Design Process and Human Interface for a Behavioral-Synthesis Environment

Daniel D. Gajski, Tedd Hadley,
Viraphol Chaiyakul, Tadatoshi Ishii

Technical Report 93-7
January 25, 1993

Dept. of Information and Computer Science
University of California, Irvine
Irvine, CA 92717
(714) 856-8059

Email: hadley@ics.uci.edu, viraphol@ics.uci.edu

Abstract

This report contains transparencies of a presentation on the design process in the University of California Irvine's Behavioral-Synthesis Environment. The human-interface aspects of the environment are discussed in detail and several design-process examples are given to demonstrate the power and usefulness of this environment for behavioral synthesis.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Goals 3

1.3 The Decision Support Environment 5

2 DSE Displays **7**

2.1 The State-Actions Table 7

2.2 The State-Actions Table: An Example 9

2.3 Component Selection and Allocation 11

2.4 The Connectivity Table and Floorplan 13

3 Displays for State-Based Quality-Measures and Statistics **15**

3.1 Delay 15

3.2 Occurrence 17

3.3 Lifetime 19

3.4 Utilization 21

4 User Scenarios **23**

4.1 Specify Table 23

4.2 Reducing behavioral operators 25

4.3 Reducing behavioral variables 27

4.4 Reducing clock period 29

4.5 Reducing execution time 31

4.6 Selection of type and quantity of component 33

4.7 Selection of implementation style 35

4.8 Selection of bitwidth for allocated components 37

4.9 Binding of functional and storage units 39

4.10 Allocating and binding of interconnect units 41

4.11 Final floorplanning 43

5 Conclusion **45**

6 References **47**

1 Introduction

This presentation describes a decision support environment (DSE) for behavioral synthesis.

1.1 Motivation

Two observations motivate the need for a decision support environment for behavioral synthesis:

1. Complete automation of the design process through every level of abstraction is not an immediately practical goal.
2. The human designer's insights into design strategy should be used to maximum effect in all phases of behavioral synthesis.

The first point is a conclusion derived more from the lack of observable uses of automated behavioral synthesis systems in commercial domains, rather than from the lack of existing systems in research and academia. Although it certainly can not be denied that progress has been considerable in this research area [Sh89] [DeRa86] [BrCa88], a practical solution to the problem of automating behavioral synthesis is still distant [CaWo91]. To develop a feasible approach to the problem, we have substituted the goal of a completely automated, "push-button" synthesis system with one which attempts to maximally utilize the human designer's methods and experience.

Decision Support Environment
Motivation

1. Complete automation of the design process through every level of abstraction is not an immediately practical goal.

2. The designer's insights into design strategy should be used to maximum effect in all phases of behavioral synthesis.

1.2 Goals

There are two primary goals in the development of our environment for decision support in behavioral synthesis:

1. *To allow user decisions and user control in every task of the design process.* These tasks are – but not limited to – unit selection: the proper hardware components are selected from a library; scheduling: the behavior is divided into timesteps according to resources and timing constraints; binding: behavioral operators and variables are bound to physical components; and floorplanning: layout and routing techniques are performed on the physical design representation ([GDWL92]).

2. *To provide rapid feedback of useable physical design characteristics and quality measures to every level of design abstraction.* Area and delay quality measures tell the designer whether or not the design meets space and time constraints. Critical path estimation reduces design development time by focusing the designer's attention on critical performance areas. Clock period estimation is used in a variety of ways to provide the designer with the necessary information for decisions in scheduling, binding, and floorplanning.

Decisions made by the user should generate immediate feedback as to the quality and functionality of the resulting design. As the design process progresses to more detailed and less abstract descriptions, the designer's experience must be utilized to make the kinds of decisions no automated tool can perceive or predict.

Decision Support Environment
Goals

1. To allow user decisions and user control throughout every phase of the design process.

   - Unit selection
   - Scheduling
   - Binding
   - Floorplanning

2. To provide rapid feedback of useable design characteristics and quality measures to the user at every level of design abstraction.

   - Area
   - Delay
   - Critical path
   - Clock period

1.3 The Decision Support Environment

The DSE supports three basic levels of interaction:

1. Graphical design capture, whereby the designer enters and/or modifies the desired behavior or structure with the aid of multiple design views,

2. Feedback of quality measures and design hints automatically derived by the system from the current design

3. The ability to incorporate automatic synthesis tools into the design process so that tedious, well-understood problems can be rapidly solved by automatic algorithms

A Decision Support Environment for Behavioral Synthesis

1. Graphical design capture
2. Quality measures and hints
3. Combined interactive and automatic synthesis

2 DSE Displays

2.1 The State-Actions Table

The state-actions table displays the behavior and schedule of the design in a tabular format. A general definition of each column in the table follows (A more detailed description of the state-actions table can be found in [HaCG93]):

1. PS is the present state.
2. SCOND gives the condition for a next-state transition.
3. NS is the next state. The condition may be any valid expression resulting in a boolean result. The condition may be any valid expression resulting in a boolean value.
4. ACOND shows the assignment condition for each action. The condition may be any valid expression resulting in a boolean value.
5. ACTIONS lists all operations in the behavior.
6. ORDER gives a level ordering of the actions within a given state and may be used by the user to explicitly indicate a dependency between actions.
7. CV lists the condition vector under which the results of the action will be used.
8. The AC # field assigns numbers to actions and is needed to make actions easier to reference in other fields.
9. DEPENDENCY (*INP DEP/OUT DEP*) field describes global dependencies among actions in terms of the numbers given in the *AC #* field. Numbers to the left of the / list the input dependencies (writes) of the operands in the corresponding action. Numbers to the right of the / list the output dependencies (reads) of the action's output variable.

The PS, SCOND, NS, ACOND, ACTIONS, and ORDER fields are user-specified through use of the menus shown below the table. The remaining fields are automatically derived by the system once the behavior is complete. The *QM/stats* field represents different quality measures, statics, and metrics that will be described in more detail later on.

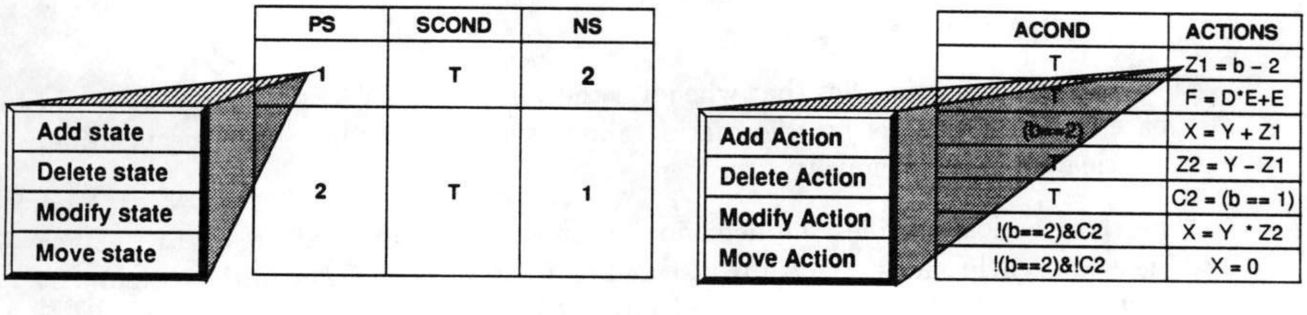
The state-actions graphical display is user-configurable in two ways.

