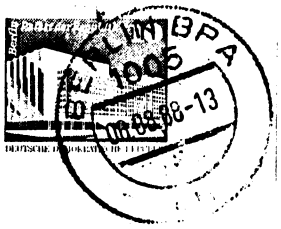
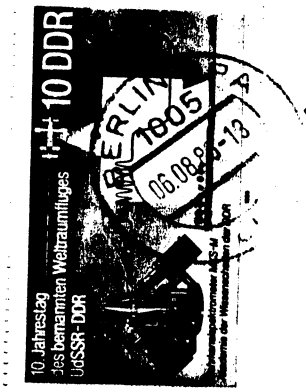
Dr. rer. nat.
R. Karl-Adolf Zech
Diplom-Mathematiker
Schliemannstrasse 28
Berlin
DDR – 1058
German Democratic Republic

DRUCKSACHE





Dr. F. Mavaddat
Dept. Computer Science
University of Waterloo
Ontario N2L 3G1
WATERLOO
CANADA

Dr. rer. nat.
R. Karl-Adolf Zech
Diplom-Mathematiker
Schliemannstrasse 28
Berlin
DDR – 1058 German Democratic Republic
Consultant Software Engineer

Dear Dr. Mavaddat,
I should be very obliged to you for sending me a copy of your paper entitled:

A functional model of RT design
CS-88-16
please, if possible.

Thanking you in advance, yours sincerely,

R. Zech

Sent
JAN 11 1989

Printing Requisition / Graphic Services

15101

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

A Functinnal Model of Register-Transfer Designs CS-88-16

DATE REQUISITIONED	DATE REQUIRED	ACCOUNT NO.
Oct. 28	ASAP	4 1 2 6 8 6 6 4 1

REQUISITIONER – PRINT	PHONE	SIGNING AUTHORITY
F. Mavaddat	4430	F. Mavaddat

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: 25 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: [X] COLLATING [X] STAPLING [ ] ____ HOLE PUNCHED [ ] PLASTIC RING

FOLDING/PADDING: CUTTING SIZE:

Special Instructions

3 down left side

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

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 [ ]

A Functional Model of Register-Transfer Designs

Farhad Mavaddat

Dept. of Computer Science
University of Waterloo
Research Report CS-88-16
October, 1988

A Functional Model of Register-Transfer Designs

Farhad Mavaddat

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

ABSTRACT

We propose a strongly-typed functional model of register-transfer-level design specifications. The model is influenced by Gordon's register-transfer model of digital design and, compared to it, is presented from a more intuitive point of view, which in a way is closer to the reality of the RT design. We use the typed nature of the design environment to develop a semantic model for our SDC design notation, reported earlier,[1] and to enforce correct composition of SDC-based designs.

The model can be used for design specification purposes as well as for analysing and reasoning about designs.

A Functional Model of Register-Transfer Designs

Farhad Mavaddat

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

1. Motivation

Register-Transfer (RT) based designs enjoy a high degree of structural regularity, which contributes to their acceptance as suitable models of VLSI design. This regularity is manifested by the explicit separation of a design into a PLA-type control-part and a slice-based data-path-part, and leads to an efficient placement and routing scheme. Layout efficiency and regularity has been the main motivation behind a number of silicon compilation activities in the past,[2-4] and will likely continue to contribute to future developments.

Theoretical interest in RT-level modeling has also gained momentum over the last few years[5, 6] and, among others, Gordon's functional model[5] has received special attention. Unfortunately, that model (among others) does not capture the above-mentioned regularities in an explicit form and so, fails to act as a direct representation of the corresponding RT design.

It is our belief that for a design abstraction to gain acceptance, there should be a one-to-one link between the objects of the design and the elements of the model, similar to the relationship between the logic-gate-based designs and their corresponding Boolean algebra model.

It is the purpose of this report to adapt a modified and extended version of Gordon's model to the direct capture of digital designs. To achieve this, we apply the model to the SDC design primitives[1, 7] and show that SDC-based designs can be modeled in a direct, one-to-one form, using the proposed functional model.

2. Defining Combinational Modules

Let **T** be the set of basic signal types used in communication with a device. We define an *m*-input, *n*-output (*m*×*n-put*) combinatorial device *D* , shown in Figure 1, to be of type:

$$D : (t_1 \times t_2 \times \cdots \times t_m) \rightarrow (t_{m+1} \times t_{m+2} \times \cdots \times t_{m+n}) \qquad (1)$$

if $t_1, t_2, \cdots, t_{m+n} \in \mathbf{T}$ represent the types of values appearing at the m input and the n output ports of device D respectively.

We define the behavior of D by:

$$D = \lambda(\eta_1, \eta_2, \cdots, \eta_m).(E_1, E_2, \cdots, E_n); \qquad (2)$$

where the right side of (2) is a short form for:

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

$\eta_j : t_j \in \mathbf{T}$, $1 \leq j \leq m$ is the *jth* input port's value, and $g_k : (t_1 \times t_2 \times \cdots \times t_m) \rightarrow t_{m+k}$, defined by $g_k = \lambda(\eta_1, \eta_2, \cdots, \eta_m).\, E_k$, $1 \leq k \leq n$, defines the *kth* output-port's value.

