Behavior Research Methods & Instrumentation
1976, Vol. 8 (2), 113-117

Hardware design style: The vital element

DAVID WINKEL and FRANKLIN PROSSER
Division of Computer Services, University of Wyoming, Laramie, Wyoming 82071

This is a tutorial in the application of digital integrated circuits to laboratory experimental design. For appropriate experiments, a hardware controller may be the most economical as well as the most enjoyable approach. By and large, the hardware profession has done a poor job of providing clear, straightforward principles of hardware design; therefore, the design process is often viewed as a "black art" accessible only to specialists. The computer software profession has begun to realize the vital need for systematic approaches to problem solving. We hope to contribute to a similar movement in the digital hardware design area.

The ideal hardware design technique should (1) be easy to learn, (2) produce clean, easy to understand devices, (3) produce documentation as a by-product of the design process, (4) provide a uniform method for designing hardware for any speed and complexity range, and (5) force the designer to think through the total design before construction starts.

This paper describes the essential elements of a method that meets the above criteria. It is the primary design technique taught to our computer science students, who have no prior digital experience. These students can rapidly produce elegant designs that typically require fewer chips than conventional designs (Clare, 1973; Peatman, 1972).

STYLE

It is important that the designer develop a design style that forces him to do things correctly. In addition to the general criteria given in the introduction, some more technical elements of our style are:

(1) Use only edge-driven synchronous (clocked) logic. This design style allows the logic to "settle" until all signals are stable. The machine then branches to the next state on the basis of stable signals only. This has two important corollaries: (a) If the machine is misbehaving, the clock can be slowed down, thereby giving more time for signals to stabilize. (b) The clock can be stopped, thereby freezing the machine in a given state. The machine can then be probed using static signals only. Removing time as a variable in this fashion vastly simplifies debugging. A simple low-cost logic probe is as useful as an oscilloscope for a "frozen" machine.

(2) Do not use single shots. A single shot is, by definition, a time-sensitive element. Single shots cannot therefore be frozen, which eliminates the possibility of static debugging. They are easily triggered by noise and, in addition, drift with time. In rare cases, single shots will be needed, but the average designer uses far more than necessary.

The second author is on leave from Indiana University.

(3) Use well-established logic families. A relatively new logic family such as I^2L will have few IC device types available to the designer, whereas an older line such as T^2L has a wide variety of debugged and useful devices. CMOS, developing rapidly toward a full line of devices, is worthy of some consideration because of its insensitivity to noise and power supply fluctuations.

(4) Do not use fast logic families. High-speed integrated circuits have important areas of applicability, for instance, in fast computers. But high-speed elements are much harder to use and require special considerations, such as terminated transmission lines. For most applications, slower logic families, such as T^2L and CMOS, are better than such high-speed families as ECL and Schottky T^2L.

(5) Flowchart the process you are trying to automate. This is much more important than flowcharting a software program. Programs are easily changed. Hardware is not.

PLAN OF ATTACK

Step A

Begin by selecting integrated circuits which can perform the functions of the target device. Some illustrations follow.

Counting. Choose a synchronous IC counter. These devices typically count in base 10 or base 16 and can be cascaded to count any desired number of events. Synchronous counters are superior to ripple counters, since all bits on the output lines will change only at clock time. Popular T^2L counters are the SN74162 for decade counting and the SN74163 for base 16 counting.

Storing data. For storing small amounts of data, a register such as the SN74174 can be used. For storing large amounts of data, a static random access memory such as the Intel 2102 is convenient.

Adding. Two cases should be distinguished: (1) Always adding one. This application is easily implemented using a counter. (2) Adding two arbitrary numbers. Use 4-bit full adders such as the SN74283. These devices can be cascaded to add larger numbers.

Switching. Do not build up data path switches from gates. Multiplexers can be used to selectively switch a number of data sources onto one output line. Demultiplexers can perform the inverse function.

Gating. A wide range of AND, OR, INVERT gates is available. The most natural design technique is mixed logic, an important component of our style (Kintner, 1971).

Step B

Develop the data paths. After the ICs are chosen to implement the various activities in the target device, they must be interconnected by data paths. At this stage, the data paths are lines on a page; later they will become wires. How the devices are controlled is not important at this stage, but it is important to be sure the proposed data paths and integrated circuits are capable of carrying out the desired operations if they receive the proper control signals.

