Microcoded Versus Hard-wired Control

A comparison of two methods for implementing the control logic for a simple CPU

Phil Koopman

THE INSTRUCTION decoding and execution control sections of modern computers are prime areas for using programmable hardware. Two of the most widely used methods for designing CPU control sections in microprocessors, minicomputers, and mainframes are microcode and hard-wired logic. Each method has its advantages, and both are natural applications for programmable hardware devices.

Architectural Description

I'll start by giving the specifications for a simple computer architecture, then walk through the implementation of this architecture using both microcoded and hard-wired design strategies. While both approaches require the same description and specification groundwork, they use different schemes to generate control signals.

I will examine the CPU architecture of Toy, a fictitious computer designed especially for this article. The CPU has an accumulator (ACC), an arithmetic logic unit (ALU), an instruction register (IR), a program counter (PC), some random-access memory (RAM), and some control logic. Figure 1 is a block diagram of the Toy architecture. All data paths are 16 bits wide with 12-bit memory-address paths. You can directly implement the ALU, ACC, IR, PC, multiplexer, and RAM sections of Toy using commonly available chips. Toy's control-logic section will require detailed design and the use of customized hardware or a large number of combinatorial logic gates.

The Toy instruction format shown in figure 2 consists of a 4-bit op code and a 12-bit address field. The 16 implemented op codes are shown in table 1. Op codes 8 through 15 do not make use of the instruction's address field.

Since Toy is a single-accumulator machine, the instructions ADD, SUB, AND, OR, and XOR combine the contents of a memory location with the accumulator and return the result to the accumulator. The instructions STORE and LOAD transfer the accumulator to and from RAM. The instructions NOT, INC, DEC, and ZERO operate on the accumulator alone. While JMPZ is the only branching instruction, you can program an unconditional branch by following ZERO with a JMPZ. Finally, the four unused op codes act as null operations (NOPs) to eliminate the annoyance of dealing with illegal op codes.

Control Logic

The control-logic section translates the op-code bit patterns into CPU-control and timing signals. Figure 1 shows the op-code inputs to the control-logic unit and the control-signal outputs required to run the rest of the CPU. The signals ALU0 through ALUCIN control the ALU. (I based the bit assignments on those for the 74181 ALU chip. See *The TTL Data Book*, listed in the Bibliography.) If ALUMODE is a 1, then the ALU will perform a logical operation; if it's a 0, the ALU will perform an arithmetic operation. ALU0 through ALU3 control which arithmetic or logic operation the ALU is performing. ALUCIN acts as the carry-in for the ALU.

When the signal CLOCK[ACC] is a 1, the ACC register is loaded with the value of its inputs at the rising edge of the system clock. This is usually referred to as "clocking in" the contents of the ACC. When the signal CLOCK[IR] is a 1, the contents of the IR are clocked in from the RAM output. This is the mechanism used to decode the next op code. When ADDR=IR is a 1, the RAM address multiplexer places the contents of the IR address field onto the RAM address bus. When it is a 0, the PC is used to address RAM. I use the descriptor ADDR=PC to mean ADDR=IR is 0. When CLOCK[PC] is a 1 and the ACC is 0, the PC is loaded from the IR address field. When INC[PC] is a 1, the program counter is incremented by 1 at the end of the current clock cycle. When WRITE[RAM] is a 1, the RAM cell addressed by the RAM address bus is loaded with the output of the ALU; when this signal is a 0, the ALU is driven from the output of RAM.

Functional Specifications

Now for the heart of how the Toy instruction set is implemented. In the Toy CPU, all instructions can be executed in just one or two clock cycles. Table 2 shows the actions required to complete each op code's function. Those actions in table 2 that are

continued

not the control signals shown in figure 1 are macros for the ALU control bits whose value is given in table 3. Let's examine some representative op codes in detail.

The STORE op code stores the contents of ACC into RAM. For the first cycle of this instruction, the low 12 bits of the IR address RAM. The ALU routes the ACC contents through without modification, then writes them out to RAM.

STORE requires two clock cycles since RAM is being used for accessing a data value during the first clock cycle. The second clock cycle is the same for all two-cycle instructions; it is simply a decoding of the next op code.

