MICROPROGRAMMING

In 1951, Maurice Wilkes first described microprogramming as a design approach for managing the datapath used in the central processing unit (CPU) found in computers (1). Microprogramming can take on a number of meanings, although there is an underlying theme that can be summed up in the following definition: A microprogram is an algorithm or program structure, used to control the sequencing of operations performed on a hardware device. The microprogrammed algorithm may be as simple as comparing two bits of information to test for logical equivalence, or it may be as complex as computing the arctangent of some angle. The programming level used can vary from microprocessor microcode and assembly code to high-level languages (e.g., C, C++, or Java). The hardware-based device being controlled may be a temperature sensor, a servo-controller, or a microprocessor pipeline. The underlying theme is that microprogramming uses simple steps (microinstructions or microoperations) to complete the task at hand.

Microprogramming provides a set of operations that can be used to control the functioning of a range of devices. These simple steps, which we will refer to hereafter as microinstructions, are purposely kept simple in order to allow the programmer to provide an efficient implementation of the desired operation. Other terms that have been associated with microprogramming include microcode, firmware, control programs, and software state machines. The most commonly discussed use of microprogramming is for the purpose described by Wilkes (i.e., controlling the internal state of the CPU). In actuality, microprogramming has been used most commonly to program simple controllers. Even though the application may be different, the underlying concepts of using simple operations remain the same. To discuss some of the issues related to microprogramming, a simple microprocessor example will be developed.

A microprogrammed control unit contains a number of elements that allow for the storage, sequencing, and interpretation of the microprogram (2). The microprogram will control sequencing of operations in the CPU datapath. In this article, we will provide an example of how various elements might be organized in a microprogrammed control unit and associated CPU datapath. The control storage maintains the microprogram image, generating the necessary control information to complete the requested programming task. The current microinstruction being addressed in the control storage is decoded, generating a set of control signals used to manipulate the datapath. The control storage addressing unit sequences through the microprogram, generating the address of the next microinstruction to execute.

CONTROL STORAGE

The set of microinstructions (i.e., the microprogram) is stored in the control storage. Traditionally, Read Only Memory (ROM) is used to maintain the contents of control storage. ROM has the characteristic that it does not need to be continuously powered to maintain its contents (i.e., ROM is nonvolatile memory). The contents of the ROM are fixed and do not change. The system designer may also provide a reserved portion of the Random Access Memory (RAM) on the system, to store the control program upon system initialization (RAM is volatile memory that will lose its contents when power is interrupted).

One feature of using RAM is that the contents of memory can be easily modified. The contents of the control storage can be dynamically updated. This class of control storage is referred to as Writable Control Storage (WCS). Early IBM 360/370 mainframes provided WCS (3). The advantages are that microprograms can be customized or modified without replacing the ROM. The microprogram can also be stored on a disk device and loaded into ROM upon system initialization.

The individual microinstructions stored in the control storage are accessed using a microsequencer. The microsequencer is driven by the series of macroinstructions (machine instructions) to be executed. Each macroinstruction is composed of a set of microinstructions. The current microinstruction being executed is buffered in the microdecode register (uDR). Multiple uDRs may be available to allow for the overlap of microinstructions (i.e., pipelining). The uDR is decoded to generate the appropriate values on the control signals associated with the control unit. The sample datapath will be used to illustrate how different control signals are used to manage the different CPU elements under control of the control unit, as shown in Fig. 1.

In our datapath, the control unit (which contains the microprogram) generates the control signals (which enable data to flow between units). The control storage unit initiates a sequence of control signal values when a new instruction image is read into the IR (Instruction Register). Other elements in our datapath include:

1. MAR—Memory Address Register (this register holds the address used to address data memory)
2. MDR—Memory Data Register (this register holds the data to be fetched from, or stored to, memory)
3. PC—Program Counter (this register holds the address of the next instruction to be fetched into the IR)
4. Register File—(contains the 32 registers r0–r31)
5. ALU—Arithmetic Logic Unit (used to perform arithmetic and logical operations on data)

There are three main busses in our datapath:

1. a-bus—transfers register and MDR data to the ALU, as well as transfers register data to the MAR, MDR, and PC.
2. b-bus—transfers register data to the ALU
3. c-bus—transfers ALU-generated data to the register file

The MAR, MDR, PC, and IR interface to memory, facilitating the loading of instructions and data from memory, and the storing of data to memory. In Fig. 1, the Add block repre-