1. Any column may be turned off at the user's discretion; for example, if screen-space limitations are a concern.
2. Any column may be interchanged with any other column

State–Actions Table

User Input | **System Derived**

PS	SCOND	NS	ACOND	ACTIONS	ORDER	CV	AC #	INP DEP/OUT DEP	*QM/stats*
1	T	2	T	Z1 = b − 2	0	(b==2)|!(b==2) & C2	1	/ 3,4	
			T	F = D*E+E	0	T	2	/	
2	T	1	(b==2)	X = Y + Z1	0	(b==2)	3	1 /	
			T	Z2 = Y − Z1	0	!(b==2)	4	1 / 6	
			T	C2 = (b == 1)	0	!(b==2)	5	/	
			!(b==2)&C2	X = Y * Z2	1	T	6	4 /	
			!(b==2)&!C2	X = 0	1	T	7	/	
									total:



PS	SCOND	NS
1	T	2
2	T	1

ACOND	ACTIONS
T	Z1 = b − 2
T	F = D*E+E
(b==2)	X = Y + Z1
T	Z2 = Y − Z1
T	C2 = (b == 1)
!(b==2)&C2	X = Y * Z2
!(b==2)&!C2	X = 0

2.2 The State-Actions Table: An Example

To get a feeling for how a behavioral description is specified in the state-actions table, let us look at a description of Armstrong's controlled counter. This is a 4-bit counter that consists of five main logic blocks: a 2-bit control register ($CONR$), a decoder, a 4-bit limit register (LIM), a 4-bit comparator, and a 4-bit counter (CNT). On the rising edge of the $STRB$ signal, the counter stores the 2-bit control input signal CON. It then performs the following operations based on the value of the control signal:

1. Clear the counter (CON = "00").
2. Load inputs from the $DATA$ signals into the LIM register on the falling edge of the $STRB$ signal (CON ="01").
3. Increment the count on the rising edge of the CLK signal (CON ="10").
4. Decrement the count on the rising edge of the CLK signal (CON = "11").

The behavior of the controlled counter consists of both synchronous and asynchronous parts. The latching of storage units in the synchronous part is performed during the rising edge of the CLK signal. The asynchronous part is performed on the rising and falling edges of the $STRB$ signal.

The first action in the state-table specifies the latching of the control input CON into a 2-bit control register $CONR$ on the rising edge of the $STRB$ signal. Since $CONR$ is clocked with the $STRB$ signal instead of the CLK signal, $CONR$ acts as the asynchronous part of the design.

The second action specifies that whenever the value of $CONR$ is "00", CNT should be cleared. Since there is no information about synchronization in the specification, CNT is cleared asynchronously.

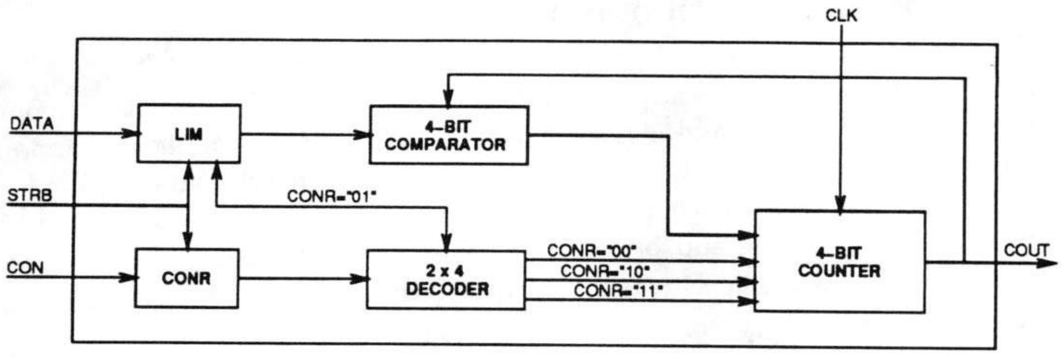
The third action specifies the behavior of the counter when $CONR$ is "01". Since LIM is also asynchronous, $DATA$ is assigned to LIM without CLK in the assignment condition. Instead, the value is latched into LIM on the falling edge of the $STRB$ signal.

The fourth action specifies the behavior of the counter when $CONR$ is "10". The counter counts up by incrementing the value of CNT on the rising edge of the CLK signal. The increment is performed only if the limit LIM is not reached.

The fifth action specifies the behavior of the counter when $CONR$ equals "11". The counter counts down by decrementing the value of CNT on the rising edge of the CLK signal. The decrement is performed only if the limit LIM is not reached.

The last action specifies the output of the counter $COUT$. The value of CNT is used for this output.

Example: 4–bit Controlled counter (Armstrong)



PS	SCOND	NS	ACOND	ACTIONS	ORDER
ST1	...	...	STRB'rising	CONR = CON	0
			CONR = "00"	CNT = "0000"	0
			STRB'falling and CONR = "01"	LIM = DATA	0
			CLK'rising and CONR = "10 and CNT /= LIM	CNT = CNT + "0001"	0
			CLK'rising and CONR = "11" and CNT /= LIM	CNT = CNT – "0001"	0
			T	COUT = CNT	0

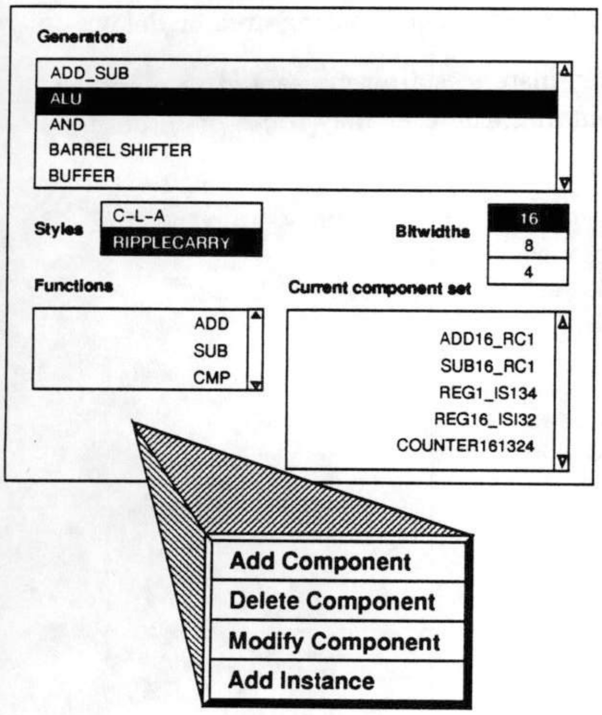
2.3 Component Selection and Allocation

The component selection and allocation display allows the designer to select components from a component library and add instances of those to the current design's component set. The **Selection** window lists the available component categories and the kinds and ranges of the parameters for each component. The designer must select parameters values, such as bitwidth, style and/or kinds of functions performed, in order to specify a unique component type. The **Current component set** window lists all component types currently in use in the design.

The allocation window lists all component instances physically present in the target design. Each instance is broken down into categories (type, unit, width), and the number of instances in each category is displayed in parentheses. Each instance may also be selected to display the list of function performed, pin-to-pin delay, and area information.