Step C

Draw a flowchart that will model the signals needed to control the data path structure. In general, the flowchart will branch on the values of status signals supplied by the controlled structure. (The next section illustrates our preferred flowchart technique.)

Step D

Before a single wire is connected, carefully study the interaction of the control algorithm (flowchart, ASM chart) and the controlled device. Some iteration of Steps A, B, and C may be necessary to correct oversights. This is *the* critical phase in the design process, since it determines the final success or failure of the project. Errors found at this stage may be corrected gracefully; errors discovered after wiring often result in confusing patches in the logic.

HARDWARE FLOWCHARTS

Clare (1973) has devised a convenient way to represent hardware flowcharts. His ASM (Algorithmic State Machine) chart notation has several advantages over the usual state diagram representation (Hill & Peterson, 1968). Perhaps most important is the natural way in which sequences of events are represented. There is considerable similarity to the familiar flowchart used by programmers, but this analogy must be applied with caution.

ASM symbols have a very exact definition which implicitly includes time. This is an important difference between an ASM chart and a normal flowchart. It is obvious that hardware flowcharts must deal with time, since their function is to issue time-sequenced control signals. In synchronous designs, time is divided into

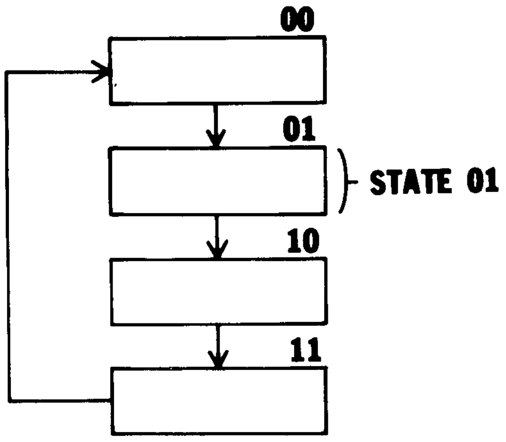

Figure 1. ASM chart of 2-bit sequential counter.

quanta by an oscillator commonly called the clock. The time quanta (clock period) can range from less than a microsecond to several seconds in typical designs. The clock period is chosen by the needs of the external logic being controlled. For example, if you are trying to measure reaction times to an accuracy of 1 msec, the clock period must be less than that; .1 msec would be a good value to start with. A move from one spot to another on the flowchart will take place only on a clock tick. A clock tick will normally be the rising edge of the clock waveform.

A state is an identifiable location on the ASM flowchart which exists for one clock duration. Consider the very simple four-state sequential flowchart in Figure 1. This flowchart could be implemented with a 2-bit binary counter sequenced by the system clock. Each rising edge of the clock will cause the counter to increment, moving the system into a new state. The system will stay in the new state until the next rising clock edge arrives. For convenience, the values assumed by the counter are shown in Figure 1 above the state boxes. Consider State 01. State 01 is reached from State 00 by applying a clock pulse to the counter. Let the top of the box be the time which starts the state. Time then flows uniformly to the bottom of the box, at which time a clock pulse causes the counter to leave State 01 and enter State 10. Thus, from a time standpoint, the top of box 10 and the bottom of box 01 are simultaneous.

A sequential flowchart can exist as a logical construct without the corresponding hardware which implements it. In this case, the states would be identified by symbolic names. These are conventionally written to the left of the states, as illustrated in Figure 2.

The meaning of the state "reset printer" is independent of the hardware used to realize the ASM chart. A state is identified by its location on the flowchart and, once initiated, exists for one clock period. Within each state box, the designer would normally indicate any

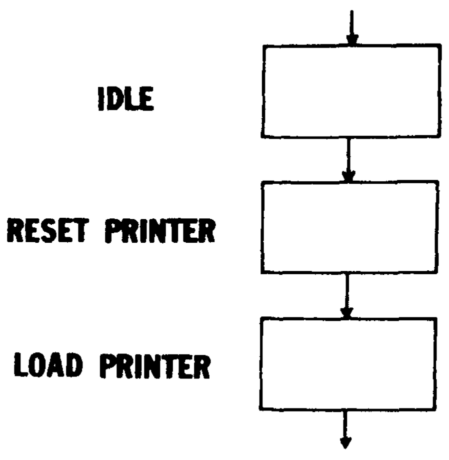

Figure 2. Labeled ASM states.

