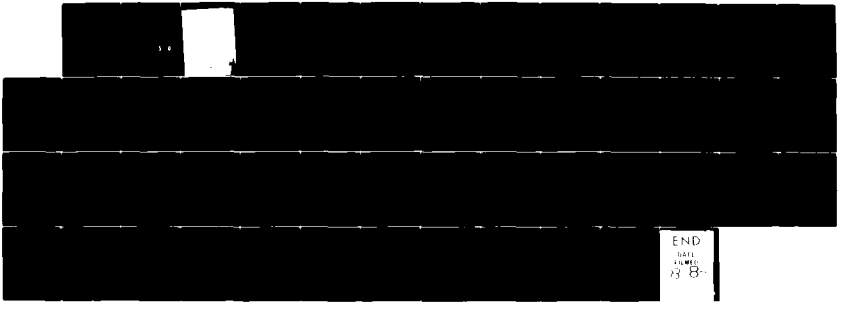
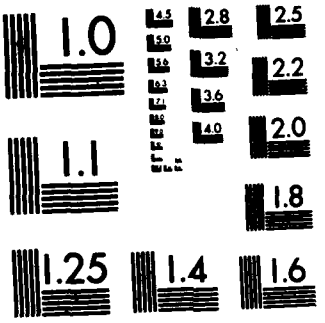
AD-A195 064 NORTHWEST LABORATORY FOR INTEGRATED SYSTEMS UNIVERSITY OF WASHINGTON(U) WASHINGTON UNIV SEATTLE NORTHWEST LAB FOR INTEGRATED SYSTEMS L SNYDER APR 88 TR-88-24-06 1/1

UNCLASSIFIED MDA903-85-K-0072 F/G 9/1 NL



END
DATE FILMED
7 8



unclassified

(4)

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 88-24-06	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Northwest Laboratory for Integrated Systems University of Washington Semiannual Technical Report No. 6		5. TYPE OF REPORT & PERIOD COVERED Technical, Interim 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Northwest Laboratory for Integrated Systems		8. CONTRACT OR GRANT NUMBER(s) MDA903-85-K-0072 ARPA-456-3/2 Code 5D30
9. PERFORMING ORGANIZATION NAME AND ADDRESS Northwest Laboratory for Integrated Systems University of Washington, Dept. of Computer Seattle, WA 98195 Science, FR-35		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS DARPA-ISTO 1400 Wilson Boulevard Arlington, VA 22209		12. REPORT DATE April, 1988 13. NUMBER OF PAGES 49
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) ONR University of Washington 315 University District Building 1107 NE 45th St., JD-16, Seattle, WA 98195		15. SECURITY CLASS. (of this report) unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this report is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

VLSI Design Generators, VLSI architectures and CAD, WIN, Network C, WireLisp, APEX, Circuit Parallelism, Planar Topology

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This document reports on the research activities of the Northwest Laboratory for Integrated Systems, for the period of November 17, 1987 to April 11, 1988, under the sponsorship on the Defense Advanced Research Projects Agency, under contract number MDA903-85-K-0072, program code number 5D30.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Contents

1 Overview of Activities

The capture of design expertise inherent in many commonly used circuits is the primary goal of design generators. Over the last three years we have taken several approaches to this problem. The first was to embed the circuit design in a common programming language such as "C". Although extremely powerful, this approach suffers from the fact that essential design information is obscured by the syntax of the language. An alternative we have recently investigated is a special purpose notation which allows the succinct description of both geometric and electrical network information. This notation ("WIN" for Washington Intermediate Notation) has been successfully applied to numerous circuit families and will soon be released to the community.

Network C is a multilevel simulation system that supports event-driven modeling (at functional, gate and switch levels) as well as continuous analog modeling. The event-driven capability has been used successfully to model both the APEX architecture as well as its software/hardware environment. Ongoing experiments with a model of MOS circuits that combines analog simulation in a stage with event-driven simulation between stages shows considerable promise. This capability seems useful in simulating small instances of circuit families whose analog character prevents effective use of switch level models, but which are too large for the continuous analog modeling of SPICE.

Several efforts address the problems of circuit specification and synthesis. One approach is through the use of functional programming languages. Besides their obvious use as a simulation language, these languages can be used to map the planar topology of a circuit to a layout. Another effort is *WireLisp* - a dialect of Lisp that facilitates the specification of circuit structure. A graphical frontend for *WireLisp* allows the mixing of symbols representing circuit elements and Lisp expressions.

Recent architectural experiments have included APEX I and II. These chip designs are two different implementations of an algorithm for generating parametric curves and surfaces. Together they constitute an interesting contrast between single and multiprocessor architectures, as well as a study of alternative design methodologies. The designs of both chips have recently been completed and are currently being fabricated.

A topic ancillary to our focus on CAD tools is that of circuit parallelism. We have performed numerous experiments on existing CMOS VLSI designs to measure the amount of simultaneous activity. The somewhat surprising result is the small amount of parallelism that these circuits exhibit.

2 WIN Descriptions of Circuit Families

(Larry Snyder, Jean-Loup Baer, Larry McMurchie, Wayne Winder, Rudolf Nottrott)

Much progress has been made over the last 6 months on the declarative notation we are using as an input language for the design generator project. The notation, which we have dubbed "WIN" (Washington Intermediate Notation), specifies the geometric and network structures of an entire family of circuits. Backend generator programs analyze WIN code and produce outputs that can be used by CAD tools. In particular the network generator *netgen* operates on WIN code to produce netlists for the simulation system Network C. The layout generator *laygen* acts as an assembler of magic-formatted leaf cells. (WIN and the accompanying generator programs were described in the November 1987 DARPA Report).

During the last six months major progress has been made on three fronts: code integration, user interface integration, and feature addition.

With respect to code integration, the generators, which previously were separate programs sharing only the notation parser, have been thoroughly integrated, with more than 75% of the code of each program being common to both. This change is transparent to the user, and has markedly increased the ease with which features can be added and problems fixed.

The user interface has been integrated to the point that options and directory searches are done in a common fashion across representations. The main unification, notation structure meaning, is now common to all the generators. Major changes were made to *netgen* to achieve this. Most important, the *netgen* output was previously hierarchical, but is now flattened. Hierarchy is still achievable by multiple notation instances where each notation instance describes one structure. Any structure made by a generator may be used as input to another notation instance. The most important achievement from this integration is that the same notation may be used to generate both layout and network circuit representations.

Many features have been added to the notation to make *netgen* and *laygen* useful tools for the designer:

1. Signal names which bind leaf cells now need appear only in the leaf cell references. This is a by-product of flattening the *netgen* hierarchy. The result is to remove most signal references from statement index lists and make a much less cluttered notation. Signal references in *laygen* now allow the addition of labels to layout to further the compatibility between layout and network output.

2. Parameter arrays have been added, allowing programmable structures, such as PLA's to be described by the notation.

3. Input, output and vector declarations may now be multi- dimensional. This allows easy description of structures using two (or more) dimensions of distinct nodes.

4. A Fill feature has been added to the notation. This allows parameterized expansion of leaf cells in the layout generator. It acts as a 'no operation' in the network generator.

Having made these improvements as well as performed considerable testing, we intend to include the network and layout generators in our next tools release. Also to be included is a comprehensive WIN reference manual.

3 Continued Development of Network C

(Bill Beckett)

Network C (*nc*) is a multilevel simulation system designed for constructing and simulating models of VLSI circuits and systems. The input language, a superset of C, supports a range of modeling capabilities including solution of Kirchoff equations at the MOS and analog levels and discrete event functional simulation at the system level (Details of nc's algorithms as well as experimental results have been described in DARPA Reports from March 1986 through November 1987).

Four basic changes have been implemented. First, a new node classification scheme was developed which allows precise determination of whether a node is an input or an output to a model. The inability to do this was causing the failure of some *nc* programs.

Second, the approach to representing node equations for both MOS and analog subnetworks has been completely redesigned. Now, rather than keeping separate data structures for the systems of equations, residual currents are kept directly in the nodes. The current-based models were changed to add their respective branch currents to these residuals. This allows very general systems of equations to be handled and allows the construction of continuous and MOS models with up to 20 terminals (the previous limit was 4).

Third, the use of single precision floating point variables has been completely eliminated throughout the system. All floating point quantities are now double.

Finally, a new scheme for circuit wide initialization was implemented. This scheme uses the activity wavefront to search for device instances that have never been invoked. Any of these that are found are invoked whether they have new inputs or not.

Aside from these changes there have been a number of smaller corrections and bug fixes, some work on improving and rationalizing the diagnostics, and some progress on the manual.

4 WireLisp – Drawing Netlist Programs

(Carl Ebeling, Zhanbing Wu)

We have completed the language definition for *WireLisp*, and the implementation of *WireLisp* in the T dialect of Lisp is almost complete. We also have the graphics frontend to *WireLisp* implemented and plan to use this facility to specify both high-level architecture descriptions for *nc* simulation and low-level circuit descriptions for *nc* and *rnl* simulation. We are currently evaluating the use of the graphics frontend and *WireLisp* to specify layout topology as well as circuit structure. This information would then be used to drive a backend program based on Coordinate Free LAP for composing circuit layout.

5 Layouts from Functional Language Specifications

(Martine Schlag, Simon Kahan)

Our work on the use of a high level functional language for the specification of integrated circuits continues and has recently focused on the problem of mapping the planar topology of a circuit to a layout. The challenge is now to exploit the information inherent in the assembly history provided by the functional language specification to determine the compaction strategy. To this end, we are looking at algorithms for structured two-dimensional compaction. That is, given an assembly history of the layout, are there efficient methods and heuristics for performing two-dimensional compaction efficiently. For example, given an array composed of a single cell, is it possible to compact the cell to obtain a certain area, aspect ratio or shape for the entire array ? This problem can be formulated as a tiling problem and we are looking for algorithms which will compact a cell to obtain a tile with a given width/height/skew.

6 APEX: An Architecture for Drawing Parametric Curves and Surfaces

(Tony DeRose, Mary Bailey, Bill Barnard, Robert Cypher, Carl Ebeling, Smaragda Konstantinidou, Larry McMurchie, Bill Yost)

During the last six months two major chip designs have been completed and are currently in fabrication. Both chips are intended to facilitate efficient drawing of curves and surfaces from sets of control points (see the November 1987 DARPA report).

APEX I employs multiple processing elements in a triangular data-flow architecture. The chip contains about 65,000 transistors and is being fabricated in an 84-pin standard frame at 2 microns. We estimate a clock rate of about 10Mhz. The design has been extensively simulated using RNL and the layout validated using Gemini.

APEX II performs the same computation in a more flexible way that allows the generation of a wider family of curve types of higher degree at the cost of lower performance. APEX II contains approximately 60000 devices and is also being fabricated in an 84-pin frame at 2 microns.

7 CAD Tools for High-Level Specification and Synthesis

(Gaetano Borriello)

The field of computer-aided design (CAD) of integrated circuits has made major strides in the last few years. Tools are now available that can transform a description of a design in a hardware description language (HDL) to the mask geometry needed to fabricate the chip. With these aids, the design cycle has been radically shortened. Previously time consuming tasks, such as mask layout, are now performed automatically by silicon assemblers and compilers. This has changed the bottlenecks of the chip design cycle. The new bottlenecks are 1) the system integration of completed designs and 2) high level system specification.

The integration of a completed custom chip into a computer system requires the design of glue logic that makes the two interfaces compatible (i.e., permits communication between the chip and the rest of the system). A concise, graphical specification method based on formalized timing diagrams has been developed. An editor has

been implemented to support the specification methodology. The diagrams can be used as a foundation for a entirely new set of CAD tools that deal with circuit interface issues from synthesis to simulation and testing.

One such tool is a glue logic synthesizer that can generate the logic circuitry given timing diagram specifications of the chip and system bus interfaces. New synthesis and optimization methods allow equal treatment of synchronous and asynchronous interfaces with results comparable in both size and performance to designs generated by experienced human designers.

A fundamental problem in the area of high level synthesis is that it takes so long for a designer to generate a complete, detailed description that there is not enough time to investigate design alternatives. Currently under development is a graphical specification method that can be used a front-end to analysis tools that provide the designer performance information early in the design cycle, while there is still time to evaluate alternatives. The graphical specification can then serve as a backbone for generating the complete description of the system.

These two phases of the design cycle have been the most overlooked by the CAD community. However, they are now beginning to dominate the time needed to bring a circuit design from idea to reality. A logical aim is to reduce this time, and with it the entire design cycle, so that ideas can be more quickly prototyped and evaluated.

8 A Model for Architectural Comparisons

(Larry Snyder, Sam Ho)

Recently, architectures for sequential computers have become a topic of much discussion and controversy. At the center of this storm is the Reduced Instruction Set Computer, or RISC, first described at Berkeley in 1980, and its counterpart the Complex Instruction Set Computer, or CISC. Applications favoring RISC over CISC abound, as well as situations where the opposite occurs. Confusing the issues further, the RISC I chip from Berkeley contained an essentially unrelated piece of hardware, that of multiple overlapping register sets. The early papers on RISC often combined the effects of the register set and the instruction set with little regard for their relationship, which was tenuous, at best.

In our analysis of the instruction set complexity, we start with a computer, defined by its functional units, such as ALU, shifter, and registers, and its control, microcode or hardwired controls. We choose a calculation, such as a matrix product or a text

formatter, and decompose it into basic actions, which are arithmetic operations and their relatives. To actually implement this calculation, we will need to generate some necessary overhead actions, such as fetches and decodes. Finally, the functional units determine the cost of each action in time units called cycles. The total cost of the calculation is the sum of the number of cycles needed. The lower this number is, the faster the computer operates.

This model is an attempt to provide a common quantitative basis for a discussion of reduced vs complex instruction sets and other architectural questions. A complete description of the model and the experiments is given in Appendix III.

9 Investigations Into Circuit Parallelism

(Larry Snyder, Mary Bailey)

We have been investigating the question: How much parallel "activity" is there on CMOS VLSI Chips? Most researchers have assumed that large chips have many transistors firing simultaneously, but because no one can measure the activity, no one can be sure. The first problem, then, is how to determine the amount of switching on a chip. Simulation is the obvious answer, but the matter is more complicated. Transistor switching is a continuous, analog activity, but parallelism, as the term is generally used in computer science, seems to be more digital, based on the concept of a "step". Also, to further complicate the use of simulation for determining on-chip parallelism, the detailed SPICE simulations that engineers generally trust are computationally infeasible for chips with more than a few hundred devices.

We have developed a methodology, using Terman's linear-level simulator RNL, for empirically determining the amount of parallelism on a CMOS VLSI chip. We also have a "calibration" study showing to what extent RNL can serve as a surrogate for SPICE, and a study of the impact that the simulation step size has on parallelism. We applied this methodology to six CMOS chips developed at the UW. The surprising results and our conclusions are given in Appendix II.