Component Selection and Allocation

Selection



Generators

ADD_SUB
[✓ ALU]
AND
BARREL SHIFTER
BUFFER

Styles C-L-A [✓ RIPPLECARRY]

Bitwidths [✓ 16] 8 4

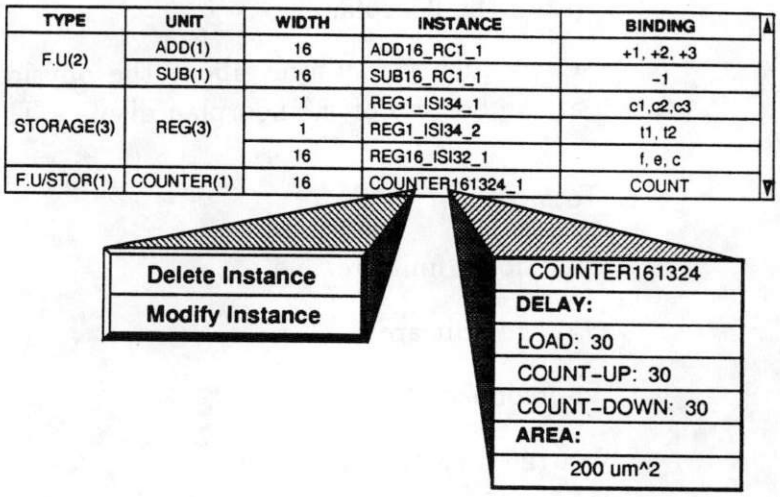
Functions

ADD
SUB
CMP

Current component set

ADD16_RC1
SUB16_RC1
REG1_IS134
REG16_ISI32
COUNTER161324

Allocation



TYPE	UNIT	WIDTH	INSTANCE	BINDING
F.U(2)	ADD(1)	16	ADD16_RC1_1	+1, +2, +3
	SUB(1)	16	SUB16_RC1_1	−1
STORAGE(3)	REG(3)	1	REG1_ISI34_1	c1,c2,c3
		1	REG1_ISI34_2	t1, t2
		16	REG16_ISI32_1	f, e, c
F.U/STOR(1)	COUNTER(1)	16	COUNTER161324_1	COUNT

COUNTER161324
DELAY:
LOAD: 30
COUNT-UP: 30
COUNT-DOWN: 30
AREA:
200 um^2

2.4 The Connectivity Table and Floorplan

The connectivity table lists component instances along the Y axis and buses (interconnections) along the X axis. A connection between an instance and a bus exists if an instance's input or output port is present in the corresponding entry. The user may add, delete, or reassign connections in order to optimize the number of interconnections needed.

The floorplan display shows the floorplan of the target design. When an instance is allocated, its layout is positioned randomly within the display initially. Afterwards, the user may reposition the instance anywhere in the design within design rule limits. Rotating an instance's layout in 90 degree increments is also permitted. The user may also relocate external i/o pads along the chip boundary to reduce routing area or delay.

During the binding process, connections between instances are represented as point-to-point connections within the floorplan. After binding, the user may route or reroute wires within the floorplan.

A set of quality measures about the physical design are constantly updated and shown at the bottom of the floorplan display. These are

1. Total area
2. Functional unit area
3. Storage unit area
4. Interconnect unit area
5. Wasted area (white space)
6. The length of the critical path
7. Total wire length

The four displays described, the state-actions table, the component selection and allocation display, the connectivity table, and the floorplan display, are integrated tightly together such that actions or highlighting in one display is reflected (if relevant) in the others. For example, if the critical state is highlighted in the state-actions table (next slide), the floorplan display will highlight the components and connections involved.

The next section describes the state-based quality-measures and statistics available to the designer from the state-actions display.

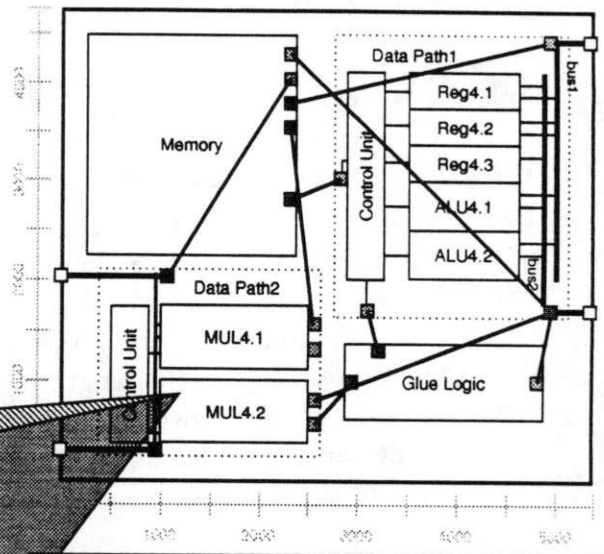
Connectivity Table and Floorplan

Connectivity Table

Instances \ Buses	b1	b2	b3	b4
Memory1	I1	I2	O2	O1
REG4.1		O	I	
REG4.2	O		I	
REG4.3	O		I	
ALU4.1		LI	RI	O
ALU4.2	RI	LI	O	

- Add Connection
- Delete Connection
- Reassign Connection

Floorplan



- Place Component
- Rotate Component
- Position Ports
- Position I/O Pads
- Route Wire

Total area:	5550 um^2	Wasted area:	1150 um^2
F.U. area:	2250 um^2	Critical path wire length:	542 um
S.U. area:	1250 um^2	Wire length:	1247 um
I.U. area:	900 um^2		

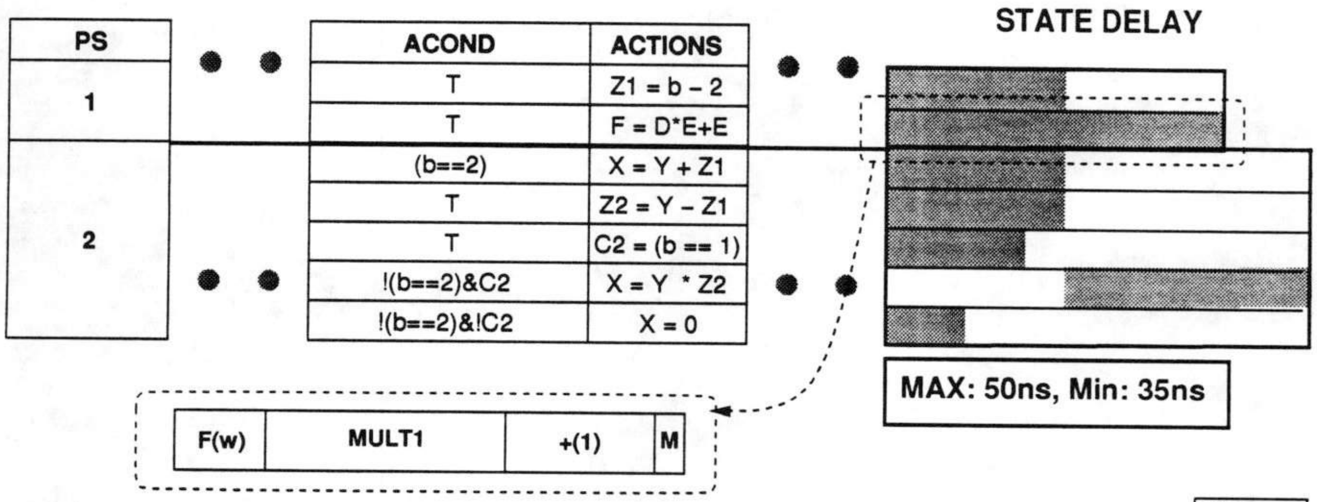
3 Displays for State-Based Quality-Measures and Statistics

3.1 Delay

The *DELAY* menu causes a bar graph to be displayed alongside the state table showing information about state delay. Delay is represented graphically along the X axis, and delay statistics are shown below the graph. The thick-lined outer boxes show the maximum delay of each state. Within each box, the delay is further broken down by actions. For example, in state 1 the action `F=D*E+E` is the longest path, so the width of its shaded box extends the length of the thick-lined state-delay box. Each action can also be further broken down by operator and operand delays if the user requests. For example, the same action is shown at the bottom of the slide with the delay for writing `F`, the delay of the multiply operation, and the delay of the addition shown.

