Printing Requisition / Graphic Services

18585

1. **Please complete unshaded areas** on form as applicable.
2. **Distribute copies** as follows: White and Yellow to Graphic Services. **Retain Pink Copies for your records.**
3. On completion of order the **Yellow copy will be returned** with the printed material.
4. **Please direct enquiries, quoting requisition number and account number,** to **extension 3451.**

TITLE OR DESCRIPTION
CS-89-08 Designing and Modeling VLSI Systems at the Register-Transfer Level

DATE REQUISITIONED: April 11, 1990

DATE REQUIRED:

ACCOUNT NO.: 4 1 2 6 8 8 6 4 1

REQUISITIONER – PRINT: Colleen Bernard for F. Mavaddat

PHONE: 2192

SIGNING AUTHORITY: F. Mavaddat

MAILING INFO – NAME: Colleen Bernard | DEPT.: CS | BLDG. & ROOM NO.: DC 2314 | [X] DELIVER [ ] PICK-UP

Copyright: I hereby agree to assume all responsibility and liability for any infringement of copyrights and/or patent rights which may arise from the processing of, and reproduction of, any of the materials herein requested. I further agree to indemnify and hold blameless the University of Waterloo from any liability which may arise from said processing or reproducing. I also acknowledge that materials processed as a result of this requisition are for educational use only.

NUMBER OF PAGES: 44

NUMBER OF COPIES: 10

TYPE OF PAPER STOCK
[X] BOND [ ] NCR ____ PT. [X] COVER [ ] BRISTOL [X] SUPPLIED [ ] ____

PAPER SIZE
[X] 8½ x 11 [ ] 8½ x 14 [ ] 11 x 17 [ ] ____

PAPER COLOUR
[X] WHITE [ ] ____

INK
[X] BLACK [ ] ____

PRINTING
[ ] 1 SIDE ____ PGS. [X] 2 SIDES ____ PGS.

NUMBERING
FROM ____ TO ____

BINDING/FINISHING: 3 down left side
[X] COLLATING [X] STAPLING [ ] ____ HOLE PUNCHED [ ] PLASTIC RING

FOLDING/PADDING

CUTTING SIZE

Special Instructions

Math fronts and backs enclosed.

COPY CENTRE — OPER. NO. | BLDG. | MACH. NO.

DESIGN & PASTE-UP — OPER. NO. | TIME | LABOUR CODE

				D 0 1
				D 0 1
				D 0 1

TYPESETTING — QUANTITY | OPER. NO. | TIME | LABOUR CODE

P A P 0 0 0 0 0				T 0 1
P A P 0 0 0 0 0				T 0 1
P A P 0 0 0 0 0				T 0 1

PROOF

P R F				
P R F				
P R F				

NEGATIVES — QUANTITY | OPER. NO. | TIME | LABOUR CODE

F L M				C 0 1
F L M				C 0 1
F L M				C 0 1
F L M				C 0 1
F L M				C 0 1

PMT

P M T				C 0 1
P M T				C 0 1
P M T				C 0 1

PLATES

P L T				P 0 1
P L T				P 0 1
P L T				P 0 1

STOCK

				0 0 1
				0 0 1
				0 0 1
				0 0 1

BINDERY

R N G				B 0 1
R N G				B 0 1
R N G				B 0 1
M I S 0 0 0 0 0				B 0 1

OUTSIDE SERVICES

$ ____ COST

TAXES – PROVINCIAL [ ] FEDERAL [ ]

GRAPHIC SERV. OCT. 85 482-2

Printing Requisition / Graphic Services

54241

1. **Please complete unshaded areas** on form as applicable.
2. **Distribute copies** as follows: White and Yellow to Graphic Services. **Retain Pink Copies for your records.**
3. On completion of order the **Yellow copy will be returned** with the printed material.
4. **Please direct enquiries, quoting requisition number and account number,** to **extension 3451.**

TITLE OR DESCRIPTION

Designing and Modeling VLSI Systems at the Register-Transfer Level CS-89-08

DATE REQUISITIONED	DATE REQUIRED	ACCOUNT NO.
Feb. 27/89	ASAP	4 1 2 6 8 6 6 4 1

REQUISITIONER – PRINT	PHONE	SIGNING AUTHORITY
F. Mavaddat	4430	

MAILING INFO – NAME	DEPT.	BLDG. & ROOM NO.	
Sue DeAngelis	C.S.	DC 2314	[X] DELIVER [ ] PICK-UP

Copyright: I hereby agree to assume all responsibility and liability for any infringement of copyrights and/or patent rights which may arise from the processing of, and reproduction of, any of the materials herein requested. I further agree to indemnify and hold blameless the University of Waterloo from any liability which may arise from said processing or reproducing. I also acknowledge that materials processed as a result of this requisition are for educational use only.

NUMBER OF PAGES 44 NUMBER OF COPIES 30

TYPE OF PAPER STOCK
[X] BOND [ ] NCR ____ PT. [X] COVER [ ] BRISTOL [X] SUPPLIED [ ] ____

PAPER SIZE
[X] 8½ x 11 [ ] 8½ x 14 [ ] 11 x 17 [ ] ____

PAPER COLOUR
[X] WHITE [ ] ____

INK
[X] BLACK [ ] ____

PRINTING
[ ] 1 SIDE ____ PGS. [X] 2 SIDES ____ PGS.

NUMBERING
FROM TO

BINDING/FINISHING 3 down left side
[X] COLLATING [X] STAPLING [ ] ____ HOLE PUNCHED [ ] PLASTIC RING

FOLDING/PADDING

CUTTING SIZE

Special Instructions

Math fronts and backs enclosed.

COPY CENTRE OPER. NO. | BLDG. | MACH. NO.

DESIGN & PASTE-UP OPER. NO. | TIME | LABOUR CODE
D 0 1
D 0 1
D 0 1

TYPESETTING QUANTITY
PAP 0 0 0 0 0 | T 0 1
PAP 0 0 0 0 0 | T 0 1
PAP 0 0 0 0 0 | T 0 1

PROOF
PRF
PRF
PRF

NEGATIVES QUANTITY | OPER. NO. | TIME | LABOUR CODE
FLM C 0 1
FLM C 0 1
FLM C 0 1
FLM C 0 1
FLM C 0 1

PMT
PMT C 0 1
PMT C 0 1
PMT C 0 1

PLATES
PLT P 0 1
PLT P 0 1
PLT P 0 1

STOCK
0 0 1
0 0 1
0 0 1
0 0 1

BINDERY
RNG B 0 1
RNG B 0 1
RNG B 0 1
MIS 0 0 0 0 0 B 0 1

OUTSIDE SERVICES

$ ____ COST

TAXES – PROVINCIAL [ ] FEDERAL [ ]

Designing and Modeling VLSI Systems at the Register-Transfer Level

Farhad Mavaddat

Dept. of Computer Science
University of Waterloo

Research Report CS-89-08
February, 1989

Designing and Modeling VLSI Systems at the Register-Transfer Level

Farhad Mavaddat

VLSI Group
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

ABSTRACT

Register transfer (RT) abstractions have several properties which make them particularly suitable for modeling VLSI designs. Nonetheless, compared to the success and widespread acceptability of *circuit*, *switch*, and *logic* abstractions, the RT abstraction has received limited recognition. In this paper we discuss some of the properties that a design abstraction must have in order to be accepted, and make suitable proposals for the RT abstraction.

To meet this goal, we propose a small set of design primitives with well-defined behaviors, and give rules for their composition, all within the framework of a strongly-typed signal environment known as the SDC Model of register-transfer design. We use the typed nature of the signal environment to enforce correct design compositions.

The primitives were chosen to permit a mathematical treatment of designs, but they are also highly design-oriented, and provide the designer with a useful and friendly set of building blocks. We also present user-friendly graphical and textual interfaces to the model.

To prove our designs correct, we use Algorithmic State Machines (ASM) as the frameworks for analyzing and reasoning about an SDC-based design. We employ a slightly modified version of the Floyd-Hoare method of inductive assertions to reason about the ASM diagrams. To this effect, we establish a link between SDC-based designs and their corresponding ASM diagrams, by formulating the SDC designs in terms of a functional model influenced by Gordon's work at Cambridge. We present the model from a different point of view, and in a way closer to the reality of RT designs.

Designing and Modeling VLSI Systems at the Register-Transfer Level

Farhad Mavaddat

VLSI Group
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

1. Motivation

The register transfer abstraction has a number of properties which make it suitable for modeling VLSI designs. Among these properties are the following:

1) RT models are a high-level abstraction with flexible modeling power, and lead to concise definitions; the importance of this property is self-evident, given the size of most VLSI designs.
2) RT abstractions are suited to capturing design regularities, a desirable feature of VLSI design [1].
3) Most other design abstractions present difficult placement and routing problems, while RT-type designs enjoy a natural and efficient layout strategy.
4) RT-based specifications are close to the normal design experience, thereby simplifying the design process and facilitating communication between designers.

These advantages have already been recognized, and RT level designs have been advocated by respected research institutions, as well as individual researchers [2-8]; several serious efforts have been made to apply them to real design environments [9-11]. Nonetheless, RT designs have received limited recognition, compared to the success and widespread acceptability of *circuit*, *switch*, and *logic* abstractions. We suggest several factors which may have contributed to this in the following section.

2. RT-Design: Problems and Prospects

For a design abstraction to be accepted, it has to provide designers with two environments, one design oriented and informal, the other analytical and formal, with well-defined, preferably automatable, mapping rules between the two.

The informal design environment should be user friendly and powerful. This is achieved by providing the designer with well-defined and simple, yet powerful, building blocks, and rules for their compositions. The building blocks form the given abstraction's *design*

primitives. Lumped circuit components (resistors, capacitors, etc.), switch level components (switches, attenuators, and wells), and logic gates (i.e. AND, OR, and NOT gates), are examples of such design primitives.

In addition to the rules for combining primitives into large systems, the designer should also be given a set of guidelines, namely, a set of *do's* and *don't's*, for producing good or acceptable designs in a reasonable time. We often refer to these guidelines as a *design methodology*; hierarchical design is a useful example of such a guideline. Other digital abstraction guidelines include the Mead-Conway methodology for nMOS designs, the synchronous design technique for facilitating design analysis, and the Huffman model to help with sequential synthesis. With this mind, we have proposed a small set of RT primitives and a design methodology which satisfies the requirements discussed above; we have also shown their application to a number of design examples [12-14].

The formal environment should be an effective mathematical system for analysing, manipulating, and verifying the design. Different mathematical tools are applied to distinct aspects of a design. The tools may already exist, as in the use of differential equations, boolean algebra, and lattice theory for *circuit*, *logic*, and *switch* models, respectively; it may be tailored to the abstraction starting from some existing mathematical technique, such as the use of denotational semantics in modeling register-transfer behaviour [8]; or it may even be invented for that purpose, for instance, Milner's Calculus of Communicating Systems [15].

Finally, there should exist simple, effective, and hopefully automatable, mapping methods between the design structures and the formalisms used in their analysis. Some design methodologies aim to make this mapping simpler, or even possible. For example, one can point to network theory, which deals with mappings from *circuit* structures into complex variables and differential equations. The existence of Spice-like simulators suggests the degree to which such mappings can be automated. At the *switch* level, the results are still new, and have not reached the maturity enjoyed by network theory; nonetheless, Hayes [16], Bryant [17], and Brzozowski [18] propose methods for mapping from *switch* level structures into lattices, boolean equations, and graphs, respectively. At the *logic* level, the mappings between the gate networks and Boolean algebra are so direct and natural that the differences between the two are often obscured. In the rest of this section we point to a few ad-hoc developments at the RT-level of design and analysis, and argue that the link between design-oriented works and mathematical models, if any, is still very weak; this is, in our opinion, the main reason behind the apparent lack of RT design maturity.

The most well-known efforts to use the RT abstraction as a design medium are: the CMU design automation activities [5, 6], work at Karlsruhe and Kaiserslautern under Hartenstein [19], and several ad-hoc silicon compiler developments at Caltech [9] and MIT [10]. To the best of our knowledge, none of these design-oriented developments are supported by or linked to a semantic model that can be used for analysis and

reasoning, nor have the mathematical models proposed to-date [8, 20] been based on a set of useful design primitives or guidelines that lead to sound RT-level designs.

One possible exception to the separation of design efforts from modeling at the register-transfer level is the work by Eveking at Darmstadt [21, 22]. Eveking proposes a mapping from members of the family of CONLAN languages [23] to mathematical structures that are based on: a language in the predicate calculus, a number of logical and non-logical axioms derived from the specification language and the design, and rules of inference. He calls this the corresponding hardware's *theory*. He then argues that the correct statements about the behaviour of a design can be derived as theorems in the corresponding theory. Although used primarily to reason between different levels of abstraction, Eveking's method can be used to reason about the designs specified in an RT-level CONLAN member and their implementations.

What distinguishes our work from that of Eveking is our use of *inductive assertions* and the ASM charts, both well-known techniques, particularly suited to specifying and reasoning about register-transfer designs.

Hanna and Daeche discuss desirable properties of the proof techniques [24], and argue that the "formal systems should already exist", "be powerful and concise", and "not too removed from the digital engineer's intuition." It is our contention that the combination of inductive assertions, algorithmic state machines, and the proposed design environment meets these criteria.

