# BEHAVIORAL-LEVEL SIMULATION AND SYNTHESIS OF DIGITAL SYSTEMS

by

J. T. Deutsch

Memorandum No. UCB?ERL M83/47

1 August 1983

# BEHAVIORAL-LEVEL SIMULATION AND SYNTHESIS OF DIGITAL SYSTEMS

by

J. T. Deutsch

Memorandum No. UCB/ERL M83/47

1 August 1983

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Jeffrey T. Deutsch
Author

" Behavioral--Level Simulation and
Synthesis of Digital Systems"

Title

RESEARCH PROJECT

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, to partial satisfaction of the requirements for the degree of Master of Sciences, Plan II.

Approval for the Report and Comprehensive Examination:

Committee: Richard Newton, Research Adviser

11/30/82 Date

[illegible], Second Reader

1 December 1982, Date

## ACKNOWLEDGEMENTS

The author would like to thank Professor A. R. Newton for the support, encouragement, and guidance he has given during this research and in all the time he has been at Berkeley.

Discussions with the members of the IC CAD research group at the University of California at Berkeley, especially J. Kleckner, K. Keller, M. Hofmann, P. Moore, M. Bales, and T. Quarles, have been extremely useful, and many of their insights have found their way into this work.

Financial support from the Army Research Office under grant (ARO) DAAG29-81-K-0021 and from Hewlett-Packard is gratefully acknowledged.

He would like to thank Dr. H.K. Gummel of Bell Laboratories for his first introduction to CAD, and for showing how exciting it can be.

Finally, he would like to thank his parents, Nathan and Gertrude, for their support and encouragement throughout his education.

TABLE OF CONTENTS

68000, Zilog Z8000, and Intel iapx 432) commonly takes between 50 and 200 person-months [Patterson81a].

According to a recent survey of the state of the art in design automation [Breuer82], even in companies with state of the art design tools, logic design takes more than 50 percent of the total design effort in digital system design. Few tools are available to aid the logic design part of the design process and often little effort is put into specifying the functional characteristics of a system and then verifying that an implementation meets the specified characteristics. Instead, the correctness of an implementation is determined by comparing it to a breadboard prototype or to earlier versions of the same machine.

The goal of the research described in this report is to decrease the time required to design large digital systems at the functional level, and to increase the designer's ability to verify that his system will function properly.

The functions-to-logic (FTL) system aids the designer in the early stages of system design – before a detailed implementation has been planned. FTL differs from other behavioral-level simulators such as ISP [Barbacci77], DDL-P [Duley68] and ADLIB/SABLE [Hill80], in that it provides *concurrent control structures*, that help the user clearly specify the behavior of their digital systems in the same way that sequential control structures aid software engineers in writing clearer, more reliable programs. FTL differs from "silicon compiler" [Johannsen79] systems in that its emphasis is on specification of the *architectural behavior* of a system rather than the *structure* of an implementation of it.

This report is divided into six chapters. The second chapter introduces several methods for specifying concurrent systems and presents techniques for automatic generation of module-level designs for digital systems. The third chapter describes data-flow analysis and data-flow based synthesis. The fourth

# CHAPTER 1

# INTRODUCTION

## 1.1. Computer Aided Design

Computer aids have been used with great success in several stages of the IC design process. Circuit simulators, such as SPICE [Nagel75], make it possible for chip designers to evaluate the detailed electrical characteristics of their circuits prior to their fabrication. Interactive graphics editors such as KIC [Keller82] and CAESAR, [Ousterhout81] and commercial products from Calma [Calma], Applicon [Applicon], and others decrease the time needed to enter mask data. Systems for automatic placement and routing of standard cells and gate arrays such as LTX [Deutsch76] make it possible to go from a completed logic design to mask layouts for semi-custom chips quickly and easily.

As advances in processing have made it possible to put more and more circuitry on a chip, CAD software designers have developed more sophisticated techniques to keep pace with the increasing complexity. In the early 1970s designs were small enough that entire chips could be simulated at the circuit level. As the complexity of IC's has increased, simulating complete circuits at the electrical level has become impractical. Instead, designers have turned to higher levels of simulation, such as timing [Chawla75], logic [Case75], and functional level simulation [Barbacci77], and to mixed-mode simulators that allow different sections of a circuit to be simulated at different levels of detail [Newton78]. [De Man81 ]. Although mixed-mode simulation has made it possible to simulate large digital systems in reasonable amounts of time, and standard-cells and gate-arrays allow layouts to be produced in almost no time at all, CAD has done relatively little to ease the task of turning initial ideas into logic designs. The logic design of large contemporary chips (such as the Motorola chapter describes the architecture of the FTL simulator and its use in some small examples. The fifth chapter shows how to describe digital systems using FTL, then introduces the RISC microprocessor and gives the FTL description of it. The sixth chapter contains a summary of this report and directions for future work.

# CHAPTER 2

# SIMULATION AND SYNTHESIS

## 2.1. Introduction

Two techniques for automatic synthesis of digital systems will be described in this chapter. They are called *control-flow based synthesis* and *data-flow based synthesis*. The two approaches differ in the way they derive sequencing information from the description of a system. Control-flow based synthesis systems use specification languages which require that the concurrency of the design is expressed explicitly in the input specification. Data-flow based synthesis systems derive sequencing information from data-dependency relationships between the operations in the input specification, i.e. the concurrency is expressed implicitly.

## 2.2. Control-flow Based Synthesis

In control-flow based synthesis, the sequence of operations that synthesized system will execute is based on the control structure of statements in the user's input description. The goal of a control-flow based synthesis system is to create a finite-state-machine controller, often implemented using a ROM (microcode) or a PLA.

**2.2.1. Language Considerations** Languages used as the input specification for a control-flow based synthesis system must contain facilities for explicitly stating the series/parallel behavior of the system. These facilities are called *control structures* [Aho77].

In a programming or description language, it is important to be able to combine several operations and treat them as a single unit or **block**. In many programming languages, (Algol, Pascal, and Ada, for example) blocks are formed by specifying the keyword **begin** followed by a sequence of statements and then the keyword **end**. It is also important to provide facilities to allow an operation to be executed a number of times in a loop. A **while** statement is an example of such a facility. In a while loop, the first expression evaluated is the test *condition*. If the condition returns true, than the statement that is the *object* of the loop is executed, and the control returns to the test at the start of the loop. If the condition returns false, the loop terminates. For example, in the while loop:

```
j = 0;
while (j < 10) do
begin
        j = j + 1;
end
```

(j < 10) is the condition, and j = j + j is the object of the loop. The successive *iterations* of the loop execute sequentially. A **for** loop provides iteration just as the while loop does, but the number of iterations is indicated by one of the parameters to the loop. The loop is executed a number of times, with the *iteration variable* bound to a different value each time through the loop. In the loop:

```
j = 0;
for i = 1 to 10 do
begin
        j = j + 1;
end
```

i is the iteration variable. In a language that supports concurrency, it is also useful to have parallel control structures to group and control concurrent operations.

There is no parallel construct that directly corresponds to the **while** statement, since the definition of **while** is that it evaluates its exit condition after every execution of the loop. There is, however, a parallel control structure that is similar to a **for** loop, called a **forall** [Ackerman79] loop. In a **forall** loop all the instances of the statement or expression that are the object of the loop execute in parallel, with each instance having the iteration variable bound to its proper value.

In sequential languages, there is only one type of block, the *serial* block. Statements in a serial block are executed one after another in sequence. In a language for describing concurrent systems it is useful to define another kind of block, called a *parallel* block, [Shaw74] where all the statements in the block execute in parallel. In existing concurrent programming languages parallel blocks are delimited by **parbegin** and **parend** [Shaw74], or **cobegin** and **coend** [Brinch Hansen76].

**2.2.2. Existing Description Languages** There are many other ways to specify the behavior of concurrent systems. The ISP hardware description language, for example, uses **begin..end** for both serial and parallel blocks, with statements within a block separated by semicolons. The rule for determining the series/parallel behavior of a block is that all statements from the beginning of a block until the occurence of a **next** statement occur in parallel then all statements from that point until the next **next** statement occur in parallel, and so on.

Another method for specifying series/parallel behavior is found in the DDL-P hardware description language [Cory79]. DDL-P divides the control and data-flow portions of a description into two separate parts. This is done to eliminate the need for data-flow analysis, which is described in more detail in the next chapter, making the job of synthesis programs easier. The data-flow portion of a DDL-P description specifies the connectivity of the circuit and the basic data-path operations, with the user giving each data-path operation a unique label. The user then specifies the behavior as a set of states and state transitions. Each state consists of a label that is the name of the state, followed by a comma-separated list of operations (labels from the data-path declaration) to be performed in that state, followed by either a conditional or unconditional transfer to a new state. Concurrency is specified by having several comma-separated data-path operations in a single state. A drawback of the DDL-P style of specification is that the decomposition of a design into separate data-path and controller portions can be difficult and/or unnatural.

The ADLIB/SABLE simulation system [Hill80], uses ADLIB, a extended version of PASCAL, to describe the behavior of components and SDL, a structural definition language to specify the interconnection of components. ADLIB extends both the syntax and the semantics of PASCAL to include the notion of *components*, *networks*, and *processes*. A strength of ADLIB is that it is built on top of a software programming language so that it provides facilities for data-type specification, iteration, and for procedure definition and invocation that are usually absent from hardware description languages. ADLIB also provides a good set of primitives for describing communication between modules. A drawback of ADLIB is that the type structure of pascal is not hierarchal and therefore procedures that work on one type of data must be re-written to work on any other type. Another is that modules are connected by way of SDL descriptions rather than by statements in ADLIB or some combination of both.

AHPL [Hill73] is a hardware description language based on a subset of the APL [Iverson62] programming language. In AHPL, operations between vectors occur in parallel for all elements of the vectors. In addition, register-transfer (assignment) statements that are on the same line in the source code also occur in parallel. AHPL descriptions contain no explicit type declarations or structure, every data object is viewed as an array of bits. AHPL inherits APL's ability to produce terse descriptions. The positive and negative aspects of this are well known in the software field [Barlow79]. AHPL also suffers from APL's lack of a good iteration facility.

The Slang [Foderaro82] simulation language is similar to ADLIB/SABLE in that connectivity between modules is given by explicit connections. A major strength of Slang is that it is embedded in lisp, so that many of the lisp primitive operations are available to a user writing a Slang description. However, Slang is similar to AHPL in that it lacks data-type declarations and thus would probably not be a good input language for an automated synthesis system.

**2.2.3. State and Statements** In control-flow based synthesis, the concepts of a *state* and a *statement* are very important since the controller being generated is a finite-state machine. The current state of the controller is given by the *state vector* which is the value of the *state counter.* The state counter performs a function similar to that of the program counter in a computer, i.e. it determines the order in which events occur. When a design specification is read, the statements of which it is comprised are translated into an internal form. This phase of the translation process is called the *compile time* phase. During this process each statement is tagged with the state in which it will execute. As each statement is translated, the state counter value to be associated with the next input statement is also determined, and microcode or logic equations for the *next-state* transition for the current state are created. In most cases, this information specifies that when the current statement is finished being executed the state counter should be incremented. However, next-state generation for **if**, **case**, and **while** statements is more complicated, because the next-state transition can depend on values from the data-path.

**2.2.4. State Counter Generation** The problem of generating optimal state assignments is a difficult one [Kang81]. A simple, although non-optimal assignment rule for a block-structured specification language follows. (Note that this rule assigns the *sequence* of states but not their *coding* For a serial block, each statement to be executed is assigned the state following the state of the previous statement. For a single parallel block, or nested parallel blocks the state assignment is even simpler: all statements in the block are assigned to execute in the same state. The state counter is therefore a vector, rather than a single value, because of the need to be able to refer to the states of statements inside of nested serial and parallel blocks. For example, in the section of code in Figure 2.1, one state assignment would be to assign statement_A to occur in State 1, and statement_B to be State 2. The entire parallel block would then execute in state 3. There is no problem in having statement_C execute in State 3, but what about statement_D and statement_E? One way to handle this program is to break the notion of time down into smaller units at this point, so that statement D executes in State 3.1 and E executes in State 3.2. Every time a "begin ... parbegin ... begin ..." sequence occurs, the current level of parallelism (plevel) is incremented, and state_vector[plevel] becomes the current state vector element.

**2.2.5. Variables and Assignment** In control-flow based synthesis, variables represent storage locations, just as they do in conventional programming languages. Assignment statements represent transfers between registers. When an assignment statement is encountered it means that the variable being

```
begin
        statement_A;
        statement_B;
        parbegin
                statement_C;
                begin
                        statement_D;
                        statement_E;
                end;
        parend;
end.
```

Figure 2.1 - A Simple Parallel Program

assigned to should be set during the current state, and that the value it should be set to is the value that the right-hand-side of the assignment statement has in that state.

**2.2.6. Data-structures** In a control-flow based synthesis system, data structures may be treated as they would be in a conventional programming language. One major difference, however, is that a single memory with one port might not be adequate because of independent parallel accesses to memory by concurrently executing portions of the generated system. There are two ways to handle this problem. The first is to employ the same methods used in multiprocessors. These methods range from local caches, to multiported memory, to centrally controlled, pre-scheduled interconnection networks [Kuck78]. The second method is to partition the "memory" into many separate memories, with the limiting case being one memory (a register) for every variable in the input specification.