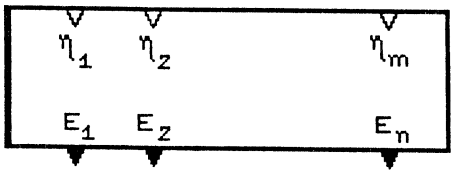


Figure 1

3. Defining Sequential Circuits

At every state, the behavior of a Mealy-type sequential machine *B*, shown in Figure 2, has two components. First, its combinational behavior, B_{cmb} , under the influence of the current state and port inputs, and second, its next state behavior, B_{seq} , under the influence of the state and the port inputs at the time of transition to the next state. Therefore, the behavior of an $m \times n$ *–put* , q *–state* sequential machine B , at state $(s_1, s_2, \cdots, s_q)$, $s_j : t_j \in \mathbf{T}$, $1 \leq j \leq q$, is modeled by two combinational circuits of types:

$$B_{cmb} : (t_1 \times t_2 \times \cdots \times t_q \times t_{q+1} \times t_{q+2} \times \cdots \times t_{q+m}) \rightarrow (t_{q+m+1} \times t_{q+m+2} \times \cdots \times t_{q+m+n}) \quad (3)$$

and,

$$B_{seq} : (t_1 \times t_2 \times \cdots \times t_q \times t_{q+1} \times t_{q+2} \times \cdots \times t_{q+m}) \rightarrow (t_1 \times t_2 \times \cdots \times t_q) \quad (4)$$

and is defined by:

$$\begin{Bmatrix} B_{cmb} \\ B_{seq} \end{Bmatrix} = \lambda(\eta_1, \eta_2, \cdots, \eta_q, \eta_{q+1}, \cdots, \eta_{q+m}) \cdot \begin{Bmatrix} (E_1, E_2, \cdots, E_n) \\ (F_1, F_2, \cdots, F_q) \end{Bmatrix} \quad (5)$$

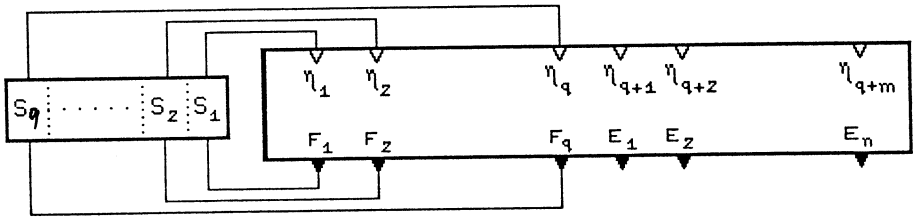


Figure 2

A few observations are appropriate at this point.

- $E_1, E_2, \cdots, E_n$ are the n output port 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 the transition to the next state.
- The $q+m$ inputs represent the m input-port (environment) and q input-state values. To distinguish between the state and the port inputs we (sometimes) move the input-state bound variables to the left of the equality symbol, 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 a next state $(F_1, F_2, \cdots, F_q)$ for B, where $F_j : t_j \in \mathbf{T}$, $1 \leq j \leq q$ is the new value for the *jth* state variable.

Combining the two components of (5) into a single definition, and following the new practice of separating the bound variables, we write

$$B\ (s_1, s_2, \cdots, s_q\ ) = \lambda(\eta_1,\eta_2, \cdots,\eta_m\ )\ .\ ((E_1,E_2, \cdots,E_n\ ), B\ (F_1,F_2, \cdots,F_q\ )) \tag{6}$$

to represent the behavior of B and re-write (3) and (4) as

$$B_{cmb}\ (s_1, s_2, \cdots, s_q\ ) = \lambda(\eta_1,\eta_2, \cdots,\eta_m\ )\ .\ (E_1,E_2, \cdots,E_n\ ) \tag{7}$$

and

$$B_{seq}\ (s_1, s_2, \cdots, s_q\ ) = \lambda(\eta_1,\eta_2, \cdots,\eta_m\ )\ .\ (F_1,F_2, \cdots,F_q\ ) \tag{8}$$

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

4. Composite Modules

So far we have concentrated on the definition of primitive modules, or the *leaf cells*.[8] It is the purpose of this section to propose a formalism for the definition of composite modules in terms of the instances of primitive and/or other (possibly predefined) composite modules. In the context of a composite module, we refer to the lower level instances as its sub-modules.

An $m \times n$ *–put* composite module $\mathbf{f}^{\,c}$ is defined as the interconnection of s submodules $\mathbf{f}^{\,0}, \mathbf{f}^{\,1}, \cdots, \mathbf{f}^{\,s-1}$, and a (hypothetical) $n \times m$ *–put* environment module $\mathbf{f}^{\,s}$, where the input and output ports of $\mathbf{f}^{\,s}$ respectively define the output and the input ports of $\mathbf{f}^{\,c}$, as shown in Figure 3. Furthermore, we define:

- $\mathbf{I} = \bigcup_{i=0}^{s} I^i$, $\mathbf{O} = \bigcup_{i=0}^{s} O^i$, as the set of internal input and the output ports respectively, where I^i, $0 \le i \le s$, and O^i, $0 \le i \le s$, are the respective sets of input and output ports of the *ith* module, and
- $P = \{p_1, p_2, \cdots, p_t\}$ 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, $h : \mathbf{O} \rightarrow P$ is *one-to-one*, and $h : \mathbf{I} \rightarrow P$ is *onto*.

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

$$(y_1, y_2, \cdots, y_{n^i}) = \mathbf{f}^{\,i}_{cmb}(s_1, s_2, \cdots, s_{q^i})\ (x_1, x_2, \cdots, x_{m^i}) \tag{9}$$

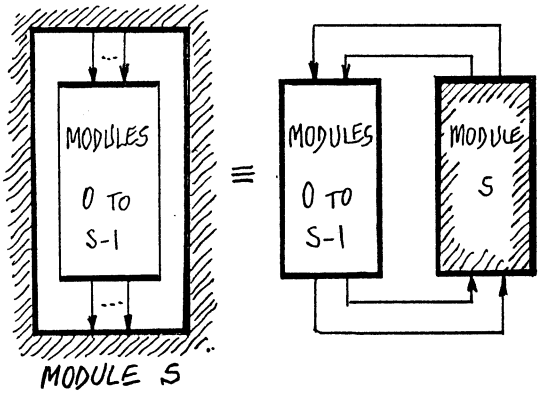


Figure 3

as a short form for

$$y_j = (\lambda(\eta_1, \eta_2, \cdots, \eta_{q^i}, \eta_{q^i+1}, \eta_{q^i+2}, \cdots, \eta_{q^i+m^i}) \cdot E_j) \qquad (10)$$
$$(s_1, s_2, \cdots, s_{q^i}, x_1, x_2, \cdots, x_{m^i}) \quad 1 \le j \le n^i ,$$

where

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

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

$$\mathbf{f}^{\,c}\ (S^1, S^2, \cdots, S^s) = \lambda(h(O^s)) \cdot (\ \mathbf{rec}$$
$$(\ Y^i = \mathbf{f}^{\,i}_{cmb}(\ S^i)(\ X^i) \quad 1 \le i \le s-1) \qquad (11)$$
$$\mathbf{in}\ (\ h(\ I^s), \mathbf{f}^{\,c}\ (\ \mathbf{f}^{\,i}_{seq}(\ S^i)(\ X^i) \quad 1 \le i \le s-1))$$

where

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

and

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

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 $\mathbf{f}^{\,i}$, $1 \le i \le s$, and **rec** and **in** are defined as in.[9]

