RTBA : A Generic Bit-Sliced Bus Architecture for DataPath Synthesis*

Kamlesh Rath
Ignacio Celis
Robert M. Wehrmeister
Steven D. Johnson

Computer Science Department
Indiana University, Bloomington, IN 47405

Abstract

Register transfer level (RTL) equations are used to specify the register and ALU datapaths of machine architectures. *RTBA (Register Transfer Bus Architecture)* is a target architecture for automatic bit-sliced VLSI implementation of RTL equations. This article discusses the automatic derivation process of a layout from a typical system of RTL equations using a series of behavior preserving transformations. The test results of a chip fabricated using the derived layout are also presented. Extensions to the RTBA transformations, allowing functions in the RTL equations, are presented by deriving the *min-max* benchmark.

*This research was supported in part by the National Science Foundation under grants numbered MIP 8707067, MIP 8921842, and DCR 8521497.

1 Introduction

Synthesis can be characterized as the use of transformation steps to translate a specification into an implementation. The behavioral equivalence of source specification and implementation is assured by constraining the synthesis process to a set of behavior preserving transformations. Datapath synthesis is comprised of three major steps, scheduling, allocation, and layout generation. Most datapath synthesis systems (e.g. [16],[10],[11]) concentrate on scheduling and allocation and defer layout considerations to logic synthesis tools. Current logic synthesis tools (*e.g. PLA, Standard Cell, Gate Matrix*) allow completely automatic hardware generation from a system of boolean equations. Such tools generate efficient layouts, but they lack in regularity of structure and testability.

RTBA (Register Transfer Bus Architecture) is an open ended architecture, well suited as a target architecture for synthesizing processor datapaths, that can be partitioned into bit-slices. It evolved from the classical single bus architecture ([9],[12]) which does not handle multiple data transfers between circuit elements simultaneously. RTBA is designed to address the synthesis of highly parallel datapaths. The multiple bus architecture distributes datapath multiplexers to a single *control decode unit* and a set of switches[8].

A processor description can be decomposed into a control description and a datapath description. The datapath is described as a system of RTL equations. A sequence of behavior preserving transformations maps the RTL equations to RTBA. Transformations on the input RTL equations allocate buses, logical units, shift units, comparator units, arithmetic units, and registers. The allocated units are tiled together and compiled for an area efficient layout.

On the front-end, the RTBA transformation system is integrated with DDD [4], an experimental transformation system that extracts control and datapath description from a high-level functional specification. On the back-end, it is integrated with the Berkeley tools [14] (*mquilt, mpla, magic*) to generate circuit layouts.

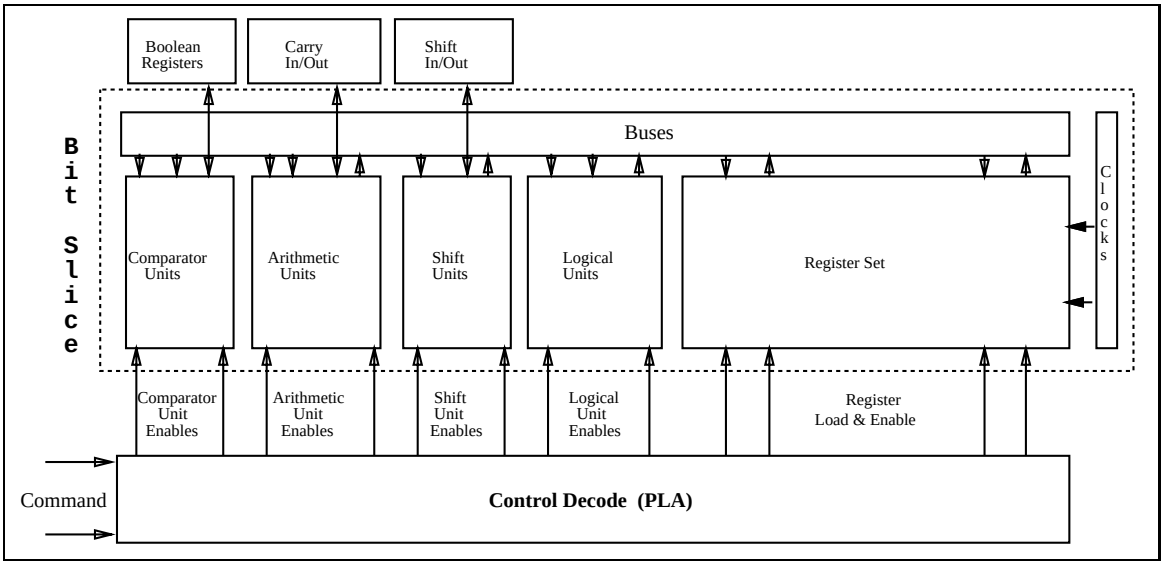

Figure 1: Block diagram for the RTBA

2 Operation of RTBA

Each bit-slice of the RTBA has an independent register transfer path. Figure 1 shows the block diagram for a bit-slice of the RTBA layout. The register set and I/O ports are partitioned into *non-conflicting* subsets using the allocation algorithm described in Section 4.1.2. Each such subset is connected to an unidirectional input or output bus. The arithmetic, shift, comparator and logic units on the value on the input bus(es) and write the result on to it's output bus. Each unique function invocation is performed by a different functional block.

The *command* of the machine with register transfer $X_i \Leftarrow (X_j \oplus X_k)$ is decoded into the control signals $load\text{–}X_i$, $enable\text{–}X_j$, $enable\text{–}X_k$, and $\oplus\text{–}enable$. The $enable\text{–}X_j$, $enable\text{–}X_k$ signals connect registers X_j, X_k to buses, which in turn connect to the inputs of the functional unit; $\oplus\text{–}enable$ switches the value of $(X_j \oplus X_k)$ to the appropriate bus; and the $load\text{–}X_i$ signal enables register X_i to take it's value from the it's input bus.

The bit-sliced [9] functional units have transmission gates on their outputs to control *writing* onto the buses. Buses connect the inputs and outputs of the registers to functional units. Left and right shift units transfer bus values to corresponding shift units in neighboring bit-slices.

The comparator units ($>$, $<$, $=$) update single-bit (boolean) registers. Logical ($\vee$, $\wedge$, $\neg$, *identity function*) and arithmetic ($+$, $-$, *inc*, *dec*) functions are performed by their respective functional units.

The register set consists of standard D-registers driven by a 4-phase clock. The registers have serial-in and serial-out capabilities driven by a separate clock. All the registers in a bit-slice can be threaded together a single I/O pin, to facilitate loading and testing of register values.

The *control decode unit* decodes the current *status* information from the control unit into appropriate *Load* and *Enable* signals for each register and the *output enable* signals for the functional units. It also generates the control inputs for arithmetic units.

3 Specification

A behavioral description of a machine in DDD is an *iterative* [5] system of functional definitions. Each function body is a tail-recursive conditional expression, which transfers control to the next state and simultaneously updates the registers and I/O ports.

3.1 High-level Specification

The input specification is of the form :

$$\begin{array}{rll} (machine = & (& \\ & (F_1 \quad [X_1\ X_2\ ...\ X_n]. & E_1) \\ & : & \\ & (F_m \quad [X_1\ X_2\ ...\ X_n]. & E_m) \\ ) & E) & \end{array}$$

where X_1, ..., X_n denote register and I/O ports, F_1, ..., F_m are control states, and E_1, ..., E_m are tail-recursive conditional expressions. E describes the initial state and register, port values of *machine*.

