5.7 Microprogramming: Simplifying Control Design

For the control of our simple MIPS subset, a graphical representation of the finite state machine, as in Figure 5.40 on page 345, is certainly adequate. We can draw such a diagram on a single page and translate it into equations (see Appendix C) without generating too many errors. Consider instead an implementation of the full MIPS-32 instruction set, which contains over 100 instructions (see Appendix A). In one implementation, instructions take from 1 clock cycle to over 20 clock cycles. Clearly, the control function will be much more complex. For such a design we would likely use a hardware design language, such as Verilog (which is explored in more detail in Section 5.8), and have the finite state control synthesized; the next section shows how this can be done.

Consider, however, an instruction set with several hundred instructions of widely varying classes, such as the IA-32 architecture. The control unit could easily require thousands of states with hundreds of different sequences. In such a case, specifying the control unit with a graphical representation will be impossible. Even using a finite state abstraction where the next state must be explicitly specified is likely to be cumbersome.

Can we use some of the ideas from programming to help create a method of specifying the control that will make it easier to understand as well as to design? Suppose we think of the set of control signals that must be asserted in a state as an instruction to be executed by the datapath. To avoid confusing the instructions of the MIPS instruction set with these low-level control instructions, the latter are called *microinstructions*. Each microinstruction defines the set of datapath control signals that must be asserted in a given state. Executing a microinstruction has the effect of asserting the control signals specified by the microinstruction.

In addition to defining which control signals must be asserted, we must also specify the sequencing—what microinstruction should be executed next? In the finite state machine shown in Figure 5.38 on page 339, the next state is determined in one of two different ways. Sometimes a single next state follows the current state unconditionally. For example, state 1 always follows state 0, and the only way to reach state 1 is via state 0. In other cases, the choice of the next state depends on the input. This is true in state 1, which has four different successor states.

When we write programs, we also have an analogous situation. Sometimes a group of instructions should be executed sequentially, and sometimes we need to branch. In programming, the default is sequential execution, while branching must be indicated explicitly. In describing the control as a program, we also assume that microinstructions written sequentially are executed in sequence, while branching must be indicated explicitly. The default sequencing mechanism can still be implemented using a structure like the one in Figure 5.37 on page 338; however, it is often more efficient to implement the default sequential state using a counter. We will see how such an implementation looks at the end of this section.

Designing the control as a program that implements the machine instructions in terms of simpler microinstructions is called *microprogramming*. The key idea is to represent the asserted values on the control lines symbolically, so that the microprogram is a representation of the microinstructions, just as assembly language is a representation of the machine instructions. In choosing a syntax for an assembly language, we usually represent the machine instructions as a series of fields (opcode, registers, and offset or immediate field); likewise, we will represent a microinstruction syntactically as a sequence of fields whose functions are related.

One other important idea from software is often incorporated into microprogrammed control: the concept of subroutines. Consider why this might make sense: suppose we are implementing a large instruction set with many complex instructions. In such an implementation it is likely that there are opportunities to reuse microcode sequences in interpreting similar instructions or in implementing operand access and decoding. Supporting subroutines in the microcode enables sharing of such microprogram sequences without having to duplicate the microinstructions. For this reason, microcoded control units that are used to implement complex microprograms (with hundreds to thousands of microinstructions) often provide support for microcode subroutines. Such subroutines are normally implemented by providing a return address stack within the control unit and using scratchpad registers to pass parameters.

Defining a Microinstruction Format

The microprogram is a symbolic representation of the control that will be translated by a program to control logic. In this way, we can choose how many fields a microinstruction should have and what control signals are affected by each field. The format of the microinstruction should be chosen so as to simplify the representation, making it easier to write and understand the microprogram. For example, it is useful to have one field that controls the ALU and a set of three fields that determine the two sources for the ALU operation as well as the destination of the ALU result. In addition to readability, we would also like the microprogram format to make it difficult or impossible to write inconsistent microinstructions. A microinstruction is inconsistent if it requires that a given control signal be set to two different values. We will see an example of how this could happen shortly.