The contents of the RAM address pointed to by the PC are put onto the RAM address bus to fetch the op code. They are then clocked into the IR, and

continued

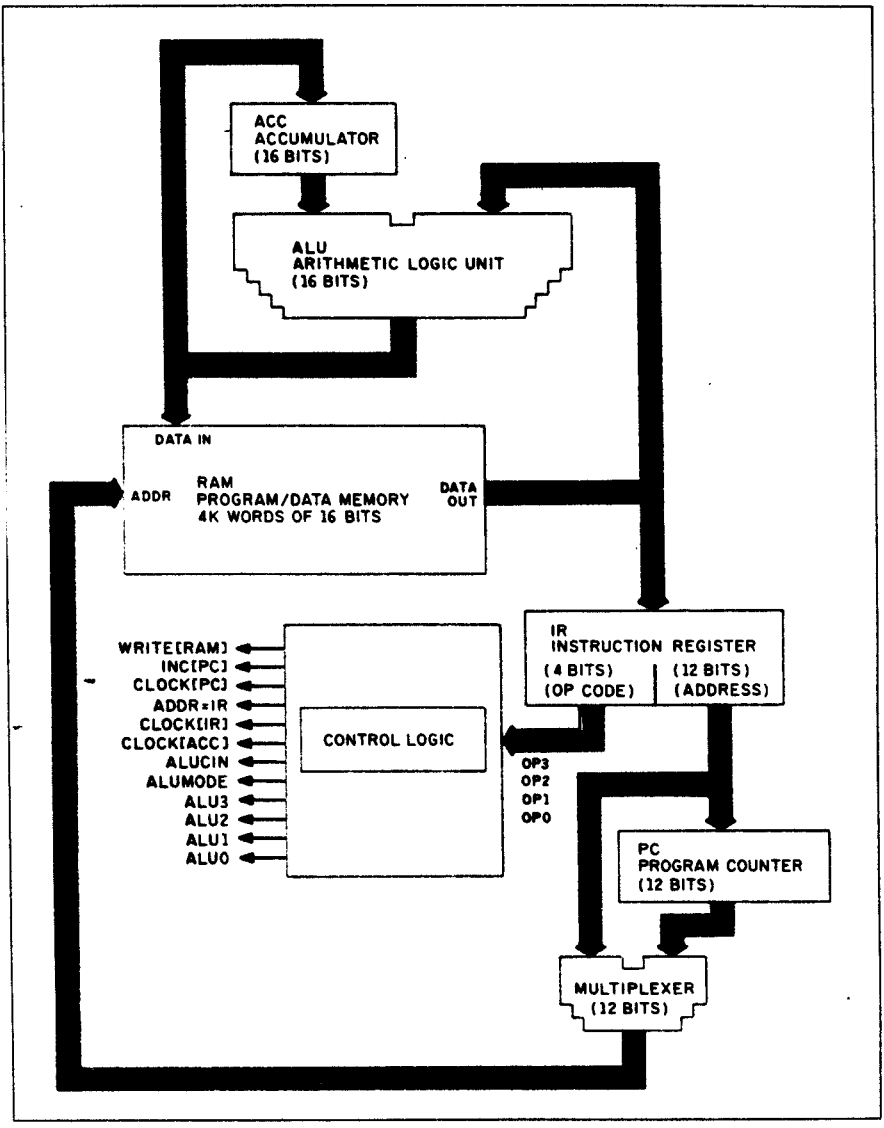


Figure 1: *Toy architecture block diagram.*

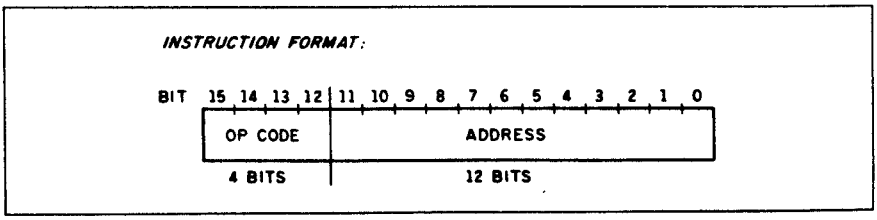


Figure 2: *Toy instruction set format.*

Table 1: *Toy instruction set.*

Op code	Operation	Description
0	STORE	store accumulator in RAM at address
1	LOAD	load ACC from RAM at address
2	JMPZ	jump to address if ACC is zero
3	ADD	add RAM to ACC
4	SUB	subtract RAM from ACC
5	OR	logical OR RAM into ACC
6	AND	logical AND RAM into ACC
7	XOR	logical XOR RAM into ACC
8	NOT	logical one's complement into ACC
9	INC	add 1 to ACC
10	DEC	subtract 1 from ACC
11	ZERO	place 0 in ACC
12	NOP	null operation — unused op code
13	NOP	null operation — unused op code
14	NOP	null operation — unused op code
15	NOP	null operation — unused op code

Table 2: *Toy functional specification. Note that ADDR=PC is equivalent to the ADDR=IR signal being 0. Also, I have used descriptive macro names for the ALU control bits (see table 3).*

Op code	Operation	Cycle	Specification
0	STORE	1	ADDR=IR ; ALU=ACC ; WRITE[RAM]
		2	ADDR=PC ; CLOCK[IR] ; INC[PC]
1	LOAD	1	ADDR=IR ; ALU=RAM ; CLOCK[ACC]
		2	ADDR=PC ; CLOCK[IR] ; INC[PC]
