Printing Requisition / Graphic Services

18587

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: The Correctness of Register-Transfer Design: Inductive Assertions on CS-89-06 Algorithmic State Machines

DATE REQUISITIONED: April 11, 1990

DATE REQUIRED:

ACCOUNT NO.: 4 1 2 6 8 6 6 4 1 (R270 41)

REQUISITIONER – PRINT: Colleen Bernard

PHONE: 2192

SIGNING AUTHORITY: per F. Mavaddat

MAILING INFO – NAME: Colleen Bernard | DEPT.: CS | BLDG. & ROOM NO.: DC-2314 | ☐ 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: 30 | 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: Math fronts and backs enclosed. Pls. staple 3 down left side
[X] COLLATING [X] STAPLING [ ] PUNCHING [ ] PLASTIC COIL

FOLDING/PADDING: | CUTTING SIZE:

Special Instructions

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

Where?

Called G.S. May 7/90

Printing Requisition / Graphic Services

54235

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-89-06

The Correctness of Register-Transfer Design: Inductive Assertions on Algorithmic State Machines

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

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

MAILING INFO – NAME: Sue DeAngelis | DEPT.: C.S. | 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: 30 | 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 | 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 [ ]

To Sue DeAngelis From Farhad

Date Feb. 1. 89

memo

University of Waterloo

89-06

Enclosed is a new manuscript for Research report production.

Please:

① Appendix I (last page - 1) should be numbered as page 26

② Appendix II (last page, numbered as page 2.) should be numbered as page 27.

③ Please put the report number in my mail box

④ Initially ~~50~~ 30 copies should be sufficient.

Thanks

Sue,

If you need more copies just send the following with your req.

Thanks
Marie.

To Farhad

From Sue DeAngelis

Date Nov. 30/89

memo

University of Waterloo

My stock of 89-06 "The Correctness Of Register-Transfer Design..." & 89-08 "Designing and Modeling VLSI Systems" has been depleted. I have not had any requests but let me know if you would like to get any more copies made for future requests.

Thanks.

Should you receive any requests please let me know. I have copies.

Sue

FM

Speed Letter 44-902

GrayLine SNAP-A-WAY® FORM

Speed Letter

To Computer Science

From Ruth Scheid
VLSI Group / EE

Subject Reports

– NO 9 & 10 FOLD

Message

Could we please have copies of the following reports. They were displayed at our recent Research Review & an interest was shown in them. We will mail them to those who showed an interest in them

CS89-06 (3) CS 89-08 (2) Thanks.

Date May 18/89 Signed Ruth Scheed

Reply

sent May 19/89

– No 9 FOLD

– NO 10 FOLD

Date Signed

RETAIN WHITE COPY, RETURN PINK COPY. TURN OVER FOR USE WITH WINDOW ENVELOPE.

SNAP-A-WAY AND RETAIN YELLOW COPY, SEND WHITE AND PINK COPIES WITH CARBON INTACT

The Correctness of Register-Transfer Design: Inductive Assertions on Algorithmic State Machines

Farhad Mavaddat

Dept. of Computer Science
University of Waterloo

Research Report CS-89-06
January, 1989

The Correctness of Register-Transfer Design: Inductive Assertions on Algorithmic State Machines

Farhad Mavaddat

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

ABSTRACT

This paper proposes a method for reasoning about *algorithmic state machines* (ASMs); the method is similar to the use of *inductive assertions* to reason about flow chart programs. The ASM's specification is derived from a candidate design, written in a small set of register-transfer primitives, using a formalism that we introduce early in the paper. The correctness of the ASM specification, and the automatic translation of the candidate design to circuit layout, strengthens our confidence in the correctness of low-level design implementations.

We assign assertions, which signify the expected behaviour of correctly designed hardware, to each *state* box of an ASM chart. To make propositions about a design's interface behaviour, the candidate hardware is extended to include its input and output strings, and the assertions are applied to the charts corresponding to the extended hardware. This extension lets us assign inductive assertions to the input, intermediate, and output states, in an integrated form. We end by discussing the proof steps needed to verify the assertions.

We argue 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 to hardware design than those methods which require newly developed skills, and possibly less well-known mathematical techniques.

The Correctness of Register-Transfer Design: Inductive Assertions on Algorithmic State Machines

Farhad Mavaddat

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

1. INTRODUCTION

A hardware design methodology employing formal verification entails [1]:

1- writing a high-level "requirement specification",

2- designing an implementation, and

3- proving mathematically that the design meets its specification.

To accomplish this, we need a suitable formal system to state the "requirement specification"; a design implementation paradigm, at some suitable level of abstraction, to implement a candidate design; and a mapping from the candidate design to the formal system, so that the designer can verify the design's correctness through symbolic reasoning.

The choice of formal system and implementation paradigm are central to how practical, and hence acceptable, the proof process will be, and to the degree of confidence one has in the final design.

In the remainder of this paper we propose a method for capturing a candidate design's behavior and "requirement specification" with a single algorithmic state machine (ASM) [2]. We also propose a method for proving the correctness of the derived ASMs. The method is similar to the *inductive assertions technique* [3] for proving the correctness of software. By way of example, we will follow a small, but non-trivial, design throughout this paper, and end by proving that its proposed implementation meets its requirement specification. Our ability to discuss a design of this size in a few pages demonstrates the power and simplicity of our method.

Hanna and Daeche discuss desirable properties of proof techniques [4], and argue that the "formal systems should already exist" ... "be powerful and concise" ... and "not too removed from the digital engineer's intuition." One can well appreciate that the use of existing and powerful systems will help with the practicality of the proof process.

It is our contention that the combination of inductive assertions and algorithmic state machines in the proof of hardware correctness meets the most important of the criteria for acceptability proposed by Hanna and Daeche.

2. Mathematical preliminaries

In this section we use an extension of the Lambda calculus to model digital designs. Later we will use this formalism as a mapping mechanism between hardware designs and the corresponding ASM charts.

A combinational circuit's behavior is modelled by a syntactic extension of the Lambda calculus. Sequential circuit behavior is harder to define, and we only give an intuitive description of the formalism used. For a more formal treatment of the subject, based on denotational semantics, the reader is referred to [5]. For a *typed* extension of the model used in the formal definition of register-transfer designs see [6].

In this section we also define the formalism needed to derive the behavior of a composite module from the behavior of its sub-modules.

2.1. Defining Combinational Modules

We define an *m*-input, *n*-output (*m*×*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 side of (2) 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.

2.2. 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 , at state $(s_1, s_2, \cdots , s_q)$, can be defined by

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

where

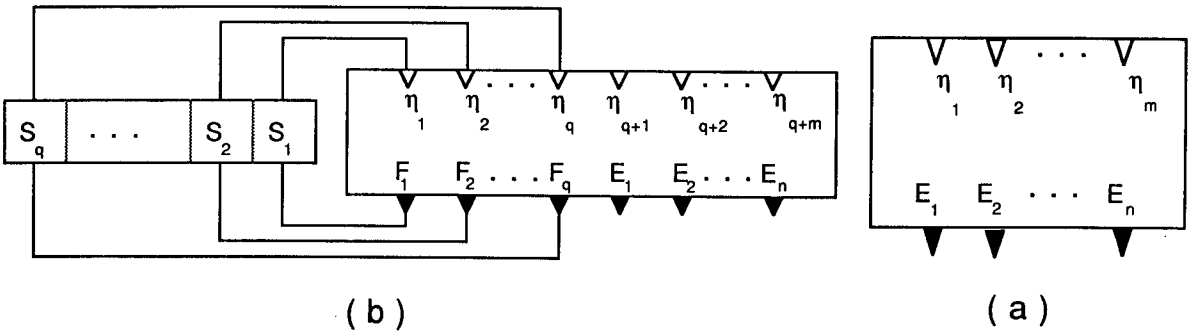

Figure 1. Graphical representation of modules with m inputs and n outputs: a) combinational module, b) sequential module with q state variables S_i, $1 \leq i \leq q$.

- 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

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

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

- to distinguish between the state and the port inputs, we have moved 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 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 *jth* state variable.

We also write

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

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.

2.3. Composite Modules

An *m×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×m-put* environment module $\mathbf{f}^{\,s}$, where the input and output ports of $\mathbf{f}^{\,s}$ define the output and the input ports of $\mathbf{f}^{\,c}$ respectively, as shown in Figure 2.

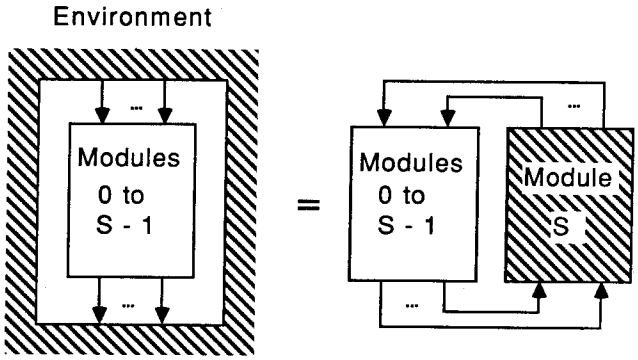

Figure 2. Modeling the environment of modules *0* to *S-1* as the module *S*.

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 output ports, respectively, where I^i, $0 \le i \le s$, and O^i, $0 \le i \le s$, are the sets of input and output ports of the i*–th* 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, 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

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

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 $y_j \in h(O^i)$, $0 \leq j \leq n^i$, 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 $\mathbf{f}^{\,0}, \mathbf{f}^{\,1}, \cdots, \mathbf{f}^{\,s}$, using the connection nets P, can be defined as

$$\mathbf{f}^{\,c}(S^1, S^2, \cdots, S^{s-1}) = \lambda(h(O^s)) \cdot (\textbf{rec}$$
$$(Y^i = \mathbf{f}^{\,i}_{cmb}(S^i)(X^i) \quad 1 \leq i \leq s-1) \qquad (8)$$
$$\textbf{in } (h(I^s), \mathbf{f}^{\,c}(\mathbf{f}^{\,i}_{seq}(S^i)(X^i) \quad 1 \leq i \leq s-1))),$$

where

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

and

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

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 $\mathbf{f}^{\,i}$, $1 \leq i \leq s$, and **rec** and **in** are defined as in [7].

3. The Register-Transfer Design Paradigm

In this section we first discuss the type of signals used in a register-transfer design. This helps us with the specification of the legal compositions of the design primitives. Only legal compositions are amenable to our design analysis and proof techniques.

Next, we present four design primitives which are the building blocks of our design environment. The primitives have three important properties:

- they are designer friendly, namely, they provide the designer with design primitives close to his normal design experience, and are thus easy to work with [8, 9].
- they are easy to implement, in regular forms, within the constraints of sound integrated-circuit design. The details of their implementations can be found in [10].
- the behaviour of the primitives and their compositions are easy to formalize and therefore amenable to mathematical treatment.

After discussing the primitives, we present a complete design that computes the greatest common divisor (GCD) of two positive integers. We use this design later in the report to demonstrate our correctness proof techniques.

3.1. Signal Types

The signals in a register-transfer design belong to one of three categories:

- *Data* signals carry values from one primitive of the *data-path* to another. The inputs and outputs of a register or an ALU are examples of these. *Data* signals also form the data inputs and data outputs of the design. In this paper we assume that *data* signals are positive integers for multi-bit data-paths, and logical values for one bit data-path slices; we use solid lines to illustrate the *data* signals.
- *Control* signals are inputs from the control-unit to the data-path; they help to dynamically reconfigure the data-path, thereby rerouting data values. Examples include the 'load' command to a register, which reconfigures the data-path to accept a new or an old value, and the 'operation-code' command to an ALU, which reconfigures the ALU into one of its several capabilities. We use dotted lines to illustrate *control* signals.
- *Status* signals indicate the status of a data-path. *Status* outputs inform the control-part of the prevailing conditions inside the data-path. Examples include the 'carry' signal out of an adder and the signal out of a comparator. The carry-in signal to an adder is an example of a *status* input signal. We use dashed lines to illustrate *status* signals.

3.2. The Selector Primitive

A *selector*, **sel** , Figure 3.a, is a 3×1–*put* combinational device whose behaviour is defined by

$$\textbf{sel} = \lambda(d_1, d_2, c) \,.\, (c \rightarrow d_2, d_1), \tag{9}$$

where '→' denotes *if_then_else* . Definition (9) indicates that the only output of a selector, a *data* signal, is equal to one of its two *data* inputs, d_1 or d_2 , and the selection is made according to the value of *control* input c .