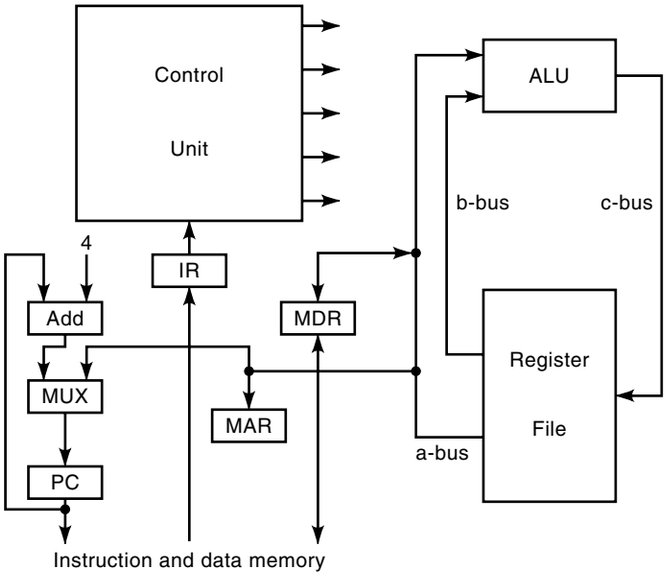

Figure 1. Sample datapath.

sents the Program Counter adder logic that increments the PC to point at the next sequential instruction (i.e., PC = PC + 4). The multiplexor block (MUX) decides whether to point the PC to the next sequential instruction or load the PC with the contents of a register (implementing a change in control flow in datapath execution) from the a-bus.

MICROINSTRUCTION DECODING

Microinstructions contain encoded information, which is used to generate the necessary signals that control the datapath of the central processing unit. The level of encoding is dictated by the constraints on the amount of control storage and the decoding performance requirements. The higher the degree of encoding, the shorter the microinstruction format will be. But more encoding implies more decoding will have to be performed (i.e., decoding will take more time).

For purposes of example, we will assume that a single microinstruction has a fixed format, 14 bits in length. The format of a microinstruction in our example control unit is shown in Fig. 2. There are 4 fields specified in this format:

1. Operation code field—3 bits wide
2. Operand 1 field—4 bits wide
3. Operand 2 field—4 bits wide
4. ALU operation code field—3 bits wide

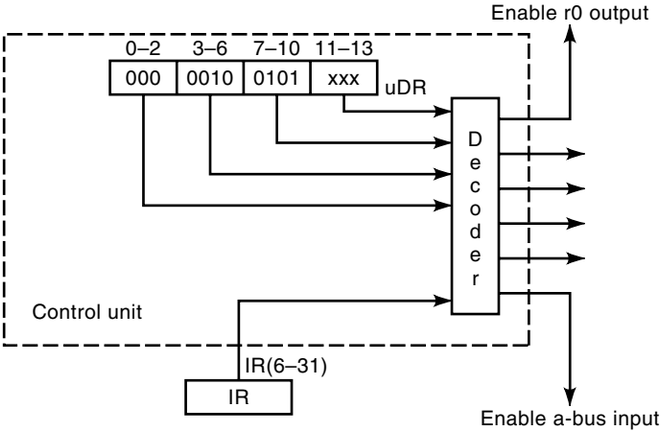

Figure 2. Decoding of the μDR for the add r1, r2, r3 instruction.

Table 1. Operation Code Definitions

Opcode	Operation
000	move op1,op2
001	read from memory into MDR
010	write to memory from MDR
011	PC = PC + 4
100	PC = op1
101	load [PC], IR
110	program ALU, ALU op
111	branch op1

The operation code field (i.e., opcode) identifies which type of microoperation is being specified. In our example microinstruction format, the 4-bit opcode field is defined in Table 1. The 4-bit operand fields 1 and 2 (op1 and op2) are defined in Table 2. The 3-bit ALU operation code field is defined in Table 3.

The operations defined in Table 1 specify either that the contents of a register (i.e., r0–r31, PC, MAR, or MDR) are to be transferred to a bus (i.e., bus-a, bus-b, or a memory bus) or that the contents of a bus (i.e., bus-a, bus-b, bus-c, or a memory bus) are to be transferred to a register (i.e., r0–r31, IR, or MDR). The enabling of register output to buses, and the latching of registers is under control of the microcoded control program located in the control storage. Control signals are generated (see Fig. 1), which control execution in the datapath.

Also notice that in Table 2 a number of operands are defined by fields in the IR. This register holds the instruction image as fetched from memory. The fields as defined can hold an immediate value or a register number (r0–r31). The data contained in these fields are used in a variety of ways to implement the macroinstructions.

