MARCH 1995

WRL Research Report 95/2

Recursive Layout Generation

Louis M. Monier
Jeremy Dion

digital **Western Research Laboratory** 250 University Avenue Palo Alto, California 94301 USA

The Western Research Laboratory (WRL) is a computer systems research group that was founded by Digital Equipment Corporation in 1982. Our focus is computer science research relevant to the design and application of high performance scientific computers. We test our ideas by designing, building, and using real systems. The systems we build are research prototypes; they are not intended to become products.

There are two other research laboratories located in Palo Alto, the Network Systems Lab (NSL) and the Systems Research Center (SRC). Another Digital research group is located in Cambridge, Massachusetts (CRL).

Our research is directed towards mainstream high-performance computer systems. Our prototypes are intended to foreshadow the future computing environments used by many Digital customers. The long-term goal of WRL is to aid and accelerate the development of high-performance uni- and multi-processors. The research projects within WRL will address various aspects of high-performance computing.

We believe that significant advances in computer systems do not come from any single technological advance. Technologies, both hardware and software, do not all advance at the same pace. System design is the art of composing systems which use each level of technology in an appropriate balance. A major advance in overall system performance will require reexamination of all aspects of the system.

We do work in the design, fabrication and packaging of hardware; language processing and scaling issues in system software design; and the exploration of new applications areas that are opening up with the advent of higher performance systems. Researchers at WRL cooperate closely and move freely among the various levels of system design. This allows us to explore a wide range of tradeoffs to meet system goals.

We publish the results of our work in a variety of journals, conferences, research reports, and technical notes. This document is a research report. Research reports are normally accounts of completed research and may include material from earlier technical notes. We use technical notes for rapid distribution of technical material; usually this represents research in progress.

Research reports and technical notes may be ordered from us. You may mail your order to:

Technical Report Distribution
DEC Western Research Laboratory, WRL-2
250 University Avenue
Palo Alto, California 94301 USA

Reports and technical notes may also be ordered by electronic mail. Use one of the following addresses:

Digital E-net:	`JOVE::WRL-TECHREPORTS`
Internet:	`WRL-Techreports@decwrl.pa.dec.com`
UUCP:	`decpa!wrl-techreports`

To obtain more details on ordering by electronic mail, send a message to one of these addresses with the word ‘‘`help`’’ in the Subject line; you will receive detailed instructions.

Reports and technical notes may also be accessed via the World Wide Web: `http://www.research.digital.com/wrl/home.html`.

Recursive Layout Generation

Louis M. Monier
Jeremy Dion

March, 1995

Abstract

We present a recursive method for generating layout for VLSI chips which combines the flexibility of gate array and standard cell layout with the control and density of custom layout. The method allows seamless integration of hand-drawn and synthesized layout, so that hand layout need only be used where the increase in density is justified. Layout is generated automatically with predictable results; small changes in the source result in small changes of the overall layout. The system is versatile enough to build dense VLSI microprocessor chips automatically.

digital **Western Research Laboratory** 250 University Avenue Palo Alto, California 94301 USA

Table of Contents

1. Introduction **1**
2. The Annotated Hierarchical Netlist **1**
2.1. Cell Generators **2**
2.2. Netlist Traversal **5**
3. Layout Generation **5**
3.1. Hand-Drawn Cells **6**
3.2. Leaf Cells **7**
3.3. Composite Cells **8**
3.4. Routing **9**
3.5. Netlist Hierarchy Equals Layout Hierarchy? **10**
4. Results **11**
References **13**

List of Figures

Figure 1: Cell schematics **3**
Figure 2: A Cell Generator **3**
Figure 3: Boolean Equations **4**
Figure 4: A hand-drawn cell **6**
Figure 5: A leaf cell after placement **7**
Figure 6: The same leaf cell after routing **7**
Figure 7: Layout by Corner Alignment **8**
Figure 8: The BIPS-1 64-bit Floating-Point Divider **12**
Figure 9: The BIPS-0 Microprocessor **12**

1. Introduction

Most effort in commercial electronic design tools concentrates on logic and layout synthesis for semi-custom gate arrays, standard cell arrays and programmable arrays. Designs done this way are highly modifiable, and can be done by small teams. But they suffer performance penalties due to the use of restricted forms of circuits and layout. In high-performance design, where wire and gate delay must be accounted for in every block of logic and memory, these tools remove too much control from the designer and are fundamentally inadequate.

At the other extreme of the design spectrum are the full-custom tools used by an ever smaller number of design teams. These tools are built around layout as a master representation, and the layout editor as the main design tool. Such designs allow complete control over all aspects of performance. Unfortunately, layout is a very rigid representation that is difficult to modify. This leads to inflexible design methodologies in which functional partitions, interfaces and floorplans are fixed too early in the design cycle. This design style is also very vulnerable to changes in the underlying technology.

Recursive layout generation bridges the gap between semi-custom and full-custom design. It requires no compromise on performance; the tools allow full control over circuit selection and layout, and all tools work at the device level, not the gate level. Recursive layout generation also produces modifiable designs. Large complicated designs are created once, but modified forever. This system allows early and continual floorplanning, global performance tuning, and tracking of technology changes.

Recursive layout has been reported before. Ayres [1] describes recursive layout of synthesized PLA's from a netlist described by a program. Barth *et. al.* [2] described the recursive composition of hand-drawn cells connected by a channel router. The system described here is novel in that it allows synthesized and hand-drawn layout to be combined easily and connected automatically by a router to produce layouts which are virtually indistinguishable from those made by hand.

The recursive layout system was designed in order to build high-performance ECL and BiCMOS microprocessors [9]. Although we shall give examples of ECL circuits, almost nothing in the layout generation system is specific to a particular VLSI technology. The full set of tools developed for our microprocessor project also includes a switch-level bipolar timing verifier based on ideas in [8], a switch-level bipolar simulator [10], electrical rules and noise margin checkers [16], and extensions to the *magic* layout editor [12].

2. The Annotated Hierarchical Netlist

There is no single best way to describe circuits and logic. For analog circuits such as RAMs, schematic drawings of interconnected transistors are the most concise specification. For control logic, Boolean equations allow easiest debugging. For a parameterized n-bit adder a program is the most flexible representation. Rather than attempting to mix several different forms of circuit description, we chose to use their greatest common divisor, the program, and to translate schematics and Boolean equations into programs. Programs are simultaneously the most expressive and most modifiable descriptions we know of. We cast as much as possible of the design

process as a problem in software development, and use all the standard programming tools to change and debug our design. Many of our CAD tools are in libraries which can be linked and run with the circuit design.

We explicitly decided *not* to develop a specific programming language for hardware description. Instead, we embedded our hardware descriptions in a common programming language, C++. This entails some syntactic inconvenience, since wires, nets and cells are not base types in the language, but this is a small price to pay for extensibility. We were able to extend and change the hardware description language over the course of our project to add code for special purposes easily, without having to undertake language and compiler changes. We were able to debug our programs using standard tools. These advantages are compelling, and far outweigh syntactic convenience of a specialized but inflexible language.

The use of C++ led us naturally to another choice. A netlist is not a file, nor a database. It is a data structure in the virtual memory of a running program, the result of executing the cell generation code. Netlists for circuits with millions of devices can be generated in one or two minutes. This is probably faster than they can be read from a file. This model sidesteps the problems of versions of netlist files and consistent updates to databases which arise with other approaches. Our netlists are never saved, but generated as needed. Our source is a program composed of many files, and we maintain versions of them with standard revision control software.