To avoid a format that allows inconsistent microinstructions, we can make each field of the microinstruction responsible for specifying a nonoverlapping set of control signals. To choose how to make this partition of the control signals for this implementation into microinstruction fields, it is useful to reexamine two previous figures:

- Figure 5.28 on page 323, which shows all the control signals and how they affect the datapath
- Figure 5.29 on page 324, which shows the function of each datapath control signal

Signals that are never asserted simultaneously may share the same field. Figure 5.7.1 shows how the microinstruction can be broken into seven fields and defines the general function of each field. The first six fields of the microinstruction control the datapath, while the Sequencing field (the seventh field) specifies how to select the next microinstruction.

Microinstructions are usually placed in a ROM or a PLA (both described in Appendix B and used to implement control in Appendix C), so we can assign addresses to the microinstructions. The addresses are usually given out sequentially, in the same way that we chose sequential numbers for the states in the finite state machine. Three different methods are available to choose the next microinstruction to be executed:

1. Increment the address of the current microinstruction to obtain the address of the next microinstruction. This sequential behavior is indicated in the microprogram by putting `Seq` in the Sequencing field. Since sequential execution of instructions is encountered often, many microprogramming systems make this the default.

Field name	Function of field
ALU control	Specify the operation being done by the ALU during this clock; the result is always written in ALUOut.
SRC1	Specify the source for the first ALU operand.
SRC2	Specify the source for the second ALU operand.
Register control	Specify read or write for the register file, and the source of the value for a write.
Memory	Specify read or write, and the source for the memory. For a read, specify the destination register.
PCWrite control	Specify the writing of the PC.
Sequencing	Specify how to choose the next microinstruction to be executed.

FIGURE 5.7.1 Each microinstruction contains these seven fields. The values for each field are shown in Figure 5.7.2.

2. Branch to the microinstruction that begins execution of the next MIPS instruction. We will label this initial microinstruction (corresponding to state 0) as `Fetch` and place the indicator `Fetch` in the Sequencing field to indicate this action.
3. Choose the next microinstruction based on the control unit input. Choosing the next microinstruction on the basis of some input is called a *dispatch*. Dispatch operations are usually implemented by creating a table containing the addresses of the target microinstructions. This table is indexed by the control unit input and may be implemented in a ROM or in a PLA. There are often multiple dispatch tables; for this implementation, we will need two dispatch tables, one to dispatch from state 1 and one to dispatch from state 2. We indicate that the next microinstruction should be chosen by a dispatch operation by placing `Dispatch i`, where `i` is the dispatch table number, in the Sequencing field.

Figure 5.7.2 gives a description of the values allowed for each field of the microinstruction and the effect of the different field values. Remember that the microprogram is a symbolic representation. This microinstruction format is just one example of many potential formats.

Elaboration: The basic microinstruction format may allow combinations that cannot be supported within the datapath. Typically, a microassembler will perform checks on the microinstruction fields to ensure that such inconsistencies are flagged as errors and corrected. An alternative is to structure the microinstruction format to avoid this, but this might make the microinstruction harder to read. Most microprogramming systems choose readability and require the microcode assembler to detect inconsistencies.

Creating the Microprogram

Now let's create the microprogram for the control unit. We will label the instructions in the microprogram with symbolic labels, which can be used to specify the contents of the dispatch tables (see Section C.5 in Appendix C for a discussion of how the dispatch tables are defined and assembled). In writing the microprogram, there are two situations in which we may want to leave a field of the microinstruction blank. When a field that controls a functional unit or that causes state to be written (such as the Memory field or the ALU dest field) is blank, no control signals should be asserted. When a field *only* specifies the control of a multiplexor that determines the input to a functional unit, such as the SRC1 field, leaving it