**2.2.7. Optimization** There several ways to reduce the amount of hardware generated without reducing the performance of the resulting system. One way is to share, or multiplex, the hardware generated for execution paths that are mutually exclusive; for example, the blocks that are executed as the disjoint consequents of **if** or **case** statements. Another way is to share specific modules between paths that use them at different times. For example, one path might use an adder and then a multiplier, and another path might use a multiplier first and then an adder. If these two paths are always started at the same time and run in lock-step, then it is possible to share the modules between them. A problem with this approach is that the user's specification might not have been written with this sharing in mind. In that case it is possible that the execution ordering specified for the two paths could stop them from sharing resources when the logic of the algorithm allowed it. The way to get around this problem is to look beyond the execution ordering specified by the user's program, and determine the optimal execution order from the *dependencies* between the variables it contains. This approach, which is similar to the one used by optimizing compilers, is the subject of the next chapter of this report.

# CHAPTER 3

# DATA-FLOW BASED SYNTHESIS

## 3.1. Introduction

The ideas in data-flow based synthesis of digital hardware have evolved from research in data-flow computer systems. Data-flow machines [Dennis80] are tightly-coupled multiprocessors; that is, they are computers with many processors that work closely together on the same problem. The programs for data-flow machines are described by directed graphs rather than by sequences of instructions. The vertices of these graphs represent functions to be performed on data values (or tokens) and the arcs represent communication paths that carry these values between the vertices. As an example, a data-flow graph for performing the evaluation of the expression (B^2)-(4*a*c) is shown in Figure 3.1

## 3.2. Data-Flow Computers

The architecture of data-flow computers is also different from that of standard von Neumann computers. Data-flow machines have no program counter, no main memory and no CPU. Instead, they have *functional units* connected by a communications network. The functional units are computational elements that serve the same purpose as functions in a program; they accept one or more operands and produce one or more results. In a data-flow machine there is no central controller; a node executes (or "fires") when, and only when, there are new values on *all* of its inputs. It then produces new values on its outputs. In classical data-flow model, as defined by Dennis [Dennis80], functional units do not store any information from one firing to the next. This insures that they are true *combinational* functions – their outputs

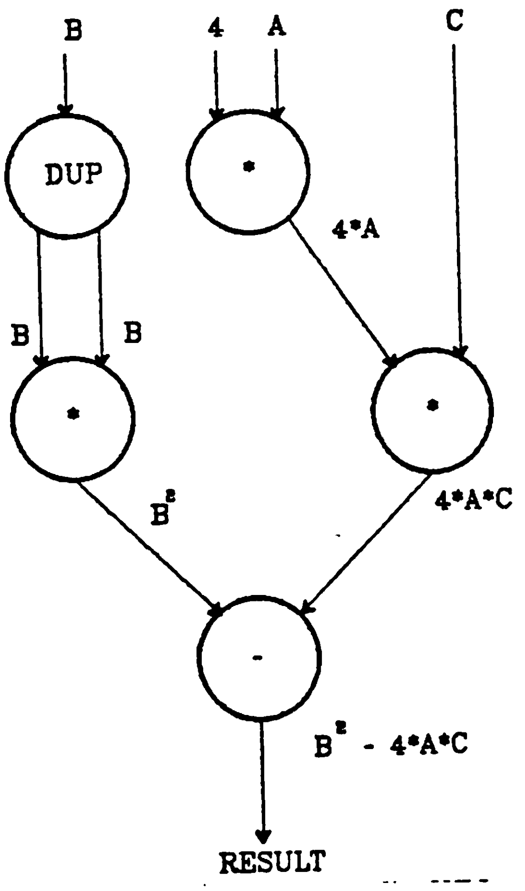

Figure 3.1 - A Data-Flow Graph

depend only on their inputs.

At a high level, data-flow graphs are similar to logic diagrams. In fact, some simulators work in what is basically a data-driven fashion [Newton78], scheduling nodes to be evaluated only when their inputs change. If the data values transmitted are the changes in the values of the inputs rather than the inputs themselves, we can see that this evaluation is data driven, although it is not a strict data-flow computation, since the nodes may also contain storage. There is also a similarity between data-flow functional units and *self-timed systems*

[Seitz80]. In a self-timed system, there is no central clock – the modules are synchronized through signals that tell when the module is ready to receive data and when it has completed its processing. An example of such a module, synchronized by *data-request*, *request-acknowledge*, *data-available*, and *data-acknowledge* signals is shown in Figure 3.2 Again, the availability of data, not a central controller, determines when each function is to be evaluated.

It is always possible to construct a special-purpose machine to execute a given data-flow graph. One simple optimization is to replace the general communications network with special purpose network (such as a set of dedicated busses) since the pattern of communications between the functional units is known. Another optimization is to provide only as many copies of each kind of functional unit that are necessary to achieve the desired level of performance. Yet another optimization is to synthesize the more complicated functional units from combinations of simpler ones in situations where speed is

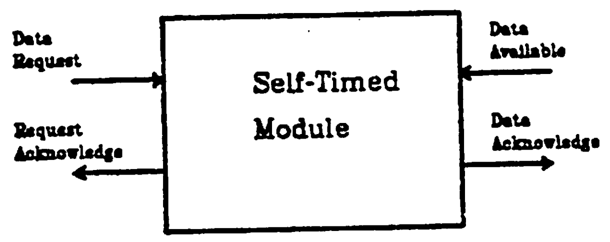

Figure 3.2 - A Self-Timed Module

not critical.

Data-flow based synthesis involves generating a data-flow graph from an input program, performing operations on that graph to get the optimum amounts of parallelism and pipelining for the particular application, and then using that graph as the net and component list input for a computer-aided design system that contains a library of modules and a program or set of programs for placing the modules on a substrate and routing the connections between them. If the modules used as primitive functions are not self-timed, then the graph can be *leveled* with respect to module delays [Bottorff82] (time or number of clock cycles) and a finite-state machine based controller can be generated that causes the nodes at each level in the graph to fire at the proper time.

## 3.3. Data-Flow Analysis

In a formal sense, a data-flow graph represents a set of functions and the dependencies that exist between them. The process of determining these dependencies is called *data-flow analysis*. An accurate data-flow analysis of programs written in most conventional programming languages is a difficult problem and can take considerable time to perform [Aho77]. However, if a program is written in a language which allows the designer or programmer to avoid constructs which make data-flow analysis difficult, the complexity of the analysis process can be reduced to a manageable level and the data-flow graphs can be produced in reasonable amounts of time. If the right language structures are provided, the user should not need these dangerous features to write programs or hardware specifications.

**3.3.1. Language Constructs** There are several constructs in conventional programming languages that complicate data-flow analysis. Unrestricted **goto** statements greatly complicate data-flow analysis and introduce unnecessary dependencies, which reduce the speed of the program. The reason for this is that the values of variables referenced in the statements following a label depend on assignments to those variables in the code sections that can jump to the label. Since the flow of control that leads to a label is almost always dependent on the values generated by the program at run-time, the data-dependencies calculated at compile time will in general be overly pessimistic.

Global variables (variables that are declared outside of any function, and therefore sharable by all of the functions) also cause problems. When a global variable is encountered, its last use cannot be determined without examining the entire program. Dependencies involving global variables can span multiple functions, making separate compilation difficult or impossible. Although global variables can cause difficult problems for a data-flow analyzer, global constants do not. Because constants do not change, a copy of the constant may be used whenever a reference to the constant appears.

Static variables (variables local to a function that retain their value between calls to that function) are not allowed in most data flow programming languages because they can create dependencies between the order of calls to the function. In effect, they represent a communication path between all functions which call the function. Like many of the language features that complicate data-flow analysis, static variables can be supported if the notion of data-flow computation is extended to allow explicit storage.

Another problem, often called *aliasing*, occurs when more than one name is used to represent the same variable. Aliasing makes data-flow analysis more difficult because it creates hidden dependencies. For example if the same variable name is allowed to occur more than once as a actual parameter in a function call and a *call by reference* or *call by name* style of passing parameters to subroutines is used, then inside the routine the formal parameter names of those arguments will be aliases for each other. If the function expects two different variables as arguments and modifies one of them, incorrect results will occur if the same variable is used for both. If a language allows separate compilation of subroutines, it may not be possible to determine at compile time if such a problem exists. Aliasing problems do not occur very often for scalar parameters because users tend to pass them by value. Arrays, however, are often passed by reference and aliasing is much more likely to be a problem when array-valued parameters are used.

Call-by-reference and call-by-name addressing can also create problems when several functions are called with name or address of the same object at the same time and one or more of them tries to modify that object. Because such errors depend on the run-time behavior of a program they can be very difficult to find.

**3.3.2. The Single Assignment Rule** The removal of goto statements, global variables, call by reference, and aliasing greatly aids the compiler in determining what expressions a variable depends on and which expressions depend on the variable.

One remaining problem in conventional languages is that variables are often used as "scratch-pad" temporaries -- they are assigned a value and then reassigned another value within the same section of a program. This complicates the data-flow analysis, because it creates a false dependency between the old value of the variable and the new one, decreasing the amount of parallelism available in the program. One way to eliminate this problem is to introduce a rule that specifies that a variable may be written only once within a given scope, while allowing the variable to be read as many times as desired. This convention, called the *single assignment rule*, is found in the data-flow languages VAL [Ackerman82], and ID [Arvind78]. While the single assignment rule makes analysis easier, it also forbids such familiar constructs as "A = A+1", the function of which must be provided by other language facilities. The most common case where such re-assignment occurs is in the index variables and auxiliary variables of loops. The way the looping constructs of data-flow languages solve this problem is described in the next section of this report. If a language does not allow **goto** statements, global and static variables, call by reference and aliasing, and enforces the single assignment rule, then that language is called an *applicative* or *functional* language. These languages are are the subject of considerable research [Morris81] since the same properties that make data-flow analysis of programs written in applicative languages tractable also cause the programs to be more modular and easier to maintain. In applicative languages, variables are not storage cells, but instead are like macro names for expressions. A data-flow graph can then be built by tracing from the constants referenced in the program and from the program's external inputs and in every place a variable name appears, substituting the expression that corresponds to that variable name. The final result of this process is a data-flow graph for the program.

**3.3.3. Loops** The single assignment rule must be modified slightly if the language is to allow loops. In general, loops require that the values of certain variables (called the *iteration variables* of the loop) change from one iteration to the next. One way avoid the prohibition of reassignment is through the use of a special keyword that specifies that the assignment is for the value of the variable for *the next time through the loop*. In the data-flow language ID this keyword is called **new**. In the body of a loop, one might see "**new** x = x + 1", meaning that the value of x in the next iteration would be one greater than its value during the current one. A loop in a program results in a loop in the data-flow graph. These loops are a form of feedback; therefore, when hardware is generated, registers must be created to hold the values of the iteration variables.

**3.3.4. Arrays and Data-Structures** The single assignment rule and call-by-value work quite well for scalar variables, but arrays and data-structures require special consideration. Arrays and structures can be viewed two ways -- either as single, large, objects or as collections of smaller objects. In data-flow languages, arrays and data-structures are almost always viewed as single objects with the result being that the entire array or data-structure must be assigned to as a single unit. Languages (such as VAL) that use the single object approach provide constructs that copy the entire array, substituting in new values for particular chosen elements. This allows the programmer to create new arrays with small changes from the original ones, rather than changing individual elements. This may seem to be more expensive than the second approach (dealing with an array as a collection of smaller elements) but it has the advantage that the entire array can have one set of control information rather than a control block for each element of the data-structure or array.

In hardware synthesis, the values of all variables, arrays, and structures, as well as scalars, are carried on busses and stored in registers. The array element substitution operation compiles to a set of busses which connect to, or go through, a functional module. Any values that are specified to be unchanged by the source code are passed through the module directly and any that are be given new values in the code receive them from the module.

**3.3.5. Functions** The evaluation of functions in a data-flow program results in an interesting set of problems because a function may be called from more than one place and it may also be necessary to allow more than one invocation of the function to be active at the same time. In hardware, this corresponds to several modules using the same type of submodule at the same time. There are two approaches to the problem. The first, and simplest, is called *code copying*. Every time a function is called, a copy of the code is created and used for the function. Thus, every caller has its own copy of the function and conflicts are avoided. The other approach is to use *colored tokens* [Arvind78], where each value in the graph has a tag associated with it that tells what function invocation it belongs to. For digital synthesis, the code copying approach is simpler and results in functions being treated like software macros, where each call of a module results in a new copy of the module being inserted in the calling circuit.

