CMU-CS-82-147

Two Papers on Circuit Extraction

Anoop Gupta

Robert W. Hon

15 December 1982

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539.

ACE: A Circuit Extractor[1]

Anoop Gupta
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

This paper describes the design, implementation and performance of a flat edge-based circuit extractor for NMOS circuits. The extractor is able to work on large and complex designs, it can handle arbitrary geometry, and outputs a comprehensive wirelist. Measurements show that the run time of the edge-based algorithm used is linear in size of the circuit, with low implementation overheads. The extractor is capable of analyzing a circuit with 20,000 transistors in less than 30 minutes of CPU time on a VAX 11/780. The high performance of the extractor has changed the role that a circuit extractor played in the design process, as it is now possible to extract a chip a number of times during the same session.

1. Introduction

The majority of layout design is done using manual methods, and is susceptible to human errors. If the designer has not verified the logic design (which is often the case in university environments), any errors in the logic design will also be translated into the layout.

Circuit extraction is the first step in eliminating layout errors and verifying circuit design. The input to a circuit extractor is the layout of a chip. The output consists of a network of electrical devices represented by the layout, and is called a wirelist. The wirelist can be fed to other CAD tools to verify the correctness of the circuit. Logic simulators help validate the logical correctness; circuit simulators help check for timing errors, overloading, and performance characteristics of the circuit. A static checker performs ratio checks, detects malformed transistors, and checks for signals that are stuck at logical 0 or 1. If a circuit's schematic diagram is available to the designer, it can be compared to the extracted circuit: if the two are equivalent, the layout corresponds to the original circuit.

The next section gives an overview of the circuit extractor (ACE) in the context of existing approaches to

[1]This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539.

circuit extraction. Section 3 has a detailed description of the algorithm used by ACE. Section 4 analyzes the expected and worst case time and space complexity of the algorithms used by ACE. Section 5 discusses the overall performance, and the final section is devoted to conclusions.

2. ACE and existing approaches to circuit extraction

Circuit extractors vary along a number of dimensions. They can differ in:

- their use of the structural information present in the description of a chip,
- the constraints they impose on the designer,
- the amount of detail present in their outputs, and
- the algorithms they use to locate devices and find connectivity.

ACE is a flat extractor in that it works on a fully-instantiated description of the chip. It does not make use of the structure and repetition present in the layout, and consequently, it must analyze every cell instance, even if the same cell has been analyzed before. In contrast, a hierarchical extractor analyzes identical cells only once [7, 10, 13, 15]. Even hierarchical extractors eventually call a flat extractor to analyze geometry inside the leaf cells, and a fast flat extractor is an advantage.

Some extractors [1] require the designer to indicate the transistors and their connection points in the layout; most extractors, including ACE, do not. Similarly, some extractors allow only manhattan geometry (rectangles with their sides parallel to the coordinate axes) in the layout. This makes the extraction algorithm simpler at the cost of constraining the designer. ACE copes with arbitrary polygons in the layout, but with somewhat less efficiency than manhattan geometry.

Extractors vary in the amount of information contained in their output. A simple node extractor provides connectivity information just sufficient for logic simulation; a sophisticated extractor provides additional information about transistor ratios, and net capacitance and resistance (a net is an electrically conducting path that does not include a transistor). ACE provides connectivity information along with data indicating the location, length and width of transistors, the location and area of capacitors, and the location and names for nets. ACE does not directly compute the capacitance and resistance for nets and devices, as it was undesirable to embed any fixed notion of a circuit model into the extractor code. It is possible, however, to obtain a list of geometry that constitutes each net and device. This information is enough for a post-processing program to compute capacitances and resistances.

Finally, extractors vary on the basis of the algorithms used to find connectivity. The fixed grid raster-scan based algorithm is popular [2]. In this algorithm the chip is examined in a raster-scan order (left to right, top to bottom) looking through an L-shaped window containing three raster elements. Using only this information it is possible to follow connectivity and locate transistors. The main advantage is simplicity [11], but a lot of time is wasted scanning over grid squares where no information is to be gained. It further requires that all geometry be aligned with the grid. ACE uses an edge-based algorithm. A scan line is moved from the top to the bottom of the chip, pausing at points corresponding to the top or bottom edges of pieces of geometry. Conceptually, this divides the chip into a number of horizontal strips where the state within the strip does not change in the vertical direction. Change in state occurs only at the interface between two strips. At the interface the extractor steps through the list of boxes touching the scanline, and makes the necessary updates to state. One of the main advantages of this algorithm over the raster based algorithm is that empty space and large device structures are extracted easily. Although the worst-case time complexity of this algorithm is $O(N^2)$, where N is the number of boxes in the artwork, the observed complexity over a large number of chips is linear in N.

3. The Algorithm

The input to the ACE program is the artwork of a chip expressed in CIF (Caltech Intermediate Form) [9]. Its output is a wirelist consisting of a list of transistors and their connectivity.

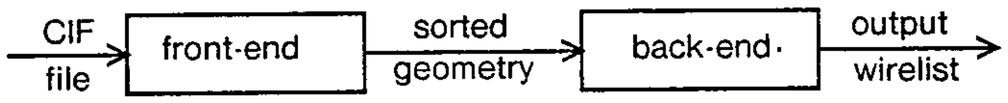

Figure 3-1: Overall organization of the extractor

It is convenient to view the extractor as consisting of two parts, a front-end and a back-end. The front-end[2] consists of routines which parse, instantiate and sort the CIF file. The front-end builds an internal database so that geometry can be output in order from top to bottom. Before being output, non-manhattan geometry is split into a number of small aligned boxes that approximate the original object. After the front-end has built the database, the back-end is initialized and the scanline is set to the top of the chip.

The back-end consists of routines that receive sorted geometry from the front-end and find transistors and connectivity information. A brief sketch of the algorithm used by the back-end is shown below.

1. Set scanline to top of chip.

[2]The front-end was written by Bob Hon at CMU

```
2. WHILE scanline > bottom of chip DO

    a. Fetch geometry from the front-end whose top coincides with the scanline, and sort each
       incoming box by x into a newGeometry list corresponding to its layer.

    b. FOR each layer DO insert new geometry into active geometry list.

    c. Compute devices.

    d. Compute y value for next scanline.

3. Output devices and nets.
```

