;
7: if (reset == 0) statebits = 0;
default: statebits = 0;
endcase
end
always @(statebits)
begin : combLogic
   case (statebits)
      0: nanoword = 17'h1FFFF;
      1: nanoword = 17'h1FFDF;
      2: nanoword = 17'h1FCFF;
      3: nanoword = 17'h1FFDF;
      4: nanoword = 17'h1FFE;
      5: nanoword = 17'h1FFCF;
      6: nanoword = 17'h17FFF;
      7: nanoword = 17'h0FFFF;
default: nanoword = 17'h1FFFF;
endcase
end
endmodule

module ere£(clk,ibus,markbus,reset,cbus,dbus);
   input clk;
   input [39:0] ibus;
   input [47:0] markbus;
   input reset;
   output [16:0] cbus;
   output [47:0] dbus;
   reg [3:0] statebits;
   reg [16:0] nanoword;
   reg [47:0] temp1;
   reg [47:0] temp2;

   assign dbus["TYPEFLAG"] = (statebits == 6)?temp2["TYPEFLAG"]: (statebits == 7)?temp1["TYPEFLAG": (statebits == 10)?'vref : 3'bz;
   assign dbus[39:8] = (statebits == 6)?temp2[39:8]: (statebits == 7)?temp1[39:8]: (statebits == 10)?32'b0 : 32'bz;
   assign dbus[7:0] = (statebits == 6)?temp2[7:0]: (statebits == 7)?temp1[7:0]: (statebits == 10)?temp2[7:0]-ibus["OPERAND1"] : 8'bz;

   assign cbus = (statebits == 0)?17'b0:nanoword;
always @(negedge clk)
begin : tempLatch
   case (statebits)
      1: begin temp1 = markbus; temp2 = markbus; end
      5: temp2 = dbus;
      default: begin temp1=temp1; temp2=temp2; end
   endcase
end
always @(posedge clk)
begin : stateTransition
   case (statebits)
      0: if (reset == 0) statebits = 0;
      else if (ibus["OPCODE"] == 'eref) statebits = 1;
      else statebits = 0;
      1: if (reset == 0) statebits = 0;
      else statebits = 2;
      2: if (reset == 0) statebits = 0;
```verilog
always @((dbus) Sdisplay("eref statebits = %h dbus=%h markreg=%h", statebits, dbus, markbus));
endmodule
ebeginicik, markbus, opcode, reset, spbus, cbus, dbus)
input die;
input [47:0] markbus;
input [7:0] opcode;
input reset;
input [7:0] spbus;
output [16:0] cbus;
output [47:0] dbus;
reg [2:0] statebits;
reg [16:0] nanoword;
assign dbus[‘TYPEFLAG] = (statebits==3 || statebits==4) ?markbus[‘BLOCK]+8‘bl:8‘bz;
assign dbus[‘DYNAMIC] = (statebits==3 || statebits==4) ?markbus[‘MP]+8‘bz;
assign dbus[‘VAREND] = (statebits==3) ?spbus:8‘bz;
assign dbus[‘RETURN] = (statebits==3) ?spbus:8‘bz;
assign dbus[‘MP] = (statebits==3 || statebits==4) ?spbus:8‘bz;
assign cbus = (statebits == 0)?17’h0:3’h0;
always @(posedge clk)
begin : stateTransistion
case (statebits)
0: if (reset == 0) statebits = 0;
else if (opcode == 'ebegin) statebits = 1;
else statebits = 0;
1: if (reset == 0) statebits = 0;
```
else statebits = 2;
2: if (reset == 0) statebits = 0;
else statebits = 3;
3: if (reset == 0) statebits = 0;
   else statebits = 4;
4: if (reset == 0) statebits = 0;
   else statebits = 5;
5: if (reset == 0) statebits = 0;
   else statebits = 0;
default: statebits = 0;
endcase
end
always @(statebits)
begin : combLogic
   case (statebits)
      0: nanoword = 17'h1FFFFF;
      1: nanoword = 17'hlFDFF;
      2: nanoword = 17'h1FFEF;
      3: nanoword = 17'h1FFCF;
      4: nanoword = 17'h1FFFB;
      5: nanoword = 17'h1FFFFF;
      default: nanoword = 17'h1FFFFF;
   endcase
end
endmodule
d宗ule eelse(clk,ibus,reset,cbus,dbus);
   input clk;
   input [39:0] ibus;
   input reset;
   output [16:0] cbus;
   output [47:0] dbus;
   reg [1:0] statebits;
   reg [16:0] nanoword;
assign dbus = (statebits == 1)?(32'b0.ibus["OPERAND1]):48'bz;
assign cbus = (statebits == 0)?17'bz:nanoword;
always @(posedge clk)
begin : stateTransistion
   case (statebits)
      0: if (reset == 0) statebits = 0;
          else if (ibus["OPCODE"] == "eelse") statebits = 1;
          else statebits = 0;
      1: if (reset == 0) statebits = 0;
          else statebits = 0;
      default: statebits = 0;
   endcase
end
always @(statebits)
begin : combLogic
   case (statebits)
      0: nanoword = 17'h1FFFFF;
      1: nanoword = 17'h1FFFFF;
      default: nanoword = 17'h1FFFFF;
   endcase
end
endmodule
d宗ule eend(clk,opcode,reset,cbus,dbus);
   input clk;
   input [7:0] opcode;
   input reset;
   output [16:0] cbus;
   inout [47:0] dbus;
   reg [3:0] statebits;
   reg [16:0] nanoword;
   reg [47:0] temp;
assign dbus = (statebits == 4) || (statebits == 7) ? temp : 48'bz;
assign cbus = (statebits == 0)?17'bz:nanoword;
always @(negedge elk)
begin : tempLatch
  case (statebits)
  1: temp = dbus;
  6: temp = dbus;
  default: temp = temp;
  endcase
end

