Simon Peyton Jones

University of Glasgow
Department of Computing Science

Lilybank Gardens
Glasgow G12 8QQ

A practical technique for designing asynchronous finite-state machines

by

Simon L Peyton Jones

CS Report Series

CSC 91/R2

Department of Computing Science
University of Glasgow
Glasgow G12 8QQ

CSC April 1991

A practical technique for designing asynchronous finite-state machines

Simon L Peyton Jones
Glasgow University

Abstract

The literature on asynchronous logic design is mostly of a fairly theoretical nature. We present below a practical technique for generating asynchronous finite-state machines from a description of their states and transitions. The technique has been used successfully to design a number of state machines in the GRIP multiprocessor.

1 Introduction

The dominant design technique for digital systems is to decompose the system into a number of interacting synchronous finite-state machines. A tremendous literature exists, which describes how to perform this decomposition, and how to implement the state machines thus specified.

Systems based on synchronous finite-state machines suffer from a serious and pervasive disadvantage, concerning clock generation and distribution. The designer of a large system is faced with two options: either he provides a single common clock for the whole system, or he generates local clocks for the various sub-systems. It is well known that both of these options have serious problems, which results in systems that are either slow or inherently unreliable (see Section 4).

A design technique that avoids these problems is to use *asynchronous* finite-state machines, which are driven by changes on their input terminals, rather than by a clock signal. Their use leads to fast and reliable systems, and they are the subject of this paper.

No familiarity with asynchronous design is assumed. The technique presented is applicable both for board-level designs based on off-the-shelf parts, and for VLSI designs.

The approach taken is pragmatic. We use manual techniques for small creative tasks, and computer-aided methods for large well-defined tasks.

2 Notation

The presentation is cast in terms of Boolean algebra, with product denoted "$*$", sum denoted "$+$", and negation denoted with an overbar. Repeated products and sums are denotes by "$\prod$" and "$\sum$" respectively. Logical truth is denoted by "1" and falsity by "0".

A ***boolean expression*** (or ***term***) over a set S is built up in the usual way from elements of S and these three operators. A ***product term*** is a term not involving the sum operator.

3 Asynchronous design is unpopular

While the fundamental ideas have been known for a long time (Unger [1969]), digital design using asynchronous finite-state machines is almost unknown among practicing engineers, who mostly operate on the rule of thumb that "asynchronous equals bad".

This attitude has two origins. Firstly, sometimes a synchronous designer needs something to happen at a time when no convenient clock edge is available to drive it. In these circumstances, designers have been known to resort to using a monostable [1] to generate a transition in the "right" place. The resulting system can be hard to understand, and may even malfunction if the time-constant of the monostable drifts too far. Such designs are justly excoriated, and have led to a mistrust of all non-clock-driven design techniques.

Secondly, glitches on the input of an asynchronous FSM can cause it to make erroneous transitions, so care has to be taken to eliminate all race hazards. (The synchronous system designer can ignore all such races, provided that everything is stable by the next clock edge.) The reliable elimination of race hazards when designing state machines is quite tricky and can increase the size of the implementation (in terms of gates), and this has led to a cultural assumption that asynchronous design is not worth considering.

This assumption may no longer be valid, however. Firstly, the objection that the gate count is increased is rapidly decreasing in importance. For board-level designs, PALs are an ideal vehicle in which to implement asynchronous FSMs, so that increased gate count does not necessarily mean increased package count. For VLSI systems, increasing the gate count comes fairly cheaply, provided the wiring is local, which it is.

Secondly, almost all hardware designers now use CAD tools of one sort or another, which should be able to deal with most of the book-keeping aspects of asynchronous design. Unfortunately, no computer-based design tools for asynchronous FSMs are readily available. This paper makes a start towards such a tool set. In it I describe and justify practical and systematic technique for generating hazard-free asynchronous finite state machines, from state-machine descriptions in a form readily understandable by engineers. I have embodied the ideas in a computer program, which takes a finite state machine description as its input, and produces a set of PAL equations which implement the FSM as its output.

4 Synchronous designs are slow or unreliable

4.1 Why synchronous systems are slow

Large synchronous systems are slow for two main reasons.

Firstly, the clock period must be long enough to ensure the design works under worst-case conditions, taking account of logic delays over the entire temperature range and manufacturer's tolerance range. It is not possible to take advantage of the fact that actual-case logic delays are (with high probability) far shorter than worst-case delays, especially when several independent components are in series.

[1] A monostable gives an output transition at some given time after the triggering input transition. The exact delay is generated independently from a resistor-capacitor network, and is subject to long-term drift.

Worse still, the clock period must be set to satisfy the longest latch-to-latch propagation delay *in the entire system*. It is not possible incrementally to improve the speed of the system, because the clock speed can only be increased when *all* parts of the system are made fast enough to cope. This leads to non-modular designs which can only be improved by a substantial effort involving the entire system.

Secondly, the problems of clock distribution enforce sloth. The designer of a large synchronous system is faced with two alternatives: a single clock may be distributed to the entire system, or each subsystem may generate a local clock.

