5

Control System Design

The design of a computer system, like that of all digital systems, requires both data manipulation capabilities (logical units, adders, multipliers, etc.) and control capabilities. The data manipulation elements form the data path of the machine, while controlling the flow of data on that data path is the responsibility of the control section. The design of the data path elements and the instructions that identify the work to be done on the data paths have been the subjects of the preceding chapters. In this chapter we will deal with methods used to control the flow of information within a computer. Our intention is not to provide an in-depth discussion of sequential design techniques; a number of excellent texts provide that material [Mano79, Flet80, McCl86, Bree89]. Our intention is to provide some insights into different ways in which those methods can be applied in the design of a control section of a computer.

The control system of a computer is basically a sequential system that implements the fetch-decode-execute function of instruction execution. Since it is a sequential system, it can be designed using the same techniques used to design counters, controllers, or any of a wide variety of digital systems. In this chapter, we examine different techniques for implementing sequential systems, and apply those techniques to the control of computational elements. Thus, our first task is to review some of the concepts used in sequential design.

The application of sequential design techniques will result in a control system that activates the appropriate clock lines, enables, and interface signals to accomplish the work of the computer system. .However, before the control system can be specified and implemented, it is necessary to identify the appropriate clocks, enables, and other signals used in the system. Therefore, the first requirement of a computer system design is to develop a detailed data path block diagram. This diagram must identify the control lines that can be used to manipulate data in the system. And with this diagram and the definition of the instructions to perform. RTL descriptions of the data transfers needed can be developed.

Thus. we will look again at the problem of data path definition and RTL specifications, and see how they are used in the process of control system design.

The designer of a control system must know what signals are available for the control and manipulation of the data in the system; these control lines are identified by the detailed block diagram. In addition, the order of operations must be specified, and this information comes from the RTL descriptions of the work to be done. With this global view of the system, a designer can select the most appropriate sequential design technique and create a system that will assert the control lines to do the work. We will look at different methods for implementing the control systems, providing examples of each.

5.1. Elements of Sequential Design: A Review

The circuits of the preceding chapters are all classified as combinational circuits: the outputs are functions only of the inputs. These can be modeled as shown in Figure 5.1; the outputs will change whenever the inputs change. This model applies to a variety of devices and circuits: random logic, ALUs ('181, '381), multiplexers ('151, '157), decoders ('138, '154), memories, PLAs, and the like. Normally we like to think that these circuits are perfect, that the outputs will change instantaneously to their new value whenever the inputs change. However, associated with real devices are real delays, and the outputs will follow the inputs after some finite time delay. Some outputs may change during the finite time delay. and resume their former values after the delay period has passed. This results in glitches that can cause problems in circuits, and care must be taken to prevent the glitches from occurring or to ascertain that, if glitches do occur, they will not cause problems. Thus, a designer must be aware of the timing restrictions in the process of creating the data path and the transfers represented by the RTL statements.

If a system is to have outputs that reflect not only the current set of inputs, but the history of the system as well, then a different model is necessary. An addition to the model of Figure 5.1 is shown in the model of Figure 5.2. Here the outputs are not only a function of the current inputs, but also the past history of the system as well. This history is reflected in the "state" of the machine, which is the value stored in the collection of memory elements within the system. If there are N memory elements, then there are 2^N possible states that the system can assume. Hence, systems with more than a few flip-flops are intractable; a system with 20 bits of memory arranged in registers or other flip-flops would have more than a million possible states.

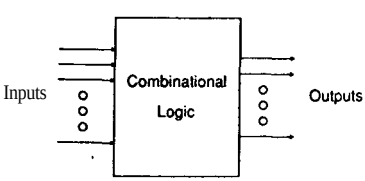


Figure 5.1. Block Diagram of **Simple** combinational Circuit.

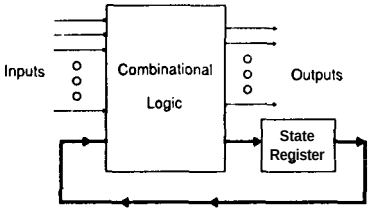


Figure 5.2. Simple Block Diagram of Sequential Circuit.

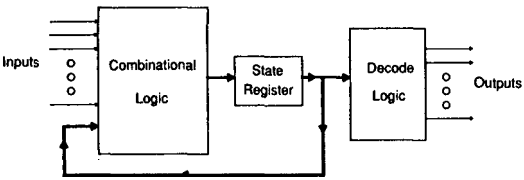


Figure 5.3. Sequential Circuit: **A** Moore Machine.

The system depicted in Figure 5.2, where the outputs are functions of both the current state and the inputs, is called a Mealy machine. This kind of machine is useful in certain circumstances, but can cause problems because of the lack of synchronization between inputs and states. That is, the outputs may be of varying lengths, since the inputs change asynchronously with respect to the states of the machine. **A** different, slightly more restricted, model for a sequential system is shown in Figure 5.3. This type of system is called a Moore machine. Here the outputs are functions of the present state only. Inputs then influence the outputs only in that they can affect the next state of the sequential machine, but the outputs are not directly functions of the inputs. This model, and variations of it, represent the controllers that we will consider for the control sections of computers and other digital systems. The outputs of the sequential machine will be the control signals needed to manipulate data and move it within the system. The inputs required to specify the desired sequence of steps consist of synchronization flags, status bits, test conditions, and other information that influences the behavior of the system. The designer's challenge, then, is to design a sequential system that will assert the outputs in an appropriate manner to accomplish the work of the system. However, before the control system can be defined and the sequence of outputs specified, the data path must be specified and the control signals on the data path identified.

5.2. Data Path Formulation

The formulation of the data path for a computer or other digital system is a complex task that is influenced by many factors. The foremost requirement is that the system be able to perform the action required by the underlying task. Just how that task is accomplished is a designer's choice; the decisions made by the designer reflect his understanding of the task and the requirements imposed by

system constraints. Consider, for example, the IBM System 360/370 family of computer systems. This was one of the first families of computers in which the different models were identified in the beginning, rather than having different models announced as permitted by customer demand and marketing strategy. The 360 family was set up to cover a variety of performance capabilities and economic ranges. Nevertheless, a program executed on different members of this family should arrive at the same answer on each machine. The instruction set architecture of the system appears the same to a programmer regardless of the model on which the program runs. However, the techniques used to implement the operations vary from model to model. The number of data paths, the arithmetic units, the memory interactions, and the control system for each model are configured to match different sets of economic and performance constraints. The same idea will be true for all digital systems: the parts used, the data paths provided, and the interfacing methods will be dictated by the intended use of the system. Some of the constraints and their implications are listed below:

- Economic: How expensive are the components used to build the system? This includes not only the integrated circuits, but other components as well, such as sockets, connectors, display elements, wire, printed circuit boards, and so on, as well as manufacturing costs.
- Interface requirements: Many devices are specifically designed to interface to TTL components. However, other technologies can require a different set of voltages and currents for information exchange. This also applies to the protocols required for the exchange.
- *Speed;* A variety of questions must be addressed. One of the first is to choose the technology in which the system will be implemented. Lower speed requirements can utilize some MOS technologies that conserve energy and do not have fast cycle times. Higher speed technologies, such as ECL and GaAs, require careful adherence to design constraints. However, another speed issue is the extent of the use of concurrency within the system, from pipeline techniques to multiple data paths. Each of these options carries with it a set of constraints that identify its range of usefulness.
- Power: The amount of power that a system utilizes may be a factor in the system. If the unit is to operate on battery power for extended periods of time, or be limited in the amount of available power, then the designer must select components and techniques accordingly.
- *Dynamic* range: Arithmetic requirements are often mandated by the intended applications of the system and the allowable signal to noise ratio. A system may be able to satisfy the data representation requirements with integer or fixed point arithmetic of a certain number of bits; or the required dynamic range may indicate that floating point operations are necessary. The data paths and arithmetic capabilities must match system needs.
- *Flexibility:* Many digital systems are created not to solve a single problem, but to provide a device that can be used in a variety of applications to achieve a reasonable solution. Therefore, the system must be flexible enough to be used easily in any of a number of target areas.
- Maintainability: Building a computer system. or other digital device, to satisfy a particular need is only part of the overall problem. Because of device failures, power surges, or other problems the system will at some time cease to function properly. One of the desirable characteristics for digital systems is

that they be maintainable. That is, the design and the implementation be done in such a way that devices and subsystems that are not functioning properly can be identified and easily replaced.

- Environment: This nebulous heading is used here to include a variety of other types of restrictions. If the system is satellite-based, it must not only conserve power used, but it also may have a radiation hardening requirement. If the unit is to operate in an airplane, it may have vibration tolerance requirements, extended temperature requirements, or other restrictions.

Acceptable limits for these and other requirements are identified by the specifications for the system to be designed. The designer must utilize the ingenuity that he has to propose a design that will meet the specifications of the system. There are many different approaches to solving a given problem; indeed, vastly different data path solutions may be proposed which satisfy the requirements of the system. These approaches may use single bus implementations, multiple bus implementations, point to point techniques, or any of a variety of approaches. In any case, the system must satisfy the requirements placed upon it by its application area and intended use.

The designer must select the data path components from the pool of available parts in the target technology, arrange the components and the interconnections so as to meet the system requirements, and identify the basic transfers and manipulations required to perform the necessary work. We wish to make two points. First, the design of the data path is basically independent of the control design. There may be factors in the intended control design that influence the data path formation, and there may be elements of the data path that bear on the control design, but basically they are two different problems. Second, having identified the elements in the data path, a designer must then identify the signals that will control the flow of information within the data path. It is the responsibility of the control section to assert the signals in such a way that the appropriate work is accomplished, and, once the signals are identified, the design of the control section can proceed to achieve that objective.

To reiterate the points made above, the designer must:

- First select an appropriate technology and a set of components in that technology to provide for the needs of the system.
- Interconnect the components in such a way that the work of the system can be accomplished.
- Then, using a register transfer language or other means of specifying the action to take place, identify the data transfers and arithmetic required by the system.
- Identify the control signals required to accomplish the work of the system.

When the data path has been defined to this level, the design of the control portion of the circuitry can proceed.

This process is best illustrated by an example. The example chosen here, and the other examples in this chapter, are contrived to illustrate specific points, and do not necessarily reflect the "trickiest" way to accomplish some work. But once the principles have been identified, the designer can then proceed to apply them to other designs. The following example, like the other examples in this chapter, is more extensive than those in earlier chapters; for instance, Section 5.7 consists entirely of two different implementation techniques applied to the same

machine. Therefore, the examples here are interwoven with the text, not separated as a short example to illustrate a single point.

Our first example of a digital system is the calculation of an inner dot product. This is used repeatedly in mathematics for doing matrix manipulations; it is also used in digital signal processing for transversal filters. The example is to design a finite impulse response (FIR) digital filter with **25** coefficients. The **equation** for this calculation is:

$$\text{output} = \sum_{i=0}^{24} S_i \times C_i$$

where S_i represents samples of an input stream and C_i represents constant coefficients. We will assume that the system is to stand alone; that is, that the system will contain an **A/D** converter to provide samples and a **D/A** converter to accept outputs. We will also assume that the coefficients are known and constant. The data manipulations involved in the FIR process are shown in Figure **5.4.** The input is sent to a delay network, which saves **25** values of the data stream. Each of the delayed values forms one of the S_i of the above equation. Each sample is multiplied by its corresponding coefficient (C_i), and all of the resulting values are summed to form the final result. The system **architect/designer** has the task of implementing the data manipulations represented by the **FIR** equation in real hardware.

The network shown in Figure **5.4** could be implemented directly in hardware. However, that would require **25** separate multipliers and some mechanism for summing **25** results in parallel. **A** more conservative solution is to build a system around a **multiply/accumulator** (MAC), a device that will perform a multiply and an add in each clock cycle. These modules have been available for several years and are applicable to a variety of different calculations; the FIR example is an ideal use for this module, since the chip performs all of the arithmetic needed to obtain the result. To present the appropriate values to the MAC, we will utilize memories to store the data and the coefficients. Thus, our problem will be to design a system that will accept a sample, store it in a memory, and then perform the calculations identified by the above equation using the current sample and the previous **24** samples.

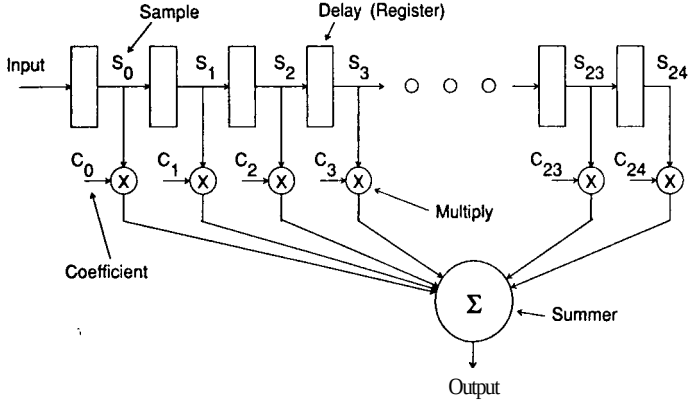


Figure 5.4. Data Manipulations Involved in Inner Dot Product.

A simple block diagram of a piece of hardware to do this is shown in Figure 5.5. The MAC can be connected to the other elements of the system in a variety of ways to satisfy different system requirements. The solution shown is capable of a fairly high computation rate, and yet is simple in its implementation. The elements of the system, their names and their responsibilities are as follows:

- ***Coefficient memory:*** (C_MEM[k]) This memory contains the constants (C_i) needed by the algorithm. The memory chosen here is a small PROM (programmable read only memory).
- ***Coefficient memory address register:*** (C_ADR) This is a counter used to identify the current coefficient. It must start at zero for each iteration and increment through the coefficient numbers, which are used directly as coefficient addresses.
- ***Sample memory:*** (S_MEM[k]) The sample memory is used to store the current sample and the previous 24; actually the memory is made of a number of RAMs, so 32 values are stored, but only 25 used for any single calculation.
- ***Sample memory address register:*** (S_ADR) This counter is initialized to the address of the current sample; it is then incremented to point at the preceding samples in order.
- ***Initial sample address register:*** (I_ADR) This register identifies the starting point of the algorithm for each pass. The correct starting point for the current iteration is one less than the starting point for the previous iteration. Thus, this counter will decrement once each pass.
- A/D ***converter:*** (ADIN) This module provides the new data for each iteration. We assume that the time at which conversion begins is controlled by an

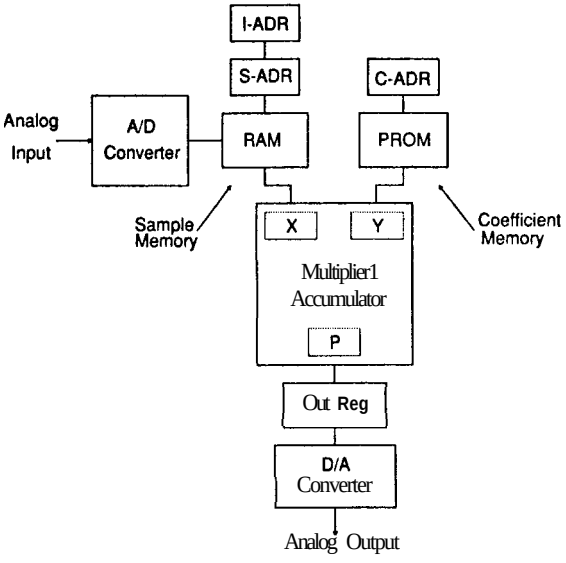


Figure 5.5. Simple Data Path Block Diagram for Finite Impulse Response System.

external source. When new data is available, a ready Rag will be asserted. Thus, testing this ready Rag will identify when the algorithm should be performed.

- ***DIA converter:*** (DAOUT) This module accepts the output of each interaction and converts it to an appropriate analog value.
- ***Output register:*** (OUT) This is a register that isolates the D/A converter from the values on the output of the MAC during the calculation process.
- ***Multiplierlaccumulator module:*** This module has the responsibility for doing the arithmetic needed by the algorithm. It will do each multiply, then add the value to a running sum. There are three registers internal to the system, two input registers (X, Y) and an output register (P).

When the designer has arrived at a data path representation such as that shown in Figure 5.5, the next step is to identify the work to take place. As identified in the formula defining the calculation, 25 coefficients will be multiplied by 25 data values, and the results of the multiplies summed to the result. Assuming that the RAM and PROM (for the data and coefficient storage) both contain 32 locations, these memories can be visualized as shown in Figure 5.6. Part a shows the coefficient storage. These values are always used in order, from location 0 to location 24. Thus, at the beginning of each iteration the address register for the coefficients will be set to zero. Figure 5.6(b) indicates that the first value received will be placed in location 0 of the RAM, and then the initial sample address register will be decremented. Thus, the next location to be filled by a sample value will be location 31. Figure 5.6(c) shows the contents of the RAM after 33 samples have been received. The 33rd sample (Sample 32) overwrites the first sample (Sample 0). The output of the FIR calculation uses the 25 most recent samples, also identified in part c. The 25 most recent samples utilize a different portion of the RAM for each iteration. The samples used for the 41st iteration (Sample 40 through Sample 16) are identified in of Figure 5.6(d). Once the basic algorithm is understood, we can specify the work to be done with RTL statements:

start: if (ADIN not ready) goto *start*			Check input data.
0	→	C-ADR	Clear coefficient address.
I_ADR	→	S_ADR	Load sample address.
ADIN	→	S_MEM[S_ADR]	Data to sample memory.
I_ADR	→	I_ADR	Decrement initial address.
S_MEM[S-ADR]	→	X	Load sample to X reg.
C_MEM[C-ADR]	→	Y	Load coefficient to Y reg.
S-ADR + I	→	S_ADR	Increment sample address.
C_ADR + I	→	C_ADR	Increment coefficient address.
X × Y	→	P	Load product register.
S_MEM[S_ADR]	→	X	Load sample to X reg.
C_MEM[C-ADR]	→	Y	Load coefficient to Y reg.
S-ADR + I	→	S-ADR	Increment sample address.
C-ADR + I	→	C-ADR	Increment coefficient address.

```
over:   P + X × Y  ┐→ P        Add product into P register.
        S_MEM[S_ADR] → X       Load sample to X reg.
        C_MEM[C_ADR] → Y       Load coefficient to Y reg.
        S_ADR + 1    → S_ADR   Increment sample address.
        C_ADR + 1    ┘→ C_ADR  Increment coefficient address.

        if (not done) goto over  Repeat for all samples.
                                 Work is done when C_ADDR is 25.

        P → OUT                  Update output value.

        goto start               Start over
```

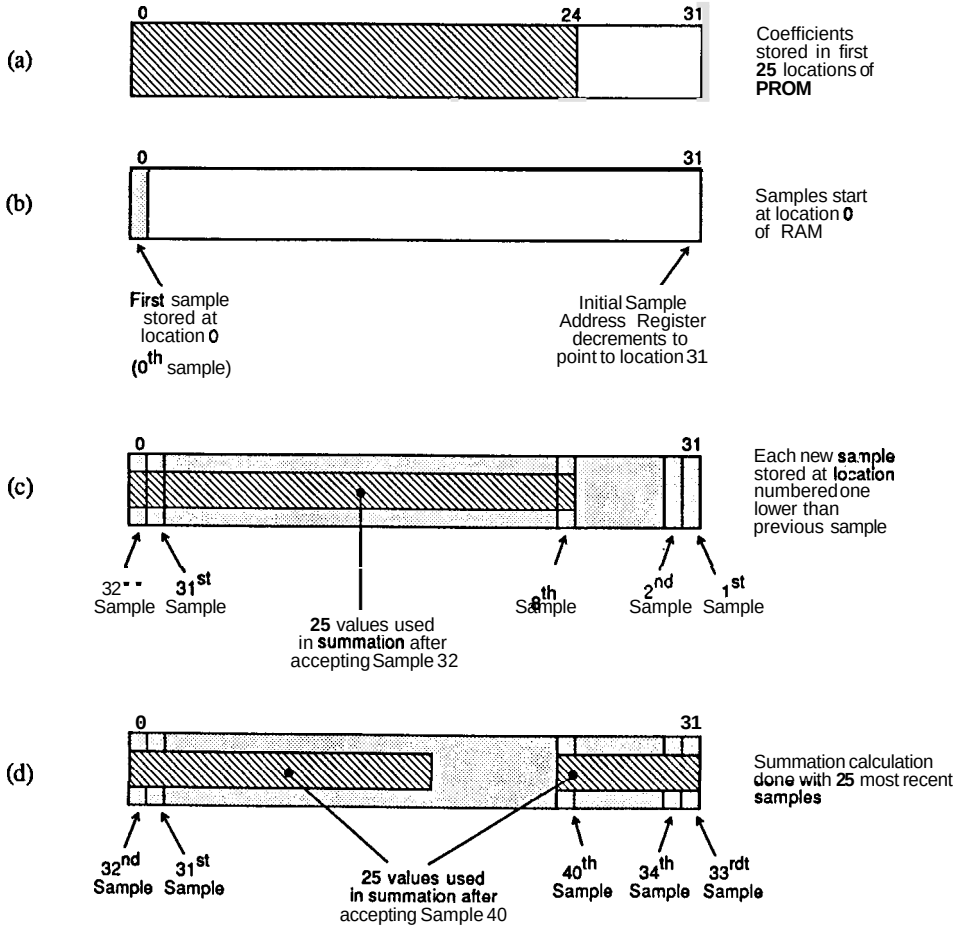


Figure 56. Coefficient and Data Storage for FIR Example.

This identifies the work needed to perform the appropriate calculations, as well as the possible parallelism of simultaneous events. The system waits for new data to become available (ADIN ready), at which point the data processing begins. The first step is to clear the coefficient address and load the sample address register from the initial sample address register. These two events can happen simultaneously. Then, the input value is loaded into the sample memory (at the address just loaded into C_ADDR from I_ADDR), and the initial sample address is decremented. Again, these two events can occur at the same time. The next step is to load the first sample and the first coefficient into the X and Y registers of the MAC, as well as increment the sample address register and the coefficient address register. The next group of transfers specify loading a product into P, a new sample into X, a new coefficient into Y, and incrementing the address registers. At this point a loop is entered, which adds the new product to the running sum, loads a new sample and a new coefficient, and increments the address registers. This continues until the process is done, which will occur when the final sample-coefficient product has been added to the running sum and is available at the inputs of the output register. This condition is checked simply by counting the number of operations, and when the result is ready moving on to the next transaction. The final transfer moves the newly calculated value to the output register (OUT), at which point control returns to the beginning to start over again.

If all of the transfers in the RTL occur instantaneously, then there is no problem with the system. However, in real systems each of the actions identified in the RTL takes a nonzero amount of time to accomplish. One of the challenges of the designer is to create a control section that will manipulate the signals in such a way that the transfers maintain the appearance of the simultaneity specified in the RTL. To prepare to design such a control section, we will create a state diagram that identifies the action of the RTL. We will see that this state diagram can be used to directly implement an appropriate control section. This state diagram is shown in Figure 5.7. It is called a preliminary state diagram, because it will be modified slightly before the actual implementation of the control section. The states that do not specify any work are added for timing purposes, and we will discuss them in connection with the actual implementation.

When the RTL description of operations and the state diagram are ready, the designer must complete the details of the data path block diagram by choosing the exact parts to be used in the system and identifying the control signals required on those parts to perform the work. Figure 5.8 shows the system of Figure 5.5 with the parts specified and the control signals identified. Note that, although the same parts are used both for the coefficient memory address register and the sample memory address register, the control lines needed are not the same. Both address registers need clocks, so that signal is shown for both blocks. However, the coefficient MAR needs to be cleared but not loaded, and the sample MAR needs to be loaded but not cleared. These differences are evident in the control signals included in Figure 5.8. Note also that control lines of the components that do not need to be manipulated during the computation are not identified in the diagram. It is assumed that the designer has studied the specifications of the components and made provisions for the other signals. Some of these will be grounded, others tied to a high level, and so on.

To summarize, the creation of the data path can be done in a manner that is relatively independent from the choice of a control mechanism for a digital system. The designer must first become familiar with the work required of the system. This includes the operations needed, the limitations of the data representation and manipulation methods, the order of events, and other considerations.

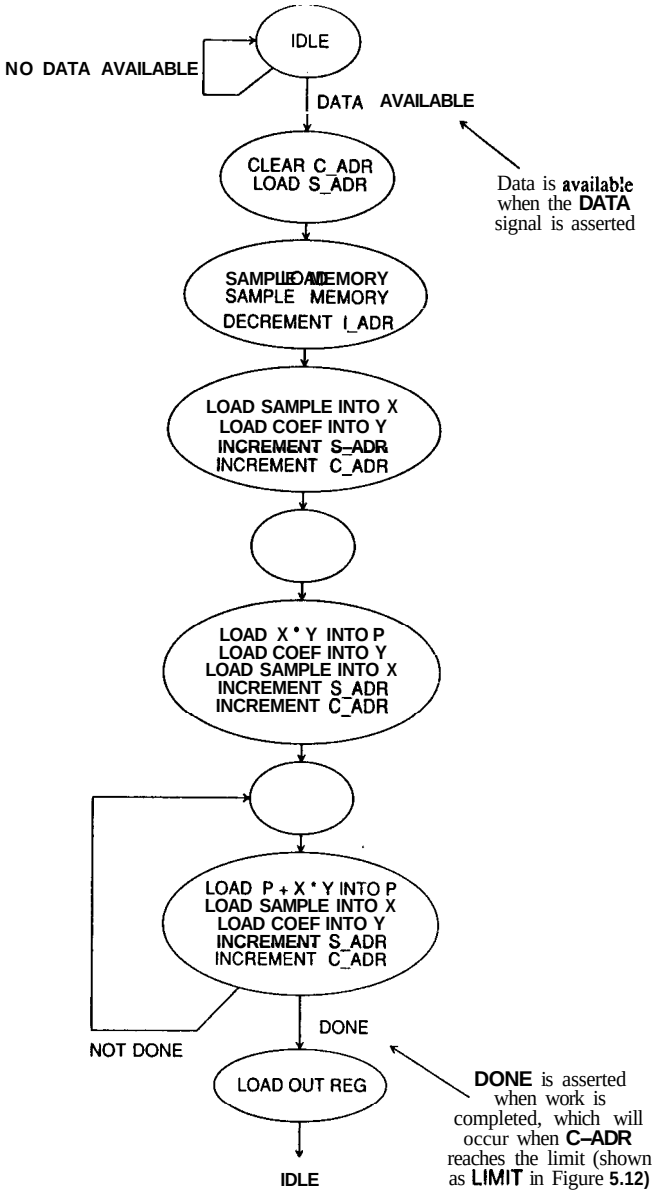


Figure 5.7. Preliminary State Diagram for Finite Impulse Response System.