2.1. Cell Generators

A circuit in our system is represented by a C++ program. A procedure in this program is a *cell generator*. It can take arbitrary parameters, and returns the netlist - a pointer to a C++ object called a *Cell* - for the requested cell. Many such generators take simple parameters, such as the amount of current drive to provide in the outputs, or the number of bits in a register, but some are quite complicated. For example, instead of having a library of OR-gates, we have an OR-gate generator, to which we pass the number of inputs, and a description of the outputs required. At this level, our form of description is quite like other hardware description languages. But our netlist fulfills one other function - it is designed to capture *all* the information needed to generate layout.

For circuits which are best described graphically, we use a conventional drawing editor for schematic capture.[1] The editor has no specialized knowledge of schematics, just as text editors have no knowledge of programs. These schematics drawings are "compiled" into C++ by our drawing interpreter *drip*, which interprets lines as wires, and names as labels of wires and devices. It uses only visible cues in the drawing to parse it into devices and wires, and can put arbitrary code, such as loops and tests, into the generated procedure. The resulting procedure is the equivalent structural description of interconnected devices. Two examples of schematics are shown in figure 1. These two cells carry annotations for layout which will cause them to be laid out by the *Leaf* cell synthesizer described below.

[1]Note to reviewers: if the companion paper on *drip* is accepted for the conference, this section will be simplified and figure 1 deleted.

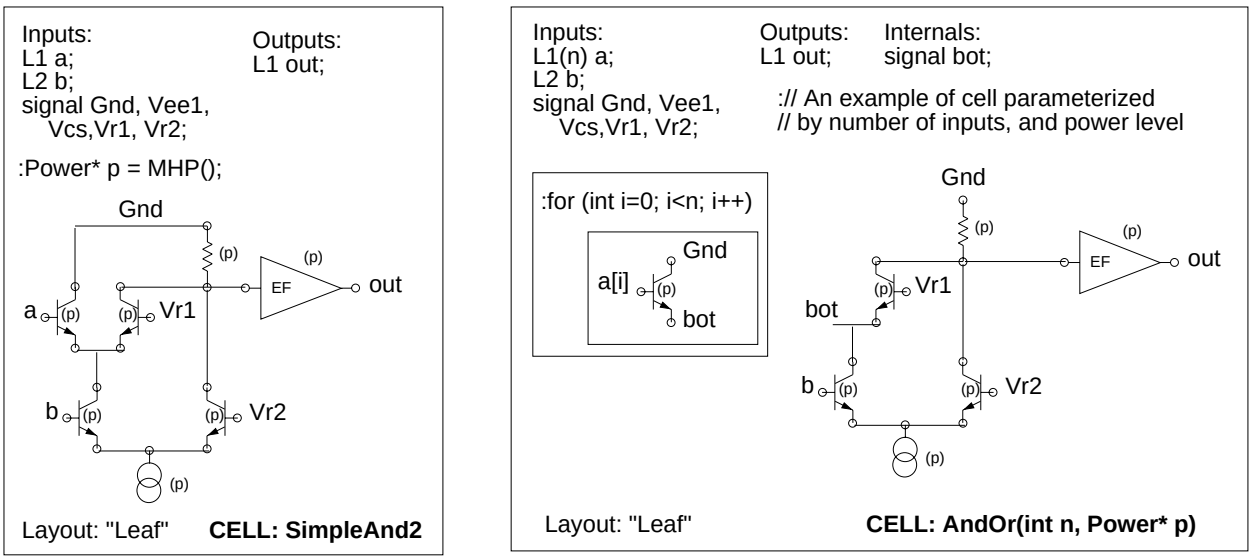

Figure 1: Cell schematics

```
 1  Cell* Register(int n, Power* p) {
 2      char key[100];
 3      sprintf(key, "Register_%d_%s", n, p->name);
 4      CACHECELL;
 5      INW(clock, L3DPair());
 6      INW(select, L2(1));
 7      INW(in, L1(n));
 8      OUTW(out, L1(n));
 9      POWER;
10      for (int i=0; i<n; i++) {
11          INST(MuxFFCell("T2", "T", "T", p));
12          BD(i, in->Sub(i));
13          BD(s, select);
14          BD(c, clock);
15          BD(o, out->Sub(i));
16      }
17      SLK(AbutUp);
18  ENDCELL
```

Figure 2: A Cell Generator

Most circuits are better described by programs rather than drawings. Programs are more concise than drawings and much easier to modify. Figure 2 shows an example of a cell generator for a register. This procedure was written by hand, but is very similar to the output of *drip* that would be generated from the equivalent schematic. Functions in capitals are C++ macros, provided for syntactic convenience. A cell is a self-contained block of logic with an interface of named wires. Wires have types, analogous to arrays and structures in C++. Lines 5-9 describe the cell's interface, consisting of inputs (INW), outputs (OUTW), and power supplies (POWER). *Clock*, for instance, is an input wire of type *differential-pair-at-ECL-level-3*, and is a structure with two single-wire subfields *true* and *complementary*. Lines 10-16 describe the contents of the cell in terms of instances of subcells, here all of the same type, a one-bit flip-flop with a multiplexor on its input. The parameters to *MuxFFCell* describe the number and logic levels of its inputs and outputs, and the desired current drive. Lines 12-15 define how the named wires *i, s, c, o* in the subcell instance are to be bound (BD) to wires in this cell; all bits get the same select and clock wires, but the *i*'th bit gets the *i*'th input and output wires.

The netlist distinguishes between a cell and an instance of the cell. Cells are shared, and there is one copy of each unique cell. There is one instance for each use of the cell. In the example of figure 2, only one *MuxFFCell* will be created, but there will be *n* instances of it. Cell sharing is provided by the programming convention in all cell generators shown on lines 3-4. Each cell generator computes a unique name for the required cell based on the procedure name and parameter values. If the netlist for this cell already exists in a global cell cache, it is returned immediately at line 4. Otherwise the generator constructs the netlist, and stores it in the cell cache at line 18 before returning it to the caller. On subsequent calls with identical parameters, the generator returns the cached netlist.

The advantages of a hierarchical netlist with shared cells are enormous, since all aspects of a cell which are common between its many instances are shared. For instance, there is exactly one RAM cell in the netlist, but thousands of instances of that cell. The RAM cell layout is generated only once, but is then instantiated in many places in the chip layout. Cell sharing speeds up layout generation by orders of magnitude.

The hierarchical netlist also carries *annotations*. Line 17 shows a *layout method*, the most important annotation for the purpose of layout generation. A layout method completely describes how the cell layout is to be generated. "SLK" means "set layout key" and *AbutUp* is a particularly simple example; it lays out the instances in a cell from bottom to top in a column. Simple layout methods like *AbutUp* are just the name of a C++ procedure, but more complex layout methods described below carry associated data. The layout method is an integral part of the definition of the cell, and is specified by the designer just like the wires defining the cell's interface. If the same circuit needs to be laid out in two different ways, it is described by a cell generator accepting the layout method as a parameter. Two different cells will result from calls supplying different layout method arguments. They will have identical netlists, but different names and different layouts.

```
EQNCELL(FPDivCtl)
    INPUT(ck, L3DPair());
    INPUT(asign, L1());
    INPUT(bsign, L1());
    OUTPUT(sign, L1());
    OUTPUT(start, L1());
    . . .
    start <<= op["div"] & ~abort & allowOp;
    sign <<= FF(ck, start, asign ^ bsign);
    . . .
    SetStdCellLayout(cell, 3500, 250);
ENDEQNCELL
```

Figure 3: Boolean Equations

Another library of C++ functions provides the syntax of Boolean equations, which are extensively used for control logic and for prototyping new blocks of logic. Figure 3 shows a small example, whose layout is defined as a block of "standard cells" 3500 units wide with 250 units of vertical space between each row. The library maps these equations into valid ECL gates (such as n-input OR gates) during netlist generation. It makes use of three-level series gating, free inversion, and wired-OR [6]. The result of calling a cell generator defined by Boolean equations is a netlist identical to that which would be obtained by explicitly interconnecting a collection of gates, flip-flops and multiplexors. We trust the equation mapper to make this translation on parts of the circuit where precise selection of the gates used is not critical.

