CHAPTER 20

MICROPROGRAMMED CONTROL

KEY POINTS

♦ An alternative to a hardwired control unit is a microprogrammed control unit, in which the logic of the control unit is specified by a microprogram. A microprogram consists of a sequence of instructions in a microprogramming language. These are very simple instructions that specify micro-operations.
♦ A microprogrammed control unit is a relatively simple logic circuit that is capable of (1) sequencing through microinstructions and (2) generating control signals to execute each microinstruction.
♦ As in a hardwired control unit, the control signals generated by a microinstruction are used to cause register transfers and ALU operations.

The term *microprogram* was first coined by M. V. Wilkes in the early 1950s [WILK51]. Wilkes proposed an approach to control unit design that was organized and systematic and avoided the complexities of a hardwired implementation. The idea intrigued many researchers but appeared unworkable because it would require a fast, relatively inexpensive control memory.

The state of the microprogramming art was reviewed by *Datamation* in its February 1964 issue. No microprogrammed system was in wide use at that time, and one of the papers [HILL64] summarized the then-popular view that the future of microprogramming "is somewhat cloudy. None of the major manufacturers has evidenced interest in the technique, although presumably all have examined it."

This situation changed dramatically within a very few months. IBM's System/360 was announced in April, and all but the largest models were microprogrammed. Although the 360 series predated the availability of semiconductor ROM, the advantages of microprogramming were compelling enough for IBM to make this move. Microprogramming became a popular technique for implementing the control unit of CISC processors. In recent years, microprogramming has become less used but remains a tool available

to computer designers. For example, as we have seen, on the Pentium 4, machine instructions are converted into a RISC-like format most of which are executed without the use of microprogramming. However, some of the instructions are executed using microprogramming.

20.1 BASIC CONCEPTS

Microinstructions

The control unit seems a reasonably simple device. Nevertheless, to implement a control unit as an interconnection of basic logic elements is no easy task. The design must include logic for sequencing through micro-operations, for executing micro-operations, for interpreting opcodes, and for making decisions based on ALU flags. It is difficult to design and test such a piece of hardware. Furthermore, the design is relatively inflexible. For example, it is difficult to change the design if one wishes to add a new machine instruction.

An alternative, which has been used in many CISC processors, is to implement a microprogrammed control unit.

Consider Table 20.1. In addition to the use of control signals, each micro-operation is described in symbolic notation. This notation looks suspiciously like a programming language. In fact it is a language, known as a **microprogramming language**. Each line describes a set of micro-operations occurring at one time and is known as a **microinstruction**. A sequence of instructions is known as a **microprogram**, or *firmware.* This latter term reflects the fact that a microprogram is midway between hardware and software. It is easier to design in firmware than hardware, but it is more difficult to write a firmware program than a software program.
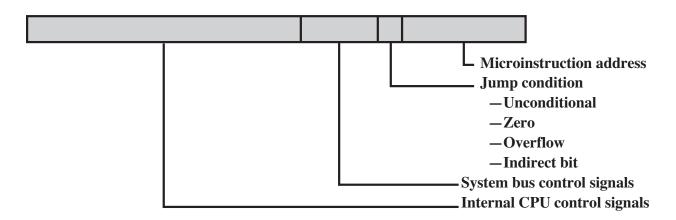
Table 20.1 Machine Instruction Set for Wilkes Example

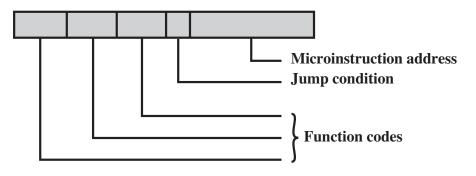| Order	Effect of Order
$A\ n$	$C(Acc) + C(n)$ to Acc_1
$S\ n$	$C(Acc) - C(n)$ to Acc_1
$H\ n$	$C(n)$ to Acc_2
$V\ n$	$C(Acc2) \times C(n)$ to Acc, where $C(n) \geq 0$
$T\ n$	$C(Acc_1)$ to $n, 0$ to Acc
$U\ n$	$C(Acc_1)$ to n
$R\ n$	$C(Acc) \times 2^{-(n+1)}$ to Acc
$L\ n$	$C(Acc) \times 2^{n+1}$ to Acc
$G\ n$	IF $C(Acc) < 0$, transfer control to n; if $C(Acc) \geq 0$, ignore (i.e., proceed serially)
$I\ n$	Read next character on input mechanism into n
$O\ n$	Send $C(n)$ to output mechanism

Notation: Acc = accumulator
Acc_1 = most significant half of accumulator
Acc_2 = least significant half of accumulator
n = storage location n
$C(X)$ = contents of X (X = register or storage location)

How can we use the concept of microprogramming to implement a control unit? Consider that for each micro-operation, all that the control unit is allowed to do is generate a set of control signals. Thus, for any micro-operation, each control line emanating from the control unit is either on or off. This condition can, of course, be represented by a binary digit for each control line. So we could construct a *control word* in which each bit represents one control line. Then each micro-operation would be represented by a different pattern of 1s and 0s in the control word.

Suppose we string together a sequence of control words to represent the sequence of micro-operations performed by the control unit. Next, we must recognize that the sequence of micro-operations is not fixed.

(a) Horizontal microinstruction

(b) Vertical microinstruction

Figure 20.1 Typical Microinstruction Formats

Sometimes we have an indirect cycle; sometimes we do not. So let us put our control words in a memory, with each word having a unique address. Now add an address field to each control word, indicating the location of the next control word to be executed if a certain condition is true (e.g., the indirect bit in a memory-reference instruction is 1). Also, add a few bits to specify the condition.

The result is known as a **horizontal microinstruction**, an example of which is shown in Figure 20.1a. The format of the microinstruction or control word is as follows. There is one bit for each internal processor control line

and one bit for each system bus control line. There is a condition field indicating the condition under which there should be a branch, and there is a field with the address of the microinstruction to be executed next when a branch is taken. Such a microinstruction is interpreted as follows:

1. To execute this microinstruction, turn on all the control lines indicated by a 1 bit; leave off all control lines indicated by a 0 bit. The resulting control signals will cause one or more micro-operations to be performed.
2. If the condition indicated by the condition bits is false, execute the next microinstruction in sequence.
3. If the condition indicated by the condition bits is true, the next microinstruction to be executed is indicated in the address field.

Figure 20.2 shows how these control words or microinstructions could be arranged in a **control memory**. The microinstructions in each routine are to be executed sequentially. Each routine ends with a branch or jump instruction indicating where to go next. There is a special execute cycle routine whose only purpose is to signify that one of the machine instruction routines (AND, ADD, and so on) is to be executed next, depending on the current opcode.

The control memory of Figure 20.2 is a concise description of the complete operation of the control unit. It defines the sequence of micro-operations to be performed during each cycle (fetch, indirect, execute, interrupt), and it specifies the sequencing of these cycles. If nothing else, this notation would be a useful device for documenting the functioning of a control unit for a particular computer. But it is more than that. It is also a way of implementing the control unit.

• • • **Jump to indirect or execute**	**Fetch cycle routine**
• • • **Jump to execute**	**Indirect cycle routine**
• • • **Jump to fetch**	**Interrupt cycle routine**
Jump to opcode routine	**Execute cycle beginning**
• • • **Jump to fetch or interrupt**	**AND routine**
• • • **Jump to fetch or interrupt**	**ADD routine**

•
•
•

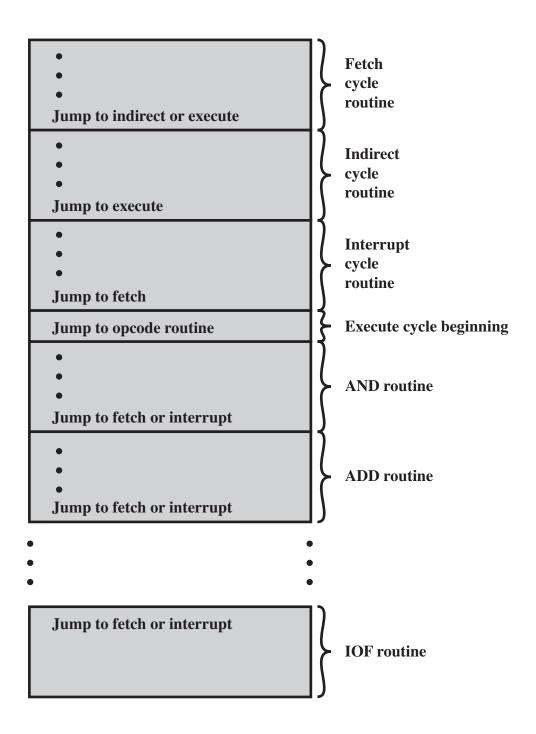|	
Jump to fetch or interrupt	**IOF routine**

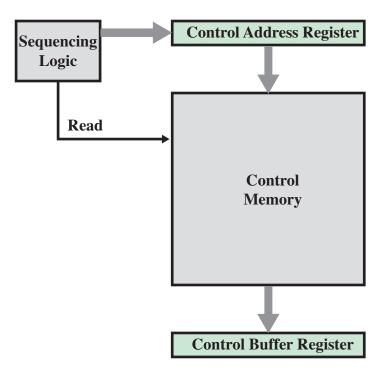**Figure 20.2   Organization of Control Memory**

Figure 20.3 Control Unit Microarchitecture

Microprogrammed Control Unit

The control memory of Figure 20.2 contains a program that describes the behavior of the control unit. It follows that we could implement the control unit by simply executing that program.

