MICROARCHITECTURE OPTIMIZATION FOR TIMING AND LAYOUT

by

Nels Vander Zanden
Daniel Gajski
Kenichi Kanehara

Technical Report 91-40

Information and Computer Science Department
University of California, Irvine
Irvine, CA. 92717

Abstract

In recent years the drive to produce more complex integrated circuits while spending less design time has driven the demand for design automation tools. The search for design automation methods has resulted in the design of numerous behavioral synthesis and logic synthesis tools. This report describes a system that fills the gap between traditional behavioral synthesis and logic synthesis tools. Techniques are introduced for improving the microarchitecture structure and using feedback from lower-level optimization tools to guide design optimizations while attempting to meet user specified area and time constraints. These techniques include the capability for mixing layout styles such as custom layout for random-logic components and bit-slicing for regularly structured components. In this manner the entire design, control logic and datapath, can be optimized at the same time. Further, this paper presents a new methodology for microarchitecture-level optimization that greatly reduces the amount of technology-specific knowledge necessary to perform the optimizations.

TABLE OF CONTENTS

CHAPTER

LIST OF FIGURES

1. Introduction

Generation of digital hardware generally passes through four stages of development: behavior, microarchitecture, logic, and layout. Behavior describes the functionality of the hardware and has often been written using simulation languages such as VHDL or programming languages such as C. Behavioral synthesis tools convert these descriptions into a microarchitecture structure called a Register-Transfer-Level design. This structure consists of components such as ALUs, memories, registers, counters, and multiplexors. Each of these components can in turn be expanded into a logic-level design consisting of gates and flip-flops. Finally a layout can be generated from transistors that compose each gate.

Today's designers are increasingly able to enter their designs at higher levels of abstraction. Recently a number of tools that can translate a behavioral description to structure have been developed. Some of these tools are tuned to a particular style of architecture and hence little further optimization is required on the microarchitecture level. Other tools produce varying styles of architecture usually involving control and datapath sections. As these architectures are more general, they tend to be less polished and more optimization of their microarchitecture structure is required.

1.1. Previous Work

Various approaches have been taken to convert the microarchitecture design into a design that can be passed to a layout tool. Some tools describe the behavior of microarchitecture components as a set of boolean equations and flip-flops, then rely heavily on logic synthesis tools to reduce the logic and make an efficient design [Br86] [StMu86] [TsWe88] [WeRo88]. They employ logic generators that produce a design of generic logic gates for each microarchitecture component's description, then use tools such as [BrRu87] to reduce the number of gates in a component, restructure critical paths, and map the design into a particular standard cell or gate-array library. The design can then be passed to a standard cell or gate-array layout tool. Thus optimization in this respect is focused on the inside of each component.

SILC [GuPa90] includes a component rearchitecting step that selects a different style of architecture for components along a critical path. For example, a ripple carry adder can be converted to a carry-lookahead adder or something in between to improve the speed. Thus the style of component can be changed after logic optimization fails to meet the necessary constraints.

Other behavioral synthesis tools designed for datapath generation can base their architecture on a standard cell layout or a bit-sliced layout [TrDi89]. Optimization is carried out for that particular layout style. Still another approach is to construct the design using off-the-shelf components [BiBr88] including microprocessors, DMA controllers, dynamic RAMS, etc.

Numerous tools for behavioral and logic synthesis have been previously reported. This chapter describes a system that fills the gap between the behavioral synthesis tools and logic synthesis tools by using a microarchitecture optimizer. Behavioral synthesis tools often use estimators in design refinement. These estimators may be technology independent and are usually not accurate enough to make decisions for fine tuning the microarchitecture design. On the other hand, logic synthesis tools can accurately gauge area and time but operate on too low of a level to adequately make microarchitecture modifications.

In this paper, techniques are introduced for improving the microarchitecture structure and for employing constraint driven synthesis based on the user's requirements for time and area. These techniques include the capability for mixing layout styles such as custom layout for random-logic components and bit-slicing for regularly structured components. In this manner the entire design, control logic and datapath, can be optimized at the same time. Further, this paper presents a new methodology for microarchitecture-level optimization that greatly reduces the amount of technology-specific knowledge necessary to perform the optimizations. Microarchitecture components are generated by a database based on a set of parameters from the microarchitecture optimization tool. Thus the microarchitecture optimizer does not need to deal with multiple logic optimization tools, layout module generators, transistor sizing tools, etc.

Often the structures produced by behavioral synthesis tools contain inefficiencies such as constants that can be propagated through a design, and common subexpressions that appear multiple times in the design, each time with replicated hardware. These can partly result from the fashion in which the user wrote the behavioral description. Also the design needs to be directed towards a certain set of constraints for time and area. Tradeoffs must be made along different paths. On critical paths optimizations that reduce time are required, possibly at the expense of increased area. Non-critical path optimizations attempt to reduce area as long as doing so does not create a new critical path. In performing these tradeoffs, the microarchitecture optimizer can select a different architectural style for the component, merge components and reoptimize their logic, insert buffers to improve drive capability, replace a set of components with a single component that performs the same function but more closely meets the constraints, restructure components to reduce delay (such as factoring multiplexors), duplicate logic to reduce delay, or change the layout style of the component (such as selecting a bit-sliced layout instead of a layout of random-logic gates). These type of improvements are nearly impossible to pursue once the design has been expanded into lower level logic.

The remainder of this paper is organized as follows. Section 3 discusses general issues related to optimization of designs from a behavioral description into layout. Section 4 examines a system architecture that performs such synthesis. Types of microarchitecture optimization are the focus of Section 5 and strategies for their

application are presented in Section 6. Finally results of using microarchitecture optimization are examined in Section 7.

2. Design Synthesis Process

Transforming a behavioral description into layout requires the work of a number of stages: behavioral synthesis, microarchitecture optimization, logic optimization, floorplanning, and layout. This section describes the goals and interactions of these tools and how microarchitecture optimization fits into the larger picture.

Behavioral synthesis tools convert a behavioral description into a dataflow graph with each node representing a functional operator (such as add or compare) [CaRo85] [OrGa86] [McPa88]. These operations must be assigned to a control step, through the process of scheduling [PaGa87] [PaKn87], that chooses a point in time at which the operation will be performed. In addition, the operator is assigned to a particular hardware module, through the process of binding [TsSi86] [PaPM86]. During this process, the synthesis tool explores multiple designs and attempts to determine which designs appear most likely to meet the set of user constraints. Estimators are employed to guide the synthesis tool towards one or more such designs.

Estimates for behavioral synthesis tools are usually obtained in one of two fashions. The first technique uses a set of formulas that when given a component type (ALU, Register, etc.) and its set of parameters (eg., number of inputs, archi-

tecture style, technology-type) produces a rough estimate of the time and area. Such estimates are not finely tuned but help to weed out unacceptable designs. A second technique is to expand the design into a lower level design consisting of gates, possibly even mapping the gates into a technology-specific library. Alternatively, a high-level floorplan of the microarchitecture components can be generated to obtain a feel for design characteristics. These methods require more time to produce the estimates and will usually be reserved for use when the number of possible designs has been greatly narrowed.

The use of estimators allows the behavioral synthesis tool to select an overall architecture by making decisions on the number of busses, use of pipelining, etc. In addition, the synthesis tool attempts to minimize the number of connections between modules and reduce the total number of modules. An appropriate architectural style must also be chosen for each microarchitecture component, such as ripple-carry or carry lookahead when using an adder. Because the estimates are only a rough predictor of the final design after layout, more rigorous analysis and optimization is required of the microarchitecture design. Thus there is a need for a microarchitecture optimization tool.

The major goals of microarchitecture optimization are to: (a) remove inefficient constructs (such as replaceing two multiplexors that have the same set of inputs with a single multiplexor), (b) select a style of architecture for each component that suits the area/time requirements, (c) insert buffers on outputs that have a high

fanout, (d) select which microarchitecture components to combine and perform logic optimization on as a single unit, and (e) select a layout style for each microarchitecture component such as PLA, random logic, bit-sliced, etc. Once the initial microarchitecture structure has been cleaned up, the optimizer has two options in producing the final design: (a) completely expand and optimiz e or (b) only partially expand and optimize. The first approach is to combine all components into a single combinational block and optimize. Logic optimization tools have been shown to be very effective for reducing the area of a design or restructuring logic to meet timing constraints. This approach may not be the best, however. First, logic optimization of large designs may require large amounts of CPU time and memory. The same will be true in the layout phase when floorplanning is performed. Second, some optimizations can be made at the microarchitecture level that cannot be made at the logic level. These optimizations include changing the architectural style of the microarchitecture component or changing its layout style.

The second approach involves only a partial expansion of the design. Various groups of the components can be combined into a single component and optimized. For example, random logic gates can be grouped together and passed to a logic optimization tool while more regularly structured components such as ALUs are optimized separately and not combined with the surrounding logic. Layout module generators can also be employed in this approach. Module generators can be used for components with a regular style of architecture such as ALUs and registers. For these components a one-bit layout slice is generated and then replicated based on

the component's bit width. The bit-sliced layout will typically be more compact than what could be generated using standard cell or custom layout generators to layout the same logic from a random logic description. Thus using module generators for components with regular structures will usually result in denser layouts. In some circumstances, however, a component such as a small ALU can be combined with surrounding logic to reduce the number of gates. This saving of gates may produce a smaller layout than if module generators had been used. Thus the microarchitecture optimizer must be able to discover such conditions.

After microarchitecture optimization, a floorplan must be generated for the design and a layout produced for each microarchitecture component. The floorplanner stacks the bit-sliced components in a vertical fashion and then places the random logic modules around the bit-sliced border. Components along the critical path should be placed close together to reduce the amount of delay caused by routing. Timing information can be supplied by the microarchitecture optimizer for this purpose.

3. System Architecture

This section describes a microarchitecture optimization tool and illustrates how it fits in to a larger system that synthesizes layouts from VHDL behavioral descriptions. The system architecture is shown in Figure 1. It consists of six major pieces: a component database, logic optimization tools, a behavioral synthesis tool, a technology mapper, a microarchitecture optimizer, and a floorplanning/layout system.

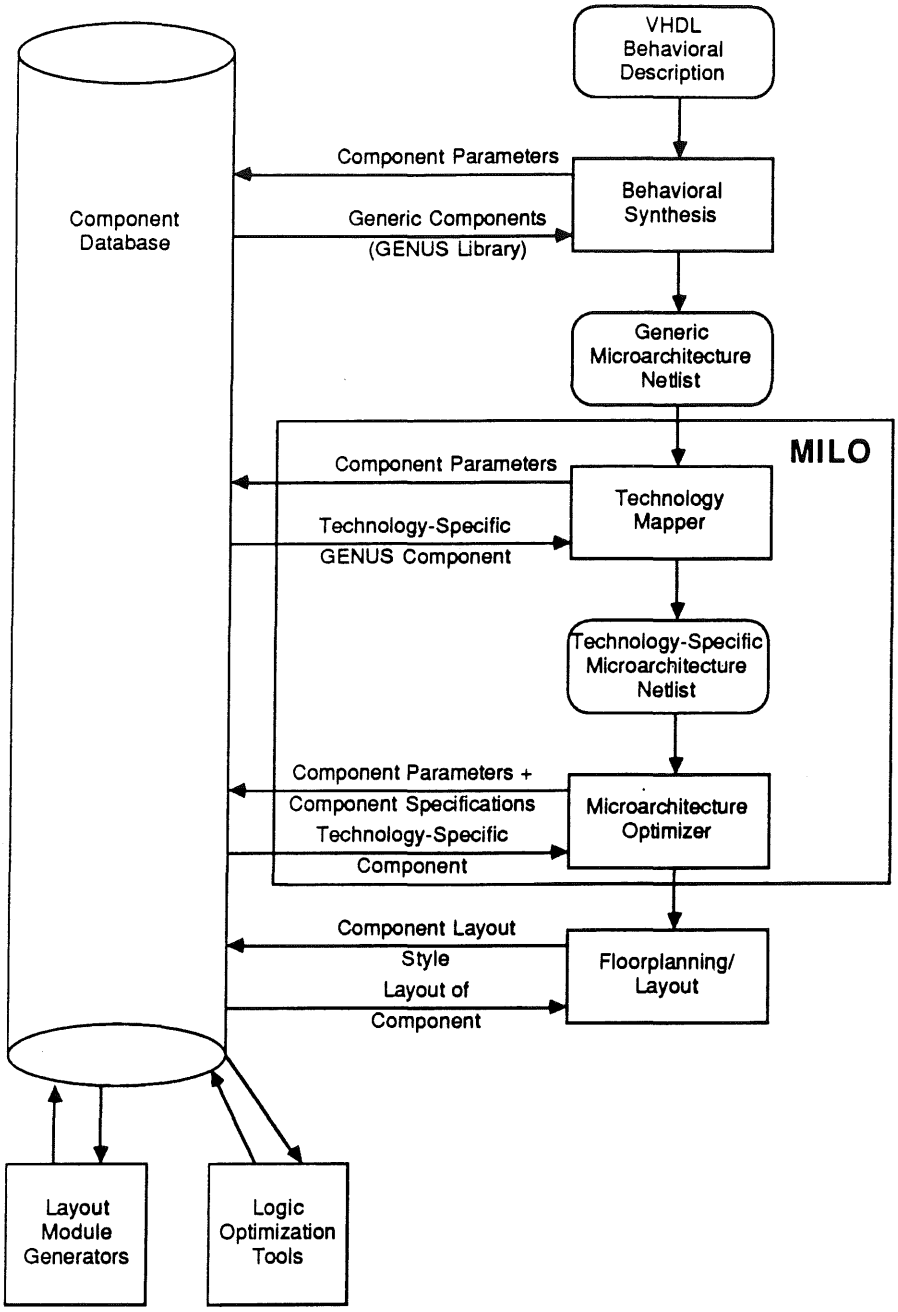


Figure 1. MILO Environment

3.1. Component Database

The central system tool is a component database [ChGa90]. It supplies components and statistics on components to the synthesis tools. Synthesis tools can pass a set of parameters and specifications to the database and then receive a list of components that meet the requirements. Parameters include the component type (eg., ALU, Counter, MUX), number of inputs, clock type (rising-edge, falling edge), etc. Specifications include the load that each output pin must drive, the maximum delay to each output pin, and an area requirement.

The component database contains a library of logic generators that produce a boolean equation representation that describes the low-level behavior of the component. One or more generators can be selected based on the parameters supplied by the synthesis tool. The boolean equations include constructs for describing sequential logic so that logic generators for components such as registers and counters can be constructed. The boolean description is passed to a logic optimizer [VaGa88] with a set of time constraints. The logic optimizer produces a technology-specific design using components from a designated library or can generate complex gates and select transistor sizes for use in a custom layout. The logic optimizer produces a report file listing delays and area. This information can be passed to synthesis tools when they request such information about a component.

The database also contains knowledge about components that can be produced by layout module generators. Estimators provide data on delay times and area

based on the bit-width.