To generate a netlist for an entire chip, we translate all schematics into C++. The C++ source code for schematics, equations and hand-written cells is then compiled and linked with the CAD libraries. We also include a short main program which calls the generator for the top-level cell of the chip. A complete microprocessor having 4 million devices was described in 25K lines of C++; 15K lines of CAD libraries were linked with the design, resulting in a 10MB executable. The chip netlist was generated in a couple of minutes.

2.2. Netlist Traversal

When the main program calls the generator for the top-level cell in the circuit, it gets in return the annotated netlist for the circuit. This netlist can then be traversed to produce input files for simulators. We use SPICE for circuit simulation of analog circuits, and *bisim* [10] for switch-level simulation of digital circuits. Each simulator requires its own input file format, so there is a different traversal of the circuit netlist for each simulator. Writing the input file for a simulator takes about as long as initial generation of the netlist.

3. Layout Generation

Layout generation is best seen as just another traversal of the in-memory netlist. It is a bottom-up, batch process. No manual intervention is required, since all the information needed to generate the layout is in the netlist and in the layout methods. Layout happens in the same way for all cells. First, the subcells are laid out recursively. Then the cell's layout method is used to place the instances of the subcells. After placement, some connections may have been made by abutment or overlap. Any which remain are completed by the *Contour* router [4]. Finally, connectivity checks are made to detect shorts and opens. An electrical short of two nets is usually a sign of overlapping subcells in an incorrect placement. Opens usually result from creating a routing problem which is too difficult. Both of these problems are solved by editing the layout method for the cell, and *never* ***ever*** by editing the generated layout directly. In this way we maintain the rule that all information needed to generate the entire layout is recorded on the netlist.

The representation of layout for a cell comes in two parts. The geometry of the instances of the subcells is defined simply by the transformations of the subcell layout which place them in the current cell. New rectangles defined in the layout of this cell - for example the wires and vias added by the router - are stored in a set of tiling planes [14], one per layer in the VLSI technology (usually one for each metal layer, and one for the active devices). Layout generation is correct by construction - unless shorts and opens are detected and reported - and two invariants are maintained for every cell:

- The layout represents the same circuit as the netlist. There are the same number of devices of the same sizes, interconnected in the same way.
- The cell geometry in the tiling planes is maintained in an extracted form; each solid tile is labelled with the net to which it belongs.

3.1. Hand-Drawn Cells

The recursion ends at hand-drawn cells, which do not depend on layout of the subcells. These cells have the layout method *ReadMagic*, and laying them out means reading a file of the same name made with the *magic* layout editor. The use of a hand-drawn cell comes at a price; its layout must be manually maintained over all future changes in technology or in pitch-matching to other cells. Our system is no better at managing large tracts of hand layout than any other, so hand-drawn cells are used either for analog cells such as pads, or for memory cells, where the gain in density is compelling. Figure 4 shows the layout of two bits of register file.

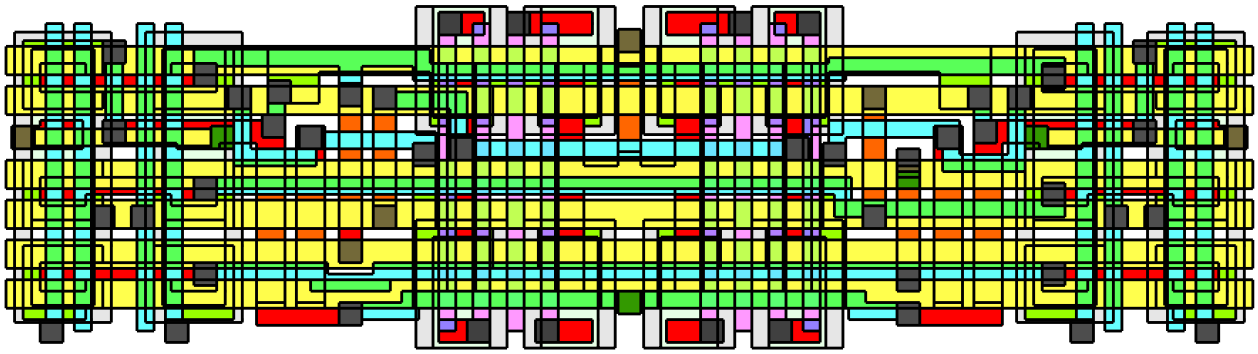

Figure 4: A hand-drawn cell

The result of reading the *magic* file for the cell is a set of tiling planes full of rectangles. Hand-drawn cells have no instances of subcells, so the entire geometry of the cell is represented in the tiling planes. But the two layout invariants must be checked; the layout must represent the same circuit as the netlist, and each rectangle has to be associated with its net in the netlist. First the netlist represented by the layout is extracted by finding the transistors and resistors, and following their interconnections. Then this netlist is matched with the netlist specified for the cell by a graph isomorphism method similar to *gemini* [5]. The matching is purely topological, though labels in the hand layout can be used to disambiguate symmetries. If the graphs are identical, each rectangle can be labelled with the correct net, and the layout has been proved to match the netlist. In case of mismatch, an error is immediately reported.

Hand-drawn cells are also routed just like any other cell, since the extracted layout makes it clear which nets are disconnected. Our router is able to make connections to arbitrary geometry and route in obstructed areas, and the designer may choose to leave some of the wiring to the router even in hand-drawn cells. A good example of this is a 64-bit decoder which has a 6-bit input bus, in which it is extremely easy to make a wiring error. In this case, labelling the device terminals and bus wires is less error-prone than wiring by hand.