Figure 3-2: Algorithm for the back-end

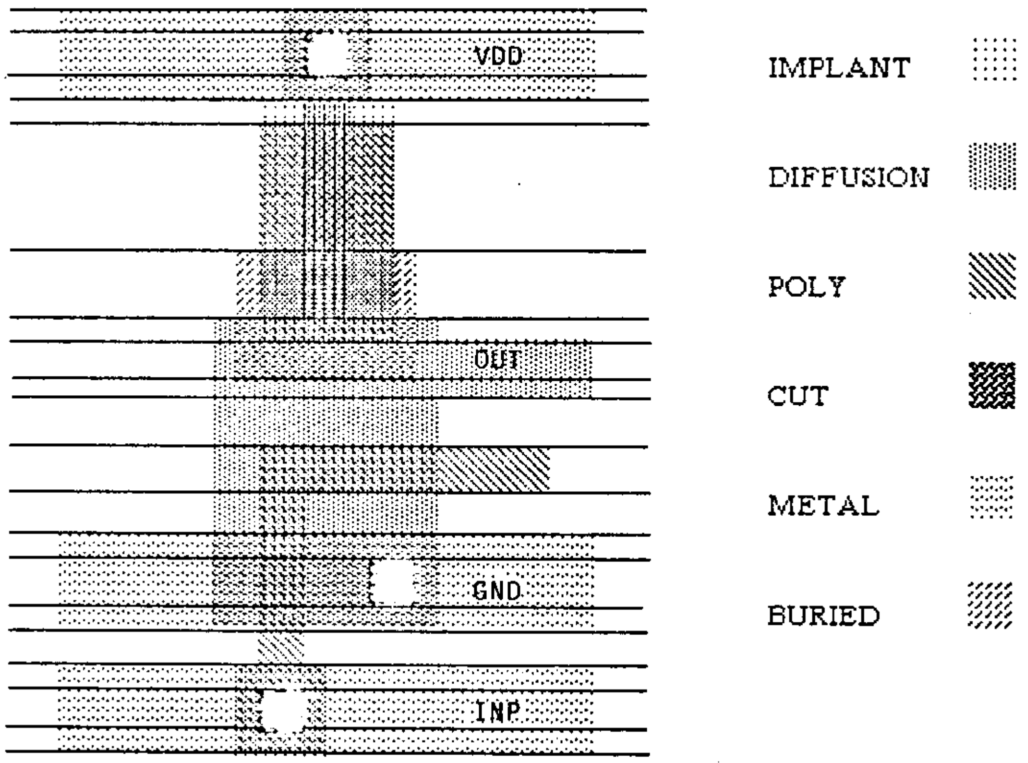

Figure 3-3: An Inverter

Within the back-end, most work is done in a loop where a scanline moves down from the top to the bottom of the chip. (The solid lines in Figure 3-3 show the positions of the scanline while extracting the inverter.) The action taken on each iteration of the loop can be divided into four steps:

- All geometry whose top coincides with the scanline is fetched from the front-end. Incoming geometry is sorted into one of a set of newGeometry lists (one list per layer) by x-coordinate of the left side of the box. Adjacent or overlapping boxes on the same layer are merged together into one box. If one of the boxes is taller than the other, the taller box is split into two before merging. Its lower portion is sent to a temporary buffer for recall when the scanline reaches its top.

- For each of the layers, boxes in the newGeometry list are inserted into the active geometry list for that layer (Geometry that intersects the current scanline is said to be active). The program maintains a sorted list of active geometry for each of the layers. If a box from the newGeometry list abuts or overlaps a box in the active geometry list the two are merged together. Otherwise that box is inserted independently and a new net is created for it. All geometry whose bottom coincides with the scanline exits the active list at this step.

- To locate devices the extractor simultaneously traverses the active lists corresponding to the four interacting layers (diffusion, poly, buried and implant). The lists are scanned from left to right, and overlap between the geometry on the four layers is computed. An overlap between diffusion and poly accompanied by the absence of buried results in a potential transistor. The presence of implant determines the type of transistor.

- The position of the next scanline is determined, and the loop iterates. The y-value for the next scanline is the larger of (i) the y-value of the top-edge of the next box output by the front-end to the back-end (recall that the front-end outputs geometry sorted from top to bottom), and (ii) the largest y-value for the bottom-edge of a box in the active list.

A wirelist containing a list of all transistors and nets is output on termination of the loop. Associated with each transistor are identities of its source, drain and gate nets, its type, length, width and location. Associated with each net is its location and all of its user defined names [12]. User options exist to force the extractor to output the geometry associated with each net and device. Under normal operation this is suppressed. Figure 3-4 shows the wirelist for the inverter in Figure 3-3. The format used for the wirelist was developed by Ed Frank, Carl Ebeling, and Robert Sproull at CMU [5]. The format is easy to parse and extend because of its LISP like syntax.

The length and width of a transistor is determined by the shape of its channel (the active region of the transistor), and the contour along which its source and drain nets touch the channel. Since both the shape of channel and the contours can be arbitrarily complex, it is not easy to determine the length and width. ACE uses a simple algorithm that works well for most transistors encountered in practice. For the purpose of this algorithm, the source edge length of a transistor is defined to be the the length of the perimeter along which the source net and the channel touch. The width of the transistor is then computed as the mean of the source and drain edge lengths. The length of the transistor is computed as the area of the channel divided by the width. Results of the use of this algorithm can be seen in the wirelist of Figure 3-4. (The user is free to obtain transistor geometry and compute length and width by a different technique if he wishes.)

```
(DefPart "inverter.cif"
 (DefPart nEnh (Export Source Gate  Drain))
(DefPart nDep (Export Source  Gate  Drain))

(Part nEnh (InstName D0)  (Location -800 -400)
 (T Gate N9)  (T Source N5)  (T Drain N11)
 (Channel (Length 400)  (Width 2800)
  ( CIF "
   L NX; B L400 W1200 C-600 -1400;  L NX; B L1600 W400 C0 -600; ")))

(Part nDep (InstName D1)(Location -400 2800)
 (T Gate N5)  (T Source N2)  (T Drain N5)
 (Channel (Length 1400)  (Width 400)
  ( CIF "   L NX; B L400 W1400 C-200 2100 ;  ")))

(Net N2    VDD    (Location -2600 3800)
 ( CIF "
   L NM; B L4800 W800 C-200 3400;  L ND; B L400 W200 C-200 2900;
   L ND; B L800 W800 C-200 3400;   L NC; B L400 W400 C-200 3400;   "))

(Net N5    OUT    (Location -800 2800)
 ( CIF "
   L NP; B L1200 W2000 C-200 1800; L ND; B L400 W1600 C-1000 -1200;
   L ND; B L2000 W400 C-200 -200;  L ND; B L3400 W600 C500 300;
   L ND; B L2000 W200 C-200 700;   L ND; B L400 W600 C-200 1100;  "))

(Net N9    INP    (Location -800 -400)
 ( CIF "
   L NP; B L800 W800 C-600 -2800;  L NP; B L400 W1600 C-600 -1600;
   L NP; B L2600 W400 C500 -600;   L NM; B L4800 W800 C-200 -2800;
   L NC; B L400 W400 C-600 -2800;   "))

(Net N11    GND    (Location -400 -800)
 ( CIF "
   L ND; B L1200 W1200 C200 -1400; L NM; B L4800 W800 C-200 -1600;
   L NC; B L400 W400 C400 -1600;   "))

(Local    N2  N5  N9  N11  ))
```

Figure 3-4: Wirelist for the inverter

4. Analysis of the algorithm

ACE is implemented in the C language [8], and runs on a VAX-11/780 under UNIX. A number of implementation decisions were influenced by the fact that the circuit extractor was to run on a VAX with a large virtual memory. Space was not considered to be at a premium, and it was frequently traded for savings in time. Care was taken, however, to preserve locality of addressing to prevent thrashing.

