# Architectural Synthesis via VHDL

Steven P. Levitan, Yee-Wing Hsieh, Alan R. Martello
Department of Electrical Engineering, University of Pittsburgh
Barry M. Pangrle
Department of Computer Science, The Pennsylvania State University

## Abstract

In this paper, we present results from our experiments integrating an architectural synthesis tool, *SandS*, into the Keystone VLSI design environment. The resulting Architectural VHDL Synthesis tool is called *TinkerTool.* It provides a means by which a designer can conceive and configure complex designs in a high level behavioral language to automatically generate hierarchical VHDL. The tool allows designers to take a "black box" approach to design, so that they do not need to know the implementation details of components or sub-components below the current one. Consequently, both top down and bottom up design methodologies can be used. Similarly, any circuit design modeled in VHDL can be ported into the component library to be used in the next level of hierarchical design.

A major impact of this hierarchical component approach is that representations, other than VHDL, can be attached to a component and one could construct the entire design along a particular implementation path. For example, mask layout of the entire design could be constructed by using layouts attached to each particular component. Such hierarchical layout generation would be much faster than current flat transistor level layout generation. This technique coupled with "global re-optimization" of the components would remove one of the more serious drawbacks to library based design styles.

## 1 Introduction

We use the term *architectural synthesis* to differentiate a class of design methodologies for large digital (VLSI) systems. These design styles can be characterized by the following attributes:

- The systems themselves are built from a "few" "large" objects rather than many small ones.
- The design is done with components from a design library.
- The design tradeoffs are in terms of *structure.* That is, we design by choosing components and interconnection strategies, rather than by choosing between algorithms for implementing functions.

The rational for this design style is based on the fact that the size of the designs are such that optimization tools take prohibitively long to run on the entire design. Rather, a library of optimized (and possibly parameterized) components is used. We assume that the designer is primarily concerned with the design of the algorithm that will be implemented and the resources available (speed and area) rather than low level implementation issues. Therefore, the tradeoffs available to the designer are in terms of variations in the algorithm, component, interconnect style, and control style selection,

To build systems using this design method, the user (and the design tool) must have access to the characteristics of available components to make informed choices. If the library is a set of templates of parameterized modules (rather than completed modules) the design tool must have ways to estimate the final size and speed of the system in order to guide the designer in decision making.

The primary advantage to this design style is that it tends to produce working systems with a minimum time and effort from both the design synthesis tool and the designer. Systems are compositions of tested components interconnected using fixed techniques, usually with simple timing models. If the components are well characterized, and the design tool can perform placement and routing effectively, it is possible to generate working systems which meet specifications.

On the other hand, the primary drawback of this design style is that it tends to preclude a level of global system optimization which is possible when an entire design is synthesized at once. Generally, design tools can not perform this level of optimization on large systems. However, there are times when global optimization among components is the only way to get the desired performance from a system.

Our goal in this work is to have a tool where we can get the benefits of a component based design style without sacrificing performance. We believe that this can come from removing the arbitrary boundaries between components which exist when they are just "cut and pasted" from a component library. We build a single hierarchical VHDL entity using components from a VHDL component library. At different stages in the synthesis process, redundant operations and operators at the boundaries between the components are removed. The resulting monolithic design is then given to behavioral synthesis tools. However, to fully utilize the advantages of a library we need to keep in the library (along with the VHDL descriptions of the components and their performance characteristics) an abstract representation of their optimized layout. During the final synthesis phase of the design, the tools will use this information to speed up the synthesis of the full design.

In the rest of this paper, we present a discussion of the tools which make up the environment for our architectural synthesis manager, TinkerTool. We first introduce SandS, our architecture synthesizer. We follow with a brief description of the rest of the tools in the Keystone VLSI design environment. We then describe the main focus of this paper, TinkerTool, the configuration management and VHDL generation tool which provides the interface between the user, the library, SandS and the rest of the Keystone system. We follow this with an example of architectural synthesis using these tools. Finally, we discuss our strategy for a system which overcomes the performance limitations of a library based design methodology.