The purpose of providing such low-level detail is to explain how macroinstructions are implemented using microprogram-

Table 2. Operand Value Definitions

Operand Values	Operands
0000	value in IR(6–18)
0001	value in IR(19–31)
0010	register in IR(6–10)
0011	register in IR(11–15)
0100	register in IR(16–20)
0101	a-bus
0110	b-bus
0111	c-bus
1000	memory
1001	MAR
1010	MDR
1011	IR
1100	uPC

Table 3. ALU Operation Code Definitions

Operand Code Values	Meaning
000	bus-c = bus-a + bus-b
001	bus-c = bus-a − bus-b
010	bus-c = bus-a SHL bus-b bits
011	bus-c = bus-a XOR bus-b
100	bus-c = bus-a AND bus-b
101	bus-c = bus-a OR bus-b
110	bus-c = NOT bus-a
111	bus-c = bus-a

ming. Each macroinstruction is defined by a sequence of microcoded words. In our example datapath, each microcode word is 14 bits wide. A variable number of microinstructions comprise a single macroinstruction. Next we will demonstrate how particular macroinstructions would be implemented in microcode.

MACROINSTRUCTIONS

A macroinstruction is just another term used to describe the fundamental instruction set associated with a CPU. The set of macroinstructions defined for a particular CPU is generally referred to as an Instruction Set Architecture (ISA). Just as a series of microinstructions is used to implement each macroinstruction, a series of macroinstructions is used to implement assembly or high-level language (e.g., C or Fortran) programming statements.

Macroinstructions can be grouped based on their functionality:

1. Arithmetic and logical operations,
2. Data transfer operations, and
3. Control transfer operations.

These three groups cover the range of operations that a typical ISA provides.

ALU Operations

Arithmetic and logical operations generally involve operations on one or two operands. At a minimum, operations include addition and subtraction, along with the basic logical operations (AND, OR, NOT, and EXCLUSIVE-OR). Other functionality that may be provided in an ALU include shifting, signed-mathematical operations, increment/decrement, and integer multiplication and division.

Data Transfer Operations

Data transfer operations include any operations that load data from, or store data to, memory. For pure Reduced Instruction Set Computer (RISC) processors [e.g., DLX (Sailer)], the ISA provides instructions that specifically perform loads and stores, but it does not combine these operations with arithmetic or control transfer instructions. In contrast, Complex Instruction Set Computer (CISC) ISA's (e.g., Intel's 80X86, Motorola's 680X0), data transfer operations can be combined with ALU operations, allowing memory accesses (both loads and stores) and arithmetic or logical operations to be combined in a single macroinstruction.

Control Transfer Operations

Control transfer operations include any that can cause instruction execution to follow a nonsequential execution path. Control transfer instructions can be conditional (the resulting control transfer outcome is dependent upon some current machine state) or unconditional (the resulting control transfer will also cause a break to a nonsequential execution path). Conditional branches allow the execution to make decisions dynamically in the program. Examples of conditional branches include jumps based on the result of an ALU operation (e.g., jump if greater than, jump if equal to zero) and loops based on a count register. Unconditional control transfers include those macroinstructions where there is no doubt whether we want the program to move to a new execution stream. Examples of unconditional branches include subroutine calls and returns, interrupts, and direct/indirect jump instructions.

MICROINSTRUCTION SEQUENCING

Next, we look at how we sequence through a microprogram. We do have many options. We will present the most fundamental, while suggesting alternative (more aggressive) implementations.

Because each macroinstruction is comprised of a number of microinstructions, we need a way to step through each microprogram that represents the desired macroinstruction. We begin by considering the microprogram for a particular macroinstruction, and then we will generalize the approach to consider a series of macroinstructions.

In Table 4 we present the microprogram for the add r1, r2, r3 instruction. This macroinstruction is read as follows: r3 = r1 + r2. The microprogram needs to send the contents of registers r1 and r2 to the inputs of the ALU along the a-bus and b-bus, respectively; program the ALU to perform an add; and then take the results from the c-bus and store it in r3.

The program shown in Table 4 is stored in ROM and is addressed using a microsequencer. Before explaining how addressing is implemented in the control unit, let us revisit how the control signals, which manage the datapath, are generated.

The bit patterns (1s and 0s) shown in Table 4 are loaded sequentially into the uDR. The first microinstruction (move r1, a-bus) translates to the bit pattern 00000000101xxx. This value is loaded into the uDR. Figure 2 shows the bit pattern in the uDR and the corresponding decoding logic used to generate the necessary control signals.

Again, notice that the IR comes into play here. The first operand is identified by decoding bits 6–10 of IR. These bits

Table 4. Microprogram Implementing Add r1, r2, r3, Including Microinstructions[a]

Microinstruction	Opcode	Operand 1	Operand 2	ALU Op
move r1,a-bus	000	0010	0101	xxx
move r2, b-bus	000	0011	0110	xxx
program ALU, add	110	xxxx	xxxx	000
move c-bus, r3	000	0111	0100	xxx

[a] x's denote don't care values in the microinstruction.

will contain an encoded value, indicating that register r1 is the first operand associated with this add. The location and definition of this bit pattern is defined by the ISA (designing the format of macroinstructions is a separate topic, and is not addressed in this article).

