ASM CHARTS IN VHDL

Dr. R. Coowar
University of Central Florida
Engineering Technology Department
P.O. Box 162450, Orlando FL 32816-2450
coowar@mail.ucf.edu

Dr. H.v.d.Biggelaar
Professor Emeritus
University of Massachusetts Dartmouth
Electrial & Computer Enginering Dept.

ABSTRACT

Although state diagrams tend to be very popular in state machine design, be it in conjunction with schematic capture or HDL-type (Hardware Description Language) input, the authors have found the use of ASM (Algorithmic State Machine) charts to be more useful and instructive. This article addresses this issue and shows an illustrative example. It also discusses the various obvious as well as some of the more subtle advantages of this approach. In addition in this lab-intensive course, "Digital Systems" which follows "Introduction to Digital Logic", emphasis is put on the careful interpretation of simulation results, which otherwise the students either ignore or at best treat rather nonchalantly. The example used in this article illustrates both aspects.

INTRODUCTION

The lab for this course is equipped with twenty workstations and a dedicated server. Each workstation is equipped with a Pentium IV (1.0 GHz), 256 MB of RAM, a 40 GB hard drive, a CD ROM read/write drive and a floppy drive. A university-wide antivirus program is maintained on the machines since the students may bring their designs stored on floppies to the lab. The textbook (Brown & Vranesic) contains a student copy of the software, which is amply sufficient for the designs done in this course, as well in it's follow-up elective course "Programmable Devices". The Altera software is stored on the server and sufficient network license tokens are available to be able to use all machines concurrently as well as a client PC in the instructor's office. The software, MAX+II, is constantly updated to its latest version. It should be noted that no "MS Office" software is loaded on the machines to prevent the students from using the computers for report writing, since ample facilities for that exist elsewhere. Access to the lab is by keycard that the students registered in this course and others who use the lab may purchase for a small fee, so that they can use it any time and day outside scheduled lab hours when there are free stations.

A variety of tool suites and hardware is available. In this case the choice was Altera's MAX+II (version 10.1) and the Altera prototype board with a Flex 10k FPGA. For all the projects in the course, even small FPGAs (Field Programmable Gate Arrays) are more than adequate, so the choice of manufacturer was driven by the fact that Altera has an in-house developed and fully integrated tool suite, in which all parts work together seamlessly and there are no third parties involved. Also the excellent technical support, readily available to the instructors makes this an attractive choice. This is not to imply, however, that Max+II does not have a few annoying and unusual quirks.

STATE MACHINE DESIGN AND SIMULATION

The first state machine in the course is a very simple one and is deliberately implemented with schematic capture. The design is done by using a state diagram and state table. The latter is then used to design the driver logic for each flip-flop and simplified by using Karnaugh maps. D-type flip-flops are used instead of T-type and JK-type, for simplicity's sake as well for the more important reason that the D-type is the one used in the FPGA (target technology).

The point of this exercise is to introduce the students to the concept of state machines and to

make them aware of the steps in the implementation. Brief mention is made of Moore and Mealy machines, although the implementations often are a mixture of the two.

When implementing the design with ICs that are wired together, it is desirable to simplify the driver circuits. However, when using FPGAs, the architecture of the logic modules is such, that it is not the complexity of the driver logic that is important but that the number of inputs be limited to four or five, depending on the LUT (look-up table) in the FPGA. This is usually the first introduction of the student to the design philosophy that it is important to know the hardware or target technology in order to be able to arrive at an efficient design and is emphasized throughout the course. In any case, the modular implementation can be depicted as shown in Fig-1.