## 2 Background

### 2.1 SandS

SandS (*slicer* and *splicer*) is a tool which performs architectural synthesis, the compilation of a high-level behavioral specification into a register transfer level architecture. The input to SandS is a behavioral specification written in a high-level language which describes an algorithm or set of algorithms that the user wants implemented on a chip [Pan87, Bre88, PG86, Pan87]. This input is generally more abstract than a VHDL structural/dataflow description that would be used as the input to a logic synthesizer. An example input file for a simple division algorithm we use throughout this paper is shown in Figure 1.

```
program divide(input,output);
        type integer = {0..7};
        port in_y, in_x, out_q : integer;
        var y, x, q: integer;
begin
    y := in_y; x := in_x; q := 0;

    if (y >= x) then
        repeat
            y := y - x;
            q := q + 1;
        when y >= x;
    out_q := q;
end.
```

Figure 1: Division Algorithm

There are three main tasks to be performed in transforming a high-level behavioral specification into an architecture. The first is the compilation of the input language into a flow-graph [OG86] representation. The second is the allocation of operations to states or control steps of the machine (also called state binding). The third task is the allocation of hardware components to operations, register assignments, and the creation of multiplexors and/or busses to complete the data path. This last task is also referred to as connectivity binding [GDP89].

SandS builds an architecture based on constraints imposed by the user. The idea is to let the user perform architectural design tradeoffs, while the system synthesizes designs that meet the user's constraints. This separates two basic types of knowledge needed to produce the designs. The first type is the "what to build" knowledge (the user's job) and the second is the "how to build" it (the synthesizer's job). This separation of knowledge in SandS frees the user from implementation details and presents the opportunity for creative exploration in the design space. The user can ask "what if" questions and have systems generated quickly to explore design alternatives.

The constraints that the user specifies are control "knobs" for the system. By altering the adjustments of the knobs, the user can direct SandS's region of operation within the design space. We have selected the following three major user constraints for the architecture synthesizer.

1. *Component selection:* The component selection mainly consists of the specification of the number of architectural components such as ALU's, multipliers, adders, subtractors, etc. that are allowed in the design along with the component delay times.
2. *Control style selection:* The control style selection specifies the type of micro-engine controlling the design. It may be of a PLA or random logic type and may have pipeline registers inserted within it.
3. *Interconnect style selection:* The interconnect style selection chooses between a tri-state bus based system or a system interconnected by multiplexors.
4. *Control cycle time:* The clock speed or control step period determines the amount of time allotted for each control step. The longer the period, the more time there is during each control step to perform each operation.

Information from the component library is necessary to produce reasonable estimates for the component and control delay times. These estimates could be guesses or results of VHDL simulations of the components. They could, in fact, be generated by running the library components individually through the module and layout synthesis tools, extracting the netlist from the layout, and simulating the result.

Early in the design process, the user must be able to obtain estimates quickly, in order to eliminate portions of the design space that aren't worth exploring. Using estimates rather than real values from low level simulation will help reduce the design space quickly. As we discuss below, it is possible, even desirable, to use "hypothetical" components at this phase of the design. As the design starts to approach a viable solution, the estimates must become more accurate and the designer (with the system) can spend more time deriving accurate estimates.

## 2.2 Keystone

Keystone is a multi-level design and synthesis system developed jointly by researchers at the University of Pittsburgh, and Pennsylvania State University. It is the result of several ongoing research efforts by researchers at both schools [LMOI89].

Keystone consists of a suite of specification, synthesis, simulation and analysis tools for designing VLSI systems. These tools are shown in Figure 2. The tools themselves are indicated by ellipses while each data abstraction level which we support is indicated by a box. The title of each box represents the format of the data within the abstraction level. The shaded boxes and ellipses indicate the path through the system which we will focus on later in this paper.

The tools are shown connecting the different design representations with which they interact. Most of the tools perform optimization within a design abstraction layer as well as transformation

