Technical Report No. 485

A Tabular Language for System Design

by

Steven D. Johnson

June 1997

To appear in the proceedings of the Fourth NASA LaRC Formal Methods Workshop (Lfm'97), *Hampton, Virginia, September 10–12, 1997*

Computer Science Department
Indiana University
Bloomington, Indiana 47405-4101

A Tabular Language for System Design

Steven D. Johnson*

sjohnson@cs.indiana.edu
Computer Science Department
Indiana University
Bloomington Indiana 47405-4101

Abstract

A tabular language for describing synchronous behaviors is developed as a visual representation for formalized design derivation. A sketch of *behavior table* syntax and semantics is given. An example illustrates the kinds of formal manipulations investigated by the research. Evidence is accumulating that tables are perspicuous for specification, design, and verification, but graphical support is essential to their effective use.

1 Introduction and background

The tabular specification language described in this article emerged as a visual representation for interactive system design. We started using tables in a casual way, generating them from the underlying expressions of a formal system for *design derivation* [13]. *Behavior tables* emerged as a bridging notation between control oriented and architecture oriented modes of description.

With better graphical support, we think tables such as these can assume a prominent role in system specification, verification, and synthesis. In our case studies of design derivation, we began using tables to visualize formal transformations on design expressions. Over time, notational features evolved that we have not found in other hardware description languages. Even though we intimately understand the underlying formalism, we believe that the tables are more expressive than the underlying modeling expressions they represent because they offer additional visual structure.

This realization prompted us to consider our tables more seriously as a formal design notation, and we began exploring features that are useful in system design applications [28, 25]. We comtemplated direct realizations in VLSI [20]. Other encouraging influences have been the emerging tools and techniques for using tables in requirements specification, verification, and synthesis. These are reviewed in the next section.

In Sections 3 through 5 we present a syntax and representative semantics for the tables we use. We think of behavior tables as denoting persistent, communicating processes rather than procedures or functions. The substance of the difference is that behavior tables cannot themselves be entries in other behavior tables. Instead, they are composed by connecting their I/O ports. Thus, behavior tables represent typed, synchronous transition systems, which we believe are closer to the intended high-level synthesis models than the "synthesizable subsets" of VHDL and Verilog now in use.

Section 6 illustrates the kind of manipulations we perform in design derivation. The example was constructed manually, but a corresponding derivation was performed using an existing transformation tool. Section 7 reviews additional syntactic features contemplated and topics of further research, including

*This research is supported, in part, by the National Science Foundation under Grants MIP-9208745 and MIP96-10358.

manipulations that we have mathematically formalized but not automated. We believe that the most urgent task of this line research is to develop graphics interface facilities.

An apology about terminology. The term "behavior table" arose spontaneously in our laboratory. In Section 3 we adopt "decision table" from [9] and "action table" from [18] for fragments of our forms. But these fragments are not identical to the previously published objects. Furthermore, there are other objects in the literature with similar names, including "behavior table," that have different, possibly incompatible forms and interpretations. We hope this terminology will stabilize in the future.

2 Related work

The work on decision tables by Hoover, Chen, and others [9, 8] inspired us to think more seriously about the behavior tables developed in our case studies of design derivation. Their *Tablewise* specification tool was developed for avionics software development, but clearly applies to reactive systems in general. In addition to a graphical front end, there are functions for verifying exclusivity and completeness of decision tables and for performing structural analyses to aid in obtaining these properties. Future topics mentioned in [9] include connections to state-machine and statechart based specification. This connection is the focus of our interest.

The *Software Cost Reduction* system of Naval Research Laboratories is also a requirements specification tool set with graphics support and aids for analysis and verification [7]. A formalization in PVS by a group at SRI is based on SCR* constructs and also contains an extensive review of tabular specification notations [19]. Their treatment is a shallow embedding, supported by tabular syntax contained in the PVS surface language. One immediate benefit of this approach is exploitation of the PVS type system, in particular, its management of *type correctness conditions*. Our experience integrating design derivation with PVS verification suggests a somewhat deeper embedding will needed to support reasoning about transformations. One reason for this is that the underlying semantic domain of *streams* is not well founded [14].

Leveson's *Requirements State Machine Language* [16] is based on Harel's state charts [6], but uses *and-or tables* to specify hyper-edges. She echos Hoover's observation that decision tables are readily accepted and used by practicing engineers.

Li and Gupta introduce *timed decision tables* as an HDL [18, 17]. Their results on optimizations exploiting don't care entries are directly applicable to the forms we use in our work. Their work is also evidence of the utility of a tabular specification language for CAD tool development. Behavior tables have been proposed as an interchange format by Gajski, Dutt, et. al. [5, 4]. The intermediate synthesis language BLIV-MV contains a very rich syntax for the tabular specification of multi-valued boolean functions [15]. We find it very encouraging that research in high-level synthesis and formal methods finds common ground in these tabular representations; it reflects new opportunities for synergy between communities that frequently encounter problems with each others' notations.

3 Syntax of behavior tables

Behavior tables are arrays of *terms* over an amalgamated abstract type giving ground syntax for constants and operations and equational laws for reasoning about them. Our examples will involve commonly understood types, but a type system is intended to support conceptual hierarchies, parameterization, and other structuring capabilities. A useful tool must have built-in reasoning for concrete types, but must also have facilities for reasoning about and between user specified type complexes.

The notion of term evaluation used here is standard. The value of a term, t, is written $\sigma[\![t]\!]$, where σ is an *assignment* or association of values to variables. A generic *don't-care* constant is written as '$\natural$'.

A finite extension of propositional logic is assumed—Hoover calls it *finite logic* [8]. Arbitrary collections of enumerated values, or *tokens*, can be formed. These finite sets come with a polymorphic selection operation. A behavior table can be thought

of as a parallel composition of selection expressions.

Behavior tables are closed expressions whose terms contain variables from three disjoint sets: I (inputs), S (data state), and C (combinational signals). Fix these sets for the remainder of this section. We will write ISC for $I \cup S \cup C$ and SC for $S \cup C$. We use the term 'register' for an element of S, but this is a euphemism that should be interpreted very abstractly. There is no assumption that these variables denote finite values, nor are tables intended only for register-transfer specification. The form of a behavior table is

Name: *Inputs* → *Outputs*	
Conditions	*Registers and Signals*
⋮	⋮
Guard	*Computation Step*
⋮	⋮

Inputs is a list of input variables and *Outputs* is a set of terms over ISC; for simplicity, assume $O \subseteq ISC$. *Conditions* is a set P of predicates over ISC, that is, terms ranging over finite types, such as truth values, token sets, etc. A *guard* is a set of constants indexed by the condition set P: $g = \{c_p\}_{p \in P}$. We say g *holds* for an assignment σ to ISC when, for each $p \in P$, either $c_p = \natural$ or $\sigma[\![p]\!] = c_p$.