With the system and device specifications in mind, the designer then organizes appropriate devices in such a way that the necessary data manipulations can be performed and the system constraints can be satisfied. The flow of information

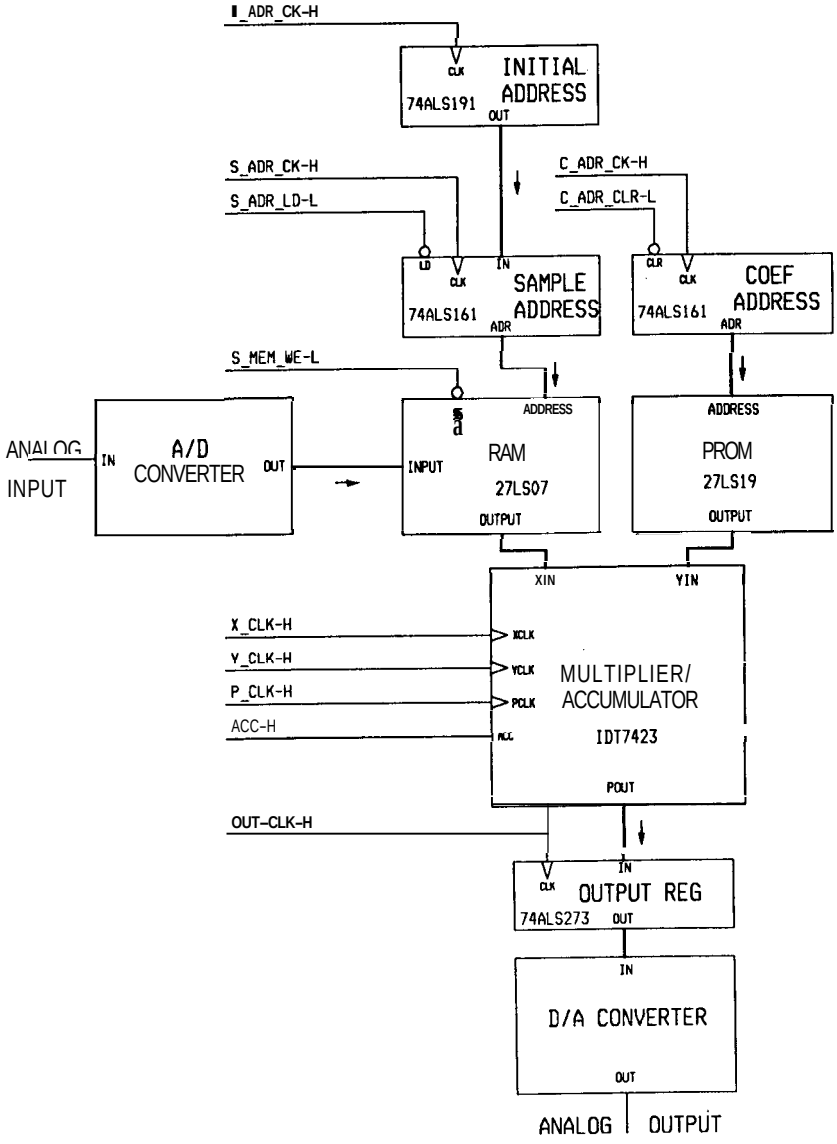


Figure 5.8. Detailed Data Path Block Diagram for Finite Impulse Response System.

within the system is then identified with register transfer specifications, state diagrams, and any other design aid that can provide insight into the operation of the system. Finally, the components are identified and the control lines of those components identified so that the detailed design of the control section can be performed.

5.3. A Simple State Machine Controller

Once the problem is understood to the point that a detailed data path block diagram and a preliminary state diagram are available, then the design of the control section can proceed. The classical approach would begin by creating a detailed state diagram, then a detailed enumeration of all possible state and input combinations. This would be translated into flip-flop excitation tables, state tables, next state and output truth tables, and logical equations for the appropriate signals. These would then be implemented with random logic, and, if all of the steps were correctly followed, the circuit should do the necessary control work. We present here a method that follows the same basic steps as the classical approach, but that is relatively simple to understand and implement. First, the state diagram is expanded as necessary to include the appropriate assertion levels for the control lines of the detailed data path block diagram. Then the system is mapped directly onto the Moore model of Figure 5.3. The simplicity of the implementation has some advantages and disadvantages, as we shall see.

A state diagram as shown in Figure 5.7 indicates the order in which events should occur to produce the desired results, but the details necessary for the control signals are missing. The designer must be sufficiently familiar with the parts being used so that the assertion of the control lines will be handled correctly. We now examine the primitive state diagram and the detailed data path block diagram in order to derive a correct and complete state diagram.

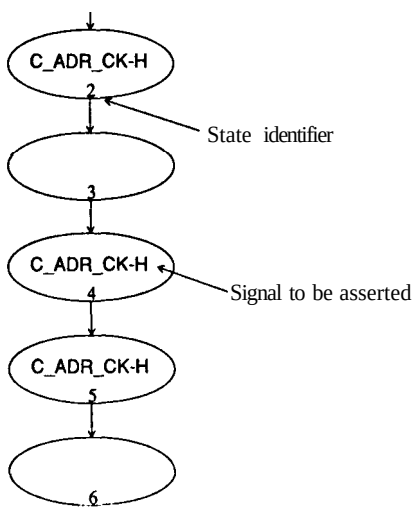
One observation concerning the state diagram of Figure 5.7 is that there are nine states in it, and to represent all of the states would require 4 bits of state information. One of the first steps of a design procedure is to attempt to reduce the number of states, if feasible, so that the number of bits required to represent the state is at a minimum. Two states in the state diagram appear to be unused, since no work is called out in these states. These states are useful, however, since they play a part in forming the control signals. Asserting signals in some states and not in others results in levels and edges that do the actual work of the system. A designer must visualize the desired behavior of the signals and create state sequences to produce that behavior.

In state diagrams we will identify signals to be asserted by naming them in the states in which they are active. The asserted level of the signal is identified by the use of polarized mnemonics included with the signal name. This is demonstrated by the segment of a state diagram shown in Figure 5.9. Five different states are indicated in the figure, and the system moves from state to state without any branching. Each state time corresponds to a single cycle of the system clock (SYS_CLK-H). In this fragment of a state diagram a single signal is called for in three diffcrcnt states (C_ADR_CK I:), and in each of those states it will be asserted, as shown by the waveform included in the figure. This signal is included on the detailed block diagram of the FIR filter implementatio for clock. ing (incrementing) the coefficient address register. However, even though this signal is asserted in three different states in Figure 5.9, the register would only be incremented by two. The implementation calls for a counter that is activated by a rising edge on the clock line, and as seen by the waveform of the figure, there are only two rising edges on C_ADR_CK-H. Thus, a designer must be aware of the shape of signal waveforms which will result from specifying assertion of the signals in a state diagram. A signal can be asserted for a single state time (C_ADR_CK-H in State 2), or a signal can last for many clock cycles (C_ADR_CK-H in States 4 and 5). We will later examine additional methods for creating control signals with state machines.

As shown by the signal waveform of Figure 5.9, removing the "empty" states in the preliminary state diagram would result in an incorrect function for the system. The states cause the signals that control the clocking of the address registers and the loading of the registers of the DAC to become unasserted, so that the proper edges are created when the signals are asserted in the following states. Thus, these states are needed, and another method must be used to try to reduce the total number of states in the state diagram.

The observation we now make is that there is some redundancy in the state diagram: if there is a method of accomplishing "LOAD PRODUCT INTO Z" and "ADD PRODUCT INTO Z" with the same signal, then two of the states can be combined. A careful examination of the specifications for the multiplied accumulator indicates that the function of the PCLK pin is determined by the level of the ACC line at the time that the X and Y registers are loaded. Thus, the desired behavior of the circuit will be obtained if the ACC line is low for the loading of the first values into X and Y, and high thereafter. This will allow combining of the appropriate states from the initial state diagram.

A detailed state diagram can now be created by identifying the desired behavior from the initial state diagram and specifying the signal assertions which



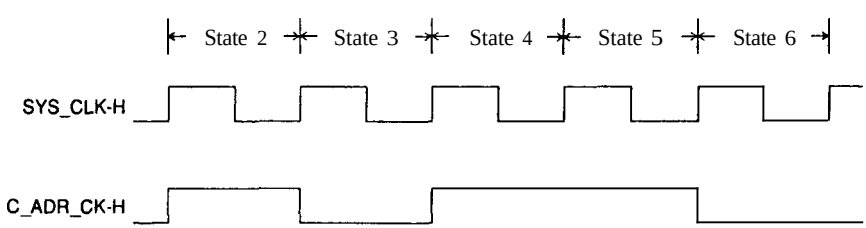


Figure 5.9. State Diagram Segment with Signal Assertion.

will cause that behavior. The new state diagram is given in Figure 5.10, and we will now explain in detail the signal assertions identified there. Two signals identified in Figure 5.10 are controlled by **SET-RESET** flip-flops to allow one behavior in one portion of the state diagram and another in a different portion of the state diagram. These signals are the **S_ADR_LD-L** line and the **ACC-H** line. The S_ADR_LD-L line is asserted by a signal in State **0 (SET–SA–LD-L)** to allow

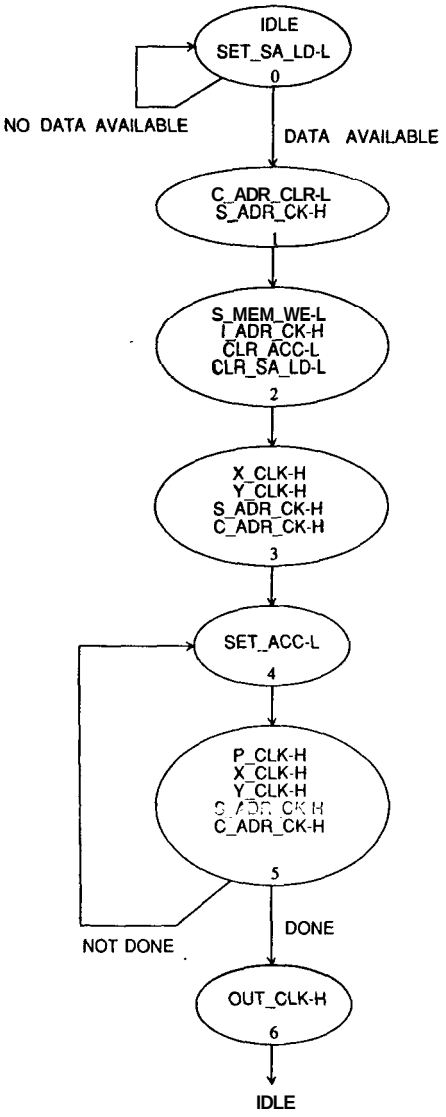


Figure 5.10. Detailed State Diagram for FIR Filter.

loading of the sample address register when its clock line is asserted in State 1. It is reset in State 2 (by CLR_SA_LD-L) to allow the address to increment when the clock line is asserted later. The ACC-H line is cleared in State 2 (by CLR_ACC-L) to set up the load of the product register. It is set in State 4 (by SET–ACC-L) to allow accumulation of results after the initial product load. We now consider each of the states, and the signal assertions needed for the process:

- ***State 0*** is the idle state; the SET_SA_LD-L signal is asserted to set up conditions for loading the sample address register, which will be accomplished in State 1.
- ***State 1*** should clear the coefficient address register and load the sample address register from the initial address register. The clear of the coefficient address register is accomplished by asserting C_ADR_CLR-L. The loading of the sample address register requires that the sample address load line be asserted, and then the clock line is asserted. The load was asserted in State 0; the clock is asserted in this state.
- ***State 2*** causes three things to happen. The S_MEM_WE-L line is asserted to write the sample into the sample memory (the appropriate address was loaded in State 1). The initial address register is decremented by asserting I_ADR_CK-H. And the product load condition is set up by asserting CLR_ACC-L.
- ***State 3*** causes load of the sample (X_CLK-H) and the coefficient (Y_CLK-H) into the MAC, then increments the two addresses (S_ADR_CK-H, C_ADR_CK-H).
- ***State 4*** sets up the accumulate condition for the product register in the multiplier/accumulator chip by asserting SET–ACC-L.
- ***State 5*** is where all of the work is done in steady state. The first time the state is entered, the assertion of P_CLK-H causes the product register to be loaded with $X \times Y$. Subsequent assertions of P_CLK-H load the product register with $X \times Y + P$. Samples and coefficients are loaded by asserting X_CLK-H and Y_CLK-H. The addresses are incremented by asserting S_ADR_CK-H and C_ADR_CK-H. The net result is that values are loaded and addresses incremented to look at the next values. The use of positive edge triggered devices assures that the current values are loaded before they change; the change will occur some time later because of propagation delays in the address registers and the memories themselves.
- ***State 6*** causes the output register to be filled by asserting OUT_CLK-H.

When the state numbers have been assigned to the state diagram, we are ready to map the controller onto the Moore machine. We will do this as shown in Figure 5.11. The present state register holds the current state of the system. The next state logic looks at the present state and the inputs and selects the next state. As shown in the figure, the logic blocks in the next state logic are multiplexers; the inputs to the multiplexers are chosen to select the correct next state from the current state. So the outputs of the multiplexers can be specified as shown in Table 5.1. The two signals included in the table have not yet been identified. The first is DATA-H, which is a flag from the A/D converter identifying that we have new data to process. A possible arrangement for this flag is shown in Figure 5.12(a). Here the end of conversion signal from the A/D converter causes a flip-flop to be set; the flip-flop is cleared by the same signal that clears the coefficient address. The second signal is DONE-H, which is asserted when the required number of iterations have been completed. We could create a new counter for

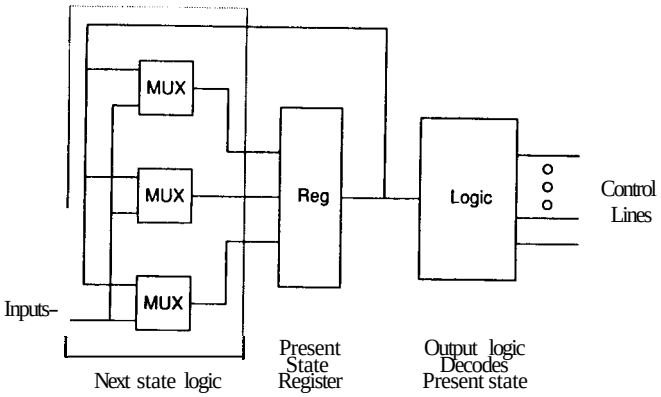


Figure 5.11. Implementation of Control System for FIR Filter.

Table 5.1. Next State Multiplexer Specifications.

	MUX 2	*MUX 1*	*MUX 0*
State 0	0	0	**DATA-H**
State 1	0	1	**0**
State 2	0	**1**	1
State 3	I	0	***0***
State 4	**1**	0	**1**
State 5	1	**DONE-H**	**0**
State 6	0	0	**0**

this, but that counter would duplicate the numbers used as the coefficient addresses. Therefore, Figure **5.12(b)** shows a comparator connected to generate a **DONE-H** signal using the numbers available from the coefficient address.

The entries in Table 5.1 specify the inputs needed for the multiplexers for the state machine. The resulting circuit is shown in Figure 5.13. Figure **5.13(a)** shows the present state register and the next state circuitry; Figure **5.13(b)** shows the decode of the present state register to generate the necessary control signals.

Several observations should be made at this point. The first is that the method described above is simple and direct, and easily applicable to state machines with up to **32** states. Larger state machines have been constructed using this method, but the number of parts involved becomes unwieldy. The simplicity of the technique allows ideas to be tested quickly; changes are easily made by moving a few wires on the inputs of the multiplexers. The basic feedback mechanism need not be disturbed. This ease of modification allows the circuit to be quickly changed to conform to the needs of the system. This basic system allows different design ideas to be implemented and and tried with a minimal investment of time and effort.

One of the tasks required when the implementation has been completed is to check out the system to verify that the unit functions correctly and that the signals are controlled in an appropriate manner. The checkout process must identify and

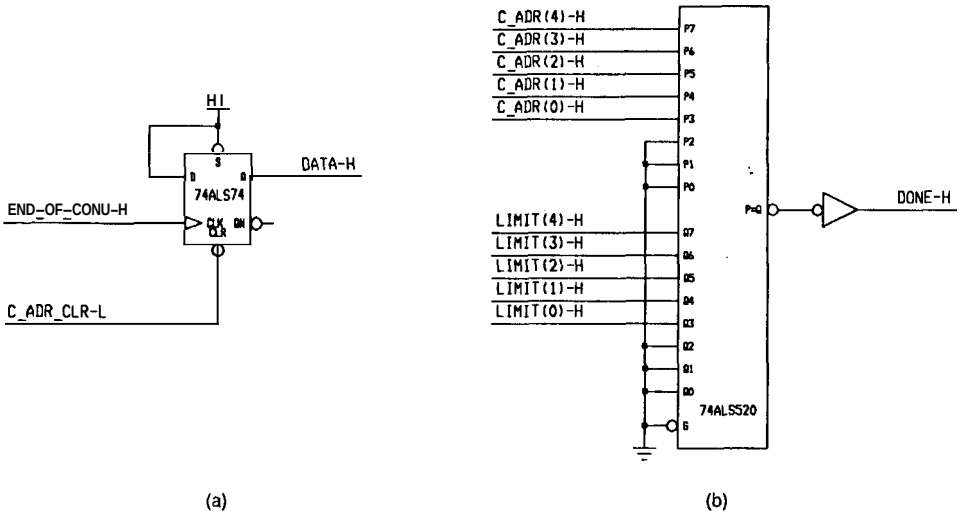


Figure 5.12. Control Signals for the State Machine.

remedy any errors which cause improper assertion sequences for the control signals. Generally errors will fall into one of two categories: either the system has wiring errors and the behavior does not follow the state diagram, or the implementation is correct but the state diagram is flawed because the designer did not thoroughly understand the system requirements. In either of these cases, modifications to a system designed in the method described above can be made easily, and the system can then be completed.

A second observation concerns the synchronization of input signals with the state machine. Figure **5.13(a)** shows that the **DATA-H** signal is not directly fed into the multiplexers, but that it is first synchronized with the system by sending it through a buffer register that is clocked with the system clock. In the example, the buffer register is the same device used as the present state register, since the device is not entirely utilized. But what is required is keeping the input synchronous with the system clock. If this provision is not made, then the inputs may change in a manner such that, when the system clock does occur, that the next state is changing and the result is an illegal state transition. If the inputs are not synchronized, the system will fail when changes on the input lines occur at the same time that the present state register is being clocked. Note also that the **DONE-H signal** is not buffered by a register. The reason for this is that the **DONE** signal changes synchronously with the system clock, and hence does not need the effect of the register.

Another observation deals with the generation of the output signals. As shown in Figure **5.13(b),** the present state is decoded to generate the appropriate signals for the system. Generally our concept is that as the system proceeds through the states identified by the state diagram, the lines of the decoder will become asserted at precisely the right time. However, since real devices contain real delays, and the delays can cause glitches, provisions must be made for the correct **operation** of the system. If the signals being activated are level sensitive,

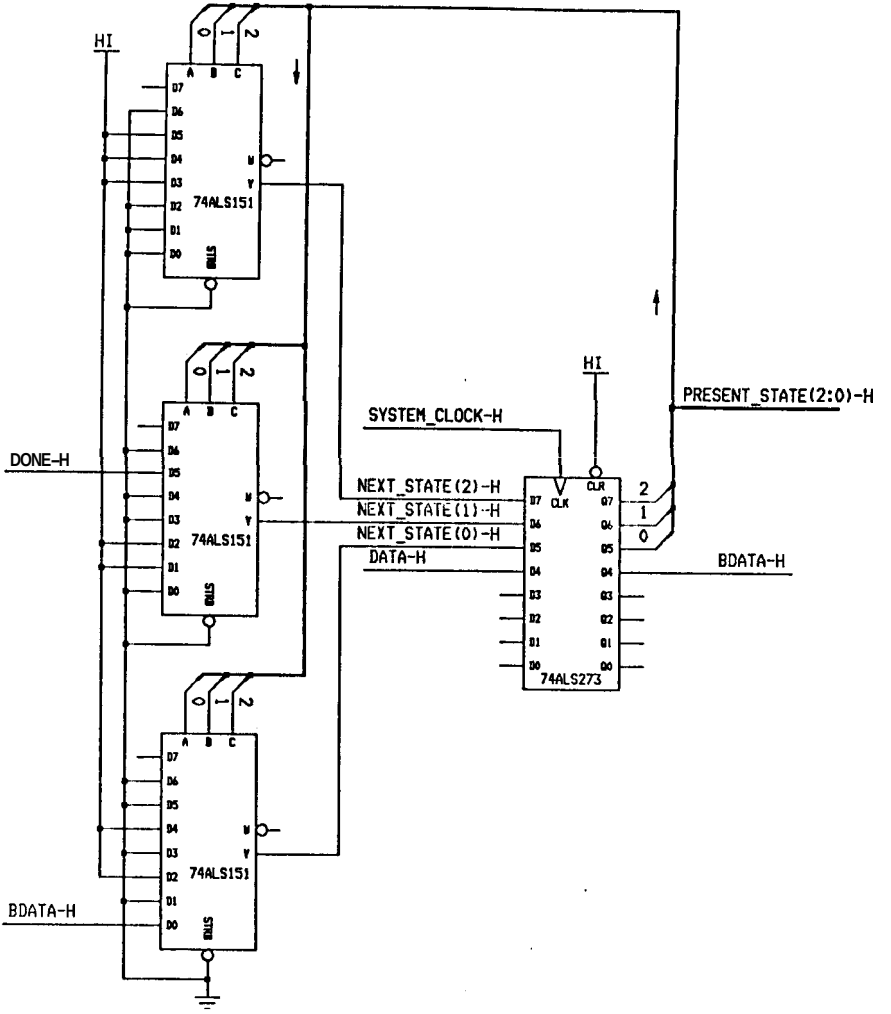


Figure 5.13(a). Present and Next State Logic for Control System.

then a glitch will not cause problems. However, if the signals are edge sensitive, as all of the clocks in our example are, then glitches on the control lines can cause problems. Figure 5.14(a) shows a decoder set up to demonstrate a number of possible combinations.

The problem of glitches on output lines is illustrated in Figure 5.14. The control line which is used in different ways in the example is the enable line of the decoder. If the decoder behaves as a perfect decoder, and no glitches occur on the output lines, then behavior similar to the waveform of Figure 5.7 can be obtained by always enabling the decoder. Figure 5.14(b) shows the results when

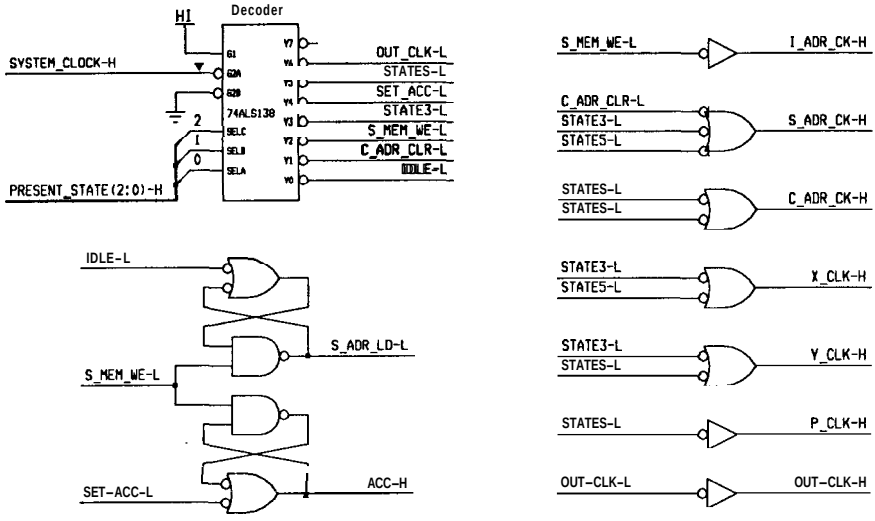


Figure **5.13(b).** Generation of Control Signals from Present State.

the enable line is always asserted: glitches occur on the output lines of the decoders. In Figure **5.14(c)** the enable has been tied to SYSTEM-CLOCK-H. The result is that the appropriate decoder output will be asserted only during the time that the system clock is low, which is the last half of the cycle. As can be seen from the figure, the assertion occurs half way through the cycle. This is the method utilized in the finite impulse response filter example. This is the reason that the ACC-H and **S_ADR_LD-L** lines are driven from flip-flops, since the decoder outputs are only asserted for half of the cycle.

The success of obtaining the last half of the cycle may prompt one to attempt to obtain the first half of the cycle by using the other phase of the clock.

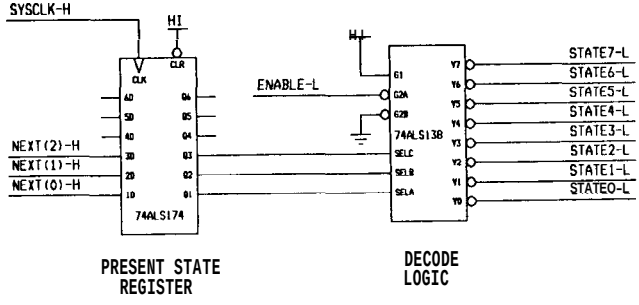


Figure **5.14(a).** Test Setup for Using State Decoder.

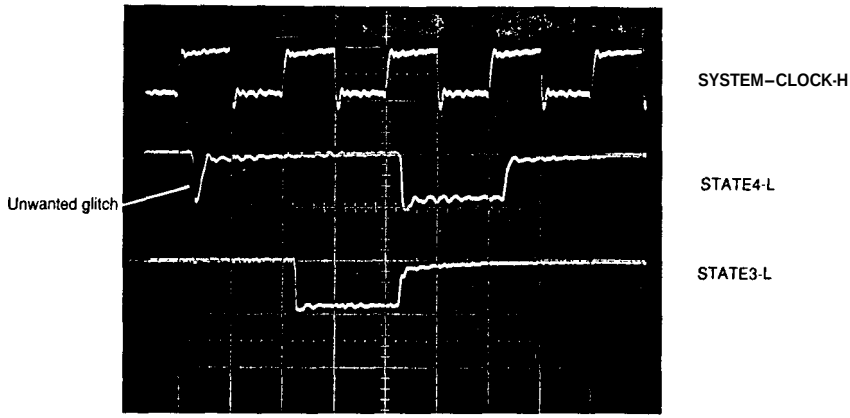


Figure 5.14(b). Waveform for Decoder with Enable Always Asserted.