APPENDIX I

The Planar Topology of Functional Programs

Martine Schlag

Department of Computer Science, FR-35
University of Washington
Seattle, Washington 98195

Abstract

The use of the applicative language (FP) in VLSI design has been advocated, because it provides not only the structure of a circuit, but the planar organization of its components and their interconnections. In this paper, the level of geometric detail implied by the functional programming style is formalized. The notion of 'planar topology' of an integrated circuit layout is defined and shown to be the appropriate level of geometric information to infer from an FP specification of a circuit. This definition provides a normal form for the representation of the planar topology of a layout which is not only unique (modulo local operations), but is optimal over all representations of the same planar topology with respect to topological cost measures. This normal form is exploited to improve the wiring of the layouts; it is realized by the application of transformations to the FP specification. The specification of a carry-save array multiplier is used as an example to illustrate how this optimization reduces the effort required to specify an integrated circuit.

1. Introduction

The current complexity of VLSI design can only be managed by the application of CAD tools at all levels of the design process. In order to be effective, these tools must be flexible enough to be tailored to any specific design. In bottom-up design, the designer is faced with the dilemma of using tools which require the complete specification of the placement of modules and the interconnections between them [Oust81, Oust84], or relinquishing control over the layout to a tool's algorithm [Rive82]. Top-down synthesis tools are capable of generating layouts from high-level specifications. Examples would include various register-transfer silicon compilers that have been proposed and built [Joha79, Sisk82]. Generally, neither type of tool provides any estimate of the area or delays of the circuit during the synthesis process. That is, designers do not know the effects of their decisions on the performance until the design is complete.

A compromise between the complete specification of an integrated circuit and the synthesis of the layout by a tool, is to use a specification which provides an intermediate level of geometric detail of the layout. Several researchers [Shee84, Pate85] have advocated the use of the applicative language (FP) [Back78] in VLSI design, precisely because it provides not only the structure of a circuit, but the planar organization of the components and their interconnections. By exploiting the geometric information embedded in FP specifications, an environment in which designers can rapidly explore various

alternative designs for their algorithms, can be provided [Pate85]. An algorithm can be specified at any arbitrary level of abstraction and the system rapidly evaluates performance parameters (e.g. speed, area, etc.) so that designers can make informed decisions during the synthesis process. Using an applicative language to specify a hardware algorithm, ties together the specification of the algorithm, the synthesis of the circuit and the evaluation of the implementation.

In this paper, the level of geometric detail implied by the functional programming style is formalized. The notion of 'planar topology' of an integrated circuit layout is defined and shown to be the appropriate level of geometric information to infer from an FP specification of a circuit. This definition provides a normal form for the representation of the planar topology of a layout which is unique (modulo local operations), and optimal over all representations of the same planar topology with respect to topological cost measures. This normal form is realized by the application of transformations to the FP specifications.

2. VLSI Design in FP

FP as proposed in [Back78] is not suitable for specifying sequential circuits due to it's lack of state. Meshkinpour [Mesh85] and Sheeran's μFP [Shee84] extend Backus' FP language with operators to handle sequential circuits. A different approach is taken in vFP [Pate85]. vFP extends the FP language proposed by Backus with additional functional forms and primitives. In contrast to μFP [Shee84], which extends FP's semantics to operate on streams, the semantics of vFP are the same as those of FP when it is used to specify algorithms. Sequential circuits are described in vFP by folding designated functional forms to implement them in time (as sequential circuits) rather than in space (as combinational circuits).

A program in vFP (as in FP) is a function that maps objects into objects. Objects are either atomic (numbers or strings) or sequences of objects. The distinguished atom $\perp$ denotes an undefined value. By definition, any sequence which contains $\perp$ as an element is itself undefined and thus equal to $\perp$. The primitive functions of vFP consist of

arithmetic functions,	+ : (1,5) → 6	* : (3,2) → 6
logical functions,	nandg : (1,0) → 1	org : (0,0) → 0
predicates,	atom : (1,2) → F	= : (3,3) → T
selector functions,	3 : (2,(4,5),6,(8,(9,10))) → 6	last : (1,4,6) → 6

and structure modifying functions,

trans : ((1,2,3),(4,5,6)) → ((1,4),(2,5),(3,6)) apndl : (1,(2,3,4)) → (1,2,3,4)

distl : (x, (a,b,c)) → ((x,a),(x,b),(x,c)) distr : ((a,b,c),x) → ((a,x),(b,x),(c,x)).

Functional forms are used to combine primitive functions into more complex functions.

compose	(f @ g) : x → f : (g : x)	**apply to all**	&f : $(x_1,\ldots,x_n)$ → (f:x_1,...,f:x_n)
construct	[f,g,h] : x → (f:x, g:x, h:x)	**right insert**	!f : $(x_1,\ldots,x_n)$ → f:(x_1, !f:($x_2,\ldots,x_n$))
constant	%k : x → k if x is not $\perp$	**seq**	seq(f) : $(x_1,\ldots,x_n)$ → $(y_1,x_1,\ldots,x_{n-1})$
			where $(y_2,x_2,\ldots,x_{n-1})$ = seq(f) : $(x_2,\ldots,x_n)$
			and (y_1,x_1) = f:(x_1,y_2)

Owing to the natural specification of parallelism in FP-like languages, they are suited to describing parallel hardware algorithms. These specifications (or programs) are executable. In addition, since such programs are *referentially transparent*, it is possible to have an algebra of programs which may be used to reason about their behavior. Either method may be used in conjunction with one another to convince the designer that the program implements the envisioned algorithm. Specifications can also be executed with a symbolic input to extract the topological structure of the algorithm. Therefore, there is a direct relationship between the structure of an algorithm written in FP and the planar topology of its layout.

3. The Level of Geometry Afforded by FP

Each functional form of an FP function implies the planar organization of the circuitry of its sub-functions. More formally, a function is a sub-graph with an input arc labeled with its input object and an output arc labeled correspondingly with its output object. Each functional form combines the sub-graphs of its sub-functions to form a new sub-graph. This may entail adding nodes which distribute and collect inputs and outputs as dictated by the particular functional form. The planar embedding of the graph of a functional form is obtained from the planar embeddings of the graphs corresponding to its the sub-functions. The graph and its embedding is obtained by the symbolic execution of the FP program. The graph on the left in Figure 1 is extracted in this fashion from its FP program.

To obtain a circuit from this planar graph, circuit elements (or sub-circuits) are substituted for the nodes and the arcs connecting the nodes are expanded. Each atom (null is not an atom) in an object will be given its own wire in the expansion of an arc. Since only one object is passed between functions, the organization of connections around function boundaries is straightforward. Each arc along which an object passed is expanded into a group of wires, one wire for each atom in the object. The ordering of these wires is inherited from the list structure of the object. The graph on the right in Figure 1 is obtained from the graph on the left by expanding its arcs. In this fashion, a planar embedding of the structure of the application of an FP program is obtained. FP programs yield planar graphs because crossings and branchings (fanout) are hidden inside nodes.

To obtain a layout from this planar graph the appropriate circuit structures (components and wiring) are be substituted for its nodes. In top-down design the exact dimensions of these structures may not be known until the final stages in the synthesis of the design. Since the placement of a structure is dependent on its dimensions as well as those of neighboring structures, only the planar topology of the FP program remains fixed as the design is refined during its synthesis. Unfortunately, interpreting the wiring implied by this graph by directly substituting the wiring required by its nodes results in either inefficient layouts or complex specifications since detailed attention must be given to the exact wiring patterns being generated. This is further complicated by uncertainty about the dimensions of components which can affect the wiring. A more flexible interpretation of the geometry implied by an FP program is the class of layouts which can be obtained from the literal interpretation by topological transformations of the plane, reshaping and stretching of the wire nets without picking them up out of the plane. Only the global routing and planar organization of the components is fixed. In the following sections, this notion of the planar topology of a circuit layout is explored: it is formally defined and its representation is shown to have a normal form. The realization of this normal form for the planar graphs

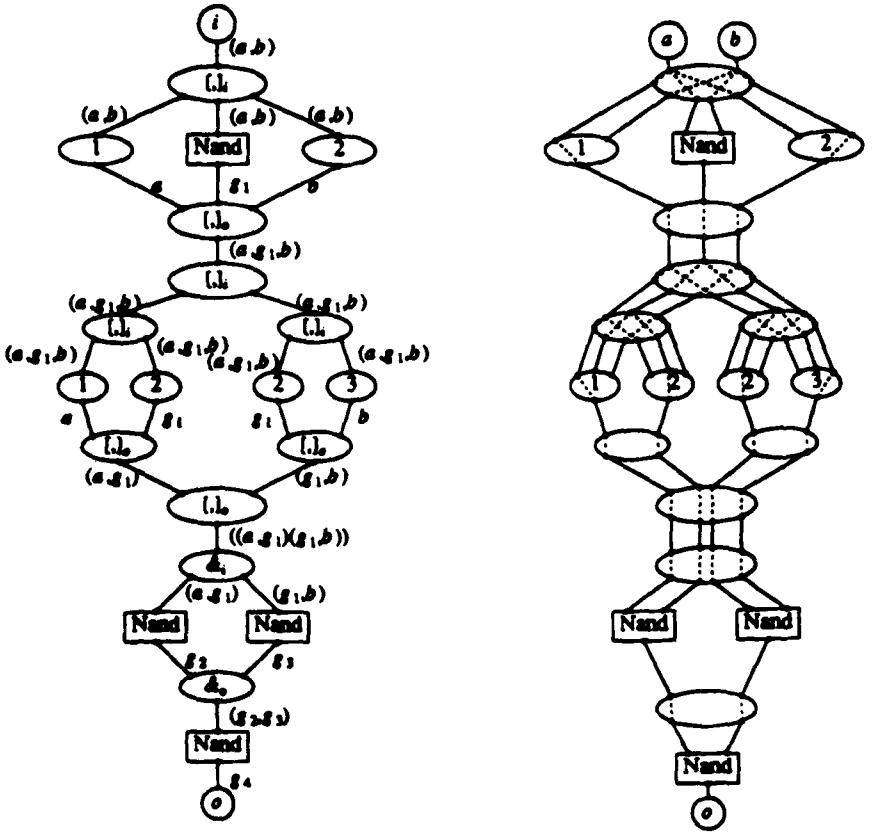


Figure 1. The planar graphs obtained from the symbolic execution of Nand @ &Nand @ [[1,2],[2,3]] @ [1,Nand,2] : (a,b).

generated by FP specifications is obtained by applying transformations to the FP programs.

4. Planar Topology of a Layout

To specify the embedding of a circuit in the plane, it is first necessary to adopt a method for specifying the embeddings of planar graphs. The cyclic ordering of edges around the vertices of a connected graph uniquely defines its embedding in a closed surface of minimal genus [Edmo60, Youn63]. Hence the embeddings of a connected planar graph in the sphere are uniquely characterized by the cyclic order of edges around their vertices in these embeddings. Using the topological equivalence of the sphere (minus it's north pole) and the plane under stereographic projection, an embedding of a graph in the plane is uniquely specified by providing (in addition to the cyclic ordering of edges around its vertices) the exterior window of the embedding in the plane. That is, the order in which the vertices which lie on the exterior face of the graph would be encountered during a clockwise traversal of the exterior window. Note that this does not imply that a planar embedding exists for any cyclic ordering of edges around the vertices, but rather that the embedding is uniquely characterized by its edge orderings and

exterior window if it does exist.

In general, circuits do not correspond to planar graphs (or hypergraphs) and even if they do, it may be desirable to lay them out in a non-planar fashion; asymptotic upper bounds on the area of layouts of planar graphs have been obtained from non-planar embeddings [Leis80, Vali81]. To extend the notion of planar topology to non-planar embeddings, a special type of node is introduced into the graph to implement crossings and branchings. A *planar circuit* is an embedded planar graph consisting of three types of nodes, *B*, *R* and *IO*-nodes. The *B*-nodes are the circuit elements (black-boxes) while the *R*-nodes are interconnection primitives. *IO*-nodes are nodes of degree one which lie on the external window of the embedding and serve as the inputs and outputs of the circuit. The embedding of this planar graph is uniquely specified by listing the ordering of edges around each node and the ordering of the input and output nodes of the circuit around the exterior; the input and output nodes are required to reside on the exterior window. Each of the *R*-nodes is accompanied by a partition of its edges. The partition indicates edges which must be interconnected within the *R*-node. Figure 2a shows a typical *R*-node and its partition. Note that nodes may have self-loops such as the one depicted in Figure 2b. Self-loops are said to be *trivial* if they do not enclose any other nodes. The loops formed by the edges e, f, and j are trivial self-loops while those formed by the edges a and g are not.

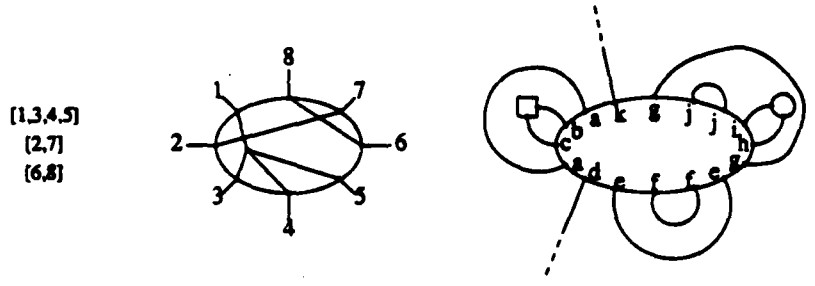


Figure 2. a) An *R*-node and its partition. b) An *R*-node with self-loops.

Given a layout, draw a set of simple disjoint closed curves in the plane such that all of the branchings and crossings of the layout lie within these curves and the circuit components lie outside of these curves. If each area enclosed by one of these curves is represented by an *R*-node, then a planar circuit is obtained. The layout is then said to be *covered* or *represented* by this planar circuit. The covering of a layout by a planar circuit is certainly not unique since there is more than one way to draw these curves. However, any two planar circuits which cover a given layout are related by a simple set of operations which can transform one planar circuit into the other. These operations not only define the equivalence between planar circuits which cover the same layout, but also provide the equivalence between planar circuits which cover layouts which can be topologically transformed into each other: obtained by the planar movement of components and the reshaping and stretching their interconnections. Three operations and their inverses are sufficient to attain this equivalence.

Merging *R*-nodes.

Any two *R*-nodes which are connected by one or more edges can be merged along those edges. Figure 3 shows an example of a merge. The edges on which the merge takes place are subsumed by the new set of partitions of the new *R*-node to preserve connectivity of the underlying circuit. A merge may create a *trivial* self-loop: a self-loop which does not enclose any other nodes of the planar circuit.

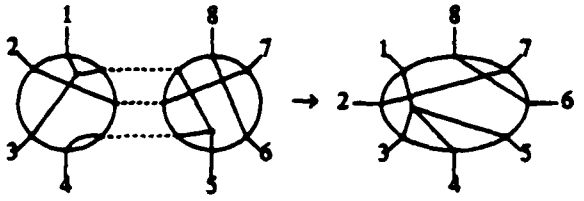


Figure 3. A merge of two *R*-nodes.

Cleanly Dividing *R*-nodes.

Cleanly dividing an *R*-node results in two new *R*-nodes with no new edges. Since the connectivity of the circuit must be preserved, this operation is only possible when the cyclic ordering of the edges of an *R*-node can be divided so that no partition is represented on both sides of the division. Figure 4 shows an example of a clean divide of an *R*-node along the dashed line. This is the only clean divide possible for this *R*-node.

