A Design Methodology for Interactive Behavioral Synthesis

Daniel D. Gajski
Hsiao-Ping Juan

Technical Report #95-25
July 26, 1995

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717-3425
(714) 824-7063

gajski@ics.uci.edu
hjuan@ics.uci.edu

Abstract

Due to the recent increases in design complexity, behavioral synthesis has become an important area of research and company interest. However, there has been market resistance to accepting the automatic behavioral synthesis approach as a practical solution in general because, first, it often produces results inferior to manual designs, and second, it allows only minimum user control. To develop a feasible approach for behavioral synthesis to overcome the hurdles faced by the automatic approach, we propose interactive behavioral synthesis, which attempts to maximally utilize the human designer's insights. Using interactive behavioral synthesis, the users can control the design process, observe the effects of design decisions, and manually override synthesis algorithms at will. In this report, we present a design methodology as how the user interacts with an interactive behavioral synthesis system, which in contrast to an automatic synthesis system, enables the human designer fine-grain control over each synthesis task and continually supplies feedback in the form of quality measures so that the user can make informed design-related decisions. To demonstrate the proposed design methodology, we also present in this report a walk-through square-root approximation (SRA) example.

Contents

List of Figures

1 Introduction

Recent advances in the VLSI technology have allowed companies to build complex designs containing over one million transistors on a single chip. As the complexity of the chips increases, so will the need for designing from the behavioral abstraction level where functionality and tradeoffs are easier to understand.

Behavioral synthesis is a process of synthesizing a design from a given behavioral description to a register-transfer-level (RTL) structure. Behavioral descriptions can be programs, algorithms, flowcharts, dataflow graphs, instruction sets or generalized finite-state machines, in which each state can perform arbitrarily complex computations. The RTL structure is a set of interconnected components that is described by a netlist. Components in the netlist can be (a) functional units such as ALUs, multipliers, (b) storage units such as memories, register files, and (c) interconnection units such as muxes and buses.

In general, the design process of behavioral synthesis can be decomposed into three major tasks. First, the number and type of resources (i.e., functional units, storage units and interconnection units), used in the design, must be defined. This task of defining necessary resources is called **allocation**. Once the resources are known, the behavioral description can be partitioned into states in such a way that all the variable assignments in each state can be computed by allocated resources. This task of partitioning the behavior into time intervals is called **scheduling**. Although the scheduling task assigns each operation to a particular state, it does not assign it to a particular component. In order to obtain the proper implementation, we have to assign each variable to a storage unit, each operation to a functional unit and each transfer from I/O ports to units and among units to an interconnect unit. This task is called **binding**. After binding, an RTL netlist is produced.

Many years of research have been dedicated in the development of automatic behavioral synthesis tools [2][4][7][11]. Several EDA vendors have also introduced recently commercial products based on behavioral synthesis. In these systems, designs are obtained with minimal user interaction. The only means of controlling the output from such systems is via constraints expressed in terms of area and/or performance. Figure 1 shows a typical design methodology of an automatic behavioral synthesis system. Note that the order of synthesis tasks being performed in an automatic synthesis system may vary.

However, automating behavioral synthesis is a very complicated issue. For instance,

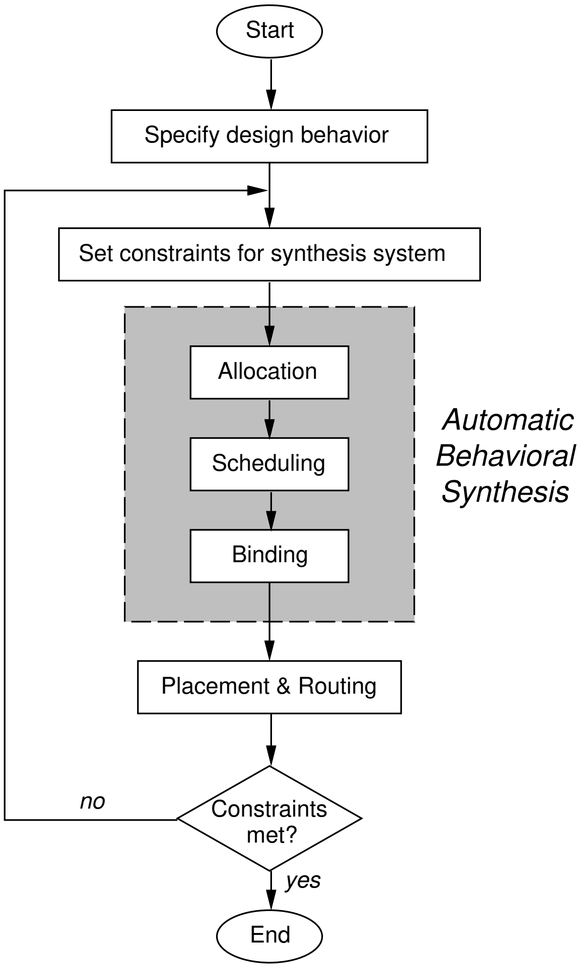


Figure 1: A typical design methodology of an automatic behavioral synthesis system

the synthesis tasks are all NP-complete problems. In addition, the order in which these synthesis tasks are performed also has an impact on both the efficiency and results of the overall synthesis process. Moreover, the behavioral synthesis tasks are always done before the physical level tasks, such as placement and routing, start. Yet, these low level tasks may have great effects, such as wiring delay, on the design and these effects are very difficult to estimate at behavioral level. Hence, the resultant designs sometimes cannot satisfy the performance or area demands of real-world constraints. Although it certainly cannot be denied that progress has been considerable, a practical solution of automating behavioral synthesis is still distant.

When the design produced by automatic behavioral synthesis is not a good one, it presents the user with the following dilemma. If the user modifies the input description and constraints and resynthesizes it, then he/she may get a completely different and unpredictable design, which still may not satisfy the constraints. Besides, low level tasks such as placement and routing, which usually require tremendous amount of time, need to done in every iteration, and consequently the time to reach an acceptable design is very long. On the other hand, if the user modifies the output design manually, then he/she needs to spend considerable effort, if at all possible, to understand the synthesized result and has to prove the correctness of the modified design.

To develop a feasible approach for behavioral synthesis, we have substituted the goal of a completely automated, "push-button" synthesis system with one which attempts to maximally utilize the human designer's insights. This approach, as opposed to automatic behavioral synthesis, is called **interactive behavioral synthesis**. Using interactive behavioral synthesis, the users can control the design process, observe the effects of design decisions, and manually override synthesis algorithms at will. This interactivity will allow the synthesis system to generate acceptable-quality high-complexity designs in the immediate future, instead of waiting for the many years of research needed to improve the current automatic synthesis techniques. With this goal in mind, we have implemented an interactive behavioral synthesis system called *Interactive Synthesis Environment (ISE)*. The main subject of this report is to propose a design methodology for using such an interactive synthesis system.

In the next section, we shall first give a brief overview of the proposed design methodology, in contrast to the methodology used by automatic synthesis. Afterwards, we shall briefly review the previous research in this area. Before we discuss the proposed design methodology, we introduce the synthesis system ISE in Section 4 in order to present the reader a clearer view as how the design methodology can be exercised. Then the proposed design methodology is discussed in detail in Section 5. Finally we present a walk-through example and give conclusions.

2 Overview of the Design Methodology

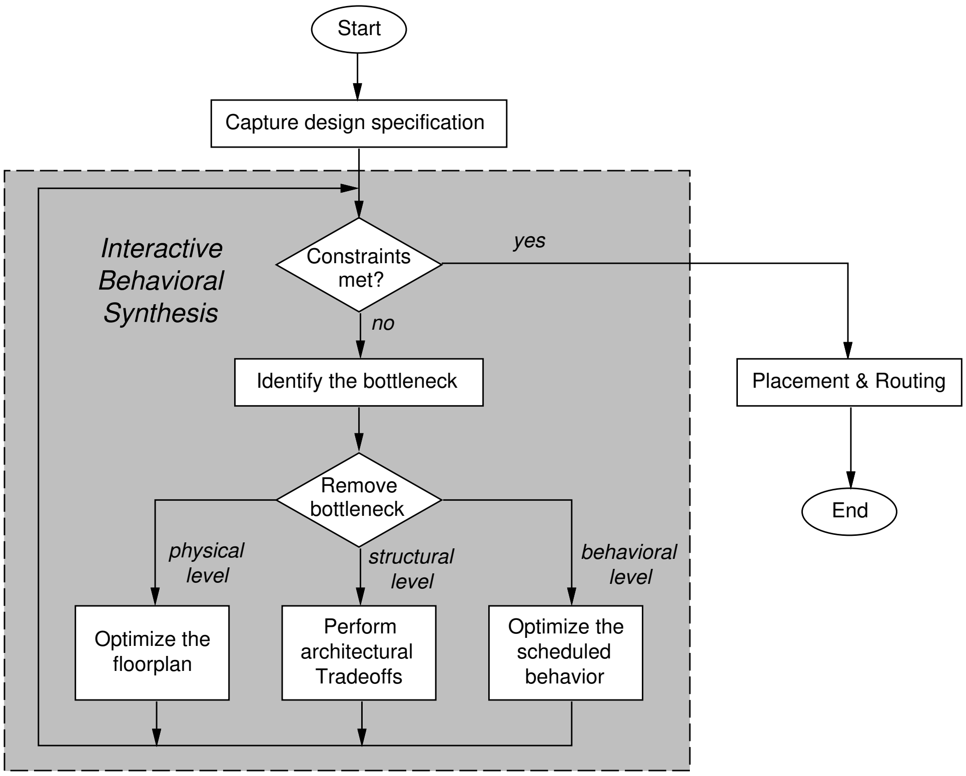


Figure 2: A design methodology for interactive behavioral synthesis

Figure 2 shows the proposed design methodology for interactive behavioral synthesis. In this methodology, the user first captures the design specification using graphical design views. Once a specification is captured, the system provides quality metrics to tell the user whether or not the design meets area and time constraints. The user can acquire design quality metrics from the system at any given step of the design process. If the information required for the computation of the acquired metric has not been established at that stage of the design process, the system will estimate the metric based on knowledge of the physical design. If there exist any constraint violations, the system can further help the user to identify the problem spots, i.e. the bottlenecks, in the design. To remove the design bottlenecks, that is, to improve the cost or performance of the current design, the user is allowed to work at either behavioral, or structural, or physical level by optimizing the scheduled behavior, or performing architectural tradeoffs, or optimizing the floorplan, respectively. Decisions made by the user generate immediate feedbacks as to the quality of the resulting design. If the updated quality metrics still indicate that there are unsatisfied constraints, the user can continue the tasks of identifying and removing design bottlenecks. This process is iterated until all the constraints are satisfied, and an RTL netlist and an implementation-specific floorplan are obtained as final results. Then this RTL netlist and the floorplan become input of the placement and routing tools, which generate the layout of the design.

There are two remarkable contrasts between the methodology for automatic behavioral synthesis, shown in Figure 1, and the proposed methodology for interactive behavioral synthesis. First of all, the proposed methodology *allows user decisions and user control in every task and at every level of the design process.* This provides the user complete control over the synthesis system. Moreover, unlike in automatic behavioral synthesis, there is no forced ordering of synthesis tasks; the user can perform any synthesis task at any time during the design process. Thus, the decision of what task to perform at a particular time during the design process depends only on how to effectively remove the bottlenecks in the design.

Secondly, the system *allows the user to start floorplanning early in the design process.* By doing so, the system can provide rapid feedbacks of useful physical design characteristics and quality metrics to every level of design abstraction. That is, the user can take the physical level floorplan into account while making design decisions at behavioral or structural level. As a result, the time-consuming tasks of placement and routing need to be done only once, when the design is completed and the floorplan is obtained. This greatly reduces the design time.

We would like to emphasize here that interactive behavioral synthesis does not mean that the user is required to do the design completely by hand. In addition to allowing the user to perform design tasks interactively, interactive behavioral synthesis should also provide automatic algorithms to complete design tasks that are simple yet tedious for the user to do unaided. At the end of each design step, the user can call automatic algorithms to either perform one design task for the user or to simply finish the design by completing the process of allocation, scheduling, binding and floorplanning. If the results of automatic algorithms are not satisfactory, the user can manually override the synthesis algorithms. Conceptually, one may think of interactive behavioral synthesis as providing quality metrics to the user during manual design, and providing automatic algorithms to rapidly complete simple yet tedious design tasks. As the design process progresses to more detailed and less abstract descriptions, the user's experience can be utilized to make the kinds of decisions no automatic tool can perceive or predict.

3 Previous Work

There are several previous papers addressed the importance of user-interaction with synthesis systems. In this section, we will differentiate our approach from previous research.

The ACE graphical interface [1] is intended to be the interface between the user and the synthesis system. It allows the user to place and connect functional nodes to create a graph that specifies the desired behavior, and thereby precludes the need for an initial textual input description. After the initial graphical specification is obtained and before synthesis tasks such as allocation, scheduling and binding start, some transformation techniques can be applied to the specification to transform it into a better, more efficient input description of the synthesis system. ACE allows the user to interact with the synthesis system by giving the user the final say in accepting or rejecting the system's transformation decisions. An experienced user can also specify transformations manually. Nevertheless, ACE does not allow the user to interact directly with the synthesis tasks.

RLEXT [9] [10] is an interactive tool which allows a user to manually reschedule a design's behavior or modify a design's structure by adding or deleting components and interconnects. The unique aspect of RLEXT is that, if the user makes changes in the datapath design or the behavior's schedule that would impair the datapath's ability to carry out the desired schedule, RLEXT will automatically repair the datapath so that it is once again able to execute the specified schedule. However, RLEXT does not provide the user feedbacks of the current design's quality to help the user making decisions of how to improve the design.

The system AMICAL [8] allows the user to mix automatic and manual design. The user may start a design manually and ask AMICAL to finish it. Alternatively, the user can execute the synthesis tasks step by step. At each step, the user has the choice to continue the synthesis automatically or manually. Yet, AMICAL has a fixed design flow, that is, the user has to perform a sequence of synthesis tasks in the order of scheduling, chaining, allocation and then architecture generation.

A unique aspect of our approach is that it allows the user to start floorplanning early in the design process. None of the previous research has ever attempted to address physical design issues with behavioral synthesis, that is, generating feedbacks from the physical level to help the user making design decisions at behavioral and structural levels. Hence, the proposed design methodology supports interactive behavioral synthesis to a degree not presently seen in this research area.

4 Interactive Synthesis Environment: ISE

design level	design view	quality metrics/hints	tasks
behavioral level	state−actions table view	operator occurrences variable lifetime state delay maximum execution time average execution time execution time utilization	add/delete assignments merge/split states
structural level	component selection and binding view	component delay component area component utilization clock utilizatoin binding hints	add/delete components change component implementations bind/unbind operators/variables to components
physical level	floorplan view	total area functional unit area storage unit area routing area wasted area wire length total wire length	change component placements alter the positions of module pins and I/O ports route/unroute interconnections

Figure 3: The design view, quality metrics, and tasks supported in ISE

We have implemented an interactive behavioral synthesis system, called ISE. To support user interactivity, ISE provides graphical design views, whereby the user enters and/or modifies the design and also perceive the consequences of design decisions. Decisions made by the user generate immediate feedbacks as to the quality of the resulting design. Furthermore, to allow the user easy control over the design tasks, such as allocation, scheduling and binding, ISE divides each tasks into small steps. For example, scheduling can be divided into splitting and merging states. Figure 3 summerizes the design views, quality metrics and tasks supported in ISE for design at behavioral, structural and physical levels. We will give a brief description of each of these views, quality metrics and tasks in the following. The detailed discussions can be found in [6].