In the remaining part of this paper we present an overview of a register-transfer level digital design and analysis environment, under development at the University of Waterloo, that satisfies many of the goals we believe to be crucial to the success of register-transfer level design. We do this by first presenting the model. This is followed by proposing a high-level user interface to the model. The paper ends by proposing a framework for proving the correctness of designs represented by model, and by proving the correctness of a design discussed throughout different stages of the paper.

3. Mathematical Preliminaries

In this section we use an extension of the Lambda calculus to model digital designs. Later we will use this formalism to map between hardware designs and the corresponding ASM charts. We introduce separate notations for specifying combinational and sequential behaviors, both of which are closely related to the formalism proposed by Gordon [8]. We also present a formalism for specifying composite designs which, although still based on Gordon's work, proposes a very different approach to the specification of *port* names and interconnection *nets*.

3.1. Defining Combinational Modules

We define an m-input, n-output ($m \times n$-*put*) combinational device D , shown in Figure 1.a, by

$$D = \lambda(\eta_1, \eta_2, \cdots, \eta_m).(E_1, E_2, \cdots, E_n), \tag{1}$$

where the right hand side of (1) is a short form for

$$\lambda(\eta_1, \eta_2, \cdots, \eta_m) \,.\, E_i \,, \qquad 1 \leq i \leq n \,,$$

and where η_j , $1 \leq j \leq m$, is the j –*th* input port's value, and $\lambda(\eta_1, \eta_2, \cdots, \eta_m).\, E_k$, $1 \leq k \leq n$, defines the k –*th* output port's value.

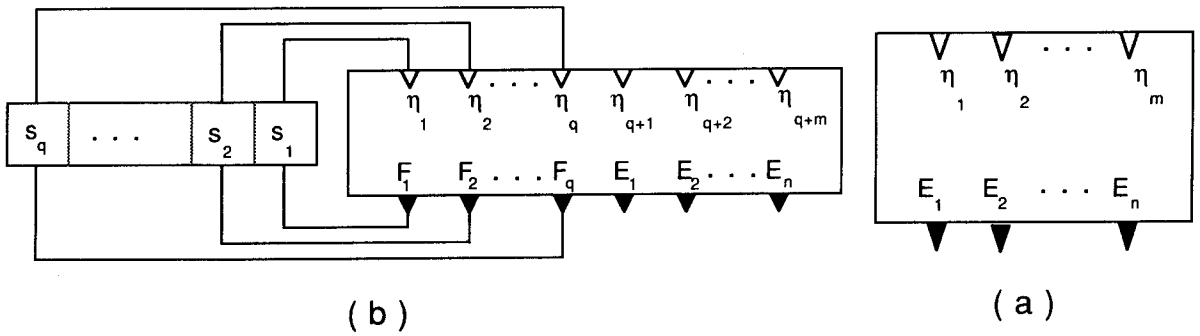


Figure 1. Graphical representation of modules with m inputs and n outputs: a) combinational module, b) sequential module with q state variables s_i , $1 \leq i \leq q$.

3.2. Defining Sequential Circuits

At every state, the behavior of a Mealy-type sequential machine B, shown in Figure 1.b, has two components. The first component is its combinational behavior, B_{cmb}, under the influence of the current state and input ports, and the second component is its next state behavior, B_{seq}, under the influence of the state and input ports at the time of transition to the next state. Therefore, the behavior of an *m×n-put, q-state* sequential machine B can be defined by

$$\left\{ \begin{matrix} B_{cmb} \\ B_{seq} \end{matrix} \right\} = \lambda(\eta_1, \eta_2, \cdots, \eta_q, \eta_{q+1}, \cdots, \eta_{q+m}) . \left\{ \begin{matrix} (E_1, E_2, \cdots, E_n) \\ (F_1, F_2, \cdots, F_q), \end{matrix} \right. \tag{2}$$

where

- the $q + m$ inputs represent the m input-port (environment) and q input-state values.
- $E_1, E_2, \cdots, E_n$ are the n output-port (environment) values produced in response to the corresponding input-port and input-state values at all times.
- $F_1, F_2, \cdots, F_q$ are the q next-state values produced in response to the corresponding input-port and input state-values at every step. They are evaluated at the time of transition to the next state.

Combining the two components of (2) into a single definition, we write

$$B (s_1, s_2, \cdots, s_q) = \lambda(\eta_1,\eta_2, \cdots,\eta_m) . \tag{3}$$
$$((E_1,E_2, \cdots,E_n), B (F_1,F_2, \cdots,F_q))$$

to explain the behavior of the sequential machine B, where:

- to distinguish between the state and the environment inputs, we have moved the input-state bound variables to the left of the equality sign, while keeping the environment inputs on the right side of the definition.
- we write $B (s_1, s_2, \cdots, s_q)$ to represent module B at state $(s_1, s_2, \cdots, s_q)$, and $B (F_1, F_2, \cdots, F_q)$, to define the next-state $(F_1, F_2, \cdots, F_q)$ for B, where F_j, $1 \leq j \leq q$, is the new value for the j *–th* state variable.

We also write

$$B_{cmb} (s_1, s_2, \cdots, s_q) = \lambda(\eta_1,\eta_2, \cdots,\eta_m) . (E_1,E_2, \cdots,E_n) \tag{4}$$

and

$$B_{seq} (s_1, s_2, \cdots, s_q) = \lambda(\eta_1,\eta_2, \cdots,\eta_m) . (F_1,F_2, \cdots,F_q) \tag{5}$$

to represent B*'s* combinational and sequential behaviors, respectively.

3.3. Composite Modules

An *m*×*n-put* composite module $\mathbf{f}^{c}$ is defined as the interconnection of *w* submodules $\mathbf{f}^{0}, \mathbf{f}^{1}, \cdots, \mathbf{f}^{w-1}$, and a (hypothetical) *n*×*m-put* environment module $\mathbf{f}^{w}$, where the input and output ports of $\mathbf{f}^{w}$ define the output and the input ports of $\mathbf{f}^{c}$, respectively, as shown in Figure 2.

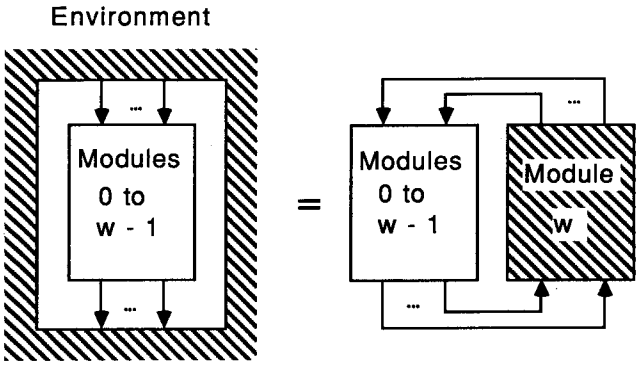


Figure 2. Modeling the environment of modules *0* to *w-1* as the module *w*.

Furthermore, we define

- $\mathbf{I} = \bigcup_{i=0}^{w} I^{i}$ and $\mathbf{O} = \bigcup_{i=0}^{w} O^{i}$ as the set of internal input and output ports, respectively, where I^{i}, $0 \leq i \leq w$, and O^{i}, $0 \leq i \leq w$, are the sets of input and output ports of the i*–th* module.
- $P = \{p_1, p_2, \cdots, p_t\}$ as the set of *nets* used in connecting the submodules, such that $h : \mathbf{O} \cup \mathbf{I} \rightarrow P$ is a total function assigning a single *net* to every port, where $h : \mathbf{O} \rightarrow P$ is *one-to-one* and $h : \mathbf{I} \rightarrow P$ is *onto*.

To model the *net* connections of a module, say $\mathbf{f}^{i}$ ($m^{i} \times n^{i}$ –*put* , q^{i} –*state*), we write

$$(y_1, y_2, \cdots, y_{n^i}) = \mathbf{f}^{i}_{cmb}(s_1, s_2, \cdots, s_{q^i})\,(x_1, x_2, \cdots, x_{m^i}) \tag{6}$$

as a short form for

$$y_j = (\lambda(\eta_1, \eta_2, \cdots, \eta_{q^i}, \eta_{q^i+1}, \eta_{q^i+2}, \cdots, \eta_{q^i+m^i}) \cdot E_j)$$
$$(s_1, s_2, \cdots, s_{q^i}, x_1, x_2, \cdots, x_{m^i}), \quad 1 \leq j \leq n^{i}, \tag{7}$$

where

$$\mathbf{f}^{\,i}_{cmb} = \lambda(\eta_1, \eta_2, \ldots, \eta_{q^i}, \eta_{q^i+1}, \eta_{q^i+2}, \cdots \eta_{q^i+m^i}) \,.\, (E_1, E_2, \cdots, E_{n^i}),$$

and $y_j \in h(O^i)$, $0 \leq j \leq n^i$, and $x_j \in h(I^i)$, $0 \leq j \leq m^i$, are the values of the *nets* connected to the corresponding ports. Thus, the behavior of module $\mathbf{f}^{\,c}$, composed of the interconnection of submodules $\mathbf{f}^{\,0}, \mathbf{f}^{\,1}, \cdots, \mathbf{f}^{\,w}$, using the connection nets P, can be defined as

$$\begin{aligned}\mathbf{f}^{\,c}\ (S^1, S^2, \cdots, S^{w-1}) = \lambda(\, h(O^w)) \,.\, (\ &\textbf{rec} \\ &(\, Y^i = \mathbf{f}^{\,i}_{cmb}(\, S^i)(\, X^i)\quad 1 \leq i \leq w-1\,) \\ &\textbf{in}\ (\, h(\, I^w), \mathbf{f}^{\,c}\ (\, \mathbf{f}^{\,i}_{seq}(\, S^i)(\, X^i)\quad 1 \leq i \leq w-1\,))),\end{aligned} \tag{8}$$

where

$$Y^i = (\, y^i_1, y^i_2, \cdots, y^i_{n^i}),\ y^i_j \in P - h(O^w),\quad 1 \leq j \leq n^i,\ 1 \leq i \leq w-1,$$

and

$$X^i = (\, x^i_1, x^i_2, \cdots, x^i_{m^i}),\ x^i_j \in P,\quad 1 \leq j \leq m^i,\ 1 \leq i \leq w,$$

are the *net* values, $S^i = (s^i_1, s^i_2, \cdots, s^i_{q^i})$ is the set of states of $\mathbf{f}^{\,i}$, q^i is the number of state variables in $\mathbf{f}^{\,i}$, $1 \leq i \leq w$, and **rec** and **in** are defined as in [25].

4. The SDC Model

The power and novelty of SDC-based modeling is characterized by the selection of its design primitives, by the enforceable methodology it expects, by the signal typing it supports, and by the ease by which the SDC-based designs can be translated into the integrated circuit layout. It is the purpose of this section to present the SDC model, and to discuss the significance of its features to the design process. In Section 5 we present a few SDC-based design examples.

4.1. Design Space

We have observed that signals flowing in most register-transfer type designs belong to one of the three categories of *data* (**D**), *control* (**C**), and *status* (**S**). A more detailed explanation of these types and their motivation is as follows:

- **D** signals carry data values from one module of the *data-path* slice to other modules of the same slice. The inputs and outputs of register and ALU slices are examples of **D** signals. **D** signal elements correspond to the basic symbols of data representation; their nature and number will depend on the type of *slice* being defined. For example, $\mathbf{D}_b = \{0, 1\}$ for binary data-path *slices*, and $\mathbf{D}_d = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ for a decimal slice. The **D** signals also form the data inputs and the data outputs of the design. We use solid lines to represent **D** signals in the graphical design examples of this paper.
- **C** signals are the command inputs from the control-unit to the data-path, where they help to dynamically re-configure the data-path and route data values through different paths of a data-path, at different times. The 'load' command to a register, and the 'operation-code' command to an ALU, are examples of the **C** signals. We use dotted lines to represent **C** signals.
- **S** signals are indicators of the status of a data-path. They inform the control-unit of the prevailing conditions inside the data-path. The 'carry' signal out of an adder or the 'greater-than' signal out of a comparator are examples of **S** signals. We represent **S** signals with dashed lines.

In the remainder of this paper we will use lower-case letters to represent signal variables, and '$\langle\,\rangle$', '[ ]', and '{ }' to enclose **S** , **C** , and **D** signals, respectively.

Elsewhere we have generalized the signal typing scheme by assigning each signal type to one of the co-ordinates of a multi-dimensional virtual design space [26, 27], and have shown that the placement and routing issues facing the designer can be formulated in terms of projection strategies from this multi-dimensional space into the two-dimensional space of integrated circuits. The informal graphical notation deals only with the two-dimensional projections of the three-dimensional register-transfer objects.

4.2. The Design Primitives

A model of a synchronous system solely as a network of combinational and unit-delay elements has already been used in theoretical studies of automata. For an example of applying this simple model to the analysis of synchronous designs, see the Leiserson and Saxe paper on the retiming of automata [28].

In this work, we keep the unit-delay element intact, but differentiate among three categories of combinational elements, namely, *selectors*, *controllers*, and *functionals*. Later in this paper we show that this differentiation leads to advantages in the design process, improves design efficiency, and presents a more useful mathematical formulation of the subject. We now present these design primitives and discuss how each new category affects the design process.

4.2.1. Functional Primitives

We refer to the family of primitives responsible for manipulating and transforming the data items in their flow through data-path slice as *functionals*; functionals preserve much of the original character of the combinationals as the data operators.