State-Based Quality-Measures and Statistics

DELAY
OCCURANCE
LIFETIME
UTILIZATION

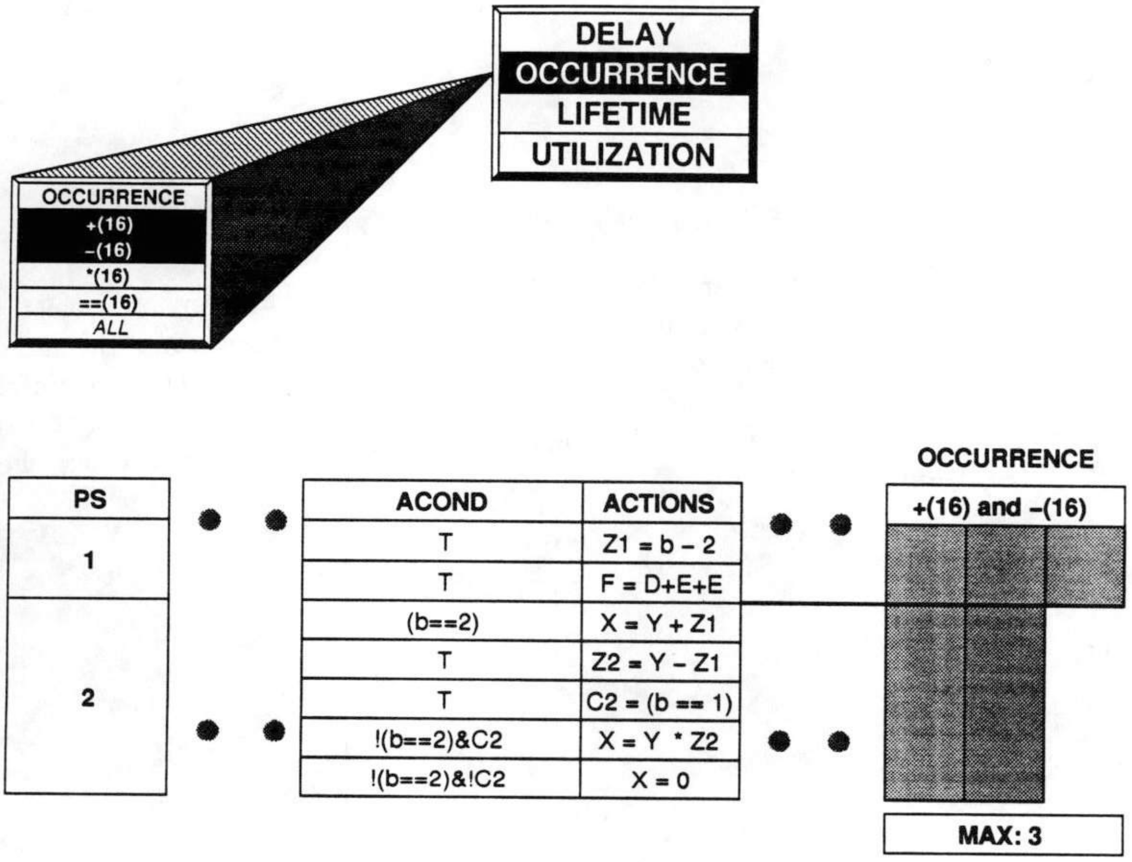


PS	ACOND	ACTIONS
1	T	Z1 = b - 2
	T	F = D*E+E
2	(b==2)	X = Y + Z1
	T	Z2 = Y - Z1
	T	C2 = (b == 1)
	!(b==2)&C2	X = Y * Z2
	!(b==2)&!C2	X = 0

3.2 Occurrence

The *OCCURRENCE* menu lists operators along with their bitwidths and allows the user to select any subset to be displayed graphically. The slide shows the bar graph resulting from selecting "`+(16)`" (all occurrences of a 16-bit `+` operation) and `-(16)` (all occurrences of a 16-bit "`-`" operation).

State-Based Quality-Measures and Statistics

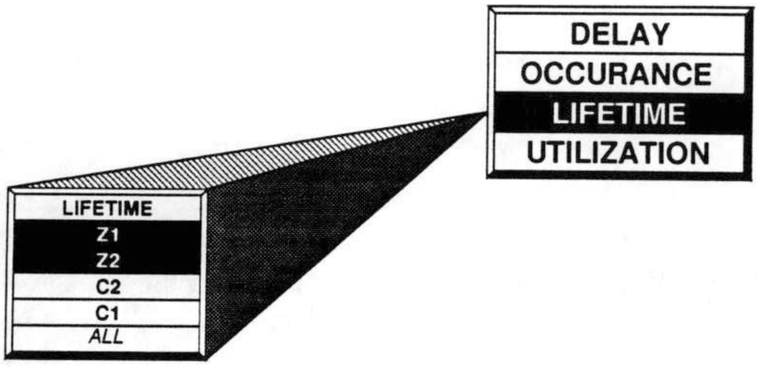


PS	ACOND	ACTIONS
1	T	Z1 = b - 2
	T	F = D+E+E
2	(b==2)	X = Y + Z1
	T	Z2 = Y - Z1
	T	C2 = (b == 1)
	!(b==2)&C2	X = Y * Z2
	!(b==2)&!C2	X = 0

3.3 Lifetime

The *LIFETIME* menu lists the variables used in the behavior. The user may select a subset of these in order to display a lifetime graph, as shown at the bottom of the slide. The lifetime of a variable is defined as all states in which the variable holds a useful value. The final state in which a variable is read is not included in the set of states making up its lifetime. The graph is further annotated to indicate the actions where the variable is read (`R`) or written (`W`).

State-Based Quality-Measures and Statistics

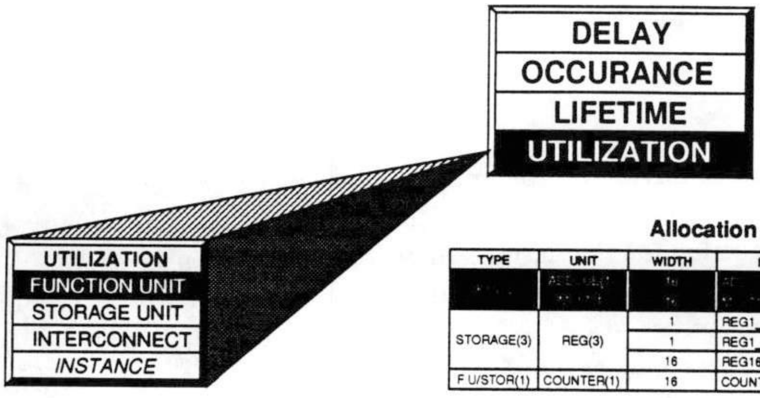


					LIFETIME	
PS	● ●	ACOND	ACTIONS	● ●	Z1	Z2
1		T	Z1 = b - 2		W	
		T	F = D+E+E			
2		(b==2)	X = Y + Z1		R	
		T	Z2 = Y - Z1		R	W
3		T	C2 = (b == 1)			
	● ●	!(b==2)&C2	X = Y * Z2	● ●		R
		!(b==2)&!C2	X = 0			

3.4 Utilization

The *UTILIZATION* menu requires the user to a select a component type (functional, storage, interconnect) or a specific component instance. Selecting *INSTANCE* from the utilization menu prompts the user to select a specific component instance from the Allocation Table. In this example, since `FUNCTION UNIT` was selected, all instances of type function-unit are selected. The utilization graph shows the states in which those instances are active. At the bottom of the graph a utilization percentage is shown, defined as the number of states in which the instance is active divided by the total number of states.