3.2. Behavioral Synthesis

A behavioral synthesis tool [LiGa89] accepts a VHDL behavioral description and produces a VHDL structural netlist consisting of generic components from GENUS [Dutt88], a library of generic microarchitecture components. One special property of GENUS components is the use of one control line per function. Thus a four-bit multiplexor has four data-in lines and four select lines -- one to control each data line. In an ALU, there are separate control lines for ADD, SUBTRACT, AND, OR, etc. This component property removes the problem of control encoding from behavioral synthesis as component encodings may depend on a particular technology library. If necessary control encoding can be performed later during technology mapping.

The behavioral synthesis tool begins by converting the input description into a dataflow graph. A graph critic then operates on the dataflow graph, removing redundancies in the behavioral description. The behavioral operators are then bound to GENUS components. The final architecture produced by the behavioral synthesis tool consists of random logic blocks of control logic and a datapath containing components such as ALUs, shifters, and registers.

The components in the generic netlist are converted to technology-specific components by a technology mapper. The technology mapper queries the database by

providing the set of component parameters. The database returns one or more components that meet the specified parameters. From this set of components the technology mapper selects the component that contains the smallest set of functions required. For example, if a component with the ADD and SUBTRACT functions is requested, the database may return two components: an ADD/SUBTRACT unit and an ALU. The technology mapper would select the ADD/SUBTRACT unit. Since the technology mapper does not pass a set of timing or area constraints to the database, the database will return the most area efficient design. Currently the technology mapper maps generic components into only components that are implemented from gates and optimized by the logic optimizer. Later implementations will include mappings of other types of components, such as those from layout module generators. In any event, these types of components are currently inserted later, during the microarchitecture optimization phase if appropriate.

3.3. Microarchitecture Optimization

At this point the design consists of two levels. One is the microarchitecture netlist, the other is a technology-specific gate-level netlist for each microarchitecture component. The microarchitecture optimizer first employs rules that make transformations that should improve both time and area. For example, converting a register and incrementer into a counter. Next the critical paths are identified. The optimizer requests faster components from the database, selects different layout styles (random logic or bit-sliced), and decides which components to merge and

apply logic optimization. Once critical paths have been processed, the microarchitecture optimizer operates on non-critical components, making similar decisions as in the critical path improvement phase but this time with an eye toward area improvements. The microarchitecture optimizer then produces a VHDL netlist that is passed to the floorplanner/layout assembler for layout.

The microarchitecture optimizer uses a new methodology for selecting microarchitecture components to be used in the design. The microarchitecture optimizer does not perform component rearchitecting and does not have knowledge of tools for logic optimization, transistor sizing, and other component reoptimization techniques. Instead, these tasks are left to the component database. The microarchitecture optimizer passes a set of time/area constraints to the database and the database examines possible ways to achieve the constraints. The database can choose from different architectural styles and can choose from multiple optimization tools to redesign the component. This frees the microarchitecture optimizer from dealing with technology concerns and having to know what set of component optimization tools exist at any one time. All of this is centralized in the database.

Integration of the database with the microarchitecture optimizer and the logic optimization tools is achieved with two servers [ChGa89] as shown in Figure 2: a component server, and a knowledge server. The component server is the part that interfaces with the microarchitecture optimizer. Queries are made from the microarchitecture optimizer to the component server through the Component Query

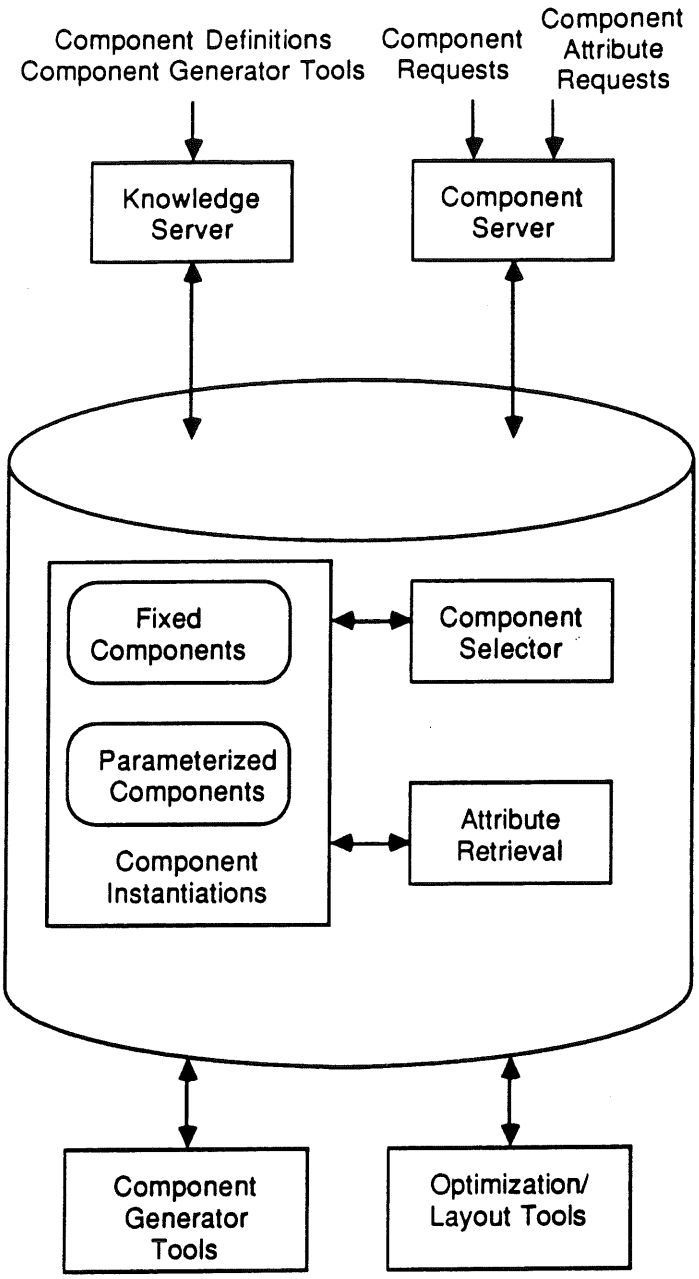


Figure 2. Tool Interface with the Component Database

Language (CQL) and a list of components or a set of component attributes are returned. In this manner, the microarchitecture optimizer can simply request the functions required of a component, an layout implementation style, and a set of delay parameters. From this information the component database checks its component list which includes fixed components (components that have already been generated) and parameterized components (those that can be generated when provided a set of parameters). The component database knows from its component list whether a component generator needs to be called to generate a design for the component or whether the component design already exists (as in the case of a fixed component). Once a component is generated, the database can call an appropriate logic optimization tool or layout tool.

The knowledge server is used to insert new fixed components, insert new component generators, and insert logic optimization and layout tools. Thus when a new logic optimization or layout tool is available, the knowledge server will be accessed to store information about how to call the new tool. Also designers can build their own components and insert them into the component database through the knowledge server.

3.4. Floorplanning/Layout

Finally the technology-specific microarchitecture netlist is passed to a layout synthesis system that performs floorplanning on the microarchitecture design and creates a layout for each component [WuGa90]. The layout tool decides how to

partition the random logic into blocks for layout. It also can modify the microarchitecture optimizer's selection of bit-sliced components, converting them to random logic if doing so will result in a better layout. Module generators are called to produce the bit-sliced layouts and a custom layout generator called to produce a layout for each random logic block. The floorplanner selects how to partition the random logic based on shape sizes that can be used to fill in the bit-sliced logic mismatches.

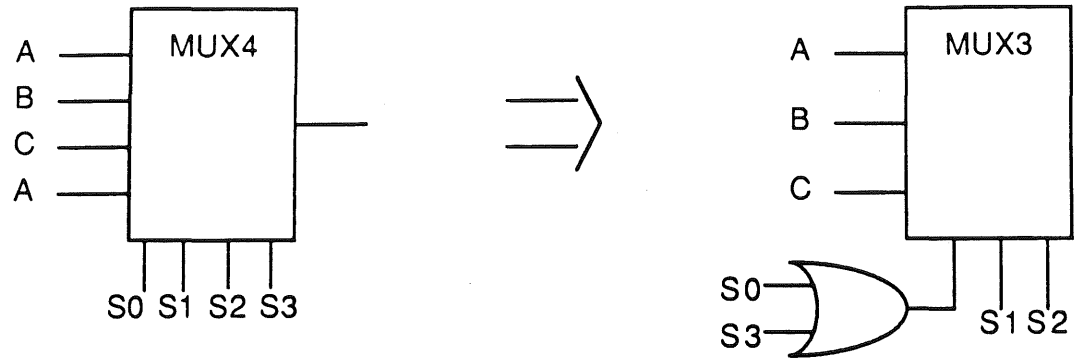
The floorplanner attempts to place similar-sized bit-sliced components together and place random logic into slots where mismatches in the length of the bit-sliced logic occurs. Other random logic is placed along the border of bit-sliced components.

4. Types of Microarchitecture Optimization

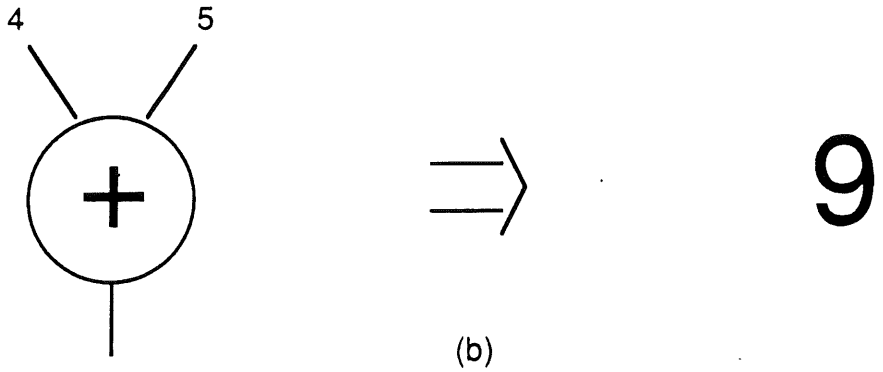
The goal of microarchitecture optimization is to optimize the design for area/time without changing the state assignment. This section describes the types of optimizations that can be performed.

4.1. Minimization

This type of optimization should be one of the first to be applied. It reduces the number of components or the amount of logic in a component. Figure 3(a) and Figure 3(b) show examples of minimization rules. Figure 3(a) shows the removal of the redundant signal A as an input to the multiplexor. Figure 3(b) shows the



(a)



(b)

Figure 3. Minimization Rules

replacement of an adder by the sum of its two constant values.

4.2. Factorization

Factorization is used to extract early arriving signals in order to speed up late arriving ones. It may also be necessary to factor components in order to meet the requirements of a layout module generator. For example, module generators may only be able to construct 4 to 1 or 2 to 1 multiplexors. Figure 4 illustrates the factorization of a multiplexor.

Procedure 4.1 describes the factoring algorithm. The algorithm factors a single component having $|\mathbf{R}|$ inputs, $\mathbf{R}$ being the set of all required input to output delays. The procedure *Factor* is recursive and takes five parameters: 1) c, which indicates

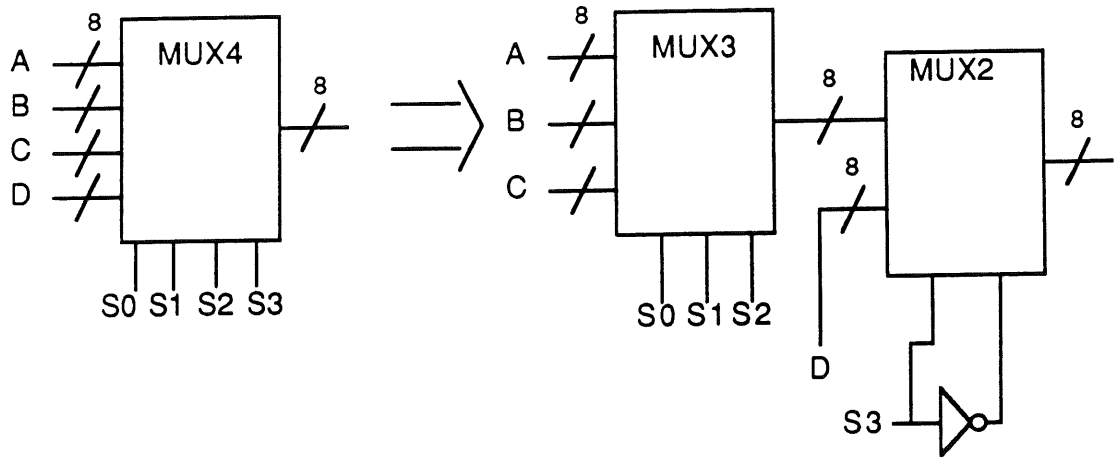


Figure 4. Factorization of Microarchitecture Components

```
Let:R={r | required delay from input to output};
      each component C_i has delay d_i and s_i inputs;
    C_0 be the component to be factored
    n=number of component inputs that still need to be assigned;
    s=last tried component size that failed to meet the constraints;
    D_s=smallest delay through any multiplexor that can be generated by the database

Function Factor(c,R,n,s,C_0)
  Begin
  start:
    n_t = n;
    R_1 = R;
    C_1 = find_new_component(R, n, s);
    if (C_1 ≠ φ)
      if (c > 0)
        assign C_1 to the cth input of C_0
      for(i=1;i≤s_1;i++)
        if(min(r-d_1)> D_s && ((n-s_1+i)>1))
          n = n - Factor(i,R={r|r=r-d_1},n-s_1+i,s_1, C_1);
        else
          r_s = smallest r in R;
          assign r_s to i-th input of C_1;
          R = R - {r_s};
          n = n - 1;
        if(n==0) return(n_t);
      if(|R|==n)
        /* Not able to assign all inputs, try again */
        n = n_t;
        s = s_1;
        R = R_1;
        goto start;
  return(n_t-n);
  End
Function find_new_component(R,n,s)
  Begin
  largest_allowable_delay = min(r);
  max_number_of_inputs = min(s-1, n);
  if (there exist database components C_i such that
      s_i ≤ max_number_of_inputs && d_i ≤ largest_allowable_delay)
    select component C_1 such that s_1 ≥ all s_i
  else
    C_1 = φ ;
  return(C_1);
  End
```

Procedure 4.1

which input of the parent component the factored out inputs should be connected to, 2) the set **R**, 3) n, the maximum number of inputs to be factored out of the parent component, 4) s, the size of the last component that failed to meet the constraints, and 5) C_0, the component to be factored.

The factoring algorithm begins by sorting the set of required delays, **R**, from smallest delays to largest delays. Then the database is queried to find the same type of component but with fewer inputs. For example, consider Figure 5. In Figure 5, the database is shown to have returned three components having six or fewer inputs. The 2-input multiplexor has a delay of 2ns, the 4-input multiplexor has a delay of 5ns, and the 6-input multiplexor has a delay of 7ns. Figure 5 shows the factoring process for a six to one multiplexor. The set of required delays, **R**, is shown to be (5, 5, 6, 6, 7, 9) for inputs A through F, respectively. Since the six input multiplexor did not meet the required delays, the next smallest one is selected. In this case it is the four input multiplexor.