Stated formally, a *functional-slice* is a $(m + k)\times(n + k)$*–put* combinational device of type $(\mathbf{D}^m \times \mathbf{S}^k) \rightarrow (\mathbf{D}^n \times \mathbf{S}^k)$, where $m \geq 1$, $k, n \in \{0, 1\}$, and $n + k \geq 1$.

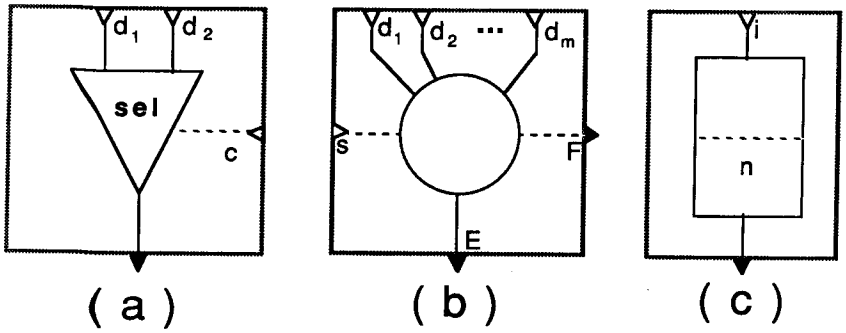


Figure 3. Data-path primitives: a) selectors, b) functionals, c) unit-delays. *Data*, *control*, and *status* signals are shown as solid, dotted, and dashed lines, respectively. Modules are enclosed in patterned boxes to indicate their interface boundaries.

The behavior of a functional primitive can be defined by one of the following three schemes:

$$\lambda\{ d_1, d_2, \cdots, d_m \} \langle s \rangle . \{ E \} \langle F \rangle \tag{9}$$

$$\lambda\{ d_1, d_2, \cdots, d_m \} \langle s \rangle . \langle F \rangle \tag{10}$$

$$\lambda\{ d_1, d_2, \cdots, d_m \} . \{ E \} \tag{11}$$

Depending on the nature of the application, the number and type of operators used in E and F expressions may vary. As an example, *multiplication* might not be allowed when the model is used as the input to a silicon compiler, while its use might be allowed in simulation applications. In a similar way, *addition* might prove to be unacceptable when the model is used for some forms of reasoning about the hardware, but acceptable when the model is intended for silicon implementation.

In Figure 3.b we have shown the graphical symbol used to represent the *functional* primitives corresponding to definition scheme (9). One or more of the signal lines will be missing from Figure 3.b in the case of definition schemes (10) and (11). We now present three typical *functional-slices* , each specified according to one of the definition schemes.

Ex.1- The Binary 'and' Slice

A binary **and** slice is a $\mathbf{D_b}^2 \rightarrow \mathbf{D_b}$ type device defined by

$$\textbf{and} = \lambda\{ a , b \} . \{ a \wedge b \} . \tag{12}$$

Ex.2- The Binary 'comparison' Slice

A binary **comparison** slice is a $\mathbf{D_b}^2 \times \mathbf{S} \rightarrow \mathbf{S}$ type device defined by

$$\textbf{comparison} = \lambda\{ a , b \} \langle s \rangle . \langle s \wedge \overline{(a \oplus b)} \rangle , \tag{13}$$

where a and b are the slice's *data* inputs, s is the *status* input indicating the result of comparisons at more significant slices, and $s \wedge \overline{(a \oplus b)}$ is the status output to the less significant neighboring slice.

Ex.3- The Decimal 'add' Slice

A decimal **add** slice is a $\mathbf{D_d}^2 \times \mathbf{S} \rightarrow \mathbf{D_d} \times \mathbf{S}$ type device defined by

$$\textbf{add} = \lambda\{ a , b \} \langle s \rangle . \{ mod_{10} (a + b + num (s))\} \langle num (s) + a + b > 9 \rangle . \tag{14}$$

Here a and b are the slice's data inputs, s is the carry input from the less significant neighboring slice, $(mod_{10} (a + b + num (s))) \in \mathbf{D_d}$ is the data output, $(num (s) + a + b > 9) \in \mathbf{S}$ is the carry to the more significant neighboring slice, and $num : \mathbf{S} \rightarrow \mathbf{D}$ is a function that converts the *status* signals into their equivalent numerical value, typically a logical '1' to a numerical '1'.

4.2.2. The Selector Primitive

Selectors are used to realize the dynamic nature of the data-path. They route the data items from one of their two *data* inputs to their only *data* output, under the control of their only **C** type input. As will become clear later, recognition of *selectors* as a category of combinational elements, distinct from *functionals* and *controllers*, is crucial to our formulation of register-transfer type designs.

Stated formally, the *selector* slice **sel** (Figure 3.a) is a combinational device of type $(\mathbf{D} \times \mathbf{D} \times \mathbf{C}) \rightarrow \mathbf{D} \times \mathbf{C}$, defined by

$$\mathbf{sel} = \lambda\{d_1, d_2\}[c] \,.\, \{c \rightarrow d_2, d_1\}[c] \,, \tag{15}$$

where '$\rightarrow$', in the context of an expression, stands for the *if-then-else* operator. Definition (15) indicates that the output of a *selector* is equal to one of its two *data* inputs, d_1 or d_2, and the selection is made according to the value of input $c : \mathbf{C}$. The *control* input c is passed to the next slice without change.

4.2.3. The Controller Primitives

Controllers are a family of $p \times q$ *–put* combinational modules, introduced to capture the properties of programmed logic arrays (PLAs), and can be used as the main component of a control-unit.

Syntactically, a *controller* **T** is defined by a two-dimensional array of m columns and n rows, where $m = p + q$, $n \leq 2^p$, $p \geq 0$, $q \geq 1$, and $t_{ij} \in \{1, 0, x\}$, $1 \leq i \leq m$, $1 \leq j \leq n$. **T** is composed of two sub-arrays: **K**, the condition sub-array, and **A**, the action sub-array, of n rows and p and q columns, respectively. Each column of **K** is associated with one of the inputs of the module and each column of **A** with one of the outputs of the module.

Operationally, we define the i *–th* row of **T** to be enabled if the input values match the corresponding **K** entries. The x entries of the table match both the 1 and the 0 values of the input. When the i *–th* row of **T** is enabled, the corresponding elements of **A** appear as the module's outputs. An x output indicates a *floating* output. Certain input values may enable more than one row of the table. When more than one row is enabled, "strong" entries (1 and 0) in the same column should not conflict; the x entries are overruled by strong 1 and 0 entries.

Behaviorally, *controllers* are a family of $p \times q$ *–put* devices of type $\mathbf{C}^s \times \mathbf{S}^t \rightarrow \mathbf{C}^u \times \mathbf{S}^v$, where $p = s + t$ and $q = u + v$, $s \geq 0$, $t \geq 0$, $u \geq 1$, and $v \geq 0$. We can express their behaviour as

$$\lambda[c_1, c_2, \cdots, c_s]\langle s_1, s_2, \cdots, s_t \rangle \,.\, [B_1, B_2, \cdots, B_u]\langle S_1, S_2, \cdots, S_v \rangle \,, \tag{16}$$

where B_i, $1 \leq i \leq u$, and S_i, $1 \leq i \leq v$, are the sums of the products of the bound variables and their complements.

Controllers are either automatically generated as the result of compiling the SDC input language to the internal form (see Section 5), or are explicitly defined whenever the designers wish to do so. We have used the tabular forms in the graphical definition of examples throughout this paper.

4.2.4. The Unit Delay primitive

Unit-delays, as the name implies, delay the transition of values from their input to their output by one clock unit without changing the value in any way. They can be used to delay signals of all types although, within our methodology, their use is limited to that of delaying the *data* signals within the data-path, and *status* and *control* signals within control-units. Stated formally, a *unit-delay* **del** is a 1×1*–put* , single-state, polymorphic [29] sequential device (see Figure 3.c) defined by

$$\mathbf{del}\ (\ n\ ) = \lambda(\ i\ )\ .\ (\ n\ ,\ \mathbf{del}\ (\ \mathrm{i}\ ))\ . \tag{17}$$

4.3. Design Methodology

To enforce some discipline in connecting the ports of the submodules of a composite module, we extend the concept of typed *ports* to that of typed *nets*. This assumes that the *nets* of a particular type connect ports of the similar type.

In section 4.3.1 we will develop a formalism for typed composition of submodules, and give a formal definition for a data-path 'slice'. This is followed by a definition of data-paths in section 4.3.2, where we also discuss an important property of slice-based design, exploited so far without proper reference to the underlying assumptions.

4.3.1. Typed connections

We continue to use the parenthesis pairs [ ], and { } and ⟨ ⟩, to enclose *control*, *data*, and *status* -type *nets*, respectively.

In this spirit we partially re-write (6) as

$$\{\ y_1, y_2, \cdots, y_{n_d}\} = \mathbf{f}_{\ cmb}\ (\ s_1, s_2, \cdots, s_q\ )(\ x_1, x_2, \cdots, x_m\ )\ , \tag{18}$$

to emphasize the *data* type connections of the output ports of **f**, where n_d is the number of *data* outputs of **f**.

We can specify the *control* and *status* type connections in a similar way, including the inputs if needed, and extend the definition (6) into a single statement of the form

$$\{\ Y_D\ \}[\ Y_C\ ]\langle\ Y_S\ \rangle = \mathbf{f}_{\ cmb}\ (\ S\ )\{\ X_D\ \}[\ X_C\ ]\langle\ X_S\ \rangle. \tag{19}$$

We also write

$$\{\mathbf{f}_{cmb}(S)\{X_D\}[X_C]\langle X_S\rangle\},$$

$$[\mathbf{f}_{cmb}(S)\{X_D\}[X_C]\langle X_S\rangle],$$

and

$$\langle \mathbf{f}_{cmb}(S)\{X_D\}[X_C]\langle X_S\rangle\rangle$$

to refer to the *net* sets: Y_D , Y_C , and Y_S , respectively.

Extending this convention to the sequential behavior, we write

$$\mathbf{f}_{seq}(S)\{X_D\}[X_C]\langle X_S\rangle \tag{20}$$

to refer to the next-state values of **f**.

We call the composition of $s+1$ sub-modules $F = \{f^0, f^1, f^2, \cdots, f^{s-1}, f^s\}$, where f^s is the environment sub-module, a *data* (*control, status*) composition of F if and only if the corresponding $h(\mathbf{O} - \mathrm{O}^s) \cap h(\mathbf{I} - \mathrm{I}^s)$ contains only *data* (*control, status*) type *nets*.

Finally, given *slice* submodules $G = \{g^0, g^1, g^2, \cdots, g^{s-1}, g^s\}$, where g^s is the environment *slice*, a *slice* composition of G is defined to be any *data* composition of G . The *functional, selector* and the *unit-delay* primitives of the SDC model form the set of primitive *slices*.

4.3.2. Multi-Slice Data-Path Definition

Given a data-path *slice* f and a positive integer n , an n -slice data-path is formed by concatenating n such slices along their *control* and *status* ports, as shown in Figure 4.

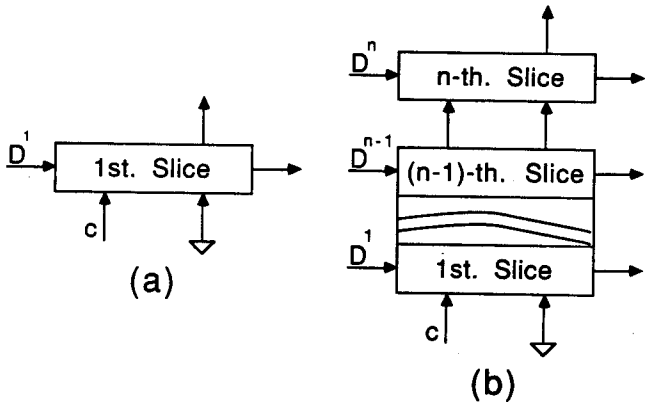


Figure 4. A graphical representation of an *n*-slice data-path. a) A single slice data-path. b) Converting an *n-1* slice data-path to an *n*-slice data-path.

Therefore, the behavior of an n -slice data-path F is defined by

$$
\begin{aligned}
&DP\,(\,f\,,\,n\,) = F\,(S^1, S^2, \cdots, S^n\,) = \\
&\quad \lambda\{D^1, D^2, \cdots, D^n\}[C\,]\,.\,(\\
&\qquad n = 1 \rightarrow (\\
&\qquad\quad \{f_{cmb}\,(S^1)\{D^1\}[C\,]\langle\nabla\rangle\} \\
&\qquad\quad \langle f_{cmb}\,(S^1)\{D^1\}\,[C\,]\langle\nabla\rangle\rangle, \qquad (21) \\
&\qquad\quad f_{seq}\,(S^1)\{D^1\}[C\,]\langle\nabla\rangle\,, \\
&\qquad)\,,\,(\\
&\qquad\quad \{f_{cmb}\,(S^n)\{D^n\}[C\,]\langle\langle DP\,(f\,,\,n-1)\{D^1, D^2, \cdots, D^{n-1}\}[C\,]\rangle\rangle\} \\
&\qquad\quad \langle f_{cmb}\,(S^n)\{D^n\}[C\,]\langle\langle DP\,(f\,,\,n-1)\{D^1, D^2, \cdots, D^{n-1}\}[C\,]\rangle\rangle\rangle, \\
&\qquad\quad f_{seq}\,(S^n)\{D^n\}[C\,]\langle\langle DP\,(f\,,\,n-1)\{D^1, D^2, \cdots, D^{n-1}\}[C\,]\rangle\rangle\,. \\
&\qquad) \\
&\quad)
\end{aligned}
$$

Here $S^i = s_1^i, s_2^i, \cdots, s_q^i, \quad 1 \leq i \leq n$, are the values, and q is the length of the state vectors of the i *–th* slice; $D^i = d_1^i, d_2^i, \cdots, d_m^i, 1 \leq i \leq n$, are the m *data* input values to the i *–th* slice; C is the same control input vector applied to each slice; and ∇ is the first slice's *status* initialization vector.