State-Based Quality-Measures and Statistics



Allocation Table

TYPE	UNIT	WIDTH	INSTANCE	BINDING
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
STORAGE(3)	REG(3)	1	REG1_ISI34_1	c1,c2,c3
		1	REG1_ISI34_2	t1, t2
		16	REG16_ISI32_1	f, e, c
F U/STOR(1)	COUNTER(1)	16	COUNTER161324_1	COUNT

UTILIZATION

PS		ACOND	ACTIONS		ADDSUB1	MULT1
1	● ●	T	Z1 = b − 2	● ●	▨	▨
		T	F = D*E+E			
2		(b==2)	X = Y + Z1		▨	
		T	Z2 = Y − Z1			
3	● ●	T	C2 = (b == 1)	● ●		▨
		!(b==2)&C2	X = Y * Z2			
		!(b==2)&!C2	X = 0			
					66 %	66 %

4 User Scenarios

This section discuses the use of quality metrics and design views in the DSE to capture and optimize an intended design. Instead of browsing through all quality metrics and design views offered by the DSE, let us consider a typical design scenario and discuss views and quality metrics that can be useful in each step of the scenario.

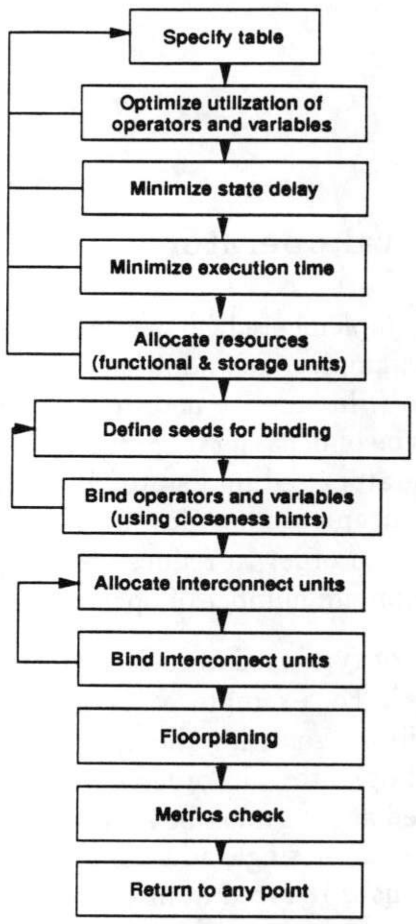
The example scenario consists of the following steps: specify table, optimize utilization of operators and variables, minimize state delay, minimize execution time, allocate functional and storage unit resources, bind functional and storage units, allocate interconnect units, bind interconnect units, finalize floorplanning, check the quality metrics and repeat any steps if necessary. Each step in the scenario is described in detail in the following sections.

4.1 Specify Table

The first step is to specify the behavior of the intended hardware in the form of a state table. The state table is constructed initially without considering the amount of resources that are available or needed for the design. In other words, the user should not pay attention to the amount of resources that are required in each state or to the amount of resources that can be shared across states. Thus, the initial description generally contains a small number of states. States are introduced only when the sequencing of control in the design is needed. For example, in a control dominated or interface circuit, states are used to control the sequencing of hand-shaking events. In contrast, states should not be introduced when specifying algorithmic behaviors, such as filters $Y_i = \sum_i^k a_i X_i$.

Once the state table is completely specified or at least the crucial part of the design has been specified, the next step is to refine or reschedule the state table to minimize required hardware resources and shorten state delays. At this early stage in the design scenario only behavioral information is available. Nonetheless, 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 memory. Thus, reducing the number of operators and variables implies that the required hardware will likely be reduced. Similarly, reducing the number of operators per state shortens state delays and the clock period in turn.

Typical Scenario


Specify table
Optimize utilization of operators and variables
Minimize state delay
Minimize execution time
Allocate resources (functional & storage units)
Define seeds for binding
Bind operators and variables (using closeness hints)
Allocate interconnect units
Bind interconnect units
Floorplaning
Metrics check
Return to any point

4.2 Reducing behavioral operators

A behavioral operator can be implemented in hardware with a functional unit that performs that operation. For example, each addition operator used in the state table can be implemented with an ALU. In order to determine the total number of functional units required to implement a type of behavioral operator, we have to find the maximum number of instances of that operator used in a state. By taking the maximum number of instances we are assuming that operators which are used in different state can share the same functional unit. Hence, in order to reduce the required number of functional units we have to reduce the maximum number of operators used in a state.

The number of a given operator type used in each state can be found from the quality metric "Occurrence of operators". For example, a maximum number of 4 additions are performed in state $ST1$. In addition, selecting the metric also highlights the actions in each state that use the selected operator. In order to reduce the number of operators used in each state, the highlighted actions must be moved to new states. In the example, three new states have been added and a single addition action moved to each one. Thus, the maximum number of additions is reduced from 4 to 1.

Reducing Behavioral operators

Reducing occ. of operators => Reducing functional unit resources

PS	ORDER	ACOND	ACTION	Occ. Addition
ST1	0	T	X1 = I1 + I2	4
	0	RESET = '1'	X2 = I2	
	1	$\overline{C}$	O1 = X1 * I2	
	0	$\overline{C}$	X3 = I1 + I3	
	1	C	O2 = I2 * X2	
	1	C	O3 = X1 + X2	
	1	T	X2 = X3 + I2	

Reschedule

(a)

MAX: 4

PS	SCOND	NS	ORDER	ACOND	ACTION	Occ. Addition	Occ. Mult.
ST1	T	ST2	0	T	X1 = I1 + I2	1	1
			0	RESET = '1'	X2 = I2		
			1	$\overline{C}$	O1 = X1 * I2		
ST2	T	ST3	0	$\overline{C}$	X3 = I1 + I3	1	1
			0	C	O2 = I2 * X2		
ST3	T	ST4	0	C	O3 = X1 + X2	1	
ST4	T	ST1	0	T	X2 = X3 + I2	1	

(b))

MAX: 1 MAX: 1

4.3 Reducing behavioral variables

Behavioral variables are generally implemented as storage units. In the implementation, variables that have non-overlapping life time 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 life time overlapping occurs. To aid minimizing storage units, the DSE offers the "Lifetime of variables" quality metric to identify states in which a selected variable is "alive" (i.e., states where the variable holds useful data). For example, a state table with the lifetimes of variables $X1$, $X2$ and $X3$ is shown. In addition, the example also show the "Union" lifetime of the three variables. The values shown in the "Union" lifetime graph are basically the sum of the lifetime values of each of the variables. Thus, values shown in the union metric suggest possible mergings of variables that have non-overlapping lifetime. The maximum value of the union metric implies the minimum number of registers required to implement all variables in the design. Thus, the example would required 3 registers.

To reduce the number of storage units that are used to implement variables, we reschedule by swapping the action $O3 = X1 + X2$ in state $ST3$ with the action $X3 = I1 + I3$ in $ST2$. The rescheduling reduces the lifetimes of variables $X2$ and $X3$ such that it is possible to merge variables $X2$ and $X3$. We can perform the merging by replacing variables $X2$ and $X3$ with a new variable $X4$. Hence, the maximum value of the union metric is reduced from 3 to 2.

