ASSASSIN: A CAD System for Self-Timed Control-Unit Design

Tony M. Carter
Department of Computer Science
University of Utah
Salt Lake City, Utah 84112

October 1982

Abstract

Many software systems exist for automatically implementing synchronous state-machines. Presented in this paper is a software system — ASSASSIN — for the design and automatic layout of self-timed (or speed-independent) control-units as integrated circuit modules. ASSASSIN provides for the editing of textual descriptions of control-flow, the functional simulation of speed-independent control-units, and the automatic layout of the implementation as a Path-Programmable Logic (PPL) program. ASSASSIN uses a well-known technique (a one-hot state encoding) for implementation of the control-unit. Examples are given illustrating the specification and implementation of simple state-machines. In addition, the design of a state-machine of interest in the University of Utah's Ada-to-Silicon project is carried out. A portion of the Ada[1] code for the "Output Side" of the Inter-Net-Module (INM_OUT), which will eventually be fabricated as part of the Ada-to-Silicon Project, is converted by hand to ASSASSIN input format and from there to an integrated circuit layout by ASSASSIN, thus illustrating the use of ASSASSIN in the context of the Ada-to-Silicon Project.

This work was sponsored in part by the Defense Advanced Research Projects Agency (DARPA) under contract number MDA 903-81-C-0414.

1. Introduction

The development of CAD tools for integrated circuit design has exploited a vast body of knowledge about synchronous computing systems. Old and new integrated circuit technologies have been well-suited for implementing synchronous computing systems. The success of these synchronous systems has been prodigious as witnessed by the recent booms in the manufacturing and purchasing of computing systems. Current research in semiconductor devices is rapidly heading toward the ability to construct computing systems which operate orders of magnitude faster and which are far more complex than those currently available. ASSASSIN treats part of problem of designing self-timed systems.

With projected room-temperature speeds of logic devices ranging down to tens of picoseconds of delay time [3], it appears that the postulate advanced by Seitz in Chapter 7 of *Introduction to VLSI Systems* [7] will be borne out. The contention is that the current methods of system synchronization (global clocks) will result in unreliable circuits as device speeds increase and as device switching energies decrease.

If Seitz is indeed right, the newer and faster integrated circuit technologies will require computing systems to be implemented using something like "Self-Timed" or "Speed-Independent" logic. In these types of logic, only sequence is of concern. The actual gate and wiring delays will not affect the function, only the absolute speed. It should be noted that any asynchronous device requires that the

---

[1] Ada is a registered trademark of the U.S. Government, Ada Joint Program Office.

surrounding environment to be suitably conditioned so as to tolerate the "un-synchronized" actions of the device.

Much work has been done in the implementation of synchronous structures in integrated circuits. Computing systems can be divided into two main parts: control and data-path. Universities and industry alike have produced many methods for generating synchronous system control, some using the PLA. Work has and is being done in the automatic generation of synchronous data-paths [9]. While there have been some successful efforts to construct self-timed or speed-independent computing systems such as DDM 1 [2] and ILLIAC II [8], there has been very little work done on the implementation of self-timed computing systems in integrated circuits. This may be because there were few integrated circuit implementation strategies which readily lent themselves to the construction of self-timed circuits.

The development of Path-Programmable Logic [1] (PPL), a derivative of the Storage/Logic Array (SLA) [10], has proven to be of great value in the generation of self-timed control in integrated circuits.

ASSASSIN is part of a research effort, being pursued at the University of Utah, to convert Ada programs into integrated circuit implementations. ASSASSIN transforms the control portions of Ada programs into their corresponding integrated circuit counterparts. In addition, ASSASSIN [1] provides a software tool for the specification, simulation and compilation of self-timed control-units to integrated circuit module layouts. As such, it begins to treat some of the low-level problems of self-timed systems design. It uses PPL as the integrated circuit implementation strategy and a one-hot encoding of the control states [4] as a mapping from the specification to the circuit implementation. It allows an implementation independent specification of control (that is, independent of fabrication technologies and circuit implementation techniques), and provides functional simulation capabilities. Layout generation (analogous to the software compiler code generation) results in self-timed circuits which functionally match the results of simulation. ASSASSIN also provides a single, convenient user interface for all of its functions.

2. The Specification of Control: Syntax

The specification of control for a given circuit can result in a labelled, directed graph similar to the one in figure 2-1. There are named nodes which are called states and labelled directed arcs which are called transitions. Associated with states are operations on output variables. These operations may be functions of only the state, or they may be functions of the state and a boolean function of a set of input variables. Transitions are labelled with a boolean function of members of the set of input variables which dictates the condition upon which that transition will take place. Transitions may also have associated operations on outputs (Mealy Machines).

The ability to specify strictly sequential control is certainly essential. Although our current understanding of concurrent processing is very limited, the ability to handle concurrent paths of control may also prove to be useful as our understanding increases. Concurrency (in the context of control) can be interpreted in two ways. The first is where two separate machines operate independently, communicating via some signalling protocol. The second is where a single machine performs some types of concurrent processing by having concurrently executing control paths. The first is handled by having control-units composed of multiple state-machines. In terms of graphs, this implies that one can draw many separate graphs, whose interconnection is implied by output and input variable names. The second is handled by allowing, within a single state-machine, some notion of forking to begin concurrently executing control paths and a notion of joining to terminate concurrently executing control paths. The addition of the concepts of FORK and JOIN to the graph model of control-flow is illustrated in figure 2-2.

Output generation from a control-unit can be either enduring or ephemeral. Enduring outputs

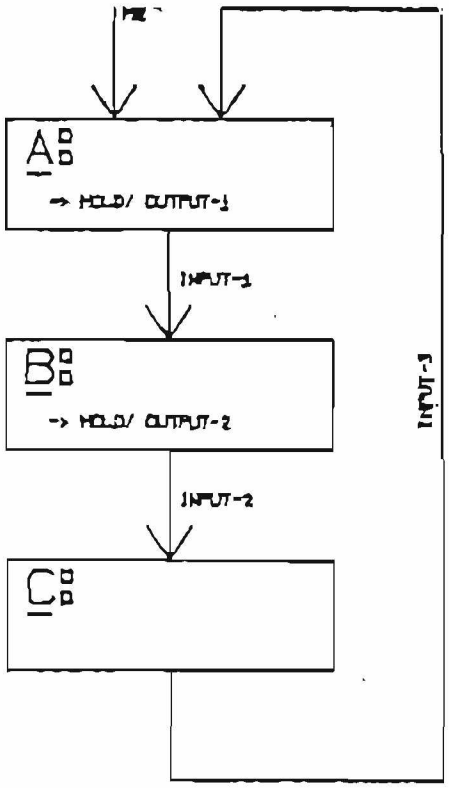

Figure 2–1: A Simple Control–Flow Graph

are latched and operated on by SET and RESET only. When an enduring output is SET it will remain on until a RESET operation is performed. Ephemeral outputs are gated and remain on only while the required condition is met (either residence in a state or execution of a transition). They are operated on by HOLD.

Figure 2–3 contains a control–flow graph which contains all of the features included in the discussion above. States are represented by rectangles with the name of the state indicated in the upper left corner, followed by a colon. Output generation is indicated by a right–arrow. To the left of the right–arrow will be a boolean expression and to the right the operations to be performed and the names of the outputs which are to be operated on. For example, State B contains three output operations. The first is unconditional (it depends only on the state of the machine) and causes the ephemeral output "O1" to be held true. The second is conditional (the boolean expression is "I3") and causes the enduring output "O3" to be SET. The third is also conditional (the boolean expression is "I4 OR I5") and causes the ephemeral outputs "O2" and "O5" to be held true and the enduring output "O4" to be RESET.

Also required in the specification of control is the concept of an initial state. In the graphs, this is indicated by the arc labelled MasterReset which has no state node at its tail.

In summary, the specification language for control should include the following features:

- the concept of an initial state,
- simple transitions from one state to another (MOVE),
- transitions from one state to many states (FORK),
- transitions from many states to one state (JOIN),
- outputs controlled only by residence in a state or by the execution of a transition,
- outputs controlled by a boolean combination of inputs AND by residence in a state or by the execution of a transition,