Time Complexity

The worst-case time complexity of the edge-based algorithm used by ACE is $O(N^2)$, where N is the number of boxes in the artwork. The worst case occurs when N horizontal poly lines intersect N vertical diffusion lines, forming a mesh with N^2 transistors. Since each of the N^2 transistors has to be found by the extractor, the complexity is at least N^2. The best that ACE can do is linear in N. This is because it must look at every box in the artwork at least once.

The time complexity of the front-end is determined by the sorting step, all other steps being linear in N. The front-end uses a tree-sort algorithm, which if everything is expanded to primitive geometry before sorting, takes $O(N \log N)$ expected time. However, the front-end does not expand everything to boxes before sorting, but instead makes use of the hierarchy present in the CIF specification of the chip, and recursively expands only those cells that intersect the current scanline. For example, if there exists a CIF symbol which lies completely below the scanline, the front-end does not have to expand that cell to determine that all geometry inside it is below the scanline. In this way the complete geometry of the chip is never instantiated (so never sorted) at the same time. The reduction in the number of items in the sorting tree reduces the complexity of the sorting step, thus reducing the complexity of the front-end. This dependence of front-end complexity on hierarchy present in the CIF description makes it very difficult to characterize it. The observed complexity of the front-end is linear in N (see Table 5-1), over a wide range of designs.

To perform an expected-time analysis for the back-end, it is necessary to have a model for the size and distribution of boxes in the artwork. The model used here is the simplest of a number of models proposed by Bentley, Haken, and Hon [3]. It assumes that in an N-rectangle design, the N rectangles are squares with edge length 7.6λ, uniformly distributed over a region $[0,8N^{1/2}\lambda]^2$. The constant λ is the size of the smallest feature resolvable by the implementation process. We further assume that the rectangles are aligned to λ boundaries, and that the total number of transistors in the circuit is proportional to N. Under these assumptions the expected number of boxes that intersect the scanline is $O(N^{1/2})$, and the expected number of stops that the scanline makes in going from top-to-bottom of the chip is $O(N^{1/2})$.

The complexity of the back-end is analyzed in context of the algorithm shown in Figure 3-2. Step 1 of the algorithm is trivial, and takes constant time. The time taken on each iteration of the loop in Step 2 is the sum of the time taken by Steps 2.a to 2.d. Step 2.a takes $O(N)$ time, because a simple insertion sort is used to place the incoming geometry into newGeometry lists. Insertion sort requires quadratic time, and since there are $O(N^{1/2})$ boxes to be inserted, the time taken is $O(N)$. The time spent in steps 2.b and 2.c is $O(N^{1/2})$, as this involves traversing down the lists of geometry that are $O(N^{1/2})$ long. The constant associated with the big-O of steps 2.b and 2.c is much larger than the constant associated with step 2.a, because the complexity of steps 2.b and 2.c is much more. Step 2.d is trivial, and takes constant time. Thus time taken by each iteration of the loop is $c_1 N + c_2 N^{1/2}$, where $c_1 \ll c_2$. Since the loop in step 2 is executed $O(N^{1/2})$ times (this is the expected number of stops made by the scanline), the total time taken by step 2 is $c_1 N^{3/2} + c_2 N$. Step 3 takes linear time as the total number of devices and nets in the circuit is $O(N)$. The time complexity of the back-end is then given by $c_1 N^{3/2} + c_3 N$, where c_3 is much greater than c_1.

The observed performance of ACE is linear in N over a wide range of chips, as shown by Table 5-1. This is not too surprising in light of the above discussion. The term containing $N^{3/2}$ can be made linear by using bin-sort instead of insertion-sort [3], but c_1 is so small that it has not been necessary to do so.

Space Complexity

ACE does not output any transistors or nets until the scanline has reached the bottom of the chip. This is because two nets that were earlier distinct can be merged after they have been output, causing the output to be in error. Thus ACE requires space at least linear in the number of transistors and nets in the circuit. Since $2N$ boxes can result in N^2 transistors (refer to the worst case time complexity analysis), the worst case space complexity is $O(N^2)$.

The expected space complexity of ACE can be calculated using the same model as was used for the time complexity. The expected space complexity for the front-end lies between $O(\log N)$ and $O(N)$ depending on the amount of hierarchy present in the layout description. The storage requirements for the geometry lists associated with the scanline is $O(N^{1/2})$, which is the expected number of boxes intersecting the scanline. Since the expected number of transistors and nets is $O(N)$, the space required for these is $O(N)$. Thus the overall expected space complexity of ACE is $O(N)$. This result corresponds to actual observations.

5. Performance

Table 5-1 shows the measured performance of ACE for a number of chips designed within the ARPA community. As the last column shows, the time complexity is linear with the number of boxes over a wide range of chips. This is consistent with the analysis carried out in the previous section.

Name	Devices	# of Boxes (in 1000's)	User + Sys Time (min:sec)	Devices/ Time (devs/sec)	# of Boxes/ Time (boxes/sec)
Cherry	881	7.4	1:05	13.69	113.84
Dchip	4884	50.7	10:12	8.00	82.84
Schip2	9473	109.0	18:12	8.69	99.81
Testram	20480	196.9	26:36	12.98	123.37
PSC	25521	251.5	41:14	10.32	101.68
Scheme81	32031	418.3	73:54	7.63	94.33
Riscb	42084	533.0	92:12	7.24	96.43

Table 5-1: Performance

The performance of ACE is compared to some other extractors in Table 5-2. Partlist [2, 14] is a run-encoded, raster-scan based extractor that was used prior to ACE at CMU. Cifplot [4] is a flat extractor provided by Berkeley. All timing measurements were made on a VAX-11/780.

chip	devices	ACE (min:sec)	Partlist (min:sec)	Cifplot (min:sec)
cherry	881	1:05	2:50	4:45
dchip	4884	10:12	18:34	46:21
schip2	9473	18:12	35:06	95:15
testram	20480	26:36	46:07	-
riscb	42084	96:43	-	-

Table 5-2: Comparison of Performance

The main reason for the better performance of ACE over other extractors is its edge-based approach. An edge-based extractor skips empty space and extracts large boxes at little cost. It does work only at the edges of a box as compared to a raster-based extractor which must visit each and every grid square spanned by the box. Since the average size of a box used in the layout is much larger than size of the grid square the savings are big. The coarse distribution of time over the extraction algorithm was found to be:

- 40% for parsing, interpreting and sorting the CIF file.
- 15% for entering new geometry into lists and updating the data structures.
- 20% for computing devices, nets, etc.
- 10% for storage allocation, input/output, and initialization.
- 15% Miscellaneous.

6. Conclusions

This paper presents the algorithms and performance of an edge-based extractor developed at CMU. The expected time complexity of the algorithms is observed to be linear in the number of boxes in the layout, and the measured performance is significantly better than the performance of other extractors.

An important measure of success for a design tool is its use by the design community. ACE is used extensively by the VLSI design group at CMU. As a result of its higher performance, it is not unusual to see a user with a 5,000 transistor chip go through a few iterations of extracting, simulating, and fixing bugs during a single two-hour session. This constitutes a major change from the way extractors have been used in the past.