2	JMPZ	1	CLOCK[PC]
		2	ADDR=PC ; CLOCK[IR] ; INC[PC]
3	ADD	1	ADDR=IR ; ALU=ACC+RAM ; CLOCK[ACC]
		2	ADDR=PC ; CLOCK[IR] ; INC[PC]
4	SUB	1	ADDR=IR ; ALU=ACC−RAM ; CLOCK[ACC]
		2	ADDR=PC ; CLOCK[IR] ; INC[PC]
5	OR	1	ADDR=IR ; ALU=ACCorRAM ; CLOCK[ACC]
		2	ADDR=PC ; CLOCK[IR] ; INC[PC]
6	AND	1	ADDR=IR ; ALU=ACCandRAM ; CLOCK[ACC]
		2	ADDR=PC ; CLOCK[IR] ; INC[PC]
7	XOR	1	ADDR=IR ; ALU=ACCxorRAM ; CLOCK[ACC]
		2	ADDR=PC ; CLOCK[IR] ; INC[PC]
8	NOTA	1	ALU=notACC ; CLOCK[ACC] ; ADDR=PC ; CLOCK[IR] ; INC[PC]
9	INCA	1	ALU=ACC+1 ; CLOCK[ACC] ; ADDR=PC ; CLOCK[IR] ; INC[PC]
10	DECA	1	ALU=ACC−1 ; CLOCK[ACC] ; ADDR=PC ; CLOCK[IR] ; INC[PC]
11	ZERO	1	ALU=0 ; CLOCK[ACC] ; ADDR=PC ; CLOCK[IR] ; INC[PC]
12-15	NOP	1	ADDR=PC ; CLOCK[IR] ; INC[PC]

finally the PC is incremented so that it is pointing to the next op code.

JMPZ accomplishes a conditional branch by loading the contents of the PC with the address in the IR. For this to be a conditional branch, the control signal to the PC loader must be ANDed with a signal that is only true if all the bits of the ACC are 0. Since the PC is loaded with the new instruction address at the end of the first clock cycle, the second cycle is a normal decoding instruction for this new address, identical to the second cycle of STORE.