The next stage is to assign the inputs to this new component. All inputs whose required delays will not be satisfied if they are factored out (ie., the delay through the new component + the smallest possible delay through any component of the same type) are connected directly to the new component. The remaining signals represent those that can be factored out. The algorithm queries the database again to find the component with the most inputs that will still meet the timing constraints when the signals are extracted. This component will then be processed recursively in a similar manner. When a solution is found that meets the time

List of multiplexors returned by the component database:

(size, delay) = (2,2) (4,5) (6,7)

Set of required delays R = (5,5,6,6,7,9) (A B C D E F)

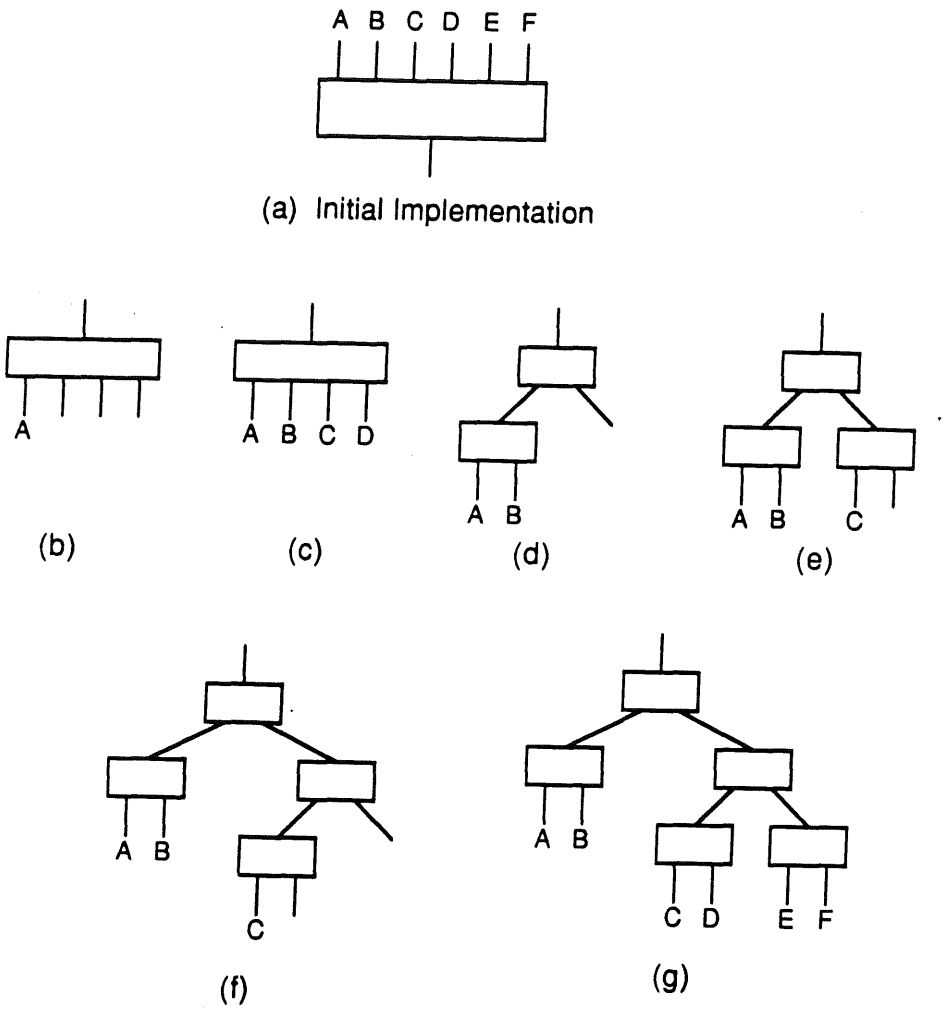


Figure 5. Example of Factorization

constraints, the algorithm ends.

For the example of Figure 5, input A cannot be factored out of the 4 to 1 multiplexor or its timing constraint of 5 will not be met. That is, the delay of the four input multiplexor (with delay of 5) plus the delay of the smallest multiplexor (2-input MUX with delay of 2) is greater than the required delay of 5 for input A. For this reason, input A is connected directly to the 4-input multiplexor (Figure 5(b)). The set **R** then becomes (5, 6, 6, 7, 9) with only five more inputs to be assigned. For similar reasons, inputs B, C, and D are assigned directly to the 4-bit multiplexor as shown in Figure 5(c). At this point, not all inputs have been assigned and there are no unused multiplexor input ports. Therefore, using the 4 to 1 multiplexor has failed. The set **R** is reset to the original (5, 5, 6, 6, 7, 9) and another attempt is made using a smaller multiplexor. If a 2 to 1 multiplexor is used, inputs A and B can be factored out using a second 2 to 1 multiplexor (Figure 5(d)). The time constraints are still met and the new set **R** is (6, 6, 7, 9). The largest multiplexor that can be used to factor out input C is a 2 to 1 multiplexor (Figure 5(e)). In addition, input C can be factored out again using another 2 to 1 multiplexor and the time constraints are still met (Figure 5(f)). In a similar fashion, inputs D, E, and F can be assigned as in Figure 5(f).

4.3. Swap Equivalent Signals on the Same Component

If two signals on a component are interchangeable and one has less delay than another, the early arriving signal can be swapped with the late arriving signal.

Figure 6(d) demonstrates how this can be accomplished. Swapping of component pins can be described as follows. Let $\mathbf{I}(c) = \{i_j \mid j = 1..n\}$ be a set of equivalent inputs to a component c, where $i_j = \{a_j, r_j, s_j\}$. a_j is the arrival time, r_j is the required time, and $s_j = r_j - a_j$ is the slack.

Let $\mathbf{T} = \{i_j \mid s_j < 0 \; j = 1..n\}$ be a set of critical inputs, $\mathbf{N} = \{i_j \mid s_j \geq 0 \; j = 1..n\}$ be a set of non-critical inputs. Sets T and N can then be sorted according to each pin's slack. Swapping of pins then takes place as shown in Procedure 4.2. The algorithm tests whether a pin from **T** can be swapped with a pin from N. If doing so does not create a new critical path, the pins are swapped.

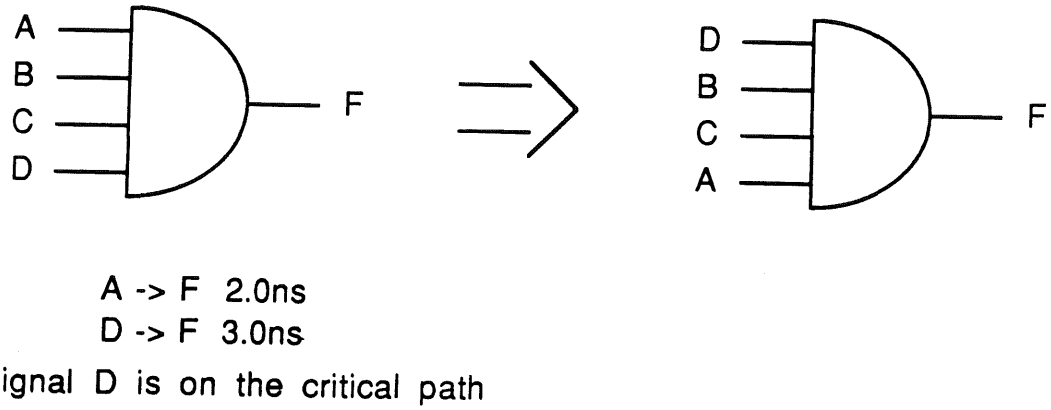


Figure 6. Signal Swapping

Let ABS() be the absolute value function

```
Procedure Swap_Pins (T, N)
  Begin
  k = 0
  For j=0 to |T|
      Begin
      i_j = the jth pin of T;
      s_t = the slack of pin i_j;
      i_k = the kth pin of N;
      s_n = the slack of pin i_k;
      If ABS(s_t) ≤ ABS(s_n) then
          Begin
          swap(i_j, i_k);
          k = k + 1;
          End If
      End
  End
```

Procedure 4.2

4.4. Merge Similar Units

Two components can be merged when one of them performs a subfunction of the other. For example, in Figure 7, combining a register and shifter into a register that performs a shift. Merging rules examine connectivity between two components and their functionality. Functionality of components can be found by querying the database for a list of functions that the microarchitecture component performs. The merge can be performed for two components, c_0 and c_1, when function(c_0) $\subset$ function(c_1). For example, in Figure 7, the function *shift* is a function that can also

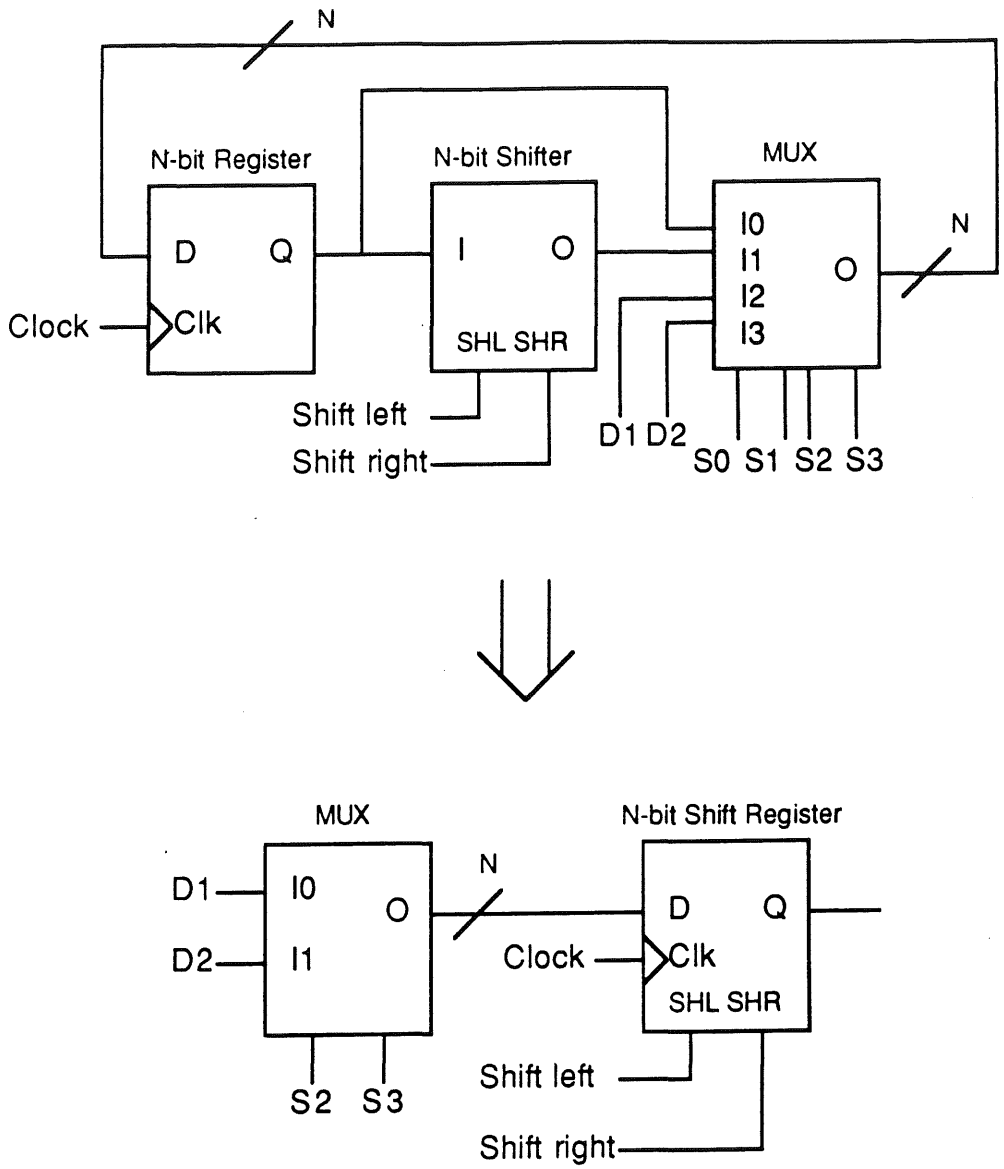


Figure 7. Merge Similar Units

be performed as part of the register component. Thus a register that does not perform a shift and a shifter can be combined into a single shift register.

Merging of similar components is accomplished in two subphases: 1) same type component merging, and 2) different type component merging. Same type component merging is accomplished by an algorithm that proceeds from the design's input pins to the design's output pins, examining whether two components that are of the same type are connected together (eg., two multiplexors, two adders, etc). The algorithm checks a list of valid component types for merging. If a match is found, the merging procedure continues, otherwise the next set of components is examined. Then, there are three cases that occur when merging components:

(1) A component is only connected to a component of similar function to itself as in Figure 8(a).

(2) A component is connected to multiple components, some of which are of a similar function, some of which are of a different function. An example of this is in Figure 8(b).

(3) A component is connected to multiple components, all of which are of the same function type.