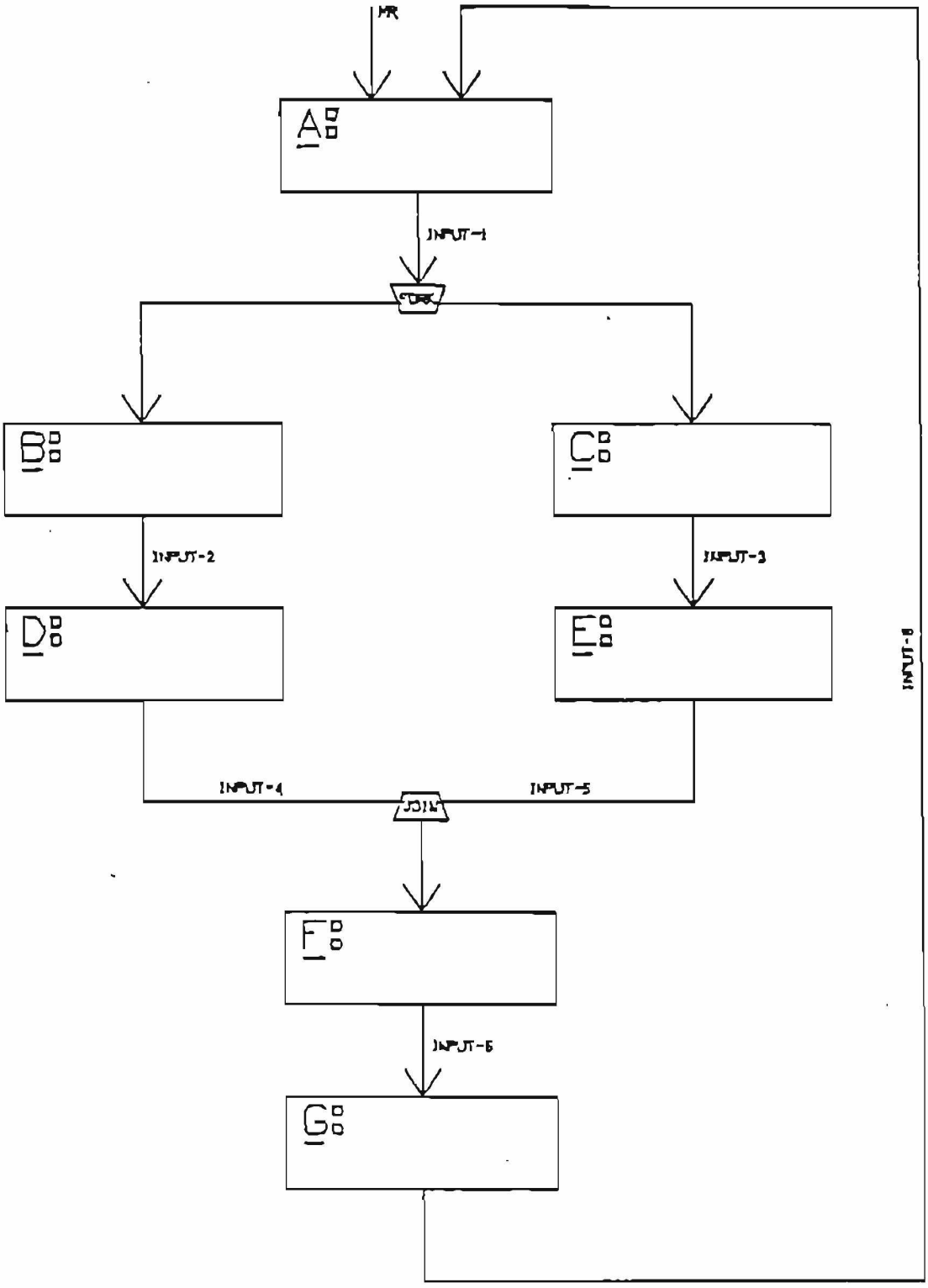

Figure 2-2: A Control-Flow Graph With Concurrency

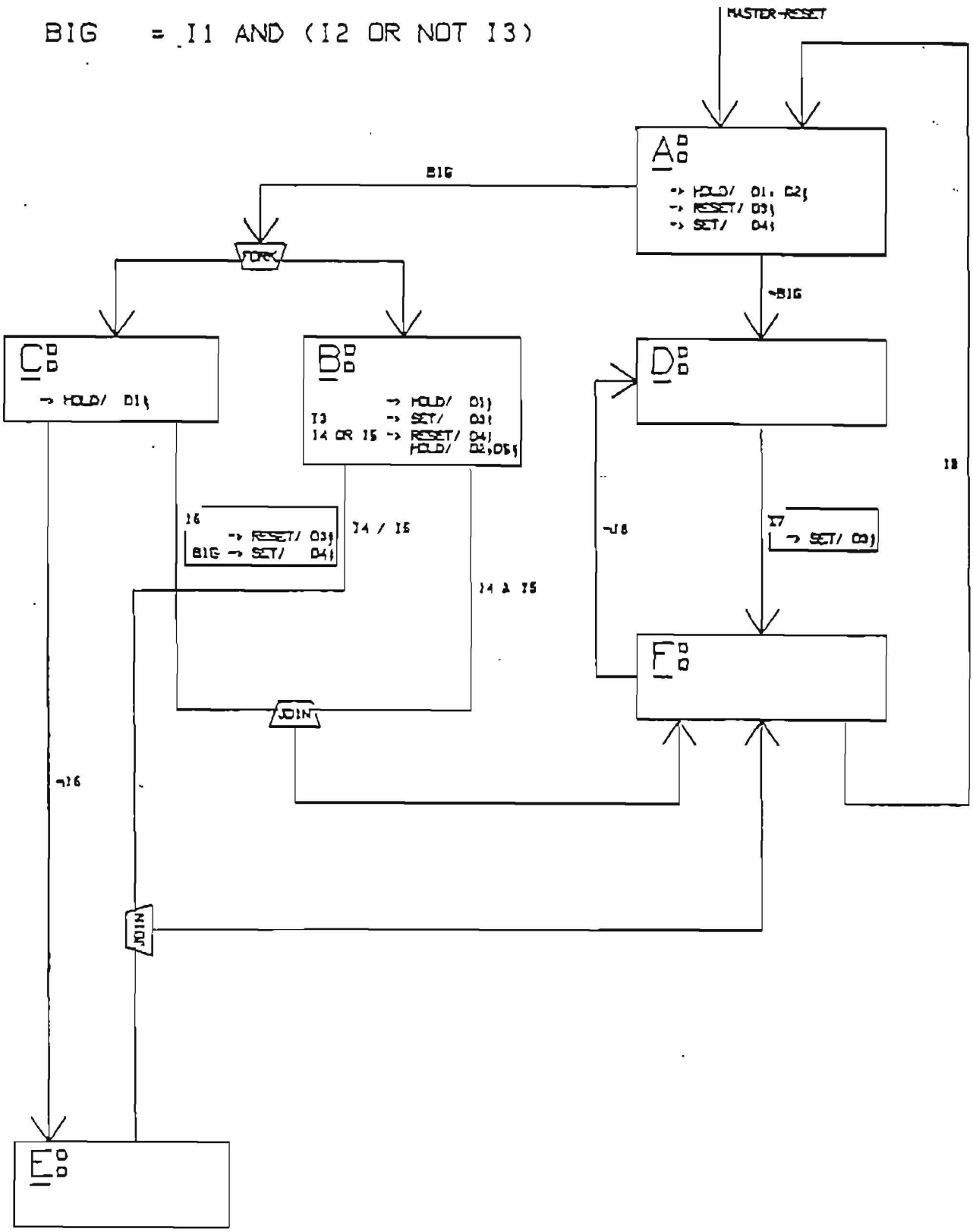

Figure 2-3: A Complex Control-Flow Graph

- —arbitrarily complex boolean expressions for conditions (controlling transitions and output generation),
- —lambda transitions (where the condition is the tautology TRUE),
- —ephemeral outputs,
- —enduring outputs,
- —multiple and varied transitions from a given state,
- —multiple and varied transitions to a given state, and
- —multiple state—machine control—units.

The task now is to codify the points listed above, such as in a grammar in BNF. It must allow for all the points listed above while limiting its expressive power to those points. The language must be easily parsed and it is desirable that parser generators be used to generate the code for the parser. A bove all, the language should be concise and intelligible to design engineers.

The complete BNF for the language (which is called CUDL) is included in Appendix I. The language has the ability to represent each of the points listed above. There are four types of blocks in the language. The first is the CONTROLUNIT block. This block indicates the name of the overall control—unit and contains STATEMACHINE blocks. It also includes the specification of "global" input expressions which assign boolean expressions to an internal variable which can significantly reduce the size of the code written to describe the control—unit. The names of "global" inputs can be used in the descriptions of transitions and output generation. Figure 2—4 contains the CUDL code describing the machine whose graph is in figure 2—3.

```
controlunit CompileTest9:

inputs: BIG := I1 and (I2 or not I3);

   selftimed statemachine Test9:
      startstate A:
         forkon BIG to B,C;
         moveon NOT BIG to D;
         hold O1,O2;
         reset O3;
         set O4;
      end;

      state B:
         joins C on I4 AND I5 to F;
         joins E on I4 OR I5 to F;
         hold O1;
         if I3 then set O3;
         if I4 OR I5 then begin reset O4; hold O2,O5; end;
      end;

      state C:
         moveon NOT I6 to E;
         joins B on I6 to F doing begin reset O3;
                                       if BIG then set O4; end;
         hold O1;
      end;

      state D:
        moveon I7 to F doing set O3;
      end;

      state E:
        joins B on TRUE to F;
      end;

      state F:
        moveon I8 to A;
        moveon NOT I8 to D;
      end;
   end;
end.
```

Figure 2—4: CUDL Code for the Graph in Figure 2—3

Eventually, given an appropriate display device, a graphical version of this language could be developed and the specification of control could be done in terms of control-flow graphs rather than a textual description of the graph. A project is underway to implement such a front end to ASSASSIN on an Apollo DOMAIN computer.

3. The Simulation of Control: Semantics

Given that the syntax of control-unit specification is defined, the designer must also understand the semantics in order to use the system. The semantics of control is directly influenced by the implementation strategy selected. Since the specification of control should allow for concurrency within a given state-machine, a scheme which allows the implementation of such concurrency must be selected. The notion of concurrency eliminates the possibility of completely and uniquely encoding the state variables. The one-hot implementation scheme (completely decoded) allows for easy implementation of concurrency. The following discussion is largely based on the assumption that a one-hot implementation is used.

The specification syntax described in the previous section can be interpreted in three ways. The interpretation depends on the particular mapping strategy being used in the compilation. The three possible types of mapping are synchronous, asynchronous, and self-timed. In order to allow for all three interpretations to be eventually simulated and compiled, the language includes the concept of a state-machine type. The choice of a state-machine level semantic interpretation is made explicit through the use of the keywords: SELFTIMED, ASYNCHRONOUS, and SYNCHRONOUS. In this way, the user can specify various types of control using the same system. Only the SELFTIMED option is currently implemented in ASSASSIN.

The simulation of self-timed control can be functional in nature. This functional simulation provides knowledge about the sequential function of the circuit. Since the implementation of the circuit is such that if sequence is correct, function is correct, the user is sure that the circuit will work if the environment in which he places it is conditioned to interact in a self-timed manner with the control-unit.

The simulation of synchronous and asynchronous control really requires the use of a detailed timing simulator. This simulator must be able to make accurate delay calculations based on variable gate delays. In the world of the integrated circuit, these delays may or may not be easily calculated since long wires and heavy loads will significantly alter the operation of any given gate. Thus, the problem of simulation for these types of systems is much more difficult that for the self-timed systems.

To interpret the semantic actions of the control-unit, one must know first the actions to be taken to execute a transition and second how outputs are generated. Transitions are operations that change the internal state of the machine. Although there may be many transitions specified for leaving a given state, it should never be possible to execute two transitions concurrently from the same state. Since the control-unit has no control over the sequence of arrival and the timing of the inputs that trigger transitions, the problem of having two transitions executed simultaneously is inherently a dynamic one and its avoidance requires a detailed knowledge of the environment into which the control-unit is to be placed. If two transitions were executed simultaneously, the result would be a state-machine which would be in two sequential and mutually exclusive states at the same time.

The three interpretations of control have somewhat different views of transitions. The one-hot implementation uses transitions that are essentially handshakes between logically adjacent states. This characteristic can be portrayed by a "token-passing-machine", with provisions made for the controlled splitting and recombination of tokens (FORK and JOIN). In a transition between state A and state B, state A will first set state B and then state B will reset state A. Consider the case (figure

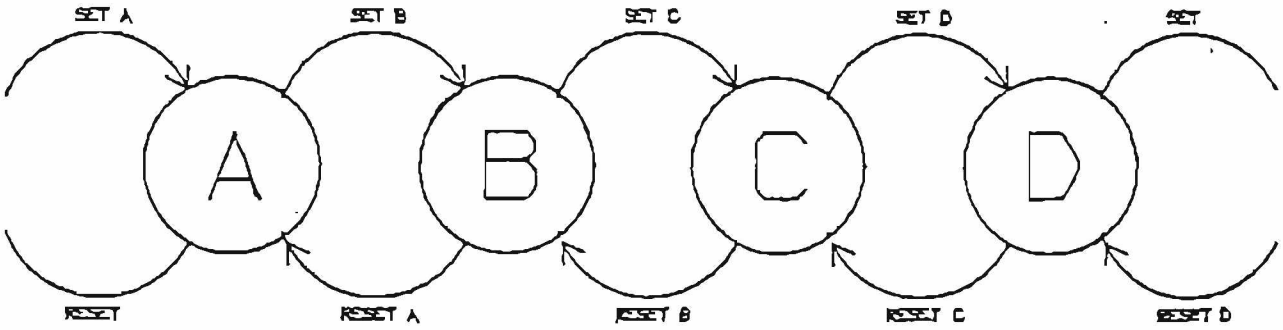

Figure 3–1: Handshaking States

3–1) where a machine contains four sequential states, A, B, C and D. Assume the machine is currently in state B. If a transition is executed, moving from state B to state C, both states B and C will be on during the time it takes state C to reset state B. Now, consider what happens if the transition from state C to state D can occur immediately after state C is set. If state C can set state D and state D can reset state C before state C can reset state B, the machine will be left in a state where both states B and D are on — resulting in a malfunction.