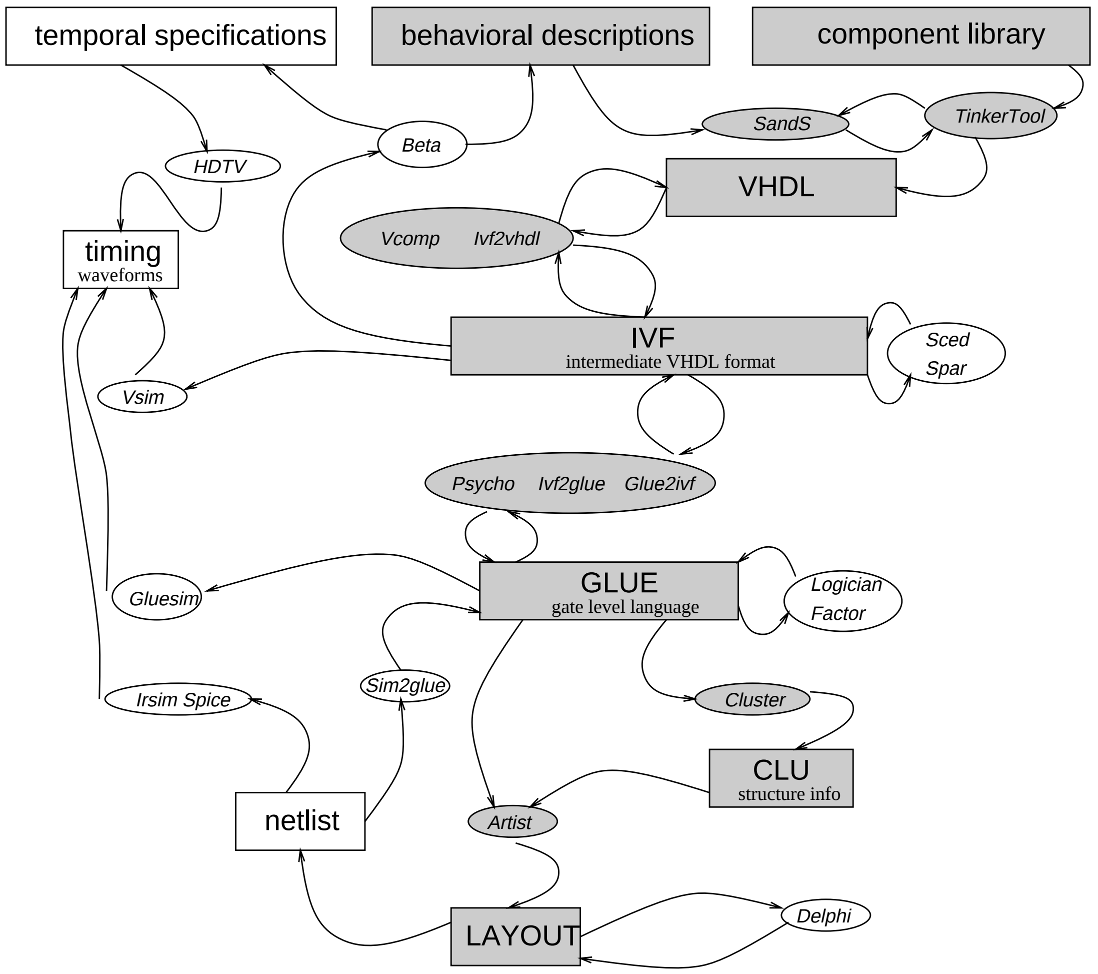

Figure 2: The Keystone System

between representations. Some of the optimization tools perform restructuring or decomposition as well.

User supplied input to the design system can come at any of several abstraction levels. It can come from an *architectural* or *temporal* specification, it can be expressed in either behavioral or data-flow VHDL text [VHD88] or it can be given as a schematic graphical description of the architecture to be implemented. System output can be generated at many abstraction levels. Perhaps the lowest levels of output of the design system are a two dimensional gate matrix layout in a form compatible with MAGIC [SMHO85] and timing waveforms from both the high-level tools and the simulators.

As indicated by the bi-directional arrows, in addition to going from "higher" abstraction levels to lower ones, we support reverse transformations, e.g., from a layout to a VHDL or a schematic description of that layout [Fre91]. We support these various transformations in order to provide the user with a variety of input and output format options including netlists, mask designs, or logic equations from other systems. By being able to move easily between representations in the design space, the system is able to more easily handle the design level interactions, user feedback operations, and support the user in the design process.

While, primarily, the tools developed at Penn State and Pittsburgh are shown in Figure 2, where appropriate, other CAD tools are used as well, including SPICE3 [Chr86], COSMOS, [BBBe87], tools from the Berkeley *magic* tool suite [SMHO85], IRSIM [Ter83], and tools from the Berkeley OCT/Vem tool suite [MHSN87].

## 3 TinkerTool

TinkerTool, the configuration and generation system, is shown in Figure 3. It consists of two parts, a library/configuration manager and a VHDL generator. These two parts, together with SandS make up the component based architectural synthesis system which feeds behavioral VHDL to the rest of the Keystone tools.

To use TinkerTool, the designer interacts with the Configurator to make the architectural decisions which control the SandS synthesis tool. The decisions which the user can make are in terms of:

- Component specification: choosing to add predefined (basic) components, adding "user defined" components, and creating "experimental" components.
- Control style selection: the system controller.
- Interconnection style selection: buses versus multiplexors.
- Overall timing requirements: cycle time for the system.
- Algorithm specification: the "Behavioral Description".

The basic components currently available in the library are shown in Figure 4. Each of these library components has a parameterized bit width. The carry look-ahead versions of the adders and subtractors also have variable block sizes for the carry chains. The actual library components are VHDL templates, written in Perl [WS90], which are expanded as necessary. This approach is more flexible than using VHDL generic entities and generate statements.