Now that we understand how a single microinstruction is executed, we need to move on to the next instruction. To do this, we need a microsequencer, which will step us through the microprogram.

MICROSEQUENCER

We will define a new register called the uPC. Like the macroinstruction PC, the uPC contains the address of the next microinstruction to execute. The uPC is modified when a new macroinstruction is loaded into the IR. After each microinstruction is executed, the microsequencer increments the uPC to point to the next sequential address (in our simplistic design here we will assume that our microprogram is explicitly specified for each macroinstruction) by incrementing the uPC by 1. In Fig. 3 we show how the add instruction might be stored in control storage and suggest how the uPC is updated. Updates to the uPC are controlled by a multiplexor. The multiplexor receives both the uPC + 1 and a decoded version of the opcode associated with the IR register. This multiplexor will select the decoder output when a new instruction is to be loaded into the IR.

After we execute each macroinstruction, we need to load the IR with the next instruction to be executed. This is accomplished using the microprogram sequence specified in Table 5. The first microinstruction increments the address in the PC to the next sequential address. The second instruction fetches the next instruction image from memory and places it in the IR. The last microinstruction in this sequence tells the multiplexor controlling the input to the uPC to pass the decoded value of the new IR value in order to update the uPC with the starting address of the next sequence of microinstructions. The uPC is latched at the beginning of each microinstruction. The MUX feeding the uPC is normally set to allow the uPC + 1 value to be loaded.

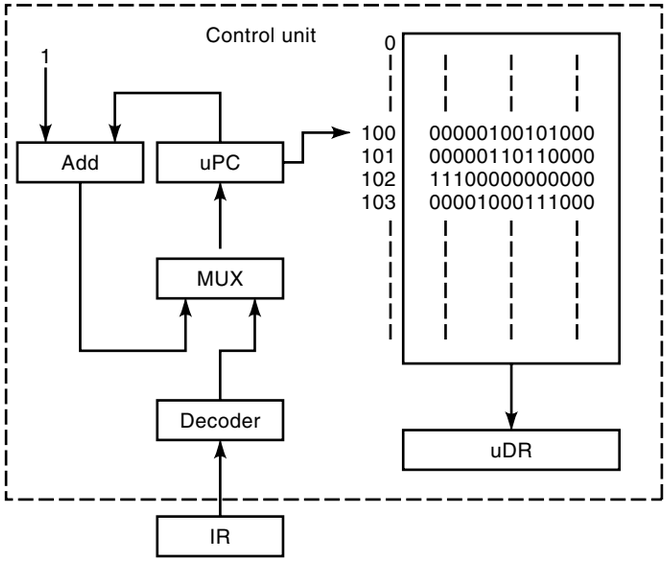

Figure 3. Control storage layout and microsequencer logic.

Table 5. Microinstruction Sequence for Updating the uPC for a Nonbranch Instruction

Microinstruction	Opcode	Operand 1	Operand 2	ALU Op
PC = PC + 4	011	xxxx	xxxx	xxx
load [PC], IR	101	xxxx	xxxx	xxx
branch op1	111	1100	xxxx	xxx

One exception to the sequence shown in Table 5 is when the current instruction is a branch instruction. In this case, the microinstruction sequence would look something like the sequence shown in Table 6 for a register-based jump.

Note that we have only discussed here one form of addressing for the target of the jump instruction. Another serious consideration that needs to be made when designing a microinstruction set is how to implement conditional branches specified in the macroinstruction set. To discuss a potential design, we first need to discuss the types of conditional branches the microinstruction set needs to support.

CONDITIONAL BRANCHES

Let us assume that our ISA only supports branches with targets contained in registers (this would be a poor choice in practice, but simplifies our example here significantly). Let us also assume that the ALU in our datapath provides a flags register that records particular characteristics regarding the last ALU operation executed. The flags register contains the following fields:

1. Zero flag—set to 1 if the last operation generated a 0 result.
2. Negative flag—set to 1 if the last operation generated a negative result.
3. Equal flag—set to 1 if the last operation generated an equal result.

We need to support the following conditional branch macroinstructions:

1. jz—jump to the address specified in the register if the previous ALU operation resulted in the zero flag being set.
2. jlz—jump to the address specified in the register if the previous ALU operation resulted in the negative flag being set.

Table 6. Microinstruction Sequence for Updating the uPC for a Register-Based Direct Jump[a]

Microinstruction	Opcode	Operand 1	Operand 2	ALU Op
mov op1, op2	000	0010	0101	xxx
PC = op1	100	0101	xxxx	xxx
load [PC], IR	101	xxxx	xxxx	xxx
branch op1	111	1100	xxxx	xxx

[a] The register value is stored in bits 6–10 of the macroinstruction.