The differences between the three semantic interpretations all center around what to do about this timing problem. In the self–timed approach, it must be guaranteed that such a malfunction cannot occur. In order to ensure this, the state–machine must verify that each transition is complete before allowing the next one. This is done by imposing an additional condition on each transition. It is no longer sufficient just to be in a state for a transition to be possible. In addition, all states which could possibly cause a transition to the current state (its predecessors) must also be off. In the asynchronous approach, it is assumed that gate delays will be well enough behaved that this problem does not arise. This approach is especially naive in the context of the integrated circuit where gate delays may vary nearly an order of magnitude depending on loading. The synchronous approach tries to avoid the problem by recognizing inputs that trigger transitions only at specified times. If the clock period is of the same order as the delays in the faster gates, the problem will not be avoided. Unfortunately, the introduction of the clock necessarily slows the response of the control–unit. Of the three approaches, only the self–timed approach guarantees a control–unit which cannot malfunction due to internal timing problems.

Looking from inside the control–unit, there are two types of outputs. The first is the ephemeral or gated output. It is turned on only while the appropriate condition is met. The second is the enduring or latched output. This type of output is controlled by setting or resetting a latch and therefore its level is maintained even after the appropriate condition has disappeared. It is possible, however, to place a latched output in a metastable condition by trying to set or reset it at the same time, so some care must be taken in working with latched outputs.

The generation of outputs from a control–unit is always conditional upon something. What we term as an unconditional output is an output that depends only on being in a particular state or only on a particular transition being executed. What we term as a conditional output depends not only on state or transition, but also on a boolean combination of input variables.

Unconditional outputs are operated on immediately upon entry into a state or upon the execution of a transition. Also, ephemeral outputs which are unconditionally operated on from a state or transition must be released when the state is left or the transition is completed.

Conditional outputs are operated on when the entire condition becomes true, including entry to a state or execution of the appropriate transition. Again, ephemeral outputs which are conditionally operated on from a state or transition must be released when either the boolean condition is no longer met or the state is left or the transition is completed.

Because of the handshake going on between logically adjacent states, there is a small amount of time when the machine is legally in both states at the same time. This allows for ephemeral outputs to be ORed in a glitch-free manner between logically adjacent states. Enduring outputs controlled by logically adjacent states pose a problem if both a set and reset are attempted at the same time — the output latch will temporarily be placed in a metastable state, possibly adversely affecting the surrounding environment.

In ASSASSIN, there is no implicit communication between any two state-machines specified as part of the same control-unit. All such inter-state-machine communication is accomplished by explicit signalling protocols using inputs to and outputs from the state-machines.

4. The Implementation of Control

The actual physical implementation of control depends on two factors: the circuit implementation technique and the control-unit implementation technique. The circuit implementation technique should be picked so as to make the physical realization of the control-unit implementation technique as simple as possible.

The selection of a control-unit implementation technique depends on the set of features to be implemented. Thus, employing FORK and JOIN prohibits using a monolithic, completely encoded control-unit. Including FORK and JOIN in a control-unit implementation technique requires either a very complex strategy for splitting out the concurrent sections of the control into physically (and perhaps logically) separate sections, a partially encoded scheme where the sequential control sections are encoded and the concurrent are not, or a completely decoded machine. The one-hot implementation is a completely decoded scheme in which FORK and JOIN are easily included. The tradeoffs involved in selecting the one-hot strategy are discussed by Hollaar [4].

Basically, the one-hot strategy involves the use of one latch for each state, two gates for each transition, a latch or driver for each output, and one gate for each condition controlling conditional outputs from a given state or transition. For complex machines, the automatic full-custom layout of a one-hot control-unit could be very difficult.

Path-Programmable Logic provides a very regular structure that is particularly well suited for implementing one-hot control-units. In the mapping of control onto PPL using a one-hot encoding, a single latch is used for each state variable. Each transition maps to two PPL row segments, one to set the next state and the other to reset the current state once the next state has been set. In addition, complex boolean conditions on transitions (or on outputs) may require the introduction of temporary gates. In PPL, the AND of several inputs is detected on a single row. The OR is formed on the columns. For this reason, extra PPL columns containing temporary variables must be inserted for forming the OR terms of boolean expressions. Outputs are controlled by using a single PPL row to drive all the unconditional outputs controlled by a state or a transition. Each separate condition for controlling conditional outputs uses a single PPL row.

4.1. The Implementation of Control: Floor Plan

With the basic mapping strategy defined above, we soon see that there are many ways to specify the global organization or floor plan of the control-unit. The one selected for use in ASSASSIN was chosen because it appears to be simple. This floor plan (see figure 4-1) has the state latches, temporary variable inverters, and input inverters in a single band across the middle of the control-unit. Output latches and inverters are placed in a band across the top of the control-unit. Inputs arrive from the bottom of the control-unit and outputs are emitted from the top of the control-unit. This stacking of inputs and outputs results in a significantly smaller area and is a direct consequence of using a PPL-like structure for the circuit implementation. State transitions are generated in the

bottom half of the control–unit and boolean expressions and outputs are generated between the state latch band and the output band. It is possible to make other area optimizations in the PPL layout of one–hot control–units.

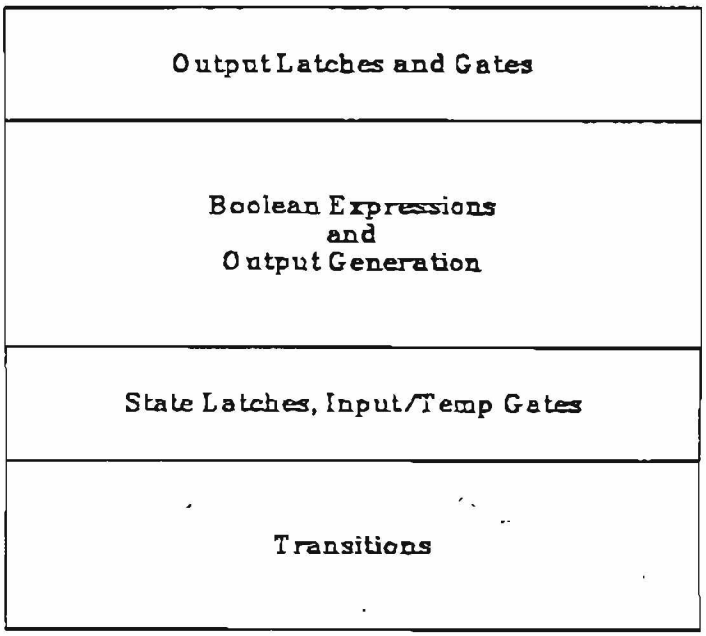

Figure 4–1: Global Organization of ASSASSIN Output

This global organization results in a simple PPL generator that needs no routing tools for constructing the control–unit. All the PPL generator has to know is which cells to place and where to place them — an easy problem when compared with routing.

4.2. The Implementation of Control: Code Generation

We have now almost fully specified the entire system. All that remains is to actually construct algorithms for generating PPL programs that implement the control–unit. The self–timed control–unit requires the use of latches for representing states. These latches must indicate their change in state after the set or reset signal has arrived. The PPL cell designed for this purpose is the four–wire latch. It contains cross–coupled NMOS inverters for the latch with inverting–buffered outputs. Thus, this cell cannot signal its change in state until after the latch has changed state. ASSASSIN can currently generate either a CIF description of the control–unit or a file written in Computervision's CADDS2 External Data Base format.

The transitions for a self–timed control–unit require two row segments. The first senses that the machine is in a certain state — say state A, that all possible predecessor states (states which could have caused a transition to state A) have been reset, and that the condition for the transition is met. If all these conditions are met, the latch for the next state is set. If there are outputs controlled by the transition, an inverter is used to appropriately control output generation from the transition. The second row segment detects that the next state has been successfully set and resets state A.

Figure 4–2 illustrates a simple transition between two states. The machine is in state B, having come from state A. State A has been reset. The first row below the state latches performs the "forward" transition, or setting of the next state. The '0' under the latch for state A detects that state A has been reset. The '1' under the latch for state B detects that state B has been set. The '1' under the inverter for input [1] detects that the input condition has been met and the 'S' under the latch for state C will set state C when the transition occurs. The second row performs the "reverse" transition,

or the resetting of the previous state. The '1' under the latch for state C detects that state C has been set and the 'R' under the latch for state B will reset state B when the forward transition has been completed. Completing the operations of both these rows constitutes a complete transition.

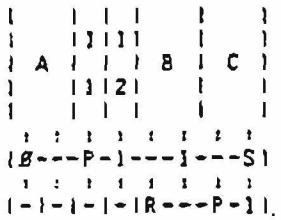

Figure 4–2: A Simple Self–Timed Transition

Asynchronous transitions are different from self–timed transitions in that they do not sense that predecessor states have been reset. If gate delays are sufficiently non–uniform, a machine constructed in the asynchronous manner would not function properly. Figure 4–3 show the same section of control as in figure 4–2, implemented asynchronously.

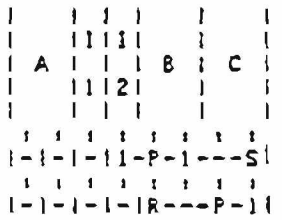

Figure 4–3: A Simple Asynchronous Transition

Synchronous transitions are implemented the same as asynchronous transitions, with the exception that the state latches are replaced by clocked flip–flops. This is illustrated in figure 4–4.

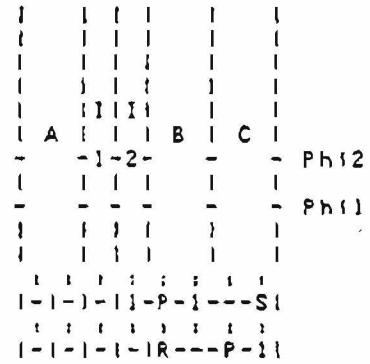

Figure 4–4: A Simple Synchronous Transition

The following discussion explains the ASSASSIN compilation of all the constructs described by Hollaar [4]. Examples are drawn from the sample control–unit whose flow–graph is contained in figure 2–3. The CUDL code for this control–unit is in figure 2–4. The complete PPL program for this example is in figure 4–5. The various constructs being discussed contain portions of this PPL program. Row segments are referred to from left to right in a given row. Row and column numbers

are as labeled in the figures.