The "experimental unit" choice allows the user to assume that a component with certain functional properties and delays exists in the library He can then run SandS to see what kind of architecture would be generated using such a component. This will tell the designer the number of busses, multiplexors, and other components used as well as the micro-code and overall timing of the system with the hypothetical component. The VHDL code output for this component will be a null body. The user can then write a VHDL process block to perform the operations implied for further simulation or synthesis. The designer can also choose to add such a component into the library.

The "user defined" unit choice reflects two ways that users can add new components to the library. The first is that arbitrary components coded in VHDL can be added. These do not give the same flexibility as the basic component templates since they are actual VHDL components and not parameterized templates. However, it does allow the user to import VHDL components from other systems.

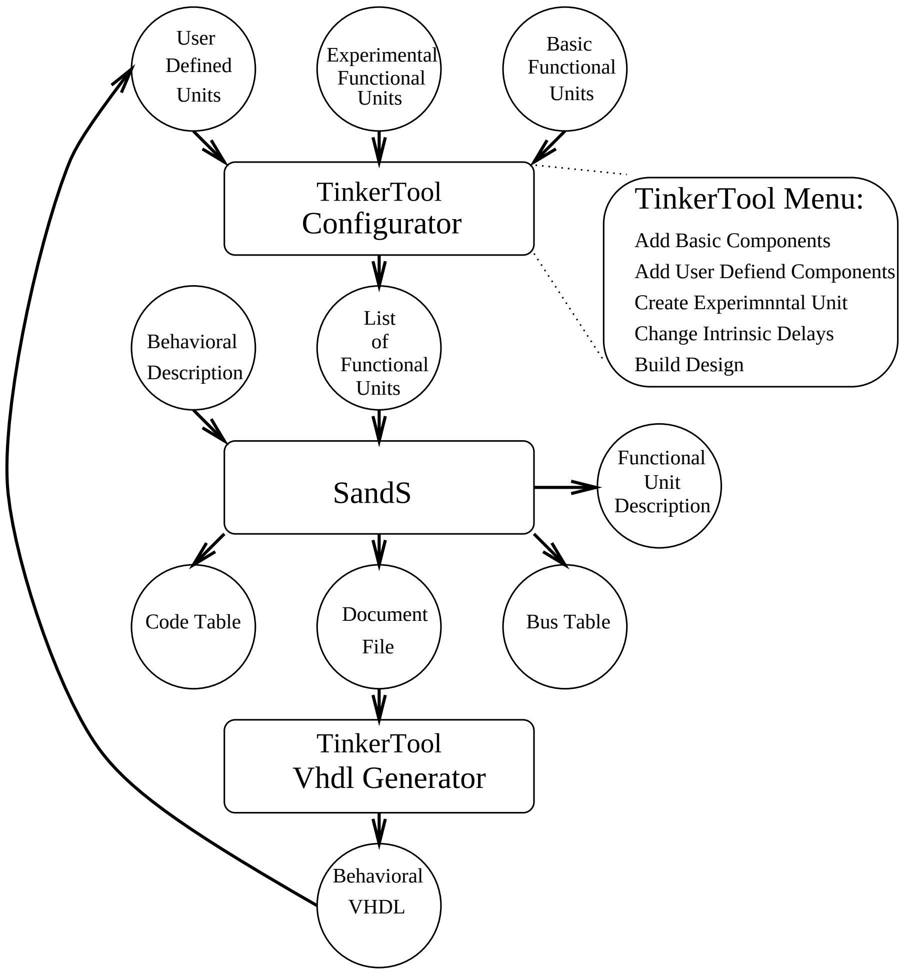

Figure 3: TinkerTool

The second way to generate more components for the library, is to use the output of TinkerTool itself to generate VHDL components which can, in turn be used in larger architectures. This path is shown by the upward pointing arrow in Figure 3. By using this path, designers can quickly build up very large designs from a few components.

Once all the choices have been made by the user and the algorithm specified, the Configurator performs the following functions:

1. Expand the templates of any basic components requested.
2. Build a description of the chosen components suitable for SandS to use for its analysis.
3. Invoke SandS to perform the analysis and synthesis necessary to generate the physical connectivity and temporal binding.
4. Invoke the the second phase of TinkerTool, th VHDL generator.