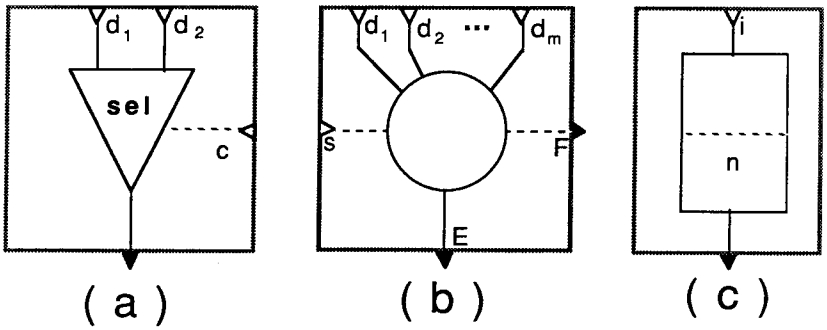

Figure 3. Data-path primitives: a) selectors, b) functionals, c) unit-delays. Data, control, and status signals are shown as solid, dotted, and dashed lines, respectively. Modules are enclosed in patterned boxes to indicate their interface boundaries.

3.3. Functional Primitives

Functionals are a family of *(m+k)×(n+k)-put* combinational devices, shown in Figure 3.b, where $m \geq 1$, $k, n \in \{0,1\}$, and $n + k \geq 1$.

The behaviour of a functional can be defined in one of the following three ways:

$$\lambda(d_1, d_2, \cdots, d_m, s) \,.\, (E, G), \tag{10}$$

$$\lambda(d_1, d_2, \cdots, d_m, s) \,.\, (G), \tag{11}$$

$$\lambda(d_1, d_2, \cdots, d_m) \,.\, (E), \tag{12}$$

where d_i , $1 \leq i \leq m$, are the *data* inputs, s is the *status* input, E is the *data* output, and G is the *status* output.

We now present a few typical functionals, specified according to the ways discussed in (10)-(12).

Ex.1- Binary 'and' Module:

A binary **and** module is defined by

$$\textbf{and} = \lambda(a, b) . (a \wedge b). \tag{13}$$

Ex.2- Binary 'equal' Module

A binary **equal** module is defined by

$$\textbf{equal} = \lambda(a, b, s) . (s \wedge \overline{(a \oplus b)}), \tag{14}$$

where a and b are the module'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' Module

A decimal **add** module is defined by

$$\textbf{add} = \lambda(a, b, s) . ((a + b + num(s)) \; mod \; 10, \; num(s) + a + b > 9), \tag{15}$$

where a and b are the module's *data* inputs, s is the carry input from the less-significant neighbouring slice, $(a + b + num(s))$ is the *data* output, $(num(s) + a + b > 9)$ is the carry to the more-significant neighbouring slice, and $num : \mathbf{B} \rightarrow \mathbf{N}$ is a function that produces the numerical equivalent of the *status* input signal.

3.4. The Table Primitives

Tables are a family of $p \times q$ *–put* combinational modules. Syntactically, a table **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: **C**, the condition sub-array, and **A**, the action sub-array, of p and q columns, respectively, and n rows. Each column of **C** 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 *ith* row of **T** to be enabled if the input values match the corresponding **C** entries. The x entries of the table match both the 1 and the 0 values of the input. When the *ith* row of **T** is enabled, the corresponding elements of **A** appear as the module's outputs. An x output indicates a *floating* output.

Typically, we capture a table's semantics with the functional form

$$T = \lambda(\eta_1, \eta_2, \cdots, \eta_p) \cdot (B_1, B_2, \cdots, B_q), \tag{16}$$

where the B_i, $1 \leq i \leq q$, are the sum of the products of the bound variables and their complements. Tables are best implemented as PLA structures.

3.5. The Unit-Delay Primitive

A *unit-delay*, **del**, is a 1×1–*put*, single-state sequential device, shown in Figure 3.c. A unit-delay's output lags its input by one system-wide clock pulse. The behaviour of a unit-delay primitive can be defined by

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

Unit-delay elements are implemented using a pair of inverters and a two-phase non-overlapping clock.

Delays are polymorphic [11] devices, and can be used to delay all three types of signals.

3.6. A Complete Register Transfer Design Example

A register-transfer design consists of a data-path and control-unit, as shown in Figure 4. The data-path part is the processing element of the design, accepting data inputs and producing data outputs. The control-unit part may or may not accept external inputs; in either case it issues action signals to the data-path, instructing it on its next action. In many applications, the control-unit has to sample the status of the data-path in order to issue its next control signal.

In our approach, explained more fully in [6, 8, 9], the data-path is made of selector, functional and unit-delay elements, connected only through their *data* ports. The unconnected data ports form the design's data ports. Any of the connected or unconnected output ports of a data-path's primitive elements can form the design's output ports. The *control* inputs of the selectors and the *status* outputs of the functionals of a data-path, form the data-path's action inputs and status outputs, and should be connected to the control unit.

The control-unit of a design is made of a table, or possibly a hierarchy of tables, and zero or more unit-delay primitives to hold the control-unit's state information. This combination of table(s) and unit-delays help to translate the data-path's status and the environment's control inputs, if any, to actions applied to the data-path and module-status, as shown in Figure 4. In the remaining part of this section, we present a circuit design based on these principles.

Figure 5 illustrates a circuit which calculates the greatest common divisor (*GCD*) of two values at its data-input *ports*, 'in_1' and 'in_2', and produces the result at its data-output *port* '*ot*'.

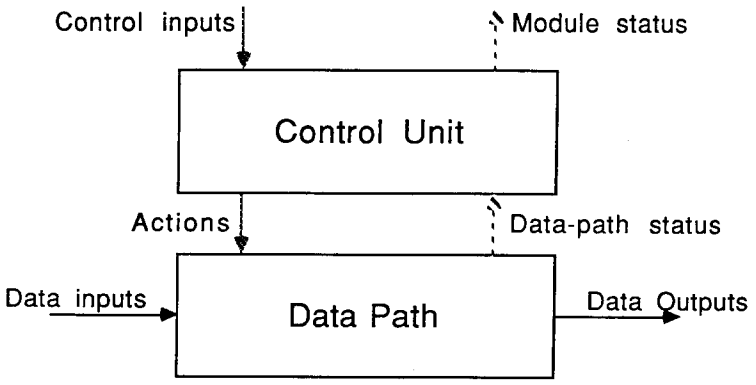

Figure 4. Schematic representation of a register-transfer model. The design is divided into a data-path and a control-unit.

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 subsection to develop the functional model of the data-path and the control-parts independently. In other applications, one may proceed to combine the two behaviours to derive an overall model of the module.

Given functionals

$$\mathbf{eql} = \lambda(\, a\,, b\,)\,.\,(\, a = b\,)$$

$$\mathbf{gt} = \lambda(\, b\,, b\,)\,.\,(\, a > b\,)$$

$$\mathbf{sub} = \lambda(\, a\,, b\,)\,.\,(\, a - b\,)$$

and the composite register module

$$\mathbf{reg}\,(\, a\,) = \lambda(\, in\,, ld\,)\,.\,(\, a\,, \mathbf{reg}\,(\, ld \rightarrow in\,, a\,)),$$

and applying the composition rule (8) to the data-path, the **gcd_path** is defined by

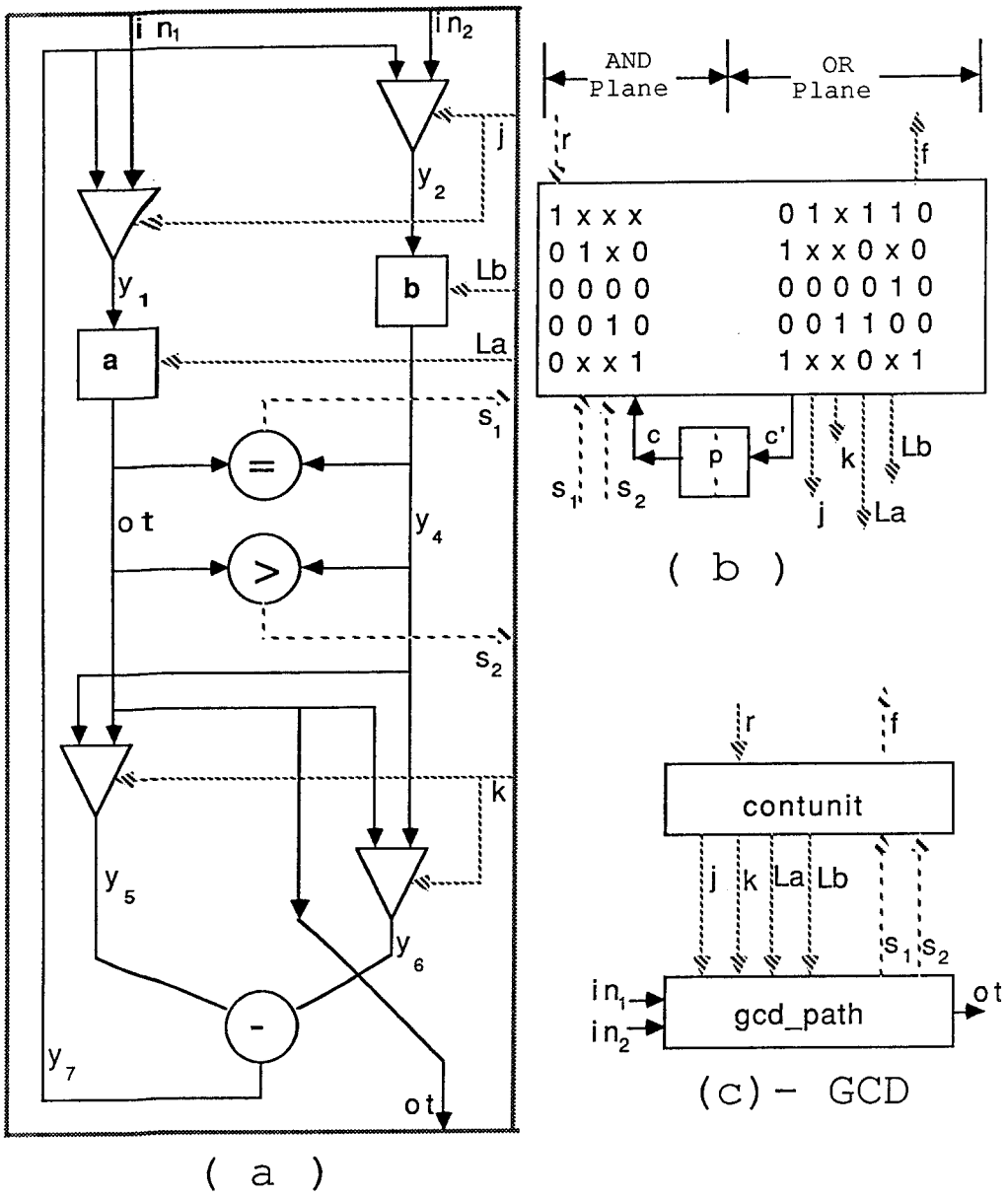

Figure 5. Separate representations of a data-path (a) and a control-unit (b) to calculate the greatest common divisor of two positive integers at inputs in_1 and in_2. Input r signals the start of the computation. Output f signals the availability of the results at ot. The combined form, called GCD, is shown in (c). Boxes labeled as **a** and **b** depict *registers*.

$$
\begin{aligned}
\textbf{gcd\_path}\ (\ a\ ,\ b\ ) = \lambda(\ in_1,\ in_2,\ j\ ,\ k\ ,\ la\ ,\ lb\ )\ .\ (\textbf{rec}\ (\\
&(\ y_1\ ) = \textbf{sel}_{cmb}\ (\ y_7,\ in_1,\ j\ );\\
&(\ y_2\ ) = \textbf{sel}_{cmb}\ (\ y_7,\ in_2,\ j\ );\\
&(\ ot\ ) = \textbf{reg}_{cmb}\ (\ a\ )\ (\ y_1,\ la\ );\\
&(\ y_4\ ) = \textbf{reg}_{cmb}\ (\ b\ )\ (\ y_2,\ lb\ );\\
&(\ s_1\ ) = \textbf{eql}_{cmb}\ (\ ot\ ,\ y_4\ );\\
&(\ s_2\ ) = \textbf{gt}_{cmb}\ (\ ot\ ,\ y_4\ );\\
&(\ y_5\ ) = \textbf{sel}_{cmb}\ (\ y_4,\ ot\ ,\ k\ \ );\\
&(\ y_6\ ) = \textbf{sel}_{cmb}\ (\ ot\ ,\ y_4,\ k\ \ );\\
&(\ y_7\ ) = \textbf{sub}_{cmb}\ (\ y_5,\ y_6\ ))\ \textbf{in}\ (\\
&\quad (\ ot\ ,\ s_1,\ s_2\ ),\ \textbf{gcd\_path}\ (\textbf{reg}_{seq}\ (\ a\ )(\ y_1,\ la\ ),\\
&\qquad\qquad \textbf{reg}_{seq}\ (\ b\ )(\ y_2,\ lb\ )))).
\end{aligned}
\tag{18}
$$

Note that in this example we assume that the *status* inputs to the data-path have been implicitly initialized.