Figure 4–6 illustrates the compilation of a move transition (from state A to state D). Rows 17 through 19 contain the state latches, input gates and temporary gates. T1 contains "I2 and not I3." T2 contains "I4 or I5." T3 indicates that the JOIN transition from states B and C to state F is currently being taken. T4 indicates that the MOVE transition from state D to state F is being taken. Row 15 is the forward transition from state A to state D. It senses that state A is active by the '1' in column 1, that "BIG" is false by the '0' in columns 2 and 3, and that state F is inactive by the '0' in column 22. State D is made active by the 'S' in column 17 and the row load is the 'P' in column 11. The reverse transition in row 14 simply senses with the '1' in column 17 that state D is active and resets state A with the 'R' in column 0.

Scale–of–two loops pose a particular problem. It is possible to get stuck in both states, with no way to get out. Scale–of–two loops therefore require some sort of mutual exclusion on transitions to avoid this problem. Figure 4–7 illustrates the compilation of a scale–of–two loop. Row 5 contains the forward transition from state D to state F. Note the '0's in columns 0 and 22 which detect the predecessors to state D. The '+' in column 18 is used in generating the outputs associated with this transition by driving T4 when the transition is in progress. The right segment on row 12 resets state D after the forward transition to state F has been finished. Note the '1' in column 19 which senses that input I8 has not yet become false. This gives the required mutual exclusion of input signals in a scale–of–two loop. Row 4 contains the forward transition from state F to state D. The '0' in column 19 detects the false state of input I6 and the other '0's detect the inactivity of the possible predecessors to state F. Row 4 contains the reverse transition associated with the transition from state F to state D. The '0' in column 15 senses that input I7 is currently false.

Figure 4–8 illustrates the FORK transition from state A to states B and C. Row 13 contains the forward FORK transition. It senses the state A is active, that state F is inactive and that input BIG is true (the '1's in columns 2 and 3). It also sets both states B and C. The reverse FORK transition is in the left segment of row 12. It detects that both states B and C have been activated and resets state A.

Figure 4–9 shows the JOIN transition from states B and C to state F. Row 9 implements the forward transition by sensing that the predecessor state (A) is inactive, states B and C are active, inputs I4, I5 and I6 are true, and by setting state F. The '+' in column 14 is used for generating the outputs associated with the JOIN transition from state C. The reverse transition is implemented in row 8 where the activation of state F is detected and states B and C are deactivated (reset).

Figure 4–10 shows the compilation of the input boolean expression BIG = I1 and (I2 or not I3). The leftmost row segments on rows 20 and 21 (I+ –I–P1 and I+ –P–01 respectively) compile the subexpression "I2 or not I3." The '+' in column 3 generate the OR of these two rows into T1. I2 is sensed by the '1' in column 4 of row 20 and "not I3" is sensed by the '0' in column 5 of row 21. To sense "BIG", the program must contain '1's in both columns 2 and 3. To sense "not BIG" it must contain '0's in both columns 2 and 3.

Figure 4–11 shows both conditional and unconditional output generation from states and transitions. Row 22 implements the unconditional outputs controlled by state A. The '1' in column 1 senses that state A is active. The '–'s in columns 6 and 13 implement the "HOLD 01,02;" statement, the 'S' in column 17 implements the "RESET 03" statement and the 'R' in column 10 implements the "SET 04" statement. The 'S' is used to reset a LATCH2 PPL cell and the 'R' is used to set it. Rows 24 and 25 implement the conditional outputs controlled by state B. Row 24 detects the "I4 or I5" condition and HOLDs 05 and 02 and resets 04. Row 25 detects the "I3" condition and sets 03. The last row segment on row 20 (I1–P———SI) implements the unconditional output (03) controlled by the JOIN transition from states B and C to F. Row 26 implements the "if BIG then set 04" statement from the JOIN transition in state C.

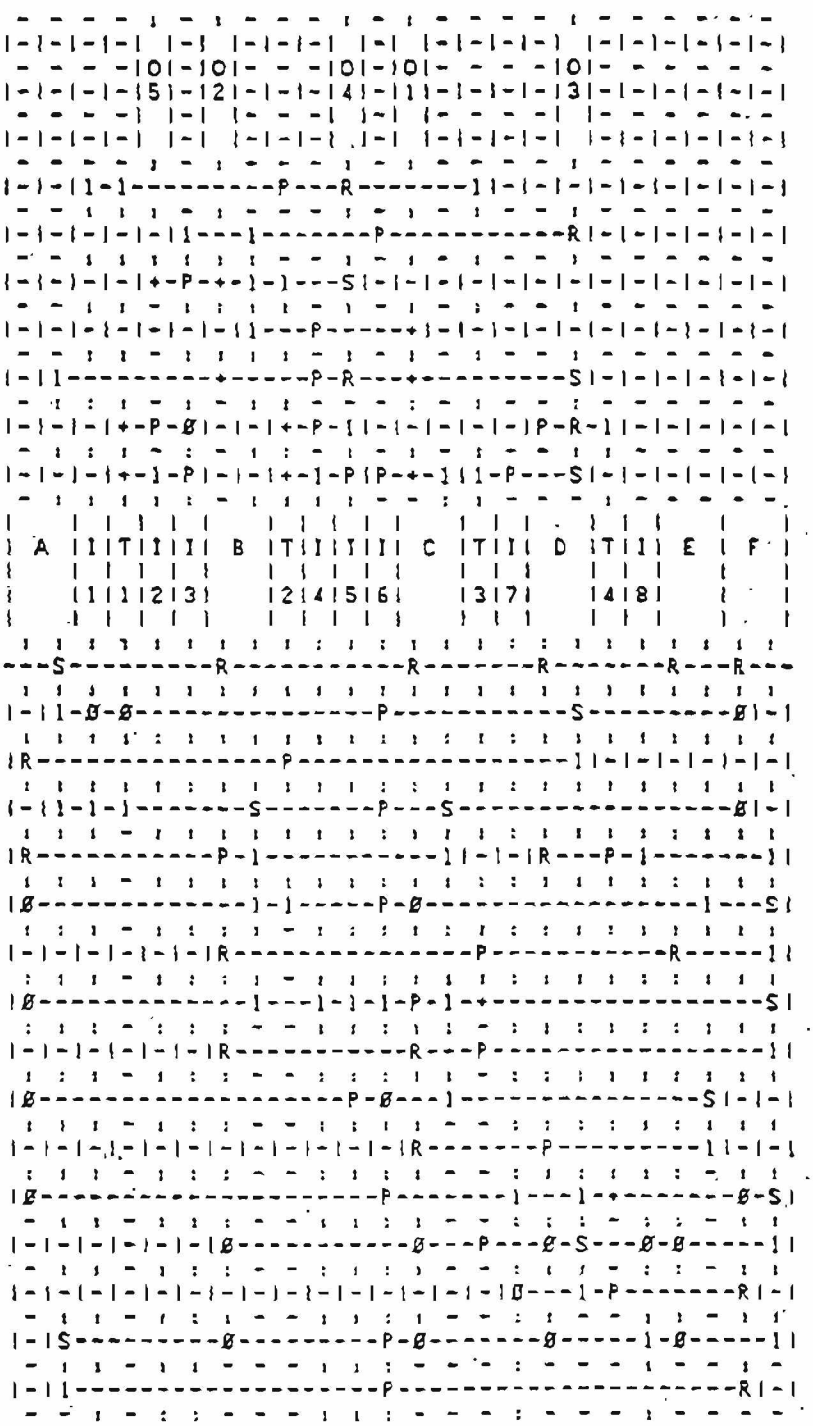

Figure 4–5: Sample PPL Program

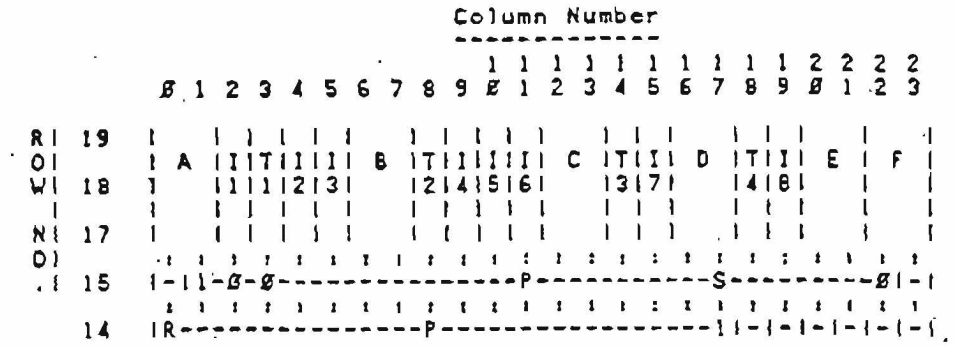

Figure 4–6: Compilation of the MOVE Transition

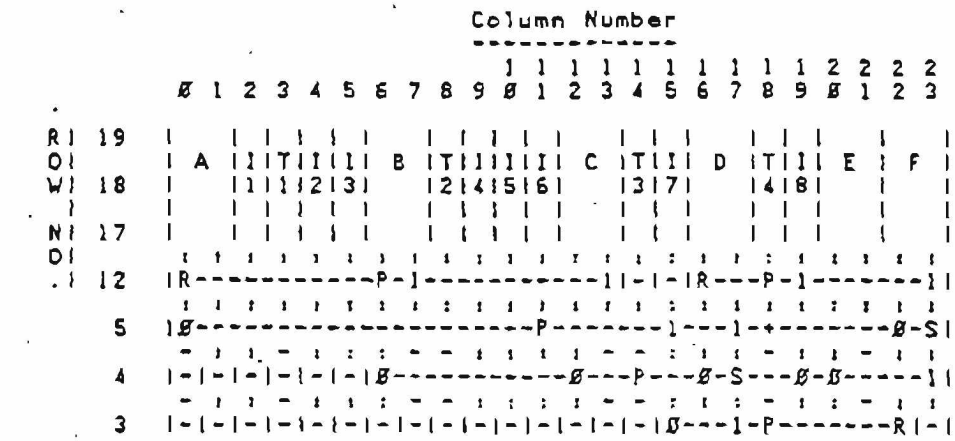

Figure 4–7: Compilation of the Scale–of–Two Loop

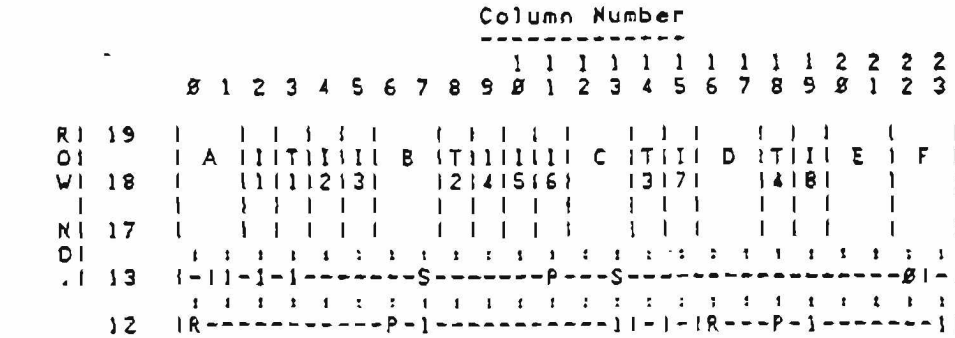

Figure 4–8: Compilation of the FORK Transition

5. The Assassination of a Control Unit

This section illustrates the complete design of a non–trivial state–machine. The control–unit to be designed comes from the Ada–to–Silicon Project underway at the University of Utah. This project has as one of its objectives the automatic transformation of Ada programs into hardware implementations using integrated circuits [5]. The Ada–to–Silicon project is using the Internet Protocol (see