Figure 20.3 shows the key elements of such an implementation. The set of microinstructions is stored in the *control memory.* The *control address register* contains the address of the next microinstruction to be read. When a microinstruction is read from the control memory, it is transferred to a *control buffer register.* The left-hand portion of that register (see Figure 20.1a) connects to the control lines emanating from the control unit. Thus, *reading* a microinstruction from the control memory is the same as *executing* that microinstruction. The third element shown in the figure is a sequencing unit that loads the control address register and issues a read command.
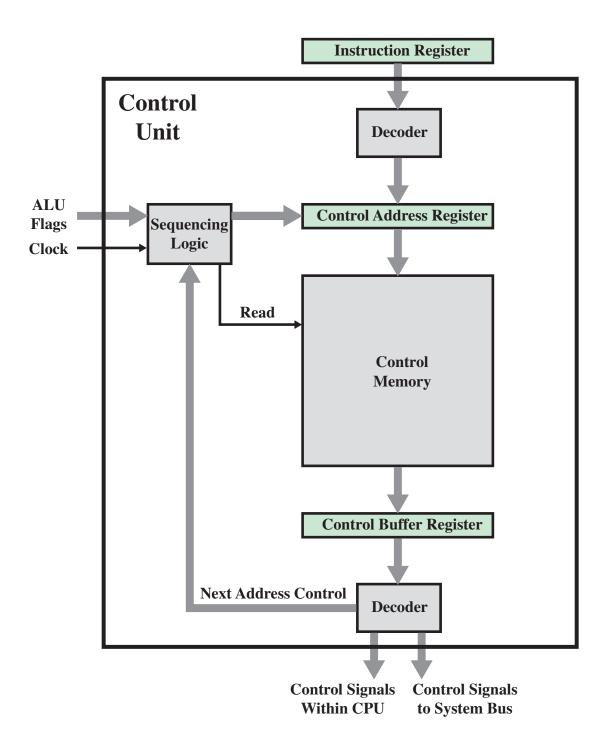
Figure 20.4 Functioning of Microprogrammed Control Unit

Let us examine this structure in greater detail, as depicted in Figure 20.4. Comparing this with Figure 20.3, we see that the control unit still has

the same inputs (IR, ALU flags, clock) and outputs (control signals). The control unit functions as follows:

1. To execute an instruction, the sequencing logic unit issues a READ command to the control memory.
2. The word whose address is specified in the control address register is read into the control buffer register.
3. The content of the control buffer register generates control signals and next-address information for the sequencing logic unit.
4. The sequencing logic unit loads a new address into the control address register based on the next-address information from the control buffer register and the ALU flags.

All this happens during one clock pulse.

The last step just listed needs elaboration. At the conclusion of each microinstruction, the sequencing logic unit loads a new address into the control address register. Depending on the value of the ALU flags and the control buffer register, one of three decisions is made:

- **Get the next instruction:** Add 1 to the control address register.
- **Jump to a new routine based on a jump microinstruction:** Load the address field of the control buffer register into the control address register.
- **Jump to a machine instruction routine:** Load the control address register based on the opcode in the IR.

Figure 20.4 shows two modules labeled *decoder.* The upper decoder translates the opcode of the IR into a control memory address. The lower decoder is not used for horizontal microinstructions but is used for **vertical**

microinstructions (Figure 20.1b). As was mentioned, in a horizontal microinstruction every bit in the control field attaches to a control line. In a vertical microinstruction, a code is used for each action to be performed [e.g., MAR ← (PC)], and the decoder translates this code into individual control signals. The advantage of vertical microinstructions is that they are more compact (fewer bits) than horizontal microinstructions, at the expense of a small additional amount of logic and time delay.

Wilkes Control

As was mentioned, Wilkes first proposed the use of a microprogrammed control unit in 1951 [WILK51]. This proposal was subsequently elaborated into a more detailed design [WILK53]. It is instructive to examine this seminal proposal.

The configuration proposed by Wilkes is depicted in Figure 20.5. The heart of the system is a matrix partially filled with diodes. During a machine cycle, one row of the matrix is activated with a pulse. This generates signals at those points where a diode is present (indicated by a dot in the diagram). The first part of the row generates the control signals that control the operation of the processor. The second part generates the address of the row to be pulsed in the next machine cycle. Thus, each row of the matrix is one microinstruction, and the layout of the matrix is the control memory.

At the beginning of the cycle, the address of the row to be pulsed is contained in Register I. This address is the input to the decoder, which, when activated by a clock pulse, activates one row of the matrix. Depending on the control signals, either the opcode in the instruction register or the second part of the pulsed row is passed into Register II during the cycle. Register II is then gated to Register I by a clock pulse. Alternating clock pulses are used to activate a row of the matrix and to transfer from Register II to Register I. The two-register arrangement is needed because the

Figure 20.5   Wilkes's Microprogrammed Control Unit

decoder is simply a combinatorial circuit; with only one register, the output would become the input during a cycle, causing an unstable condition.

This scheme is very similar to the horizontal microprogramming approach described earlier (Figure 20.1a). The main difference is this: In the previous description, the control address register could be incremented by one to get the next address. In the Wilkes scheme, the next address is contained in the microinstruction. To permit branching, a row must contain two address parts, controlled by a conditional signal (e.g., flag), as shown in the figure.

Having proposed this scheme, Wilkes provides an example of its use to implement the control unit of a simple machine. This example, the first

known design of a microprogrammed processor, is worth repeating here because it illustrates many of the contemporary principles of microprogramming.

The processor of the hypothetical machine (the example machine by Wilkes) includes the following registers:

A   multiplicand
B   accumulator (least significant half)
C   accumulator (most significant half)
D   shift register

In addition, there are three registers and two 1-bit flags accessible only to the control unit. The registers are as follows:

E   serves as both a memory address register (MAR) and temporary storage
F   program counter
G   another temporary register; used for counting

Table 20.2 Microinstructions for Wilkes Example

Notation: $A, B, C,\dots$ stand for the various registers in the arithmetical and control register units. C to D indicates that the switching circuits connect the output of register C to the input register D; $(D + A)$ to C indicates that the output register of A is connected to the one input of the adding unit (the output of D is permanently connected to the other input), and the output of the adder to register C. A numerical symbol n in quotes (e.g., $'n'$) stands for the source whose output is the number n in units of the least significant digit.

	Arithmetical Unit	Control Register Unit	Conditional Flip-Flop		Next Micro-instruction	
			Set	Use	0	1
0		F to G and E			1	
1		(G to '1') to F			2	
2		Store to G			3	
3		G to E			4	

	4	*E* to decoder				—	
A	5	*C* to *D*				16	
S	6	*C* to *D*				17	
H	7	Store to *B*				0	
V	8	Store to *A*				27	
T	9	*C* to Store				25	
U	10	*C* to Store				0	
R	11	*B* to *D*	*E* to *G*			19	
L	12	*C* to *D*	*E* to *G*			22	
G	13		*E* to *G*	$(1)C_5$		18	
I	14	Input to Store				0	
O	15	Store to Output				0	
	16	$(D + \text{Store})$ to *C*				0	
	17	$(D - \text{Store})$ to *C*				0	
	18				1	0	1
	19	*D* to *B* (R)*	$(G - \text{'1'})$ to *E*			20	
	20	*C* to *D*		$(1)E_5$		21	
	21	*D* to *C* (R)			1	11	0
	22	*D* to *C* (L)†	$(G - \text{'1'})$ to *E*			23	
	23	*B* to *D*		$(1)E_5$		24	
	24	*D* to *B* (L)			1	12	0
	25	'0' to *B*				26	
	26	*B* to *C*				0	
	27	'0' to *C*	'18' to *E*			28	
	28	*B* to *D*	*E* to *G*	$(1)B_1$		29	
	29	*D* to *B* (R)	$(G - \text{'1'})$ to *E*			30	
	30	*C* to *D* (R)		$(2)E_5$	1	31	32
	31	*D* to *C*			2	28	33
	32	$(D + A)$ to *C*			2	28	33
	33	*B* to *D*		$(1)B_1$		34	
	34	*D* to *B* (R)				35	
	35	*C* to *D* (R)			1	36	37
	36	*D* to *C*				0	
	37	$(D - A)$ to *C*				0	

*Right shift. The switching circuits in the arithmetic unit are arranged so that the least significant digit of the register *C* is placed in the most significant place of register *B* during right shift micro-operations, and the most significant digit of register *C* (sign digit) is repeated (thus making the correction for negative numbers).

†Left shift. The switching circuits are similarly arranged to pass the most significant digit of register *B* to the least significant place of register *C* during left shift micro-operations.

Table 20.1 lists the machine instruction set for this example. Table 20.2 is the complete set of microinstructions, expressed in symbolic form, that implements the control unit. Thus, a total of 38 microinstructions is all that is required to define the system completely.

The first full column gives the address (row number) of each microinstruction. Those addresses corresponding to opcodes are labeled. Thus, when the opcode for the add instruction (A) is encountered, the microinstruction at location 5 is executed. Columns 2 and 3 express the actions to be taken by the ALU and control unit, respectively. Each symbolic expression must be translated into a set of control signals (microinstruction bits). Columns 4 and 5 have to do with the setting and use of the two flags (flip-flops). Column 4 specifies the signal that sets the flag. For example, $(1)C_s$ means that flag number 1 is set by the sign bit of the number in register C. If column 5 contains a flag identifier, then columns 6 and 7 contain the two alternative microinstruction addresses to be used. Otherwise, column 6 specifies the address of the next microinstruction to be fetched.

Instructions 0 through 4 constitute the fetch cycle. Microinstruction 4 presents the opcode to a decoder, which generates the address of a microinstruction corresponding to the machine instruction to be fetched. The reader should be able to deduce the complete functioning of the control unit from a careful study of Table 20.2.

Advantages and Disadvantages

The principal advantage of the use of microprogramming to implement a control unit is that it simplifies the design of the control unit. Thus, it is both cheaper and less error prone to implement. A *hardwired* control unit must contain complex logic for sequencing through the many micro-operations of

the instruction cycle. On the other hand, the decoders and sequencing logic unit of a microprogrammed control unit are very simple pieces of logic.

The principal disadvantage of a microprogrammed unit is that it will be somewhat slower than a hardwired unit of comparable technology. Despite this, microprogramming is the dominant technique for implementing control units in pure CISC architectures, due to its ease of implementation. RISC processors, with their simpler instruction format, typically use hardwired control units. We now examine the microprogrammed approach in greater detail.

20.2 MICROINSTRUCTION SEQUENCING

The two basic tasks performed by a microprogrammed control unit are as follows:

- **Microinstruction sequencing:** Get the next microinstruction from the control memory.
- **Microinstruction execution:** Generate the control signals needed to execute the microinstruction.