For case 1 occurrences, the two components are merged. Thus the design of Figure 8(a) becomes the design of Figure 9(a). In a case 2 occurrence, the two components c_1 and c_2 are merged to create a new component, but c_1 must remain connected to those components which are of different types. Thus the design of Figure

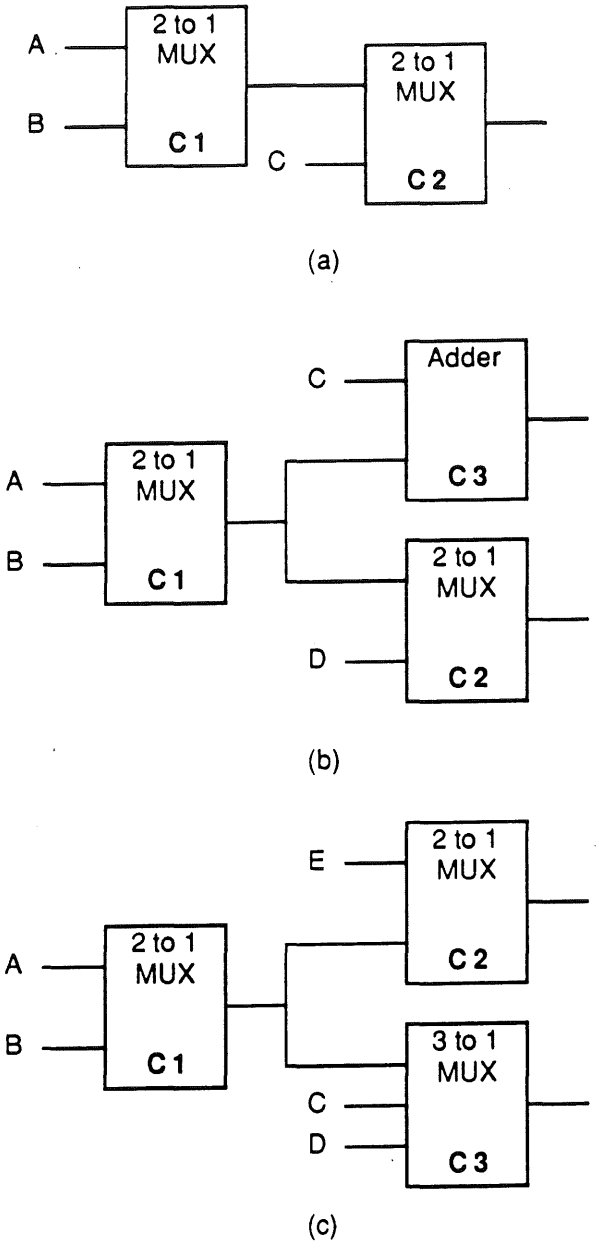


Figure 8. Three Possible Merging Cases

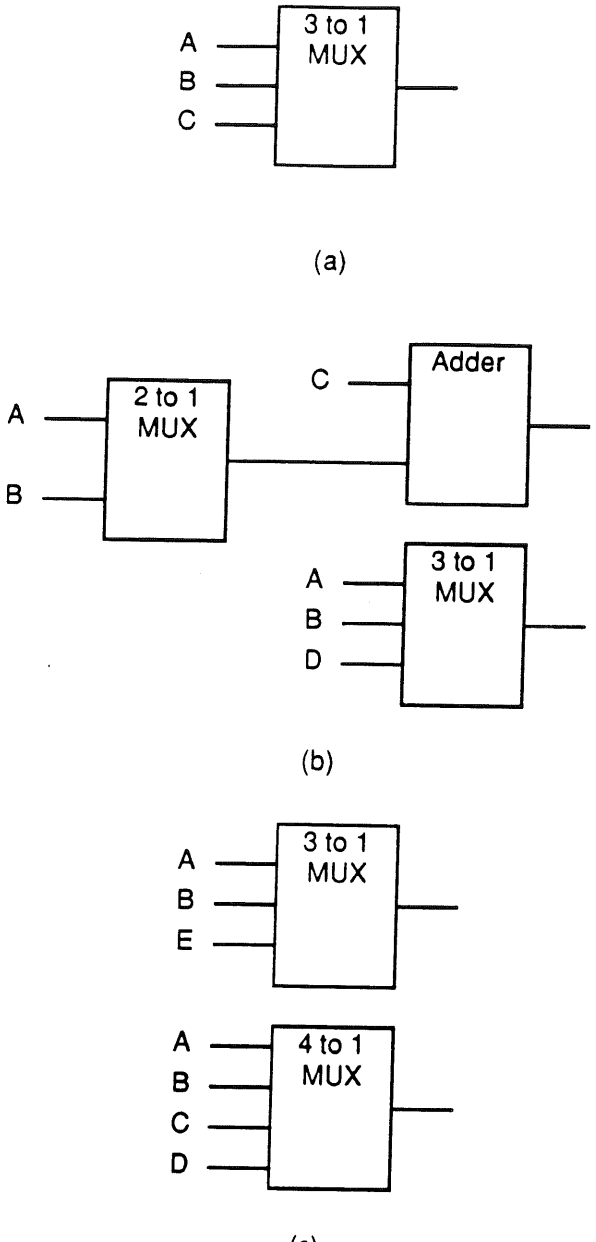


Figure 9. Results of Merging

8(b) becomes that of Figure 9(b). In case 3 occurrences, component c_1 is merged with all components that its output is connected to. For example, the design of Figure 8(c) becomes the design of Figure 9(c). Though the design of Figure 9(c) is more expensive than that of Figure 8(c) it is used as an intermediate step in optimization. This is discussed in further detail later.

Merging of different type components is performed using rules. There is one rule for each type of merge operation. For example, a rule to perform the optimization of Figure 7 is shown in Figure 10. If the connectivity of the components is found to be similar to that of Figure 7, then the component database is queried to produce the new set of components which are substituted into the design.

If there is a component C1 with functionality = register
AND there is a component C2 with functionality = shifter
AND output Q of C1 is connected to input I of component C2
AND there is a component C3 with functionality = multiplexor
AND output Q of C1 is connected to input I of C3
AND output O of C3 is connected to input D of C1

Then

C4 = Query Component Database for a shift register
C5 = Query Component Database for a multiplexor with two fewer inputs than C3
Replace C1, C2, and C3 with C4 and C5

Figure 10. Rule for Merging

4.5. Merge Unsimilar Units

Two components can be merged into a single component that performs a different function than any of the original units. For example, combining a register and incrementer into a counter, as in Figure 11. Rules for this type of merging are similar to those for merging similar functional units. In this case, however, for two components, c_0 and c_1, function(c_0) $\cup$ function(c_1) $\subseteq$ function(C_l), where C_l is a component that can be generated by the component database. For example, in Figure 11 the register and incrementer are both subfunctions of a counter component that can be generated by the database. Mergeability can be determined by querying the database with a list of functions desired in a component to determine if such a component can be created.

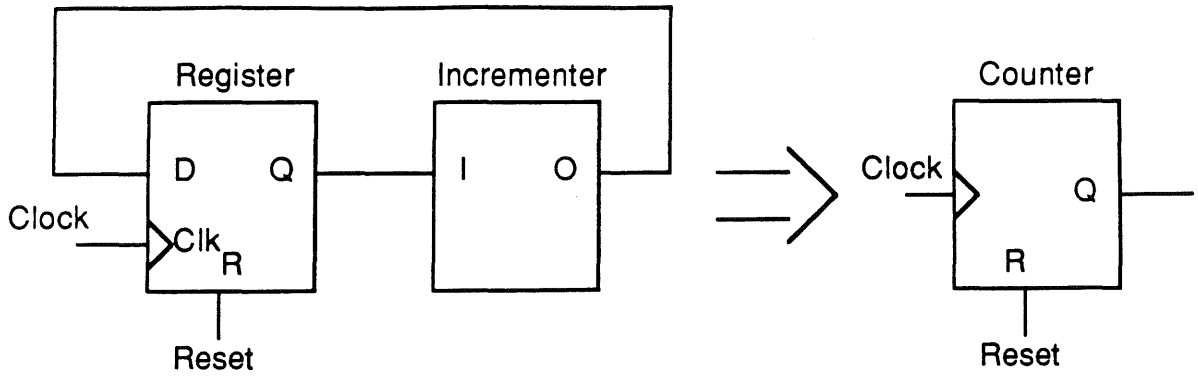


Figure 11. Merging Unsimilar Units

4.6. Style Change

The optimizer can query the component database to request a component that performs the same function(s) but is faster or has a smaller area. The database returns a list of components from which the optimizer can select one based on the time and area requirements. Part of the database query can include a layout style request. For components having a bit-sliceable architecture, such as ALUs, the optimizer will request a bit-sliced layout style. By placing the component in the bit-sliced datapath, routing area can be reduced. As mentioned earlier, bit-slices usually tend to be faster and smaller than their equivalent random-logic implementation. Transistor sizes in the designs produced by layout module generators are fixed, however. In some cases larger transistor sizes may be required for drive capability and speed. Buffers can usually be inserted to add greater drive capability. For greater speed, however, larger transistor sizes for gates in a design typically decrease the delay. Thus producing a random-logic design with larger transistor sizes than those used in the bit-sliced cell may result in a faster design. As standard cells have fixed transistor sizes, a transistor sizing program, such as [WuVG90], can be combined with a custom-layout generator to produce the layout for the component. Estimators that calculate delays for bit-slice logic, based on a single slice, and for random-logic, based on gate type and transistor sizes, assist the microarchitecture optimizer in determining which design will be faster.

The database searches through its list of different architectural styles for the component to select one that it estimates will come closest to meeting the specified constraints [ChGa90]. For each style, the database maintains a range of delays and area that can be obtained. Then, depending on the layout style, the database can call tools such as logic optimizers, transistor sizing tools, etc., to generate the low level design in terms of gates or a layout. In this manner, the microarchitecture optimizer is freed from the low level details and is not concerned with which low level optimization tools should be called.

4.7. Duplicate Logic

Duplication of components is a technique designed to improve the speed of a path at the cost of additional area. It is the reverse of factorization. Figure 12 shows the duplication of the two-input multiplexor in order to reduce the delay along a critical path.

4.8. Merge Multiple Components and Optimize

This technique combines components performing different functions into a single unit and then applies logic optimization. Optimization of this type can be particularly effective when some of the inputs to the components are constants. The optimization of the constants will propagate through the logic. Thus in cases where the microarchitecture optimizer believes constant propagation in the logic will occur, it will merge even bit-sliceable components, optimize them, and treat them as

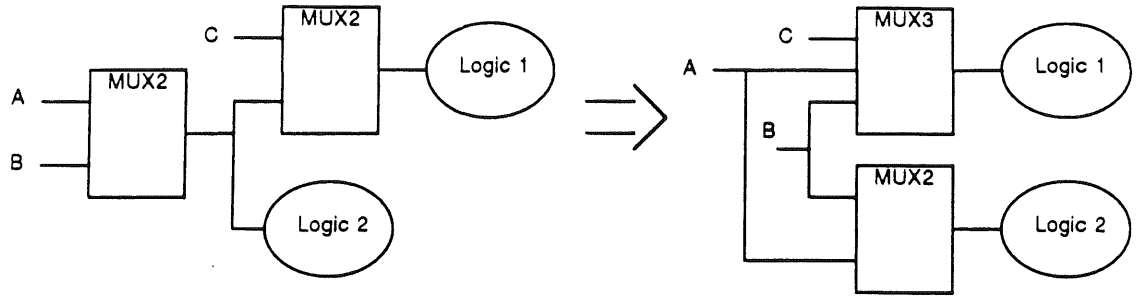


Figure 12. Component Duplication

random logic. Constant propagation is obvious when a number of the component's inputs are constants. Components connected to the output of such a component should also be combined into the random logic since the constants can usually be propagated through several levels of microarchitecture components.

4.9. Extraction of Common Subexpressions

Designs can often have the same logic duplicated in different parts of the design. Local transformations will not detect this. Therefore, global analysis is required to find and extract such common subexpressions. An example of common subexpression extraction is shown in Figure 13.

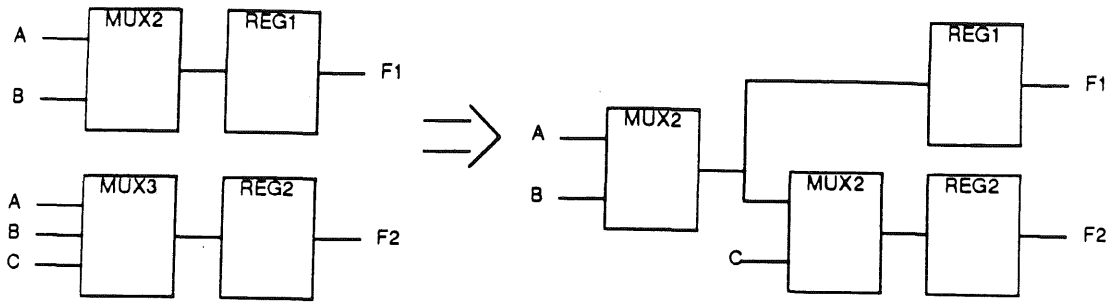
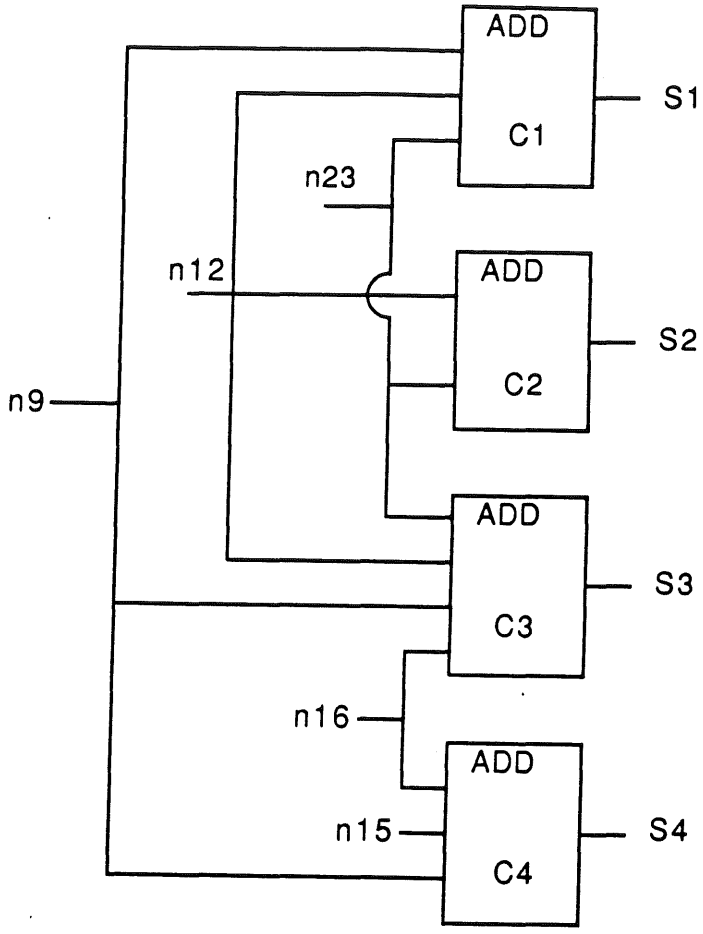


Figure 13. Common Subexpression Extraction

Common subexpression elimination is performed for each component type. For example, it will be performed separately for multiplexors and adders. The algorithm consists of three steps: 1) for each component which is of the selected component type, a set **N** is generated that contains all the inputs to that component, 2) a set **L** of possible subexpressions is generated, 3) a common subexpression is selected and extracted. This process is repeated until no more subexpressions are present.

As an example, consider Figure 14. In this example, the component type is *adder*. For each adder (components c_1, c_2, c_3, and c_4), the set $\mathbf{N}_i$ is generated. Every net is assigned a unique id number (for example, nets n12 and n23 in Figure 14). Two components that have an input connected to the same net have that net id number in common. Each set **N** is sorted by the net id numbers. From Figure 14,



N1 = {n9, n12, n23}

N2 = {n12, n23}

N3 = {n9, n12, n16, n23}

N4 = {n9, n15, n16}

Figure 14. Common Subexpression Elimination

the N sets generated are: N_1 = {n9, n12, n23}, N_2 = {n12, n23}, N_3 = {n9, n12, n16, n23}, and N_4 = {n9, n15, n16}.

The second step is to identify possible common subexpressions. A common subexpression s_e is present if the following expression is true: $|N_i \cap N_j| > 1$. A set of common subexpressions, S_{ij}, is generated for each $N_i \cap N_j$. Each set S_{ij} must have at least two elements or be the null set as only two or more inputs can be extracted. From the four N sets generated above, the sets S_{ij} are as follows: S_{12} = {n12, n23}, S_{13} = {n9, n12, n23}, $S_{14} = \phi$, S_{23} = {n12, n23}, $S_{24} = \phi$, and S_{34} = {n9, n16}. A set **L** is created from the **S** sets. It contains no duplicated entries but instead keeps a count of the number of occurrences for each subexpression. Thus the set **L** is {{n12,n23}:2, {n9,n12,n23}:1, {n9,n16}:1}.