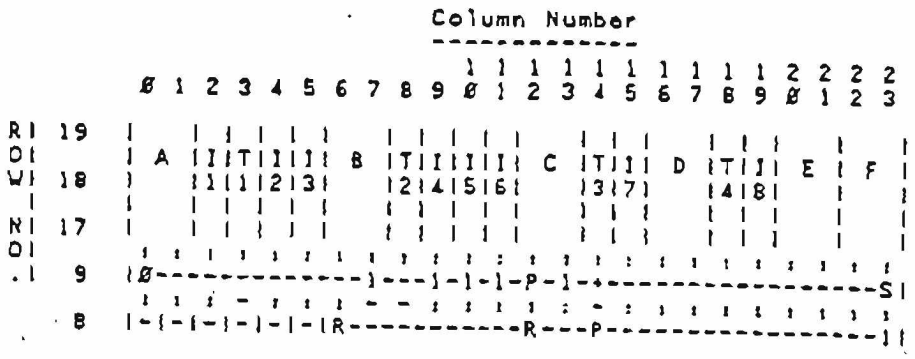

Figure 4–9: Compilation of the JOIN Transition

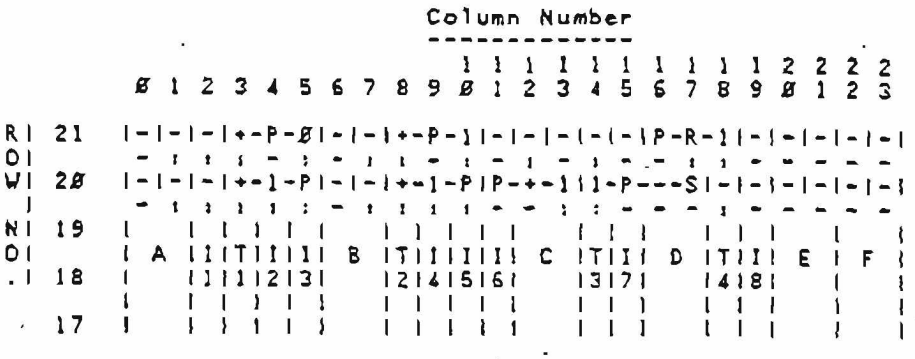

Figure 4–10: Compilation of Boolean Expressions – BIG

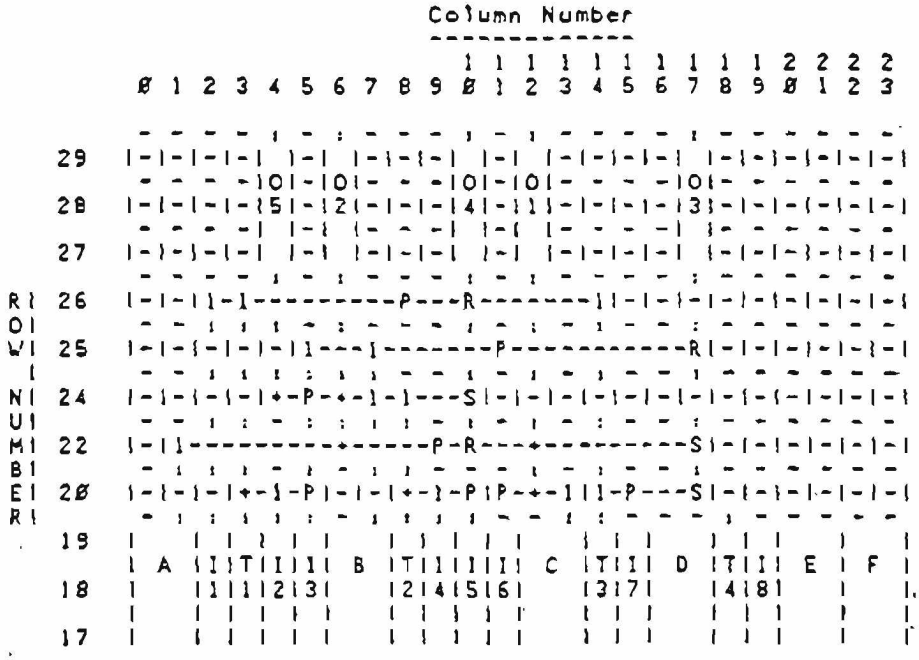

Figure 4–11: Compilation of Outputs

Postel [6]) as a test vehicle. The Internet Protocol has been decomposed into three communicating hardware (and software) submodules [5]. Figure 5–1 illustrates this division. The protocol consists

of N INM_IN submodules, each of which receives transmitted data and assembles datagrams from a single local area network, N INM_OUT submodules, each of which appropriately fragments and transmits datagrams to a single local area network, and a single INM_SRV submodule that interfaces the N INM_OUT and N INM_IN submodules to one or more host computers. The complete Ada code describing the INM_OUT submodule has been written and compiled and willis presented in a forthcoming report.

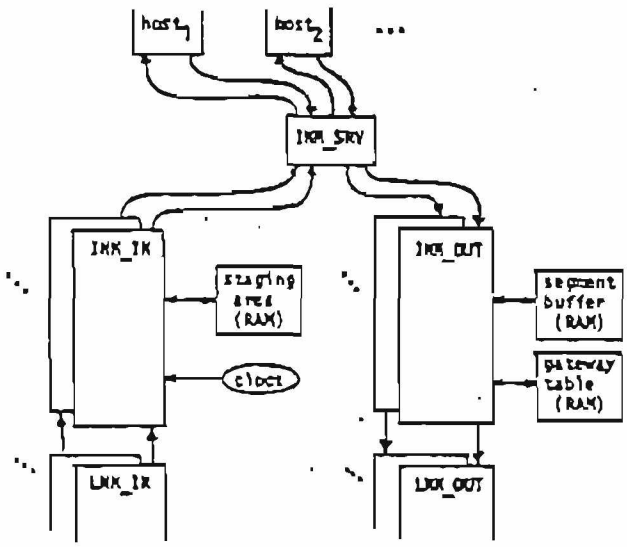

Figure 5–1: Internet Protocol Hardware Submodules

The INM_OUT submodule of the Internet Protocol has been selected as the initial test case. Preliminary Ada code in the form of a complete task has been written and compiled. INM_OUT consists of three separate tasks, Main, Read_Init_Parameters and Translate_TOS_Table. Of these, the hardware architectural design has been completed for the Read_Init_Parameters task. Read_Init_Parameters deals with the initialization parameters of INM_OUT and loads various registers with data related to the transmission of datagrams through a local area network. Illustrated in figure 5–3 is a block diagram of the hardware implementation of this task. Professor Al Davis performed the mapping of the initial version of Ada code into a block diagram. Several modifications have been made since that time. The block marked "Read_Init_Pars – FSM" is the control–unit derived from the Ada code for the Read_Init_Parameters task. Figure 5–2 contains the Ada code for a section of Read_Init_Parameters. The complete code is found in Appendix .

Figure 5–4 contains the control flow–graph for the Read_Init_Parameters task as extracted from the Ada program. It should be noted that this particular flow–graph does not use the FORK and JOIN transitions available in CUDL. Indeed, FORK and JOIN will probably not be used in implementing tasking, but may be used for more fine grained parallelism based on data independency. Ada accept statements are translated into request–acknowledge handshakes with the appropriate module. These are indicated by the name of the accept (GO or SRV) concatenated with ".REQ" and ".ACK". State RIP0 is the initial state of the machine and sends initialization signals to several of the datapath modules in the environment of Read_Init_Parameters. Of particular interest, the signal INITNUM.REG.LOD is held during this state. This signal indicates to the register holding the initialization number to watch the associated three–wire bus and assume its value at all times. When this signal is dropped (in state RIP1), this register latches the value on the bus. The first accept statement ("accept GO(...) ") is begun with the transition from state RIP0 to state RIP1.

```
begin
  loop
    accept Go(  -- Get OP Code from Main (size of Memory Address)
        init_num_formal:            bit3;
        response:               out out_response)
      do
        response := sent_ok;                              -- Also means init_ok.
        for index in 1 .. init_num_formal
        loop
          accept Srv_req( -- Process Memory Address
              server_command_datum:      srv_command;
              response_to_server:    out out_response)
          do
            Memory_request( -- Put chunk out to the Memory module.
                request_type_formal      => load_address,
                chunk_of_address_formal  => server_command_datum,
                octet_formal             => dont_care_octet);
          end Srv_req;
        end loop;

        -- Get the 8 individual initialization parameters (contained in the
        -- next 8 octets received) from the Memory Module.
        for index in 0 .. 7
        loop

          Memory_request(
              request_type_formal      => receive_datum_octet,
              chunk_of_address_formal  => dont_care_X_datum,
              octet_formal             => octet_register);

          case index is
            when 1 => lna_max_packet_lo          := octet_register; -- 8 bits
            when 2 => lna_max_packet_hi          := octet_register; -- 8 bits
            when 3 => lna_address_length         := octet_register; -- 8 bits
            when 4 => lna_time_out_lo            := octet_register; -- 8 bits
            when 5 => lna_time_out_hi            := octet_register; -- 8 bits
            when 6 => ack_type                   := octet_register; -- 1 bit
            when 7 => local_net_type_of_service_table_row_size
                                                 := octet_register; -- 3 bits
            when 8 => number_of_local_net_types_of_service
                                                 := octet_register; -- 3 bits
          end case;
        end loop;

        -- Read in type-of-service translation table.

        declare
          row_number: integer range 0 .. number_of_local_net_types_of_service;
          col_number: integer range 0 .. local_net_type_of_service_row_size;

          index:      integer range 0 .. number_of_local_net_types_of_service
                                       * local_net_type_of_service_row_size
                                       := 0;
        begin
          row_number := 0;
          loop                      -- Outer loop reads all rows of TOS table.
            col_number := 0;
            loop                    -- Inner loop reads in one row of TOS table.
              Memory_request(
                  request_type_formal      => receive_datum_octet,
                  chunk_of_address_formal  => dont_care_X_datum,
                  octet_formal             => tos_table(index));
              col_number := col_number + 1;
              exit when col_number = local_net_type_of_service_row_size;
              index := index + 1;
              if index > max_tos_table_size then
                response := bad_srv_command;
                return;             -- Exit the current accept statement.
              end if;
            end loop;               -- End inner loop.

            row_number := row_number + 1;
            exit when row_number = number_of_types_of_service;
          end loop;                 -- End outer loop.
        end;                        -- End declare block.
      end Go;                       -- End of init processing.
  end loop;                         -- End of outer-most (infinite) loop.
end Read_Init_Parameters;
```