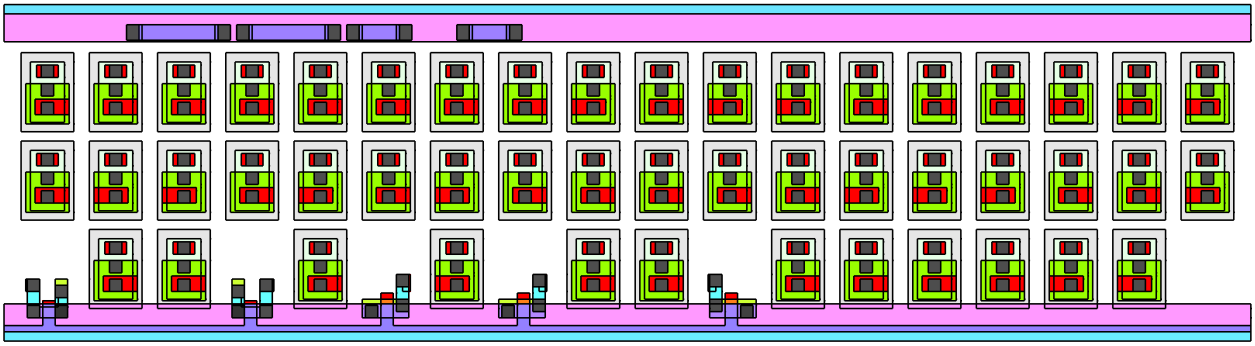

Figure 5: A leaf cell after placement

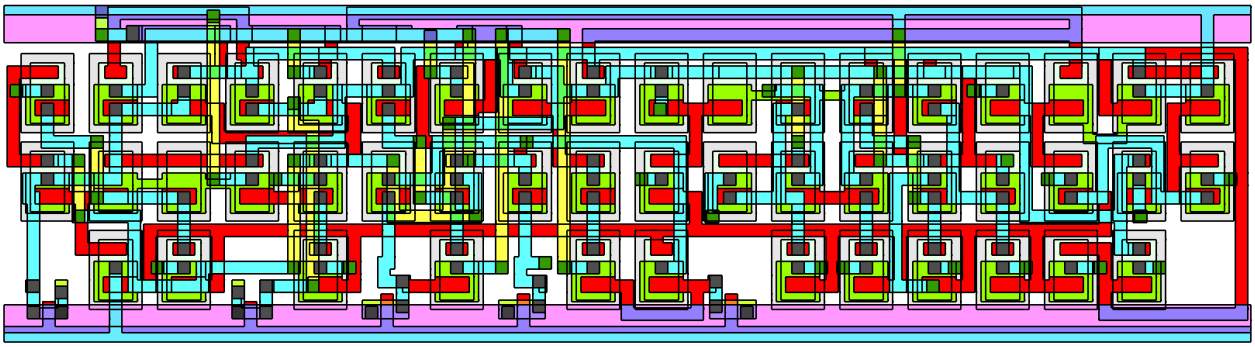

Figure 6: The same leaf cell after routing

3.2. Leaf Cells

Many small cells are synthesized using the *Leaf* layout method. Typically, these are gate-level cells, containing up to a hundred transistors. Figure 5 shows a flip-flop with a 10-input integrated multiplexor after placement by *Leaf*. Figure 6 shows the same cell with the wiring added by *Contour*.

Leaf is a general ECL cell synthesizer. It produces finished layout given the netlist of devices, the layouts for the devices (they all have the *ReadMagic* layout method), and a vertical pitch. This synthesizer is one of the few parts of the layout system dependent on ECL technology. The placement uses no hints from the designer, but selects positions of the transistors and resistors which maximize the number of connections which can be made using polysilicon interconnect. The resulting placement is very close to the density of hand designs. In part this is due to the regularity of current trees in ECL logic and the similar sizes of bipolar devices. The current algorithm - the sixth and by far the best in the history of the project - is based on exhaustive combinatorial search. It finds very good placements in milliseconds. The key ideas are to consider the transistors in the right order (by current tree, then by closeness to the current source within each current tree), and to avoid the use of a cost function to compare a large number placements. In place of a cost function, the algorithm has a *rejection criterion* and an *acceptance criterion*. The rejection criterion is used to prune the search tree by detecting partial placements which can never lead to a good complete placement, perhaps because a current tree can not be placed contiguously. The *acceptance criterion* is a strict definition of a good placement. The first complete placement passing the criterion is accepted as the final solution.

After placement, routing of leaf cells is done by our *Contour* router in polysilicon and up to three levels of metal. Routing in second- and third-level metal is permitted but discouraged by the cost parameters given to *Contour*. Complete layout generation for a typical leaf cell takes from 1 to 50 seconds on a DECStation 5000/200.

This style of leaf cell synthesis may be contrasted with semi-custom design. In our system there is no cell library, and we cannot predict in advance which 2000 or so of the enormous number of legal ECL gates will actually be used. The particular gate selection is determined by the parameters passed to the gate generator procedures during creation of the netlist. Each distinct gate is made exactly once - because of the sharing enforced by the cell cache during netlist generation - and is instantiated one or more times. During layout generation therefore, each unique ECL gate is placed and routed exactly once, and that layout is shared among all its instances.

3.3. Composite Cells

Most higher-level cells are neither hand-drawn nor synthesized, and a variety of layout methods are used for them. For completely regular cells such as multi-bit registers, simple layout methods like *AbutUp* and *AbutRight* are used to arrange the subcells in rows or columns. For completely random cells, such as blocks of control logic, *StdCell*, a fully automatic placer based on the conjugate gradient method [11] generates rectangular blocks with minimal wire distance. For cells which are neither fully random, nor fully regular, a layout method based on corner and edge alignment is used to place subcells to the last micron.

```
 1  // four RAM cells in a symmetrical 2x2 square

 2  CELL(RamQuad)
 3      INW(wwl0, oneBit());    INW(wwl1, oneBit());
 4      INW(rwl0, oneBit());    INW(rwl1, oneBit());
 5      OUTW(rbl0, DiffPair()); OUTW(rbl1, DiffPair());
 6      OUTW(wbl0, DiffPair()); OUTW(wbl0, DiffPair());
 7      POWER;
 8      AlignRegion quad("quad");

 9      INST(RamBit());
10      BD(rwl, rwl0); BD(wwl, wwl0); BD(rbl, rbl0); BD(wbl, wbl0);
11      InstRegion bit0(instance); /* lower left */

12      INST(RamBit());
13      BD(rwl, rwl0); BD(wwl, wwl0); BD(rbl, rbl1); BD(wbl, wbl1);
14      InstRegion bit1(instance, TopToBottom);
15      quad.Align(bit1, LL, bit0, UL); /* upper left */

16      INST(RamBit());
17      BD(rwl, rwl1); BD(wwl, wwl1); BD(rbl, rbl0); BD(wbl, wbl0);
18      InstRegion bit2(instance, RightToLeft);
19      quad.Align(bit2, LL, bit0, LR); /* lower right */

20      INST(RamBit());
21      BD(rwl, rwl1); BD(wwl, wwl1); BD(rbl, rbl1); BD(wbl, wbl1);
22      InstRegion bit3(instance, Rotate180);
23      quad.Align(bit3, LL, bit0, UR); /* upper right */

24      SetLayout(cell, quad);
25  ENDCELL
```

Figure 7: Layout by Corner Alignment

Figure 7 shows how corner alignment is used to assemble four RAM cells into a 2x2 array which is symmetrical around both the horizontal and vertical center lines. This cell is the repeating unit of four bits in a cache RAM. Here, an *AlignRegion* object is declared at line 8, and corner alignments are added to it at lines 15, 19 and 23. In line 24, the resulting data structure is made the layout method of the cell. The instructions for generating the layout (lines 14-15: "flip bit1 top to bottom, then align its lower left corner to the upper left of bit0") are simply recorded as annotations on the cell during netlist generation. When layout is required for this cell, the alignments are retrieved and executed.

Note that layout by corner and edge alignment is independent of the size of the underlying *RamBit* cell. This is important; editing the hand-drawn *RamBit* cell will not invalidate this layout. We contrast this with normal hand assembly of layout, in which the series of keystrokes and mouse clicks to arrange the subcells is lost. If the *RamBit* changes size, these keystrokes and mouse clicks must be repeated manually. With recursive layout generation, only a program needs to be re-executed.

Integrating corner alignment directives with the netlist has proved very successful. It might be objected that the layout directives obscure the connectivity of the netlist, but in practice this has not been a problem. Having all the information about a cell in one place makes circuit modifications much easier. Whenever a cell's netlist needs to be changed, it is usually quite easy to make the corresponding layout changes, especially when the reward of a new color plot can be generated quickly.