Distributing a single clock over a large system suffers from two disadvantages:

- The maximum clock skew between different parts of the system must be added to the worst-case logic delay to give the minimum clock period. Thus the larger the system the slower the clock.
- The shortest clock period of the various subsystems may differ; yet each must be driven by a sub-multiple of a single clock. Thus, most sub-systems are running slower than they need, even when performing internal operations only.

On the other hand, if the various subsystems generate local clocks, the following objections may be made:

- Any control or data signals from subsystem A to subsystem B must be synchronised by B. Generally, B will feed external inputs to a synchronisation latch, which samples them on each clock edge, so that the latch outputs can then be used internally in the same way as any other signal. This leads to an average half-clock delay before B even sees the signal from A, and another whole clock cycle before B can respond.
- If the external signal is changing just as the B's clock edge occurs, the register may hang up in a metastable state for an indefinite period. This may mean that the register output is not stable by the next clock edge, which may cause internal malfunction in B. This is called *synchronisation failure*, and is well documented (Chaney & Molnar [1973]).

  One approach to this problem is to reduce the *probability* of failure by slowing the clock down, to give more time for the output of the latch to settle. Alternatively, the output of the latch may be taken to another synchronisation latch, and only the output of the second latch used inside B. Both methods slow down the system.

  Another approach eliminates the possibility of failure altogether by stopping the clock until the system has stabilised. This is possible in VLSI but no off-the-shelf parts are available to support this approach for board-level designs. Furthermore, it imposes an unbounded possible delay on the system operation.

To conclude, both alternatives lead to slow systems, and the latter alternative leads to unreliable systems too.

A typical example would be a bus-based multiprocessor system. If each board has a separate clock then data transfer is hampered by the spectre of synchronisation failure and synchronisation delays. If there is a global clock, then its frequency must allow for worst-case clock skew across the entire system, and all boards are constrained to run off a sub-multiple of this clock.

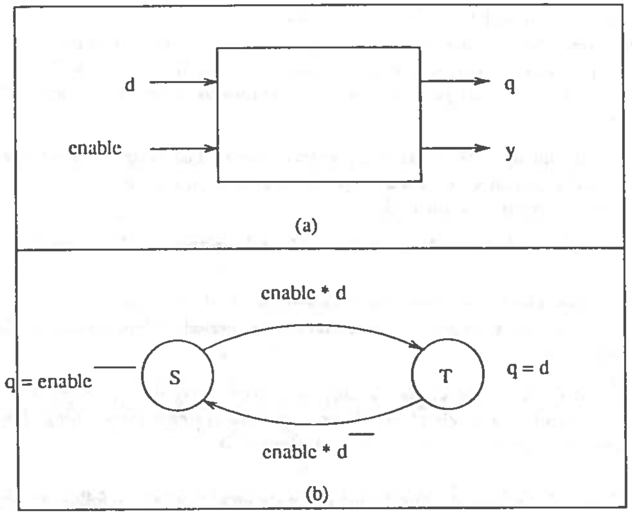

Figure 1: A transparent latch

5 Asynchronous designs are fast and reliable

In an asynchronous design, none of these problems arise:

- The system runs at actual-case logic delays.
- A signal from subsystem A to subsystem B can be acted on immediately, rather than waiting for a clock signal.
- There is no synchronisation problem since there is no clock.

An example of a system based on asynchronous principles is the IEEE P896 Futurebus draft standard. The bus arbitration and data transfer (including broadcast and broadcall), are handled using entirely asynchronous handshakes between participating boards.

5.1 An example

As an example, consider the asynchronous FSM shown in Figure 1, which shows its inputs/output connections, and its state transition diagram. It is a latch, whose state follows the d input when the *enable* input is asserted, and freezes when *enable* is released. It also posesses a slightly curious output q, which follows the d input when in state T and follows $\overline{enable}$ when in state S. q serves no particularly useful function, but makes the design more interesting.

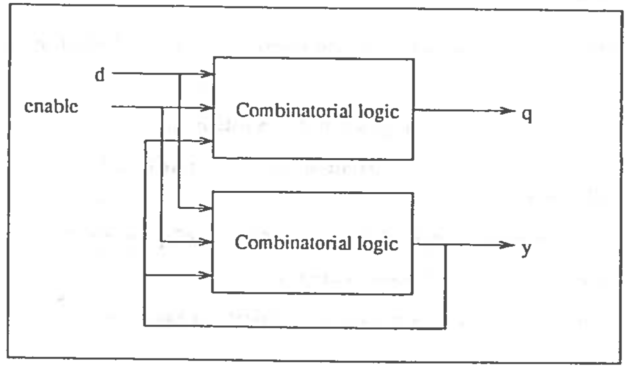

Figure 2: An implementation of the latch

Naturally, there would be a metastability problem if the d input was changing just as the *enable* input was released. It is the responsibility of the designer of the system incorporating the latch to avoid this event. Expressing and reasoning about constraints of this kind, which concern the protocols which must be obeyed by users of an asynchronous component, is the purpose of *trace theory* (Brzozowski & Ebergen [1989]), but is beyond the scope of the present paper.