Table 3: *Macros for the ALU control bits (based on bit assignments in the 74181 ALU chip).*

Macro	ALU0	ALU1	ALU2	ALU3	ALUMODE	ALUCIN
ALU = ACC	1	1	1	1	1	x
ALU = RAM	0	1	0	1	1	x
ALU = ACC + RAM	1	0	0	1	0	0
ALU = ACC − RAM	0	1	1	0	0	1
ALU = ACC OR RAM	0	1	1	1	1	x
ALU = ACC AND RAM	1	1	0	1	1	x
ALU = ACC XOR RAM	0	1	1	0	1	x
ALU = NOT ACC	0	0	0	0	1	x
ALU = ACC + 1	0	0	0	0	0	1
ALU = ACC − 1	1	1	1	1	0	0
ALU = 0	1	1	0	0	1	x

Table 4: *Control signal value specification.*

Values for first clock cycle of each instruction

Control signal	Op code 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
ALU0	1	0	x	1	0	0	1	0	0	0	1	1	x	x	x	x
ALU1	1	1	x	0	1	1	1	1	0	0	1	1	x	x	x	x
ALU2	1	0	x	0	1	1	0	1	0	0	1	0	x	x	x	x
ALU3	1	1	x	1	0	1	1	0	0	0	1	0	x	x	x	x
ALUMODE	1	1	x	0	0	1	1	1	1	0	0	1	x	x	x	x
ALUCIN	x	x	x	0	1	x	x	x	x	1	0	x	x	x	x	x
CLOCK[ACC]	0	1	0	1	1	1	1	1	1	1	1	1	0	0	0	0
CLOCK[IR]	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
ADDR = IR	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
CLOCK[PC]	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
INC[PC]	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
WRITE[RAM]	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Values for second clock cycle of each instruction

Control signal	Op code 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
ALU0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
ALU1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
ALU2	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
ALU3	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
ALUMODE	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
ALUCIN	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
CLOCK[ACC]	0	0	0	0	0	0	0	0	x	x	x	x	x	x	x	x
CLOCK[IR]	1	1	1	1	1	1	1	1	x	x	x	x	x	x	x	x
ADDR = IR	0	0	0	0	0	0	0	0	x	x	x	x	x	x	x	x
CLOCK[PC]	0	0	0	0	0	0	0	0	x	x	x	x	x	x	x	x
INC[PC]	1	1	1	1	1	1	1	1	x	x	x	x	x	x	x	x
WRITE[RAM]	0	0	0	0	0	0	0	0	x	x	x	x	x	x	x	x

The single-clock-cycle instructions, such as NOTA, do not require a RAM access for an operand. This means that the usual second-cycle decoding sequence can occur during the same clock cycle as the ALU operation that modifies the ACC contents. In the case of NOTA, the RAM input to the ALU is ignored while the ALU computes the one's complement (logical inverse) of the current ACC contents.

Control-Logic Outputs

Table 4 gives a complete listing of all the control-logic output values that you need to specify the Toy functional description. Each X corresponds to a signal whose value does not matter, either because the controlled resource is unused (as in the ALU signals for op code 2) or because the second clock cycle is unused for op codes 8 to 15. These "don't-care" signals become crucial when you are designing hard-wired control circuitry.

Hard-wired Control

A CPU designed with hard-wired control uses random logic such as AND, OR, and NOT gates and either flip-flops or counters to decode each op code and control the processing flow. The hard-wired design process usually consists of identifying all the states needed to implement the instruction set, then deriving the Boolean logic equations required to control the computer's resources for each step.

Figure 3 shows the hard-wired implementation of the functional specifications given in table 4. It requires a controller with two states: first clock cycle and second clock cycle. The flip-flop in figure 3 is forced to the CLOCK1 state whenever a new instruction is clocked into the IR and changes to the CLOCK2 state whenever the IR is not clocked.

The most tedious part of a hard-wired control design is creating the logic gate networks to decode instructions into control signals. I have derived the required logic equations shown in figure 4 from the functional specifications in table 4. Figure 5 shows the Karnaugh map for deriving the first equation (ALU0) in figure 4. (See W. Fletcher's *An Engineering Approach to Digital Design* [Prentice-Hall, 1980] for a discussion of Karnaugh maps.)