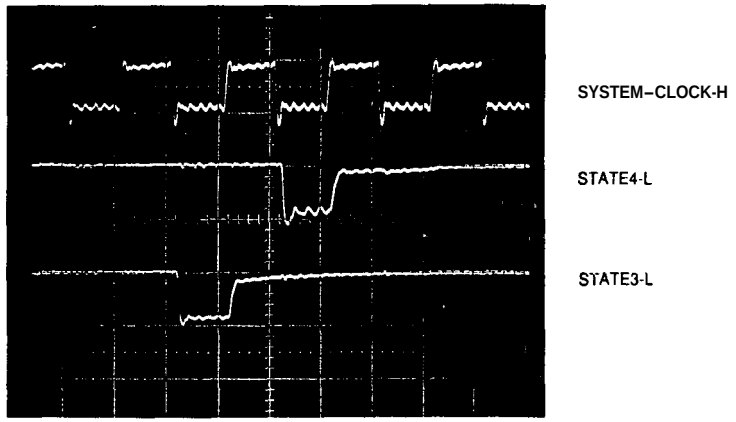


Figure 5.14(c). Waveform for Decoder with Enable Tied to SYSTEM_CLOCK-H.

The result is shown in Figure **5.14(d),** which indicates that unwanted pulses occur; this is a result of the propagation delay from clock assertion to change of decoder output. As can be seen from the waveforms of Figure 5.14, a number of options are available to a designer, and the merits of each option must be considered before selecting a design method.

The FIR filter example demonstrates some of the basic principles of controller design. It is imperative that the designer first understand the system specifications; this includes aspects often neglected, such as the implications of

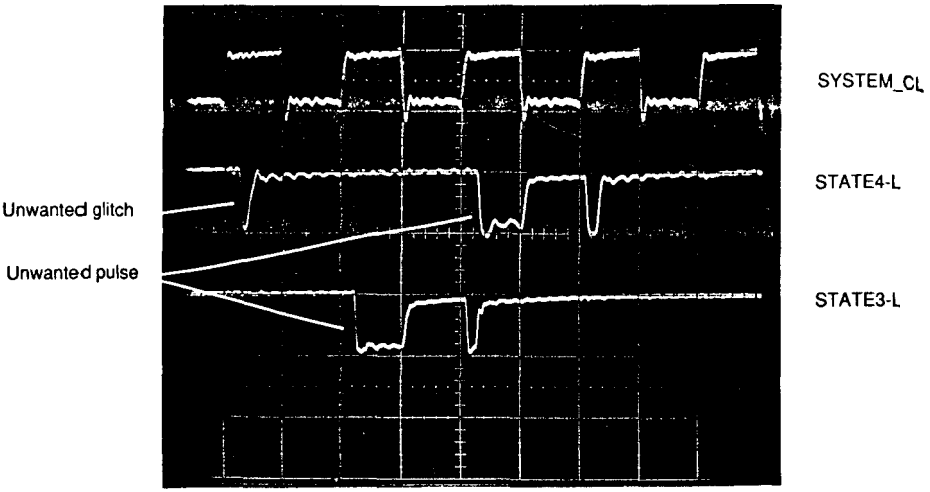


Figure 5.14(d). Waveform for Decoder with Enable Tied to SYSTEM–CLOCK-L.

the arithmetic methods, number of bits on the data path, and interaction protocols. The designer then generates a data path block diagram, an RTL description of the desired system behavior, and a preliminary state diagram. These tools assure the designer that the system specifications will be satisfied, and that the necessary data transfers can be made. When the block diagram is defined, the control signals of the components of the system are identified and labeled with appropriate polarized mnemonics. The state diagram can now be refined to specify the assertion of the control signals that will cause the desired work to be accomplished. The state diagram can then be mapped onto the Moore model to provide a working control system. The result is a system that will activate the control signals in the proper sequence to achieve the necessary results.

The state diagram approach is easy to understand, and it is also fairly easy to implement for small systems. We have shown the next state decode logic to be multiplexers; classical methods dictate the use of random logic. Manufacturers now provide registered PLAs (programmable logic arrays) that allow the designer to put both the present state register and the next state logic inside a single chip, which is then programmed to follow some specified state diagram. Outputs are handled in much the same way. One use of these controllers will be used in Section 5.6. However, historically other methods have been applied to the control systems of computers. We now look at some of these methods.

5.4. Sequential Systems with Individual Delays

As we have seen, the first step in any control design is to derive a block diagram that meets the system specifications, and then to identify on that block diagram the control lines needed. In this section we will look at an extremely simple computer, and use that machine to exemplify the delay method of sequential control

systems. The principles here are similar to those used in the state machine control of the previous section, but the application methods are slightly different. Rather than have the state of a system stored in a single register, and the state changes reflected by changes in the state number, the action of the delay type system is governed by a control pulse that traverses the elements of the control system.

The technique of individual delays described here has been used in the past for a number of computer systems, but is not widely used in new systems. However, in some systems constructed entirely within an integrated circuit **chip,** delay lines play a prominent part in generating control signals.

The block diagram for our example is given in Figure 5.15. The diagram shows a simple single address machine, with enough detail represented to illustrate the principles of this section. The diagram does not by any means represent a complete system, since a diagram of that complexity would be overwhelming. The data paths are patterned after some of the first computers: the connections arc

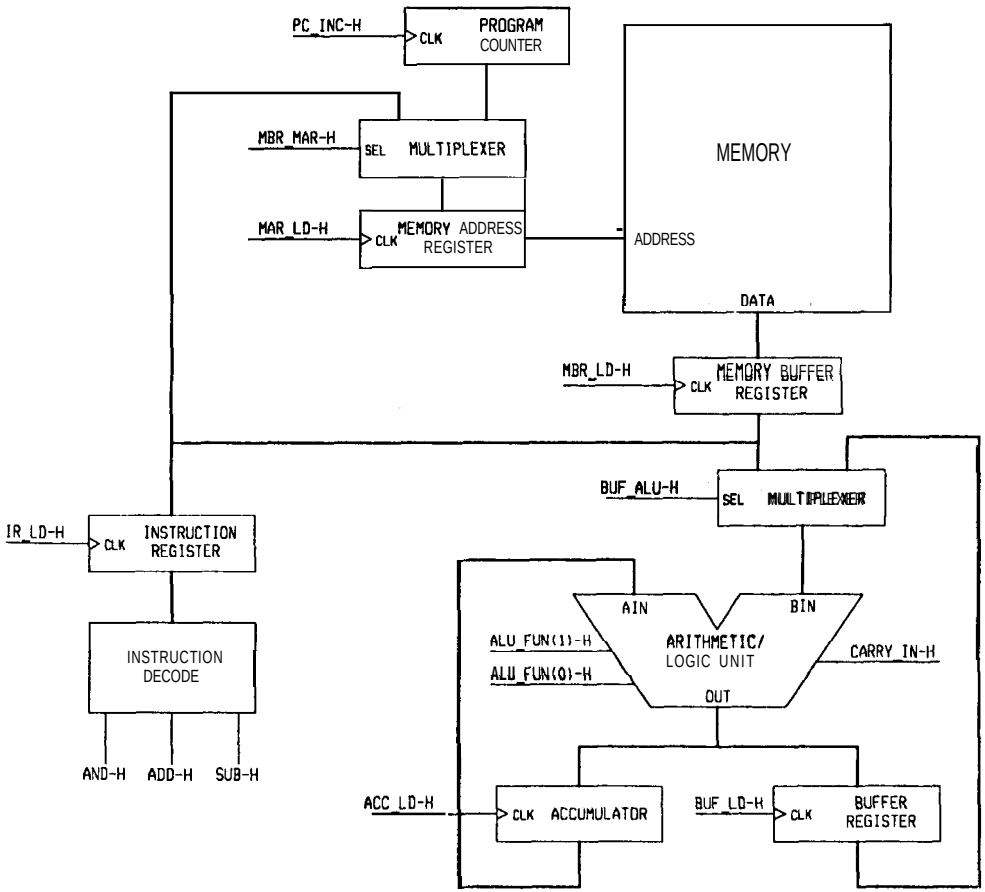


Figure 5.15. Block Diagram for a Simple Computer

basically point to point rather than bused. Note that the data paths are not complete, as exemplified by the fact that there is no path to the program counter.

The desired behavior for this example is to implement three simple instructions: ADD, SUBTRACT, and AND. All three of these instructions require two operands. Since this is a single address machine, one operand is found in the accumulator, and the other is found in memory at a location specified by the instruction. The task required of the control section is to cause the requested action on the data and leave the result in the accumulator.

As in the previous example, the first task is to create a suitable data path block diagram, which was given as part of the definition of the example. The designer then must arrange for the required action, utilizing the capabilities of the data path hardware. The hardware capabilities of this example include:

- ***Program counter:*** The content of this register identifies a location in memory where the instruction to be executed can be found. The process of instruction execution should increment this register to point at the next instruction. This can be accomplished by asserting **PC_INC-H.**
- ***Memory address register:*** This register holds an address to identify a location in the memory.
- ***MAR multiplexer:*** The multiplexer selects the source of information for the MAR. Normal operation is for the PC to be output to the MAR. However, when **MBR_MAR-H** is asserted, the address is obtained from the memory buffer register.
- ***Memory:*** The memory will provide to the memory buffer register the contents of the address specified by the memory address register within some specified delay. For this example we will assume that the delay is 200 nsec.
- ***Memory buffer register:*** For destructive readout memory technologies this register remembers the data just read so that it can be restored to the memory. In general, modem semiconductor memories do not need this capability.
- ***ALU multiplexer:*** This device selects the BIN operand of the **arithmetic/logic** unit. Normal operation selects the contents of the memory buffer register; when **BUF_ALU-H** is asserted, the ALU receives the contents of the buffer register.
- ***Buffer register:*** This register is used for internal operations that need a temporary storage location. It is not visible to assembly level programmers.
- ***Accumulator:*** This is the known register of the machine. All instructions that manipulate data will find information in this register, and instructions that produce data results will leave their information in this register.
- ***Arithmetic/logic unit:*** This functional unit is capable of some rudimentary actions, as specified by the following table:

ALU_FUN		*OUT Function*
0	0	Bitwise **AND** of AIN, BIN
0	1	Bitwise OR of **AIN, BIN**
1	0	Inverse of BIN
I	1	Binary **ADD** of AIN. BIN

This ALU has the characteristic that logical operations **(AND,** OR, INVERT) take 40 nsec to complete; the arithmetic operation (ADD) takes 80 nsec to complete.

- ***Instruction register:*** This register is used to hold the instruction during its execution.
- ***Instruction decode:*** The decode circuitry identifies the type of instruction to be performed. In this example there are only three, but generally there will be many instructions. The appropriate output line will be asserted to identify which of the instructions has been decoded.

In addition to the times specified for the ALU and memory functions, we will assume that register to register transfers require 40 nsec.

The designer utilizes knowledge of the data path connections and the capabilities of the components used on the data path to specify the required action of the control system. The first step is to identify the required register transfers, and for this example these transfers are given in RTL form in Table **5.2.** The table specifies the order in which the transfers are to be accomplished. Our task is now to take these transfers and implement them in hardware. The first step in this process is to generate a flow chart that identifies the required steps. The flow chart for these three instructions is given in Figure 5.16. Note that the flow chart identifies the signal assertions required to accomplish the transfers specified by Table **5.2,** as well as the delays necessary between the assertion of those signals. Also note that there is a one to one correspondence between the operations identified in Table **5.2** and the operations caused by the signal assertions identified in the flow chart.

To illustrate the process of instruction execution, we will examine the subtract instruction. A timing diagram showing the control lines involved in this instruction is shown in Figure **5.17.** The process begins by transferring the address of the instruction from the program counter into the memory address register with **MAR–LD-H.** Note that the multiplexer normally supplies this information to the **MAR,** so no action is required on the control lines of the multiplexer. The memory has a 200 nsec delay, so the **MBR_LD-H** signal is delayed by that amount after loading the **MAR.** The program counter is also incremented at the same time. The instruction register is loaded from the **MBR** 40 nsec later, since 40 nsec is required for register transfers; after a period of time for instruction decode, the **MBR_MAR** line is asserted so that the **MAR** receives its information from the **MBR.** A delay time later the **MAR–LD-H** line is asserted again, loading the address of the operand required for the operation. After the memory

Table 5.2. Register Transfers for Three Instructions.

Register Transfers for Example		
AND Instruction	***ADD Instruction***	**SUBTRACT *Instruction***
PC → MAR	PC → MAR	PC → MAR
M[MAR] → MBR	M[MAR] → MBR	M[MAR] → MBR
PC + 1 → PC MBR → IR	PC + 1 → PC MBR → IR	PC + 1 → PC MBR → IR
MBR → MAR	MBR → MAR	MBR → MAR
M[MAR] → MBR	M[MAR] → MBR	M[MAR] → MBR
MBR • ACC → ACC	MBR + ACC → ACC	¬ MBR → BUF
		BUF + ACC + 1 → ACC

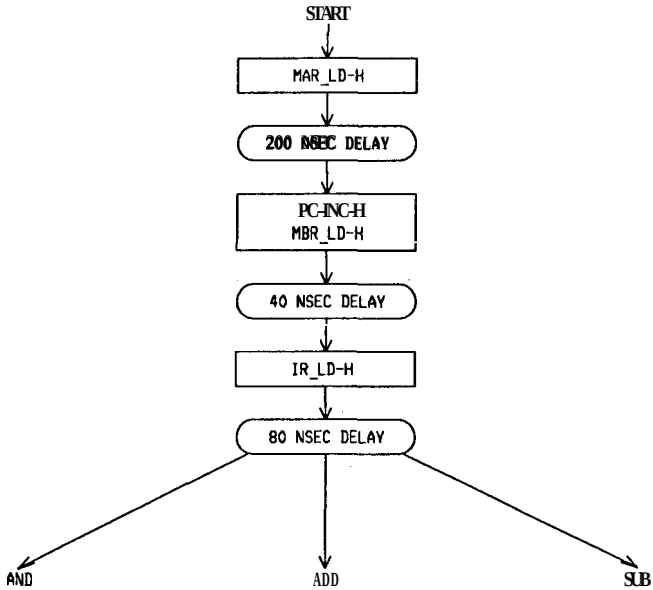


Figure 5.16(a). Row **Chart** for Delay Implementation of Three Instructions.

delay the **MBR** is loaded and the **MBR_MAR-H** line reset. Since the subtraction method specified calls for inversion of the **MBR**, the **ALU_FUN(1)-H** line is set high to present to the input of the buffer register the inverse of the **MBR**. This information is then loaded into the buffer register, and the **ALU** prepared for an addition operation. By also forcing the **carry** input to be a "I," the final operation is the desired subtraction, and after the required delay the **ACC_LD-H** line is asserted to load the information into the accumulator. The ***fetch-execute*** cycle then repeats itself, beginning with the assertion of the **MARLD-H** signal.

The flow diagram and timing diagram together specify the action to occur and the timing relationship between control signals. The individual delay method of sequencer design consists of directly implementing the flow diagram with delay elements. A delay element consists of either a semiconductor device or an analog equivalent that will accept a signal, usually a pulse, and delay the signal by a preset amount. In this example we need delays of 40 nsec for the register accesses — 80 nsec. 120 nsec. and 200 nsec. With these available. a designer matches the flow diagram with timing elements, and then uses logic gates and flip-flops to create the appropriate control signals. The delay elements for this example are shown in Figure 5.18(a), and the additional logic required is shown in Figure 5.18(b) and Figure 5.18(c).

The system would begin action by injecting into the delay network a single pulse at **RUN-H**. This would assert **START-H**, which in **turn** asserts **MAR-LD-H**. After a delay of 200 nsec, **I_FETCHED-H** is asserted. This causes the assertion of both **PC-INC-H** and **MBR_LD-H**. Another delay element is used to place the required time between the load of the **MBR** and the assertion of **IR_LOAD-H**. The **AND** gates then direct the pulse down the appropriate set of delays, depending on the instruction decoded. And so the process continues, with the pulse traversing

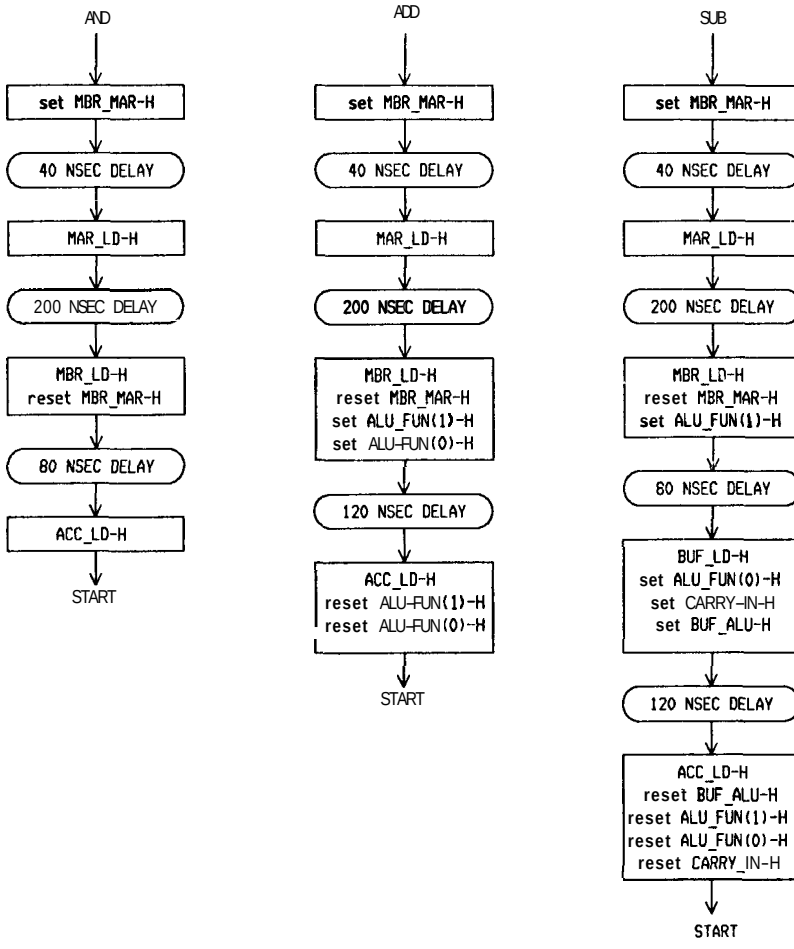


Figure **5.16(b).** *(con!)* Flow **Chart** for **Delay** Implementation of Three Instructions.

the delay network and doing work as required. The control signals are created by tapping the appropriate spots in the delay network, as specified by the flow diagram. For example, **ACC_LD-H** is created by **ORing** the signals from the **AND.** ADD, or SUB delay sections together. For signals that need to remain set **for** lengths of time, the flip-flop arrangement shown for **MAR_MBR-H** can be used. The signal is set when it is first needed, and then reset when it is no longer needed. This allows both pulses and levels to be used in the system.

The preceding example has shown that systems can be designed in a straightforward manner using delay elements and gates to cause the appropriate action. The data path block diagram identifies the control points that need to be activated, and the flow diagram and timing diagram specify the actions and delays to take place to accomplish the appropriate tasks. This example can easily be extrapolated to include other instructions: the flow diagram will require additional

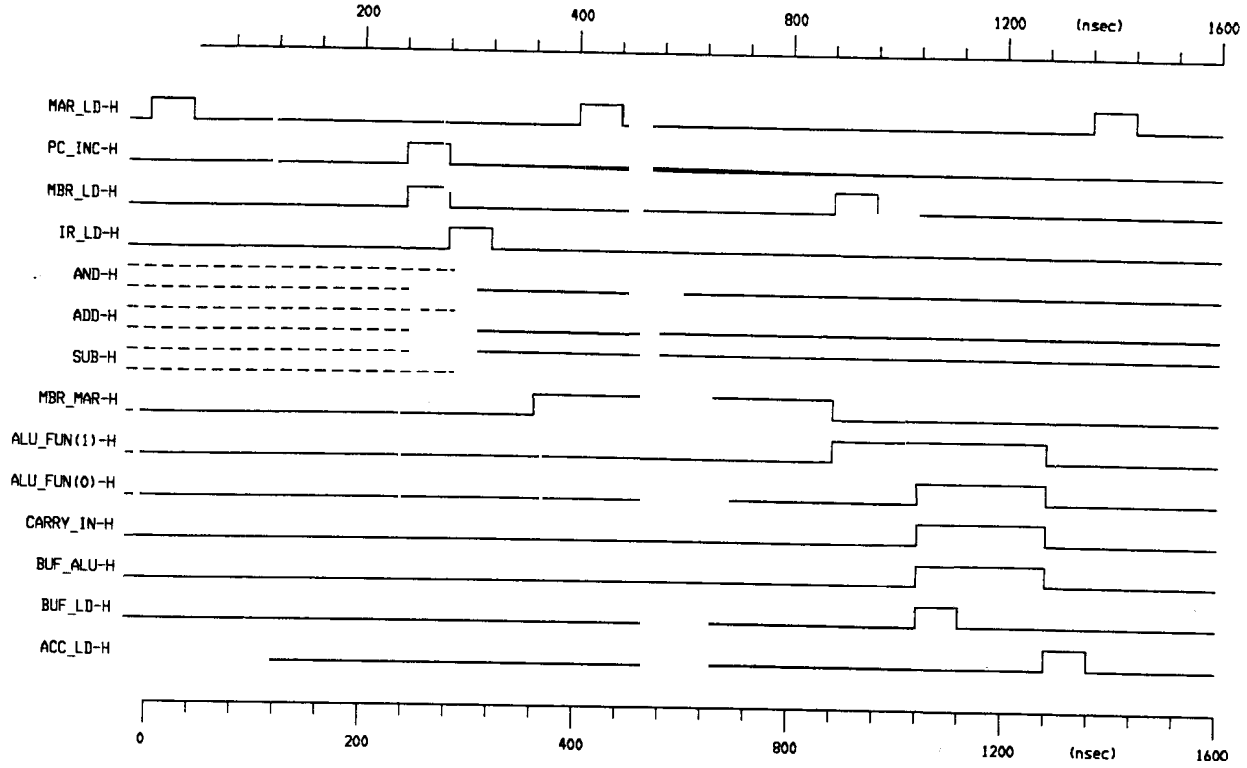


Figure 5.17. Timing Diagram for Control Signals: Subtract Instruction.

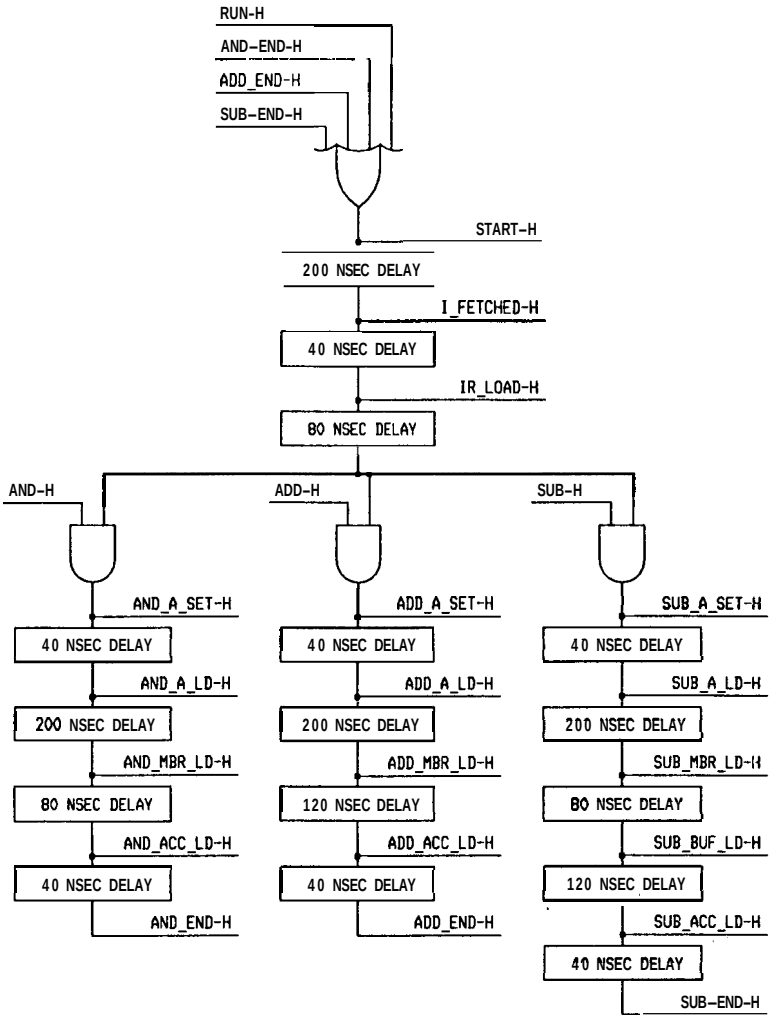


Figure **5.18(a).** Delay Elements **Needed** for Simple Machine.

branches in the decode section, and additional register transfer level specifications will identify the work required for arithmetic or procedural instructions. For example, not all instructions will require action from the ALU, and other data paths will be required for jumps and other activity.

This method of design has an advantage in that the control can be tuned to provide the fastest action possible. That is, if it is known that the ALU will do an **AND** action in **38** nsec, then the 38 nsec delay can be placed in the appropriate spot in the system, and the **AND** instruction **will** take 2 nsec less than an OR instruction. But offsetting the speed advantages **are** some of the practical problems. The fidelity of the pulse as it travels through the system must be carefully

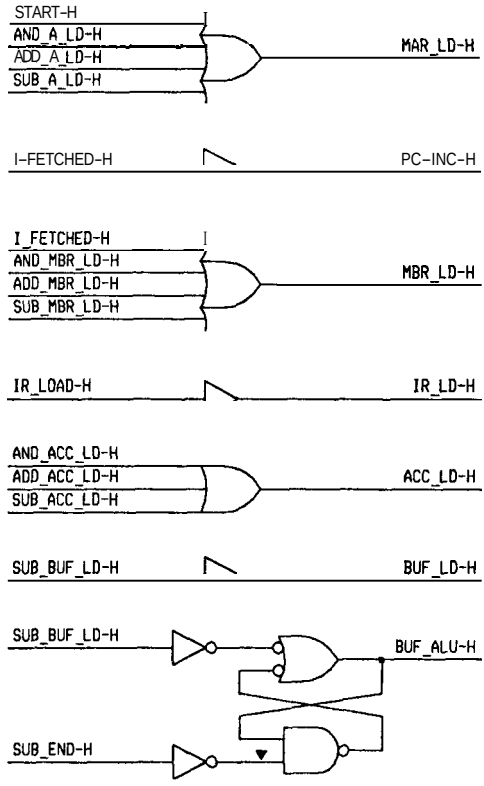


Figure **5.18(b).** Creation of Control Signals for Delay Method.

maintained, and this can cause additional problems. The system must be carefully designed to prevent spurious pulses from entering the network; an interesting error mode is when two pulses are traversing the system simultaneously.