4.1 Behavioral Level

4.1.1 Design View

At the behavioral level, ISE provides the **state-actions table view** to the user for capturing a design's behavior. The state-actions table format allows a design's behavior to be captured as a series of states, each state containing a set of operations to be performed in the state. Note that when a behavior is completely non-scheduled, it can be specified using one state.

PS	SCOND	NS	AC	ACTIONS
ST1	T	ST2	T	O1 = I1 + I2
			T	O2 = I1 + I3
ST2	T	ST1	T	O3 = I1 + I2

Figure 4: The state-actions table view

The state-actions table view displays the behavior and schedule of a design in a tabular format. Figure 4 shows an example of the state-actions table. The following is a brief description of each column in the table.

- *PS* is the present state.
- *SCOND* gives the condition for a next-state transition.

- *NS* is the next state.
- *AC* shows the assignment condition for each action.
- *ACTIONS* lists all operations in the behavior.

Using this view, the user can specify a new behavior, modify an existing behavior, or schedule a behavior. Before the user finalizes the design, the schedule represented in the state-actions table view is considered "partial" and reflects only the user's conceptualization of the flow of the behavior. That is, the user is allowed to interactively modify a schedule at any time in the design process.

4.1.2 Quality Metrics

Since the state-actions table view is used for behavioral capture and scheduling, several scheduling metrics are implemented to help the user decide how to partition a behavioral description into control steps.

Two important metrics of design cost are operator occurrences and variable lifetimes. **Operator occurrences** metric shows the number of operators of each type used in each state. The maximum number of occurrences of a certain operator type over all states determines the required minimum number of functional units to perform that type of operation. **Variable lifetimes** metric identifies states in which a variable holds a useful value. The maximum number of variables with overlapped lifetimes over all states determines the required minimum number of storage units.

Where performance is concerned, ISE provides the user metrics of clock period and execution time. Since the clock period of a synchronous design can be estimated by the maximum state delay over all states in the design, ISE provides the user the **state delay** metric, which gives the time needed to execute all operations in a state. In addition to the delay time, the metric can also show the register transfer path that causes the longest delay in the state, that is, **the critical path**. By shortening the critical path, the user can reduce the clock period.

If a performance constraint has been specified, the user must be able to evaluate the impact of any design decision to determine whether any time constraints are being violated. Depending on how the performance constraint is specified, the user can use either of two metrics: **the maximum execution time** and **the average execution time**. The maximum execution time metric shows the longest execution time required by the behavior from start to finish, considering all possible state branching, while the average execution time shows the average. The maximum execution time is computed by the product of the maximum state delay and the total number of states on **the longest execution path**, which is the sequence of states that causes the maximum execution time. ISE can also highlight the states reside on the longest execution path to help the user identify the bottleneck of reducing the maximum execution time.

Other than reducing the clock period and the number of states on execution paths, in order to improve the performance of a design, the user can also try to minimize the idle time of components. The **clock slack** associated with a state represents the portion of the clock cycle for which the components are idle, that is, the difference between the state delay and the clock period. Since the total execution time is equivalent to the sum of all state delays plus the sum of clock slacks of all states, we may postulate that a smaller sum of clock slacks would result in a shorter execution time. ISE provides a metric, **execution time utilization**, which indicates the percentage of maximum execution time during which at least one of the components are being used. This metric is defined as *(the sum of state delays of all states on the longest execution path/the maximum execution time)* $\times$ *100%*. Hence, the higher the execution time utilization is, the lower the sum of clock slacks is, the shorter the execution time is. When the execution time utilization is 100%, there is no clock slack and the total execution time is minimized.

4.1.3 Tasks

In the state-actions table view, ISE provides the user a minimum set of tasks for capturing and scheduling a design's behavior.

To capture or modify a behavior, the user could either re-write a behavior or re-order the existing assignments. Re-writing a behavior involves adding new assignments, deleting existing assignments or modifying existing assignments. Re-ordering assignments involves deleting an assignment and adding the same assignment in a different state. In ISE, the user could either **add assignments** to or **delete assignments** from the state-actions table. By performing either one or both of these two tasks, the user can re-write a behavior or re-order assignments in a behavior. For instance, if the user wants to modify an assignment, he/she can simply delete the assignment which needs to be modified, and then add the new assignment to the state-actions table.

Furthermore, the user can modify a design's schedule by either **merging states** or **splitting states**. State merging can be done by selecting two consecutive states and commanding the system to merge the selected states. As a result, the system will generate a new state in place of those two selected states. The new state will encompass the operators performed in the two original states and all the data dependencies will be maintained automatically. State splitting can be done by selecting a set of operators in a state and asking the system to split the state at where those operators are. That is, the system will insert a new state before the state where the selected operators are currently executed and move the selected operators and their predecessors to the new state. The user can also modify a schedule by adding/deleting assignments. For example, if the user wants to move one assignment from one state to another, he/she can first delete the assignment from the state where it is executed and then add the assignment to another state.

4.2 Structural Level

4.2.1 Design View

At the structural level, the user needs to be able to determine the type and number of resources used to implement the design and also assign operators or variables to functional or storage units respectively. In order to allow the user to perform these design tasks, it would require a view of behavior as well as available physical components. In ISE, these tasks can be done in **the component selection and binding view**.

The component selection and binding view consists of four displays: unit selection display, component capture display, allocation table display and state-actions table display. Figure 5 shows an example of different displays in the component selection and binding view.

The unit selection display and component capture display allow the user to select components from a component library and add instances of those components to the current design's component set, which is shown in the allocation table display. The unit selection display shows the available component categories and the parameters for each component. The user must select parameters values, such as bitwidth, style and functions performed, in order to specify a unique component type. These parameters can be derived from the

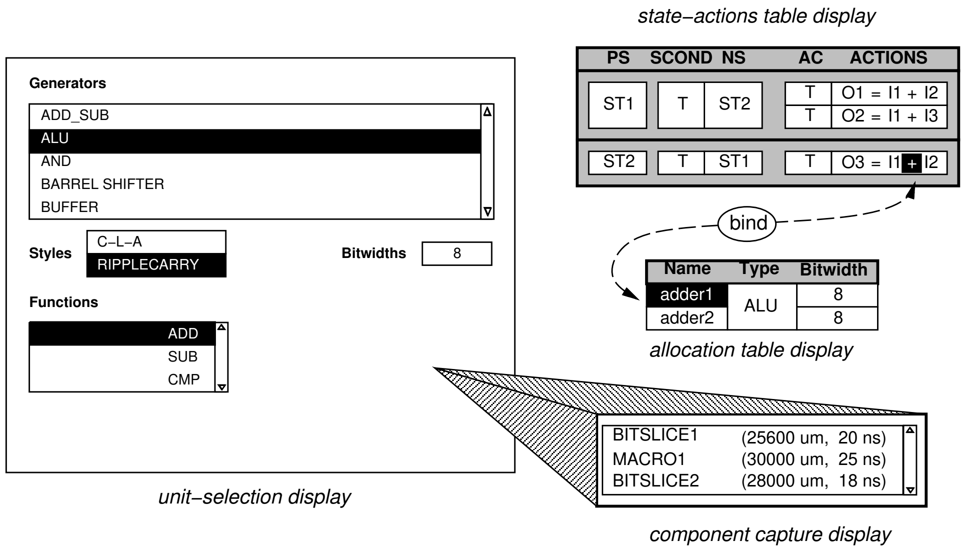


Figure 5: The component selection and binding view

behavior shown in the state-actions table display since the components selected must be able to perform the operations defined in the behavior. Moreover, the number of components of each type can also be derived from the behavior because there have to be enough components allocated to perform the scheduled behavior. Once a component type is given, the component capture display shows all the available implementations of the given component type along with the area and maximum pin-to-pin delay of each implementation. The user can select one implementation with satisfactory area and delay. After components are allocated, the user can assign operators or variables to components by first selecting one operator/variable from the state-actions table display and then one functional/storage unit among the allocated components shown in the allocation table display.

4.2.2 Quality Metrics

Two basic metrics are provided in the allocation table display to indicate to the user the cost and speed of each component: **component area** gives the area of a component in microns squared and **component delay** gives the maximum pin-to-pin delay of a component in picoseconds.

After the user assigned some operators to components, **component utilization** and **clock utilization** metric can be calculated. Component utilization is defined as *(the number of states in which the component is used/the total number of states)$\times$ 100%.* Clock utilization metric measures the average percentage of the clock period that is utilized by the component. The slack of a component in a particular state is defined as the time the component waits after it completes its computation and before the state changes. Assume the component is used in n states, its clock utilization can be computed by *(1-(the sum of slacks in all states in which the component is used)/($n\times$ clock period)) $\times$ 100%.*

4.2.3 Binding Hints

There are several criteria that must be considered when deciding what hardware component can be bound to a given operator. The most basic criterion is examining **function compatibility** to indicate whether or not the component can perform the function required by the operator. In some cases, the component may be able to perform only part of the computation and additional logic will be needed. In other cases, the component may perform more then the required operation, in which case part of the component's circuit is wasted in doing useless computation. When considering binding alternatives, the **performance** of components must be taken into account so that no state will violate clock-width constraints. Similarly, the **cost** of a component must also be considered so that the design does not exceed any cost constraint. The **bitwidths** of ports on the component must be compared to the bitwidths of variables or operators in the behavior. Mismatches may require additional techniques if a binding is to be performed. If component port bitwidths are greater than that required by the behavior, the unused bits must be initialized properly (usually by a connection to ground) during each computation. Component bitwidths smaller than that needed by the corresponding behavior may require extra components and additional logic to perform the operation. Moreover, after some bindings have been performed, examining hardware component **sources** and **sinks** can be done to gain information useful in avoiding additional bus drivers or interconnect units. When considering a binding for a specific operator/variable, if common sources or sinks can be found in operators/variables previously bound to a component, performing the binding will not require additional interconnect unit and thus extra cost can be avoided.

Since the binding tasks require the user to consider many criteria at the same time in order to determine which component should an operator or a variable be bound to, ISE provides **binding hints** to help the user to make the decision. Different from quality metrics, which give the user only a set of numbers indicating the quality of the current design, hints make suggestions to the user of the appropriate actions to take. For example, when the user selects an operator, binding hints highlight components using different color shades. These highlighted components are selected by the system as the most likely candidates to be bound to the selected operator. Binding the selected operator to a component with the brighter shade will give a better resulting cost than binding to the component with the darker shade, according to the binding hints algorithm. The binding hints are estimated based on six factors, as explained above: function compatibility, bitwidth compatibility, sources closeness, sinks closeness, performance and area. The user can specify weights for different factors to emphasize their degrees of importance.

4.2.4 Tasks

ISE supports interactive allocation by providing a minimum set of tasks that the user can perform: **adding components to the allocation table**, **deleting components from the allocation table**, and **changing component implementations**. By performing any of these tasks, the user can easily modify the allocation table. For instance, if the user would like to replace an already allocated component by a component of different type, he/she could simply delete the allocated component from the allocation table and add the new component to it. If the user would like to replace an allocated component by a component of the same type but different speed or cost, it could be done by changing the implementation of the allocated component. The task of changing component implementations is achieved by requesting the system to show all the implementations of a selected component in the component capture display and then selecting a desired implementation. The system will then automatically update the component characteristics of speed and cost in the allocation table.

As mentioned in the discussion of the component selection and binding view, the task of **binding operators/variables to components** interactively can be done by the user selecting an operator or a variable and also a component from the allocation table, and then requesting the system to perform the binding between the selected operator/variable and the selected component. The task of **unbinding** is also required so that the user can modify binding decisions once a particular binding results in unsatisfactory design.

4.3 Physical Level

4.3.1 Design View

At the physical level, ISE allows the user to perform floorplanning once some hardware components are chosen to implement the design. Therefore, a floorplan view showing the shapes of hardware components and the interconnections between them is the basic requirement. Figure 6 shows an example of the floorplan view.

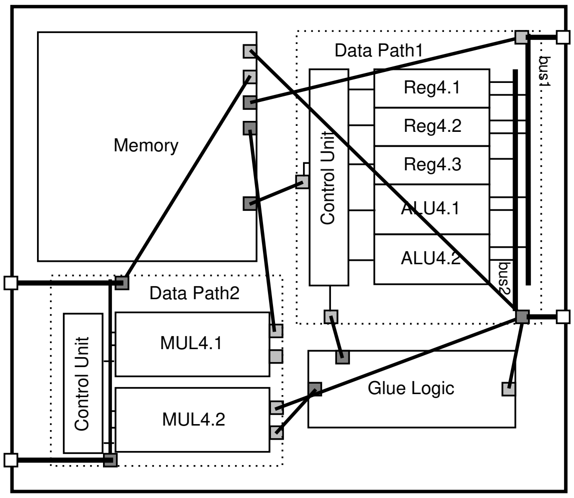


Figure 6: The floorplan view

4.3.2 Quality Metrics

Specific floorplan metrics to facilitate area optimization include the following. **Total area** metric gives the estimated chip area of the design. **Functional unit area**, **storage unit area** and **routing area** show the area in microns squared as well as the percentage of the entire chip area being occupied by functional units, storage units and routing respectively. **Wasted area** indicates the amount of "white space" in the floorplan, calculated by subtracting the sum of component areas plus routing area from the total area of the current design.

Other than specific area metrics, ISE also provides metrics of wire length since wire length is often an indication of interconnection delay. For instance, if the user selects one wire in the floorplan view, **wire length** metric can tell the user the length of the selected wire in microns. Moreover, **total wire length** metric can show the user the sum of the lengths of all wires in the floorplan. Furthermore, to help the user identify performance bottleneck, ISE can also highlight the **critical path**, including the components and interconnections, in the floorplan.

4.3.3 Tasks

The floorplan view in ISE allows the user to perform interactive placement and routing by doing one of the following tasks: **changing the placement of components**, **altering the positions of module pins and I/O ports**, and **routing interconnections**.

The task of changing component placement can be done by rotating and moving components. The task of altering the positions of module pins and I/O ports can be done by simply moving selected module pins or I/O ports. A module is a grouping of several hardware components. In ISE, the user can perform floorplanning in a hierarchical fashion, that is, he/she can group several components into a module, and then perform floorplanning on a set of modules. This hierarchical approach greatly reduces the complexity of floorplanning. To route an interconnection, the user can select a not yet routed connection, which is shown as a point-to-point connection in the floorplan, and then position and size the wire segments according to the route desired. The user may also **unroute** a previously routed set of wire segments resulting in a point-to-point connection being displayed.