Field name	Values for field	Function of field with specific value
Label	Any string	Used to specify labels to control microcode sequencing. Labels that end in a 1 or 2 are used for dispatching with a jump table that is indexed based on the opcode. Other labels are used as direct targets in the microinstruction sequencing. Labels do not generate control signals directly but are used to define the contents of dispatch tables and generate control for the Sequencing field.
ALU control	`Add`	Cause the ALU to add.
	`Subt`	Cause the ALU to subtract; this implements the compare for branches.
	`Func code`	Use the instruction's funct field to determine ALU control.
SRC1	`PC`	Use the PC as the first ALU input.
	`A`	Register A is the first ALU input.
SRC2	`B`	Register B is the second ALU input.
	`4`	Use 4 for the second ALU input.
	`Extend`	Use output of the sign extension unit as the second ALU input.
	`Extshft`	Use the output of the shift-by-two unit as the second ALU input.
Register control	`Read`	Read two registers using the rs and rt fields of the IR as the register numbers, putting the data into registers A and B.
	`Write ALU`	Write the register file using the rd field of the IR as the register number and the contents of ALUOut as the data.
	`Write MDR`	Write the register file using the rt field of the IR as the register number and the contents of the MDR as the data.
Memory	`Read PC`	Read memory using the PC as address; write result into IR (and the MDR).
	`Read ALU`	Read memory using ALUOut as address; write result into MDR.
	`Write ALU`	Write memory using the ALUOut as address; contents of B as the data.
PCWrite control	`ALU`	Write the output of the ALU into the PC.
	`ALUOut-cond`	If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.
	`Jump address`	Write the PC with the jump address from the instruction.
Sequencing	`Seq`	Choose the next microinstruction sequentially.
	`Fetch`	Go to the first microinstruction to begin a new instruction.
	`Dispatch i`	Dispatch using the ROM specified by `i` (1 or 2).

FIGURE 5.7.2 Each field of the microinstruction has a number of values that it can take on. The second column gives the possible values that are legal for the field, and the third column defines the effect of that value. Each field value, other than the label field, is mapped to a particular setting of the datapath control lines; this mapping is described in Appendix C, Section C.5. That section also shows how the label field is used to generate the dispatch tables. As we will see, the microcode implementation will differ slightly from the finite state machine control, but only in ways that do not affect instruction semantics.

blank means that we do not care about the input to the functional unit (or the output of the multiplexor).

The easiest way to understand the microprogram is to break it into pieces that deal with each component of instruction execution, just as we did when we designed the finite state machine.

The first component of every instruction execution is to fetch the instructions, decode them, and compute both the sequential PC and branch target PC. These actions correspond directly to the first two steps of execution described on pages 325 through 329. The two microinstructions needed for these first two steps are shown below:

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshft	Read			Dispatch 1

To understand what each microinstruction does, it is easiest to look at the effect of a group of fields. In the first microinstruction, the fields asserted and their effects are the following:

Fields	Effect
ALU control, SRC1, SRC2	Compute PC + 4. (The value is also written into ALUOut, though it will never be read from there.)
Memory	Fetch instruction into IR.
PCWrite control	Causes the output of the ALU to be written into the PC.
Sequencing	Go to the next microinstruction.

The label field, containing the label `Fetch`, will be used in the Sequencing field when the microprogram wants to start the execution of the next instruction.

For the second microinstruction, the operations controlled by the microinstruction are the following:

Fields	Effect
ALU control, SRC1, SRC2	Store PC + sign extension (IR[15:0]) << 2 into ALUOut.
Register control	Use the rs and rt fields to read the registers placing the data in A and B.
Sequencing	Use dispatch table 1 to choose the next microinstruction address.