5. The SDC Model of Register-Transfer Design

Let $\mathbf{T} = \{ \mathbf{S}, \mathbf{C}, \mathbf{D} \}$ be the set of signal types used in the design of a *register-transfer-based* design, where:

- **S** is the type of signal indicating the truth values of the assertions made about the status of the *data-path*.
- **C** is the type of signal selecting among the path alternatives inside the *data-path*.
- **D** is the type of signal carrying data values among the path *slice* components. Such data values depend on the width of *slice* being defined. For example, for a binary *slices*, we have $\mathbf{D}_b = \{ 0, 1 \}$, and in case of a decimal slice, we have $\mathbf{D}_d = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

In the remainder of this report we will use small letter identifiers as variables ranging over the above sets of values and '$\langle\,\rangle$', '[ ]', and ' { } ' to enclose **S**, **C**, and **D** type variables, respectively.

We now introduce four design primitives which constitute the building blocks of our **SDC** model.

5.1. The Selector-Slice Primitive

A *selector –slice* **sel**, Figure 4, is a combinational device of type $\mathbf{sel} : \mathbf{D} \times \mathbf{D} \times \mathbf{C} \rightarrow \mathbf{D} \times \mathbf{C}$, defined by:

$$\mathbf{sel} = \lambda\{d_1, d_2\}[c] \,.\, \{c \rightarrow d_2, d_1\}[c], \tag{12}$$

where '$\rightarrow$', in the context of an expression, stands for the *if_then_else* operator. Definition (12) indicates that the output of a *selector* is equal to one of its two data inputs and the selection is made according to the value of input $c : \mathbf{C}$.

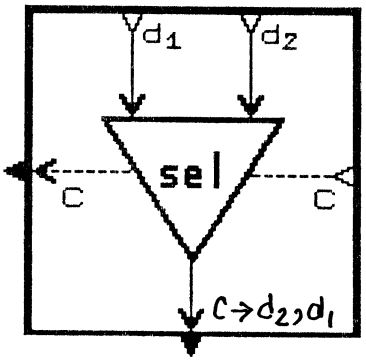


Figure 4

5.2. Functional Primitives

Functional –slices are a family of $(m + k)\times(n + k)$–*put* combinational devices of type $(\mathbf{D}^m \times \mathbf{S}^k) \rightarrow (\mathbf{D}^n \times \mathbf{S}^k)$, as shown in Figure 5, where: $m \geq 1$, $1 \geq k \geq 0$, $1 \geq n \geq 0$, and $n + k \geq 1$.

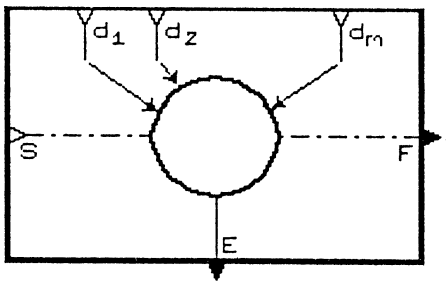


Figure 5

Thus, the behavior of a functional primitive can be defined by one of the following three definition schemes:

$$\lambda\{d_1, d_2, \cdots, d_m\}\langle s\rangle . \{E\}\langle F\rangle \tag{13}$$

$$\lambda\{d_1, d_2, \cdots, d_m\}\langle s\rangle . \langle F\rangle \tag{14}$$

$$\lambda\{d_1, d_2, \cdots, d_m\} . \{E\}. \tag{15}$$

Depending on the nature of application, the number and type of operators used in E and F may vary. For example, *multiplication* might not be allowed when the model is used as the input to a silicon compiler, while its use in simulation applications might be allowed. In a similar way, *addition* might prove to be un-acceptable when the model is used for reasoning about hardware, but acceptable when the model is intended for silicon implementation. We now present a few typical *functional-slices* specified according to the techniques discussed in (13)-(15).

Ex.1- Binary 'and' Slice:

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

$$\textbf{and} = \lambda\{a, b\} . \{a \wedge b\}. \tag{16}$$

Ex.2- Binary 'equal' Slice:

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

$$\textbf{equal} = \lambda\{ a , b \}\langle s \rangle . \langle s \wedge \overline{(a \oplus b)} \rangle, \tag{17}$$

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 neighbouring slice.

Ex.3- 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:

$$\textbf{add} = \lambda\{ a , b \}\{s \rangle . \{(a + b + num (s)) \; mod \; 10\}\{num (s) + a + b > 9 \rangle \tag{18}$$

Where a and b are the slice's data inputs, s is the carry input from the less significant neighbouring slice, $(a + b + num (s)) \; mod \; 10 \in \mathbf{D}$ is the data output, $(num (s) + a + b > 9) \in \mathbf{S}$ is the carry to the more significant neighbouring slice, and $num : \mathbf{S} \rightarrow \mathbf{D}$ is a function that produces the numerical equivalent of the status input signal.