In designing a control unit, these tasks must be considered together, because both affect the format of the microinstruction and the timing of the control unit. In this section, we will focus on sequencing and say as little as possible about format and timing issues. These issues are examined in more detail in the next section.

Design Considerations

Two concerns are involved in the design of a microinstruction sequencing technique: the size of the microinstruction and the address-generation time.

The first concern is obvious; minimizing the size of the control memory reduces the cost of that component. The second concern is simply a desire to execute microinstructions as fast as possible.

In executing a microprogram, the address of the next microinstruction to be executed is in one of these categories:

- Determined by instruction register
- Next sequential address
- Branch

The first category occurs only once per instruction cycle, just after an instruction is fetched. The second category is the most common in most designs. However, the design cannot be optimized just for sequential access. Branches, both conditional and unconditional, are a necessary part of a microprogram. Furthermore, microinstruction sequences tend to be short; one out of every three or four microinstructions could be a branch [SIEW82]. Thus, it is important to design compact, time-efficient techniques for microinstruction branching.

Sequencing Techniques

Based on the current microinstruction, condition flags, and the contents of the instruction register, a control memory address must be generated for the next microinstruction. A wide variety of techniques have been used. We can group them into three general categories, as illustrated in Figures 20.6 to 20.8. These categories are based on the format of the address information in the microinstruction:

- Two address fields
- Single address field

• Variable format

The simplest approach is to provide two address fields in each microinstruction. Figure 20.6 suggests how this information is to be used. A multiplexer is provided that serves as a destination for both address fields plus the instruction register. Based on an address-selection input, the multiplexer transmits either the opcode or one of the two addresses to the control address register (CAR). The CAR is subsequently decoded to produce the next microinstruction address. The address-selection signals are provided by a branch logic module whose input consists of control unit flags plus bits from the control portion of the microinstruction.

Although the two-address approach is simple, it requires more bits in the microinstruction than other approaches. With some additional logic, savings can be achieved. A common approach is to have a single address field (Figure 20.7). With this approach, the options for next address are as follows:

• Address field
• Instruction register code
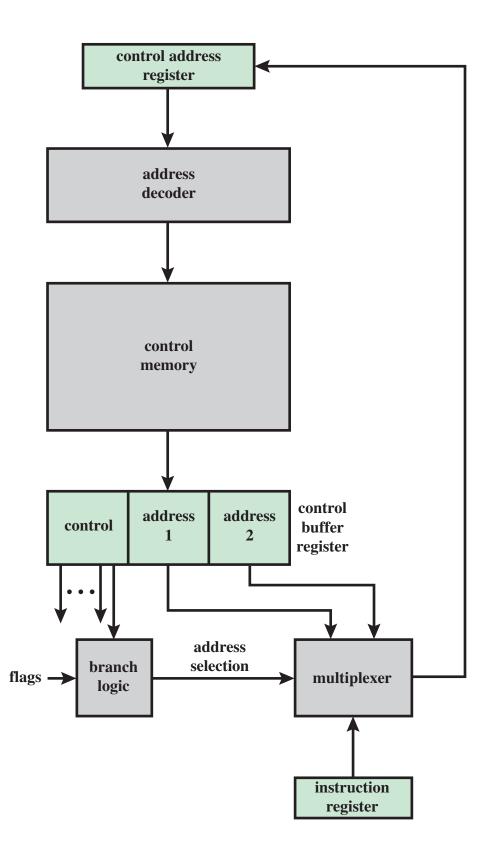• Next sequential address

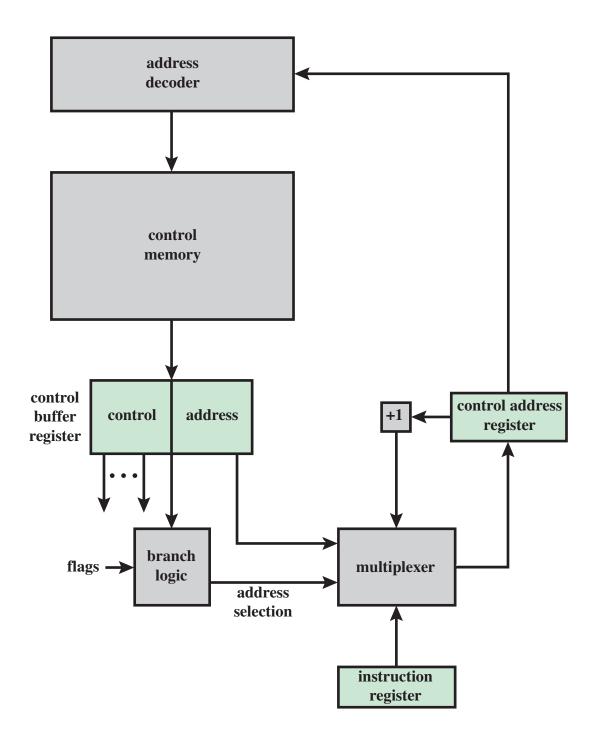**Figure 20.6   Branch Control Logic: Two Address Fields**

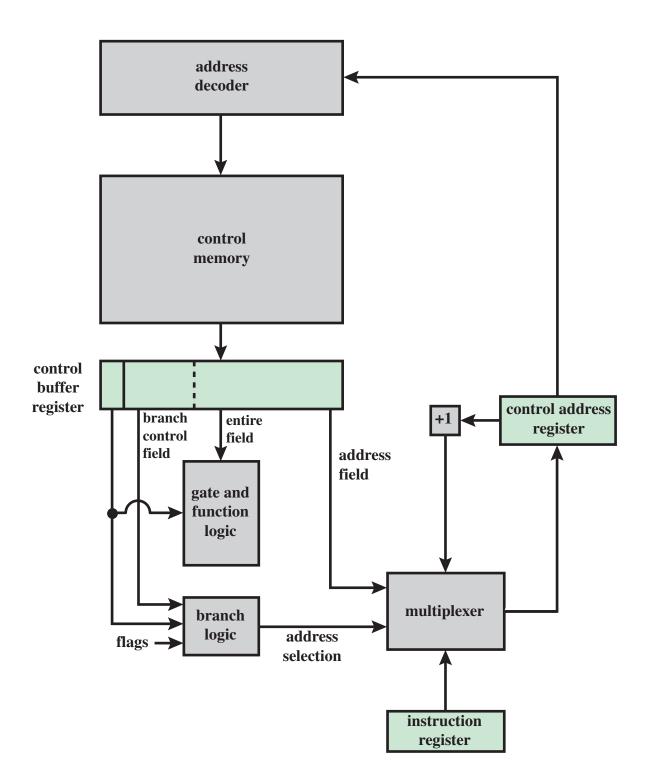**Figure 20.7   Branch Control Logic: Single Address Field**

Figure 20.8   Branch Control Logic: Variable Format

The address-selection signals determine which option is selected. This approach reduces the number of address fields to one. Note, however, that the address field often will not be used. Thus, there is some inefficiency in the microinstruction coding scheme.

Another approach is to provide for two entirely different microinstruction formats (Figure 20.8). One bit designates which format is being used. In one format, the remaining bits are used to activate control signals. In the other format, some bits drive the branch logic module, and the remaining bits provide the address. With the first format, the next address is either the next sequential address or an address derived from the instruction register. With the second format, either a conditional or unconditional branch is being specified. One disadvantage of this approach is that one entire cycle is consumed with each branch microinstruction. With the other approaches, address generation occurs as part of the same cycle as control signal generation, minimizing control memory accesses.

The approaches just described are general. Specific implementations will often involve a variation or combination of these techniques.

Address Generation

We have looked at the sequencing problem from the point of view of format considerations and general logic requirements. Another viewpoint is to consider the various ways in which the next address can be derived or computed.

Table 20.3 Microinstruction Address Generation Techniques

Explicit	Implicit
Two-field	Mapping
Unconditional branch	Addition
Conditional branch	Residual control

Table 20.3 lists the various address generation techniques. These can be divided into explicit techniques, in which the address is explicitly available in the microinstruction, and implicit techniques, which require additional logic to generate the address.

We have essentially dealt with the explicit techniques. With a two-field approach, two alternative addresses are available with each microinstruction. Using either a single address field or a variable format, various branch instructions can be implemented. A conditional branch instruction depends on the following types of information:

- ALU flags
- Part of the opcode or address mode fields of the machine instruction
- Parts of a selected register, such as the sign bit
- Status bits within the control unit

Several implicit techniques are also commonly used. One of these, mapping, is required with virtually all designs. The opcode portion of a machine instruction must be mapped into a microinstruction address. This occurs only once per instruction cycle.

A common implicit technique is one that involves combining or adding two portions of an address to form the complete address. This approach was taken for the IBM S/360 family [TUCK67] and used on many of the S/370 models. We will use the IBM 3033 as an example.
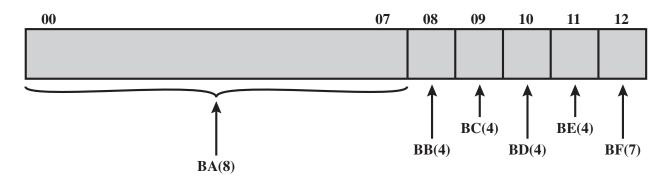
Figure 20.9   IBM 3033 Control Address Register

The control address register on the IBM 3033 is 13 bits long and is illustrated in Figure 20.9. Two parts of the address can be distinguished. The highest-order 8 bits (00–07) normally do not change from one microinstruction cycle to the next. During the execution of a microinstruction, these 8 bits are copied directly from an 8-bit field of the microinstruction (the BA field) into the highest-order 8 bits of the control address register. This defines a block of 32 microinstructions in control memory. The remaining 5 bits of the control address register are set to specify the specific address of the microinstruction to be fetched next. Each of these bits is determined by a 4-bit field (except one is a 7-bit field) in the current microinstruction; the field specifies the condition for setting the corresponding bit. For example, a bit in the control address register might be set to 1 or 0 depending on whether a carry occurred on the last ALU operation.