Such a machine can be implemented using two output pins of a PAL, as illustrated in Figure 2. One output implements the state variable y, and is fed back, and the other implements the q output. Suppose we decide that the machine is in state S when y is released, and in state T when it is asserted. Then the following two equations define y and q:

$$\begin{aligned} y &= y * \overline{enable} + enable * d + y * d \\ q &= \overline{enable} * \overline{y} + d \end{aligned}$$

The equation for y can be interpreted as follows. The $enable * d$ term asserts y when *enable* and d are both asserted, which makes the latch track the state of d when *enable* is asserted. The $y * \overline{enable}$ term keeps y asserted when *enable* is released; and the $y * d$ term is a *cover term* which makes sure there is no glitch in y when *enable* is making a transition.

The equation for q is can be interpreted in a similar way. The $\overline{enable} * \overline{y}$ term makes q follow $\overline{enable}$ when in state S. The d term is more complex. When in state T, q should follow d, and d will itself be released before a transition from T to S can be made. On the other hand, in state S, if *enable* is asserted and d makes a transition from released to asserted, then the machine will make a transition to T, and q should be asserted (since d is). Finally, in state S, if *enable* is released then q is asserted. So we conclude that q should be asserted when d is, regardless of the state of the machine.

There is some subtlety in these equations, and they are far from easy to generate by hand, especially for a large state machine. What is required is an automatic technique for generating the equations from a description of the state machine. We now begin the presentation of just such a technique.

6 The design method

The design method to be presented is split into several steps, each of which is treated in the following sections:

- The asynchronous FSM is specified by a state transition diagram.
- The number of state variables is determined, and each state is assigned a suitable coding of these state variables.
- Boolean equations for each output, including the state variables, are generated.
- The right-hand-sides of these equations are minimised.
- The resulting set of equations is implemented directly in hardware logic.

7 Specification of an asynchronous FSM

An *asynchronous finite state machine* consists of

- A set, $\mathcal{I}$, of *inputs*.
- A set, $\mathcal{X}$, of *outputs*.
- A set, $\mathcal{S}$, of *states*. The states themselves are denoted with upper case letters $R, S, T \ldots$.
- An *output specification*, which specifies for each output $x \in \mathcal{X}$ and each state $S \in \mathcal{S}$ a boolean expression x_S over $\mathcal{I}$, whose value x should take when in state S. Notice that outputs may depend directly on inputs, so an input change may cause an ouput change without causing a change of state.
- A *transition specification* which, for each pair of states S and T, gives a boolean expression κ^{S-T}, over $\mathcal{I}$. The idea is that if the machine is in state S and κ^{S-T} is asserted, it should make an immediate transition to state T. The term κ^{S-S} expresses the conditions under which the machine should remain in state S.

The transitions should be *complete*; that is, in any state S at least one of the κ^{S-T} is asserted. This ensures that the behaviour of the machine is specified under all circumstances. The completeness condition can be written as follows:

$$\sum_{T \in \mathcal{S}} \kappa^{S-T} = 1 \qquad \text{for any } S \in \mathcal{S} \tag{1}$$

Without loss of generality, therefore, we assume that

$$\kappa^{S-S} = \prod_{T \in \mathcal{S}, T \neq S} \overline{\kappa^{S-T}} \tag{2}$$

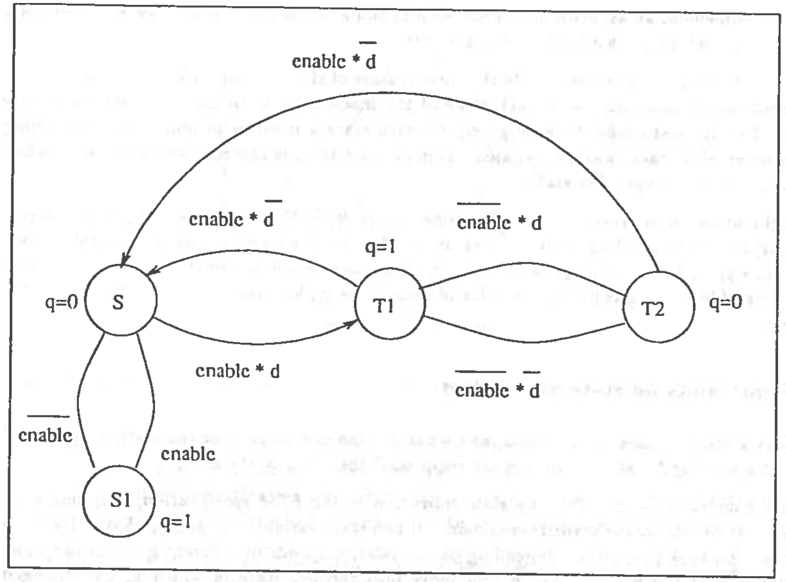

Figure 3: Another transparent latch

The transitions should also be *unambiguous*; that is, in any state S at most one of the κ^{S-T} is asserted. This can be expressed as follows:

$$\kappa^{S-T} * \kappa^{S-T'} = 0 \qquad \text{for any } S, T, T' \in \mathcal{S}, T \neq T' \tag{3}$$