5.3. The Controller Primitives

Controllers are a family of $m \times n$ *–put* devices of type $\mathbf{C}^p \times \mathbf{S}^q \rightarrow \mathbf{C}^s \times \mathbf{S}^t$, where $m = p + q$ and $n = s + t$, $p \geq 0$, $q \geq 0$, $s \geq 1$, and $t \geq 0$; a typical way of defining this might be as:

$$\lambda[\eta_1, \eta_2, \cdots, \eta_p]\langle \eta_{p+1}, \eta_{p+2}, \cdots, \eta_{p+q} \rangle . [B_1, B_2, \cdots, B_s] \; \langle S_1, S_2, \cdots, S_t \rangle, \tag{19}$$

where B_i, $1 \leq i \leq s$, and S_i, $1 \leq i \leq t$, are the sum of the products of the bound variables and their complements.

5.4. The Unit Delay Primitive

A *unit-delay* **del** is a 1×1*–put*, single state, polymorphic [10] sequential device of type **del** : $(* \times *) \rightarrow (* \times *)$, shown in Figure 6 and defined by:

$$\textbf{del} (n) = \lambda(i) . (n , \textbf{del} (i)) \tag{20}$$

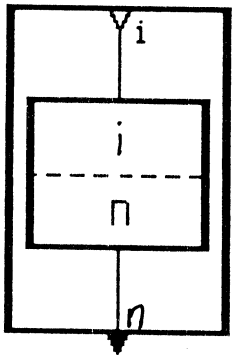


Figure 6

6. Typed Connections

We extend the concept of typed ports to that of typed *nets*. This assumes that the *nets* of a particular type connect the ports of the same type. We continue to use the parenthesis pairs [ ], and { } and ⟨ ⟩, to enclose *control*, *data*, and *status nets*, respectively.

In this spirit we partially re-write (9) as:

$$\{y_1, y_2, \cdots, y_{n_d}\} = \mathbf{f}_{cmb}(s_1, s_2, \cdots, s_q)(x_1, x_2, \cdots, x_m) \qquad (21)$$

to emphasize the output *data* conections of **f**, where n_d is the number of *data* outputs of **f**.

In a similar way, one may specify the *control* and *status* type connections, or extend the definition into a single statement of the form:

$$\{Y_D\}[Y_C]\langle Y_S\rangle = \mathbf{f}_{cmb}(S)\{X_D\}[X_C]\langle X_S\rangle \qquad (22)$$

so as to re-write (9) in a completely-typed form.

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

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

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

Extending this convention to the sequential behavior, we write:

$$\mathbf{f}_{seq}(S)\{X_D\}[X_C]\langle X_S\rangle, \qquad (23)$$

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

Finally, we call a composition a *data-* (or *control-*, or *status-*) *composition*, if only *data* (or *control,* or *status*) type *nets* are used in making the connections.

7. Register-Transfer Slices

A Register-Transfer (RT) *slice* is defined as the *data-composition* of primitives and/or other *data-composed* sub-modules. This guarantees that the *control* and *status* ports of an RT-slice remain unconnected. We now present a number of RT slice examples.

- The *m-bit* "register-counter" slice:

  Following are the behavioral definitions of three submodules: **sel**, **del**, and **inc** used in the composition of the *m-bit* "register-counter" slice shown in Figure 7.

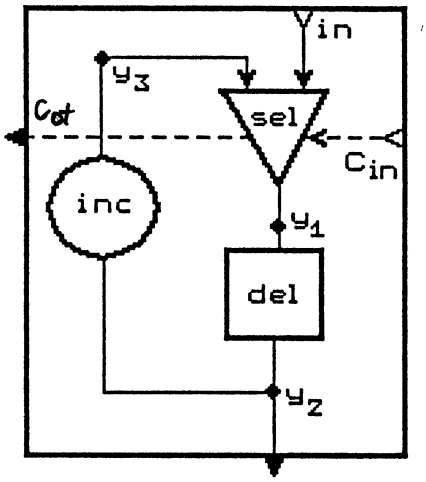


Figure 7

$$\textbf{sel} = \lambda\{ in_1, in_2 \}[c] . \{ c \rightarrow in_2, in_1 \} [c]$$

$$\textbf{del} (n) = \lambda\{ in \} . (\{ n \}, \textbf{del} (in))$$

$$\textbf{inc} = \lambda\{ in \} . \{(in +1) \textbf{ mod } 2^m \}.$$

Next we write the composition rule for the *m-bit* "register-counter" using the above sub-modules, the nets y_1, y_2, y_3, in , c_{in} , and c_{ot} . According to the (11),

$$\textbf{count } (n) = \lambda\{ in \}[c_{in}] . (\textbf{rec} (\{ y_1\}[c_{ot}] = \textbf{sel}_{cmb} \{ y_3, in \}[c_{in}];$$

$$\{ y_2\} = \textbf{del}_{cmb} (n)\{y_1\};$$

$$\{ y_3\} = \textbf{inc}_{cmb} \{y_2\}) \textbf{ in } (\{y_2\}[c_{ot}], \textbf{count } (\textbf{del}_{seq} (n)(y_1)))).$$

This can be expanded to

count (n) = λ\{in \}[c_{in}] . (**rec** (

$y_1 = c_{in} \rightarrow in , y_3$;

$c_{ot} = c_{in}$;

$y_2 = n$;

$y_3 = (y_2+1)$**mod** 2^m) **in** ({y_2}[c_{ot}] , **count** (y_1))),

and reduced to

count (n) = λ\{ in \}[c_{in}] . ({ n }[c_{in}] , **count** ($c_{in} \rightarrow$ in , (n +1) **mod** 2^m).

Verbally, **count** is a single state (n) sequential device with one data input (in), one data output, one *control* input (c_{in}) and one *control* output. The **counter**'s next state value is controlled by the value of the control input which selects between (n +1)**mod** 2^m and in as the next state value.

- The "shift-register" slice:

The following is the behavioral definition of one slice of a shift register shown in Figure 8. The sub-modules **sel**, and **del** used in the composition of the "shift-register" slice are defined as usual.