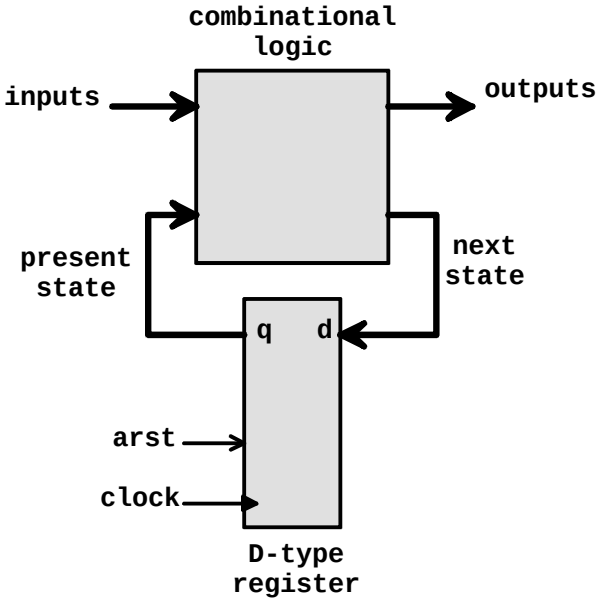

Fig. 1. Modular block diagram of a state machine.

This incidentally, also leads quite naturally to a brief mention that the combinational portion of the logic can be implemented with a ROM (read-only memory), in other words as a ROM-centered design. This only requires a mental switch in terminology, i.e. think "data-in" instead of "address". It is then pointed out that if the number of states and inputs stays the same and only the logic needs to be modified, it only requires reprogramming the ROM, which usually amounts to just editing the ROM-contents file and recompiling the design. An important point that is made at the completion of this design is the by then obvious fact that it is awkward and time-consuming to begin with and difficult to make even the smallest modifications. Changing the number of states and/or the number of inputs and outputs for instance, as well as the transition conditions, usually requires a complete redesign. Finally the design is simulated functionally as well as with full timing information included and the hardware is programmed. This last step is highly satisfying for the students as it makes the design very real. It is also used as an important criterion by the instructor to sign off on the design as "working". This is a particularly important step, since a successful timing simulation not always automatically guarantees a working design and some more troubleshooting may be necessary.

From that point on, all designs that require a state machine are done using an HDL, to wit VHDL. The software also allows the use of Verilog, so that is a matter of preference. The thrust of this article is to emphasize the advantage and ease of use of the combination of ASM Charts and VHDL. ASM Charts and State Diagrams have much in common and in fact they contain exactly the same information as do State Tables and any one can be derived from any of the others. However, the ASM Chart shows more clearly the state transitions and signal flow and has the advantage that there is practically a one-to-one correspondence with the VLDL code, as is shown in Fig-2.

Before we elaborate on one project using the above-mentioned approach, it should be pointed out that the students are made aware that there are essentially two different methods in the way the code can be written. One uses two processes, one dedicated to the clock circuitry, the other describing exclusively the state transitions. The single-process approach is one in which both parts of the state machine are combined. Although the two-process method requires a