5 Design Methodology

In this section, we shall discuss in detail each step in the proposed design methodology shown in Figure 2. In the discussion, we shall also demonstrate how the quality metrics and tasks explained in previous section are utilized in the design process. Throughout the discussion, there are small examples for explanation purpose. In all the examples, when it is not otherwise specified, the delay of a multiplier is assumed to be 40 ns, the delay of an adder is assumed to be 20 ns and the setup time of a register is assumed to be 5 ns.

5.1 Capture Design Specification

The first step in the proposed design methodology is to capture the design specification. Unlike in automatic behavioral synthesis, where the user could only write behavioral descriptions using the hardware description language required by the synthesis system, the user here is allowed to capture a specification at mixed levels. That is, the user can specify the behavior and schedule of the design using a state-actions table, as well as a set of components to be used in the implementation and the floorplan of the components. Note that this is the starting point of the design process; therefore, the specifications at each of these behavioral, structural and physical levels are most likely to be incomplete. However, after the initial specification is obtained, the user can modify the specification at any level in an interactive and iterative fashion until a register-transfer level implementation along with a floorplan, which satisfy all design constraints, are completed.

5.2 Identify Design Bottlenecks

At any stage in the design process, the user needs to identify the problem spots, i.e. the bottlenecks, in the current design and remove these bottlenecks, that is, to improve either the cost or the performance of the current design.

ISE can identify design bottlenecks for the user by highlighting where the bottlenecks are in all levels of design. For instance, a design's clock period constraint is violated. In this case, not only the operators which cause the longest state delay would be highlighted in the state-actions table, the components to which the operators are bound and the interconnections between the components would also be highlighted in the floorplan. This helps the user to identify the critical path in the behavior, the structure, as well as in the floorplan of the current design. The importance of this is, in general, bottlenecks can be removed by modifying the design at either the behavioral, the structural or the physical level. For example, after the critical path is identified, it can be shorten by either rescheduling the behavior, or assigning the operators on the critical path to faster components, or by shortening the wire lengths between the components on the critical path.

In the subsequent sections, we shall discuss sets of techniques as how to improve a design's quality by optimizing its scheduled behavior, performing architectural tradeoffs on the design's datapath structure, or optimizing its floorplan. Each of the techniques may produce different degrees of effects and also side-effects.

5.3 Optimize the Scheduled Behavior

At the behavioral level, the user can modify or reschedule the state-actions table to minimize required hardware resources and shorten execution time. Note that each behavioral construct has its corresponding hardware implementation. For example, behavioral operators will be implemented with functional units and behavioral variables will be implemented with registers or memories. Thus, reducing the number of operators and variables implies that the required hardware will likely be reduced. Similarly, reducing the number of operators on the critical path shortens state delay and the clock period in turn. Figure 7 shows a set of such techniques that can be applied at the behavioral level to improve either area or execution time of the design.

5.3.1 Reduce Area

A design's area is contributed by three factors: functional unit area, storage unit area and interconnection unit area. At the behavioral level, it is very difficult to obtain indications of interconnection unit area. However, as mentioned before, the functional unit area and storage unit area can be roughly approximated by the maximum operator occurrence and the maximum number of variables with overlapped lifetimes, respectively. By reducing the maximum operator occurrence, the scheduled behavior is likely to result in a design which requires less number of functional units, which in turn may result in smaller chip area. Similarly, the storage unit area may be reduced by reducing the maximum number of live variables in the scheduled behavior.

Goal A: reduce the maximum operator occurrence

This goal can be achieved by either *balancing the operator occurrences over all states by moving assignments* or *splitting those states which have maximum operator occurrence.*

For example, the operator occurrences (Op. Occ.) metric in Figure 8(a) shows that a maximum number of three additions are performed in state ST2, while there is only one addition performed in either state ST1 or ST3. To implement this behavior, at least three adders are required. Therefore, to reduce the number of adders, the user could try to reduce the number of additions in state ST2 by moving one of the additions to either ST1 or ST3.

However, moving operators inappropriately may introduce side-effects. Notice that the

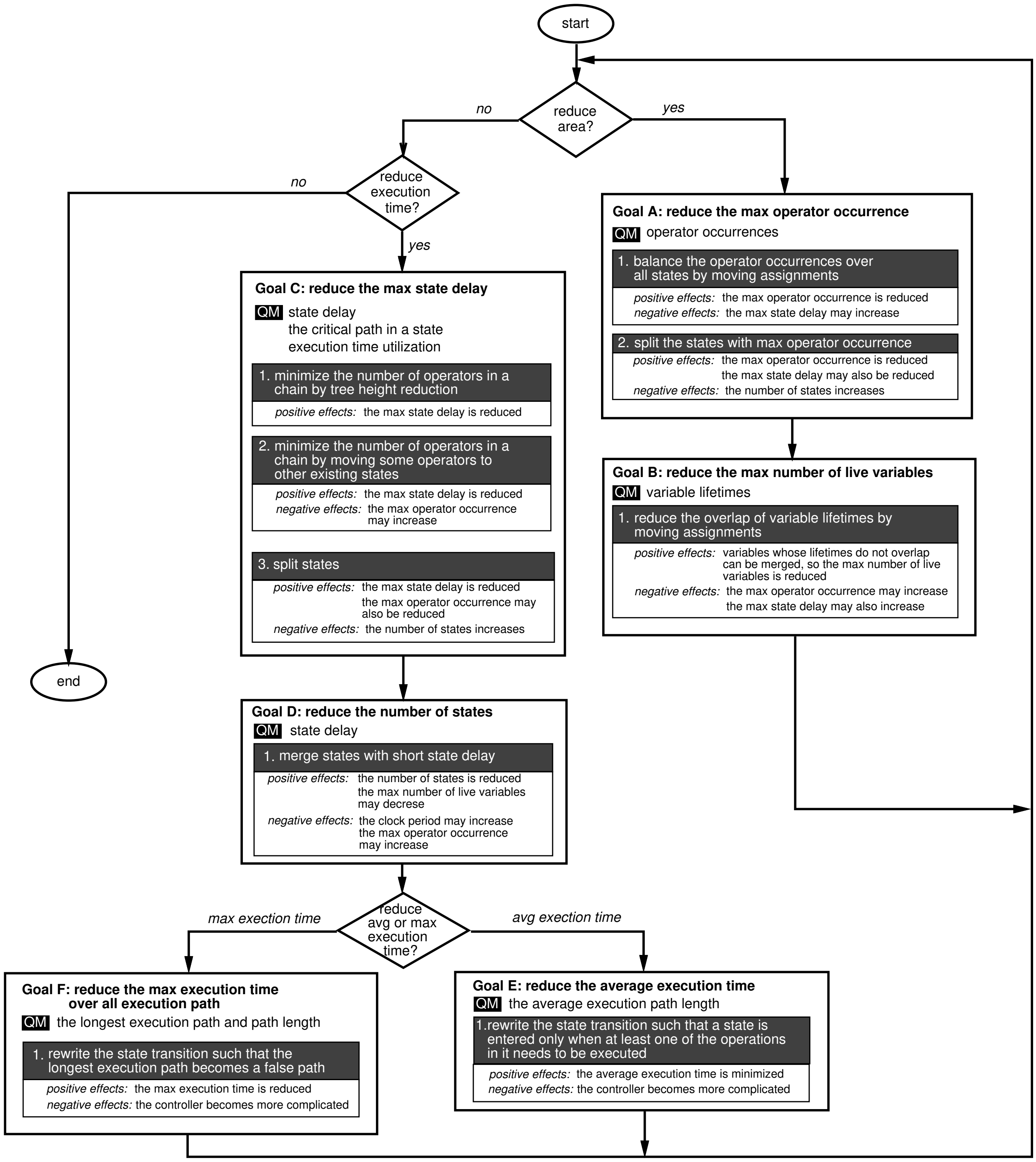


Figure 7: Techniques to optimize the scheduled behavior

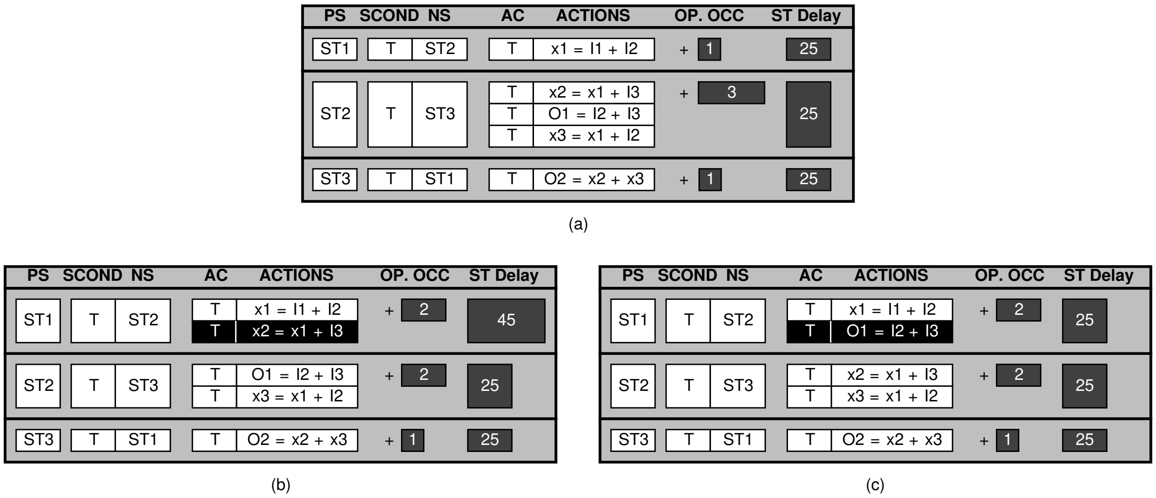


Figure 8: The example of reducing area by moving assignments

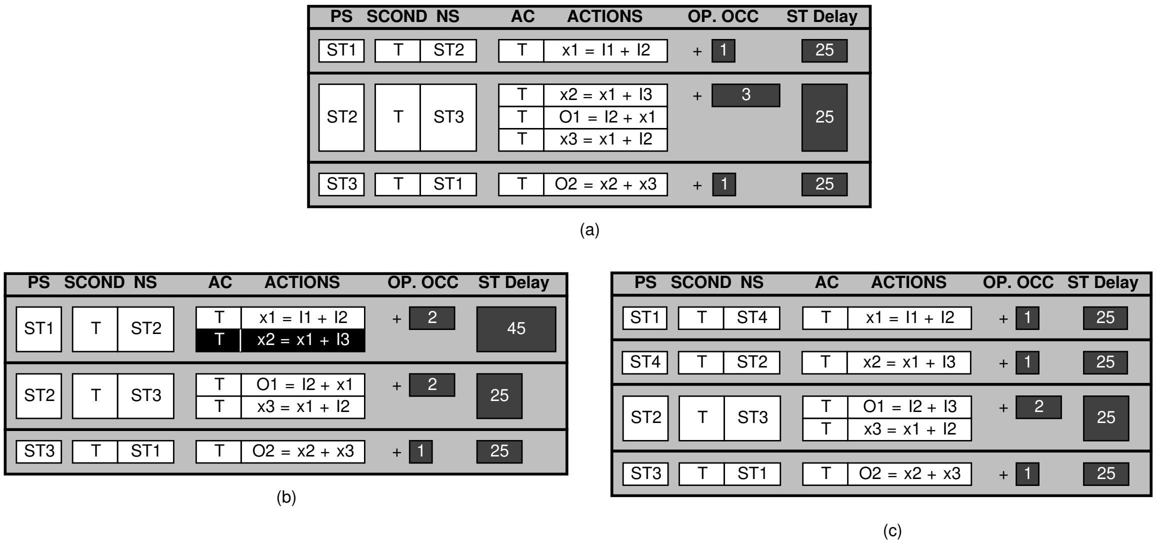


Figure 9: The example of reducing area by splitting a state

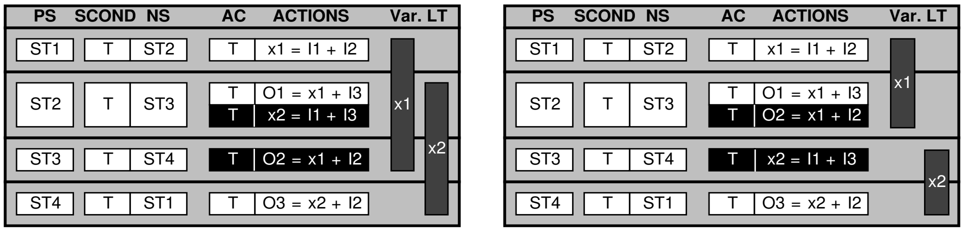
state delay metric in Figure 8(a) shows that the maximum state delay is 25 ns. Figure 8(b) shows the result after moving the assignment $x2 = x1 + I3$ to state ST1. Because of the data dependency between the assignments $x1 = I1 + I2$ and $x2 = x1 + I3$, the state delay of ST1 is estimated to be 45 ns by summing up the delays of the two additions and the register setup time. Thus, the clock period is increased from 25 ns to 45 ns. On the other hand, there is no data dependency between $O1 = I2 + I3$ and $x1 = I1 + I2$; therefore, moving $O1 = I2 + I3$ from state ST2 to ST1 as shown in Figure 8(c) incurs no penalty in performance.

If there is no possible way to move the assignments without increasing the maximum state delay, the user could try state splitting. Figure 9 shows such an example. To implement the behavior in Figure 9(a) requires three adders and the clock period is 25 ns. To reduce the required number of functional units, the user could try to move either one of the additions in state ST2 to the other states. However, because of the data dependencies, moving either one of the additions increases the maximum state delay. For instance, Figure 9(b) shows the result of moving $x2 = x1 + I3$ to state ST1. The new clock period is now 45 ns, Instead of moving operators, the user could split state ST2 as shown in Figure 9(c). A new state ST4 is inserted between states ST1 and ST2 and $x2 = x1 + I3$ is moved to the new state. The maximum operator occurrence of additions is now reduced from three to two, and the maximum state delay is maintained at 25 ns. Therefore, only two adders are required now and the clock period does not increase.

However, the total number of states increases after a state is splitted. Since the total execution time of a synchronous design is the product of the clock period and the total number of states, the total execution time increases after state splitting. If reducing the required functional units is the major goal, the user should decide whether to move operators or to split states by evaluating the amount of extra execution time required. For example, Figure 9(b) (result of moving operators) requires total execution time 45×3=135 ns, while Figure 9(c) (result of splitting states) requires total execution time 25×4=100 ns. Thus, in this example, splitting state ST2 is a better choice.

Goal B: reduce the maximum number of live variables

In the implementation, variables that have non-overlapping lifetimes can share the same storage unit. Thus, one way to reduce the total number of storage units required in the design is to schedule the usage of variables such that minimal lifetime overlapping occurs.



(a) (b)

Figure 10: The example of reducing area by moving variables

For example, a state table with the lifetimes of variables $x1$ and $x2$ are shown in Figure 10(a). Since the lifetimes of variables $x1$ and $x2$ overlap, this example would require two registers.