This method allows a straightforward combination of data path block diagram, flow diagram, and timing information to result in a tunable, high performance control system. The control system provides both pulse and level capabilities, and can be easily modified either by changing the delays or by including other points in the delay network in the creation of control signals. Many of these characteristics are also evident in the shift register method of control design.

5.5. Sequential Systems Using Shift Register Timing

The concepts of the shift register timing method for control design follow closely those of the individual delay method. The data path block diagram is used to identify the control signals, the flow diagram identifies the register transfers and other work that need to **be** done, and the timing diagram specifies the interaction of the control signals required to accomplish the work. However, the timing

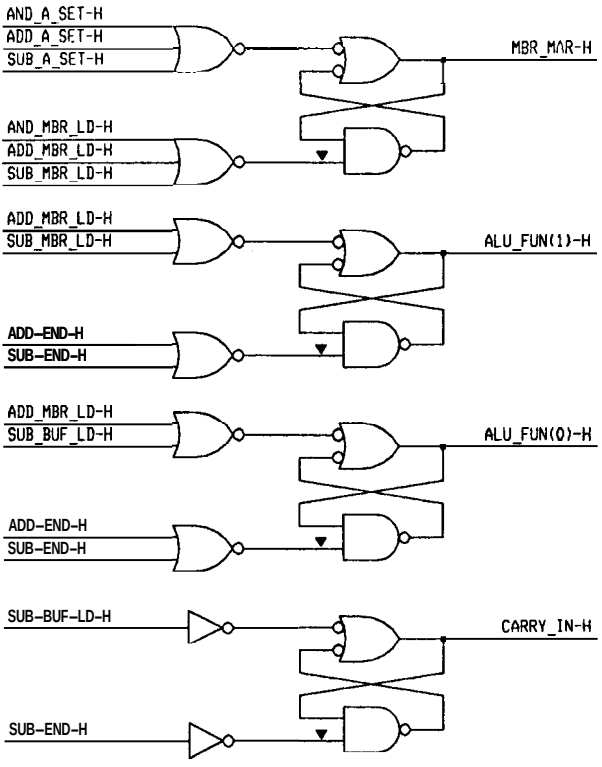


Figure **5.18(c).** *(cont)* Creation of Control Signals for Delay Method.

diagram must now represent events that occur at multiples of the system clock. That is, the delays of a flow chart such as Figure 5.16 must all be multiples of the system clock. The preceding example was chosen so that all of the delays were multiples of 40 nsec — so that will be the assumed system clock rate for this section. The concept for the shift register method is to identify the work to be done, and then to create the proper waveforms by using gates to harness a pulse proceeding down a shift register.

The creation of the timing action is accomplished by the action of a shift register. One such arrangement is shown in Figure 5.19. The desired pulse action is initiated by asserting START–PULSE-L. On the next clock pulse the signal **PULSE_0-H** will be asserted. If a pulse duration of one cycle is desired, the STOP–PULSE-L control line can be created by inverting **PULSE_0-H.** Thereafter, on each leading edge of the system clock the pulse "moves" down the shift register. The resulting pattern is shown in Figure 5.20, called Method One. The pulses depicted for Method One form a precise timing capability for the system. If an event is to occur 80 nsec after initiation of the instruction, then **PULSE_2-H** can be used to cause the event. However, if a control line needs to be asserted for more than one clock period, then more than one time period is needed. That is, if a signal is to be asserted from 80 to **160** nsec after initiation of the instruction,

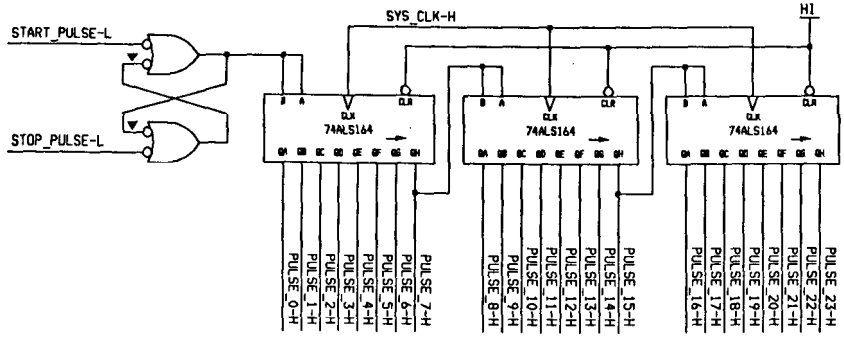


Figure 5.19. Pulse Creation with Shift Registers.

then the signal can be created by ORing PULSE–2-H and PULSE–3-H. This will indeed result in a signal of duration 80 nsec; however, there may be a glitch in the signal caused by the timing difference of deasserting PULSE–2-H and asserting PULSE–3-H.

One way to get around the problem of glitches on the control lines is to use set-reset flip-flops as we did with the delay line method. Another solution to the problem is to use overlapping pulses, as shown in Method Two of Figure 5.20. Pulses with a length of two system clock periods can easily be created by using the inverse of PULSE–I-H to be STOP–PULSE-L. When these signals are ORed together, the resulting signal is free of glitches caused by the hazards associated with pulse assertion.

The similarities between this method and the individual delay method are apparent from the approaches both take in implementing the control signals. The principal difference is that one method uses individual delays and a pulse that traverses a control network to accomplish work, while the other method achieves the correct timing relationships by the use of measured delays in a shift register. Both methods create the control signals by gating appropriate delayed values with the necessary enable conditions. The result is a system that asserts the control signals needed to accomplish the necessary work.

An example of gating for the shift register method for the system of the previous section is given in Figure 5.21. The gates shown are derived directly from the timing and flow diagrams. The MAR–LD-H signal is always asserted at PULSE–0 time, or it is asserted at PULSE–10 if the instruction is an AND. ADD. or SUBtract instruction. For this example, this is the entire collection of instructions. so the AND and OR gates are superfluous. However, if a number of other instructions were included in the system, then the gates would be needed. The PC_INC-H instruction always occurs at PULSE–6 time, so no additional gating is needed. The ALU_FUN(1)-H signal is asserted during PULSE–16. PULSE–17. or PULSE–18. if the instruction is an ADD instruction, or during PULSE–16. PULSE–17, PULSE–IS, PULSE–19, or PULSE–20. if the instruction is a SUBtract instruction. It is not asserted during an AND instruction. The other control signals are created in a similar fashion. Note that the STOP–PULSE-L signal occurs at PULSE–1 time, resulting in overlapped pulse operation. Also note that the START–PULSE-L

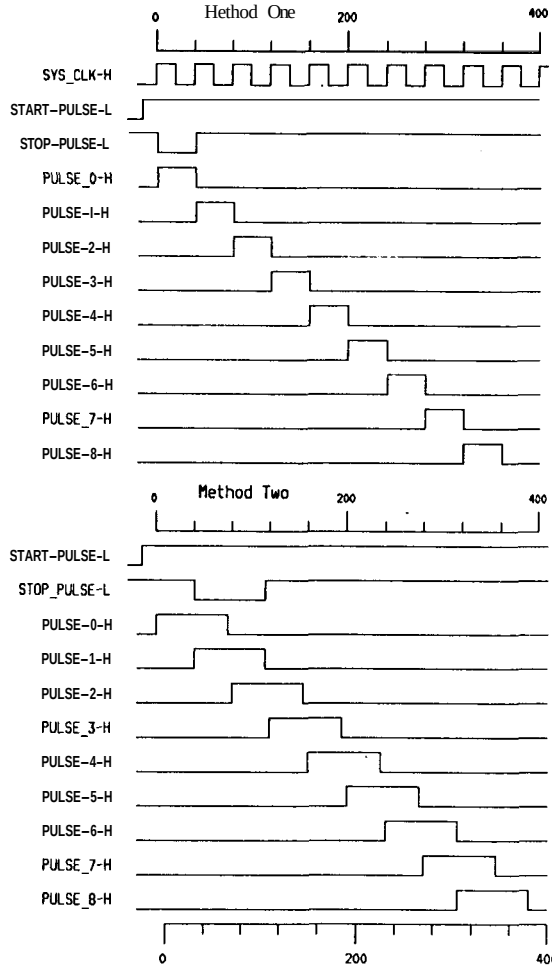


Figure 5.20. Timing Diagrams for the Shift Register Method.

signal occurs at different times for the different instructions, and that initialization comes from some external circuitry.

Both the delay method and the shift register method provide straightforward approaches of building control circuitry, mapping the information from the flow and timing diagrams directly into hardware. Both methods allow the designer flexibility to implement the necessary signals to match the constraints of

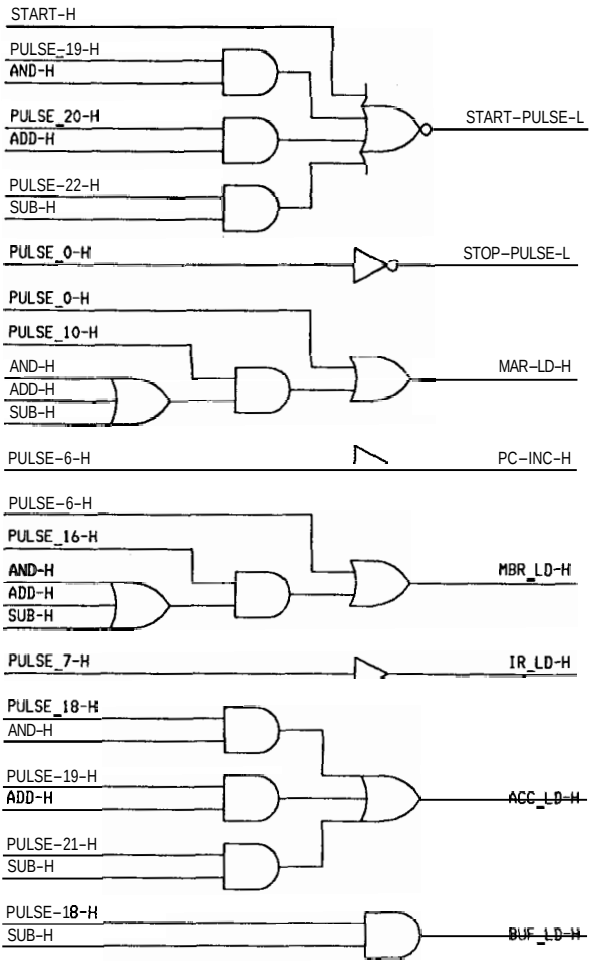


Figure 5.21(a). Control Signal Generation with the Shift Register Method.

technology and application. And both methods have interesting error modes when more than one signal enters the delay network/shift register. Nonetheless, both of these methods have been utilized in the design of many types of digital equipment. However, perhaps the most extensively utilized control design method in recent years is microcode.

5.6. Microcode Controllers: A Regular Control Structure

In 1951 Wilkes presented a paper in which he suggested that the design of control systems was entirely too complicated. He went on to suggest that the process could be greatly simplified by the use of a regular method for making decisions

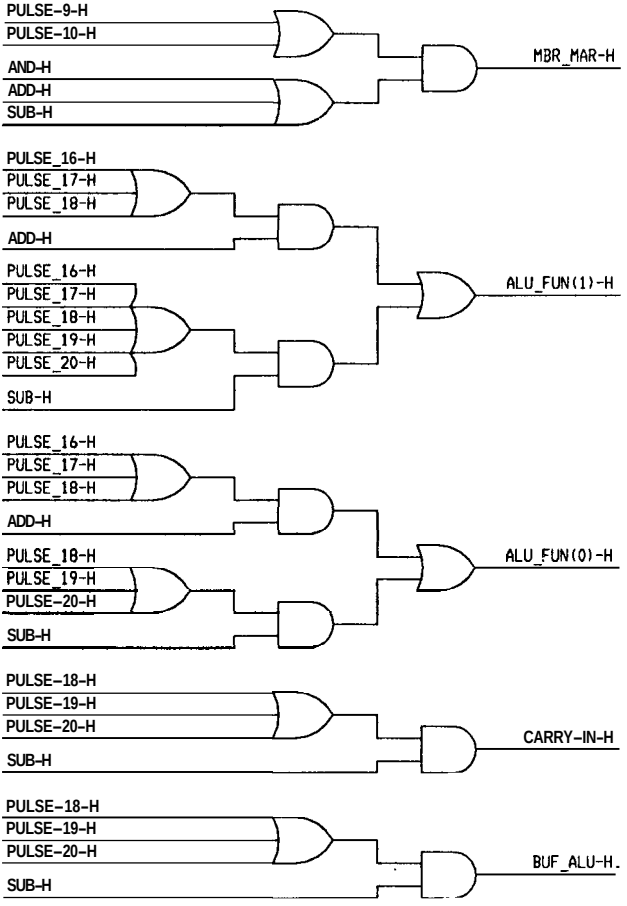


Figure 5.21(b). *(cont)* Control Signal Generation with the Shift Register Method.

concerning the next thing to do and what control signals to assert. The heart of this method was a high speed memory element needed to remember the appropriate sequence of information. However, at that time the memory technology was not as fast as random logic, nor as readily available. Hence, for many years Wilkes' suggestions went unheeded. Instead, designers utilized classical techniques, as well as the delay line and shift register methods, to implement sequential controllers. However, in the mid-1960s memory technology advanced to the point that it was an attractive alternative to use high speed memory to govern the action of a control system. We will introduce the method by taking another look at state machine control, and transfer the state machine ideas to the use of microcode.

We begin our examination of memory-based control methods by reorganizing the block diagram of Figure 5.15. The same basic components are utilized,

but the organization is changed. The reason for changing the block diagram will become apparent as we discuss the implementation methods of this section. The main organizational change is the inclusion of a single data path that is utilized by all of the components. This single bus organization is very useful in systems where universal communication is desirable. Each component can transfer information to any other component; however, only one value can be transferred in a clock period. The component required to accomplish this is a bus driver, which isolates the register outputs from the bus except when the information in that particular register is required. At that time, the bus driver is enabled and information from the register is made available to the other elements on the bus.

Transferring the contents of the program counter to the memory address register is achieved by asserting PC_BUS-L to place the contents of the program counter on the bus, and then after a time required for propagation delay, settling time, and setup time, MAR_LD-H is asserted to load the information into the MAR. One method of implementation is to make the various registers from simple register devices such as the '273, and the drivers from tri-state drivers, such as '244. For situations where the data is not necessary except to drive the bus, such as the buffer register, it is possible to obtain both register and driver in a single package, such as the '574. However, not all registers can take advantage of this capability, since the output of the accumulator is always needed at AIN of the ALU, and the value in the memory address register is required at the memory.

As with the other control implementations, our first requirement is a complete data path block diagram. with control points identified. This is given in Figure 5.22. We can now generate a state diagram that identifies the assertions required in order to accomplish the desired results. These results have already been identified by the flow chart given in Figure 5.16; we can now generate a state diagram to do the same work. One such state diagram is given in Figure 5.23. This state diagram illustrates some interesting points, and represents a fairly conservative approach to system design. Let us consider the methods illustrated by Figure 5.23, and then consider some alternatives.

The method used for transferring information across the bus is illustrated in the first two states, which cause the MAR to be loaded with the contents of the PC. In State A the signal PC_BUS-L is asserted, which causes the contents of the program counter to be placed on the bus. This same signal is asserted in State B, which guarantees that the value will be present during that state also. The loading of the MAR is caused by the assertion of MAR_LD-H in State B; this signal causes the register to accept the information while the bus is held steady by the PC–BUS line. The relationship between these signals is shown in Figure 5.24. The method described in the state diagram, and shown pictorially in Figure 5.24, requires two states, and guarantees that the data is loaded into the MAR at the beginning of State B. The same work can be accomplished by generating both the MAR signal and the PC_BUS signal simultaneously, as shown in the alternative method. The key to success of this method is that the register is loaded on the rising edge of the MAR–LD line. Thus, for the duration of State X the PC_BUS signal is causing the data to be placed on the bus, and sufficient time is allotted for the delay in that process, as well as the setup time on the inputs to the MAR. Then when the low-to-high edge occurs on the MAR–LD line at the end of the state, the data available is loaded into the register. For most logic families (LS, ALS, AS, etc.), the delay in turning off the driver is sufficient to guarantee that the data is stable long enough to be correctly loaded into the MAR. This alternative method requires only one state to transfer the information, instead of the two states shown for

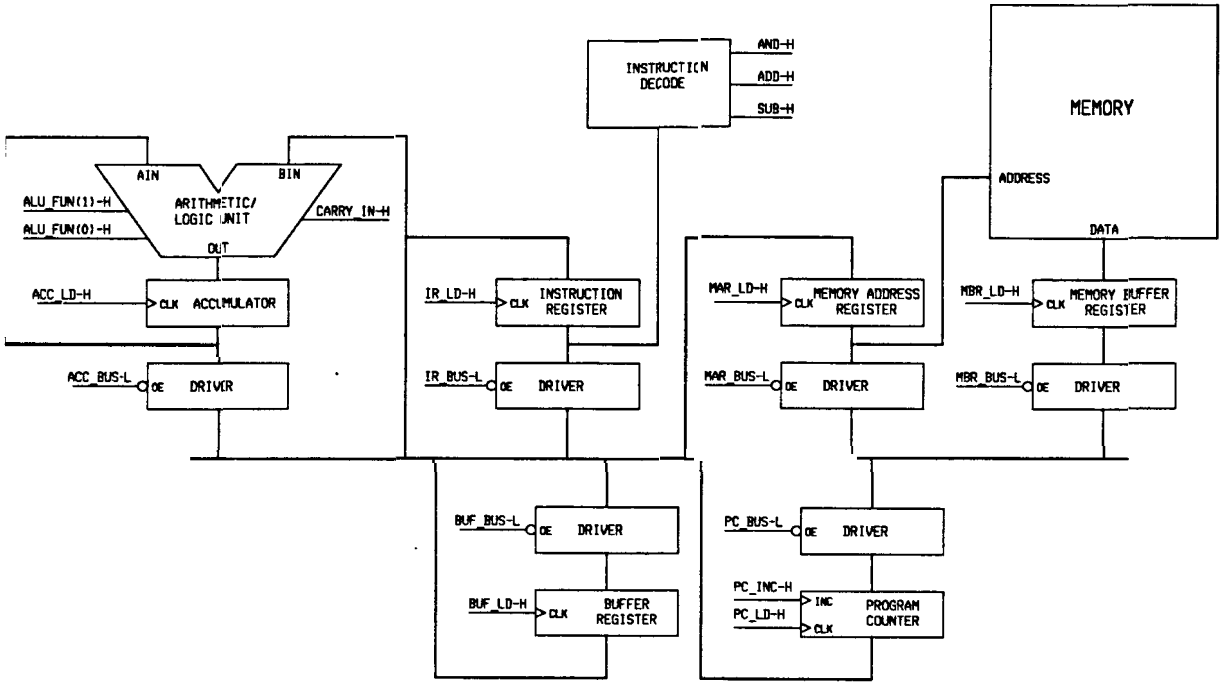


Figure 5.22. Block Diagram of Processor with Single Bus Organization.

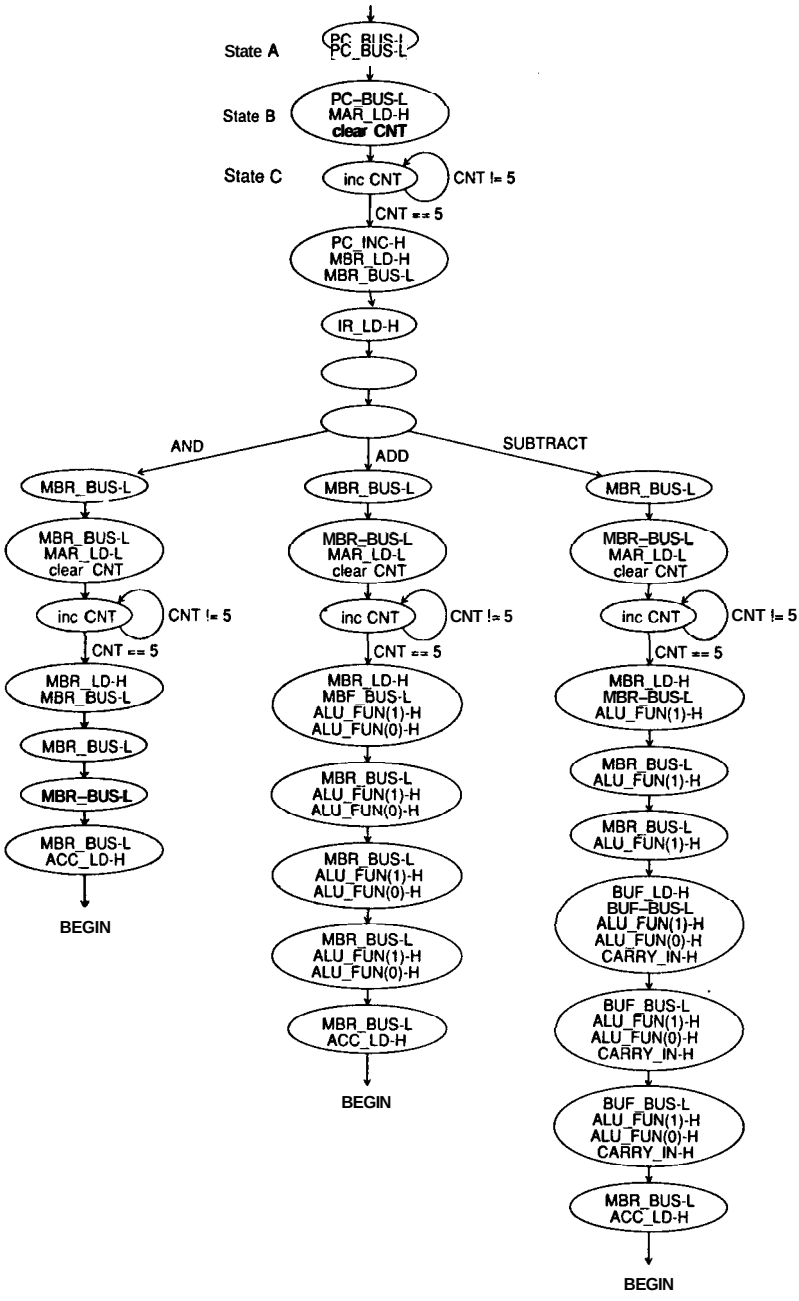


Figure 5.23. State Diagram for Single Bus Processor.

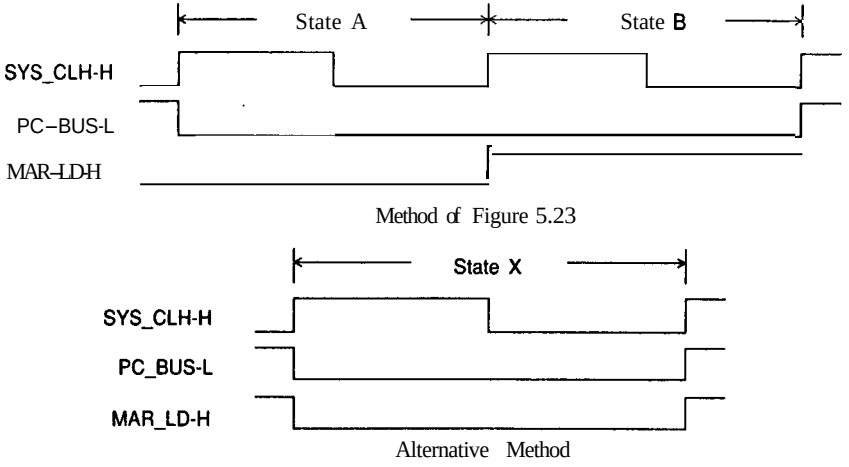


Figure 5.24. Timing for Loading MAR from PC.

Figure 5.23. One caution with this method of information transfer is that the designer must ascertain that the data has been stable in the loaded register for a sufficiently long period to guarantee desired results for the next operatian. That is. the propagation delay, from clock assertion to data available, must be accounted for in any subsequent data manipulation.

This method is applicable to registers and other edge-triggered devices whose clock lines are driven directly from signals generated by the state machine. Another method to achieve this result is to use devices with separate clock and enable lines. One such device is the 74F550, shown in Figure 5.25. This register

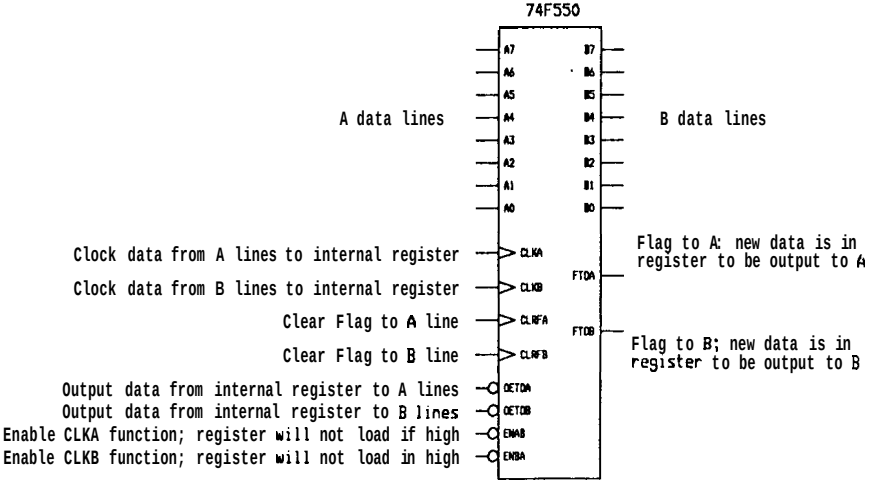


Figure 5.25. Register with Separate Clock and Enable Lines.

has a clock enable control line, which controls the effect of the clock. This allows the clock line to be connected directly to the system clock, and then the line that needs to be asserted by the control section is the enable line. This is particularly useful for systems in which all events are to happen at precisely the same time, and that time is defined by the rising edge of the system clock. A number of devices utilize this strategy for their operation, including registers (2950, 2952, '550, etc), arithmetic units (2903), and controllers.