After expansion and simplification, the **gcd_path** behaviour reduces to

$$
\begin{aligned}
\textbf{gcd\_path}\ (\ a\ ,\ b\ \ ) = \lambda(\ in_1,\ in_2,\ j\ ,\ k\ ,\ la\ ,\ lb\ )\ .\ ((\ a\ ,\ a\ = b\ ,\ a > b\ ),\\
\textbf{gcd\_path}\ ((\ la \rightarrow (\ j \rightarrow in_1,\ (\ k \rightarrow (\ a - b\ ),(b - a\ ))),\ a\ ),\\
(\ lb \rightarrow (\ j \rightarrow in_2,\ (\ k \rightarrow (\ a - b\ ),(b - a\ ))),\ b\ )))
\end{aligned}
\tag{19}
$$

This completes the definition of the data-path part.

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

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

which is expanded to

$$
\begin{aligned}
\textbf{pla}\ = \lambda(\ r\ ,\ s_1,\ s_2\ ,\ c\ \ )\ .\ (\ (\ \bar{r} \wedge \bar{c} \wedge s_1\ ) \vee (\bar{r} \wedge c\ ),\\
r\ ,\ (\ \bar{r} \wedge \overline{s_1} \wedge s_2 \wedge \bar{c}\ ),\ r \vee (\ \bar{r}\ \wedge \overline{s_1} \wedge s_2 \wedge \bar{c}\ ),
\end{aligned}
\tag{20}
$$

$$r \vee (\bar{r} \wedge \overline{s_1} \wedge \overline{s_2} \wedge \bar{c}), \bar{r} \wedge c),$$

and

$$\begin{aligned} \textbf{contunit}\ (\ p\ ) = \lambda(\ r\ ,\ s_1,\ s_2).(\textbf{rec}\ (& \\ (\ c\ ',\ j\ ,\ k\ ,\ la\ ,\ lb\ ,\ f\ ) = \textbf{pla}_{\ cmb}\ (\ r\ ,\ s_1,\ s_2,\ c\ ); & \qquad (21) \\ (\ c\ ) = \textbf{del}_{\ cmb}\ (\ p\ )(\ c\ '))\ \textbf{in}\ (& \\ (\ j\ ,\ k\ ,\ la\ ,\ lb\ ,\ f\ ),\ \textbf{contunit}\ (\ \textbf{del}_{\ seq}\ (\ p\ )(\ c\ '\ )))). & \end{aligned}$$

contunit can be reduced to

$$\begin{aligned} \textbf{contunit}\ (\ p\ ) = \lambda(\ r\ ,\ s_1,\ s_2\ ).(\ r\ ,\ \bar{r} \wedge \overline{s_1} \wedge s_2 \wedge \bar{p}\ , & \\ r \vee(\ \bar{r} \wedge \overline{s_1} \wedge s_2 \wedge \bar{p}\ ),\ r \vee(\ \bar{r} \wedge \overline{s_1} \wedge \overline{s_2} \wedge \bar{p}\ ) & \qquad (22) \\ (\ \bar{r} \wedge p\ ),\ \textbf{contunit}\ ((\bar{r} \wedge \bar{p} \wedge s_1) \vee (\bar{r} \wedge p\ ))). & \end{aligned}$$

4. 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 [2, 12], and can be regarded as a variation of the state-diagram method for specifying state-machines.

Each ASM chart consists of the appropriate interconnection of three basic elements: a *state* box, a *condition* box, and a *conditional-output* box. 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. The behaviour of a combinational circuit can be represented by an ASM with a single *state* box.

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. In Figure 6 we show the ASM-based behaviour of a *JK*-flipflop. In this and other ASM charts 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 register-transfer-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.

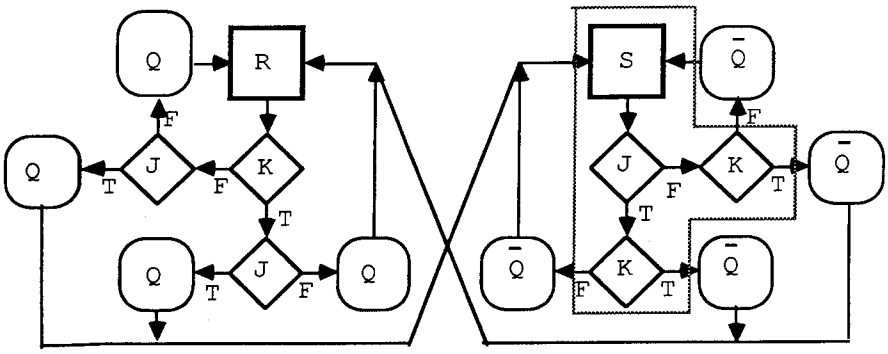

Figure 6. ASM definition of a synchronous *JK* flip-flop. Output lists Q and $\overline{Q}$ are moved to the output boxes. Using standard ASM conventions, Q and $\overline{Q}$ appear in the S and R boxes, respectively, with no need for output boxes. The dotted line encloses the boxes forming a *state block*.

Similarities between the ASM specification of hardware and the flow-chart specification of computer programs suggests the use *inductive assertions* [4] 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 wider user familiarity with the method, since it is the widely taught method 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 5, 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 greatest common divisor 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 7 we show an i/o_extension of the GCD module to include 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 5, is simulated by the design extension 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 the $gcd(m, n)$ are both single values, so unit-delay elements f and *ot* are used to represent them, respectively.

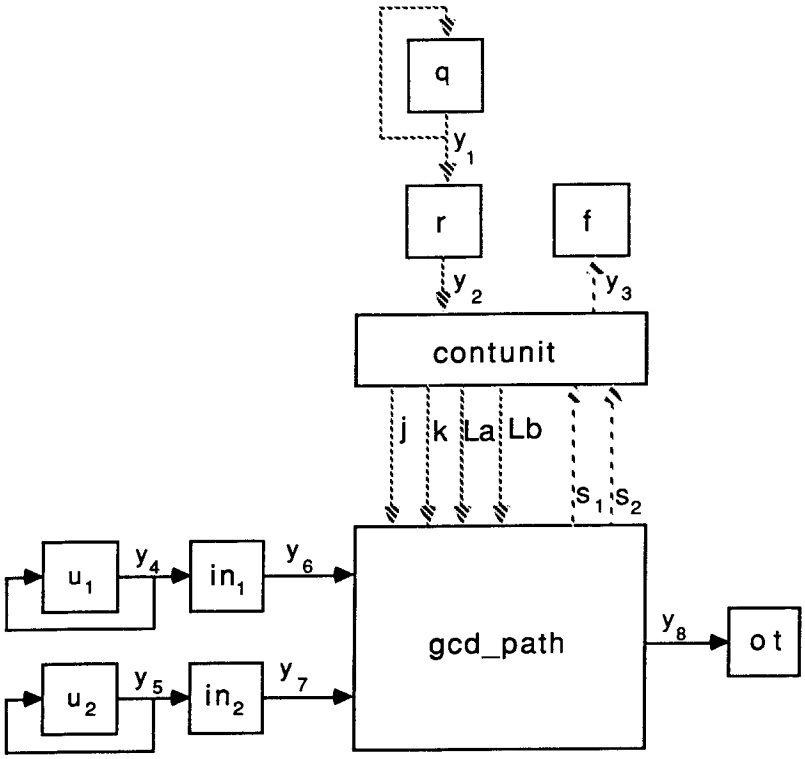

Figure 7. Extending the **gcd** design of Figure 5.c to include its inputs, output, control, and status signal strings.

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

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

holds between the new states of 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 total 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 control-unit and data-path of respectively. We have

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

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

$$\begin{aligned}
&\textbf{e_contunit}\ (p, r, q, f) = \lambda(s_1, s_2).\ ((r,\ (\bar{r} \wedge \bar{p} \wedge \overline{s_1} \wedge s_2),\\
&\qquad (r \vee (\bar{r} \wedge \bar{p} \wedge \overline{s_1} \wedge s_2)), (r \vee (\bar{r} \wedge \bar{p} \wedge \overline{s_1} \wedge \overline{s_2}))),\\
&\qquad \textbf{e_contunit}\ ((\bar{r} \wedge \bar{p} \wedge s_1) \vee (\bar{r} \wedge p), q, q, \bar{r} \wedge p)).
\end{aligned} \tag{23}$$

Similarly,

$$\begin{aligned}
&\textbf{e_gcd_path}\ (a, b, in_1, in_2, u_1, u_2, ot) = \lambda(j, k, La, Lb).\ (\textbf{rec}\ (\\
&\qquad (y_4) = \textbf{del}_{cmb}\ (u_1)(y_4);\\
&\qquad (y_5) = \textbf{del}_{cmb}\ (u_2)(y_5);\\
&\qquad (y_6) = \textbf{del}_{cmb}\ (in_1)(y_4);\\
&\qquad (y_7) = \textbf{del}_{cmb}\ (in_2)(y_5);\\
&\qquad (y_8, s_1, s_2) = \textbf{gcd_path}\ (a, b)(y_6, y_7, j, k, La, Lb))\ \textbf{in}\ (\\
&\qquad\quad (s_1, s_2), \textbf{e_gcd_path}\ (\textbf{gcd_path}_{seq}\ (a, b)(y_6, y_7, j, k, La, Lb),\\
&\qquad\quad \textbf{del}_{seq}\ (in_1)(y_4), \textbf{del}_{seq}\ (in_2)(y_5), \textbf{del}_{seq}\ (u_1)(y_4),\\
&\qquad\quad \textbf{del}_{seq}\ (u_2)(y_5), \textbf{del}_{seq}\ (ot)(y_8)))).
\end{aligned}$$

This simplifies to

$$\textbf{e\_gcd\_path}\ (a, b, in_1, in_2, u_1, u_2, ot) = \lambda(j, k, La, Lb).\ ($$

$$((a = b), (a > b)), \textbf{e\_gcd\_path}\ (\ (La \rightarrow (j \rightarrow in_1, (k \rightarrow (a - b), (b - a))), a), \quad (La \rightarrow (j \rightarrow in_2, (k \rightarrow (a - b), (b - a))), b), \quad u_1, u_2, u_1, u_2, a)) . \tag{24}$$

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 the sequential (5) and combinational (4) behaviour models of the extended control-unit to derive the corresponding next state and action outputs. Due to the closed nature of the extension process, the 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** and definitions (4) and (5), we obtain the following sequential and combinational behaviours:

$$\textbf{e\_contunit}_{seq}\ (p, r, q, f) = \lambda(s_1, s_2) . \tag{25}$$
$$((\bar{r} \wedge \bar{p} \wedge s_1) \vee (\bar{r} \wedge p), q, q, (\bar{r} \wedge p))$$

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

The results of the search process using (25) and (26) are listed in Appendix I. The same results are illustrated in Figure 8.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 *output* and *condition* boxes. These additions complete the derivation of the ASM chart.