The edge-based algorithms are well suited for hierarchical and incremental extractors. A modified version of ACE is used as a part of an experimental hierarchical extractor being developed at CMU [6]. The results obtained so far are very promising.

7. Acknowledgments

Robert Sproull suggested the original idea and algorithm to me. Edward Frank and Robert Hon helped me with useful suggestions and comments throughout the development of the program. I wish to thank Jon Bentley, HT Kung, and Hank Walker for careful reading of earlier versions of the manuscript. Finally, thanks go to all the users of the program, who helped discover the bugs in the program.

References

[1] Bryan Ackland and Neil Weste.
Functional Verification in an Interactive Symbolic IC Design Environment.
In *2nd. Caltech Conference on VLSI.* 1981.

[2] Clark Baker.
Artwork Analysis Tools for VLSI Circuits.
Master's thesis, M.I.T., 1980.

[3] Jon Louis Bentley, Dorothea Haken, and Robert Hon.
Statistics on VLSI Designs.
Technical Report, Carnegie-Mellon University, April, 1980.

[4] Daniel Fitzpatrick.
Circuit Analysis from CIF Layouts
Computer Science Division, University of California at Berkeley, 1981.

[5] Edward Frank, Carl Ebeling, and Robert Sproull.
Hierarchical Wirelist Format.
VLSI Document V085, Carnegie-Mellon University, 1981.

[6] Anoop Gupta and Robert Hon.
Two Papers on Circuit Extraction.
Technical Report, Carnegie-Mellon University, 1982.

[7] Robert Hon.
The Hierarchical Analysis of VLSI Designs.
VLSI Document V073, Carnegie-Mellon University, 1981.

[8] Brian W. Kernighan and Dennis M. Ritchie.
The C Programming Language.
Prentice-Hall, 1978.

[9] Carver Mead and Lynn Conway.
Introduction To VLSI Systems.
Addison-Wesley, 1980.

[10] Martin E.Newell and Daniel T. Fitzpatrick.
Exploiting Structure in Integrated Circuit Design Analysis.
In *Conference on Advanced Research in VLSI.* M.I.T., 1982.

[11] Larry Seiler.
A Hardware Assisted Design Rule Check Architecture.
In *19th Design Automation Conference.* 1982.

[12] Robert Sproull.
Names in CIF.
VLSI Document V062, Carnegie-Mellon University, 1980.

[13] Mike Tucker and Lou Scheffer.
A Constrained Design Methodology for VLSI.
VLSI Design, May/June, 1982.

[14] James Wendorf.
NMOS Circuit Partlist Extractor.
VLSI Document V047, Carnegie-Mellon University, 1980.

[15] Telle Whitney.
A Hierarchical Design-Rule Checking Algorithm.
Lambda Magazine, First Quarter, 1981.

HEXT: A Hierarchical Circuit Extractor[1]

Anoop Gupta
Robert W. Hon[2]
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

This paper describes the algorithms, implementation, and performance of a hierarchical circuit extractor for NMOS designs. The input to the circuit extractor is a description of the layout of the chip, and its output is a hierarchical wirelist describing the circuit. The extractor is divided into two parts, a front-end and a back-end. The front-end analyzes the CIF description of a layout and partitions it into a set of non-overlapping rectangular regions called *windows*; redundant windows are recognized and are extracted only once. The back-end analyzes each unique window found by the front-end. The back-end determines the electrical circuit represented by the window, and computes an interface that is later used to combine the window with others that are adjacent. The paper also presents a simple analysis of the expected performance of the algorithm, and the results of running the extractor on some real chip designs.

1. Introduction

Computer aided design (CAD) tools are used to design and debug VLSI circuits. A *circuit extractor* is one such tool. Circuit extractors work on a geometric description (i.e., a specification of the features that appear on the chip surface) of the design and find the equivalent electrical circuit. The circuit is represented as a list of transistors and their interconnections; this is called a *wirelist*. Once the wirelist for a chip is available, a number of other tools can be used to find properties of the circuit: logic simulators, circuit simulators, wirelist comparators, and static checkers all help to validate and debug the designed chip. A circuit extractor is run over the layout frequently during the design process, so good performance is desirable. Most extractors operate on a list of all the geometric shapes on a chip. These *fully-instantiated* descriptions are lengthy and difficult to work with for several reasons.

---

[1]This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

[2]Now at Atari Research and Department of Computer Science, Columbia University.

The number of devices on a VLSI chip is increasing at an exponential rate [5]. Fully-instantiated descriptions are therefore increasing in length at an exponential rate. Since the basic element on which circuit extractors operate is the rectangle, and most extractors have performance that is $O(N)$ [2] or $O(N \log N)$, where N is the number of rectangles in the layout, run-times are becoming very long.

People who design complex artifacts have long known that an effective way to speed the design process is to build objects *hierarchically*. In a hierarchical design, objects are formed by composing a number of less complex objects. Each object in turn is built of subpieces until the most primitive blocks are reached (in VLSI, these primitive building blocks are the rectangles). Hierarchical designs are usually structured so that objects that are related are grouped together. This helps the designer by limiting the amount of detail that must be considered at a given time. Low-level design is time-consuming, so designers are motivated to reuse building blocks where possible. This *regularity*, the repeated use of a symbol or a group of symbols, serves to further reduce design time. Because of the recent emergence of hierarchical integrated circuit mask description formats, the structure and regularity in hierarchical designs are available to tools that work on mask descriptions.

Structure and repetition are of use in reducing the run-time for artwork analysis tools such as circuit extractors. A key observation is that much of the area of current VLSI chips is covered by similar structures (for example, repeated memory cells or bit-sliced data paths). Thus determining which parts are identical can eliminate redundant work. Recognizing identical groups of rectangles in a fully-instantiated description is difficult; structure in a hierarchical description makes this task possible. Instead of trying to determine if a given group of 100 rectangles matches some other group, a hierarchical extractor need only decide if the groups are instances of the same symbol. The more regular a design is, the greater the reduction in the repeated analysis. A symbol call that is repeated 1000 times costs little more than the same call repeated 100 times: only one analysis is done in either case. The same situation is much more expensive for an extractor that works on a fully-instantiated description, since at least 10 times the work must be done.

While exploiting structure and repetition can reduce the running-time of CAD tools, a problem remains. In integrated circuits, cells can physically overlap on the chip surface, thus changing their electrical characteristics. This implies that a cell cannot be circuit extracted without considering what other structures interfere with it. A number of papers that address this problem [7, 8, 9] have appeared in literature. HEXT, the circuit extractor described here, solves the problem by first transforming an IC design into one that has no overlapping cells, and then analyzing the transformed design.

2. An Overview of the Hierarchical Circuit Extractor

The input to the HEXT program consists of a description of the layout of the chip in CIF (Caltech Intermediate Form) [6]. CIF allows the user to place geometry (rectangles, polygons, and wires) on the various mask layers of a VLSI design. It further allows the user to define symbols that may consist of geometry and calls to other symbols. There is no array command, so arrays must be constructed explicitly, for example by calling a single cell repeatedly, or by defining a row-of-cells symbol and calling it repeatedly. Because CIF is a hierarchical format, the structure in the chip design is available to layout analysis tools.