always @ (posedge elk)
begin : stateTransition
  case (statebits)
  0: if (reset == 0) statebits = 0;
    else if (opcode == 'end) statebits = 1;
    else statebits = 0;
  1: if (reset == 0) statebits = 0;
    else statebits = 2;
  2: if (reset == 0) statebits = 0;
    else statebits = 4;
  3: if (reset == 0) statebits = 0;
    else statebits = 6;
  4: if (reset == 0) statebits = 0;
    else statebits = 5;
  5: if (reset == 0) statebits = 0;
    else statebits = 4;
  6: if (reset == 0) statebits = 0;
    else statebits = 3;
  7: if (reset == 0) statebits = 0;
    else statebits = 2;
  8: if (reset == 0) statebits = 0;
    else statebits = 0;
  default: statebits = 0;
  endcase
end

always @ (statebits)
begin : combLogic
  case (statebits)
  0: nanoword = 17’h1FFFFF;
  1: nanoword = 17’h1FFDF;
  2: nanoword = 17’h1FEFF;
  3: nanoword = 17’h1FEPF;
  4: nanoword = 17’h1FFCF;
  5: nanoword = 17’h1FFBF;
  6: nanoword = 17’h1FF9F;
  7: nanoword = 17’h1FFFB;
  8: nanoword = 17’h17FFFF;
  default: nanoword = 17’h1FFFF;
  endcase
end
endmodule
module ein(elk,opcode,reset,cbus);
input elk;
input [7:0] opcode;
input reset;
output [16:0] cbus;
reg [2:0] statebits;
reg [16:0] nanoword;

assign cbus = (statebits == 0)?17'hFFFF:nanoword;

always @(posedge clk)
begin : stateTransition
    case (statebits)
        0: if (reset == 0) statebits = 0;
           else if (opcode == 'ein) statebits = 1;
           else statebits = 0;
        1: if (reset == 0) statebits = 0;
           else statebits = 2;
        2: if (reset == 0) statebits = 0;
           else statebits = 3;
        3: if (reset == 0) statebits = 0;
           else if (label[TYPEFLAG]'label) statebits = 10;
           else statebits = 4;
        4: if (reset == 0) statebits = 0;
           else statebits = 5;
        5: if (reset == 0) statebits = 0;
           else statebits = 6;
        6: if (reset == 0) statebits = 0;
           else statebits = 7;
        7: if (reset == 0) statebits = 0;
           else statebits = 8;
        8: if (reset == 0) statebits = 0;
           else statebits = 9;
        9: if (reset == 0) statebits = 0;
           else statebits = 10;
        10: if (reset == 0) statebits = 0;
            else statebits = 10;
        default: statebits = 0;
    endcase
end
endmodule
# Spaceship Engine