To reduce the number of storage units, the user can reschedule the behavior by swapping $x2 = I1 + I3$ in state ST2 with $O2 = x1 + x2$ in state ST3. That is, the user can move $x2 = I1 + I3$ from ST2 to ST3 and also move $O2 = x1 + x2$ from ST3 to ST2. As shown in Figure 10(b), moving the variables reduces the lifetimes of variables $x1$ and $x2$ such that it is possible now that $x1$ and $x2$ share the same register. Hence the required number of registers is reduced from two to one. However, this technique has to be applied with care since moving variables can be done only by moving assignments and moving assignments may change the operator occurrences of some states. As an undesired side-effect, the maximum operator occurrence may increase.

5.3.2 Reduce Execution Time

If a behavior is described by straight-line code, that is, there is no state branches in the schedule, the start-to-finish execution time for the behavior is proportional to the clock period, where the clock period is defined as the maximum time needed to execute a state, and the number of states. Therefore, to reduce the execution time of a design, the user could try to reduce either or both of the maximum state delay and the number of states. In the general case, a scheduled behavior may consist of state branches because of the conditional (such as if-then-else) and iteration (such as loops) constructs in the behavior. Depending on the performance constraint, the user may need to reduce either the maximum execution time or the average execution time. For example, if the throughput of the design is given as a constraint, the user has to assure that the maximum execution time of the design will not violate the throughput constraint.

Goal C: reduce the maximum state delay

To reduce the maximum state delay, the user must first identify the state which has the longest delay and the operators on the critical path which cause the longest delay. And then the maximum state delay can be reduced by minimizing the number of operators in a chain, this in turn can be achieved by either *applying tree height reduction*, *moving some operators to other existing states*, or *splitting states*.

Tree height reduction is a well-known transformation technique, which uses the commutativity and distributivity properties of operators to decrease the height of a long expression chain. For example, the assignment $x = (((a + b) + c) + d) + (e + f)$ has a critical-path length of four, while after applying tree height reduction, the resulting assignment $x = ((a+b)+c)+(d+(e+f))$ has a critical-path length of three. Both assignments have the same behavior since they compute $x = a+b+c+d+e+f$. However, the second assignment has a shorter tree height, resulting in a shorter critical path, and therefore a shorter clock period.

Figure 11 gives an example of how to reduce the maximum state delay by moving operators. In this example, the delay of ST1 is estimated as 65 ns due to the chaining of a multiplication, an addition and the register setup time. Since this state has the maximum state delay, the clock period of the design is 65 ns.

In addition to display state delay, the ISE also highlights operators that determine the delay of a selected state. For example, the two chained operators in ST1 requiring a total delay time of 65 ns are highlighted. The user can reduce this delay by moving the addition in ST1 to ST2. By doing so, the state delay of ST1 is reduced to 45 ns, since only one multiplication is performed in this state. The maximum state delay for the new schedule shown in Figure 11(b) is reduced from 65 ns to 45 ns; thus, the new clock period is 45 ns. However, there is also an undesired side-effect. For example, to implement the original behavior in Figure 11(a) requires one adder and one multiplier. But to implement the new schedule in Figure 11(b) will require one more adder.

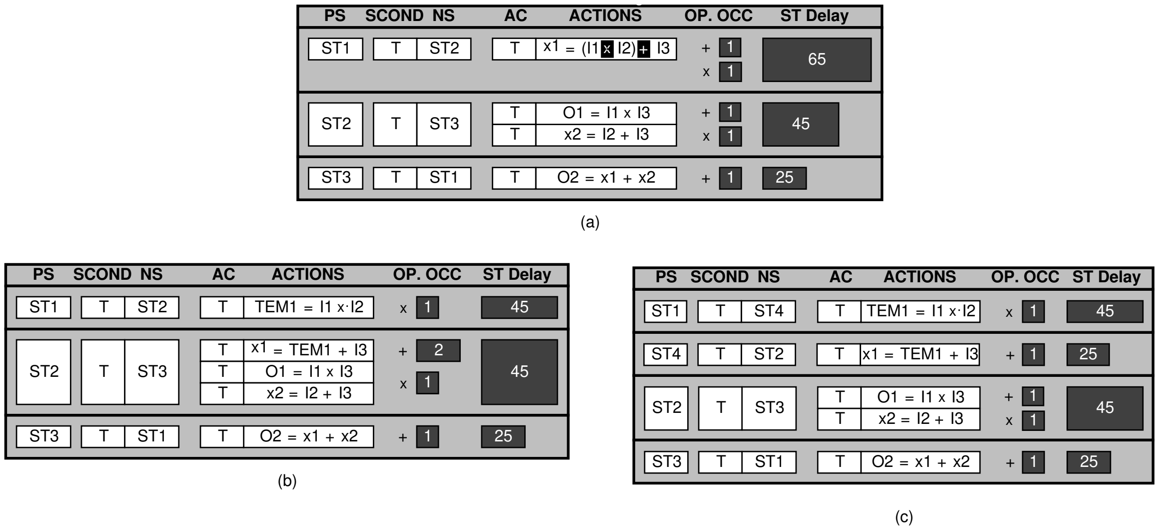


Figure 11: The example for reducing maximum state delay by moving operators or splitting states

On the other hand, Figure 11(c) shows the result of splitting state ST1 into two states. A new state ST4 is inserted and the addition is moved to the new state. In this behavior, the clock period is also reduced from 65 ns to 45 ns, but only one adder and one multiplier are required.

As mentioned previously, the number of states increases after state splitting. Since the execution time is equivalent to the product of the maximum state delay and the number of states, it is not necessarily true that reducing maximum state delay by splitting states would reduce the execution time. Moreover, there are many ways to split states: one state may be splitted into two or three or even more states and the user can also split more than one state. To decide how to perform state splitting, the execution time utilization metric is a good indication: the higher the resulting execution time utilization is, the shorter the execution time would be. For instance, the execution time utilization of the schedule in Figure 11(a) is $(65+45+25)/(65\times3)$=69.2%, which shows an improvement of execution time is possible. After splitting ST1 into two states, as shown in Figure 11(b), the resulting execution time utilization is $(45+25+45+25)/(45\times4)$=77.8% and the execution time is reduced from 195 to 180 ns.

On the other hand, moving operators does not increase the number of states. Therefore, the total execution time resulted from moving operators is generally shorter than the total execution time resulted from state splitting. For example, the behavior in Figure 11(b) requires the total execution time 45×3=135 ns, while the behavior in Figure 11(c) requires 45×4=180 ns. Therefore, in this example, since both moving operators and splitting states result in the same reduction of the clock period (from 65 ns to 45 ns), the user has to trade-off between the increase of the number of functional units (one extra adder in Figure 11(b)) and the longer execution time (180 ns in Figure 11(c) as opposed to 135 ns in Figure 11(b)).

Goal D: reduce the number of states

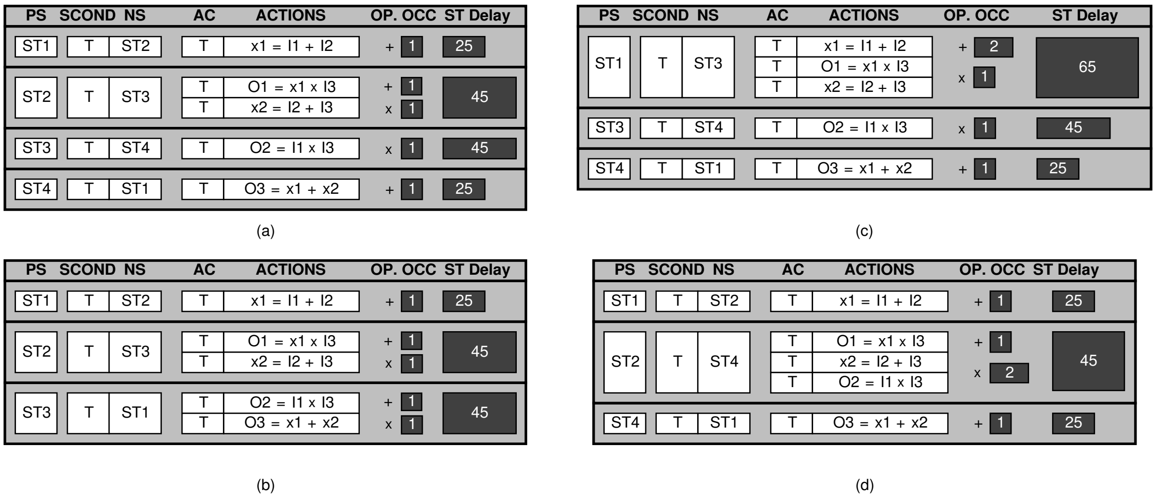


Figure 12: The example for reducing the number of states by merging two states

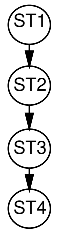
Other than trying to reduce the clock period, the user can also try to reduce the number of states in order to reduce the total execution time. Figure 12(a) shows an example. This example consists of four states that are sequentially executed one after another; therefore, the total execution time is 45×4=180 ns. To reduce the number of states, the user can merge either two of those four states. Figure 12(b) shows the result of merging states ST3 and ST4. The execution time is now 45×3=135 ns.

Like the other techniques discussed so far, merging states inappropriately could also introduce side-effects, that is, the operator occurrences or the state delay may increase. For example, Figure 12(c) shows the result of merging states ST1 and ST2. Because of the data dependency between $x1 = I1 + I2$ and $O1 = x1 \times I3$, the state delay is increased from 45 ns to 65 ns. Thus, the new clock period is 65 ns and the new execution time is 65 $\times$ 3=195 ns. Note that the total execution time increases instead of decreases in this case. Figure 12(d) shows the result of merging states ST2 and ST3. Although the execution time is reduced to 45 $\times$ 3=135 ns, the behavior now requires two adders and one multiplier as opposed to one adder and one multiplier for the behavior in Figure 12(a).

Goal E: reduce the average execution time

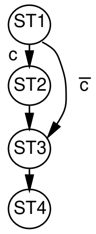
The average execution time can be reduced by rewriting the state transitions such that the schedule still performs the same behavior but a state is entered only when at least one of the operators in it needs to be executed. This is explained by the example shown in Figure 13.

PS	SCOND	NS	AC	ACTIONS
ST1	T	ST2	T	x1 = I1 + I2
ST2	T	ST3	c	O1 = x1 x I3
			c	x2 = I2 + I3
ST3	T	ST4	T	O2 = I1 x I3
ST4	T	ST1	T	O3 = x1 + x2



(a)

PS	SCOND	NS	AC	ACTIONS
ST1	c	ST2	T	x1 = I1 + I2
	$\bar{c}$	ST3		
ST2	T	ST31	T	O1 = x1 x I3
			T	x2 = I2 + I3
ST3	T	ST4	T	O2 = I1 x I3
ST4	T	ST1	T	O3 = x1 + x2



(b)

Figure 13: The example for reducing the average execution time

The average execution time of the schedule shown in Figure 13(a) is 45 $\times$ 4=180 ns since the clock period is 45 ns and the schedule consists of four states that are sequentially executed. However, notice that in state ST2, if the condition c is false, then none of the operations executed will be assigned to storage units. Hence, all operations performed in ST2, when c is false, are useless. Therefore, if c is false, the user could skip state ST2 by branching from ST1 directly to ST3. Thus, the state transition can be modified such that ST1 transits to ST2 only when c is true; otherwise ST1 transits to ST3. The resulting schedule is shown in Figure 13(b). This schedule consists of two execution paths: ST1 $\rightarrow$ ST2 $\rightarrow$ ST3 $\rightarrow$ ST4, while c is true, and ST1 $\rightarrow$ ST3 $\rightarrow$ ST4, while c is false. If we assume that the probability of c being true is 0.5, the average number of states required to be executed during one single execution will be $0.5 \times 4 + 0.5 \times 3 = 3.5$. Since the rewriting of state transitions does not affect the clock period, the average execution time is reduced from 180 ns to $45 \times 3.5 = 157.5$ ns.

Goal F: reduce the maximum execution time

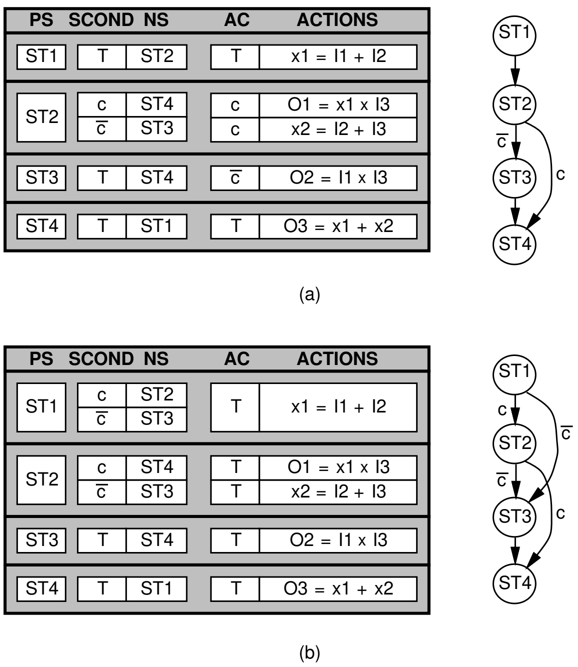


Figure 14: The example for reducing the maximum execution time

To reduce the maximum execution time, the user should first identify the longest execution path in the schedule. After the longest execution path is known, the user can then rewrite the state transitions such that the longest execution path becomes a false path. This will be explained in the following using the example shown in Figure 14.

The longest execution path of the schedule shown in Figure 14(a) is when the condition c is false, in which case, the execution path has to go through all four states sequentially. However, notice that when c is false, none of the operators in ST2 need to be executed. Therefore, the user can factor out the condition c and use it as the state transition condition to enter the state ST2. That is, only when c is true, ST2 needs to be executed; otherwise, ST1 transists to ST3. By doing so, the longest execution path in the original schedule, ST1 $\rightarrow$ ST2 $\rightarrow$ ST3 $\rightarrow$ ST4 has now become a false path. And the resultant schedule consists of only two paths: ST1 $\rightarrow$ ST2 $\rightarrow$ ST4 and ST1 $\rightarrow$ ST3 $\rightarrow$ ST4. Hence, the maximum execution time is reduced from 45×4=180 ns to 45×3=135 ns.

5.4 Perform Architectural Tradeoffs

At the structural level, the user needs to determine the type and quantity of resources used in the chip architecture. This task, called allocation, requires the user to make appropriate tradeoffs between the design's cost and performance. For example, if the original description contains inherent parallelism, allocating more resources increases area and cost, but it also creates more opportunities for parallel operations or storage accesses, resulting in better performance. On the other hand, allocating fewer resources decreases area and cost, but it also forces operations to execute sequentially, resulting in poorer performance. After each or all components have been allocated, the user can determine the binding of behavioral operators/variables to physical components.

However, at the early stage of design process, choices in choosing components to be used in the architecture are difficult because they are made with relatively little information. The tradeoffs may not still work after component binding or place and route. At times, the user may need to modify the allocation or binding in order to satisfy cost or performance constraints. Figure 15 shows a set of such techniques.