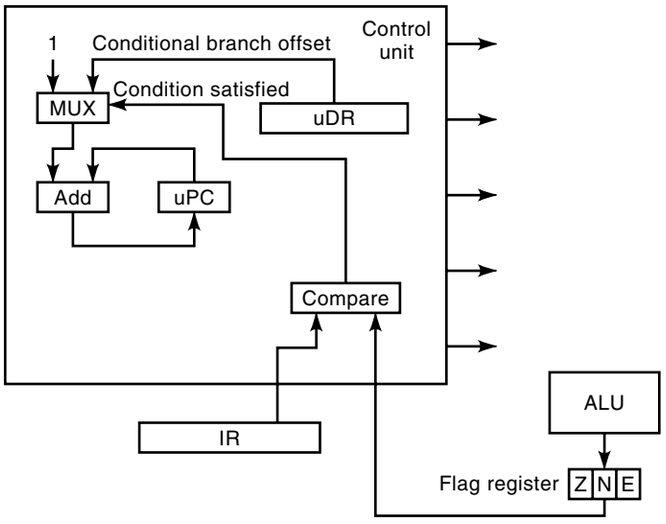

Figure 4. Additional control logic needed to implement conditional branch macroinstructions.

3. je—jump to the address specified in the register if the previous ALU operation resulted in the equal flag being set.

The difficulty with handling conditional branch macroinstructions is that we then must provide conditional branch microinstructions. We can implement these macroinstructions by using the circuitry suggested in Fig. 4. Here we see that bits in the macroinstruction IR are compared against the current status of the ALU flags. The multiplexor controlling the setting of the uPC is controlled by the comparison logic. This macroinstruction opcode currently in the IR provides information on the type of condition desired (jz, jnz, jlz, je or jne) and the multiplexor conditionally selects which next value to load in uPC. Additionally, we must supply the microinstruction target offset that will be branched to in the microprogram if this condition is satisfied. Table 7 modifies the definition of the ALU operation field. As a result, rename this field the Operand 3 field. When the microinstruction opcode is a branch and the macroinstruction IR contains a conditional branch, the Operand 3 field will contain the branch offset to branch to. Table 8 shows the microinstruction sequence to implement the conditional branch jz. Again, the branch target address is stored in a register.

Table 7. Additional Operand Value Definitions to Implement Conditional Branching

Opcode	Operand 3	Meaning
110	000	bus-c = bus-a + bus-b
110	001	bus-c = bus-a − bus-b
110	010	bus-c = bus-a SHL bus-b bits
110	011	bus-c = bus-a XOR bus-b
110	100	bus-c = bus-a AND bus-b
110	101	bus-c = bus-a OR bus-b
110	110	bus-c = NOT bus-a
110	111	bus-c = bus-a
111	000	load new decode IR to uPC
111	000–111	uPC = uPC + Operand Value

In this example, a conditional branch microinstruction is executed at address 200 upon entry into the microroutine for jz. If the zero flag is set, the uPC = uPC + 4 (a 4 is stored in the Operand 3 field), causing a branch to microinstruction address 204. If the zero flag is reset, then the sequential path of microinstructions is followed (microinstructions 201–203). At microinstruction address 202, we will load a new IR value, and in microstruction 203 the uPC is updated with the starting address of the next microinstruction sequence.

If the zero flag was set, the microsequencer would continue execution from microinstruction address 204. At microinstruction address 206, the macroinstruction IR is updated, and in the next microinstruction, the uPC is updated to point at the starting address of the next microinstruction sequence. Notice that there is considerable commonality in these two sequences. This will be addressed when we discuss possible optimizations.

PUTTING IT ALL TOGETHER

Now that we have covered the fundamentals of microinstruction sequencing while considering the necessary support for a general class of macroinstructions (including conditional branches), we will see how multiple macroinstructions might be processed. Let us consider the following snipit of high-level language (C) code:

```
for (x = 0; x < 100; x++)
y = y + x;
```

The corresponding macroinstructions generated by a C compiler for this code snipit might look like the sequence shown in Table 9. The microinstruction sequences for the unique macroinstructions used in Table 9 is shown in Table 10. Notice that there are only five unique macroinstructions present in our code snipit:

1. add rx, rx, rx (addresses 100 - 106)
2. add IR(6-18), rx, rx (addresses 107 - 113)
3. sub rx, rx, rx (addresses 114-120)
4. sub IR(6-18), rx, rx (addresses 121 - 127)
5. jz rx (addresses 128 - 135)

Within the five unique microinstruction sequences comprising 135 microinstructions, there are only 11 unique microinstructions in our code snipit:

1. move rx,a-bus
2. move rx, b-bus
3. program ALU, add
4. move c-bus, rx
5. PC = PC + 4
6. load[PC],IR
7. branch op1
8. move IR(6-18),a-bus
9. program ALU, sub

Table 8. Microinstruction Sequence for Conditionally Updating the uPC for a Register-Based Direct Conditional Jump[a]

Address	Microinstruction	Opcode	Operand 1	Operand 2	Operand 3
200	cbranch op1, op3	111	1100	xxxx	100
201	PC = PC + 4	011	xxxx	xxxx	xxx
202	load[PC],IR	101	xxxx	xxxx	xxx
203	branch op1	111	1100	xxxx	000
204	mov op1, op2	000	0010	0101	xxx
205	PC = op1	100	0101	xxxx	xxx
206	load [PC], IR	101	xxxx	xxxx	xxx
207	branch op1	111	1100	xxxx	000

[a] The register value is stored in bits 6–10 of the macroinstruction. The sequence implements the jz macroinstruction.