Appendix II gives the results of applying the **e_contunit**'s action outputs to the following sequential and combinational behaviours of **e_gcd_path**:

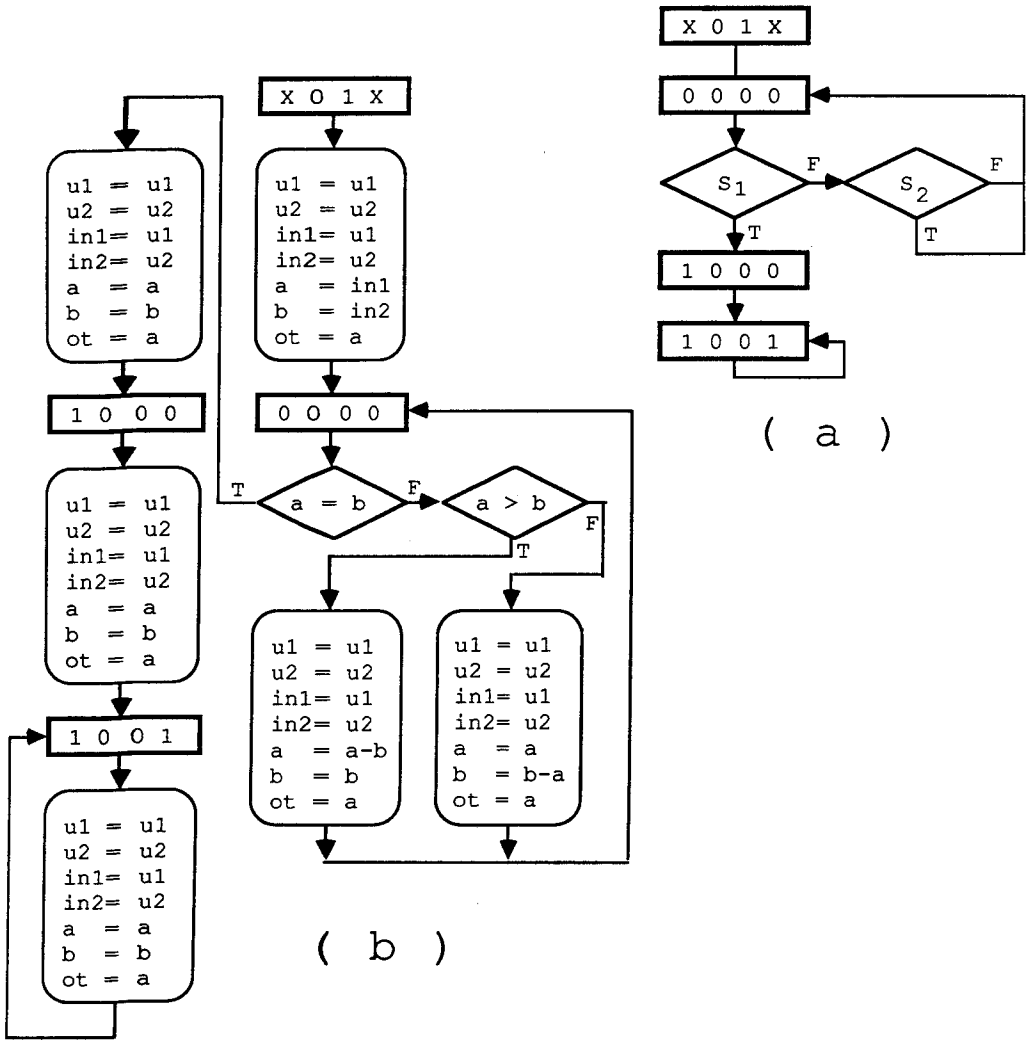

Figure 8. The ASM charts corresponding to the GCD design of Figure 7. (a) The flow-control-part, depicting the behaviour of the control-unit; strings in each state box represent the values in unit-delays *p*, *q*, *r* and *f* at that state, where 'X' stands for an unknown state. (b) The combined behaviour of the control-unit and the data-path.

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

$$\textbf{e\_gcd\_path}\,(\,a\,,\,b\,,\,in_1,\,in_2,\,u_1,\,u_2,\,ot\,) = \lambda(\,j\,,\,k\,,\,La\,,\,Lb\,)\,.\,(\,(\,a = b\,),\,(\,a > b\,)\,) \tag{28}$$

The ASM chart corresponding to the **e-gcd** module is shown in Figure 8.b.

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 and expected to be true at the start and 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,

$$P\,(\,i\,)\,\{R\,,\,Q\,\}\,P\,(\,j\,) \tag{28}$$

holds. In (28), 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 (28) is due to Hoare [13], 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 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 (28) 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*. Obviously, 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 9 shows a version of the ASM chart given in Figure 8.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.

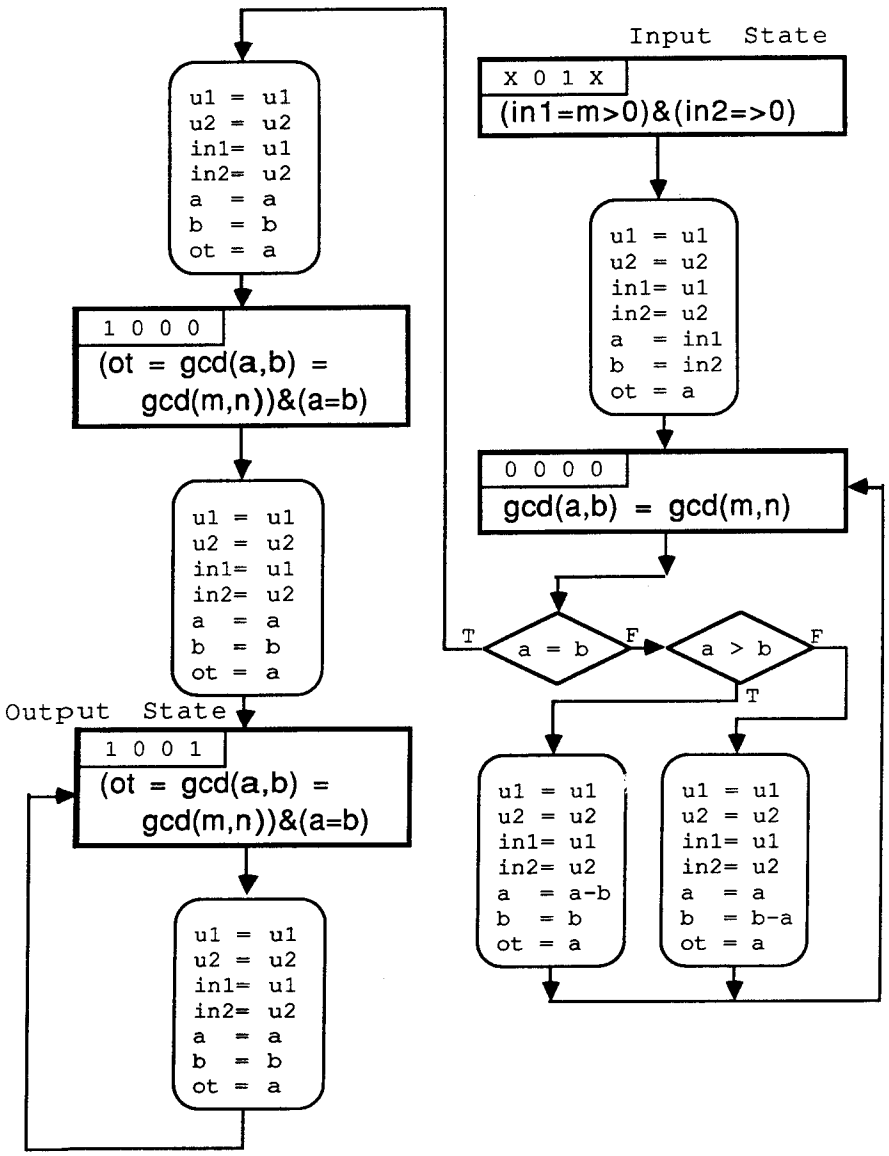

Figure 9. The GCD module's ASM chart, with the correctness propositions for each state boxes. Each proposition can be derived from the preceding proposition(s); the input proposition is assumed *true*.

5. SUMMARY

In this paper we integrated established techniques with several novel methods to form the main components of a hardware verification methodology; these consist of the "requirement specification", "design definition", and "reasoning" phases of a register-transfer design paradigm.

For the design definition, we proposed a friendly register-transfer design environment based on a small set of primitives, and a signal-typing scheme that controls the composition of legal designs. As well as being close to the designer's normal design experience, the primitives are amenable to a mathematical treatment. We proposed a complete functional model for specifying the primitives and the behaviour of their compositions. Later, we used the formalism to derive ASM specifications of candidate designs.

We then introduced automatic translations between the design and the implementation, and the design and the proof environment, and argued that performing the design at a level lower than that of the proof environment leads to improved confidence in the final implementation.

We showed that in order to make assertions about the behaviour of a candidate design at the interfaces, we had to extend the design to include its input and output strings. Finally, the ASM charts were derived from the extended designs, and 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 is 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.

ACKNOWLEDGEMENT

We gratefully acknowledge the University of Waterloo funding and computing facilities used to carry out the work reported here.

6. References

1. Gordon, Mike, Hardware Verification by Formal Proof, Technical Report No. 74, Computer Laboratory, University of Cambridge, Cambridge, England (August 1985).

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

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

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

5. Gordon M., 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. Mavaddat, F., A Functional Model of Register-Transfer Design, CS-88-16, Dept. of Computer Science, University of Waterloo, Waterloo, Ontario (April 1988).

7. Landin, P. J., The Mechanical Evaluation of Expressions, *Computer Journal* **6**(4) pp. 308-320 (Jan. 1984).

8. 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).

9. Mavaddat, F., Designing and Modeling VLSI Systems at Register Transfer Level, *to appear in International Journal of Computer Aided VLSI Design* **1**(2) pp. 41-1 (June 1989).

10. Carver Mead and Lynn Conway, *Introduction to VLSI Systems,* Addison Wesley, Reading, Mass. (1980).

11. Cardelli, Luca and Wegner, Peter, On Understanding Types, Data Abstraction, and Polymorphism, *Computing Surveys* **17**(4) pp. 471-522 (December 1985).

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

13. Hoare, C. A. R., An Axiomatic Basis for Computer Programming, *Communications of the ACM* **12**(10) pp. 576-583 (October 1969).

APPENDIX I

State-Table for the Flow-Control Part of the GCD chart

Present-State				Status-inputs		Next-State				Action-outputs			
p	q	r	f	s1	s2	p	q	r	f	j	k	La	Lb
x	0	1	x	0	0	0	0	0	0	1	0	1	1
x	0	1	x	0	1	0	0	0	0	1	0	1	1
x	0	1	x	1	0	0	0	0	0	1	0	1	1
x	0	1	x	1	1	0	0	0	0	1	0	1	1
0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	1	0	0	0	0	0	1	1	0
0	0	0	0	1	0	1	0	0	0	0	0	0	0
0	0	0	0	1	1	1	0	0	0	0	0	0	0
1	0	0	0	0	0	1	0	0	1	0	0	0	0
1	0	0	0	0	1	1	0	0	1	0	0	0	0
1	0	0	0	1	0	1	0	0	1	0	0	0	0
1	0	0	0	1	1	1	0	0	1	0	0	0	0
1	0	0	1	0	0	1	0	0	1	0	0	0	0
1	0	0	1	0	1	1	0	0	1	0	0	0	0
1	0	0	1	1	0	1	0	0	1	0	0	0	0
1	0	0	1	1	1	1	0	0	1	0	0	0	0

Next-state and action-outputs are obtained by substituting the corresponding present-state and status-input values in the behaviour expressions of **e_contunit**, i.e. (25) and (26), respectively. Only states reachable from the initial state of 'x 0 1 x' are listed. An 'x' entry indicates an unknown unit-delay state. With reference to the **e_gcd_path**, $(s\,1 = 1) \wedge (s\,2 = 1)$ is not possible.