The final approach listed in Table 20.3 is termed *residual control.* This approach involves the use of a microinstruction address that has previously been saved in temporary storage within the control unit. For example, some microinstruction sets come equipped with a subroutine facility. An internal register or stack of registers is used to hold return addresses. An example of this approach is taken on the LSI-11, which we now examine.

LSI-11 Microinstruction Sequencing

The LSI-11 is a microcomputer version of a PDP-11, with the main components of the system residing on a single board. The LSI-11 is implemented using a microprogrammed control unit [SEBE76].

The LSI-11 makes use of a 22-bit microinstruction and a control memory of 2K 22-bit words. The next microinstruction address is determined in one of five ways:

- **Next sequential address:** In the absence of other instructions, the control unit's control address register is incremented by 1.
- **Opcode mapping:** At the beginning of each instruction cycle, the next microinstruction address is determined by the opcode.
- **Subroutine facility:** Explained presently.
- **Interrupt testing:** Certain microinstructions specify a test for interrupts. If an interrupt has occurred, this determines the next microinstruction address.
- **Branch:** Conditional and unconditional branch microinstructions are used.

A one-level subroutine facility is provided. One bit in every microinstruction is dedicated to this task. When the bit is set, an 11-bit return register is loaded with the updated contents of the control address register. A subsequent microinstruction that specifies a return will cause the control address register to be loaded from the return register.

The return is one form of unconditional branch instruction. Another form of unconditional branch causes the bits of the control address register to be loaded from 11 bits of the microinstruction. The conditional branch instruction makes use of a 4-bit test code within the microinstruction. This code specifies testing of various ALU condition codes to determine the

branch decision. If the condition is not true, the next sequential address is selected. If it is true, the 8 lowest-order bits of the control address register are loaded from 8 bits of the microinstruction. This allows branching within a 256-word page of memory.

As can be seen, the LSI-11 includes a powerful address sequencing facility within the control unit. This allows the microprogrammer considerable flexibility and can ease the microprogramming task. On the other hand, this approach requires more control unit logic than do simpler capabilities.

20.3 MICROINSTRUCTION EXECUTION

The microinstruction cycle is the basic event on a microprogrammed processor. Each cycle is made up of two parts: fetch and execute. The fetch portion is determined by the generation of a microinstruction address, and this was dealt with in the preceding section. This section deals with the execution of a microinstruction.

Recall that the effect of the execution of a microinstruction is to generate control signals. Some of these signals control points internal to the processor. The remaining signals go to the external control bus or other external interface. As an incidental function, the address of the next microinstruction is determined.

The preceding description suggests the organization of a control unit shown in Figure 20.10. This slightly revised version of Figure 20.4 emphasizes the focus of this section. The major modules in this diagram should by now be clear. The sequencing logic module contains the logic to perform the functions discussed in the preceding section. It generates the address of the next microinstruction, using as inputs the instruction register, ALU flags, the control address register (for incrementing), and the control buffer register. The last may provide an actual address, control bits, or both.
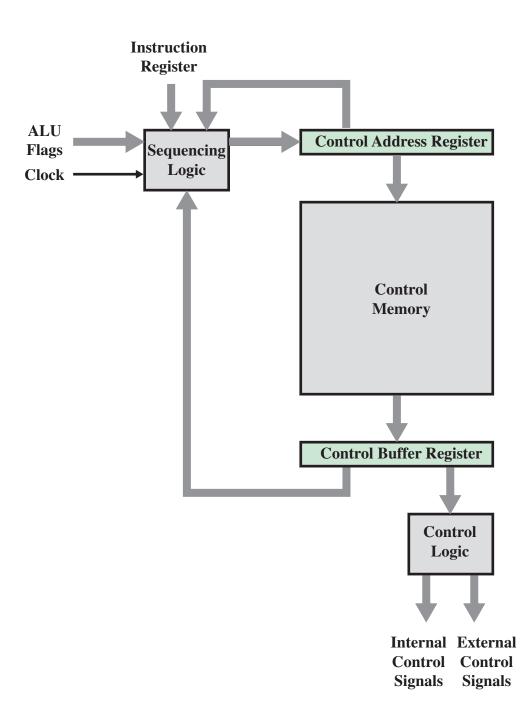
Figure 20.10 Control Unit Organization

The module is driven by a clock that determines the timing of the microinstruction cycle.

The control logic module generates control signals as a function of some of the bits in the microinstruction. It should be clear that the format and

content of the microinstruction will determine the complexity of the control logic module.

A Taxonomy of Microinstructions

Microinstructions can be classified in a variety of ways. Distinctions that are commonly made in the literature include the following:

- Vertical/horizontal
- Packed/unpacked
- Hard/soft microprogramming
- Direct/indirect encoding

All of these bear on the format of the microinstruction. None of these terms has been used in a consistent, precise way in the literature. However, an examination of these pairs of qualities serves to illuminate microinstruction design alternatives. In the following paragraphs, we first look at the key design issue underlying all of these pairs of characteristics, and then we look at the concepts suggested by each pair.

In the original proposal by Wilkes [WILK51], each bit of a microinstruction either directly produced a control signal or directly produced one bit of the next address. We have seen, in the preceding section, that more complex address sequencing schemes, using fewer microinstruction bits, are possible. These schemes require a more complex sequencing logic module. A similar sort of trade-off exists for the portion of the microinstruction concerned with control signals. By encoding control information, and subsequently decoding it to produce control signals, control word bits can be saved.

How can this encoding be done? To answer that, consider that there are a total of K different internal and external control signals to be driven by the

control unit. In Wilkes's scheme, K bits of the microinstruction would be dedicated to this purpose. This allows all of the 2^K possible combinations of control signals to be generated during any instruction cycle. But we can do better than this if we observe that not all of the possible combinations will be used. Examples include the following:

- Two sources cannot be gated to the same destination (e.g., C_2 and C_8 in Figure 20.5).
- A register cannot be both source and destination (e.g., C_5 and C_{12} in Figure 20.5).
- Only one pattern of control signals can be presented to the ALU at a time.
- Only one pattern of control signals can be presented to the external control bus at a time.

So, for a given processor, all possible allowable combinations of control signals could be listed, giving some number $Q < 2^K$ possibilities. These could be encoded with a minimum $\log_2 Q$ bits, with $(\log_2 Q) < K$. This would be the tightest possible form of encoding that preserves all allowable combinations of control signals. In practice, this form of encoding is not used, for two reasons:

- It is as difficult to program as a pure decoded (Wilkes) scheme. This point is discussed further presently.
- It requires a complex and therefore slow control logic module.

Instead, some compromises are made. These are of two kinds:

- More bits than are strictly necessary are used to encode the possible combinations.
- Some combinations that are physically allowable are not possible to encode.

The latter kind of compromise has the effect of reducing the number of bits. The net result, however, is to use more than $\log_2 Q$ bits.

In the next subsection, we will discuss specific encoding techniques. The remainder of this subsection deals with the effects of encoding and the various terms used to describe it.

Based on the preceding, we can see that the control signal portion of the microinstruction format falls on a spectrum. At one extreme, there is one bit for each control signal; at the other extreme, a highly encoded format is used. Table 20.4 shows that other characteristics of a microprogrammed control unit also fall along a spectrum and that these spectra are, by and

Table 20.4 The Microinstruction Spectrum

Characteristics	
Unencoded	Highly encoded
Many bits	Few bits
Detailed view of hardware	Aggregated view of hardware
Difficult to program	Easy to program
Concurrency fully exploited	Concurrency not fully exploited
Little or no control logic	Complex control logic
Fast execution	Slow execution
Optimize performance	Optimize programming
Terminology	
Unpacked	Packed
Horizontal	Vertical
Hard	Soft

large, determined by the degree-of-encoding spectrum.

The second pair of items in the table is rather obvious. The pure Wilkes scheme will require the most bits. It should also be apparent that this extreme presents the most detailed view of the hardware. Every control signal is individually controllable by the microprogrammer. Encoding is done in such a way as to aggregate functions or resources, so that the microprogrammer is viewing the processor at a higher, less detailed level. Furthermore, the encoding is designed to ease the microprogramming burden. Again, it should be clear that the task of understanding and orchestrating the use of all the control signals is a difficult one. As was mentioned, one of the consequences of encoding, typically, is to prevent the use of certain otherwise allowable combinations.

The preceding paragraph discusses microinstruction design from the microprogrammer's point of view. But the degree of encoding also can be viewed from its hardware effects. With a pure unencoded format, little or no decode logic is needed; each bit generates a particular control signal. As more compact and more aggregated encoding schemes are used, more complex decode logic is needed. This, in turn, may affect performance. More time is needed to propagate signals through the gates of the more complex control logic module. Thus, the execution of encoded microinstructions takes longer than the execution of unencoded ones.

Thus, all of the characteristics listed in Table 20.4 fall along a spectrum of design strategies. In general, a design that falls toward the left end of the spectrum is intended to optimize the performance of the control unit. Designs toward the right end are more concerned with optimizing the process of microprogramming. Indeed, microinstruction sets near the right end of the spectrum look very much like machine instruction sets. A good example of this is the LSI-11 design, described later in this section. Typically, when the objective is simply to implement a control unit, the

design will be near the left end of the spectrum. The IBM 3033 design, discussed presently, is in this category. As we shall discuss later, some systems permit a variety of users to construct different microprograms using the same microinstruction facility. In the latter cases, the design is likely to fall near the right end of the spectrum.
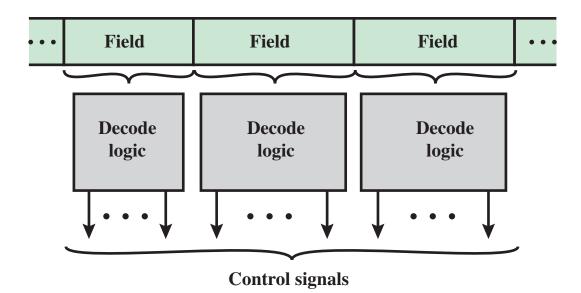
We can now deal with some of the terminology introduced earlier. Table 20.4 indicates how three of these pairs of terms relate to the microinstruction spectrum. In essence, all of these pairs describe the same thing but emphasize different design characteristics.