3.2 RTL Equations

A series of correctness preserving transformations [6] in DDD converts the machine specification into a control specification and a system of RTL equations which form the input for the RTBA.

$$\begin{aligned}((X_1 &\Leftarrow (select\ \ status\ \ g_{11}\, g_{12} \ldots g_{1k}))\\ (X_2 &\Leftarrow (select\ \ status\ \ g_{21}\, g_{22} \ldots g_{2k}))\\ &\cdot\\ (X_n &\Leftarrow (select\ \ status\ \ g_{n1}\, g_{n2} \ldots g_{nk})))\end{aligned}$$

where g_{ij} is a function of X_1, ..., X_n, and *status* is the controlling signal. The *select* function selects a particular value of g_{ij} to load into the corresponding X_p, depending on status. The transposed view of the system of RTL equations is called a *register transfer table*.

3.3 Type Information

RTBA requires the user to specify the type of each variable used in the input specification. The supported types are *register*, *input port*, *output port*, and *boolean*. The type information is used by RTBA to allocate registers, buses and functional units.

Variable names with `?` as the first character are input ports. Output ports have `!` as the first character of the variable name. Boolean variables are identified by a `?` as the last character in their names. The rest of the variables are of type register.

4 Example 1 : A Simple Machine

A simple specification was chosen to layout a chip using RTBA, and to simulate, fabricate, test the chip. The example includes multiple register transfers and transfers from and to the same register in the same datapath cycle. All register transfers are through *identity* function units. This specification does not have any predicates or arithmetic, logical, comparison functions.

The *iterative specification* of the example is written in *Scheme*. The specification has 5 registers, and one input port. It was then transformed through DDD into a control specification and a system

Type	Registers				
Status	a	b	c	d	e
0	*?i*	b	c	d	e
1	a	*?i*	c	d	e
2	a	b	*?i*	d	e
3	a	b	c	*?i*	e
4	a	b	c	d	*?i*
5	*b*	b	*d*	d	e
6	*e*	*a*	c	d	e
7	a	b	*b*	*a*	e
8	a	b	c	*c*	*b*
9	*d*	b	c	d	*d*
10	a	*e*	c	d	*a*

Table 1: Register Transfer Table

of RTL equations. The register transfer table corresponding to the system of RTL equations is given in Table 1.

4.1 Allocating Registers, Buses and Functional Units

RTBA is an open ended architecture. There are no constraints on the number of registers, buses, and functional units allocated to implement the given machine specification. The synthesis process generates a cost effective layout, without altering the behavior of register transfers. Derivations illustrated in Sections 4.1.1 - 4.1.2 are used to allocate different units in the layout. The script used to allocate registers, buses, functional units, and to generate the layout of a bit-slice is given in Appendix A.

4.1.1 Register Allocation

All variables of type *register* are allocated registers. This example is allocated 5 registers (`a, b, c, d, e`). The layout of the registers is arranged in pairs for proper tiling in the floor-plan.

4.1.2 Bus Allocation

All entries in the register transfer table of type *register* or *input port* whose values form inputs to functional units (in italics font in Table 1) are called *sources*, and all entries of *register* or *output port* types which get *loaded* with a value from a functional unit are called *destinations*. Sources (Destinations) which are never used for the same status value can be allocated to the same bus.

The problem of allocating buses for a given number of sources or destinations can be reduced to a *graph vertex-coloring* problem[3]. Consider a graph with nodes for each source (destination) and an edge for each *dependency*. There exists a dependency between 2 sources (destinations), if both are used for the same datapath state. The solution to the bus allocation problem is equivalent to identifying the minimum number of colors (*chromatic number*) in the graph in which no two nodes with the same color have an edge between them. Each color in the graph could then be assigned to a source (destination) bus.

The *graph vertex-coloring* problem is a known NP-complete [3] problem. Our solution to the problem, is based on heuristics and is not necessarily optimal. The source (destination) with the maximal number of dependencies is first allocated to a bus. On each iteration of the algorithm, the source (destination) with the next highest number of dependencies is checked for dependencies with all the allocated buses in the order they were allocated. It is allocated to the first bus with which it does not share a dependency. It is allocated an independent bus if it has dependencies with all the allocated buses.

The sources in the example are `(?i c b d e a)`, and the destinations are `(a b c d e)`. The sources and destinations in each datapath state are given in Table 2. The dependencies of the source and destination registers are shown in the *dependency table* in Table 3. The designer can also choose to allocate input/output ports as independent buses. In this example, the input port `?i` is allocated a separate bus.

Processing the register dependencies using the Bus Allocation algorithm results in the 3 destination and 3 source buses (shown in Table 4).The *Bus Transfer Table* shown in Table 5 is generated by substituting each register with it's allocated bus and merging together the columns with the same names. If the buses are allocated properly, then the merging operation will not result in any

Status	Destinations	Sources
0	a	?i
1	b	?i
2	c	?i
3	d	?i
4	e	?i
5	a c	b d
6	a b	e a
7	c d	b a
8	d e	c b
9	a e	d d
10	b e	e a

Table 2: Sources and Destinations

Destinations	*Dependencies*	*Sources*	*Dependencies*
a	c b e	a	e b
b	a e	b	d a c
c	a d	c	b
d	c e	d	b
e	a b d	e	a
		?i	-

Table 3: Dependency Table

Destinations		*Sources*	
Bus	*Registers / Outputs*	*Bus*	*Registers / Inputs*
d0	c e	so	?i
d1	d a	s1	e b
d2	b	s2	c d a

Table 4: Bus Allocation Table

Status	d0	d1	d2
0	?	s0	?
1	?	?	s0
2	s0	?	?
3	?	s0	?
4	s0	?	?
5	s2	s1	?
6	?	s1	s2
7	s1	s2	?
8	s1	s2	?
9	s2	s2	?
10	s2	?	s1

Table 5: Bus Transfer Table

position of table having more than one bus name. The *Bus Transfer Table* is used to generate the enable signals for the functional units. Each non-don't-care term in the transfer table is assigned an identity functional unit.

4.2 Generating Control Decode

The *Load* and *Enable* signals for the registers are generated for each command from the register transfer table. For each command value, the registers being loaded with a different value is given a *Load* signal for the *command*, and each register whose value is used as an input to a functional block is given an *Enable* signal.

The command information from the *control unit* is decoded by the decoder into enable signals for the transmission gates at the output of functional blocks, and the control signals for the arithmetic unit.

The register transfer table is transformed into register loads and enables. The bus transfer table and the allocated functional units are used to generate the enable signals. There are 9 different functional units, each for a different bus transfer `(d0←s0, d0←s1, d0←s2, d1←s0, d1←s1, d1←s2, d2←s0, d2←s1, d2←s2)` . An enable signal is generated for each functional unit for

	a		b		c		d		e		d0	d0	d0	d1	d1	d1	d2	d2	d2
											↑	↑	↑	↑	↑	↑	↑	↑	↑
Status	Ld	En	Ld	En	Ld	En	Ld	En	Ld	En	s0	s1	s2	s0	s1	s2	s0	s1	s2
0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
2	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0
4	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0
5	1	0	0	1	1	0	0	1	0	0	0	0	1	0	1	0	0	0	0
6	1	1	1	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	1
7	0	1	0	1	1	0	1	0	0	0	0	1	0	0	0	1	0	0	0
8	0	0	0	1	0	1	1	0	1	0	0	1	0	0	0	1	0	0	0
9	1	0	0	0	0	0	0	1	1	0	0	0	1	0	0	1	0	0	0
10	0	1	1	0	0	0	0	0	1	1	0	0	1	0	0	0	0	1	0

Table 6: Register Loads & Output Enables and Functional Unit Enables

the appropriate status value. The boolean equations for the control decode (shown in Table 6) are used to automatically generate the layout of a PLA using *mpla* [14].

4.3 Layout, Simulation and Fabrication

The layout of the PLA to implement the control decode is shown in Appendix B. The layout for each bit-slice is generated automatically using *mquilt* [14], as shown in Appendix C.

The layout was simulated using the logical simulator COSMOS [1], and was verified to have behavioral equivalence with the specified RTL equations. The script of the COSMOS simulation for the layout of Example 1 is included in Appendix D.