3.4. Routing

A large fraction of the development effort was spent on *Contour*, a general router based on a hybrid maze/line search principles [3, 13, 7] and the corner-stitched data structure [14, 15]. *Contour* is used in each cell of the design to complete connections not made by placement. In general, the router is adding wires to cells on top of wires already routed in the subcells. Routing over the top of active logic is one of the characteristics of custom VLSI, and is largely responsible for its density. For this reason conventional channel routers, which route only over unobstructed rectangular channels, are unacceptable. *Contour* reads design rules from the same file used by the design-rule checker. It can generate routing with minimum dimensions and clearances from obstacles on all wiring layers simultaneously. *Contour* is used repeatedly at all levels of the design, from routing polysilicon in the leaf cells to chip assembly. In fact, part of a layout method is a set of directions to the router on the correct use of metal and/or polysilicon layers to connect the nets.

Routing a cell is done by breaking each net into a spanning tree of pairwise *connections* between disconnected *terminals*. Terminals are not simple rectangular connection points, but arbitrary collections of wires and devices - in general they are the result of previous routing in subcells. These connections are then ordered by likely difficulty in order to produce a connection schedule for the cell. The connections are then attempted in order of increasing difficulty. Each connection is completed by finding a design-rule-correct path between its terminals. If a path can be found, the next connection is attempted. In case of blockage, previously made connections are removed and re-routed later.

Contour uses a breadth-first routing algorithm based on a single principle; *postpone arbitrary choices*. When such choices arise, such as "should the connection start with a wire or a via?", or "should we turn left or right around this obstacle?", *all* the alternatives are explicitly represented, maintained and propagated until there is enough information to discriminate between them. During early attempts to implement the router, we did not rigorously adhere to this principle, believing it to be too complicated or too costly to implement. The result was always a router that would surprise us by the paths it had chosen for some connections ("Why did it do *that*?"). Only when we finally eliminated all arbitrary choices in the algorithm did the router choose exactly the paths a person would.

3.5. Netlist Hierarchy Equals Layout Hierarchy?

Is the hierarchy described by the annotated netlist strictly the same as the hierarchy of the cells which are laid out? Yes, but with a single exception. The layout method for any cell may choose to flatten the netlist for its cell selectively. For the purposes of layout only, intermediate cells of netlist may be exploded, and their instances promoted to be instances of the top-level cell. Two uses of this flattening are in the layout methods *StdCell* and *ReadMagic*. *StdCell*, the "standard-cell" layout method for random logic, removes some, but not all layers of its cell's hierarchy. It flattens the netlist until it contains only gate-level instances. So for instance, a 10-bit register would be considered as ten independent bit layouts by *StdCell*. *ReadMagic* flattens its cell completely until it contains only bare devices; all intervening cells in the hierarchy are ignored for this purpose. This flattened netlist can then be matched against the extracted hand-drawn layout. Allowing *ReadMagic* to be used for complex cells is a kind of escape mechanism from the failure of our automatic placement and routing; when all else fails, draw it by hand. We have never used it this way to date.

Even though the netlist and layout hierarchies must be similar, we do not believe that this results in an unnatural logical partitioning of circuits. In general, a piece of logic is defined as a cell in our system whenever one or more of the following conditions holds:

- *logical*: the number of wires in the cell's interface is much smaller than the number of wires used to connect the instances of its subcells.
- *sharing*: there will be many instances of this cell in the circuit.
- *layout*: the cell represents a natural unit of layout.
- *simulation*: the cell carries a functional model for some level of simulation.

Very often, these conditions occur naturally together. A frequently used piece of logic with a small interface is likely to be a natural unit of layout. When these conditions are kept in mind at the start of a design, a netlist fulfilling the requirements of layout and simulation can be built just as easily as any other. Imposing these conditions after the fact can be painful.

4. Results

Figure 8 shows the floating-point divider from BIPS-1, a 4,000,000-device BiCMOS microprocessor. The divider contains 20,000 transistors, and occupies about 5% of the chip area. It consists of 118 unique cells, 97 of which are synthesized leaf cells, and 21 of which are composite cells such as 64-bit registers or blocks of control logic. The large block on the left and center is the mantissa data path. It consists of multiplexors and registers laid out by abutment (the long dark horizontal bands) and three carry-lookahead adders assembled by corner alignment (the areas with irregular lower borders are the carry-lookahead trees). The exponent data path, with one smaller adder, is on the upper right. The sequencer is the irregular block on the lower right, laid out by *StdCell*. Assembly of the top-level cell was by corner alignment. The divider was designed, simulated, laid out, and tuned for performance by two people in two months, and survived several design-rule changes without further modification. Layout of the divider takes 10 minutes on a DEC 3000/800 Alpha workstation.

Recursive layout generation has permitted a small team to design large chips. It was used to construct BIPS-0, a bipolar processor [9], shown in figure 9. The entire design team for this chip was twelve people, including two mechanical engineers and three people working entirely on CAD. The layout generation of this complete 700,000-device circuit took ten hours with no manual intervention. This allowed one complete iteration of the design per day.