The component descriptions passed to SandS are only in terms of the key parameters (size, function, delay, etc.) of the chosen subset of the components from the library. SandS might not

```
abs : absolute value
adc : adder with c carry look ahead and carry in
add : adder with c carry look ahead
add_sub : adder and subtracter
asc : adder, subtracter, comparator
dec : decrementer
eq : logic  = operator
ge : logic >= operator
ge2 : logic >= operator (2's complement)
gt : logic  > operator
gt2 : logic  > operator (2's complement)
inc : incrementer
le : logic <= operator
le2 : logic <= operator (2's complement)
lt : logic  < operator
lt2 : logic  < operator (2's complement)
mult : multiplier (2's complement)
mult2 : multiplier
ne : logic != operator
neg : negater
sbc : subtracter with c carry look ahead and carry in
sub : subtracter with c carry look ahead
```

Figure 4: Basic Component Menu

use all components specified, depending on timing constraints. For instance, it might chose not to use two adders, if one can be shared. But it will not use any (or any more) than those listed. Similarly, SandS does not generate a VHDL description of the complete architecture. Its result is an encoded description of the connectivity and timing information needed to complete the architectural synthesis.

An example of the "micro code" generated by the division example is shown in Figure 5. This shows a four step micro instruction sequence for the division algorithm. In the first step, `MI 1a and MI 1b`, the input variables are read, and the initial test of $y \geq x$ is performed. In the second step, `MI 2`, both the subtraction and the addition are performed. The test for completion is done in the third step, `MI 3`, and conditionally the program either exits, in `MI 4`, or loops. The code table also shows the connectivity binding for the synthesized architecture, giving the registers, buses, and function units used in each micro operation.

The VHDL generation part of TinkerTool takes the results of SandS and, together with the VHDL descriptions from the library, generates a complete VHDL description of the architecture. This involves:

- collecting the selected library components
- adding "intrinsic" components like registers, and multiplexors of the correct bit width
- checking components for sub-components and expanding if necessary
- generating all the VHDL "syntactic glue" for the component use, signal declaration, and port maps

```
---------------------------------------------------------------------------
| MI#|           ACTIONS                        | Nxt| Conditions            |
|----|------------------------------------------|----|-----------------------|
|1 a |r002,B02 = FU04( in_y.r:)                 |2   |(y>=x)       :     TRUE|
|    |r003,B03 = FU05( in_x.r:)                 |4   |(y>=x)       :    FALSE|
|    |r004 = ( RMOVE:r000,b01)                  |    |                       |
|1 b |FU03( >=:B02;B03)                         |    |                       |
|----|------------------------------------------|----|-----------------------|
|2 a |r002,B02 = FU02( -:r002,b06;r003,b07)     |3   |                       |
|    |r004,B04 = FU01( +:r004,b08;r001,b05)     |    |                       |
|----|------------------------------------------|----|-----------------------|
|3 a |FU03( >=:r002,b06;r003,b07)               |2   |y>=x         :     TRUE|
|----|------------------------------------------|----|-----------------------|
|4 a |FU06( out_q.w:r004,b08)                   |1   |                       |
---------------------------------------------------------------------------
```

Figure 5: Generated "Micro Code"

- inserting feedback latches into the finite state machine used for the controller

The resulting VHDL file can then be further processed by the rest of the Keystone environment. An abbreviated version of the VHDL output for the division example is shown in Figure 6.

Returning to Figure 2 we can see that in the Keystone environment, the VHDL compiler, and simulator can be used to verify the complete design. For simulation, timing information on the system is based on the timing data from the libraries and their interaction with the finite state machine control specified by SandS.

After compilation and simulation, the design is passed to *Psycho* [Hou90], our behavioral synthesis tool. *Psycho* performs the mapping of the finite state machine VHDL process block into structure. But it also does much more. It completely evaluates the entire design, and eliminates many redundant signals, multiplexors, and control lines. It also re-encodes the finite state machine into a more optimal design. In this process the individual VHDL components are merged into a monolithic entity with many of the borders between the blocks no longer visible.

The layout synthesis tools, ClusterII [OI88] and ArtistII [OI89] are then used to re-partition the design, and produce *magic* layouts. The magic layout for the division example is shown in Figure 7. After extraction, simulation tools can be used to verify the functionality and, more importantly the timing of the system.