Unambiguity ensures that the machine has only one course of action; if two transition conditions became asserted simultaneously, the machine would have to make a nondeterministic choice about which one to follow, with possibly metastable results. It is up to the designer to ensure that this cannot happen; again, trace theory seems to be the best tool to express and verify this constraint. Sometimes this requires some external synchronising elements when a nondeterministic choice genuinely has to be made, and this must lead either to an unbounded delay, or a non-zero probability of failure.

There are many FSMs which posess a given behaviour, some of which will have more states than others. For example, Figure 3 shows another version of the latch in Figure 1 which has three states, but in which the output q is always either asserted or released in any given state. In general it seems to result in more economical designs to reduce the number of states to the minimum required to "remember" sufficient history to implement the desired behaviour. There are formal techniques to do this (Fletcher [1980, Chapter 10]), but they are quite complex, and it seems to be a process that is rather easy to perform by hand.

8 State assignment

In order to implement an asynchronous FSM we first add a new set of outputs, the *state variables* $\mathcal{Y} = y_1, \ldots, y_n$, which are fed back as extra inputs.

The values of the state variables encode the current state of the machine. We must devise a *state assignment*, which associates with each state of the machine a particular value for each state variable. This is most easily done by giving for each state a boolean product term containing every member of $\mathcal{Y}$, each possibly negated. This product term is asserted when and only when the machine in in the specified state.

By a slight abuse of notation, we use the same letters $R, S, T\ldots$ to denote the product terms identifying the corresponding states. Thus, in our Figure 3, we could encode the states thus: $S = \overline{y_1} * \overline{y_2} * \overline{y_3}$, $T1 = \overline{y_1} * y_2 * \overline{y_3}$, and so on. Another form which is sometimes convenient is to give a string of binary digits giving the value of each of the y_i; for example, $S = 000$, $T1 = 010$, and so on.

8.1 Constraints on state assignment

How many state variables are required, and what are the constraints on the state assignment? Clearly at least $\lceil \log N \rceil$ state variables are required if there are N states.

The main constraint is this: *for any state transition in the FSM specification, only one state variable must change as the transition is made.* If two state variables y_a and y_b changed when a state transition took place, then depending on the relative speeds of different gates in the logic, it is possible that the machine would first move into another state in which y_a had changed but not y_b or vice versa. These other states might be part of some other part of the machine altogether, so this would clearly be a disaster.

For example, supposing state S of Figure 3 were coded by $S = 000$, $T1 = 110$, and $T2 = 010$. Then, in the transition from S to T, the machine might go through the intermediate states 010 or 100, depending on the relative rates at which the state bits changed. But one of these "intermediate" states is actually $T2$, which has a whole set of transitions of its own!

In short, unlike state assignments for synchronous machines, the state assignment for an asynchronous machine must juxtapose (in the boolean state space) states between which a transition can be made.

8.2 Practical techniques

It is therefore necessary to devise a state assignment which respects the single-change constraint. Much the easiest approach is based on the observation that orthogonally adjacent squares in a Karnaugh map represent terms which differ by only one bit. Hence, ***finding a consistent state assignment boils down to embedding the state diagram in a Karnaugh map, with each state corresponding to a square, so that all the state transitions are from one square to an orthogonal neighbour.***

Sometimes it is not possible to perform such a state assignment. For example, a machine with three states, each of which has a single transition to a successor state (Figure 4), has no direct

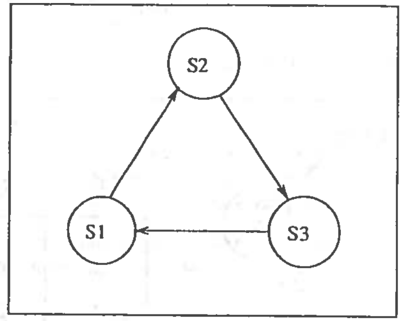

Figure 4: A three-state FSM

state assignment. Under these circumstances three techniques can help:

- Extra intermediate states can be inserted to bridge the gap between two states between which a transition must be made. The intermediate states make an immediate transition into the desired destination state. For example, an extra state could be inserted in Figure 4, giving Figure 5(a), for which a possible state assignment is given in Figure 5(b).
- Existing states can be duplicated. For example, each state of the three-state machine of Figure 4 could be duplicated to make a six-state machine, which can be neatly embedded in a Karnaugh map as shown in Figure 6 [2]. This is also useful when one state is very heavily connected to others.
- It is possible to make a "diagonal" transition, in which two state variables change simultaneously, provided that the two "phantom" states are also programmed to make an immediate transition into the destination state. For example, suppose the state S is encoded 000, and T by 110. It is still possible to make a direct transition from S to T, provided that the states encoded by 010 and 100 are both programmed to make an unconditional transfer into state T. Thus, if the transition from S to T happens to move first into one of these phantom states, the transition programmed for the phantom state will be consistent with the transtion from S to T. It follows that the phantom states are unusable for any other part of the machine.