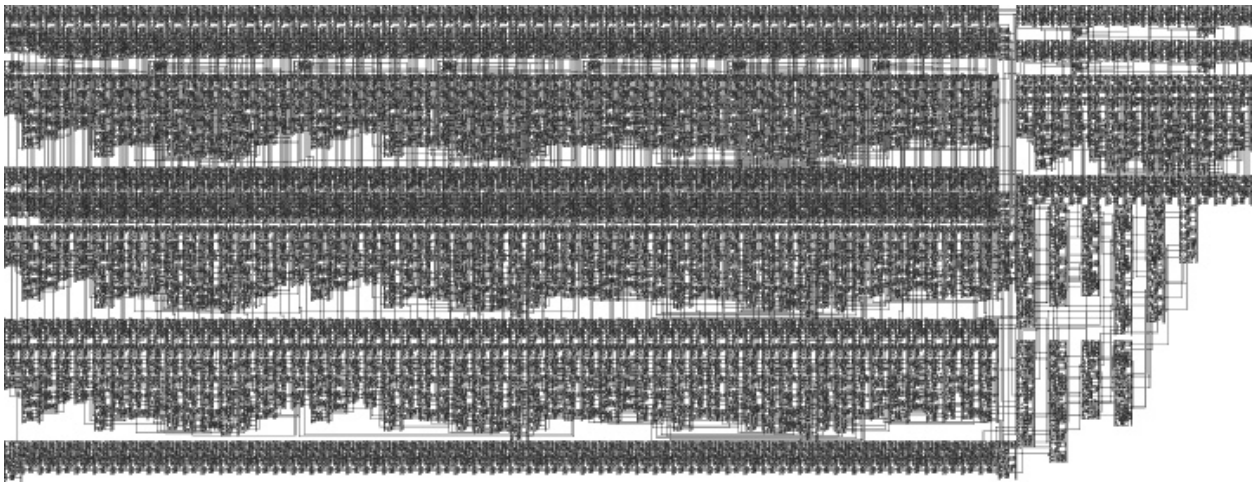

Figure 8: The BIPS-1 64-bit Floating-Point Divider

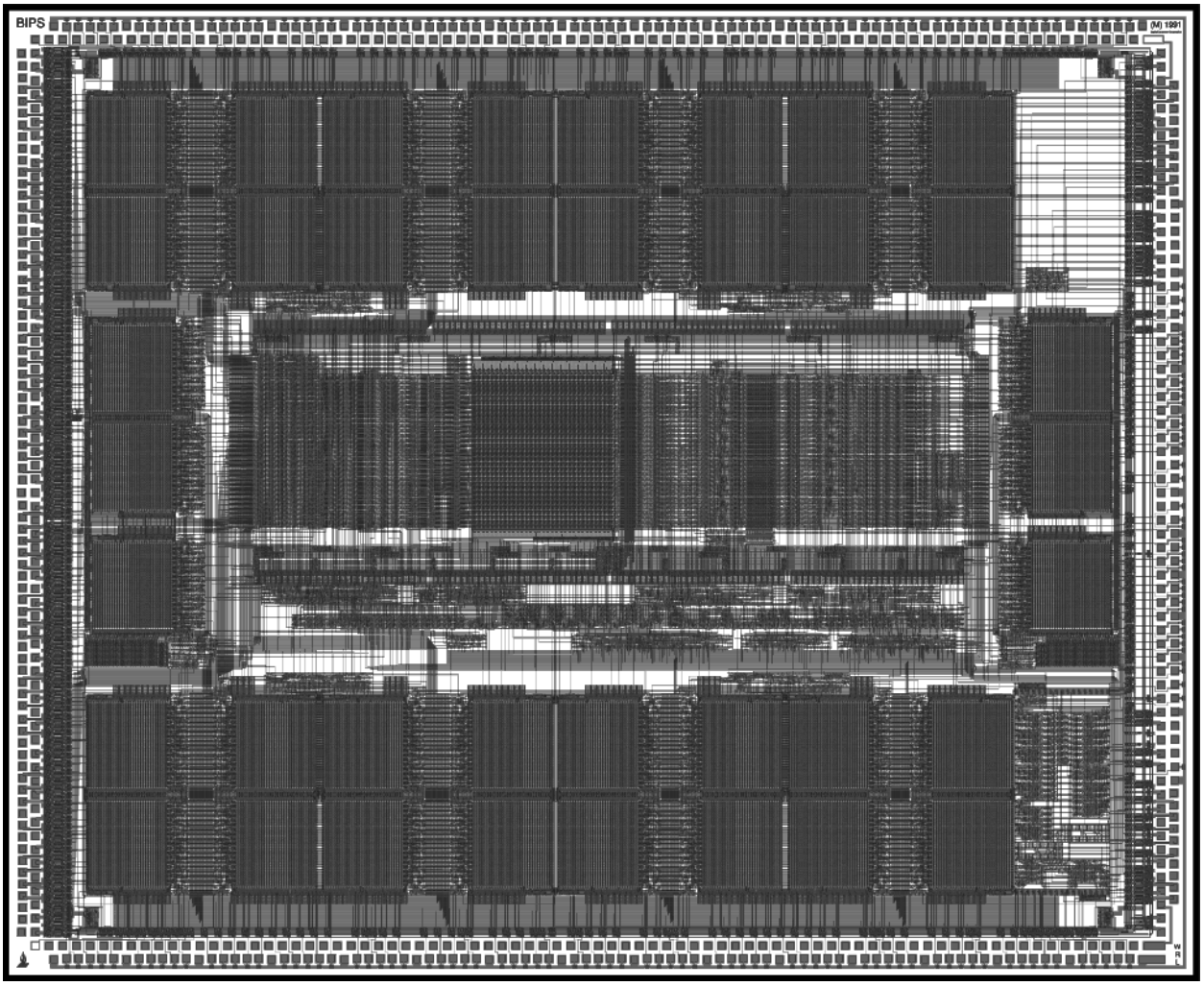

Figure 9: The BIPS-0 Microprocessor

References

[1] R.M. Ayres. *VLSI: Silicon Compilation and the Art of Automatic Microchip Design.* Prentice-Hall, 1983.

[2] R. Barth, L. Monier, B. Serlet. Patchwork: Layout From Schematic Annotations. In *25th Design Automation Conference*, pages 250-255. Sydney, June, 1988.

[3] J. Dion. *Fast Printed Circuit Board Routing*. WRL Research Report 88/1, Digital Equipment Western Research Laboratory, 1988.

[4] J. Dion, L.M. Monier. CONTOUR: A Tile-Based Gridless Router. *submitted to ACM/IEEE Design Automation Conference* , 1995.

[5] C. Ebeling, O. Zajicek. Validating VLSI Layout by Wirelist Comparison. In *IEEE International Conference on Computer-Aided Design*, pages 172-173. 1983.

[6] R.N. Mayo, H. Touati. Boolean Matching for Full-Custom ECL Gates. In *Proc. IEEE/ACM International Conference on Computer-Aided Design*, pages 472-477. November, 1993.

[7] D. Hightower. A Solution to Line Routing Problems on the Continuous Plane. *Proc. Design Automation Workshop* :1-24, 1969.

[8] N.P. Jouppi. Timing Analysis and Performance Improvement of MOS VLSI Designs. *IEEE Transactions on Computer Aided Design* 6(4):650-665, 1987.

[9] N.P. Jouppi, P. Boyle, J. Dion, M.J. Doherty, A. Eustace, R.W. Haddad, R. Mayo, S. Menon, L.M. Monier, D. Stark, S. Turrini, J.L. Yang, W.R. Hamburgen, J.S. Fitch, R. Kao. A 300-MHz 115-W 32-b Bipolar ECL Microprocessor. In *IEEE Journal of Solid-State Circuits.* November, 1993.

[10] R. Kao, R. Alverson, M. Horowitz, D. Stark. Bisim: A Simulator for Custom ECL Circuits. In *IEEE International Conference on Computer-Aided Design*, pages 62-65. Santa Clara, California, November, 1988.

[11] J.M. Kleinhans, G. Sigl, F.M. Johannes, K.J. Antreich. GORDIAN: VLSI Placement by Quadratic Programming and Slicing Optimization. *IEEE Transactions on Computer-aided Design* 10(3):356-365, March, 1991.

[12] R.N. Mayo, M.H. Arnold, W.S. Scott, D. Stark, G.T. Hamachi. *1990 DECWRL/Livermore Magic Release*. WRL Research Report 90/7, Digital Equipment Western Research Laboratory, 1990. see also: http://www.research.digital.com/wrl/magic/magic.html.

[13] E.F. Moore. Shortest Path Through a Maze. In *Annals of the Computation Laboratory of Harvard University*, pages 285-292. Harvard Univ. Press, Cambridge Mass., 1959.

[14] J.K. Ousterhout. Corner Stitching: A Data Structuring Technique for VLSI Layout Tools. *IEEE Transactions on Computer-Aided Design* CAD-3(1):87-89, January, 1984.

[15] J. Ousterhout, G. Hamachi, R. Mayo, W. Scott, and G.S. Taylor. The Magic VLSI Layout System. *IEEE Design and Test of Computers* 2(1):19-30, February, 1985.

[16] D. Stark. *Analysis of Power Supply Networks in VLSI Circuits*. WRL Research Report 91/3, Digital Equipment Western Research Laboratory, 1991.

WRL Research Reports

‘‘Titan System Manual.’’
Michael J. K. Nielsen.
WRL Research Report 86/1, September 1986.

‘‘Global Register Allocation at Link Time.’’
David W. Wall.
WRL Research Report 86/3, October 1986.

‘‘Optimal Finned Heat Sinks.’’
William R. Hamburgen.
WRL Research Report 86/4, October 1986.

‘‘The Mahler Experience: Using an Intermediate Language as the Machine Description.’’
David W. Wall and Michael L. Powell.
WRL Research Report 87/1, August 1987.