APPENDIX II

Data-Path Assignments Table

0000		0100		1000		1100	
u1	= u1	u1	= u1	u1	= u1	u1	= u1
u2	= u2	u2	= u2	u2	= u2	u2	= u2
in1	= u1	in1	= u1	in1	= u1	in1	= u1
in2	= u2	in2	= u2	in2	= u2	in2	= u2
a	= a	a	= a	a	= a	a	= a
b	= b	b	= b	b	= b	b	= b
ot	= a	ot	= a	ot	= a	ot	= a
0001		**0101**		**1001**		**1101**	
u1	= u1	u1	= u1	u1	= u1	u1	= u1
u2	= u2	u2	= u2	u2	= u2	u2	= u2
in1	= u1	in1	= u1	in1	= u1	in1	= u1
in2	= u2	in2	= u2	in2	= u2	in2	= u2
a	= a	a	= a	a	= a	a	= a
b	= b - a	b	= a - b	b	= in2	b	= in2
ot	= a	ot	= a	ot	= a	ot	= a
0010		**0110**		**1010**		**1110**	
u1	= u1	u1	= u1	u1	= u1	u1	= u1
u2	= u2	u2	= u2	u2	= u2	u2	= u2
in1	= u1	in1	= u1	in1	= u1	in1	= u1
in2	= u2	in2	= u2	in2	= u2	in2	= u2
a	= b - a	a	= a - b	a	= in1	a	= in1
b	= b	b	= b	b	= b	b	= b
ot	= a	ot	= a	ot	= a	ot	= a
0011		**0111**		**1011**		**1111**	
u1	= u1	u1	= u1	u1	= u1	u1	= u1
u2	= u2	u2	= u2	u2	= u2	u2	= u2
in1	= u1	in1	= u1	in1	= u1	in1	= u1
in2	= u2	in2	= u2	in2	= u2	in2	= u2
a	= b - a	a	= a - b	a	= in1	a	= in1
b	= b - a	b	= a - b	b	= in2	b	= in2
ot	= a	ot	= a	ot	= a	ot	= a

Each block of entries represents a unique action-input and the corresponding set of register-transfer assignments. The first entry of each block represents, from left to right, the action-input values corresponding to the *j*, *k*, *La*, and *Lb* ports of the data-path respectively.

The Correctness of Register-Transfer Design: Inductive Assertions on Algorithmic State Machines

Farhad Mavaddat

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

ABSTRACT

This paper proposes a method for reasoning about *algorithmic state machines* (ASMs); the method is similar to the use of *inductive assertions* to reason about flow chart programs. The ASM's specification is derived from a candidate design, written in a small set of register-transfer primitives, using a formalism that we introduce early in the paper. The correctness of the ASM specification, and the automatic translation of the candidate design to circuit layout, strengthens our confidence in the correctness of low-level design implementations.

We assign assertions, which signify the expected behaviour of correctly designed hardware, to each *state* box of an ASM chart. To make propositions about a design's interface behaviour, the candidate hardware is extended to include its input and output strings, and the assertions are applied to the charts corresponding to the extended hardware. This extension lets us assign inductive assertions to the input, intermediate, and output states, in an integrated form. We end by discussing the proof steps needed to verify the assertions.

We argue 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 to hardware design than those methods which require newly developed skills, and possibly less well-known mathematical techniques.

The Correctness of Register-Transfer Design: Inductive Assertions on Algorithmic State Machines

Farhad Mavaddat

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

1. INTRODUCTION

A hardware design methodology employing formal verification entails [1]:

1- writing a high-level "requirement specification",

2- designing an implementation, and

3- proving mathematically that the design meets its specification.

To accomplish this, we need a suitable formal system to state the "requirement specification"; a design implementation paradigm, at some suitable level of abstraction, to implement a candidate design; and a mapping from the candidate design to the formal system, so that the designer can verify the design's correctness through symbolic reasoning.

The choice of formal system and implementation paradigm are central to how practical, and hence acceptable, the proof process will be, and to the degree of confidence one has in the final design.

In the remainder of this paper we propose a method for capturing a candidate design's behavior and "requirement specification" with a single algorithmic state machine (ASM) [2]. We also propose a method for proving the correctness of the derived ASMs. The method is similar to the *inductive assertions technique* [3] for proving the correctness of software. By way of example, we will follow a small, but non-trivial, design throughout this paper, and end by proving that its proposed implementation meets its requirement specification. Our ability to discuss a design of this size in a few pages demonstrates the power and simplicity of our method.

Hanna and Daeche discuss desirable properties of proof techniques [4], and argue that the "formal systems should already exist" ... "be powerful and concise" ... and "not too removed from the digital engineer's intuition." One can well appreciate that the use of existing and powerful systems will help with the practicality of the proof process.

It is our contention that the combination of inductive assertions and algorithmic state machines in the proof of hardware correctness meets the most important of the criteria for acceptability proposed by Hanna and Daeche.

2. Mathematical preliminaries

In this section we use an extension of the Lambda calculus to model digital designs. Later we will use this formalism as a mapping mechanism between hardware designs and the corresponding ASM charts.

A combinational circuit's behavior is modelled by a syntactic extension of the Lambda calculus. Sequential circuit behavior is harder to define, and we only give an intuitive description of the formalism used. For a more formal treatment of the subject, based on denotational semantics, the reader is referred to [5]. For a *typed* extension of the model used in the formal definition of register-transfer designs see [6].

In this section we also define the formalism needed to derive the behavior of a composite module from the behavior of its sub-modules.

2.1. Defining Combinational Modules

We define an *m*-input, *n*-output (*m*×*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 side of (2) 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.

2.2. 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 , at state $(s_1, s_2, \cdots , s_q)$, can be defined by

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

where

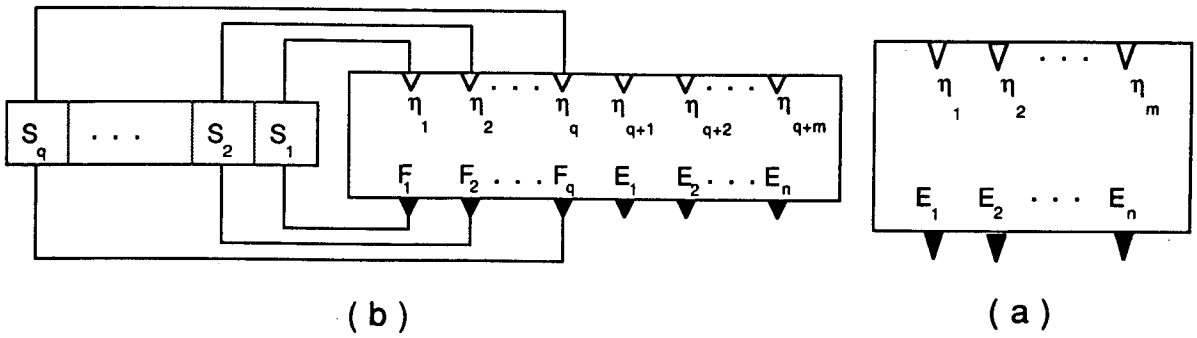

Figure 1. Graphical representation of modules with m inputs and n outputs: a) combinational module, b) sequential module with q state variables S_i, $1 \leq i \leq q$.

- 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

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

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

- to distinguish between the state and the port inputs, we have moved 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 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 *jth* state variable.

We also write

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

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.

2.3. Composite Modules

An *m×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×m-put* environment module $\mathbf{f}^{\,s}$, where the input and output ports of $\mathbf{f}^{\,s}$ define the output and the input ports of $\mathbf{f}^{\,c}$ respectively, as shown in Figure 2.

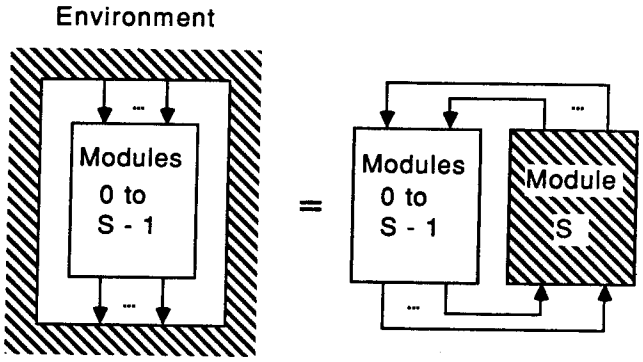

Figure 2. Modeling the environment of modules *0* to *S-1* as the module *S*.

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 output ports, respectively, where I^i, $0 \le i \le s$, and O^i, $0 \le i \le s$, are the sets of input and output ports of the i*–th* 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, 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

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

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 $y_j \in h(O^i)$, $0 \leq j \leq n^i$, 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 $\mathbf{f}^{\,0}, \mathbf{f}^{\,1}, \cdots, \mathbf{f}^{\,s}$, using the connection nets P, can be defined as

$$\mathbf{f}^{\,c}(S^1, S^2, \cdots, S^{s-1}) = \lambda(h(O^s)) \cdot (\textbf{rec}$$
$$(Y^i = \mathbf{f}^{\,i}_{cmb}(S^i)(X^i) \quad 1 \leq i \leq s-1) \qquad (8)$$
$$\textbf{in}\ (h(I^s), \mathbf{f}^{\,c}(\mathbf{f}^{\,i}_{seq}(S^i)(X^i) \quad 1 \leq i \leq s-1))),$$

where

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

and

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

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 $\mathbf{f}^{\,i}$, $1 \leq i \leq s$, and **rec** and **in** are defined as in [7].

3. The Register-Transfer Design Paradigm

In this section we first discuss the type of signals used in a register-transfer design. This helps us with the specification of the legal compositions of the design primitives. Only legal compositions are amenable to our design analysis and proof techniques.

Next, we present four design primitives which are the building blocks of our design environment. The primitives have three important properties:

- they are designer friendly, namely, they provide the designer with design primitives close to his normal design experience, and are thus easy to work with [8, 9].
- they are easy to implement, in regular forms, within the constraints of sound integrated-circuit design. The details of their implementations can be found in [10].
- the behaviour of the primitives and their compositions are easy to formalize and therefore amenable to mathematical treatment.

After discussing the primitives, we present a complete design that computes the greatest common divisor (GCD) of two positive integers. We use this design later in the report to demonstrate our correctness proof techniques.

3.1. Signal Types

The signals in a register-transfer design belong to one of three categories:

- *Data* signals carry values from one primitive of the *data-path* to another. The inputs and outputs of a register or an ALU are examples of these. *Data* signals also form the data inputs and data outputs of the design. In this paper we assume that *data* signals are positive integers for multi-bit data-paths, and logical values for one bit data-path slices; we use solid lines to illustrate the *data* signals.
- *Control* signals are inputs from the control-unit to the data-path; they help to dynamically reconfigure the data-path, thereby rerouting data values. Examples include the 'load' command to a register, which reconfigures the data-path to accept a new or an old value, and the 'operation-code' command to an ALU, which reconfigures the ALU into one of its several capabilities. We use dotted lines to illustrate *control* signals.
- *Status* signals indicate the status of a data-path. *Status* outputs inform the control-part of the prevailing conditions inside the data-path. Examples include the 'carry' signal out of an adder and the signal out of a comparator. The carry-in signal to an adder is an example of a *status* input signal. We use dashed lines to illustrate *status* signals.

3.2. The Selector Primitive

A *selector*, **sel** , Figure 3.a, is a 3×1–*put* combinational device whose behaviour is defined by

$$\textbf{sel} = \lambda(d_1, d_2, c) \,.\, (c \rightarrow d_2, d_1), \tag{9}$$

where '→' denotes *if_then_else* . Definition (9) indicates that the only output of a selector, a *data* signal, is equal to one of its two *data* inputs, d_1 or d_2 , and the selection is made according to the value of *control* input c .

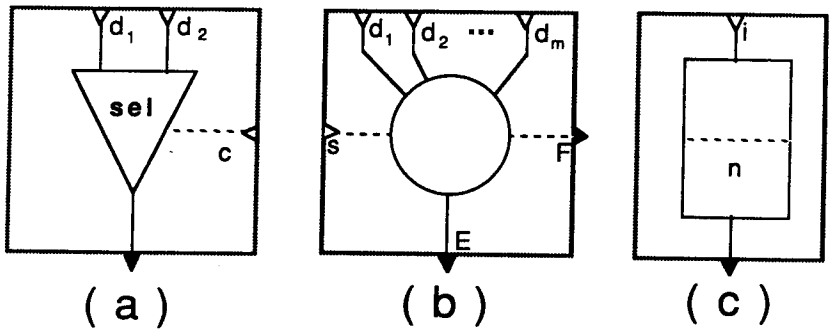

Figure 3. Data-path primitives: a) selectors, b) functionals, c) unit-delays. Data, control, and status signals are shown as solid, dotted, and dashed lines, respectively. Modules are enclosed in patterned boxes to indicate their interface boundaries.