The degree of packing relates to the degree of identification between a given control task and specific microinstruction bits. As the bits become more *packed,* a given number of bits contains more information. Thus, packing connotes encoding. The terms *horizontal* and *vertical* relate to the relative width of microinstructions. [SIEW82] suggests as a rule of thumb that vertical microinstructions have lengths in the range of 16 to 40 bits and that horizontal microinstructions have lengths in the range of 40 to 100 bits. The terms *hard* and *soft* microprogramming are used to suggest the degree of closeness to the underlying control signals and hardware layout. Hard microprograms are generally fixed and committed to read-only memory. Soft microprograms are more changeable and are suggestive of user microprogramming.

The other pair of terms mentioned at the beginning of this subsection refers to direct versus indirect encoding, a subject to which we now turn.

Microinstruction Encoding

In practice, microprogrammed control units are not designed using a pure unencoded or horizontal microinstruction format. At least some degree of encoding is used to reduce control memory width and to simplify the task of microprogramming.

(a) Direct encoding

(b) Indirect encoding

Figure 20.11 Microinstruction Encoding

The basic technique for encoding is illustrated in Figure 20.11a. The microinstruction is organized as a set of fields. Each field contains a code, which, upon decoding, activates one or more control signals.

Let us consider the implications of this layout. When the microinstruction is executed, every field is decoded and generates control signals. Thus, with N fields, N simultaneous actions are specified. Each action results in the activation of one or more control signals. Generally, but not always, we will want to design the format so that each control signal is activated by no more than one field. Clearly, however, it must be possible for each control signal to be activated by at least one field.

Now consider the individual field. A field consisting of L bits can contain one of 2^L codes, each of which can be encoded to a different control signal pattern. Because only one code can appear in a field at a time, the codes are mutually exclusive, and, therefore, the actions they cause are mutually exclusive.

The design of an encoded microinstruction format can now be stated in simple terms:

- Organize the format into independent fields. That is, each field depicts a set of actions (pattern of control signals) such that actions from different fields can occur simultaneously.
- Define each field such that the alternative actions that can be specified by the field are mutually exclusive. That is, only one of the actions specified for a given field could occur at a time.

Two approaches can be taken to organizing the encoded microinstruction into fields: functional and resource. The *functional encoding* method identifies functions within the machine and designates fields by function type. For example, if various sources can be used for transferring

data to the accumulator, one field can be designated for this purpose, with each code specifying a different source. *Resource encoding* views the machine as consisting of a set of independent resources and devotes one field to each (e.g., I/O, memory, ALU).

Another aspect of encoding is whether it is direct or indirect (Figure 20.11b). With indirect encoding, one field is used to determine the interpretation of another field. For example, consider an ALU that is capable of performing eight different arithmetic operations and eight different shift operations. A 1-bit field could be used to indicate whether a shift or arithmetic operation is to be used; a 3-bit field would indicate the operation. This technique generally implies two levels of decoding, increasing propagation delays.

Figure 20.12 is a simple example of these concepts. Assume a processor with a single accumulator and several internal registers, such as a program counter and a temporary register for ALU input. Figure 20.12a shows a highly vertical format. The first 3 bits indicate the type of operation, the next 3 encode the operation, and the final 2 select an internal register. Figure 20.12b is a more horizontal approach, although encoding is still used. In this case, different functions appear in different fields.

Simple register transfers

| 0 | 0 | 0 | 0 | 0 | 0 | | MDR ← Register

| 0 | 0 | 0 | 0 | 0 | 1 | | Register ← MDR

| 0 | 0 | 0 | 0 | 1 | 0 | | MAR ← Register

Register select

Memory operations

| 0 | 0 | 1 | 0 | 0 | 0 | | Read

| 0 | 0 | 1 | 0 | 0 | 1 | | Write

Special sequencing operations

| 0 | 1 | 0 | 0 | 0 | 0 | | CSAR ← Decoded MDR

| 0 | 1 | 0 | 0 | 0 | 1 | | CSAR ← Constant (in next byte)

| 0 | 1 | 0 | 0 | 1 | 0 | | Skip

ALU operations

| 0 | 1 | 1 | 0 | 0 | 0 | | ACC ← ACC + Register

| 0 | 1 | 1 | 0 | 0 | 1 | | ACC ← ACC – Register

| 0 | 1 | 1 | 0 | 1 | 0 | | ACC ← Register

| 0 | 1 | 1 | 0 | 1 | 1 | | Register ← ACC

| 0 | 1 | 1 | 1 | 0 | 0 | | ACC ← Register + 1

Register select

(a) Vertical microinstruction format

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18

Field   1   2   3   4   5   6

Field definition

1 - register transfer   4 - ALU operation
2 - memory operation   5 - register selection
3 - sequencing operation 6 - Constant
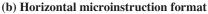
(b) Horizontal microinstruction format

Figure 20 .12 Alternative Microinstruction Formats for a Simple Machine

LSI-11 Microinstruction Execution

The LSI-11 [SEBE76] is a good example of a vertical microinstruction approach. We look first at the organization of the control unit, then at the microinstruction format.
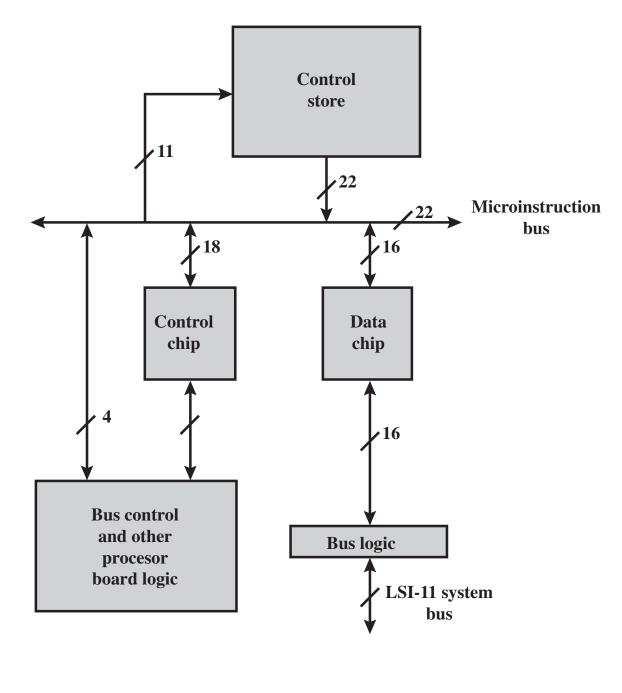
LSI-11 CONTROL UNIT ORGANIZATION

The LSI-11 is the first member of the PDP-11 family that was offered as a single-board processor. The board contains three LSI chips, an internal bus known as the *microinstruction bus* (MIB), and some additional interfacing logic.

Figure 20.13 depicts, in simplified form, the organization of the LSI-11 processor. The three chips are the data, control, and control store chips. The data chip contains an 8-bit ALU, twenty-six 8-bit registers, and storage for several condition codes. Sixteen of the registers are used to implement the eight 16-bit general-purpose registers of the PDP-11. Others include a program status word, memory address register (MAR), and memory buffer register. Because the ALU deals with only 8 bits at a time, two passes through the ALU are required to implement a 16-bit PDP-11 arithmetic operation. This is controlled by the microprogram.

The control store chip or chips contain the 22-bit-wide control memory. The control chip contains the logic for sequencing and executing microinstructions. It contains the control address register, the control data register, and a copy of the machine instruction register.

The MIB ties all the components together. During microinstruction fetch, the control chip generates an 11-bit address onto the MIB. Control store is accessed, producing a 22-bit microinstruction, which is placed on the MIB. The low-order 16 bits go to the data chip, while the low-order 18 bits go to the control chip. The high-order 4 bits control special processor board functions.

Control store

11

22

22 **Microinstruction bus**

18

16

Control chip

Data chip

4

16

Bus control and other procesor board logic

Bus logic

16

LSI-11 system bus

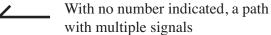With no number indicated, a path with multiple signals

Figure 20.13 Simplified Block Diagram of the LSI-11 Processor

Figure 20.14 provides a still simplified but more detailed look at the LSI-11 control unit: the figure ignores individual chip boundaries. The address

sequencing scheme described in Section 20.2 is implemented in two modules. Overall sequence control is provided by the microprogram sequence control module, which is capable of incrementing the microinstruction address register and performing unconditional branches. The other forms of address calculation are carried out by a separate translation array. This is a combinatorial circuit that generates an address based on the microinstruction, the machine instruction, the microinstruction program counter, and an interrupt register.

The translation array comes into play on the following occasions:

- The opcode is used to determine the start of a microroutine.
- At appropriate times, address mode bits of the microinstruction are tested to perform appropriate addressing.
- Interrupt conditions are periodically tested.
- Conditional branch microinstructions are evaluated.

LSI-11 MICROINSTRUCTION FORMAT

The LSI-11 uses an extremely vertical microinstruction format, which is only 22 bits wide. The microinstruction set strongly resembles the PDP-11 machine instruction set that it implements. This design was intended to optimize the performance of the control unit within the constraint of a vertical, easily programmed design. Table 20.5 lists some of the LSI-11 microinstructions.

Figure 20.15 shows the 22-bit LSI-11 microinstruction format. The high-order 4 bits control special functions on the processor board. The translate bit enables the translation array to check for pending interrupts. The load return register bit is used at the end of a microroutine to cause the next microinstruction address to be loaded from the return register.