1. if (reset == 0) statebits = 0;
   else statebits = 2;
2. if (reset == 0) statebits = 0;
   else statebits = 3;
3. if (reset == 0) statebits = 0;
   else statebits = 4;
4. if (reset == 0) statebits = 0;
   else statebits = 0;
   default: statebits = 0;
endcase
end

always @(statebits)
begin : combLogic
    case (statebits)
    0: nanoword = 17'hFFFF;
    1: nanoword = 17'hFFFD;
    2: nanoword = 17'hFFFE;
    3: nanoword = 17'hFFCF;
    4: nanoword = 17'h7FFF;
    default: nanoword = 17'hFFFF;
endcase
end
endmodule

module emul(clk, opcode, reset, cbus, dbus);
    input clk;
    input [7:0] opcode;
endmodule
module enew(clk, markbus, opcode, reset, cbus, dbus);

input clk;
input [47:0] markbus;
input [7:0] opcode;
input reset;
output [16:0] cbus;
output [47:0] dbus;

reg [2:0] statebits;
reg [47:0] temp;

always @(negedge clk)
begin : tempLatch
  case (statebits)
    1: number2 = dbus;
    3: number1 = dbus;
    default: begin number1 = number1; number2 = number2; end
  endcase
end

always @(posedge clk)
begin : stateTransition
  case (statebits)
    0: if (reset == 0) statebits = 0;
      else if (opcode == "emuD") statebits = 1;
      else statebits = 0;
    1: if (reset == 0) statebits = 0;
      else statebits = 3;
    2: if (reset == 0) statebits = 0;
      else statebits = 3;
    3: if (reset == 0) statebits = 0;
      else statebits = 4;
    4: if (reset == 0) statebits = 0;
      else if (number1[TYPEFLAG] == 'numb && number2[TYPEFLAG] == 'numb)
        statebits = 5;
      else statebits = 7;
    5: if (reset == 0) statebits = 0;
      else statebits = 6;
    6: if (reset == 0) statebits = 0;
      else statebits = 6;
    7: if (reset == 0) statebits = 0;
      else statebits = 0;
    default: statebits = 0;
  endcase
end

always @statebits
begin : combLogic
  case (statebits)
    0: nanoword = 17'h1FFFF;
    1: nanoword = 17'h1FFDF;
    2: nanoword = 17'h1FFFF;
    3: nanoword = 17'h1FFDF;
    4: nanoword = 17'h1FFFF;
    5: nanoword = 17'h1FFDF;
    6: nanoword = 17'h1FFFF;
    7: nanoword = 17'h0FFFF;
    default: nanoword = 17'h1FFFF;
  endcase
end
endmodule
reg [16:0] nanoword;
assign dbus = (statebits == 3)?"undf,40'b0":
(statebits == 4 || statebits == 6)?48'bz;
assign cbus = (statebits == 0)?17'bz:nanoword;

always @(negedge elk)
begin : latchTemp
  case (statebits)
    1: temp = markbus;
    2: begin
      temp['BLOCK] = temp['BLOCK];
      temp['DYNAMIC] = temp['DYNAMIC];
      temp['STATIC] = temp['STATIC];
      temp['VAREND] = temp['VAREND] - 1;
      temp['RETURN] = temp['RETURN];
      temp['MP] = temp['MP];
    end
    default: temp = temp;
  endcase
end

always @(posedge clk)
begin : stateTransiscion
  case (statebits)
    0: if (reset == 0) statebits = 0;
       else if (opcode == 'enew) statebits = 1;
       else statebits = 0;
    1: if (reset == 0) statebits = 0;
       else statebits = 2;
    2: if (reset == 0) statebits = 0;
       else statebits = 3;
    3: if (reset == 0) statebits = 0;
       else statebits = 4;
    4: if (reset == 0) statebits = 0;
       else statebits = 5;
    5: if (reset == 0) statebits = 0;
       else statebits = 6;
    6: if (reset == 0) statebits = 0;
       else statebits = 7;
    7: if (reset == 0) statebits = 0;
       else statebits = 0;
    default: statebits = 0;
  endcase
end