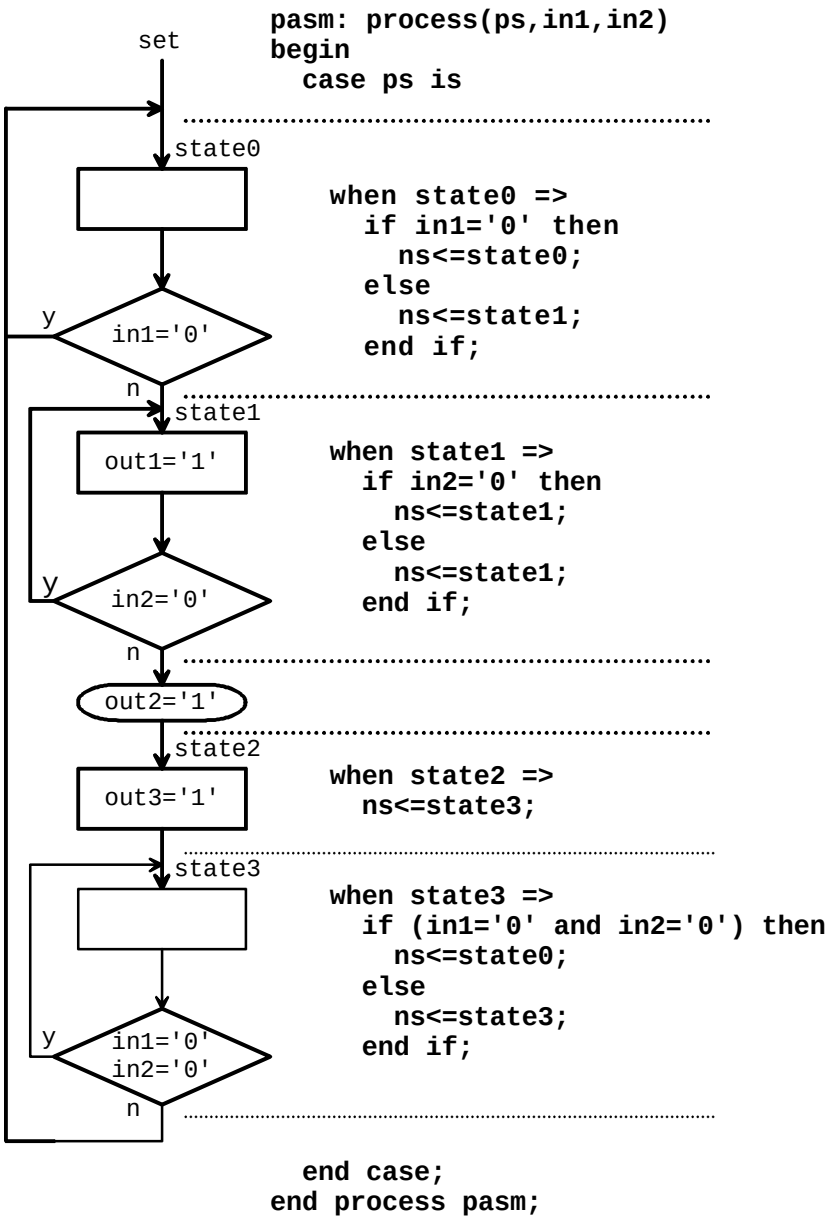

Fig. 2. State-transition section of the state machine.

little more code, it has the distinct advantage that the clock-related portion and the state transition aspects of the state machine are treated separately and thus modifications in one do not affect the other. There is also a more subtle advantage in that the principal statement in the clock process is written in the form "present-state <= next-state", which is consistent with the one that describes a D-type flip-flop, i.e. "q <= d". The student is familiar with that construct from previous simpler exercises. Additionally, the ASM chart and state diagram are all written in the form "next-state <= previous-state", which is much more intuitively obvious than in the single-process method, where they would have to be written as "present-state <= next-state". There is also a difference in the sensitivity lists. In the two-process method, the sensitivity list for the transition process clearly shows, in addition to the present state, only the inputs that directly affect the transitions, while in the clock process it only contains the clock and asynchronous inputs, such as a set or clear and enable.

Finally, all the outputs are written as separate concurrent statements, i.e. outside the processes, using the "when – else" construct. This keeps the state-related process simple and avoids difficulties with initialization of the outputs. As a bonus, the Moore and Mealy outputs are handled essentially the same ways and can be easily changed if necessary, without affecting the state transition process.

There are some further points to be made with the simulation. When the state machine arrives in state0 from state3, "out1", which is supposed to appear only in state1, shows up as a short pulse as shown in Fig-3.

Although some students may shrug this of with an "oh well", the more curious ones will want to understand the reason. First it should be pointed out that although the pulse is very short, it is long enough to be captured by a directly following latch or flip-flop. Thus it should not be ignored. This leads to a discussion of state assignments. It was necessary to address this when doing schematic capture since the state assignments have to be made explicitly in order to construct the state table. However, with the HDL method there is no mention of this issue, although it is the one that probably has the most effect on the implementation of the design. This does not mean that there are no state assignments; they just are hidden in the statement "type state is (state0, state1, state3)". They are automatically assigned the values, "00", "01", "10" and "11" respectively. When the state changes from state3 to state0, i.e. from "11" to "00", the simulation shows that the value "10" apparently is the brief interim value, which seems to represent state1 since that one has output out1.