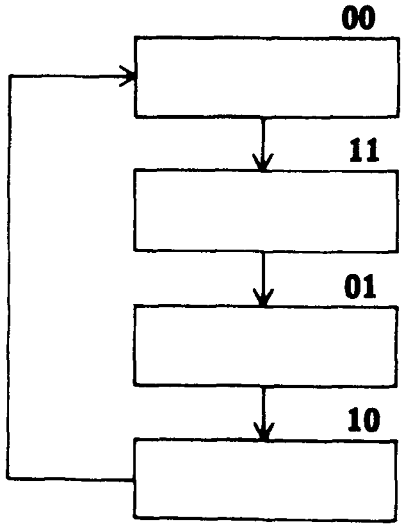

Figure 3. An awkward state assignment.

signals to be generated during that state time. Such signals, called outputs, are used to control the movement of data in the data path structure; illustrations might be "clear register T," "set RUN flip-flop," "enable memory read." The flow of control from state to state specified by an ASM chart is independent of such signals. In the remainder of this exposition, we will concentrate on the implementation of ASM chart structures, and to avoid unnecessary particularization of the examples, we have omitted signals generated during state times.

The general design problem can be stated: "How can I build hardware to implement any given ASM chart?"

An implied problem is the representation of a given state by some convenient hardware device. This is most easily done with flip-flops that are set and reset by the system clock (the unit of time will then be the duration between clock pulses). States are commonly represented by an encoding. With n flip-flops, 2^n combinations are possible. A state assignment results when each state is represented by a unique encoding. It should be emphasized that the encoding is the designer's choice. Some choices will lead to neater solutions than others. For example, the state assignment shown in Figure 1 yields a ready made implementation, namely, the 2-bit binary counter. If the state assignment shown in Figure 3 is made, a binary counter cannot be used to sequence through the ASM chart. (The design technique discussed later in this paper could, however, be used to implement this ASM chart.)

While purely sequential ASM charts are useful in some cases, most algorithms require branching. A state with a two-way branch can be represented as in Figure 4.

Time flows uniformly from the top to the bottom of the state. The clock pulse at the bottom causes the state controller to jump to one of two states, b or c. This decision is based on the value of the test variable X. If X = 0, then the jump is to State b; if X = 1, the jump is to State c. The voltage representing the logic variable X must, of course, be stable at the clock edge causing exit from State a. It is conventional to label only the

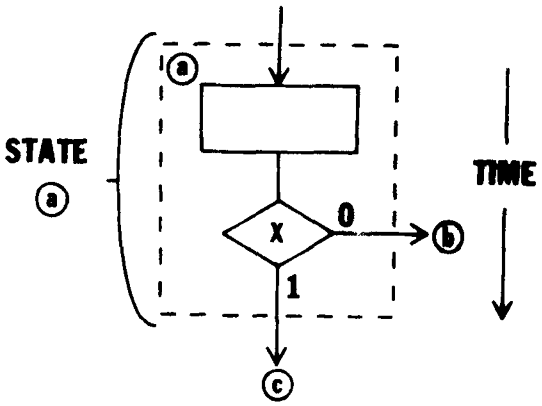

Figure 4. Conditional branch in ASM chart.

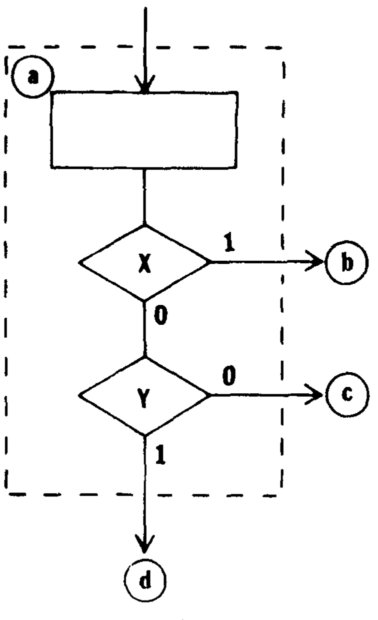

Figure 5. Multiple branches.