A *decision table* $\mathbf{D} = [P, G]$, consists of a condition set and a list of guards. Following [9], we say a decision table is *functional* when G describes a proper partitioning of the possible assignments to ISC. In other words, the guards are "exclusive and exhaustive." A *computation step* or *action* is a set of terms, one for each register and signal: $a = \{t_v\}_{v \in SC}$. An *action table* is an indexed set of actions.

A *behavior table for* $I \to O$ consists of a decision table, $\mathbf{D}$, with guards $G = \{g_1, \ldots g_n\}$, and an action table indexed by G, $\mathbf{A} = \{t_{v,k} \mid v \in SC \text{ and } g_k \in G\}$.

4 Synchronous semantics

A behavior table $[\mathbf{D}, \mathbf{A}]$ for $O \subseteq ISC$ denotes a relation between infinite input and output sequences. We call these sequences *streams* because in prior work we obtain a semantics by interpreting a table as a (co)recursive system of stream-defining equations [13]. More directly, suppose we are given a set of initial values for the registers, $\{x_s\}_{s \in S}$ and a stream for each input variable in I. Construct a sequence of assignments, $\langle \sigma_0, \sigma_1 \ldots \rangle$ for ISC as follows:

(a) $\sigma_n(i)$ is given for all $i \in I$ and all n.

(b) For each $s \in S$, $\sigma_0(s) = x_s$.

(c) $\sigma_{n+1}(s) = \sigma_n[\![t_{s,k}]\!]$ if guard g_k holds for σ_n.

(d) For each $c \in C$, $\sigma_n(c) = \sigma_n[\![t_{c,k}]\!]$ if guard g_k holds for σ_n.

The stream associated with each $o \in O$ is $\langle \sigma_0(o), \sigma_1(o), \ldots \rangle$. This semantic relation is well defined if there are no circular dependencies among the combinational actions $\{t_{c,k} \mid c \in C,\ g_k \in G\}$. The relation is a function (i.e. deterministic) if decision table $\mathbf{D}$ is functional.

This semantics is at odds with both TDTs and Tablewise (Section 2), but the differences are reconcilable, and are by no means special to tabular notations. We think of behavior tables as denoting persistent, communicating processes rather than procedures to be invoked.

In other words, behavior tables cannot themselves be entries in other behavior tables, but instead are composed by interconnecting their I/O ports. Compositions give rise to hierarchical network descriptions in which the "leaves" are tables. This is closer to the intended high-level synthesis models than the "synthesizable HDL subsets" now in use. For example, Borrione, et.al., have recently proposed *hierarchical finite state machines* (HFSMs) as a common basis for HDL interoperability [1]. The semantic relationship between HFSMs and behavior tables is very close.

Composition is specified by giving a connection map that is faithful to each component's arity. Valid maps must preserve I/O directionality, excluding both combinational cycles and output conflicts. In our function-oriented modeling methodology, such compositions are expressed as recursive systems of

equations,

$$\lambda(U_1,\ldots,U_n).(V_1,\ldots,V_m)\ \textit{where}$$

$$\begin{aligned}(X_{11},\ldots,X_{1q_1}) &= \mathcal{T}_1(W_{11},\ldots,W_{1\ell_1})\\ &\vdots\\ (X_{p1},\ldots,X_{pq_p}) &= \mathcal{T}_p(W_{p1},\ldots,W_{p\ell_p})\end{aligned}$$

in which the defined variables X_{ij} are all distinct, each $\mathcal{T}_k$ is the name of a behavior table or other composition, and the outputs V_k and internal connections W_{ij} are all simple variables coming from the set $\{U_i\} \cup \{X_{jk}\}$.

Provided they are well formed, deterministic systems are readily animated in modeling languages that allow recursive stream networks to be expressed [10]. As long as each register has an initial value, the streams are constructed head-first as a fixed-point computation. Translation to event-based simulation languages is also relatively straightforward for systems expressed over concrete types.

A synchronous semantics is simple and suited to the clocked implementation models most high-level synthesizers use. In fact, behavior tables will acquire a range of semantics, depending on their applications, just as HDLs and programming languages do. Even with a variety of interpretations, their inherent structure helps reduce the mathematical bookkeeping that often obscures semantic definitions.

5 Examples of behavior tables

The behavior table shown in Figure 1 describes a process that computes the Fibonacci function: The inputs are control signal `go` and data input `in`; the outputs are control signal `done*` and the data signal `v`. The '`*`' is a notational convention for distinguishing combinational signals from state-holding registers. Three other representations in Figure 1 depict various aspects of the design. The labeled transition diagram is keyed to the table's rows; its labels consist of a condition under which the transition is taken, the outputs associated with the transition, and an update to the data state; the same information as a row of the table. The control automaton is represented in the table by the `now` register. Throughout this paper, we reserve the name `now` for this purpose. A timing diagram shows the interface abstraction. The textual expression of the algorithm at the lower left of Figure 1 describes is the well-known iterative computation of $\mathit{fib}(n)$, where

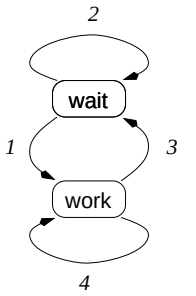
$$\begin{aligned}\mathit{fib}(0) &= 0\\ \mathit{fib}(1) &= 1\\ \mathit{fib}(k+2) &= \mathit{fib}(k)+\mathit{fib}(k+1)\end{aligned}$$

The behavior table in Figure 2 describes the garbage collector of a list processing computer [2]. It is representative of the tables we work with in our case studies. Its level of specification is more abstract, with two of the registers of type *memory*. An implementation of this table was realized in about 5,000 ACTEL FPGA cells, of which 1,500 4-input MUX elements compute the behavior. A behavior table for the same computer's CPU is about twice as big, when expressed at a level where garbage collection is an abstract operation. A table closer the the register transfer level, as in Figure 2, would be more than ten times larger, but even at that scale the tables are useful in our derivation methodology—and would be even more useful with better display automation.

As these examples may illustrate, behavior tables are not the best representation for understanding the specification of an algorithm. However, they seem (in our experience) to serve well as a bridging notation for simultaneously contemplating control and architecture. Furthermore, studies (e.g. [16]) suggest that complicated control functions are clearer when presented as decision tables. In hardware design, the intuitive sense of control flow is quickly overwhelmed when processes are composed.