The output with the "glitch" of 0.5ns duration is produced by the LUT (Look Up Table) that in the FPGA implements the dual-input AND gate with one input from each of the two flip-flops. Since the inputs of that gate switch from "11" to "00", there apparently is an interim state "01" or "10" briefly present on the gate. It is tempting to think that this is due to the flip-flops not changing their outputs at exactly the same time. However, even though this might be possible in the actual hardware, where the characteristics of the flip-flops and gates in general, may differ, dependent on their location in the FPGA, the flip-flops as modeled in the simulator are ideal and identical. Thus they must change at the same time unless there is clock skew. It is unreasonable to assume, however, that adjacent flip-flops, as evident from the layout in the floorplanner, there would be 0.5ns clock skew. This is further confirmed by the fact that the timing analyzer shows that the delay from clock edge to output is identical for both flip-flops. This leaves just one source, i.e. the difference in

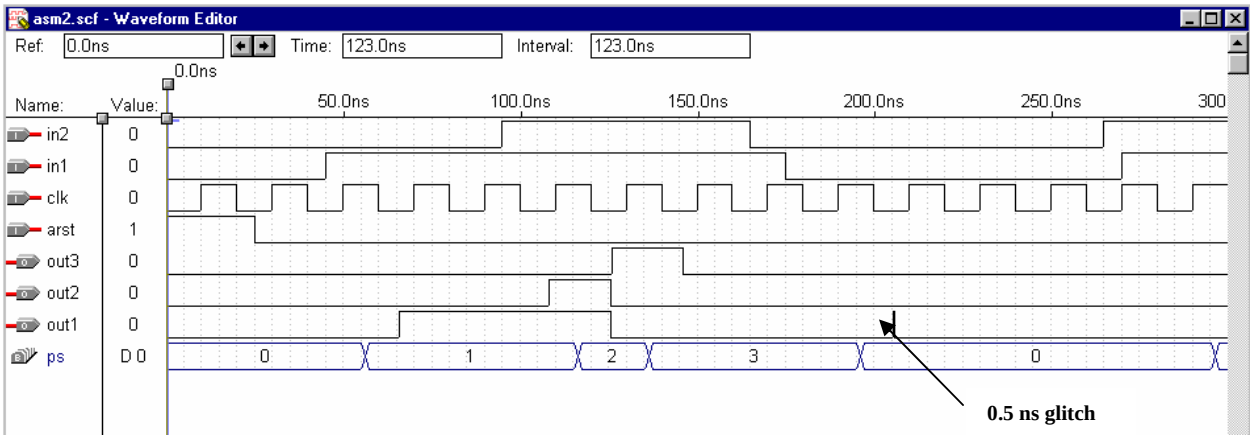

Fig. 3. Timing simulation of the entire state machine.

the interconnect delays between the flip-flops and the AND gate. To prove this, the floorplanner was used to move one of the flip-flops to a very different location, making one of the two flip-flop => AND gate connections much longer than the other. Recompilation and resimulation showed an increase in the glitch length from "0.5ns" to "1.4ns", this confirming the assumption.

It was then discussed how the design might be changed to avoid this glitch. If there are no interim "01" or "10" signal values on the AND gate, the glitch cannot appear. This can be accomplished by changing the binary sequence of the state assignments to unit-step or Gray code. All this required was a change in the "type" statement from (state0, state1, state2, state3) to (state0, state1, state3, state2). Recompilation and resimulation indeed showed no more glitch. It was then pointed out that although in this case the Gray code state assignments solved the glitch problem, that it may be unavoidable in a more complex state machine where not all the transitions are in sequence and certainly for those encoded as one-hot machines in which always two bits change per transition.

In preparation for an assignment we discuss a more complex state transition example. When there are three decision paths from a state one has the choice of two constructs both with respect to the code and the asm chart. An example of each is given in Fig. 4-a and Fig. 4-b.