  Diagonal transtions where more than two bits change simultaneously are also possible, but the more bits which are changed simultaneously, the larger the area of the state space which is thereby rendered unusable for other parts of the machine.

Whilst mechanical assistance would be a help for this step, for machines with up to twenty states it is an easy task to perform by hand.

[2] Remember that opposite edges of a Karnaugh map "wrap around" to each other.

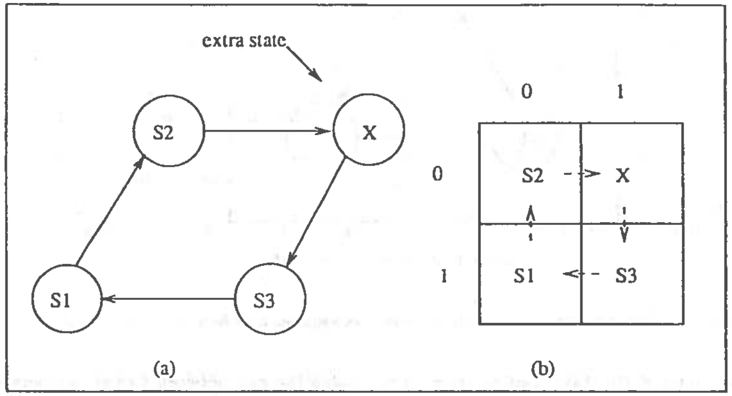

Figure 5: Adding an extra state to get a legal state assignment

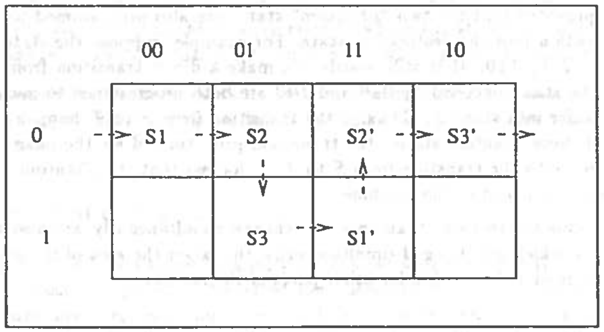

Figure 6: Karnaugh map embedding a replicated three-state FSM in a six-state cycle

9 Generating the equations

Finally comes the step of generating the equations which implement the FSM, which is the main contribution of this paper.

Rather than treat the state variables and the outputs differently, we regard the state variables themselves as outputs, which are asserted in each state whose encoding asserts the variable and released otherwise. This reduces the problem to generating a combinatorial term for each output $x \in \mathcal{X} \cup \mathcal{Y}$, in terms of the inputs $\mathcal{I}$ and the state variables $\mathcal{Y}$.

The approach we take is to generate a highly redundant term for each output x, and subsequently to minimise it. The latter process is described in the next section, while in this section we show how to generate the term.

Each such term is the sum of a term S_x for each state S, which describes when x should be asserted while in state S, and during the transition from S to any other state:

$$x = \sum_{S \in \mathcal{S}} S_x \tag{4}$$

The term S_x has two components. The first component, S_x^{hold}, tells when x should be asserted while the machine is in state S and is not making a transition. The second component, S_x^T, is a cover term which covers x during an $S \rightarrow T$ transition.

$$S_x = S * S_x^{hold} + \sum_{T \in \mathcal{S}, T \neq S, \kappa^{S \rightarrow T} \neq 0} (S + T) * S_x^{cover(T)} \tag{5}$$

The $(S+T)$ part of the second term is used to bring S_x^T into play either in state S or in state T. Since the states are encoded so that only one state variable, y_{ST} say, changes during any transition, the product terms for S and T will be identical except that one will contain y_{ST} asserted and the other will contain y_{ST} negated. Hence $(S+T)$ will be a product term identical to that for S and T but omitting y_{ST}. For example, suppose that S was $y_1 * \overline{y_2} * \overline{y_3}$ and T was $y_1 * y_2 * \overline{y_3}$, then $S+T$ would be $y_1 * \overline{y_3}$.

The hold term S_x^{hold} is easy:

$$\begin{aligned} S_x^{hold} &= x_S * \prod_{T \in \mathcal{S}, T \neq S} \overline{\kappa^{S-T}} \\ &= x_S * \overline{\kappa^{S-S}} \end{aligned}$$

That is, in state S the output x should be asserted if x_S (the output specified for x in S) is asserted, *and none of the exit conditions are asserted.* The big product switches off the S_x^{hold} term as soon as a transition begins, in case x should be released in the destination state. At first this seems like a luxury: surely x will be released as soon as the transition actually occurs, so that we could use the simpler equation $S_x^{hold} = x_S$. This is true for ordinary ouputs, but suppose that x is a state variable which is asserted in S and released in the destination state. Then, if the big product were not present in S_x^{hold}, the state variable would never be released, so the transition would never occur!