Another example of the separate clock/enable function is demonstrated by the use of counters in this system. The 200 nsec delay required by the memory is obtained by waiting for five state times before proceeding. This wait time is governed by a counter similar to those used in the example of Section 5.2. The control design used in that section caused control signals (specifically, clock lines of counters) to be asserted when the action was needed. Another method to achieve the same result is to use counters which will increment only when enabled, even though a clock signal is present at the clock input. The counters will increment only when the enable line is asserted, and the enable line is controlled by the state machine. This is the method which is illustrated in State B and State C of Figure 5.23. The counter is cleared in State B, and then State C calls for incrementing the counter. This cannot be accomplished if the clock signal is fed directly from the decode of the state, since the state does not change. (As pointed out earlier, ANDing the clock signal with the system clock would result in a pulsating clock line.) However, if the State C signal is utilized to enable a counter, then the desired result is obtained. For the '161 of Section 5.2, the action can be obtained by using a signal generated in State C to assert the Enable P line of the counters.

Other delays are implemented by repeating the action of one state in another state. The 40, 80 and 120 nsec delays can be obtained by using one, two, or three states. Thus, delays can either be obtained by staying in one state for a predetermined number of system clock times, or by using multiple states, assuring that the required signals are asserted within those states.

The state diagram of Figure 5.23 is specifically constructed to follow the flow diagram of Figure 5.16. No attempt has been made to try to save on the number of states utilized. An examination of the state diagram reveals that there are some duplications, specifically in the area of obtaining the operand of the instruction. One method of reducing the number of states would be to delay decoding of the instruction until the operand has been obtained. This results in a system that partially decodes instructions at appropriate times to attempt to minimize the number of states. For example, the system under consideration always requires an operand for each instruction, but in a real system instructions such as "increment" or "clear" affect only the accumulator, and do not need to obtain another operand. Thus, the organization of the system hardware, the complexity of the instruction set, and the goals of the system all influence the designer in the creation of the state diagram that describes the control algorithms of the system.

Using classical methods, or those described in Section 5.2, we can implement a control section that operates as described by the state diagram of Figure 5.23. A block diagram of such an implementation is shown in Figure 5.26. The current state of the system is stored in a register labeled "Present State Register." The next state logic uses the current state, the instruction, and the start signal to select the appropriate next state. In the direct implementation method of Section 5.2, this logic consists of multiplexers and perhaps some minimal logic. With classical methods, this would be some type of random logic implementation.

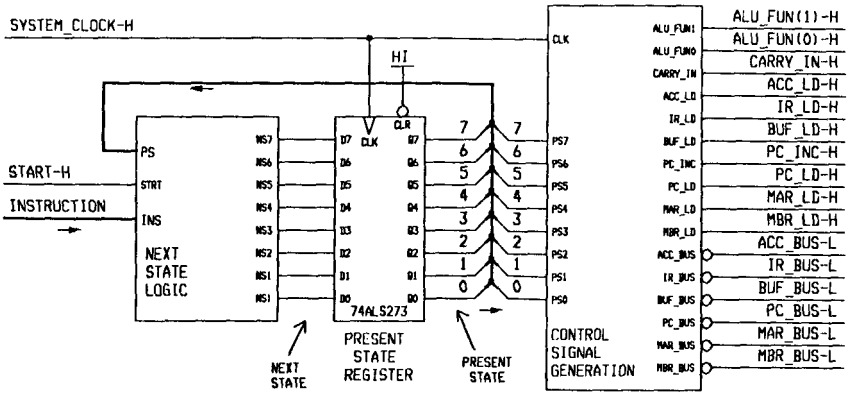


Figure 5.26. **Block** Diagram of State Machine Controller for Simple Computer.

Regardless of the implementation method, every clock period a new determination is made as to the next state, and if the implementation is correct, the state diagram of Figure **5.23** will **be** followed. The control signals are generated by decoding the present state; these signals may or may not include the system clock in their implementation. The following observation can **be** made concerning the control signal generation: the signals asserted at any given time are functions only of the present state (and clock), and the signals to be asserted in any given state are determined during the design process. Since the signal assertions are set up at design time, the same information used to select the appropriate next state (present state and inputs) can also be used to determine the signals that will be asserted at that time. Therefore, during the same period that the next state is determined, the appropriate signal assertions for that state can also be determined. This leads to the implementation shown in Figure **5.27.**

The block diagram of a system controller as shown in Figure **5.27** is extremely simple. The next state logic determines the state to which the system will proceed from the present state, based on the present state and the external inputs. At the same time, this same information will be used by the next state control logic block to determine the control signals to be asserted in the next state. As stated earlier, this information is available at design time, and will not change during the useful life of the product. Both the state infonnation and the control lines will **be** held in registers, so that transitions on control signals will occur at the same time that the state changes. If it is deemed desirable to do so, some control signals can be conditioned with the clock to create appropriate timing pulses. This arrangement eliminates the use of a present state decoder for generation of control signals, since all of the signal generation is determined prior to the active edge of the clock.

The logic utilized by a system for the **next** state logic and next state control logic blocks can be created by any appropriate means open to a designer. But it is instructive to note that it need not be random logic nor the multiplexer arrangement presented earlier. Some manufacturers build devices specifically designed to do this function, and they provide means to create the appropriate logic, depending on the mechanism used for implementation of the device. Figure **5.28** shows a block diagram representation of the 82S105, which is called a "Field

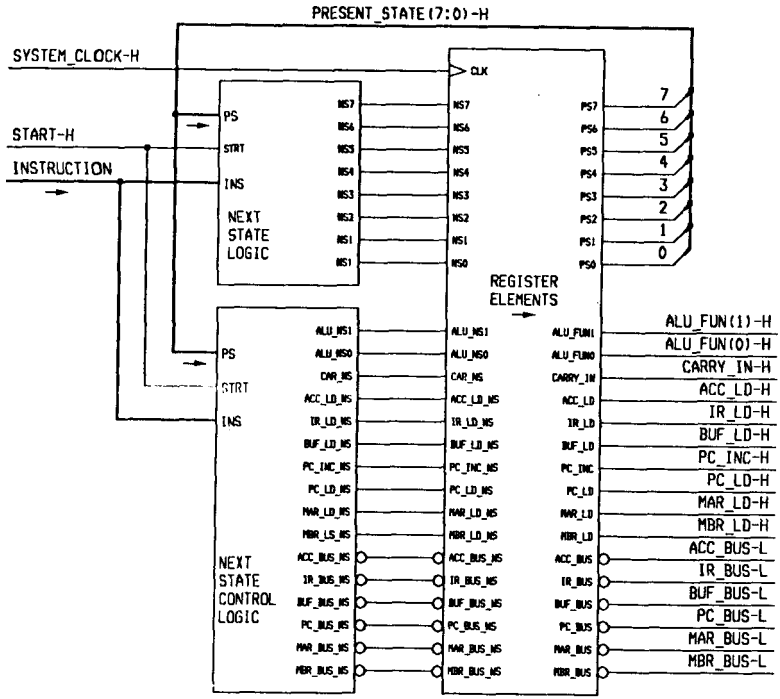


Figure 5.27. Block Diagram of State Machine Controller Combining Generation of Next State and Control Signals.

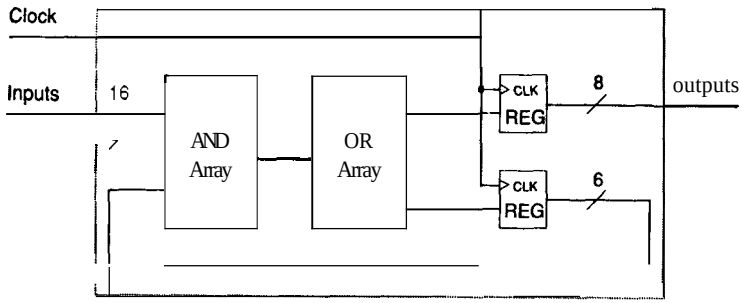


Figure 5.28. Internal Makeup of a Field Programmable Logic Sequencer

Programmable Logic Sequencer." This device allows 16 external inputs; in addition there is a reset/output enable line (function is defined at time of programming). The device has eight outputs, all of which are registered so that the outputs will change only after a clock transition. Internal to the device are six feedback lines; this allows creation of a state machine with up to 64 states. The determination of the next state and the output levels is accomplished by a programmable AND/OR array; the limitation is that the device is capable of only 48

AND terms and 14 OR terms. This places some limits to the complexity of the state machines that can be implemented by the device, but a variety of very useful controllers is feasible. A large number of similar devices can be utilized to implement sequencers, such as registered PALs, state machine controllers, and registered PLAs.

One "feature" of this type of unit is that the feedback variables are internal to the device. This is a benefit in that the speed is not hampered by going off chip, which leads to a higher clock rate. Note that the inputs will need to be synchronized to the system in some way. The drawback to this feature is that the state variables are not available to the user to aid in the debug process. Thus, the only way to ascertain the state of the machine is to observe the output pins. The designer must be careful about his assumptions concerning the correctness of the machine during checkout. Nevertheless, programming aids. available from both manufacturers and third party vendors, greatly enhance the ability of the designer to create a correct system.

As a result of making state machine devices compact and easy to generate, many of the designs utilized in recent digital systems are created with a number of individual state machines. In these systems, each state machine is a single IC created to perform its own task, and the units function together to control the system. Thus, rather than have a single controller to control all of the action of the system, the control is divided between smaller units, and these units each activate a subset of the control lines. An example of this is described in Section **6.5**, where one state machine controls the action of an interface module, while different state machine controls the signals used to interface to the bus.

The use of logic arrays for the next state and output generation allows creation of a variety of useful devices, but does not permit arbitrarily complex systems. Also, since the feedback is internal to the device, the number of outputs is limited to those available from that chip. One way to expand the use of this technique is to use memory instead of AND/OR arrays for the logic blocks. That is, if we consider the correct next state/output information as a pattern of ones and zeros stored in a memory, and the address of the correct pattern is formed by the feedback variables and inputs, then all combinations of states and input variables are possible. The memory utilized in this arrangement can be ROM or PROM, and the number of input variables and feedback variables can be increased by adding more memory chips. For example, one such device is the 27S55, a PROM with eight registered outputs and 4,096 locations, which requires 12 address lines. Three of these devices clocked together would give 24 outputs, and these could be used in any combination required by a design. A system implementing the state diagram of Figure **5.23** would require six feedback variables; these would form six of the 12 address lines on each device. That allows six other lines to be used for the start signal and instruction lines, as well as any other inputs required in the system. The 24 outputs would then be utilized for six feedback variables and 18 control lines; we have identified 16 lines in the block diagram of Figure **5.22.**

This arrangement has several practical advantages. The controller is completely contained in three 24 pin devices, requiring about 1.4 square inches of board space. The fact that it is programmable allows a designer to try different state diagrams or implementation ideas by merely changing the devices, not physically changing any wires. The net result is a very versatile system controller of arbitrary complexity. No limitations have been made concerning the complexity of the state diagram, nor concerning the number of states in which control signals can be asserted.

At this point we will pause in our discussion of control system construction to identify a technique that can be beneficial in the checkout and maintenance of sequential machines. One basic model for a sequential system was given in Figure **5.2,** and this basic model is reflected in the diagram of the sequencer shown in Figure **5.28.** One of the basic problems facing system designers is the checkout of equipment that has been constructed. For simple designs made from individual gates, or for any system in which access to major system components is readily available, a brute force method of checkout is often utilized. With this method. the outputs of the system are observed under the necessary conditions of input and history to check for correctness. If improper behavior of output signals is observed, then the logic required to generate those signals is meticulously checked for correcmess. The problem may lie in improper implementation of the logic, or the problem may concern an improper design based on flawed assumptions about the problem to be solved and the available inputs. Thus, not only the logic network, but also the design of the logic, must be checked for errors.

If the system to be checked is a sequential IC, such as that shown in Figure **5.28,** then it is difficult, if not impossible, to test the actual logic. Access is needed for controllability and observability: we need to control the inputs to the system, and we need to be able to observe the outputs of the system. Control over the external inputs of Figure **5.28** is easily obtained, but control over the feedback variables is not readily available, since they exist solely internal to the device. Similarly, the outputs of the chip can be readily observed, but the contents of the internal state register is not available to the external to the device. One of the techniques used to provide both controllability and observability is called the scan technique, which is used to provide access to the internal registers of a svstem.

The basic idea of the scan technique is to provide a method for controlling and observing the contents of the registers internal to a system. Rather than providing additional pins for all of the desired points, the internal registers are configured as either a normal register or as a shift register. In normal operation, the registers behave as we have discussed to this point: at the active edge of the clock, the register is loaded with either the output information or the next state information. In diagnostic mode, the registers are reconfigured as a single serial shift register, and activating the clock shifts out the bits in a serial fashion. Thus, all of the internal register bits can be observed. Similarly, as the bits are shifted out, new bits can be input to the system to allow external control of the levels inside the device.

The application of the idea requires some modifications to a system. This is indicated by modifying the organization of the device shown in Figure **5.28** to include the elements shown in Figure 5.29. The additional lines required are minimal: a control line to normal or diagnostic operation, a serial input, and a serial output. A good description of the technique and its application is available in **[McC186].** Some manufacturers provide integrated circuits with this capability built into the register elements. Advanced Micro Devices refers the additional registers in their devices as "shadow registers," but the idea remains the same: provide ability to control and observe needed points in a system [see **AMD88, Lee87,** and **Schm87].** This need not apply only to integrated circuits, but can be used in any sequential module. **IBM** utilizes this technique in a number of systems, where it is known **as** level sensitive scan design (LSSD) **[TeSw82].**

The state machines implemented to this point have been created to match a timing constraint given in the problem statement. One of the questions to be

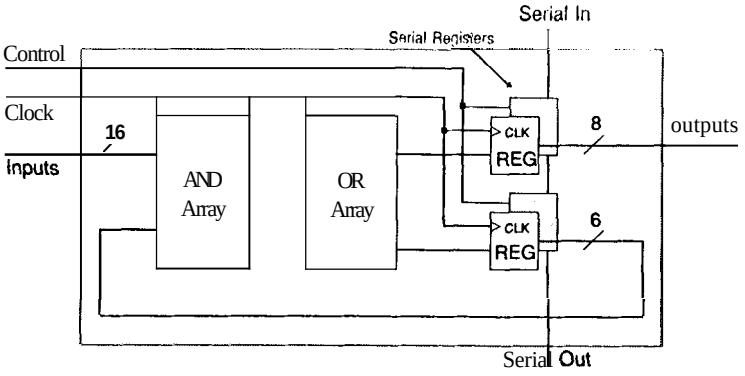


Figure 5.29. Internal Makeup of a Field Programmable Logic Sequencer with Registers for Scan Technique.

addressed concerns the speed of a state machine: how fast can it run, or how fast should it run? These two questions, in general. have different answers. One answer comes from the speed at which the controller can operate. The other answer comes from the speed at which the elements of the data path can operate.

All of the examples included in this chapter use edge-triggered registers, in which the outputs change to coincide with the values at the inputs when the active edge of the clock occurs. Thus, the timing requirements of the system must satisfy the constraints of edge triggered devices. Another design approach is to use devices that operate on a latching principle, in which the level of the outputs (of the latch) follow the level of the inputs so long as the clock (or enable) of the latch is asserted. A description of the differences in designing with edge-triggered devices and latched devices is found in Section 7.2.

A state machine controller of the type shown in Figure 5.27 consists of a register and some logic for generation of control signals and next state determination. The minimum cycle time for the system clock (SYSTEM_CLOCK-H) must include times sufficient for each of these functions. This time can be broken down into three basic components, as shown in Figure 5.30. When the active

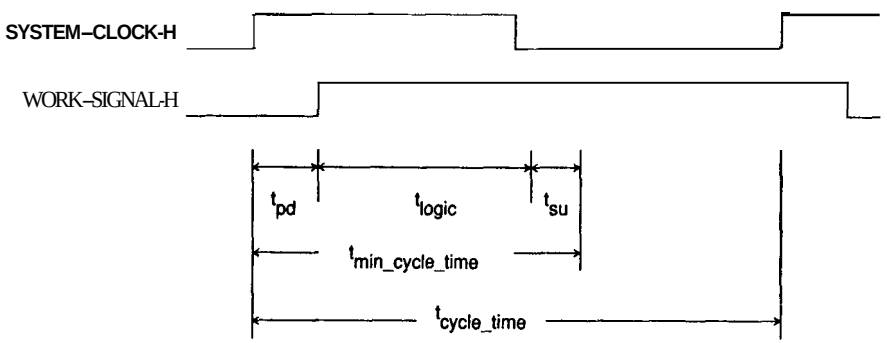


Figure 5.30. Component Times for Controller Cycle Time.

edge of the clock occurs, there is a propagation delay time (t_{pd}) after which signals become asserted, both in the present state register and in the signals that cause work in the data path. Once the present state is stable (and also the register holding the synchronized inputs), there is a time required for the choice of a next state to become stable. This is labeled in the figure as t_{logic}, but the decision could be made by random logic, PLAs, or memory. Whatever mechanism is utilized to determine the next state and the correct levels for the control lines in that state, t_{logic} must be sufficient to allow the these signals to become stable. When these values are stable, another time must be accounted for, which is the setup time of the register being used as the present state register (and the registers for the work signals). This is shown as t_{su} in the figure. The setup time is the amount of time prior to the active edge of the clock that a signal must be present at the input of a device to guarantee that the output stays at the required level after the clock occurs. Any time after the setup time requirement has been satisfied, the next active edge of the clock can occur. The sum of these three times provides a minimum cycle time that must be met by the system. For some high speed TTL parts, t_{pd} = 8 nsec, t_{logic} = 19 nsec, and t_{su} = 3 nsec, and the minimum cycle time would be 30 nsec, which gives a system clock frequency of 33.3 MHz.

As can be seen from Figure 5.30, the actual cycle time is often much longer than the minimum cycle time. The reason for the longer cycle time is not that the controller is incapable of running faster, but rather that the functions occurring on data path require a longer time to complete. Consider the timing relationships shown in Figure 5.31. which shows some of the signals required to add a value from the MBR to the accumulator. After the active edge of the clock, time is required for the work signals to become asserted, as shown in both Figure 5.30 and 5.31. Once the work signal becomes asserted, another propagation delay time is required; in this instance, it is for the driver to assert the value contained in the memory buffer register onto the bus. Once the value on the bus is stable, another time is required, which is the addition time of the ALU. It is assumed that the function lines and the carry input line of the ALU for the addition function became stable at the same time that the MBR_BUS-L signal became asserted; hence, these

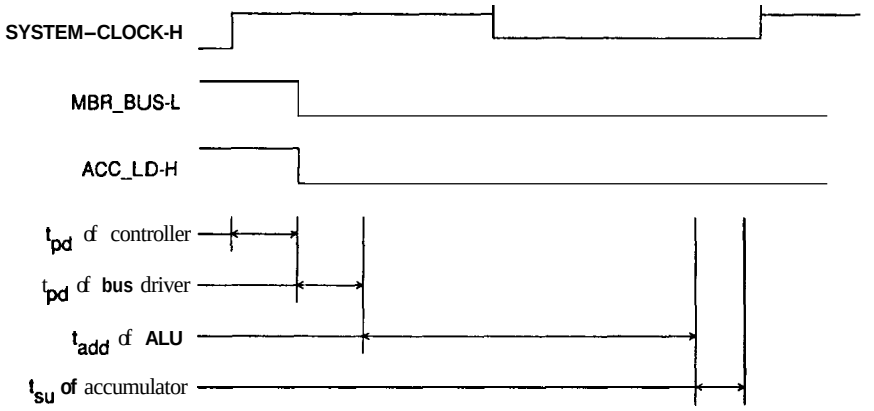


Figure 5.31. Component Times for Data Path Cycle Time.

lines are not shown in the figure. If for some reason these lines were not stable until after the data at the input to the ALU became stable, then the t_{add} of the ALU must be adjusted to reflect the delay time from the last stable signal. Once the correct results are stable on the outputs of the ALU, a t_{su} is required before the clock line of the accumulator can be asserted. The sum of the times required for the longest sequence of events in a single cycle establishes the minimum allowable cycle time for the data path, which is usually longer than the minimum cycle time for the controller. So the controller time is adjusted to match the cycle time requirements for the data path in the operation of the system.

Two basic functions are provided by the systems shown in Figure 5.26 and Figure 5.27. These are the choice of a next state, and the assertion levels of the control signals for each state. As we have seen, the next state logic and the next state control logic of Figure 5.27 can be implemented with memory devices. We now formulate a different view of the function provided by Figure 5.27, and present this view in Figure 5.32. The same functionality is shown: part of the system is used to control the function of the device by sequencing through the proper states, and the other part of the system controls the flow of data in the system by asserting the appropriate signals on the data path devices. The only addition to the process included in Figure 5.32 is the address selection portion of the system. The function provided by this section is to determine the address in the logic memory that contains the correct next state and control line assertion information. This address is a function of the present state and the external inputs. When the memory address is provided, the memory responds with the location of that address, and this information (the new state and new levels for the control signal assertions) is available for the control registers. The number of states, the complexity of the state diagram, and the number of inputs to the system determine the amount of logic memory needed for the selection of a next state. In addition to the complexity of function and number of inputs, the number of control signals generated by the unit determines the amount of logic memory needed for control signal generation.

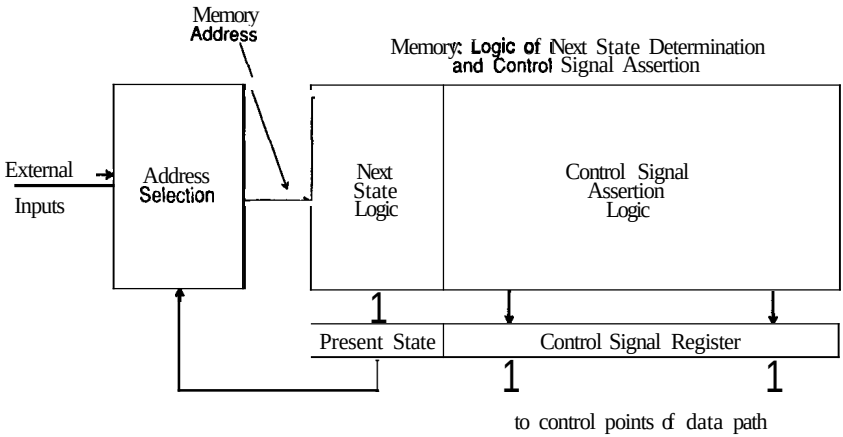


Figure 5.32. State Machine Implementation with Memory Implementing Logic Functions.

The diagrams in Figure 5.27 and Figure 5.32 represent the same function. and the difference is basically cosmetic. Figure 5.32 shows the elements of Figure 5.27 rotated by 90'. However, the diagram shown in Figure 5.32 matches most of the diagrams shown for another control technique which is called microcode. We expand the detail of Figure 5.32 slightly to obtain the system shown in Figure 5.33.

The present state register function and the registered control lines are combined in a register called the Microinstruction Register. All of the logic for control signal generation and next state selection is combined into one functional unit called Microcode Memory. Instead of a location in the microcode memory being strictly a function of the current state and all inputs, a functional unit labeled Microcode Address Generation performs the function shown as Address Selection in Figure 5.32. This unit selects the appropriate address for the next set of control lines; this forms the next microinstruction to execute. The complexity of the microcode address generation unit reflects the designer's tradeoff choices for speed versus complexity. We will describe different approaches in our next design example. Just like the present state register identified the state of a state machine controller, the registers and memory elements included in the microcode address generation module identify the state of a microcode machine. The address issued by the microcode address generation module identifies the next micminstruction to execute; the address is then analogous to the information contained in a present state register. However, the analogy is not exact, since the functions of the microcode address generation module can include things like subroutine linkage capabilities and loop control. However, the analogy does demonstrate the close conceptual relationship between microcode systems and systems designed with a state machine approach. The micminstruction register provides

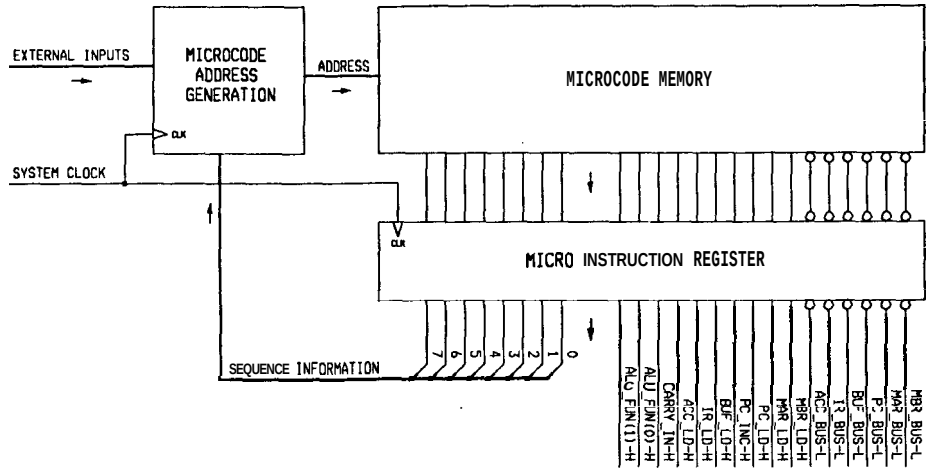


Figure 5.33. Implementation of Microcode Control.

the sequential control functions of the present state register, and it also specifics the assertion of the control lines of the data path.

This model is in agreement with the intuitive concept that work is accomplished by activating control signals in a sequential fashion. The sequential action of the control signals is specified by the sequence of microinstructions. and the address selection of the next microinstruction is derived during the execution of the current microinstruction. Thus, if the microinstructions were strictly sequential, the address could be provided by a simple counter. which would increment from one address to the next. The counter would be reset when the process needed to start again. The next address selection process can become arbitrarily complex; the address of the next microinstruction to execute can be one of many determined by a complex algorithm. In any case. a number of bits are used to control the selection of the next address; the remainder of the bits in the microinstruction register are used to control the data flow through the data path section of the device.

Let us begin our discussion of the contents of the microinstruction register by including whatever bits are required to specify selection of the address of the next microinstruction. In Figure 5.33 these bits are merely labeled "Sequence Information." The next bits to include in the microinstruction register are the control lines identified on the data path block diagram; we include one bit for each control line needed. The result is an extremely wide microinstruction register; the number of bits is the same as the number of control lines required to select the next address and control the data path. This style of microcode has received the name of "horizontal microcode," because the microcode grows wider as more functions are added.