The don't-care conditions are vital in reducing the complexity of the gate networks, since they allow freedom to ignore some op-code bits or state bits to minimize decoding logic. A good example of a don't-care condition is the ALU control signals; they do not depend on whether the controller is currently in the CLOCK1 or CLOCK2 mode.

continued

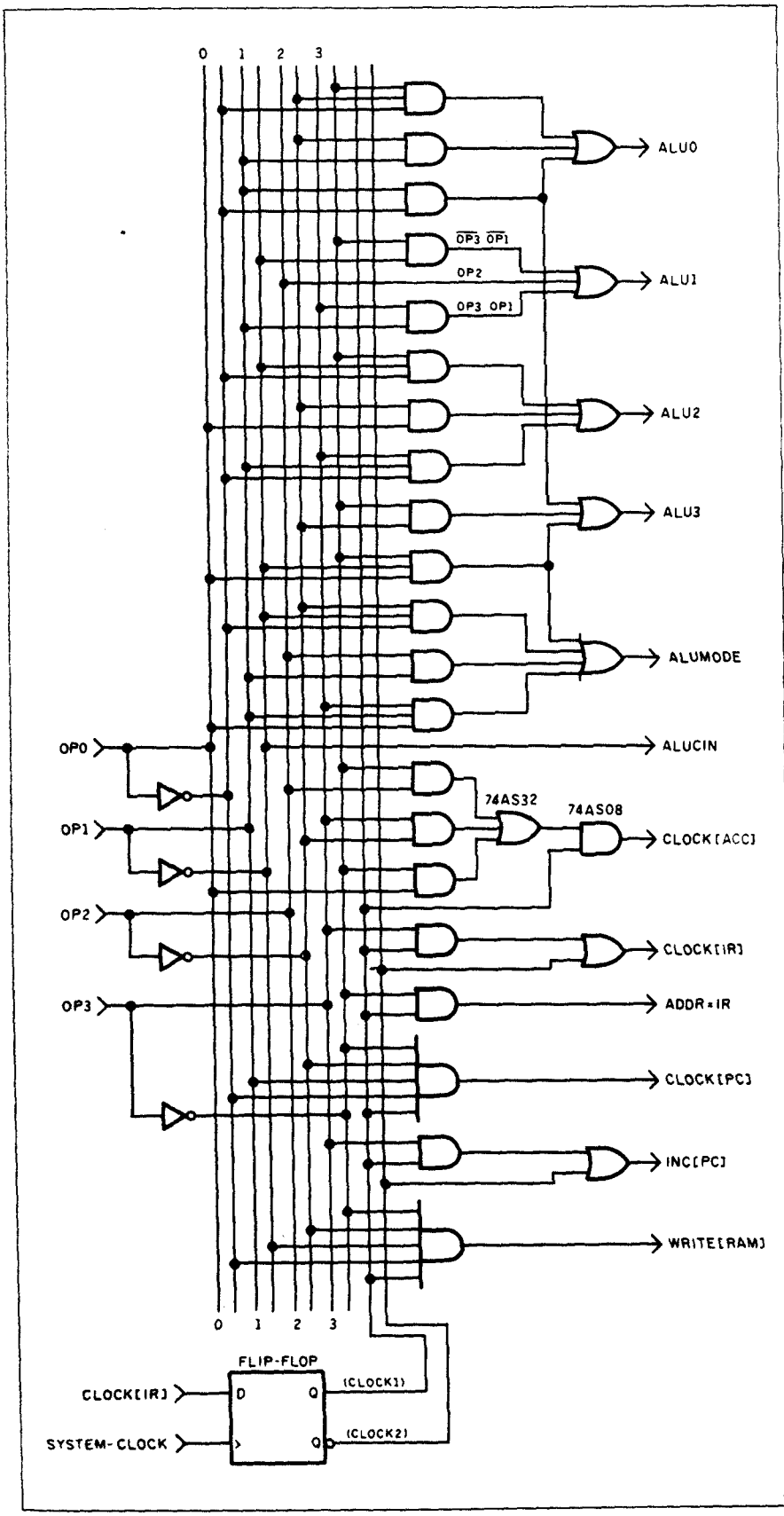


Figure 3: *Hard-wired controller schematic. Note that none of the ALU signals depend on whether the controller is in the CLOCK1 or CLOCK2 mode.*

$$ALU0 = \overline{OP3}\ \overline{OP2}\ \overline{OP0} + \overline{OP2}\ OP1 + OP1\ \overline{OP0}$$