A few observations are in order at this point:

- Since slices are *identical*, structural concatenations are realized by the abuttment [1] of the corresponding layouts.
- We have assumed that the status information is passed from the lower indexed slices to the higher indexed ones. Assuming that the smallest indexed slice is also the least significant slice of the representation scheme, this formulation satisfies the requirements of certain functionals, such as the carry propagations in a sliced adder.
- A similar formulation exists for cases in which the status signal has to propagate in the opposite direction (for example, a sliced comparator†). Since *control* signals are passed through the slices without any modification, simultaneous flows of both formulations does not lead to infinite recursion.

---

†- A comparator can also be designed with lsb-to-msb signal propagation but it tends to become rather complicated.

- In the past, designers have automatically extended the properties of a *slice* to that of its n-slice data-path. Definition (21) can be used to extend the slice properties to that of the data-path itself, using the structural induction proof.

4.4. SDC-Based Design Examples

In this section we present two examples of SDC-based designs. In each case we present an informal graphical representation of the design and a formal functional model of its behavior.

The first example is simple, representing the data-path of a shift register. This example may be helpful because of the way its shift element is defined. The last example is that of a complete design for calculating the greatest common divisor (GCD) of two positive integers presented at its inputs. This example will also be used in the subsequent sections of this paper.

4.4.1. The "shift-register" Slice:

The following is the behavioral definition of one slice of a shift register, shown in Figure 5. The sub-modules **sel**, and **del** used in the composition of the "shift-register" slice are defined in the usual way.

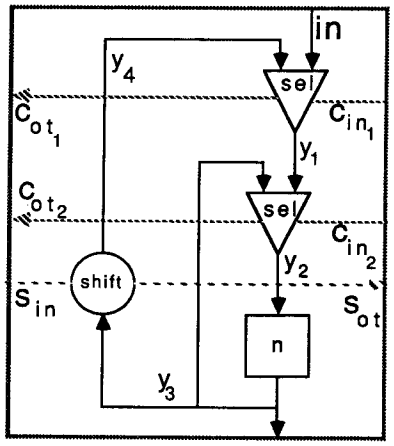


Figure 5. Graphical representation of a shift register slice.

The **shift** slice is a new *functional* primitive defined by

$$\textbf{shift} = \lambda\{in_1\}\langle in_2\rangle \,.\, \{in_2\}\,\langle in_1\rangle \,.$$

We now write the composition rule for the **shift-register** slice, using the above submodules and the *nets* y_1, y_2, y_3, y_4, in, c_{in_1}, c_{in_2}, s_{in}, c_{ot_1}, c_{ot_2}, and s_{ot}. According to (8) we get

$$\begin{aligned}
\textbf{shift -register}\ (\ n\ ) = \lambda\{in\}[c_{in_1}, c_{in_2}]\langle s_{in}\rangle \,.\, (\textbf{rec}\ (\\
&\{y_1\}[c_{ot_1}] = \textbf{sel}_{cmb}\ \{y_4, in\}[c_{in_1}];\\
&\{y_2\}[c_{ot_2}] = \textbf{sel}_{cmb}\ \{y_3, y_1\}[c_{in_2}];\\
&\{y_3\} = \textbf{del}_{cmb}\ (n\ )\{y_2\};\\
&\{y_4\}\langle s_{ot}\rangle = \textbf{shift}_{cmb}\ \{y_3\}\langle s_{in}\rangle)\\
&\textbf{in}\ (\{y_3\}[c_{ot_1}, c_{ot_2}]\langle s_{ot}\rangle\ ,\\
&\quad \textbf{shift -register}\ (\textbf{del}_{seq}\ (n\ )(y_2)))) \,.
\end{aligned}$$

This is expanded to

$$\begin{aligned}
\textbf{shift -register}\ (\ n\ ) = \lambda\{in\}[c_{in_1}, c_{in_2}]\langle s_{in}\rangle \,.\, (\textbf{rec}\ (\\
&y_1 = c_{in_1} \rightarrow in\ ,\ y_4;\\
&c_{ot_1} = c_{in_1};\\
&y_2 = c_{in_2} \rightarrow y_1, y_3;\\
&c_{ot_2} = c_{in_2};\\
&y_3 = n\ ;\\
&y_4 = s_{in}\ ;\\
&s_{ot}\ = y_3\ )\\
&\textbf{in}\ (\{y_3\}[c_{ot_1}, c_{ot_2}]\langle s_{ot}\rangle\ ,\ \textbf{shift -register}\ (y_2))) \,,
\end{aligned}$$

and reduced to

$$\begin{aligned}
\textbf{shift -register}\ (\ n\ ) = \lambda\{\ in\ \}[c_{in_1}, c_{in_2}]\langle s_{in}\rangle \,.\, (\{\ n\ \}[c_{in_1}, c_{in_2}]\langle n\rangle\ ,\\
\textbf{shift -register}\ (\ c_{in_2} \rightarrow (\ c_{in_1} \rightarrow in\ ,\ s_{in}\ )\ ,\ n\ )) \,.
\end{aligned}$$

4.4.2. SDC-based design of a GCD Hardware Module

The **SDC**-based graphical representation of the *GCD* circuit is shown in Figure 6. It calculates the *GCD* of the two values at its data-input *ports* 'in_1' and 'in_2', and producing the result at its data-output *port* '*ot*'.

The input values are sampled at the last assertion of the '*r*' (*reset*) *control* input; the availability of the result is signaled by the first assertion of the '*f*' (*finish*) *status* output. The hardware follows the usual *GCD* algorithm of repeated subtraction of the smaller value from the larger value until the two values match. We will develop the functional models of the data-path and the control-unit parts independently, and then combine them to form the total module's behavioral model.

We start by applying composition rule (8) to the data-path. Given functional primitives

$$\textbf{eql} = \lambda\{ a , b \} . \langle a = b \rangle ,$$

$$\textbf{gt} = \lambda\{ a , b \} . \langle a > b \rangle ,$$

and

$$\textbf{sub} = \lambda\{ a , b \} . \{ a - b \} ,$$

the **gcd_path**, shown in Figure 6.a, is defined by

$$\begin{aligned}
\textbf{gcd_path}\ (a , b) = \lambda\{ in_1, in_2 \} [\ j , k , la , lb\ ] .\ (\textbf{rec}\ (& \\
& \{ y_1 \} = \textbf{sel}_{cmb} \{ y_7, in_1 \} [\ j\ ]; \\
& \{ y_2 \} = \textbf{sel}_{cmb} \{ y_7, in_2 \} [\ j\ ]; \\
& \{ ot \} = \textbf{reg}_{cmb} (a) \{ y_1 \} [\ la\ ]; \\
& \{ y_4 \} = \textbf{reg}_{cmb} (b) \{ y_2 \} [\ lb\ ]; \\
& \langle s_1 \rangle = \textbf{eql}_{cmb} \{ ot , y_4 \}; \\
& \langle s_2 \rangle = \textbf{gt}_{cmb} \{ ot , y_4 \}; \\
& \{ y_5 \} = \textbf{sel}_{cmb} \{ y_4, ot \} [\ k\ ]; \\
& \{ y_6 \} = \textbf{sel}_{cmb} \{ ot , y_4 \} [\ k\ ]; \\
& \{ y_7 \} = \textbf{sub}_{cmb} \{ y_5, y_6 \})\ \textbf{in}\ (\\
& \quad \{ ot \} \langle s_1, s_2 \rangle, \textbf{gcd_path}\ (\textbf{reg}_{seq} (a) \{ y_1 \} [\ la\ ], \\
& \qquad\qquad (\textbf{reg}_{seq} (b) \{ y_2 \} [\ lb\ ]))))\ .
\end{aligned}$$

Note that in this example we have used a single *slice* and a data-path of those *slices* in an interchangeable form. As a result, we have also assumed that the *status* inputs to the data-path receive proper initialization without explicitly referencing them.

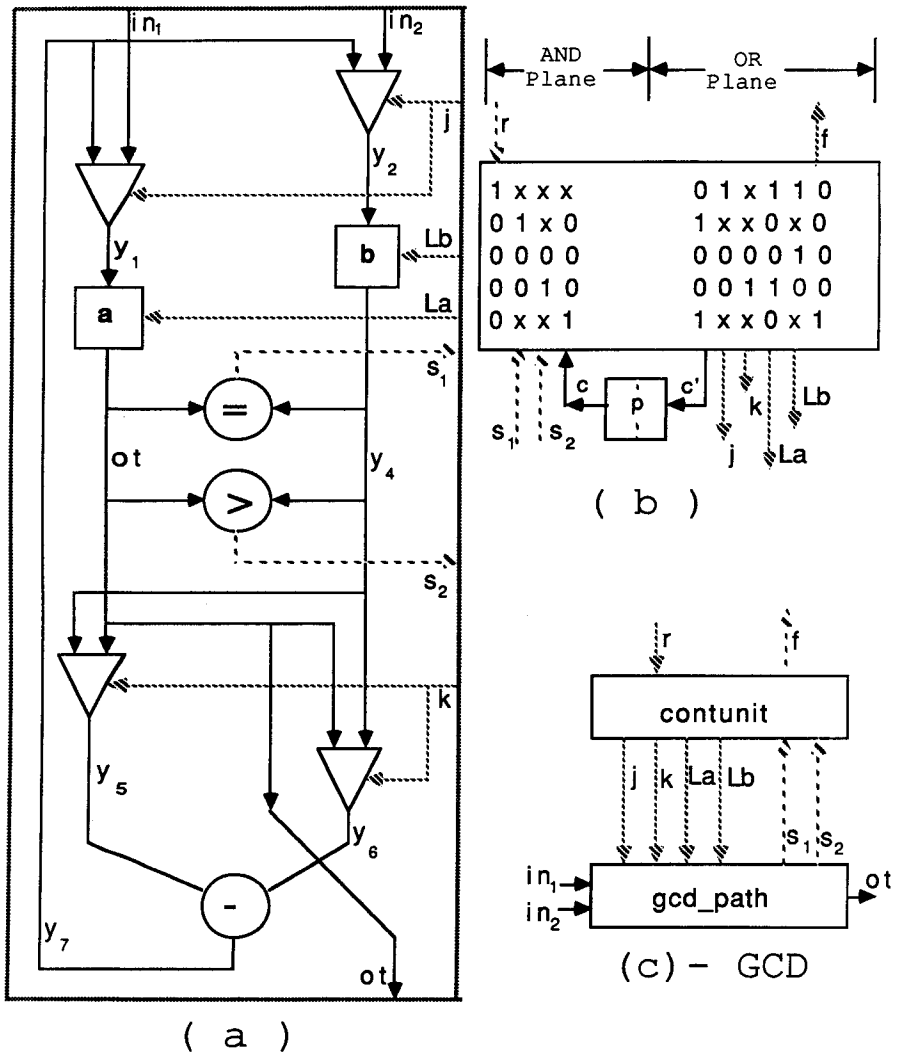


Figure 6. Separate representations of a data-path (a) and a control-unit (b) to calculate the greatest common divisor of two positive integers at inputs in_1 and in_2. Input r signals the start of the computation. Output f signals the availability of the results at ot. The combined form, called GCD, is shown in (c). Boxes labeled as **a** and **b** depict *registers*.

After expansion and simplifications, **gcd_path** behavior reduces to

$$\begin{aligned}\textbf{gcd\_path}\ (\ a\ ,\ b\ ) = \lambda\{\ in_1,\ in_2\ \}[\ j\ ,\ k\ ,\ la\ ,\ lb\ ]\ .\ (\{\ a\ \}\ \langle\ a\ = b\ ,\ a > b\ \rangle,\\ \textbf{gcd\_path}\ ((\ la \rightarrow (\ j \rightarrow in_1,\ (\ k \rightarrow (\ a - b\ ),(b - a\ ))),\ a\ ),\\ (\ lb \rightarrow (\ j \rightarrow in_2,\ (\ k \rightarrow (\ a - b\ ),(b - a\ ))),\ b\ )))\ .\end{aligned} \tag{22}$$

This completes the definition of the data-path part.