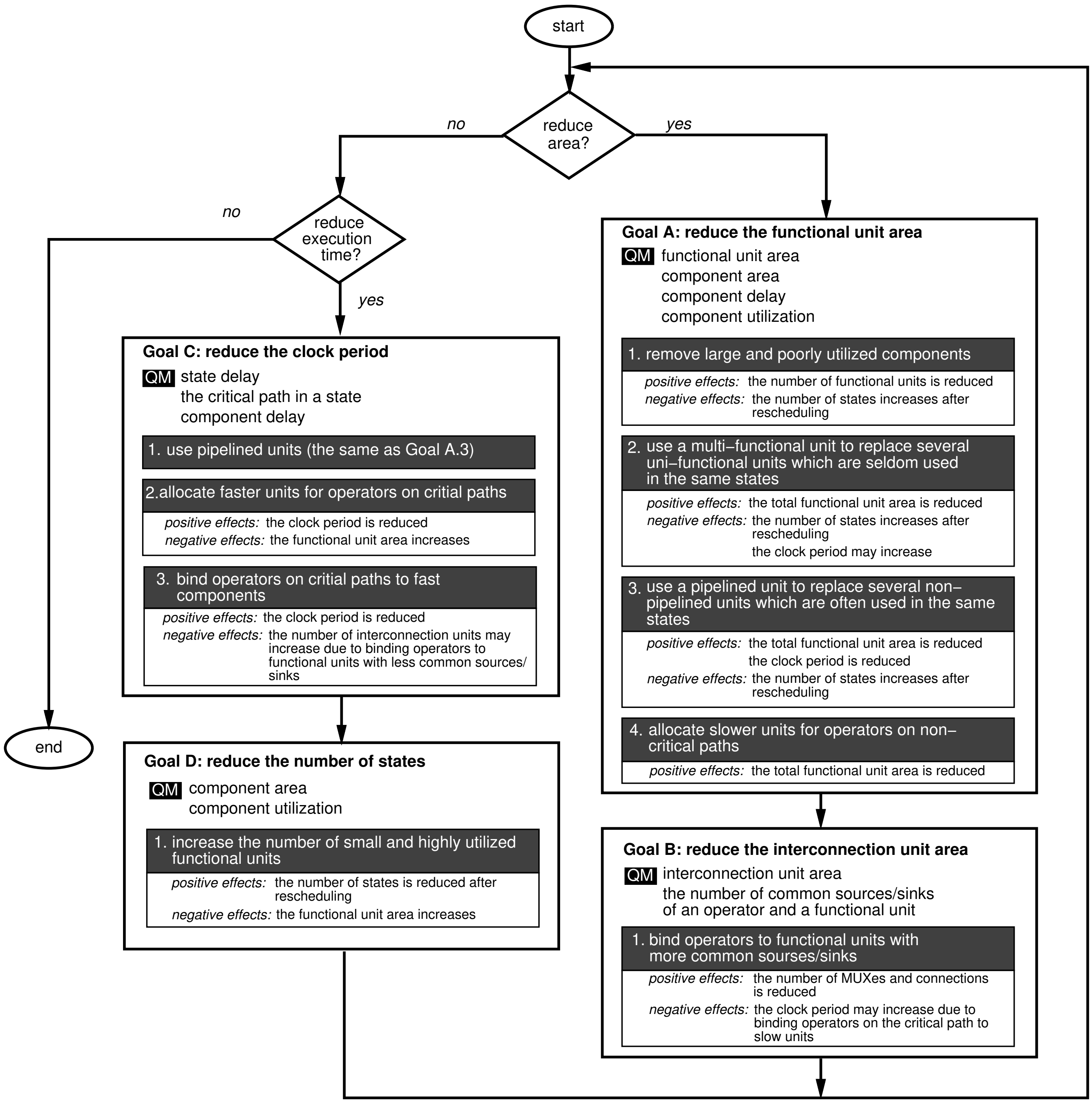


Figure 15: Techniques to perform architectural tradeoffs

5.4.1 Reduce Area

It is obvious that allocation directly affects the total functional unit area, thus, the functional unit area may be reduced by modifying allocation. Similarly, since the binding of operators to components determines the interconnections between allocated components, the user can try to reduce the interconnection unit area by modifying binding results.

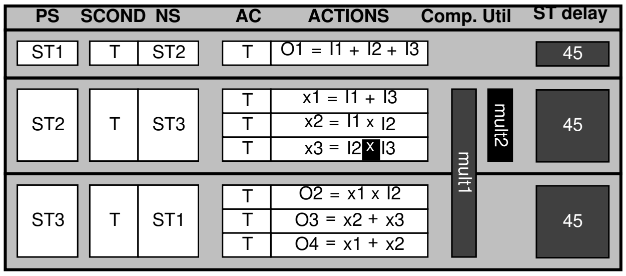
Goal A: reduce the functional unit area

The most straight-forward way to reduce the functional unit area is to *remove some of the allocated components*. This also gives the greatest functional unit area reduction. However, once some components are removed from the allocation, the behavior needs to be rescheduled such that the operations in each state can be computed by the remaining components. The resultant schedule needs longer execution time since there are less resources and the operators are forced to executed sequentially.

The component utilization metric in the unit selection and binding view, which calculates the number of states in which the component is used versus the total number of states, can help the user to decide which component to remove. Low utilization indicates that the component is inefficiently used. Large components with low utilization are the better candidates to be removed because the removal can give great area reduction but small performance penalty. For example, Figure 16(a) shows that the multiplier *mult2* is utilized in only one state. By removing *mult2*, the total functional unit area can be reduced by 100,000 μm^2. However, removing *mult2* requires the state ST2 to be splitted into two states such that the remaining multiplier *mult1* can be used to perform two multiplications sequentially. The result is shown in Figure 16(b). The total execution time is increased from 135 to 180 ns due to the increase in the number of states.

The user can also use a large component to replace several small components, as far as the resulting total area is smaller. This can be done in two ways: *replacing two components which perform different functions by one multi-functional component, which is capable of performing both of the functions*, or *replacing two components which perform the same functions by one pipelined component.*

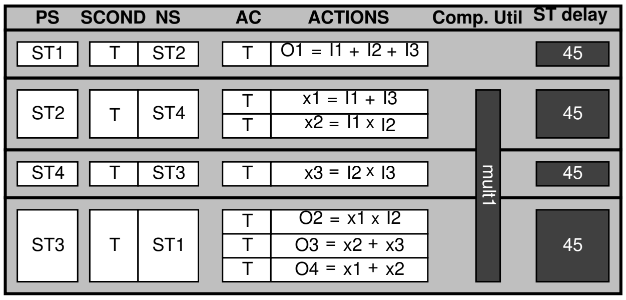
In practice, multi-functional components cost less than a set of uni-functional components that perform the same operations. For example, although an adder-subtractor component costs twenty percent more than an adder or subtractor separately, it is consid-



Name	Type	Bitwidth	Area	Delay	Util
adder1	ADD	8	25600	20	100%
adder2		8	25600	20	67.7%
mult1	MULT	8	100000	40	67.7%
mult2		8	100000	40	33.3%

total functional unit area = 251200
total execution time = 45 x 3 = 135 ns

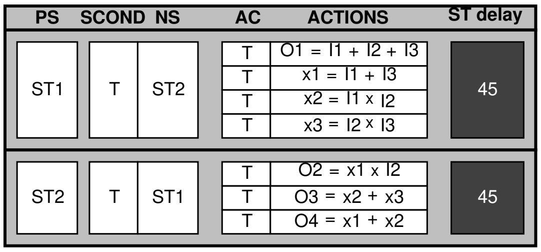
(a)



Name	Type	Bitwidth	Area	Delay	Util
adder1	ADD	8	25600	20	100%
adder2		8	25600	20	75%
mult1	MULT	8	100000	40	75%

total functional unit area = 151200
total execution time = 45 x 4 = 180 ns

(b)

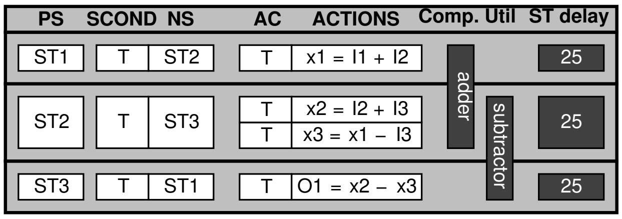


Name	Type	Bitwidth	Area	Delay	Util
adder1	ADD	8	25600	20	100%
adder2		8	25600	20	100%
adder3		8	25600	20	50%
mult1	MULT	8	100000	40	100%
mult2		8	100000	40	50%

total functional unit area = 276800
total execution time = 45 x 2 = 90 ns

(c)

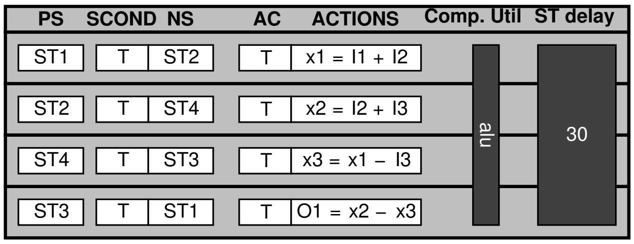
Figure 16: The example for adding or removing components.



Name	Type	Bitwidth	Area	Delay
adder	ADD	8	25600	20
subtractor	SUB	8	28000	20

total functional unit area = 53600 um^2
total execution time = 25 x 3 = 75 ns

(a)



Name	Type	Bitwidth	Area	Delay
alu	ALU	8	34000	25

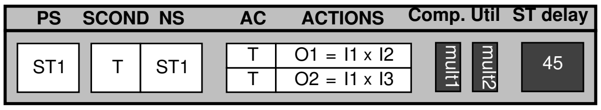
total functional unit area = 34000 um^2
total execution time = 30 x 4 = 120 ns

(b)

Figure 17: The example for replacing two components by one multi-functional component

erably cheaper than using one adder and one subtractor. For example, Figure 17(a) shows a design using one adder and one subtractor. The maximum state delay of this design is 25 ns. Assume that there exists an ALU, which can perform both addition and subtraction, in the component library and the area and delay of the ALU are 34,000 μm^2 and 25 ns. By using one of such ALUs to replace the allocated adder and subtractor, the total functional unit area can be reduced by 33%. However, the bar graph Comp. Util in Figure 17(a), which indicates the states in which each component is used, shows that both adder and subtractor need to be used in state ST2. Consequently, ST2 needs to be splitted into two states such that the ALU performs the addition in one state and the subtraction in the other. Notice that the clock period needs also to be increased from 25 ns to 30 ns because the ALU is slower than the adder and the subtractor. Figure 17(b) shows the result of using one ALU instead of one adder and one subtractor. The total execution time is now increased from 75 to 120 ns. In conclusion, by using one multi-functional component to replace two components, the total functional unit area is usually reduced, but the execution time is increased.

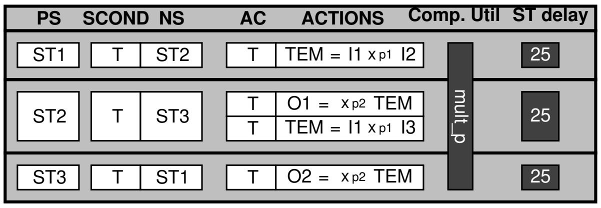
On the other hand, when there are two operators executed concurrently in the same state, these two operators can share the same two-stage pipelined component. This sharing is possible because each operator uses a different stage of the pipelined component. As a result, one pipelined component instead of two non-pipelined component can be used for these two concurrently executed operators. This can give a great area reduction.



Name	Type	Pipeline	Bitwidth	Area	Delay
mult1	MULT	none	8	100000	40
mult2	MULT	none	8	100000	40

total functional unit area = 200000 um^2
total execution time = 45 ns

(a)



Name	Type	Pipeline	Bitwidth	Area	Delay
mult_p	MULT	2–stage	8	120000	20

total functional unit area = 53600 um^2
total execution time = 25 x 3 = 75 ns

(b)

Figure 18: The example for replacing two components by one pipelined component

Figure 18(a) shows a one-state design. There are two multiplications executed concurrently in the state and two non-pipelined multipliers are allocated. In Figure 18(b), a two-stage pipelined multiplier *mult_p* is used to replace the non-pipelined multipliers. To use *mult_p*, the state ST1 is splitted into three states. The symbol $\times_{p1}$ denotes the first stage of the pipelined multiplication, while the symbol $\times_{p2}$ denotes the second stage. The variable TEM indicates the pipeline register in *mult_p*. In state ST1, the multiplication in $O1 = I1 \times I2$ uses the first stage of *mult_p*. In state ST2, while the multiplication progresses to the second stage of *mult_p*, the multiplication in $O2 = I1 \times I3$ uses the first stage. Finally, in state ST3, the multiplication is completed. By using the two-stage pipelined multiplier, the area is reduced from 200000 to 120000 μm^2 and the clock period is reduced from 45 ns to 25 ns.

Notice that using pipelined components also requires the original behavior to be rescheduled. And as a consequence, the number of states increases. Hence, although the clock period is reduced, the total execution time may become longer. In the example shown in Figure 18, the total execution time is increased from 45 to 75 ns.

When the required area reduction is not too large, there is one technique which gives no performance loss. This can be done because component libraries in reality often have multiple implementations of the same component, each implementation having a different area/delay characteristic. For example, an addition can be done either quickly with a large (hence, costly) carry-look-ahead adder or slowly with a small (hence, inexpensive) ripple-carry adder. Therefore, for each operator in the behavior, an efficient component can be selected such that the slower and cheaper components would be used by operators on non-critical paths and the faster and more expensive components would be used only by operators on the critical path.

PS	SCOND	NS	AC	ACTIONS	ST Delay
ST1	T	ST2	T	O1 = I1 + I2 + I3	45
			T	x1 = I1 + I3	
ST2	T	ST1	T	O2 = x1 x I2	55

Name	Type	Bitwidth	Area	Delay	C. Util
adder1	ADD	8	25600	20	72.7%
adder2		8	25600	20	72.7%
adder3		8	25600	20	36.4%
mult	MULT	8	100000	50	90.9%

total functional unit area = 176800
total execution time = 55 x 2 = 110 ns

(a)

PS	SCOND	NS	AC	ACTIONS	ST Delay
ST1	T	ST2	T	O1 = I1 + I2 + I3	45
			T	x1 = I1 + I3	
ST2	T	ST1	T	O2 = x1 x I2	55

Name	Type	Bitwidth	Area	Delay	C. Util
adder1	ADD	8	25600	20	72.7%
adder2		8	25600	20	72.7%
adder4		8	18000	40	72.7%
mult	MULT	8	100000	50	90.9%

total functional unit area = 169200
total execution time = 55 x 2 = 110 ns

(b)

PS	SCOND	NS	AC	ACTIONS	ST Delay
ST1	T	ST2	T	O1 = I1 + I2 + I3	45
			T	x1 = I1 + I3	
ST2	T	ST1	T	O2 = x1 x I2	45

Name	Type	Bitwidth	Area	Delay	C. Util
adder1	ADD	8	25600	20	90.9%
adder2		8	25600	20	90.9%
adder4		8	18000	40	90.9%
mult	MULT	8	120000	40	90.9%

total functional unit area = 189200
total execution time = 45 x 2 = 90 ns

(c)

Figure 19: The example for performing component cost/speed tradeoff

The clock utilization metric in the unit selection and binding view gives the average percentage of clock cycle that is utilized by the component. Components which are used by operators on non-critical paths are idle after they complete their computation and before state changes. This is indicated by low clock utilization. Therefore, the components with low clock utilization can be replaced by slower components which save area and incur no performance penalty. For example, Figure 19(a) shows that the clock utilization of *adder3* is as low as 36.4%. By replacing it with a slower (40 ns) and smaller (18,000 μm^2) adder *adder4*, the total functional unit area is reduced from 176800 to 169200 μm^2 while the total execution time remains the same.