Figure 8 shows the results of an *irsim* simulation of the division example. In that figure we show the circuit performing the division of $16_{hex} \div 7$. The data registers `reg_2, reg_3, and reg_4` are used to hold the dividend, divisor, and quotient. One can trace the `ps/ns` present state, next state vectors as the finite state machine loops during the division process. At the end of the algorithm, the signal `fu_3_output_a` goes low, indicating that the $y \geq x$ test is false. The algorithm generates the output, `out_q,` and terminates leaving the remainder in `reg_2`.

```
ENTITY div IS
    PORT(clock: IN BIT; in_y, in_x: IN BIT_VECTOR(7 DOWNTO 0);
         out_q: OUT BIT_VECTOR(7 DOWNTO 0));
END div;
ARCHITECTURE div_struct OF div IS
    SIGNAL ......
    COMPONENT reg
        PORT(clk, write_en: IN BIT; d: IN BIT_VECTOR(7 DOWNTO 0);
         q: OUT BIT_VECTOR(7 DOWNTO 0));
    END COMPONENT;
    COMPONENT add
        PORT(oe: IN BIT; a , b: IN BIT_VECTOR(7 DOWNTO 0);
         sum: OUT BIT_VECTOR(7 DOWNTO 0));
    END COMPONENT;
    COMPONENT sub
        PORT(oe: IN BIT; a , b: IN BIT_VECTOR(7 DOWNTO 0);
         sum: OUT BIT_VECTOR(7 DOWNTO 0));
    END COMPONENT;
    COMPONENT ge
        PORT(oe: IN BIT; a, b: IN BIT_VECTOR(7 DOWNTO 0); y: OUT BIT);
    END COMPONENT;
    COMPONENT .....
BEGIN
    REG_2: reg PORT MAP(clock, reg_en(2), BUS_2, REG_2_output_a);
    REG_3: reg PORT MAP(clock, reg_en(3), BUS_3, REG_3_output_a);
    REG_4: reg PORT MAP(clock, reg_en(4), MUX_3_output_a, REG_4_output_a);
    FU_1: add PORT MAP(op_sel_en(0), BUS_8, BUS_5, FU_1_output_a);
    FU_2: sub PORT MAP(op_sel_en(1), BUS_6, BUS_7, FU_2_output_a);
    FU_3: ge PORT MAP(op_sel_en(2), MUX_1, MUX_2, FU_3);
    BUS_6 <= REG_2_output_a;
.....
    PROCESS(ps,FU_3_output_a)
    BEGIN
        CASE ps IS
            WHEN "001" =>  reg_en    <= "11100";
                           mux_f_en  <= "0101";
                           mux_b_en  <= "10";
                           mux_r_en  <= "01";
                           op_sel_en <= "011100";
                           IF (FU_3_output_a = '1') THEN
                               ns <= "010";
                           ELSE
                               ns <= "100";
                           END IF;
            WHEN "111" =>  ....
        END CASE;
    END PROCESS;
    FSM: reg_b3 PORT MAP(clock, ns, ps);
END div_struct;
```

Figure 6: Generated VHDL Implementation

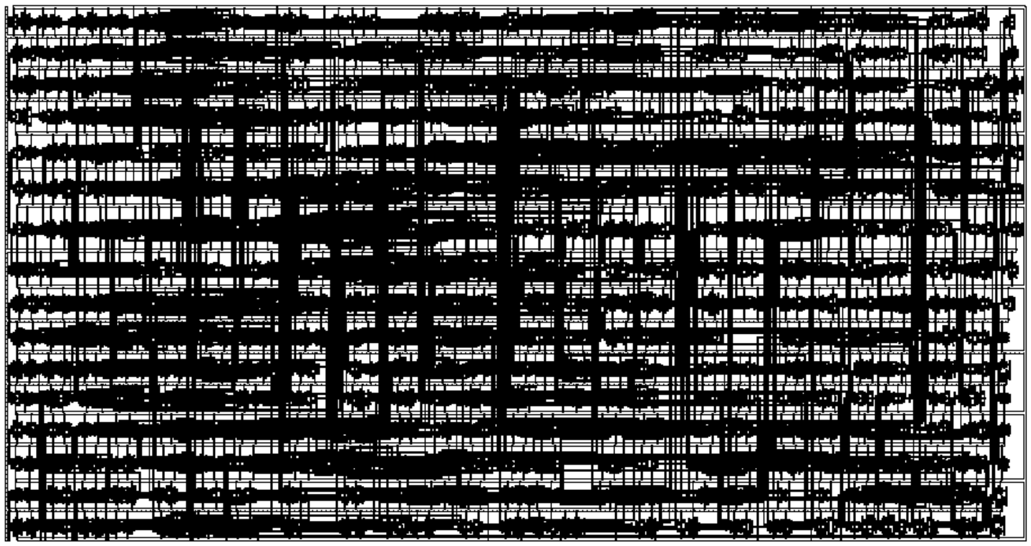

Figure 7: Layout of Division Circuit

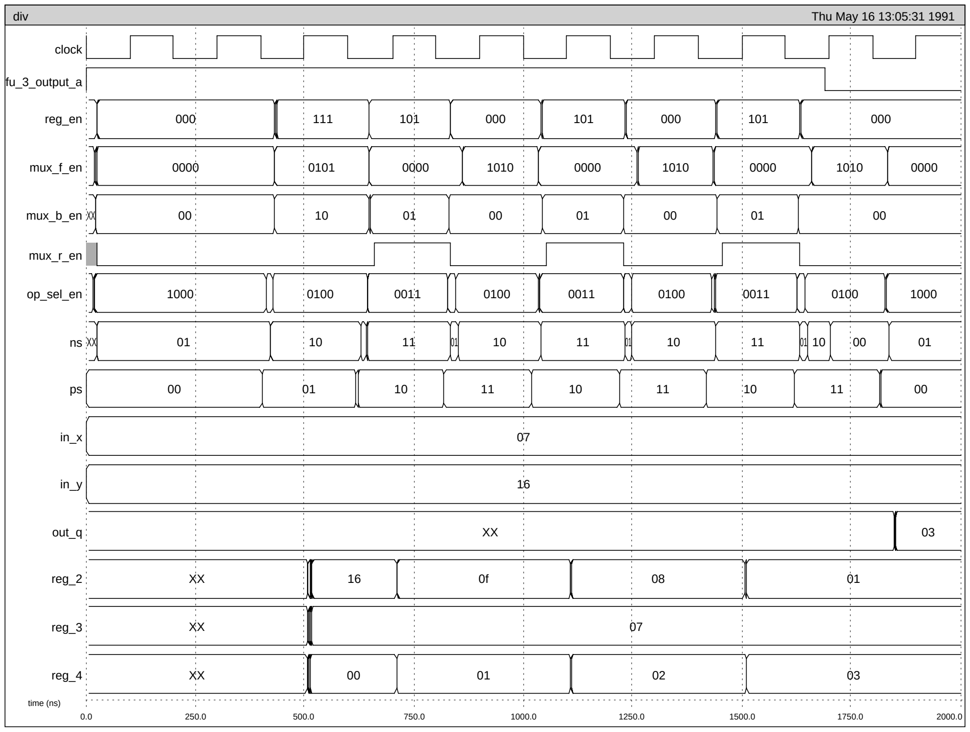

Figure 8: Simulation Results

| Quantity | Component | Gates | Transitors | Length | | Width | $Q \times area$ |
|---|---|---|---|---|---|---|---|
| 1 | add_b8 | 164 | 572 | 347 | $\times$ | 1220 | 423340 |
| | clh_b4 | 70 | 264 | 339 | $\times$ | 569 | 0 |
| 1 | ge_b8 | 84 | 326 | 355 | $\times$ | 681 | 241755 |
| 1 | sub_b8 | 183 | 632 | 363 | $\times$ | 1360 | 493680 |
| | mux_b1 | 3 | 12 | 186 | $\times$ | 100 | 0 |
| 4 | mux_b8 | 40 | 128 | 299 | $\times$ | 324 | 387504 |
| | dff_b1 | 18 | 52 | 251 | $\times$ | 177 | 0 |
| 1 | reg_b3 | 60 | 168 | 299 | $\times$ | 429 | 128271 |
| 3 | reg_we_b8 | 185 | 546 | 323 | $\times$ | 1423 | 1378887 |
| | tb_b1 | 7 | 22 | 186 | $\times$ | 135 | 0 |
| 3 | tb_b8 | 42 | 148 | 299 | $\times$ | 338 | 303186 |
| 1 | FSM | 197 | | 307 | $\times$ | 799 | 239153 |
| | Area Sum | | | | | | 3595776 |
| 1 | divide | 1555 | 4944 | 1899 | $\times$ | 3628 | 6889572 |
| | "Overhead" | | | | | | 3293796 |

Table 1: Component and Design Sizes

## 4 Results and Conclusions

Table 1 shows the sizes for the components used in the generation of an 8 bit division circuit with a carry lookahead adder and subtractor (block length of 4). The quantity field shows how many of each component were used. Some of the sub-components are shown for illustration purposes, (e.g., the clh carry lookahead circuit) although their sizes are incorporated in the major components. All of these sizes come from running the layout synthesis tools with their default settings on a Sun SparcII. The ClusterII tool takes approximately four hours of CPU time on the complete divider. TinkerTool, SandS, Vcomp and Psycho each took negligible time (under a minute) compared to this. The layout tool ArtistII was run in "default mode" with minimal optimization and took approximately 10 minutes, most of which was taken reading and writing the files.