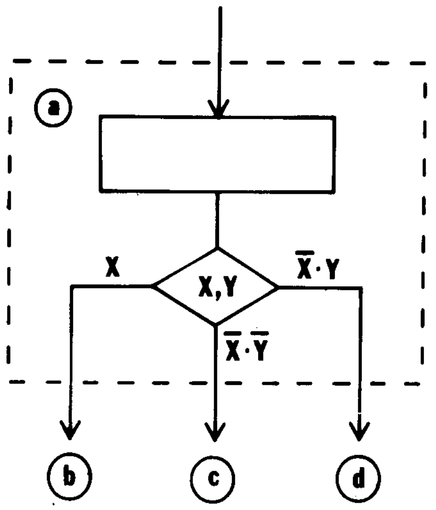

Figure 6. Emphasizing parallel nature of test evaluation.

rectangle with the state name (here a). Actually State a encompasses the entire area enclosed by the dotted line.

Conditional branching during a state time can be extended to cover more than one test. For example, Figure 5 shows transfers from State a dependent on two test variables, with variable X having priority over variable Y. At the moment of leaving State a, we jump to only one next state: b, c, or d. The jump conditions are: to State b, X = 1; to State c, (X = 0) • (Y = 0) or $\overline{X} \cdot \overline{Y} = 1$; to State d, (X = 0) • (Y = 1) or $\overline{X} \cdot Y = 1$.

Note that all of these test conditions are computed in *parallel* during State a. Only *one* jump condition will be satisfied, so there will be only *one* next state. An ASM chart must not be interpreted as if it were a software program. In software, you would interpret the flowchart as specifying first a test on X, then a test on Y.

The power of hardware lies in its ability to make parallel instead of sequential decisions. In the above case, we could emphasize the parallel nature of the decisions by rewriting the block as in Figure 6. However, the designer will usually find it more convenient to use the serial form of Figure 5, since in that form the priority of the X test over the Y test is immediately obvious.

GENERAL METHOD FOR IMPLEMENTING ASM CHARTS

We are now ready to consider a universal design technique for implementing ASM charts of arbitrary complexity. As preliminaries we note the following:

(1) We will assume that the state identifiers are held in "D"-type flip-flops. (2) The sequencing problem is fundamentally one of computing the jump address to the next state. (3) The starting point of the implementation is a well thought out ASM chart.

The design process proceeds in the following steps

Step 1

Start with the ASM chart. For example, consider the simple three-state chart of Figure 7.

Step 2

Make a state assignment. In this case, 2 bits are sufficient to encode four states (1 bit will only encode two states; therefore, we will need 2 bits). For example, assume the state assignment shown in Figure 8. Note that the choice is arbitrary and up to the designer. However, some choices may lead to simpler logic, although the technique described here tends to minimize complexity generated from different assignments. We may refer to a given state either by its symbolic name or by its assignment, as convenient.

Step 3

Select the state generator configuration. Assume there are n state flip-flops. Each state flip-flop will have 2^n input multiplexer (MUX) feeding its D input. Each MUX will be controlled by all the state flip-flop outputs. Thus, each MUX will be addressed to the current state of the system. If the corresponding MUX inputs develop the next address (jump address), the controller will properly follow the flow chart.

For the case above, label the most significant bit of the state code "B," and the least significant bit "A." Figure 9 shows the desired configuration.

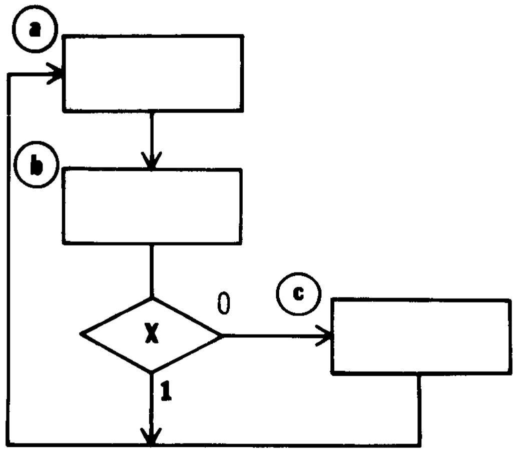

Figure 7. An example ASM chart.