HEXT is divided into two parts, a front-end and a back-end. The HEXT program does not assume that the input CIF description of the layout contains no overlapping instances. The program contains code that takes the CIF description and builds an internal description containing no overlapping instances. The rest of the program operates on the transformed representation.

In its simplified form, the main algorithm used by the HEXT program may be viewed as a three-step process.

1. Find all distinct non-overlapping windows. Determine how these windows should be composed to cover the entire chip.

2. For each of the windows compute:

    a. The electrical network represented by the window.

    b. An interface that can be later used to compose adjacent windows.

3. Combine the windows together (according to the composition order specified in first step) to obtain the circuit corresponding to the complete layout.

The front-end is responsible for eliminating overlapping symbol instances. The result is a set of non-overlapping *windows* (rectangular regions containing some number of geometric shapes), and a specification of how to place the windows so that the entire chip is covered. The set of windows is similar to a set of jigsaw puzzle pieces and the composition specification tells how to place the pieces to complete the puzzle. The difference is that identical pieces are highly valued in a hierarchical extractor, since a new piece represents work that must be done.

The back-end is responsible for analyzing the geometry contained in each of the distinct windows found by the front-end. For each window the back-end computes the electrical circuit represented by the geometry inside it and an interface. The interface contains information about geometry that touches the boundary of the window. Since overlap has been removed, communication with the external environment is only possible through the interface. The back-end also combines adjacent windows to obtain a circuit for the entire chip.

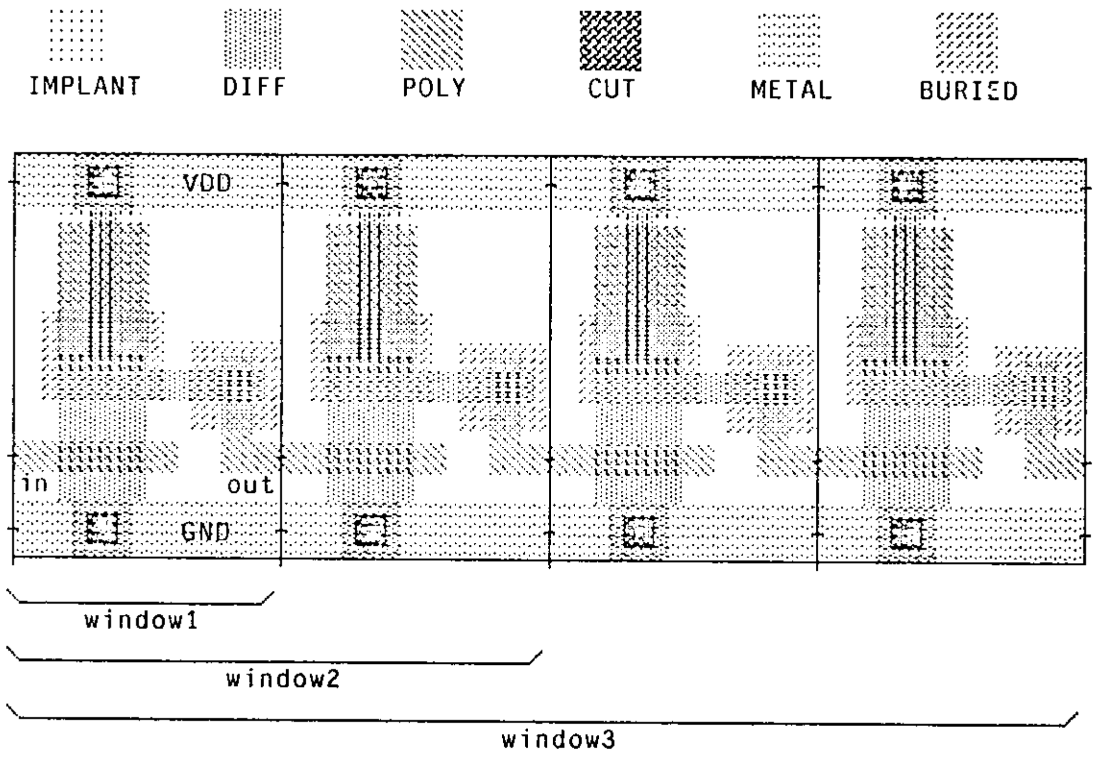

Figure 2-1: Four inverters

The output of the HEXT program is a hierarchical wirelist. This is a hierarchical representation of the electrical circuit denoted by the layout. The format used for the wirelist was developed by Ed Frank, Carl Ebeling, and Bob Sproull at CMU [1]. In this format circuits are represented in terms of *parts* (devices) and *nets* (signals). Parts can either be *primitive* (such as enhancement or depletion transistors) or *non-primitive* (defined in terms of other more simple parts). The format permits nested definitions to describe hierarchical circuits and provides scope rules to resolve naming conflicts. In the wirelist a DefPart statement is used to encapsulate a circuit fragment. This circuit can then be instantiated any number of times using the Part statement. The export list of a circuit is the list of all signal names internal to the circuit that may be referenced from outside, while the local list refers to the set of internal signals that may not be referenced from outside. The Net statement is used by the wirelist to establish equivalence of signal names.

Figure 2-2 shows a hierarchical wirelist for the four inverters shown in Figure 2-1. In the wirelist fragment associated with Window1, N0 refers to the *VDD* signal, N3 refers to the *out* signal, N8 refers to the *in* signal, and N10 refers to the *GND* signal. All four signals are present in the export list as they are located on the boundary of the window and may be referenced from outside. The window consists of two primitive devices, one enhancement and one depletion mode transistor, that constitute the inverter. The wirelist fragments for the remaining windows may be interpreted in a similar manner.

```
(DefPart nDepl (Exports G S D))
(DefPart nEnh  (Exports G S D))

(DefPart Window1
 (Exports  N3  N0  N10  N8  )
 (Part nDepl (Name D1) (Loc 1000 4600) (T G N3) (T S N0) (T D N3))
 (Part nEnh  (Name D2) (Loc 600 1600) (T G N8) (T S N3) (T D N10))
 (Local  ) )

(DefPart Window2
 (Exports  N13  N23  N16  N0  N10  N8  )

 (Part Window1 (Name P1)  (NetOffset 13)  (LocOffset 3600 0))
 (Net P1/N3 N16)   (Net P1/N0 N13)   (Net P1/N10 N23)
 (Net P1/N8 N21)

 (Part Window1 (Name P2)  (NetOffset 0)  (LocOffset 0 0))
 (Net P2/N3 N3)   (Net P2/N0 N0)   (Net P2/N10 N10)
 (Net P2/N8 N8)

 (Net N0 N13)    (Net N10 N23)    (Net N3 N21)
 (Local   N21   N3    ) )

(DefPart Window3
 (Exports  N26  N13  N36  N23  N39  N49  N42  N0  N10  N8  )

 (Part Window2 (Name P1)  (NetOffset 26)  (LocOffset 7200 0))
 (Net P1/N13 N39)   (Net P1/N23 N49)   (Net P1/N16 N42)
 (Net P1/N0 N26)   (Net P1/N10 N36)   (Net P1/N8 N34)

 (Part Window2 (Name P2)  (NetOffset 0)  (LocOffset 0 0))
 (Net P2/N13 N13)   (Net P2/N23 N23)   (Net P2/N16 N16)
 (Net P2/N0 N0)   (Net P2/N10 N10)   (Net P2/N8 N8)

 (Net N13 N26)    (Net N23 N36)    (Net N16 N34)
 (Local   N34   N16   ) )

(Part Window3  (Name Top))
```

Figure 2-2: Hierarchical wirelist for the four inverters

3. The Algorithm

The previous section gave a brief overview of the functions that are performed by the hierarchical extractor. This section describes in detail the algorithms used to perform those functions and discusses some design and implementation issues.