The code shown in Fig. 4-a is simpler and so is the asm chart, however, it is not obvious if all possibilities are covered, in other words if there are no dangling branches, which when synthesized, would cause undesirable extra flip-flops. Also the "elsif" does not allow pairing of the "end-if" statements. In Fig. 4-b on the other hand, both the code and the asm chart section clearly show that all possibilities are covered, thus guarantying that no unnecessary flip-flops will be implemented. There are of course exceptions, for instance when the state transitions depend on two variables, but the conditions are such that one condition is the opposite of the other, thus in essence there are only two paths or choices, which then can be easily depicted and coded with a single decision box. An example is x&y and $\bar{x}+\bar{y}$ $(=\overline{x\&y})$.

Next is an example of a student assignment which uses the constructs discussed above.

"Design an alarm system for a residence that monitors the doors and windows. The alarm has five outputs: a green, an amber and a red light, a sustained note and a siren. Normally the alarm is in the ready state and shows a green light and opening and closing windows or doors has no effect. A binary code "1101" arms or disarms the system. When the system is armed, the green light goes off and the amber one goes on, however, if the correct code is entered within 30 seconds, the alarm resets to the ready state. If in the armed state, a door or window is opened, the red light goes on and there is a sustained note, indicating that the system is armed. It can then be disarmed within 30 seconds, i.e. reset to the ready state, by entering the code. Otherwise the siren goes off and to reset the system the code must be entered."

Fig. 5 shows the segment of the asm chart and the corresponding code for the state sequence.. The complete code can be found in Appendix A. The designs were implemented with a schematic as the top level, which included the timing and display modules. The Altera demo board was used to test the downloaded designs.

Regardless of the fact that the authors emphasize at the beginning of the introduction of this topic, that the state machine is strictly a controller, used to turn external modules on and off, the first returned assignments often include considerable functionality in the state machine itself. The students have then experienced that such a design is difficult to debug and modify and are more likely to appreciate and adopt the controller philosophy in their next design. Typically an external module requires as inputs

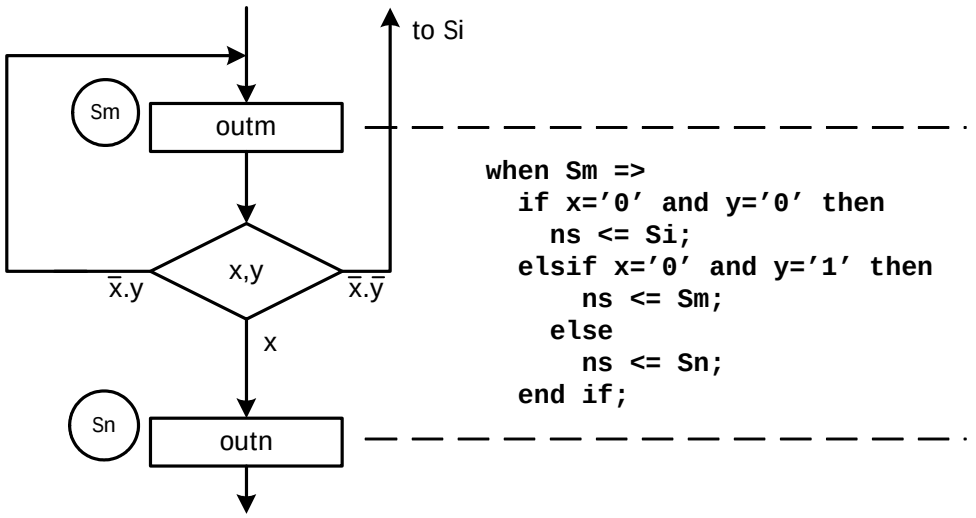

Fig. 4-a. Singe (nested) if asm transition diagram and associated code with logic expressions.