The control part consists of two submodules: a PLA and a unit-delay (Figure 6.b). The PLA realizes the microprogram to be executed by the module. The unit-delay holds the state of the control-unit. We start by defining the PLA part, called **pla**, and then combine it with a unit-delay element to form the complete control-part, called **contunit**. These two steps follow:

$$\textbf{pla}\ = \lambda[\ r\ ]\ \langle\ s_1,\ s_2,\ c\ \rangle\ .\ ([\ c\ ',\ j\ ,\ k\ ,\ la\ ,\ lb\ ]\langle\ f\ \rangle)\ ,$$

which is expanded to

$$\begin{aligned}\textbf{pla}\ = \lambda[\ r\ ]\ \langle\ s_1,\ s_2\ ,\ c\ \rangle\ .\ ([\ (\ \bar{r} \wedge \bar{c} \wedge s_1\ ) \vee (\bar{r} \wedge c\ ),\\ r\ ,\ (\ \bar{r} \wedge \overline{s_1} \wedge s_2 \wedge \bar{c}\ ),\ (\ r \vee (\ \bar{r} \wedge \overline{s_1} \wedge s_2 \wedge \bar{c}\ )),\\ (\ r \vee (\ \bar{r} \wedge \overline{s_1} \wedge \overline{s_2} \wedge \bar{c}\ ))]\ \langle\ \bar{r} \wedge c\ \rangle),\end{aligned}$$

and

$$\begin{aligned}\textbf{contunit}\ (\ p\ ) = \lambda[\ r\ ]\langle\ s_1,\ s_2\rangle.(\textbf{rec}\ (\\ [\ y_1,\ j\ ,\ k\ ,\ la\ ,\ lb\ ]\langle\ f\ \rangle = \textbf{pla}_{\ cmb}\ [\ r\ ]\langle\ s_1,\ s_2,\ y_2\ \rangle;\\ (\ y_2) = \textbf{del}_{\ cmb}\ (\ p\ )[\ y_1]\ )\ \textbf{in}\ \ (\\ [\ j\ ,\ k\ ,\ la\ ,\ lb\ ]\ \langle\ f\ \rangle,\ \textbf{contunit}\ (\ \textbf{del}_{\ seq}\ (\ p\ )[\ y_1\ ]))),\end{aligned}$$

which is reduced to

$$\begin{aligned}\textbf{contunit}\ (\ p\ ) = \lambda[\ r\ ]\ \langle\ s_1,\ s_2\ \rangle.([\ r\ ,\ \bar{r} \wedge \overline{s_1} \wedge s_2 \wedge \bar{p}\ ,\\ r \vee (\ \bar{r} \wedge \overline{s_1} \wedge s_2 \wedge \bar{p}\ ),\ r \vee (\ \bar{r} \wedge \overline{s_1} \wedge \overline{s_2} \wedge \bar{p}\ )]\\ \langle\ \bar{r} \wedge p\ \rangle,\ \textbf{contunit}\ ((\bar{r} \wedge \bar{p} \wedge s_1) \vee (\bar{r} \wedge p\ )))\ .\end{aligned} \tag{23}$$

Combining **gcd** and **contunit** to form the complete module, called **gcd**, as shown in Figure 6.c, leads initially to

$$\begin{aligned}\textbf{gcd}\ (\ a\ ,\ b\ ,\ p\ ) = \lambda\{in_1,\ in_2\}[\ r\ ]\ .\ (\ \textbf{rec}\ (\\ \{\ out\ \}\langle\ s_1,\ s_2\rangle = \textbf{gcd\_path}_{\ cmb}\ (\ a\ ,\ b\ )\{\ in_1,\ in_2\}[\ j\ ,\ k\ ,\ la\ ,\ lb\ ];\end{aligned}$$

$$[\ j\ ,\ k\ ,\ la\ ,\ lb\ ]\langle\ f\ \rangle = \textbf{contunit}_{\ cmb}\ (\ p\ )[\ r\ ]\langle\ s_1,\ s_2\rangle)$$

in ($\{\ out\ \}\langle\ f\ \rangle$,

gcd (**gcd_path** $_{seq}$ (a , b)$\{\ in_1,\ in_2\}[\ j\ ,\ k\ ,\ la\ ,\ lb\ ]$,

contunit $_{seq}$ (p)$[\ r\ ]\langle\ s_1,\ s_2\rangle))))$;

this is expanded to

gcd (a , b , p) $= \lambda\{in_1,\ in_2\}[\ r\ ]$. (**rec** (

$out\ = a$;

$s_1 = a\ = b$;

$s_2 = a > b$;

$j\ = r$;

$k\ = \bar{r} \wedge \overline{s_1} \wedge s_2 \wedge \bar{p}$;

$la\ = r \vee (\ \bar{r} \wedge \overline{s_1} \wedge s_2 \wedge \bar{p}\ )$;

$lb\ = r \vee (\ \bar{r} \wedge \overline{s_1} \wedge \overline{s_2} \wedge \bar{p}\ )$;

$f\ = \bar{r} \wedge p\ )$ **in** ($\{\ out\ \}\langle\ f\ \rangle$,

gcd (($la \rightarrow (\ j \rightarrow in_1,\ (\ k \rightarrow (\ a - b\ ),\ (b - a\ )))$, a),

(($lb \rightarrow (\ j \rightarrow in_2,\ (\ k \rightarrow (\ a - b\ ),\ (b - a\ )))$, b)),

$(\ \bar{r} \wedge \bar{p} \wedge s_1) \vee (\ \bar{r} \wedge p\ ))))$,

and can eventually be reduced to

gcd (a , b , p) $= \lambda\{in_1,\ in_2\}[\ r\ ]$. ($\{a\}\langle\ \bar{r} \wedge p\ \rangle$, **gcd** (

$((\ r \wedge q\ ) \rightarrow (\ r \rightarrow in_1,\ (\ q \rightarrow (\ a - b\ ),\ (\ b - a\ )))\ ,\ a\ )$, (24)

$((\ r \wedge q\ ') \rightarrow (\ r \rightarrow in_2,\ (\ q \rightarrow (\ a - b\ ),\ (\ b - a\ )))\ ,\ b\ )$,

$(\ \bar{r} \wedge \bar{p} \wedge s_1 \vee (\ \bar{r} \wedge p\ ))))$

where

$q\ = \bar{r} \wedge \overline{(\ a\ = b\ )} \wedge (\ a > b\ ) \wedge \bar{p}$

$q\ ' = \bar{r} \wedge \overline{(\ a\ = b\ )} \wedge \overline{(\ a > b\ )} \wedge \bar{p}$.

5. The SDCL Hardware Description Language

The SDC model presented so far is a precise and powerful form for specifying RT designs, but it lacks a friendly design interface. The graphic interface used is too informal, and lacks a suitable control specification mechanism; the functional representation, although precise and flexible, is hard to read and write.

To overcome this problem we have developed a hardware description language, called SDCL (for SDC Language), which is tuned to the SDC model and provides a friendly high-level interface to it. This section gives a brief description of this language and its highlights.

5.1. SDCL Objects

SDCL is an object-oriented synchronous RT-level digital design specification language. SDCL objects operate on data values at their *data-input(s)* and produce results at their *data-output(s)*, under the control of zero or more *command-input(s)* (see Figure 7.a).

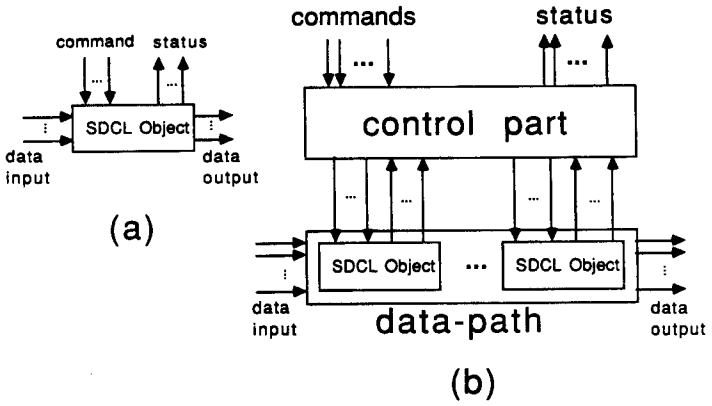


Figure 7. Graphical representation of an SDCL object and its breakdown to a control-unit part and a data-path part. a) The SDCL object. b) The SDCL object with its control-unit and data-path parts defined. The data-path part itself is made of one or more SDCL objects.

Each command-input activation initializes the execution of one or more register transfer steps, which last over as many number of clock periods. An object with no input commands or one whose commands are all inactive, performs the same (default) operation at every clock period. Distinct command executions should not overlap. Objects may also possess one or more status-outputs, which indicate truth values of the assertions made about the internal state of the object.

An example of a complete object, with all four categories of input and output terminals, is a special type of register with the usual data input and output terminals, a 'load' command, and a 'zero' status flag. Later in this section we further explain this register.

SDCL objects consist of a control-unit part and a data-path part, where the data-path part itself consists of one or more objects (see Figure 7.b.) The resulting tree structure of objects is expanded recursively until all the leaf nodes of the tree are primitive objects. Primitive objects are atomic, and we will not attempt to analyze them further into control and data-path parts. The leaf nodes of the expanded design form SDCL's primitive objects, and are either a **Selector**, a **Delay**, or one of several functional objects available to the designer. SDCL is designed to help the designer generate only proper tree-based design hierarchies.

5.1.1. Selectors

A **Selector** is an SDCL primitive object with two data-inputs, one data-output, and a single command input called **enable**. It has no status outputs. **Selector** instances are defined by declaring them as instances of the object **Selector**. If 's' is a **Selector** instance, '**enable**.*s*' selects the second data-input value as the value of the data-output terminal. By default, a non-enabled **Selector** instance will select the first data-input terminal. The data terminals of '*s*' are referenced as the arguments of '*s* (in_1, in_2, *ot*)' where 'in_1' and 'in_2' are the names of the nets connected to the two data-inputs, respectively, and '*ot* ' is the name of its data-output net.

5.1.2. Delay Objects

A **Delay** is an SDCL primitive object, with one data-input and one data-output terminal. It has no command-inputs or status-outputs. A **Delay** object delays the input to output transfer between its two terminals by one clock period for all clock periods. **Delay** instances are generated by declaring them as instances of the **Delay** object. If '*d* ' is a **Delay** instance, its terminals are referenced as the arguments of '*d* (*in* , *ot*)', where '*in* ' and '*ot* ' are the names of the nets connected to the input and output terminals, respectively.

5.1.3. Functional Objects

SDCL *functional* primitive objects have one or two data-inputs, and zero or one data-outputs. None have a command-input, and thus repeat their default operation during every clock period. They may or may not have a single status-output. They must have at least one output of some form.

Instances of SDCL (functional) primitive objects are defined by declaring them as instances of their corresponding SDCL object. If '*f* ' is one such instance, then '**cond**.*f* ' returns the truth value of the only assertion defined on '*f* '. '**cond**.*f* ' is undefined if '*f* ' does not have a status-output. Such errors are detectable at compile time.

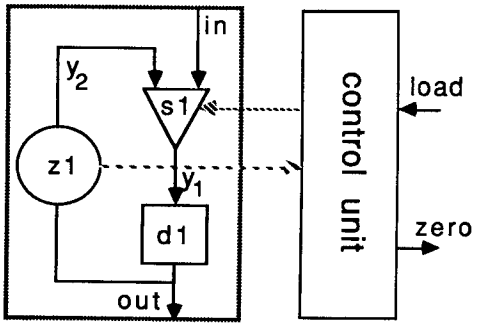
The data terminals of '*f* ' are accessible by '*f* (< *net_list* >)', where the *net_list* elements correspond to the data-input and data-output terminals of '*f* ' in intuitively obvious ways.

5.2. Defining Modules

In this part we use the example of the register with a 'zero' flag, called 'zregister', to explain the writing of SDCL objects. Figure 8.a shows the SDCL definition of the 'zregister'; its graphic form is shown in Figure 8.b.

```
1   module zregister (){
2       s1: Selector;
3       d1: Delay;
4       z1: Ztest;
5       datapath ( in, out )
6       in: I_signal;
7       out: O_signal {
8           y1, y2: node;
9           s1 ( y2, in, y1 );
10          d1 ( y1, out );
11          z1 ( out, y2 )
12      }
13      control () {
14          load: command;
15          zero: status; {
16              load = {
17                  enable.s1
18              }
19              zero = {
20                  cond.z1
21              }
22          }
23      }
24  }
```

(a)



(b)

Figure 8. The definition of a register with zero flag, called 'zregister'. a) The SDCL definition of zregister. b) The graphical representation of zregister.

A composite module's SDCL specification has three sections. The header section (lines 1-4) starts with a module declaration line (line 1), and is followed by the submodule instantiations (lines 2-4).

The data-path section (lines 5-12) starts with a data-path declaration statement (line 5), through which the composite module's data-input and data-output terminals are specified (arguments in line 5) and their types declared (lines 6 & 7). The body of the data-path part (lines 8-11) is used to declare the internal nets, defined as **nodes** (line 8), and a set of connection statements (one for each instantiated submodule), which specify the data-path topology by assigning submodule ports to the net names to which they are connected (lines 9-11). To enforce proper register-transfer design procedures, only the data-in and data-out terminals of each submodule are available inside the data-path section.

Finally, the third section (lines 13-24) is used to define the control and status parts of the definition. This is done by declaring a *command procedure* and a *status function* for each of the commands or status terminals of the composite module. Each function or procedure has a name (e.g. 'load' and 'zero'), which is the name given to the corresponding command or status terminals, and a body, which specifies the steps or tests to be performed in order to execute the procedure or evaluate the function.

The body of a status function is a set of assertions separated by ';', each specifying the condition under which the corresponding function must return the 'true' value. Assertions are well-formed propositions, with the status outputs of submodules as their 'literals', and the proposition operators ($\wedge$, $\vee$) as their operators. In the zregister example, the status function 'zero' has a single literal assertion, **cond**.z1, signifying that the 'zero' function will return a true value if and only if the status output of z1 is true. When a composite module is used in the specification of a higher level module, the status function names defined in this form (if any) are used by the higher level object to form new assertions as needed. A composite module can have one or more status outputs.