The horizontal microcode technique results in the fastest microcode controlle n for two separate reasons. The first is that, since all of the bits are independent, multiple operations can be specified in the same microcode word. For example, assume that the value in the program counter is to be loaded into the memory address register and into the accumulator. In a horizontal scheme the clocks of both registers could be activated simultaneously. resulting in the transfer of information to two destinations in one cycle. This concurrent operation is not limited to information transfers over a bus, but can be observed in any independent operations. The second reason for enhanced speed is that no decoding is required for the control signals. This reduces to a minimum the cycle time required for operation of the system.

In contrast to the horizontal microcode method is a technique called "vertical microcode." This method emphasises not speed, but rather conserving system resources -- power and microcode bits. The method calls for combining the bits required for basically independent functions. For example, in Figure 5.33 there are six lines which, when activated, assert the data lines on the bus. These functions are not totally independent: we do not want more than one of these asserted at any time. Therefore, we can specify a single line to be asserted by encoding this information in fewer bits. In this case, we can specify one of the six lines with 3 bits; for example, a decoder such as a '138 could be used.

The encoding of information in this manner has two effects, both of which tend to slow down the operation of the system. The first is that the decoding of the bits is not free; more time is required in each cycle to allow for the decode function. This increases the time required for each cycle. The second cause of slowdown is that the system has a reduced capability of performing operations in parallel. Consider, for example, encoding the choice of N bus destination lines in

$\log_2 N$ bits. Combining bits in this manner precludes sending information to two destinations simultaneously; such an operation would require two cycles with the encoded scheme. Thus, the time required for accomplishment of work is increased because the individual cycle time is increased, and because more microinstructions are required. This increase in the number of microinstructions causes the required microcode memory to increase "vertically," which leads to the name of this technique.

The horizontal and vertical microcode methods both control the action of a system by sequencing through a set of microinstructions. However, each approach uses the system resources in a different way. The horizontal approach chooses to consume resources (power, number of bits in microcode word, etc.) to make the system run faster, both from concurrency of many simultaneous operations and from the minimal cycle times available. The vertical approach chooses to conserve the resources, limit concurrent operations, and accept a slower overall system speed. However, both mechanisms share many common characteristics, as demonstrated by the microcoded system in the following section.

5.7. A Microcode Controller

To demonstrate both the vertical and horizontal concepts of microcoded control, we will design two different microcode controllers for a computer system. The computer we will use for this example is patterned after the Data General Nova, which has been used for many years. This is not as exotic a machine as many newer machines. In fact, in many real aspects this system has been superseded by the 16- and 32-bit microprocessors available today. This system has been chosen to illustrate the ideas presented because it is simple enough to present in the confined space of a this section, and at the same it is complex enough to provide an fairly comprehensive example. A block diagram of the system is shown in Figure 5.34. Note that the diagram identifies the registers known to a

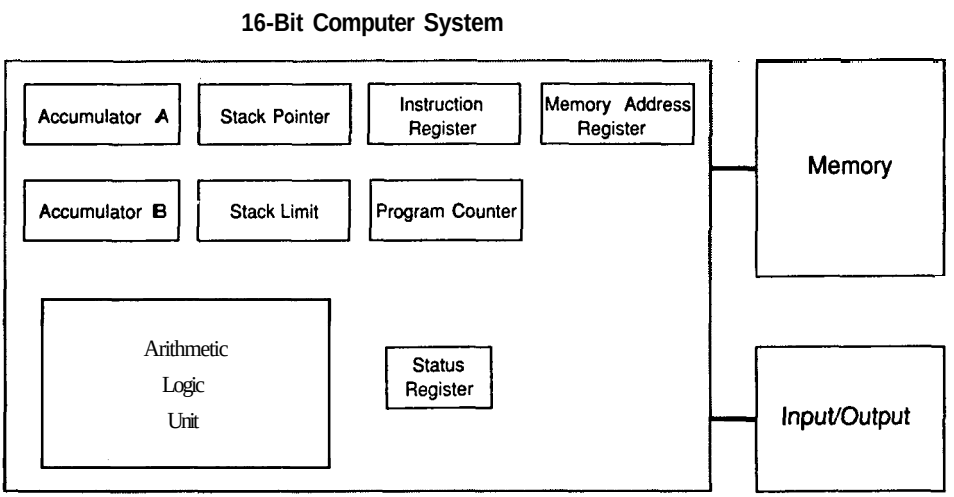


Figure 5.34. Block Diagram of 16-Bit Computer System.

programmer, without attempting to identify the physical links that connect the registers. The possible data transfers and arithmetic operations are defined by the instruction set. We will not describe the entire instruction set of such a machine; rather, we will select a few instructions and examine the rudimentary operations required to accomplish those instructions. If our instruction set matches the Nova exactly, we would like our module to execute the instructions in such a fashion that an observer would not be able to differentiate between our machine and a Nova: the "macro" machine behavior would be equivalent. We would then accomplish the work of the "macro" machine with our "micro" machine.

The machine depicted in Figure 5.34 has two 16-bit accumulators. In the Nova architecture these accumulators are also the first two locations of memory. Thus, actual registers are not required for this information; it will reside in the first two locations of main store. The memory address register, program counter, stack pointer, and stack limit register provide 24 bits of address information. The instruction register holds 16 bits, not all of which are needed by all instructions. The status register is composed of 4 bits that are both controlled and utilized by many of the instructions of the computer. The Nova instruction set utilizes I/O instructions instead of having strictly memory mapped I/O. Thus, to allow for these instructions we will need some interface lines as well. These will be described in more detail as we discuss the implementation.

The concepts of microcoded control can be applied at different levels. The address control, the microcode memory, and the microinstruction register can be composed of individual registers and memories, or the entire system can be part of a single integrated circuit. Many microprocessors utilize a microcoded control section internal to the chip. However, one family of components, called bit sliced processors, has been specifically designed to utilize microcoded control. The members of this family are so constructed that they can be put together in systems to satisfy a variety of constraints. In order to implement a microcoded system to perform the action of the system of Figure 5.34, we will use two of the most common microcoded devices. These are the 2901 Four Bit Microprocessor Slice and the 2910 Microprogram Controller. These units are available from several manufacturers, as well as newer units with extended capabilities.

A simplified block diagram of the 2901 is shown in Figure 5.35. This diagram shows the main data paths, but the control lines are merely suggested, and some of the data paths are not shown. What the diagram does indicate is that internal to the 2901 are a 4-bit ALU, a register bank holding 16 registers, a Q register, and multiplexers to control the flow of data. The registers (implemented in RAM) have two sets of addresses; the value of the register identified by the A address is loaded into the A latch, and the value of the register identified by the B address is loaded into the B latch. These loads occur at the beginning of a cycle, so that the values are available to the operand select function. The operand selec. portion of the device selects one of eight available combinations of the Data In, A, B, Q, and zero values. The two values chosen are fed into an ALU capable of AND, OR, EXCLUSIVE-OR, ADD, and SUBTRACT, as well as some variations of these operations. The output of the ALU can then be used as an output of the chip, as well as providing information to the registers. Writing can occur to the Q register, or to a register specified by the B address lines. In addition, there are data lines that allow cascading the modules to form units of higher numbers of bits, as well as lines that can provide status information. A data sheet should be referenced for a complete specification, but for the purposes of our example we need to know the data and control lines to be concerned with for the data path and

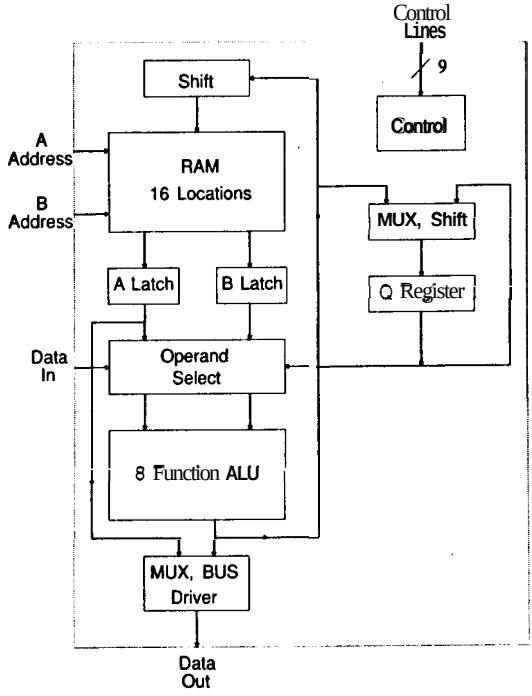


Figure 5.35. Simplified Block Diagram of a 2901 Bit Slice Microprocessor.

microcode sections. The data path is basically taken care of by the data in and data out lines; in addition some status lines need to be utilized. The control lines include the A and B addresses, which are each 4 bits, and the nine control lines. These we will need to include in the microinstruction register.

The address selection portion of our microcoded machine will be handled by a **2910.** a simplified diagram of which is shown in Figure 5.36. Like the **2901.** this diagram does not show all of the features of the **2910,** but points out the major capabilities. The unit is capable of handling a 12-bit address, which will address up to 4,096 words of microcode memory. This is sufficient for most applications; however, similar units are available that will control more address bits. The multiplexer in the unit selects one of four sources:

- The ***data path*** allows for an external source to specify what the next microinstruction address will be. This is useful for jumps, subroutine calls, and similar activities.
- The ***register path*** allows the functional unit to specify at some previous time an address used to specify a microinstruction.
- The ***stack path*** is used for returning from subroutines, and for providing an address during special function operations.
- The ***microprogram counter-register path*** is used to proceed to the next instruction.

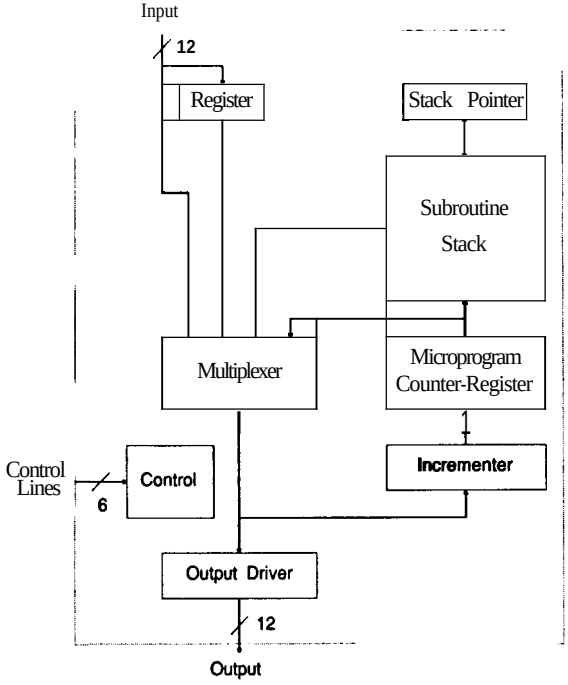


Figure 536. Simplified Block Diagram of a 2910 Sequencer.

The incrementer is not built into the microprogram counter-register, a situation that seems unreasonable. However, consider the desired action when a subroutine address is provided via the data path. In this situation, the microprogram counter-register contains the address of the next instruction in sequence (after the currently executing microinstruction), and this address is to be placed on the stack as a return address. The output lines contain the address of the subroutine (supplied as an input on the data lines), and the microinstruction at that address will be fetched for execution. The address of the microinstruction that should be obtained next is the second instruction in the subroutine; therefore, the value presented to the microprogram counter-register is one more than the subroutine address, not one more than the current address. Hence, the incrementer is connected to the output lines of the multiplexer, not to the microprogram counter-register containing the current address.

The control lines of the sequencer select one of sixteen instructions, many of which have a conditional nature associated with them. The conditional mechanism allows the address at the output to be one of the four values available at the input to the multiplexer, the selection of which depends upon the conditional inputs. Not shown on the block diagram are three output lines, which can be used to control the source of information presented to the input lines: **PL-L**, **MAP-L**, and **VEC-L**. For most of the instructions **PL-L** is asserted; this can be used to enable information from the microinstruction register to provide the necessary data. One of the instructions causes the **MAP-L** line to be asserted; another

instruction causes the *VEC-L* to be asserted. These signals are provided so that when some external information, such as the op code of an instruction, is to cause the system to jump to an externally supplied address, that address can be made available to the sequencer. In this case, the normal source of address information is disabled *(PL-L* is deasserted), and the alternate source of address information is enabled *(MAP-L* or *VEC-L* is asserted). The mapping between opcodes and the address of their respective microcode implementations can be easily stored in a *PROM.* The output enable of the *PROM* can be directly connected to the *MAP-L* signal, and when the system requires the mapping function to be performed, the appropriate address is supplied to the inputs of the sequencer. The *VEC-L* line is utilized in the same fashion: an external address, such as a vector supplied by the user, is enabled onto the input lines at the appropriate time.

Like the 2901, the **2910** contains both control lines and data lines. However, in this implementation the data lines of the **2910** are all concerned with microcode addresses in the control section, and do not have a direct bearing on the data path section of the system. The data path for our example is shown in Figure 5.37. This diagram indicates that we are going to simulate the action of the 16-bit machine with an 8-bit system; 16-bit transactions will then require two transfers. The 8-bit system is composed of two **2901** processing elements that have been combined to provide 8-bit arithmetic and logic capabilities. The address and instruction information needed by the processor section are provided by the microinstruction register (MIR). In addition, the MIR supplies 8 bits of data to provide a constant load capability. Often a system designer will need the capability to place a known value in a register, or provide a constant for comparison or masking purposes. The other modules of the processor shown in Figure 5.37 are for the data and address paths. The data path is composed of bidirectional registers; this allows our system to load information in 8-bit quantities, and these quantities are then available on a 16-bit bus. The reverse path is also available, allowing our module to accept 16-bit values 8 bits at a time. The *DATA* HIGH and *DATA LOW* blocks of Figure 5.37 can be constructed from individual registers and hi-state drivers as shown in Figure 5.38(a). The address path is broken into three 8-bit quantities, which together form a 24-bit address. These registers can be read individually by the 8-bit system, or they can provide an address under the control of an arbitration module, which is not considered here. The address modules of Figure 5.37 can be created with the register and driver configurations shown in Figure 5.38(b).

Not shown on the diagram are the control signals used to interact with the memory and the *UO.* This system is patterned after the Nova, but many of the features are different. We will assume that there is a separate memory address space and I/O address space; this will require a method to identify the address currently on the address bus. That is, the control lines must establish a different protocol (either different physical lines or a different accessing mechanism) for the I/O devices than that used for the memory locations. The Nova I/O structure calls for three functions/registers at each interface address, and these are labeled *A,* B, and *C.* In this implementation, there are write and read control lines to each of these elements. In addition, there are some other control signals for testing conditions and causing action at the interface. Table 5.3 identifies the various control signals that we will include in our design, in addition to those found in Figure 5.37.

With the detailed data path block diagram available, the signals needed to control the flow of information in the system have been identified. In addition,

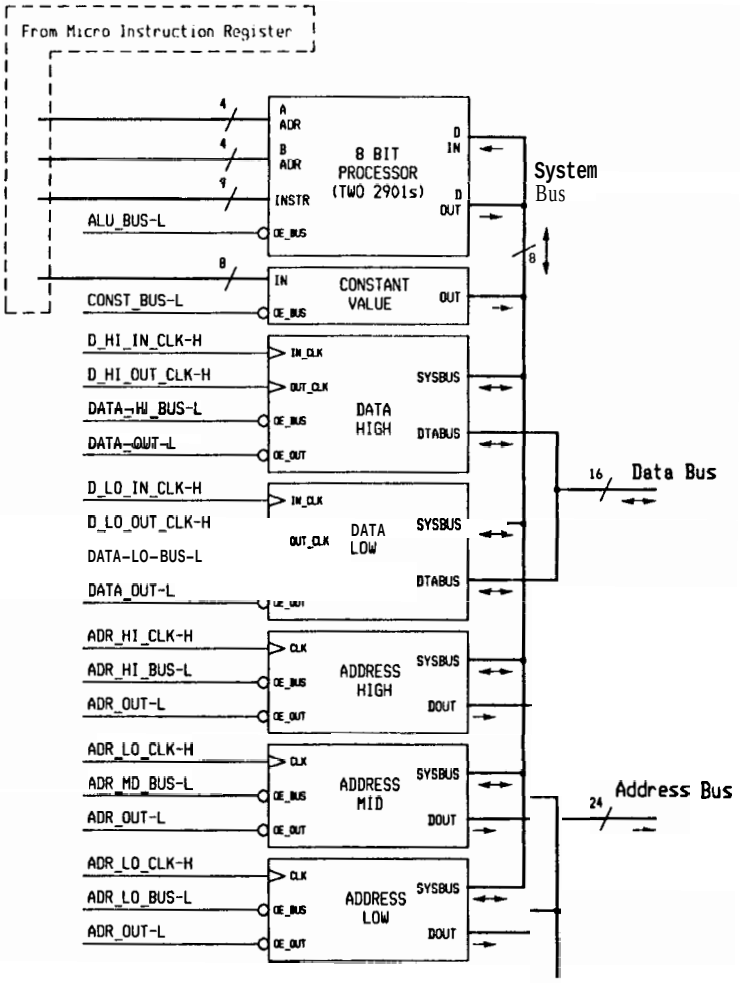


Figure 5.37. Data Path Block Diagram of Implementation of the 16-Bit Computer System.

the other control signals needed in the system have been specified. We are ready to set up the microcode control section of the computer. Before doing that, however, we will consider one more portion of the circuitry. This system is using an 8-bit processor to simulate the action of a 16 bit-processor, and hence must be able to do 16-bit arithmetic. In fact, to increment the program counter, a 24-bit addition must be possible. To accomplish that we have included the circuitry shown in Figure 5.39 to control the carry into the processor. As seen by the logic, under the control of the **MIR** the carry into the ALU **(ALU_C_IN-H)** can be forced to zero, forced to one, set to the carry out of the previous cycle **(uCRY-H),** or set to MCRY-H. MCRY-H is the **carry** bit from the status register of the 16-bit machine, which is not shown. This control of the **carry** input allows the designer

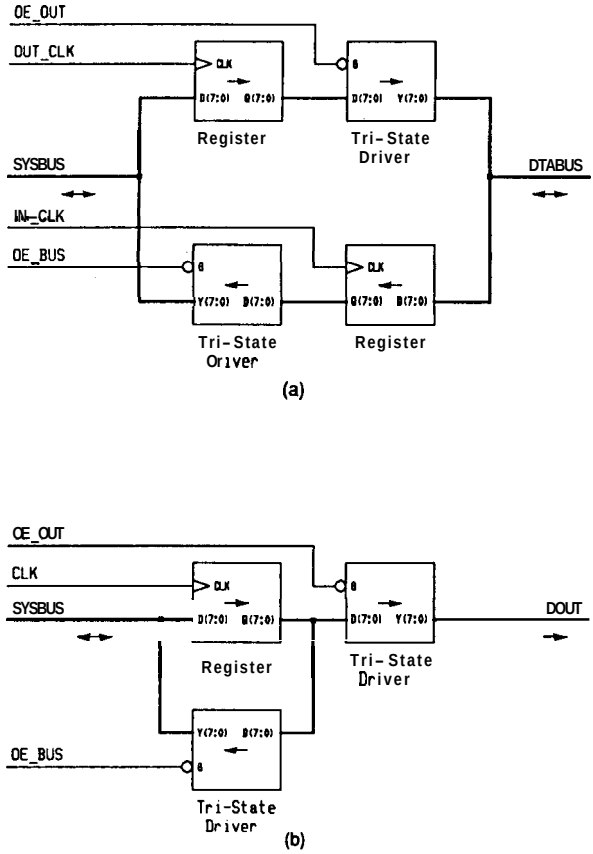


Figure 5.38. Components of the Data and Address Portions of Microcoded System; (a) Bidirectional Data Register for Microcode Example, (b) Address Driver for Microcode Example.

total flexibility; multiprecision adds can be achieved by doing 8 bits at a time and feeding the carry to the next cycle through uCRY-H. Adds of 32-bit words can be done at the assembly language level, and the microcode would then select MCRY-H as the carry in the appropriate cycle.

With this set of control bits identified, we will proceed with the design process. It should be noted that the design is not complete, and more control lines would be required for the entire system. In the horizontal microcode approach there will be 1 bit in the MIR for each control line. This results in a very wide word, but the clock cycle time is as small as possible, and the available parallelism is at a maximum. Figure 5.40 shows a diagram of the resulting system. A more detailed schematic diagram is found in Appendix B.

The diagram shows a system with the same general organization as seen in Figure 5.33. The 2910 provides the address information, and the microcode memory modules supply the microinstruction to the MIR. The bits comprising the

Table 5.3. Additional Control Lines for 16-Bit Computer.

Signal Name	*Definition*
MEM-H	Asserted when the transfer is for memory.
READ-H	Asserted when the transfer is a read (to CPU).
ADR_VALID-H	Asserted when address valid; deasserted at end of cycle.
DATA–VALID-H	Asserted (by CPU for write, by device for read) when data valid.
DATOA-H	A control line; asserted on output.
DATOB-H	B control line; asserted on output.
DATOC-H	C control line; asserted on output.
DATIA-H	A control line; asserted on input.
DATIB-H	B control line; asserted on input.
DATIC-H	C control line; asserted on input.
STRT-H	Start control line; asserted when needed by I/O instruction.
CLR-H	Clear control line; asserted when needed by I/O instruction.
IOPLS-H	YO pulse; asserted when needed by YO instruction.
MSKO-H	Mask out; asserted during MSKO instruction.
INTA-H	Interrupt acknowledge; asserted during INTA instruction.
DCHA-H	Data channel acknowledge; asserted at beginning of data channel cycle.
DCHI-H	Data channel input; asserted for channel input.
DCHO-H	Data channel output; asserted for channel output.
IORST-H	I/O reset; asserted during IORST instruction. console reset.

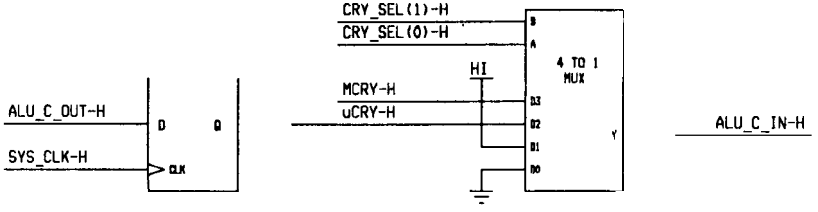


Figure 5.39. Carry Control Circuitry for 16-Bit Computer System.

MIR can be loosely grouped into three categories: bits controlling the microcode address system, bits controlling data flow on the data path, and bits controlling interaction with other machines. We will briefly discuss some points concerning each of these sections.

The address control section has four lines (SEQ_INSTR) to control the function of the 2910. When these lines identify a conditional type of instruction, the action of the module is further specified by the condition code and condition code

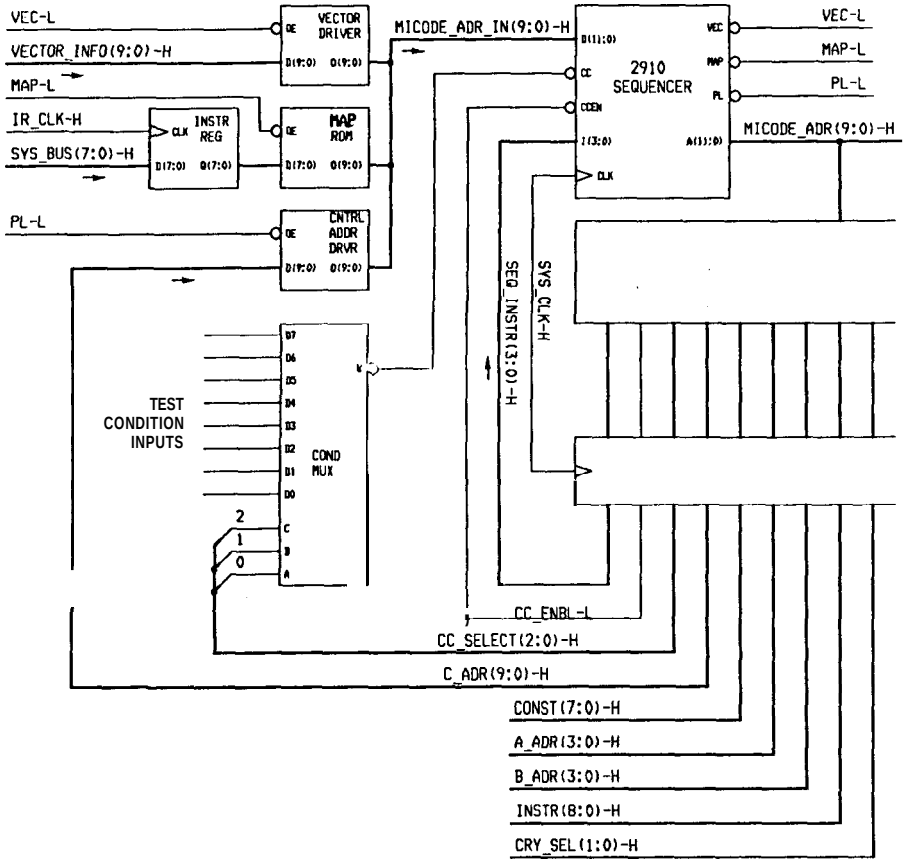


Figure 5.40. Logic Diagram for Horizontal Microcode Control.

enable (CC_ENBL) lines. The condition code is actually selected from a number of available possibilities by CC_SELECT. We have shown eight possible inputs; additional inputs could be considered by allowing a larger multiplexer and another select line. These lines control the function of the 2910, but occasionally additional information is required by the sequencer. For example, if the instruction to execute is a jump or jump subroutine, then the target address must be supplied, which is the function of the MICODE_ADR_IN lines.

There are three sources of information into the sequencer, each of which has responsibility for a different kind of information.