Goal B: reduce the interconnection unit area

As discussed in previous section, ISE offers hints for binding operators and variables to allocated components. To use these hints, the user selects one unbound operator or variable, then asks the hints to suggest components to bind to. The user can further control the estimation algorithm of these hints by giving different weights to different factors: function compatibility, bitwidth compatibility, sources closeness, sinks closeness, performance and area. By varying the weights, the user can make the hints suggesting components so that the binding can minimize different cost functions such as interconnection area or the clock period.

The sources/sinks closeness factors measure the commonality between the sources/sinks of the selected operator/variable and the sources/sinks of the operators/variables that are already bound to a component. Therefore, given the highest weights to the sources closeness and the sinks closeness factors, the component highlighted with the brightest shade by binding hints algorithm should result in a binding such that the number of interconnection units is minimized.

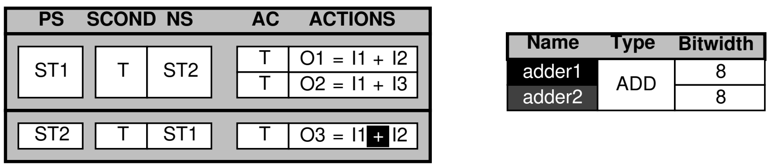


Figure 20: The example for minimizing the interconnection area during binding

For example, Figure 20 shows an allocation of two adders. Assume that the first addition in state ST1 has been bound to *adder1* and the second addition has been bound to *adder2*. Given the highest weights to the sources closeness and the sinks closeness factors, the hints highlight both of the adders since both of them can be used to perform the addition. However, the different shading suggests that *adder1* is the better choice for binding the addition in state ST2.

5.4.2 Reduce Execution Time

As mentioned in previous section, a design's execution time can be reduced by either reducing its clock period, or reducing the number of states.

Goal C: reduce the clock period

By *using a pipelined component to replace several non-pipelined components*, the user can reduce not only the functional unit area but also the clock period. This is because, if the longest state delay is dominated by the component delay of a slow component, by using a pipelined component to replace this slow component, the clock period can be reduced from the original component delay to the pipeline stage delay of the pipelined component. Notice that in the example shown in Figure 18, after the non-pipelined multipliers *mult1* and *mult2* are replaced by the pipelined multiplier *mult_p*, the clock period is reduced from 45 to 25 ns.

Another technique to reduce the clock period is to *allocate faster components for operators on the critical path.* Components frequently used by operators on critical paths have high clock utilization. If they are replaced by faster components, the clock period can often be reduced. However, since faster components are usually larger, the functional unit area increases. For example, notice that the multiplier *mult* in Figure 19(b) has the highest clock utilization, 90.9%. By replacing it with a faster (40 ns) and larger (120,000 μm^2) multiplier, the clock period is reduced to 45 ns and the total execution time is reduced from 110 ns to 90 ns. Yet, the total functional unit area is now increased from 169200 to 189200 μm^2 due to the larger multiplier.

The user can also minimize the clock period by binding the operators on the critical path to the fast components and the operators which are not on the critical path to slow components. To do this, the user needs to first identify the state with maximum state delay by looking at the state delay metric, and the system will indicate to the user the critical path in the state. Then the user can select one of the operators on the critical path and ask for binding hints. By giving the highest weight to the performance factor, the fastest component available in the allocation should be suggested. Such a binding will leave slow components to the operators which are not on the critical path.

For example, Figure 21(a) shows an allocation of three adders with delays of 20, 30 and 40 ns respectively. Notice that the addition highlighted in Figure 21(a) is on the critical

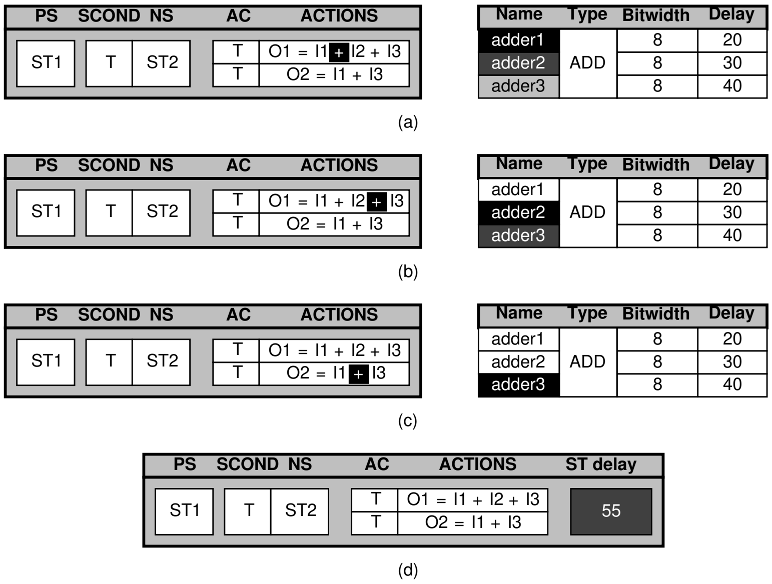


Figure 21: The example for minimizing the clock period during binding

path. Giving the highest weight to the performance factor, binding hints highlight all three adders but suggest that *adder1*, which is the fastest adder, is the better choice. After the addition is bound to *adder1*, if the user asks for binding hints for the second addition highlighted in Figure 21(b), only *adder2* and *adder3* are highlighted since *adder1* has been used to perform the first addition in Figure 21(a). Now *adder2*, which is the faster one between *adder2* and *adder3*, is suggested as the better choice. Binding the addition to *adder2* leaves only *adder3* to the third addition, as shown in Figure 21(c). After the binding is done, Figure 21(d) shows the state delay, which is 55 ns. If the binding is done by any other way, the state delay will be longer.

Goal D: reduce the number of states

If the behavior contains inherent parallelism, allocating more components requires larger chip area but it allows more operators to be performed in parallel, resulting in less number of states and better performance.

In the unit selection and binding view, the component utilization metric helps the user to decide what component to add to produce the largest performance gain while increasing the least area. In an allocation, a component having high component utilization indicates that this component is frequently used. Therefore, increasing the number of this component may allow some operators which are currently executed sequentially to be executed in parallel. For example, the adders in Figure 16(a) have high component utilization. By adding one more adder, the total functional unit area is increased by only 25600 μm^2 while the required number of states is decreased from three to two. The result can now be executed within 90 ns, as shown in Figure 16(c).

5.5 Optimize the Floorplan

Once some of the operators or variables are bound to hardware components, the floorplan view in ISE can show the user the placement of those components, I/O ports, routing and wasted area. Also, the user is allowed to modify the floorplan to reduce the chip area or wiring delay. Figure 22 shows a set of floorplanning techniques that the user can perform to optimize either area or performance of the design at the physical level.

5.5.1 Reduce Area

At the physical level, the user can reduce the chip area by reducing the wasted area or the routing area.

Goal A: reduce the wasted area

In the floorplan view, the user can reduce the wasted area by *changing the placement of components such that the floorplan becomes more compact.* Changing the component placement can be done by rotating or moving components. However, if the components on the critical path happen to be placed far apart, then the clock period increases. For example, Figure 23(a) and (b) show different floorplans of the same design. The floorplan view highlights the components on the critical path, which are the adder and the multiplier in this example. Figure 23(a) shows a more compact floorplan where components on the critical path are placed far apart. Figure 23(b) shows a less compact floorplan but the adder and the multiplier are placed close together. Since the wiring delay may be an important portion in the critical path delay, the floorplan in Figure 23(b) gives shorter clock period but larger chip area, while the floorplan in Figure 23(a) gives longer clock period but smaller chip area.

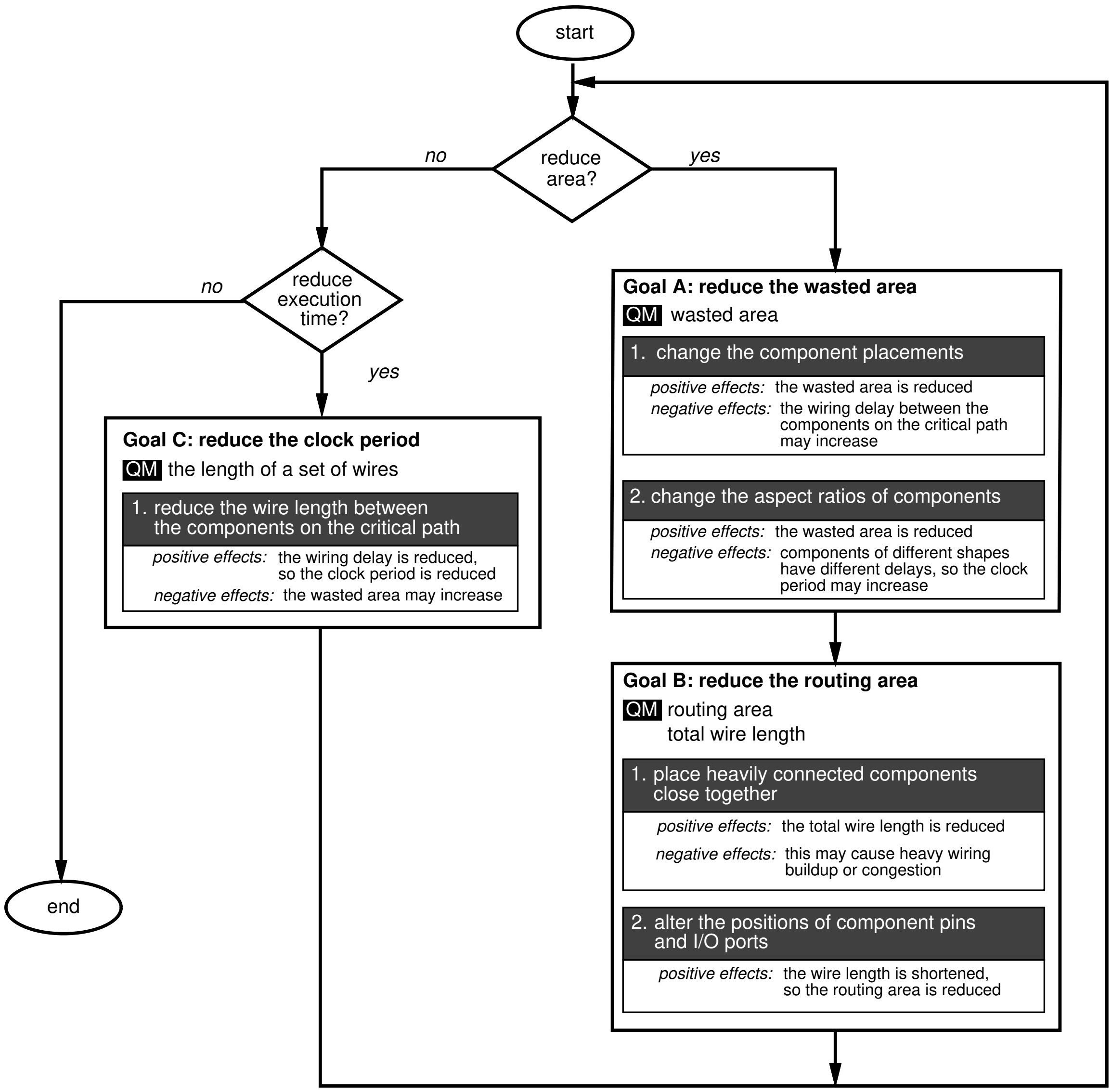
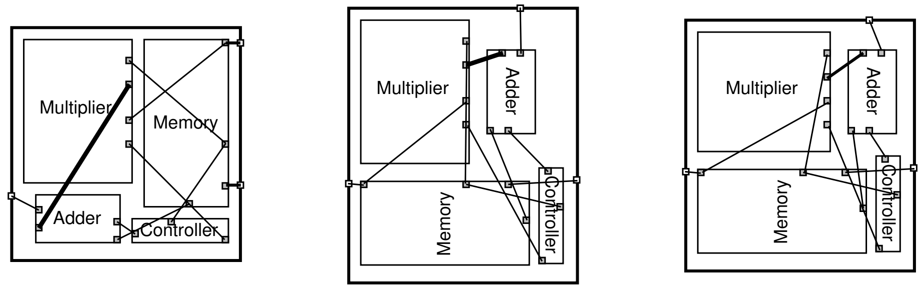


Figure 22: Techniques to optimize the floorplan

Changing the aspect ratios of components can also make a floorplan more compact. However, a component's shape is closely related to its pin-to-pin delay. Thus, by changing a component's aspect ratio, its delay is also changed and consequently, the clock period of the design may be affected. For example, Figure 23(c) shows the result of changing the aspect ratio of the multiplier in Figure 23(b). Assume the new multiplier used in Figure 23(c) has a longer delay, the clock period is increased because the multiplier is on the critical path.



(a)

Total area:	5000 um^2
F.U. area:	2250 um^2
S.U. area:	1250 um^2
Routing area:	700 um^2
Wasted area:	800 um^2
Critical path wire length:	542 um
Total wire length:	1247 um

(b)

Total area:	5300 um^2
F.U. area:	2250 um^2
S.U. area:	1250 um^2
Routing area:	800 um^2
Wasted area:	1000 um^2
Critical path wire length:	100 um
Total wire length:	1247 um

(c)

Total area:	5250 um^2
F.U. area:	2400 um^2
S.U. area:	1250 um^2
Routing area:	700 um^2
Wasted area:	900 um^2
Critical path wire length:	100 um
Total wire length:	1250 um

Figure 23: The example for changing the placement and aspect ratios of components

Goal B: reduce the routing area

The floorplan view provides the user total wire length metric, which can be an indication of the routing area. By *placing heavily connected components close to each other*, the total wire length can be reduced and the eventual routing area may consequently be reduced. However, this may sometimes lead to heavy wiring buildup or congestion. For example, Figure 24 shows two possible component placements of a netlist. The one shown in Figure 24(b) has shorter wire length than the one shown in Figure 24(a). However, the one shown in Figure 24(a) would require only two routing tracks, but the one in Figure 24(b) would require three because of the connection between components D and G. Therefore, in this example, the placement that minimizes wire length requires more routing area than the placement with longer wire length.