## 3.4. Parallelism, Pipelining, and Graph-folding

Data-flow graphs can contain both parallelism and pipelining. The parallelism results from more than one functional unit at the same level in the graph being active at the same time. Pipelining results from units on more than one level in the graph being active at the same time. The output of the data-flow analysis section of the compiler is a data-flow graph where the only dependencies are those that result from data dependencies in the input program. This graph gives the maximum possible parallelism and pipelining for that program. It is possible to restrict one or both of these properties by performing operations on the graph that introduce additional constraints. These operations correspond to folds in either the *width* (decreasing the number of units on a given level) or *length* (decreasing the number of levels that may be active at one time) of the graph. To prevent two units of the same type at the same level from being active at the same time we make one unit depend on the output of the other unit by creating a *symbolic* data dependency. The dependent node is then constrained not to fire until the node on which it depends has finished just as if it depended on the particular value that node produced. This is called *widthwise folding*. An example of widthwise folding is shown in Figure 3.3 where the graph of the expression (a + b) + (c + d) which can require that two adder functional units be available at the same time is folded to produce a graph that can only require a single adder at a time. It is also possible to reduce the maximum amount of pipelining, creating *lengthwise folds*,

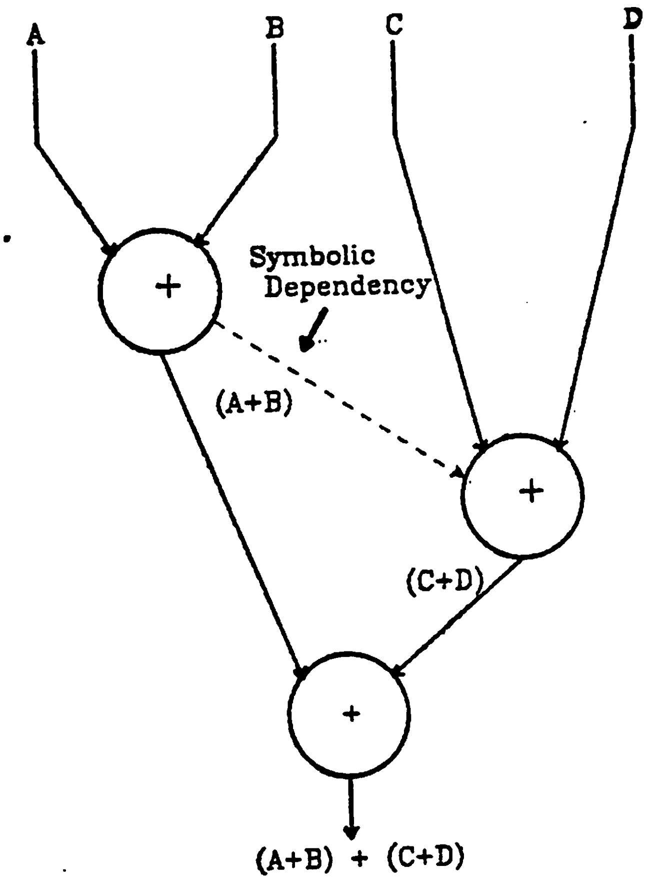

Figure 3.3 - Widthwise Folding

decreasing the number of levels in the graph. Lengthwise folds are created by overlaying one or more levels in the graph with previous levels. This is done by generating symbolic connections from the outputs of a level of the graph to the inputs of a previously generated level. A symbolic multiplexer node is then generated at each input of each functional unit that receives a signal which is fed back. The number of inputs to the multiplexer is equal to the number of signals that share that functional unit and the multiplexer contains a counter that determines which input it should direct to the functional unit. Each time the multiplexer generates a value, the counter is incremented. This technique is similar to the use of colored tokens in data-flow computer systems [Arvind78], except that the "color" is kept on the inputs to the functional unit not as part of the data-value. An implementation of a folded graph need not use actual multiplexers: any technique that insures that the correct data-values are transported to the functional units can be used. One set of levels of the graph can now be thought of as the first iteration of a pseudo-loop, the next set as the second iteration, and so on. Note that this technique does not need a central controller or a central clock. An example of lengthwise folding is given in Figure 3.4 where the multiplexers have two states, called **Odd** and **Even**. When a fold is generated between two functional units, additional control information must be generated to make sure that they receive their input values at the right time. In addition, when two functional units of different types are folded together, a functional unit that can perform both operations must be generated. The limiting case of lengthwise folding is a parallel processor with an interconnection network to switch data-values between the processors and memories. The limiting case of widthwise folding is a multi-stage pipeline processor with feedback and memories between the stages. If both maximum lengthwise and widthwise folding is performed then the result is a microcode representation for use in a general-purpose processor and memory system.

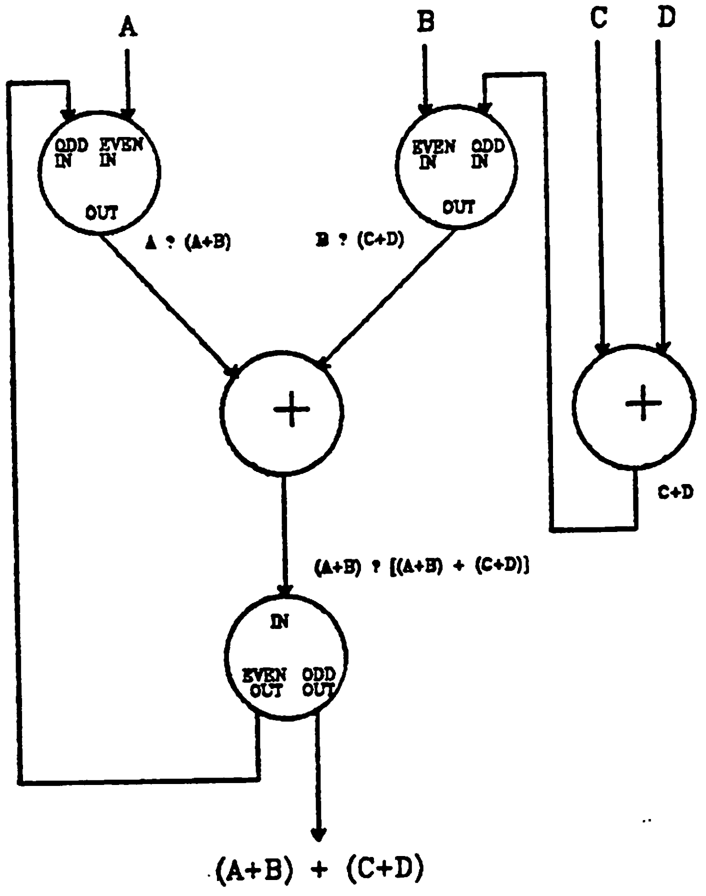

Figure 3.4 - Lengthwise Folding

It may be desirable to limit the parallelism and pipelining of an implementation synthesized from a data-flow graph to reduce the total amount of logic, total chip-area, or to create implementations of the same architecture with different cost-performance tradeoffs. An attempt can also be made to minimize crossovers, i.e. to make the graph *planar* or as close to planar as possible. In general, the larger the number of crossovers in the graph, the more difficult the wiring of the chip or circuit board with a limited number of interconnection layers will be.

## 3.5. Combined Methods

Both control-flow and data-flow based systems have their respective advantages. To be able to combine the best elements of both approaches it is necessary to have a specification language that is facilitates data-flow analysis and also provides constructs for the explicit specification of concurrency. With these restrictions, it is possible to have the translation system compare the explicit and the implicit control and data flow aspects of the user's specification and check for errors. These errors fall into two classes. The first kind of error is when the user's specification fails to specify that two computations can be performed in parallel but the data dependencies imply that they can. An error of this kind can mean two things: either the user has neglected to recognize some potential concurrency (a relatively harmless error) or there is some external constraint that the user neglected to include in the specification. The second kind of error occurs when the user specifies that two events can go on in parallel, but the dependencies prevent them from doing so. Catching this kind of error is extremely useful for high performance design since it points out potential bottle-necks to the user so that they can be examined and corrected.

# CHAPTER 4

# THE ARCHITECTURE OF FUNCTIONS TO LOGIC

## 4.1. Introduction

This chapter describes the background of the FTL project and the architecture of the FTL simulator. The goal of the Functions to Logic (FTL) system is to make it easy for a user to describe the behavior of a digital system in a way that makes its natural structure and concurrency available for simulation and synthesis. VLSI digital systems are very complex and highly concurrent. These two problems of complexity and concurrency have been examined in two different prototype systems, FTL1 and FTL2.

## 4.2. The Syntax of FTL

The input to FTL is a lisp-like language, which has a uniform syntax that shows the tree structure of a user's input specification and makes it easy to add extensions the language. Alternate syntaxes are possible through the use of input pre-processors. The input to FTL is viewed as a sequence of *symbolic expressions*. A symbolic expression (or *s-expression* for short) is either an atomic symbol (such as a variable or function name, a number, or a quoted string) or a *form*. Some examples of atomic symbols are: 10, "Cycles used", and **clk**. A form may be a control structure (such as a **while** or **if**), a language facility (such as a data-type declaration) a primitive function, or a user defined module, macro, or *alias*. An alias in FTL is a parameter-less macro (note that the term "alias" is used differently here than it is in the chapter of this thesis that describes data-flow analysis). Arguments to forms may themselves be forms. All forms may be thought of as functions, since they all return a value. There are no separate "statements" in the sense of Pascal or 'C'. A form consists of an open parenthesis, the name of the form, the arguments to the form, and a close parenthesis. For example:

```
(+ x y)
```
returns the sum of x and y,
```
(< a b)
```
returns true if a is less than b and false otherwise.

and
```
(for ((x from 0)
      (i from 0 to 10))
     (new x (+ x i)))
```

returns the sum of the numbers from 1 to 10.

## 4.3. Object Oriented Programming

FTL1 is an interpreter for an object-oriented programming language similar to ICL [Ayres79] and Smalltalk [Xerox80] Object-oriented languages help users manage complexity by providing a hierarchal type specification mechanism. Hierarchal type specification allows the user to create generic *classes* of objects and define functions that operate on them, and then create more specialized objects without necessarily having to create new functions to manipulate them. For example, FTL1 provides integer and floating point numbers as most languages do. In FTL1, however, integer and floating point numbers are created as special cases of the class *number*, where number is a special case of the type of the class *object* (the most basic data-type in FTL1). Therefore, if a user defines a function that works on *numbers* it will also work on *integers* and *floats* without any special effort. For example, the functions in Figure 4.1 define addition and multiplication for complex numbers. Note that it is possible to define several functions of the same name, and that it is also possible to define functions with the same names as primitive functions such as "+" and "*". The only thing that must be different is the types of the arguments they take. This ability is useful in VLSI where a user may have several modules that only differ from each other by such characteristics as word-length, drive, input and output impedance, etc. and it is desirable to create *generic functions* that operate on them. The class *number* is the *parent* of the class *integer*. Integer is the parent type of the classes *bit* and *byte*. In FTL it is possible to have several functions that have the same name but differ in the type of the arguments they expect. These are called *variants* of the function. Whenever a function is called on an argument, the variant of that function whose formal argument type most closely matches the type of the actual argument is called. An exact match is sought, and if one is not found then the variant that matches the closest ancestor of the actual parameters is chosen. The implementation of FTL1 is approximately 1200

```
(define-type complex slots ((type integer name real)
                            (type integer name imaginary)))

(define-function + ((type complex name a) (type complex name b))
  (make-complex
   real (+ (a real) (b real))
   imaginary (+ (a imaginary) (b imaginary))))

(define-function * ((type complex name a) (type complex name b))
  (make-complex
   real (-
        (* (a real) (b real))
        (* (a imaginary) (b imaginary)))
   imaginary (+
             (* (a real) (b imaginary))
             (* (b real) (a imaginary)))))
```

Figure 4.1 - Polymorphic functions

lines of Franz lisp [Foderaro81], a dialect of the lisp programming language developed at UC Berkeley.

## 4.4. Introduction to FTL2

FTL2 is a language for describing concurrent systems. It provides constructs to allow the user to explicitly specify the concurrent behavior of their system while avoiding language features that make data-flow analysis extremely difficult. FTL2 evaluates all arguments to a function call in parallel, and provides a set of primitive mechanisms for describing concurrency in a structured fashion. These structures help the user in writing correct descriptions in the same way as normal control structures help users write correct programs in standard high-level sequential languages.. In FTL2 there is a matching concurrent control structure for almost every sequential one. For example, the **serial** function evaluates all of its arguments in sequence and serves the same purpose as a block in a programming language like pascal or 'C'. The **parallel** function is the matching concurrent version of serial. Parallel evaluates all of its arguments (statements) in parallel. To step through the elements of a vector sequentially, the **for** function can be used. To step through them in parallel, the **forall** function may be used instead.

FTL evaluates functions and moves data in such a way that the results from a FTL program running on a sequential machine are the same as they would be on the concurrent digital system that the user's program describes. The current implementation of FTL2 is approximately 2000 lines of Franz lisp.

**4.4.1. Explicit Delay Mode** Many new questions arise in the design of a system like FTL that are do not occur in normal sequential programming languages. The most interesting ones concern the notion of time. The model of time in normal programming languages is very simple. Statements in a program are executed in sequence.

In a language that provides concurrency, things are much more complex. FTL currently supports two models of time. The first one is called *explicit delay mode* because time delays must be provided explicitly by the user program. The model provides sequencing, so that statements inside a **serial** or **for** happen sequentially and those inside a **parallel** or **forall** happen in parallel. However, the current time is only advanced when the **delay** function is used; normally it takes zero time to go from one statement in a serial block to the next. Thus, if the system is set to report an error when a variable is written and then read at the same point in time, it will complain when a variable is set and then used in two adjacent steps in a serial, unless the user puts an explicit delay between the two steps. Explicit delay mode is useful for modeling asynchronous systems, or systems with more than two phases of clocking.