The cover terms, $S_x^{cover(T)}$, from Equation 5 are more complicated. Each is designed to cover the transition from S to any other state T to which it can make a transition. At first it seems that $S_x^{cover(T)}$ should be as follows:

$$S_x^{cover(T)} = \kappa^{S-T} * x_T \tag{6}$$

That is, it should be asserted when a transition is being made (κ^{S-T} asserted), and when the output in the destination state (x_T) is asserted. But matters are a litte more complicated than this. To begin with, the hold term for S, S_x^{hold}, has a term including $\overline{\kappa^{S-T}}$, so there is a possible hazard when κ^{S-T} makes a transition from released to asserted. To avoid this, we must add a term representing the negation of the *other* exit conditions:

$$S_x^{cover(T)} = S_x^{trans(T)} * x_T \tag{7}$$

$$S_x^{trans(T)} = \kappa^{S-T} + x_S * \prod_{T' \neq T, T' \neq S} \overline{\kappa^{S-T'}} \tag{8}$$

Even this is not quite right yet. Suppose a transition has been made into state T, and κ^{S-T} remains asserted; then $S_x^{cover(T)}$ will remain asserted too, and hence so will x. But suppose that $\kappa^{T-T'}$ now becomes asserted as well, so that the machine should make a transition from T to T', and that x is released in state T'. It is wrong for $S_x^{cover(T)}$ to remain asserted under these conditions; if x were the state variable which distinguished T from T', the transition would never be made, because x would never be released.

We need to modify $S_x^{cover(T)}$ to make it release under these circumstances, to take account of a "lookahead" from T. If any of the $\kappa^{T-T'}$ are true, we need to take account of the value of x in state T'; if they are all false then $S_x^{cover(T)}$ should be as before:

$$S_x^{cover(T)} = S_x^{trans(T)} * x_T * T_x^{lookahead} \tag{9}$$

$$\begin{aligned} T_x^{lookahead} &= \prod_{T' \neq T} \overline{\kappa^{T-T'}} + \sum_{T' \neq T} x_{T'} * \kappa^{T-T'} \\ &= \kappa^{T-T} + \sum_{T' \neq T} x_{T'} * \kappa^{T-T'} \end{aligned} \tag{10}$$

This completes the description of how the equations can be generated. The terms generated are quite large, and it is essential to provide mechanical support for this process. Figure 7 [3] summarises the relevant equations.

The final step is to minimise the resulting equations to remove redundancy.

[3] The use of $x_{T'}^{early}$ in these equations is a forward reference to an optimisation discussed in Section 11.1, but meanwhile can safely be read as $x_{T'}$

$$\begin{aligned}
\kappa^{S-S} &= \prod_{T \in S, T \neq S} \overline{\kappa^{S-T}} \\
x &= \sum_{S \in S} S_x \\
S_x &= S * S_x^{hold} + \sum_{T \in S, T \neq S, \kappa^{S-T} \neq 0} (S+T) * S_x^{cover(T)} \\
S_x^{hold} &= x_S * \kappa^{S-S} \\
S_x^{cover(T)} &= S_x^{trans(T)} * x_T * T_x^{lookahead} \\
T_x^{lookahead} &= \kappa^{T-T} + \sum_{T' \neq T} x_{T'}^{early} * \kappa^{T-T'} \\
S_x^{trans(T)} &= \kappa^{S-T} + x_S * \prod_{T' \neq T, T' \neq S} \overline{\kappa^{S-T'}}
\end{aligned}$$

Figure 7: Summary of equation generation

10 Minimisation

The equations generated in the previous section are highly redundant, and would consume an inordinate amount of hardware if implemented directly. Some care needs to be taken, however, with minimising the expressions, because uncontrolled minimisation can eliminate hazard cover. For example, given the term $x * y + \overline{x} * z + y * z$, many minimisers would remove the $y * z$ term, thereby creating a possible hazard when x makes a transition.

It is possible to apply some simple rules of Boolean algebra to perform some minimisation without removing hazard cover:

$$\begin{aligned}
a * 1 &= a \\
a * 0 &= 0 \\
a + 1 &= 1 \\
a + 0 &= a \\
a + a &= a \\
a + \overline{a} &= 1 \\
a * a &= a \\
a * \overline{a} &= 0 \\
a + a * b &= a
\end{aligned}$$

Each of these rules preserves hazard cover, because each only involve the absorption of one term into another which covers it, or combines with it to form a larger term.

Unfortunately, using only these rules misses out on some useful minimisations. For example, the term $x * y * z + \overline{x} * z + x * \overline{y}$ is equivalent to $x * \overline{y} + z$, but the simplifications given above will not discover this fact. In short, *a minimiser powerful enough to discover all simplifications will also eliminate essential hazard cover.*

The solution is quite simple. For any equation $x = T$, where T is a term, first minimise T with the most powerful minimiser available to obtain a new term T'. Since $T = T'$ it is certainly true that $x = T + T'$, and the term $T + T'$ preserves all the hazard cover in the original T.