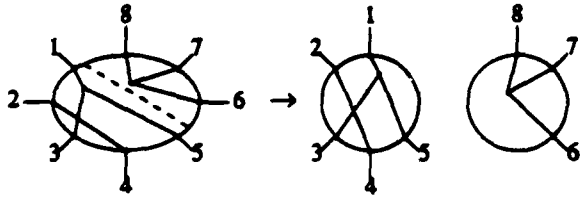


Figure 4. A clean divide of an *R*-node.

Removing trivial *R*-nodes.

A *trivial* *R*-node is an *R*-node of degree two whose edges are connected within the *R*-node. A trivial *R*-node can be removed leaving behind a wire. Figure 5 illustrates a removal.

These three operations and their inverses are guaranteed to produce planar circuits, providing the planar circuit on which they are applied possesses some connectivity properties. Otherwise, it is possible for a clean divide operation to disconnect a planar circuit's graph. However, if the underlying circuit represented by a planar circuit is connected when its inputs and outputs are joined (shorted), then the operations cannot disconnect the planar circuit's graph. This connectivity property is a reasonable restriction since it merely requires that each circuit component and wire be connected (via wires and components) to at least one input or output. Under this restriction, the operations and their inverses define equivalence classes of planar circuits which cover layouts with the same planar topology.

Figure 5. A removal of a trivial *R*-node.

Definition

Two planar circuits, A and A', are *homeomorphic* if there exists a finite sequence of operations transforming A into A'.

By showing that any representation of a layout by a planar circuit is homeomorphic to the 'smallest' planar circuit representing the same layout (the one in which each *R*-node covers as few branch points and crossings as possible), the equivalence between planar circuits which cover the same layout is obtained.

Proposition

Two planar circuits which cover the same layout are homeomorphic.

The operations on planar circuits also provide the means by which to simulate the changes in the layout which occur as components are moved around and their interconnections are stretched and reshaped while staying in the plane. These changes in the layout can be simulated by operations on the planar circuits which cover the layouts. Conversely, the operations on planar circuits can be realized by these changes in the layout.

Proposition

Layouts can be transformed into each other by orientation-preserving topological transformations of the plane and individual path deformations in the space which is obtained by removing the interior of the modules from the plane, if and only if any two planar circuits covering their respective layouts are homeomorphic.

The planar topology of an integrated circuit is then defined as follows.

Definition

The *planar topology* of a layout is the class of homeomorphic planar circuits which contains a planar circuit representing the layout.

5. Optimizing Planar Topology

If an FP program specifies the planar topology of a layout rather than a particular planar circuit, then an entire class of homeomorphic planar circuits must be considered in obtaining the layout of the FP program. The layout procedure may be sensitive to the particular planar circuit chosen from a class of homeomorphic planar circuits. Selecting the best layout with a given planar topology, would require

the consideration of all planar circuits within a class of homeomorphic planar circuits. However, the planar circuits within a class which need to be considered can be limited by requiring their R-nodes to have certain properties. Only those planar circuits within each class of homeomorphic planar circuits possessing these properties need be considered, since these planar circuits are optimal (under topological cost measures) within their class of planar circuits, if the layout procedure satisfies some assumptions. Since a planar circuit with these properties is unique (modulo one operation) within it's class of homeomorphic planar circuits, these properties provide a normal form for planar circuits.

Definitions

An R-node is *indivisible* if it cannot be cleanly divided even after permuting the order of adjacent self-loops.

An R-node is *maximal* if it does not have an edge connected to another R-node, is not trivial and does not have any trivial self-loops.

A planar circuit is *maximal*, (*indivisible*), {*maximal-indivisible*} if all its R-nodes are maximal, (indivisible), {maximal and indivisible} respectively.

One last operation involving a single R-node (which is in fact a composite of two previously defined operations) is required. This operation is treated as a single operation rather than as two separate operations since the uniqueness of the normal form can be only up to this operation.

Refoldings

A refolding of a node is in fact two operations, a divide (clean or inverse merge) of a node followed by a merge of the two nodes created by the divide. If the divide is clean, then the original node must have had some self-loops since otherwise no merge could take place.

The uniqueness, which gives a one-to-one correspondence between the R-nodes of homeomorphic maximal-indivisible planar circuits, is stated as follows.

Theorem

If A and A' are maximal-indivisible homeomorphic planar circuits, then A can be transformed into A' by a sequence of operations consisting only of refoldings.

The uniqueness of a homeomorphic maximal-indivisible planar circuit can only be up to refoldings since an R-node can never completely surround a B-node of the planar circuit. Merging an R-node with itself along a set of its self-loops may result in a doughnut R-node which encloses B-nodes. In this case, the information as to the relative position of the wires internal to the R-node with respect to the B-node, would be lost. Figure 6 contains a layout which can be covered by both of the maximal-indivisible planar circuits in Figure 7. The planar circuit on the right in Figure 7 could be obtained from the one on the left by dividing it's R-node along the diagonal in the lower left and then merging the two resulting R-nodes along the wire connecting them at the upper right.

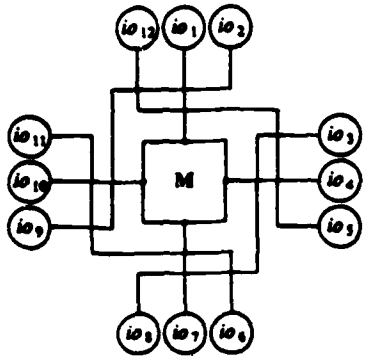


Figure 6. A layout which can be covered two distinct maximal-indivisible planar circuits.

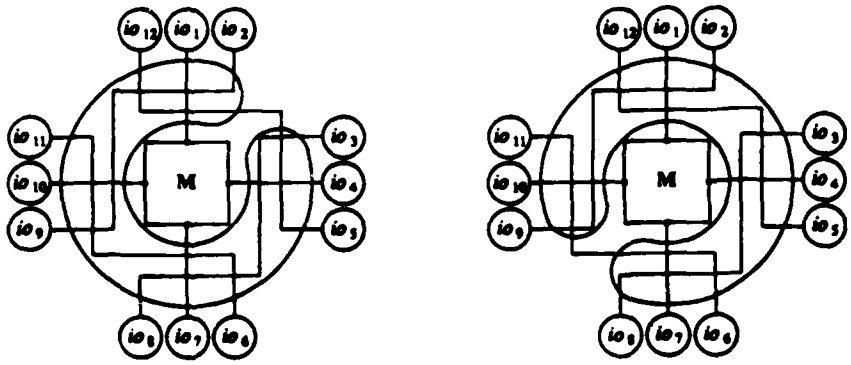


Figure 7. Two maximal-indivisible planar circuits which cover the layout in Figure 6.

The uniqueness of a normal form is often shown by deducing the Church-Rosser property from a local confluence property [Newm42]. Unfortunately, homeomorphic planar circuits do not lend themselves to this technique since the uniqueness can only be up to refoldings and refoldings can accomplish 'forward moves' when the planar circuit is not maximal-indivisible. Instead, the uniqueness is established by showing that an arbitrary sequence of operations leading to a maximal-indivisible planar circuit can be replaced by a sequence of operations consisting of specific types of operations in a particular order and resulting in the same planar circuit. The proof which is given in detail in [Schl86] consists of several steps in which the sequence of operations is rearranged by moving inverse merges and inverse clean divides to the end of the sequence and placing the other operations in a specific order. In the process, these inverse operations must cancel and can be dropped from the sequence since the final planar circuit is maximal-indivisible. If the original planar circuit is also maximal-indivisible then the sequence can be further rearranged by removing operations whose effects cancel, until only refoldings remain. Since homeomorphic maximal-indivisible planar circuits are unique modulo refolding operations and admit no merges or clean divides, this is defined to be the normal form for planar circuits. By examining the *R*-nodes and applying the appropriate operations, a planar circuit can be transformed into an equivalent maximal-indivisible planar circuit.

The rationale for obtaining a maximal-indivisible planar circuit is an issue which affects the procedure which will produce the layout from a planar circuit. The particular choice of planar circuit from among homeomorphic ones to which the layout procedure is applied may affect the quality of the layout. The underlying assumption is that the procedure produces a layout which is represented by the given planar circuit. With a few assumptions on the cost functions which measure the layout and the layout procedure, maximal-indivisible planar circuits can be shown to be optimal with respect to other homeomorphic planar circuits. The assumptions needed are the natural ones implied by the normal form.

1. Merges do not increase the cost.

2. Clean divides do not increase the cost.

3. Permuting self-loops to allow clean divides does not increase the cost.

4. Trivial R-nodes have zero cost.

At the topological level, these are natural assumptions to be made since the cost functions measure topological characteristics. Examples of topological cost functions which meet these assumptions are, the number of pairwise crossings required, the minimum number of layers needed if each net is restricted to one layer, and the minimum number of connections (module pins) which need to be dropped so that the remaining modules pins can all be connected on a single layer. At the geometrical level, these assumptions imply that the procedures which generate the layout implicitly examine the inverses of the operations of merging and cleanly dividing nodes. Such an assumption is also quite natural since the procedure which transforms topology to geometry cannot be decomposed; it must consider the geometric interactions of neighboring structures of the layout. It is easier for the layout procedure to determine how to 'glue' unconnected adjacent sections of the layout and to decompose R-nodes rather than to have to deal with non-maximal or divisible R-nodes. In the latter case, an optimal layout procedure would realize operations which are equivalent to merges and clean divides.

To assert that the costs of all homeomorphic maximal-indivisible planar circuits are the same requires an additional assumption. The problem is that even though an R-node is maximal-indivisible, permuting the relative order of a pair of adjacent self-loops can affect its internal wiring complexity. If the layout procedure is allowed to order these adjacent self-loops of an R-node as it sees fit, then the costs of homeomorphic maximal-indivisible planar circuits are the same. With this additional assumption, the following result is obtained from the theorem.

Lemma

Under the given assumptions, a maximal-indivisible planar circuit is optimal within its class of homeomorphic planar circuits.

6. The Planar Topology of FP Programs

Using the results of the previous section, it suffices to apply the layout procedure to a maximal-indivisible planar circuit in the class defined by the FP program. In this section, the transformation of an FP program's planar circuit into a maximal-indivisible one is explored. The planar circuit of an FP program is specified recursively in terms of its functional forms. Rather than flattening an FP program into a planar circuit and losing the structural information provided by its functional forms, this information is retained by representing the planar circuit as a tree in terms of its functional forms (it's computation tree). This information is exploited so as to efficiently map the planar circuit to a layout.

Each sub-tree in the computation tree corresponds to a function application. The functional form or primitive function as well as the input and output objects are stored in the root of the sub-tree. A primitive function is a leaf, and a functional form has as its children, the sub-trees corresponding to its sub-function applications. This computation tree is extracted by symbolically executing the FP program. Figure 8 shows the computation tree corresponding to the FP program of the planar circuits in Figure 1. The normal form of the planar circuit is realized by rearranging its computation tree. In general, it is not possible to rearrange the tree so that only maximal *R*-nodes are generated. However, the rearrangement of the computation tree is often sufficient to allow the layout procedure to effectively consider maximal-indivisible *R*-nodes.

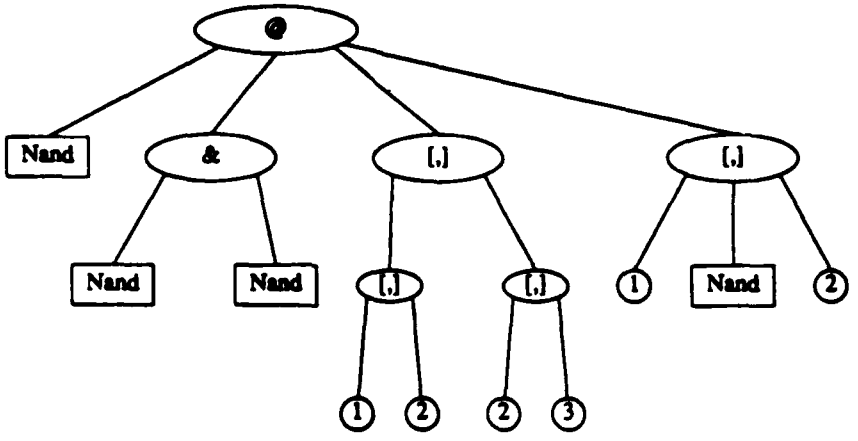


Figure 8. The computation tree of Nand @ &Nand @ [[1,2],[2,3]] @ [1,Nand,2]

In order to perform the operations to transform a planar circuit into its normal form, the planar circuit must possess certain connectivity properties. The planar circuits obtained from FP programs do not in general satisfy these connectivity properties. Much of the structure implied by an eager evaluation of a functional program may not be useful. This not only poses a problem in the application operations to the planar circuit, but may result in inefficient layouts since unnecessary components and wiring are generated. Before operating on the planar circuit, it is 'pruned' to remove useless wiring and components. The pruning is accomplished by a second pass in the reverse direction (from output to input) over the computation tree in which the objects stored in the nodes have their atoms tagged as useful or

not. These tagged objects are used by the layout procedure to avoid the generation of useless circuit structures.

The merging of R-nodes is accomplished by rearranging the computation tree to collect adjacent R-nodes (R-nodes which share edges) within a sub-tree. The planar circuit of a sub-tree which does not contain any B-nodes (a routing sub-tree) can be enclosed by itself within a simply connected region of the plane with only its input and output wires emerging from this region. Since there are no B-nodes within this region, this region forms a single R-node. Treating the routing sub-tree as a single R-node produces the same effect as combining all of the sub-tree's R-nodes using merges and inverse clean divides. If the computation tree can be rearranged so that adjacent R-nodes are located within a the same routing sub-tree, then maximal R-nodes will be obtained. The rearrangements of the tree which attempt to produce this effect are based on FP identities.

The R-nodes of a planar circuit are the result of the application of the primitive (structure modifying) functions and the distribution of inputs or collection of outputs required by a functional form. Only the distribution of the inputs to the sub-functions of a construct functional form can result in an R-node which cannot be decomposed into trivial R-nodes and removed. To extract this possibly non-trivial R-node generated by the construct functional form, a new functional form called a projection is introduced. It is denoted by $(h_1, \ldots, h_n)$ and is equivalent to $[h_1@1, \ldots, h_n@n]$. It differs from the construct functional form in that instead of applying each h_i to the input object, h_i is applied only to x_i where the input object is of the form $(x_1, \ldots, x_n)$. If the input object, x, is not of this form then $(h_1, \ldots, h_n):x$ is undefined ($\perp$). This new functional form provides the following identity for FP functions,

$$[h_1@g_1, \ldots, h_n@g_n] \equiv (h_1, \ldots, h_n)@[g_1, \ldots, g_n] \equiv (h_1:(g_1:(x)), \ldots, h_n:(g_n:(x))).$$

If $[g_1, \ldots, g_n]$ does not contain any B-nodes then it forms a routing sub-tree and can be treated with as a single R-node. By replacing a function, f by $f@Id$, $[f_1, \ldots, f_n]$ can be rewritten as $(f_1, \ldots, f_n)@[Id, \ldots, Id]$. The sub-tree corresponding to $[Id, \ldots, Id]$ can then be implemented as a single R-node. This R-node is the same as the R-node that would have been generated to distribute the input object to each of the sub-functions of $[f_1, \ldots, f_n]$.