**4.4.2. Implicit Delay Mode** The other time model is called *implicit delay mode*. In implicit delay mode, sequencing and time are combined into one. The basic idea is that time is broken down into finer and finer grains as necessary. Each step in the outermost serial block of the program is defined to take place one time unit (clock tick) apart. The time when statements in the inner blocks occur is defined as follows. Suppose one of the statements in the main block is a parallel. If the parallel block is, for example, the fourth statement in the main block, then it will be evaluated at time = 4, and since it is a parallel block, the statements inside it will also be evaluated at time = 4. Now, if one of those statements is a serial block, then the statements in the block must be evaluated in sequence - one after another - therefore they cannot all be evaluated at time = 4; the first one will be evaluated at time = 4, the second one at time = 4.1, the third at time = 4.2, etc. If there is a parallel block inside *that* serial block, and a serial block inside that, the problem repeats, and it would be necessary to be able to talk about time = 4.1.1, time = 4.1.2, etc. The rule is that whenever a serial block..parallel block..serial block combination is seen, a new level is added to the state vector. The width of the state vector is proportional to the maximum depth of parallelism in the source program.

**4.4.3. The Compiler** The translation and execution of FTL programs is broken into two major phases. The compilation phase reads the input program and produces an internal tree representation of it. The interpreter phase then walks over the tree, emulating a hypothetical parallel machine. The compiler or *reader* portion of FTL reads in symbolic expressions from the user's input description and creates an internal tree representation. Each node in the expression tree contains an operator, a node name, the name of the node's parent, a list of the names of its children, the names of functions to call when evaluating the expression (called *message handlers*) and some other assorted fields. The action taken by the reader when an expression is first read is based on the object type of the expression. If the expression is an atomic symbol, then that symbol is either a number, a variable name, or an *alias*. If it is not an alias, then a tree node is created to hold the symbol and normal processing continues. If it is an alias, however, then the text for the alias is read as input, replacing the alias name. In FTL, the **alias** function is used to For example, if the expression:

**(alias index (+ index-register offset))**

is seen in the text, then if

**(get-register index)**

was read, the result would be the same as if the expression

**(get-register (+ index-register offset))**

was seen in the source.

If the expression is a form, then action taken is based on the class of the first element of the form, the form name. If the form name is the name of a macro, then the text for the macro is expanded with the arguments to the form being the arguments to the macro. For example:

If

**(macro square(x) (* x x))**

is read, and later on

**(square a)**

is read, then the result is the same as if

**(* a a)**

was seen in the source.

If the form name is the name of a function that only declares a variable or sets a switch in the compiler and doesn't generate any code during simulation or hardware during synthesis, then the handler routine for the function (linked via the property list of the function) is called and is given the current tree and input context as arguments. (In the code for the compiler, these functions are called *magic compile* functions because they don't produce any code.) If the form name is the name of a built-in facility (such as a looping function) then that function is called a *special compile* function and the handler function for that facility will be called and in most cases will modify the input before passing it to the next level of the compiler. If the form name is the name of a primitive function, (such as + - * or /), then the compiler generates code to call the primitive directly. Finally, if the compiler does not know what the form name is, it is assumed to be the name of a user defined module, and code is generated to call the module with the proper linking. Modules in FTL represent the modules of a digital system and serve the same purpose as user defined functions in a normal programming language. Because of the concurrency provided in FTL it is possible that more than one parent module might try to use the same child module at the same time. In FTL, if two modules try to use the same module at the same time it is considered an error. This insures the designer that there are no access conflicts to the module. If more than one copy of a module is needed at the same time in a digital system being described, then a macro can be written that generates the circuitry inside the module, and as many modules as necessary can use that macro. An example of this technique is found in chapter 5.

The decisions about which functions are "special" and what functions should be called to handle them is almost completely table driven. To add a primitive or special form to the compiler, all that is necessary is to enter the form name into the table for either magic or special compile and write the handler function. The handler function is passed the code tree being constructed, the name of the node being constructed, and the input text.

**4.4.4. The Interpreter** After an expression has been read by FTL and the tree that represents it has been constructed, the tree is walked by the interpreter and the expression is evaluated. The interpreter represents a hypothetical augmented-data-flow tree-machine, [Deutsch80], just as the pascal *p-machine* [Jensen79] represents a hypothetical stack machine. The way the machine executes programs is by way of a top-down flow of control, followed by a bottom-up flow of data (see Figure 4.1). The top-down messages are called *eval* messages. An eval message asks a node to determine its value and return it. The bottom-up messages are called *data* messages. These messages are used by child nodes to transmit values to their parents. Conceptually, the interpretation of the program is done by concurrently sending these messages between nodes of the tree. Actually, there is a central evaluation queue, and when a message is sent to a node the message handler in the node that is responsible for dealing with that type of message is called. The message handler takes whatever action is necessary and then returns a list of nodes to be entered onto the end of the queue. There are three basic messages a node can receive: *eval*, *data*, and *fire*. An eval message asks the node to evaluate itself, which usually requires it to evaluate its children (put them into the evaluation queue). A data message asks

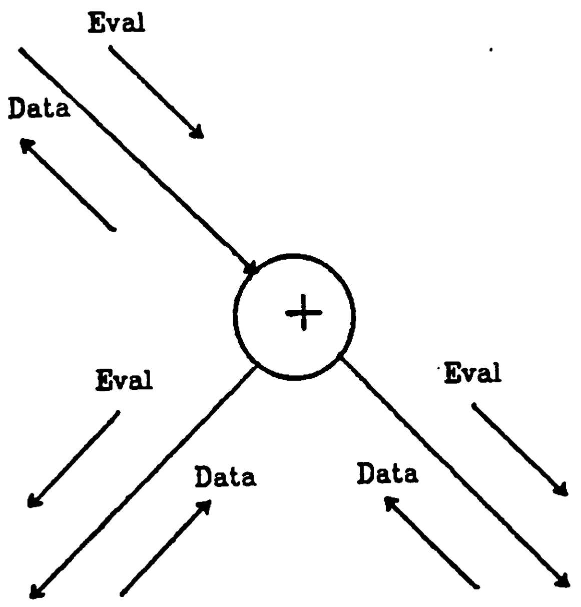

Figure 4.1 - expression evaluation

a node how it would like to handle the data value that one of its children wants to pass back to it; for example, when a statement in a serial block returns its value to its parent, the parent will schedule the next statement in the block, or the parent will schedule its parent if no statements are left. Finally, a fire handler tells how a function is supposed to be executed once it has all its arguments. The **walk** routine in the interpreter is responsible for taking nodes off the evaluation queue and calling their eval and fire message handlers. If there are no nodes left in the evaluation queue then walk calls the **step-time** routine. Step-time finds the nodes in the delay-list with the smallest delay, then advances the clock and moves those nodes from the delay-list to the evaluation queue. The way that the walk routine and the message handlers decide which handler to call on a particular node is by way of the **state** field in the node. If the state of the node is DORMANT, then it means that the code that sub-tree represents is idle – it is not in the process of being executed. If a node is not dormant, it means that the code below it is in some stage of evaluation. For example if the node state is WAIT-DATA, then it means that the eval message has been sent to one or more children of the node and the node is waiting for the child or children to call the node's data-handler to send it their values. If the node state is READY, then it means that the node has all values it needs from its children and is ready to have its fire-handler called. Since the walk routine only distinguishes between READY and non-READY states (calling the fire-handler in the first case and the eval-handler in the second one) most of the eval-handlers have a case statement used to choose a code segment to execute based on the state of the node.

**4.4.5. Message Handlers** The simplest set of message handlers are those used for **parallel** blocks. When a parallel block is entered, the parallel-eval-handler is called and returns a list of its children – the forms in the block. When each one of these functions returns, it calls the parallel-data-handler to enter its value into the parent's list of values. Finally, when all the functions in the block have returned, the parallel-fire-handler is called, which sets the **parallel** dormant, and calls its parent's data-handler. The message handlers for **serial** blocks are slightly different. When first called, the serial-eval-handler schedules the first form inside the serial. Then, every time a form in the serial returns, the serial-data-handler causes the next form in the serial to be scheduled for execution. Finally, when the last form returns, the serial-fire-handler makes the value of the last form the value of the serial, sets the node dormant, and calls its parent's data-handler. Many special forms use the message handlers from other forms. For example, primitive functions use the "parallel-eval-handler" to evaluate their arguments, and the "parallel-data-handler" to gather the values of their arguments. The difference between primitives and parallel blocks, however, is that when primitives fire, they use the "primitive-fire-handler". The primitive-fire-handler calls the lisp function that the primitive represents giving it the now completely evaluated arguments as its arguments. User-defined modules are simulated by evaluating all their arguments in parallel, then calling the "module-fire-handler" *linking in* the user's module. The linking process involves examining the "node-parent-name" field in the root node of the module. If the value of that field is non-null then it means the module is already in use and the error is reported to the user. If not, then the "node-parent-name" field of the module is set to the node-name of the node that calls it, and the code of the module is executed.

The handlers for conditionals are more complicated. The **if** form looks like a function call, but it evaluates its arguments in a special way. **If** evaluates its first argument (the condition) and if the condition returns **true** it then evaluates its second argument. If the condition returns **false** it evaluates its third argument if there is one, otherwise it just returns.

The while-eval-handler evaluates (sends an eval message to) its first argument, the condition, and if the condition returns true it evaluates its second argument the object of the loop. If the condition does not return true, the while loop returns its value (the value of the while loop) to its parent. The while-data-handler simply schedules while-loop for re-evaluation. Thus, while loops are evaluated in a style more like tail-recursion [Steele78] than straight iteration.

**4.4.6. Variable Management** Like most software programming languages, FTL is "block structured" - it allows a variable to be declared inside a block and for that definition variables to be supercede a definitions of a variable with the same name in an outer block. This is an example of *scoping*. There are two different scoping rules in common use today. The scoping rule used in most current programming languages such as 'C' or Pascal is called *lexical scoping* because the instance of a variable name seen at any point in the program is the one whose declaration is closest to the point in the text where the variable name appears. The rule used in most lisp systems is called *dynamic scoping* because the instance of a variable seen is dependent on the dynamic behavior of the program. When a variable is referenced that is not declared in the function that references it, the environment of the caller of that function is examined, and then the caller of the caller, etc. until a caller is found that has declared a variable of that name. The advantage of dynamic scoping is that the environment for a function or module may be completely defined by its caller. The advantage of lexical scoping is that the source of all variables can be known at compile time. FTL currently uses dynamic scoping because that method fits cleanly into the machine model, but changing it to use lexical scoping would not be difficult.

# CHAPTER 5

# EXAMPLES

## 5.1. Introduction

This chapter begins with some small FTL examples, followed by a FTL description of a large digital system, the RISC [Patterson81a]. microprocessor. RISC was chosen because it has a simple, clear architecture that makes it a good choice for formal specification. It is also a large enough example to demonstrate the use of FTL on real systems.

## 5.2. Low Level Descriptions

FTL may be thought of as a multi-level simulator in the style of ADLIB/SABLE [Hill80]. In FTL, the same system can be described at various levels of detail depending on the type and amount of information it is desired to get from the simulation. As an example, we can model a recognizer for a simple pattern. The recognizer, a sample problem from [Roth79], outputs a 1 if its input pattern is 1101 or 0011, and a 0 otherwise. In the original solution, the recognizer was implemented as a finite-state-machine. The FTL description of the system reads four bits from the input stream, determines whether to put out a 1 or a 0, and then goes back for more input. In example 5.1, the *repeat* form does just that.

Another example is shown in Figure 5.2. Here, the goal is to simulate a pseudo-random number generator based on a shift-register with feed-back through an exclusive-or function. This example demonstrates the use of a numeric parameter combined with the *while* function to limit the number of cycles of a simulation.

```
(module Roth()
 (serial
  (declare (inports (bit x))
           (exports (bit y))
           (locals (bit a b c d)))
   (repeat
    (serial
     (setq a (input x))
     (setq b (input x))
     (setq c (input x))
     (setq d (input x))
     (output y
            (logior
             (logand (lognot a)
                     b
                     (lognot c)
                     d)

             (logand a
                     (lognot b)
                     c
                     d)))))))
```

Figure 5.1 - a simple module

```
(module shift-seq()
 (serial
  (declare (locals (word x0_in x0_out x1_in x1_out x2_in x2_out)))
  (setq x0_out 1)
  (setq x1_out 0)
  (setq x2_out 0)
  (while true
        (serial
         (delay 1)
         (parallel
          (setq x0_in x2_out)
          (setq x1_in (l-xor x2_out x0_out))
          (setq x2_in x1_out)
          (msg x2_out x1_out x0_out))
         (delay 1)
         (parallel
          (setq x0_out x0_in)
          (setq x1_out x1_in)
          (setq x2_out x2_in))))))
```

Figure 5.2 - Shift-register pseudo-random number generator

In the previous example, the time delay of the parts of the module were specified separately from its series/parallel behavior, and temporary variables were used to carry values from one iteration to the next. FTL also has the ability to deal with the previous value of a variable. When operating in *PC* mode, the value of a variable is not updated until after the end of the cycle it is set in, and explicit delays are replaced with an implicit delay between the forms in a serial block, while loop, if statement, or any control structure that involves sequencing. Using this mode, the previous example may be re-written as example 5.3.