- The control address driver (CNTRL ADR DRVR) is the selected source when PL is asserted, and provides information from the MIR. This allows address information to be directed to the sequencer directly from the microcode.
- The second source, the map ROM (MAP ROM), is selected when the MAP signal is asserted. This allows a designer to map bit patterns loaded into the

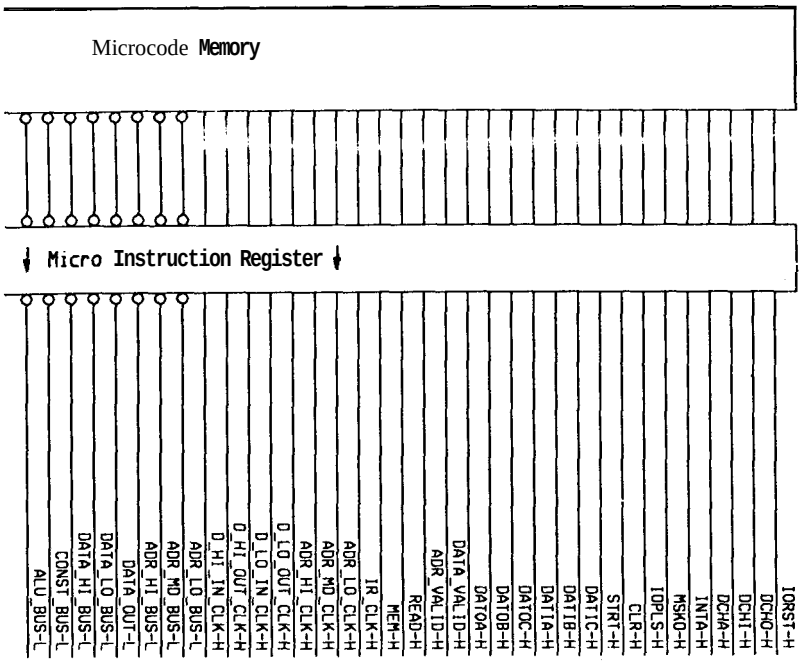


Figure 5.40. *(cont.)* Logic Diagram for Horizontal Microcode Control.

instruction register **(INSTR REG)** to specific addresses in microcode memory at which the necessary instruction sequence is found.

- The final source for address information is the vector driver, which accepts an address from an external source and supplies that address when the VEC signal is asserted. This allows an address to be supplied directly from a user-defined source.

As can be seen from the diagram, the sequencer requires 18 lines to control these functions, and these lines form the first part of the **MIR.**

The control and data lines of the data path elements are the next to be included in the MIR. The **CONST** lines allow a known value to be injected into the system. This is very useful for providing values to be used as constants, masks, and match values. The **A** and B address lines **(A-ADR, B_ADR)** are required by the 2901 **ALU** to specify the addresses needed for register identification. The instruction lines **(INSTR)** specify the action to be taken by the

Table 5.4. Bit Fields for Microcode of Table 5.5.

Label	Number of bits	Radix of Representation	Function
ADR		16	Address; number of bits depends on implementation (10 bits in this example).
SEQ INS	4	16	Sequencer instruction; bits to I lines on 2910.
CC EN	1	2	Condition code enable.
CC SEL	3	8	Condition code select; address of condition to test.
SEQ ADDR	10	16	Sequencer address; lines to provide address for jumps.
DATA CONS	8	16	Data constant; for constants to bus.
A ADR	4	16	A address lines of 2901.
B ADR	4	16	B address lines of 2901.
ALU INS	9	8	Instruction lines of 2901.
CRY SEL	2	2	Carry select lines.
BUS SRC	8	2	Bus source lines.
BUS DEST	8	2	Bus destination lines.
MEM I/O BITS	19	2	Bits for memory and I/O interaction.

ALU. If the instruction to be executed requires a carry input. then the CRY–SEL lines specify the appropriate carry information. These 27 lines control the processing accomplished by the processing section of the system, in addition to providing constants as needed.

The remainder of the bits are those required to control the individual elements of the system. By allowing total control of the lines (since each control line is independent of the others) the maximum parallelism is possible. For example, all of the extemal registers can be cleared by creating a zero value on the bus (all bits zero in constant field, assert CONST_BUS-L) and assert all of the appropriate clock lines. This would allow loading of six registers simultaneously. The bits in the microcode word also contain the control bits identified in Table 5.3 for interaction with external units. This last section of the MIR, which contains the control lines for the data path and the interaction with extemal devices, contains 35 bits.

One of the best ways to become familiar with the system and its capabilities and weaknesses is to prepare microcode for it. To this end we will look at a few lines of code that do two simple functions: simulate the action of an ADD A, B instruction and a JUMP instruction. These instructions are broken into separate fetch and execute portions, and the following assumptions are made: the 24-bit stack limit register comprises registers 5, 6, and 7 in the 2901s; the 24-bit stack pointer comprises registers 8, 9, and 10 in the 2901s; the 24-bit program counter is contained in registers 11, 12, and 13; 16 bits of instruction are located in registers 14 and 15 in the 2901s; the A and B registers are coincident with memory locations 000_{16} and 002_{16} (byte addresses), and are stored in memory.

The microcode is contained in Table 5.5, which can be confusing if not approached in a regular fashion. Each of the headings in Table 5.5 identifies a group of bits, and their definitions representations are given in Table 5.4. Each microcode word identifies a single operation, but since there are a large number of bits, a correspondingly large number of things can happen during each microcode cycle. Each group of bits identifies an action to be performed by the system. Required macro operations are accomplished stepwise by a succession of micro operations. Table 5.5 contains three sections of micro operations.

The first section (addresses 0A0 to 0A8) is the code for the fetch portion. Note that the function of this code is to do PC → MAR, which takes three 8-bit transfers, at the same time that it does PC + 2 → PC. Note also that the bus destination lines are normally high in this implementation, so that (see the instruction at 0A1) when a value is available on the bus (a bus source line is asserted), that value can be loaded into the appropriate register at the end of the cycle. This occurs because the 0 becomes a 1 at the next clock pulse, creating a rising edge that causes the required load of information. Another thing to note is that the memory interaction is started by the instruction at location 0A4, and the instruction at location 0A5 waits for the memory to respond before continuing. Instructions located at 0A6 and 0A7 move 16 bits of instruction into the internal IR, as well as the 8 most significant bits to the IR that addresses the MAP ROM. The instruction at location 0A8 causes the sequencer to jump to the address specified by the *MAP* ROM, which will be the beginning of the code for the appropriate macro instruction. For this example, that address will either be 0F2 or 121; in general, the address can map to any appropriate location.

The second section of code (addresses 0F2 to 100) performs the ADD A, B instruction. The address zero is forced into the MAR, and that value (the A value) is copied to temporary locations in the 2901s (registers 2 and 3). Then the address two is placed in the MAR, and that value is added to the temporary already in the 2901s. Then the result is written back, and control moves to the fetch portion to continue execution. Again note the memory interaction: the action is initialized by the microcode, and the microcode continues when the memory responds.

- The final section of code (addresses 121 to 129) is for the JUMP instruction. The assumption here is that the instruction is actually 32 bits long — 8 bits of op code and 24 bits for the target address. The fetch section has placed 8 of the 24 address bits in the internal IR, in register 15. So, the first part of this code duplicates the fetch action to obtain the next 2 bytes. These bytes are then transferred from the data registers to the address registers, along with the value contained in register 15. The microcode then moves back to the fetch portion to proceed with the program. Note here that the maximum speed is not attained, since the work done by the instruction at location 127 could be done with the work done by the instruction at location 125, and the time required for the execution of the instruction would be reduced by one cycle.

Other observations can be made concerning the microcode in Table 5.5. Some of the fields are not used much of the time, and some of the fields have only a small number of legal patterns. This is one of the observations that gives rise to the use of vertical microcode. We wish to reduce the number of bits required in a microcode word, but we still wish to be able to do all of the necessary functions. The resulting system uses single fields for multiple functions, and combines patterns into decoded information. For our example, we will combine the functions of the 2910 address, the data constant. the A and B addresses for the 2901s, and the ALU carry select into a single field. In addition, we will combine the bits required for bus source into one field, bus destination into another, and further encode the bits required for the memory-I/O interaction. The resulting system is shown in Figure 5.41. A schematic showing the detailed connections of the components is included in Appendix B.

This system is very much like the system shown in Figure 5.40. The control signals are identical; however, there are limitations on how many control

Table 55. Horizontal Microcode for Fetch. **ADD A.B. and JUMP.**

ADR	SEQ INS 4 bits	CC EN 1 bit	CC SEL 3 bits	SEQ ADR 10 bits	DATA CONS 8 bits	A ADR 4 bits	B ADR 4 bits	ALU INS 9 bits	CRY SEL 2 bits	BUS SRC 8 bits	BUS DEST 8 bits
ddd_{16}	d_{16}	d_2	d_8	ddd_{16}	dd_{16}	d_{16}	d_{16}	ddd_8	dd_2	$d \cdots d_2$	$d \cdots d_2$
0A0	E	x	x	xxx	02	x	x	037	xx	10111111	11111111
0A1	E	x	x	xxx	xx	B	B	200	00	01111111	11111101
0A2	E	x	x	xxx	xx	C	C	203	10	01111111	11111011
0A3	E	x	x	xxx	xx	D	D	203	10	01111111	11110111
0A4	E	x	x	xxx	xx	x	x	144	xx	11111111	11111111
0A5	3	0	1	0A5	xx	x	x	144	xx	11111111	01011111
0A6	E	x	x	xxx	xx	x	E	337	xx	11101111	11111111
0A7	E	x	x	xxx	xx	x	F	337	xx	11011111	11111110
0A8	2	x	x	xxx	xx	x	x	144	xx	11111111	11111111
0F2	E	x	x	xxx	00	x	x	144	xx	10111111	11110001
0F3	E	x	x	xxx	xx	x	x	144	xx	11111111	11111111
0F4	3	0	1	0F4	xx	x	x	144	xx	11111111	01011111
0F5	E	x	x	xxx	xx	x	2	337	xx	11101111	11111111
0F6	E	x	x	xxx	xx	x	3	337	xx	11011111	11111111
0F7	E	x	x	xxx	02	x	x	144	xx	10111111	11111101
0F8	E	x	x	xxx	xx	x	x	144	xx	11111111	11111111
0F9	3	0	1	0F9	xx	x	x	144	xx	11111111	01011111
0FA	E	x	x	xxx	xx	2	2	305	00	11101111	11111111
0FB	E	x	x	xxx	xx	3	3	305	10	11011111	11111111
0FC	E	x	x	xxx	xx	2	x	134	xx	01111111	11101111
0FD	E	x	x	xxx	xx	3	x	134	xx	01111111	10111111
0FE	E	x	x	xxx	xx	x	x	144	xx	11110111	11111111
0FF	3	0	1	0FF	xx	x	x	144	xx	11110111	11111111
100	3	1	x	0A0	xx	x	x	144	xx	11111111	11111111
121	E	x	x	xxx	02	x	x	037	xx	10111111	11111111
122	E	x	x	xxx	xx	B	B	200	00	01111111	11111101
123	E	x	x	xxx	xx	C	C	203	10	01111111	11111011
124	E	x	x	xxx	xx	D	D	203	10	01111111	11110111
125	E	x	x	xxx	xx	x	x	144	xx	11111111	11111111
126	3	0	1	126	xx	x	x	144	xx	11111111	01011111
127	E	x	x	xxx	xx	F	B	334	xx	11111111	11111111
128	E	x	x	xxx	xx	x	C	337	xx	11101111	11111111
129	3	1	x	0A0	xx	x	D	337	xx	11011111	11111111

signals can be asserted at any given time. For example, one of the functions of the horizontal microcode example was to load a zero value into three registers simultaneously; in this implementation that would require three separate instructions, since only one destination line can be asserted at any time. Also, the horizontal microcode method has independent address and data fields in the code; it would be possible to jump to one address and load a constant in the same cycle. With the vertical microcode implementation one field is used for both these functions; hence, one could not load an arbitrary constant and perform a microcode jump at the same time. This type of system is, in general, slower than the horizontal microcode system, since more instructions are required, and the cycle time is longer. However, the number of bits required in the microcode word is smaller, and the total number of bits (number of words × number of bits/word) is, in

Table 5.5. *(cont.)* Horizontal Microcode for Fetch, ADD A.B. and JUMP.

MEM I/O BITS 19 bits $d \cdots d_2$	ADR ddd_{16}	*Comment*
		PC → MAR; increment **PC (PC = R11, R12, R13)**
0000000000000000000	0A0	Move constant 02_{16} to Q reg of **2901.**
0000000000000000000	0A1	**R11 → MAR$_{7-0}$;** increment **R11.**
0000000000000000000	0A2	**R12 → MAR$_{15-8}$;** inc **R12** with previous carry.
0000000000000000000	0A3	**R13 → MAR$_{23-16}$;** inc **R13** with previous carry.
1110000000000000000	0A4	**MEM** bits initiate memory read action.
1110000000000000000	0A5	Stay here till memory ready; **0 → 1** on destination lines loads result.
0000000000000000000	0A6	Move **LSB** to **R14.**
0000000000000000000	0A7	Move **MSB** to **R15** and **IR.**
0000000000000000000	0A8	Jump to address provided by **MAP ROM.**
0000000000000000000	0F2	**0 → MAR**
1110000000000000000	0F3	**MEM** bits initiate memory read action.
1110000000000000000	0F4	Stay here till memory ready; **0 → 1** on destination lines loads result.
0000000000000000000	0F5	**LSB** of mem value to **R2.**
0000000000000000000	0F6	**MSB** of mem value to **R3.**
0000000000000000000	0F7	02_{16} → MAR$_{}$
1110000000000000000	0F8	**MEM** bits initiate memory read action.
1110000000000000000	0F9	Stay here till memory ready; **0 → 1** on destination lines load; result.
0000000000000000000	0FA	Add **LSB** of memory to **R2.**
0000000000000000000	0FB	Add **MSB** of memory to **R3** with previous carry.
0000000000000000000	0FC	**R2 → MEM$_{7-0}$**
0000000000000000000	0FD	**R3 → MEM$_{15-8}$**
1010000000000000000	0FE	**MEM** bits initiate memory write action.
1010000000000000000	0FF	Wait here till memory done.
0000000000000000000	100	Jump back to address $0A0_{16}$ for next fetch.
0000000000000000000	121	Move constant 02_{16} to Q register of **2901.**
0000000000000000000	122	**R11 → MAR$_{}$;** increment **R11.**
0000000000000000000	123	**R12 → MAR$_{}$;** inc **R12** with previous carry.
0000000000000000000	124	**R13 → MAR$_{}$;** inc **R13** with previous carry.
1110000000000000000	125	**MEM** bits initiate memory read action.
1110000000000000000	126	Stay here till memory ready; **0 → 1** on destination lines loads result.
0000000000000000000	127	Move first byte to PC from **R15.**
0000000000000000000	128	Move second byte to PC from **MBR.**
0000000000000000000	129	Move third byte to PC from **MBR;** jump to fetch.

general, smaller. For this example, the number of bits in the microcode word decreased from 80 bits (horizontal) to 45 bits (vertical), a decrease of over 40 percent in the number of lines required.

The method a designer uses to combine functions and lines into groups, and the amount of overlap used in a system, reflect the design choices made in the design process. If a designer is using 8-bit parts, he may attempt to end up with a system that uses a multiple of 8 bits. If a designer is constrained by power requirements, he may combine as many fields as possible into one. Any of a number of different requirements will influence the choices made in the process. This example brings out several techniques that can be used, which we point out here. The constant lines, the address lines, and the A and B addresses from the ALU have been combined into a single field. This constrains what can be done at

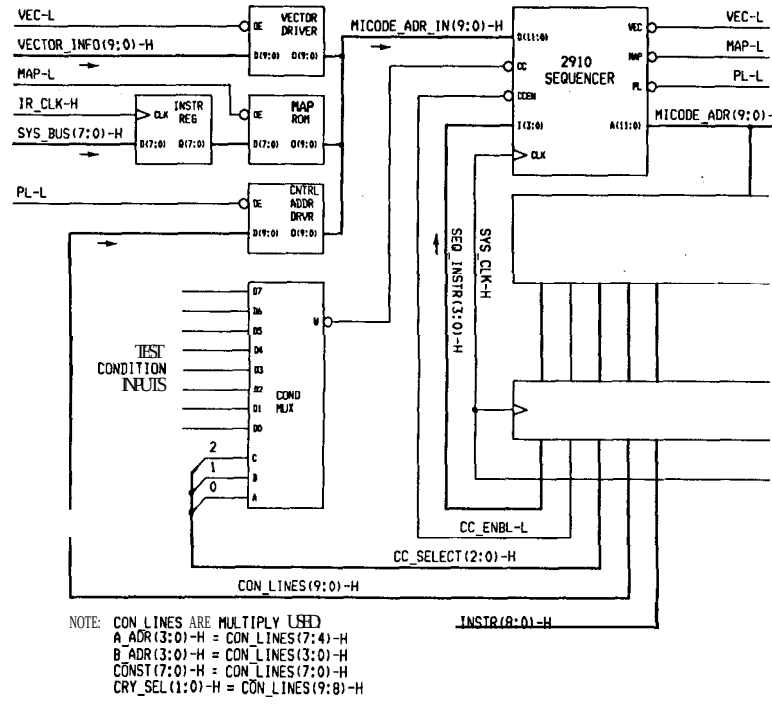


Figure 5.41. Logic Diagram for Vertical Microcode Control.

one time, since only one function can be performed by the CON_LINES at any one time. One example is the jump/load conflict. Another example is loading constants into the ALU: since the B address lines specify the destination of a write function, information that is to go to an arbitrary register in the ALU is first placed in the Q register, then moved to the appropriate register. Thus, what was accomplished in a single cycle in the previous system requires two cycles in this system.

In general. it is not a good idea to permit more than one driver to place a value on a bus at any one time. Thus, there is danger in the scheme of the horizontal microcode implementation, since it is possible to enable more than one source to the bus simultaneously. This possibility has been removed in the vertical microcode example by the use of a decoder; now only one line can be asserted at any one time. A decoder has also been used to identify the destination, which removes the possibility of sending information to two or more destinations simultaneously. Since there were eight destinations in the original system. a three to eight decoder has been used to do this decoding; however, since we want the ability to select none of the destinations. an additional line has been added to enable the entire destination function. A different technique for not asserting any line

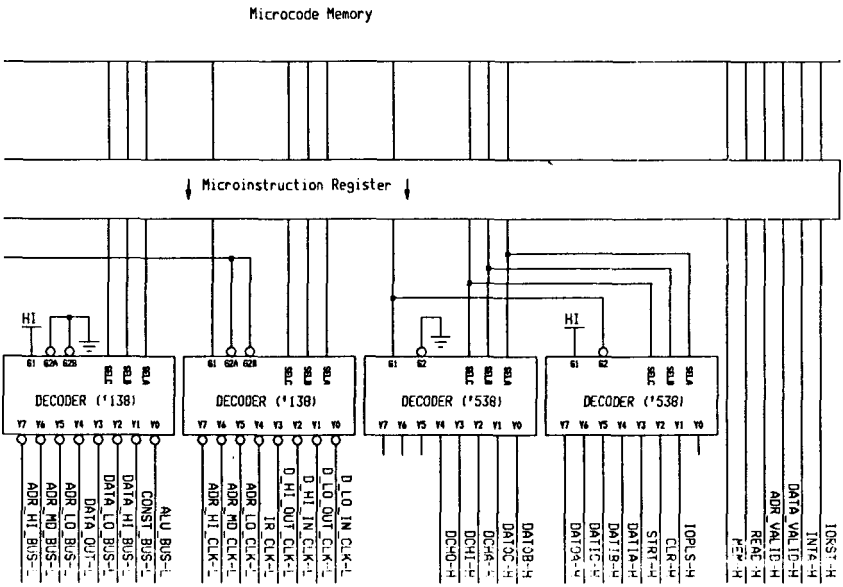


Figure 5.41. *(cont)* Logic Diagram for Vertical Microcode Control

has been used for the I/O bits: the zero line is left unused. Thus a value of zero on the I/O bit lines will result in no action. Finally. those lines that cannot be independently asserted, such as READ, **MEM,** and the like, are not combined, but left to be asserted as needed by the **system.**

Some care must be taken to guarantee correct system function with respect to the nonideal nature of the decoders. We decoder used for the bus source is always enabled. and the microcode system can control only the address lines. This will result in glitches on the decoder output lines as the internal logic changes to agree with changes of the address lines. The outputs of this decoder are connected to **tri-state** enable lines of devices that can assert information onto the data bus. These glitches will not cause problems with the data being enabled onto the bus, since the glitches always occur at the beginning of the cycle when the bus lines have not assumed the correct assertion level.

Table 5.6.

ADR	SEQ INST 4 bits	CC EN 1 bit	CC SEL 3 bits	CON LINES 10 bits	ALU INST 9 bits	BUS SRC 3 bits	BUS DEST 4 bits	I/O BITS 4 bits	MEM BITS 6 bits	ADR	Comment
ddd_{16}	d_{16}	d_2	d_1	ddd_{16}	ddd_8	d_8	d_{16}	d_{16}	$d \cdots d_2$	ddd_{16}	
OAO	E	x	x	002	037	1	0	0	000000	OAO	PC → MAR; PC is stored in R11, R12, R13; first, 02_{16} to Q reg.
OA1	E	x	x	OBB	200	0	D	0	000000	0A1	R11 → MAR; increment R11.
0A2	E	x	x	2CC	203	0	E	0	000000	0A2	R12 → $MAR_{7..8}$; increment R12 with last carry.
0A3	E	x	x	2DD	203	0	F	0	000000	0A3	R13 → $MAR_{15..}$ increment R13 with last carry.
0A4	E	x	x	xxx	144	0	0	0	111000	0A4	MEM lines initiate read action.
0A5	3	0	1	0A5	144	0	0	0	111000	0A5	Wait for memory to respond.
0A6	E	x	x	xxx	144	0	8	0	111000	0A6	Strobe MEM LSB to accept info.
0A7	E	x	x	xxE	337	3	A	0	111000	0A7	Strobe MEM MSB to accept info; transfer LSB to R14.
0A8	E	x	x	xxF	337	2	C	0	000000	0A8	Transfer MEM MSB to R15 and IR.
0A9	2	x	x	xxx	144	0	0	0	000000	0A9	Jump to address provided by MAP ROM.
0F2	E	x	x	000	144	1	D	0	000000	0F2	Set MAR_7 to zero.
0 n	E	x	x	000	144	1	E	0	000000	0 n	Set $MAR_{8..}$ to zero.
OF4	E	x	x	000	144	1	F	0	000000	OF4	Set $MR_{7..}$ lo zero.
OF5	E	x	x	xxx	144	0	0	0	111000	OF5	MEM lines initiate read action.
OF6	3	0	1	OF6	144	0	0	0	111000	OF6	Wait for memory to respond.
O R	E	x	x	xxx	144	0	8	0	111000	0F7	Strobe MEM LSB to accept info.
0F8	E	x	x	xx2	337	3	A	0	111000	0F8	Strobe MEM MSB to accept info; transfer LSB lo R2.
OF9	E	x	x	xx3	337	2	0	0	111000	OF9	Transfer MEM MSB to R3.
OFA	E	x	x	002	144	1	D	0	000000	OFA	Load MAR, with 02_{16}.
0FB	E	x	x	xxx	144	0	0	0	111000	0FB	MEM lines initiate read action.
OFC	3	0	1	OFC	144	0	0	0	111000	0FC	Wait for memory to respond.
0FD	E	x	x	xxx	144	0	8	0	111000	0FD	Strobe MEM LSB to accept info.
0FE	E	x	x	022	305	3	A	0	111000	0FE	Strobe MEM MSB to accept info; add LSB to R2.
OFF	E	x	x	233	305	2	0	0	111000	OFF	Add MEM MSB to R3 with last carry.
100	E	x	x	x2x	134	0	9	0	000000	100	Transfer contents of R2 to MEM LSB.
101	E	x	x	x3x	134	0	B	0	000000	101	Transfer conlents of R3 to MEM MSB.
102	E	x	x	xxx	144	0	0	0	101000	102	MEM lines initiate write action.
103	3	0	1	103	144	0	0	0	101000	103	Wait for memory to respond.
104	3	1	0	0A0	144	0	0	0	000000	104	Jump back to fetch microcode (address 0A0).
121	E	x	x	002	037	1	0	0	000000	121	PC → MAR; 02_{16} to Q reg.
122	E	x	x	OBB	200	0	D	0	000000	122	R11 → MAR_{7-0}; increment R11.
123	E	x	x	2CC	203	0	E	0	000000	123	R12 → MAR_{15-8}; increment R12 with last carry.
124	E	x	x	2DD	203	0	F	0	000000	124	R13 → $MAR_{...}$ increment R13 with last carry.
125	E	x	x	xxx	144	0	0	0	111000	1254	MEM lines initiate read action.
126	3	0	1	126	144	0	0	0	111000	126	Wait for memory to respond.
127	E	x	x	xFB	334	0	8	0	111000	127	Strobe MEM LSB to accepl info: also copy R15 to R11.
128	E	x	x	xxC	337	3	A	0	111000	128	Strobe MEM MSB to accept info; transfer LSB to R12.
129	E	x	x	xxD	337	2	0	0	000000	129	Transfer MEM MSB to R13.
12A	3	1	0	OAO	144	0	0	0	000000	12A	Jump back to fetch microcode (address OAO).

The decoder associated with the bus destination control is connected in a different fashion. Note that the system clock has been connected to the low true enable of this decoder. Connecting the clock to the decoder enable in this fashion will assert the designated signal only during the last half of the cycle, which will prevent glitches from occurring on the decoder output lines. This prevents unwanted action to occur since these lines activate edge triggered functions. The I/O and memory bits do not have this enabling function, which indicates that the system designer was willing to live with the glitches which would occur on these

lines. If this is unacceptable. then steps must be taken to be sure that glitches do not cause unwanted results.

As with the horizontal microcode example, one of the best ways to get a feel for the system capabilities is to prepare microcode for it. Table 5.6 contains the microcode for the same instructions. ADD A, B and JUMP. The fields of Table 5.6 are similar to the fields of Table 5.5. The two differences are that the CON lines in Table 5.6 (10 bits, base 16 representation) combine the function of the SEQ ADR. DATA CONS. A ADR, and B ADR, fields of Table 5.5, and the bus and I/O lines are encoded in the vertical example, and hence represented in base 16. The code in Table 5.6 performs the same functions as that in Table 5.5, but more instructions are required. For example, at 0F2 of the horizontal code is an instruction that loads zero into three registers simultaneously. With the vertical example this requires the three instructions located at 0F2, 0F3, and 0F4. This is an example of the way that code will "grow" in the vertical dimension to perform a function, when compared to a horizontal implementation.

The above examples demonstrate that microcode is a technique that enables a designer to perform work with a state machine type of controller, and have the action dictated by the contents of a memory. The microinstruction register of the microcode machine serves the function of the present state register to follow the progress of the work to be performed, and the MIR and microcode memory combine do the work of the decode portion of a state machine. The net result is to allow assertion of control signals using techniques of low level programming. This permits nested subroutines and conditional jumps to be pan of a hardware designer's collection of usable techniques. The designer can then make design choices based on the constraints of his particular design to accomplish the goals of his system, using whatever combination of horizontal and vertical techniques may be most beneficial.