‘‘The Packet Filter: An Efficient Mechanism for User-level Network Code.’’
Jeffrey C. Mogul, Richard F. Rashid, Michael J. Accetta.
WRL Research Report 87/2, November 1987.

‘‘Fragmentation Considered Harmful.’’
Christopher A. Kent, Jeffrey C. Mogul.
WRL Research Report 87/3, December 1987.

‘‘Cache Coherence in Distributed Systems.’’
Christopher A. Kent.
WRL Research Report 87/4, December 1987.

‘‘Register Windows vs. Register Allocation.’’
David W. Wall.
WRL Research Report 87/5, December 1987.

‘‘Editing Graphical Objects Using Procedural Representations.’’
Paul J. Asente.
WRL Research Report 87/6, November 1987.

‘‘The USENET Cookbook: an Experiment in Electronic Publication.’’
Brian K. Reid.
WRL Research Report 87/7, December 1987.

‘‘MultiTitan: Four Architecture Papers.’’
Norman P. Jouppi, Jeremy Dion, David Boggs, Michael J. K. Nielsen.
WRL Research Report 87/8, April 1988.

‘‘Fast Printed Circuit Board Routing.’’
Jeremy Dion.
WRL Research Report 88/1, March 1988.

‘‘Compacting Garbage Collection with Ambiguous Roots.’’
Joel F. Bartlett.
WRL Research Report 88/2, February 1988.

‘‘The Experimental Literature of The Internet: An Annotated Bibliography.’’
Jeffrey C. Mogul.
WRL Research Report 88/3, August 1988.

‘‘Measured Capacity of an Ethernet: Myths and Reality.’’
David R. Boggs, Jeffrey C. Mogul, Christopher A. Kent.
WRL Research Report 88/4, September 1988.

‘‘Visa Protocols for Controlling Inter-Organizational Datagram Flow: Extended Description.’’
Deborah Estrin, Jeffrey C. Mogul, Gene Tsudik, Kamaljit Anand.
WRL Research Report 88/5, December 1988.

‘‘SCHEME->C A Portable Scheme-to-C Compiler.’’
Joel F. Bartlett.
WRL Research Report 89/1, January 1989.

‘‘Optimal Group Distribution in Carry-Skip Adders.’’
Silvio Turrini.
WRL Research Report 89/2, February 1989.

‘‘Precise Robotic Paste Dot Dispensing.’’
William R. Hamburgen.
WRL Research Report 89/3, February 1989.

‘‘Simple and Flexible Datagram Access Controls for Unix-based Gateways.’’
Jeffrey C. Mogul.
WRL Research Report 89/4, March 1989.

‘‘Spritely NFS: Implementation and Performance of Cache-Consistency Protocols.’’
V. Srinivasan and Jeffrey C. Mogul.
WRL Research Report 89/5, May 1989.

‘‘Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines.’’
Norman P. Jouppi and David W. Wall.
WRL Research Report 89/7, July 1989.

‘‘A Unified Vector/Scalar Floating-Point Architecture.’’
Norman P. Jouppi, Jonathan Bertoni, and David W. Wall.
WRL Research Report 89/8, July 1989.

‘‘Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU.’’
Norman P. Jouppi.
WRL Research Report 89/9, July 1989.

‘‘Integration and Packaging Plateaus of Processor Performance.’’
Norman P. Jouppi.
WRL Research Report 89/10, July 1989.

‘‘A 20-MIPS Sustained 32-bit CMOS Microprocessor with High Ratio of Sustained to Peak Performance.’’
Norman P. Jouppi and Jeffrey Y. F. Tang.
WRL Research Report 89/11, July 1989.

‘‘The Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance.’’
Norman P. Jouppi.
WRL Research Report 89/13, July 1989.

‘‘Long Address Traces from RISC Machines: Generation and Analysis.’’
Anita Borg, R.E.Kessler, Georgia Lazana, and David W. Wall.
WRL Research Report 89/14, September 1989.

‘‘Link-Time Code Modification.’’
David W. Wall.
WRL Research Report 89/17, September 1989.

‘‘Noise Issues in the ECL Circuit Family.’’
Jeffrey Y.F. Tang and J. Leon Yang.
WRL Research Report 90/1, January 1990.

‘‘Efficient Generation of Test Patterns Using Boolean Satisfiablilty.’’
Tracy Larrabee.
WRL Research Report 90/2, February 1990.

‘‘Two Papers on Test Pattern Generation.’’
Tracy Larrabee.
WRL Research Report 90/3, March 1990.

‘‘Virtual Memory vs. The File System.’’
Michael N. Nelson.
WRL Research Report 90/4, March 1990.

‘‘Efficient Use of Workstations for Passive Monitoring of Local Area Networks.’’
Jeffrey C. Mogul.
WRL Research Report 90/5, July 1990.

‘‘A One-Dimensional Thermal Model for the VAX 9000 Multi Chip Units.’’
John S. Fitch.
WRL Research Report 90/6, July 1990.

‘‘1990 DECWRL/Livermore Magic Release.’’
Robert N. Mayo, Michael H. Arnold, Walter S. Scott, Don Stark, Gordon T. Hamachi.
WRL Research Report 90/7, September 1990.

‘‘Pool Boiling Enhancement Techniques for Water at Low Pressure.’’
Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.
WRL Research Report 90/9, December 1990.

‘‘Writing Fast X Servers for Dumb Color Frame Buffers.’’
Joel McCormack.
WRL Research Report 91/1, February 1991.

‘‘A Simulation Based Study of TLB Performance.’’
J. Bradley Chen, Anita Borg, Norman P. Jouppi.
WRL Research Report 91/2, November 1991.

‘‘Analysis of Power Supply Networks in VLSI Circuits.’’
Don Stark.
WRL Research Report 91/3, April 1991.

‘‘TurboChannel T1 Adapter.’’
David Boggs.
WRL Research Report 91/4, April 1991.

‘‘Procedure Merging with Instruction Caches.’’
Scott McFarling.
WRL Research Report 91/5, March 1991.

‘‘Don’t Fidget with Widgets, Draw!.’’
Joel Bartlett.
WRL Research Report 91/6, May 1991.

‘‘Pool Boiling on Small Heat Dissipating Elements in Water at Subatmospheric Pressure.’’
Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.
WRL Research Report 91/7, June 1991.

‘‘Incremental, Generational Mostly-Copying Garbage Collection in Uncooperative Environments.’’
G. May Yip.
WRL Research Report 91/8, June 1991.

‘‘Interleaved Fin Thermal Connectors for Multichip Modules.’’
William R. Hamburgen.
WRL Research Report 91/9, August 1991.

‘‘Experience with a Software-defined Machine Architecture.’’
David W. Wall.
WRL Research Report 91/10, August 1991.

‘‘Network Locality at the Scale of Processes.’’
Jeffrey C. Mogul.
WRL Research Report 91/11, November 1991.

‘‘Cache Write Policies and Performance.’’
Norman P. Jouppi.
WRL Research Report 91/12, December 1991.

‘‘Packaging a 150 W Bipolar ECL Microprocessor.’’
William R. Hamburgen, John S. Fitch.
WRL Research Report 92/1, March 1992.

‘‘Observing TCP Dynamics in Real Networks.’’
Jeffrey C. Mogul.
WRL Research Report 92/2, April 1992.

‘‘Systems for Late Code Modification.’’
David W. Wall.
WRL Research Report 92/3, May 1992.

‘‘Piecewise Linear Models for Switch-Level Simulation.’’
Russell Kao.
WRL Research Report 92/5, September 1992.