6 Table manipulations

Let us explore some basic transformations, starting with the table in Figure 1. As in any derivation, the order of presentation is not necessarily the order in which the transformations were conceived. In practice, backtracking is involved as the architectural goals develop and concrete representations are intro-



(go, in) → (done*, v)							
now	go	u=0	now	done*	u	v	w
wait	*true*	↯	work	*false*	in	0	1
"	*false*	↯	wait	*true*	↯	↯	↯
work	↯	*true*	wait	*true*	↯	v	↯
"	↯	*false*	work	*false*	u-1	w	v+w

await *go*;
  u, v, w := *input*$(in), 0, 1$;
while $(u \neq 0)$ **do**
  $\{v = fib(in_0 - u) \wedge w = fib(in_0 - u + 1)\}$
  $u, v, w := u - 1, w, v + w$;
output(v);
assert *done**

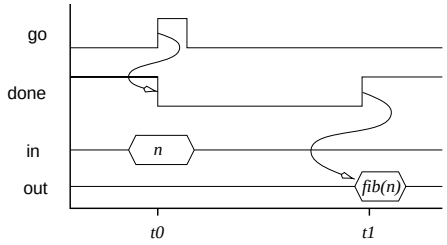


Figure 1: A behavior table and related diagrams

duced. The final derivation is just a residual proof of the design.

This example was carried out and formatted by hand, but Tuna was able to mimic the entire derivation [12] using the DRS mechanized transformation system [3] (Appendix A), which operates on recursive systems of functional expressions. That exercise exposed one significant error in the manual derivation.

The `now` and `done*` columns suggest an assignment of concrete values `0` to `work` and `1` to `wait`. To reduce clutter, let us also assign `1` to `true` and `0` to `false`.

(go, in) → (done*, v)							
now	go	u=0	now	done*	u	v	w
1	1	↯	done*	¬go	in	0	1
"	0	↯	"	¬go	↯	↯	↯
0	↯	1	"	u=0	↯	v	↯
"	↯	0	"	u=0	u-1	w	v+w

With these changes, the first and second rows have become identical up to don't-cares, so we can merge them to obtain

(go, in) → (done*, v)							
now	go	u=0	now	done*	u	v	w
1	↯	↯	done*	¬go	in	0	1
0	↯	1	"	u=0	↯	v	↯
"	↯	0	"	u=0	u-1	w	v+w

The predicate `go` has become irrelevant will be removed. We note in passing that the last two rows could be merged by replacing the term for `v` with `select(u=0,v,w)`. Behavior tables seem especially useful for this kind of interplay between control and computation—all the more so with the provisions for *indirection* discussed in Section 7.

Next is a scheduling transformation on the third row that puts the arithmetic terms `u-1` and `v+w` into different computation steps. The goal is to assign these operations to a single arithmetic component.

go, in → done*, v						
now	u=0	now	done*	u	v	w
1	↯	done*	¬go	in	0	1
0	1	"	u=0	↯	v	↯
0	0	2	false	u-1	v	w
2	↯	done*	u=0	u	w	v+w

	NOW	RQ	(= U A)	(pointer? H)	(bvec-h? H)	(eq? 'forward (tag d))	(tag H)	(= C 0)	(= C 1)	NOW	OLD:*mem*	NEW:*mem*	H:*cont*	D:*cont*	C:*addr*	U:*cont*	A:*addr*	AK
1	idle	1	↯	↯	↯	↯	↯	↯	↯	driver	OLD	NEW	H	D	C	(rd OLD H)	0	0
2	"	0	↯	↯	↯	↯	↯	↯	↯	idle	OLD	NEW	H	↯	↯	↯	↯	1
3	driver	↯	1	↯	↯	↯	↯	↯	↯	idle	NEW	OLD	0	D	C	U	A	1
4	"	↯	0	↯	↯	↯	↯	↯	↯	nextobj	OLD	NEW	(rd NEW U)	D	C	U	A	0
5	nextobj	↯	↯	1	↯	↯	↯	↯	↯	objtype	OLD	(wt NEW U (cell H A))	H	(rd OLD H)	C	U	A	0
6	"	↯	↯	0	1	↯	↯	↯	↯	driver	OLD	NEW	H	(rd OLD H)	C	(+ U (btow-u (pr-pt H)) (cin 1))	A	0
7	"	↯	↯	0	0	↯	↯	↯	↯	driver	OLD	NEW	H	(rd OLD H)	C	(+ U (const 0) (cin 1))	A	0
8	objtype	↯	↯	↯	↯	1	↯	↯	↯	driver	OLD	(wt NEW U (cell H d))	H	D	C	(+ U (const 0) (cin 1))	A	0
9	"	↯	↯	↯	↯	0	fixed	↯	↯	copy	(wt OLD H (cell forward A))	NEW	(cell H (add1-ptr (pr-pt H)))	D	(add1-2 (fixed-size H))	U	A	0
10	"	↯	↯	↯	↯	0	vec	↯	↯	vec	(wt OLD H (cell forward A))	NEW	(cell H (add1-ptr (pr-pt H)))	D	(btow-c (pr-pt d))	U	A	0
11	"	↯	↯	↯	↯	0	bvec	↯	↯	vec	(wt OLD H (cell forward A))	NEW	(cell H (add1-ptr (pr-pt H)))	D	(btow-c (pr-pt d))	U	A	0
12	vec	↯	↯	↯	↯	↯	↯	1	↯	driver	OLD	(wt NEW A d)	(cell H (add1-ptr (pr-pt H)))	D	C	(+ U (const 0) (cin 1))	(add1-a A)	0
13	"	↯	↯	↯	↯	↯	↯	1	↯	copy	OLD	(wt NEW A d)	(cell H (add1-ptr (pr-pt H)))	(rd OLD H)	C	U	(add1-a A)	0
14	copy	↯	↯	↯	↯	↯	↯	↯	1	driver	OLD	(wt NEW A d)	H	D	C	(+ U (const 0) (cin 1))	(add1-a A)	0
15	"	↯	↯	↯	↯	↯	↯	↯	0	copy	OLD	(wt NEW A d)	(cell H (add1-ptr (pr-pt H)))	(rd OLD H)	(sub1 C)	U	(add1-a A)	0

Figure 2. Behavior Table for a Garbage Collector

The newly created control token, '2', induces a type mismatch with boolean `done*` in the `now` column. This is a problem to be resolved by underlying type inference system. In addition to implicit coercions, this transformation's validity depends *both* on the fact that the sequence of two steps preserves the original computation *and* the fact that the surrounding synchronization protocol is preserved. Verifying the latter of these conditions is not automatic, in general. In this case, we are relying on the interface specification of Figure 1, which says that the result is ready only when `done*` is asserted.

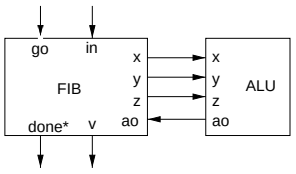
The next table is a simple example of *system factorization*, a decomposition technique that is central to the derivation formalism. As desired, terms `u-1` and `v+w` are allocated to a single combinational arithmetic component, called `ALU`.