The body of each command procedure specifies the sequence of steps needed to complete the task defined by that procedure. Each step in the execution (corresponding to a micro-instruction) is a set of micro-orders separated by ','. Each micro-order, defined syntactically as a '.'-separated concatenation of a command name and a module instantiation name, instructs the corresponding submodule to execute that command. All micro-orders within the same step are executed in parallel. In the zregister example, the command procedure 'load' has a single micro-order, **enable**.s1, signifying that in response to each external activation of the 'load' command it activates the **enable** command input to 's1: **Selector**'. The 'load' command, as defined, has a single execution step.

The execution of any micro-order or groups of micro-orders within a single micro-instruction can be made conditional on the existence of expected conditions in the system, using *if-then* statements. The applicable tests are the same as those used as single assertions within the status functions.

Certain steps within a command procedure can be labeled, and their status ('true' when the corresponding statement is executing, 'false' otherwise) made available to the status functions of the same **control** section. Such a facility, called a *command-status* literal, is defined syntactically as a '.'-separated concatenation of the label and the corresponding function name, and is used in a manner similar to the use of other literals. This enables the status functions to qualify a module's status on the basis of specific steps in execution sequence of commands in the same **control** section.

The control mechanisms *if-then-else* and *while* are used to control the sequence of operations within a command procedure. In both cases, logical statements, similar to the single assertions within the body of status functions, are used as the conditional parts of the control structures.

The second example (Figure 9), involves the SDCL definition of the *GCD* module first shown in Figure 6. The label 'end' within the 'reset' command procedure, and the literal 'end.reset' within the 'finish' status function, exemplify the command-status reporting mechanism discussed above.

```
module gcd() {
        s1, s2, s3, s4: Selector;
        a, b, zregister;
        eq1: equal;
        gt1: greaterthan;
        sub1: subtractor;
        datapath (~ in1, in2, ot )
        in1, in2: I_signal;
        ot: O_signal {
                y1, y2, ot, y4, y5, y6, y7: node;
                s1 ( y7, in1, y1 );
                s2 ( y7, in2, y2 );
                a ( y1, ot );
                b ( y2, y4 );
                s3 ( y4, ot, y5 );
                s4 ( ot, y4, y6 );
                eq1 ( ot, y4 );
                gt1 ( ot, y4 );
                sub1 ( y5, y6, y7 )
        }
        control () {
                reset: command;
                fin: status;
                reset = {
                        enable.s1, load.a, enable.s2, load.b;
                        while ( not cond.eq1 )
                                if cond.gt1 then {
                                        enable.s3, enable.s4, load.a
                                                }
                                                else load.b;
                        end: goto end
                }
                finish = { end.reset }
        }
}
```

Figure 9. The SDCL definition of the GCD module (the graphical representation of the same module is shown in Figure 6). In this design we have used the zregister module shown in Figure 8 without using its "zero" status test capability. We could have used a simpler register.

6. Translation to Layout

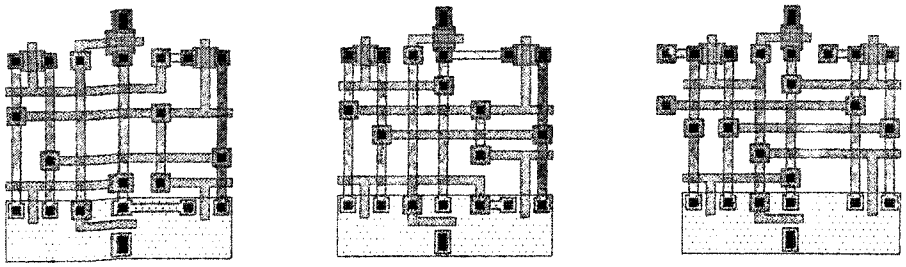
Translating SDC-based designs into their corresponding layout involves three steps. These consist of leaf-cell design, placement, and routing. All these activities have been well-researched in the past, and efficient methods exist for implementing them. Regarding leaf-cell design, SDC uses a fairly standard set of primitives whose layouts are available as standard designs in many IC libraries. The placement and routing of RT designs are also well researched, and all follow a more or less standard and highly efficient scheme reported in most silicon compilation activities.

Due to the inherent efficiency of the RT layouts and the availability of good layout libraries, the three steps are relatively easy to perform, with reasonable overall efficiency; nonetheless, to achieve better results, each step should be performed with attention to the others, and tailored to the specific characteristics of each design. As part of our work we have implemented an experimental data-path generator program that we believe is capable of generating a superior design compared to those achievable through the standard methods discussed above.

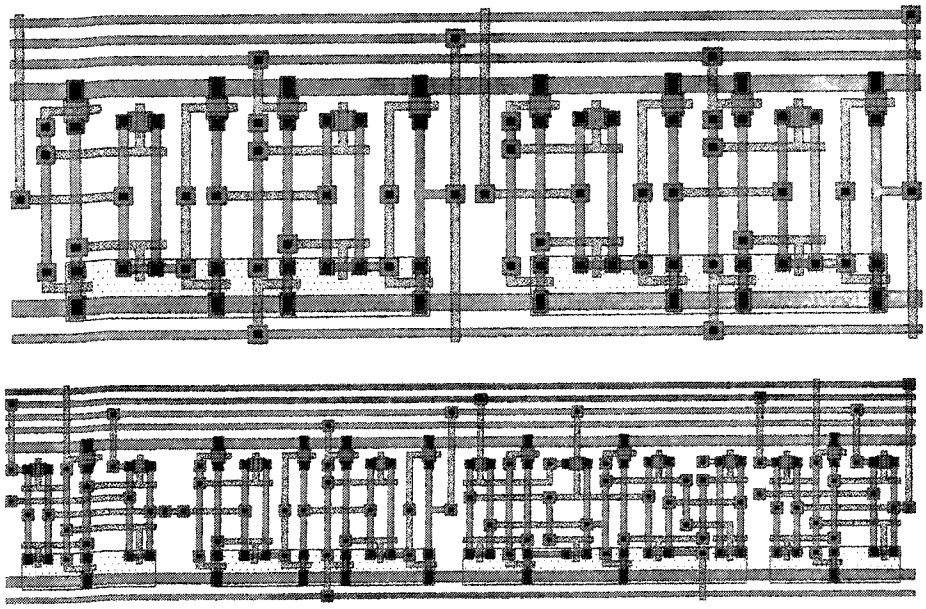
To make the interaction between the data-path program and cell layouts flexible, we have taken the module generator approach to the creation of the leaf-cell layouts. In this approach, each SDC primitive corresponds to a parametrized module generator program written in IC^2 layout language [30, 31]. IC^2 is a locally developed layout language with silicon compilation in mind. In IC^2 each module generator, through its parametrized inputs, can be instructed to generate one of several variations of the functionally equivalent cells, with inputs and outputs strategically placed on a subset of the four edges of the bounding box, and implemented in either metal or polysilicon. In addition, the module generator can be instructed to generate modules of different height, to match the overall slice width to the specific requirements of each design. In Figure 10.a we have shown a number of such leaf cells generated by our module generator program.

On the matter of placement and routing, the optimum linear placement of a number of interconnected modules, with the aim of finding the minimum width solution, has been the subject of study in the past. The models used in these studies are often too simple to reflect the realities of design in a useful way. To overcome this problem, we have implemented a heuristic program within our data-path generator program which places the flattened design (consisting solely of a network of interconnected primitive modules) as closely and as naturally as possible, while taking into account such matters as the commutative nature of inputs, or their need for clock inputs.

A detailed description of the operation of the data-path generator program is beyond the scope of this paper, and will be reported in the future. In Figure 10.b we show two examples of two data-path slices generated by our data-path generator program.



(a)



(b)

Figure 10. Examples of cell and slice layouts generated by the experimental silicon compiler directly from high-level definitions. a) Examples of cell layouts. b) Slice layouts. *Data* lines are generated as horizontal connections above and below the cells array.

7. Correctness of the Implementation

An *algorithmic state machine* (ASM) is a flow-chart representation of the state-transition functions and output-functions of a state-machine [32, 33], and can be regarded as a variation of the state-diagram method for specifying state-machines.

Each state of a state-machine is represented by a unique *state* box in the corresponding ASM chart. A state-machine's transition from one state to the next is represented by the flow of a hypothetical control-pointer from one *state* box to the next. Transitions from a given state in the state-machine to one of several next states are shown in the ASM chart by cascading one or more *condition* boxes at the exit of the originating *state* box. The combination of a *state* box and the *condition* boxes at its output, if any, is called a *state block*, and corresponds roughly to the state circles used in state diagrams.

Each *condition* box contains a proposition on the inputs, and has two exit paths. The choice of exit path, and therefore, the next *state* box, depends on the truth value of the proposition at the time the control-pointer visits the *condition* box.

An ASM's output is a list of signal names, where each name is a command for activating the corresponding signal. The ASM formalism distinguishes between outputs which are activated unconditionally whenever a particular state is reached, and outputs whose activations depend on certain input conditions. When ASM charts are used to specify register-transfer designs, signal names may be replaced by the assignments they activate in the data-path.

Traditionally, an ASM's unconditional outputs are written inside the *state* box in which they occur, while lists of conditional outputs are written inside one of possibly several *conditional output* boxes placed at the appropriate exit of a *condition* box. Later in this paper, we will write the assertions inside the state boxes. To avoid confusion, we move the unconditional output lists from their state boxes to all of the *conditional output* boxes associated with the state boxes. Thus, we will refer to the *conditional output* boxes of our ASM charts simply as *output* boxes. We illustrate the state boxes as solid rectangles, the condition boxes as diamonds, and the output boxes as rectangles with rounded corners.

An ASM chart is particularly suited for specifying RT-type designs, since it explicitly separates a specification into a flow-control-part, representing a design's control-unit, and *output lists*, representing the data-path operations.

Similarities between the ASM specification of hardware and the flow-chart specification of computer programs suggests the use *inductive assertions* [34] for proving the correctness of hardware. This method has several advantages over other hardware proof methods; among these are:

- The existence of a body of experience, know-how, and techniques accumulated over the past two decades.

- A wide user familiarity with the method, which is widely taught for proving correctness of algorithms in computer science and engineering programs.
- The potential to be a more practical tool than it is when applied to software, due to the size of the useful hardware that can be proven correct compared to the size of typical software undergoing a similar proof activity.

However, these advantages are somewhat eroded, for the following reason. ASM-based specifications often go through ad-hoc steps of translation to hardware; thus, unless the translation is fully automatic, confidence in the correctness of ASM representations cannot be transferred to their implementation. To overcome this problem we propose a different approach to the use of ASMs in digital design: instead of using them as inputs to the design activity, we derive them from the appropriately specified designs using the derivation techniques discussed below.

In the remaining parts of this section, we first discuss a method for extending designs to include their input and output strings. We then convert the extended designs into their functional form. This is followed by presenting a method for deriving a module's ASM specification from its functional specification. In the final step, motivated by goals similar to those applicable to the proof of correctness of software, we assign suitable *assertions* to every state box of the derived ASM chart. We then show that the requirement specification will hold between the state variables of the extended module if and when the hardware reaches its output states.

Step 1- Extending Modules to Include I/O Sequences

Assertions on the behaviour of correctly designed modules are made on the sequences of inputs and outputs of those modules. For example, in the case of the GCD hardware of Figure 6, we would expect that: "Each activation of the control input *r* will eventually lead to an activation of the status output *f*, signaling the availability at data output *ot* of the *GCD* of values *m* and *n* present at inputs *in1* and *in2* at the time of *r*'s activation."

Software modules communicate with their environment in an explicit and sequential form through the use of input and output statements. However, hardware modules communicate only in implicit forms, and this complicates the formalization of the above assertion. In order to make explicit the form of the hardware communication, we extend the hardware to include its input and output sequences.

Consider a hardware module *M* and an assertion *A* about some sequence of input-output activities over *M*. The *i/o_extension of M*, denoted by *M '*, is an extension of *M* by a suitable amount of hardware to simulate the generation of inputs and acceptance of outputs according to *A*.