$$ALU1 = \overline{OP3}\ \overline{OP1} + OP2 + OP3\ OP1$$

$$ALU2 = \overline{OP3}\ \overline{OP1}\ \overline{OP0} + OP2\ OP0 + OP3\ OP1\ \overline{OP0}$$

$$ALU3 = \overline{OP3}\ \overline{OP2} + \overline{OP3}\ \overline{OP1}\ OP0 + OP1\ \overline{OP0}$$

$$ALUMODE = \overline{OP2}\ \overline{OP1}\ \overline{OP0} + \overline{OP3}\ \overline{OP1}\ OP0 + OP2\ OP1 + OP3\ OP1\ OP0$$

$$ALUCIN = \overline{OP1}$$

$$CLOCK[ACC] = (OP3\ \overline{OP2} + \overline{OP3}\ OP2 + \overline{OP3}\ OP0)\ CLOCK1$$

$$CLOCK[IR] = OP3\ CLOCK1 + CLOCK2$$

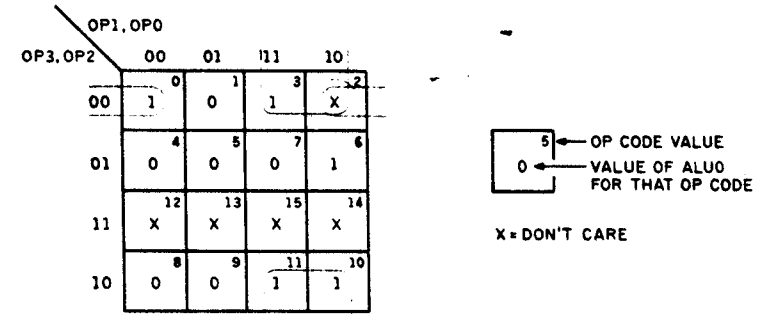
$$ADDR{=}IR = \overline{OP3}\ CLOCK1$$

$$CLOCK[PC] = \overline{OP3}\ \overline{OP2}\ OP1\ \overline{OP0}\ CLOCK1$$

$$INC[PC] = OP3\ CLOCK1 + CLOCK2$$

$$WRITE[RAM] = \overline{OP3}\ \overline{OP2}\ \overline{OP1}\ \overline{OP0}\ CLOCK1$$

Figure 4: *Logic equations for Toy's hard-wired implementation.*



	OP CODE NUMBERS	CORRESPONDING BIT VALUES	EQUATION
REGION	: 2, 6, 10, 14	OP1 = 1, OP0 = 0	$OP1 \cdot \overline{OP0}$
REGION	: 2, 3, 10, 11	OP2 = 0, OP1 = 1	$\overline{OP2} \cdot OP1$
REGION	: 0, 2	OP3 = 0, OP2 = 0, OP0 = 0	$\overline{OP3} \cdot \overline{OP2} \cdot \overline{OP0}$

$$ALU0 = \overline{OP3} \cdot \overline{OP2} \cdot \overline{OP0} + \overline{OP2} \cdot OP1 + OP1\ \overline{OP0}$$

Figure 5: *To show how the Boolean equations in figure 4 were derived from table 4, here is the Karnaugh map used to minimize the ALU0 Boolean equation. The Xs are the don't-care bits, and the number in the upper right corner of each box is the op code.*

To implement the hard-wired controller, the complementary outputs of the CLOCK1/CLOCK2 flip-flop and the inputs from the current op code in the IR are fed throughout the system by the lines at the left of figure 3. These inputs are then fed through logic-gate combinations specified by the equations in figure 4. You can implement these logic-gate combinations with TTL logic gates or, if you want to save board space, program them into hardware, such as a PAL.

As an example of how these decoding gates work, consider the generation of the signal INC[PC]. The INC[PC] signal should be a 1 for op codes 8 to 15 on the first clock cycle and for op codes 0 to 7 on the second clock cycle. But, since op codes 8 to 15 are all single-cycle op codes, any signals generated from them during the second cycle can be ignored. This gives the result that INC[PC] can be 1 for all op codes during the second cycle. The logic for INC[PC] then becomes the AND of the highest op-code bit (OP3) and CLOCK1, with the result ORed with CLOCK2.