Figure 5-2: ADA Code for Read_Init_Parameters

Note that the condition for the transition includes, in addition to GO.REQ, INITNUM.REG.DON and INITNUM.CTR.DON. The machine cannot proceed until it is sure that the initialization number register contains the correct value and the associated counter has been reset. In state RIP1, the machine begins the second accept loop. When the SRV.REQ signal arrives, a transition is made to

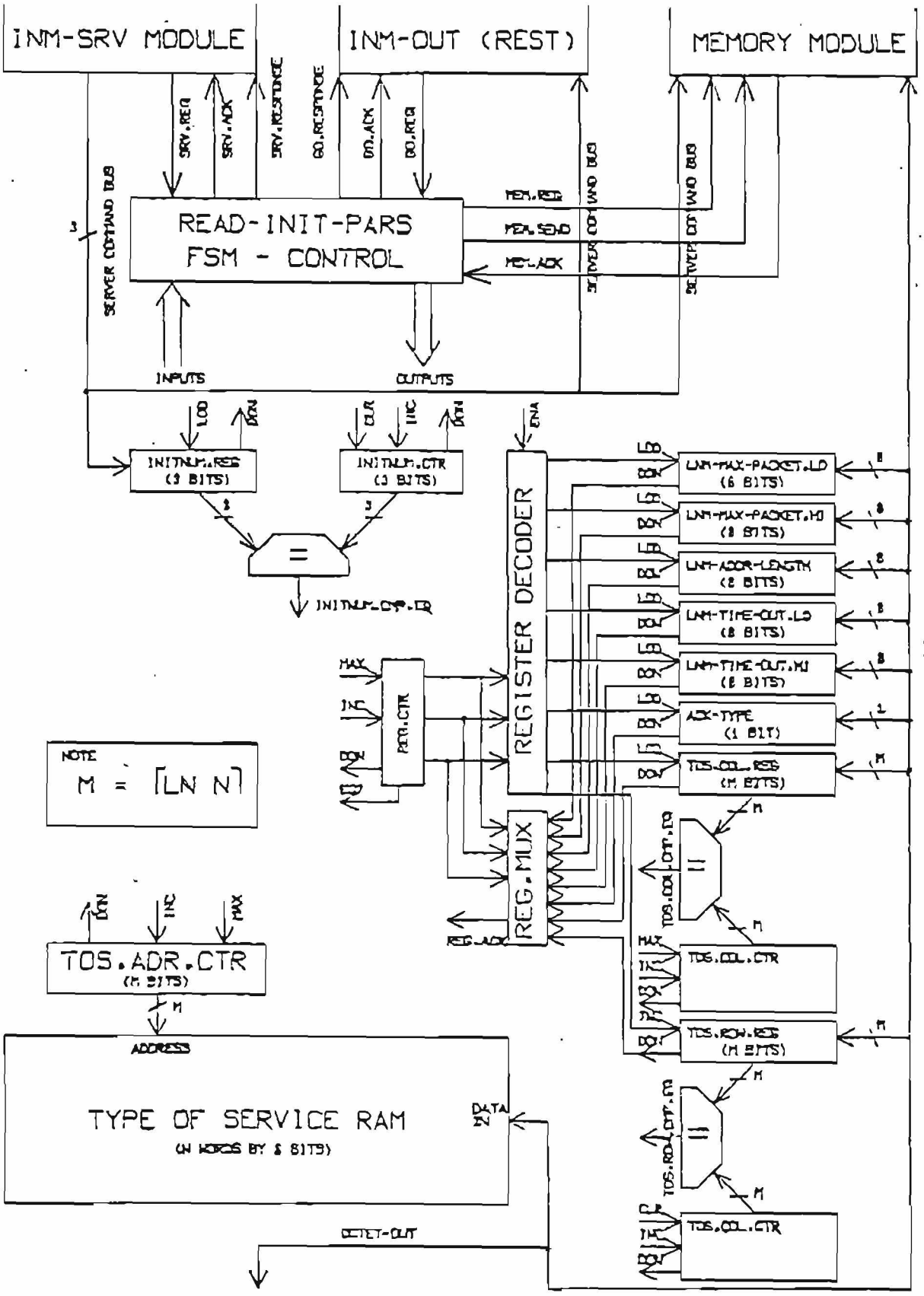

Figure 5-3: Block Diagram of Read_ Init_ Parameters

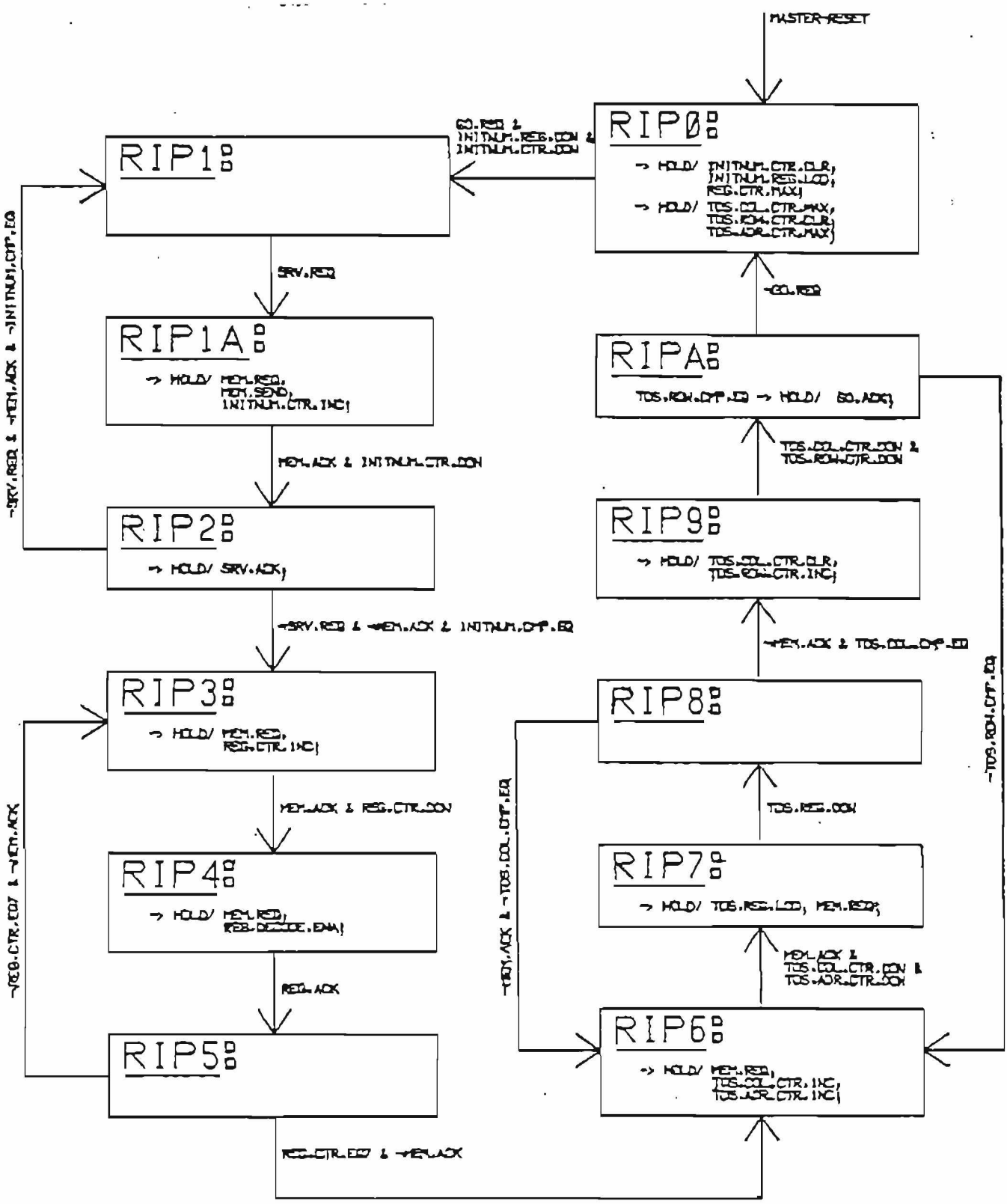

Figure 5-4: Control Flow-Graph for Read_ Init_ Parameters

state RIP2, where the counter is incremented (indicating that another byte of address is to be transmitted to the memory module), and a request–acknowledge handshake is performed between READ_INIT_PARS and the memory module. The signal MEM.SEND indicates to the memory that it is to receive data. When the counter has been incremented (INITNUM.CTR.DON) and an acknowledgement from the memory (MEM.ACK) have been received, a transition is made to state RIP2. State RIP2 terminates the handshake with the INM_SRV module by asserting the signal SRV.ACK. Once both SRV.REQ and MEM.ACK have been lowered, the output of the comparator between the initialization counter and register is examined. One of the two transitions from state RIP2 is executed based on the value of INITNUM.CMP.EQ. If INITNUM.CMP.EQ is on, the initialization loop is terminated. If it is off, the initialization loop is continued.

The memory module now has the complete address of the parameter block which needs to be transmitted to INM_OUT. State RIP3 begins an interaction between the memory module and Read_Init_Parameters that loads a set of registers appropriately. The handshake with the memory module is begun by holding MEM.REQ. At the same time, the register counter (which was initialized to 7) is incremented (and is now 0). When an acknowledgement is received from the memory (MEM.ACK), and the register counter is finished counting up by one, a transition is made to state RIP4 where the signal REG.DECODE.ENA signals the appropriate latch to gate in the value from the memory bus. MEM.REQ is left on here so that the valid data on the memory bus does not disappear before it can be latched. When the appropriate register signals that it has the data loaded (REG.ACK), a transition is made to state RIP5. When the memory acknowledges the termination of a transmission cycle (not MEM.ACK), a comparator with the register counter is made to see if all required registers have been loaded (REG.CTR.EQ7). If not, the loop is repeated, incrementing the register counter each time. If so, a transition is made to state RIP6 and the processing of the Type–of–Service (TOS) table is performed.