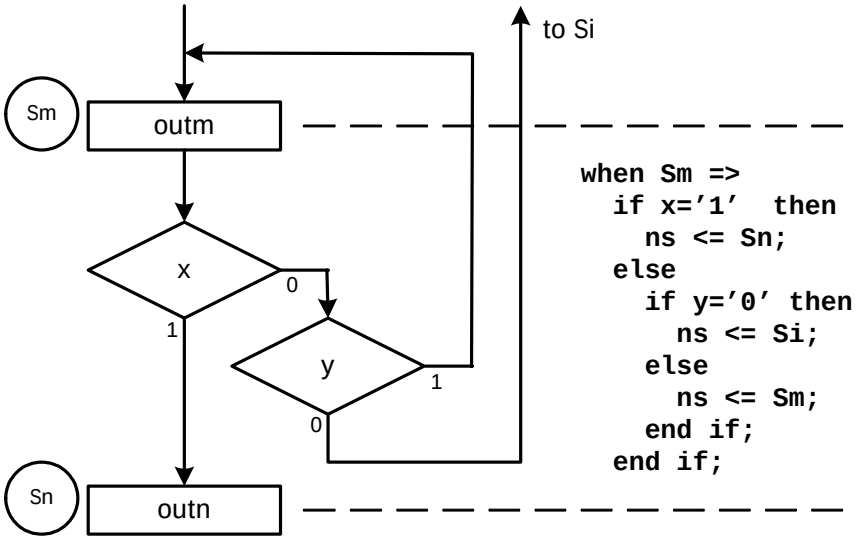

Fig. 4-b.Double if asm transition diagram and associated code with one if per condition.

a "reset" and an "enable" signal and produces as output a "done" signal. In Fig. 5 this is shown in the segment with states: "arming1", "arming2" and "armed". The first state outputs an asynchronous reset (arst) and immediately transitions to the next state which outputs a synchronous enable (ena) signal. This choice allows decoupling the state machine clock from the module clock, the former being usually much faster than the latter. The transition to the third state in this segment takes place when the module has indicated with its "done" signal, that it has completed its task. This implementation serves a large variety of functions and both the three-state segment in the asm chart and the associated code become practically boilerplate and self-documenting code. This also works well if the external module is another state machine. Occasionally it is desirable to run both controllers at the same clock speed but this often causes timing problems, which can be avoided, however, by triggering one state machine with the rising edge and the other on the falling edge of that same clock.

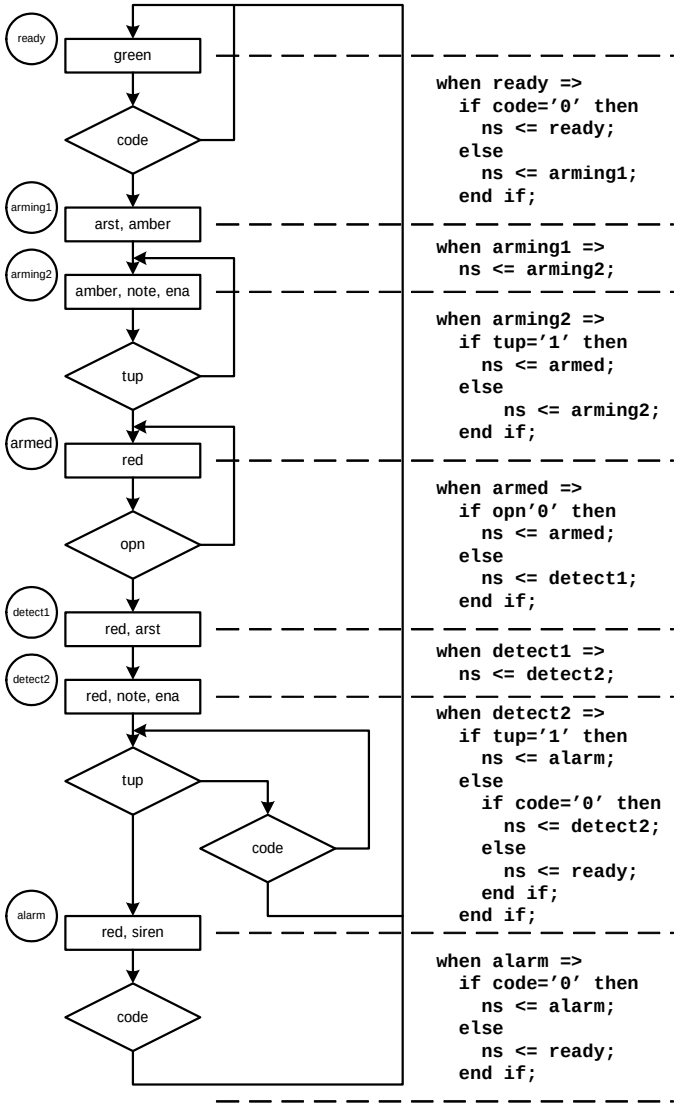

Fig. 5. ASM chart for the state transitions and code for the alarm system.