FIB: (go, in, ao) → (done*, v, x*, y*, z*)

now	u=0	now	done*	u	v	w	x*	y*	z*
1	↯	done*	¬go	in	0	1	↯	↯	↯
0	1	"	u=0	↯	v	↯	↯	↯	↯
0	0	2	false	ao	v	w	sub	u	1
2	↯	done*	u=0	u	w	ao	add	v	w

ALU:(x,y,z) → ao*

x	ao*
add	x+y
sub	x-y



A system factorization encapsulates a set of subject terms in a new table and generates residual interface signals [11]. Here, the interface signal `x*` generates instruction tokens, `sub` and `add`, telling `ALU` which operation to perform. The transformation tool keeps track of the connectivity. In particular, factorizations preserve well-formedness even when one of the factors is entriely combinational, as is `ALU` in this case.

To finish the example, we make some assignments to the don't-care entries whose ultimate effect is to isolate control. As a second example of system factorization, we decompose into a control process generating an encoded command signal, `cmd`, to the data path `DP`, as shown in Figure 3.

7 Directions

We are encouraged by the number of recent papers centering on tabular specification languages. It is usually reported that such tables are a good "engineering notation," and they seem also to be relatively easy to represent formally. If this consensus grows, the most urgent task may be the mundane one of building tools to manipulate table syntax. It is our hope that such graphics tools will be general enough to accommodate the range of applications tables are finding in the design community.

At this stage, we are investigating a number of additions to and variations of behavior table syntax.

7.1 Assertions

Both Tablewise and TDTs (Section 2 have provisions for *assertions* that are not yet in our behavior tables, but which should be included in any graphics support. Tablewise incorporates type-declarative fields that we would defer to a background type system in our applications. TDTs contain time parameters used in optimization. In Tablewise the primary intent seems to be the verification of invariant properties, but assertions could also be used to state constraints, measures, and, for that matter, computational actions.

The algorithmic specification in Figure 1 contains a loop invariant that might be attached to row 4 of the corresponding behavior table. It is interesting to contemplate how subsequent manipulations, especially decompositions, might affect this assertion. Since system design verification often involves liveness, safety, and other eventualities, assertions would likely take the form of temporal logic predicates on the current state (i.e., row).

7.2 Decompositions

Our notion of system factorization, involving both data abstraction and interface specification, has yet to be fully reflected in our behavior tables. The underlying ideas are more general than the example shows, having evolved over several years of research.

CTL: (go, u) → (done*, cmd*, x*)					
now	u=0	now	done*	cmd*	x*
1	↯	done*	¬go	0	↯
0	1	"	u=0	1	sub
0	0	2	false	1	sub
2	↯	done*	"	2	add

DP: (cmd, in, ao) → (u, v, y*, z*)					
cmd	u	v	w	y*	z*
0	in	0	1	↯	↯
1	ao	v	w	v	w
2	u	w	ao	u	1

ALU:(x,y,z) → ao*	
x	ao*
add	y+z
sub	y-z

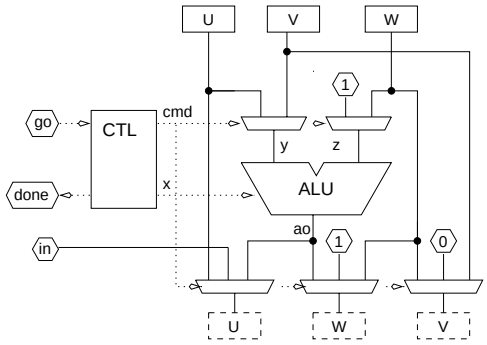


Figure 3: Final decomposition of the example

As an illustration, let us consider a two-phase ALU that takes operands sequentially.

ALU2:(op,in) → (phase, out*)				
phase	op	phase	hold	out*
1	↯	2	in	↯
2	add	1	in	hold + in
2	sub	1	in	hold - in

We wish to use ALU2 to determine a factorization of the table below, a variant of the specification in Figure 1. A *reset* input, r, has been added to illustrate why tables are sometimes better than algorithmic languages at expressing features of global control flow. Row 1 of the table says that whenever r is asserted the FSM moves to state A.

	FIB:(r, go, in) → (d*, v)								
	r	now	go	u=0	now	d*	u	v	w
1	0	↯	↯	↯	A	0	↯	↯	↯
2	1	A	1	↯	B	0	in	0	1
3	"	"	0	↯	A	1	↯	↯	↯
4	"	B	↯	T	A	1	↯	v	↯
5	"	"	↯	F	B	0	u-1	w	v+w

A targeted factorization has to instantiate the protocol ALU2 expects, synchronizing with its phases, and presenting the operands sequentially. First, we serialize the arithmetic as before:

	FIB:(r, go, in) → (d*, v)								
	r	now	go	u=0	now	d*	u	v	w
1	0	↯	↯	↯	A	0	↯	↯	↯
2	1	A	1	↯	B	0	in	0	1
3	"	"	0	↯	A	1	↯	↯	↯
4	"	B	↯	T	A	1	↯	v	↯
5a	"	"	↯	F	C	0	u	w	v+w
5b	"	C	↯	↯	B	0	u-1	v	w

To decompose according to ALU2, add a wait state to get into phase and graft the addition and subtraction paths into the control flow. One possible factorization is shown in Figure 4. We have have investigated several constructive approaches to this family of decompositions [21, 23, 30, 29, 24]. We cannot yet claim a universal construction, but we do have transformations general enough to handle many common interface specifications [22]. Performing simultaneous decompositions—which is necessary for practical application of formal derivations—remains a topic of research.

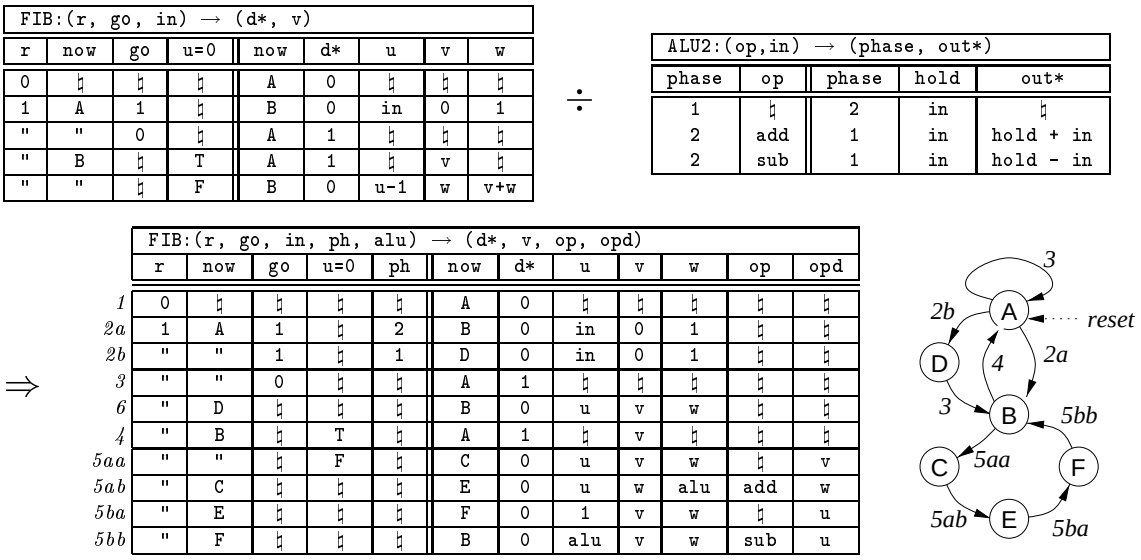


Figure 4: The factorization developed in Section 7.2

7.3 Other syntax

The tabular languages we have seen exhibit a great variety of abbreviation techniques. Typically, these serve to condense decision conditions by specifying sets of values. *BLIF-MV*, for example, allows subrange, subset, and complementation expressions in its table specifications [15].