FTL may also be used at a much higher level. Since it provides true parallel processing, it may be used to simulate communicating processes in problems such as simulating computer networks on to test the validity of solutions to problems involving multiple parallel communicating processes such as those posed by Dijkstra [Dijkstra76]. FTL provides a function called **lock** that tests a

```
(module shift-seq()
 (serial
  (declare (locals (word x0 x1 x2)))
  (setq x0 1)
  (setq x1 0)
  (setq x2 0)
  (while true
        (parallel
         (setq x0 x2)
         (setq x1 (l-xor x2 x0))
         (setq x2 x1)
         (msg x2 x1 x0))))).
```

Figure 5.3 - Shift-register example using PC mode

variable to see if a variable is 0 and sets it to 1 if it is not. This testing and setting is performed as an *atomic operation*. This means that while the testing and setting are being done no other processes are allowed to run, and thus no other process can interfere. **Lock** returns as its value the old state of the lock. An example of using FTL and the **lock** function for this type of simulation is given in Figure 5.2.

## 5.3. The RISC Microprocessor

One aspect of the RISC project is to determine if a processor with a simple architecture, executing one simple instruction per cycle, can run as fast or faster than more complicated conventional machines. If the number of instructions and addressing modes in a machine are reduced, the size of the machine's control section can also be reduced. The silicon area that is saved can then be used for other purposes. In the case of RISC-1 that area is used to hold a large number of on-chip registers. This concept of optimizing a simple architecture to execute one instruction per cycle was developed independently by researchers at IBM, and is used in the IBM 801 minicomputer [Radin82].

```
(module tlock()
 (serial
  (declare (local (word l))) ;;; node we'll be locking

  (unlock l)

  (parallel

   (serial
    (if (locked? (lock l))
         (serial
          (dumpmsg "A locked l ")
          (delay 1)
          (unlock l)
          (dumpmsg "A unlocked l ")
          (delay 1))
         (serial
          (dumpmsg "A couldn't lock l")
          (delay 1))))

   (serial
    (if (locked? (lock l))
         (serial
          (dumpmsg "B locked l ")
          (delay 1)
          (unlock l)
          (dumpmsg "B unlocked l ")
          (delay 1))
         (serial
          (dumpmsg "B couldn't lock l")
          (delay 1)))))))
```

Figure 5.3 - A locking problem with two communicating processes

RISC-1 has 78 registers and 31 instructions. The instructions are grouped into four categories -- *LOAD*, *JUMP*, *ALU*, and *OTHER*. The instructions in the LOAD group are the only ones that reference memory. The rest of the instructions manipulate data from the register file and/or immediate data values given in the instructions. The RISC chip is composed out of five logical blocks. The blocks are the *Register file*, the *Shifter*, the *ALU*, the *Pc Group*, and the *Control Pla's*.

**5.3.1. The Register File** The RISC-1 register file contains 78 registers. The registers are broken up into two groups. The first group is a set of 18 global registers (numbered 0-17) that are accessable at any time by any routine. Register zero is special in that it always always contains the value 0. All the rest are general purpose registers. The second group is a set of 60 registers broken up into six *overlapping register windows*. Whenever a function is called, a special register called the *Current Window Pointer* (or CWP) is decremented; it is incremented whenever a function returns. The CWP is used to select one of the 6 register windows. Each window consists of 14 registers. The windows are conFigured so that the lower 4 (register numbers 18-21) are shared with the caller of the function, the middle 6 (22-27) are local to the function, and the upper 4 are shared with any function the current function calls. A diagram of the scheme is shown in Figure 5.4. The use of overlapping register windows greatly reduces the cost of procedure calls. The only time registers have to be stored to main memory is when the depth of procedure calls is greater than the number of register windows. Statistics of sample programs show that this should be a relatively infrequent event [Patterson81b].

**5.3.2. The Shifter** The RISC chip contains a 32 bit barrel shifter [Mead80]. The barrel shifter can shift a 32 bit word by from 0 to 31 bits in either direction in one cycle. Both arithmetic and logical shifting are provided.

**5.3.3. The ALU** The operations that the ALU can perform are: add, add with carry, subtract, subtract with carry, subtract reversed (b-a rather than a-b), subtract reversed with carry, AND, OR, and XOR. A bit is available in most instructions to specify whether or not the result of the instruction should set the condition codes. In the instruction set *order code* (the binary format of the instructions), and in the FTL description of RISC, the shift operations are grouped with the ALU instructions.

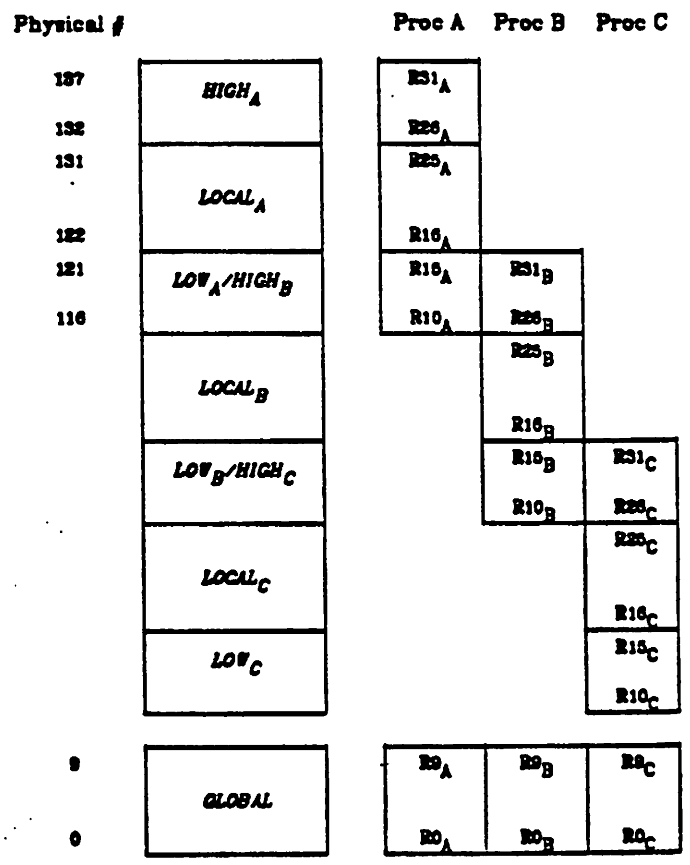

Figure 5.4 - Overlapping register windows

**5.3.4. The PC Group** RISC uses instruction prefetch, and an interesting branch strategy called a *delayed jump*. The delayed jump method means that whenever a jump instruction is executed, the jump is not actually taken until after the instruction following the jump is executed. This method is used to allow pipelining of the CPU without requiring hardware to clear out the pipeline when a branch occurs. Using this strategy it is normally necessary to follow each branch instruction with a no-op so that the results are the same whether the branch is taken or not. However, it is possible to have a post-processor remove most of these no-op's [Patterson81a]. Because of pipelining, the RISC chip actually has three program counters; the last pc, the current pc, and the next pc.

**5.3.5. The Control PLA's** In most computers, the control unit is the single most complicated part of the machine [Lampson81], and in most current microprocessors, the control section takes up from 40-70% of the chip area [Patterson81b]. In RISC, however, the main controller takes only a small portion of the total chip area. The controller is implemented in two PLA's and is positioned in the upper right corner of the chip. The decoder for the overlapping windows register file is also considered part of the control logic.

## 5.4. The FTL specification of RISC

The FTL description of Risc is broken up into 7 files: *opcodes.ftl*, *regfile.ftl*, *macros.ftl*, *alu.ftl*, *jump.ftl*, *memory.ftl*, *other.ftl*, *exec.ftl*, and *testrisc.ftl*.

**5.4.1. Opcodes.ftl** The file opcodes.ftl contains constant definitions for all of the RISC opcodes, with the top two bits of each opcode set to zero. The reason for this is that the top bits are used to decide which of the four classes of RISC instructions *(LOAD, JUMP, ALU, or OTHER)* an instruction belongs to.

**5.4.2. Macros and Aliases** The file macro.ftl Contains some useful *aliases*. The alias facility allows a user to associate a name with a piece of text, and have the FTL compiler replace every occurrence of the name seen after that point with the corresponding text. In the description of RISC, aliases are defined for all the important bit-fields in an instruction, and the word "index" is made an alias for the indexed addressing computation.

**5.4.3. Regfile.ftl** This file defines constants for the major parameters of the register file (i.e. the total number of registers, the size of the register windows, the number of local registers in a window, and between a routine and a routine it calls, of registers global to all routines). The macro "logical-to-physical" takes care of the conversion from a logical register number to the actual physical register number (which depends on the value of the RISC's current window pointer). Since many RISC instructions reference two registers in the same cycle. two paths (modules) must be provided to read from the regfile. This is done by defining a macro "get-register" that contains code to do the work of reading a register, and then creating two modules (called "get-register-a" and "get-register-b"), each of which uses the macro. The last definition in the file is the module "set-register", used to set a register to a value.

**5.4.4. Alu.ftl** Alu.ftl Contains the constant, variable, macro, and module declarations for the RISC's ALU. The macro "alu-worker" takes an opcode, two operands, and the carry flag as inputs, and returns the result of that ALU operation. The module "alu" calls alu-worker and if the "set condition-codes" or "SCC" bit in the instruction is on, sets the condition codes and is used for conditional branches (note that all condition code bits are set in parallel).

**5.4.5. Jump** The file "jump.ftl" handles the execution of "jump" and "call" instructions. Because of the register window scheme, it is necessary to decrement the CWP on procedure calls, and increment it on returns. The *status-equal* module compares the the condition specified in an instruction with the processor's condition codes, and is used for conditional branches.

**5.4.6. Memory** The file "memory.ftl" serves as an interface between RISC and main memory. It declares a small test memory (100 words) and two modules – "get-memory" and "set-memory". Get-memory and set-memory do the sign-extension necessary when accessing bytes and halfwords as signed quantities.

**5.4.7. Other** The file "other.ftl" handles the fourth class of instructions, as well as a pseudo-instruction that allows the FTL description of RISC to print out the contents of registers and memory locations.

**5.4.8. Exec** Exec.ftl defines the program counter, the current window pointer, and a one bit running/halted flag. During simulation, it examines the top two bits of each instruction to determine its type, and then passes the instruction to the correct functional module. Exec.ftl also defines a routine that simulates the top-level instruction fetch finite-state machine, and a reset function that sets the PC to zero, and puts the machine into a fetch-execute loop.

## 5.5. Support Files

The other files included in this section are simple RISC assembler, which can be called from FTL. and a simple absolute loader. The FTL description of RISC and the assembler and loader are included in an appendix.

# CHAPTER 6

# CONCLUSIONS

## 6.1. Results

A new language and associated behavioral-level simulator has been described. FTL allows the description of systems from a very high behavioral level to a fairly low logic level. FTL allows users to describe the sequential and parallel behavior of their systems in a structured fashion. These structuring facilities lead to clearer, more useful specifications and should give designers increased confidence that different implementations of a system will be function in the same way. FTL has been demonstrated on several small pieces of logic, and one large digital system, the RISC microprocessor. The profile and trace facilities in FTL make it possible to analyze the characteristics of such behavioral-level models so that different implementations of the same architecture can be compared and their functional correctness can be verified.

## 6.2. Future Work

More facilities must be added to FTL before it can be a generally useful tool. A dependency analysis package would be a good step in the implementation of an FTL based synthesis system. A program to take an infix input specification language and convert it to FTL would be a useful addition for users that are not comfortable with lisp syntax. The data-type management of FTL1 would have been a great help in writing the RISC description and is likely to be added in the future.

## APPENDIX 1: LISP

One of the most important decisions in a large software project is the choice of a language to implement it in. Two things were apparent from the beginning of the FTL effort. The first was that it was possible, and even likely, that it would be necessary to make one or more major design changes during the implementation; the second was that, because of the experimental nature of the project, it would be very important to be easily able to add new constructs and features to the language. For these reasons, LISP [Winston81] was chosen as the implementation language. LISP is a good language for building prototype software systems because of its excellent debugging facilities, run time interpreter, and human readable linked structures. LISP also has a powerful macro facility which allows implementation decisions in a package (as to what objects are data, vs. what objects are functions) to be hidden from users of that package without any loss of efficiency.

LISP uses a single representation (linked lists) for both programs and data. Thus it is possible for a lisp program to build lisp code which is then evaluated. The FTL compiler works basically in this fashion. It takes an input program in lisp syntax and creates data-structures that represent the structure of the program. Finally, at run-time, the "interpreter" executes the program by traversing this data-structure.

LISP is a easy language to learn because of its simple syntax and semantics. Every programming language must provide a set of basic facilities, among which are control and data-structures. In lisp, all primitive language facilities (control structures, function definitions, variable declarations, and even arithmetic) are provided through functions. The syntax of a function call in lisp is slightly different than in most other programming languages. In lisp that the function name is placed after the left parenthesis, rather than before it. Thus, the form of a function call is an open parenthesis, the function name, the arguments (if any), and a close parenthesis. For example, "(print x)" calls the function "print" on the variable x, and as a result that the value of x is printed. As another example, "(+ a b)" is a call the function "+" with the variables a and b as arguments, returning their sum. Data structures are represented using the same notation. The main two data-types in lisp are *atoms* and *lists*. Numbers and variable names (such as "x" or "y") are the examples of atoms. A list is a dynamic (linked) data structure that is very easy to add and delete from. In lisp, a list is represented as an open parenthesis, followed by the elements of the list (which may be either atoms or lists), followed by a close parenthesis. For example, "(this is a list)" is a list of four elements, and "(this is a (nested) list)" is a list of five elements, one of which is a list. Inside of lisp, a linked list is made up of a collection of two element cells, called *cons* cells for historical reasons. Also for historical reasons, the first element of a cons cell is called its *car* and the second is called its *cdr*. The function **cons** (which is a function of two arguments) creates a new cons cell; the first element of the cell has the value of the first argument, and the second element of the cell having the value of the second one. The function **car** , (which takes a list cell as its argument), returns the first element of that cell, and the function **cdr** returns the second element (the rest of the list). Armed with these simple facts, the user should be able to understand the lisp examples used in this report.