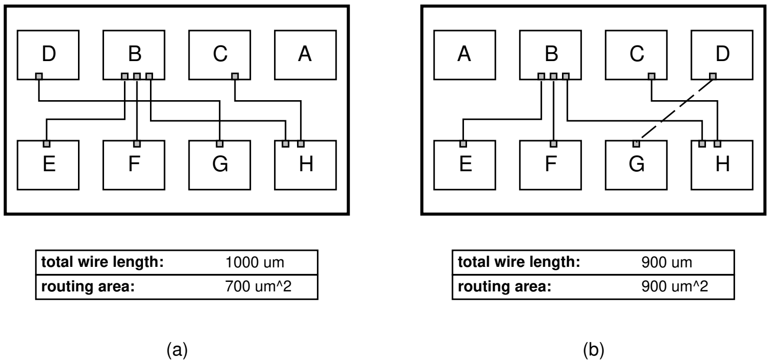


Figure 24: The example for minimizing the total wire length

When there are a large number of components, the user may perform floorplanning hierarchically. In hierarchical floorplan, the components are first grouped into modules. The floorplanning is then further performed on the set of modules. In this approach, the I/O port positions of the modules will determine the quality of the routing between modules. Similarly, on the chip level, the positions of I/O pads will affect the routing.

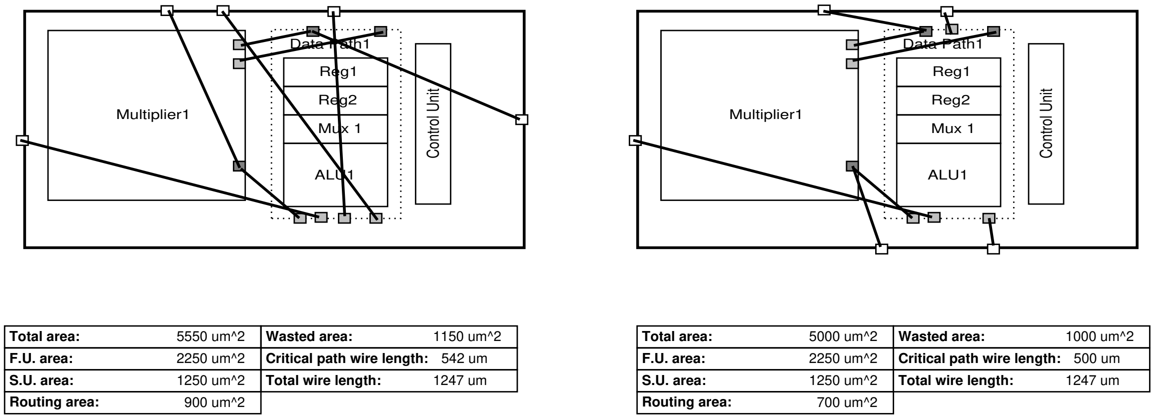


Figure 25: The example for altering the positions of I/O ports

In the floorplan view, the quality metric routing area gives the estimated total area consumed by routing. To reduced the routing area, the user can alter the I/O port positions of the modules or the positions of chip's I/O pads manually. Figure 25 shows the floorplan of a design before a possible improvement and the floorplan of the same design after. The improvement is accomplished by rearranging the I/O pads and altering the I/O port positions of the datapath module.

5.5.2 Reduce Execution Time

At the physical level, there is no way that the user can reduce the number of states by simply optimizing the floorplan. However, the clock period could be reduced by reducing the wiring delay between components on the critical path.

Goal C: reduce the clock period

We have demonstrated in the example shown in Figure 23(a) and (b) that the clock period can be reduced by *reducing the wire length between the components on the critical path.* And the side-effect of this technique, which is the wasted area being increased as the floorplan becoming less compact, has also been explained previously.

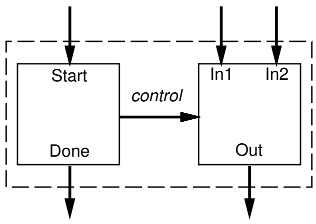
6 Sample Example Walk-Through

To illustrate the application of the proposed methodology, we shall walk through a simple design and annotate the key decision points in the design.

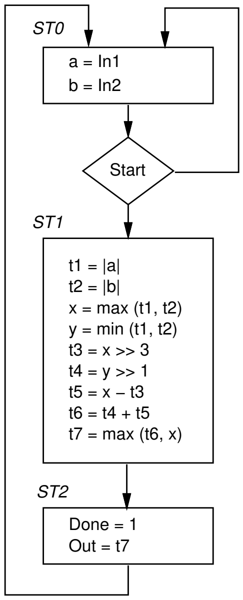
Figure 26 shows the specification of this walk-through example, which is designed to compute the square-root approximation (SRA) [5] of two signed integers, a and b, by the following formula:

$$\sqrt{a^2 + b^2} \approx max((0.875x + 0.5y), x)$$

where $x = max(|a|, |b|)$, and $y = min(|a|, |b|)$. According to Figure 26(a), this design has two input ports, *In1* and *In2*, which are used to read integers a and b, and one output port *Out*. As shown in the flow-chart in Figure 26(b), the design reads the input ports and starts the computation whenever the input control signal *Start* becomes equal to 1. After the computation is done, it makes the result available through the *Out* port for one clock cycle. At the same time, it sets the control signal *Done* to 1, in order to signal to the environment that the data that has appeared at the *Out* port is a valid result. Figure 26(c) shows the component library that will be used in implementing this design. This component



(a)



(b)

component	**functions**	**delay(ns)**	**area(um^2)**
add	*+*	16.4	110,880
sub	*–*	17.5	119,808
alu	*+, –*	19.8	160,416
min	*min*	23.2	149,472
max	*max*	26.5	162,432
max_min	*max, min*	30.9	180,576
abs	*absolute*	23.3	149,472
		25.5	123,886
reg	*register*	3.5(setup) 5.4(hold)	49,824
2–1 mux	*2 to 1 mux*	5.7	29,664
3–1 mux	*3 to 1 mux*	6.0	49,536

(c)

Figure 26: The specification of the SRA example

library is created based on the VLSI Technology, Inc. 1.0 micron CMOS VDP370 datapath cell library [12]. The constraints for the design are area smaller than 2,500,000 μm^2 and maximum execution time no longer than 300 ns.

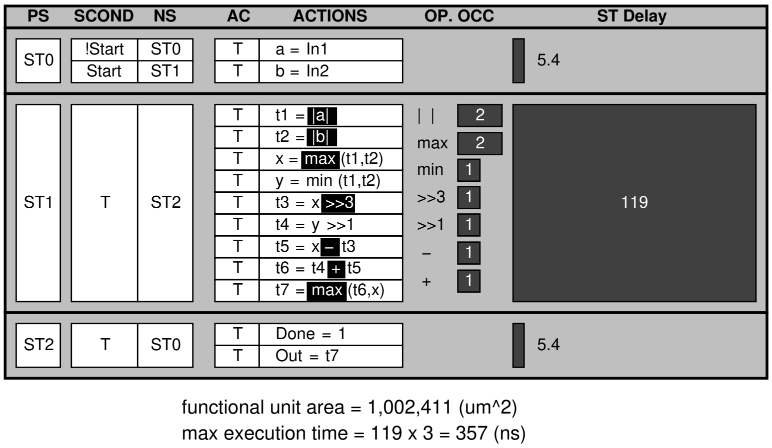


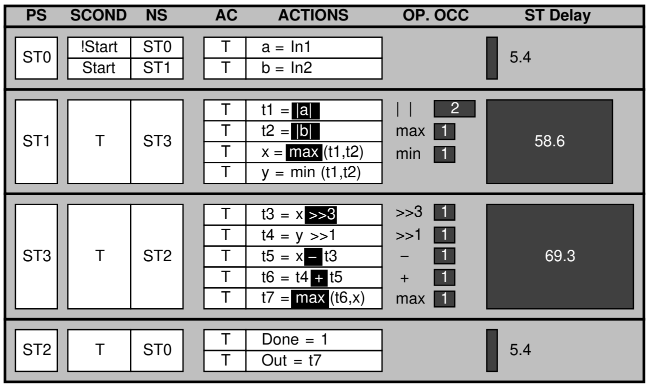
Figure 27: The state-action table of the SRA example

Figure 27 shows the state-action table representation of the design, obtained from Figure 26(b). Also shown in this figure are the quality metrics, operator occurrences (OP. OCC) and state delay (ST Delay). From the operator occurrences metric, it is obvious that the current schedule requires at least two components for the computation of absolute value, two components for the computation of maximums, one component each for the computation of minimum, addition, and subtraction. Note that the two shift operations can be implemented by signal rearrangement and do not require any logic. Therefore, the functional unit area is estimated to be 1,002,411 μm^2, which is the sum of the areas of all the required components. At the same time, the state delay metric shows that the longest state delay is 119 ns; therefore, the clock period is 119 ns. Since the longest execution path consists of three states (ST0 $\rightarrow$ ST1 $\rightarrow$ ST2), the maximum execution time would be 119 $\times$ 3 = 357 ns, which clearly violates the performance constraint. To help the user identifying the performance bottleneck, ISE highlights the operators on the critical path, as shown in Figure 27. To shorten the critical path, ST1 is splitted into two states, ST1 and ST3, as shown in the state-action table in Figure 28(b). After splitting ST1, the new longest state is now ST3 whose state delay is 69.3 ns. Hence, the clock period is reduced from 119 to 69.3

ns and the maximum execution time is reduced from 357 to 277.2 ns, which now satisfies the performance constraint. At this point, we can switch our attention to the area of the design.

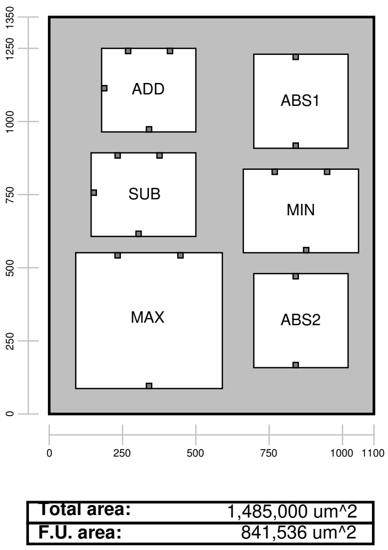
Name	Type	Bitwidth	Dealy	Area
ABS1 ABS2	abs	16	23.3	149,472
MAX	max	16	26.5	162,432
MIN	min	16	23.2	149,472
ADD	add	16	16.4	110,880
SUB	sub	16	17.5	119,808

(a)



max execution time = 69.3 x 4 = 277.2 (ns)

(b)



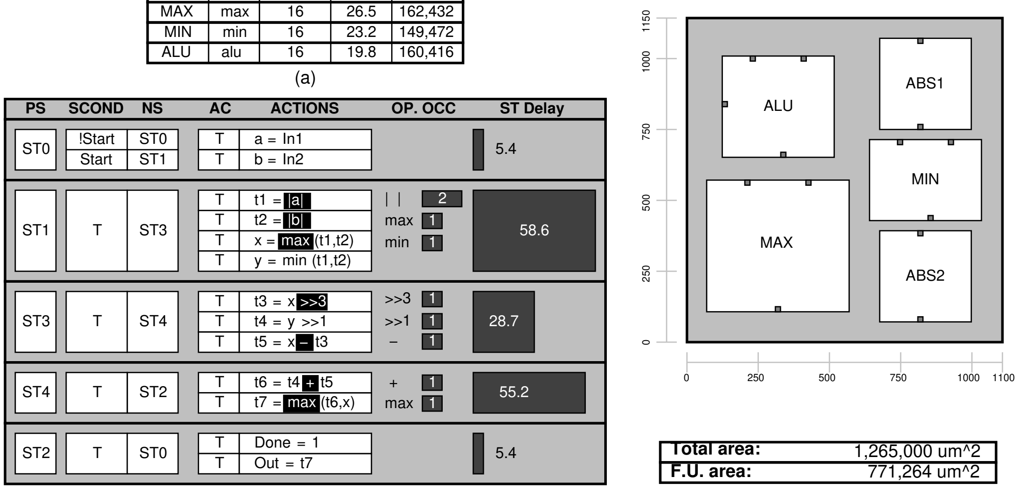
(c)

Figure 28: The design of the SRA example after splitting ST1 and allocation

From the operator occurrences metric shown in Figure 28(b), we can see that the maximum operator occurrence of the computation of maximums decreases to one after ST1 was splitted. Therefore, the current schedule requires two components for the computation of absolute value, and one component each for the computation of maximum and minimum, one adder and one subtractor. The allocation is shown in Figure 28(a). After the components are allocated, ISE allows the user to start floorplanning. Figure 28(c) shows a possible floorplan. The total area metric estimates that the current design would require 1,485,000 μm^2. Note that this does not include the storage unit area, interconnection unit area, routing area, and the controller. Knowing that the storage units, interconnection units and the controller, etc. may very well occupy more than half of the final design area, we should see whether it is possible to further reduce the functional unit area.

Name	Type	Bitwidth	Dealy	Area
ABS1	abs	16	23.3	149,472
ABS2				
MAX	max	16	26.5	162,432
MIN	min	16	23.2	149,472
ALU	alu	16	19.8	160,416

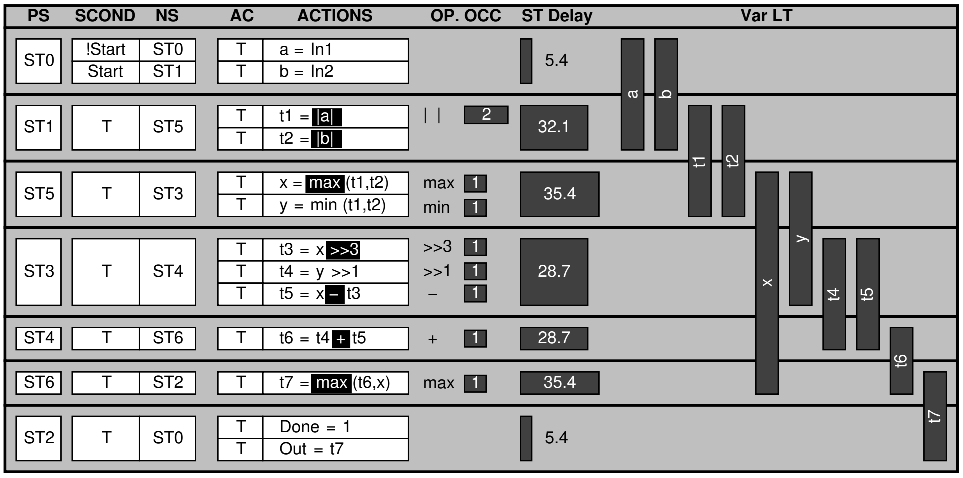
(a)



max execution time = 58.6 x 5 = 293 (ns)
execution time utilization = 52.3%

(b) (c)

Figure 29: The design of the SRA example after the first re-allocation



max execution time = 35.4 x 7 = 247.8 (ns)
execution time utilization = 69.1%

Figure 30: The design of the SRA example after splitting ST1 and ST4

In the component library, there is an ALU which can perform both addition and subtraction. Knowing that replacing the adder and subtractor by the ALU can reduce the functional unit area, a new allocation is obtained and shown in Figure 29(a). After modifying the floorplan, the total area is now approximately 1,265,000 μm^2. However, since the addition and subtraction are both executed in ST3, ST3 now needs to be splitted into two states such that one ALU can be used to execute the addition in one state and the subtraction in another. The state-action table after splitting ST3 is shown in Figure 29(b). The longest state delay, and the clock period in turn, is now 58.6 ns and the maximum execution time is increased from 277.2 to 293 ns.