SPICE [13] was used to simulate the electrical characterstics of the layout of a single bit-slice. The spice deck and the graph corresponding to the simulations of states 0 and 6 are shown in Appendix E.

The pin diagram for the chip layout is given in Appendix F. The layout was fabricated at the

MOSIS fabrication facility (Fabrication ID number : N05F-BR1). The chip was tested using the Logic Engine[17] (test programs given in Appendix G).

5 Example 2 : The Min-Max Machine

The *min-max* machine is a simple machine that computes and stores the running maximum and minimum values on it's *input port*. It also computes the average of the current maximum and minimum and displays it on it's *output port*. This specification is a simpler version of the *min-max* benchmark specification used in IMEC '89 [15] conference. The control signals CLEAR, ENABLE and RESET were eliminated to simplify the control unit and concentrate on the datapath.

The RTBA synthesis process comprises of four major steps, generating RTL equations by datapath extraction (using DDD), allocating registers, buses, and functional units, floor-planning and layout generation, and generating control decoder.

5.1 Specification

The behavioral description for the *min-max* machine :

```
(min-max = (
     (START    [min max last-in tmp !out lt? gt?].
            (CHECKMIN min max ?in tmp (-> tmp) (< min ?in) (> max ?in)))
     (CHECKMIN [min max last-in tmp !out lt? gt?].
            (if ( lt? )
               (UPDATE last-in max last-in tmp (-> tmp) lt? gt?)
               (CHECKMAX min max last-in tmp (-> tmp) lt? gt?)))
     (CHECKMAX [min max last-in tmp !out lt? gt?].
            (if ( gt? )
               (UPDATE min last-in last-in tmp (-> tmp) lt? gt?)
               (UPDATE min max last-in tmp (-> tmp) lt? gt?)))
     (UPDATE   [min max last-in tmp !out lt? gt?].
            (START min max last-in (+ min max) (-> tmp) lt? gt?))
) (START HI_VALUE 0 0 0 0 F F))
```

Type	Registers				Output port	Boolean	
Command	min	max	last-in	tmp	!out	lt?	gt?
0	min	max	*?in*	tmp	(-> *tmp*)	(< *min ?in*)	(> *max ?in*)
1	*last-in*	max	last-in	tmp	(-> *tmp*)	lt?	gt?
2	min	max	last-in	tmp	(-> *tmp*)	lt?	gt?
3	min	*last-in*	last-in	tmp	(-> *tmp*)	lt?	gt?
4	min	max	last-in	tmp	(-> *tmp*)	lt?	gt?
5	min	max	last-in	(+ *min max*)	(-> *tmp*)	lt?	gt?

Table 7: Register Transfer Table

where -> is the right shift operator; min, max, last-in, tmp are *registers*, !out is an *output port*, lt?, gt? are *boolean* (single-bit) registers, and ?in is an *input port*. START, CHECKMIN, CHECKMAX, UPDATE are the control states. START is the initial state of the machine. The specification contains only simple arithmetic, shift and comparison functions, all of which can be directly implemented as functional blocks. Also, predicates test only *boolean* registers which can interact with the *control*.

The flow-chart corresponding to the DDD specification for the *min-max* machine is shown in Figure 2, where ellipses denote control states, diamonds denote predicates and rectangles denote datapath states. The datapath states have been numbered 0-5.

DDD transformations are used to extract the datapath RTL equations from the *min-max* specification. The register transfer table corresponding to the datapath RTL equations for the *min-max* machine, is shown in Table 7. Datapath transfers (shown in italics in Table 7) correspond to datapath states in the flow-chart.

```
((min     <== (select status min last-in min min min min))
 (max     <== (select status max max max last-in max max))
 (last-in <== (select status ?in last-in last-in last-in last-in last-in))
 (tmp     <== (select status tmp tmp tmp tmp tmp (+ min max)))
 (!out    <== (select status (-> tmp)(-> tmp)(-> tmp)(-> tmp)(-> tmp)(-> tmp)))
 (lt?     <== (select status (< min ?in) lt? lt? lt? lt? lt?))
 (gt?     <== (select status (> max ?in) gt? gt? gt? gt? gt?)))
```

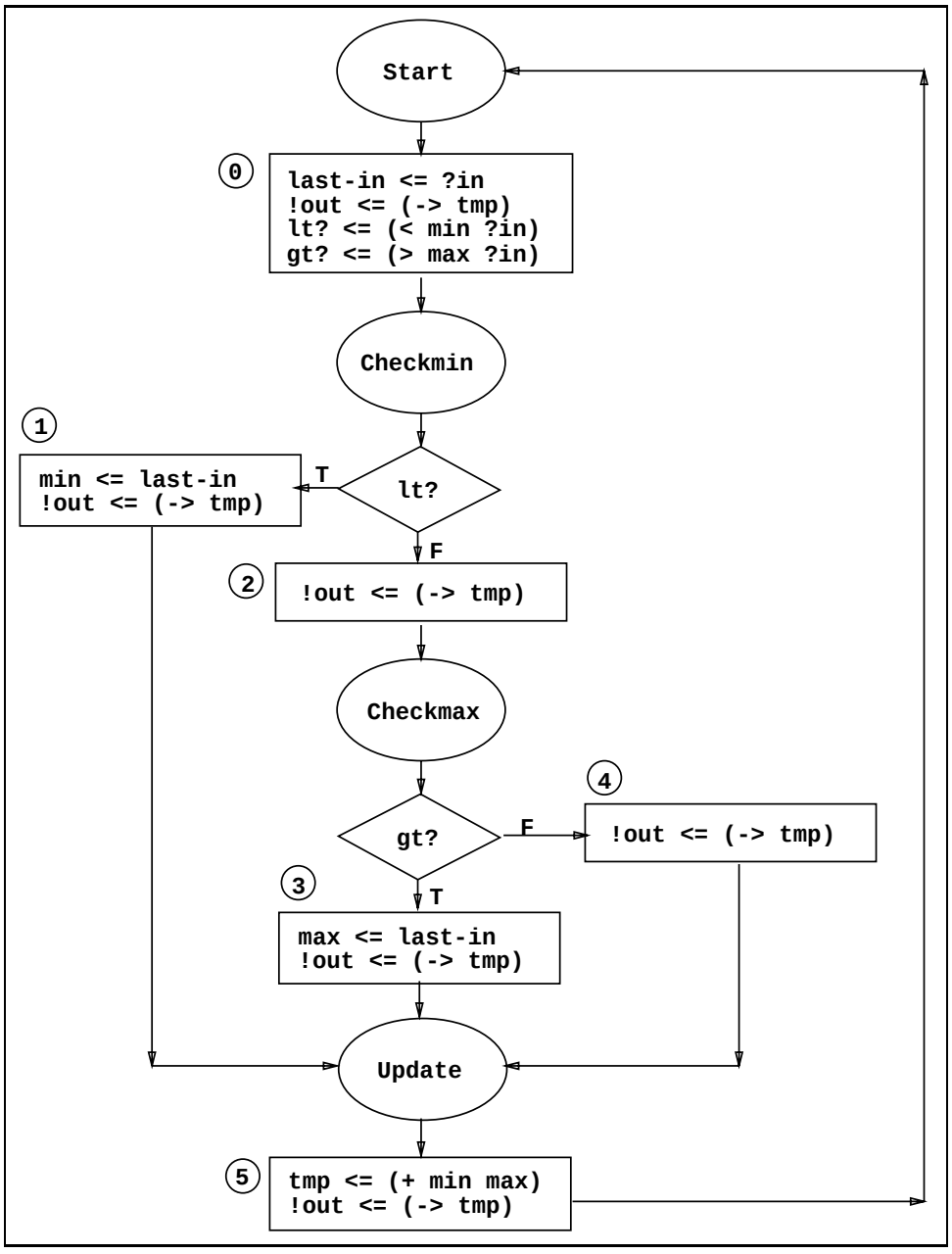

Figure 2: Flow Chart of the Min-Max Machine

Sources	*Dependencies*	*Destinations*	*Dependencies*
min	?in tmp max	min	!out
max	?in tmp min	max	!out
last-in	tmp	last-in	!out
tmp	min max ?in last-in	tmp	!out
?in	tmp min max	!out	min max tmp last-in

Table 8: Dependency Table

Sources		*Destinations*	
Bus	*Registers / Inputs*	*Bus*	*Registers / Outputs*
s0	tmp	d0	!out
s1	min last-in	d1	min max tmp last-in
s2	max		
s3	?in		

Table 9: Bus Allocation Table

5.2 Allocating Registers, Buses and Functional Units

The entries in the register transfer table with type *register* are allocated registers in the layout, *e.g.* `min, max, last-in, tmp` are registers in the *min-max* machine. Entries with type *boolean* are connected to outputs of comparator units in the least significant bit-slice and not allocated registers in the bit-slice.