3.3. Functional Primitives

Functionals are a family of *(m+k)×(n+k)-put* combinational devices, shown in Figure 3.b, where $m \geq 1$, k , $n \in \{0,1\}$, and $n + k \geq 1$.

The behaviour of a functional can be defined in one of the following three ways:

$$\lambda(d_1, d_2, \cdots, d_m, s) \,.\, (E, G), \tag{10}$$

$$\lambda(d_1, d_2, \cdots, d_m, s) \,.\, (G), \tag{11}$$

$$\lambda(d_1, d_2, \cdots, d_m) \,.\, (E), \tag{12}$$

where d_i , $1 \leq i \leq m$, are the *data* inputs, s is the *status* input, E is the *data* output, and G is the *status* output.

We now present a few typical functionals, specified according to the ways discussed in (10)-(12).

Ex.1- Binary 'and' Module:

A binary **and** module is defined by

$$\textbf{and} = \lambda(a, b) . (a \wedge b). \tag{13}$$

Ex.2- Binary 'equal' Module

A binary **equal** module is defined by

$$\textbf{equal} = \lambda(a, b, s) . (s \wedge \overline{(a \oplus b)}), \tag{14}$$

where a and b are the module'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' Module

A decimal **add** module is defined by

$$\textbf{add} = \lambda(a, b, s) . ((a + b + num(s)) \; mod \; 10, \; num(s) + a + b > 9), \tag{15}$$

where a and b are the module's *data* inputs, s is the carry input from the less-significant neighbouring slice, $(a + b + num(s))$ is the *data* output, $(num(s) + a + b > 9)$ is the carry to the more-significant neighbouring slice, and $num : \mathbf{B} \rightarrow \mathbf{N}$ is a function that produces the numerical equivalent of the *status* input signal.

3.4. The Table Primitives

Tables are a family of $p \times q$ *–put* combinational modules. Syntactically, a table **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: **C**, the condition sub-array, and **A**, the action sub-array, of p and q columns, respectively, and n rows. Each column of **C** 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 *ith* row of **T** to be enabled if the input values match the corresponding **C** entries. The x entries of the table match both the 1 and the 0 values of the input. When the *ith* row of **T** is enabled, the corresponding elements of **A** appear as the module's outputs. An x output indicates a *floating* output.

Typically, we capture a table's semantics with the functional form

$$T = \lambda(\eta_1, \eta_2, \cdots, \eta_p) \,.\, (B_1, B_2, \cdots, B_q), \tag{16}$$

where the B_i, $1 \leq i \leq q$, are the sum of the products of the bound variables and their complements. Tables are best implemented as PLA structures.

3.5. The Unit-Delay Primitive

A *unit-delay*, **del**, is a 1×1–*put*, single-state sequential device, shown in Figure 3.c. A unit-delay's output lags its input by one system-wide clock pulse. The behaviour of a unit-delay primitive can be defined by

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

Unit-delay elements are implemented using a pair of inverters and a two-phase non-overlapping clock.

Delays are polymorphic [11] devices, and can be used to delay all three types of signals.

3.6. A Complete Register Transfer Design Example

A register-transfer design consists of a data-path and control-unit, as shown in Figure 4. The data-path part is the processing element of the design, accepting data inputs and producing data outputs. The control-unit part may or may not accept external inputs; in either case it issues action signals to the data-path, instructing it on its next action. In many applications, the control-unit has to sample the status of the data-path in order to issue its next control signal.

In our approach, explained more fully in [6, 8, 9], the data-path is made of selector, functional and unit-delay elements, connected only through their *data* ports. The unconnected data ports form the design's data ports. Any of the connected or unconnected output ports of a data-path's primitive elements can form the design's output ports. The *control* inputs of the selectors and the *status* outputs of the functionals of a data-path, form the data-path's action inputs and status outputs, and should be connected to the control unit.

The control-unit of a design is made of a table, or possibly a hierarchy of tables, and zero or more unit-delay primitives to hold the control-unit's state information. This combination of table(s) and unit-delays help to translate the data-path's status and the environment's control inputs, if any, to actions applied to the data-path and module-status, as shown in Figure 4. In the remaining part of this section, we present a circuit design based on these principles.

Figure 5 illustrates a circuit which calculates the greatest common divisor (*GCD*) of two values at its data-input *ports*, 'in_1' and 'in_2', and produces the result at its data-output *port* '*ot*'.

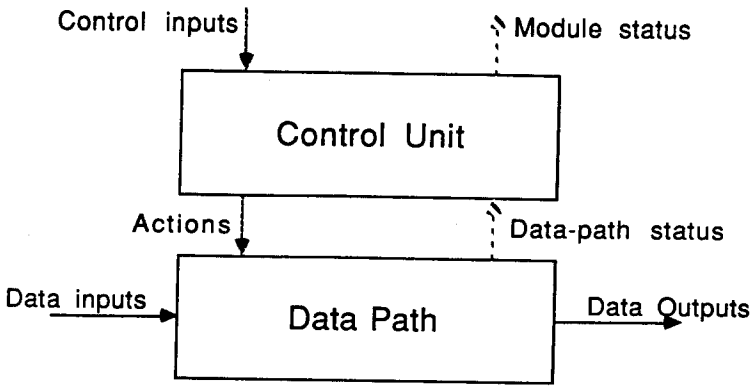

Figure 4. Schematic representation of a register-transfer model. The design is divided into a data-path and a control-unit.

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 subsection to develop the functional model of the data-path and the control-parts independently. In other applications, one may proceed to combine the two behaviours to derive an overall model of the module.

Given functionals

$$\mathbf{eql} = \lambda(\, a\,,\, b\,)\,.\,(\, a = b\,)$$

$$\mathbf{gt} = \lambda(\, b\,,\, b\,)\,.\,(\, a > b\,)$$

$$\mathbf{sub} = \lambda(\, a\,,\, b\,)\,.\,(\, a - b\,)$$

and the composite register module

$$\mathbf{reg}\,(\, a\,) = \lambda(\, in\;,\, ld\,)\,.\,(\;\, a\;\,,\, \mathbf{reg}\,(\, ld \rightarrow in\,,\, a\,)),$$

and applying the composition rule (8) to the data-path, the **gcd_path** is defined by

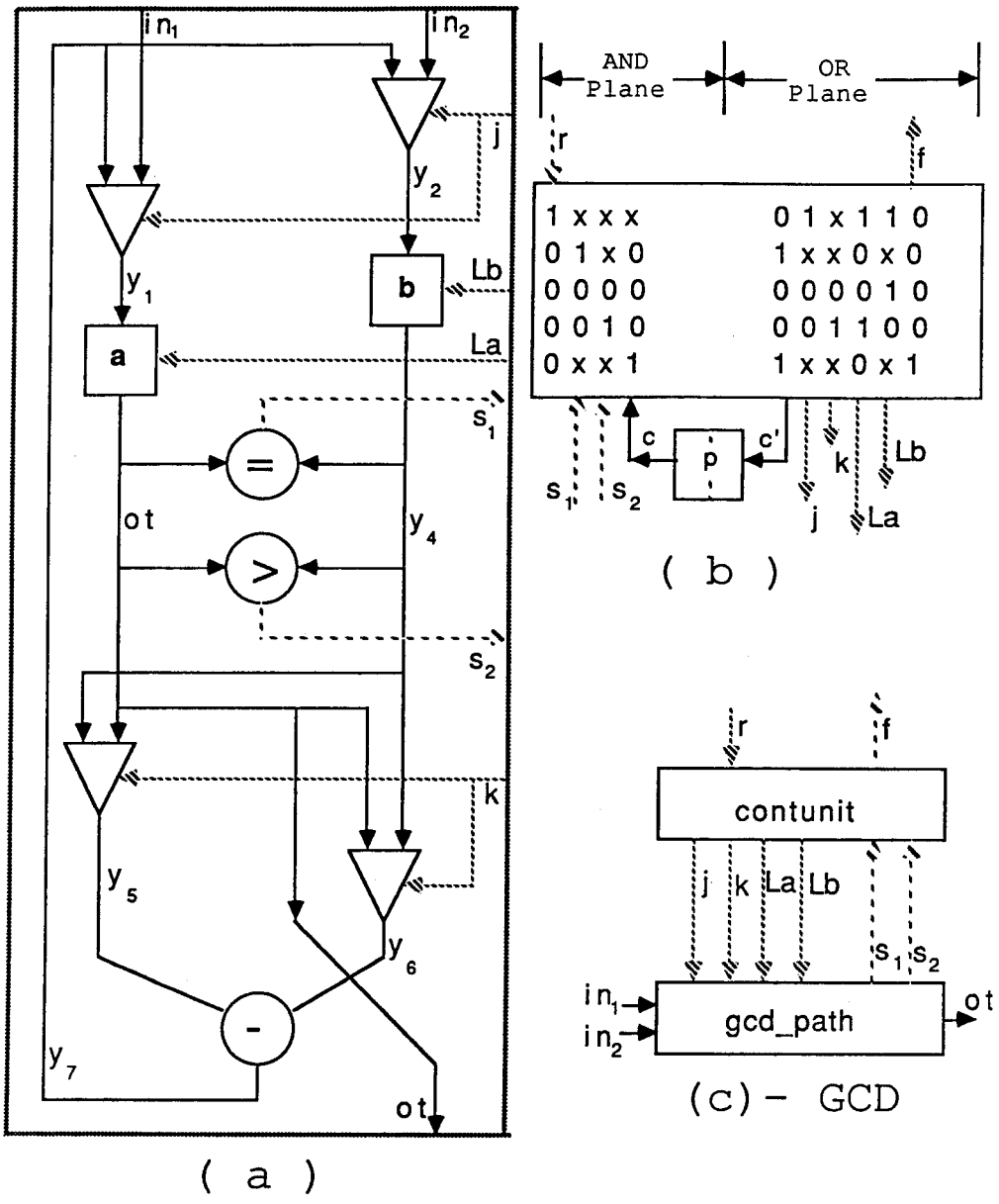

Figure 5. Separate representations of a data-path (a) and a control-unit (b) to calculate the greatest common divisor of two positive integers at inputs in_1 and in_2. Input r signals the start of the computation. Output f signals the availability of the results at ot. The combined form, called GCD, is shown in (c). Boxes labeled as **a** and **b** depict *registers*.

$$
\begin{aligned}
\textbf{gcd\_path}\ (\ a\ ,\ b\ ) = \lambda(\ in_1,\ in_2,\ j\ ,\ k\ ,\ la\ ,\ lb\ )\ .\ (\textbf{rec}\ (\\
&(\ y_1\ ) = \textbf{sel}_{cmb}\ (\ y_7,\ in_1,\ j\ );\\
&(\ y_2\ ) = \textbf{sel}_{cmb}\ (\ y_7,\ in_2,\ j\ );\\
&(\ ot\ ) = \textbf{reg}_{cmb}\ (\ a\ )\ (\ y_1,\ la\ );\\
&(\ y_4\ ) = \textbf{reg}_{cmb}\ (\ b\ )\ (\ y_2,\ lb\ );\\
&(\ s_1\ ) = \textbf{eql}_{cmb}\ (\ ot\ ,\ y_4\ );\\
&(\ s_2\ ) = \textbf{gt}_{cmb}\ (\ ot\ ,\ y_4\ );\\
&(\ y_5\ ) = \textbf{sel}_{cmb}\ (\ y_4,\ ot\ ,\ k\ \ );\\
&(\ y_6\ ) = \textbf{sel}_{cmb}\ (\ ot\ ,\ y_4,\ k\ \ );\\
&(\ y_7\ ) = \textbf{sub}_{cmb}\ (\ y_5,\ y_6\ ))\ \textbf{in}\ (\\
&\quad (\ ot\ ,\ s_1,\ s_2\ ),\ \textbf{gcd\_path}\ (\textbf{reg}_{seq}\ (\ a\ )(\ y_1,\ la\ ),\\
&\qquad\qquad \textbf{reg}_{seq}\ (\ b\ )(\ y_2,\ lb\ )))).
\end{aligned}
\tag{18}
$$

Note that in this example we assume that the *status* inputs to the data-path have been implicitly initialized.