The front-end

The front-end performs three basic operations:

- Recognize redundant windows.
- Divide a window into a set of non-overlapping sub-windows.
- Determine how to connect each sub-window to its neighbors.

The window given initially to the front-end contains the entire design, which may contain symbol instances and primitive geometry. The front-end divides the window into a set of sub-windows (the second step) and then applies the algorithm to each sub-window recursively. If a window is reached that cannot be subdivided, because it contains only geometry, it is sent to the back-end for circuit extraction. Windows that contain only geometry are always rectangular in shape. (The front-end is independent of the back-end task, i.e., the same front-end can be used for plotting, design-rule checking, or other tasks.) The front-end remembers each unique window in a table along with a circuit fragment for the window. Each time a window is considered for sub-division, the front-end checks a table to see if the window was previously analyzed. If so, the associated circuit fragment is returned, otherwise a new entry is made in the table and the window is subdivided. The last step results in a circuit fragment for the window. This fragment is associated with the window's table entry and returned.

A window is divided into sub-windows by the following algorithm:

1. Does this window contain only geometry? If so, it need not be subdivided, and is sent to the back-end for analysis.

2. Expand all symbol instances one level, that is, replace each symbol instance by its constituent parts—other symbol instances and geometry. See Figure 3-1a, 3-1b.

3. Whenever the bounding boxes of two or more symbols overlap, create a new window using the boundaries of the bounding boxes to define the edges[3]. We are left with a number of sub-windows in the original window. See Figure 3-1c.

[3]This *disjoint* transformation was first suggested by Newell and Fitzpatrick [7]. See [3] for a discussion of other transformations.

4. Slice the original window into a set of sub-windows, using the sub-windows found in step 3 for guidance. See Figure 3-1d.

When the subdivision phase is completed, the front-end decides in what order to visit each sub-window. The order determines how a window is covered; many strategies exist. The particular strategy implemented is quite simple: the sub-windows are sorted by the lower-left corner, bottom to top, left to right, and then visited in sorted order.

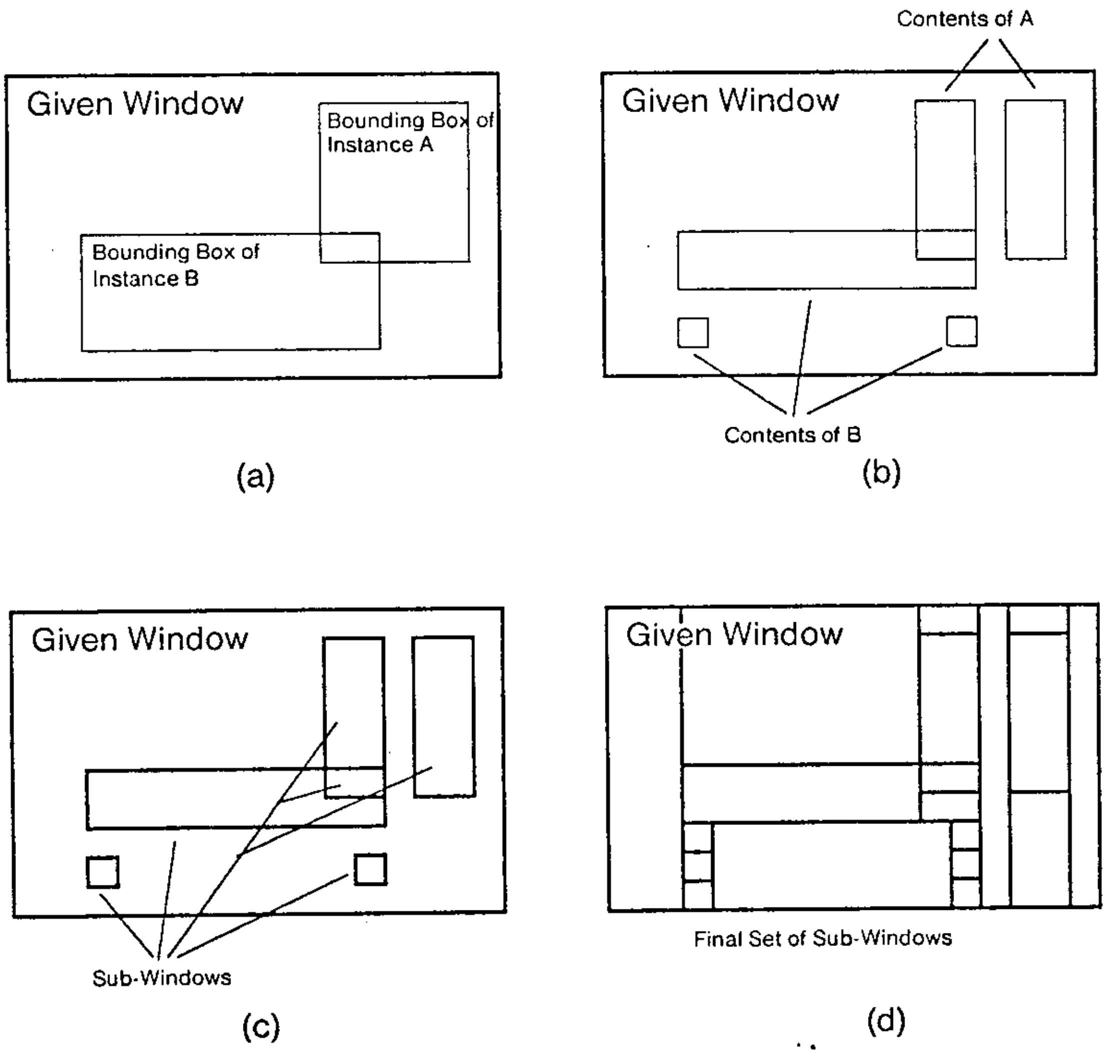

Figure 3-1: Sub-division of Windows

This composition strategy has a number of important implications for the back-end. The *Compose* routine called by the front-end takes two windows as arguments and returns a third that is the result of merging the two along their common boundaries. Notice that the result of combining two rectangular windows may not be rectangular (for example, a large window and a small one may produce a "L"-shaped window). In our discussion, rectangular windows are called *simple* windows while non-rectangular windows are called *complex*.

Returning to the composition strategy used here, two observations can be made:

1. *Compose* will only have to combine two simple windows or one complex and one simple window, never two complex ones.

2. Complex windows never have holes in them.

Both of these facts are exploited by the back-end to make the *Compose* routine simpler.