The dependencies in the *min-max* machine are shown in Table 8. Single-bit registers are not used in the dependency transformations because they are not allocated buses in each bit-slice. The bus allocation algorithm results in 4 source buses`(s0 s1 s2 s3)` and 2 destination buses`(d0 d1)` for the *min-max* machine, as shown in Table 9. The bus transfer table shown in Table 10 is generated by substituting each register and port with it's allocated bus and merging together the columns with the same names in the register transfer table. The `?` entries in tables denote don't-care terms. Further serialization of the input specification would result in fewer buses and more control states.

The RTBA transformation system allocates a functional unit for each function. This is a tradeoff for a smaller and faster interconnect area[7]. The *bus transfer table* for *min-max* results in

Command	d0	d1	lt?	gt?
0	(-> s0)	s3	(< s1 s3)	(> s2 s3)
1	(-> s0)	s1	?	?
2	(-> s0)	?	?	?
3	(-> s0)	s1	?	?
4	(-> s0)	?	?	?
5	(-> s0)	(+ s1 s2)	?	?

Table 10: Bus Transfer Table

the following set of function applications, each of which is assigned a functional unit :

{ (d0 ⇐ (-> s0)), (d1 ⇐ s3), (d1 ⇐ s1), (d1 ⇐ (+ s1 s3)),

(lt? ⇐ (< s1 s3)), (gt? ⇐ (> s2 s3))}

The *min-max machine* is allocated 1 right-shift unit, 2 identity function units, 1 adder unit, 1 less-than comparator unit and 1 greater-than comparator unit. `lt?` and `gt?` are single-bit registers connected to the corresponding comparator outputs of the least significant bit-slice.

5.3 Floor-planning and Layout Generation

The layout for different functional units, registers, and bus fragments are kept in a cell library. All cell layouts have the same height for proper tiling. The floor-plan for the VLSI layout of a bit-slice of the *min-max* datapath is shown in Figure 3. Successive bit-slices are stacked together to connect common data transfer paths and load/enable signals.

The automated floor-planner of the RTBA transformation system generates a template map for each bit-slice. The layout for a bit-slice is generated using *mquilt*[14], by substituting each template on the map with it's corresponding cell layout. The layout for a *min-max* bit-slice is shown in Appendix H.

5.4 Generating Control Decode

The register transfer table is transformed into register loads and enables in Table 11. The bus transfer table and the allocated functional units are used to generate the enable signals, as shown

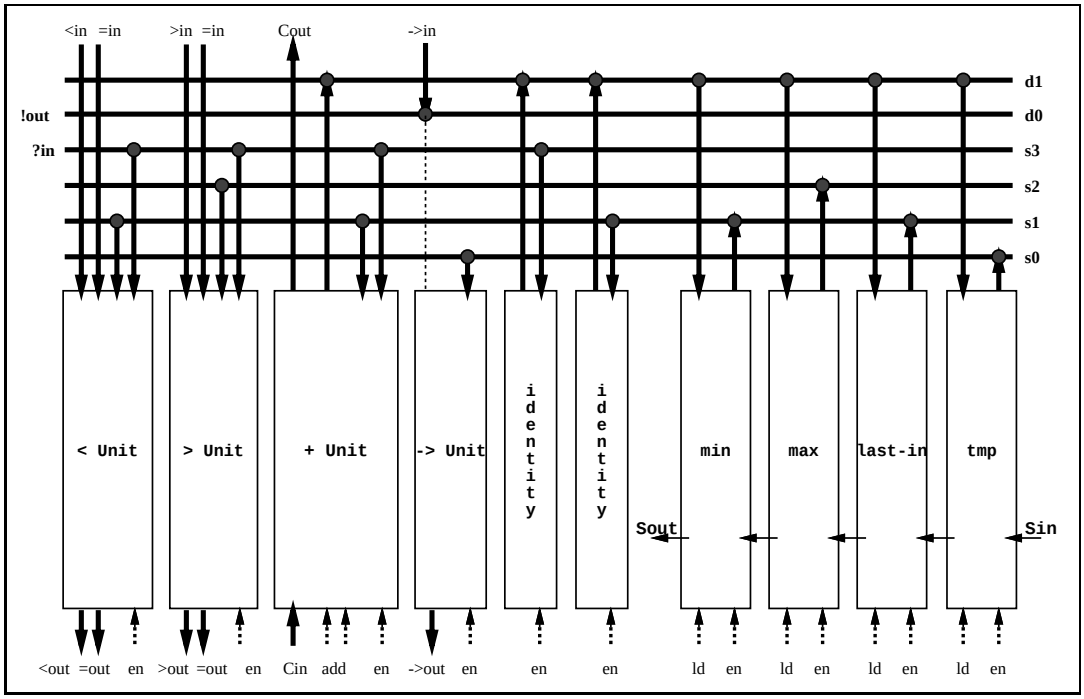

Figure 3: Floor-plan of Min-Max layout

	min		max		last-in		tmp	
Command	Load	Enable	Load	Enable	Load	Enable	Load	Enable
0	0	1	0	1	1	0	0	1
1	1	0	0	0	0	1	0	1
2	0	0	0	0	0	0	0	1
3	0	0	1	0	0	1	0	1
4	0	0	0	0	0	0	0	1
5	0	1	0	1	0	0	1	1

Table 11: Register Loads & Output Enables

DataPath State	-> unit	d1 ⇐ s3	d1 ⇐ s1	+ unit	< unit	> unit
0	1	1	0	0	1	1
1	1	0	1	0	0	0
2	1	0	0	0	0	0
3	1	0	1	0	0	0
4	1	0	0	0	0	0
5	1	0	0	1	0	0

Table 12: Functional Unit Enables & Control Code

in Table 12. Tables 11 and 12 are merged to generate boolean equations, which are processed through *mpla* [14] to generate a PLA to implement the control decoder. The layout for the control decode PLA is shown in Appendix I.

6 Observations and Directions for Future Work

The RTBA transformation system can be used to implement any system of RTL equations that can be partitioned into bit-slices, if all the function applications correspond to some functional unit in the library of functional units and all predicates can be implemented using boolean registers. Complex functions need to be *serialized* into a sequence of implementable functions, or be *factored* [4] from the RTL equations into a set of I/O ports and implemented by the user. Data transfers across bit-slice partitions are also implemented as a set of I/O ports.

Future work on the RTBA will focus on automation of transformations, support for user defined functional units, and improved layout generation. Correctness proofs for the transformations leading to the layout would give greater confidence in VLSI implementations of machines through the synthesis process.

References

[1] Randy Bryant, *Compiled Simulator for MOS Circuits*, Carnegie Mellon University.

[2] R. Camposano, *Behavior Preserving Transformations for High-Level Synthesis*, Hardware Specification, Verification and Synthesis: Mathematical Aspects, Springer-Verlag, July 1989.

[3] M. R. Garey and D. S. Johnson, *Computers and Intractibility*, W.H.Freeman and Company, San Francisco, 1979.