10. cbranch op1, op3
11. PC = op1

It should be evident from this example that there are several opportunities to improve the utilization of the ROM space. Next we will explore the questions of speed of execution and of storage space, in the context of the control unit design and the accompanying microprogram.

OPTIMIZATIONS

To this point, we have not considered the practicality of implementing the described control unit and accompanying microcode. We have only been concerned with presenting general principles that would apply, independent of the eventual implementation. Microprogram execution performance is a critical issue in the design of high-performance microprocessors (actually most microprocessors today use hardwired control, even though all the concepts presented so far can be implemented in hardwired logic). One key issue related to the performance of microcode is directly related to the amount of encoding that has been used in the microinstruction format.

We could choose to perform as little encoding as possible, providing a bit in the microinstruction word for each control line bit and performing little or no decoding of the microinstruction. Microinstructions designed using this principle are generally referred to as horizontal microcode. The overhead of using horizontal microcode is felt in the width of the ROM used to store the microprogram. The width of each microinstruction will be very wide (the number of control signals generated by the control unit is typically greater than 100 and may be more than 200).

Table 9. Macroinstruction Sequence Needed to Execute Code snipet

Address	Macroinstruction	Description
100	sub r1, r1, r1	clears register r1, where x will be stored
104	sbu r10, r10, r10	clears register r10, where the conditional branch target will be stored
108	add r1, 99, r1	stores the loop counter in register r1
112	add r10, r10, r10	stores branch target
116	add r1, r2, r2	y is stored in r2, add x to y
120	sub r1, 1, r1	decrements the loop counter
124	jz r10	conditionally exits loop

Encoding allows for the reduction in the length of a typical microinstruction. Deeply encoded microinstructions are typically referred to as vertical microcode. This will reduce the amount of ROM needed to store the microprogram but may introduce unwanted performance degradation. In many microprocessor designs, the generation of the control signals produced by the control unit are on the critical path in the design (the critical path refers to a timing path in a design where time requirements are an issue). Heavily encoded microcode can add multiple gate levels of delay to a timing path. This may limit our ability to speed up the oscillator clock used to control the design.

So there is much room for compromise between horizontal and vertical microcode. The implementation just presented is basically a middle ground in encoding and complexity (although it lacks a number of performance and storage space considerations, which will be clear shortly in this article). One obvious advantage of using a wider (i.e., more horizontal) microcode word is the ability to control multiple control signals by performing encoding carefully (such that any two control signals that must be generated on the same cycle will be encoded in separate fields in the microinstruction). Even though we have done a reasonable job of providing some parallelism in the execution of the microprogram in our present implementation, further optimizations could easily be performed.

TWO-LEVEL CONTROL STORE DESIGN

As pointed out in our previous discussion of the microprogram shown in Table 9, there were a total of 135 microinstructions, but only 11 of these were unique. What if instead of storing the complete microcode sequence for each macroinstruction, we stored only the unique microinstructions that are needed to generate the same sequence of operations as found in the full program in Table 9. We refer to such a design as a two-level control store and refer to the microcode as a nanoprogram.

Table 11 shows the 11 unique microinstructions stored in our control storage. The only difference is that now a level of indirection takes place. Each macroinstruction is implemented by providing the sequence of control store addresses associated with this macroinstruction. The main benefit of using a two-level control store is that we can reduce the amount of duplication of in our microprogram. If we consider only the number of bits needed to store the unique microinstructions, we would wind up with 154 bits of storage (as opposed to the