Once the routing node generated for the construct is extracted and placed within its own routing sub-tree, adjacent R-nodes in the planar circuit are a result of only the compose, right insert and seq functional forms. Ignoring for the moment the right insert and seq functional forms, the following identities can be used to group adjacent R-nodes within the same sub-tree of the computation tree.

1. $f_1@(f_2@f_3) \equiv (f_1@f_2)@f_3$
   The associativity of the compose functional form is used to gather R-nodes within routing sub-trees.

2. $[h_1@g_1, \ldots, h_n@g_n] \equiv (h_1, \ldots, h_n)@[g_1, \ldots, g_n]$
   This identity is used to extract the R-node which distributes the input object from its construct functional form. In practice, the g_i's will correspond to routing sub-trees.

3. $[h_1@g_1, \ldots, h_n@g_n] \equiv [h_1, \ldots, h_n]@[g_1, \ldots, g_n]$
This identity is used to extract routing sub-trees from projection functional forms.

4. $\&f:\langle x_1, \ldots, x_n\rangle \equiv [f_1, \ldots, f_n]:\langle x_1, \ldots, x_n\rangle$ where $f_i = f$ for $1 \le i \le n$.
The apply-to-all functional form is handled as projection using this identity.

5. $[h_1, \ldots, h_n]@[g_1, \ldots, g_n] \equiv [h_1@g_1, \ldots, h_n@g_n]$
This identity is applied in order to maximize the parallelism. If there is a compose functional form which has two adjacent children which are projection functional forms, then they are combined using this identity.

The identities above are applied by a bottom-up recursive procedure which rearranges a computation tree without seq's and right-inserts. These rearrangements have no effect on the planar circuit generated since the connectivity and the embedding of the planar circuit is preserved. However, if the layout procedure treats a routing sub-tree as a single R-node, then these rearrangements realize effects equivalent to the application of operations to the planar circuit to merge its R-nodes. Hence these rearrangements do not alter the planar topology, but bring the planar circuit closer to a maximal-indivisible representative of its planar topology.

If the layout procedure treats a sub-tree with no B-nodes in it as a single R-node, instead of generating its planar circuit, then two R-nodes in the rearranged computation tree will be adjacent if and only if they correspond to nodes f and f' which occur as $f@[g_1, \ldots, g_n]@f'$ where one of the g_i's is *Id*. Thus, each maximal R-node corresponds either to some routing sub-tree in the computation tree or to a set of routing sub-trees under a compose which are separated by nested projections, one of which has an *Id* as a sub-tree. Only the latter can have self-loops and these must correspond to an additional *Id* within the nested projections.

Unfortunately this property does not extend readily to trees containing right inserts and seqs. Each right insert and seq could be rewritten, using the appropriate identity, as a combination of composes and projections. However, these transformations would sacrifice the structural information provided by the presence of these forms. Instead, the rearrangement procedure is applied individually to the children of right inserts and seqs functional forms and any routing sub-trees which result, are pulled out.

The techniques described in this section have been extended to the sequential versions of the apply-to-all, right insert and seq functional forms of vFP. Only the 'pruning' of the planar circuit is substantially different. Pruning is more complex in this case, since the usefulness of structure must be determined for the several applications which are mapped to it, rather than the single application in the combinational case.

7. Example

The specification of a simple carry-save array multiplier serves to illustrate the flexibility offered by fixing only the planar topology of the design as it is refined. A multiplication algorithm for unsigned digits written in FP will be developed starting with the following high-level description:

```
mult      =   2@repeat@[1,initialize,2]
repeat    =    (nul@1->id; repeat@stage)
stage     =   [tlr@1, +@[*@[last@1,3],2], *@[radix,3] ]
radix     =   %2
initialize  =   %0
```

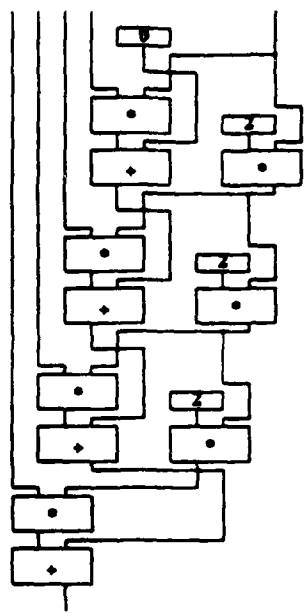


Figure 9. The abstract sketch of the mult:$((y_4,y_3,y_2,y_1),x)$.

Three arguments are passed from stage to stage. The first is the multiplier, the second is the sum of the partial products, and the last is the multiplicand. One bit of the multiplier is consumed at each stage. The partial product is added to the product of one bit of the multiplier and the multiplicand. The multiplicand is shifted (by multiplying it by the radix) left to prepare for the next stage. Eventually, the second argument is the product. The layout of the planar circuit implied by this multiplication algorithm is given in Figure 9.

This high-level algorithm is realized in hardware by implementing the radix 2 multiplication with a shift to realign the multiplicand. The multiplication of the multiplicand by a single digit of the multiplier is achieved by the function **select**. A carry propagate adder performs the addition with half and full adders. Conditionals are used in **CPA** and **HA** to determine the number of bits to be summed in each column. Figure 10 contains the layout of the planar circuit of this version of **mult** which is given below.

```
mult      =    2@repeat@[1,&initialize@2,2]
repeat    =     (nul@1->id; repeat@stage)
stage     =     [tlr@1, addthem@[select@[last@1,3],2], shift@3]
select    =     &(nul@2->2;andg)@distl
shift     =    apndr@[id,%()]
addthem   =      CPA@apndr@[trans,%()]
CPA       =     seq((nul@2->HA@1;(nul@1@1->HA@[2@1,2];FA)))
HA        =    (nul@1->id;(nul@2->[2,1];[andg,xorg]))
FA        =    [org@[1,2],3]@apndl@[1,HA@[2,3]]@apndr@[HA@1,2]
initialize =     %0
```

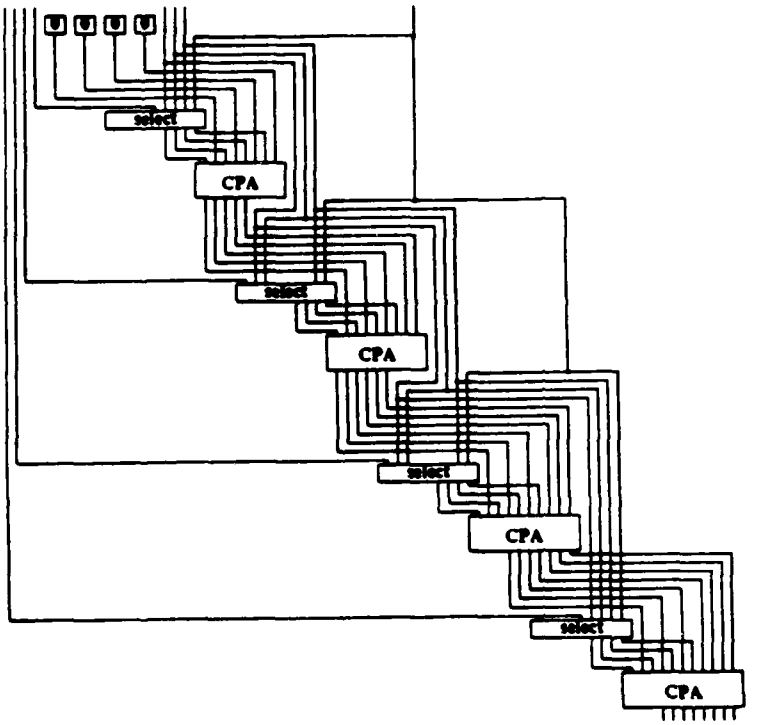


Figure 10. The abstract sketch of the **mult**:$((y_4,y_3,y_2,y_1),(x_4,x_3,x_2,x_1))$.

To increase the performance of the multiplier, the technique of carry-save addition is employed to avoid lengthy carry propagations at the intermediate stages. Instead of generating the sum at each stage, the partial product is generated in the form of two vectors and passed to the next stage. Hence, separate vectors of carry and sum bits are retained instead of the actual sum. The object being passed from stage to stage is now more complex. It consists of the multiplier, sum and carry bit vectors and the multiplicand. Note that the null object, '0', is appended to the left of the sum vector at each stage so that it matches the length of the carry vector and the multiplicand. Since the null object does not contain any atoms, wiring is not a concern. The specification of **mult** is now as follows.

```
mult     ≡    finish@repeat@setup
repeat   ≡     (nul@1->id; repeat@stage)
setup    ≡    apndl@[1,trans@2]@[1,&[initialize,initialize,id]@2]
stage    ≡    unweave@[1,addthem@2]@weave
weave    ≡     [tlr@1,&([2,3,1,4]@apndl)@distl@[last@1,trans@[2,3,4]]]
addthem      ≡     &(nul@2->[2,4,1];(nul@1->HA*;FA*))
unweave      ≡     [1,apndl@[%(),3@2],apndr@[1@2,%()],apndr@[2@2,%()]]@[1,trans@2]
FA*      ≡    [1@1,2,2@1]@[FA@[tlr,3]@tlr,4]@[1,2,andg@[3,4],4]
HA*      ≡    [1@1,2,2@1]@[HA@tl@tlr,4]@[1,2,andg@[3,4],4]
finish   ≡     CPA@apndr@[trans@[2,3],%()]
CPA      ≡    seq((nul@2->HA@1;(nul@1@1->HA@[2@1,2];FA)))
FA       ≡    [org@[1,2],3]@apndl@[1,HA@[2,3]]@apndr@[HA@1,2]
HA       ≡    (nul@1->id;(nul@2->[2,1];[andg,xorg]))
initialize   ≡     %0
```

At each stage, the sum and carry vectors and the multiplicand are *woven* together into the proper columns. The rightmost bit of the multiplier is distributed to each column and a full adder (except in the leftmost column where a half adder is used since there is no sum bit) is used at each column to add its sum bit, carry bit and the product of the multiplier bit with the multiplicand bit. The resulting sum and carry bits and the multiplicand are then *unwoven* for the next stage. As in the previous version, conditionals are used to detect the existence of a sum bit and invoke either the half or full adder function accordingly. Finally, the product is generated by the function **finish** which employs a carry-propagate adder to sum up the sum and carry bit vectors. Note that the sum and carry bit vectors have been initialized to zeros to simplify matters. In practice, the first two partial products should be used in their place.

The layout is extracted with the functions **HA*** and **FA*** defined as primitives. Figures 11 and 12 were both obtained from the application of the previous FP program to a symbolic object consisting of two 4-bit vectors. The same layout procedure (in which routing sub-trees are treated as single *R*-nodes) produced both layouts. However, the layout in Figure 11 was obtained by first rearranging the computation tree as described in the previous sections, before extracting its planar circuit. This specification of a carry-save array multiplier is simpler than the FP specifications appearing in [Schl84] where details of the wiring between stages had to be considered explicitly so as to produce a reasonable layout.

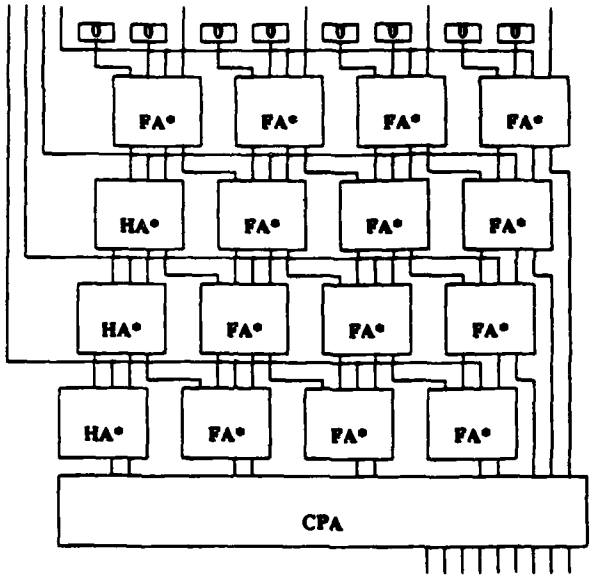


Figure 11. The layout of the optimized planar circuit of **mult**:$((y_4,y_3,y_2,y_1),(x_4,x_3,x_2,x_1))$.

8. Concluding Remarks

The objective of this research is to develop a formal high-level language approach to the specification, simulation, performance evaluation, and chip layout planning for VLSI systems. The use of a high-level applicative language (νFP) is motivated by the geometric detail provided by this programming style. The level of geometry afforded by the functional programming style is the subject of this paper. The planar topology of an integrated circuit is formally defined and demonstrated to be the desired level of geometric information to infer from a functional program. A normal form which is optimal with respect to topological cost measures is derived and used to improve the wiring of the layouts. The specification of a carry-save array multiplier is used to illustrate how this optimization reduces the effort required to specify integrated circuits.

9. Acknowledgements

The author is indebted to Milos Ercegovac, Sheila Greibach and Dorab Patel for their help, suggestions and support. A special thank you to Chan Pak Kuen for his help, and in particular for staying up to read late night versions of this paper.

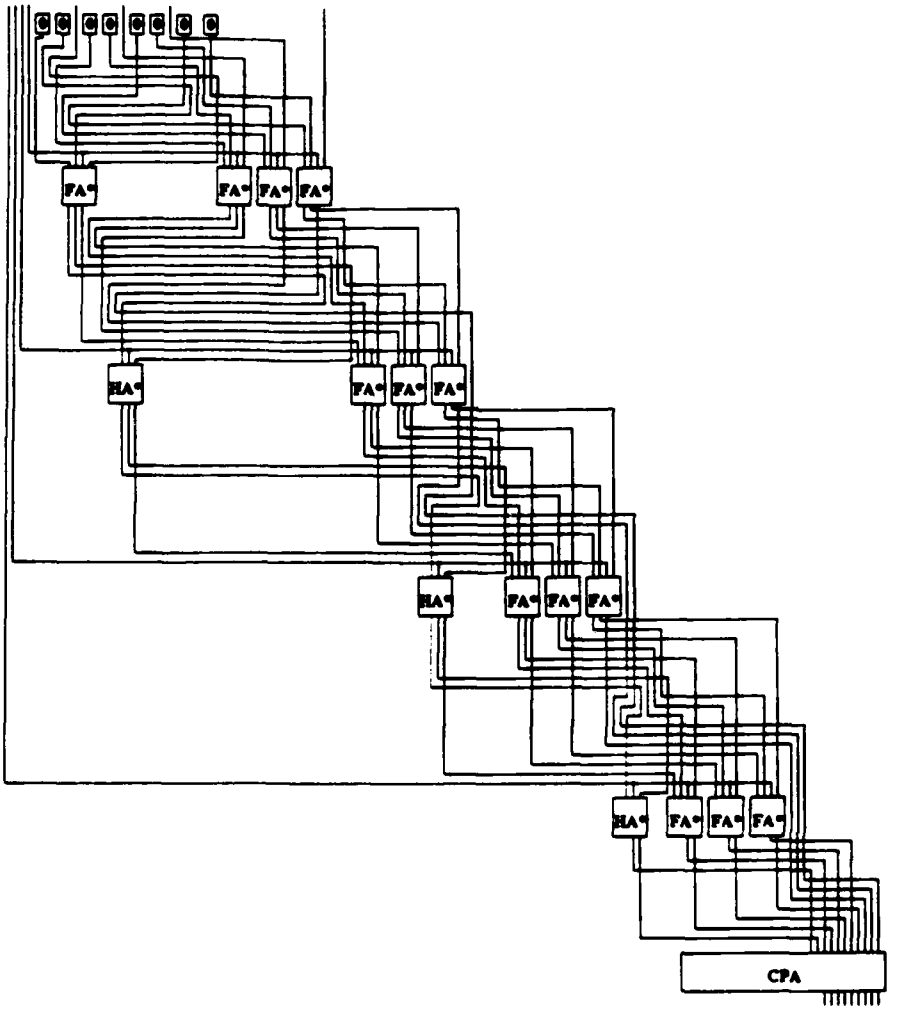


Figure 12. The layout of the untreated planar circuit of mult:$((y_4,y_3,y_2,y_1),(x_4,x_3,x_2,x_1))$.