[4] Steven D. Johnson and Bhaskar Bose, *A System for Mechanized Digital Design Derivation*, to appear at International Workshop on Formal Methods in VLSI Design, Miami, January 1991.

[5] Steven D. Johnson, *Synthesis of Digital Designs from Recursion Equations*, The MIT Press, Cambridge, 1984.

[6] Steven D. Johnson, Robert M. Wehrmeister and Bhaskar Bose, *On the Interplay of Synthesis and Verification*, Formal VLSI Specification and Synthesis, Elsevier Science Publishers B.V. (North Holland), IFIP, 1990.

[7] K. Kucukcakar and A.C. Parker,*Data Path Tradeoffs using MABAL*, 27th ACM/IEEE Design Automation Conference, June 1990.

[8] T.A. Ly, W.L. Elwood and E.F. Girczyc, *A Generalized Interconnect Model for Data Path Synthesis*, 27th ACM/IEEE Design Automation Conference, June 1990.

[9] M. Morris Mano, *Digital Logic and Computer Design*, Prentice Hall, Englewood Cliffs, New Jersey, USA.

[10] A.C. Parker, J "T" Pizarro and M. Mlinar, *MAHA: A Program for Datapath Synthesis*, 23rd ACM/IEEE Design Automation Conference, July 1986.

[11] P.G. Paulin, J.P. Knight and E.F. Girczyc, *HAL: A Multi Paradigm Approach to Automatic Data Path Synthesis*, 23rd ACM/IEEE Design Automation Conference, July 1986.

[12] F.P. Prosser and D.E. Winkel, *The Art of Digital Design*, Prentice Hall, Englewood Cliffs, New Jersey, USA.

[13] Thomas L. Quarles, *SPICE 3C1 Users Guide*, UCB/ERL M89/46, Electronics Research Laboratory, College of Engineering, University of California, Berkeley.

[14] Walter S. Scot, Robert N Mayo, Gordon Hamachi, and John K. Ousterhout, *1986 VLSI Tools*, Report No. UCB/CSD 86/272,(Computer Science Division-EECS, University of California at Berkeley, 1985). California, Berkeley.

[15] Diederik Verkest, Luc Claesen, and Hugo De Man, *Formal Design Benchmark Example*, Formal VLSI Specification and Synthesis, Elsevier Science Publishers B.V. (North Holland), IFIP, 1990.

[16] R.S. Wei, S. Rothweiler and J.Y. Jon, *BECOME: Behavior Level Synthesis Based On Structure Mapping*, 25th ACM/IEEE Design Automation Conference, June 1988.

[17] David Winkel, Franklin Prosser, Robert Wehrmeister, William Hunt and Caleb Hess, *A Student VLSI Hardware Tester*, Indiana University.

A Script for Allocation and Layout

```
; ---- Script for deriving RTL Equations ----
(load "/u/bose/ddd/ddd.ss")
(define x1 (readfile "x0"))
(define x2  (iterativesystem->singleloop x1))
(define x3  (singleloop->streqns x2))
(define xsel (singleloop->control x2))
(writefile x3 "x3")
(writefile xsel "xsel")
(define state  (list (extractstreqn 'state x3)))
(define registers (removestreqn 'state x3))
(writefile registers "registers.con")
(makepaltable registers "registers.rtt" "registers.tab")

; ---- Script to allocate units and plot -----------------
(load "/u/bose/ddd/ddd.ss") ; --- used DDD v0.2
(load "/u5/install/ddd/generic/buses.ss")
(load "/u5/install/ddd/generic/bitslice.ss")
(load "/u5/install/ddd/generic/dcare.ss")
(load "/u5/install/ddd/generic/gen.ss")
(load "predinfo")

(define registers (readfile "registers.con"))
(define rtt (streqn->paltable registers))
(define en.rtt (enables rtt))
(define ld.rtt (loads rtt))
(define enld.str (make-enld.str en.rtt ld.rtt))

(writefile enld.str "enld.str")

(define srcs (sources rtt))
(define dests (destinations rtt))
(define ldest-bus (alloc-dest-buses rtt dests #t))
(define lsource-bus (alloc-source-buses rtt srcs #t))

(writefile ldest-bus "ldb.nam")
(writefile lsource-bus "lsb.nam")

(define lbus (mk-source-buses 'l. lsource-bus (mk-dest-buses 'd. ldest-bus rtt)))
(writefile lbus "bus.rtt")
```

```
(define d-list (dest-list lbus))
(define s-list (source-list lbus))
(define xfer.str (rtt->sw_enable lbus))
(define inverted_xfer.str (invert_sw xfer.str))

(writefile inverted_xfer.str "xfer.str")

(define sh_enld.str (shuffle_ld_en enld.str))
(define pla (append inverted_xfer.str sh_enld.str))
(define plaout (map car pla))

(writefile plaout "pla.out")
(writefile pla "pla")

(streqns->mpla pla 'static "pla.mpla")

(bitslice-layout lbus rtt lsource-bus ldest-bus "slice.in")


; ------------- Source specification "x0" ------------------
(define test
    (letrec(
            (state0 (lambda(a b c d e) (state1 ?i b c d e)))
            (state1 (lambda(a b c d e) (state2 a ?i c d e)))
            (state2 (lambda(a b c d e) (state3 a b ?i d e)))
            (state3 (lambda(a b c d e) (state4 a b c ?i e)))
            (state4 (lambda(a b c d e) (state5 a b c d ?i)))
            (state5 (lambda(a b c d e) (state6 b b d d e)))
            (state6 (lambda(a b c d e) (state7 e a c d e)))
            (state7 (lambda(a b c d e) (state8 a b b a e)))
            (state8 (lambda(a b c d e) (state9 a b c c b)))
            (state9 (lambda(a b c d e) (state10 d b c d d)))
            (state10 (lambda(a b c d e) (state0 a e c d a))))
      (lambda()(state1 1 2 3 4 5))))
```