## APPENDIX 2: Detailed Implementation

FTL is implemented in the lisp programming language. Lisp was chosen because of the ease with which the language may be modified and extended, the large number of primitive functions that it provides, and its excellent program development and debugging environment. An introduction to the lisp language is given in an appendix.

### Files

The source code for the ftl system resides in several different files. Currently these files are "ftl.l", "comm.l", "structs.l", "comp.l", "runt.l", and "handlers.l".

| | |
|---|---|
| **ftl.l** | loads the other files. |
| **comm.l** | contains a collection of macros and global variables that are used by the other files. |
| **structs.l** | contains structure declarations and accessor macros for the main data structures used in the ftl system. |
| **comp.l** | contains the functions that form the "compiler" of the ftl. The compiler functions take the ftl program as input and produce the internal tree format as output. |
| **runt.l** | contains the functions for the "run time system". The run time system manages the evaluation queue, provides variable storage and lookup, and handles other support functions. |
| **handlers.l** | contains all message handlers. The normal procedure for adding a new primitive function to ftl is to create any new handler functions it needs, put them into handlers.l, and then enter the name of the primitive and its handlers into the data-structures in comp.l. |

**FTL.l**

The file ftl.l contains calls on the lisp system's "load" function, which causes the files containing the function and variable declarations, and function calls that make up the ftl program, to be loaded. This file is machine dependent because the syntax of file names is different on different machines.

**Comm.l**

The file comm.l contains the declarations of the global variables that are shared across all the phases of ftl. As is common practice in Lisp programming, all global variable names begin and end with an '*' character so that they can be recognized as global by humans reading functions that reference them. The following is a list of the global variables used in ftl:

| | |
|---|---|
| ***show-time*** | is a flag that, when set, causes FTL to show the time-clock any time it changes. |
| ***maxtime*** | gives the maximum value that the argument to the delay function can have. |
| ***show-clash*** | is a flag that causes an error message to be output when a variable is set and then read, read and then set, or set twice in the same instant of simulated time. |
| ***show-root*** | tells the system to print out the name of the root of the tree whenever it compiles a program. |
| ***debug-mode*** | is a general flag that specifies that the compiler is in a debug mode, and should dump internal status information. |
| ***pc-mode*** | puts the compiler is in "implicit delay mode" (see earlier chapters). |
| ***plevel*** | holds what the current level of parallelism is. |

| | |
|---|---|
| ***block-stack*** | is a stack of the blocks outside of the current one. |
| ***level-array*** | is an array (hunk) of the maximum values reached at each level of concurrency. |
| ***wall-clock*** | holds the simulator's idea of time. |
| ***delay-list*** | is a list of all "streams" (processes) that are blocked, waiting on the *wall-clock*. |
| ***final-value*** | holds the final-value of the last computation. |
| ***mode-x*** | says that all delays are one unit long. |

Comm.l also defines some very useful macros.

| | |
|---|---|
| **debug!** | takes a list of names and gives them the "ftl-debug" property. |
| **debug?** | takes a name and a list of expressions and if the name has the ftl-debug property, expands into the list of expressions. Otherwise, it expands into nothing at all. |
| **enum** | that takes a variable number of arguments and gives each one its name as its value. This is used to provide an enumeration facility for such things as node states. |

## Structs.l

Structs.l contains the structure declarations and accessor macros for the trees that are the heart of FTL, and for user declared ftl variables. The program tree is a doubly linked structure composed out of tree-nodes. In FTL2, version 1.0, the tree nodes have 15 fields.

| | |
|---|---|
| **node-op** | stores the constant or the name of the variable that the node represents if the node is a leaf. otherwise, the node represents a function, and node-op field gives the name of the function. |

| | |
|---|---|
| **node-name** | stores the symbol-name of the node. Every node has a unique symbol-name that is of the form <function-name>-<number>. |
| **node-parent-name** | contains the symbol-name of the parent of the node and is used as a pointer back to it. |
| **node-offset** | tells what child number of its parent node the node is. |
| **node-status** | tells whether the node is dormant, ready to fire, waiting for the values of its children, or in some other state. |
| **node-current-caller?** | tells whether the node is the current active function, and thus, whether its environment should be searched when variables are looked up. The exact way this works will be explained in the section on variable management. |
| **node-type** | contains the data-type of the node. |
| **node-value** | contains its value. These fields are used by the data-type management facility of FTL. |
| **node-n-children** | field specifies the number of children of this node, and at any given point in the evaluation of a sub-tree the |
| **node-n-children-left** | gives the number of children that have not propagated their values up to the parent yet. |
| **node-child-name-list** | is a list of the names of the node's children. |
| **node-env** | holds the local environment (list of variable-name, variable-value pairs) for the node. |

**Var**

There are currently five fields in the structure that describes each variable in a user's FTL description.

| | |
|---|---|
| **var-value** | holds the current value of the variable. |
| **var-type** | gives the data-type of the variable. |
| **var-time-read** | holds the value of the clock the last time the variable was read. |
| **var-time-set** | holds the value of the clock the last time the variable was set. |
| **var-scope** | holds the scope of the variable. |

**Accessor Macros**

The last section of structs.l is a collection of macros to access nodes. These macros hide implementation decisions in the choice of node fields. There are four portions of the accessor macros section. The first is a set of macros that return the name of the first, second, third, nth, and last children of a node. The second set returns the node of the first, second, last, etc. children of a node. The third section defines macros for accessing the values of the children of a node. The final section defines macros to access the parent node of a node and the value of the parent node.

**Comp.l**

There are several parts to the file "comp.l". The first is a list of global variables. These variables are used to hold compile-time information that is not local to any single routine.

**Variables**

***Root*** is the root of the tree and holds the top-level environment.

***ftl-user-functions*,**

***ftl-user-macros*, *ftl-user-aliases*** , and ***ftl-user-constants***

hold the names of the functions, macros, aliases, and constants that are currently defined. For now, FTL stores its function definitions in the lisp environment, although this is likely to change in the future. Whenever a function is compiled, a unique symbol name is created for each node in the tree. The name is formed by a prefix that is the routine name, a "-" and a number; for example, the first symbol in the routine "alu" would be "alu-1".

***prefix*** contains the current prefix, and

***prefix-stack*** is a stack of prefixes, so that nested module definitions can be supported.

***symnum*** is the current symbol number, and

***symnum-stack***
is a stack previous symbol numbers.

**Functions**

**reset-global** is the function to reset the entire FTL system. it calls **reset-symbols**, **reset-level-array**, **reset-user-extensions** , and **reset-root** as worker functions.

**Push-prefix** is a function that is called whenever compilation of a new function is started. It's purpose is to save the current prefix and symnum, make its argument the new value of current-prefix and set symnum down to zero.

**my-makesym** is a function that returns a new symbol, suitable to use as a tree-node, formed out of the current prefix and symnum. As a side effect, it increments the symnum.

**make-primitives**

puts the appropriate eval, data, and fire handlers on the property lists of all the lisp primitives callable from FTL.

**make-things** takes two arguments; an association list (a list of (name.value) pairs) and a symbol to be used as a property name. Make-things examines the each element in the association list and gives that car of the element (the name) the cdr of the element (the value) as the value of the given property. The functions "make-magic-compiles", "make-special-compiles", "make-special-evals", "make-special-datas", and "make-special-fires", all use make-things to give the symbols in their argument list the property given by their names.

**ftl-compile** takes a list representing a ftl program and compiles it into internal tree form. It calls

**ftl-compile-prolog**

to set up things for a new compilation,

**ftl-compile-node**

to compile the top-level function, and then

**ftl-compile-epilogue**

to do any final cleaning-up. FTL-compile-node takes four arguments. They are: a symbol to use as the name of the node, the name of the parent of the node, the offset of the current node in its parent's child-list, and the list to be compiled. FTL-compile-node looks at the node it has been passed and checks to see if it is a "special function". If it is, then it calls the compiler function associated with that name. If not, then it checks to see if the node is the name of a previously defined user function, and in that case it calls "ftl-compile-function" to take care of expanding the definition. Finally, if no other option matches, it calls "ftl-compile-node-worker" to do the standard expansion. FTL-compile-node-worker creates a structure for the node, makes a list of names for the node's children and enters it into the structure, sets-up the message handler fields of the node, and finally calls ftl-compile-node on each child.

## Runt.l

There are four major sections in Runt.l. The first portion contains a small number of utility routines to dump trees and print out the contents of nodes. The second section contains routines to manage the evaluation queue. The third section contains routines for management of variables and reporting errors. The fourth section contains some support functions for vectors and bit operations. Currently, the event queue is broken up into two separate parts. The **walk** function manages the evaluation queue which contains nodes scheduled for the current time point. When that queue empties, **step-time** is called. Step-time finds the nodes in the delay-list with the smallest delay, then advances the clock by that amount and moves those nodes from the delay-list to the evaluation queue.

The queue management routines currently support two different versions of explicit delay mode. The first, *variable delay mode*, allows delays to be any length at all but is less time efficient than *unit delay mode* which assumes that all delays are of one time unit.

## Variable Management

Like most modern programming languages, FTL is "block structured" - it allows a variable to be declared inside a block and for that definition of the variable to supercede a definitions of a variable with the same name in an outer block. This idea is commonly called *scoping*. There are two different scoping rules in common use today. The scoping rule used in most current programming languages such as 'C' or Pascal is called *lexical scoping* because the instance of a variable name seen at any point in the program is the one whose declaration is closest to the point in the text where the variable name appears. The rule used in most lisp systems is called *dynamic scoping* because the instance of a variable seen is dependent on the dynamic behavior of the program. When a variable is referenced that is not declared in the function that references it, the environment of the caller of that function is examined, and then the caller of the caller, etc. until a caller is found that has declared a variable of that name. FTL currently uses dynamic scoping but may be changed to use lexical scoping some time in the future.

The routine **lookup** is responsible for looking up variables in the environment. It searches the environment of current node for the variable name. If it is not found there, lookup calls itself recursively on the parent of the node, until it either finds the variable or it reaches a node whose parent is nil, (signaling that the variable is undeclared). Lookup also checks the node to see if the *current caller* flag is set before searching for the variable in it so that formal argument names will not be seen when looking up actual parameter names in a function call, and prints out an error message if ***show-clash*** is t, and it finds that the variable has been set and then read at the current time point. The routine

**node-bound-on?**

returns nil if the variable is not declared, and the node the variable is bound on if it is.

**bound?** calls "node-bound-on?" and returns true (t) if the variable is bound and false (nil) if it is not.

**set-value** is responsible for setting user variables. It is similar in structure to lookup, but gives error messages when *show-clash* is true and a variable is set twice or read and then set at the same point in time.

**Support Functions**

The final section of runt.l contains a collection of functions for manipulating vectors and bit strings.

**Handlers.l**

The file handlers.l contains the functions that are called when messages are sent to nodes in the program tree. These functions manipulate the state of the nodes in tree, and return lists of new nodes to be evaluated. As was mentioned before, there are three types of messages a node can receive, (eval, data, and fire) and therefore three types of message handlers. The message handlers determine the forms that the ftl system understands. handlers.l starts out by including comm.l and structs.l and then defines some useful macros.

**make-stream-list**
makes a stream-list from an atom, and **make-stream-list-worker** makes a stream-list from a list. Currently, stream-lists are just normal lists.

**set-node-status**
and **set-node-value** set the status and value fields of a node. The next section defines some worker macros for the handler functions.

**fire-handler-worker**

is called by all fire handlers to do the general clean-up needed after a node fires. It sets the status of the node back to DORMANT, sets node-n-children-left to node-n-children, and then calls the data-handler for the node.

**data-handler-worker**

decrements the node-n-children-left field of a node and returns **t** when node-n-children drops to zero.

**profiled?** is a macro that returns **t** if the name it is given is being profiled, and **traced?** does the same thing if the name it is given is being traced. The next (and largest) section of handlers.l contains the actual message handlers. Only the most important of these will be described here.

**root-data-handler,**

and **root-fire-handler** are the data and fire handlers for the root. Root-data-handler is used to pass values up to the root, and puts the value being passed into the global variable ***final-value***. Root-fire-handler prints out an error message, since the root node should never be set READY. Because no one ever calls the root (it is at the top of the program tree), there is no root-eval-handler.

outputs from the module, and a list of forms that make up the body of the module. For example,

```
(module adder
        (declare (inputs (integer a b)))
        (+ a b))
```

defines a module called adder that takes two integers as input and returns their sum as its result.

Files containing FTL descriptions may be loaded using the **load** function. the syntax is: (**load** <file-name>). Each form is the file specified by the load function is then read into FTL with the same result as if the forms in the file were typed in from the user's terminal.

Any user defined module may be traced with the **trace** function. The syntax is "(**trace** <module-1> <module-2> ...)". Modules may also be monitored for activity, by using the **profile** function. Entire sessions may be monitored by using **profile-all** which will profile all currently defined modules. If the **prof-report** function is invoked, FTL will print out statistics on all profiled modules.