Figure 20.14 Organization of the LSI-11 Control Unit

4	1	1	16
Special functions			Encoded micro-operations

↑ ↑
Load return register
Translate

(a) Format of the full LSI-11 microinstruction

5	11
Opcode	Jump address

Unconditional jump microinstruction format

4	4	8
Opcode	Test code	Jump address

Conditional jump microinstruction format

4	8	4
Opcode	Literal value	A register

Literal microinstruction format

8	4	4
Opcode	B register	A register

Register jump microinstruction format

(b) Format of the encoded part of the LSI-11 microinstruction

Figure 20.15   LSI-11 Microinstruction Format

Table 20.5 Some LSI-11 Microinstructions

Arithmetic Operations
Add word (byte, literal)
Test word (byte, literal)
Increment word (byte) by 1
Increment word (byte) by 2
Negate word (byte)
Conditionally increment (decrement) byte
Conditionally add word (byte)
Add word (byte) with carry
Conditionally add digits
Subtract word (byte)
Compare word (byte, literal)
Subtract word (byte) with carry
Decrement word (byte) by 1

Logical Operations
AND word (byte, literal)
Test word (byte)
OR word (byte)
Exclusive-OR word (byte)
Bit clear word (byte)
Shift word (byte) right (left) with (without) carry
Complement word (byte)

General Operations
MOV word (byte)
Jump
Return
Conditional jump
Set (reset) flags
Load G low
Conditionally MOV word (byte)

Input/Output Operations
Input word (byte)
Input status word (byte)
Read
Write
Read (write) and increment word (byte) by 1
Read (write) and increment word (byte) by 2
Read (write) acknowledge
Output word (byte, status)

The remaining 16 bits are used for highly encoded micro-operations. The format is much like a machine instruction, with a variable-length opcode and one or more operands.

Table 20.6 IBM 3033 Microinstruction Control Fields

ALU Control Fields

AA(3)	Load A register from one of data registers
AB(3)	Load B register from one of data registers
AC(3)	Load C register from one of data registers
AD(3)	Load D register from one of data registers
AE(4)	Route specified A bits to ALU
AF(4)	Route specified B bits to ALU
AG(5)	Specifies ALU arithmetic operation on A input
AH(4)	Specifies ALU arithmetic operation on B input
AJ(1)	Specifies D or B input to ALU on B side
AK(4)	Route arithmetic output to shifter
CA(3)	Load F register
CB(1)	Activate shifter
CC(5)	Specifies logical and carry functions
CE(7)	Specifies shift amount

Sequencing and Branching Fields

AL(1)	End operation and perform branch
BA(8)	Set high-order bits (00–07) of control address register
BB(4)	Specifies condition for setting bit 8 of control address register
BC(4)	Specifies condition for setting bit 9 of control address register
BD(4)	Specifies condition for setting bit 10 of control address register
BE(4)	Specifies condition for setting bit 11 of control address register
BF(7)	Specifies condition for setting bit 12 of control address register

0										35
P	AA	AB	AC	AD	AE	AF	AG	AH	AJ	AK	AL

A, B, C, D registers — Arithmetic — Shift

36							71
P	BA	BB	BC	BD	BE	BF	BH

Next address — Storage address

72									107
P	BH	CA	CB	CC	CD	CE	CF	CG	CH

Storage address — Shift control — Local storage — Miscellaneous controls

108					125
P	DA	DB	DC	DD	DE

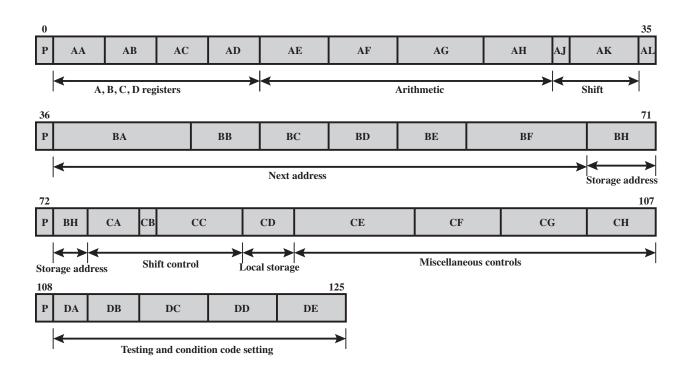Testing and condition code setting

Figure 20.16 IBM 3033 Microinstruction Format

IBM 3033 Microinstruction Execution

The standard IBM 3033 control memory consists of 4K words. The first half of these (0000–07FF) contain 108-bit microinstructions, while the remainder (0800–0FFF) are used to store 126-bit microinstructions. The format is depicted in Figure 20.16. Although this is a rather horizontal format, encoding is still extensively used. The key fields of that format are summarized in Table 20.6.

The ALU operates on inputs from four dedicated, non-user-visible registers, A, B, C, and D. The microinstruction format contains fields for loading these registers from user-visible registers, performing an ALU function, and specifying a user-visible register for storing the result. There are also fields for loading and storing data between registers and memory.

The sequencing mechanism for the IBM 3033 was discussed in Section 20.2.

20.4 TI 8800

The Texas Instruments 8800 Software Development Board (SDB) is a microprogrammable 32-bit computer card. The system has a writable control store, implemented in RAM rather than ROM. Such a system does not achieve the speed or density of a microprogrammed system with a ROM control store. However, it is useful for developing prototypes and for educational purposes.

The 8800 SDB consists of the following components (Figure 20.17):

- Microcode memory
- Microsequencer
- 32-bit ALU
- Floating-point and integer processor
- Local data memory

Two buses link the internal components of the system. The DA bus provides data from the microinstruction data field to the ALU, the floating-point processor, or the microsequencer. In the latter case, the data consists of an address to be used for a branch instruction. The bus can also be used for the ALU or microsequencer to provide data to other components. The System Y bus connects the ALU and floating-point processor to local memory and to external modules via the PC interface.

The board fits into an IBM PC-compatible host computer. The host computer provides a suitable platform for microcode assembly and debug.
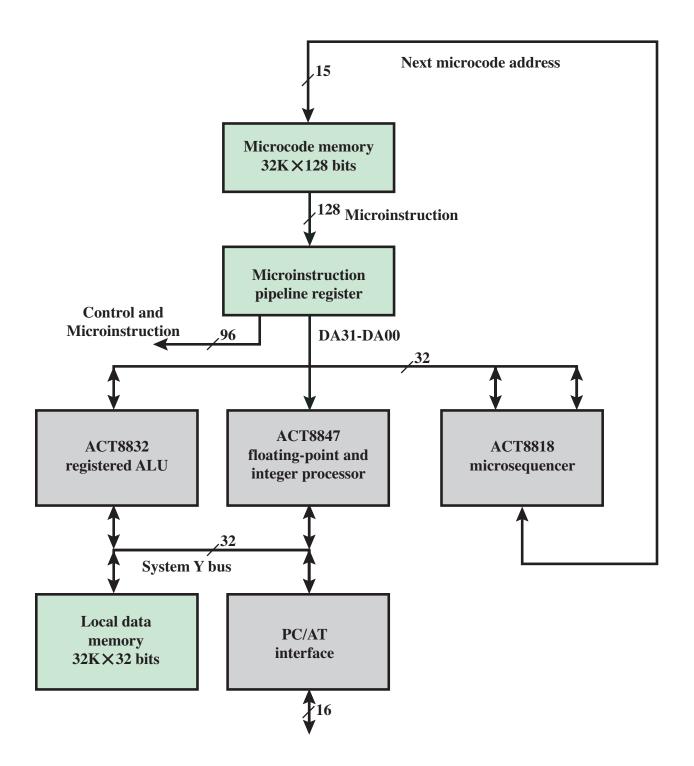
Figure 20.17   TI 8800 Block Diagram

Table 20.7 TI 8800 Microinstruction Format

Field Number	Number of Bits	Description
		Control of Board
1	5	Select condition code input
2	1	Enable/disable external I/O request signal
3	2	Enable/disable local data memory read/write operations
4	1	Load status/do no load status
5	2	Determine unit driving Y bus
6	2	Determine unit driving DA bus
		8847 Floating Point and Integer Processing Chip
7	1	C register control: clock, do not clock
8	1	Select most significant or least significant bits for Y bus
9	1	C register data source: ALU, multiplexer
10	4	Select IEEE or FAST mode for ALU and MUL
11	8	Select sources for data operands: RA registers, RB registers, P register, 5 register, C register
12	1	RB register control: clock, do not clock
13	1	RA register control: clock, do not clock
14	2	Data source confirmation
15	2	Enable/disable pipeline registers
16	11	8847 ALU function
		8832 Registered ALU
17	2	Write enable/disable data output to selected register: most significant half, least significant half
18	2	Select register file data source: DA bus, DB bus, ALU Y MUX output, system Y bus
19	3	Shift instruction modifier
20	1	Carry in: force, do not force
21	2	Set ALU configuration mode: 32, 16, or 8 bits
22	2	Select input to 5 multiplexer: register file, DB bus, MQ register
23	1	Select input to R multiplexer: register file, DA bus
24	6	Select register in file C for write
25	6	Select register in file B for read
26	6	Select register in file A for write
27	8	ALU function
		8818 Microsequencer
28	12	Control input signals to the 8818
		WCS Data Field
29	16	Most significant bits of writable control store data field
30	16	Least significant bits of writable control store data field

Microinstruction Format

The microinstruction format for the 8800 consists of 128 bits broken down into 30 functional fields, as indicated in Table 20.7. Each field consists of one or more bits, and the fields are grouped into five major categories:

- Control of board
- 8847 floating-point and integer processor chip
- 8832 registered ALU
- 8818 microsequencer
- WCS data field

As indicated in Figure 20.17, the 32 bits of the WCS data field are fed into the DA bus to be provided as data to the ALU, floating-point processor, or microsequencer. The other 96 bits (fields 1–27) of the microinstruction are control signals that are fed directly to the appropriate module. For simplicity, these other connections are not shown in Figure 20.17.

The first six fields deal with operations that pertain to the control of the board, rather than controlling an individual component. Control operations include the following:

- Selecting condition codes for sequencer control. The first bit of field 1 indicates whether the condition flag is to be set to 1 or 0, and the remaining 4 bits indicate which flag is to be set.
- Sending an I/O request to the PC/AT.
- Enabling local data memory read/write operations.
- Determining the unit driving the system Y bus. One of the four devices attached to the bus (Figure 20.17) is selected.

The last 32 bits are the data field, which contain information specific to a particular microinstruction.

The remaining fields of the microinstruction are best discussed in the context of the device that they control. In the remainder of this section, we discuss the microsequencer and the registered ALU. The floating-point unit introduces no new concepts and is skipped.