The second step is to minimise $T + T'$ with an absorption-only minimiser. Any large covering terms in T', discovered with the powerful minimiser, will absorb their component terms in T, resulting in a final term for x, which is both minimal and preserves all hazard cover.

Sadly, not all minimisers provide control over the level of minimisation to be performed. We used a PC-based minimiser which is part of a PAL programming package called CUPL, which does provide this feature.

11 Improvements

Even after minimisation the equations can be too large to fit in a PAL, so we searched for ways of further reducing the complexity of the equations. In this section we give three techniques we have found useful. Each can simplify the final equations, but at the cost of complicating the generation process somewhat.

11.1 Intelligent lookahead

The term $T_x^{lookahead}$ "looks ahead" to the value of x in each state T' accessible from T. While the value of x in state T' may be a complex expression, it is often the case that the designer knows that this expression will always be asserted or released *at the moment a transition is made into* T'. In general, the designer can give an expression $x_{T'}^{early}$ which gives the value of x as a transition into T' is made, where $x_{T'}^{early}$ is simpler than $x_{T'}$ [4]. Typically $x_{T'}^{early}$ is either 0 or 1.

Since $T_x^{lookahead}$ is only significant until the machine leaves state T, it suffices to use $x_{T'}^{early}$ instead of $x_{T'}$ in the expression for $T_x^{lookahead}$. We have found that this sometimes results in a significant simplification of the equations.

11.2 Multi-way exits

Frequently, a state is left when some condition α becomes asserted, but which of several destination states are entered is then controlled by some other conditions β, γ, etc. For example, suppose there are just two transitions out of state S, to $T1$ and $T2$ respectively, such that:

$$\begin{aligned} \kappa^{S-T1} &= \alpha * \beta \\ \kappa^{S-T2} &= \alpha * \overline{\beta} \end{aligned}$$

[4] Theoretically, the designer might wish to give an expression $x_{T'}^{early(T)}$ for each source state T, but in practice we have found this level of specification to be unnecessary.

That is, S will be left when (and only when) α becomes asserted, while β controls which of $T1$ and $T2$ is entered. Then it follows from Equation 8 that:

$$\begin{aligned} S_x^{trans(T1)} &= \alpha * \beta + x_S * \overline{(\alpha * \overline{\beta})} \\ &= \alpha * \beta + x_S * \overline{\alpha} + x_S * \beta \end{aligned}$$

This expression is more complex than necessary. The term $S_x^{trans(T)}$ is concerned with covering a transition from S to T, and if α is released then no such transition can take place. Hence the term $x_S * \overline{\alpha}$ can be eliminated from $S_x^{trans(T)}$, giving:

$$S_x^{trans(T1)} = \alpha * \beta + x_S * \beta \qquad (11)$$

We have not been able to find a way to formalise this transformation. What we do in practice is to try to spot common terms between κ^{S-T} and $\kappa^{S-T'}$, and eliminate them from $S_x^{trans(T)}$.

11.3 Use of don't care terms

Consider the term $S_x^{trans(T)}$ again:

$$S_x^{trans(T)} = \kappa^{S-T} + x_S * \prod_{T' \neq T, T' \neq S} \overline{\kappa^{S-T'}} \qquad (12)$$

The big product is required to release the term when a transition should be made to some other state T'. But, if x is asserted in T' then $S_x^{trans(T)}$ does not need to be released when $\kappa^{S-T'}$ becomes asserted; remember that $S_x^{trans(T)}$ is ANDed with $(S + T)$, so that it will become irrelevant anyhow as soon as the transition to T' takes place.

The conclusion is that $\kappa^{S-T'}$ can be omitted from the big product if $x_{T'}^{early}$ is 1. But this is not necessarily helpful! The expression might be simplifiable further if $\kappa^{S-T'}$ was left in the product. What is really required is to put $X * \overline{\kappa^{S-T'}}$ in the product, where X is the "don't care" value. This should leave the minimiser free to choose an arbitrary value for X which most benefits the minimisation process.

Unfortunately, not many minimisers can deal with "don't care" terms. Our's did not, and we compromised by choosing $X = 0$ all the time.

12 Related work

Recent years have seen renewed interest in "self-timed" systems (which seems to be another name for asynchronous systems), particularly in the VLSI world where the problems of clocking synchronous machines are particularly serious (Chu, Leung & Wanuga [1985]; Seitz [1980]). Most papers on the subject are rather academic, however, and do not purport to suggest practical design techniques for engineers. Furthermore, the VLSI bias of the papers often makes them inapplicable to designers working at the chip level. Brzozowski and Ebergen give a useful and up-to-date survey (Brzozowski & Ebergen [1989]).

Hollar (Hollar [1982]) describes a systematic design technique using the so-called "one-hot" method. This works fine, but uses a lot of PAL outputs (the scarce resource). The method we describe allows a more flexible state-assignment.