After expansion and simplification, the **gcd_path** behaviour reduces to

$$
\begin{aligned}
\textbf{gcd\_path}\ (\ a\ ,\ b\ \ ) = \lambda(\ in_1,\ in_2,\ j\ ,\ k\ ,\ la\ ,\ lb\ )\ .\ ((\ a\ ,\ a\ = b\ ,\ a > b\ ),\\
\textbf{gcd\_path}\ ((\ la \rightarrow (\ j \rightarrow in_1,\ (\ k \rightarrow (\ a - b\ ),(b - a\ ))),\ a\ ),\\
(\ lb \rightarrow (\ j \rightarrow in_2,\ (\ k \rightarrow (\ a - b\ ),(b - a\ ))),\ b\ )))
\end{aligned}
\tag{19}
$$

This completes the definition of the data-path part.

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

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

which is expanded to

$$
\begin{aligned}
\textbf{pla}\ = \lambda(\ r\ ,\ s_1,\ s_2\ ,\ c\ \ )\ .\ (\ (\ \bar{r} \wedge \bar{c} \wedge s_1\ ) \vee (\bar{r} \wedge c\ ),\\
r\ ,\ (\ \bar{r} \wedge \overline{s_1} \wedge s_2 \wedge \bar{c}\ ),\ r \vee (\ \bar{r}\ \wedge \overline{s_1} \wedge s_2 \wedge \bar{c}\ ),
\end{aligned}
\tag{20}
$$

$$r \vee (\bar{r} \wedge \overline{s_1} \wedge \overline{s_2} \wedge \bar{c}), \bar{r} \wedge c),$$

and

$$\begin{aligned} \textbf{contunit}\ (\ p\ ) = \lambda(\ r\ ,\ s_1,\ s_2).(\textbf{rec}\ (& \\ (\ c\ ',\ j\ ,\ k\ ,\ la\ ,\ lb\ ,\ f\ ) = \textbf{pla}_{\ cmb}\ (\ r\ ,\ s_1,\ s_2,\ c\ ); & \qquad (21) \\ (\ c\ ) = \textbf{del}_{\ cmb}\ (\ p\ )(\ c\ '))\ \textbf{in}\ (& \\ (\ j\ ,\ k\ ,\ la\ ,\ lb\ ,\ f\ ),\ \textbf{contunit}\ (\ \textbf{del}_{\ seq}\ (\ p\ )(\ c\ '\ )))). & \end{aligned}$$

contunit can be reduced to

$$\begin{aligned} \textbf{contunit}\ (\ p\ ) = \lambda(\ r\ ,\ s_1,\ s_2\ ).(\ r\ ,\ \bar{r} \wedge \overline{s_1} \wedge s_2 \wedge \bar{p}\ , & \\ r \vee (\ \bar{r} \wedge \overline{s_1} \wedge s_2 \wedge \bar{p}\ ),\ r \vee (\ \bar{r} \wedge \overline{s_1} \wedge \overline{s_2} \wedge \bar{p}\ ) & \qquad (22) \\ (\ \bar{r} \wedge p\ ),\ \textbf{contunit}\ ((\bar{r} \wedge \bar{p} \wedge s_1) \vee (\bar{r} \wedge p\ ))). & \end{aligned}$$

4. 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 [2, 12], and can be regarded as a variation of the state-diagram method for specifying state-machines.

Each ASM chart consists of the appropriate interconnection of three basic elements: a *state* box, a *condition* box, and a *conditional-output* box. 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. The behaviour of a combinational circuit can be represented by an ASM with a single *state* box.

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. In Figure 6 we show the ASM-based behaviour of a *JK*-flipflop. In this and other ASM charts 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 register-transfer-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.

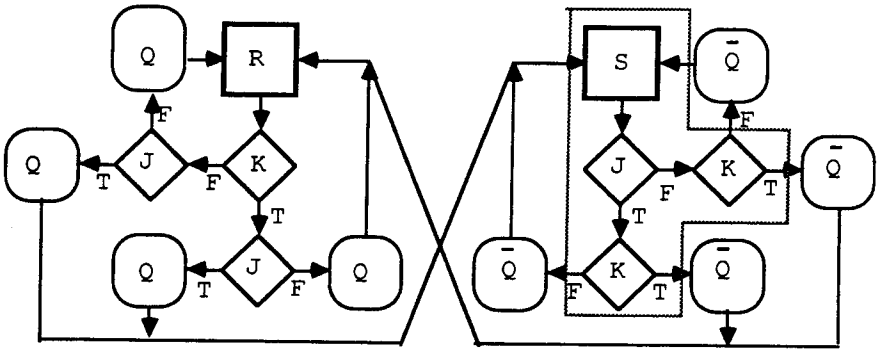

Figure 6. ASM definition of a synchronous *JK* flip-flop. Output lists Q and $\bar{Q}$ are moved to the output boxes. Using standard ASM conventions, Q and $\bar{Q}$ appear in the S and R boxes, respectively, with no need for output boxes. The dotted line encloses the boxes forming a *state block*.

Similarities between the ASM specification of hardware and the flow-chart specification of computer programs suggests the use *inductive assertions* [4] 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 wider user familiarity with the method, since it is the widely taught method 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 5, 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 greatest common divisor 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 7 we show an i/o_extension of the GCD module to include 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 5, is simulated by the design extension 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 the $gcd(m, n)$ are both single values, so unit-delay elements f and *ot* are used to represent them, respectively.

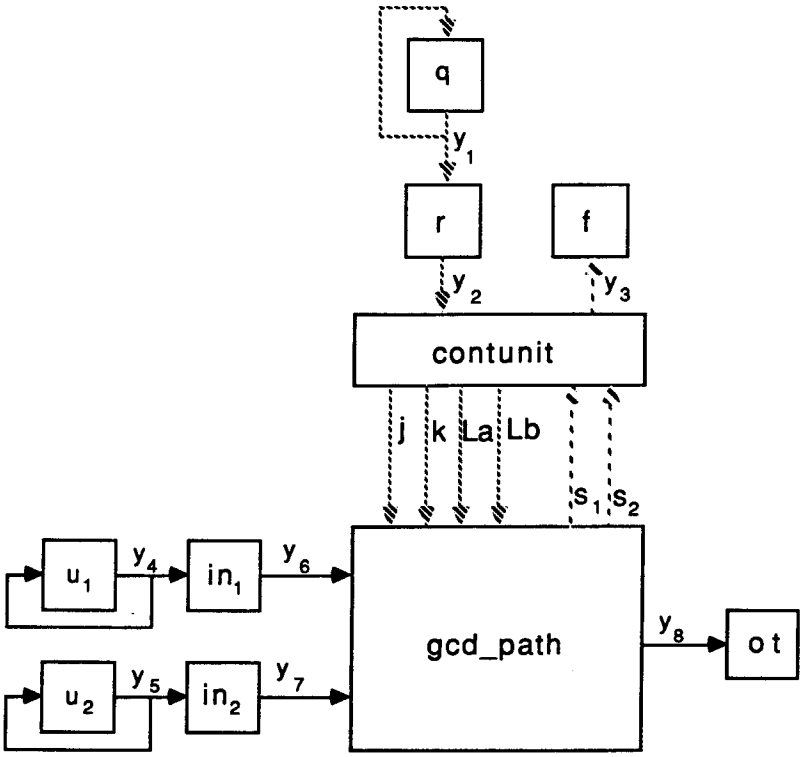

Figure 7. Extending the **gcd** design of Figure 5.c to include its inputs, output, control, and status signal strings.

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

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

holds between the new states of 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 total 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 control-unit and data-path of respectively. We have

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

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

$$\begin{aligned}\textbf{e\_contunit}\ (p, r, q, f) = \lambda(s_1, s_2).\ (&(r,\ (\bar{r} \wedge \bar{p} \wedge \overline{s_1} \wedge s_2),\\ &(r \vee (\bar{r} \wedge \bar{p} \wedge \overline{s_1} \wedge s_2)), (r \vee (\bar{r} \wedge \bar{p} \wedge \overline{s_1} \wedge \overline{s_2}))),\\ &\textbf{e\_contunit}\ ((\bar{r} \wedge \bar{p} \wedge s_1) \vee (\bar{r} \wedge p), q, q, \bar{r} \wedge p)).\end{aligned} \tag{23}$$

Similarly,

$$\begin{aligned}\textbf{e\_gcd\_path}\ (a, b, in_1, in_2, u_1, u_2, ot) = \lambda(j, k, La, Lb).\ (\textbf{rec}\ (&\\ &(y_4) = \textbf{del}_{cmb}\ (u_1)(y_4);\\ &(y_5) = \textbf{del}_{cmb}\ (u_2)(y_5);\\ &(y_6) = \textbf{del}_{cmb}\ (in_1)(y_4);\\ &(y_7) = \textbf{del}_{cmb}\ (in_2)(y_5);\\ &(y_8, s_1, s_2) = \textbf{gcd\_path}\ (a, b)(y_6, y_7, j, k, La, Lb))\ \textbf{in}\ (\\ &\quad (s_1, s_2), \textbf{e\_gcd\_path}\ (\textbf{gcd\_path}_{seq}\ (a, b)(y_6, y_7, j, k, La, Lb),\\ &\quad \textbf{del}_{seq}\ (in_1)(y_4), \textbf{del}_{seq}\ (in_2)(y_5), \textbf{del}_{seq}\ (u_1)(y_4),\\ &\quad \textbf{del}_{seq}\ (u_2)(y_5), \textbf{del}_{seq}\ (ot)(y_8)))).\end{aligned}$$

This simplifies to

$$\textbf{e\_gcd\_path}\ (a, b, in_1, in_2, u_1, u_2, ot) = \lambda(j, k, La, Lb).\ ($$

$$((a = b), (a > b)), \textbf{e\_gcd\_path}\ (\ (La \rightarrow (j \rightarrow in_1, (k \rightarrow (a - b), (b - a))), a), \quad (La \rightarrow (j \rightarrow in_2, (k \rightarrow (a - b), (b - a))), b), \quad u_1, u_2, u_1, u_2, a)) . \tag{24}$$

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 the sequential (5) and combinational (4) behaviour models of the extended control-unit to derive the corresponding next state and action outputs. Due to the closed nature of the extension process, the 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** and definitions (4) and (5), we obtain the following sequential and combinational behaviours:

$$\textbf{e\_contunit}_{seq}\ (p, r, q, f) = \lambda(s_1, s_2) . \tag{25}$$
$$((\bar{r} \wedge \bar{p} \wedge s_1) \vee (\bar{r} \wedge p), q, q, (\bar{r} \wedge p))$$

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

The results of the search process using (25) and (26) are listed in Appendix I. The same results are illustrated in Figure 8.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 *output* and *condition* boxes. These additions complete the derivation of the ASM chart.