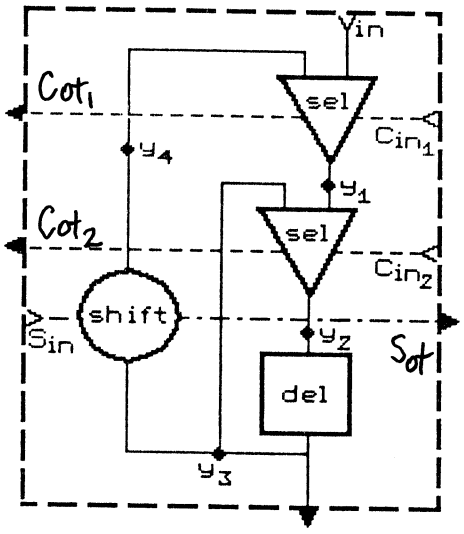


Figure 8

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

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

Next, we write the composition rule for forming the "shift register" slice using the above sub-modules 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 the (11),

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

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

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

- Hierarchical Design Example:

  In the following example we first give the behavioral definition of one slice of a register with add capability, called **radd**, shown in Figure 9.

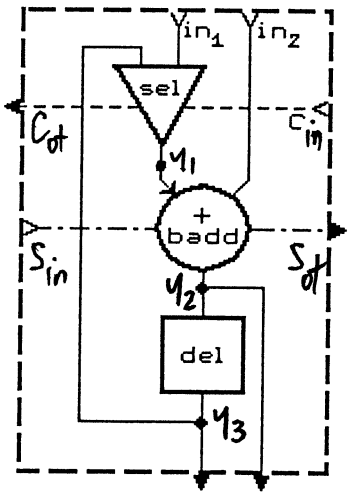


Figure 9

We then show the use of this unit in a hierarchical design using two such modules. The sub-modules used in composition of the **radd** are **sel**, **del**, and a binary adder called **badd** defined by

$$\textbf{badd} = \lambda\{in_1, in_2\}\langle s_{in}\rangle \,.\, \{in_1 \oplus in_2 \oplus s_{in}\}\,\langle in_1 \wedge in_2 + in_1 \wedge s_{in} + in_2 \wedge s_{in}\rangle.$$

Next, we write the composition rule for forming the **radd** slice, using the above sub-modules, the nets y_1, y_2, y_3, in_1, in_2, c_{in}, s_{in}, c_{ot}, and s_{ot}. According to the (11),

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

This is expanded to

$$\begin{aligned}\textbf{radd}\ (n) = \lambda\{\,in_1, in_2\}[c_{in}\,]\langle s_{in}\rangle\ .\ (\textbf{rec}\ (&\\ y_1 = c_{in} \longrightarrow in_1, y_3;&\end{aligned}$$

$$
\begin{aligned}
& c_{ot} = c_{in} ; \\
& y_2 = y_1 \oplus in_2 \oplus s_{in} ; \\
& s_{ot} = y_1 \wedge in_2 + y_1 \wedge s_{in} + in_2 \wedge s_{in} ; \\
& y_3 = n \ ) \\
& \textbf{in } (\{y_3, y_2\}[c_{ot}]\langle s_{ot} \rangle \ , \ \textbf{radd } (y_2))),
\end{aligned}
$$

and reduced to

$$
\begin{aligned}
\textbf{radd } (n) = \lambda\{ in_1, in_2\}[c_{in}]\langle s_{in} \rangle \ . \\
& (\{ n \ , \ (c_{in} \rightarrow in_1, n) \oplus in_2 \oplus s_{in} \}[c_{in}] \\
& \quad \langle ((c_{in} \rightarrow in_1, n) \wedge in_2 + (c_{in} \rightarrow in_1, n) \wedge s_{in} + in_2 \wedge s_{in} \rangle \ , \\
& \quad \textbf{radd } (((c_{in} \rightarrow in_1, n) \oplus in_2 \oplus s_{in})).
\end{aligned}
$$

A further reduction yields

$$
\begin{aligned}
\textbf{radd } (n) = \lambda\{in_1, in_2\}[c_{in}]\langle s_{in} \rangle \ . \\
& (\{ n \ , \ (c_{in} \rightarrow in_1, n) \oplus in_2 \oplus s_{in} \}[c_{in}] \\
& \langle ((in_2 + s_{in}) \wedge (c_{in} \rightarrow in_1, n) + in_2 \wedge s_{in} \rangle, \\
& \textbf{radd } (((c_{in} \rightarrow in_1, n) \oplus in_2 \oplus s_{in})).
\end{aligned}
$$

We are now interested in deriving the behavior of the serial connection of two **radds**, as shown in Figure 10.

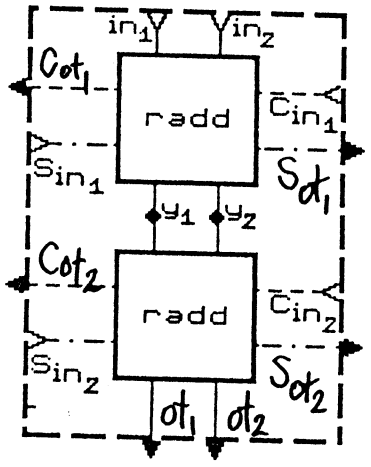


Figure 10

The following derivations lead to the definition of combined behavior. First,

$$\textbf{Dradd}\ (n\ ,m\ ) = \lambda\{in_1,\ in_2\}[c_{in_1},\ c_{in_2}]\langle s_{in_1},\ s_{in_2}\rangle\ .\ (\textbf{rec}\ (\{y_1,\ y_2\}[c_{ot_1}]\langle s_{ot_1}\rangle = \textbf{radd}_{cmb}\ (n\ )\{in_1,\ in_2\}[c_{in_1}]\langle s_{in_1}\rangle$$
$$\{ot_1,\ ot_2\}[c_{ot_2}]\langle s_{ot_2}\rangle = \textbf{radd}_{cmb}\ (m\ )\{y_1,\ y_2\}[c_{in_2}]\langle s_{in_2}\rangle)$$
$$\textbf{in}\ (\{ot_1,\ ot_2\}[c_{ot_1},\ c_{ot_2}]\langle s_{ot_1},\ s_{ot_2}\rangle,$$
$$\textbf{Dradd}\ (\textbf{radd}_{seq}\ (n\ )\{in_1,\ in_2\}[c_{in_1}]\langle s_{in_1}\rangle,$$
$$\textbf{radd}_{seq}\ (m\ )\{y_1,\ y_2\}[c_{in_2}]\langle s_{in_2}\rangle))).$$