Reducing Behavioral Variables Usage

Reducing max. life all vars => Reducing registers

PS	SCOND	NS	ACOND	ACTION	Life X1	Life X2	Life X3	U(X1,X2,X3)
ST1	T	ST2	T	X1 = I1 + I2	W	W		W
			RESET = '1'	X2 = I2	1	1		W 2
			$\overline{C}$	O1 = X1 * I2	R			
ST2	T	ST3	$\overline{C}$	X3 = I1 + I3		R	W	W
			C	O2 = I2 * X2	1	1	1	R 3
ST3	T	ST4	C	O3 = X1 + X2	R	R	1	R R 1
ST4	T	ST1	T	X2 = X3 + I2		W 1	R	W R 1

(a)

MAX: 3

Reschedule

PS	SCOND	NS	ACOND	ACTION	Life X1	Life X4	U(X1,X4)
ST1	T	ST2	T	X1 = I1 + I2	W	W	W
			RESET = '1'	X4 = I2	1	1	W 2
			$\overline{C}$	O1 = X1 * I2	R		
ST2	T	ST3	C	O3 = X1 + X4	R	R R	R R R
			C	O2 = I2 * X4			
ST3	T	ST4	$\overline{C}$	X4 = I1 + I3		W 1	W 1
ST4	T	ST1	T	X4 = X4 + I2		R W 1	R W 1

(b)

MAX: 2

4.4 Reducing clock period

The performance of a synchronous design is proportional to the clock period, where the clock period is defined as the maximum time needed to execute a state. The DSE provides a quality metric to measure the clock period, the "Delay per state" metric. Each value is the time needed to completely execute all actions in that state. Since at this stage of the scenario only behavioral information is available, delay information is estimated. An example state table with the quality metric "Delay per state" is shown. The delay of a multiplier is assumed to be 40 ns, the delay of an adder is assumed to be 20ns and the setup time of a register is assumed to be 5 ns. Each value of the "Delay per state" is estimated as the sum of the operator delays and the register setup time. For example, the delay of $ST1$ is estimated as 65 ns due to the chaining of a multiplication (40 ns.), an addition (20 ns.) and the register setup time (5 ns.). Since this state has the maximum value of the "Delay per state" metric, the clock period of the design is also 65ns.

In addition to displaying delay per state, the DSE will also highlight actions that determine the delay of a selected state. For example, the two chained actions in state $ST1$ requiring a total delay time of 65 ns are shown. We can reduce this delay by moving the action ($O1 = X1 * X2$) to state $ST3$. By doing so, the "Delay per state" of state $ST1$ is reduced to 25 ns, since only addition is now performed in this state. On the other hand, the delay of $ST3$ increases from 25 ns. to 45 ns. which is the maximum of the addition and multiplication delays since both operators are performed concurrently. The maximum value of "Delay per state" for the new schedule is reduced from 65 ns. to 45 ns.; thus, the new clock period is 45 ns.

Reducing Clock Period

Reducing max. Delay/state => Reducing clock period

PS	SCOND	NS	ACOND	ACTION	Delay/state
ST1	T	ST2	T	[1] X1 = I1 + I2	1 2 65
			RESET = '1'	X2 = I2	
			$\overline{C}$	[2] O1 = X1 * I2	
ST2	T	ST3	C	O3 = X1 + X2	45
			C	O2 = I2 * X2	
ST3	T	ST4	$\overline{C}$	X3 = I1 + I3	25
ST4	T	ST1	T	X2 = X3 + I2	25

Reschedule

(a)

MAX: 65

PS	SCOND	NS	ACOND	ACTION	Delay/state
ST1	T	ST2	T	X1 = I1 + I2	25
			RESET = '1'	X2 = I2	
ST2	T	ST3	C	O3 = X1 + X2	45
			C	O2 = I2 * X2	
ST3	T	ST4	$\overline{C}$	O1 = X1 * I2	45
			$\overline{C}$	X3 = I1 + I3	
ST4	T	ST1	T	X2 = X3 + I2	25

(b)

MAX: 45

4.5 Reducing execution time

The total execution time of a design is the time required to execute all states that lie on the critical path from the starting state to the final state. Thus, one way to reduce execution time is to reduce the number of states on the critical path. To accomplish this, first we have to identify states that are on the critical path by using the quality metric "Critical execution path". An example of a state table with the "Critical execution path" metric is shown.

The critical execution path of this example consists of four states that are sequential executed one after another. Since this is a synchronous design, the state transition is assumed to be performed at the rising edge of every clock. Thus, if the design is in state $ST1$ then on the rising edge of the clock the new state will be $ST2$. In $ST2$, if the condition C is false then none of the actions executed will be assigned to storage units. Hence, all actions performed in $ST2$, when C is false, are useless. Therefore, if C is false we should skip state $ST2$ by jumping from $ST1$ directly to $ST3$. Thus, we can reschedule the state by changing the transition from $ST1$ to $ST2$ only when C is true; otherwise $ST1$ transits to $ST3$ when C is false. The result of the rescheduling is shown. The rescheduling changes the critical execution path from 4 states to 3 states, and the total execution time has been reduced from 120 ns. to 90 ns.

Reducing Execution Time

Reducing max. # states on critical execution path => Reducing execution time

PS	SCOND	NS	ACOND	ACTION	Execution path
ST1	T	ST2	T	X1 = I1 + I2	1
			RESET = '1'	X2 = I2	
ST2	T	ST3	C	O3 = X1 + X2	1
			C	O2 = I2 * X2	
ST3	T	ST4	$\overline{C}$	O1 = X1 * I2	1
			$\overline{C}$	X3 = I1 + I3	
ST4	T	ST1	T	X2 = X3 + I2	1

MAX: 120 ns

Reschedule

PS	SCOND	NS	ACOND	ACTION	Execution path
ST1	C	ST2	T	X1 = I1 + I2	1
	$\overline{C}$	ST3	RESET = '1'	X2 = I2	
ST2	T	ST4	T	O3 = X1 + X2	
			T	O2 = I2 * X2	
ST3	T	ST4	T	O1 = X1 * I2	1
			T	X3 = I1 + I3	
ST4	T	ST1	T	X2 = X3 + I2	1

MAX: 90ns

4.6 Selection of type and quantity of component

Having a scheduled state table that satisfies resources constraints, the next step is to allocate hardware components that can be used to implement the operators and variables in the description. The first task in allocation is to determine the type and quantity of such components. We can determine the type and quantity of functional units by using the occurrence of operators metric. To determine the number of storage units we can use the variable lifetime metric. The example shows a scheduled state table with the life times of all variables, occurrences of addition and occurrences of multiplication quality metics. The maximum value of the life times of all variables is 2. Thus, the design would required 2 registers to implement. The maximum occurrence of additions is 1, hence, 1 adder is required. Similarly, the maximum occurrence of multiplications is 1, which means 1 multiplier is required.

Selection of Type and Quantity of Component

PS	SCOND	NS	ACOND	ACTION	U. Life all vars	Occ. Addition	Occ. Mult.
ST1	C	ST2	T	X1 = I1 + I2	W		
	$\overline{C}$	ST3	RESET = '1'	X2 = I2	W 2	1	
ST2	T	ST4	T	O3 = X1 + X2	R		
			T	O2 = I2 * X2	R 2	1	1
ST3	T	ST4	T	O1 = X1 * I2	W		
			T	X3 = I1 + I3	R 2	1	1
ST4	T	ST1	T	X2 = X3 + I2	W R 1	1	
					MAX: 2	MAX: 1	MAX: 1

2 Registers, 1 Adder, 1 Multiplier

4.7 Selection of implementation style

The next task in allocation is to select the implementation style for each component. The selection might depend on the area and/or the performance of the design. The example shows a selection of implementation style for the adder and the multiplier based on the performance of the design. Since the clock period is constrained by the delay of the multiplier, we can reduce the clock period by selecting either a high-speed, a multi-clock, or a pipelined implementation for it. In this example we select a two-stage pipelined implementation for the multiplier because the outputs of the multiplications, $O2$ in state $ST2$ and $O1$ in state $ST3$, are not used in the immediately successive states. Addition, by comparison, is not a time consuming operation, and we can select a slow implementation for the adder. In this example we select a ripple implementation for the adder.