From **L** a subexpression for extraction is chosen using the following criteria: a) most number of occurrences, and b) smallest subexpression. In the case of Figure 14, the set in **L** with the largest number of occurrences is {n12,n23}. Figure 15(a) shows the new design after the common subexpression is extracted. The set {n12,n23} is removed from **L** and any set containing {n12,n23} as a subset (for example the set {n9, n12, n23}) is replaced with the net id for the extracted subexpression (in this case, n30 as shown in Figure 15). The new set **L** is {{n9, n30}:1, {n9,n16}:1}. Since both sets have the same number of occurrences and the same size, the first set {n9,n30} is selected. Figure 15(b) shows the design after the extraction of this set. The new set **L** for the design of Figure 15(b) is {ϕ }. Therefore there are

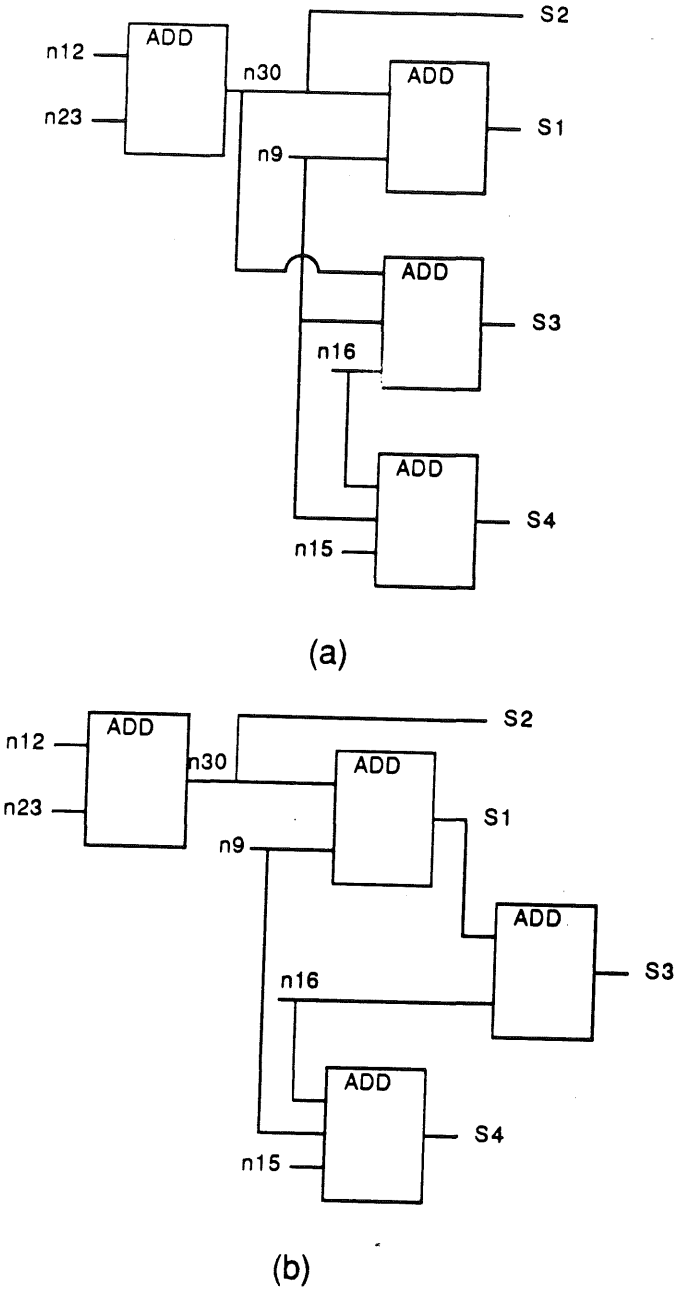


Figure 15. Common Subexpression Elimination Example

no more subexpressions to extract.

4.10. Addition of Buffers

Some components that drive large loads may require the addition of buffers at their outputs. This can reduce the delay by providing greater drive power. Methods of doing this have been discussed in [GuPa90] [SiSV90]. One solution is to partition the load by constructing a fanout tree from buffers. This tree should be constructed in a manner that does not violate the time constraints yet minimizes the amount of area increase.

[GuPa90] also mentions that in standard cell designs, components with higher drive capacity can be selected. Alternatively, some duplication of logic can be used to reduce fanout. In our case, the component database contains tools for transistor sizing and can generate a layout using a custom layout generator. This allows the design to be more finely tuned than when using standard cells. With the custom layout capability component transistor sizes are not fixed at discrete intervals. Rather, transistor sizes can be selected on a continuous basis in order to meet delay requirements.

5. Strategies for Microarchitecture Optimization

Having examined types of optimization techniques, we now describe an algorithm for applying them. A block diagram of the optimization process is shown in

Figure 16. It is divided into three parts: a data structure, a control section, and a set of optimization procedures. The data structure contains the design netlist, statistics for delay and area, a set of user constraints, a set of critical paths, and a set of non-critical paths. Critical and non-critical path sets are determined by the timing analyzer in the control section. The controller also selects which optimization

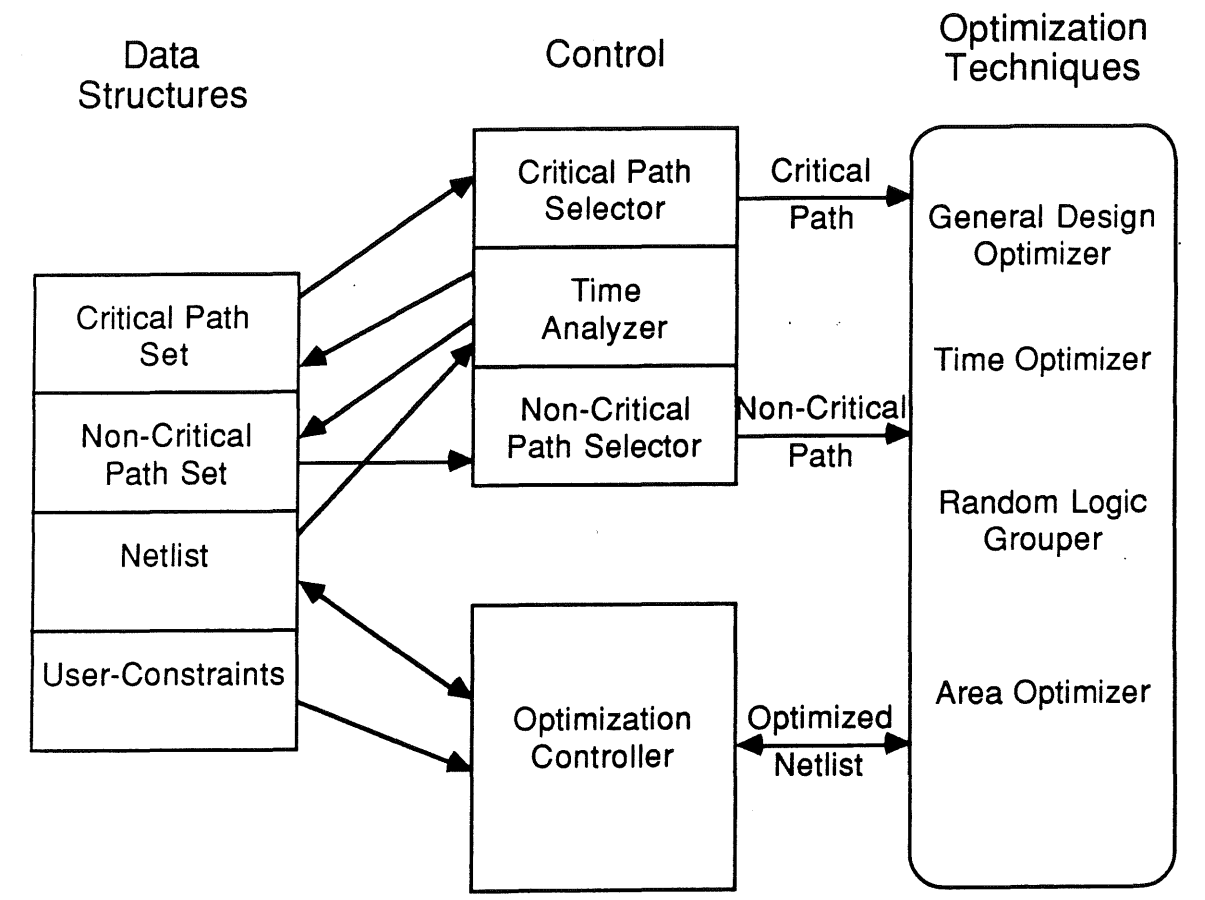


Figure 16. Overview of Microarchitecture Optimization

procedure to use. Each optimization procedure corresponds to one phase of the optimization process. The microarchitecture optimization is carried out in four phases: general design improvement, random logic grouping, timing optimization and area optimization.

Phase 1 of the algorithm, general design improvement, attempts to reduce the number of components and to prepare the design for timing optimization should that be necessary. For example, to be able to refactor multiplexors along the critical path, all multiplexors that can be merged should be merged into a single multiplexor. Then the timing optimizer can decide how to refactor the single multiplexor. Thus optimizations in this phase set up techniques that will be performed later or employ techniques that improve both the time and area of a design.

Phase 2 groups random logic components for logic optimization. Microarchitecture optimizations are not performed on random logic gates. Instead, they are passed to the database which has tools for restructuring the logic to meet a set of constraints passed by the microarchitecture optimizer. Thus Phase 2 prepares the design for Phases 3 and 4 by reducing the number of components that the microarchitecture optimizer must deal with. In doing so, it groups components that will be implemented using random logic gates rather than a bit-sliced layout.

Phase 3 applies time reduction techniques. The microarchitecture design is originally tuned for area by the technology mapper. Therefore, in this phase, the microarchitecture optimizer operates on critical paths, making necessary time for

area tradeoffs.

Phase 4 works on non-critical paths, attempting to reduce the design's area. During area optimization, some microarchitecture components may be merged with others for logic optimization. These types of optimizations must be performed after timing optimization because once components are merged, the microarchitecture optimizer cannot recognize the original component functionality. This information is necessary for some of the timing optimization techniques.

```
Procedure Optimize_Microarchitecture (microarchitecture design)
Begin
General_Design_Improvements(microarchitecture_design)
Random_Logic_Grouping(microarchitecture_design)
Identify critical path set
While (Critical path set is not empty)
   Begin
   critical_path = select_critical_path(critical_path_set)
   Timing_Optimization(critical_path)
   Remove critical_path from critical_path_set
   End
Identify non critical path set
While (Non critical path set is not empty)
   Begin
   non_critical_path = select_non_critical_path(non_critical_path_set)
   Area_Optimization(non_critical_path)
   Remove non_critical_path from non_critical_path_set
   End
End
```

5.1. General Design Improvements

To improve the overall design, Phase 1 proceeds as follows:

Procedure General_Design_Improvements (microarchitecture design)
Begin
Merge Similar Units (from inputs to outputs of the design)
Merge UnSimilar Units (from inputs to outputs of the design)
Apply Minimization Rules (from inputs to outputs of the design)
Perform Common Subexpression Elimination
End

Phase 1 begins with components of similar types being merged. As mentioned earlier, this is necessary for refactoring and common subexpression recognition. Further it reduces the number of components and hence makes minimization rules easier to apply. For example, performing the optimization of Figure 3 would be more difficult to discover if the multiplexor containing the common signal A were factored into two multiplexors, each containing the signal A.

Next unsimilar components are merged using a set of rules. These rules also reduce the number of components in the design. Once all merging is complete, a set of minimization rules can be applied to clean up redundant and unnecessary logic in the microarchitecture design. Up to this point, all optimizations reduce the number of microarchitecture components in the design. The next step, common subexpression extraction, increases the number of microarchitecture components but reduces the actual amount of logic required to implement them. It allows hardware that is redundant in a number of components to be shared. Common subexpression elimination is not performed on random logic as this can be performed by logic optimization tools.

5.2. Random Logic Grouping

Phase 2 collects certain types of components to be optimized together as random logic. This is performed as follows:

Procedure Random_Logic_Grouping (microarchitecture design)
Begin
Group Logic Gates into random logic components
Group Non-Bit-Sliceable Components into random logic components
Group Components with Constant Inputs into random logic components
End

Phase 2 groups components that will then be optimized as a single random logic component. During this phase, three types of components can be grouped: 1) gates, 2) components for which bit-slicing is difficult, and 3) bit-sliceable components that have constants as inputs. Type 1 components, gates such as NAND, AND, and XOR, each have a lower-level technology specific implementation. For example, at the microarchitecture level, one could have a 12-input AND gate. Of course, a gate with this many inputs is usually not physically implementable as a single gate. Thus the technology-specific design is constructed from smaller gates that are available in the specified technology. Phase 2 groups all random logic gates at the microarchitecture level that are connected together and forms a single component of type "random logic", as shown in Figure 17.

Type 2 components, for which bit-slicing is difficult, such as a decoder, will also be grouped with the random logic gates that they are connected with. Finally,

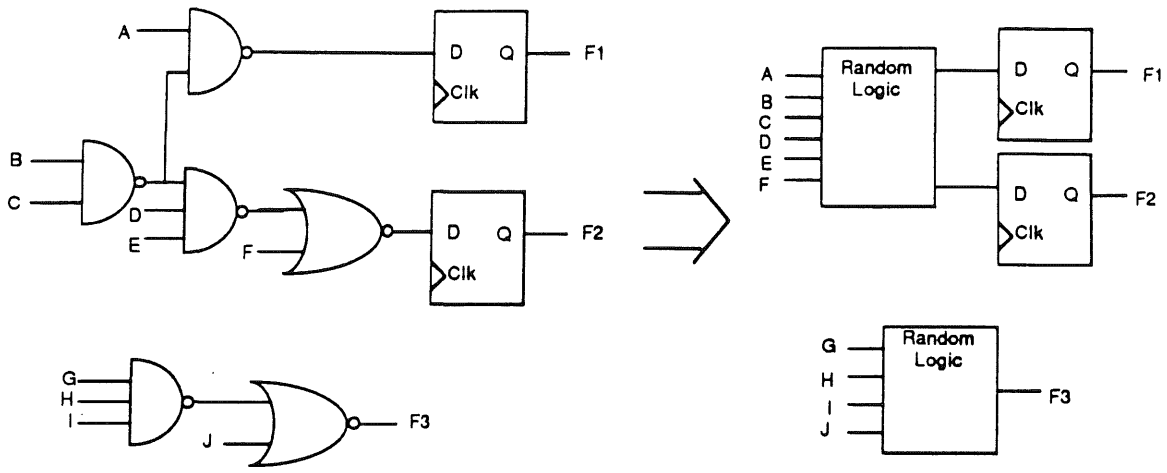


Figure 17. Random Logic Grouping

bit-sliceable components with constant inputs will be added to the random logic set. Components connected to the outputs of the type 3 components will also be grouped into the random logic since during logic optimization, the constants will often propagate through.