References

[Back78] Backus, J., "Can Programming Be Liberated from the Von Neumann Style? A Functional Style and Its Algebra of Programs," *CACM, Turing Award Lecture* 21(8), pp.613-641 (August 1978).

[Edmo60] Edmonds, J. R., "A combinatorial representation for polyhedral surfaces," *American Mathematical Society Notices*(7), p.646 (1960).

[Joha79] Johannsen, D., "Bristle Blocks: A Silicon Compiler," pp. 310-313 in *Proceedings 16th Design Automation Conference*, San Diego, California (June 1979).

[Leis80] Leiserson, C. E., "Area-Efficient Graph Layouts (for VLSI)," pp. 270-281 in *Proceedings 21st IEEE Symposium on Foundations of Computer Science* (1980).

[Mesh85] Meshkinpour, F. and M. D. Ercegovac, "A Functional Language for Description and Design of Digital Systems: Sequential Constructs," pp. 238-244 in *Proceedings of the 22nd Design Automation Conference* (June 1985).

[Newm42] Newman, M. H. A., "On Theories with a Combinatorial Definition of `Equivalence'," *Annals of Mathematics* 43(2), pp.223-243 (April 1942).

[Oust81] Ousterhout, J. K., "Caesar: An Interactive Editor for VLSI Layouts," *VLSI Design* II(4), pp.34-38 (fourth quarter 1981).

[Oust84] Ousterhout, J. K, G. T. Hamachi, R. N. Mayo, W. S. Scott, and G. S. Taylor, "Magic: A VLSI Layout System," pp. 152-159 in *Procceedings of the 21st Design Automation Conference* (June 1984).

[Pate85] Patel, D., M. Schlag, and M. Ercegovac, "vFP: An Environment for the Multi-level Specification, Analysis, and Synthesis of Hardware Algorithms," pp. 238-255 in *Functional Programming Languages and Computer Architecture*, ed. J.P. Jouannaud, Springer-Verlag Lecture Notes in Computer Science, Nancy, France (September 1985).

[Rive82] Rivest, R. L., "The 'PI' (Place and Interconnect) System," pp. 475-481 in *Proceedings 19th Design Automation Conference*, Las Vegas, Nevada (June 1982).

[Schl84] Schlag, M., "Extracting Geometry from FP for VLSI Layout," CSD-840043, University of California, Los Angeles, Los Angeles, California (October 1984).

[Schl86] Schlag, Martine, "Layout from a Topological Description," PhD Dissertation, Available as Technical Report CSD-860039, University of California, Los Angeles (July 1986).

[Shee84] Sheeran, M., "muFP, a language for VLSI design," pp. 104-112 in *Proceedings ACM Symposium on LISP and Functional Programming* (1984).

[Sisk82] Siskind, J. M., J. R. Southard, and K. W. Crouch, "Generating Custom High Performance VLSI Designs from Succinct Algorithms," *MIT Conference on Advanced Research in VLSI*, pp.28-40 (January 1982).

[Vali81] Valiant, L. G., "Universality Considerations in VLSI Circuits," *IEEE Transactions on Computers* C-30(2), pp.135-140 (February 1981).

[Youn63] Youngs, J. W. T., "Minimal Imbeddings and the Genus of a Graph," *Journal of Mathematics and Mechanics* 12(2), pp.303-315 (March 1963).

APPENDIX II

An Empirical Study of On-Chip Parallelism*

Mary L. Bailey Lawrence Snyder

Department of Computer Science
University of Washington
Seattle, WA 98195

Abstract

This paper presents a methodology for empirically determining the amount of parallelism on a CMOS VLSI chip. Six chips are measured, and the effect of input choice and circuit size is studied. The unexpectedly low parallelism measured here suggests that certain strategies for parallel simulators may be doomed, and earlier efforts to extrapolate parallelism from small circuits to large circuits may have been overly optimistic.

1 Introduction

Simulation has continued to be a bottleneck in circuit design; parallel simulation is a potential solution to this problem. One oft-proposed strategy for parallel simulation is to partition the circuit among many processors, with each simulating its subcircuit [Smi86]. In particular, for switch-level simulation, circuits are often partitioned by transistor groups [Arn85, DB85]. This circuit partitioning strategy assumes that there is sufficient "activity" on the chip to keep all of the processors busy.

However, several instances of chip-level simulators designed using this strategy have not performed up to expectations. For example, Frank estimated a speedup of only 12 with a 64-processor simulator engine, assuming no bus contention [Fra85]. Arnold obtained a speedup of 3.8 for a 6 processor version of PRSIM running on MSPLICE [Arn85], and noticed that on average, each processor was idle 29% of the time. Are these problems due to poor partitioning strategies, or is there a more fundamental problem?

This paper addresses the question: How much parallel "activity" is there on CMOS VLSI chips? We begin by presenting a methodology for measuring the "activity" or parallelism on CMOS chips. We use this methodology to measure the parallelism of 6 circuits. We then discuss the effect of circuit size on parallelism. Finally, we draw our conclusions and discuss future work.

*This research is supported by the Defense Advanced Research Projects Agency, ARPA Contract Number MDA903-85-K-0072

2 Methodology

Before measuring circuit parallelism, we must provide a clear definition for it. Parallelism, as used in computer science, is a digital measurement, while the circuits necessarily have measurable analog properties. For our purposes, circuit parallelism is the average number of transistors switching at a given time. At this point we leave vague the quantization of time, but will return to it later in this section.

There are several ways to measure chip parallelism. Perhaps the most accurate method is to use a SEM probe. However, this requires many probes, one for each node on the chip. One could use a small number of probes and repeat the test until all nodes are measured, but this is impractical. Therefore, direct measurement is not possible.

An alternative, indirect method for measuring a circuit's parallelism would measure the chip's power requirements. Since static CMOS circuits derive power completely from transistor switching, the average power should correlate quite well with the amount of parallelism on the chip. However, because power dissipation depends on capacitive loads, pads and big drivers present in the circuit distort the power measurement. Eliminating these factors is impractical. Thus, external measurements are not viable for measuring circuit parallelism.

A third alternative is to use circuit simulation. Circuit simulation is a long trusted method for gathering information about the behavior of circuits. SPICE [Nag75] is recognized as the best indicator of circuit performance, but it can only be used on small circuits. Since we need to consider larger circuits for our measurements, we eliminated SPICE as a candidate for our circuit simulator, but use it to calibrate the selected simulator.

We chose to use RNL [Ter83], a linear level simulator that can handle circuits that are quite large. RNL is similar to a switch-level simulator, but includes timing information as a part of the simulation output. Given this choice, we considered two metrics for measuring parallelism. The first metric is the average number of events in the input queue at each timestep. The transistors in the queue are those that are changing. Thus, the average queue length is the average number of transistors that are changing. This is similar to the power measurements, but does not suffer from

the distortion of driver sizing, and thus may be the most intuitive measure of parallelism.

The second, alternative, metric is the average number of events executed in a timestep. This defines parallelism as the number of transistors that cross threshold values at a given time. This definition is commonly used when one wishes to measure the potential success of parallel simulators, since the events occurring in a single timestep can be executed in parallel by multiple processors.

Because this investigation was originally driven by the parallel simulation perspective, we use this second metric. In fact, to make the results more appropriate for event-driven simulators, we ignore timesteps where no events occur when computing this average parallelism.

Two issues remain in using this definition of parallelism. First, we must "calibrate" RNL. We do this by comparing it to SPICE. Second, we discuss how the parallelism measurements are affected by the choice of timestep.

2.1 Calibrating RNL

Since SPICE is the preferred simulator for measuring parallelism, but fails to handle large circuits, it is important to compare RNL and SPICE on small circuits to see if they reveal the same information. While we have used RNL at the University of Washington for five years with good success, RNL does not have the universal acceptance of SPICE.

The first issue in this calibration process is to determine exactly how the measurements will be taken, and which numbers will be compared. For RNL, an event occurs for one of two reasons: a node is changing between stable states ($0 \rightarrow 1$ or $1 \rightarrow 0$), or a node is changing due to charge-sharing. Charge-sharing may cause nodes to make transitions between stable states and the X state. Node changes from stable states to the X state occur with no delay. However, there may be a delay for the transition from the X state to the stable states. Also, RNL computes realistic delays only for nodes that it considers "interesting". These are all nodes *except* those that connect exactly two transistors in a chain. For the calibration experiments, we trace only the "interesting" nodes, and track events that give rise to stable states.

While RNL has a digital output, SPICE outputs a voltage which varies between 0 and 5 Volts. We first must determine how to "digitize" the SPICE output so that we can easily compare it to the output of RNL. In fact, the important issue is to determine when a signal makes a transition. We used a $4V$ threshold for rising signals, and a $1V$ threshold for falling signals. Thus a rising signal will have a value of 0 until its voltage rises above $4V$ at which time its value will become 1. These thresholds were selected because they provided the best timing correlation with the RNL output. Also, for the SPICE runs, we used a $0.1ns$ time increment in the transient analysis since the internal timebase in RNL is $0.1ns$.

We compared SPICE and RNL on three small circuits. We will discuss one of these, a 2 to 4 decoder, here. The others are discussed in [BS87]. The decoder has 32 transistors and 20 nodes. It is small, and is similar to a larger circuit used in later parallelism measurements. Thus, it was a good candidate for the calibration analysis. Of the 20 nodes in the circuit, 14 were "interesting." The inputs were initially set at 5 Volts and then simultaneously pulled down to 0 Volts. The circuit's reaction to this change ($5 \rightarrow 0$) was measured. In SPICE, each input was modeled as a pulse with $0ns$ rise and fall time. The results are shown in Table 1. The times listed are relative to the time when the select lines became 0.

Time (ns)	SPICE	RNL	Time (ns)	SPICE	RNL
0.0	s_1 = 0	s_1 = 0	1.5	o_4 = 0	
	s_2 = 0	s_2 =0	1.6	56 = 1	56 = 1
0.6	19 = 1	19 = 1	1.9		o_4 = 0
		20 = 1	2.6	o_1 = 1	
0.7	20 = 1		2.7	143 = 0	
	17 = 1	17 = 1	2.9		143 = 0
	18 = 1	18 = 1	3.2		o_1 = 1

Table 1: RNL vs. SPICE in the Decoder

There is a close correlation between RNL and SPICE for all of the signals except o_1 and o_4. These signals change much faster in the SPICE simulation. Each of these signals is the output of an inverter driven by either node 143 (o_1) or 56 (o_4). In RNL, the outputs cannot change before its input changes, thus the input changes first and the output changes a little later. In SPICE, however, the threshold of the n-channel transistors in the inverters have a lower threshold than $4.0V$ and the p-channel transistors have a higher threshold than $1.0V$, so the inverter starts driving its output signal before the input reaches its threshold. Since the input changes slowly compared to the output, the output reaches its threshold *before* the input does.

The other two small circuits, a small shift register and a modified full adder cell used in the Baugh-Wooley multiplier, yield similar results [BS87]. The results of SPICE and RNL are not identical, but are quite similar. Thus RNL provides a reasonable substitute for SPICE in these parallelism measurements.

2.2 Timestep Effects

One concern in using RNL for measuring parallelism is its $0.1ns$ timestep. Because we measure parallelism over a timestep instead of instantaneously, we need to understand the effects of the timestep on our parallelism metric. When considering the plausibility of creating parallel simulators, the $0.1ns$ timestep may be too small. This section presents the effect of changing RNL's timestep.

If larger timesteps give more parallelism, this could be useful for parallel simulators. However, expanding the size of the time step may affect the validity of the simulation. In particular, one must be careful to insure that an event and its predecessor, the event causing this one to occur, are *not* evaluated in the same timestep.

In order to examine this issue, we performed three types of experiments:

1. Retain the 0.1*ns* timebase, and measure the parallelism obtained by grouping the events into larger time slices. The results obtained here will have the same total number of events as in the 0.1*ns* timestep measurements, but may result in two events being evaluated at the same time when in fact one of them causes the other event.

2. Change RNL's timebase by changing the queuing delay (Δt, the number of 0.1*ns* units) of events. When an event is queued, if the new timebase is n times the old timebase, its Δt is changed to be:

$$\Delta t = \begin{cases} 0 & \text{if } \Delta t_{old} = 0 \\ n & \text{if } 0 < \Delta t_{old} < n \\ n(\Delta t_{old}/n) & \text{otherwise} \end{cases}$$

   where "/" is integer divide. This appropriately changes the timebase while guaranteeing that events dependent on each other will not occur in the same timestep.

3. Use the switch-level option in RNL to get parallelism data for a unit-delay time model. One would expect that the second experiment's data will approach the value obtained in this experiment as the timebase gets larger.

The circuit data we present here is from a 16 x 16 Baugh-Wooley multiplier, an instance of a generator developed at the University of Washington by Wayne Winder [Sys87]. It is a signed multiplier designed in static CMOS, and is purely combinatorial in nature. The test data consisted of 20 sets of random inputs, with each set consisting of a pair of inputs to initialize the circuit followed by a pair of inputs for the parallelism measurement.

The results for this experiment are shown in Figure 1, with 95% confidence intervals [BJ77]. The dashed confidence intervals are the average parallelism results for the "bucket" timebase measurements (experiment 1), and the solid line confidence intervals are the average parallelism results for the modified timebase measurements (experiment 2). The dotted lines at the top of the figure shows the confidence interval for the switch-level timebase (experiment 3).

The "bucket" experiments show parallelism increasing linearly as a function of timebase. The parallelism for the modified experiments increases also, but starts to level off around the switch level measurements. The curves differ due to the data dependency built into the modified experiments. In the modified timebase experiments the parallelism is ultimately limited by the data dependencies of the events, unlike the "bucket" experiment where the parallelism will continue to increase as the timebase increases.