B Status Decoder PLA Plot

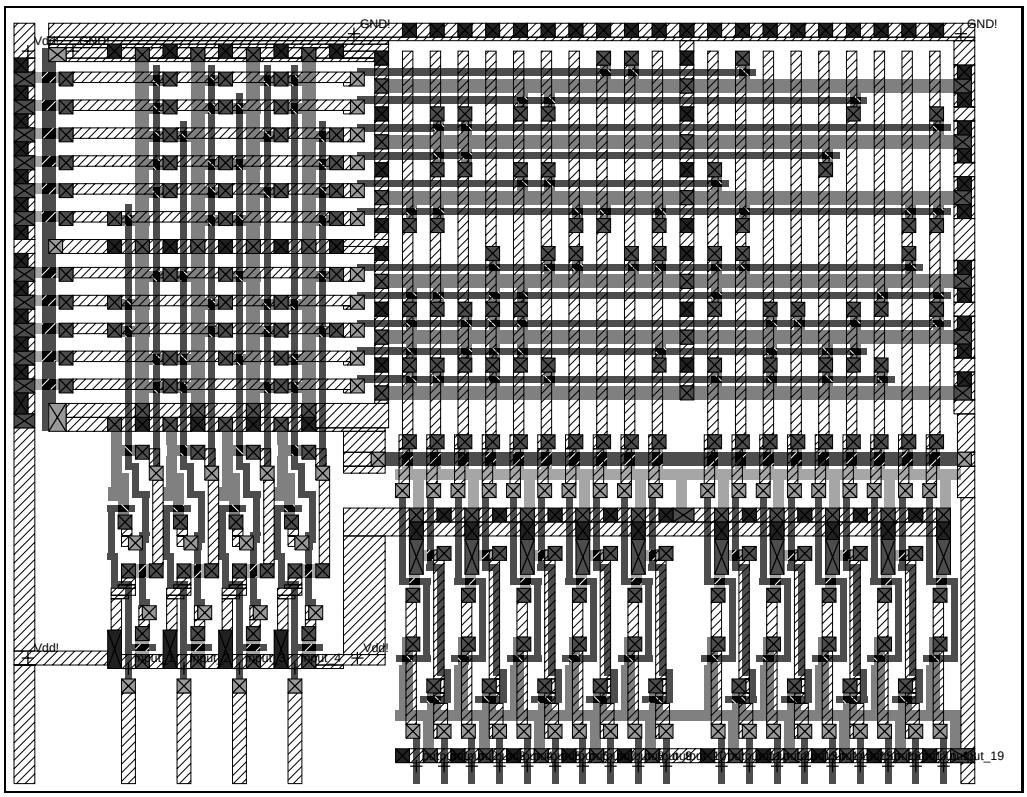

Figure 4: Magic plot status decode PLA

C Bit-slice Plot

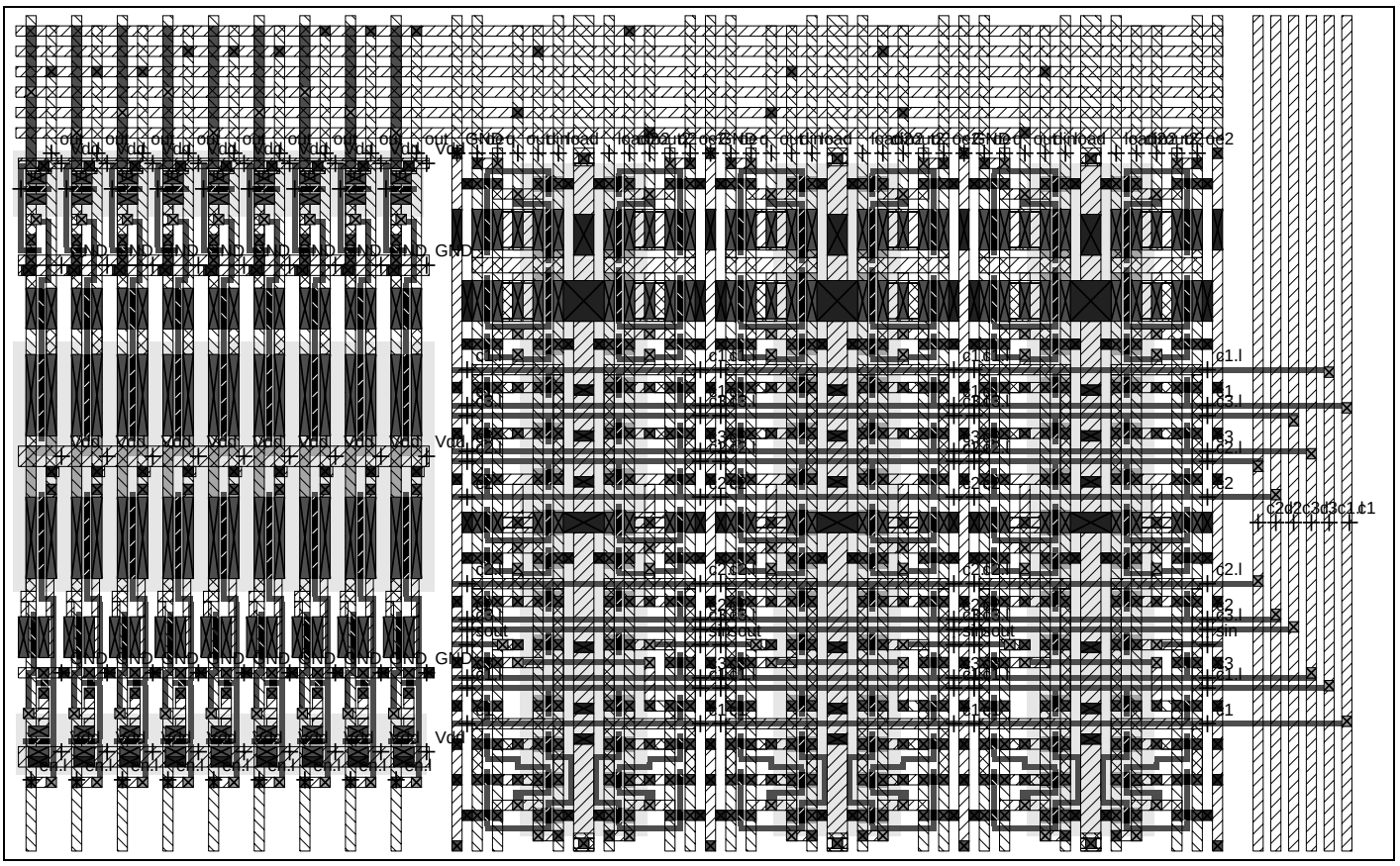

Figure 5: Magic plot of a bit-slice

D COSMOS Script

```
>vector state in1 in2 in3 in4
>vector rega qa1 qa4
>vector regb qb1 qb4
>vector regc qc1 qc4
>vector regd qd1 qd4
>vector rege qe1 qe4
>vector Ibus I1 I2 I3 I4
>
>clock c1.l:1011 c2.l:1110 c3:0000
>
>initialize 1
>
>watch %4 rega regb regc regd rege
>
>set %h state:0
>set %h Ibus:0
>cycle
1.4| rega:00 regb:XX regc:XX regd:XX rege:XX
>
>set %h state:1
>set %h Ibus:f
>cycle
2.4| rega:00 regb:11 regc:XX regd:XX rege:XX
>
>set %h state:2
>set %h Ibus:f
>cycle
3.4| rega:00 regb:11 regc:11 regd:XX rege:XX
>
>set %h state:3
>set %h Ibus:f
>cycle
4.4| rega:00 regb:11 regc:11 regd:11 rege:XX
```

```
>
>set %h state:4
>set %h Ibus:0
>cycle
5.4| rega:00 regb:11 regc:11 regd:11 rege:00
>
>set %h state:5
>cycle
6.4| rega:11 regb:11 regc:11 regd:11 rege:00
>
>set %h state:1
>set %h Ibus:0
>cycle
7.4| rega:11 regb:00 regc:11 regd:11 rege:00
>
>set %h state:6
>cycle
8.4| rega:00 regb:11 regc:11 regd:11 rege:00
>
>set %h state:1
>set %h Ibus:0
>cycle
9.4| rega:00 regb:00 regc:11 regd:11 rege:00
>
>set %h state:7
>cycle
10.4| rega:00 regb:00 regc:00 regd:00 rege:00
>
>set %h state:2
>set %h Ibus:f
>cycle
11.4| rega:00 regb:00 regc:11 regd:00 rege:00
>
>set %h state:1
>set %h Ibus:f
```

```
>cycle
12.4| rega:00 regb:11 regc:11 regd:00 rege:00
>
>set %h state:8
>cycle
13.4| rega:00 regb:11 regc:11 regd:11 rege:11
>
>set %h state:4
>set %h Ibus:0
>cycle
14.4| rega:00 regb:11 regc:11 regd:11 rege:00
>
>set %h state:9
>cycle
15.4| rega:11 regb:11 regc:11 regd:11 rege:11
>
>set %h state:4
>set %h Ibus:0
>cycle
16.4| rega:11 regb:11 regc:11 regd:11 rege:00
>
>set %h state:a
>cycle
17.4| rega:11 regb:00 regc:11 regd:11 rege:11
```