The type–of–service table is to be a linear array of registers (or ram cells), indexed by row and column. Initially this indexing was done via a multiplication (in the Ada code). It was replaced with a doubly nested loop to make the hardware implementation easier and more straightforward. In state RIP6, the type–of–service column counter and type–of–service address counter are incremented. They were initialized to their maximum value in state RIP0. At the same time, a handshake with the memory module is begun (by raising MEM.REQ) When the memory has placed the data on the line and replied by using MEM.ACK, and when the two counters, TOS.COL.CTR and TOS.ADR.CTR have been incremented, a transition is made to state RIP7. Here the TOS table is signalled to load the value from the memory bus (TOS.REG.LOD). MEM.REQ is held high so that the data on the memory bus remains valid. When the data is in the TOS table, TOS.REG.DON is asserted and the next state becomes RIP8. This state terminates the handshake with the memory module. When the acknowledgement from the memory arrives, if all columns in the current TOS table entry have been processed, a transition is made to state RIP9 to proceed to the next TOS table entry. If more columns in the entry need to be processed, the TOS.COL.CMP.EQ signal will be false and the transition from state RIP8 to state RIP6 will be taken.

In state RIP9, the column counter (TOS.COL.CTR) is cleared and the row counter (TOS.ROW.CTR) is incremented. When these two operations are complete, the next state becomes RIPA where a check is performed to see of the entire TOS table has been loaded. If it has not, TOS.ROW.CMP.EQ will be false and the a transition occurs from state RIPA to state RIP6. If TOS.ROW.CMP.EQ is true, the output GO.ACK is asserted, terminating the "Accept GO (...)" statement. When GO.REQ is lowered, the next state becomes RIP0 to begin over again when necessary. Figure 5–5 contains the CUDL code for the Read_Init_Parameters state machine.

The CUDL code in figure 5–5 was run through ASSASSIN. The code was simulated to verify that it matched the flow–graph; the associated PPL program was then generated through compilation of the CUDL code. Figure 5–6 contains a plot of the PPL program for the Read_Init_Parameters control.

```
ControlUnit ReadInitParms;

 StateMachine RIP;

   StartState RIP0;
     moveon GO_Req and (InitNum_REG_DON and InitNum_CTR_DON) to RIP1;
     hold InitNum_CTR_CLR, InitNum_REG_LOD, REG_CTR_MAX;
     hold TOS_Col_CTR_MAX, TOS_Row_CTR_CLR, TOS_ADR_CTR_MAX;
   end;

   state RIP1:
     moveon SRV_Req to RIP1A;
   end;

   state RIP1A:
     moveon MEM_Ack and InitNum_CTR_DON to RIP2;
     hold MEM_Req, MEM_Send, InitNum_CTR_INC;
     set GO_Response;
   end;

   state RIP2:
     moveon not SRV_Req and (not MEM_Ack and     InitNum_CMP_EQ) to RIP3;
     moveon not SRV_Req and (not MEM_Ack and not InitNum_CMP_EQ) to RIP1;
     hold SRV_Ack;
   end;

   state RIP3:
     moveon MEM_Ack and Reg_CTR_DON to RIP4;
     hold MEM_Req, Reg_CTR_INC;
   end;

   state RIP4:
     moveon Reg_ACK to Rip5;
     hold MEM_Req, Reg_Decode_ENA;
   end;

   state RIP5:
     moveon     Reg_CTR_EQ7 and not MEM_Ack to RIP6;
     moveon not Reg_CTR_EQ7 and not MEM_Ack to RIP3;
   end;

   state RIP6:
     moveon MEM_Ack and (TOS_Col_CTR_DON and TOS_Adr_CTR_DON) to RIP7;
     hold MEM_Req, TOS_Col_CTR_INC, TOS_Adr_CTR_INC;
   end;

   state RIP7:
     moveon TOS_Reg_DON to RIP8;
     hold TOS_Reg_LOD, MEM_Req;
   end;

   state RIP8:
     moveon not MEM_Ack and     TOS_Col_CMP_EQ to RIP9;
     moveon not MEM_Ack and not TOS_Col_CMP_EQ to RIP6;
   end;

   state RIP9:
     moveon TOS_Col_CTR_DON and TOS_Row_CTR_DON to RIPA;
     hold TOS_Col_CTR_MAX, TOS_Row_CTR_INC;
   end;

   state RIPA:
     moveon not TOS_Row_CMP_EQ to RIP6;
     moveon not GO_Req to RIP0;
     if TOS_Row_CMP_EQ then hold GO_Ack;
   end;
 end;
end.
```

Figure 5-6: CUDL Code for Read_ Init_ Parameters Control

Figure 5-7 shows the composite layout.

The compilation of the control unit took approximately 2 minutes of DEC-System 20 CPU time. The resulting circuit is 2028 microns by 1050 microns (39 PPL columns by 30 PPL rows using 6-micron geometry). The datapath related to the Read_ Init_ Parameters task cannot be layed out until the relationship of some of the registers, which represent global variables (with respect to

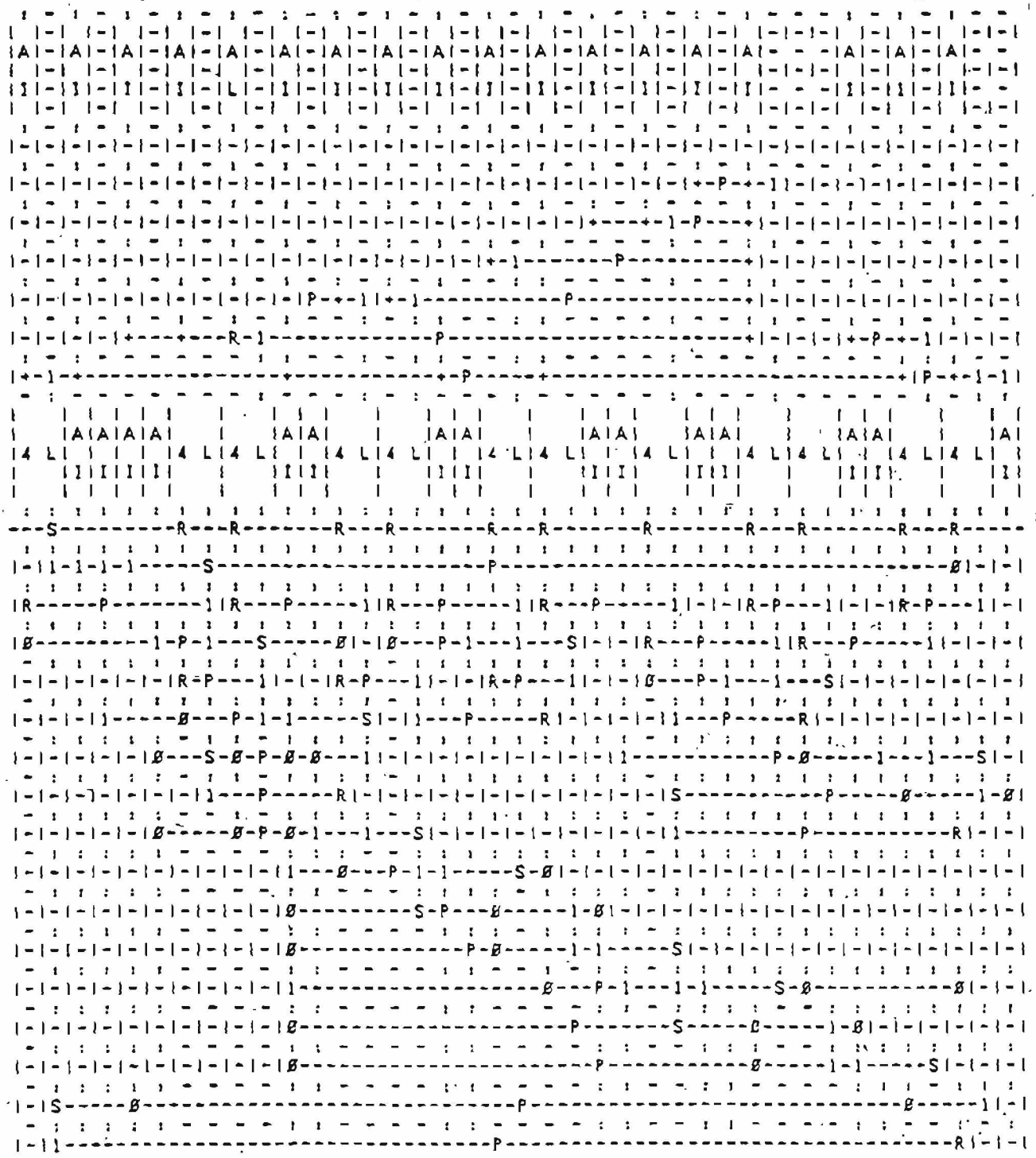

Figure 5-6: PPL Program for Read_ Init_ Parameters Control

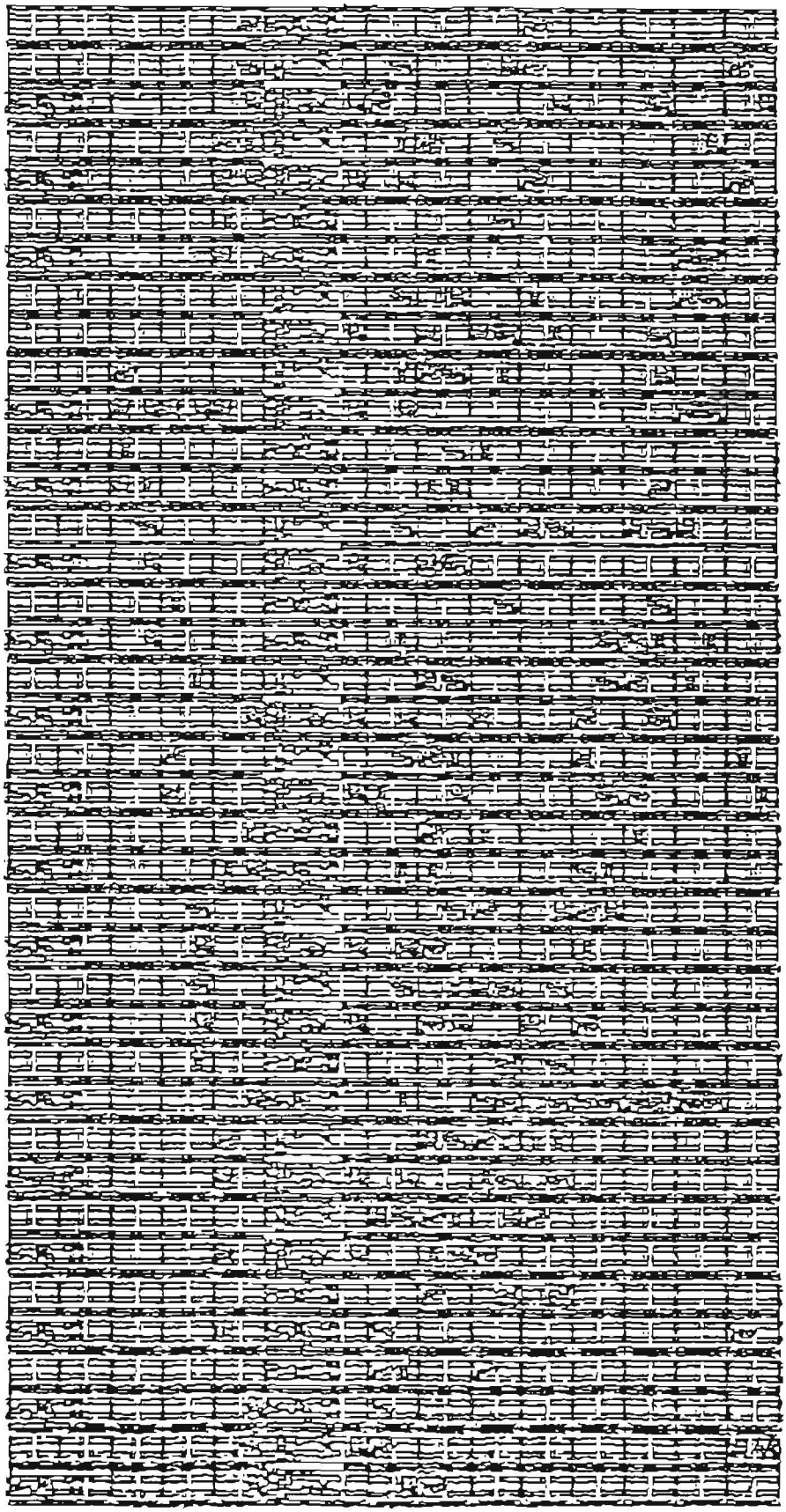

Figure 5–7: Composite Layout (NMOS) for Read_ Init_ Parameters Control

Read_ Init_ Parameters), with other associated control and datapath elements has been established.