For specific inputs, the parallelism in a given modified timebase can exceed the parallelism for the analogous switch-level simulation, and vice versa. For example, if three events occur in the order i, j, k, and i causes j, increasing the timestep can place i, j, and k in the same timestep. The modified timebase experiment will move j to the next timestep because it is caused by the earlier event, i, in the same timestep. This reverses the evaluation order of j and k, which may generate a different sequence of subsequent events. Thus different timesteps may result in more or fewer total events.

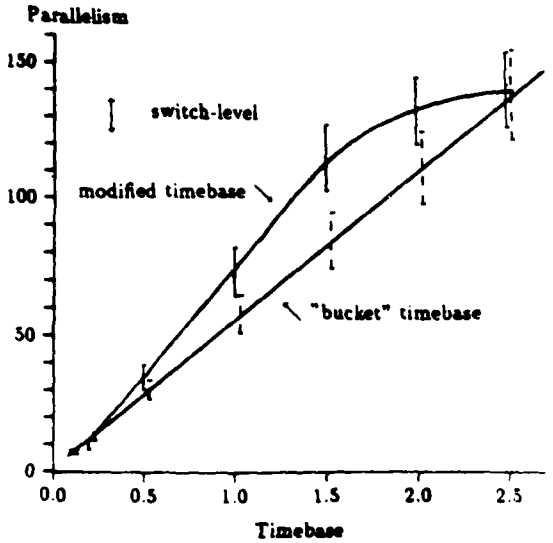


Figure 1: Effects of Timebase on Average Parallelism in the Baugh-Wooley Multiplier

Our measurements show that parallelism will increase as the timestep increases, and that the modified timebase measurements will converge to the parallelism measured by a unit-delay algorithm. Increasing the timebase does not dramatically change the measured parallelism, but may affect the accuracy of the simulation. In this paper, we use the more accurate 0.1*ns* timebase for our measurements.

3 Parallelism of Circuits

The above metric was applied to six CMOS circuits developed at the University of Washington. Two of the circuits have over 20,000 transistors; the others are much smaller. For all of the circuits, we used extracted circuits for the simulation so that the actual topology of the circuits would be reflected in the measurements. The results are shown in Table 2.

The Quarter Horse is a 32-bit RISC microprocessor [HJK+85]. It has 32 general purpose dual-ported registers, two internal busses, an ALU, shifter, memory address register, and a program counter structure with PLA control. In addition, it has an LSSD for testing purposes. As our test data we used a single run of a character load instruction, which takes 18 PLA cycles. The designers thought this instruction was highly parallel.

The IIR digital filter was designed by Hyong Lee [Lee85]. It includes a 16 × 16 multiplier, a 32-bit ripple adder, a 9-bit ripple counter, a 17 stage, 16-bit shift register, four 3 stage, 16-bit shift registers, and a PLA. Here we measured

Circuit	Transistors	Nodes	Parallelism (S.D.)						Percent	
			Percent		Average		Maximum		Serial (S.D.)	
Quarter Horse, 32-bit RISC Microprocessor	24,068	10,500	0.06%		6.3		140		60%	
IIR Digital Filter	27,360	14,399	0.04%		6.4		280		53%	
8 × 8 Baugh-Wooley Multiplier	2,162	1,083	0.26%	(0.046)	2.8	(0.50)	22	(7.3)	38%	(8.6)
8 × 8 Booth Multiplier	2,013	1,088	0.31%	(0.031)	3.4	(0.34)	41	(12)	36%	(3.8)
9-stage, 16-bit Shift Register	1,536	1,048	2.4%	(0.23)	25	(2.4)	69	(6.0)	3.4%	(4.9)
4 to 16 Decoder	208	110	2.9%	(0.91)	3.2	(0.47)	11	(3.0)	42%	(9.6)

Table 2: Circuit Parallelism.

one macrocycle containing 401 microcycles.

The other three circuits are instances of generators, programs that produce families of circuits. All were developed at the University of Washington. The Baugh-Wooley multiplier is a 8 × 8 instance of the Baugh-Wooley multiplier generator used in the timestep measurements above.

The Booth multiplier is a generator developed as a group project in a VLSI design class. Its design is based on the modified Booth multiplier using the sign generate method described in [Ann86]. The multiplier has a static CMOS multiplier plane and clocked pipeline registers between the multiplier plane and the final adder. An additional carry resolve unit is placed at each row in the multiplier plane to compute the carry generated by unnecessary low order bits. The final carry is then used as the *carryin* to the final 18-bit adder. The final adder is a precharged Manchester carry adder with carry bypass.

The shift register is a CMOS generator developed by Smaragda Konstantinidou [Sys87]. It uses two-phase non-overlapping clocks. The shift register latch is a master-slave dynamic latch implemented with two clocked inverters.

The decoder is a static gate style CMOS generator written by Bill Yost [Sys87]. It is parameterized by the number of select lines.

For these instances, we averaged 20 sets of random inputs. Each set consisted of enough random inputs to initialize the circuit followed by a random input for the measurement.

For each circuit in Table 2 we measured the percentage of parallelism, average parallelism, maximum parallelism, and fraction of serial steps. The percentage of parallelism is the percentage of nodes that are changing, or the average parallelism divided by the number of nodes in the circuit. This allows easy comparison of the parallelism of different sized circuits. In this table the standard deviations are shown in parentheses, and calculated values are shown to two significant digits.

The resulting parallelism measurements are remarkably small. For instance, for the Quarter Horse, if we used 139 processors, and there was no overhead for synchronization and communication, the speedup would be only 6.3. The Digital filter faces a similar fate; the speedup is slightly larger, but at the cost of many more (430) processors. The surprise in these circuits is the amount of serial activity. Both show over 40% of serial activity, during which only one processor can be doing useful work. In fact, the only circuit with much parallelism is the shift register, which was selected for this reason. Even here, only 2.4% of the nodes are changing at any instance.

These results are comparable to those reported by Ed Frank [Fra85]. He found that the potential parallelism for the proposed Fast-1 processor ranged from 4.1 to 192.1 with a mean of 29.2. The measure he used was essentially the same as our definition: he took the total number of instructions executed by the single processor version of the Fast-1 and divided this by the number of parallel simulation steps. The number of parallel simulation steps is roughly equivalent to the number of distinct timesteps in RNL. The difference in the measured parallelism is explained by the fact that the Fast-1 is a unit-delay simulator, and in Section 2.2 we saw that the parallelism values for switch-level simulation can be up to an order of magnitude greater than the values we measure using a $0.1ns$ timebase.

4 Influence of Circuit Size

We now consider the effects of circuit size on parallelism using the two multipliers and the shift register shown in Table 2. For each circuit and size we used 20 random data sets and computed the mean and standard deviation for each circuit instance. Each data set consisted of random numbers generated by successively using the UNIX function *random* to obtain bit values, and combining these to form random numbers of the appropriate lengths. For each data set, we first supplied enough inputs to obtain an output, and then supplied an additional input for the parallelism measurement. In the case of purely combinatorial circuits this meant using one input set to initialize the circuit and a second set for the measurement.

Size ($n \times n$)	Transistors	Parallelism (S.D.) Percent		Average		Maximum		Percent Serial (S.D.)	
4 × 4	594	0.54%	(0.082)	1.6	(0.24)	6.8	(3.2)	64%	(12)
8 × 8	2,162	0.26%	(0.046)	2.8	(0.50)	22	(7.3)	38%	(8.6)
16 × 16	8,178	0.18%	(0.039)	7.2	(1.6)	79	(24)	14%	(4.0)
24 × 24	18,034	0.16%	(0.016)	14	(3.0)	190	(41)	8.6%	(2.7)
32 × 32	31,730	0.15%	(0.024)	24	(3.8)	320	(3.9)	5.7%	(1.2)

Table 3: Baugh-Wooley Multiplier: Random Inputs

Size ($n \times n$)	Transistors	Parallelism (S.D.) Percent		Average		Maximum		Percent Serial (S.D.)	
8 × 8	2,013	0.31%	(0.031)	3.4	(0.34)	41	(12)	36%	(3.8)
16 × 16	6,867	0.21%	(0.020)	7.8	(0.74)	110	(30)	20%	(2.2)
24 × 24	14,665	0.19%	(0.016)	14	(1.2)	230	(43)	14%	(1.8)
32 × 32	25,407	0.17%	(0.0076)	23	(1.0)	340	(46)	11%	(1.4)

Table 4: Booth Multiplier: Random Inputs

For each experiment we again measured the percentage of parallelism, average parallelism, maximum parallelism, and fraction of serial steps. As before, the standard deviations are shown in parentheses and calculated values are shown to two significant digits.

Each circuit family was represented by a group of circuits, parameterized here by input size. All of the circuits in the same family are similar in design and construction.

For the Baugh-Wooley multiplier, five instances were generated ranging from a 4 × 4 multiplier to a 32 × 32 multiplier. The data set consisted of two sets of x and y inputs, the first set for initializing the circuit, and the second set for measuring the parallelism. The results of these simulations are shown in Table 3. Note that even though the average parallelism increases as the size of the circuit increases, the percentage of parallelism decreases for the first three instances and then stays constant(statistically). Thus, one cannot predict the parallelism of the 32 × 32 multiplier by multiplying the percentage of parallelism of the 4 × 4 instance by the number of transistors in the larger instance.

For the Booth multiplier four instances were generated, ranging from 8 × 8 to 32 × 32. A 4 × 4 instance was not generated due to limitations in the generator software. For these experiments, each data set consisted of three pairs of x and y inputs. Two input sets were needed to initialize the circuit due to the presence of the pipeline between the multiplier plane and the adder. Table 4 shows the results of these simulations. The average parallelism is higher in all of the Booth instances than in the corresponding Baugh-Woolev instances with the exception of the 16 × 16 instance. However, the percentage of parallelism is higher in the Booth multiplier for *all* instances. As in the Baugh-Wooley multiplier, the parallelism of the larger instances cannot be accurately computed by extrapolating from the percentage of parallelism of the smaller instances.

Five instances of the shift register were tested. Three instances had 16 bits with varying stages, and three instances had 8 stages with different numbers of bits. Each data set for the shift register consisted of $s + 1$ data points, where s is the number of stages in the instance. The results of the simulations are shown in Table 5. For the 16-bit instances, we see that the increase in parallelism grows linearly with the number of stages. Analogously, parallelism for the 8-stage instances grows linearly with the number of bits. The parallelism per 1,000 transistors is not statistically different for any of the instances. Thus, for shift registers, using the parallelism per transistor for a small instance to estimate the parallelism for a larger circuit should provide a realistic result.

The results in Tables 3 through 5 reveal a problem with estimating parallelism as a linear function of small circuits. For instance, Wong and Franklin [WF87] report parallelism experiments using a gate/switch level simulator and circuits ranging from 650 to 8,000 transistors. They scaled the parallelism values for 100,000 components and obtained parallelism measurements (the average number of simultaneous events) ranging from 80 to 3,294. As predicted by our results, the largest circuit had the smallest scaled parallelism while the smallest circuit had the largest. Our results show that the percentage of parallelism is typically not constant

Size Bits	Stages	Transistors	Parallelism (S.D.) Percent		Average		Maximum		Percent Serial (S.D.)	
8	8	768	2.5%	(0.32)	13	(1.7)	36	(4.0)	8.1%	(7.1)
16	4	768	2.4%	(0.29)	13	(1.6)	36	(4.2)	1.9%	(4.0)
16	8	1,536	2.4%	(0.23)	25	(2.4)	69	(6.0)	3.4%	(4.9)
16	16	3,072	2.4%	(0.18)	50	(3.8)	130	(9.0)	2.6%	(4.9)
32	8	3,072	2.4%	(0.16)	50	(3.4)	130	(8.7)	0%	(0)

Table 5: Shift Register: Random Inputs

and usually decreases as the circuit size increases. Thus the large parallelism measurements reported in that study may be overly optimistic.

5 Conclusions and Future Work

We have described a methodology for measuring the parallelism of CMOS VLSI circuits. We selected one of the parallelism definitions, the one most suited to the parallel simulation problem, "calibrated" the simulation tool, and measured the parallelism of six circuits. Three of the circuits were further analyzed to see how circuit size affects the resulting parallelism.

Parallelism in these circuits is remarkably low. It is unlikely that the circuits are not representative, either by the choice of circuits or by virtue of the design methodology at the University of Washington, since our results are comparable to those found by Frank. We also show that the percentage of parallelism is not necessarily constant over circuit size, and in fact often decreases as the circuit size increases. Measurements taken on small circuits may not be reliable predictors of the parallelism performance of larger circuits.

If chips are not very parallel, the conventionally accepted method for speeding up simulations through parallelism may be doomed. Specifically, the method of partitioning a circuit among processors may not be appropriate for parallel switch-level simulation, except, possibly, for simulators using a very small number of processors. These results go a long way toward explaining the poor observed performance of parallel event-based simulators. Moreover the implication is that the results apply, to a lesser extent, to gate-level event-based simulations for the same reasons.

There are several areas for future investigation. We are exploring alternative definitions for parallelism. Additional chips, including chips from other sources, should be tested to validate our results.

Most importantly, we need to discover why chips are not parallel. Are there better design methodologies or specific circuit designs that produce chips with more parallelism? Are parallel chips indeed better performers? Intuitively more activity should imply more performance for a given design. Is this intuition correct?

Finally, we may be able to use parallelism to assist at the architectural design level. One of the problems with the two large chips we examined is that at the architectural level, there is only one functional block active at any time. Pipelining should help here, and thus raise the overall parallelism.

References

[Ann86] Marco Annaratone. *Digital CMOS Circuit Design*. Kluwer Academic Publishers, Norwell, Mass., 1986.

[Arn85] Jeffrey M. Arnold. *Parallel Simulation of Digital LSI Circuits*. Master's Thesis, Massachusetts Institute of Technology, 1985.

[BS87] Mary L. Bailey and Lawrence Snyder. *Measurement of On-Chip Parallelism in CMOS VLSI Circuits*. Technical Report TR87-11-03, University of Washington Department of Computer Science, 1987.

[BJ77] Gouri K. Bhattacharyya and Richard A. Johnson. *Statistical Concepts and Methods*. John Wiley and Sons, Inc., New York, 1977.

[DB85] William J. Dally and Randal E. Bryant. A Hardware Architecture for Switch-Level Simulation. In *IEEE Transactions on Computer-Aided Design*, vol. CAD-4, no. 3, pp. 239-250.

[Fra85] Edward H. Frank. *A Data-Driven Multiprocessor for Switch-Level Simulation of VLSI Circuits*. PhD Thesis, Carnegie-Mellon University, November 1985.

[HJK*85] S. Ho, B. Jinks, T. Knight, J. Schaad, L. Snyder, A. Tyagi, and C. Yang. The Quarter Horse: A Case Study in Rapid Prototyping of a 32-bit Microprocessor Chip. In *Proceedings of the International Conference on Computer Design: VLSI in Computers*, pp. 261-266. IEEE, 1985.

[Lee85] Hyong Lee. *A Variable Digital Filter Design in 3um CMOS*. Master's Thesis, University of Washington, 1985.

[Nag75] L. Nagel. *SPICE2: A Computer Program to Simulate Semiconductor Circuits*, Technical Report UCB ERL-M250, Electronics Research Laboratory, University of California, Berkeley, 1975.