FTL users can define their own data-types. New data-types are defined by using the **types** function.

The syntax is:
(**types** (<type-name> <type-specifier>) (<type-name> <type-specifier>) ... )

The type-specifier is either the name of a primitive type, or a previously defined user defined data-type. The primitive data-types provided in FTL are **integer, byte, bit,** and **float**

There are three types of blocks provided in FTL. The **serial** form takes zero or more forms as arguments, evaluates them sequentially, and then returns the value of the last one as its value. The **parallel** form takes zero or more forms as its arguments, evaluates them in parallel, and after all of them have finished it

## Appendix 3: Using FTL under UNIX

To use the FTL simulator under UNIX, type "ftl". FTL will respond with the date and version number of the version you are using. At this point the FTL system is ready for input. This level of the system is called "top-level". Any form typed in at top level will be evaluated and the results of evaluation will be printed out on your terminal.

If you type: (+ 1 1)

the system will respond with:

```
(new root = s-1)
2
```

The "new root" message simply says that the compiler has taken your input, converted it into its internal tree form, and that the name of the root of that tree is "s-1". If you wish to re-run a form, without having to re-compile that form, You can use the "run" function, with the name of the root of the tree as its argument.

For example: (run s-1)

will re-run the code generated from "(+ 1 1)" and will print out the result

2.

Variables are declared by using the **declare** function.

The syntax of declare is:

(**declare** (<scope> (<type> <vars>))
For example,

(declare (local (integer a b c)))

declares three integer local variables called "a", "b", and "c". If a variable is referenced or set without being declared, an error message will is printed.

Functions are declared by using the **module** function. The arguments to the module function are the name of the module, the inputs to the module, the returns a vector which contains the values of each of the forms. The "any" form is similar to **parallel** but it returns after any of its arguments returns. Finally, the **block** form evaluates its arguments sequentially, as the **serial** does, but does not advance the time clock when in *pc-mode*.

The control structures currently provided in FTL are **while**, **for**, **forall**, **if**, **cond**, **parallel-cond**, and **which**.

The **while** statement takes two arguments. The first is called the *condition* and the second is called the *object*. The while statement first evaluates its condition. If the condition is true, it then evaluates the form that is the object of the loop, and then goes back and repeats the process by re-evaluating the condition.

The **for** statement takes two arguments. The first is list of the form: (<variable> [from <start-value>] [to <final-value>]), the second is a statement (form) to evaluate as long as the elements of the for are within their limits. The **forall** is exactly the same, except that all iterations of the loop execute in parallel.

The **if** statement can be given either two or three arguments. The first argument is the condition, the second is a form to evaluate if the condition evaluates to **true** and the third (optional) argument is a form to evaluate if the condition returns **false**.

The **cond** statement is like a **case** statement in Pascal. The cond statement takes one or more arguments, called *cond-clauses*. Each cond-clause is a list of the form *(<condition> <action>)*. When a cond statement is evaluated, each clause is examined in sequence. If the condition part of the clause returns true, the action part of that clause is evaluated and the cond is exited. The **parallel-cond** is just the same, except the clauses are examined in parallel, and **all** clauses whose condition evaluates to true have their action's evaluated.

Finally, the **which** statement is like the **switch** statement of 'C': The which statement takes two or more arguments. The first argument is an expression, called the *control*. The remaining arguments are *which-clauses*. Each which-clause is of the form *(<value> <action>)* Where the value field must be a constant. When a **which** statement is evaluated, it first evaluates its control expression. Then, it examines all which-clauses in parallel. If the value part of a which-clause is equal to the value of the control expression, the action portion of that which-clause is evaluated.

```
; Simulate the behavior of the RISC alu.
; Add two n-bit signed operands and the carry, producing a result and flags

; The Most significant bit

(constant *high-bit-num* 31)
(macro sign(x) (get-bit x *high-bit-num*))

; The Condition codes

(declare
  (globals
   (bit Carry Zero Negative Overflow)))

(<- Carry 0)
(<- Zero 0)
(<- Negative 0)
(<- Overflow 0)

; The guts of the alu

(macro do-alu( op x y carry )
  (which? op
           (add
            (+ x y))
           (addc
            (+ x y carry))
           (sub
            (- x y))
           (subc
            (- x y carry))
           (subi
            (- y x))
           (subci
            (- y x carry))
           (and
            (AND x y))
           (or
            (OR x y))
           (xor
            (XOR x y))
           (sll
            (lsh x y))
           (srl
            (rsh x y))))

; The alu as called

(module alu( op a b )
  (declare (inputs (word op a b)))
  (serial
   (declare (locals (word output)))
   (<- output (do-alu op a b Carry))
   (if (= Scc 1)
       (parallel
         (<- Negative
               (not (zero? (sign output))))
         (<- Zero
               (zero? output))
         (<- Overflow
               (which? op
                       (add
                         ; If the sign of the two operands was the same,
                         ; but the sign of the result if different -> overflow!

                          (and
                           (= (sign a) (sign b))
                           (not (= (sign a) (sign output)))))

                         (sub

                          ; For subtraction its exactly the reverse

                          (and
                           (not (= (sign a) (sign b)))
                           (= (sign a) (sign output)))))))
   output))
```

```
; Executive (controller) for the FTL version of the risc simulator.

; The Run/Halt Flag, and current window pointer
(declare
  (globals (word *Running* *Pc*) (CWPTYPE *Cwp*)))

(<- *Pc* 0)
(<- *Cwp* 5)

; Constants for the four classes of instructions

(constant OTHER  (hex 00))
(constant JUMP   (hex 01))
(constant ALU    (hex 02))
(constant LOAD   (hex 03))

;;; THE MAIN LOOP (CONTROLER)
;;; Takes care of fetching the instruction,
;;; fetching the operands if needed
;;; doing the instruction
;;; updating the pc
;;; and going back for more!

(module exec( Inst )
  (declare (inputs (word Inst)))
  (serial
   (declare (local (word Md)))
   (which? Op-high
            (OTHER (other Op-low))

            (JUMP  (jump Op-low Rd))


            ; For alu op's, if Im is set, use a register and an Immediate,
            ;                  else use two registers.

            (ALU   (serial
                    (set-register
                     Rd
                     (which? Im
                             (1 (alu Op-low (get-register-a R1) Off))
                             (0 (alu Op-low
                                     (get-register-a R1)
                                     (get-register-b R2)))))))

            (LOAD (which? Load-or-Store
                          (0 (serial
                               (<- Md (get-memory Op-low index))

                               ; Load instuctions take an extra cycle

                               (delay 1)
                               (set-register Rd Md)))
                          (1 (serial
                               (<- Md (get-register-a Rd))

                               ; Store instructions take an extra cycle

                               (delay 1)
                               (set-memory Op-low index Md))))))

   ; All instructions take a minimum of 1 cycle
   (delay 1)))

; Fetch an instruction and increment the PC
(module fetch()
  (declare (modifies (word Inst *Pc*)))
  (serial
   (declare (local (word Inst)))
   (<- Inst (get-vector *memory* *Pc*))
   (<- *Pc* (1+ *Pc*))
   Inst))

; Reset the machine, and run until it halts

(module reset-risc()
  (declare
   (modifies (word *Pc* *Running*)))
  (serial
   (<- *Pc* 0)
   (<- *Running* t)
   (while *Running*
          (exec (fetch)))))
```

```
;;; Jump conditions

; Enum takes a list of symbols and sets each equal to their position in
; the list
; i.e. ALWAYS=0, NE=1, EQ=2, etc.

(enum ALWAYS NE EQ NCAR CAR NV V LT LE GT GE LOS HI P M)

; Cond is the kind of condition the jump or call is to be made on

; Note the use of "modifies" in the description to tell the compiler we
; plan to change a variable inherited from an outer scope

(module jump( kind Cond )
  (declare
   (inputs
    (word kind)
    (condtype Cond))
   (modifies
    (pctype *Pc*)
    (cwptype *Cwp*)))

  (which? kind
           (jumpx        (if (status-equal? Cond)
                              (<- *Pc* index)))

           (jumpr        (if (status-equal? Cond)
                              (<- *Pc* (+ *Pc* Y))))

           (callx        (serial
                          (set-register Rd *Pc*)
                          (<- *Pc* index)
                          (<- *Cwp* (sub1 *Cwp*))))

           (callr        (serial
                          (set-register Rd *Pc*)
                          (<- *Pc* (+ *Pc* Y))
                          (<- *Cwp* (sub1 *Cwp*))))

           (ret          (serial
                          (<- *Pc* (get-register-a Rd))
                          (<- *Cwp* (add1 *Cwp*))))))

; Compare the condition with the current values of the condition codes

(module status-equal?( Cond )
  (declare
   (inputs (condtype Cond)))
   (which? Cond
           (ALWAYS t)
           (NE (not Zero))
           (EQ Zero)
           (NCAR (not Carry))
           (CAR Carry)
           (NV (not Overflow))
           (V Overflow)
           (LT Negative)
           (LE (or Negative Zero))
           (GT (not Negative))
           (GE (or Zero (not Negative)))
           (LOS (or Zero Carry))
           (HI (and (not Carry) (not Zero)))
           (P (Not Negative))
```

```
        (M  Negative)))
```

A4.6

```
; A simple absolute loader
; Takes in an assembled program as a list of integers
; and loads it into memory at the given starting location

(module load-risc( program start )
  (declare (inputs (list program) (word start)))
  (serial
   (while program
             (serial
              (set-vector *memory* start (first program))
              (<- start (+ 1 start))
              (<- program (rest program))))))
```

```
; Here we define aliases for all the important components of an instruction

(alias Op (get-field Inst 25 31))  ;the opcode
(alias Op-high (get-field Op 4 6)) ;the opcode class (LOAD,JUMP,ALU,OTHER)
(alias Op-low (get-field Op 0 3))  ;the specific instruction in the class
(alias Scc (get-bit Inst 24))       ;the "set condition codes" bit
(alias Rd (get-field Inst 19 23))  ;the destination register number
(alias Rx (get-field Inst 14 18))  ;the index register number
(alias R1 Rx)
(alias Im (get-field Inst 13 13))  ;the immediate addressing flag
(alias R2 (get-field Inst 0 4))     ;the second register for two register insts
(alias Off (get-field Inst 0 12))  ;short offsets or small (13 bits) immediates
(alias Y (get-field Inst 0 19))     ;long (20 bit) immediates or offsets
(alias Load-or-Store (get-bit Op 3)) ;bit that distinguishes loads from stores

; index
; an index register plus a short offset if the immediate bit is set (Rx+Off)
; otherwise, an index register plus a second register (Rx+R1)

(alias index
  (+
   (get-register-a Rx)
   (which? Im
           (0 (get-register-b R1))
           (1 Off))))
```

```
; The main memory for RISC

(declare (globals (word *memory*)))

(<- *memory* (make-vector 100))

; bit-mask produces a mask with the lower N bits trned on.
; NOT is the logical negation, so high24 = !low-8.

(constant HIGH24 (NOT (bit-mask 8)))
(constant HIGH16 (NOT (bit-mask 16)))


(macro get-memory-worker( loc )
  (get-vector *memory* loc))

; Read words, halfwords, or bytes.

(module get-memory( op-low loc )
  (declare (inputs (word op-low loc)))
  (serial
   (declare (locals (word Md Mdl Mdb)))
   (<- Md (get-memory-worker loc))
   (<- Mdl (get-field Md 0 15))
   (<- Mdb (get-field Md 0 7))
   (which? op-low
            (0 Mdb)
            (1 (if (zero? (get-bit Mdb 7))
                   Mdb
                   (OR HIGH24 Mdb)))
            (2 Mdl)
            (3 (if (zero? (get-bit Mdl 15))
                   Mdl
                   (OR HIGH16 Mdl)))
            (4 Md))))

(macro set-memory-worker( loc value )
  (set-vector *memory* loc value))

; Setting is a little more tricky; have to read the old value,
; merge in the new one, and then write the composite value back to memory

(module set-memory( op-low loc value )
  (declare (inputs (word op-low loc value)))
  (serial
   (declare (locals (word Md)))
   (<- Md (get-memory-worker loc))
   (which? op-low
            (8 (set-memory-worker loc value))
            (9 (set-memory-worker loc (OR (AND HIGH16 Md)
                                          (get-field value 0 15))))
            (10 (set-memory-worker loc (OR (AND HIGH24 Md)
                                           (get-field value 0 7)))))))
```