6. Conclusions

ASSASSIN demonstrates several significant points.

1. Control can be specified at an abstract level and then automatically and easily implemented as an integrated circuit module. It is possible to map control specified at even higher levels of abstraction to something ASSASSIN understands, thereby enabling us to make progress toward a true silicon compiler. Such work is reported in [11].
2. Self-timed (or asynchronous) control-units with concurrency can be easily implemented. ASSASSIN shows that the control for self-timed machines can be designed with relative ease.
3. The successful use of Path-Programmable Logic in ASSASSIN shows that PPL has great value as a circuit implementation technique, at least for this type of control-unit. This also shows that PPL is indeed amenable to the development of sophisticated CAD tools that use it as the underlying circuit implementation technique.
4. The mapping of Ada's rich set of control constructs is very straightforward as illustrated by the generation of the control for the Read_ Init_ Parameters task. ASSASSIN represents a step forward in the design of integrated circuits by allowing high level descriptions of integrated circuit modules to be automatically compiled to a layout.

7. Acknowledgements

I would like to acknowledge the help of Dr. Lee Hollaar without whose help and encouragement this work would not have been done. The work of Dr. Kent Smith on the development of PPL is greatly appreciated. Acknowledgements are also due to Dr. Elliott Organick, Dr. Alan Davis, Dr. Gary Lindstrom, and Dr. Alan Hayes for work on development of the example of the transformation of Ada to a hardware module.

I The Syntax for ASSASSIN

The following is a BNF description of CUDL – the Control Unit Description Language. The following are to be used in understanding this description:

```
<>    - a non-terminal symbol
{}    - 0 or more repetitions
:=    - is defined as
|     - OR

language terminals are indicated by uppercase
```

```
<control-unit>          := CONTROLUNIT <identifier> :
                           {<input-descriptor>} <sm-list> END .

<identifier>            := <letter> <id-tail>

<id-tail>               := <letter> <id-tail> | <digit> <id-tail> |
                           <letter> | <digit>

<input-descriptor>      := INPUTS: <input-reduction-list>

<input-reduction-list>:= <reduction-statement>
                               <input-reduction-list> |
                           <reduction-statement>

<reduction-statement>   := <identifier> := <condition> ;

<condition>             := <term> OR <condition> | <term>

<term>                  := <primary> | <primary> AND <term>

<primary>               := <identifier> | (<condition>) |
                           NOT <primary> | TRUE | FALSE

<sm-list>               := <sm-descriptor> |
                           <sm-descriptor> <sm-list>

<sm-descriptor>         := <sm-type> STATEMACHINE <identifier> :
                           <state-list> END ;

<sm-type>               := SELFTIMED | ASYNCHRONOUS | SYNCHRONOUS

<state-list>            := <state-descriptor> |
                           <state-descriptor> <state-list>

<state-descriptor>      := STARTSTATE <state-name> :
                                <statement-list> END ; |
                           STATE <state-name> :
                                <statement-list> END ;

<state-name-list>       := <state-name> , <state-name-list> |
                           <state-name>

<state-name>            := <identifier>

<statement-list>        := <statement> ; <statement-list> |
                           <statement>

<statement>             := <transition-statement> |
                           <action-statement>
```

```
<transition-statement>:= <transition-op> <transition>

<transition-op>       := MOVEON | FORKON |
                         JOINS <state-name-list> ON

<transition>          := <condition> TO <state-name-list> ; |
                         <condition> TO <state-name-list>
                              DOING <action-statement-list> ;

<action-statement-list>:= <action-statement> |
                          BEGIN {<action-statement> ;} END

<action-statement>    := <action-op> <output-list> |
                         <if-action-statement>

<action-op>           := HOLD | SET | RESET

<output-list>         := <output-name> , <output-list> |
                         <output-name>

<output-name>         := <identifier>

<if-action-statement> := IF <condition> THEN
                              <action-statement-list>;
```

II. Ada Code for the Read_Init_Parameters Task of the INM_OUT Submodule

```
separate (Inm_Out_Module)

task body Read_Init_Parameters is

  -- Accessed globals:
  --------------------
  -- number_of_local_net_types_of_service:          octet_type
  -- local_net_type_of_service_table_row_size:      octet_type
  -- tos_table:                                     octet_buffer_type

  -- Local variable declaration:
  ------------------------------
  -- The following variable is commented out. It appeared only in the
  -- "high-level" used to read in the TOS table.  See below.
  --   number_of_tos_table_octets: integer range 2 .. max_tos_table_size - 1;
  octet_register:                 octet_type;

begin
  loop
    accept Go(
        init_num_formal:       bit3;
        response:          out out_response)
      do
        response := sent_ok;                              -- Also means init_ok.

        -- Get from the server all of the addr_chunks needed to form the
        -- base address in memory that holds the initialization parameters
        -- and sends these chunks to the Memory module.
        for index in 1 .. init_num_formal
        loop
          accept Srv_req(                                 -- Get next address
                                                          -- chunk from the
                                                          -- Server Module.
              server_command_datum:     srv_command;
              response_to_server:   out out_response)
          do
            Memory_request(                               -- Put chunk out to
                                                          -- the Memory module.
                request_type_formal      => load_address,
                chunk_of_address_formal  => server_command_datum,
                octet_formal             => dont_care_octet);
          end Srv_req;
```

```
        end loop;

        -- Get the 8 individual initialization parameters (contained in the
        -- next 8 octets received) from the Memory Module.
        for index in 1 .. 8
        loop

          Memory_request(
              request_type_formal      => receive_datum_octet,
              chunk_of_address_formal  => dont_care_X_datum,
              octet_formal             => octet_register);

          case index is
            when 1 => lnm_max_packet.lo          := octet_register;
            when 2 => lnm_max_packet.hi          := octet_register;
            when 3 => lnm_address_length         := octet_register;
            when 4 => lnm_time_out.lo            := octet_register;
            when 5 => lnm_time_out.hi            := octet_register;
            when 6 => ack_type                   := octet_register;
            when 7 => local_net_type_of_service_table_row_size
                                                 := octet_register;
            when 8 => number_of_local_net_types_of_service
                                                 := octet_register;
          end case;
        end loop;

        -- Read in type of service translation table.

        declare
          row_number: integer range
                          0 .. number_of_local_net_types_of_service;
          col_number: integer range
                          0 .. local_net_type_of_service_row_size;

          index:      integer range
                          0 .. number_of_local_net_types_of_service
                             * local_net_type_of_service_row_size
                          := 0;
        begin
          row_number := 0;
          loop                      -- Outer loop reads all rows of TOS table.
            col_number := 0;
            loop                    -- Inner loop reads in a row of TOS table.
              Memory_request(
                  request_type_formal     => receive_datum_octet,
                  chunk_of_address_formal => dont_care_X_datum,
                  octet_formal            => tos_table(index));

              col_number := col_number + 1;
              exit when col_number = local_net_type_of_service_row_size;

              index := index + 1;
              if  index > max_tos_table_size  then
                response := bad_srv_command;
                return;             -- Exit the current accept statement.
              end if;
            end loop;               -- End inner loop.

            row_number := row_number + 1;
            exit when row_number = number_of_types_of_service;
          end loop;                 -- End outer loop.
        end;                        -- End declare block.

      end Go;                                   -- End of init processing.

    end loop;                                   -- End of outer-most (infinite)
                                                -- loop.
end Read_Init_Parameters;
```

References

1. T. M. Carter, "ASSASSIN: An Assembly, Specification and Analysis System for Speed-Independent Control-Unit Design in Integrated Circuits Using PPL," Master's thesis, Department of Computer Science, University of Utah, June 1982.

2. A. L. Davis and P. J. Drongowski, "Dataflow Computers: A Tutorial and Survey," Computer Science Department Technical Report UUCS-80-109, University of Utah, Jul. 1980.

3. S. Hiyamizu et al., "Extremely High Mobility of Two-Dimensional Electron Gas in Selectively Doped GaAs/N-AlGaAs Heterojunction Structures Grown by MBE," *Japanese Journal of Applied Physics*, Vol. 20, No. 4, Apr. 1981, pp. L245-L248.

4. L. A. Hollaar, "Direct Implementation of Asynchronous Control Units", Submitted to IEEE Transactions on Computers, to be published December 1982

5. Organick, E. I., and Lindstrom, G., "Mapping high-order language units into VLSI structures," *Proc. COMPCON 82*, IEEE, Feb. 1982, pp. 15-18.

6. Postel, Jon: editor, "Internet Protocol: DARPA Internet Program, Protocol Specification," Tech. report RFC 791, Information Sciences Institute, USC, Sept. 1981.

7. C. L. Seitz, *System Timing*, Addison-Wesley, Reading MA, 1980, pp. 218-262, .

8. J. H. Shelly, "Design of Speed-Independent Circuits," File 226, UIDCL, July 1957.

9. H. E. Shrobe, "The Data Path Generator," *Digest of Papers: CompCon Spring 82*, IEEE Computer Society, 1982, pp. 340-344.

10. K. F. Smith; T. M. Carter; and C. E. Hunt, "Structured Logic Design of Integrated Circuits Using the Stored Logic Array," *IEEE Transactions on Electron Devices*, Vol. ED-29, No. 4, April 1982, pp. 765-776.

11. P. A. Subrahmanyam, "Automated Design of VLSI Architectures: Some Preliminary Explorations", Draft Version