Sutherland's Turing Award lecture (Sutherland [1989]) provides a highly accessible introduction to asynchronous design, based on *micro-pipelines*. He shows how to design event-driven hardware in a similar way to drawing a flow diagram for a program. This is very suitable for pipelines, but rather less so for state machines.

Most of this background work is based on *transition signalling*, in which an event is associated with *either* the assertion *or* the release of a signal line. Each is regarded as the *same* event. This contrasts with conventional digital design, in which an event is normally associated only with an assertion of a signal, and the signal must subsequently be released so that it can be re-asserted to indicate the next event.

Symmetrical transition signalling is *elegant*, because one transition is exactly the minimum required to signal an event, and *fast*, because it eliminates the need to reset the signal line. It does require a completely symmetrical approach to the design of state machines, and Boolean algebra is not a suitable tool for this purpose. The OR operation is implemented by an EXCLUSIVE-OR gate, and the AND operation requires a Muller C-element which has state.

Nevertheless, a number of the machines we have designed using the techniques in this paper have used a form of transition signalling. The entire state machine was duplicated, with one half expecting assertions on inputs and generating assertions on outputs, and the complementary half expecting and generating releases.

13 Experience and conclusions

The techniques given in this paper seem to work, and have been used successfully in practice. Of the four steps described, namely specification, state assignment, equation generation and minimisation, the latter two have been completely automated.

The equation generation is done by a 350-line program in the functional programming language Miranda [5], most of which concerns expression manipulation and output formatting. The heart of the program implements the equations of Figure 7, and is only 20 lines long.

The minimisation is done by the minimiser packaged with the CUPL PAL programming system. More sophisticated minimisation programs, such as Expresso, seem to lack control over the level of minimsation, which rendered them useless for the purpose (see Section 10).

State encoding is done manually. It would be nice to automate this step as well, but it is relatively hard to do so, and it seems rather easy to do with a pencil and paper for small machines. The effort of automation would be excessive for the examples we have tackled.

The GRIP multiprocessor (Peyton Jones et al. [1987]) contains a number of asynchronous FSMs designed in this way, including an arbiter, a memory access sequencer, send and receive machines for block data transfer over an asynchronous multi-master bus (the IEEE Futurebus), and a distributed arbiter for the same bus. A companion paper gives further details of the asynchronous Futurebus interface (Peyton Jones & Hardie [1991]).

[5] Miranda is a trade mark of Research Software Ltd.

The largest of these machines contains a dozen states, ten inputs and five outputs. In all cases, the time required to perform state encoding and equation generation was very substantially less than the time taken to design the state transition diagram and verify correct operation.

The production of the state machine only became tricky if the equations produced had too many product terms to fit into a PAL, which happened on a couple of occasions. Under these circumstances the designer has to understand exactly why, and figure out a way to modify the state transitions to produce fewer product terms. This is a difficult process, but it is just another aspect of the general problem of fitting a design into a fixed set of components. A VLSI designer would have no such difficulties.

Apart from this difficulty, there seemed to be no increase in design effort associated with asynchronous FSMs, compared with using a synchronous techniques. Given the benefits of asynchronous design outlined above, this is an encouraging result.

This work was driven by a very pragmatic objective: we had some asynchronous FSMs to design and we needed a design tool. The major shortcoming is the absence of any formal model for guaranteeing the correctness of the equations. All the arguments given are informal, and it would be reassuring to have a proof of correctness to back them up.

14 References

JA Brzozowski & JC Ebergen [Aug 1989], "Recent developments in the design of asynchronous circuits," in *Proc 7th Intl Conference on Fundamentals of Computation Theory, Szeged, Hungary*.

TJ Chaney & CE Molnár [Apr 1973], "Anomalous begaviour of synchroniser and arbiter circuits," *IEEE Trans Computers* C-22, 421–422.

T-A Chu, CKC Leung & TS Wanuga [Oct 1985], "A design methodology for concurrent VLSI systems," *Proceedings of ICCD-85*.

WI Fletcher [1980], *An engineering approach to digital design*, Prentice Hall.

LA Hollar [Dec 1982], "Direct implementation of asynchronous control units," *IEEE Trans Computers* C-31, 1133–1141.

SL Peyton Jones, Chris Clack, Jon Salkild & Mark Hardie. [Sept 1987], "GRIP - a high-performance architecture for parallel graph reduction," in *Proc IFIP conference on Functional Programming Languages and Computer Architecture, Portland*, G Kahn, ed., Springer Verlag LNCS 274, 98–112.

SL Peyton Jones & M Hardie [Feb 1991], "A Futurebus interface from off-the-shelf parts," *IEEE Micro*.

CL Seitz [1980], "System timing," in *Introduction to VLSI systems*, C Mead & L Conway, eds., Addison Wesley, 218–262.

IE Sutherland [June 1989], "Micropipelines," *CACM* 32, 720–738.

SH Unger [1969], *Asynchronous sequential switching circuits*, Wiley.