always @(statebits)
begin : combLogic
  case (statebits)
    0: nanoword = 17'hlFFFF;
    1: nanoword = 17'hlFDFF;
    2: nanoword = 17'hlFFEE;
    3: nanoword = 17'hlFFFB;
    4: nanoword = 17'hlFF2F;
    5: nanoword = 17'hlFF0F;
    6: nanoword = 17'hlFFFF;
    default: nanoword = 17'hlFFFF;
  endcase
end

module enumber(clk,ibus,reset,cbus,dbus);
  input clk;
  input [39:0] ibus;
  input reset;
  output [16:0] cbus;
  output [47:0] dbus;

  reg [2:0] statebits;
  reg [16:0] nanoword;

  assign dbus['TYPEFLAG] = (statebits==3)?'numb:8'bz;
  assign dbus[39:16] = (statebits==3)?24'b0:24'bz;
endmodule
assign dbus[\"NUMBER\] = (statebits==1)?ibus[\"OPERAAND\]:16'bz;
assign cbus = (statebits == 0)?17'bz:nanoword;

always @(posedge clk)
begin : stateTransistion
    case (statebits)
        0: if (reset == 0) statebits = 0;
           else if (ibus[\"OPCODE\] == \"enumfaerl statebits = 1;
                    else statebits = 0;
        1: if (reset == 0) statebits = 0;
           else statebits = 2;
        2: if (reset == 0) statebits = 0;
           else statebits = 3;
        3: if (reset == 0) statebits = 0;
           else statebits = 4;
        4: if (reset == 0) statebits = 0;
           else statebits = 0;
        default: statebits = 0;
    endcase
end

always @(statebits)
begin : combLogic
    case (statebits)
        0: nanoword = 17'hlFFFF;
        1: nanoword = 17'hlFFDF;
        2: nanoword = 17'hlFFE7;
        3: nanoword = 17'hlFFCF;
        4: nanoword = 17'hl7FFF;
        default: nanoword = 17'hlFFFF;
    endcase
end
endmodule

module eout(clk, opcode, reset, cbus);
input clk;
input [7:0] opcode;
input reset;
output [16:0] cbus;
reg [1:0] statebits;
reg [16:0] nanoword;

assign cbus = (statebits == 0)?17'bz:nanoword;

always @(posedge clk)
begin : stateTransistion
    case (statebits)
        0: if (reset == 0) statebits = 0;
           else if (opcode == \"eout\) statebits = 1;
                    else statebits = 0;
        1: if (reset == 0) statebits = 0;
           else statebits = 2;
        2: if (reset == 0) statebits = 0;
           else statebits = 0;
        3: if (reset == 0) statebits = 0;
           else statebits = 4;
        default: statebits = 0;
    endcase
end

always @(statebits)
begin : combLogic
    case (statebits)
        0: nanoword = 17'hlFFFF;
        1: nanoword = 17'hlFFDE;
        2: nanoword = 17'hl7FFF;
        default: nanoword = 17'hlFFFF;
    endcase
end
endmodule

module esub(clk, opcode, reset, cbus, dbus);
input clk;
input [7:0] opcode;
input reset;
output [16:0] cbus;
inout [47:0] dbus;
module echen(clk,ibus,reset,cbus,dbus);
  input clk;
  input [39:0] ibus;
  input reset;
  output [16:0] cbus;
  input [47:0] dbus;

  reg [2:0] statebits;
  reg [16:0] nanoword;
  reg [47:0] number1;
  reg [47:0] number2;

  assign dbus[47:16] = (statebits == 5)?32'b0:32'bz;
  assign dbus[15:0] = (statebits == 5)?ibus[OPERAND1]:16'bz;

always @(negedge clk)
  begin : tempLatch
    case (statebits)
      1: number2 = dbus;
      3: number1 = dbus;
      default: begin number1 = number1; number2 = number2; end
    endcase
  end