[Smi86] Robert J. Smith II. Fundamentals of Parallel Logic Simulation. In *Proceedings of the 23rd Design Automation Conference* , pp. 2-12. IEEE, June, 1986.

[Sys87] Northwest Laboratory For Integrated Systems. *VLSI Design Tools Reference Manual Release 3.1*. Technical Report TR87-02-01, University of Washington Department of Computer Science, 1987.

[Ter83] Christopher J. Terman. *Simulation Tools for Digital LSI Design*. PhD Thesis, Massachusetts Institute of Technology, September 1983.

[WF87] Ken Wong and Mark A. Franklin. Performance Analysis and Design of a Logic Simulation Machine. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pp. 49-55. IEEE, 1987.

APPENDIX III

A Model for Architectural Comparison

Sam Ho and Larry Snyder [1]
University of Washington
Seattle, WA 98195

1 Light and Heat

Recently, architectures for sequential computers have become a topic of much discussion and controversy. At the center of this storm is the Reduced Instruction Set Computer, or RISC, first described at Berkeley in 1980. [P80] While the merits of the RISC architecture cannot be ignored, its opponents have tried to do just that, while its proponents have expanded and frequently exaggerated them. This state of affairs has persisted to this day. This paper attempts not to settle the controversy, since indeed there likely is no one answer, but to provide a quantitative framework for a rational discussion of the issues.

In this paper, we seek to shed some light on this topic. The model we present takes an architecture and a computation. It has the following features.

- Quantitatively measures an architecture.
- Examines an architecture working on a computation.
- Separates the overall computation into logical pieces.
- Determines from the architecture how long each piece takes.
- Considers how much parallelism is available.
- Compares the results with other architectures.

While the early Crisp processor [D87] and the IBM 801 [R82] embodied similarly small instruction sets, the controversy began with Patterson's paper, "The Case for a Reduced Instruction Set Computer." [PD80] In this paper, he extolled the virtues of a smaller, simpler instruction set, as opposed to the large and complex instruction set typified by the VAX architecture. Unfortunately, he also impugned the motives of the designers of such architectures, by suggesting marketing strategy to be the moving force behind the choice of instruction set. The designers of the VAX rose to the bait, and papers and articles of either persuasion began to pour out.

The problem with these arguments was that they were not speaking of the same things. Each writer, quite naturally, chose examples that most supported his own view. The opposing writers, then, chose different examples with different results. On top of that, the RISC I chip from Berkeley contained an essentially unrelated piece of hardware, that of multiple overlapping register sets. The

[1] Supported in part by DARPA MDA903-85-K-0072 and Minority Affairs Fellowship

early papers on RISC often combined the effects of the register set and the instruction set with little regard for their relationship, which was tenuous, at best. When the RISC I chip turned out to have an error that caused it to run extremely slowly, it provided no vindication for the proponents of the CISC, since the problem had nothing to do with the complexity of the instruction set.

The model presented here is an attempt to provide a common quantitative basis for a discussion of this and other architectural questions. It is important to remember that a model will not, and is not intended to, settle a question once and for all. Instead, given a task, and several processors differing in some of their parameters, it provides a numerical basis to compare the results with the results from other possible sets of parameters. This dependence on the example cannot be ignored, and reflects the truth that the performance of any system depends greatly on what it is being used for, as compared to what it was designed for. It is unreasonable to expect a Lisp machine to perform matrix inversions efficiently, or conversely, to expect a matrix processor to execute Lisp well.

Another key point is to keep the comparison simple. If there are two or more changes between the two architectures being compared, the results from the model will reflect their combined effect. This is fine, if the changes are closely related, but it will lead only to confusion if the changes are not related.

2 A Model of Computation

In this model, we will define what a unit of computation is, and how long those units take to execute. To start with, we need a computer to examine. The definitions below hold true for a general computer, with an arbitrary instruction set and hardware capability.

In some of our examples, we will use the QuarterHorse, a 32 bit microprocessor designed at the University of Washington that bears close resemblance to the Berkeley RISC, except that it is microprogrammed.

2.1 The Processor

The *Functional* units are the basic blocks of the datapath of the processor. That is, they are such things as registers, shifters, arithmetic units, multipliers and the like. They act on the data being processed in the processor in some way. Registers do not, strictly speaking, act on the data, but the perform the "function" of making the data available to the other units. One way to change a processor is to change the functional units.

The *Control* portion of the processor determines what actions the functional units execute, and when they do so. On many commercial processors, this is a read-only memory containing microinstructions for a microprogram. On

other processors, particularly microprocessors, it is a collection of logic spread throughout the chip. The Berkeley RISC is one such processor.

Again, in any given comparison, only one of the functional units or the control should be changed. If there are several changes, it is difficult to separate the effects of the various changes. As an example, the RISC has a smaller instruction set than the VAX. In addition, the RISC has hardwired control, while the VAX is microprogrammed. On top of all that, the RISC has multiple register sets, but the VAX does not. With so many changes, it is not surprising that there is so much dispute on the merits of various aspects of the RISC.

2.2 The Calculation

Now that we have a computer, we need to give it a problem to work on. The problem (or family of problems) we choose will be called the *Calculation*. This choice is crucial to the evaluation we will perform, since the calculation determines which operations are likely to be executed, and worth optimising for.

For example, computing a matrix product will bias the results towards processors which can multiply quickly, while running an operating system which does frequent block input will bias the results towards a processor with a block movement instruction. This bias is not inherently wrong. It is quite reasonable for any given processor to do well on some tasks but poorly on others. Indeed, it is a wise choice to design a processor with knowledge of its intended application, and to optimise its operation for a specific class of problems. A processor designed to do everything will likely not be outstanding in doing any one thing.

In the examples below, we will use a fairly generic arithmetic computation, that of computing the greatest common divisor of two integers, to illustrate the point.

From this calculation, a compiler generates a set of data dependencies, and the transformations of the data as they pass through a graph. That is, it generates a dataflow graph for the calculation, with each node of the graph corresponding to some action being performed on the data. How this graph is produced is more of a topic for writers of compilers, and will not be discussed in this model.

The computation of the greatest common divisor can be represented by a graph, where the nodes are such operations as subtractions and comparisons. Unfortunately, this example is somewhat too simple, as it does not allow any room for parallelism, in the form of branches in the graph.

1. *If the first number is larger than the second, exchange them.*
2. *Subtract the first number from the second.*
3. *If the difference is not zero, return to step one, using the first number and the difference. Otherwise, we have the answer.*

The steps of this algorithm correspond to the nodes of the graph. In each step, we perform one operation.

In the more general case, there would be steps that do not depend on each other. For these steps, it would not matter if one were executed first, or the other, or both simultaneously on a parallel processor.

We shall restrict ourselves to sequential processors, and so, where the calculation does not determine an ordering, we shall impose one. The linear arrangement of actions will be called the *Sequence*, denoted S. Construction of the sequence is also commonly done by the compiler.

Now that we have defined the sequence of actions to take in performing a calculation, we need to define these actions.

In the GCD example above, the natural set of actions would be subtraction, exchange, and conditional branch, along with the more housekeeping activities of operand and instruction fetches, decode, and operand storage. These are simple actions because this is a simple calculation.

Since our model concerns itself with how long these actions take to execute on various processors, we must choose the actions carefully. If the actions are too small, larger-scale optimizations will affect sequences of actions instead of single actions, and we will not be able to model them. On the other hand, actions that are too large mean that we will have an unmanageable number of actions, each of which is affected in the same way by the same change.

2.3 Action

We define the *Action* to be the fundamental unit of computation. What it is is restricted by the conditions above, and by the experiment we are examining. For example, if we are considering the effect of a multiplier on the processor, multiplication should certainly be an action. However, if we are considering the effect of overlapping register files, there is no compelling reason to make multiplication an action.

When we have decided on the set of actions we will refer to it as C.

In the example above, we describe the second half of the set of actions as housekeeping. This distinction is worth noting, since we do not want to have the processor spend all its time on such ancillary activities, but rather on actions with some bearing on the calculation.

The *Overhead* actions, which we will denote as C_O, are just those actions which do not contribute to the calculation, but need to be executed anyway. These are usually instruction fetches, decodes and the like.

The remaining actions are *Computational*. These actions, denoted C_C, are actually part of the sequence associated with the calculation. In general, arithmetic and comparisons will fall into this category.

In addition, even among computational actions some of them are *Wasted*. These actions are computations that do not do anything towards the overall calculation, but are executed anyway, for lack of anything better to do, or

because a limitation in the instruction set requires its execution in order to perform some other activity.

In RISCs, such problems are generally caused by a limited overall clock structure which allocates time for an arithmetic operation whether it is wanted or not, and having the ALU to serve as the only channel between the input and output ports of the registers. An example is the RISC I, which insists on an addition when a register to register move is desired.

The remainder, and hopefully majority of the actions are *Useful*. These are the actions that do contribute to the calculation.

2.4 Time

And now, we need to see how long it takes for a processor to execute these actions. We measure time in basic units, which are *Cycles* of a master clock. This virtual clock does not necessarily correspond to the actual system clock of the processor, because some processors, particularly those with hardwired control, divide the incoming clock into many parts for different actions, while other processors, to prevent race conditions, use various arrangements of several clocks in what is really one cycle.

On microprogrammed processors, it is easier to determine what a cycle is, since the clock rate of the microprogram is generally the right measure for a cycle. The QuarterHorse falls into this category.

The key to the model is the mapping of each action to the number of cycles it needs to execute. We assign to each action a fixed cost which is the time, in cycles, needed to execute it. Generally, this will be a small integer, such as one or two, but it could be quite large, for more complex actions, such as multiplication, or of medium size, for actions of intermediate complexity. It is also quite common to have the carry chain for additions and subtractions take a slightly larger number of cycles than logical operation which do not involve a carry.

Finally, we can multiply the cycle by the *Clock Rate*, the time needed for one cycle, to get the time. In most cases, we can ignore this step, since the architectures in the experiment will have the same clock rate, but sometimes the clock rate is noticeably affected by the architecture. As an example, proponents of the RISC frequently claim that adding instructions to an architecture will slow it down. While the model cannot verify this claim, if it is true and measurable, we can take it into account.

2.5 The Interesting subset

Since the differences between the processors in any given experiment should be small, to avoid the mixing of effects warned about earlier, the majority of the actions will have identical numbers of cycles in their implementations in the two processors. To reduce the difficulty of computing the effects of the change, and

to see how much of the calculation is affected at all by the modifications, we can separate the actions into two categories.

The *Interesting* actions are those which are in some way different between the processors. The remaining, unchanged, actions are called *Common*. Then, the fraction of the cycles attributed to interesting actions is a measure of how much the modification affects the processor. Furthermore, the change in the overall time of the calculation is equal to the fractional change for the interesting actions multiplied by the share the interesting actions have in the overall computation.

2.6 Parallelism

While we are discussing sequential processors, we cannot entirely ignore parallelism. In particular, at the level of the action, even sequential processors allow some parallelism. This is, for example, why many processors have two data buses, allowing two operands to be simultaneously fetched, or even three buses, allowing yet another operand to be stored at the same time. Alternatively, actions that do not depend on each other can also proceed in parallel. This type of parallelism typically occurs between the instruction stream and the execution stream, in the form of prefetch buffers.

The *Maximum Parallel Set* of a processor is the set, P, of all the actions it can perform simultaneously. More strictly, $P = C_1 \times \ldots \times C_n$, the Cartesian product of several sets of actions. This notation means that in any given cycle, each one of the C_i can have an action in progress. The *Maximum Parallelism* is n, the largest number of actions that could conceivably be executing at one time. Sometimes, we will also use this cross product notation to denote cases where we want to point out that one specific group of actions that may execute in parallel with the rest. In such cases, the C_i may themselves have further structure. When that is the case, the maximum parallelism is the dimension of the maximal parallel set, and not just the visible portion with which we concern ourselves.

3 Derived Numbers

3.1 Derived Numbers in Time

Now that we have a model of what the processor is doing, we can combine this with the calculation the processor is performing to get quantities that reflect on the time the processor is taking.

The *Length* of the calculation is the number of cycles needed for the processor to perform the calculation. This is not quite adding up the actions performed in the calculation and multiplying by the appropriate number of cycles for each action, because we have actions that can proceed in parallel. During such times, we can, in the time of one cycle, be working on more than one action.

The *Time* of the calculation is the length, which is expressed in cycles, of the calculation multiplied by the cycle rate, giving a value in units of time. Clearly, the less time it takes for the calculation, the faster the processor. In most cases, we will deal with the length, rather than the time, of a calculation, and assume that whatever the clock rate is, it is the same for the processors we examine.

The *Efficiency* of the processor is the ratio of the length of the calculation to the number of actions in the sequence of the calculation. That is, it is the number of cycles needed to execute, on average, one useful action. This number is somewhat misleading, since it depends on what we chose as the actions, but among processors within one comparison, where the set of actions is the same, it provides a measure of how well the processor is doing, which is normalized for the size of the various calculations within one family.

3.2 Derived Classes of Processors

Within the general framework of processors defined previously, we can point out several interesting types of processors.

Any description of the controversy between the RISC and the CISC would be incomplete without an attempt at defining the differences between them. These differences, however, do not all fit within the realm of this model. In particular, the issue of the complexity of the control is difficult to quantify. Thus, the question of whether design time and chip area would be better spent on some other optimisation is left unanswered. Also, we cannot determine how much, if at all, faster the clock rate of a RISC would be, compared to the CISC implemented in equivalent technology, although if we accept the claims of some other person as to what this difference is, we could incorporate it into the model.

3.2.1 RISC and CISC

A *RISC* is an architecture with few actions, either overhead or computational, in each instruction, and a small number of total instructions. To some extent, whether a processor can be considered to be a RISC depends on the calculation, since the calculation determines the choice of actions. As an example, an instruction with floating point support in its instruction set would be a RISC if the calculation performed floating point computation, making the floating point operations a basic action. If the calculation did not perform such operations, the processor would then be carrying considerable excess baggage, and would be harder to justify as a RISC.

A *CISC*, by contrast is the opposite of a RISC. This processor has more instructions, and each of which performs more actions. The usual example of this is the VAX, which has over two hundred instructions, performing such varied tasks as manipulation of doubly-linked lists, and about a dozen addressing modes, which may involve up to three memory references and two additions and

shifts for each operand fetched. This means that each instruction invokes more computational, but also more overhead, actions.

We earlier defined the classes of possible parallelism. Now, we point out some common types of parallelism within a processor. These are, by no means, the only ways in which parallelism is possible, but they are the ones most frequently used in what are basically sequential processors.

3.2.2 Instruction Prefetch

Instruction Prefetch is the technique of fetching the next instruction to be executed while the current instruction is still executing. Symbolically, if we call C_{fetch} the set containing the action or actions necessary for an instruction fetch, and C_{other} the set containing the remainder of the possible actions, then the maximum parallel set $P = C_{fetch} \times C_{other}$. In such a processor, we can effectively discount the time necessary for instruction fetches from the overall time of the calculation. The RISC processor fetches one instruction ahead, while more complex processors, such as the VAX, typically fetch several bytes ahead, and provide them as needed to the instruction decoder.