We have added syntax for *bounded indirection* which often significantly reduces the size of action tables [25] and is novel for hardware description languages. If r is a signal or register, then `#`r denotes a token referring to r. If register s contains such a token, then `@`s denotes the entity to which s refers; that is,

$$\texttt{@}s \;\equiv\; \mathsf{case}\ s\ \mathsf{of}\ \ldots \texttt{\#}r\colon r \ldots$$

In [28] we show how a behavior table describing a bus reduces to one row when indirection is used to specify sources and destinations. In [27] we explore control indirection.

The *behavior FSMs* proposed by Takach, Wolf, and Leeser contain constructs to constrain events to occur within sets of transitions. BFSM are intended to serve as specification models for high-level synthesis. Any implementation of a BFSM refines this constraint by assigning particular transition to each event, subject to the constraints [26]. As the example of Section 6 suggests, one row of an action table can represent a number of transitions at a finer time scale. However, BFSMs are more expressive than behavior tables in the sense that unstructured programs are more expressive than structured ones. This suggests to us that action tables may need provisions to relax their output behaviors.

References

[1] Dominiquie Borrione, Fredrik Vestman, and Hakim Bouamama. An approch to Verilog-VHDL interoperability for synchronous designs. In *Procedings of the IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'97)*. To appear.

[2] Robert G. Burger. The scheme machine. Technical Report 413, Indiana University, Computer Science Department, August 1994. 59 pages.

[3] Derivation Systems, Inc., Carlsbad, California. *DRS: Derivational Reasoning System*, 1.2.1 edition, December 1995. Contact `drs@derivation.com`.

[4] Nikil D. Dutt and Daniel D. Gajski. Exel: A language for interactive behavioral synthesis. In John A. Darringer and Franz J. Rammig, editors, *Computer Hardware Description Languages and their Applications*, pages 3–18, 1989.

[5] D. Gajski, N. Dutt, A. Wu, and S. Lin. *High-level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, 1992.

[6] D. Harel. Statecharts: a visual formalism for complex systems. *The Science of Computer Programming*, 8:231–274, 1987.

[7] Constance Heitmeyer, Alan Bull, Carolyn Gasarch, and Bruce Labaw. SCR*: a toolset for specifying and analyzing requirements. In *Proceedings of the Tenth Annual Conference on Computer Assurance (COMPASS '95)*, pages 109–122, 1995.

[8] D. N. Hoover and Zewei Chen. Tbell: A mathematical tool for analyzing decision tables. Contractor Report 195027, NASA/LRC, Hampton VA 23681-0001, November 1994.

[9] D. N. Hoover, David Guaspari, and Polar Humenn. Applications of formal methods to specification and safety of avionics software. Contractor Report 4723, NASA/LRC, Hampton VA 23681-0001, November 1994.

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

[11] Steven D. Johnson. Manipulating logical organization with system factorizations. In Leeser and Brown, editors, *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, volume 408 of *LNCS*, pages 260–281. Springer, July 1989.

[12] Steven D. Johnson. A tabular language for system design, Appendix A. Technical Report 485, Indiana University Computer Science Department, June 1997.

[13] Steven D. Johnson and Bhaskar Bose. A system for mechanized digital design derivation. In *IFIP and ACM/SIGDA International Workshop on Formal Methods in VLSI Design*, 1991. Available as Indiana University Computer Science Department Technical Report No. 323 (December 1990).

[14] Steven D. Johnson and Paul S. Miner. integrated reasoning support in system design: design derivation and theorem proving. In *IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'97)*, 1997. To appear.

[15] Yuji Kukimoto. BLIF-MV. `http://www-cad.eecs.berkeley.edu/~/vis/`.

[16] Nancy G. Leveson, Mats Per Erik Heimdahl, Holly Hildreth, and Jon Damon Reese. Requirements specifiation for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, September 1994.

[17] Jian Li. Timed decision tables: A behavioral model for embedded system specification and optimization. Technical Report UIUCDCS-R-96-1971, Univeristy of Illinois Department of Computer Science, 1304 West Springfield Ave, Urbana IL 61801, 1996.

[18] Jian Li and Rejash K. Gupta. HDL optimization using timed decision tables. In *33rd ACM/IEEE Design Automation Conference*, 1996.

[19] Sam Owre, John Rushby, and N. Shankar. Integration in PVS: Tables, types, and model checking. In *Tools and Algorithms for the Construction and Analysis of Systems TACAS '97*, Springer *LNCS*. To appear.

[20] K. Rath, I. Celis, and R. M. Wehrmeister. RTBA: A generic bit-sliced bus architecture for

datapath synthesis. Technical Report 321, Department of Computer Science, Indiana University, December 1990.

[21] Kamlesh Rath. *Sequential System Decomposition.* PhD thesis, Computer Science Department, Indiana University, USA, 1995. Technical Report No. 457, 90 pages.

[22] Kamlesh Rath, Bhaskar Bose, and Steven D. Johnson. Derivation of a DRAM memory interface by sequential decomposition. In *Proceedings of the International Conference on Computer Design (ICCD)*, pages 438–441. IEEE, October 1993.

[23] Kamlesh Rath, Venkatesh Choppella, and Steven D. Johnson. Decomposition of sequential behavior using interface specification and complementation. *VLSI Design Journal*, 3(3-4):347–358, 1995.

[24] Kamlesh Rath and Steven D. Johnson. Toward a basis for protocol specification and process decomposition. In D. Agnew, L. Claesen, and R. Camposano, editors, *Proceedings of IFIP Conference on Hardware Description Languages and their Applications*, pages 157–174. Elsevier, April 1993.

[25] Kamlesh Rath, M. Esen Tuna, and Steven D. Johnson. Behavior tables: A basis for system representation and transformational system synthesis. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, pages 736–740. IEEE, November 1993.

[26] Andreś Takach, Wayne Wolf, and Miriam Leeser. An automaton model for scheduling constraints in synchronous machines. *IEEE Transactions on Computers*, 44(1):1–12, January 1995.

[27] M. Esen Tuna, Steven D. Johnson, and Bob Burger. Continuations in hardware-software codesign. In *Proceedings of the International Conference on Computer Design (ICCD)*, pages 264–269. IEEE, October 1994.