Because the time required for a signal to pass through a simple logic gate is only a few nanoseconds with most current technologies, hard-wired control can provide the fastest possible decoding of machine language instructions. It also is the most flexible design method for specifying unique and complex control flows within a CPU because the designer can specify any decoding gate combinations and any control-flow hardware.

One drawback to using hard-wired control methodology is that it requires a considerable amount of Boolean algebra manipulation. Another drawback is that the CPU must be completely and correctly specified before you design a hard-wired control unit.

Any additions or modifications to the specification can require a major redesign of the control unit. If you want a feel for the impact a design change can have on a hard-wired controller, try redoing the logic equations with two op codes switched, such as op codes 5 and 9, or with op code 15 defined as a two-cycle logical NAND instruction.

Microcoded Control

Microcoded design differs from hard-wired design in that the control-logic gates are replaced by a memory array (usually a ROM) to generate the required control-logic signals. While ROMs are slower than random logic within the same price and performance categories, using a ROM simplifies the design process and significantly reduces time and costs for implementing a CPU control circuit.

Figure 6 shows the schematic for a

microcoded control circuit for Toy. The op code and a flip-flop similar to the one used in the hard-wired controller are fed in as an address to the microprogram ROM. The outputs of the ROM directly drive the control signals for the CPU. Each ROM location contains the proper bit settings to control a single clock cycle of an op code's execution, as shown in figure 7.

The control signals for the first cycle of each op code are placed in the even memory addresses (which are addressed when the flip-flop in the controller outputs a 0 for the first clock cycle), and the second cycle op codes are placed in odd memory addresses. I have arbitrarily assigned the value 0 to all don't-care bits from table 4 and copied the rest of the bits directly from table 4 to figure 7.

The main advantage to microcoded control is that it lets the designer change the CPU's functional description by changing the bits in any ROM address without having to redesign the machine's logic-decoding gate structure. Microcoded machine design also lends itself to simply structured, low-component-count computers such as those built using bit-slice technology. Most modern microproces-