This is expanded to

$$\textbf{Dradd}\ (n\ ,m\ ) = \lambda\{in_1,\ in_2\}[c_{in_1},\ c_{in_2}]\langle s_{in_1},\ s_{in_2}\rangle\ .\ (\textbf{rec}\ (y_1 = n\ ;$$
$$y_2 = (c_{in_1} \rightarrow in_1,\ n\ ) \oplus in_2 \oplus s_{in_1;}$$
$$c_{ot_1} = c_{in_1;}$$
$$s_{ot_1} = (in_2,\ s_{in_1}) \wedge (c_{in_1} \rightarrow in_1,\ n\ ) + in_2 \wedge s_{in_1;}$$
$$ot_1 = m\ ;$$
$$ot_2 = (c_{in_2} \rightarrow y_1,\ m\ ) \oplus y_2 \oplus s_{in_2;}$$
$$c_{ot_2} = c_{in_2;}$$
$$s_{ot_2} = (y_2 + s_{in_2}) \wedge (c_{in_2} \rightarrow y_1,\ m\ ) + y_2 \wedge s_{in_2})$$
$$\textbf{in}\ (\{ot_1,\ ot_2\}[c_{ot_1},\ c_{ot_2}]\langle s_{ot_1},\ s_{ot_2}\rangle,\ \textbf{Dradd}\ ($$
$$(c_{in_1} \rightarrow in_1,\ n\ ) \oplus in_2 \oplus s_{in_1},$$
$$(\ c_{in_2} \rightarrow y_1,\ m\ ) \oplus y_2 \oplus s_{in_2}),$$

and reduced to

$$\begin{aligned}
\textbf{Dradd}\ (\ n\ ,m\ ) = \lambda\{in_1,\ in_2\}[c_{in_1},\ c_{in_2}]\langle s_{in_1},\ s_{in_2}\rangle\ .\ (&\\
&\{m\ ,\ (c_{in_2}\rightarrow n\ ,\ m\ )\oplus((c_{in_1}\rightarrow in_1,\ n\ )\oplus in_2\oplus s_{in_1})\oplus s_{in_2}\}\\
&[c_{in_1},\ c_{in_2}]\\
&\langle(in_2,\ s_{in_1})\wedge(c_{in_1}\rightarrow in_1,\ n\ )+in_2\wedge s_{in_1},\\
&(((c_{in_1}\rightarrow in_1,\ n\ )\oplus in_2\oplus s_{in_1}\ )+s_{in_2})\wedge(c_{in_2}\rightarrow n\ ,\ m\ )+\\
&\quad((c_{in_1}\rightarrow in_1,\ n\ )\oplus in_2\oplus s_{in_1}\ )\wedge s_{in_2}\rangle,\\
&\textbf{Dradd}\ ((c_{in_1}\rightarrow in_1,\ n\ )\oplus in_2\oplus s_{in_1},\\
&\quad(c_{in_2}\rightarrow n\ ,\ m\ )\oplus((c_{in_1}\rightarrow in_1,\ n\ )\\
&\quad\oplus in_2\oplus s_{in_1}\ )\oplus s_{in_2}))).
\end{aligned}$$

8. Multi-Slice Data-Path Definition

Given a *data –slice* f and a positive integer n, an n-wide data-path is formed by concatenating n such slices along their *control* and *status* ports, as shown in Figure 11.

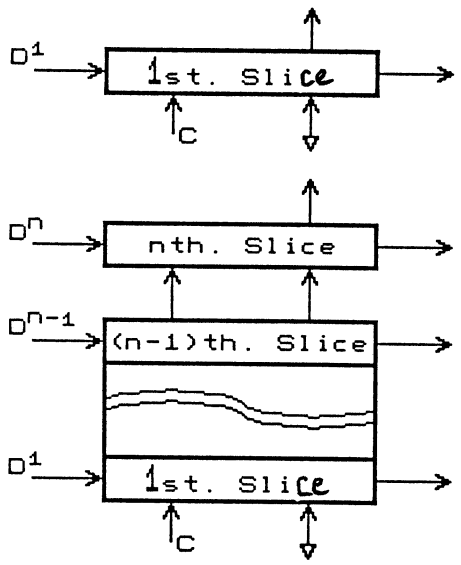


Figure 11

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

$$DP(f, n) = F(S^1, S^2, \cdots, S^n) = \lambda\{D^1, D^2, \cdots, D^n\}[C] .$$

$$(n = 1 \rightarrow (\{f_{cmb}(S^1)\{D^1\}[C]\langle\nabla\rangle\}\langle f_{cmb}(S^1)\{D^1\}[C]\langle\nabla\rangle\rangle, \quad (24)$$

$$f_{seq}(S^1)\{D^1\}[C]\langle\nabla\rangle),$$

$$(\{f_{cmb}(S^n)\{D^n\}[C]\langle\langle DP(f, n-1)\{D^1, D^2, \cdots, D^{n-1}\}[C]\rangle\rangle\}$$

$$\langle f_{cmb}(S^n)\{D^n\}[C]\langle\langle DP(f, n-1)\{D^1, D^2, \cdots, D^{n-1}\}[C]\rangle\rangle\rangle,$$

$$f_{seq}(S^n)\{D^n\}[C]\langle\langle DP(f, n-1)\{D^1, D^2, \cdots, D^{n-1}\}[C]\rangle\rangle)),$$

where $S^i = s_1^i, s_2^i, \cdots, s_q^i$, $1 \le i \le n$, are the values and q is the length of the state vector of *ith.* slice; $D^i = d_1^i, d_2^i, \cdots, d_m^i$, $1 \le i \le n$, are the m data input values; C is the control input vector of each slice; and ∇ is the first slice's status initialization vector.