[28] M. Esen Tuna, Kamlesh Rath, and Steven D. Johnson. Specification and synthesis of bounded indirection. In *Proceedings of the Fifth Great Lakes Symposium on VLSI*, pages 86–89. IEEE, March 1995.

[29] Zheng Zhu and Steven D. Johnson. Automatic synthesis of sequential synchronizations. In D. Agnew, L. Claesen, and R. Camposano, editors, *Proceedings of IFIP Conference on Hardware Description Languages and their Applications*, pages 285–301. Elsevier, April 1993.

[30] Zheng Zhu and Steven D. Johnson. Capturing synchronization specifications for sequential compositions. In *Proceedings of the 1994 IEEE International Conference on Computer Design (ICCD 94)*, pages 117–121. IEEE, October 1994.

A DDD derivation of the example

This appendix shows a formal derivation corresponding to the example in Section 6. The derviation was done by E. Esen Tuna using the *DRS* system under development by Derivations Systems, Inc. [3].

The DRS derivation involves a sequence of 28 transformation commands

```
(new-design "fib")
```

0 ———

```
(define fib
   (lambda (go in)
      (letrec ([add (lambda (x y) (+ x y))]
               [sub (lambda (x y) (- x y))]
               [zero-p (lambda (x) (eq? x 0))]
               [zero 0]
               [one 1])
         (letrec ([wait
                   (lambda (u v w)
                      (if go
                          (let ([done* #f]) (work in zero one))
                          (let ([done* #t]) (wait ? ? ?))))]
                  [work
                   (lambda (u v w)
                      (if (zero-p u)
                          (let ([done* #t]) (wait ? v ?))
                          (let ([done* #f])
                             (work (sub u 1) w (add v w)))))])
            (wait ? ? ?)))))
```

Introduce a control state to schedule `sub` *and* `add` *in distinct time steps.*

```
(serialize
   '((appl work 2))
   '(define work2 (lambda (u v w) (work u w (add v w)))))
```

1

```
(define fib
   (lambda (go in)
      (letrec ([add (lambda (x y) (+ x y))]
               [sub (lambda (x y) (- x y))]
               [zero-p (lambda (x) (eq? x 0))]
               [zero 0]
               [one 1])
         (letrec ([wait
                   (lambda (u v w)
                      (if go
                          (let ([done* #f]) (work in zero one))
                          (let ([done* #t]) (wait ? ? ?))))]
                  [work
                   (lambda (u v w)
                      (if (zero-p u)
                          (let ([done* #t]) (wait ? v ?))
                          (let ([done* #f]) (work2 (sub u 1) v w))))]
                  [work2
                   (lambda (u v w)
                      (let ([done* #f]) (work u w (add v w))))])
            (wait ? ? ?)))))
```

Translate specification to an explicit-state behavioral specification. The token representing the contol state is called `now`.

```
(behavior->singleloop 'now)
```

2

```
(define fib
   (lambda (go in)
      (letrec ([add (lambda (x y) (+ x y))]
               [sub (lambda (x y) (- x y))]
               [zero-p (lambda (x) (eq? x 0))]
               [zero 0]
               [one 1]
               [wait 'wait]
               [work 'work]
               [work2 'work2])
         (letrec ([fib
                   (lambda (now u v w)
                      (case now
                         [wait
                          (if go
                              (let ([done* #f]) (fib work in zero one))
                              (let ([done* #t]) (fib wait ? ? ?)))]
                         [work
                          (if (zero-p u)
                              (let ([done* #t]) (fib wait ? v ?))
                              (let ([done* #f])
                                 (fib work2 (sub u 1) v w)))]
                         [work2
                          (let ([done* #f]) (fib work u w (add v w)))]))])
            (fib wait ? ? ?)))))
```

This transformation begins the process of incorporating concrete representations for symbolic constants, such as `'wait`, `'work`, `'work2`. `state` is a file name.

```
(change-representation "state)
```

3

```
(define fib
   (lambda (go in)
      (letrec ([add (lambda (x y) (+ x y))]
               [sub (lambda (x y) (- x y))]
               [zero-p (lambda (x) (eq? x 0))]
               [zero 0]
               [one 1]
               [wait #t]
               [work #f]
               [work2 2])
         (letrec ([fib
                   (lambda (now u v w)
                      (case now
                         [wait
                          (if go
                              (let ([done* #f]) (fib work in zero one))
                              (let ([done* #t]) (fib wait ? ? ?)))]
                         [work
                          (if (zero-p u)
                              (let ([done* #t]) (fib wait ? v ?))
                              (let ([done* #f])
                                 (fib work2 (sub u 1) v w)))]
                         [work2
                          (let ([done* #f]) (fib work u w (add v w)))]))])
            (fib wait ? ? ?)))))
```

Replace occurances of `work` *and* `wait` *with their bindings.*

```
(expand-all '((letrec 2)) '(work wait))
```

4

```
(define fib
   (lambda (go in)
      (letrec ([add (lambda (x y) (+ x y))]
               [sub (lambda (x y) (- x y))]
               [zero-p (lambda (x) (eq? x 0))]
               [zero 0]
               [one 1]
               [wait #t]
               [work #f]
               [work2 2])
         (letrec ([fib
                   (lambda (now u v w)
                      (case now
                         [wait
                          (if go
                              (let ([done* #f]) (fib #f in zero one))
                              (let ([done* #t]) (fib #t ? ? ?)))]
                         [work
                          (if (zero-p u)
                              (let ([done* #t]) (fib #t ? v ?))
                              (let ([done* #f])
                                 (fib work2 (sub u 1) v w)))]
                         [work2
                          (let ([done* #f]) (fib #f u w (add v w)))]))])
            (fib #t ? ? ?)))))
```

The next ten commands are local rewrites whose effect is to complete the tactical instantiation of undetermined constant symbols in the example.

```
(substitute '((const #f 3)) 'done*)
(substitute '((const #t 3)) 'done*)
(substitute '((const #t 4)) 'done*)
(substitute '((const #f 5)) 'done*)
(substitute '((const #t 4)) 'wait)
(instantiate-boolean '((const #f 2)) 'go)
(instantiate-boolean '((const #t 2)) 'go)
(reduce-if '((if 1)))
(instantiate-boolean '((const #t 2)) '(zero-p u))
(instantiate-boolean '((const #f 2)) '(zero-p u))
```

5–14

```
(define fib
   (lambda (go in)
      (letrec ([add (lambda (x y) (+ x y))]
               [sub (lambda (x y) (- x y))]
               [zero-p (lambda (x) (eq? x 0))]
               [zero 0]
               [one 1]
               [wait #t]
               [work #f]
               [work2 2])
         (letrec ([fib
                   (lambda (now u v w)
                      (case now
                         [wait
                          (let ([done* (not go)]) (fib done* in zero one))]
                         [work
                          (if (zero-p u)
                              (let ([done* (zero-p u)]) (fib done* ? v ?))
                              (let ([done* (zero-p u)])
                                 (fib work2 (sub u 1) v w)))]
                         [work2
                          (let ([done* #f]) (fib done* u w (add v w)))]))])
            (fib wait ? ? ?)))))
```