Selection of Implementation Style

PS	SCOND	NS	ACOND	ACTION	Delay/state
	C	ST2	T	X1 = I1 + I2	25
ST1	$\overline{C}$	ST3	RESET = '1'	X2 = I2	
ST2	T	ST4	T	O3 = X1 + X2	45
			T	O2 = I2 * X2	
ST3	T	ST4	T	O1 = X1 * I2	45
			T	X3 = I1 + I3	
ST4	T	ST1	T	X2 = X3 + I2	25

MAX: 45

Fast multiplier (e.g., 2 statges–pipelined multiplier)

Slow adder (e.g., Ripple adder)

4.8 Selection of bitwidth for allocated components

At this point in the design scenario we have allocated components and selected component types and implementation styles. The next task is to determine the bitwidth of each allocated component. For each allocated component we can determine the bitwidth of each operator by using the "Occurrence of units with given bitwidth" metric. The example shows a 3-state design with 6 addition operations. The bitwidth of each addition operation is shown in parenthesis. The metric shows that there are two 8-bit additions performed in state $ST1$ and $ST3$ and one 6-bit addition performed in $ST2$. Thus, using an 8-bit adder, the two 8-bit additions and the 6-bit addition can be performed with the same 8-bit adder. The same procedure can be used for the allocation of a 4-bit adder. Hence, the resulting allocation for this example would be one 8-bit adder and one 4-bit adder.

Selection of Bitwidth for Allocated Components

Bitwidth

		8 bit addition	6 bit addition	4 bit addition	2 bit addition
ST1	A = B +(8) C D = E +(2) F	1		1	
ST2	G = H +(4) I J = A +(6) D		1		1
ST3	K = L +(8) M N = O +(4) P	1		1	

Resources: one 8-bit adder, one 4-bit adder

4.9 Binding of functional and storage units

Binding can be performed after each or all components have been allocated. In addition to the quality metrics that have been discussed earlier (e.g., Occurrence of operators, Life time of variables, etc.), the DSE also offers hints for binding operators and variables to allocated components. These hints suggest the next operator or variable to be bound, or suggest different binding configurations that can improve the design cost. The binding hints can be divided into two category: hints for constructive binding and hints for iterative improvement in binding.

Hints for constructive binding are used to select a particular unbound operator or a variable. To use these hints, the user selects as a seed a component or a register that is newly allocated or that already has some bound operators/variables. The user then asks the DSE to suggest an unbound operator/variable that bears a "closeness" to the selected seed. There are three kinds of closeness measures available for constructive binding:

1. **Closeness in bitwidth**

   The DSE will select an unbound operator/variable that has a bitwidth less then or equal to the selected seed. The example shows an 8-bit $ALU1$ seed that has pre-bound 8-bit operators $+_1$ and $+_2$. The DSE will suggest $+_4$ as the next operator to be bound to $ALU1$.

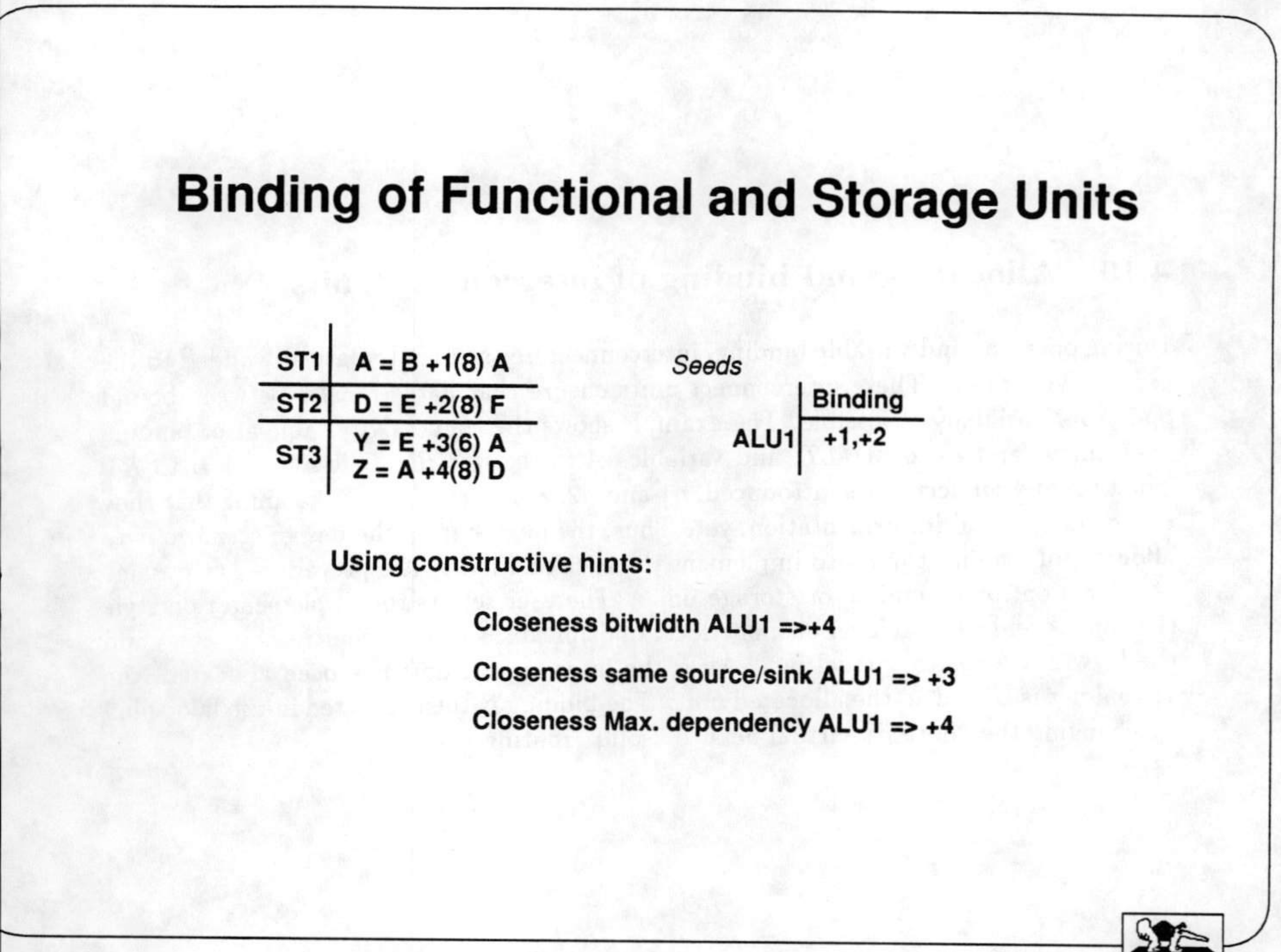
2. **Closeness in sources and sinks**

   This measure selects an operator/variable that has the most number of common sources or common sinks as operators/variables that have been previously bound to the seed component. Such a binding should reduce the number of interconnect units. The example shows that using the closeness in sources and sinks measure, the DSE will select $+_3$ as the hint for the next binding. This is because $+_3$ has two common sources with $+_1$ (A) and $+_2$ (E).

3. **Closeness in dependencies**

   Closeness in dependencies is defined as the number of dependency edges between an operator/variable and operators/variables that are bound to the selected component. Let us consider the data flow graph of the example state table. $+_3$ has one dependency on $ALU1$ from $+_1$, while $+_4$ has two dependencies, which are due to $+_1$ and $+_2$. Thus, the closeness in dependencies measure will suggest $+_4$ as the next binding candidate for $ALU1$.