always @(posedge clk)
  begin : stateTransition
    case (statebits)
      0: if (reset == 0) statebits = 0;
         else if (opcode == 'esub) statebits = 1;
         else statebits = 0;
      1: if (reset == 0) statebits = 0;
         else statebits = 2;
      2: if (reset == 0) statebits = 0;
         else statebits = 3;
      3: if (reset == 0) statebits = 0;
         else statebits = 4;
      4: if (reset == 0) statebits = 0;
         else if (number1[TYPEFLAG1 == numb & number2[TYPEFLAG1 == numb)
          statebits = 5;
         else statebits = 7;
      5: if (reset == 0) statebits = 0;
         else statebits = 6;
      6: if (reset == 0) statebits = 0;
         else statebits = 0;
      7: if (reset == 0) statebits = 0;
         else statebits = 7;
      default: statebits = 0;
    endcase
  end

always @(statebits)
  begin : combLogic
    case (statebits)
      0: nanoword = 17'h1FFFF;
      1: nanoword = 17'h1FFFDF;
      2: nanoword = 17'h1FCFF;
      3: nanoword = 17'h1FFDF;
      4: nanoword = 17'h1FFFF;
      5: nanoword = 17'h1FCFF;
      6: nanoword = 17'h1FFFF;
      7: nanoword = 17'h0FFFF;
      default: nanoword = 17'h1FFFF;
    endcase
  end
endmodule
module eva1ue(clk, opcode, markbus, reset, cbus, dbus);

assign cbus = (statebits == 0) ? 17'b0 : nanoword;

always @(negedge clk)
begin :
begin : tempLatch
case (statebits)
1: temp = dbus;
default: begin temp = temp; end
endcase
end