We can think of the dispatch operation as a *case* or *switch* statement with the opcode field and the dispatch table 1 used to select one of four different microinstruction sequences with one of four different labels (all ending in "1"):

- `Mem1` for memory-reference instructions
- `Rformat1` for R-type instructions
- `BEQ1` for the branch equal instruction
- `JUMP1` for the jump instruction

The microprogram for memory-reference instructions has four microinstructions, as shown below. The first instruction does the memory address calculation. A two-instruction sequence is needed to complete a load (memory read followed by register file write), while the store requires only one microinstruction after the memory address calculation:

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
`Mem1`	`Add`	`A`	`Extend`				`Dispatch 2`
`LW2`					`Read ALU`		`Seq`
				`Write MDR`			`Fetch`
`SW2`					`Write ALU`		`Fetch`

Let's look at the fields of the first microinstruction in this sequence:

Fields	Effect
ALU control, SRC1, SRC2	Compute the memory address: Register (rs) + sign-extend (IR[15:0]), writing the result into ALUOut.
Sequencing	Use the second dispatch table to jump to the microinstruction labeled either `LW2` or `SW2`.

The first microinstruction in the sequence specific to `lw` is labeled `LW2`, since it is reached by a dispatch through table 2. This microinstruction has the following effect:

Fields	Effect
Memory	Read memory using the ALUOut as the address and writing the data into the MDR.
Sequencing	Go to the next microinstruction.

The next microinstruction completes execution with a microinstruction that has the following effects:

Fields	Effect
Register control	Write the contents of the MDR into the register file entry specified by rt.
Sequencing	Go to the microinstruction labeled `Fetch`.

The store microinstruction, labeled `SW2`, operates similarly to the load microinstruction labeled `LW2`:

Fields	Effect
Memory	Write memory using contents of ALUOut as the address and the contents of B as the value.
Sequencing	Go to the microinstruction labeled `Fetch`.

The microprogram sequence for R-type instructions consists of two microinstructions. The first does the ALU operation (and is labeled `Rformat1` for dispatch purposes), while the second writes the result into the register file:

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
`Rformat1`	`Func code`	`A`	`B`				`Seq`
				`Write ALU`			`Fetch`

You might think that because the fields of these two microinstructions do not conflict (i.e., each uses different fields), you could combine them into one. Indeed, microcode optimizers perform such operations when compiling microcode. In this case, however, the result of the ALU instruction is written into the register ALUOut, and the written value cannot be read until the next clock cycle; hence we cannot combine them into one microinstruction. (If you did combine them, you'd end up writing the wrong thing into the register file!) You could try to remove the ALUOut register to allow the two microinstructions to be combined, but this would require lengthening the clock cycle to allow the register file write to occur in the same clock cycle as the ALU operation.

The first microinstruction initiates the ALU operation:

Fields	Effect
ALU control, SRC1, SRC2	The ALU operates on the contents of the A and B registers, using the function field to specify the ALU operation.
Sequencing	Go to the next microinstruction.

The second microinstruction causes the ALU output to be written in the register file:

Fields	Effect
Register control	The value in ALUOut is written into the register file entry specified by the rd field.
Sequencing	Go to the microinstruction labeled `Fetch`.

Because the immediately previously executed microinstruction computed the branch target address, the microprogram sequence for branch, labeled with `BEQ1`, requires just one microinstruction:

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
`BEQ1`	`Subt`	`A`	`B`			`ALUOut-cond`	`Fetch`

The asserted fields of this microinstruction are the following:

Fields	Effect
ALU control, SRC1, SRC2	The ALU subtracts the operands in A and B to generate the Zero output.
PCWrite control	Causes the PC to be written using the value already in ALUOut, if the Zero output of the ALU is true.
Sequencing	Go to the microinstruction labeled `Fetch`.

The jump microcode sequence also consists of one microinstruction:

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
`JUMP1`						`Jump address`	`Fetch`

Only two fields of this microinstruction are asserted:

Fields	Effect
PCWrite control	Causes the PC to be written using the jump target address.
Sequencing	Go to the microinstruction labeled `Fetch`.