A few observations are in order here.

- Since the slices are identical†, concatenations can be realized through the abutment[11] of the corresponding layouts.

† - In practice f can be parametrized and the *ith* slice can receive i as its parameter.

- 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 (under some number representation scheme), this formulation satisfies the requirements of certain functionals, such as the carry propagations in a sliced adder.
- A similar formulation exists for the case where the status signal has to propagate in the opposite direction, for example a sliced comparator. Since *control* signals are simply passed through the slices without any modification, simultaneous flows of both formulations will not lead to infinite recursion, as might be feared.
- Definition (24) can be used to extend certain slice properties to that of the data-path itself, through structural induction proof techniques. In the past, designers have assumed this in an implied way and have used the properties of the slice and the corresponding n -wide data-paths in an interchangeable form. We also do this in the next example by defining a single slice, and applying the control part to the slice, assuming that the multi-bit version of the data-path leads to the identical behavior.

9. A Complete Example

The **SDC**-based graphical representation of a circuit designed to calculate the greatest common divisor (*GCD*) of two values at its data-input *ports* '*in* $_1$' and '*in* $_2$', and producing the result at its data-output *port* '*out* ' is shown in Figure 12.

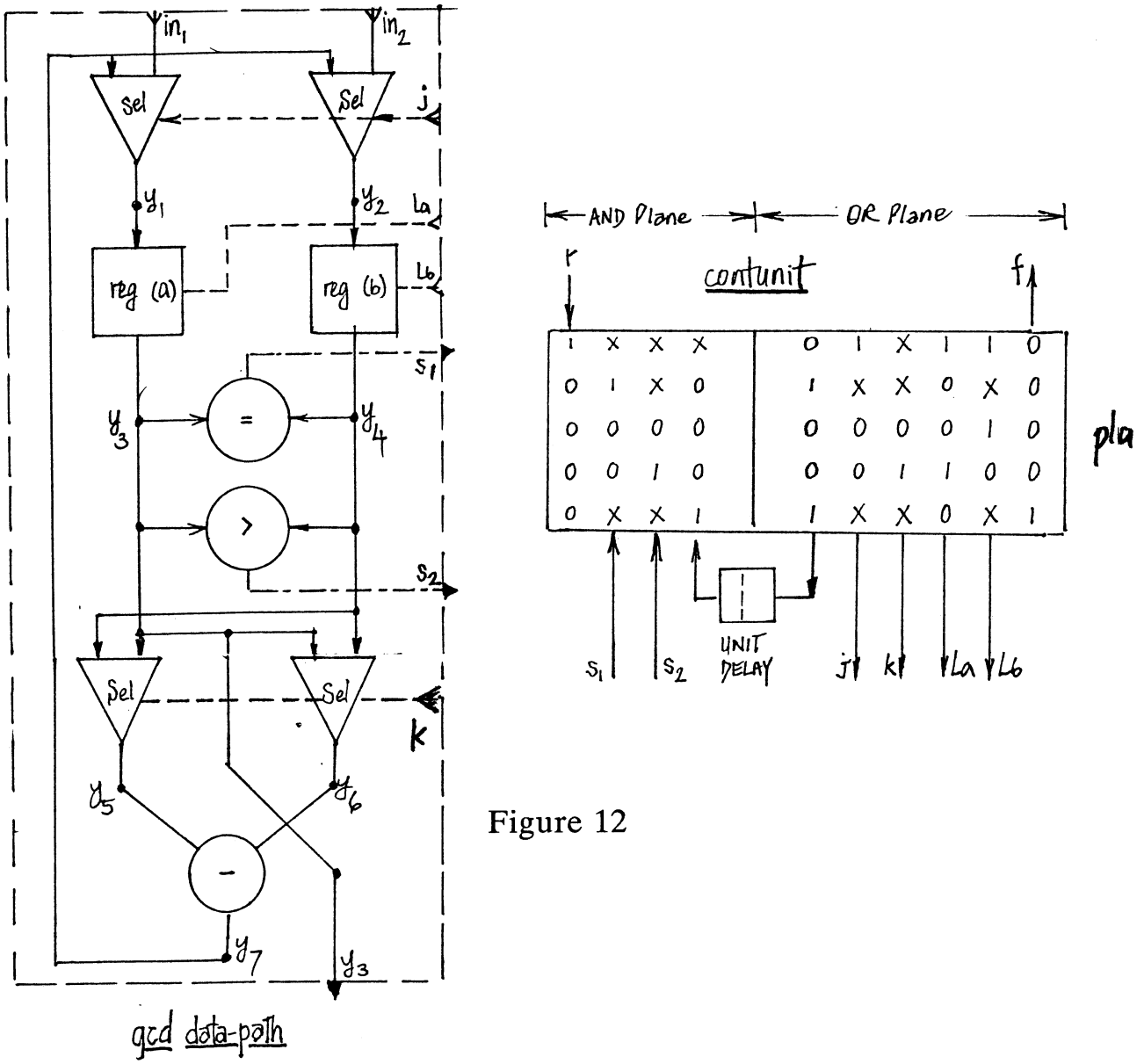


Figure 12

The input values are sampled at the last assertion of the '*r*' (*reset*) *control* input and 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 repeatedly subtracting the smaller value from the larger value until the two values match. It is the purpose of this section to develop the functional models of the data-path and the control parts independently, and to combine them to form the total module's behavioral model.