continued

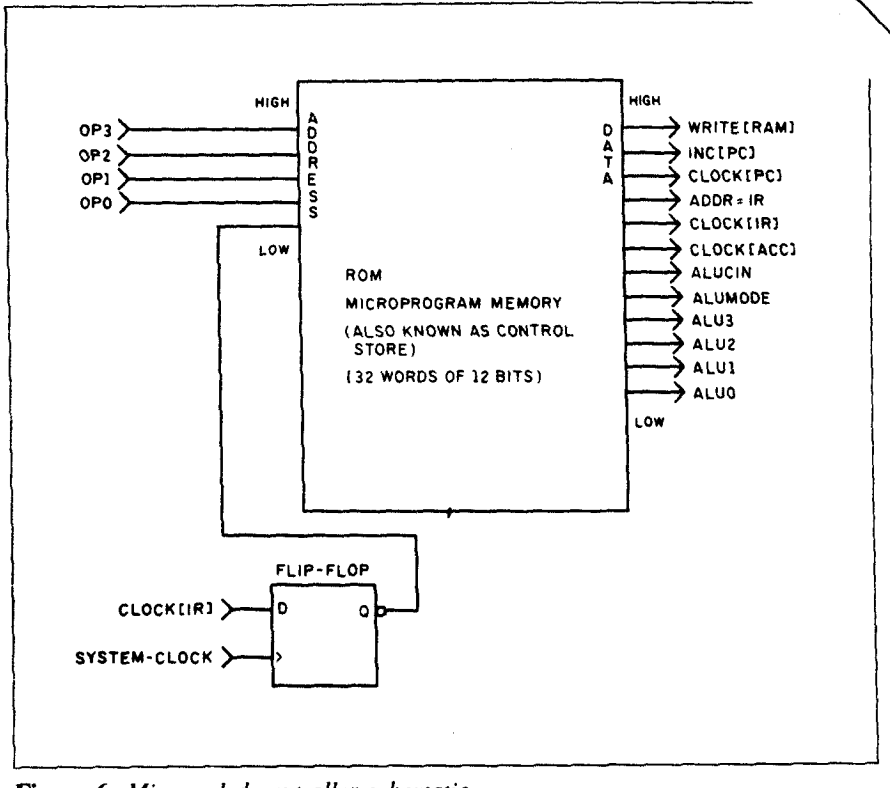


Figure 6: *Microcoded controller schematic.*

OP CODE	ROM ADDRESS	WRITE[RAM]	INC[PC]	CLOCK[PC=IR]	ADDR[IR]	CLOCK[ACC]	ALUCIN	ALUMODE	ALU3	ALU2	ALU1	ALU0
0	0	1	0	0	1	0	0	0	1	1	1	1
	1	0	1	0	0	1	0	0	0	0	0	0
1	2	0	0	0	1	0	1	0	1	1	0	1
	3	0	1	0	0	1	0	0	0	0	0	0
2	4	0	0	1	1	0	0	0	0	0	0	0
	5	0	1	0	0	1	0	0	0	0	0	0
3	6	0	0	0	1	0	1	0	0	1	0	0	1
	7	0	1	0	0	1	0	0	0	0	0	0	0
4	8	0	0	0	1	0	1	1	0	0	1	1	0
	9	0	1	0	0	1	0	0	0	0	0	0	0
5	10	0	0	0	1	0	1	0	1	1	1	1	0
	11	0	1	0	0	1	0	0	0	0	0	0	0
6	12	0	0	0	1	0	1	0	1	1	0	1	1
	13	0	1	0	0	1	0	0	0	0	0	0	0
7	14	0	0	0	1	0	1	0	1	0	1	1	0
	15	0	1	0	0	1	0	0	0	0	0	0	0
8	16	0	1	0	0	1	1	0	1	0	0	0	0
	17	0	0	0	0	0	0	0	0	0	0	0	0
9	18	0	1	0	0	1	1	1	0	0	0	0	0
	19	0	0	0	0	0	0	0	0	0	0	0	0
10	20	0	1	0	0	1	1	0	0	1	1	1	1
	21	0	0	0	0	0	0	0	0	0	0	0	0
11	22	0	1	0	0	1	1	0	1	0	0	1	1
	23	0	0	0	0	0	0	0	0	0	0	0	0
12	24	0	1	0	0	1	0	0	0	0	0	0	0
	25	0	0	0	0	0	0	0	0	0	0	0	0
13	26	0	1	0	0	1	0	0	0	0	0	0	0
	27	0	0	0	0	0	0	0	0	0	0	0	0
14	28	0	1	0	0	1	0	0	0	0	0	0	0
	29	0	0	0	0	0	0	0	0	0	0	0	0
15	30	0	1	0	0	1	0	0	0	0	0	0	0
	31	0	0	0	0	0	0	0	0	0	0	0	0

Figure 7: *Contents of ROM for the microcode.*

sors and large computers use microcoded design techniques because the design costs associated with hard-wired control are too high.

In some cases, a computer will use RAM instead of ROM for its microcoded memory, providing a "writable control store." A sophisticated programmer can use this to modify and extend the machine's instruction set for special applications. By using multiple sets of ROM or RAM within a machine, the programmer can make a computer emulate more than one machine-code instruction set for different computing environments.

The method of microcoding I used in Toy is called horizontal microcoding, since each bit of the ROM directly feeds a control line for the CPU. A hybrid design method known as vertical microcoding compacts some control signals together to save ROM bits. It then uses decoding logic much like that used by the hard-wired approach to regenerate the signals.

In general, hard-wired control is used for computer designs that are simple or that require fast execution speeds, while microcoded control is used in complex computer designs to keep design costs low. Both design methods can implement CPUs that are much more complex than the Toy architecture. ■

BIBLIOGRAPHY

Hill, F., and Peterson, G. *Digital Systems: Hardware Organization and Design*. (2nd ed.) New York: John Wiley & Sons, 1978.

The TTL Data Book, volume 2, Dallas, TX: Texas Instruments Inc., 1985, pages 3-712.