Microsequencer

The principal function of the 8818 microsequencer is to generate the next microinstruction address for the microprogram. This 15-bit address is provided to the microcode memory (Figure 20.17).

The next address can be selected from one of five sources:

1. The microprogram counter (MPC) register, used for repeat (reuse same address) and continue (increment address by 1) instructions.
2. The stack, which supports microprogram subroutine calls as well as iterative loops and returns from interrupts.
3. The DRA and DRB ports, which provide two additional paths from external hardware by which microprogram addresses can be generated. These two ports are connected to the most significant and least significant 16 bits of the DA bus, respectively. This allows the microsequencer to obtain the next instruction address from the WCS data field of the current microinstruction or from a result calculated by the ALU.
4. Register counters RCA and RCB, which can be used for additional address storage.
5. An external input onto the bidirectional Y port to support external interrupts.

Figure 20.18 is a logical block diagram of the 8818. The device consists of the following principal functional groups:

- A 16-bit microprogram counter (MPC) consisting of a register and an incrementer
- Two register counters, RCA and RCB, for counting loops and iterations, storing branch addresses, or driving external devices
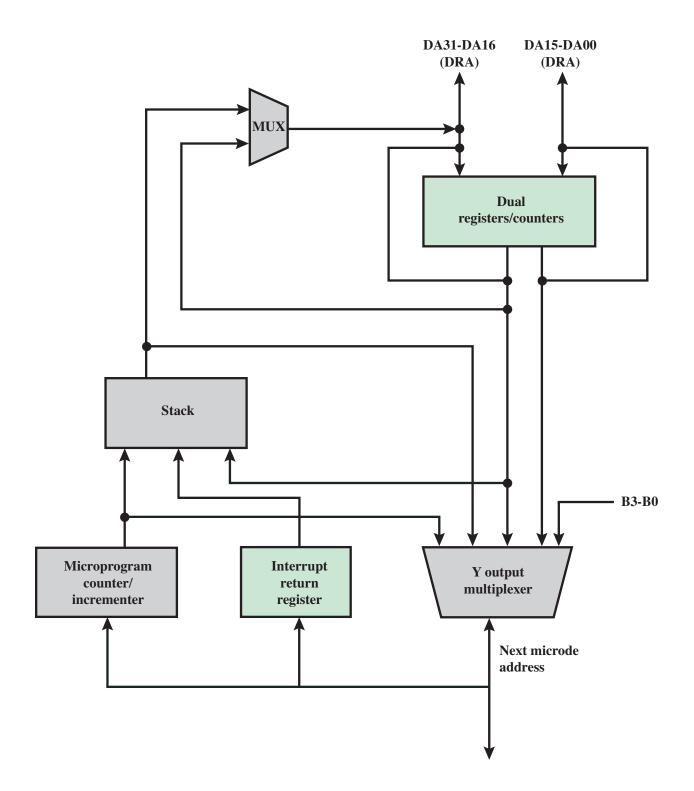
Figure 20.18 TI 8818 Microsequencer

- A 65-word by 16-bit stack, which allows microprogram subroutine calls and interrupts
- An interrupt return register and Y output enable for interrupt processing at the microinstruction level
- A Y output multiplexer by which the next address can be selected from MPC, RCA, RCB, external buses DRA and DRB, or the stack

REGISTERS/COUNTERS

The registers RCA and RCB may be loaded from the DA bus, either from the current microinstruction or from the output of the ALU. The values may be used as counters to control the flow of execution and may be automatically decremented when accessed. The values may also be used as microinstruction addresses to be supplied to the Y output multiplexer. Independent control of both registers during a single microinstruction cycle is supported with the exception of simultaneous decrement of both registers.

STACK

The stack allows multiple levels of nested calls or interrupts, and it can be used to support branching and looping. Keep in mind that these operations refer to the control unit, not the overall processor, and that the addresses involved are those of microinstructions in the control memory.

Six stack operations are possible:

1. Clear, which sets the stack pointer to zero, emptying the stack
2. Pop, which decrements the stack pointer
3. Push, which puts the contents of the MPC, interrupt return register, or DRA bus onto the stack and increments the stack pointer
4. Read, which makes the address indicated by the read pointer available at the Y output multiplexer

5. Hold, which causes the address of the stack pointer to remain unchanged

6. Load stack pointer, which inputs the seven least significant bits of DRA to the stack pointer

CONTROL OF MICROSEQUENCER

The microsequencer is controlled primarily by the 12-bit field of the current microinstruction, field 28 (Table 20.7). This field consists of the following subfields:

- **OSEL (1 bit):** Output select. Determines which value will be placed on the output of the multiplexer that feeds into the DRA bus (upper-left-hand corner of Figure 20.18). The output is selected to come from either the stack or from register RCA. DRA then serves as input to either the Y output multiplexer or to register RCA.
- **SELDR (1 bit):** Select DR bus. If set to 1, this bit selects the external DA bus as input to the DRA/DRB buses. If set to 0, selects the output of the DRA multiplexer to the DRA bus (controlled by OSEL) and the contents of RCB to the DRB bus.
- **ZEROIN (1 bit):** Used to indicate a conditional branch. The behavior of the microsequencer will then depend on the condition code selected in field 1 (Table 20.7).
- **RC2–RC0 (3 bits):** Register controls. These bits determine the change in the contents of registers RCA and RCB. Each register can either remain the same, decrement, or load from the DRA/DRB buses.
- **S2–S0 (3 bits):** Stack controls. These bits determine which stack operation is to be performed.

Table 20.8 TI 8818 Microsequencer Microinstruction Bits (Field 28)

Mnemonic	Value	Description
RST8818	000000000110	Reset Instruction
BRA88181	011000111000	Branch to DRA Instruction
BRA88180	010000111110	Branch to DRA Instruction
INC88181	000000111110	Continue Instruction
INC88180	001000001000	Continue Instruction
CAL88181	010000110000	Jump to Subroutine at Address Specified by DRA
CAL88180	010000101110	Jump to Subroutine at Address Specified by DRA
RET8818	000000011010	Return from Subroutine
PUSH8818	000000110111	Push Interrupt Return Address onto Stack
POP8818	100000010000	Return from Interrupt
LOADDRA	000010111110	Load DRA Counter from DA Bus
LOADDRB	000110111110	Load DRB Counter from DA Bus
LOADDRAB	000110111100	Load DRA/DRB
DECRDRA	010001111100	Decrement DRA Counter and Branch If Not Zero
DECRDRB	010101111100	Decrement DRB Counter and Branch If Not Zero

- **MUX2−MUX0:** Output controls. These bits, together with the condition code if used, control the Y output multiplexer and therefore the next microinstruction address. The multiplexer can select its output from the stack, DRA, DRB, or MPC.

These bits can be set individually by the programmer. However, this is typically not done. Rather, the programmer uses mnemonics that equate to the bit patterns that would normally be required. Table 20.8 lists the 15 mnemonics for field 28. A microcode assembler converts these into the appropriate bit patterns.

As an example, the instruction INC88181 is used to cause the next microinstruction in sequence to be selected, if the currently selected condition code is 1. From Table 20.8, we have

$$INC88181 = 000000111110$$

which decodes directly into

- **OSEL = 0:** Selects RCA as output from DRA output MUX; in this case the selection is irrelevant.
- **SELDR = 0:** As defined previously; again, this is irrelevant for this instruction.
- **ZEROIN = 0:** Combined with the value for MUX, indicates no branch should be taken.
- **R = 000:** Retain current value of RA and RC.
- **S = 111:** Retain current state of stack.
- **MUX = 110:** Choose MPC when condition code = 1, DRA when condition code = 0.

Registered ALU

The 8832 is a 32-bit ALU with 64 registers that can be configured to operate as four 8-bit ALUs, two 16-bit ALUs, or a single 32-bit ALU.

The 8832 is controlled by the 39 bits that make up fields 17 through 27 of the microinstruction (Table 20.7); these are supplied to the ALU as control signals. In addition, as indicated in Figure 20.17, the 8832 has external connections to the 32-bit DA bus and the 32-bit system Y bus. Inputs from the DA can be provided simultaneously as input data to the 64-word register file and to the ALU logic module. Input from the system Y bus is provided to the ALU logic module. Results of the ALU and shift operations are output to

the DA bus or the system Y bus. Results can also be fed back to the internal register file.

   Three 6-bit address ports allow a two-operand fetch and an operand write to be performed within the register file simultaneously. An MQ shifter and MQ register can also be configured to function independently to implement double-precision 8-bit, 16-bit, and 32-bit shift operations.

   Fields 17 through 26 of each microinstruction control the way in which data flows within the 8832 and between the 8832 and the external environment. The fields are as follows:

17. **Write Enable.** These two bits specify write 32 bits, or 16 most significant bits, or 16 least significant bits, or do not write into register file. The destination register is defined by field 24.

18. **Select Register File Data Source.** If a write is to occur to the register file, these two bits specify the source: DA bus, DB bus, ALU output, or system Y bus.

19. **Shift Instruction Modifier.** Specifies options concerning supplying end fill bits and reading bits that are shifted during shift instructions.

20. **Carry In.** This bit indicates whether a bit is carried into the ALU for this operation.

21. **ALU Configuration Mode.** The 8832 can be configured to operate as a single 32-bit ALU, two 16-bit ALUs, or four 8-bit ALUs.

22. **S Input.** The ALU logic module inputs are provided by two internal multiplexers referred to as the S and R multiplexers. This field selects the input to be provided by the S multiplexer: register file, DB bus, or MQ register. The source register is defined by field 25.

23. **R Input.** Selects input to be provided by the R multiplexer: register file or DA bus.

24. **Destination Register.** Address of register in register file to be used for the destination operand.
25. **Source Register.** Address of register in register file to be used for the source operand, provided by the S multiplexer.
26. **Source Register.** Address of register in register file to be used for the source operand, provided by the R multiplexer.

Finally, field 27 is an 8-bit opcode that specifies the arithmetic or logical function to be performed by the ALU. Table 20.9 lists the different operations that can be performed.