In addition to constructive binding hints, the DSE also provide hints for improving the cost of the design by changing the binding of operators or variables. There are three kind of design costs that can be improved, namely, gain in the number of interconnect units, gain in the number of drivers on an interconnect unit, and gain in the number of routing tracks. When one of these three is selected the DSE will suggest an alternate binding for a set of operators/variables and their corresponding functional/storage units that will improve the selected design cost.



Binding of Functional and Storage Units

ST1	A = B +1(8) A
ST2	D = E +2(8) F

ST3	Y = E +3(6) A
	Z = A +4(8) D

Seeds

	Binding
ALU1	+1,+2

Using constructive hints:

Closeness bitwidth ALU1 =>+4

Closeness same source/sink ALU1 => +3

Closeness Max. dependency ALU1 => +4

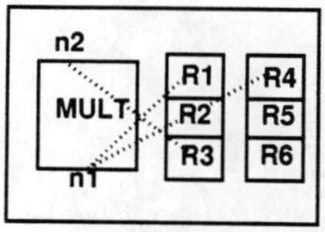
4.10 Allocating and binding of interconnect units

During operator and variable binding, interconnect units are automatically added to the connectivity table. These interconnect units ensure that data transfer between bound operators/variables is possible. The example shows the connectivity table after binding operator $*_1$ and $*_2$ to $MULT$, and variables A to register $R1$, D to $R4$ and B to $R3$. The two interconnect units introduced, $n1$ and $n2$, are "virtual nets", meaning that they do not have a real implementation, yet. Thus, the next step in the design scenario is to allocate interconnect units to implement these virtual nets. The procedure is the same as allocation for functional or storage units. The user selects the implementation style (i.e. one level bus, two level bus, one level multiplexor, two level multiplexor, etc.) and the bitwidth for each virtual net. After the interconnect unit has been allocated, the virtual net is bound to the allocated unit. The binding is then reflected in the floor plan by changing the "dotted" virtual nets to "solid" routing tracks.

Allocation and Binding of Interconnect Units

Creation of virtual nets after operators and variables binding

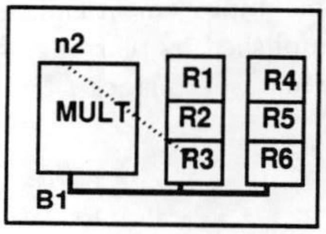
ST1	A = B *1 C
ST2	D = E *2 F



Binding *1 to MULT, A to R1
*2 to MULT, D to R4, and B to R3

	n1	n2	Total #drivers
Mult	O	//	1
R1	/		
R2			
R3		O	1
R4	/		
⋮			
Fan in	1	1	
Fan out	2	1	

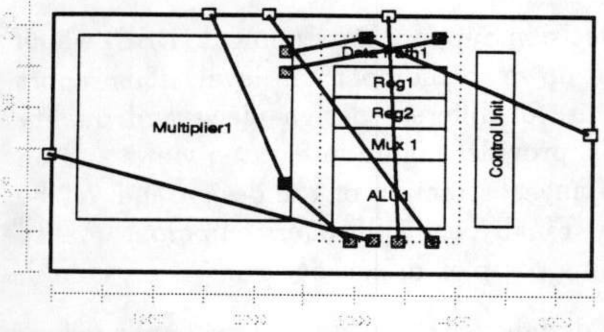
Floorplan after Interconnect binding



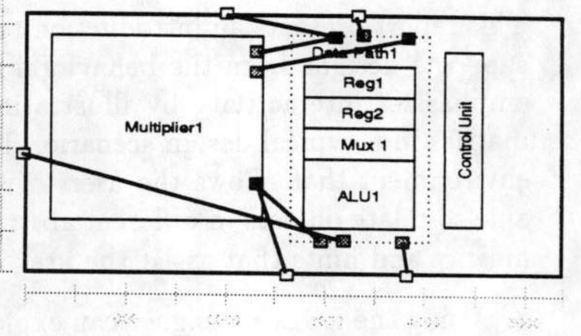
4.11 Final floorplanning

Once all operators, variables and virtual nets have been bound, the floorplan shows the placement of components, I/O pads, routing and wasted area. The next thing the user may wish to do is to improve the floorplan by changing the placement of components, altering the positions of I/O pads, or rerouting buses. Feedback from these changes is reflected directly in the floorplan view by quality metrics displaying total functional unit area, total routing area, total wasted area, etc. The example shows the floorplan of a design before a possible improvement and the floorplan of the same design after. The improvement was accomplished by rearranging input and output ports and by reassigning pin positions for the data path stack.

Floorplaning



Total area:	5550 um^2	Wasted area:	1150 um^2
F.U. area:	2250 um^2	Critical path wire length:	542 um
S.U. area:	1250 um^2	Wire length:	1247 um
I.U. area:	900 um^2		



Total area:	5000 um^2	Wasted area:	1000 um^2
F.U. area:	2250 um^2	Critical path wire length:	500 um
S.U. area:	1250 um^2	Wire length:	1247 um
I.U. area:	700 um^2		

5 Conclusion

This report presents an introduction to the Decision Support Environment (DSE) which supports designs from the behavioral level on down to the floorplan level. This report emphasizes interactivity by illustrating the use of different design views and quality metrics in a typical design scenario. The DSE provides highly integrated views and an environment that allows the user to work in any abstraction of the design and yet be able to relate objects in different abstractions. Finally, the DSE offers numerous quality metrics and hints that assist the user in making complex design decisions.

Using the DSE, a designer can explore the design space quickly, and expect to reduce design time by 50%, and the number of iterations in the design cycle to only few. In addition, the DSE provides excellent design documentation covering the main design abstractions from the state table through the floorplan. The relationships inherent in the environemnt between components or objects of different abstractions allows the user of the DSE to determine implementation information for each behavior operators and variables, or to determine behavioral operations that are implemented by each hardware component. For example, the user can determine the component that is used to implement an add operation in the behavior, and at the same time, he/she can find out the exact placement of that component in the floor plan. Last but not the least, DSE can be used as a front end for other automatic synthesis tools.

Decision Support Environment

Conclusion

Aided design capture from Behavior to Floorplan

- Highly Integrated Multi-view
- Quality metrics

Gain:

- Expected 50% reduction in design time
- Expected large reduction in number of iterations in the design cycle
- Excellent documentation
- Good front end for other synthesis tools

6 References

[BrCa88] Brayton, R., Camposano, R., et al., "The Yorktown Silicon Compiler," *Silicon Compilation*, Addison Wesley, 1988.

[CaWo91] Camposano, R., Wolf, B., *High Level Synthesis of VLSI Systems*, Kluwer, 1991.

[DeRa86] DeMan, H., Rabaey, J., et al., "Cathedral II: A Silicon Compiler for Digital Signal Processing," IEEE Design and Test, Dec., 1986.

[GDWL92] D.D. Gajski, et. al., *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer, 1992.

[HaCG93] Hadley, T., Chaiyakul, V., & Gajski, D. D., "A Data Structure for Interactive Synthesis," Info. & Computer Science Dept., UCI, Tech. Rep. 93-6, January 1993.

[HaWG92] Hadley, T., Wu, A. C-H., & Gajski, D. D., "An Efficient Multi-View Design Model for Real-Time Interactive Synthesis," Info. & Computer Science Dept., UCI, Tech. Rep. 92-35, April 1992.

[HaGa93] Hadley, T. & Gajski, D. D., "A Decision Support Environment," Info. & Computer Science Dept., UCI, Tech. Rep. 91-17, August 1991.

[Sh89] Shung, C., et al., "An Integrated CAD System for Algorithm-Specific IC Design," Proc. Int'l Conf. System Design, Jan., 1989.