The entire microprogram appears in Figure 5.7.3. It consists of the 10 microinstructions appearing above. This microprogram matches the 10-state finite state machine we designed earlier, since they were both derived from the same five-step execution sequence for the instructions. In more complex machines, the microprogram sequence might consist of hundreds or thousands of microinstructions and would be the representation of choice for the control. Datapaths of more complex machines typically require additional scratch registers used for holding intermediate results when implementing complex multicycle instructions. Registers A and B are like such scratch registers, but datapaths for more complex instruction sets often have a larger number of such registers with a richer set of

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshft	Read			Dispatch 1
Mem1	Add	A	Extend				Dispatch 2
LW2					Read ALU		Seq
				Write MDR			Fetch
SW2					Write ALU		Fetch
Rformat1	Func code	A	B				Seq
				Write ALU			Fetch
BEQ1	Subt	A	B			ALUOut-cond	Fetch
JUMP1						Jump address	Fetch

FIGURE 5.7.3 The microprogram for the control unit. Recall that the labels are used to determine the targets for the dispatch operations. `Dispatch 1` does a jump based on the IR to a label ending with a 1, while `Dispatch 2` does a jump based on the IR to a label ending with 2.

interconnections to other datapath elements. These registers are available to the microprogrammer and make the analogy of implementing the control as a programming task even stronger.

Implementing the Microprogram

Translating a microprogram into hardware involves two aspects: deciding how to implement the sequencing function and choosing a method of storing the main control function. The microprogram can be thought of as a text representation of a finite state machine, and implemented in exactly the same way we would implement a finite state machine: using a PLA to encode both the sequencing function as well as the main control (see Figure 5.37 on page 338). Often, however, both the implementation of the sequencing function, as well as the implementation of the main control function, are done differently, especially for large microprograms.

The alternative form of implementation involves storing the control function in a read-only memory (ROM) and implementing the sequencing function separately. Figure 5.7.4 shows this different way to implement the sequencing function: using an incrementer to choose the next microinstruction. In this type of implementation, the microcode storage would determine the values of the datapath control lines, as well as *how to select* the next state (as opposed to *specifying* the next state, as in our finite state machine implementation). The address select logic would contain the dispatch tables, implemented in ROMs or PLAs, and would, under the control of the address select outputs, determine the next microinstruction to execute. The advantage of this implementation of the sequencing