For each microarchitecture component, the database has a file containing a set of boolean equations that describe the behavior of the component. As mentioned earlier, the equations can represent sequential logic as well as combinational logic. The microarchitecture optimizer can request that the database create a new component by merging two component's equation files. Logic optimization on this new component can then be performed by tools in the database.

5.3. Timing Optimization

Timing analysis is performed as the first stage of Phase 3. Delays and setup times for each component can be found by querying the database. The timing analyzer calculates four types of worst delay: 1) input pins to registers, 2) register to register, 3) registers to output pins, and 4) input pins to output pins. The worst delays at the design's output pins are compared with the required delays that are entered by the user. Output pins with negative slacks do not meet the delay constraints. Slack is computed as:

slack = actual delay - required delay

Required delays are also calculated at each register data input. The required delay is calculated based on the required maximum clock width, which is entered by the user:

required delay = max clock width - setup time

Actual delays are calculated based on the worst delay to the register's input, the setup time for that input, and the worst delay to the clock input of the register:

actual delay = worst delay to register input + setup time -
worst delay to register clock input

The slack is then computed from the actual and required delay values. A slack value is found for every component's output pin in a similar manner by subtracting

the actual delay from the required delay.

After timing analysis, the goal of timing optimization is to make sure that no component's outputs have a negative slack value. Any component having such an output is said to be on a critical path. Ideally these negative slack values are raised to zero, with any value over zero representing over optimization (assuming area/time tradeoffs must be made).

Timing optimization is performed for each critical path. The worst critical path (ie., the one having the largest negative slack) is processed first. Timing optimization along the critical path proceeds as follows:

Procedure Timing_Optimization (Critical Path)
Begin
  Swap Equivalent Signals
  Factor
  New Component Style Selection
  Merge Multiple Components for Optimization
End

Phase 3 operates on microarchitecture components along the critical paths. It uses factoring, signal swapping, new component selection, and merging of components in order to reduce delay. The timing optimization phase ends as soon as there are no critical paths.

Signal swapping is performed first since there is no area increase associated with it. Usually, however, improvements in delay from this type of optimization are

small. Factoring is employed in the second step to produce shorter paths for critical signals. This technique usually increases the area only slightly. The set of required delays is calculated for each input to the output of the factorable component. These delays are passed to the factoring routine which then attempts to factor in a manner that will meet those delays.

In the third step of Phase 3, the optimizer selects new component styles by querying the database to find out what components are available with smaller delays. The component that comes closest to satisfying the required delays at each output is selected. Thus the optimizer tries to set each of the slacks at the output pins to zero.

Having failed to fix all critical paths with the previous three steps, the microarchitecture optimizer attempts to combine bit-sliceable components into a random logic component and query the database to apply logic optimization. In addition, the database can use a transistor sizing program to size the transistors in a fashion that will meet the time constraints. By using larger transistor sizes than those used in the bit-sliced approach (where transistor sizes are fixed), it may be possible to produce a faster component. If indeed the database returns a faster component, the microarchitecture optimizer will switch the layout style to a custom layout. Of course, the random logic approach combined with the large transistor sizes results in larger area.

5.4. Area Optimization

Finally, Phase 4 performs area optimizations along non-critical paths. It mainly employs new component selection and component merging. Components that have outputs with positive slacks are examined for possible area/time tradeoffs. Area optimization operates as follows:

Procedure Area_Optimization (Non-Critical Path)
Begin
  New Component Style Selection
  Merge Multiple Comps for Optimization
End

New component selection includes choosing a bit-sliced layout style for components where doing so results in an area reduction. Some components, such as multiplexors, may need to be factored in order to use a layout module generator. For example, only 4-to-1 and 2-to-1 multiplexors may be available. An 8 to 1 multiplexor would then need to be factored. This can be achieved by the algorithm presented earlier.

In some cases, a layout module generator exists but contains more functions than are required. For example, consider an ALU. The bit-slice of the module generator may perform addition, subtraction, eight logical functions (eg., NAND, AND), and a set of comparison functions (eg., equal, greater than, zero). If only the addition operation, the comparison functions, and a logical AND are required, the layout module generator performs more functions than are necessary. Thus a

random logic implementation will probably produce the smallest area design. If there is already a random logic component connected to the ALU, the ALU can be merged into the random logic component and the logic reoptimized.

An alternative approach to generating the random logic design is to separate the groups of functions that need to be performed. For example. Figure 18 shows that three groups of functionality can be generated for our example of the ALU: an arithmetic unit (adder), a comparator, and a logical AND. A multiplexor is used to choose the addition function or the logical AND. In this case all components can be

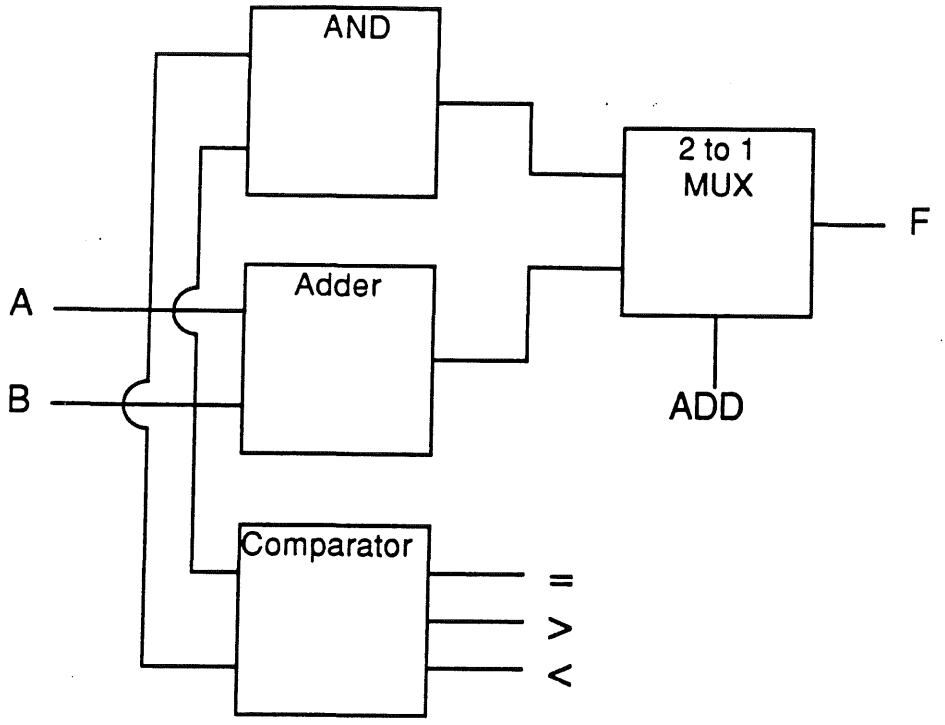


Figure 18. Option for performing ALU functions

implemented using the layout module generators and placed in the bit-sliced data-path during layout.

6. Experimental Results

This section presents experiments performed using MILO. A number of design examples were written in VHDL, then run through VSS to generate the initial microarchitecture design. The designs were then run through MILO with the following four strategies:

(1) Optimize the design for area and produce an underlying gate-level design for each microarchitecture component.

(2) Optimize the design for time and produce an underlying gate-level design for each microarchitecture component.

(3) Optimize the design for area and use the module generators for all bit-sliceable microarchitecture components, gate-level designs for all other components.

(4) Optimize the design for time and use the module generators for all bit-sliceable microarchitecture components, gate-level designs for all other components.

To get an idea of how good these optimizations were compared to a traditional straight logic optimization, the output design from VSS consisting of microarchitecture components was completely expanded into a flat gate-level design. This design was run through MISII and then through a transistor sizing program. The logic

optimization was also performed for both area and time. Thus each example was run six different ways.

Five different benchmarks were run through MILO: Rockwell Counter, Armstrong Counter, and three different versions of DRACO: Draco2, Draco3, and Draco Schematic. A short description of each of these designs and their results are shown in the following sections. In the final section a comparison of all of the optimizations performed by MILO and MISII is made and conclusions are drawn.

6.1. Benchmark Experiments

6.1.1. Rockwell Counter

The Rockwell Counter benchmark was supplied by Rockwell International and is a design used in telephone switching networks. It has four inputs as shown in the block diagram of Figure 19: 1) CLK, the system clock, 2) RST, which performs a synchronous reset of the counter, 3) DTI, a 12-bit data input, and 4) LDE, a control line which loads the counter with input DTI. It has only one output, DTO, which represents the value of the count.

The counter is a divide by 3328 counter that operates as follows:

(1) The counter has a start count of 0 and a terminal count of 3327.

(2) The counter increases by 208 on each clock edge. If the count is greater than 3327, the counter will start at the previous start count plus 26 (eg., the first

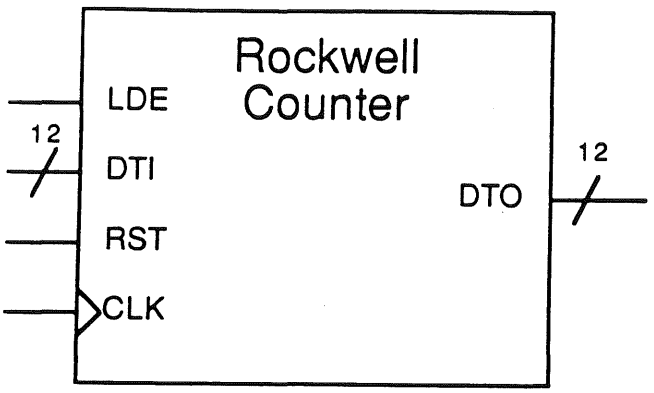


Figure 19. Block Diagram of the Rockwell Counter

time: 0 + 26). If the previous start count plus 26 is greater than 207, then the count will start at the previous start count plus 1.

(3) There are 26 sequences (ie., 26 start counts) before the counter reaches 3327 and wraps back to 0.

(4) The counter has an active high load enable which synchronously loads the counter. The state machine must adjust to the new state so as to keep the same counting sequence.

Table 1 shows the optimization results for the Rockwell Counter when optimizing for time, while Table 2 shows the results when optimizing for area. In this example, the design with the fewest transistors is achieved using the module generators

Optimization Tool	Optimization Style	time (ns)	area (# of transistors)
MILO	Gates	206.0	1800
MILO	Module Generators	233.5	1484
MISII	Gates	222.5	1344

Table 1. Time Optimization Results for the Rockwell Counter

Optimization Tool	Optimization Style	time (ns)	area (# of transistors)
MILO	Gates	327.0	1158
MILO	Module Generators	337.5	1056
MISII	Gates	413.0	1170

Table 2. Area Optimization Results for the Rockwell Counter

during microarchitecture optimization. The area of the designs employing only gates are roughly equal. When comparing time results, MILO's optimization with gates produced the smallest delay, followed by MILO's optimization using the

module generators. The optimization by MISII produced the largest delay.

Table 3 displays the tradeoff of time for area when comparing the time optimized designs with the area optimized designs. The change in time and area is shown as a percentage. For example, MILO's optimization using only gates achieves a 37% improvement in time at a cost of a 55% increase in area when comparing the time optimized design with the area optimized design. This table illustrates that fairly substantial reductions in time can be achieved at a cost of additional area. Finally, Figure 20 compares the three optimization approaches (MILO with gates, MILO with module generators and gates, and MISII) graphically. The curve represents the potential to achieve area/time tradeoffs between the best area optimized design and the best time optimized design, although this ability has not actually been

Optimization Tool	Optimization Style	time difference (%)	area difference (%)
MILO	Gates	-37.0	+55.4
MILO	Module Generators	-30.8	+40.5
MISII	Gates	-46.4	+14.8

Table 3. Time/Area Tradeoffs for Rockwell Counter

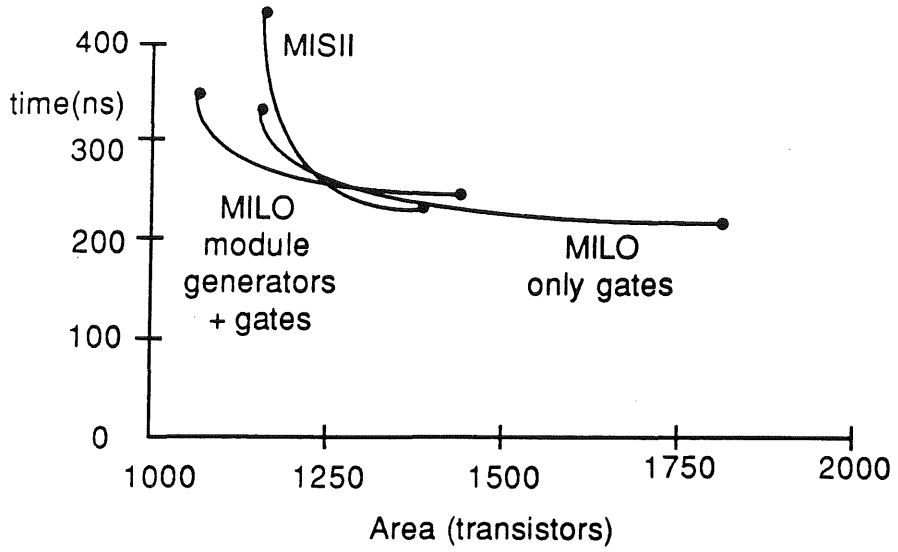


Figure 20. Three Optimization Approaches for the Rockwell Counter tested.

6.1.2. Armstrong Counter

The Armstrong Counter is a benchmark adapted from [Arms89]. As shown in the block diagram of Figure 21, it has four inputs: 1) CLK, the system clock, 2) CON, a two-bit input that selects which function the counter will perform, 3) DATA, a four-bit input that determines the end count for the counter, and 4) STRB, an asynchronous line that loads the two values DATA and CON into registers. It has a single four-bit output, CON_OUT.

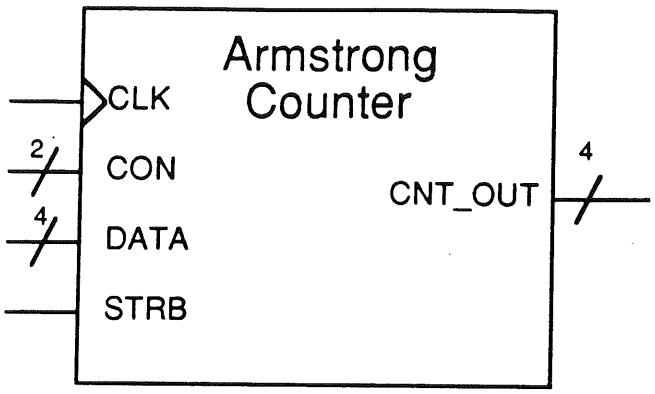


Figure 21. Block Diagram of Armstrong Counter

The behavior of the Armstrong Counter is as follows:

(1) On the rising edge of STRB, the values of DATA and CON will be loaded.

(2) The counter can perform four functions as specified by the value of CON: clear the counter, load a limit register, count up to a limit, or count down to a limit.