A4.9

A4.10

```
;;; The risc instruction set...

;The LOAD instructions

(constant ldbu    (hex 00)) ; Load byte unsigned
(constant ldbs    (hex 01)) ; Load byte signed
(constant ldsu    (hex 02)) ; Load short unsigned
(constant ldss    (hex 03)) ; Load short signed
(constant ldl     (hex 04)) ; Load long
(constant stb     (hex 08)) ; Store byte
(constant sts     (hex 09)) ; Store short
(constant stl     (hex 0a)) ; Store long

;The JUMP instructions

(constant jumpx   (hex 00)) ; Jump indexed
(constant jumpr   (hex 01)) ; Conditional jump relative
(constant callx   (hex 02)) ; Call indexed
(constant callr   (hex 03)) ; Call relative
(constant ret     (hex 04)) ; Return

;The ALU instructions

(constant ldhi    (hex 00)) ; Load immediate high
(constant and     (hex 01)) ; And
(constant or      (hex 02)) ; Or
(constant xor     (hex 03)) ; Xor
(constant sub     (hex 04)) ; Subtract
(constant subc    (hex 05)) ; Subtract with carry
(constant subi    (hex 06)) ; Subtract interchanged
(constant subci   (hex 07)) ; Subtract interchanged with carry
(constant add     (hex 08)) ; Add
(constant addc    (hex 09)) ; Add with carry
(constant sla     (hex 0c)) ; Shift left arithmetic
(constant sra     (hex 0\d)); Shift right arithmetic
(constant sll     (hex 0\e)); Shift left logical
(constant srl     (hex 0f)) ; Shift right logical

; The OTHER instructions

(constant wait    (hex 00)) ; Wait
(constant trap    (hex 01)) ; Trap
(constant gtlpc   (hex 02)) ; Get last Pc (to restart delayed jump)
(constant gtin    (hex 03)) ; Disable interrupts
(constant reti    (hex 04)) ; Enable interupts
(constant out     (hex 05)) ; Output memory location or register (simulator op)
```

A4.11

```
;;; Handle unusual instructions

(module other( kind )
  (declare (inputs (word kind))
           (imports (word *Pc*))
           (modifies (word *Running*)))
  (which? kind
          (out
           (which? Im

                   ; print out the copy of a rgegister

                   (0 (dumpmsg "R" R1 "=" (get-register-a R1)))

                   ; print out a memory location

                   (1 (serial
                        (declare (local (word addr)))
                        (<- addr index)
                        (dumpmsg "Memory[" addr "] = "
                                 (get-memory-worker addr))))))
          ; Halt the machine

          (wait (serial
                  (<- *Running* nil)
                  (dumpmsg "Halt at Pc=" *Pc*)))))
```

```
; FTL simulation of register file: allows reading and writing registers
; Using Cwp.

; Currently in the actual chip, there are 18 globals, [addresses 0-17],
; 4 overlapped with the next higher window (the previous caller) [18-21]
; 6 locals [22-27], and 4 shared with the next procedure to be called [28-31].
; When a call is made, the Cwp is decremented and when it returns it is
; incremented

; The register file itself
(declare
  (globals (vector *regfile*)))

; Number of registers, windows, globals, etc.
(constant *n-regs* 78)
(constant *n-windows* 6)
(constant *n-globals* 18)
(constant *n-locals* 6)
(constant *n-shared* 4)
(constant *n-in-window* 10)
(constant *n-non-global* 60)

; Make the register file

(<- *regfile* (make-vector *n-regs*))

; Macro for converting from a logical register number to a physical one
; as a function of the current window-pointer

(macro logical-to-physical(regnum)
  (cond
   ((< regnum *n-globals*)
    regnum
    (t (serial
         (declare (local (word rnum)))
         (<- rnum
             (+ regnum
                (* *Cwp* *n-in-window*)))
         (if (< rnum *n-regs*) rnum (- rnum *n-non-global*)))))))

; The worker macro for reading registers

(macro get-register-macro( regnum )
  (get-vector *regfile*
              (logical-to-physical regnum)))

; Bus A
(module get-register-a( regnum )
  (declare (inputs (word regnum)))
  (get-register-macro regnum))

; Bus B
(module get-register-b( regnum )
  (declare (inputs (word regnum)))
  (get-register-macro regnum))

; Bus C
(module set-register( regnum value )
  (declare (inputs (word regnum value)))
  (set-vector *regfile* (logical-to-physical regnum) value))
```

```
(load /devels/jtd/risc/opcodes.ftl)
(load /devels/jtd/risc/regfile.ftl)
(load /devels/jtd/risc/macros.ftl)
(load /devels/jtd/risc/alu.ftl)
(load /devels/jtd/risc/jump.ftl)
(load /devels/jtd/risc/memory.ftl)
(load /devels/jtd/risc/other.ftl)
(load /devels/jtd/risc/exec.ftl)
(lisp (load '/devels/jtd/asm.l))
(load /devels/jtd/risc/loader.ftl)
(load /devels/jtd/risc/testrisc.ftl)
```

```
(declare (globals (vector program) (word Inst)))

;       This program puts 5 into register 1
;loop   Then it outputs the value in register 1.
;       decrements register 1,
;       and if it's greater than 0, go back to loop

(<- program
  (lisp
   (asm '((add 0 1 0 1 5)
          (out 0 1 1 0 0)
          (sub 1 1 1 1 1)
          (jumpx 0 1 0 1 1)
          (wait 0 0 0 0 0)))))

(load-risc program 0)

; The value of the file is the value of the last thing evaluated --
; in this case it's the string "test program-loaded"


"test program loaded"
```

```
(setq program
  (lisp
   (asm '((add 1 19 0 1 10)
          (out 0 1 19 0 0)
          (jumpx 0 2 0 1 4)
          (callx 0 18 0 1 5)
          (wait 0 0 0 0 0)
          (out 0 1 29 0 0)
          (add 1 29 0 1 0)
          (ret 0 28 0 0 0)
          (wait 0 0 0 0 0)))))
```

A4.16

```
(setq program
   (lisp
    (asm '((add 0 1 0 1 5)
           (out 0 1 1 0 0)
           (sub 1 1 1 1 1)
           (jumpx 0 1 0 1 1)
           (wait 0 0 0 0 0)))))

(risc-load program 0)
```

A4.17

## APPENDIX 5: SOURCE PROGRAM LISTING FOR FTL

To obtain copies of this Appendix, contact:

Pamela Bostelmann
Industrial Liaison Program
499A Cory Hall
University of California
Berkeley, CA 94720

tel: (415) 642-8312

## REFERENCES

[Nagel75],
L.N. Nagel,
"SPICE2: A Computer Program to Simulate Semiconductor Circuits",
University of California Electronics Research Laboratory,
Memo ERL-M520, May 1975

[Keller82]
K. Keller and A. R. Newton,
"KIC2: A Low Cost Interactive Editor for Integrated Circuit Design",
Digest of Papers
IEEE Compcon Conference, 1982

[Ousterhout81]
J. Ousterhout,
"Caesar: An Interactive Editor for VLSI Layouts",
VLSI Design, Vol. 2, No. 4, Fourth Quarter 1981.

[Calma],
Commercial products available from
the Calma Corporation.

[Applicon],
Commercial products available from
the Applicon corporation.

[Deutsch76]
D. N. Deutsch, G. Persky, and D. Schweikert,
"LTX - A System For the Directed Automatic Design of LSI Circuits",
Proceedings,
IEEE/ACM Design Automation Conference, 1976

[Chawla75],
B.R. Chawla, H.K. Gummel, and P. Kozak,
"MOTIS - An MOS Timing Simulator",
IEEE Transactions of Circuits and Systems,
Volume 22, No. 13, December 1975, pp 901-909.

[Case75],
G. R. Case,
"SALOGS - A CDC 6600 Program to Simulate Digital Logic Networks,
Vol. 1 - User's Manual",
Sandia Laboratory Report No. SAND 74-0441, 1975

[Barbacci77],
M. Barbacci,
"The ISPL Language",
Carnegie Mellon University,
Department of Computer Science, 1977.

[Newton78].
A.R. Newton,
"The Simulation of Large-Scale Integrated Circuits",
Memo No. UCB/ERL-M78/52, Electronics Research Laboratory,
University of California, Berkeley, July 1978

[De Man81 ].
H. De Man, G. Arnout, and P. Reynaert,
"Mixed-mode Circuit Simulation Techniques and Their Implementation in DIANA"
Computer Design Aids for VLSI Circuits,
Nato Advanced Study Institutes Series,
Sijthoff & Noordhoff, 1981

[Patterson81a].
D. A. Patterson and C. H. Sequin,
"A VLSI RISC",
IEEE Computer Magazine,
September 1982

[Breuer82].
M. A. Breuer,
"A survey of the state-of-the-art of design automation*
Proceedings, 1982 Design Automation Conference,
IEEE

[Duley68]
J. R. Duley and D. L. Dietmeyer,
"A Digital System Design Language (DDL)",
IEEE Transactions on Computers, Vol C, No 17, September 1968,
pp. 850-861

[Hill80].
D. D. Hill and W. M. Van Cleemput,
"SABLE: Multi-Level Simulation for Hierarchal Design",
Proceedings of the 1980 IEEE International Symposium on Circuits and Systems,
pp. 431-434,
IEEE, 1980

[Johannsen79]
D. J. Johannsen,
"Bristle Blocks: a silicon compiler",
Proceedings of the 16th Design Automation Conference,
pp. 310-313
IEEE, 1979

[Aho77].
A.V. Aho, and J.D. Ullman,
"Principles of Compiler Design",
Addison Wesley, 1977

[Ackerman79]
W. B. Ackerman and J. B. Dennis
"Val – A Value Oriented Algorithmic Language: Preliminary Reference

Manual", MIT Laboratory for Computer Science TR-218,
MIT, 1979.

[Shaw74]
A. C. Shaw,
"The Logical Design of Operating Systems"
Prentice-Hall, 1974

[Brinch Hansen76].
P. Brinch Hansen,
"The Programming Language Concurrent Pascal"
Springer Verlag, 1976

[Cory79].
W.E. Cory, J.R. Duley and W.M. Van Cleemput,
"An Introduction to the DDL-P language",
Tech. Report No. 164,
Stanford University,
March 1979

[Hill80].
D. D. Hill,
"Language and Environment for Multi-Level Simulation",
Ph.D. Thesis,
Stanford University, 1980

[Hill73]
F.J. Hill and G.R. Peterson,
"Digital Systems: Hardware Organization and Design",
John Wiley and Sons, 1973

[Iverson62]
K.E. Iverson,
"A Programming Language",
John Wiley and Sons, 1962

[Barlow79].
"The Pro's and Con's of APL"
Computer Science Department publication
Worcester Polytechinc Institute, 1979

[Foderaro82]
J. K. Foderaro and K. S. Van Dyke,
"The Slang Reference Manual"
Internal memorandum,
Computer Science Division,
Electrical Engineering and Computer Science Department,
U. C. Berkeley, 1982

[Kang81].
"Automatic Generation of PLA Based Systems",
S.K. Kang
Ph.D. Thesis,

Stanford University, 1981

[Kuck78].
D. J. Kuck,
"The Structure of Computers and Computations, Vol. 1"
John Wiley and Sons, 1978

[Dennis80]
J.B. Dennis, "Data-Flow Supercomputers",
IEEE Computer, Vol 13, No.11, Nov 1980, pp 48-56

[Newton78],
A.R. Newton, "The Simulation Of Large Scale Integrated Circuits"
University of California, Berkeley, Electronics Research Laboratory
Memorandum No. M78/52, July 1978

[Seitz80].
Charles L. Seitz, "System Timing",
Chapter 7,
C. Mead and L. Conway, "Introduction to VLSI Systems",
Addison-Wesley, 1980

[Bottorff82]
P. S. Bottorff,
"Computer Aids to Testing - An Overview",
In Computer Design Aids for VLSI Circuits,
see above

[Ackerman82],
W.B. Ackerman,
"Data Flow Languages"
IEEE Computer,
Volume 15, Number 2,
February 1982

[Arvind78],
Arvind, K.P. Gostelow, and W. Plouffe,
"An data-flow Programming Language and Computing Machine,"
Department of Information and Computer Science, Technical Report TR-113,
University of California, Irvine, Feb. 1978.

[Morris81]
J. H. Morris,
"Real programming in functional languages",
Xerox Palo Alto Research Center Technical Report CSL-81-11,
Xerox, July 1981

[Ayres79]
R. Ayres,
"A Language Translator and Sample Language",
Ph.D. Thesis,
California Institute of Technology,
1979

[Xerox80]
Special Issue on Smalltalk,
BYTE magazine, July 1980

[Foderaro81],
J. K. Foderaro and K. L. Sklower,
"The Franz lisp Manual",
University of California at Berkeley, 1981

[Deutsch80],
J. Deutsch,
"Computer Architecture",
Major Qualifying Project,
Worcester Polytechnic Institute,
May 1980

[Jensen79]
J. Jensen and N. Wirth,
"Revised Report on the Programming Lauguage Pascal",
Springer-Verlag, 1976

[Steele78]
G. L. Steele and G. J. Sussman,
"The art of the interpreter",
AI memo 321,
MIT, 1978

[Patterson81a].
D. A. Patterson and C. H. Sequin,
"A VLSI RISC",
EECS Department Computer Science Division Memorandum,
University of California at Berkeley, 1981

[Roth79],
C. H. Roth,
"Fundamentals of Logic Design",
West, 1979

[Dijkstra76].
E. W. Dijkstra,
"A Discipline of Programming",
Prentice-Hall, 1976

[Radin82].
G. Radin,
"The 801 Minicomputer"
pp 31-47,
Symposium on Architectural Support for Programming Languages
and Operating Systems,
ACM, 1982

[Patterson81b].
D. A. Patterson,

"A RISCy approach to computer design"
pp 8-14, Digest of Papers,
IEEE COMPCON,
Spring 1982
IEEE Computer Society Press

[Mead80].
C. Mead and L. Conway,
"Introduction to VLSI systems",
Addison Wesley, 1980

[Lampson81],
B. Lampson,
"The Dorado Personal Computer",
Xerox Palo Alto Research Technical Report,
Xerox, 1979

[Winston81]
P.H. Winston and B.K.P. Horn,
"LISP",
Addison-Wesley, 1981