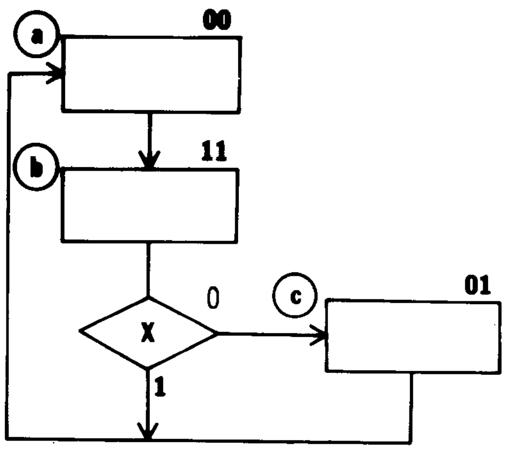

Figure 8. A state assignment for the example ASM chart.

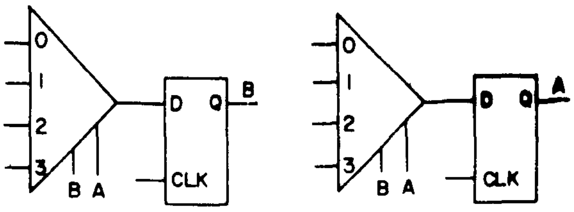

Figure 9. Structure for a four-state generator.

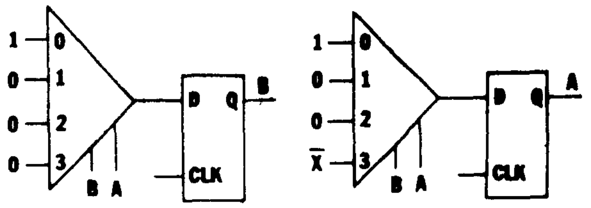

Figure 11. MUX-driven state generator for example ASM chart.

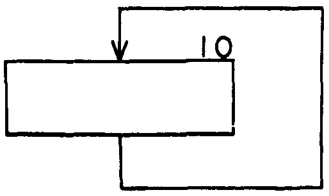

Figure 10. Lockup in unused state.

Step 4

Derive the MUX inputs. In our simple example, we can develop the MUX inputs by inspection. (a) In State 00 the next state address is 11. The output of each MUX must therefore be a one, since both bit A and bit B must become one. This will be true if the zero address input of each MUX is a one. (b) In State 01 the next state address is 00. Therefore, a zero must be connected to the one address input of both MUXs. (c) State 11 involves a conditional branch, which in turn will involve conditional inputs to the MUXs. State 11 may go either to State 00 or to State 01. For either case, the B bit must go to zero. Therefore, the three address input to the B MUX is an unconditional zero. The A bit, however, is a one only if $X = 0$; therefore, the three address input to the A MUX must be $\overline{X}$. (d) One of the nice features of this design technique is the ease of handling unused states, for example, State 10 in Figure 10. With conventional designs, it is sometimes possible to get "lockup" in an endless loop unless careful provision is made to avoid the condition. "Lockup" in the present example would be shown in Figure 10. With the present technique, it is a simple matter to force a transition to any state. Suppose the designer chooses State 00, so that if the flip-flops ever happen to be set to the unused State 10 (e.g., at power-up time) the system will always go to State 00. The complete design is shown in Figure 11.

SUMMARY

Good hardware design style is the most important ingredient in the successful application of digital logic. In this paper, we outlined the elements of a systematic design style that we have found useful. Using the ASM chart notation (an important ingredient in our style), we have proposed a straightforward procedure for implementing an arbitrary next-state generator. Future contributions will emphasize other aspects of the design process.

REFERENCES

Clare, C. R. *Designing logic systems using state machines.* New York: McGraw-Hill, 1973.

Hill, F. J., & Peterson, G. R. *Introduction to switching theory and logical design.* New York: Wiley, 1968.

Kintner, P. M. Mixed logic: A tool for design simplification. *Computer Decisions*, 1971, **10**, 55.

Peatman, J. L. *The design of digital systems.* New York: McGraw-Hill, 1972.