3.2.3 Pipelining

Pipelining is the technique of sequentially partitioning the actions in an instruction into several classes, each of which can execute independently, but in order. Each step requires the same number of cycles to execute, so that several instructions in various stages of execution can be simultaneously processed.

The MIPS processor is pipelined. This processor divides its instruction processing into three stages. The first is instruction fetch and decode, the second operand decode, execution and store, and the third operand load. The processor is arranged so that one action from each of these classes can be executing at any given cycle. That is, if we call the first class C_{fetch}, the second C_{ex}, and the third C_{load}, the parallel set $P = C_{fetch} \times C_{ex} \times C_{load}$.

In the more general case, there can be many stages in the pipeline, and many instructions can be simultaneously in their respective states of execution.

4 Some example results

4.1 Block movement

A *Block Move* is the copying of a significant quantity of data from one location to another, with minimal change. In applications such as operating systems, block moves frequently result from the transfer of information from a buffered device to the user's address space. For such devices as disk and tape drives, this can mean moving a thousand bytes of data at a time. In applications such

as these, where such block movement is common, having such an instruction available will save a considerable amount of time.

In Clark and Levy's paper, [CL82] where the example instruction load consisted of an interactive operating system, such moves occurred frequently. Here, (Multiuser/All modes) the highest ranked, by time, instruction was the MOVC3 instruction, which is the block move instruction, consuming 13 percent of the total time. By frequency of occurrence, however, it was less than one percent of the instructions. In fact, in this benchmark, the average block move was of 20 words.

To analyze the costs of implementing the move in various ways, let us assume that each instruction executed incurs one overhead action for the decode, and each word moved incurs one computational action. Furthermore, subtraction and branch are also each computational actions. We assume that prefetching hides the cost of instruction fetch, except after a branch. Further, let all of these actions require one cycle each to execute.

In the architecture with such a block move instruction, we see that a block move of 20 words is implemented with a single instruction. This instruction incurs one decode and twenty moves, for a total of 21 cycles.

$$Time = (Decode + 20 * Move) * 1\frac{Cycle}{Action} = 21$$

Now let us consider the architecture without a block move. If the equivalent operation were implemented by unrolling the instruction into a number of individual moves, we then have twenty decode actions as well as twenty move actions. In a strictly sequential machine, except for the instruction prefetch, we would then require 40 cycles to do the same thing.

$$Time = 20 * (Decode + Move) * 1\frac{Cycle}{Action} = 40$$

However, the decode of one instruction can go in parallel with the move of the previous word. In this case, we have the pipeline $(Prefetch \rightarrow)Decode \rightarrow Execute$. Here, except for the first word, on each of the twenty following cycles, we complete one move at the same time as the decode for the next move, for a time of 21 cycles. In this case, the pipelining allows the block move to run as fast as it would if there were a special instruction for it. Even so, there is still a twenty-to-one space penalty for the unrolled instructions versus the single instruction.

$$Time = (Decode + 19 * (Decode \wedge Move) + Move) = 21$$

If, instead, the move were implemented as a tight loop, the space penalty would be minimal, but there would be a time penalty. In this case, let us examine the replacement sequence.

- Move word

- Decrement counter
- Jump if not zero

In this case, even assuming that the branch at the end disrupts the pipeline only for the loop exit, we see that each time through the loop requires three cycles, and outside the loop we have a cycle for decode at the beginning, and two cycles for fetching and decoding the instruction after the loop, to repair the pipeline disruption. This makes a total time of 63 cycles. Here, pipelining cannot make up for the loss of the special instruction.

$$Loop = Move + Decrement + Jump = 3.$$

$$Time = Decode + 20 * Loop + 2 * Repair = 63.$$

4.2 Operand Address Modes

One of the characteristics of a RISC is the small number of address modes. Most RISCs have a load-store architecture, in which fetching data from external memory is separate from the computation involving that data. These are instruction sequences of the form *Load A* followed by *Add A (from register)*. The alternative is called memory-to-memory. Here, the computational instruction is allowed to reference data from memory directly: *AddM A (from memory)*. If we make the assumption that the two instructions *Load* and *Add* require one more cycle of overhead than the single instruction *AddM*, we discover that the load-store machine requires one extra cycle each time on operand is referenced from memory. If we let L_{MM} be the length of the calculation on the memory-to-memory machine, and L_{LS} be the length of the same calculation on the load-store machine, and N_S be the number of operands fetched from memory, then we find that $L_{LS} = L_{MM} + N_S$.

In Wiecek's paper, [W82] just over 55 percent of the total operand references are to immediate or register data, and thus do not reference memory. Almost all the rest refer to memory once, with virtually none referring to memory twice. Since this same paper also indicates that 1.74 operands are referenced in the average instruction, multiplying this by the 45 percent of operands that reference memory produces 0.96 data memory references in each instruction. If, as we stated above, each such reference costs an additional cycle in the load-store machine, this means that having these memory reference modes saves almost one cycle for each instruction.

4.3 Floating point

Another case where a few instructions can make a major difference in the computation is that of floating point operations. Floating point operations require a large amount of time to execute, so many processors have special hardware

to assist in this computation. In such processors, the control of the processor has little bearing on this portion of the calculation, since we have a functional unit to do all of the work. If we give up this functional unit, we will need to emulate floating point operations with an equivalent series of additions, shifts and other operations available to us with the functional units we chose. Typically, just these actions will require on the order of ten times as much time as a well-designed floating point coprocessor.

In the system described by Clark and Levy, [CL82] which has a floating point processor, the time spent in floating point multiplication, T_{FP} is 3.5 percent of the total time used in the workload. The remaining instructions, then, account for the other 96.5 percent T_{other} of the time.

$$Time = T_{FP} + T_{other} = 3.5 + 96.5 = 100.$$

If we say that deleting the floating point hardware assistance will increase this time by a factor of ten, that increases the time fraction of the instruction from 3.5 percent to 35 percent of the original computation. The other instructions, of course, are unchanged.

$$Time = T_{FP} + T_{other} = (10 * 3.5) + 96.5 = 131.5$$

If we take our previous assumption that an unpipelined RISC will break each action into a separate instruction, each with one cycle of overhead, as well as the cycle of computation, we now have ten cycles for overhead, as well as ten cycles of computation for the equivalent floating point operation.

$$Time = T_{FP} + T_{other} = (20 * 3.5) + 96.5 = 166.5$$

With these assumptions, using a RISC processor for an unpipelined machine without hardware floating point support more than doubles the cost of not having such support, as compared with a machine with single instructions for floating point operations.

4.4 The CRISP processor: an example architecture

The Crisp processor was designed at Bell Laboratories for the efficient execution of C programs. It embodies many of the same ideas the RISC processor from Berkeley. However, it includes some special features, which will will describe briefly below.

The first idea is the stack cache. This cache allows dynamic choice of which variable to keep locally and which to keep in external memory. The idea is that the most frequently referenced variables will be at the top of the stack, and thus in fast registers, while less often used material is in external storage.

Here, based on numbers from the Crisp paper "Register Allocation for Free", are numbers for the register activity on a VAX. The basic numbers are 0 77

memory references actions per instruction for the standard VAX, 1.34 for a VAX without registers, and 0.24 for a VAX with Stack Cache registers. These then are divided by the ratio of memory reference time to overall time to produce a number for the time saved. That is, if we let N be the number of instructions, and N_M be the number of memory references, for the standard VAX, $N_M = 0.77N$, for a VAX making no use of registers, $N_M = 1.34N$, and for the Stack Cache version of the processor, $N_M = 0.24N$.

By contrast, the Wiecek paper gives 1.18 memory references per instruction and 1.8 register references per instruction. This results in total data references of 3.0 per instruction. Note here that the register usage count includes references to FP and AP, the frame and argument pointers. Another count, that of operands per instruction, gives a total of 1.8 operands per instruction, some of which, reference both memory and registers one or more times. These are the displacement operands, which reference both register and memory. Also, the displacement deferred operands refer to registers once, and memory twice. Adding the indexed modifier adds yet another reference for each of registers and memory, for a possible maximum of two register references and three memory references in a single instruction.

These differences can, in part, be attributed to differences in what is being measured. In fact, an accurate evaluation of branch folding requires the the measurement of memory usage be reasonable. Clearly, from these data, the instruction mix can greatly affect, here by a factor of two, the number of memory references in a processor.

In addition, even with the same instruction mix, determining which instructions can eliminate memory references by using a Stack cache can be difficult. The VAX does not distinguish in its instruction set which operands are stackable, and so, we must guess at which operands these are. In particular, the displacement operands are frequently used for all of stack, global, argument, local, and indirect operation. Sorting these apart is a major task.

Another aspect of the Crisp processor is branch folding. This optimization, which presupposes accurate branch prediction, allows branching in the preferred direction to be eliminated during the prefetch stage.

Here, this allows taken branches to be optimized from a typical one cycle to zero. Since branches occur every four instructions or so, in Wiecek, this can be a major saving of computation time.

Evaluating Branch folding is somewhat easier. In the simplest case, without using branch spreading, there is a saving of a cycle, or the entire fetch and execute pair in the pipeline, when a branch goes in the preferred direction. Thus we need only to count the number of branches executed and the fraction in the correct direction to get an estimate of the savings.

By applying these numbers to the one-cycle saving for each correctly predicted branch, we have another test for the value of folded, but unspread, branches.

The effect of branch spreading is similar to that of the earlier technique of

delayed branch. Both allow additional cycles to be saved if the branch and the comparison on which it relies can be separated by a few instructions to be executed unconditionally. Finding such instructions is the task of the compiler, and, as such, depends both on the details of the optimization in the compiler and the problem being examined. The RISC group at Berkeley has also published claims about the value of the delayed branch.

4.5 The RISC and the CISC

As we well know, this dichotomy has attracted much debate recently. This is an attempt to consider the claims of each side in the framework of the model. As such, it measures only those things which are within the view of the model, and not other effects we have seen claimed. Here we use the same simple model of the RISC we have used above. In this model, the RISC has exactly as many cycles of overhead as of computation, and they alternate. Furthermore, each instruction has exactly one computational action in it. In this case, the RISC will use two cycles for each useful action, since every computation is useful, and is accompanied by a cycle of overhead. Numerically, if N_U is the number of useful actions in the calculation, the number of wasted actions $N_W = 0$, so the total number of computational actions $N_C = N_U$. The number of overhead actions $N_O = N_C$, giving a total of $2N_U$ actions. We assumed that each action take one cycle, so there are $2N_U$ cycles to do N_U useful actions. This leaves the efficiency $E = 1/2$.

On the other hand, the CISC needs to have more cycles of overhead, but if it can also get more cycles of computation. Furthermore, some of the actions in the CISC may be wasted, if the instruction packages more actions than are necessary. Here, the efficiency is equal to the number of useful actions divided by the total number of cycles. This means that if all actions take one cycle, the total number of cycles L is equal to twice the number of computational cycles N_C, which is equal to the wasted actions N_W plus the useful actions N_U. $L = 2N_W + 2N_U$. The efficiency E is $\frac{L}{N_U}$.

5 Conclusion

This model is not an answer for all the arguments that have gone by in architectural discussion. Instead, it is a basis for fair and reasoned discussion. We have seen how to:

- Describe an architecture quantitatively.
- Identify where architectures differ.
- Remove irrelevant or unrelated factors.
- Separate logical actions from the implementation.

- Determine how much time each action consumes.

While it is true that honest differences of opinion will always remain, and all bias can never be removed, a clear separation of the parts of an architecture and their effects on overall performance will help prevent one aspect of a processor for erroneously receiving credit for what properly must be ascribed to some other part or effect. When there is a quantitative basis for evaluating architectures, is easier for discussions to shed more light and less heat.

6 References

1. Berenbaum, Ditzel, McLellan. The Hardware Architecture of the CRISP Microprocessor. *14th Symposium on Computer Architecture* June 1987.

2. Clark, D., Levy, H. Measurement and Analysis of Instruction Use in the VAX 11/780. *9th Symposium on Computer Architecture* April, 1982.

3. Colwell, Hitchcock, Jensen, Sprunt, Kollar. Computers, Complexity and Controversy. *Computer* Sep. 1985, p.8

4. Ditzel, D., McLellan, H. Register Allocation for Free: The C Machine Stack Cache. *Symposium on Architectural Support for Programming Languages and Operating Systems*. p.48 1982

5. Ditzel, D., McLellan, H. Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero. *14th Symposium on Computer Architecture* June 1987.

6. Fitzpatrick, Foderaro, Katevenis, Landman, Patterson, Peek, Peshkess, Sequin, Sherburne, VanDyke. A RISCy approach to VLSI. *VLSI Design*. Fourth Quarter 1981.

7. Ho, Jinks, Knight, Schaad, Snyder, Tyagi, Yang. The QuarterHorse: A Case Study in Rapid Prototyping of a 32-bit Microprocessor Chip. *IEEE International Conference on Computer Design: Very Large Scale Integration*, p.161, 1985.

8. Katevenis, Sherburne, Patterson, Sequin. The RISC II Micro-Architecture. *VLSI 83*.

9. Patterson, D. A RISCy Approach to Computer Design. *COMPCON Spring*. 1982

10. Patterson, D. Reduced Instruction Set Computers *Communications of the ACM* 28,1. 1985.

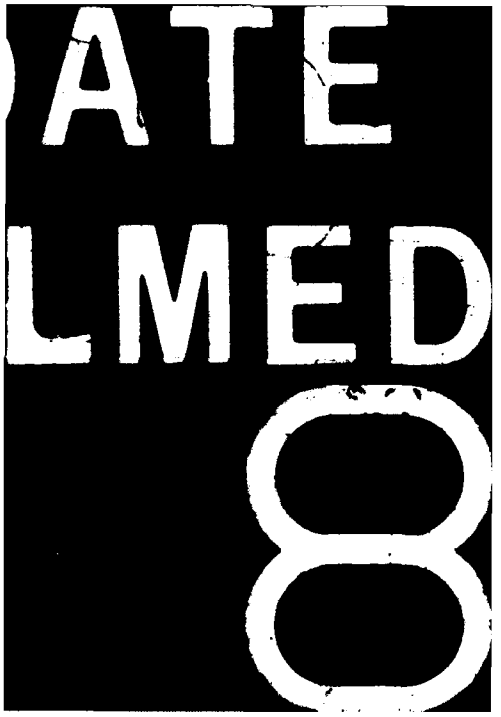
11. Patterson, D., Ditzel, D. The Case for the Reduced Instruction Set Computer. *Computer Architecture News* 9,3 p.25 1980

12. Patterson, D., Sequin, C. RISC I: A Reduced Instruction Set VLSI Computer. *8th Symposium on Computer Architecture*, p.443 1981

13. Przybylski, Gross, Hennessy, Jouppi, Rowen. Organization and VLSI Implementation of MIPS. *Journal of VLSI and Computer Systems* 1,2. 1984

14. Radin, G. The 801 Minicomputer. *Symposium on Architectural Support for Programming Languages and Operating Systems*. p.39 1982

15. Wiecek, C. A Case Study of VAX-11 Instruction Set Usage for Compiler Execution. *Symposium on Architectural Support for Programming Languages and Operating Systems*. March 1982.


ATE
LMED
8