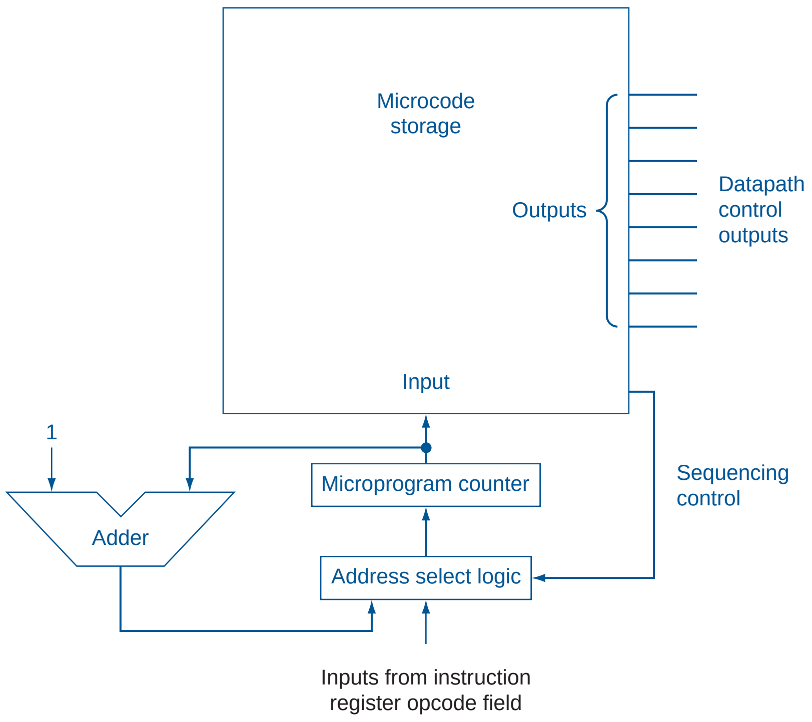

FIGURE 5.7.4 A typical implementation of a microcode controller would use an explicit incrementer to compute the default sequential next state and would place the microcode in a read-only memory. The microinstructions, used to set the datapath control, are assembled directly from the microprogram. The microprogram counter, which replaces the state register of a finite state machine controller, determines how the next microinstruction is chosen. The address select logic contains the dispatch tables as well as the logic to select from among the alternative next states; the selection of the next microinstruction is controlled by the sequencing control outputs from the control logic. The combination of the current microprogram counter, incrementer, dispatch tables, and address select logic forms a sequencer that selects the next microinstruction. The microcode storage may consist either of read-only memory (ROM) or may be implemented by a PLA. PLAs may be more efficient in VLSI implementations, while ROMs may be easier to change. Further discussions of the advantages of these two alternatives can be found at the end of this section.

function is that it removes the logic to implement normal sequencing of microinstructions, implementing such sequencing with a counter. Thus, in cases where there are long sequences of microinstructions, the explicit sequencer can result in less logic in the microcode controller.

In Figure 5.7.4, the main control function could be implemented in ROM, rather than implemented in a PLA. With a ROM implementation, the microprogram is assembled and stored in microcode storage and is addressed by the microprogram counter, in much the same way as a normal program is stored in program memory and the next instruction is chosen by the program counter. This analogy with programming is both the origin of the terminology (microcode, microprogramming, etc.) and the initial method by which microprograms were implemented (see Section 5.12).

Although the type of sequencer shown in Figure 5.7.4 is typically used to implement a microprogram control specification, it can also be used to implement a finite state specification. Section C.4 of Appendix C describes how to generate such a sequencer in more detail. Section C.5 describes how a microprogram can be translated to such an implementation. Similarly, Appendix C shows how the control function can be implemented in either a ROM or a PLA and discusses the trade-offs. In total, Appendix C shows how to go from the symbolic representations of finite state machines or microprograms shown in this chapter to either bits in a memory or entries in a PLA. If you are interested in detailed implementation or the translation process, you may want to proceed to Appendix C.

The choice of which way to represent the control (finite state diagram versus microprogram) and how to implement control (PLA versus ROM and encoded state versus explicit sequencer) are independent decisions, affected by both the structure of the control function and the technology used to implement the control.

Trade-offs in Control Approaches

Much has changed since Wilkes [1953] wrote the first paper on microprogramming. The most important changes are the following:

- Control units are implemented as integral parts of the processor, often on the same silicon die. They cannot be changed independent of the rest of the processor. Furthermore, given the right computer-aided design tools, the difficulty of implementing a ROM or a PLA is the same.
- ROM, which was used to hold the microinstructions, is no longer faster than RAM, which holds the machine language program. A PLA implementation of a control function is often much smaller than the ROM implementation, which may have many duplicate or unused entries. If the PLA is smaller, it is usually faster.
- Instruction sets have become much simpler than they were in the 1960s and 1970s, leading to reduced complexity in the control.
- Computer-aided design tools have improved so that control can be specified symbolically and, by using much faster computers, thoroughly simulated

before hardware is constructed. This improvement makes it plausible to get the control logic correct without the need for fixes later.

These changes have blurred the distinctions among different implementation choices. Certainly, using an abstract specification of control is helpful. How that control is then implemented depends on its size, the underlying technology, and the available CAD tools.

Check Yourself

Suppose we had an instruction set with five different classes of instructions; each class had some operations in common, and some of them required further separate micoprogram flows as follows:

Class 1: 1 flow for all instructions.

Class 2: 10 separate flows.

Class 3: 25 separate flows.

Class 4: 1 flow for all instructions.

Class 5: 15 separate flows.

Assuming a separate dispatch table for each dispatch operation, how many dispatch tables are needed and how many total entries are there?