Transform the control oriented expression to a structure oriented expression, interepreted as a network of stream equations

```
(singleloop->structure 'select 'status)
```

15

```
(define fib
   (lambda (go in)
      (stream-letrec
         ((add (lambda (x y) (+ x y)))
          (sub (lambda (x y) (- x y)))
          (zero-p (lambda (x) (eq? x 0)))
          (zero 0)
          (one 1)
          (wait #t)
          (work #f)
          (work2 2))
         (select-letrec
            ((select
                (lambda ((p0 p1) v0 v1 v2 v3)
                   (case p0 [wait v0] [work (if p1 v1 v2)] [work2 v3])))))
            (system-letrec
               ((status (list now (zero-p u)))
                (now (! wait (select status done* done* work2 done*)))
                (u (! ? (select status in ? (sub u 1) u)))
                (v (! ? (select status zero v v w)))
                (w (! ? (select status one ? w (add v w))))
                (done* (select status (not go) (zero-p u) (zero-p u) #f)))
               (list now u v w done*))))))
```

A system factorization encapsulates `add` *and* `sub` *as a subsystem called* `alu`.

```
(factor-ops
   '((select-letrec 1))
   'alu
   '((ao* ins inp (((appl add 1)) ((appl sub 1))))))
```

16

```
(define fib
   (lambda (go in)
      (stream-letrec
         ((add (lambda (x y) (+ x y)))
          (sub (lambda (x y) (- x y)))
          (zero-p (lambda (x) (eq? x 0)))
          (zero 0)
          (one 1)
          (wait #t)
          (work #f)
          (work2 2)
          (ao*-? 'ao*-?)
          (alu_ins-nop 'alu_ins-nop)
          (alu_ins-sub 'alu_ins-sub)
          (alu_ins-add 'alu_ins-add)
          (alu_inp_a-? 'alu_inp_a-?)
          (alu_inp_b-? 'alu_inp_b-?))
         (letrec ([alu
                   (lambda (ins inp_a inp_b)
                      (stream-letrec
                         ((construct0
                             (lambda (ins inp_a inp_b)
                                (case ins
                                   [alu_ins-nop ao*-?]
                                   [alu_ins-sub (sub inp_a inp_b)]
                                   [alu_ins-add (add inp_a inp_b)])))))
                         (system-letrec
                            ((x0 (construct0 ins inp_a inp_b)))
                            x0)))])
           (select-letrec
              ((select
                  (lambda ((p0 p1) v0 v1 v2 v3)
                     (case p0
                        [wait v0]
                        [work (if p1 v1 v2)]
                        [work2 v3])))))
              (system-letrec
                 ((status (list now (zero-p u)))
```

```
 (now (! wait (select status done* done* work2 done*)))
 (u (! ? (select status in ? ao* u)))
 (v (! ? (select status zero v v w)))
 (w (! ? (select status one ? w ao*)))
 (done*
    (select status (not go) (zero-p u) (zero-p u) #f))
 (ao* (alu alu_ins alu_inp_a alu_inp_b))
 (alu_ins
    (select
       status
       alu_ins-nop
       alu_ins-nop
       alu_ins-sub
       alu_ins-add))
 (alu_inp_a (select status alu_inp_a-? alu_inp_a-? u v))
 (alu_inp_b (select status alu_inp_b-? alu_inp_b-? 1 w)))
(list now u v w done*)))))))
```

A partitioning transformation reorganizes the structural expression into controller `ctl`*, data path* `dp`*, and function unit* `alu`*, as depicted at the end of the example*

```
(partition
   '((select-letrec 1))
   '((abs_alu (ao*))
     (ctl (now done* alu_ins))
     (dp (u v w alu_inp_a alu_inp_b))))
```

17

```
(define fib
   (lambda (go in)
      (stream-letrec
         ((add (lambda (x y) (+ x y)))
          (sub (lambda (x y) (- x y)))
          (zero-p (lambda (x) (eq? x 0)))
          (zero 0)
          (one 1)
          (wait #t)
          (work #f)
          (work2 2)
          (ao*-? 'ao*-?)
          (alu_ins-nop 'alu_ins-nop)
          (alu_ins-sub 'alu_ins-sub)
          (alu_ins-add 'alu_ins-add)
          (alu_inp_a-? 'alu_inp_a-?)
          (alu_inp_b-? 'alu_inp_b-?))
         (letrec ([alu
                   (lambda (ins inp_a inp_b)
                      (stream-letrec
                         ((construct0
                             (lambda (ins inp_a inp_b)
                                (case ins
                                   [alu_ins-nop ao*-?]
                                   [alu_ins-sub (sub inp_a inp_b)]
                                   [alu_ins-add (add inp_a inp_b)])))))
                         (system-letrec
                            ((x0 (construct0 ins inp_a inp_b)))
                            x0)))])
           (letrec ([abs_alu
                     (lambda (now u alu_ins alu_inp_a alu_inp_b)
                        (select-letrec
                           ((select
                               (lambda ((p0 p1) v0 v1 v2 v3)
                                  (case p0
                                     [wait v0]
```

```
                [work (if p1 v1 v2)]
                [work2 v3])))
       (system-letrec
          ((status (list now (zero-p u)))
           (ao* (alu alu_ins alu_inp_a alu_inp_b)))
          ao*)))]
[ctl
 (lambda (u)
    (select-letrec
       ((select
           (lambda ((p0 p1) v0 v1 v2 v3)
              (case p0
                 [wait v0]
                 [work (if p1 v1 v2)]
                 [work2 v3])))
       (system-letrec
          ((status (list now (zero-p u)))
           (now (! wait
                   (select
                      status
                      done*
                      done*
                      work2
                      done*)))
           (done*
              (select
                 status
                 (not go)
                 (zero-p u)
                 (zero-p u)
                 #f))
           (alu_ins
              (select
                 status
                 alu_ins-nop
                 alu_ins-nop
                 alu_ins-sub
                 alu_ins-add)))
          (list now done* alu_ins))))]
[dp
 (lambda (now ao*)
    (select-letrec
       ((select
           (lambda ((p0 p1) v0 v1 v2 v3)
              (case p0
                 [wait v0]
```

```
                    [work (if p1 v1 v2)]
                    [work2 v3])))))
            (system-letrec
               ((status (list now (zero-p u)))
                (u (! ? (select status in ? ao* u)))
                (v (! ? (select status zero v v w)))
                (w (! ? (select status one ? w ao*)))
                (alu_inp_a
                   (select
                      status
                      alu_inp_a-?
                      alu_inp_a-?
                      u
                      v))
                (alu_inp_b
                   (select
                      status
                      alu_inp_b-?
                      alu_inp_b-?
                      1
                      w)))
               (list u v w alu_inp_a alu_inp_b))))])
(system-letrec
   ((ao* (abs_alu now u alu_ins alu_inp_a alu_inp_b))
    ((now done* alu_ins) (ctl u))
    ((u v w alu_inp_a alu_inp_b) (dp now ao*)))
   (list now u v w done*)))))))
```