E SPICE Simulation

```
.
.
* GND           0
Vdd             1 0 DC 5V
VCMOSN          7 0 DC 0V
VCMOSP          5 0 DC 5V

Vc1             80 0 PULSE(0 5 10NS 0NS 0NS 10NS 40NS)
Vc1.l           81 0 PULSE(5 0 10NS 0NS 0NS 10NS 40NS)
Vc2             85 0 PULSE(0 5 30NS 0NS 0NS 10NS 40NS)
Vc2.l           86 0 PULSE(5 0 30NS 0NS 0NS 10NS 40NS)
Vc3             82 0 DC 0V
Vc3.l           83 0 DC 5V

Vin1            43 0 DC 0V
Vin2            47 0 PULSE(0 5 80NS 0NS 0NS 40NS)
Vin3            48 0 PULSE(0 5 80NS 0NS 0NS 80NS)
Vin4            49 0 DC 0V

Vregle2_0/sin   92 0 DC 0V
Vregle2_0/din2  99 0 DC 0V
Vregle2_0/oe2   71 0 DC 5V
Vregle2_0/load2 96 0 DC 5V

VI              148 0  PULSE(0 5 40NS 0NS 0NS 80NS)
* s0            72
* s1            102
* d0            98
* d1            119
* d2            138

.TRAN .5NS 120NS 40NS
.
.
.
```

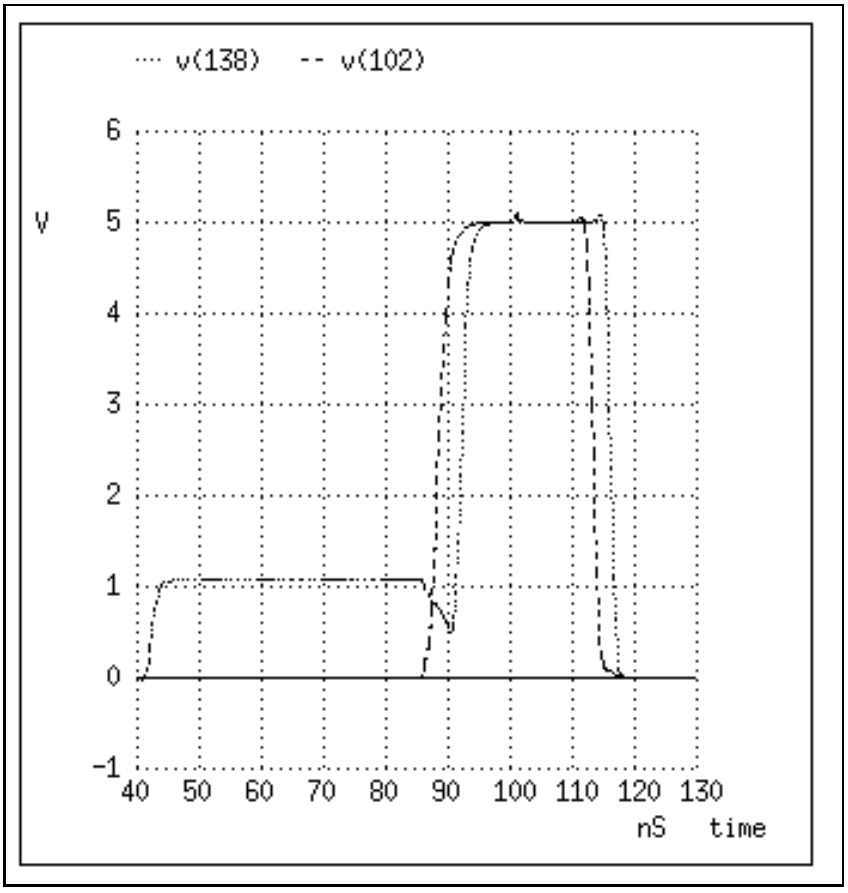

Figure 6: SPICE plot for buses s1(102) and d2(138)

F Pin Configuration

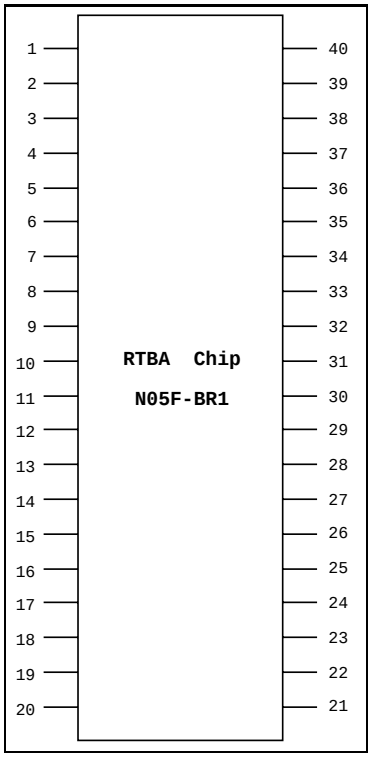

Figure 7: Pin Configuration for the RTBA chip

Pin #	*Description*	*Pin #*	*Description*	*Pin #*	*Description*	*Pin #*	*Description*
1	Slice1/qe	11	Slice4/qd	21	PLA/input3	31	Slice1/s1
2	Slice2/sin	12	Slice4/qc	22	PLA/input2	32	Slice1/I
3	Slice2/I	13	Slice4/qb	23	PLA/input1	33	Slice1/d0
4	Slice3/sin	14	Slice4/qa	24	Clock/c2.low	34	Slice1/d1
5	Vdd	15	Vdd	25	GND	35	GND
6	Slice3/I	16	Slice4/sout	26	Clock/c3	36	Slice1/d2
7	Slice4/sin	17	Slice3/sout	27	Clock/c1.low	37	Slice1/qa
8	Slice4/I	18	Slice2/sout	28	Slice1/sin	38	Slice1/qb
9	Slice4/qe	19	Slice1/sout	29	Slice1/s0	39	Slice1/qc
10	GND	20	PLA/input4	30	Vdd	40	Slice1/qd