Table 10. Microinstruction Sequences for the Macroinstructions Shown in Table 9

Address	Microinstruction	Opcode	Operand 1	Operand 2	Operand 3
100	move rx,a-bus	000	0010	0101	xxx
101	move rx, b-bus	000	0011	0110	xxx
102	program ALU, add	110	xxxx	xxxx	000
103	move c-bus, rx	000	0111	0100	xxx
104	PC = PC + 4	011	xxxx	xxxx	xxx
105	load[PC],IR	101	xxxx	xxxx	xxx
106	branch op1	111	1100	xxxx	000
107	move IR(6–18),a-bus	000	0010	0101	xxx
108	move rx, b-bus	000	0011	0110	xxx
109	program ALU, add	110	xxxx	xxxx	000
110	move c-bus, rx	000	0111	0100	xxx
111	PC = PC + 4	011	xxxx	xxxx	xxx
112	load[PC],IR	101	xxxx	xxxx	xxx
113	branch op1	111	1100	xxxx	000
114	move rx,a-bus	000	0010	0101	xxx
115	move rx, b-bus	000	0011	0110	xxx
116	program ALU, sub	110	xxxx	xxxx	001
117	move c-bus, rx	000	0111	0100	xxx
118	PC = PC + 4	011	xxxx	xxxx	xxx
119	load[PC],IR	101	xxxx	xxxx	xxx
120	branch op1	111	1100	xxxx	000
121	move IR(6–18),a-bus	000	0000	0101	xxx
122	move rx, b-bus	000	0011	0110	xxx
123	program ALU, sub	110	xxxx	xxxx	001
124	move c-bus, rx	000	0111	0100	xxx
125	PC = PC + 4	011	xxxx	xxxx	xxx
126	load[PC],IR	101	xxxx	xxxx	xxx
127	branch op1	111	1100	xxxx	000
128	cbranch op1, op3	111	1100	xxxx	100
129	PC = PC + 4	011	xxxx	xxxx	xxx
130	load[PC],IR	101	xxxx	xxxx	xxx
131	branch op1	111	1100	xxxx	000
132	mov rx, a-bus	000	0010	0101	xxx
133	PC = op1	100	0101	xxxx	xxx
134	load [PC], IR	101	xxxx	xxxx	xxx
135	branch op1	111	1100	xxxx	000

1890 bits needed in the single-level control storage scheme). But we have forgotten to consider how we plan to sequence through these microinstructions. A simple alternative is to provide a 135-entry control program that contains only the addresses of the microinstructions shown in Table 11. In Table 12, we provide just such a table, completing the design of the two-level control storage unit.

To make a fair assessment of the benefits of a two-level control store, we need to consider the extra storage needed to store the 4-bit index stored in the second-level address field

Table 11. Unique Microinstructions Stored in our Two-Level Control Storage

Address	Microinstruction	Opcode	Operand 1	Operand 2	Operand 3
000	move rx,a-bus	000	0010	0101	xxx
001	move rx, b-bus	000	0011	0110	xxx
002	program ALU, add	110	xxxx	xxxx	000
003	move c-bus, rx	000	0111	0100	xxx
004	PC = PC + 4	011	xxxx	xxxx	xxx
005	load[PC],IR	101	xxxx	xxxx	xxx
006	branch op1	111	1100	xxxx	000
007	move IR(6–18),a-bus	000	0010	0101	xxx
008	program ALU, sub	110	xxxx	xxxx	001
009	cbranch op1, op3	111	1100	xxxx	100
010	PC = op1	100	0101	xxxx	xxx

Table 12. Microinstruction Sequences for the Macroinstructions Shown in Table 9

1st-Level Address	Microinstruction	2nd-Level Address
100	move rx,a-bus	0000
101	move rx, b-bus	0001
102	program ALU, add	0010
103	move c-bus, rx	0011
104	PC = PC + 4	0100
105	load[PC],IR	0101
106	branch op1	0110
107	move IR(6–18),a-bus	0111
108	move rx, b-bus	0001
109	program ALU, add	0010
110	move c-bus, rx	0011
111	PC = PC + 4	0100
112	load[PC],IR	0101
113	branch op1	0110
114	move rx,a-bus	0000
115	move rx, b-bus	0001
116	program ALU, sub	1000
117	move c-bus, rx	0011
118	PC = PC + 4	0100
119	load[PC],IR	0101
120	branch op1	0110
121	move IR(6–18),a-bus	0111
122	move rx, b-bus	0001
123	program ALU, sub	1000
124	move c-bus, rx	0011
125	PC = PC + 4	0100
126	load[PC],IR	0101
127	branch op1	0110
128	cbranch op1, op3	1001
129	PC = PC + 4	0100
130	load[PC],IR	0101
131	branch op1	0110
132	mov rx, a-bus	0000
133	PC = op1	1010
134	load [PC], IR	0101
135	branch op1	0110

in the first-level table. This adds up to 694 bits, versus 1890 bits used in the single-level table design. Two points need to be made here. First, we have implemented only a small fraction of the total number of macroinstructions typically present in an instruction set architecture. The difference in the amount of storage used grows dramatically because, as we add additional microinstructions to our microcoded control program, the amount of storage used in the two-level control store implementation will increase by 4 bits per microinstruction, whereas the corresponding single-level control store will grow by a full 14 bits per microinstruction. In general, if a microprogram contains x unique microinstructions, the full microprogram contains y microinstructions, and a single microinstruction is z bits wide; the total number of bits needed for a single-level store is given in Eq. (1),

$$\text{Number of bits} = (y * z) \tag{1}$$

whereas the number of bits need for a two-level store is given in Eq. (2).

$$\text{Number of bits} = (y * \log_2 x) + (x * z) \tag{2}$$

An issue that we failed to address so far is the extra hardware cost of providing sequencing and addressing for two independent storage units. This is an issue. There is also the issue of performing two memory accesses to satisfy the execution of a single microinstruction (this could make our design too costly).