The next sequence of transformations eliminates the `abs_alu` *interface to use the* `alu` *combinator more directly in the system.*

```
(inline '((ref ao* 1)))
(reduce-letrec '((system-letrec 2)))
(reduce-letrec '((select-letrec 1)))
(inline '((appl abs_alu 1)))
(reduce-letrec '((letrec 2)))
```

18–22

```
(define fib
   (lambda (go in)
      (stream-letrec
         ((add (lambda (x y) (+ x y)))
          (sub (lambda (x y) (- x y)))
          (zero-p (lambda (x) (eq? x 0)))
          (zero 0)
          (one 1)
          (wait #t)
          (work #f)
          (work2 2)
          (ao*-? 'ao*-?)
          (alu_ins-nop 'alu_ins-nop)
          (alu_ins-sub 'alu_ins-sub)
          (alu_ins-add 'alu_ins-add)
          (alu_inp_a-? 'alu_inp_a-?)
          (alu_inp_b-? 'alu_inp_b-?))
         (letrec ([alu
                   (lambda (ins inp_a inp_b)
                      (stream-letrec
                         ((construct0
                             (lambda (ins inp_a inp_b)
                                (case ins
                                   [alu_ins-nop ao*-?]
                                   [alu_ins-sub (sub inp_a inp_b)]
                                   [alu_ins-add (add inp_a inp_b)])))))
                         (system-letrec
                            ((x0 (construct0 ins inp_a inp_b)))
                            x0)))])
           (letrec ([ctl
                     (lambda (u)
                        (select-letrec
                           ((select
                               (lambda ((p0 p1) v0 v1 v2 v3)
                                  (case p0
                                     [wait v0]
```

```
                [work (if p1 v1 v2)]
                [work2 v3])))))
        (system-letrec
           ((status (list now (zero-p u)))
            (now (! wait
                    (select
                       status
                       done*
                       done*
                       work2
                       done*)))
            (done*
               (select
                  status
                  (not go)
                  (zero-p u)
                  (zero-p u)
                  #f))
            (alu_ins
               (select
                  status
                  alu_ins-nop
                  alu_ins-nop
                  alu_ins-sub
                  alu_ins-add)))
           (list now done* alu_ins))))]
[dp
 (lambda (now ao*)
    (select-letrec
       ((select
           (lambda ((p0 p1) v0 v1 v2 v3)
              (case p0
                 [wait v0]
                 [work (if p1 v1 v2)]
                 [work2 v3])))))
       (system-letrec
          ((status (list now (zero-p u)))
           (u (! ? (select status in ? ao* u)))
           (v (! ? (select status zero v v w)))
           (w (! ? (select status one ? w ao*)))
           (alu_inp_a
              (select
                 status
                 alu_inp_a-?
                 alu_inp_a-?
                 u
```

```
                        v))
                   (alu_inp_b
                      (select
                         status
                         alu_inp_b-?
                         alu_inp_b-?
                         1
                         w)))
                  (list u v w alu_inp_a alu_inp_b))))])
(system-letrec
   ((ao* (alu alu_ins alu_inp_a alu_inp_b))
    ((now done* alu_ins) (ctl u))
    ((u v w alu_inp_a alu_inp_b) (dp now ao*)))
   (list now u v w done*)))))))
```

Instantiate don't cares in order to simplify the dp *subsystem.*

```
(instantiate-? '((ref ? 2)) 'ao*)
(instantiate-? '((ref ? 4)) 'w)
(instantiate-? '((ref alu_inp_a-? 2)) 'u)
(instantiate-? '((ref alu_inp_b-? 2)) '1)
(optimize '((select-letrec 2)))
(expand-status '((select-letrec 2)) 'select)
```

23–28

```
(define fib
   (lambda (go in)
      (stream-letrec
         ((add (lambda (x y) (+ x y)))
          (sub (lambda (x y) (- x y)))
          (zero-p (lambda (x) (eq? x 0)))
          (zero 0)
          (one 1)
          (wait #t)
          (work #f)
          (work2 2)
          (ao*-? 'ao*-?)
          (alu_ins-nop 'alu_ins-nop)
          (alu_ins-sub 'alu_ins-sub)
          (alu_ins-add 'alu_ins-add)
          (alu_inp_a-? 'alu_inp_a-?)
          (alu_inp_b-? 'alu_inp_b-?))
         (letrec ([alu
                   (lambda (ins inp_a inp_b)
                      (stream-letrec
                         ((construct0
                             (lambda (ins inp_a inp_b)
                                (case ins
                                   [alu_ins-nop ao*-?]
                                   [alu_ins-sub (sub inp_a inp_b)]
                                   [alu_ins-add (add inp_a inp_b)])))))
                         (system-letrec
                            ((x0 (construct0 ins inp_a inp_b)))
                            x0)))])
           (letrec ([ctl
                     (lambda (u)
                        (select-letrec
                           ((select
                               (lambda ((p0 p1) v0 v1 v2 v3)
                                  (case p0
```

```
                        [wait v0]
                        [work (if p1 v1 v2)]
                        [work2 v3])))))
               (system-letrec
                  ((status (list now (zero-p u)))
                   (now (! wait
                           (select
                              status
                              done*
                              done*
                              work2
                              done*)))
                   (done*
                      (select
                         status
                         (not go)
                         (zero-p u)
                         (zero-p u)
                         #f))
                   (alu_ins
                      (select
                         status
                         alu_ins-nop
                         alu_ins-nop
                         alu_ins-sub
                         alu_ins-add)))
                  (list now done* alu_ins))))]
         [dp
          (lambda (now ao*)
             (select-letrec
                ((select
                    (lambda (p0 v0 v1 v3)
                       (case p0
                          [wait v0]
                          [work v1]
                          [work2 v3])))
                (system-letrec
                   ((u (! ? (select now in ao* u)))
                    (v (! ? (select now zero v w)))
                    (w (! ? (select now one w ao*)))
                    (alu_inp_a (select now alu_inp_a-? u v))
                    (alu_inp_b (select now alu_inp_b-? 1 w)))
                   (list u v w alu_inp_a alu_inp_b))))])
(system-letrec
   ((ao* (alu alu_ins alu_inp_a alu_inp_b))
    ((now done* alu_ins) (ctl u))
```

```
 ((u v w alu_inp_a alu_inp_b) (dp now ao*)))
(list now u v w done*)))))))
```