Table 4 shows the optimization results for the Armstrong Counter when optimizing for time, while Table 5 shows the results when optimizing for area. In this example, MILO's optimization with module generators produced the smallest delay, followed by MILO's optimization using the gates. The optimization by MISII produced the largest delay. When comparing area results, the design with the fewest transistors is again achieved using the module generators during microarchitecture

Optimization Tool	Optimization Style	time (ns)	area (# of transistors)
MILO	Gates	28.0	486
MILO	Module Generators	20.0	393
MISII	Gates	43.5	484

Table 4. Time Optimization Results for the Armstrong Counter

Optimization Tool	Optimization Style	time (ns)	area (# of transistors)
MILO	Gates	38.0	486
MILO	Module Generators	20.0	395
MISII	Gates	74.5	460

Table 5. Area Optimization Results for the Armstrong Counter

optimization. The area of the MISII design is smaller that that produced by MILO using gates.

Table 6 displays the tradeoff of time for area when comparing the time optimized designs with the area optimized designs. The change in time and area is shown as a percentage. Optimization by MILO using only gates shows a 26% reduction in delay with no increase in transistor count. This indicates that the improvement in time was mainly due to changes in transistor sizing. Finally, Figure 22 compares the three optimization approaches (MILO with gates, MILO with module generators and gates, and MISII) graphically. Again, the curve represents the potential to achieve area/time tradeoffs between the best area optimized design and the best time optimized design. For the Armstrong Counter, MISII has the largest distance between the area and time optimized designs.

Optimization Tool	Optimization Style	time difference (%)	area difference (%)
MILO	Gates	-26.3	+0.0
MILO	Module Generators	-0.0	-0.5
MISII	Gates	-41.6	+5.2

Table 6. Area/Time Tradeoffs for the Armstrong Counter

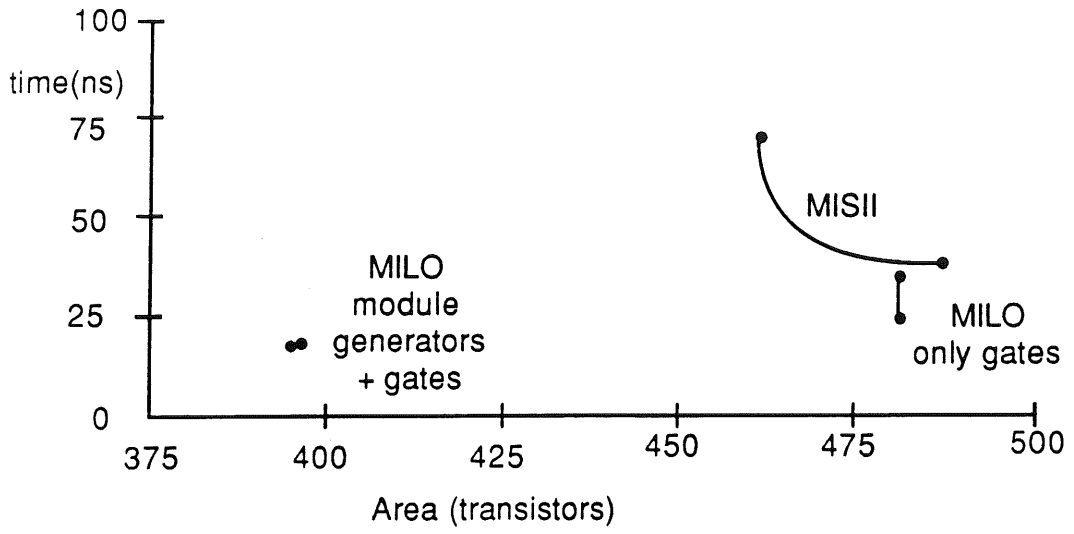


Figure 22. Three Optimization Approaches for Armstrong Counter

6.1.3. DRACO

DRACO is another benchmark obtained from Rockwell International and is the most complex of all our benchmarks. A block diagram of DRACO is shown in Figure 23, consisting of nine inputs and one output. DRACO is primarily intended to interface 16 I/O ports to a microprocessor's 8-bit multiplexed address/data bus and control signals. DRACO was developed by Rockwell as an ASIC chip.

Three VHDL descriptions of the DRACO chip were written. Each description represented DRACO at a different level of abstraction. "Draco Schematic" was derived from the logic schematic provided by Rockwell International. "Draco2" and "Draco3" were more abstract versions and each used a different style of modeling in

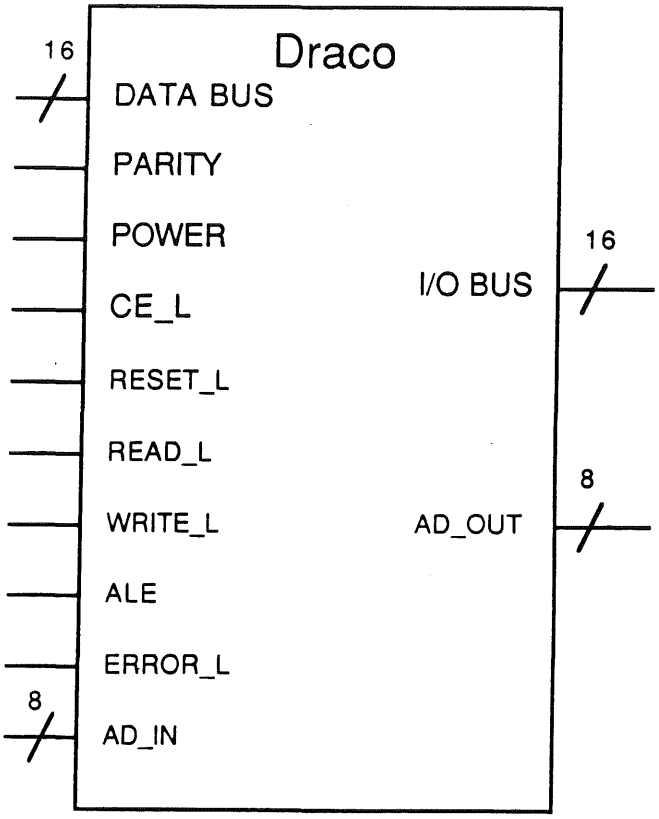


Figure 23. Block Diagram of DRACO

VHDL. Thus the designs produced by VSS from each of these descriptions are quite different.

Table 7 through Table 12 demonstrate optimization results for time and area on the DRACO examples. For examples "Draco2" and "Draco3", MILO's optimizations proved to be the best in terms of delay. Optimization by MILO using module generators resulted in the best designs in terms of area. Table 13 through Table 15 show the tradeoff of time for area when comparing the time optimized and area

Optimization Tool	Optimization Style	time (ns)	area (# of transistors)
MILO	Gates	194.5	5868
MILO	Module Generators	109.0	3390
MISII	Gates	283.5	5800

Table 7. Time Optimization Results for Draco2

Optimization Tool	Optimization Style	time (ns)	area (# of transistors)
MILO	Gates	226.5	5152
MILO	Module Generators	117.5	3390
MISII	Gates	342.0	4668

Table 8. Area Optimization Results for Draco2

Optimization Tool	Optimization Style	time (ns)	area (# of transistors)
MILO	Gates	101.5	5544
MILO	Module Generators	135.0	3968
MISII	Gates	138.5	4202

Table 9. Time Optimization Results for Draco3

Optimization Tool	Optimization Style	time (ns)	area (# of transistors)
MILO	Gates	115.5	5298
MILO	Module Generators	176.5	3492
MISII	Gates	174.5	4206

Table 10. Area Optimization Results for Draco3

Optimization Tool	Optimization Style	time (ns)	area (# of transistors)
MILO	Gates	205.0	5658
MILO	Module Generators	135.5	3026
MISII	Gates	149.0	4216

Table 11. Time Optimization Results for Draco Schematic

Optimization Tool	Optimization Style	time (ns)	area (# of transistors)
MILO	Gates	206.0	4486
MILO	Module Generators	136.5	3018
MISII	Gates	258.0	3762

Table 12. Area Optimization Results for Draco Schematic

Optimization Tool	Optimization Style	time difference (%)	area difference (%)
MILO	Gates	-14.2	+13.9
MILO	Module Generators	-7.2	+7.5
MISII	Gates	-17.1	+24.1

Table 13. Time/Area Tradeoffs for Draco2

Optimization Tool	Optimization Style	time difference (%)	area difference (%)
MILO	Gates	-12.1	+4.6
MILO	Module Generators	-23.5	+13.6
MISII	Gates	-20.6	-0.1

Table 14. Time/Area Tradeoffs for Draco3

Optimization Tool	Optimization Style	time difference (%)	area difference (%)
MILO	Gates	-0.5	+26.1
MILO	Module Generators	-0.7	+0.3
MISII	Gates	-42.2	+12.1

Table 15. Time/Area Tradeoffs for Draco Schematic

optimized designs. Figure 24 through Figure 26 compare the three optimization approaches.

In addition to comparisons of transistor counts, two layouts were generated by SLAM for Draco2 designs as an additional comparison. Figure 27 shows the layout for Draco2 that was produced from the design optimized by MILO using the module generators. The layout consists of two sections: the left hand portion is a custom layout consisting of random logic. The right hand portion of the layout is the bit-sliced datapath produced by the module generators. Figure 28 shows the layout for Draco2 that was produced from the design optimized by MISII. It consists entirely of a custom layout for random logic. As would be expected, the design with the module generators is smaller than the random logic design. The total layout area of the random logic design is 14,668,600 square micrometers compared with only