G Logic Engine Programs

```
/******* generic.c **********/
#include <stdio.h>
#include "\le\include\lelib.h"
#include <time.h>

#define ALUAND 0x10
#define ALUOR  0x20
#define ALUADD 0x42
#define ALUSUB 0x43
#define ALUXOR 0x40
#define ALUSHR 0x08
#define ALUSHL 0x04
#define A 0
#define B 1
#define C 2
#define D 3
#define E 4
#define F 5

void main(void);
void dotest(void);
void put_random(void);
void sh_Clock(int n);
void Clock(int n);
int rand(void);
void read_reg(void);
void print_reg(void);
void move_arr(short source[4][6],short destination[4][6]);
void Failure(int n);

short regs[4][6],temp[4][6],new_val[4][6];
int new_i;
int comm;
void main()
{
   if(declare("generic.dec")!=0){
      fprintf(stderr,"Error in declaration file\n");
      exit(-1);
   }
```

```
   while(initboard()<0){
      printf("LE board is not connected\n<enter>");
      getchar();
   }
   dotest();
   restoreboard();
}

void Failure(int n)
{
   printf("Fail %d\n",n);
   printf("<enter>");getchar();
   exit(1);
}

void sh_Clock(int n)
{
int i;
for(i=0;i<n;i++)
      {
         mask("C2.L=1, C3=0");
         mask("C2.L=1, C3=1");
         mask("C2.L=1, C3=0");
         mask("C2.L=0, C3=0");
    }
}

void Clock(int n)
{
int i;
for(i=0;i<n;i++)
      {
         mask("C1.L=1, C2.L=1");
         mask("C1.L=0, C2.L=1");
         mask("C1.L=1, C2.L=1");
         mask("C1.L=1, C2.L=0");
    }
}

void read_reg(void)
```

```
{
int i;
   for (i=0; i<6; i++)
   {
      regs[0][i] = readval("SOUT1");
      regs[1][i] = readval("SOUT2");
      regs[2][i] = readval("SOUT3");
      regs[3][i] = readval("SOUT4");
      command("SIN1=%d,SIN2=%d,SIN3=%d,SIN4=%d",
         regs[0][i],regs[1][i],regs[2][i],regs[3][i]);
      sh_Clock(1);
   }
}

void put_random(void)
{
   int i;
   for(i=0;i<6;i++)
   {
      command("SIN1=%d,SIN2=%d,SIN3=%d,SIN4=%d",
         1 & rand(),1 & rand(),1 & rand(),1 & rand());
      sh_Clock(1);
   }
}

void print_reg(void)
{
   int i,j;
   printf("\nslice   A    B    C    D    E    F\n");
   for(i=3;i>=0;i--)
   {
      printf("  %d     ",i+1);
      for(j=0;j<6;j++)
         printf("%d    ",regs[i][j]);
      printf("\n");
   }
}

void move_arr(short source[4][6],short destination[4][6])
{
```

```
   int i,j;
   for (i=0;i<4;i++)
      for(j=0;j<6;j++)
         destination[i][j] = source[i][j];
}

void move_to(int source, int dest)
{
   int i;
   for (i=0;i<4;i++)
      temp[i][dest] = new_val[i][source];
}

void compare(void)
{
   int i,j,k,l;
    for (i=0;i<4;i++)
      for(j=0;j<5;j++)
         if (temp[i][j] != regs[i][j])
            {
               printf("*************** FAILURE **************\n");
               printf("command = %d I = %d\n",comm,new_i);
               printf("Real Values\n");
               print_reg();
               printf("Expected Values\n");
               move_arr(temp,regs);
               print_reg();
               printf("Original Values\n");
               move_arr(new_val,regs);
               print_reg();
               Failure(i*4+j);
            }
}

void put_i(int new_i,int n_reg)
{
   int i;
   for (i=0;i<4;i++)
      temp[i][n_reg] = (0 != (new_i & (1 << (3-i))));
}
```

```
void do_command(int comm)
{
   new_i = rand() % 15;
   command("IN=%d,I=%d",comm,new_i);
   Clock(1);
   switch (comm)
   {
      case 0:
         put_i(new_i,A); break;
      case 1:
         put_i(new_i,B); break;
      case 2:
         put_i(new_i,C); break;
      case 3:
         put_i(new_i,D); break;
      case 4:
         put_i(new_i,E); break;
      case 5:
         move_to(B,A); move_to(D,C); break;
      case 6:
         move_to(E,A); move_to(A,B); break;
      case 7:
         move_to(B,C); move_to(A,D); break;
      case 8:
         move_to(C,D); move_to(B,E); break;
      case 9:
         move_to(D,A); move_to(D,E); break;
      case 10:
         move_to(E,B); move_to(A,E); break;
       default:
          printf("Unkown command %d\n",comm); Failure(0);
       }
}

void begin_comm(void)
{
   move_arr(new_val,temp);
}
```

```
void end_comm(void)
{
   move_arr(temp,new_val);
}

void dotest(void)
{
   int i;
   long j;

   time(&j);
   srand((unsigned) j);
   printf("Board Initialized\n");
   for (j=0;j<40000;j++)
      {
      printf("New Test = %ld\n",j);
      put_random();
      read_reg();
      move_arr(regs,new_val);
      for (i=0;i<35;i++)
         {
            comm = rand() % 10;
            begin_comm();
            do_command(comm);
            end_comm();
            read_reg();
            compare();
         }
      }
   fprintf(stderr,"The test completed with no errors\n");
}

/****** generic.dec ********/
C1.L       SW(31), D=%1
C2.L       SW(30), D=%1
C3         SW(29), D=%0
SIN1       SW(28), D=%0
SIN2       SW(27), D=%0
SIN3       SW(26), D=%0
SIN4       SW(25), D=%0
```

```
I          SW(20:23), D=%0000
IN         SW(16:19), D=%0000
SOUT1      LT(47)
SOUT2      LT(46)
SOUT3      LT(45)
SOUT4      LT(44)
```

H Min-Max Bit-Slice Plot

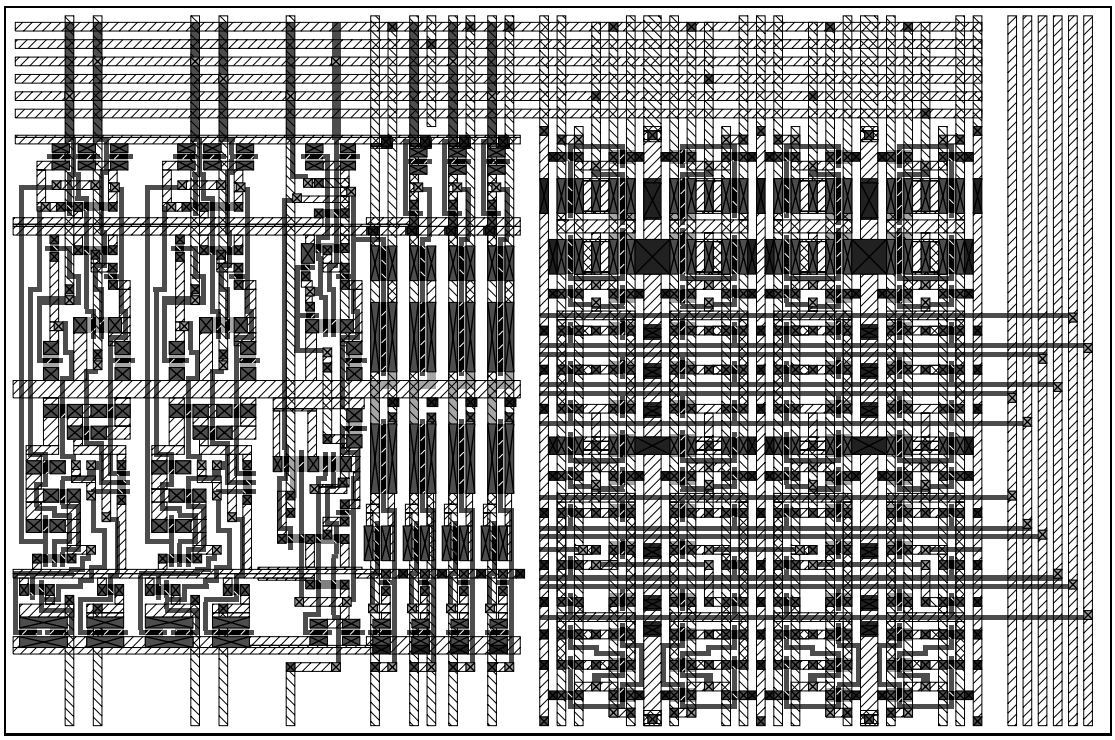

Figure 8: Min-Max bit-slice plot

I Min-Max Control Decode PLA Plot

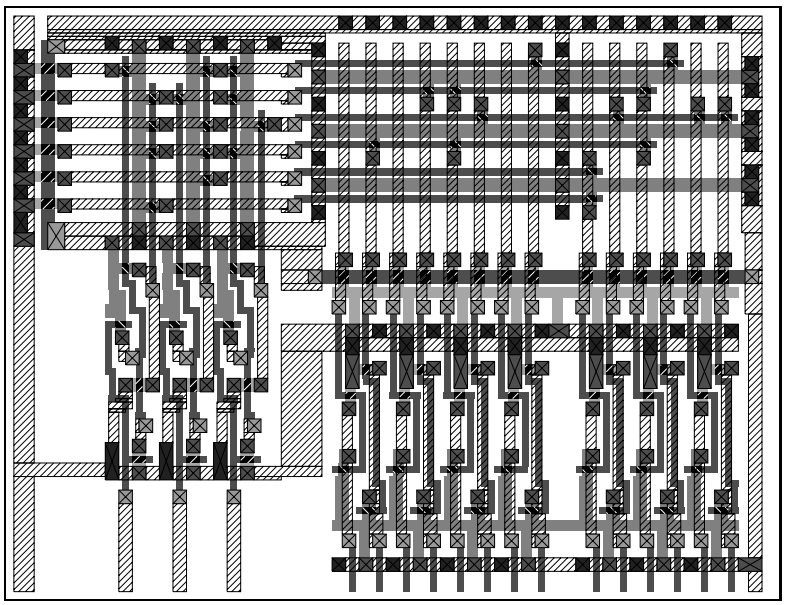

Figure 9: Min-Max Control Decode PLA plot