Appendix II gives the results of applying the **e_contunit**'s action outputs to the following sequential and combinational behaviours of **e_gcd_path**:

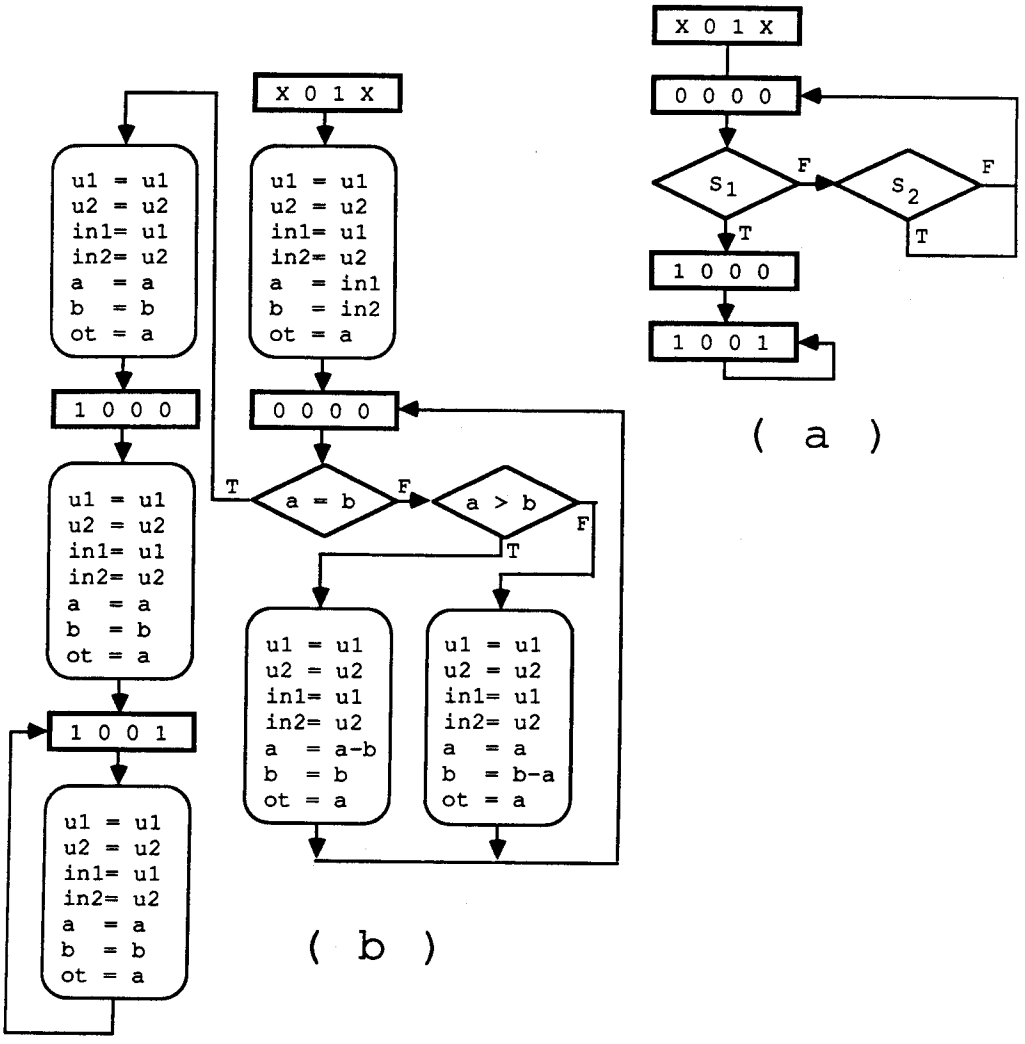

Figure 8. The ASM charts corresponding to the GCD design of Figure 7. (a) The flow-control-part, depicting the behaviour of the control-unit; strings in each state box represent the values in unit-delays *p*, *q*, *r* and *f* at that state, where 'X' stands for an unknown state. (b) The combined behaviour of the control-unit and the data-path.

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

$$\textbf{e\_gcd\_path}\,(a, b, in_1, in_2, u_1, u_2, ot) = \lambda(j, k, La, Lb).\,(\\ (a = b), (a > b)) \tag{28}$$

The ASM chart corresponding to the **e-gcd** module is shown in Figure 8.b.

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 and expected to be true at the start and 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,

$$P(i)\{R, Q\}P(j) \tag{28}$$

holds. In (28), 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 (28) is due to Hoare [13], 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 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 (28) 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*. Obviously, 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 9 shows a version of the ASM chart given in Figure 8.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.

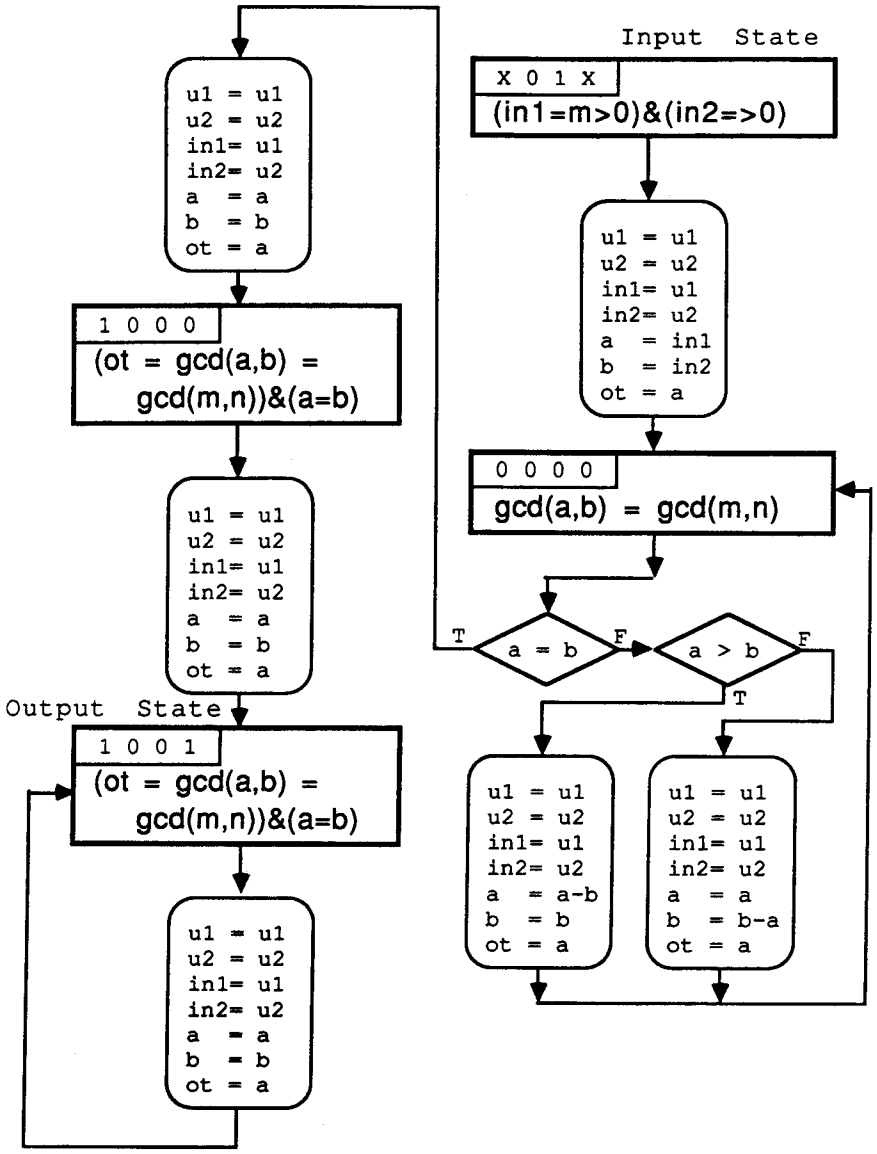

Figure 9. The GCD module's ASM chart, with the correctness propositions for each state boxes. Each proposition can be derived from the preceding proposition(s); the input proposition is assumed *true*.

5. SUMMARY

In this paper we integrated established techniques with several novel methods to form the main components of a hardware verification methodology; these consist of the "requirement specification", "design definition", and "reasoning" phases of a register-transfer design paradigm.

For the design definition, we proposed a friendly register-transfer design environment based on a small set of primitives, and a signal-typing scheme that controls the composition of legal designs. As well as being close to the designer's normal design experience, the primitives are amenable to a mathematical treatment. We proposed a complete functional model for specifying the primitives and the behaviour of their compositions. Later, we used the formalism to derive ASM specifications of candidate designs.

We then introduced automatic translations between the design and the implementation, and the design and the proof environment, and argued that performing the design at a level lower than that of the proof environment leads to improved confidence in the final implementation.

We showed that in order to make assertions about the behaviour of a candidate design at the interfaces, we had to extend the design to include its input and output strings. Finally, the ASM charts were derived from the extended designs, and 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 is 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.

ACKNOWLEDGEMENT

We gratefully acknowledge the University of Waterloo funding and computing facilities used to carry out the work reported here.

6. References

1. Gordon, Mike, Hardware Verification by Formal Proof, Technical Report No. 74, Computer Laboratory, University of Cambridge, Cambridge, England (August 1985).

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

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

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

5. Gordon M., 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. Mavaddat, F., A Functional Model of Register-Transfer Design, CS-88-16, Dept. of Computer Science, University of Waterloo, Waterloo, Ontario (April 1988).

7. Landin, P. J., The Mechanical Evaluation of Expressions, *Computer Journal* **6**(4) pp. 308-320 (Jan. 1984).

8. 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).

9. Mavaddat, F., Designing and Modeling VLSI Systems at Register Transfer Level, *to appear in International Journal of Computer Aided VLSI Design* **1**(2) pp. 41-1 (June 1989).

10. Carver Mead and Lynn Conway, *Introduction to VLSI Systems,* Addison Wesley, Reading, Mass. (1980).

11. Cardelli, Luca and Wegner, Peter, On Understanding Types, Data Abstraction, and Polymorphism, *Computing Surveys* **17**(4) pp. 471-522 (December 1985).

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

13. Hoare, C. A. R., An Axiomatic Basis for Computer Programming, *Communications of the ACM* **12**(10) pp. 576-583 (October 1969).

APPENDIX I

State-Table for the Flow-Control Part of the GCD chart

Present-State				Status-inputs		Next-State				Action-outputs			
p	q	r	f	s1	s2	p	q	r	f	j	k	La	Lb
x	0	1	x	0	0	0	0	0	0	1	0	1	1
x	0	1	x	0	1	0	0	0	0	1	0	1	1
x	0	1	x	1	0	0	0	0	0	1	0	1	1
x	0	1	x	1	1	0	0	0	0	1	0	1	1
0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	1	0	0	0	0	0	1	1	0
0	0	0	0	1	0	1	0	0	0	0	0	0	0
0	0	0	0	1	1	1	0	0	0	0	0	0	0
1	0	0	0	0	0	1	0	0	1	0	0	0	0
1	0	0	0	0	1	1	0	0	1	0	0	0	0
1	0	0	0	1	0	1	0	0	1	0	0	0	0
1	0	0	0	1	1	1	0	0	1	0	0	0	0
1	0	0	1	0	0	1	0	0	1	0	0	0	0
1	0	0	1	0	1	1	0	0	1	0	0	0	0
1	0	0	1	1	0	1	0	0	1	0	0	0	0
1	0	0	1	1	1	1	0	0	1	0	0	0	0

Next-state and action-outputs are obtained by substituting the corresponding present-state and status-input values in the behaviour expressions of **e_contunit**, i.e. (25) and (26), respectively. Only states reachable from the initial state of 'x 0 1 x' are listed. An 'x' entry indicates an unknown unit-delay state. With reference to the **e_gcd_path**, $(s\,1 = 1) \wedge (s\,2 = 1)$ is not possible.

APPENDIX II

Data-Path Assignments Table

0000	0100	1000	1100
u1 = u1	u1 = u1	u1 = u1	u1 = u1
u2 = u2	u2 = u2	u2 = u2	u2 = u2
in1 = u1	in1 = u1	in1 = u1	in1 = u1
in2 = u2	in2 = u2	in2 = u2	in2 = u2
a = a	a = a	a = a	a = a
b = b	b = b	b = b	b = b
ot = a	ot = a	ot = a	ot = a
0001	**0101**	**1001**	**1101**
u1 = u1	u1 = u1	u1 = u1	u1 = u1
u2 = u2	u2 = u2	u2 = u2	u2 = u2
in1 = u1	in1 = u1	in1 = u1	in1 = u1
in2 = u2	in2 = u2	in2 = u2	in2 = u2
a = a	a = a	a = a	a = a
b = b - a	b = a - b	b = in2	b = in2
ot = a	ot = a	ot = a	ot = a
0010	**0110**	**1010**	**1110**
u1 = u1	u1 = u1	u1 = u1	u1 = u1
u2 = u2	u2 = u2	u2 = u2	u2 = u2
in1 = u1	in1 = u1	in1 = u1	in1 = u1
in2 = u2	in2 = u2	in2 = u2	in2 = u2
a = b - a	a = a - b	a = in1	a = in1
b = b	b = b	b = b	b = b
ot = a	ot = a	ot = a	ot = a
0011	**0111**	**1011**	**1111**
u1 = u1	u1 = u1	u1 = u1	u1 = u1
u2 = u2	u2 = u2	u2 = u2	u2 = u2
in1 = u1	in1 = u1	in1 = u1	in1 = u1
in2 = u2	in2 = u2	in2 = u2	in2 = u2
a = b - a	a = a - b	a = in1	a = in1
b = b - a	b = a - b	b = in2	b = in2
ot = a	ot = a	ot = a	ot = a

Each block of entries represents a unique action-input and the corresponding set of register-transfer assignments. The first entry of each block represents, from left to right, the action-input values corresponding to the *j*, *k*, *La*, and *Lb* ports of the data-path respectively.