ADVANCED MICROPROGRAMMING CONCEPTS

One design alternative is to use a single-level control store and allow each microinstruction to act as a conditional branch, based on the opcode present in the macroinstruction IR. If we know all possible sequences through the microcode, then given the current microinstruction and the opcode of the current macroinstruction being executed, we can compute the offset of the next microinstruction that is to follow. What we have created is a Finite State Machine (FSM), that given the current input to the control unit (the macroinstruction opcode present in the IR) and the current state stored in the control unit (the in-flight microinstruction), we can determine the next state for the controller and the associated control signals. This can be represented using Boolean logic, and a number of tools such as truth tables and Karnaugh maps can quickly provide us with an optimized design (4).

Many control units built today do not use microcode, but instead use hardwired control logic. The FSM described previously is simply a large decoder block that produces the desired values on the output control signals on each microsequencer clock tick. The value of the control signals is gated into latches, and these are used to control the datapath. The decoder logic can be efficiently realized using a standard combinational logic circuit call a Programmable Logic Array (PLA). These arrays consist of an array of AND gates, followed by an array of OR gates.

Another design alternative is to provide the capability of executing multiple macroinstructions in a pipelined CPU (5). This would require the addition of multiple uPCs, each handling the sequencing through the microprogram for an individual active macroinstruction. We would also need to buffer the IR contents for a number of cycles in order to allow the next IR value to enter the control unit. One additional level of complexity here is that we need to make sure that no two instructions attempt to manipulate the same control line during the same clock cycle.

Another issue we need to consider is how to realize a control unit that will manage a superscalar datapath implementation (6). Superscalar pipelines allow multiple instructions to be active in the pipeline concurrently. We will need to be able to handle the same basic operations already described (i.e., microinstruction storage, sequencing, decoding), but now we need to consider doing this for multiple instructions concurrently. The complexity of these designs grows quickly. As a result, most superscalar control units are designed using custom logic.

SUMMARY

Even though the beginnings of microprogramming date back to the early 1950s, we still see extensive use of microprogramming today. This article has given one complete example of how microprogramming is used to control a CPU datapath. In

many CPUs developed today, microcode has been replaced with hardwired control logic. Many of the same principles apply to hardwired control, which are fundamental to the design of microprograms, because both are variations of a finite state machine (one developed in hardware, another developed in software and supporting control logic).

The major application of microprogramming is seen in the microcontroller world. These simple CPUs are used to control a variety of electrical and mechanical devices. Microprogramming is used to control these devices. Other discussion of microcoding and microprogramming can be found in Refs. 7–10. We refer the reader to these recognized textbooks for further information on this topic.

BIBLIOGRAPHY

1. M. V. Wilkes, The best way to design an automatic calculating machine, *Rep. Manchester Univ. Comput. Inaug. Conf.,* 16–18, July 1951.
2. F. J. Hill and G. R. Peterson, *Digital Systems: Hardware Organization and Design,* 3rd ed., New York: Wiley, 1987.
3. P. M. Sailer and D. R. Kaeli, *The DLX Instruction Set Architecture Handbook,* San Mateo, CA: Morgan-Kaufmann, 1996.
4. J. F. Wakerly, *Digital Design: Principle and Practices,* 2nd ed., Englewood Cliffs, NJ: Prentice-Hall, 1994.
5. M. J. Flynn, *Computer Architecture: Pipelined and Parallel Processor Design,* Boston: Jones and Bartlett, 1995.
6. M. Johnson, *Superscalar Microprocessor Design,* Englewood Cliffs, NJ: Prentice-Hall, 1990.
7. C. Hamacher, Z. G. Vranesic, and S. G. Zaky, *Computer Organization,* 4th ed., New York: McGraw-Hill, 1996.
8. S. Tannenbaum, *Structured Computer Organizations,* 3rd ed., Englewood Cliffs, NJ: Prentice-Hall, 1990.
9. M. M. Mano and C. R. Kime, *Logic and Computer Design Fundamentals,* Upper Saddle River, NJ: Prentice-Hall, 1997.
10. D. Patterson and J. L. Hennessy, *Computer Organization and Design—The Hardware / Software Interface,* San Mateo, CA: Morgan-Kaufmann, 1994.

DAVID R. KAELI
Northeastern University