always @(posedge clk)
begin :
begin : stateTransiscion
case (statebits)
0: if (reset == 0) statebits = 0;
   else if (ibus[OPCODE] == 'ethen) statebits = 1;
   else statebits = 0;
1: if (reset == 0) statebits = 0;
   else statebits = 2;
2: if (reset == 0) statebits = 0;
   else if (temp[TYPEFLAG] == 'bool) statebits = 6;
   else statebits = 4;
3: if (reset == 0) statebits = 0;
   else if (temp[15:0] == 16'b0) statebits = 5;
   else statebits = 4;
4: if (reset == 0) statebits = 0;
   else statebits = 6;
5: if (reset == 0) statebits = 0;
   else statebits = 6;
6: if (reset == 0) statebits = 0;
   else statebits = 6;
default: statebits = 0;
endcase
end
always @(statebits)
begin :
begin : combLogic
case (statebits)
0: nanoword = 17'h1FFFF;
1: nanoword = 17'h1FFDF;
2: nanoword = 17'h1FFFF;
3: nanoword = 17'h1FFFF;
4: nanoword = 17'h1FFFF;
5: nanoword = 17'h1FFFF;
6: nanoword = 17'h1FFFF;
default: nanoword = 17'h1FFFF;
endcase
end
endmodule
end
always @(posedge clk)
begin : stateTransition
    case (statebits)
        0: if (reset == 0) statebits = 0;
           else if (opcode == 'evalue) statebits = 1;
                else statebits = 0;
        1: if (reset == 0) statebits = 0;
                else statebits = 2;
        2: if (reset == 0) statebits = 0;
                else if (templ['TYPEFLAG]!'vref) statebits = 10;
                        else statebits = 3;
        3: if (reset == 0) statebits = 0;
                else statebits = 4;
        4: if (reset == 0) statebits = 0;
                else statebits = 5;
        5: if (reset == 0) statebits = 0;
                else statebits = 6;
        6: if (reset == 0) statebits = 0;
                else statebits = 7;
        7: if (reset == 0) statebits = 0;
                else statebits = 8;
        8: if (reset == 0) statebits = 0;
                else statebits = 9;
        9: if (reset == 0) statebits = 0;
                else if (reset == 0) statebits = 10;
                        else if (templ['TYPEFLAG]!'vref) statebits = 12;
                        else statebits = 11;
        10: if (reset == 0) statebits = 0;
            else if (templ['TYPEFLAG]!'vref) statebits = 12;
                        else statebits = 11;
        11: if (reset == 0) statebits = 0;
                else statebits = 0;
        12: if (reset == 0) statebits = 0;
            else statebits = 12;
        default: statebits = 0;
    endcase
end
always @(statebits)
begin : combLogic
    case (statebits)
        0: nanoword = 17'hFFFFF;
        1: nanoword = 17'hFFFFDF;
        2: nanoword = 17'hFFFFEF;
        3: nanoword = 17'hFFFFFFFF;
        4: nanoword = 17'hFFFFFB;
        5: nanoword = 17'hFFFFE5;
        6: nanoword = 17'hFFFFF1;
        7: nanoword = 17'hFFFFFB;
        8: nanoword = 17'hFFFFEF;
        9: nanoword = 17'hFFFFF5;
        10: nanoword = 17'hFFFFFFFF;
        11: nanoword = 17'hFFFFFB;
        12: nanoword = 17'hFFFFEF;
        default: nanoword = 17'hFFFFF;
    endcase
end
module machine(clk, error, import, export, reset);
    input clk;
    output error;
    input[47:0] import;
    output [47:0] export;
    input reset;
    wire [47:0] dbus;
    wire [7:0] spbus;     /* busses */
    wire [7:0] sbus;
    wire [7:0] iabusbus;
    wire [47:0] markbus;
    wire [39:0] ibus;
    wire [16:0] cbus;
    wire [47:0] inputbus;
    /* instantiate members of R from M = (R,R) */
mreg inputreg(clk, inpor, reset, 1'b1, 1'b0, inputbus);
tabuf inputbuf(inputbus, cbus['dr1], dbus);
mreg outputreg(clk, dbus, reset, 1'b1, cbus['ldor], outport);
mreg #8 sp(clk, markbus['VAREND], reset, cbus['s1sp], cbus['s0sp], spbus);
mreg markeq(clk, dbus, reset, cbus['s1mp], cbus['s0mp], markbus);
mreg #(8,8'b00000000) iadr(clk, dbus[7:0], reset, cbus['s1iap], cbus['s0iap], iabusebus);
mux41 #(8) smux(spbus, markbus['DYNAMIC], markbus['STATIC], markbus['MP], sabus, cbus['smux1], cbus['smux0]);
mem stack(sabus, cbus['scs], dbus, cbus['sw1]);
rom imemory(iabus, ibus);
assign error = cbus['error];
/* instantiate members of 0 from M = {R, O} */
assign assignop(clk, ibus['OPCODE], markbus, reset, cbus, dbus);
esemi semio(clk, ibus['OPCODE], reset, cbus, dbus);
egthan gethanop(clk, ibus['OPCODE], reset, cbus, dbus);
eref refop(clk, ibus, markbus, reset, cbus, dbus);
begin beginop(clk, markbus, ibus['OPCODE], reset, spbus, cbus, dbus);
else elseop(clk, ibus, reset, cbus, dbus);
end endop(clk, ibus['OPCODE], reset, cbus, dbus);
eggoto gotoop(clk, ibus['OPCODE'], reset, cbus, dbus);
ein inop(clk, ibus['OPCODE'], reset, cbus, dbus);
elabel labelop(clk, ibus['OPCODE'], reset, cbus, dbus);
emul mulop(clk, ibus['OPCODE'], reset, cbus, dbus);
enew newop(clk, markbus, ibus['OPCODE'], reset, spbus, cbus, dbus);
eout outop(clk, ibus['OPCODE'], reset, cbus, dbus);
esub subop(clk, ibus['OPCODE'], reset, cbus, dbus);
ethen thenop(clk, ibus, reset, cbus, dbus);
evalue evalop(clk, ibus['OPCODE'], markbus, reset, cbus, dbus);
endmodule
/* ******************************************
 * simulation module follows: modify *
 * events within this module as needed. *
 ****************************************** */
module testmachine;
reg clk;
wire error;
reg [47:0] inport;
wire [47:0] outport;
reg reset;

machine undertest(clk, error, inport, outport, reset);

initial
begin : stimuli
$monitor($time, "reset= %b, error= %b, outport= %h", reset, error, outport);

// initial values for simulation
// clock at "power off"
// input port at "switch settings for power on"
reset = 0;
clk = 0;
inport = 48'h050000000007;
// exercise ports through time
#20 reset = 1;
#30000 $finish;
end

always #5 clk = ~clk;
endmodule

#******************************************************************************
# * fact.log
# * an output file created by the Cadence Verilog simulator while simulating fact.v
#******************************************************************************
Host command: verilog
Command arguments:
  fact.v