One final comment is in order concerning the design examples used in this chapter. The examples have become increasingly complex, starting with the simple, seven state controller for the FIR filter function, and ending with a controller capable of implementing the necessary control for an entire computer system. Thus, the microcode mechanisms can appear to be much more complex than the state machine or delay methods of control, when the principles on which all of the controllers are based are the same. The apparent complexity stems from the complexity of the data path being controlled, not from an inherently complex technique.

5.8. Microcode Machine Example: VAX 111780

The microcoded method of control implementation has been used by many machines since the appropriate memory technology became available. Each of these machines has a unique blend of techniques to generate its control signals. One of theese examples was introduced by Digital Equipment Corporation in the 1970s. This system. the VAX 11/780, is a 32-bit machine with general purpose computing capabilities. The system has been utilized for scientific, business, and office applications, and a variety of models with different speeds and complexity are now available.

The VAX 11/780 itself has a microcoded engine to control about 2,600 integrated circuits on 19 circuit boards. The clock cycle time of the system is 200 nsec, applicable to both the internal modules and the bus that allows con-

nection of memory modules and peripherals. The organization of the system is shown in Figure 5.42. The diagram is done to reflect the physical division of the system as well as the logical connections available. As can be seen from the figure. a number of data paths are used to transfer information between system components. The synchronous backplane interconnect (SBI) is the mechanism used to transfer information from memory and peripheral devices into the CPU itself. This bus is time shared between address and data, and the highest data rate will occur when an address is transferred, followed by two 32-bit data words. This results in a data rate of 8 bytes in 3 cycles (600 nsec). or 13.3 Mbytes per second. The SBI control interacts with devices on the SBI to perform whatever transfers are required by the system.

The internal data bus Is used to move information between any of the major system components as required by the system. This is in contrast to the other buses with a more specific purpose. The control store bus is composed of the microcode bits, and is used to control the action of all of the system components. The memory data bus is used to transfer information to and from memory. This includes the cache memory as well as the memory accessible via the SBI. The virtual address provides the address of information requested by the program in the virtual address space; this must be converted to an appropriate physical address, which will be placed on the physical address bus. Finally, the microprogram control bus is used to address the appropriate microcode word, which will be extracted from the control store and used to specify the appropriate action.

Some of the blocks connected by these buses are self-explanatory. The SBI control is used to contml the interaction with devices that transfer information via the SBI. The data cache is a small cache used to store the most recently used pieces of information. The translation buffer and decode, which ascertains the

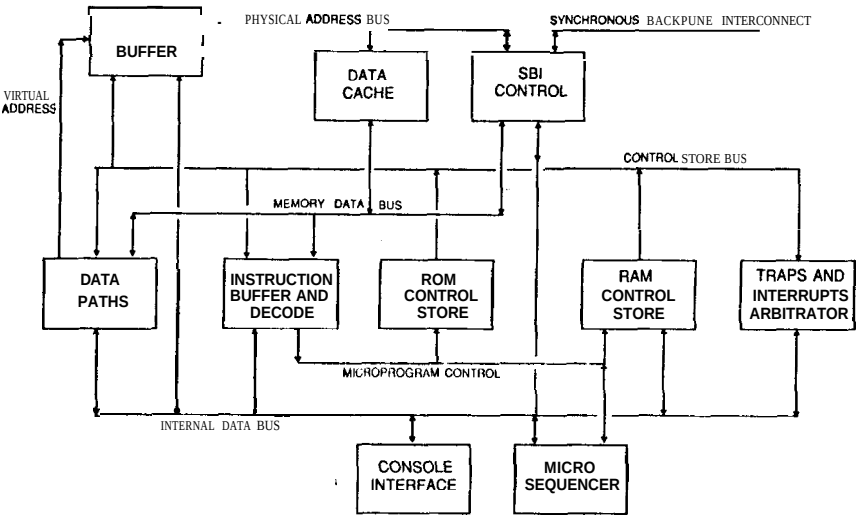


Figure 5.42. Block Diagram of the VAX 11/780 Computer.

machine instruction to be performed, provides the appropriate control to the microcode machine.

The microcode machine itself is contained in the micro sequencer, the ROM control store, and the RAM control store. The ROM control store contains the microcode for the basic instructions of the system; this includes a comprehensive set of variable length instructions for general computing, and some other functions needed by the system. The ROM control store provides storage for 4.096 microinstructions. The RAM control store serves the same basic function, storing microinstructions for system use. However, the content of the RAM control store must be provided by the user at an appropriate time, usually when the system is initialized. This capability of writing new information to the control memory is often called writable control store (WCS). The WCS can be used to provide corrections to faulty operations in permanent control store, or to speed up certain often executed sequences. such as operating system primatives. Having these functions in WCS allows changing them to grow with system needs or to correct faulty operation. It also allows users to tailor their system to enhance its operation in a specific environment. In any case, the system operates by having the micro sequencer supply a microcode address, and the ROM or RAM control store supplying the micminstruction.

The microinshuction is a 96-bit word whose format is given in Figure 5.43. The 96 bits are broken into 30 different fields, each of which controls part of the function of the machine. Table 5.7 identifies the various fields and the elements they control. Each microinstruction is capable of controlling the hardware of the system to do the work required. The techniques used are the same techniques we have already identified. Some fields provide an address to a multiplexer function (SMX, EBMX. RMX, KMX, etc.) to select one operand or source of data for a

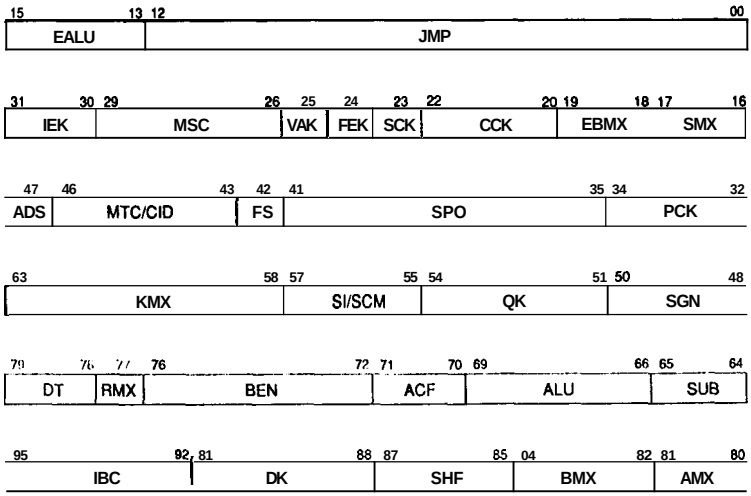


Figure 5.43. Fields of VAX 11/780 Microinstructions.

Table 5.7. Microcode Fields in VAX 111780.

Mnemonic	Name	Function
JMP	Jump Address	Address of next microinstruction.
EALU	Exponent ALU	Control ALU for exponent arithmetic.
SMX	SMX Select	Control selection of value for S operand.
EBMX	EBMX Select	Control selection of value for EB operand.
CCK	Condition Code	Identify bit for condition code test.
SCK		
FEK		
VA		
MSC	Miscellaneous	Control various functions not included elsewhere.
IEK	Interrupt and Exception	Control function of interrupt logic.
PCK	Address Count Control	Control program counter and address specification.
SPO	Scratch Pad Operation	Control function of scratch pad area.
FS	Function Select	Identify function of MCT/CID bits.
MCT/CID	Memory and Control Bus	Control bus transfers.
ADS	Address Select	Identify source of effective address.
SGN	Sign Control	Identify source of sign bit.
QK	Q Reg Control	Control action of Q register.
SI/SCM	Shift Input Control	Control action of shift network.
KMX	Constants Select	Select source, value of constants.
SUB	Subroutine Control	Provide control for subroutine linkage.
ALU	ALU Control	Specify function performed by ALU.
ACF	Accelerator Control	Identify function of accelerator.
BEN	Branch Enable	Control branching function.
RMX	Reg Mux Control	Specify source of operand in reg mux.
DT	Data Type	Identify type of data being operated on.
AMX	A Mux Select	Control value supplied by A Mux.
BMX	B Mux Select	Control value supplied by B Mux.
SHF	ALU Shift Control	Control action of ALU shifter.
DK	D Reg Control	Control action of operand in D register.
IBC	Instruction Buffer Control	Specify action of instruction buffer.

specific register or function. Other fields use a single bit to identify the function of another field (FS). The ALU bits directly control the function of the arithmetic element in the system. The SUB bits identify the action to be taken on a microcode subroutine. Each of the fields controls action to occur somewhere in the system in each 200 nsec clock cycle.

One of the interesting techniques exemplified by this system is the selection of the next microinstruction. The JMP field of each microinstruction, which is *not* multiplexed with any other function, identifies the next microinstruction to be executed Thus, there is no requirement that successive microinstructions be located in successive locations in microstore. The microinstructions can be located in any

available locations in memory. If some choice of next microinstruction is required — that is, if a conditional branch of some type is needed — this is controlled by the BEN field. The effect is to modify the address in the JMP field in some predetermined fashion. For example, the JMP field can provide the most significant portion of an address, and the least significant bits can be provided by sign bits, processor state indicators, or other machine information. This provides a multiway branch capability, so that the next instruction is one of 8, 16, or 32 possible instructions, based on the function selected by the BEN field. The multiway branch ability permits multiple decisions simultaneously, since more than one bit can be used in the selection of the next microinstruction. The requirement for this to be effective is that the set of instructions that are possible next instructions for a specific microinstruction be located at an appropriate address boundary in the microcode. This is one of the reasons that each microinstruction carries the address of the next microinstruction, since placing sets of next instructions on address boundaries fragments the available microcode memory.

The VAX 11/780 is a good example of a microcoded system, but certainly not the only example. The technique has been used in a host of different machines to provide a programmable control system that can be utilized in a regular system fashion.

5.9. Control System Design: Asserting Control Lines In a Timely Fashion

This chapter has dealt with the concepts and practices involved in producing a control system for a digital device. The device can be as simple as a counter or as complex as a computer, but the principles involved in the process are the same. Before the design of the control system can begin, it is imperative that the data path be defined, and that the appropriate control lines be identified. This process must not only identify the lines to be controlled, but also specify the assertion levels required to perform the work. Armed with this information the designer can then proceed to provide a control section which will assert the lines in an appropriate fashion.

Once the set of signals required for control of a system has been identified, then the order of assertion and other specific information must be determined. This process requires that the designer be familiar with the system components. their uses, and their limitations. But the action of the control section can be specified by utilizing system knowledge, design techniques, and desired behavioral characteristics. This specification may take the form of a state diagram, which is useful for direct implementation of state machines. Or it may take the form of a flow diagram, which can provide the basis for a delay line or shift register method of control signal implementation. The state diagram or flow diagram can also be useful in preparing a system that utilizes microcode techniques for asserting the control lines. Each of the techniques can be effectively utilized where the system characteristics call for behavior of one type or another. For example, RISC machines generally need extremely fast control, but being relatively simple the amount of logic required for direct implementation permits random logic in the control system. On the other hand, CISC machines often require numerous steps and decisions to perform a specific instruction, so microcode is very appropriate. The designer, then, has the responsibility of selecting the implementation technique that will maximize effective use of system resources.

5.10. Problems

5.1 Consider the block diagram given in Figure P5.1 for the data path of a computer. This data path is to be used to implement a single address machine. The following information about the machine may or may not be useful. The add is not cascadable (no double precision adds.) The memory is fast RAM. The MAR ignores bits higher than its address space, which are transferred to it. The only control lines you have access to are those listed on the diagram. No initialization is needed for any of the logic. Design a sequencer for this data path that will do SUBTRACT, LEFT-SHIFT, and NEGATE. Use microcoding techniques. Include an RTL description of the transfers necessary. Give a logic diagram (at a reasonable level) of the control section, specify the bits in the microinstruction word, and give the microcode needed. The available microcode sequencer has the following pins: address out, address in, JUMP-H (test input) and CONTROL-H. When CONTROL-H is L (normal case) the instruction obtained is the next in sequence. When CONTROL-H is H, then an conditional jump is performed, with the address input being used as the source of address if JUMP-H is asserted. The ALU is capable of the following operations:

C1	C0	*Function*
0	0	F = A + B
0	1	F = A nand B
1	0	F = not B
1	1	F = A or B

5.2 For the single address machine shown in Figure P5.1, design a sequencer that will do ADD or OR, using some technique other than microcoding. Include RTL description of the transfers needed and the logic of the control section.

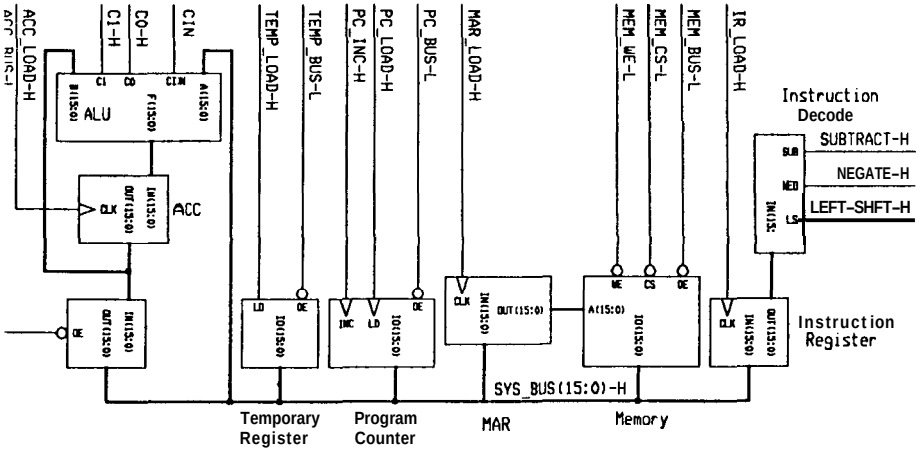


Figure P5.1. Block Diagram for Single Address Machine.

5.3 Consider the controller for the finite impulse response digital filter of Section 5.3. Implement the controller using the technique of individual delays. How can the loop be implemented?

5.4 Consider the controller for the finite impulse response digital filter of Section 5.3. Implement the controller using the shift register technique to develop the timing pulses. How can the loop be implemented?

5.5 The state machine controller shown in Figure 5.27 can be used to implement the state diagram shown in Figure 5.23. Provide the necessary additional details, and specify the contents of the memory to create the system. That is:

a. Identify which of the address lines of the memory are provided by the present state lines, and which are provided by external inputs.

b. Identify which of the memory outputs are used for next state determination and which are used for signal generation.

c. Create a table that specifies the next state patterns and the signal line patterns for each of the appropriate addresses.

A computer system can be a very valuable tool for this project, using a simple program to help develop the basic patterns, and an editor system to modify the patterns as necessary.

5.6 Design a control system using the state machine technique for the shift and add multiply system of Figure 3.9. Implement the state machine with the multiplexer method (like Figure 5.13). and then do the design with random logic for next state determination. Compare the two implementations based on board space, power consumption, and ease of implementation.

5.7 Design a controller for the Booth's algorithm multiplier of Figure 3.14. Use a state machine implementation mechanism.

5.8 Consider the block diagram of a portion of the control system for a unit that utilizes a microcoded organization. The contents of the microcode memory for the original organization are as follows:

addr	*contents of addr*
0	01001000111110100111011000110 10
1	10110111000110100100100100010 01
2	01001000111110100111011000110 10
3	11010010001001000100100010000 10
4	10110111000110100100100100010 01
5	00101001001001010010010010010 00
6	01001000111110100111011000110 10
7	11010010001001000100100010000 10
8	00111001001010100010100100010 01
9	10110111000110100100100100010 01
10	10010010010100100101001001010 00
11	11010010001001000100100010000 10
12	01001000111110100111011000110 10
13	10110111000110100100100100010 01
14	11010010001001000100100010000 10

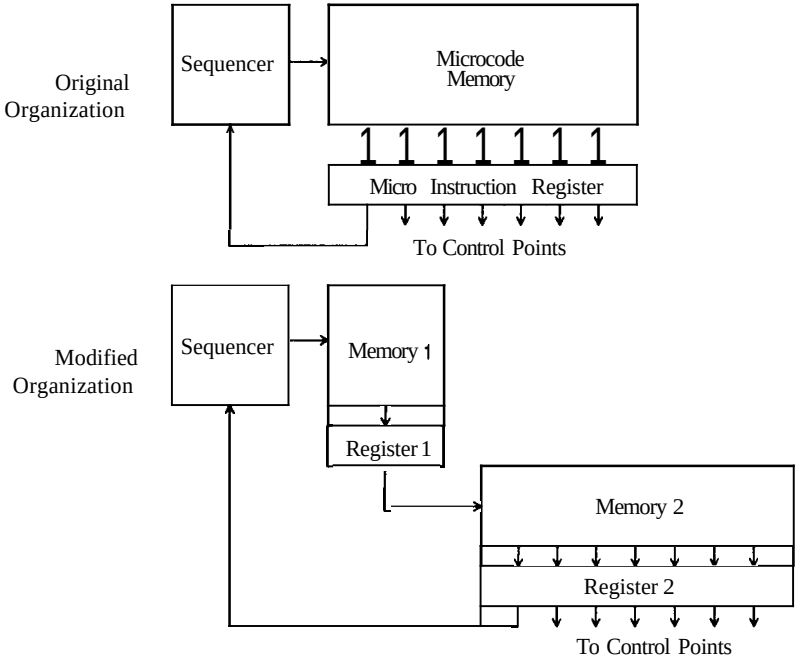


Figure P5.8. Alternative Microcode Organizations.

Give the contents of Memory 1 and Memory 2 for the modified organization. What are the advantages, if any, of the second organization?

5.9 Design a state machine controller for the divider shown in Figure 3.23. (See also Appendix B.) Use a memory for the next state logic, and specify the contents of the memory for all of the appropriate locations.

5.10 Consider the horizontally organized microcode system of Figure **5.40;** details of this implementation are shown in Appendix B. Some microcode appears in Table **5.5.** Write microcode to implement a **JUMP** instruction, a **JUMP TO SUBROUTINE** instruction, and a **RETURN** instruction. State whatever assumptions that you need to make.

5.11 Modify the microcode shown in Table **5.5** so that the fetch portion of the microcode checks for the existence of an interrupt. Include microcode to handle the interrupt. State whatever assumptions you need to make.

5.12 Consider the vertically organized microcode system of Figure **5.41.** Some of the microcode appears in Table **5.6.** Write microcode for a memory-to-memory add instruction with the format ADD **<addr1> <addr2>.** Assume that the two addresses **(<addr1>** and **<addr2>)** are stored in locations directly following the instruction in program memory.

5.13 Obtain data sheets for the devices of Figure **5.40** (see also Appendix B) and determine the minimum cycle time for the data path and the minimum cycle time for the control path.

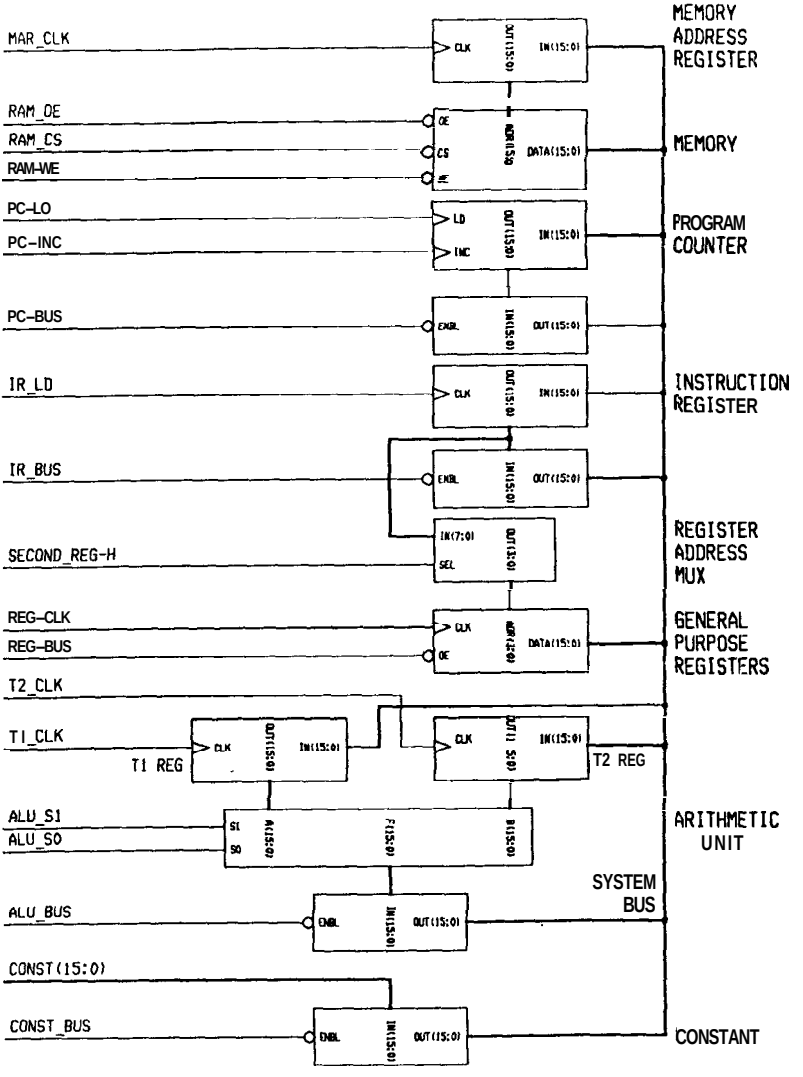


Figure P5.14. Logic for Data Path of 16-Bit Computer System.

5.14 The diagram shown in Figure P5.14 is a single bus system that can be used to implement a simple 16-bit machine. Using horizontal microcode techniques, create microcode to perform the ADD instruction with possibilities for register direct, register indirect, register indirect autoincrement (post increment), and register indirect auto-decrement (predecrement). That is:

a. determine the contents of the microinstruction register

b. complete the naming for the control signals of the system

c. identify the work needed **(RTL)** to perform the **ADD** instruction

d. give the microcode bit patterns needed to accomplish the work

State any assumptions needed to complete the specification.

5.15 The diagram shown in Figure P5.14 can be used to implement a 16-bit machine. Using vertical microcode techniques, create microcode to perform a **NEGATE** instruction, an **INCREMENT** instruction, and a **CLEAR** instruction. To accomplish this:

a. determine which control lines can be activated by a decoded field in a **microinstruction** word, and which lines can share a field

b. specify the contents of the microinstruction register

c. complete the naming for the control signals of the system

d. give the RTL needed to perform the instructions

e. give the microcode bit patterns needed to perform the work of the **RTL** specified

State any assumptions needed to complete the specification.

5.16 For the **data** path shown in Figure P5.14, implement with horizontal microcode techniques a **JUMP SUBROUTINE** instruction and a **RETURN** (from subroutine) instruction. Include with the answer the following information:

a. the format of the microinstruction

b. the **proper** naming of the control signals

c. the RTL for the instructions

d. the microcode bit patterns that will implement the instructions

Assume that R15 of the register set is designated as the stack pointer. Also assume that the memory for this problem is slow **RAM,** and that once the **MAR** has been set, three cycles are needed to **write/read** the memory.

5.17 Repeat **Problem** 5.16, using vertical microcode techniques. How can the bus **destination** signals be created in a manner that will prevent glitches from occurring? For the system of Figure P5.14, the following information may be useful:

The memory write enable, chip select, and output enable are all asserted low.

The program counter can be loaded or incremented; both actions occur on **the** low-to-high transition of the control signal.

When one register is identified by an instruction, the register address mux will feed the correct lines to the general purpose registers.

When **two** registers are specified by an instruction, the register address **mux** will feed the pattern identifying the first register to the general purpose registers unless **SECOND–REG-H** is asserted, at which point the pattern identifying the second register will be directed to the general purpose registers.

The arithmetic unit operates according to the following table:

S1	S0	Function
0	0	F = A plus B
0	1	F = A nand B
1	0	F = B
1	1	F = A or B

The CONST field allows the controller to place a known value into the system.

5.11. References and Readings

[AMD85] Advanced Micm Devices, ***Bipolar Microprocessor Logic and Interface Data Book.*** Sunnyvale. CA: Advanced Micro Devices, 1985.

[AMD88] Advanced Micro Devices. PAL ***Device Handbook.*** Sunnyvale, CA: Advanced Micm Devices. 1988.

[AnLe81] Anderson, T., and P. A. Lee. ***Fault Tolerance, Principles and Practice.*** Englewood Cliffs, NJ: Prentice Hall International, 1981.

[Andr80] Andrews, M.. ***Principles of Firmware Engineering in Microporgram Control.*** Potomac, MD: Computer Science Press. 1980.

[Arms81] Armstrong, R. A., "Applying CAD to Gate Arrays Speeds 32-bit Minicomputer Design," ***Electronics.*** Vol. 54, No. 1, January 13, 1981, pp. 167–173.

[BaRa82] Banerji, D. K., and J. Raymond, ***Elements of Microprogramming.*** Englewood Cliffs, NJ: Prentice-Hall, 1982.

[Bart85] Bartee, T. C., ***Digital Computer Fundamentals.*** 6th edition. New Yo*: McGraw Hill Book Company. 1985.

[Boot84] Booth. T. L.. ***Introduction lo Computer Engineering: Hardware and Software Design.*** New Yo*: John Wiley & Sons. 1984.

[Bree89] Breeding. K. J., ***Digital Design Fundamentals.*** Englewood Cliffs, NJ: Prentice Hall, 1989.

[BuCa84] Burger, R. M., R. K. Calvin, W. C. Holton, et al., "The Impact of ICs on Computer Technology," ***Computer.*** Vol. 17, No. 10, October 1984, pp. 88–96.

[Damm85] Damm, W., "Design and Specification of Microprogrammed Computer Architectures." ***Proceeding of the 18th Annual Workshop on Microprogramming.*** Washington, DC: IEEE Computer Society Press. 1985, pp. 3–9.

[Dasg80] Dasgupta, S., "Some Aspects of High Level Microprogramming," ***ACM Computing Surveys.*** Vol. 12, No. 3, September 1980, pp. 295–324.

[Dasg79] Dasgupta, S., "The Organization of Microprogram Stores," ***ACM Computing Surveys.*** Vol. 11, No. 1, March 1979, pp. 39–65.

[Davi86] Davidson. S., "Progress in High Level Microprogramming," ***IEEE Software.*** Vol. 3, No. 4, July 1986, pp. 19–26.

[ErLa85] Ercegovac, M. D., and T. Lang, ***Digital Systems and Hardware/Firmware Algorithms.*** New York: John Wiley & Sons. 1985.

[Flet80] Fletcher, W. L ***An Engineering Approach to Digital Design.*** Englewood Cliffs, NJ: Prentice Hall, 1980.