When the front-end completes its partitioning of the chip, a number of windows have been extracted by the back-end and then composed into ever larger windows until the chip has been covered (i.e., completely extracted).

The back-end

The back-end is responsible for two major functions. First, it must compute the electrical circuit and interface for each of the *primitive* (geometry only) windows. Second, it must combine adjacent windows to obtain a circuit for the chip. Notice that primitive windows are always simple (rectangular) windows, but the converse is not true since a simple window may result from composing a complex and a simple window.

To analyze the geometry in the primitive windows the hierarchical extractor uses a modified version of the *flat extractor* ACE [2]. (A flat extractor disregards hierarchy and regularity, and works on a fully-instantiated description of the layout.) ACE is a fast edge-based circuit extractor written in the C language and runs on a VAX-11/780 at CMU. Modifications to ACE were necessary because:

- ACE is optimized to handle large designs (hundreds of thousands of rectangles), while the primitive windows encountered in the course of hierarchical extraction are very small (a few hundred to a few thousand rectangles). Overhead that is insignificant for the large chips is substantial for the small primitive windows. For example, ACE uses a large static table for storing information about net equivalences. The initialization overhead for this is comparatively small for large designs, but is a large fraction of the total time for small windows. Selective initialization is used in the modified version of ACE to reduce this overhead. The complete data structure is initialized once when the first window is analyzed. For all subsequent windows only the portion of the data structure used during the analysis of the previous window is initialized.

- ACE does not compute an interface for the layouts it analyzes (this is not necessary). The modified version of ACE has extra code to output an interface for each window that it analyzes.

- Storage allocation and reclamation are more frequent, necessitating changes to increase their efficiency.

The hierarchical extractor makes use of the interface computed for the primitive and non-primitive windows when combining them into larger windows. The data structures used for the interface are critical because it is observed that for most chips, a large fraction of the total time is spent in merging interfaces (see Table 5-2). The structure of the interface was partly based on the following observations:

- Windows communicate with the external environment via geometry on the conducting layers (metal, poly and diffusion) that touches the boundary of the window. As the non-conducting layers (implant, cut, buried and overglass) do not carry any electrical signals, geometry on these layers cannot transfer any information to the external environment.

- The boundary of a window may cross the active region of transistors, leading to the formation of partial transistors inside the window. The final form of these transistors is determined by the contents of the windows adjacent to the partial transistor.

- The boundaries of windows are always aligned to the coordinate axes, although it is not necessary that the windows be rectangular.

The interface consists of a list of *boundary segments* for each of the four *faces* (left, right, bottom, and top) of a window. Simple windows have only one boundary segment for each face, while complex windows may have one or more boundary segments for each face. For example, the complex window shown in Figure 3-2 has one boundary segment each for its left and bottom faces, but has two segments each for the top and right faces. Associated with each boundary segment is information about its endpoints, and a sorted list of rectangle edges (one list for each of the conducting layers) touching the boundary segment. These lists are called the *interface-segment* lists. Associated with each element in the interface-segment list is data about the extent of contact between the rectangle edge and the boundary segment, and the identity of the signal carried by the rectangle.

The interface for a window also contains a list of partial transistors, i.e., transistors whose channels touch the boundary. Each partial transistor keeps pointers to the elements in the interface-segment list that belong to it, and conversely, elements in the interface-segment list have a pointer to the partial transistor they are a part of. When abutting windows are merged and the interface data structure is updated, all partial transistors that do not have corresponding elements in the interface-segment list are output as completed transistors.

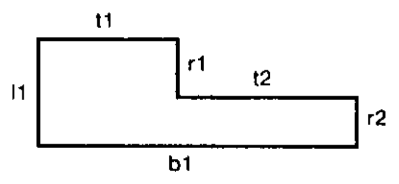

Figure 3-2:

Adjacent windows are composed by the following steps:

1. Find all pairs of boundary segments that touch from the two windows that are to be merged.

2. For each pair of touching boundary segments, step through the elements of the interface-segment lists (for corresponding layers) and establish signal equivalences.

3. Compute the interface for the new window.

In the compose operation the circuit and interface information for the component windows must not be destroyed, because they maybe referenced again in another compose operation. The resulting new window does not copy the contents of its component windows, but simply stores pointers to them. For the interface, however, all relevant contents are copied over. This is an expensive operation and its cost is proportional to the number of rectangles touching the boundary of the new window. Storing all information about the interface with the window greatly simplifies the *Compose* routine.

4. Algorithm Analysis

The performance of a hierarchical extractor is a strong function of the regularity and hierarchy present in the layout description. The performance can vary anywhere from worse than to exponentially better[4] than that of a flat extractor. It is easy to give an example for the case where the hierarchical extractor performs worse than the flat extractor. A layout containing no hierarchy and no repetition takes longer on a hierarchical extractor, as it gains nothing from hierarchy or repetition but wastes time trying to find regularity anyway.

Although numerical measures of hierarchy and regularity have started emerging [3, 7], it is very difficult to parametrize the performance of the hierarchical extractor. One can predict general trends, but anything more than that is difficult. It is possible, however, to predict the performance in a few extreme (boundary) cases,

[4]This is the theoretical speedup that can be attained. The algorithms used in HEXT do not give exponential speedup.

which also lend insight into the algorithm. In the following paragraphs we examine the performance of the extractor on a square array of identical cells. This is the best case for the hierarchical extractor, as maximal hierarchy and regularity are present.

Consider a square array containing N identical cells, where N is an even power of 2 (the array is constructed as a complete binary tree with the leaves forming the N cells). Each side of the array has $N^{1/2}$ cells touching it. To extract this square array HEXT first extracts a single cell using the flat extractor. It then combines the extracted circuit with another copy to obtain the circuit corresponding to two cells. The operation is repeated on the new circuit to obtain the circuit for four cells, and so on. The cost of determining the circuit for the entire array is the cost of extracting a single cell using the flat extractor plus the cost of $\log N$ compose operations. Note that the windows being composed are not of fixed size, but double at each step.

Before going further with the analysis of the array, it is necessary to determine the costs associated with the steps listed above. The cost associated with extracting a single cell is a constant, say k. There are two costs associated with combining windows, when the circuit and interface for the component windows are already known:

1. The cost of computing the interface for the new window. In the HEXT program this is proportional to the number of rectangles touching the boundary of the new window. For the array this is proportional to the number of cells touching the perimeter of the new window. Let this cost be $c_1 n$, where n is the number of cells touching the perimeter.

2. The cost of establishing signal equivalences at the common boundary. This cost is also proportional to the number of rectangles on the common boundary. Let this cost for the array be $c_2 m$, where m is number of cells on the common boundary.

The recurrence equation for the cost of extracting the array is obtained by looking at the process of extraction in a top-down manner, instead of the bottom-up manner actually followed by the extractor. Under the cost model described above the recurrence equations are:

$$T(N) = T(N/4) + 7c_1 N^{1/2} + (3/2)c_2 N^{1/2},$$

$$T(1) = k.$$

The closed form solution of the above recurrence is $T(N) = (14c_1 + 3c_2)\, N^{1/2} + k$, indicating that we should expect $O(N^{1/2})$ behavior from the extractor. Table 4-1 below shows the actual run-time of the hierarchical extractor on square arrays of increasing dimensions. The basic cell here contained a single transistor formed by the overlap of diffusion and polysilicon. The table also lists the performance of the flat extractor on the same array, which is equivalent to the case when hierarchy and regularity are ignored.