In Figure 11 we show an i/o_extension of the GCD module that includes the sequence of input-output activities described in the foregoing verbal assertion. In this extension, the sequence of values input to *r* of **contunit** (Figure 6), is simulated by the design extension

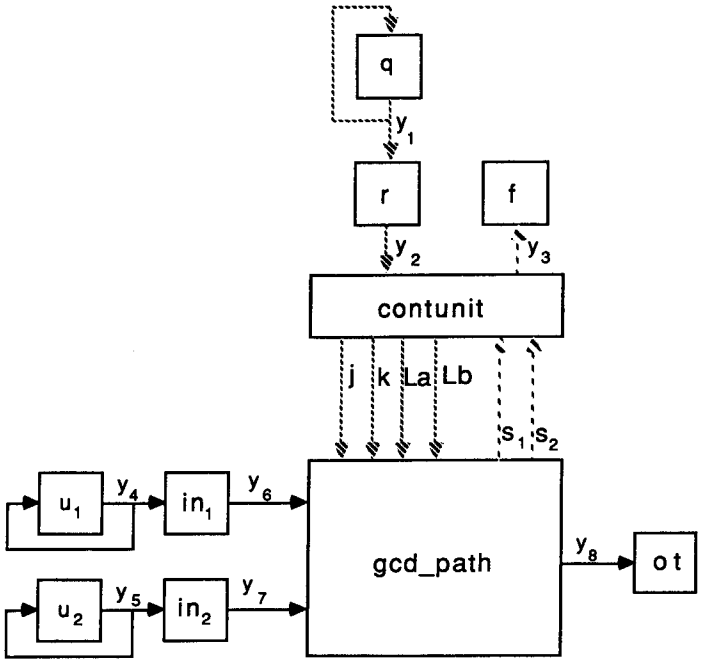


Figure 11. Extending the **gcd** design of Figure 6.c to include its inputs, output, control, and status signal strings.

part corresponding to the unit-delay elements r and q, initialized to 1 and 0, respectively. The choice of the same identifier to refer to an unextended port and its corresponding unit-delay extension (e.g., r in this case) was made for the sake of readability. As a result of this extension, the **contunit** will initially receive a '1' on its r input, followed by an infinite sequence of '0's. This guarantees the proper behaviour of the environment as expected by the input r of **contunit**.

A similar extension of the **gcd_path** with unit-delay elements in_1, in_2, u_1, and u_2, initialized to data values m, n, $\perp$, and $\perp$, respectively, where symbol '$\perp$' indicates an undefined value, simulates the proper input of the data values into the **gcd_path** unit. The f and $gcd(m, n)$ are both single values, so unit-delay elements f and ot are used to represent them, respectively.

The completed extension, called **e_gcd**, lets us reformulate the assertion about the expected behaviour of a correctly operating GCD module, as follows:

"Given the initial relationship

$$(r = 1) \wedge (q = 0) \wedge (in_1 = m) \wedge (in_2 = n)$$

between the states of an extended **gcd** module, and a sufficient number of state transitions, the circuit will eventually reach a new state in which the relationship

$$(ot = gcd (m , n)) \wedge (f = 1)$$

holds between the new states of the extended **gcd**."

Later in this paper, we will show that the output assertion indeed follows the input assertion after a finite number of state transitions. We do so by assigning the two assertions to the input and the output states of the ASM chart corresponding to the extended **gcd** module.

Step 2- Functional Model of The Extended Module

We now write separate functional models for the extended forms of the control-unit and the data-path of a design. In the following formulations, **e_contunit** and **e_gcd_path** refer to the extended forms of the **contunit**, (23), and **gcd-path**, (22), respectively. We have

$$\begin{aligned}
&\textbf{e_contunit}\ (p , q , r , f) = \lambda\langle s_1, s_2 \rangle . (\textbf{rec} (\\
&\qquad [j , k , La , Lb]\langle y_3 \rangle = \textbf{contunit}_{cmb} (p)[y_2]\langle s_1, s_2 \rangle;\\
&\qquad \{ y_1 \} = \textbf{del}_{cmb} (r)\{ y_1 \}\\
&\qquad \{ y_2 \} = \textbf{del}_{cmb} (q)\{ y_1 \}) \textbf{ in } (\\
&\qquad\quad [j , k , La , Lb] , \textbf{e_contunit } (\textbf{contunit}_{seq} (p)[y_2]\langle s_1, s_2 \rangle ,\\
&\qquad\quad \textbf{del}_{seq} (r)\{ y_1 \}, \textbf{del}_{seq} (q)\{ y_1 \}, \textbf{del}_{seq} (f)\{ y_3 \}))).
\end{aligned}$$

We expand and simplify **e_contunit**'s behavioural equations to the following form:

$$\begin{aligned}
&\textbf{e_contunit}\ (p , q , r , f) = \lambda\langle s_1, s_2 \rangle . ([q , (\bar{q} \wedge \bar{p} \wedge \overline{s_1} \wedge s_2),\\
&\qquad (q \vee (\bar{q} \wedge \bar{p} \wedge \overline{s_1} \wedge s_2)) , (q \vee (\bar{q} \wedge \bar{p} \wedge \overline{s_1} \wedge \overline{s_2}))]\\
&\qquad \textbf{e_contunit } ((\bar{q} \wedge \bar{p} \wedge s_1) \vee (\bar{q} \wedge p), r , r , (\bar{q} \wedge p))).
\end{aligned}$$

Similarly,

$$\begin{aligned}
&\textbf{e_gcd_path}\ (a , b , n , m , u_1, u_2, u_3) = \lambda[j , k , La , Lb] . (\textbf{rec} (\\
&\qquad \{ y_4 \} = \textbf{del}_{cmb} (u_1)\{ y_4 \};
\end{aligned}$$

$$\{ y_5 \} = \mathbf{del}_{cmb} (u_2)\{ y_5 \};$$

$$\{ y_6 \} = \mathbf{del}_{cmb} (n)\{ y_4 \};$$

$$\{ y_7 \} = \mathbf{del}_{cmb} (m)\{ y_5 \};$$

$$\{ y_8 \}\langle s_1, s_2 \rangle = \mathbf{gcd\_path} (a , b)\{ y_6, y_7 \}[j , k , La , Lb]) in ($$

$$\langle s_1, s_2 \rangle , \mathbf{e\_gcd\_path} (\mathbf{gcd\_path}_{seq} (a , b)\{ y_6, y_7 \}[j , k , La , Lb],$$

$$\mathbf{del}_{seq} (n)\{ y_4 \}, \mathbf{del}_{seq} (m)\{ y_5 \}, \mathbf{del}_{seq} (u_1)\{ y_4 \},$$

$$\mathbf{del}_{seq} (u_2)\{ y_5 \}, \mathbf{del}_{seq} (u_3)\{ y_8 \}))).$$

This simplifies to

$$\mathbf{e\_gcd\_path} (a , b , n , m , u_1, u_2, u_3) = \lambda[j , k , La , Lb] . ($$

$$\langle (a = b), (a > b) \rangle, \mathbf{e\_gcd\_path} ($$

$$(La \rightarrow (j \rightarrow n , (k \rightarrow (a - b), (b - a))), a),$$

$$(Lb \rightarrow (j \rightarrow m , (k \rightarrow (a - b), (b - a))), b),$$

$$u_1, u_2, u_1, u_2, a)).$$

Step 3- Translating Functional Models into ASM Charts

An extended functional model is translated into a corresponding ASM chart in two phases. The first phase derives the ASM chart's flow-control part, i.e., the interconnection of the *state* and *condition* boxes. The second phase derives the output lists, and completes the chart by adding the *output* boxes.

Given the current state and the environment inputs, we use sequential behaviour model (5) and combinational behaviour model (4) of the extended control-unit to derive the corresponding next state and action outputs. Due to the closed nature of the extension process, the only environment inputs contributing to these derivations are from the data-path parts of the designs. Only a few of the possible next states are ever reachable, due to the special architecture of the extension hardware; so rather than enumerating all possible transitions, we can use a search strategy starting from the input state to save on the amount of computation required.

Considering the **e_contunit** behaviour and definitions (4) and (5), we obtain the following sequential and combinational behaviours:

$$\mathbf{e\_contunit}_{seq} (p , r , q , f) = \lambda\langle s_1, s_2 \rangle . \qquad (25)$$

$$((\bar{r} \wedge \bar{p} \wedge s_1) \vee (\bar{r} \wedge p), q, q, (\bar{r} \wedge p))$$

$$\textbf{e\_contunit}_{cmb}(p, r, q, f) = \lambda\langle s_1, s_2 \rangle . [r, (\bar{r} \wedge \bar{p} \wedge \overline{s_1} \wedge s_2), \quad (26)$$
$$(r \vee (\bar{r} \wedge \bar{p} \wedge \overline{s_1} \wedge s_2)), (r \vee (\bar{r} \wedge \bar{p} \wedge \overline{s_1} \wedge \overline{s_2}))].$$

The results of the search process using (25) and (26) are illustrated in Figure 12.a.

The second phase derives the data-path's register-transfer assignments and the status expressions, and assigns them to the *output* and *condition* boxes, respectively. To do this, the action vectors derived during the first phase are applied to the sequential and combinational models of the data-path; symbolic statements, which indicate the nature of the transfers and the status, are derived and assigned to the *output* and *condition* boxes. These additions complete the derivation of the ASM chart.

Figure 12.b shows the results of applying the **e_contunit**'s action outputs to the following sequential and combinational behaviours of **e_gcd_path**:

$$\textbf{e\_gcd\_path}_{seq}(a, b, in_1, in_2, u_1, u_2, ot) = \lambda[j, k, La, Lb] . ($$
$$(La \rightarrow (j \rightarrow in_1, (k \rightarrow (a - b), (b - a))), a), \quad (27)$$
$$(Lb \rightarrow (j \rightarrow in_2, (k \rightarrow (a - b), (b - a))), b),$$
$$u_1, u_2, u_1, u_2, a)$$

$$\textbf{e\_gcd\_path}_{cmb}(a, b, in_1, in_2, u_1, u_2, ot) = \lambda[j, k, La, Lb] . \langle \quad (28)$$
$$(a = b), (a > b) \rangle.$$

Step 4- Proving the ASM Specification Correct

To verify the ASM chart, and thus the corresponding candidate hardware, we start by proposing a mapping P from the ASM chart's state boxes to propositions whose free variables are the unit-delay names of the candidate hardware. The propositions assigned to the initial and final states of the computation are those known to be true at the start of the computation and expected to be true at the end of the computation, respectively. We refer to these as the 'input' and 'output' states.

Next, we show that for every *state* box i, should the control-pointer starting from the input state reach i, if at all, then $P(i)$ should be true. To prove this, we have to show that for every pair of *state* boxes i and j, where i is a predecessor of j,

$$\{P(i)\}(R, Q)\{P(j)\} \quad (29)$$

holds. In (29), R is the conjunction of zero or more Boolean expressions assigned to the *condition* boxes between i and j, and Q is one or more assignment statements that the control-pointer visits on its path from state box i to state box j. The notation in (29) is due to Hoare [35], and can be interpreted as "If $P(i)$ is *true* and the conditions and the actions specified by R and Q are, respectively, *true* and *executed*, then $P(j)$ must

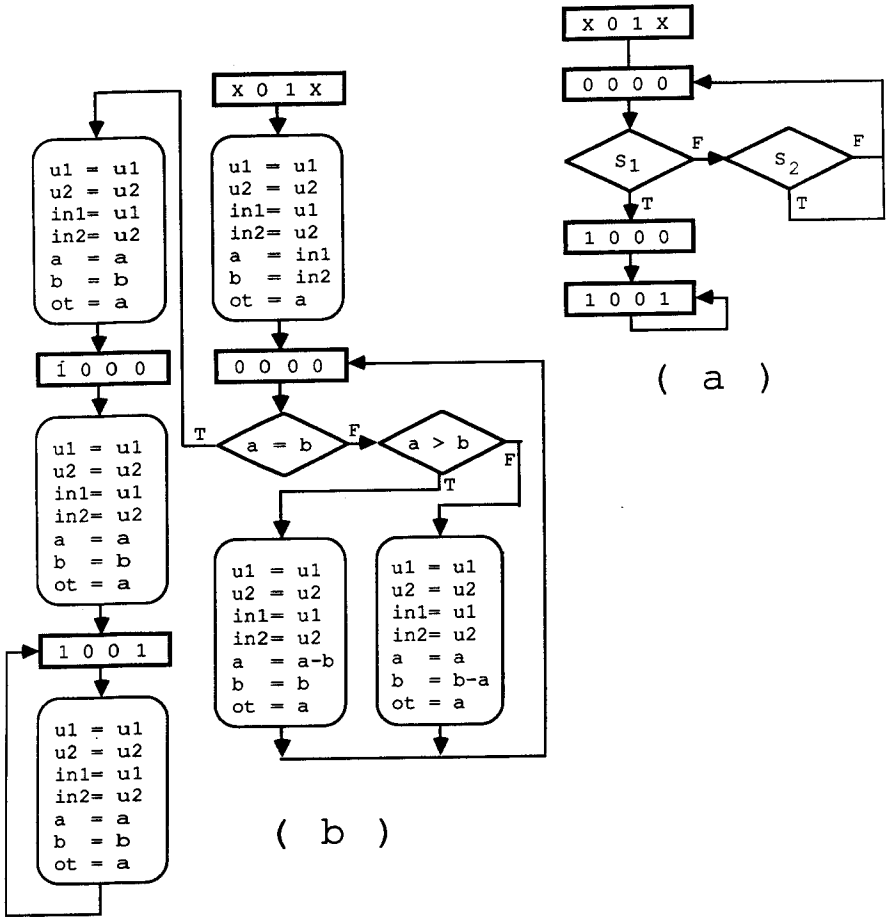


Figure 12. The ASM charts corresponding to the GCD design of Figure 7. (a) The flow-control-part, depicting the behaviour of the control-unit part; strings in each state box represent the values in unit-delays *p*, *q*, *r* and *f* at that state, where 'X' stands for an unknown state. (b) The combined behaviour of the control-unit and the data-path.

also be *true*."

Any rigorous demonstration of this requires a formal definition of the ASM chart, and familiarity with the theory of inductive-assertions; these matters are beyond the scope of this paper. Nonetheless, one can argue informally that, starting from an initial condition satisfying the input proposition, if (29) is proven correct for all adjacent pairs of *state* boxes, then for all subsequent *state* boxes along any path, say k, $P(k)$ is also *true*. Of course, should the candidate hardware reach any of possibly several output *state* boxes, if at all, then the corresponding output proposition must also be *true*.