‘‘A Practical System for Intermodule Code Optimization at Link-Time.’’
Amitabh Srivastava and David W. Wall.
WRL Research Report 92/6, December 1992.

‘‘A Smart Frame Buffer.’’
Joel McCormack & Bob McNamara.
WRL Research Report 93/1, January 1993.

‘‘Recovery in Spritely NFS.’’
Jeffrey C. Mogul.
WRL Research Report 93/2, June 1993.

‘‘Tradeoffs in Two-Level On-Chip Caching.’’
Norman P. Jouppi & Steven J.E. Wilton.
WRL Research Report 93/3, October 1993.

‘‘Unreachable Procedures in Object-oriented Programing.’’
Amitabh Srivastava.
WRL Research Report 93/4, August 1993.

‘‘An Enhanced Access and Cycle Time Model for On-Chip Caches.’’
Steven J.E. Wilton and Norman P. Jouppi.
WRL Research Report 93/5, July 1994.

‘‘Limits of Instruction-Level Parallelism.’’
David W. Wall.
WRL Research Report 93/6, November 1993.

‘‘Fluoroelastomer Pressure Pad Design for Microelectronic Applications.’’
Alberto Makino, William R. Hamburgen, John S. Fitch.
WRL Research Report 93/7, November 1993.

‘‘A 300MHz 115W 32b Bipolar ECL Microprocessor.’’
Norman P. Jouppi, Patrick Boyle, Jeremy Dion, Mary Jo Doherty, Alan Eustace, Ramsey Haddad, Robert Mayo, Suresh Menon, Louis Monier, Don Stark, Silvio Turrini, Leon Yang, John Fitch, William Hamburgen, Russell Kao, and Richard Swan.
WRL Research Report 93/8, December 1993.

‘‘Link-Time Optimization of Address Calculation on a 64-bit Architecture.’’
Amitabh Srivastava, David W. Wall.
WRL Research Report 94/1, February 1994.

‘‘ATOM: A System for Building Customized Program Analysis Tools.’’
Amitabh Srivastava, Alan Eustace.
WRL Research Report 94/2, March 1994.

‘‘Complexity/Performance Tradeoffs with Non-Blocking Loads.’’
Keith I. Farkas, Norman P. Jouppi.
WRL Research Report 94/3, March 1994.

‘‘A Better Update Policy.’’
Jeffrey C. Mogul.
WRL Research Report 94/4, April 1994.

‘‘Boolean Matching for Full-Custom ECL Gates.’’
Robert N. Mayo, Herve Touati.
WRL Research Report 94/5, April 1994.

‘‘Software Methods for System Address Tracing: Implementation and Validation.’’
J. Bradley Chen, David W. Wall, and Anita Borg.
WRL Research Report 94/6, September 1994.

‘‘Performance Implications of Multiple Pointer Sizes.’’
Jeffrey C. Mogul, Joel F. Bartlett, Robert N. Mayo, and Amitabh Srivastava.
WRL Research Report 94/7, December 1994.

‘‘How Useful Are Non-blocking Loads, Stream Buffers, and Speculative Execution in Multiple Issue Processors?.’’
Keith I. Farkas, Norman P. Jouppi, and Paul Chow.
WRL Research Report 94/8, December 1994.

‘‘Recursive Layout Generation.’’
Louis M. Monier, Jeremy Dion.
WRL Research Report 95/2, March 1995.

‘‘Contour: A Tile-based Gridless Router.’’
Jeremy Dion, Louis M. Monier.
WRL Research Report 95/3, March 1995.

‘‘The Case for Persistent-Connection HTTP.’’
Jeffrey C. Mogul.
WRL Research Report 95/4, May 1995.

‘‘Network Behavior of a Busy Web Server and its Clients.’’
Jeffrey C. Mogul.
WRL Research Report 95/5, June 1995.

WRL Technical Notes

‘‘TCP/IP PrintServer: Print Server Protocol.’’
Brian K. Reid and Christopher A. Kent.
WRL Technical Note TN-4, September 1988.

‘‘TCP/IP PrintServer: Server Architecture and Implementation.’’
Christopher A. Kent.
WRL Technical Note TN-7, November 1988.

‘‘Smart Code, Stupid Memory: A Fast X Server for a Dumb Color Frame Buffer.’’
Joel McCormack.
WRL Technical Note TN-9, September 1989.

‘‘Why Aren’t Operating Systems Getting Faster As Fast As Hardware?’’
John Ousterhout.
WRL Technical Note TN-11, October 1989.

‘‘Mostly-Copying Garbage Collection Picks Up Generations and C++.’’
Joel F. Bartlett.
WRL Technical Note TN-12, October 1989.

‘‘The Effect of Context Switches on Cache Performance.’’
Jeffrey C. Mogul and Anita Borg.
WRL Technical Note TN-16, December 1990.

‘‘MTOOL: A Method For Detecting Memory Bottlenecks.’’
Aaron Goldberg and John Hennessy.
WRL Technical Note TN-17, December 1990.

‘‘Predicting Program Behavior Using Real or Estimated Profiles.’’
David W. Wall.
WRL Technical Note TN-18, December 1990.

‘‘Cache Replacement with Dynamic Exclusion’’
Scott McFarling.
WRL Technical Note TN-22, November 1991.

‘‘Boiling Binary Mixtures at Subatmospheric Pressures’’
Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.
WRL Technical Note TN-23, January 1992.

‘‘A Comparison of Acoustic and Infrared Inspection Techniques for Die Attach’’
John S. Fitch.
WRL Technical Note TN-24, January 1992.

‘‘TurboChannel Versatec Adapter’’
David Boggs.
WRL Technical Note TN-26, January 1992.

‘‘A Recovery Protocol For Spritely NFS’’
Jeffrey C. Mogul.
WRL Technical Note TN-27, April 1992.

‘‘Electrical Evaluation Of The BIPS-0 Package’’
Patrick D. Boyle.
WRL Technical Note TN-29, July 1992.

‘‘Transparent Controls for Interactive Graphics’’
Joel F. Bartlett.
WRL Technical Note TN-30, July 1992.

‘‘Design Tools for BIPS-0’’
Jeremy Dion & Louis Monier.
WRL Technical Note TN-32, December 1992.

‘‘Link-Time Optimization of Address Calculation on a 64-Bit Architecture’’
Amitabh Srivastava and David W. Wall.
WRL Technical Note TN-35, June 1993.

‘‘Combining Branch Predictors’’
Scott McFarling.
WRL Technical Note TN-36, June 1993.

‘‘Boolean Matching for Full-Custom ECL Gates’’
Robert N. Mayo and Herve Touati.
WRL Technical Note TN-37, June 1993.

‘‘Ramonamap - An Example of Graphical Groupware’’
Joel F. Bartlett.
WRL Technical Note TN-43, December 1994.

‘‘Circuit and Process Directions for Low-Voltage Swing Submicron BiCMOS’’
Norman P. Jouppi, Suresh Menon, and Stefanos Sidiropoulos.
WRL Technical Note TN-45, March 1994.

‘‘Experience with a Wireless World Wide Web Client’’
Joel F. Bartlett.
WRL Technical Note TN-46, March 1995.

‘‘I/O Component Characterization for I/O Cache Designs’’
Kathy J. Richardson.
WRL Technical Note TN-47, April 1995.

‘‘Attribute caches’’
Kathy J. Richardson, Michael J. Flynn.
WRL Technical Note TN-48, April 1995.

‘‘Operating Systems Support for Busy Internet Servers’’
Jeffrey C. Mogul.
WRL Technical Note TN-49, May 1995.