N, total # of cells	HEXT (seconds)	HEXT - k (seconds)	flat extractor (seconds)
1	6.0 (k, the cost of extracting one cell)		
1024 (1K)	7.6	1.6	25.5
4096 (4K)	9.2	3.2	103.6
16384 (16K)	12.8	6.8	410.1
65536 (64K)	18.7	12.7	1844.1
262144 (256K)	33.8	27.8	-

Table 4-1: Performance of HEXT in the ideal case

In the above table the third column is of main interest. In this column we have subtracted the cost of initialization and the cost of extracting a single cell (this corresponds to k in the above analysis) from the total cost of extracting the array. The table shows that for every four-fold increase in the number of cells, the extraction time in the third column increases only by a factor of two, which is exactly as predicted by the analysis. The flat extractor exhibits linear behavior in the number of cells, which is asymptotically the best it can do, as it must look at each and every cell.

It is often necessary to flatten the hierarchical wirelist produced by the HEXT program. This is because most CAD tools, especially simulators, require a flat wirelist as their input. The hierarchical wirelist can be flattened by recursively instantiating all calls to subparts of the top level cell. In this case the performance of the hierarchical extractor is linear in the number of devices in the circuit.

5. Performance

The hierarchical extractor, HEXT, is written in the C language [4] and runs on a VAX-11/780 under UNIX. The code for the front-end was written by Bob Hon and the back-end was written by Anoop Gupta.

Table 5-1 below, shows the performance of the HEXT program. While it is easy to characterize the performance of the flat extractor as linear in the number of devices, it is very difficult to characterize the performance of the HEXT program. The performance is best for the testram chip, which is a regular memory array. This result is expected from the analysis in the previous section. The performance for schip2 and psc chips, however, is much worse. The extraction time for these designs can be divided into two parts. First, the time taken to extract regular structures such as memory and register arrays. For HEXT this is only a small fraction of the total time. Second, the time taken to extract irregular structures like data paths and control. The front-end of HEXT divides these structures into a large number of small distinct windows. The time taken to extract these windows using the flat extractor is small, but it takes an extremely long time to compose them together. This is the main cause for the poor performance of HEXT on these designs.

chip	devices	HEXT front-end (min:sec)	HEXT back-end (min:sec)	HEXT Total (min:sec)	ACE flat (min:sec)
cherry	881	0:49	1:12	2:01	1:05
dchip	4884	3:07	3:57	7:04	10:12
schip2	9473	8:42	19:06	27:48	18:12
testram	20480	0:24	1:12	1:36	26:36
psc	25521	18:57	30:14	49:11	41:14
riscb	42084	8:57	18:19	27:16	92:12

Table 5-1: Performance of HEXT

Tradeoffs exist between the amount of work done by the front-end and that done by the back-end. The front-end uses a number of heuristics to partition a chip. The complexity of these heuristics can be varied to produce inferior or superior partitionings. If the front-end spends little time and produces an inferior partitioning of the chip, the back-end has to spend a lot of time in analyzing and composing the circuit for the chip. If instead the front-end spends a large amount of time and produces a good partitioning, the back-end will only take a small amount of time to construct the circuit. Beyond a certain limit, however, the extra time spent in the front-end does not lead to a corresponding or larger decrease in the time spent by the back-end. It is worthwhile (and still an open issue) to determine the point of match between the front-end complexity and the back-end.

The back-end consists of two relatively independent parts, (i) the flat extractor and (ii) the routines which compose windows. It is important to have a knowledge of the distribution of time between the two parts to be able to optimize them. Table 5-2 presents data for the percentage of total back-end time spent in composing windows. The table shows that on an average 72% of total time is spent in composing windows. The time spent in composing is large both for regular and irregular designs. This indicates that it is more important to optimize the algorithms for the compose routine than those for the flat extractor.

chip	devices	Calls to flat extractor	Calls to compose routine	HEXT back-end (min:sec)	time for compose (min:sec)	% of time spent in composing
cherry	881	205	463	1:12	0:34	47%
dchip	4884	375	1886	3:57	2:37	66%
schip2	9473	538	6409	19:06	17:58	94%
testram	20480	45	1089	1:12	1:02	86%
psc	25521	3756	11565	30:14	23:44	79%
riscb	42084	1499	8785	18:19	11:03	60%

Table 5-2: Analysis of back-end

6. Conclusions

The performance of the HEXT program demonstrates the usefulness of hierarchical tools for VLSI. In particular, we obtain more than an order of magnitude speedup for regular designs (e.g., the testram chip). While the performance is worse for some designs, primarily because of the large number of compose operations, preliminary work indicates that the number of compose operations may be reduced by a more intelligent fracturing algorithm. One such algorithm avoids creating many small, unique windows by coalescing adjacent windows in some cases. It is clear that there is still work to be done, in particular, trying to understand how to reduce the overhead in hierarchical analysis of designs with little hierarchy. Robert Hon has explored a number of different algorithms to do so in his PhD thesis [3]. While other hierarchical tools may exhibit better performance, it is often the case that they do so by forcing the designer to use a restrictive design style (e.g., restricted overlapping of cells [8]). We believe that it is possible to use hierarchy to make VLSI design tools faster without constraining the designer.

7. Acknowledgments

We wish to thank Jon Bentley, Allan Fisher, Edward Frank, Robert Sproull, and Hank Walker for valuable comments and the careful reading of early drafts of this paper.

References

[1] Edward Frank, Carl Ebeling, and Robert Sproull.
Hierarchical Wirelist Format.
VLSI Document V085, Carnegie-Mellon University, 1981.

[2] Anoop Gupta and Robert Hon.
Two Papers on Circuit Extraction.
Technical Report, Carnegie-Mellon University, 1982.

[3] Robert Hon.
The Hierarchical Analysis of VLSI Designs.
PhD thesis, Carnegie-Mellon University, (in preparation).

[4] Brian W. Kernighan and Dennis M. Ritchie.
The C Programming Language.
Prentice-Hall, 1978.

[5] Bill Lattin.
VLSI Design Methodology: The Problem of the 80's for Microprocessor Design.
In *Caltech Conference on VLSI.* January, 1979.

[6] Carver Mead and Lynn Conway.
Introduction To VLSI Systems.
Addison-Wesley, 1980.

[7] Martin E.Newell and Daniel T. Fitzpatrick.
Exploiting Structure in Integrated Circuit Design Analysis.
In *Conference on Advanced Research in VLSI.* M.I.T., 1982.

[8] Mike Tucker and Lou Scheffer.
A Constrained Design Methodology for VLSI.
VLSI Design , May/June, 1982.

[9] Telle Whitney.
A Hierarchical Design-Rule Checking Algorithm.
Lambda Magazine , First Quarter, 1981.