As an example of the coding used to specify fields 17 through 27, consider the instruction to add the contents of register 1 to register 2 and place the result in register 3. The symbolic instruction is

CONT11 [17], WELH, SELRYFYMX, [24], R3, R2, R1, PASS+ADD

The assembler will translate this into the appropriate bit pattern. The individual components of the instruction can be described as follows:

- CONT11 is the basic NOP instruction.
- Field [17] is changed to WELH (write enable, low and high), so that a 32-bit register is written into
- Field [18] is changed to SELRFYMX to select the feedback from the ALU Y MUX output.
- Field [24] is changed to designate register R3 for the destination register.
- Field [25] is changed to designate register R2 for one of the source registers.

Table 20.9 TI 8832 Registered ALU Instruction Field (Field 27)

Group 1		Function
ADD	H#01	R + S + Cn
SUBR	H#02	(NOT R) + S + Cn
SUBS	H#03	R = (NOT S) + Cn
INSC	H#04	S + Cn
INCNS	H#05	(NOT S) + Cn
INCR	H#06	R + Cn
INCNR	H#07	(NOT R) + Cn
XOR	H#09	R XOR S
AND	H#0A	R AND S
OR	H#0B	R OR S
NAND	H#0C	R NAND S
NOR	H#0D	R NOR S
ANDNR	H#0E	(NOT R) AND S
Group 2		**Function**
SRA	H#00	Arithmetic right single precision shift
SRAD	H#10	Arithmetic right double precision shift
SRL	H#20	Logical right single precision shift
SRLD	H#30	Logical right double precision shift
SLA	H#40	Arithmetic left single precision shift
SLAD	H#50	Arithmetic left double precision shift
SLC	H#60	Circular left single precision shift
SLCD	H#70	Circular left double precision shift
SRC	H#80	Circular right single precision shift
SRCD	H#90	Circular right double precision shift
MQSRA	H#A0	Arithmetic right shift MQ register
MQSRL	H#B0	Logical right shift MQ register
MQSLL	H#C0	Logical left shift MQ register
MQSLC	H#D0	Circular left shift MQ register
LOADMQ	H#E0	Load MQ register
PASS	H#F0	Pass ALU to Y (no shift operation)
Group 3		**Function**
SET1	H#08	Set bit 1
Set0	H#18	Set bit 0
TB1	H#28	Test bit 1
TB0	H#38	Test bit 0
ABS	H#48	Absolute value
SMTC	H#58	Sign magnitude/twos-complement
ADDI	H#68	Add immediate
SUBI	H#78	Subtract immediate
BADD	H#88	Byte add R to S
BSUBS	H#98	Byte subtract S from R
BSUBR	H#A8	Byte subtract R from S
BINCS	H#B8	Byte increment S
BINCNS	H#C8	Byte increment negative S
BXOR	H#D8	Byte XOR R and S
BAND	H#E8	Byte AND R and S
BOR	H#F8	Byte OR R and S
Group 4		**Function**
CRC	H#00	Cyclic redundancy character accum.
SEL	H#10	Select S or R

SNORM	H#20	Single length normalize
DNORM	H#30	Double length normalize
DIVRF	H#40	Divide remainder fix
SDIVQF	H#50	Signed divide quotient fix
SMULI	H#60	Signed multiply iterate
SMULT	H#70	Signed multiply terminate
SDIVIN	H#80	Signed divide initialize
SDIVIS	H#90	Signed divide start
SDIVI	H#A0	Signed divide iterate
UDIVIS	H#B0	Unsigned divide start
UDIVI	H#C0	Unsigned divide iterate
UMULI	H#D0	Unsigned multiply iterate
SDIVIT	H#E0	Signed divide terminate
UDIVIT	H#F0	Unsigned divide terminate
Group 5		**Function**
LOADFF	H#0F	Load divide/BCD flip-flops
CLR	H#1F	Clear
DUMPFF	H#5F	Output divide/BCD flip-flops
BCDBIN	H#7F	BCD to binary
EX3BC	H#8F	Excess -3 byte correction
EX3C	H#9F	Excess -3 word correction
SDIVO	H#AF	Signed divide overflow test
BINEX3	H#DF	Binary to excess –3
NOP32	H#FF	No operation

• Field [26] is changed to designate register R1 for one of the source registers.

• Field [27] is changed to specify an ALU operation of ADD. The ALU shifter instruction is PASS; therefore, the ALU output is not shifted by the shifter.

Several points can be made about the symbolic notation. It is not necessary to specify the field number for consecutive fields. That is,

CONT11 [17], WELH, [18], SELRFYMX

can be written as

CONT11 [17], WELH, SELRFYMX

because SELRFYMX is in field 18.

ALU instructions from Group 1 of Table 20.9 must always be used in conjunction with Group 2. ALU instructions from Groups 3–5 must not be used with Group 2.

20.6 RECOMMENDED READING

There are a number of books devoted to microprogramming. Perhaps the most comprehensive is [LYNC93]. [SEGE91] presents the fundamentals of microcoding and the design of microcoded systems by means of a step-by-step design of a simple 16-bit processor. [CART96] also presents the basic concepts using a sample machine. [PARK89] and [TI90] provide a detailed description of the TI 8800 Software Development Board.

[VASS03] discuss the evolution of microcode use in computer design and its current status.

CART96 Carter, J. *Microprocesser Architecture and Microprogramming.* Upper Saddle River, NJ: Prentice Hall, 1996.

LYNC93 Lynch, M. *Microprogrammed State Machine Design.* Boca Raton, FL: CRC Press, 1993.

PARK89 Parker, A., and Hamblen, J. *An Introduction to Microprogramming with Exercises Designed for the Texas Instruments SN74ACT8800 Software Development Board.* Dallas, TX: Texas Instruments, 1989.

SEGE91 Segee, B., and Field, J. *Microprogramming and Computer Architecture.* New York: Wiley, 1991.

TI90   Texas Instruments Inc. *SN74ACT880 Family Data Manual.* SCSS006C, 1990.

VASS03 Vassiliadis, S.; Wong, S.; and Cotofana, S. "Microcode Processing: Positioning and Directions." *IEEE Micro*, July-August 2003.

Additional References

HILL64 Hill, R. "Stored Logic Programming and Applications." *Datamation,* February 1964.

SEBE76 Sebern, M. "A Minicomputer-compatible Microcomputer System: The DEC LSI-11." *Proceedings of the IEEE,* June 1976.

SIEW82 Siewiorek, D.; Bell, C.; and Newell, A. *Computer Structures: Principles and Examples.* New York: McGraw-Hill, 1982.

TUCK67 Tucker, S. "Microprogram Control for System/360." *IBM Systems Journal,* No. 4, 1967.

WILK51 Wilkes, M. "The Best Way to Design an Automatic Calculating Machine." *Proceedings, Manchester University Computer Inaugural Conference,* July 1951.

WILK53 Wilkes, M., and Stringer, J. "Microprogramming and the Design of the Control Circuits in an Electronic Digital Computer." *Proceedings of the Cambridge Philosophical Society,* April 1953. Reprinted in [SIEW82].

20.7 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

control memory	microinstruction	microprogrammed
control word	encoding	control unit
firmware	microinstruction	microprogramming
hard microprogramming	execution	language
horizontal	microinstruction	soft microprogramming
microinstruction	sequencing	unpacked
	microinstructions	microinstruction
	microprogram	vertical microinstruction

Review Questions

20.1 What is the difference between a hardwired implementation and a microprogrammed implementation of a control unit?

20.2 How is a horizontal microinstruction interpreted?

20.3 What is the purpose of a control memory?

20.4 What is a typical sequence in the execution of a horizontal microinstruction?

20.5 What is the difference between horizontal and vertical microinstructions?

20.6 What are the basic tasks performed by a microprogrammed control unit?

20.7 What is the difference between packed and unpacked microinstructions?

20.8 What is the difference between hard and soft microprogramming?

20.9 What is the difference between functional and resource encoding?

20.10 List some common applications of microprogramming.

Problems

20.1 Describe the implementation of the multiply instruction in the hypothetical machine designed by Wilkes. Use narrative and a flowchart.

20.2 Assume a microinstruction set that includes a microinstruction with the following symbolic form:

$$\text{IF } (AC_0 = 1) \text{ THEN CAR} \leftarrow (C_{0\text{-}6}) \text{ ELSE CAR} \leftarrow (\text{CAR}) + 1$$

where AC_0 is the sign bit of the accumulator and C_{0-6} are the first seven bits of the microinstruction. Using this microinstruction, write a microprogram that implements a Branch Register Minus (BRM) machine instruction, which branches if the AC is negative. Assume that bits C_1 through C_n of the microinstruction specify a parallel set of micro-operations. Express the program symbolically.

20.3 A simple processor has four major phases to its instruction cycle: fetch, indirect, execute, and interrupt. Two 1-bit flags designate the current phase in a hardwired implementation.
a. Why are these flags needed?
b. Why are they not needed in a microprogrammed control unit?

20.4 Consider the control unit of Figure 20.7. Assume that the control memory is 24 bits wide. The control portion of the microinstruction format is divided into two fields. A micro-operation field of 13 bits specifies the micro-operations to be performed. An address selection field specifies a condition, based on the flags, that will cause a microinstruction branch. There are eight flags.
   a. How many bits are in the address selection field?
   b. How many bits are in the address field?
   c. What is the size of the control memory?

20.5 How can unconditional branching be done under the circumstances of the previous problem? How can branching be avoided; that is, describe a microinstruction that does not specify any branch, conditional or unconditional.

20.6 We wish to provide 8 control words for each machine instruction routine. Machine instruction opcodes have 5 bits, and control memory has 1024 words. Suggest a mapping from the instruction register to the control address register.

20.7 An encoded microinstruction format is to be used. Show how a 9-bit micro-operation field can be divided into subfields to specify 46 different actions.

20.8 A processor has 16 registers, an ALU with 16 logic and 16 arithmetic functions, and a shifter with 8 operations, all connected by an internal processor bus. Design a microinstruction format to specify the various micro-operations for the processor.