The arguments needed to show that the path from the input state will eventually lead to an output state are similar to those given for the termination of software programs. In the case of our candidate design we can also verify, by inspection, that the hardware will never falsely signal the availability of the output data. The output state is the only *state* box in which $f = 1$.

Figure 13 shows a version of the ASM chart given in Figure 12.b, with suitable propositions. The reader may wish to verify the propositions, keeping in mind the following properties of the greatest common divisor of integers:

$$a = gcd(a, a)$$

$$gcd(a, b) = gcd(b, a)$$

$$gcd(a, b) = gcd(a + b, b).$$

The search for suitable assertions to be placed at each state box, signifying the expected relationship between the state variables if and when the control pointer visits that box, requires some skill and ingenuity. Of course, this applies to other proof techniques as well, and is by no means unique to the method of inductive assertions.

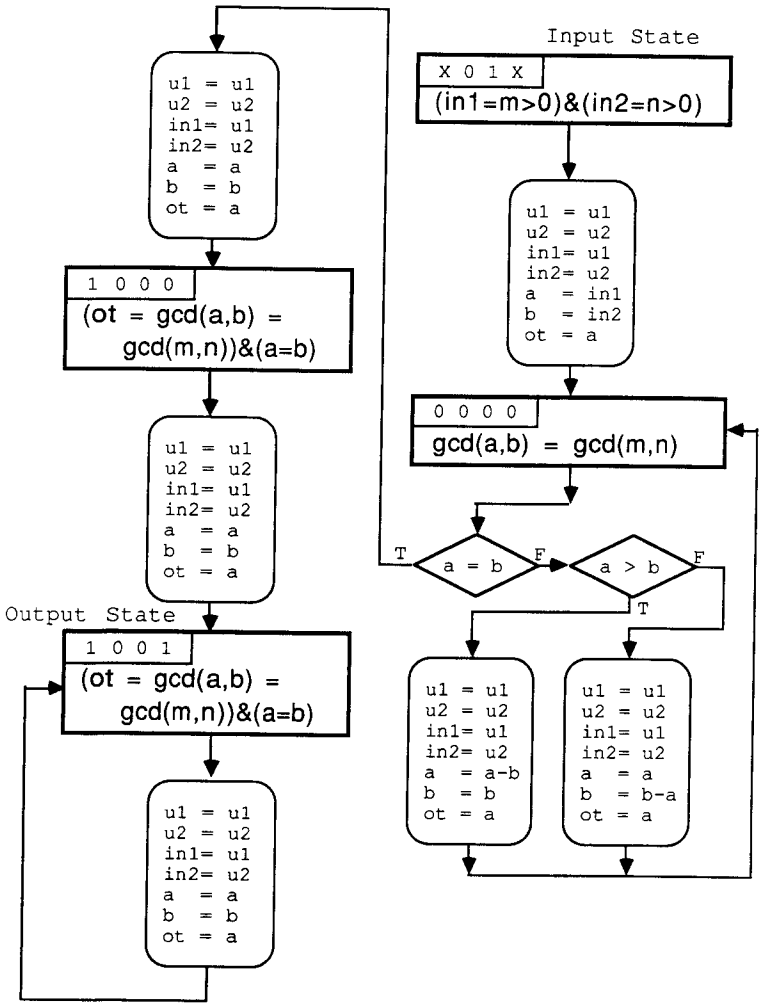


Figure 13. The GCD module's ASM chart, with the correctness propositions for each state box. Each proposition can be derived from the preceding proposition(s); the input proposition is assumed *true*.

8. Summary

In this paper we discussed the various components of an integrated register-transfer-level digital design and verification environment under development at the University of Waterloo.

We proposed a simple but powerful model of RT design, based on a small set of primitives and a typed interconnection environment, as the central structure around which these components are designed and implemented. The primitives are close to the designer's normal design experience, and are amenable to a mathematical treatment.

We also proposed a functional model for specifying the primitives and the behaviour of their compositions. Our functional model is influenced by Gordon's work [8]; nonetheless, our approach is simpler, and differs in the following ways:

1- Our model closely follows the structural aspects of RT design by separating the design into a data-path and a control part, and by defining the data-path in terms of a number of data-slices. This brings the formalism much closer to the reality of design.

2- We distinguish between device *port* names as bound variables and *net* names as free variables. This makes the interconnection definition task considerably simpler. By distinguishing between port and net names we do not need to use the *restriction* operator to hide the internal connections.

3- We have introduced three signal types, and have shown that each primitive element interacts through a subset of these types. The signal types correspond to the signal flows within the design, and thus capture the RT layout strategies.

We then used the functional model to derive the ASM specifications of candidate designs.

We introduced a friendly interface to the model, in the form of a synchronous hardware description language. The main features of the language are its support of hierarchy, handling of design modules as abstract objects, and the explicit separation of module definitions into data-path and control-unit components.

In order to reason about the candidate design we had to make assertions about the behaviour of the design at its interfaces; thus we extended the design to include its input and output strings. The ASM charts corresponding to the extended objects were proved correct by showing the existence of suitable assertions about the states that the hardware has to step through, including the input and the output states.

It is our contention that, given the wealth of experience and know-how developed over many years of applying similar methods to proving the correctness of software, the method of inductive assertions may be better suited for use by researchers and the design community than those methods which require newly developed skills, and possibly less well-known mathematical techniques.

The model has been used to specify a number of small and medium-size designs. To date by far the largest specification attempted so far has been the specification of a speech processing chip [36]. An experimental silicon compilation system is available for converting the high-level specifications to data-path slices; the high-level specification language is an earlier version of the SDCL. Plans are under way to integrate the complete SDCL, to develop a symbolic computation package to help with the reasoning process, and to introduce a second generation of the silicon compiler.

9. Acknowledgement

The author would like to acknowledge the University of Waterloo funding and computing facilities used to carry out the work reported here. The author also gratefully acknowledges the useful comments and suggestion of two anonymous referees, whose detailed reading of the manuscript helped to improve the readability and completeness of the paper.

References

1. C. Mead and L. Conway, *Introduction to VLSI Systems,* Addison Wesley, Reading, Mass. (1980).

2. R. W. Hartenstein, "Basics of Structured Design Methodologies: Data-Path and Finite State Machines", pp. 73-107 in *Design Methodologies for VLSI Circuits*, ed. P. G. Jespers, C. H. Sequin, and F. Van De Wiele, Sijthoff & Noordhoff, Rockville, Maryland (1982).

3. A. C. Parker, F. Kurdahi, and M. Milnar, "A General Methodology for Synthesis and Verification of Register-Transfer Designs", *Proceedings of 21st. Design Automation Conference*, pp. 329-335 (June 1984).

4. J. M. Siskind, J. R. Southard, and K. W. Crouch, "Generating Custom High Performance VLSI Designs from Succinct Algorithmic Descriptions", *Proceedings of 1982 Conference on Advanced Research in VLSI*, pp. 28-39 (1982).

5. D. E. Thomas, C. Y. Hitchcock III, T. J. Kowalski, and R. A. Walker, "Topics in Automatic Data-Path Synthesis", Research report CMUCAD-83-9, Carnegie Mellon University, Pittsburgh, Pen. (April 1983).

6. C. Tsen and D. P. Siewiorek, "The Modeling and Synthesis of Bus Systems", DRC-18-24-82, Digital Research Center, Carnegie Mellon University, Pittsburgh, PA 15213 (April 1982).

7. M. R. Barbacci and D. P. Siewiorek, *The Design and Analysis of Instruction Set Processor,* McGraw-Hill Book Company, New York (1982).

8. M. Gordon, "A Model of Register Transfer Systems with Application to Microcode and VLSI Correctness", CSR-82-81, University of Edinburgh, Dept. of Computer Science, Edinburgh, Scotland (March 1981- revised May 1982).

9. D. Johannsen, "Bristle Blocks: A Silicon Compiler", *Proceedings of the 16th. Design Automation Conference*, pp. 310-313 (July 1979).

10. J. R. Southard, "MacPitts: An Approach to Silicon Compilation", *IEEE Computer Magazine* **Vol. 16.** No. 12. pp. 74-82 (December 1983).

11. F. Anceau, "CAPRI: A Design Methodology and a Silicon Compiler for VLSI Specified by Algorithms", pp. 15-31 in *Third Caltech Conference on Very Large Scale Integration*, ed. Bryant, R., Computer Science Press, Rockville, Maryland (1983).

12. F. Mavaddat, "A Model for Register-Transfer Level Design Specification: The SDC Notation", CS-84-34, Department of Computer Science,, University of Waterloo, Waterloo, Ontario (October 1984).

13. F. Mavaddat, "An Architecture and Layout for Register Transfer Level IC Design", Report 85-4, Institute for Computer Research, University of Waterloo, Waterloo, Ontario (January 1985).

14. F. Mavaddat, "A Functional Model of Register-Transfer Designs", Research report CS-88-16, Department of Computer Science, University of Waterloo, Waterloo, Ontario (October 1988).

15. R. Milner, *A Calculus of Communicating Systems, Lecture Notes in Computer Science, No. 92,* Springer Verlag, New York (1980).

16. J. P. Hayes, "Pseudo Boolean Logic Circuits", CRL-TR-33-84, Computing Res. Lab., The University of Michigan,, Ann Arbor, Michigan (August 1984).

17. R. E. Bryant, "A Switch-Level Model and Simulator for MOS Digital Systems", *IEEE Transactions on Computers* **Vol. C-33.** No. 2. pp. 160-177 (Feb. 1984).

18. J. A. Brzozowski and M. Yoeli, "Combinational Static CMOS Networks", *INTEGRATION, the VLSI Journal,* (Vol. 5.) pp. 103-122 (1987).

19. R. W. Hartenstein, K. Lammert, and A. Wodtko, *Karl III Manual,* Kaiserslautern University, Kaiserslautern, Germany (1986).

20. L. J. Hafer and A. C. Parker, "A formal Method for Specification, Analysis, and Design of Register-Transfer Level Digital Logic", *IEEE Transactions on Computer-Aided Design* **Vol. CAD 2.** No. 1. pp. 4-18 (January 1983).

21. H. Eveking, "The Application of CHDL's to the Abstract Specification of Hardware", pp. 167-178 in *Computer Hardware Description Languages and Their Applications*, ed. C. J. Koomen and T. Moto-oka, Elsevier Science Publishers, North Holland, Amsterdam, The Netherlands (1985).

22. H. Eveking, "Formal Verification of Synchronous Systems", pp. 137-149 in *Formal Aspects of VLSI Design*, ed. G. J. Milne and P. A. Subrahmanyam, Elsevier Science Publishers, North Holland, Amsterdam, The Netherlands (1986).

23. R. Piloty, M. Barbacci, D. Borrione, D. Dietmeyer, F. Hill, and P. Skelly, *CONLAN Report, Lecture Notes in Computer Science, No. 51,* Springer Verlag, New York (1983).

24. F. K. Hanna and N. Daeche, "Specification and Verification using Higher Order Logic: A Case Study", Electronics Laboratory, University of Kent, Canterbury, England (November 1985).

25. P. J. Landin, "The Mechanical Evaluation of Expressions", *Computer Journal* **Vol. 6.** No. 4. pp. 308-320 (Jan. 1964).

26. F. Mavaddat, "Physical Design Capture at the Structural Level", unpublished report, Dept. of Computer Science, University of Waterloo, Waterloo, Ontario (1988).

27. H. Aboel-Foto and F. Mavaddat, "A Language for VLSI Physical Design at Register-Transfer Level", *Technical Digest, 1986 Canadian Conference on VLSI,* (October 1986).

28. C. E. Leiserson and J. B. Saxe, "Optimizing Synchronous Systems", *Proc. 22nd. Annual Symposium on Foundations of Computer Science*, pp. 23-36 (1981).

29. L. Cardelli and P. Wegner, On Understanding Types, Data Abstraction, and Polymorphism, *Computing Surveys* **Vol. 17.** No. 4. pp. 471-522 (December 1985).

30. C. S. Yeung and F. Mavaddat, "Procedural Cell Design, Placement, and Routing, for VLSI Design Automation", *Invited contribution, VLSI Systems Design* **Vol. VI.** No. 9. pp. 122-125 (September 1985).

31. C. S. Yeung and F. Mavaddat, "A Procedural Cell Layout Specification Environment for VLSI Design", *Proceedings of the IEEE Electronicum 85*, pp. 18-23 (October 1985).

32. C. Clare, *Designing Logic Systems using State Machines,* McGraw Hill, Maidenhead (1972).

33. D. Green, *Modern Logic Design,* Addison-Wesley Publishing Company, Workingham, England (1986).

34. R. W. Floyd, "Assigning Meaning to Programs", pp. 19-32 in *Proc. Symposium Applied Math*, American Mathematical Society, Providence, R. I. (1967).

35. C. A. R. Hoare, "An Axiomatic Basis for Computer Programming", *Communications of the ACM* **Vol. 12.** No. 10. pp. 576-583 (October 1969).

36. F. Mavaddat, "Architecture and Layout of a Speech Recognition VLSI Chip", *International Journal of Mini and Microcomputers* **Vol. 9.** No. 1. pp. 1-5 (1987).