Although the estimated maximum execution time still satisfies the performance constraint, the quality metric, execution time utilization, shows that only 52.3% of the execution time is being utilized by the components, that is, due to clock slacks, the components are idled during 47.4% of the execution time. That means, by splitting states to reduce the clock slacks, the maximum execution time can be improved. Noticing that states ST1 and ST4 are approximately twice as long as ST3, we split ST1 and ST4 and the resulting state-action table is shown in Figure 30. The execution time utilization is improved from 52.3% to 69.1% and the maximum execution time is reduced from 293 to 247.8 ns.

Now that according to the quality metrics, the performance and area constraints are both satisfied, we can proceed with the binding task. The operator binding is straight-forward since there are one component each for maximum, minimum, addition and subtraction, and two identical components for the computation of absolute values. The operator binding is shown in Figure 31(a). Variable binding requires us to determine the lifetimes of each variable since a register can be shared by those variables with non-overlapping lifetimes. Figure 30 shows the variable lifetime metric (Var. LT) of the current schedule. One of the common goals during variable binding is to try to have as few registers as possible. Figure 31(a) shows one possible variable binding which requires only four registers.

After the operator and variable bindings are done, the interconnections between components and registers can be automatically determined. Multiplexers are also automatically inserted at the input ports of the components and registers when they have more than one sources. The controller can also be generated. Figure 31(c) shows a complete netlist and floorplan of the current design. The total area of the design increases tremendously and the area constraint is now violated. Moreover, after including the wiring delay and multi-

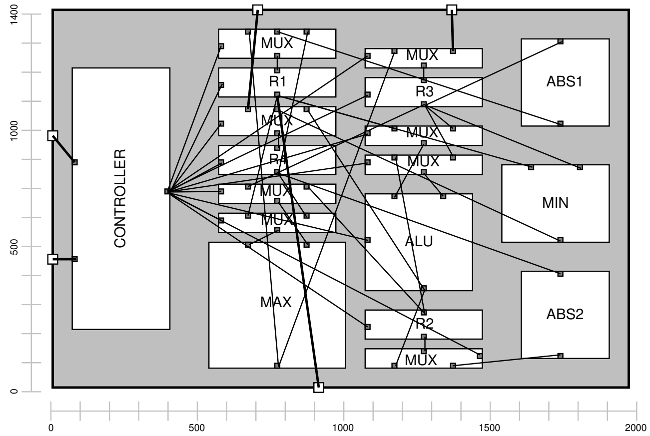
<table>
<tr><th>Name</th><th>Type</th><th>Bitwidth</th><th>Dealy</th><th>Area</th><th>Binding</th></tr>
<tr><td>ABS1</td><td rowspan="2">abs</td><td rowspan="2">16</td><td rowspan="2">23.3</td><td rowspan="2">149,472</td><td>|a|</td></tr>
<tr><td>ABS2</td><td>|b|</td></tr>
<tr><td>MAX</td><td>max</td><td>16</td><td>26.5</td><td>162,432</td><td>max(t1,t2), max(t6,x)</td></tr>
<tr><td>MIN</td><td>min</td><td>16</td><td>23.3</td><td>149,472</td><td>min(t1,t2)</td></tr>
<tr><td>ALU</td><td>alu</td><td>16</td><td>19.8</td><td>160,416</td><td>+, –</td></tr>
<tr><td>R1</td><td rowspan="4">reg</td><td rowspan="4">16</td><td rowspan="4">3.5/5.4</td><td rowspan="4">49,824</td><td>t1, t4, t7</td></tr>
<tr><td>R2</td><td>t2, t5</td></tr>
<tr><td>R3</td><td>x, a</td></tr>
<tr><td>R4</td><td>y, t6, b</td></tr>
</table>

(a)

<table>
<tr><th>PS</th><th>SCOND</th><th>NS</th><th>AC</th><th>ACTIONS</th><th>OP. OCC</th><th></th><th>ST Delay</th></tr>
<tr><td rowspan="2">ST0</td><td>!Start</td><td>ST0</td><td>T</td><td>a = In1</td><td></td><td></td><td rowspan="2">11.4</td></tr>
<tr><td>Start</td><td>ST1</td><td>T</td><td>b = In2</td><td></td><td></td></tr>
<tr><td rowspan="2">ST1</td><td rowspan="2">T</td><td rowspan="2">ST5</td><td>T</td><td>t1 = |a|</td><td rowspan="2">| |</td><td rowspan="2">2</td><td rowspan="2">38.1</td></tr>
<tr><td>T</td><td>t2 = |b|</td></tr>
<tr><td rowspan="2">ST5</td><td rowspan="2">T</td><td rowspan="2">ST3</td><td>T</td><td>x = max (t1,t2)</td><td>max</td><td>1</td><td rowspan="2">46.8</td></tr>
<tr><td>T</td><td>y = min (t1,t2)</td><td>min</td><td>1</td></tr>
<tr><td rowspan="3">ST3</td><td rowspan="3">T</td><td rowspan="3">ST4</td><td>T</td><td>t3 = x >>3</td><td>>>3</td><td>1</td><td rowspan="3">40.4</td></tr>
<tr><td>T</td><td>t4 = y >>1</td><td>>>1</td><td>1</td></tr>
<tr><td>T</td><td>t5 = x – t3</td><td>–</td><td>1</td></tr>
<tr><td>ST4</td><td>T</td><td>ST6</td><td>T</td><td>t6 = t4 + t5</td><td>+</td><td>1</td><td>40.4</td></tr>
<tr><td>ST6</td><td>T</td><td>ST2</td><td>T</td><td>t7 = max (t6,x)</td><td>max</td><td>1</td><td>47.1</td></tr>
<tr><td rowspan="2">ST2</td><td rowspan="2">T</td><td rowspan="2">ST0</td><td>T</td><td>Done = 1</td><td></td><td></td><td rowspan="2">5.4</td></tr>
<tr><td>T</td><td>Out = t7</td><td></td><td></td></tr>
</table>

max execution time = 47.1 x 7 = 329.7 (ns)

(b)



Total area:	2,660,000 um^2
F.U. area:	1,089,397 um^2
S.U. area:	198,916 um^2
I.U. area:	276,926 um^2
Routing area:	775,000 um^2
Wasted area:	319,761 um^2

(c)

Figure 31: The design of the SRA example after binding

plexer delay, etc., the maximum execution time is now estimated to be 329.7 ns, which also violates the performance constraint. Note that floorplanning at this early stage enables us to discover that both performance and area constraints are being violated. Without floorplanning, the total area can only be computed by summing up the functional unit area, storage unit area and the interconnection unit area. That gives an estimate of 1,565,239 μm^2, which is only 58.8% of the total area.

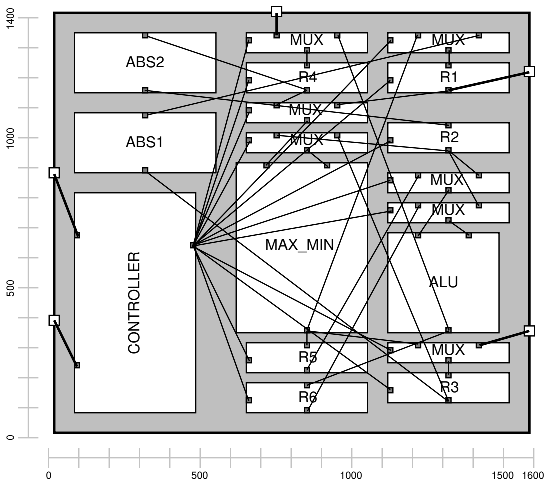
Name	Type	Bitwidth	Dealy	Area	Binding
ABS1	abs	16	25.5	123,886	|a|
ABS2					|b|
MAX_MIN	max_min	16	30.9	180,625	max(t1,t2), max(t6,x), min(t1,t2)
ALU	alu	16	19.8	160,416	+, –
R1	reg	16	3.5/5.4	49,824	t1, t7
R2					t2
R3					x, a
R4					t6, b
R5					t4
R6					t5

(a)

PS	SCOND	NS	AC	ACTIONS	OP. OCC		ST Delay
ST0	!Start	ST0	T	a = In1			11.1
	Start	ST1	T	b = In2			
ST1	T	ST5	T	t1 = |a|	| |	2	40.0
			T	t2 = |b|			
ST5	T	ST3	T	x = max(t1,t2)	max	1	46.8
ST3	T	ST4	T	y = min(t1,t2)	min	1	37.8
			T	t3 = x >>3	>>3	1	
			T	t4 = y >>1	>>1	1	
			T	t5 = x – t3	–	1	
ST4	T	ST6	T	t6 = t4 + t5	+	1	40.1
ST6	T	ST2	T	t7 = max(t6,x)	max	1	46.8
ST2	T	ST0	T	Done = 1			5.4
			T	Out = t7			

max execution time = 46.8 x 7 = 327.6 (ns)
execution time utilization = 69.6%

(b)



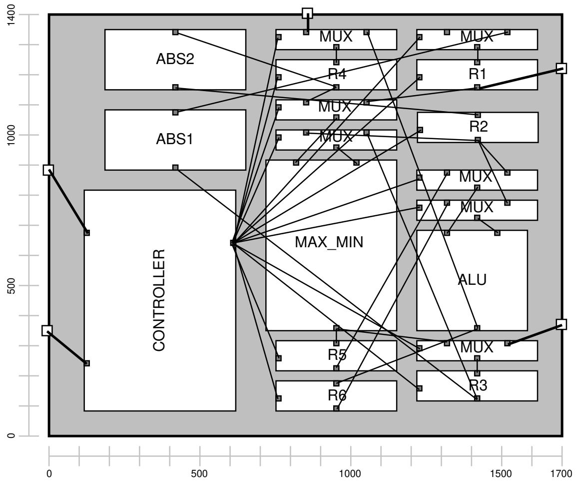
Total area:	2,240,000 um^2
F.U. area:	838,764 um^2
S.U. area:	298,944 um^2
I.U. area:	207,648 um^2
Routing area:	750,000 um^2
Wasted area:	144,640 um^2

(c)

Figure 32: The design of the SRA example after the second re-allocation

To reduce the area, we can go back to the architecture of the current design. Note that there exists one component which can perform both maximum and minimum. Also, there is one slower but smaller implementation of the components ABS1 and ABS2. Therefore, we can replace components MAX and MIN by a new component MAX_MIN, and use the smaller implementation for ABS1 and ABS2. The resulting allocation is shown in Figure 32(a).

However, the computation of minimum in ST5 needs to be moved to ST3 such that the component MAX_MIN can be used to execute both maximum and minimum sequentially. Figure 32(b) shows the state-action table after moving the computation of minimum to ST3. After re-allocation and re-scheduling, part of the operator and variable bindings, which are affected by the re-allocation and re-scheduling, need to be modified. Also, the controller needs to be re-generated. Figure 32(c) shows the final floorplan. The total area is now 2,240,000 μm^2, which satisfies the area constraint.



Total area:	2,380,000 um^2
F.U. area:	938,764 um^2
S.U. area:	298,944 um^2
I.U. area:	207,648 um^2
Routing area:	750,000 um^2
Wasted area:	184,640 um^2

Figure 33: The final design of the SRA example

However, the maximum execution time of the current design is 327.6 ns, which still violates the performance constraint. Note that the execution time utilization is only 69.6%, which implies that the maximum execution time can be further improved by reducing clock slacks. A simple computation tells us that if we split all the states except ST0 and ST2 into two states, the clock period will be reduced to 23.4 ns and the total number of states on the longest execution path will be increased to 12 states. Hence, the maximum execution time will be 280.8 ns, which satisfies the performance constraint. The splitting of states does not change any operator and variable bindings, only the controller needs to be re-generated.

Since there are more states in the new schedule, the controller becomes larger. The final floorplan is shown in Figure 33 and the design of this SRA example is completed with the total area 2,380,000 μm^2 and maximum execution time 280.8 ns.

7 Conclusion

This report details a design methodology for interactive behavioral synthesis. As opposed to the typical design methodology for automatic behavioral synthesis systems, the proposed methodology allows user decisions and user control in every task and at every level of the design process. Moreover, it allows the user to start floorplanning early in the design process. To demonstrate the design methodology, we also presented a walk-through square-root approximation example. Note that during the design process of this example, we utilized different quality metrics and made design improvements while working at behavioral, structural and even physical levels at the same time.

To realize the idea of interactive behavioral synthesis, we have implemented a system called ISE. An overview of ISE was also presented in this report. So far in ISE, we have developed and are continuing to develop new metrics that evaluate a partial or complete design and return some numerical value of its quality. What is missing at this moment is **bottleneck** metrics which will direct the designer's attention to congested areas at different levels of the design. These metrics are also important for developing design hints, which are procedures running in the environment background which compute design alternatives for the user that may help remove bottlenecks. The binding hint discussed in this report is an example of design hints. Therefore, the important future work would be to develop and implement bottleneck metrics and design hints.

8 References

[1] O. A. Buset, and M. I. Elmasry, "ACE: A Hierarchical Graphical Interface for Architectural Synthesis," *Proc. 26th DAC*, 1989.

[2] R. Camposano, and W. Wolf, *High-Level VLSI Synthesis,* Kluwer Academic Publishers, 1991.

[3] C. M. Chu, M. Potkonjak, M. Thaler, and J. Rabaey, "HYPER: An Interactive Synthesis Environment for High Performance Real Time Applications," *Proc. ICCD 89*, 1989.

[4] D. D. Gajski, N. Dutt, A. Wu, and S. Lin, *High-Level Synthesis: Introduction to Chip and System Design,* Kluwer Academic Publishers, 1992.

[5] D. D. Gajski, *Principles of Digital Design,* Prentice Hall, 1996.

[6] T. Hadley, *A System for Interactive High-Level Synthesis,* PhD Thesis, UC Irvine, 1995.

[7] P. Hilfinger, and J. Rabey, *Anatomy of a Silicon Compiler,* Kluwer Academic Publishers, 1992.

[8] A. Jerraya, I. Park, and K. O'Brien, "Amical: An Interactive High-Level Synthesis Environment," *Proc. EDAC 93*, 1993.

[9] D. W. Knapp, "An Interactive Tool for Register-Transfer Level Structure Optimization," *Proc. 26th DAC*, 1989.

[10] D. W. Knapp, "Manual Rescheduling and Incremental Repair of Register-Level Datapaths," *Proc. ICCAD 89*, 1989.

[11] D. E. Thomas, E. D. Langese, R. A. Walker, J.A. Nestor, J. V. Rajan, and R. L. Blackburn, *Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench,* Kluwer Academic Publishers, 1990.

[12] VLSI Technology, Inc. *VDP370 Datapath Element Library,* 1992.