CONCLUSIONS AND SUMMARY

The instructors have found that one pitfall with introducing VHDL too soon, is that the students do not get an appreciation for the actual implementation of the design. That is one reason the first state machine is implemented strictly with schematic capture. The design with VHDL is a large step removed from the hardware, but knowledge and understanding of the limitations and the special features of the hardware are very necessary for an efficient design. It's for that reason that throughout the course the philosophy "Know your Hardware" is strongly emphasized. Also by taking the time and making the effort to look at the details of a carefully scripted simulation, the students develop an appreciation for the large amount of information that can be obtained from
such an analysis, but that it is generally not easy or obvious and learn that there is much more to it than seeing what you want to see.

REFERENCES

1. Fundamentals of Digital Logic with VHDL Design. S. Brown and Z. Vranesic, McGraw-Hill, ISBN 0-07-012591-0.

2. VHDL for Programmable Logic. K. Skahill, Addison-Wesley, ISBN 0-201-89573-0.

3. Altera Max plus 2 software and hardware and documentation available from their University Program. Web site www.altera.com.

4. Three-Day VHDL workshop for industry at UCF – Dr. H.v.d.Biggelaar.

BIOGRAPHICAL INFORMATION

Rosida Coowar, Ph.D, is an Associate Professor in the Department of Engineering Technology at the University of Central Florida, Orlando, FL. She teaches a variety of courses including Digital Systems Design, Programmable Digital Devices, Applied Quality Assurance and Applied Reliability. Her research interests include applied statistics and simulation in the improvement of processes and systems. She is a member of the IEEE, ASEE and Tau Alpha Pi.

Hans v.d.Biggelaar, Ph.D, is Professor Emeritus from the Electrical and Computer Engineering Department of the University of Massachusetts Dartmouth where he was Director of the VLSI Laboratory. He consults in the area of digital electronics and VHDL. He is a member of Sigma Xi, Eta Kappa Nu and Senior Life member of the IEEE, a member of the National Association of Flight Instructors and an FAA Safety Counselor.

APPENDIX A

```
-- 2-process state machine
-- ~\max2work\hvdb\asm2.vhd

library ieee;
use ieee.std_logic_1164.all;

entity asm2 is
port(arst,clk,in1,in2: in std_logic;
        out1, out2, out3: out std_logic);
end entity asm2;

architecture a_asm1 of asm2 is
type state is(state0,state1,state2,state3);
   signal ps, ns: state;
begin

   -- clock with asynchronous reset:
   pclk: process(arst,clk)
   begin
      if arst='1'
         then ps <= state0;
      else
         if rising_edge(clk) then
            ps <= ns;
         end if;
      end if;
   end process pclk;

   -- state transitions:
   pasm: process(ps,in1,in2)
   begin
      case ps is
         when state0 =>
            if in1='0' then
               ns <= state0;
            else
               ns <= state1;
            end if;
         when state1 =>
            if in2='0' then
               ns <= state1;
            else
               ns <= state2;
            end if;
         when state2 =>
               ns <= state3;
         when state3 =>

            if (in1='0' and in2='0') then
               ns <= state0;
            else
               ns <= state3;
            end if;
         when others =>
            ns <= state1;
      end case;
   end process pasm;

   -- outputs:
   out1 <= '1' when ps=state1 else '0'; -- Moore output
   out2 <= '1' when (ps=state1 and in2='1') else '0'; -- Mealy output
   out3 <= '1' when ps=state2 else '0'; -- Moore output

end architecture a_asm1;
```

Complete code for the alarm system.