We start by applying the composition rule (11) to the data-path. Given *functional* primitives:

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

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

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

and the composite register module

$$\textbf{reg}\,(a) = \lambda\{ in \}[ld] . (\{ a \}, \textbf{reg}\,(ld \rightarrow in , a))$$

the **gcd_path** is defined by:

$$\begin{aligned}
\textbf{gcd_path}\,(a , b) = \lambda\{ in_1, in_2 \}[j , k , la , lb] . (\textbf{rec}\,(& \\
& \{ y_1 \} = \textbf{sel}_{cmb}\,\{ y_7, in_1 \}[j]; \\
& \{ y_2 \} = \textbf{sel}_{cmb}\,\{ y_7, in_2 \}[j]; \\
& \{ y_3 \} = \textbf{reg}_{cmb}\,(a)\,\{ y_1 \}[la]; \\
& \{ y_4 \} = \textbf{reg}_{cmb}\,(b)\,\{ y_2 \}[lb]; \\
& \langle s_1 \rangle = \textbf{eql}_{cmb}\,\{ y_3, y_4 \}; \qquad (24) \\
& \langle s_2 \rangle = \textbf{gt}_{cmb}\,\{ y_3, y_4 \}; \\
& \{ y_5 \} = \textbf{sel}_{cmb}\,\{ y_4, y_3 \}[k]; \\
& \{ y_6 \} = \textbf{sel}_{cmb}\,\{ y_3, y_4 \}[k]; \\
& \{ y_7 \} = \textbf{sub}_{cmb}\,\{ y_5, y_6 \})\ \textbf{in}\ (\\
& \quad \{ y_3 \}\langle s_1, s_2 \rangle, \textbf{gcd_path}\ (\textbf{reg}_{seq}\,(a)\{ y_1 \}[la], \\
& \qquad (\textbf{reg}_{seq}\,(b)\{ y_2 \}[lb])).
\end{aligned}$$

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

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

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

This completes the definition of the data-path part.

The control part is made of two sub-modules: the PLA and the *unit-delay* parts. The PLA realizes the microprogram to be executed by the module. The *unit-delay* holds the state of the control-part. We start by first defining the PLA part, called **pla**, and combine it with a *unit-delay* element to form the complete control-part, called **contunit**. Following are these two steps.

$$\textbf{pla}\ = \lambda[\ \mathrm{r}\ ]\ \langle\ \mathrm{s}_1,\ \mathrm{s}_2,\ \mathrm{c}\ \rangle\ .\ ([\ \mathrm{c}\ ',\ \mathrm{j}\ ,\ \mathrm{k}\ ,\ \mathrm{la}\ ,\ \mathrm{lb}\ ]\langle\ \mathrm{f}\ \rangle)$$

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

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

contunit can be reduced to

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

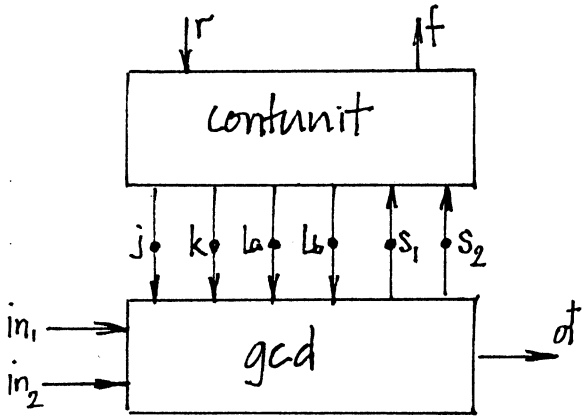


Figure 13

Combining **gcd-path** and **contunit**, as shown in Figure 13, to form the complete module, called **gcd**, leads initially to:

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

$\{\ out\ \}\langle\ s_1, s_2\rangle$ = **gcd_path** $_{cmb}$ (a , b)$\{\ in_1, in_2\}[\ j$, k , la , $lb\ ]$;

$[\ j$, k , la , $lb\ ]\langle\ f\ \rangle$ = **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 exapnds to:

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

$out = a$;

$s_1 = a \equiv b$;

$s_2 = a > b$;

$j = r$;

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

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

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

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

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

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

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

and can eventually be reduces to:

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

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

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

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

where

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

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

10. Acknowledgement

Useful discussions with Dr. Mantis H. M. Cheng of the University of Victoria, British Columbia, especially on formulating design composition rules in terms of bound and free variables, are gratefully acknowledged, as is funding from the University of Waterloo Operating Grant.

11. References

1. Mavaddat, F., A Model for Register-Transfer Level Design Specification: The SDC Notation, CS-84-34, Department of Computer Science, submitted for publication and under revision, University of Waterloo, Waterloo, Ontario (October 1984).
2. Johannsen, D., Bristle Blocks: A Silicon Compiler, *Proceedings of the 16th. Design Automation Conference*, pp. 310-313 (July 1979).
3. Southard, J. R., MacPitts: An Approach to Silicon Compilation, *IEEE Computer Magazine* **16**(12) pp. 74-82 (December 1983).
4. Anceau, F., 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. R. Bryant, Computer Science Press, Rockville, Maryland (1983).
5. 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).
6. Hafer L. J. and A. C. Parker, A formal Method for Specification, Analysis, and Design of Register-Transfer Level Digital Logic, *IEEE Transactions on Computer-Aided Design* **CAD 2**(1) pp. 4-18 (January 1983).
7. Mavaddat, F., An Architecture and Layout for Register Transfer Level IC Design, Report 85-4, Institute for Computer Research, University of Waterloo, Waterloo, Ontario (January 1985).
8. Rowson, James A., Understanding Hierarchical Design, Ph.D Thesis, California Institute of Technology, Pasadena, California (April 1980).
9. Landin, P. J., The Mechanical Evaluation of Expressions, *Computer Journal* **6**(4) pp. 308-320 (Jan. 1984).
10. Cardelli, Luca and Wegner, Peter, On Understanding Types, Data Abstraction, and Polymorphism, *Computing Surveys* **17**(4) pp. 471-522 (December 1985).
11. Carver Mead and Lynn Conway, *Introduction to VLSI Systems,* Addison Wesley, Reading, Mass. (1980).