--- some header lines removed to save space in this appendix ---

Compiling source file "fact.v"
Compiling included source file "fact.rom"
Continuing compilation of source file "fact.v"

Highest level modules:
  testmachine

  0reset= 0, error= x, outport= 000000000000
  12670reset= 1, error= 1, outport= 05000000001b0
  127S5reset= 1, error= z, outport= 0500000013b0

L1348 "fact.v": $finish at simulation time 30020

#* ** poly
#* * an Euler algorithm describing a polynomial
#* *

begin
  new a;
  new b;
  new c;
  new d;

  new x;
  a:=3;
  b:=4;
  c:=5;
  d:=1;

  x:=in;
  out a*x*x*x - b*x*x - c*x - d
end

#* ** poly.sym
#* *
#* * the synthesis personality for the poly algorithm
#* * created by the Euler-97 compiler
#* *

EULER Synthesis Information
-----------------------------
5  eassign
  6  esemi
12  eref
16  eadd
18  ebeg
22  eend
26  ein
44  emul
46  enew
48  enumber
51  eout
57  evalue

Total Opcodes Required: 12
Total Opcode Calls: 57

#* ** poly2
#* *
#* * a different polynomial algorithm --- higher degree
#* *

begin
  new a;
  new b;
  new c;
  new d;
new x;
    a:=3;
b:=4;
c:=5;
    d:=1;
x:=in;
out a'x*x*x - b'x*x - c*x - d end

#* * poly2.syn
#* * the synthesis personality for the poly2 algorithm
#* * created by the Euler-97 compiler
#* *****************************************************
EULER Synthesis Information
----------------------------------
    5  eassign  5
    6  esemi  5
    12  eref  17
    16  eadd  3
    18  ebegin  1
    22  end  1
    26  ein  1
    44  emul  8
    46  enew  5
    48  enumber  4
    51  eout  1
    57  evalue  12
----------------------------------
Total Opcodes Required: 12
Total Opcode Calls: 53

#* * poly3.syn
#* * a different polynomial algorithm --- coefficient
#* * not stored in variables
#* *****************************************************
begin
    new x;
x:=in;
out 3'x*x*x - 4'x*x - 5*x - 1 end

#* *****************************************************
#* poly3.syn
#* * the synthesis personality for the poly3 algorithm
#* * created by the Euler-97 compiler
#* *****************************************************
EULER Synthesis Information
----------------------------------
    5  eassign  1
    6  esemi  1
    12  eref  7
    16  eadd  3
    18  ebegin  1
    22  end  1
    26  ein  1
    44  emul  6
    46  enew  1
    48  enumber  4
    51  eout  1
    57  evalue  6
----------------------------------
Total Opcodes Required: 12
Total Opcode Calls: 33
This data path was generated by a commercial synthesis tool called Synplify for the polynomial algorithm of chapter 5. Note that the input port, output port, and Verilog opcodes are shown. The other ports, such as the stack bus, and instruction bus are not shown due to space limitations.
This circuit was synthesized by the commercial tool called Synplify for the `ebegin` opcode.
This data path was generated by a commercial synthesis tool called Synplify for the factorial algorithm of chapter 5. Note that the input port, output port, and Verilog opcodes are shown. The other ports, such as the stack bus, and instruction bus are not shown due to space limitations.
REFERENCES

1. Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines

2. Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines

3. Proceedings of the Third Canadian Workshop on Field Programmable Devices:
   Technology, Tools, and Applications Polytechnical University of Montréal and the
   University of Québec at Montréal: Montréal (Québec) Canada, 1995

   PWS Publishing Company: Boston, MA, 1997

5. Skahill, K. VHDL for Programmable Logic


7. Milutinovic, V. Tutorial on Advanced Microprocessors and High-Level Language
   Computer Architecture

8. Ward, S.; Halstead, R. Computation Structures

9. Pratt, T.; Zelkowitz, M. Programming Languages: Design and Implementation (3rd
   Edition)


12. Reported Errata

13. Galloway, D. The Transmogrifier C Hardware Description Language and Compiler
    for FPGAs
    Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines