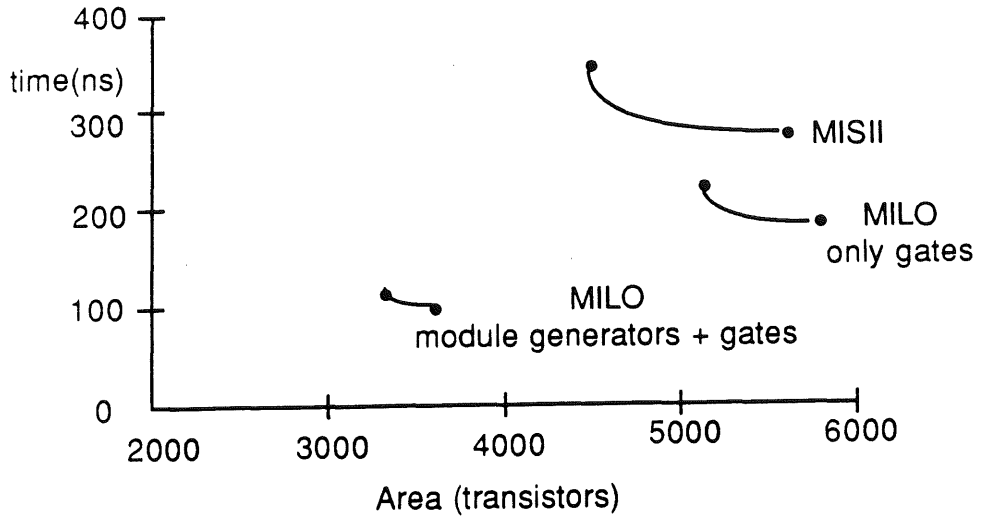


Figure 24. Three Optimization Approaches for Draco2

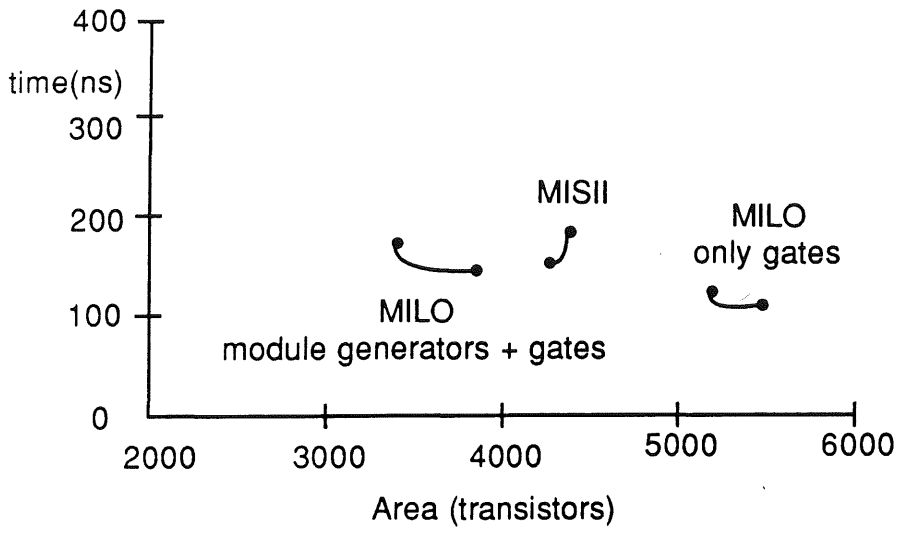


Figure 25. Three Optimization Approaches for Draco3

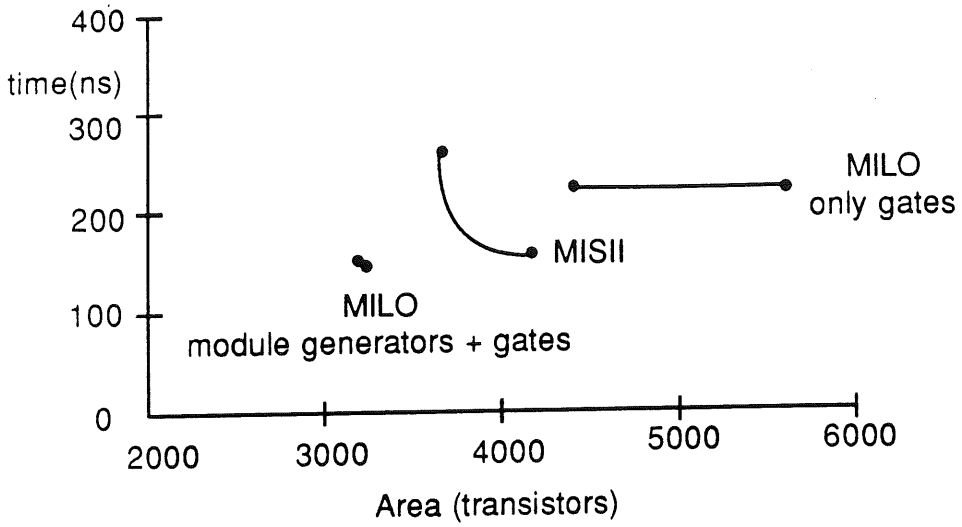


Figure 26. Three Optimization Approaches for Draco Schematic

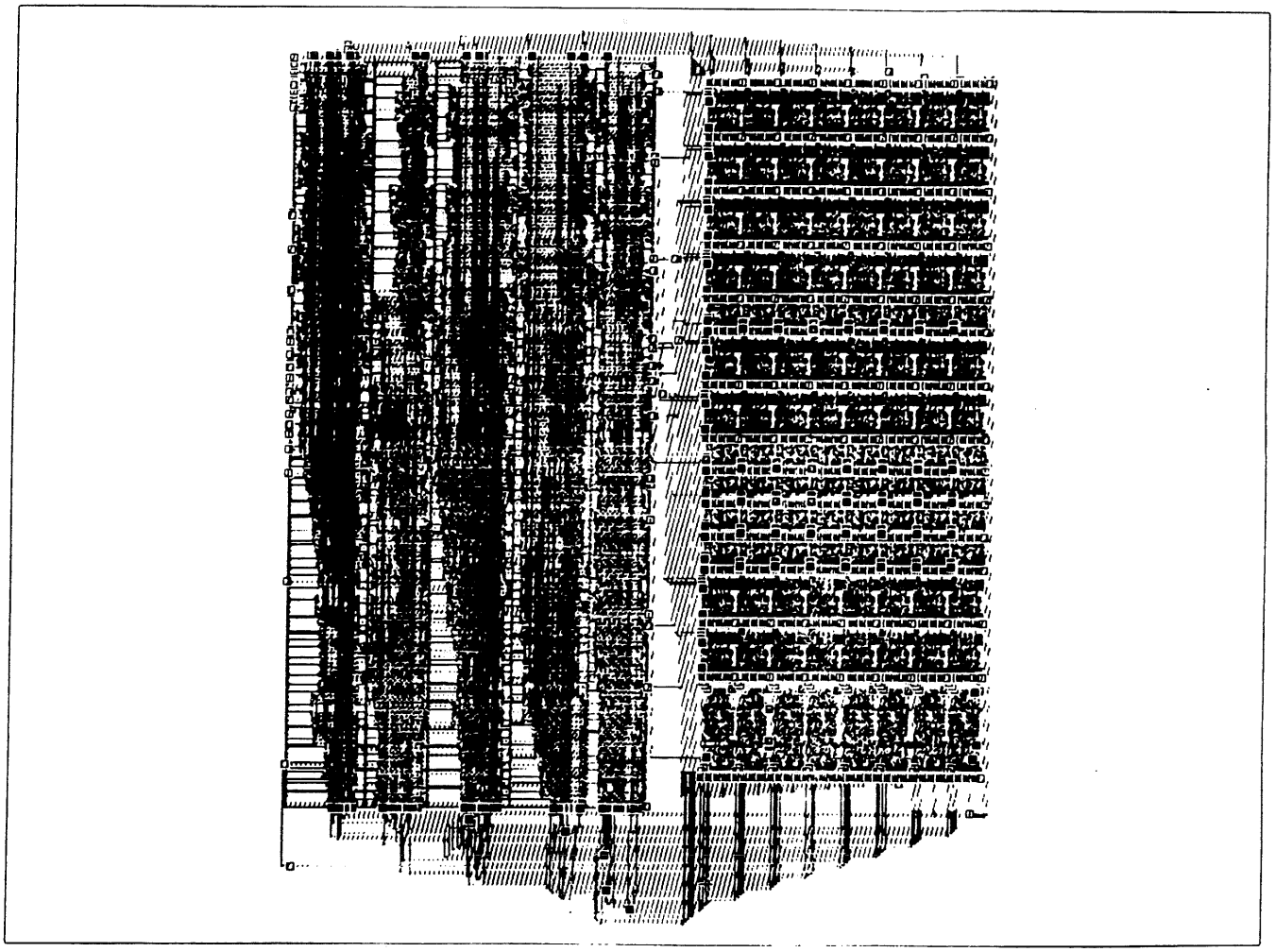


Figure 27. Layout of Module Generator Design for Draco2

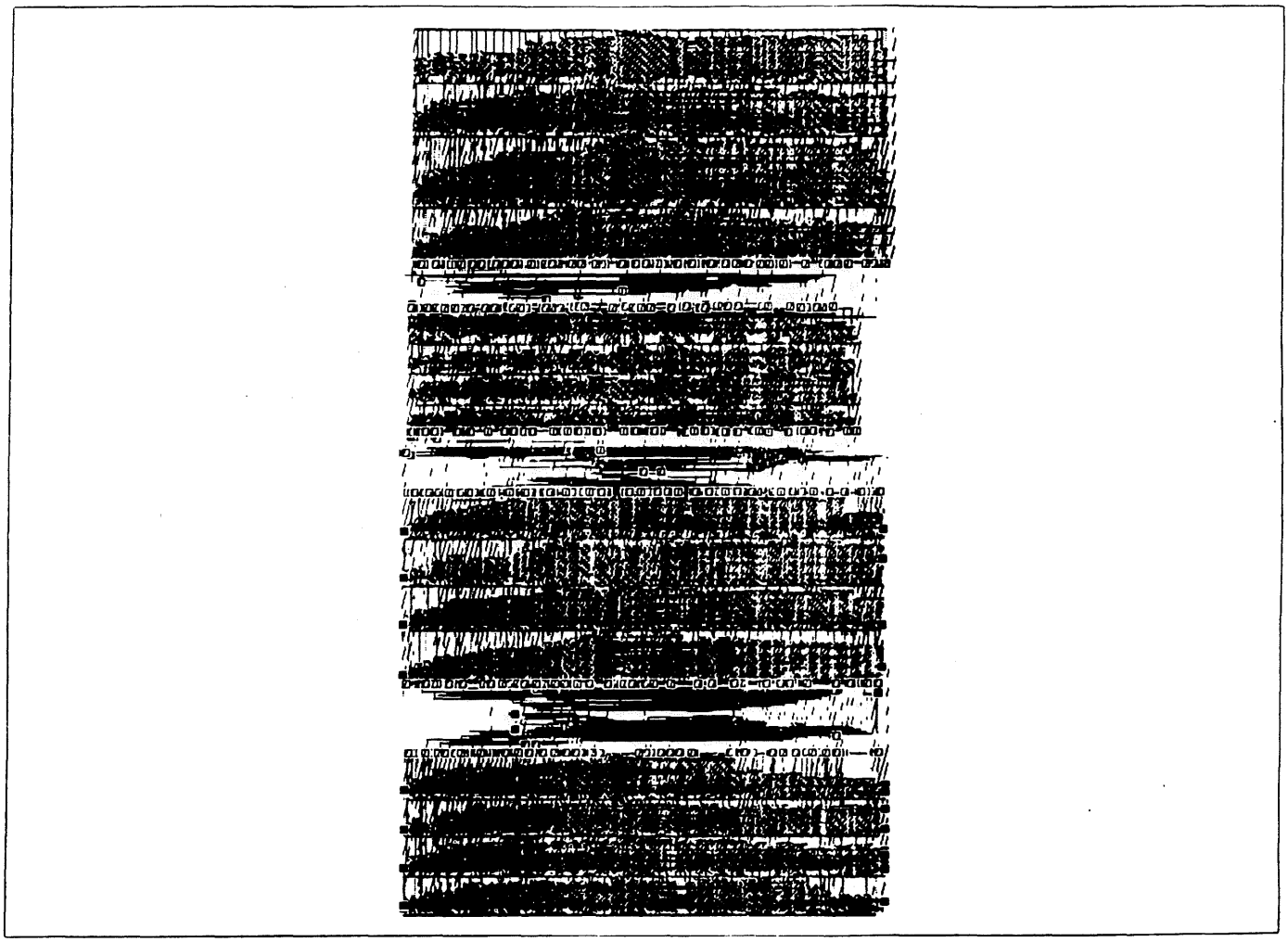


Figure 28. Layout of MISII Design for Draco2

8,592,672 square micrometers in the MILO module generator design. This represents an area difference of 70%.

6.2. Analysis

Table 16 compares optimization by MILO using only gates and straight logic optimization by MISII. MILO when using only gates produces faster designs in four of the five cases, ranging from 7% faster to 35% faster. In one of the five cases MILO is slower by 37%. This demonstrates that MILO can produce faster designs

Benchmark	MILO (%)	MISII (%)
Draco2	69	100
Draco3	73	100
Draco Schematic	138	100
Armstrong Cntr.	64	100
Rockwell Cntr.	93	100

Table 16. Comparison of MILO and MISII Timing Optimization

on average.

Table 17 compares optimization by MILO using only gates and straight logic optimization by MISII of the examples for area. In four of the five cases, MISII produces a design with a smaller area. This is to be expected as the MILO logic optimizer is primarily geared for time optimization. However, MILO's optimization with module generators compensates by providing area efficient bit-sliced layouts. The best designs in terms of area were usually achieved when using module generators as shown in the tables that follow.

Benchmark	MILO (%)	MISII (%)
Draco2	111	100
Draco3	132	100
Draco Schematic	119	100
Armstrong Cntr.	106	100
Rockwell Cntr.	99	100

Table 17. Comparison of MILO and MISII Area Optimization

Table 18 compares optimization using modules generators with optimization using only gates and optimizing for time. Table 19 shows the same comparison for area. The table shows that in most of the cases, optimization with the module generators produced a design with the smallest area and fastest speed. Table 20 and Table 21 make the same comparison with module generators but use the MISII results as the base for comparison.

These experiments demonstrate the effectiveness of MILO in generating efficient designs for either time or area. By optimizing the microarchitecture design

Benchmark	MILO (module gen.) (%)	MILO (gates) (%)
Draco2	56	100
Draco3	133	100
Draco Schematic	85	100
Armstrong Cntr.	71	100
Rockwell Cntr.	113	100

Table 18. MILO gate vs. MILO module generator designs (Time)

Benchmark	MILO (module gen.) (%)	MILO (gates) (%)
Draco2	66	100
Draco3	66	100
Draco Schematic	67	100
Armstrong Cntr.	81	100
Rockwell Cntr.	91	100

Table 19. MILO gate vs. MILO module generator designs (Area)

Benchmark	MILO (module gen.) (%)	MISII (gates) (%)
Draco2	38	100
Draco3	97	100
Draco Schematic	91	100
Armstrong Cntr.	46	100
Rockwell Cntr.	105	100

Table 20. MISII vs. MILO module generator designs (Time)

Benchmark	MILO (module gen.) (%)	MISII(gates) (%)
Draco2	73	100
Draco3	83	100
Draco Schematic	80	100
Armstrong Cntr.	86	100
Rockwell Cntr.	90	100

Table 21. MISII vs. MILO module generator designs (Area)

instead of simply expanding the design and performing logic optimization, superior designs can be produced. Further, the results demonstrate flexibility in generating designs with different layout styles -- those using only gates and those incorporating a bit-slice capacity.

7. Conclusion

In this report, we presented a tool for optimization of register-transfer level designs. The tool operates on top of a set of logic synthesis tools that provide area and delay information for individual microarchitecture components. This informa-

tion is used to modify the microarchitecture design by employing techniques such as changing a component's architectural or layout style and grouping selected components for optimization as a random-logic component. Further, a new methodology was presented for microarchitecture-level optimization that greatly reduces the amount of technology-specific knowledge necessary to perform the optimizations. Microarchitecture components are generated by a database based on a set of parameters from the microarchitecture optimization tool. Thus the microarchitecture optimizer does not need to deal with multiple logic optimization tools, layout module generators, transistor sizing tools, etc. Finally, a set of experiments were presented indicating that microarchitecture optimization techniques can produce faster designs or designs with smaller area than those obtained by logic optimization alone.

BIBLIOGRAPHY

[BiBr88] Birmingham, W.P, Brennan, A., Gupta, A.P., and Siewiorek, D.P., "MICON: A Single-Board Computer Synthesis Tool", *IEEE Circuits and Devices Magazine*, January, 1988.

[Br86] Brayton, R., et al., "Multiple-Level Logic Optimization System", *ICCAD*, 1986.

[BrRu87] Brayton, R., Rudell, R., Sangiovanni-Vincentelli, and Wang, A., "MIS: A Mutiple-Level Logic Optimization System", *IEEE Transactions on Computer-Aided Design*, Vol. CAD-6, No. 6, Nov. 1987.

[CaRo85] Camposano, R., and Rosenstiel, W., "A Design Environment for the Synthesis of Integrated Circuits", *11th EUROMICRO Symposium on Microprocessing and Microprogramming*, Sept. 1985.

[ChGa90] Chen, G.D., and Gajski, D.D., "An Intelligent Component Database For Behavioral Synthesis", *27th DAC*, 1990.

[Dutt88] Dutt, N.D., "GENUS: A Generic Component Library for High Level Synthesis", *Technical Report* 88-22, University of California, Irvine, Sept. 1988.

[GuPa90] Gupta, G., Pastorello, D., and House, G., "Timing Optimizations in a High-Level Synthesis System", *ICCD*, 1990.

[LiGa89] Lis, J.S., and Gajski, D.D., "VHDL Synthesis Using Structured Modeling", *26th DAC*, 1989.

[OrGa86] Orailoglu, A., and Gajski, D., "Flow Graph Representation", *23rd DAC*, June, 1986.

[PaGa87] Pangrle, B.M., and Gajski, D.D., "Design Tools for Intelligent Silicon Compilation", *IEEE Transactions on Computer-Aided Design*, Vol. 6, No. 6, November 1987.

[PaKn87] Paulin, P.G., and Knight, J.P., "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's", *IEEE Transactions on Computer-Aided Design*, Vol. 8, No. 6, June 1987.

[PaPM86] Parker, A.C., Pizarro, J., and Milnar, M., "MAHA: A Program for Datapath Synthesis", *23rd DAC*, 1986.

[SiSV90] Singh, K.J., and Sangiovanni-Vincentelli, A., "A Heuristic Algorithm for the Fanout Problem", *27th DAC*, 1990.

[StMu86] Stroud, C.E, Munoz, R.R., and Pierce, D.A., "CONES: A System for Automated Synthesis of VLSI and Programmable Logic From Behavioral Models", *ICCAD*, 1986.

[TrDi89] Trick, M.T., and Director, S.W., "LASSIE: Structure to Layout for Behavioral Synthesis Tools", 26*th DAC*, 1989.

[TsSi86] Tseng, C.J., and Siewiorek, D.P., "Automated Synthesis of Data Paths in Digital Systems", *IEEE Transactions on Computer-Aided Design*, Vol. 5, No. 3, July 1986.

[TsWe88] Tseng, C.J., Wei, R.S., Rothweiler, S.G., Tong, M.M., and Bose, A.K., "Bridge: A Versatile Behavioral Synthesis System", 25*th DAC*, 1988.

[VaGa88] Vander Zanden, N.B., and Gajski, D.D., "MILO: A Microarchitecture and Logic Optimizer", 25*th DAC*, 1988.

[WeRo88] Wei, R.S, Rothweiler, R., and Jou, J.Y., "BECOME: Behavior Level Circuit Synthesis Based On Structure Mapping", 25*th DAC*, 1988.

[WuGa90] Wu, C.H., and Gajski, D.D., "Silicon Compilation from Register-Transfer Schematics", *International Symposium on Circuits and Systems*, 1990.

[WuVG90] Wu, C.H., Vander Zanden, N., and Gajski, D.D., "A New Algorithm for Transistor Sizing in CMOS Circuits", *European Design Automation Conference*, 1990.