The running times for the tools, and the resulting circuit sizes shows an important point. Even when using a monolithic circuit approach, there is still a significant amount of area "overhead" which comes from interconnecting large blocks. This comes about from two very different sources. One is the actual need for the wires which interconnect the blocks. The divider uses seven eight bit buses. Assuming these run most of the distance across the circuit they account for approximately $2,000,000$ $sq$ $\mu m$. The rest of the overhead comes from the fact that the layout tool needs considerable time to operate on a large circuit. Therefore, it did not do the level of optimization of which it is capable. These two problems compound each other, since the lack of optimization makes the interconnection wires longer, and the circuit that much bigger.

Therefore, we conclude that it is essential that module generation tools be able to generate "intermediate geometry" files which can be kept in the library along with the VHDL, and also that module generation tools be able to accept such a representation as input, to allow them to speed up the final layout phase without sacrificing circuit performance, by only performing a global re-optimization on the assembled components.

To summarize, the work reported here is based on a first pass at implementing a component based, architectural synthesis system. One thing we have learned is the need to support and use intermediate geometric representations in our module generation tools. Besides this enhancement, we plan to extend this work in the following ways:

- Move to a purely behavioral VHDL input language, for consistency, and to take advantage of course-grained high level parallelism in the input language.
- Use optimization tools to perform better FSM synthesis.
- Provide more control to the user to allow for more choices in interconnect styles.
- Provide more sophisticated timing characteristics in the library and in SandS to allow for data dependent delays in components.
- Use related work [Hou90] to extend the simple data-path/control paradigm.

We feel that supporting a component based architectural design style has many advantages, and that the assumed disadvantages in performance penalties can be overcome using the global re-optimization techniques outlined here.

## 5 References

[BBBe87] R. Bryant, D. Beatty, D. Brace, and et. al. COSMOS: a compiled simulator for MOS circuits. In *Design Automation Conference*, June 1987.

[Bre88] F.D. Brewer. *Constraint Driven Behavioral Synthesis*. PhD thesis, Dept. of Computer Science, University of Illinois, May 1988.

[Chr86] Wayne A. Christopher. *A Short Introduction to SPICE3*. CAD Group, U. C. Berkeley, March 1986.

[Fre91] Stephen T. Frezza. Spar: A system for the automatic generation of schematic diagrams. Master's thesis, Dept. of Electrical Eng., Univ. Of Pittsburgh, Pittsburgh, PA, April 1991.

[GDP89] D. Gajski, N. D. Dutt, and B.M. Pangrle. Silicon compilation (tutorial). In *Custom Integrated Circuits Conference*, May 1989.

[Hou90] Pao-Po Hou. *Mid-Level Hardware Synthesis*. PhD thesis, The Pensylvania State University, 1990.

[LMOI89] S.P. Levitan, A.R. Martello, R.M. Owens, and M.J. Irwin. *Proc. 9th Inter. Symposium on Comp. Hard. Description Lang.*, chapter Using VHDL As A Language For Synthesis Of CMOS VLSI Circuits, pages 331–346. Elsevier Science Publishers, B.V., June 1989.

[MHSN87] Peter Moore, David S. Harrison, Rick L. Spickelmier, and A. Richard Newton. Oct, vem, and rpc. Technical report, EECS, Univ. of California, Berkeley, March 1987.

[OG86] A. Orailoglu and D.D. Gajski. Flow graph representation. In *Design Automation Conference*, 1986.

[OI88] R.M. Owens and M.J. Irwin. Exploiting gate clustering in VLSI layout. Technical Report CS-88-09, Computer Science, Penn State Univ., Univ. Park, PA, 1988.

[OI89] R. M. Owens and M. J. Irwin. A comparison of four two-dimensional gate matrix layout tools. In *Design Automation Conference*, pages 698–701, 1989.

[Pan87] B. Pangrle. *A Behavioral Compiler for Intelligent Silicon Compilation.* PhD thesis, Dept. of Computer Science, University of Illinois, June 1987.

[PG86] B. Pangrle and D. Gajski. Slicer: A state synthesizer for intelligent silicon compilation. In *ICCAD*, Santa Clara, CA, October 1986.

[SMHO85] Walter S. Scott, Robert N. Mayo, Gordon Hamachi, and John K. Ousterhout. 1986 VLSI tools: Still more works by the original artists. Technical report, Computer Sciences Division (EECS) University of California, Berkeley, CA 94720, December, 1985.

[Ter83] C.J. Terman. Simulation tools for digital lsi design. Technical report, Cambridge, MA, 1983.

[VHD88] IEEE, New York. *VHDL Language Reference Manual, IEEE Standard 1076-1987*, March 1988.

[WS90] Larry Wall and Randal L. Schwartz. *Programming Perl.* O'Reilly and Associates, Sebastopol, CA, 1990.